PSOC™ Control C3M/P6 MCU architecture

Reference manual

About this document

Scope and purpose

This document provides comprehensive information about Infineon's PSC3M/P6 MCU series of microcontrollers. The devices are based on 32-bit, Arm® Cortex®-M33 running at up to 180 MHz and include DSP and FPU capabilities.

A high-performance programmable analog subsystem is provided, featuring 12-bit, 9-MSPS ADC, hardware 3P3Z filters, DACs, OpAmps with programmable-gain amplifier mode, and comparator slope generators. Advanced high-resolution timers are included for high-performance real-time closed-loop control applications, along with peripherals that enable flexible communication and security.

This document describes the following subfamilies of PSC3M/P6 MCUs:

- PSOC™ Control C3M (Motor Control) series: PSC3M6
- PSOC™ Control C3P (Power Control) series: PSC3P6

For an overview of the PSC3M/P6 MCU product family, refer to the device datasheet.

Intended audience

This document helps hardware engineers, embedded software developers, and system architects understand the architecture of Infineon's PSC3M/P6 MCU microcontrollers series.

Table of contents

About this document . . . 1
Table of contents . . . 2
1 Introduction . . . 26
1.1 Detailed features . . . 26
1.2 Architecture . . . 29
1.3 Document organization and conventions . . . 29
1.3.1 Major sections . . . 29
1.3.2 Document conventions . . . 30
1.3.2.1 Register conventions . . . 30
1.3.2.2 Numeric naming . . . 30
1.3.2.3 Units of measurement . . . 30
1.3.2.4 Acronyms and abbreviations . . . 31
2 CPU subsystem (CPUSS) . . . 37
2.1 Features . . . 37
2.2 Architecture . . . 37
2.2.1 CPUSS . . . 37
2.2.1.1 Key security features . . . 37
2.2.1.2 Memory and bus architecture . . . 38
2.2.1.3 Peripherals and interconnect . . . 38
2.2.1.4 Power and clocking . . . 38
2.2.1.5 Power-on behavior . . . 38
2.2.2 Address and memory maps . . . 38
2.2.2.1 Wait state lookup tables . . . 40
2.3 Operating modes and privilege levels . . . 40
2.3.1 Security states . . . 40
2.3.2 Operating modes . . . 40
2.3.3 Operating modes . . . 41
2.3.4 Operating states . . . 41
2.3.5 Privilege levels . . . 41
2.4 Instruction set . . . 41
2.5 Registers summary . . . 42
3 Bus infrastructure . . . 46
3.1 SYSCPUSS AHB5 interconnect . . . 47
3.1.1 AHB5 bus matrix interconnect . . . 48
3.1.2 SYSCPU buses . . . 49
3.1.2.1 Cortex® -M33 C-AHB bus . . . 49
3.1.2.2 Cortex® -M33 S-AHB bus . . . 49
3.1.2.3 DMA bus . . . 49

Table of contents

3.2 AHB5 MMIO 49
3.3 Peripheral interconnect (PERI) 49
3.3.1 AHB bus error handling 51
3.4 Bus master peripherals 51
3.4.1 Direct memory access (DMA-DW) 51
3.4.2 CryptoLite 51
3.4.3 USBFS 51

4 Inter processor communication (IPC) 52
4.1 Features 52
4.2 Architecture 52
4.2.1 IPC channel 52
4.2.2 IPC interrupt 53
4.2.3 IPC channels and interrupts 54
4.3 Implementing locks 55
4.4 Message passing 56
4.5 Typical usage models 58
4.5.1 Full-duplex communication 58
4.5.2 Half-duplex with independent event handling 58
4.5.3 Half-duplex with shared event handling 59

5 Flash 61
5.1 Features 61
5.2 Configuration 61
5.2.1 FLASHC 61
5.2.2 Flash geometry 63
5.2.3 Bank modes and address mapping 64
5.3 Power modes 65
5.4 Read While Write support 65
5.5 ECC 66
5.5.1 ECC generation/encoding 66
5.5.2 ECC (error) injection 66
5.6 Security and protection 67
5.7 SROM APIs 67

6 SRAM controller 70
6.1 Features 70
6.2 Architecture 70
6.2.1 AHB5 read transfers 70
6.2.2 32-bit AHB5 write transfers 70
6.2.3 Partial AHB5 write transfers 70
6.2.4 Power partition control 70
6.3 Error Correcting Code (ECC) 71
6.3.1 ECC Error injection 71

Table of contents

6.3.2 ECC error report functionality . . . 71
6.3.2.1 Partial AHB5 write transfers . . . 71
6.3.2.2 AHB5 read transfers . . . 71
6.3.2.3 Correctable ECC error reporting . . . 71
6.3.2.4 Non-correctable ECC error reporting . . . 72
6.4 Wait states . . . 72

7 Interrupts . . . 73
7.1 Features . . . 73
7.2 Architecture . . . 73
7.3 Interrupts and exceptions operation . . . 74
7.3.1 Interrupt/exception handling . . . 74
7.3.2 Level interrupts . . . 74
7.3.3 Exception vector table . . . 75
7.4 Exception sources . . . 77
7.4.1 Reset exception . . . 77
7.4.2 Non-maskable interrupt exception . . . 78
7.4.3 HardFault exception . . . 78
7.4.4 MemManage fault exception . . . 79
7.4.5 BusFault exception . . . 79
7.4.6 UsageFault exception . . . 79
7.4.7 SecureFault exception . . . 79
7.4.8 Supervisor Call (SVCall) exception . . . 79
7.4.9 DebugMonitor exception . . . 80
7.4.10 PendSupervisory (PendSV) exception . . . 80
7.4.11 System Tick (SysTick) exception . . . 80
7.5 Interrupt sources . . . 80
7.6 Interrupt/exception target security state and banking . . . 86
7.6.1 Interrupt target security state . . . 87
7.6.2 GPIO secure and non-secure interrupts . . . 87
7.7 Interrupt/exception priority . . . 87
7.8 Enabling and disabling interrupts . . . 88
7.9 Interrupt/exception states . . . 89
7.9.1 Pending interrupts/exception . . . 89
7.10 Stack usage for interrupts/exceptions . . . 90
7.11 Interrupts and low-power modes . . . 91
7.12 Interrupt/exception - initialization/configuration . . . 91

8 Fault monitoring . . . 93
8.1 Features . . . 93
8.2 Architecture . . . 93
8.2.1 Fault report . . . 94
8.2.2 Signaling interface . . . 94

8.2.3 Monitoring . . . 95
8.2.4 Low-Power mode operation . . . 95
8.2.5 Using a fault structure . . . 96
8.2.6 CPU exceptions versus fault monitoring . . . 96
8.3 Fault sources . . . 97
8.4 Fault encodings . . . 99
8.4.1 PERI Master0 interface PPC violation . . . 99
8.4.2 PERI Master1 interface PPC violation . . . 100
8.4.3 PERI peripheral PPC PC mask violation . . . 100
8.4.4 PERI peripheral Group1 AHB timeout . . . 101
8.4.5 PERI peripheral Group2 AHB timeout . . . 101
8.4.6 PERI peripheral Group3 AHB timeout . . . 101
8.4.7 PERI peripheral Group4 AHB timeout . . . 102
8.4.8 PERI peripheral Group5 AHB timeout . . . 102
8.4.9 PERI peripheral Group0 AHB violation . . . 102
8.4.10 PERI peripheral Group1 AHB violation . . . 102
8.4.11 PERI Peripheral Group2 AHB violation . . . 103
8.4.12 PERI Peripheral Group3 AHB violation . . . 103
8.4.13 PERI Peripheral Group4 AHB violation . . . 104
8.4.14 PERI Peripheral Group5 AHB violation . . . 104
8.4.15 RAMC-0 MPC violation . . . 105
8.4.16 RAMC-1 MPC violation . . . 106
8.4.17 AHB bridge error fault . . . 106
8.4.18 PROMC MPC violation . . . 107
8.4.19 RAMC-0_C ECC mmio . . . 107
8.4.20 RAMC-0_NC ECC mmio . . . 107
8.4.21 RAMC-1_C ECC mmio . . . 107
8.4.22 RAMC-1_NC ECC mmio . . . 108
8.4.23 FLASHC MPC fault . . . 108
8.4.24 FLASHC bus access fault . . . 108
8.4.25 FLASHC main_c ECC mmio fault . . . 109
8.4.26 FLASHC main_nc ECC mmio fault . . . 109
8.4.27 FLASHC FM CTL fault . . . 109
8.4.28 FLASHC FM r_q fault . . . 110
8.4.29 FLASHC flash CTL fault . . . 110
8.4.30 FLASHC addr hole fault . . . 111
8.4.31 FLASHC lock fault . . . 111
8.4.32 SRSS clock supervisor fault . . . 112
8.4.33 SRSS supply supervisor fault . . . 112
8.4.34 Supply voltage glitch sensor fault . . . 112
8.4.35 Advanced encryption standard fault . . . 112
8.5 Registers summary . . . 113

Table of contents

9 Direct memory access (DMA - DW) . . . 114
9.1 Features . . . 114
9.2 Architecture . . . 115
9.2.1 Standard flow . . . 115
9.3 Channels . . . 116
9.3.1 Channel priority . . . 116
9.3.2 Channel security attributes . . . 116
9.3.3 Channel state . . . 116
9.3.4 Channel interrupt generation . . . 117
9.4 Descriptors . . . 118
9.4.1 Descriptor memory transfer types . . . 118
9.4.1.1 Single transfer . . . 119
9.4.1.2 1D memory transfer . . . 119
9.4.1.3 2D memory transfer . . . 120
9.4.1.4 Cyclic Redundancy Check (CRC) transfer . . . 121
9.4.2 Descriptor size . . . 122
9.4.3 Descriptor configuration . . . 122
9.4.3.1 Descriptor control (DESCR_CTL) . . . 123
9.4.3.2 Source and destination address (DESCR_SRC/DESCR_DST) . . . 124
9.4.3.3 Descriptor X loop control (DESCR_X_CTL) . . . 124
9.4.3.4 Descriptor Y loop control (DESCR_Y_CTL) . . . 124
9.4.3.5 Descriptor next pointer (DESCR_NEXT_PTR) . . . 124
9.4.4 Descriptor transfer Size . . . 124
9.4.5 Descriptors chaining . . . 125
9.5 Triggers . . . 126
9.5.1 Input triggers . . . 126
9.5.2 Pending triggers . . . 127
9.5.3 Output triggers . . . 127
9.6 Performance . . . 128
9.6.1 Single transfer . . . 128
9.6.2 1D transfer . . . 128
9.6.3 2D transfer . . . 129
9.6.4 CRC transfer . . . 129

10 Trigger multiplexer (Trigger MUX) . . . 130
10.1 Features . . . 130
10.2 Architecture . . . 130
10.3 Functional description . . . 131
10.4 Software triggers . . . 151
10.5 Trigger functionality . . . 151
10.6 Registers summary . . . 154

11 Debug and Trace system . . . 156

Table of contents

11.1 Features . . . 156
11.2 Architecture . . . 156
11.2.1 Debug access port (DAP) . . . 157
11.2.1.1 DAP power domain . . . 158
11.2.1.2 DAP security . . . 158
11.2.2 ROM tables . . . 159
11.2.3 Trace . . . 159
11.2.4 Embedded cross triggering . . . 159
11.2.4.1 Freeze Peripherals in Debug Mode . . . 161
11.3 Serial wire debug (SWD) interface . . . 161
11.4 JTAG interface . . . 162
11.5 Programming the MCU . . . 162
11.5.1 SWD port acquisition . . . 162
11.5.2 SWD programming mode entry . . . 163
11.5.3 SWD programming routine executions . . . 163
11.6 PSC3M/P6 MCU SWD debug interface . . . 163
11.6.1 Debug control and configuration registers . . . 163
11.6.2 Break point units (BPU) . . . 164
11.6.3 Data watchpoints (DWT) . . . 164
11.6.4 Debugging the PSC3M/P6 MCU device . . . 164
11.7 Registers summary . . . 164

12 Cryptographic accelerator . . . 165
12.1 Features . . . 165
12.2 Architecture . . . 165
12.2.1 Randomized clock . . . 166
12.2.2 Hardware/software interface . . . 166
12.3 Block functions . . . 168
12.3.1 AES-128 . . . 168
12.3.2 SHA . . . 169
12.3.2.1 SHA-256/384/512 . . . 169
12.3.3 Vector unit (VU) . . . 171
12.3.4 True random number generator (TRNG) . . . 174
12.3.4.1 Fixed ring oscillators: RO11, RO15, GARO15, FIRO15 . . . 176
12.3.4.2 Ring oscillator: GARO31 . . . 176
12.3.4.3 Ring oscillator: FIRO31 . . . 177
12.3.4.4 TRNG result . . . 178
12.3.4.5 TRNG health monitor . . . 178
12.4 Interrupts . . . 180
12.4.1 TRNG interrupt . . . 180
12.4.2 Error interrupt . . . 180
12.5 Registers summary . . . 181

Table of contents

13 TrustZone . . . 183
13.1 Features . . . 183
13.2 Architecture . . . 183
13.3 Memory regions . . . 185
13.4 Attribution units . . . 186
13.4.1 Implementation Defined Attribution Unit (IDAU) . . . 186
13.4.2 Secure Attribution Unit (SAU) . . . 187
13.5 State transition . . . 189
13.5.1 Secure to non-secure state transition . . . 189
13.5.2 Non-secure to Secure state transition . . . 190
13.5.3 State transition from exception or interrupt . . . 190

14 Protection units . . . 191
14.1 Architecture . . . 192
14.2 Protection context . . . 195
14.2.1 Special protection contexts . . . 196
14.3 Master protection attributes . . . 196
14.3.1 Master Security Controller (MSC) . . . 197
14.3.1.1 Memory Protection Unit (MPU) . . . 197
14.4 Slave protection attributes . . . 197
14.4.1 Memory Protection Controller (MPC) . . . 197
14.4.1.1 Software view . . . 198
14.4.1.2 MPC MMIO register protection . . . 200
14.4.2 Peripheral Protection Controller (PPC) . . . 200
14.4.3 PPC access control rules . . . 201
14.4.3.1 Security aware peripherals . . . 202
14.4.3.2 PPC regions . . . 203

15 Secure boot . . . 213
15.1 Basic definitions . . . 213
15.2 Features . . . 214
15.3 Architecture . . . 214
15.3.1 Policies and keys . . . 214
15.3.1.1 Policies . . . 215
15.3.1.2 Keys . . . 215
15.3.2 Life cycle stages (LCS) . . . 215
15.3.2.1 IFX_MFG . . . 216
15.3.2.2 Development LCS . . . 216
15.3.2.3 Production LCS . . . 216
15.3.2.4 RMA . . . 217
15.3.3 Boot sequence . . . 217
15.3.3.1 Secure boot . . . 218
15.3.3.2 Protection states . . . 219

15.3.4 BootROM serial interface . . . 220
15.3.5 Secure provisioning and re-provisioning . . . 221
15.3.6 Protection context used in secure boot . . . 221

16 Advanced Encryption Standard (AES) . . . 222
16.1 Introduction . . . 222
16.2 Overview . . . 222
16.3 Programming model . . . 223
16.4 Functional description . . . 225
16.4.1 Module reset . . . 225
16.4.2 Endianess . . . 225
16.4.3 BusFault scenarios . . . 225
16.4.4 Handling of key state clearing . . . 225
16.4.5 Fault structure . . . 226
16.4.6 Timing . . . 226
16.4.7 Debug support . . . 226
16.5 AES registers interface . . . 226
16.5.1 Registers overview . . . 227
16.5.2 Access conditions . . . 228
16.5.3 Resets . . . 229
16.5.4 AES configuration register . . . 229
16.5.5 AES command register . . . 230
16.5.6 AES status register . . . 231
16.5.7 AES configuration register 2 . . . 233
16.5.8 AES interrupt cause register . . . 234
16.5.9 AES interrupt set register . . . 235
16.5.10 AES interrupt mask register . . . 236
16.5.11 AES masked interrupt register . . . 236
16.5.12 AES clear register . . . 237
16.5.13 AES bus counter register . . . 238
16.5.14 AES Q-channel response register . . . 239
16.5.15 AES user seeding register . . . 240
16.5.16 AES Key registers . . . 240
16.5.16.1 AES key register (x for index of shares; y for index of word) . . . 241
16.5.17 AES Data registers . . . 241
16.5.17.1 AES data register (x for index of shares; y for index of word) . . . 242
16.5.17.2 AES data reread register (x for index of shares; y for index of word) . . . 243

17 Power supply and monitoring . . . 244
17.1 Features . . . 244
17.2 Architecture . . . 245
17.3 Power supply . . . 246
17.3.1 Active regulator . . . 246

Table of contents

17.3.2 Deep Sleep regulator . . . 246
17.3.3 Retention regulator . . . 246
17.3.4 Backup domain . . . 247
17.4 SRSS reference block . . . 247
17.5 Voltage monitoring . . . 247
17.5.1 Power-on-reset (POR) . . . 247
17.5.2 Brown-out detection (BOD) . . . 247
17.5.3 Low voltage detect (LVD) . . . 247
17.6 Sleep control . . . 248
17.7 Registers summary . . . 248

18 Device power modes . . . 249
18.1 Features . . . 249
18.2 Architecture . . . 249
18.2.1 Power modes . . . 250
18.2.1.1 Active and Sleep . . . 251
18.2.1.2 Deep Sleep . . . 252
18.2.1.3 Deep Sleep RAM . . . 252
18.2.1.4 Deep Sleep OFF . . . 252
18.2.1.5 Hibernate . . . 253
18.2.2 Other operation states . . . 253
18.2.2.1 Reset state (XRES) . . . 253
18.2.2.2 Off state (OFF) . . . 254
18.3 Power mode transitions . . . 254
18.4 Summary . . . 256
18.5 Registers summary . . . 257

19 Clocking system . . . 258
19.1 Features . . . 258
19.2 Architecture . . . 258
19.3 Clock sources . . . 260
19.3.1 Internal main oscillator (IMO) . . . 260
19.3.2 Internal high-frequency oscillator (IHO) . . . 260
19.3.3 Internal low-speed oscillator (ILO) . . . 260
19.3.4 External clock (EXT_CLK) . . . 260
19.3.5 External crystal oscillator (ECO) . . . 261
19.3.6 Watch crystal oscillator (WCO) . . . 261
19.3.7 ECO prescaler . . . 261
19.4 Clock generation . . . 261
19.4.1 Digital phase-locked loop (DPLL) . . . 261
19.4.2 Frequency lock loop (FLL) . . . 264
19.5 Clock trees . . . 267
19.5.1 Clock path . . . 267

Table of contents

19.5.2 High-frequency root clocks (CLK_HF) . . . 268
19.5.3 Low-frequency clock (CLK_LF) . . . 269
19.5.4 Backup clock (CLK_BAK) . . . 270
19.5.5 Timer clock (CLK_TIMER) . . . 270
19.5.6 Power clock (CLK_PWR) . . . 270
19.6 CLK_HF distribution . . . 271
19.7 Peripheral clock dividers . . . 272
19.7.1 Fractional clock dividers . . . 272
19.7.2 Peripheral clock divider configuration . . . 272
19.7.3 Peripheral clock divider groups . . . 274
19.8 Clock calibration counters . . . 278
19.9 Clock supervision (CSV) . . . 279
19.10 Registers summary . . . 281

20 Watchdog timer . . . 284
20.1 Features . . . 284
20.2 Architecture . . . 284
20.3 Basic watchdog timer . . . 285
20.3.1 Overview . . . 285
20.3.2 Watchdog reset . . . 287
20.3.3 Watchdog interrupt . . . 288
20.4 Multi-counter WDT . . . 289
20.4.1 Overview . . . 289
20.4.1.1 MCWDT0_WDT0 and MCWDT0_WDT1 counters operation . . . 289
20.4.1.2 MCWDT0_WDT2 counter operation . . . 292
20.4.2 Enabling and disabling MCWDT . . . 294
20.4.3 Watchdog cascade options . . . 295
20.4.4 MCWDT reset . . . 298
20.4.5 MCWDT interrupt . . . 298
20.5 Reset cause detection . . . 299
20.6 Registers summary . . . 299

21 Backup . . . 301
21.1 Features . . . 301
21.2 Architecture . . . 301
21.3 Power supply . . . 302
21.4 Clocking . . . 302
21.4.1 WCO with external clock/sine wave input . . . 303
21.4.2 Calibration . . . 303
21.5 Reset . . . 304
21.6 Real-time clock . . . 304
21.6.1 Reading RTC User registers . . . 305
21.6.2 Writing to RTC User registers . . . 305

Table of contents

21.7 Alarm feature . . . 305
21.8 Backup registers . . . 307
21.9 Registers summary . . . 308

22 Reset system . . . 309
22.1 Features . . . 309
22.2 Reset sources . . . 309
22.2.1 Brownout reset . . . 311
22.2.2 WDT/MCWDT reset . . . 311
22.2.3 Software reset . . . 311
22.2.4 External reset . . . 312
22.2.5 Debugger reset . . . 312
22.2.6 Fault detection reset . . . 312
22.2.7 **High frequency clock error reset** . . . 312
22.2.8 Reference clock error reset . . . 312
22.2.9 Hibernate wakeup reset . . . 312
22.3 Identifying reset sources . . . 313
22.4 Registers summary . . . 314

23 SVGS (Supply Voltage Glitch Sensor) . . . 315
23.1 Introduction . . . 315
23.2 Features . . . 315
23.3 Architecture . . . 315
23.4 Functional description . . . 316
23.5 Registers summary . . . 321

24 I/O subsystem . . . 322
24.1 Features . . . 322
24.2 Architecture . . . 323
24.2.1 I/O cell architecture . . . 323
24.2.2 HSIO to map alternate functions on GPIO . . . 324
24.2.3 I/O state on power-up . . . 328
24.2.4 Behavior in low-power modes . . . 329
24.3 Functional description . . . 329
24.3.1 Digital input buffer . . . 329
24.3.2 Digital output driver . . . 330
24.3.2.1 GPIO-OVT pins . . . 330
24.3.2.2 Drive strength . . . 330
24.3.2.3 Slew rate control . . . 331
24.3.2.4 Drive mode . . . 331
24.3.3 Firmware-controlled GPIO . . . 335
24.3.4 Interrupt . . . 335
24.3.5 Analog I/O . . . 338
24.3.6 Secure GPIO . . . 338

Table of contents

24.4 Programmable digital (Smart I/O) . . . 338
24.4.1 Overview . . . 338
24.4.2 Block components . . . 339
24.4.2.1 Clock and reset . . . 340
24.4.2.2 Synchronizer . . . 342
24.4.2.3 Lookup table (LUT) . . . 342
24.4.2.4 Data unit (DU) . . . 345
24.4.3 Routing . . . 350
24.4.4 Operation . . . 350
24.5 Registers summary . . . 351

25 Timer, counter, and pulse-width modulator (TCPWM) . . . 354
25.1 Features . . . 354
25.1.1 Supported features of TCPWM block . . . 355
25.2 Architecture . . . 356
25.2.1 Enabling and disabling counters in a TCPWM block . . . 357
25.2.2 Clocking . . . 357
25.2.2.1 Clock pre-scaling . . . 357
25.2.2.2 Count event . . . 357
25.2.3 Trigger inputs . . . 357
25.2.4 Glitch filtering . . . 361
25.2.5 Synchronization of multiple counters . . . 362
25.2.6 Trigger outputs . . . 363
25.2.7 Internal events . . . 364
25.2.7.1 Underflow event . . . 364
25.2.7.2 Overflow event . . . 364
25.2.7.3 Terminal count event . . . 365
25.2.7.4 cc0_match (cc1_match) event . . . 366
25.2.8 Interrupts . . . 367
25.2.9 Debug mode . . . 368
25.2.10 PWM outputs . . . 369
25.2.11 Power modes . . . 370
25.3 Counter operation modes . . . 371
25.3.1 Timer mode . . . 372
25.3.1.1 Up/Down Count Modes . . . 375
25.3.1.2 One Shot Mode . . . 377
25.3.1.3 Clock Pre-scaling . . . 378
25.3.1.4 CC Auto Reload . . . 378
25.3.1.5 **External Input Event** . . . 379
25.3.1.6 External Direction Control . . . 380
25.3.1.7 How to configure counter for timer mode . . . 382
25.3.2 Capture mode . . . 382
25.3.2.1 Capture COUNTER value into CC0/CC0_BUFF . . . 384

25.3.2.2 Up/Down Count Modes . . . 386
25.3.2.3 One Shot Mode . . . 386
25.3.2.4 Clock Pre-scaling . . . 386
25.3.2.5 How to configure counter for capture mode . . . 386
25.3.3 Quadrature mode . . . 387
25.3.3.1 Quadrature mode functionality overview . . . 387
25.3.3.2 QUAD_RANGE0 . . . 391
25.3.3.3 QUAD_RANGE0_CMP . . . 394
25.3.3.4 QUAD_RANGE1_CMP . . . 397
25.3.3.5 QUAD_RANGE1_CAPT . . . 400
25.3.4 Pulse-width modulation (PWM) mode . . . 401
25.3.4.1 Up/down Count Modes . . . 407
25.3.4.2 One Shot Mode . . . 409
25.3.4.3 Clock Pre-scaling . . . 409
25.3.4.4 CC and PERIOD auto reload with switch event . . . 409
25.3.4.5 Left/Right/Center align PWM with CC0/CC0_BUFF auto reload . . . 411
25.3.4.6 PWM output selection for stepper motor control . . . 425
25.3.4.7 How to configure counter for PWM mode . . . 428
25.3.5 Pulse width modulation with dead time mode (PWM_DT) mode . . . 429
25.3.5.1 How to configure counter for PWM with dead time mode . . . 433
25.4 Dithering . . . 433
25.5 Motion interface (MOTIF) . . . 435
25.5.1 Overview . . . 436
25.5.1.1 Features . . . 436
25.5.1.2 Block diagram . . . 436
25.5.2 MOTIF functional description . . . 437
25.5.2.1 MOTIF MMIO registers . . . 438
25.5.2.2 MOTIF Pin mapping . . . 439
25.5.2.3 MOTIF clocks . . . 440
25.5.2.4 MOTIF reset . . . 441
25.5.2.5 Input trigger mapping . . . 441
25.5.2.6 Output trigger mapping . . . 441
25.5.2.7 Function selector connection scheme . . . 442
25.5.3 Hall sensor control . . . 443
25.5.3.1 Hall Sensor standard operation . . . 443
25.5.3.2 Hall Sensor operation with LUT . . . 449
25.5.4 Quadrature decoder control . . . 454
25.5.4.1 Quadrature clock and direction decoding . . . 459
25.5.4.2 Index control . . . 461
25.5.5 Standalone multi-channel mode . . . 462
25.5.6 Modulation interface from MOTIF . . . 464
25.5.6.1 Modulation inside counter logic . . . 467

Table of contents

25.5.7 Synchronous start . . . 468
25.5.8 Output selector . . . 468
25.5.9 Using the MOTIF . . . 469
25.5.10 Service request generation . . . 477
25.5.10.1 Hall sensor mode flags . . . 477
25.5.10.2 Quadrature decoder flags . . . 479
25.5.11 MOTIF debug behavior . . . 481
25.5.12 Initialization and System dependencies . . . 482
25.5.12.1 Initialization . . . 482
25.5.12.2 System dependencies . . . 483
25.6 High-resolution PWM (HRPWM) . . . 483
25.6.1 Overview . . . 483
25.6.1.1 Features . . . 483
25.6.1.2 Block diagram . . . 484
25.6.1.3 Clocks . . . 484
25.6.2 PWM . . . 485
25.6.2.1 Computation concept . . . 486
25.6.2.2 PWM output mode . . . 488
25.6.2.3 PWM output mode - Left align . . . 489
25.6.2.4 PWM output mode - Right align . . . 491
25.6.2.5 PWM output mode – Asymmetric with cc0 & cc1 . . . 494
25.6.2.6 PWM output mode – Center align . . . 495
25.6.2.7 PWM output mode – Center align asymmetrical with cc0 . . . 497
25.6.2.8 PWM output mode – Center align asymmetric with cc0 and cc1 . . . 498
25.6.3 PWM_DT . . . 498
25.6.3.1 Computation concept . . . 499
25.6.3.2 PWM_DT – Left align . . . 500
25.6.3.3 PWM_DT – Right align . . . 500
25.6.3.4 PWM_DT – Center align . . . 501
25.7 Recovery . . . 503
25.8 Initialize . . . 503
25.9 Pin status . . . 503
25.10 Registers summary . . . 503
26 Controller area network flexible data-rate (CAN FD) . . . 506
26.1 Overview . . . 506
26.1.1 Features . . . 506
26.1.2 Features not supported . . . 507
26.2 Architecture . . . 507
26.2.1 Block diagram . . . 507
26.2.2 Dual clock sources . . . 507
26.2.3 Interrupt line . . . 507
26.3 Functional description . . . 508

26.3.1 Modes of operation . . . 508
26.3.1.1 Software initialization . . . 508
26.3.1.2 Normal operation . . . 509
26.3.1.3 CAN FD operation . . . 509
26.3.1.4 Transmitter delay compensation . . . 510
26.3.1.4.1 Description . . . 510
26.3.1.4.2 Transmitter delay compensation measurement . . . 511
26.3.1.5 Restricted Operation mode . . . 512
26.3.1.6 Bus Monitoring mode . . . 512
26.3.1.7 Disable automatic retransmission . . . 513
26.3.1.8 Power Down (Sleep mode) . . . 513
26.3.1.9 Test mode . . . 514
26.3.1.9.1 External Loop Back mode . . . 514
26.3.1.9.2 Internal Loop Back mode . . . 514
26.3.1.10 Application watchdog . . . 515
26.3.2 Timestamp generation . . . 515
26.3.3 Timeout counter . . . 516
26.3.4 Receive handling . . . 516
26.3.4.1 Acceptance filtering . . . 516
26.3.4.2 Receive FIFOs . . . 519
26.3.4.2.1 RX FIFO blocking mode . . . 520
26.3.4.2.2 RX FIFO overwrite mode . . . 521
26.3.4.2.3 RX FIFO top pointer . . . 521
26.3.4.3 Dedicated RX buffers . . . 522
26.3.4.4 Debug on CAN support . . . 523
26.3.4.4.1 Filtering debug messages . . . 523
26.3.4.4.2 Debug message handling . . . 523
26.3.5 Transmission handling . . . 524
26.3.5.1 Transmit pause . . . 525
26.3.5.2 Dedicated TX buffers . . . 525
26.3.5.2.1 Addressing dedicated TX buffers . . . 525
26.3.5.3 Transmit FIFO . . . 526
26.3.5.4 TX queue . . . 526
26.3.5.5 Mixed dedicated TX buffers/TX FIFO . . . 527
26.3.5.6 Mixed dedicated TX buffers/TX queue . . . 527
26.3.5.7 Transmit cancellation . . . 528
26.3.5.8 Transmit event handling . . . 528
26.3.6 FIFO acknowledge handling . . . 528
26.3.7 Configuring the CAN bit timing . . . 529
26.3.7.1 CAN bit timing . . . 529
26.3.7.2 CAN bit rate . . . 531
26.4 Message RAM (MRAM) . . . 532

Table of contents

26.4.1 Message RAM configuration . . . 532
26.4.2 RX buffer and FIFO element . . . 533
26.4.3 TX buffer element . . . 536
26.4.4 TX event FIFO element . . . 539
26.4.5 Standard message ID filter element . . . 540
26.4.6 Extended message ID filter element . . . 542
26.4.7 Trigger memory element . . . 544
26.4.8 RAM watchdog (RWD) . . . 546
26.5 Time trigger CAN . . . 547
26.5.1 Reference message . . . 547
26.5.2 TTCAN configuration . . . 548
26.5.2.1 TTCAN timing . . . 548
26.5.2.2 Message scheduling . . . 549
26.5.2.3 Trigger memory . . . 550
26.5.2.3.1 Trigger types . . . 551
26.5.2.3.2 Restrictions for the node's trigger list . . . 552
26.5.2.3.3 Examples of trigger handling . . . 552
26.5.2.3.4 Detection of configuration errors . . . 554
26.5.2.4 TTCAN schedule initialization . . . 554
26.5.2.4.1 Time slaves . . . 554
26.5.2.4.2 Potential time master . . . 554
26.5.3 TTCAN gap control . . . 555
26.5.4 Stop watch . . . 556
26.5.5 Local time, cycle time, global time, and external clock synchronization . . . 556
26.5.6 TTCAN error level . . . 559
26.5.7 TTCAN message handling . . . 560
26.5.7.1 Reference message . . . 560
26.5.7.2 Message reception . . . 561
26.5.7.3 Message transmission . . . 561
26.5.7.3.1 Transmission in exclusive time windows . . . 562
26.5.7.3.2 Transmission in arbitrating time windows . . . 562
26.5.7.3.3 Transmission in merged arbitrating time windows . . . 562
26.5.8 TTCAN interrupt and error handling . . . 563
26.5.9 Level 0 . . . 563
26.5.9.1 Synchronizing . . . 564
26.5.9.2 Handling error levels . . . 564
26.5.9.3 Master slave relation . . . 565
26.5.10 Synchronization to external time schedule . . . 565
26.6 Setup procedures . . . 566
26.6.1 General program flow . . . 566
26.6.2 Clock stop request . . . 567
26.6.3 MRAM OFF operation . . . 568

26.6.4 MRAM ON operation . . . 569
26.6.5 Procedures specific to M_TTCAN channel . . . 569
26.6.6 CAN Bus configuration . . . 571
26.6.7 Message RAM configuration . . . 571
26.6.8 Interrupt configuration . . . 574
26.6.9 Transmit frame configuration . . . 575
26.6.10 Interrupt handling . . . 576
26.7 Registers summary . . . 583

27 Serial Communication Block (SCB) . . . 584
27.1 Features . . . 584
27.2 Architecture . . . 584
27.2.1 Trigger interface . . . 584
27.2.2 Clocking modes . . . 585
27.2.3 Buffer modes . . . 586
27.2.3.1 FIFO mode . . . 586
27.2.3.2 EZ mode . . . 586
27.2.3.3 CMD RESP mode . . . 586
27.2.4 Block operation . . . 587
27.2.4.1 Block enable and recovery . . . 587
27.2.4.2 Initialization flow after reset . . . 587
27.2.5 Power modes . . . 587
27.3 Serial peripheral interface (SPI) . . . 588
27.3.1 Features . . . 588
27.3.2 General description . . . 588
27.3.3 SPI modes of operation . . . 590
27.3.3.1 Motorola SPI . . . 590
27.3.3.2 Texas Instruments SPI . . . 592
27.3.3.3 National semiconductors SPI . . . 594
27.3.4 SPI buffer modes . . . 595
27.3.4.1 FIFO mode . . . 596
27.3.4.2 EZSPI mode . . . 596
27.3.4.3 Command-response mode . . . 599
27.3.5 Clocking and oversampling . . . 601
27.3.5.1 Clock modes . . . 601
27.3.5.2 Using SPI master to clock slave . . . 602
27.3.5.3 Oversampling and bit-rate . . . 602
27.3.6 SPI master SELECT output timing control . . . 604
27.3.7 SPI parity functionality . . . 606
27.3.8 SPI SSEL lines . . . 606
27.3.9 Loop-back . . . 606
27.3.10 Enabling and initializing SPI . . . 607
27.3.11 Fast SPI Master/Slave support . . . 607

Table of contents

27.3.12 I/O pad connection . . . 607
27.3.12.1 SPI master . . . 607
27.3.12.2 SPI slave . . . 609
27.3.13 Registers summary . . . 611
27.4 UART . . . 611
27.4.1 Features . . . 611
27.4.2 General description . . . 612
27.4.3 UART modes of operation . . . 613
27.4.3.1 Standard protocol . . . 613
27.4.3.1.1 Parity . . . 614
27.4.3.1.2 Start skipping . . . 615
27.4.3.1.3 Half-duplex mode . . . 616
27.4.3.1.4 Break detection . . . 617
27.4.3.1.5 UART flow control . . . 618
27.4.3.1.6 Multi-processor mode . . . 618
27.4.3.2 SmartCard (ISO 7816) . . . 620
27.4.3.3 IrDA . . . 621
27.4.3.4 UART local interconnect network (LIN) mode . . . 622
27.4.4 Clocking and oversampling . . . 624
27.4.5 Enabling and initializing UART . . . 625
27.4.6 I/O pad connection . . . 625
27.4.6.1 UART mode . . . 625
27.4.6.1.1 Standard UART mode . . . 625
27.4.6.1.2 Half-duplex mode . . . 627
27.4.6.2 SmartCard mode . . . 628
27.4.6.3 IrDA mode . . . 628
27.4.6.4 LIN mode . . . 629
27.4.7 Registers summary . . . 629
27.5 Inter-Integrated Circuit (I2C) . . . 629
27.5.1 Features . . . 630
27.5.2 General description . . . 630
27.5.3 Terms and definitions . . . 631
27.5.3.1 Clock stretching . . . 632
27.5.3.2 Bus arbitration . . . 632
27.5.4 I2C modes of operation . . . 632
27.5.4.1 Write transfer . . . 633
27.5.4.2 Read transfer . . . 634
27.5.5 I2C buffer modes . . . 634
27.5.5.1 FIFO mode . . . 634
27.5.5.2 EZI2C mode . . . 636
27.5.5.3 Command-response mode . . . 638
27.5.6 Clocking and oversampling . . . 639

27.5.6.1 Glitch filtering . . . 641
27.5.6.2 Oversampling and bit rate . . . 642
27.5.7 Loop-back . . . 643
27.5.8 Enabling and initializing I2C . . . 643
27.5.8.1 Configuring for I2C FIFO mode . . . 643
27.5.8.2 Configuring for EZ and CMD_RESP modes . . . 644
27.5.9 I/O pad connection . . . 644
27.5.10 4-wire I2C feature . . . 645
27.5.10.1 Configurable inversion of 4-wire I2C outputs . . . 645
27.5.11 Registers summary . . . 646
27.6 Timeout Generation Support (TGS) . . . 646
27.6.1 Key features of TGS functionality . . . 646
27.6.2 Functional description . . . 646
27.6.3 Intended use case . . . 648
27.6.4 Special cases and protocol-specific behavior . . . 648
27.7 Multiple Slave Address(MSA) . . . 650
27.8 SCB interrupts . . . 651
27.8.1 SPI interrupts . . . 651
27.8.2 UART interrupts . . . 652
27.8.3 I2C interrupts . . . 654

28 Improved inter integrated circuit (I3C) . . . 656
28.1 I3C overview . . . 656
28.1.1 Block diagram . . . 656
28.2 Features . . . 657
28.3 Terms and definitions . . . 658
28.4 I/O pad connections . . . 659
28.5 Functional description . . . 659
28.5.1 Operating modes . . . 659
28.5.2 Bus configurations . . . 660
28.5.3 Speed modes . . . 660
28.5.4 Controller data structures . . . 660
28.5.4.1 Command data structure . . . 661
28.5.4.1.1 Transfer command data structure . . . 661
28.5.4.1.2 Transfer argument data structure . . . 665
28.5.4.1.3 Short data argument data structure . . . 666
28.5.4.1.4 Address assignment command data structure . . . 667
28.5.4.2 Response data structure . . . 668
28.5.5 Target data structures . . . 670
28.5.5.1 Transmit command data structures . . . 670
28.5.5.2 Response data structure . . . 671
28.5.6 Bus communication protocols . . . 673
28.5.6.1 Single data rate (SDR) transfers . . . 673

28.5.6.1.1 Broadcast CCC write transfers . . . 673
28.5.6.1.2 Directed CCC write and read transfers . . . 675
28.5.6.1.3 Directed CCC transfer targeted to multiple targets . . . 676
28.5.6.1.4 I3C private write or read transfers . . . 677
28.5.6.1.5 I2C private write or read transfers . . . 678
28.5.6.1.6 Tx-FIFO empty and Rx-FIFO full conditions . . . 679
28.5.6.1.7 TOC and ROC bit settings for SDR transfers . . . 679
28.5.6.2 High data rate (HDR) transfers . . . 679
28.5.6.2.1 High data rate - double data rate (HDR-DDR) transfers . . . 680
28.5.6.2.2 Tx-FIFO empty and Rx-FIFO full conditions . . . 681
28.5.6.2.3 TOC and ROC bit settings for HDR transfers . . . 681
28.6 Interrupts . . . 682
28.7 I3C block initialization . . . 682
28.7.1 Initializing for I3C primary controller operation . . . 683
28.7.2 Initializing for I3C target operation . . . 684
28.7.3 Enabling the I3C block . . . 684
28.8 Using the initialized I3C in primary controller role . . . 685
28.8.1 Target address assignment . . . 685
28.8.2 Receiving IBI (all types) . . . 686
28.8.2.1 I3C Hot-join (HJ) request . . . 686
28.8.2.2 I3C controller role (CR) request . . . 687
28.8.2.2.1 CR reject control . . . 687
28.8.2.3 I3C target interrupt request (TIR) . . . 687
28.8.2.3.1 TIR reject control . . . 687
28.8.2.4 IBI queue data structure . . . 688
28.8.2.5 IBI notification control . . . 691
28.8.3 CCC read/write transfers . . . 691
28.8.4 Private read/write transfers . . . 692
28.8.5 Flow to prepare the I3C block to switch to target mode . . . 693
28.8.6 Flow for disable and FIFO/queue reset . . . 694
28.9 Using the initialized I3C in target mode . . . 694
28.9.1 Private receive (controller write) transfer in target mode . . . 694
28.9.2 Private transmit (controller read) transfer in target mode . . . 694
28.9.3 Programming flow for generating target interrupt request . . . 695
28.9.4 Programming flow for generating controller request . . . 695
28.9.5 Programming flow to disable HJ capability . . . 696
28.9.6 Programming flow to prepare the I3C block to switch to controller mode . . . 696
28.9.7 Command pipeline and aggregation of response queue threshold interrupt . . . 697
28.9.8 Error recovery flow . . . 697
28.9.8.1 T0-T5 error handling . . . 697
28.9.9 CCC updated interrupt flow . . . 698
28.9.10 Flow for disable and Tx/Rx/CMD/Response queue request . . . 698

Table of contents

28.10 List of supported CCCs . . . 699
28.10.1 Controller mode . . . 699
28.10.2 Target mode . . . 701
28.11 Power modes . . . 702
28.12 Registers summary . . . 703

29 Universal serial bus - full speed (USB-FS) . . . 713
29.1 Overview . . . 713
29.2 USB device mode . . . 714
29.2.1 Features . . . 714
29.2.2 Architecture . . . 714
29.2.2.1 USB physical layer (USB PHY) . . . 715
29.2.2.2 Serial interface engine (SIE) . . . 715
29.2.2.3 Arbiter . . . 716
29.2.2.3.1 SIE interface module . . . 716
29.2.2.3.2 CPU/DMA interface block . . . 716
29.2.2.3.3 Memory interface . . . 716
29.2.2.3.4 Arbiter logic . . . 716
29.2.3 Operation . . . 716
29.2.3.1 USB clocking scheme . . . 716
29.2.3.2 USB PHY . . . 717
29.2.3.2.1 Power scheme . . . 717
29.2.3.2.2 VBUS detection . . . 717
29.2.3.2.3 USB D+ pin pull-up enable logic . . . 718
29.2.3.2.4 Transmitter and receiver logic . . . 718
29.2.3.2.5 GPIO mode logic . . . 718
29.2.3.2.6 Link power management (LPM) . . . 718
29.2.3.3 Endpoints . . . 718
29.2.3.4 Transfer types . . . 719
29.2.3.5 Interrupt sources . . . 719
29.2.3.5.1 USB start of frame (SOF) event . . . 719
29.2.3.5.2 USB bus reset event . . . 719
29.2.3.5.3 Data endpoint interrupt events . . . 720
29.2.3.5.4 Control endpoint interrupt event . . . 720
29.2.3.5.5 Link power management (LPM) event . . . 720
29.2.3.5.6 RESUME interrupt . . . 720
29.2.3.5.7 Arbiter interrupt event . . . 721
29.2.3.6 DMA support . . . 722
29.2.4 Logical transfer modes . . . 722
29.2.4.1 Manual memory management with no DMA access . . . 725
29.2.4.2 Manual memory management with DMA access . . . 728
29.2.4.3 Automatic DMA mode . . . 731
29.2.4.4 Control endpoint logical transfer . . . 734

29.2.5 USB power modes . . . 737
29.3 USB host mode . . . 738
29.3.1 Features . . . 738
29.3.2 Architecture . . . 738
29.3.2.1 USB physical layer (USB PHY) . . . 738
29.3.2.2 Clock control block . . . 739
29.3.2.3 Interrupt control block . . . 739
29.3.2.4 Endpoint n (n=1, 2) . . . 739
29.3.2.5 DMA request (DREQ) control . . . 739
29.3.3 USB host operations . . . 739
29.3.3.1 Detecting device connection . . . 740
29.3.3.2 Obtaining transfer speed of the USB device . . . 740
29.3.3.3 USB bus reset . . . 741
29.3.3.4 USB packets . . . 742
29.3.3.4.1 Token packet . . . 743
29.3.3.4.2 Data packet . . . 748
29.3.3.4.3 Handshake packet . . . 748
29.3.3.5 Retry function . . . 748
29.3.3.6 SOF interrupt . . . 749
29.3.3.7 Error status . . . 751
29.3.3.8 End of package . . . 752
29.3.3.9 Interrupt source . . . 752
29.3.3.10 DMA transfer function . . . 754
29.3.3.10.1 Packet transfer mode . . . 754
29.3.3.10.2 Automatic data transfer mode . . . 756
29.3.3.11 NULL transfer function . . . 758
29.3.3.12 Suspend and resume operations . . . 759
29.3.3.12.1 Suspend operation . . . 759
29.3.3.12.2 Resume operation . . . 759
29.3.3.12.3 DeepSleep operation . . . 760
29.3.3.13 Device disconnection . . . 760
29.4 USB registers . . . 760

30 Coordinate rotation digital computer (CORDIC) . . . 764
30.1 Features . . . 764
30.2 Architecture . . . 765
30.2.1 Block diagram . . . 765
30.3 Functional description . . . 766
30.3.1 Operation . . . 766
30.3.2 Normalized result data . . . 767
30.3.3 Operating modes . . . 768
30.3.4 Domains of convergence . . . 769
30.3.5 Overflow considerations . . . 770

30.3.6 Data format . . . 770
30.3.7 Performance . . . 770
30.3.8 Look-up tables . . . 771
30.3.9 Interrupt generation . . . 772
30.4 Power, clock, and reset . . . 773
30.4.1 Power supply . . . 773
30.4.2 Clock . . . 773
30.4.3 Reset . . . 773
30.5 Registers summary . . . 773

31 High-performance programmable analog subsystem (HPPASS) . . . 775
31.1 Features . . . 775
31.2 Architecture . . . 777
31.2.1 Infrastructure (INFRA) . . . 779
31.2.1.1 Startup . . . 779
31.2.1.2 Input trigger . . . 782
31.2.2 SAR ADC . . . 783
31.2.2.1 Channels . . . 784
31.2.2.2 Sequencer . . . 787
31.2.2.3 MUX routing control . . . 788
31.2.2.4 Illustration of triggering, sampling and conversion priority . . . 790
31.2.2.5 Error status . . . 793
31.2.2.6 Post processing . . . 793
31.2.2.7 Result handling . . . 797
31.2.2.8 SAR ADC interrupts . . . 798
31.2.2.9 Calibration . . . 798
31.2.3 Comparator and slope generator (CSG) . . . 799
31.2.3.1 10-bit DAC . . . 801
31.2.3.2 DAC modes . . . 803
31.2.3.3 DAC mode start/stop . . . 806
31.2.3.4 DAC update triggers . . . 807
31.2.3.5 Hysteresis . . . 808
31.2.3.6 Post processing . . . 809
31.2.3.7 CSG interrupts . . . 815
31.2.4 3P3Z Filter (3P3ZFILT) . . . 816
31.2.4.1 Overview . . . 816
31.2.4.2 Trigger filter processing . . . 818
31.2.4.3 Anti-Windup . . . 819
31.2.4.4 Data path connection . . . 820
31.2.5 Digital to analog converter (DAC) . . . 821
31.2.5.1 Features . . . 821
31.2.5.2 DAC Digital . . . 822
31.2.5.2.1 LUT . . . 824

Table of contents

31.2.5.2.2 FIFO . . . 824
31.2.5.2.3 ADC Loop back control . . . 825
31.2.5.2.4 DAC input modes . . . 825
31.2.5.2.5 DAC Mode Start/Stop . . . 828
31.2.5.2.6 DAC update triggers . . . 829
31.2.5.2.7 DAC BUF analog data interface . . . 829
31.2.6 Operational Amplifiers (OpAmps) . . . 830
31.2.6.1 OpAmp digital . . . 831
31.2.6.2 OpAmp configurations . . . 834
31.2.7 Autonomous controller (AC) . . . 839
31.2.7.1 Overview . . . 839
31.2.7.2 Block interface . . . 839
31.2.7.3 State transition table . . . 840
31.2.7.4 Initialization . . . 844
31.2.7.5 Examples . . . 844
31.2.8 Temperature sensor . . . 847
31.3 Clock . . . 849
31.4 Output interrupts and triggers . . . 850
31.5 Registers summary . . . 852

32 Low-power comparator (LPCOMP) . . . 858
32.1 Features . . . 858
32.2 Architecture . . . 858
32.2.1 Input configuration . . . 859
32.2.2 Output and interrupt configuration . . . 859
32.2.3 Power mode and speed configuration . . . 861
32.2.4 Hysteresis . . . 862
32.2.5 Wake up from low-power modes . . . 862
32.2.6 Comparator clock . . . 862
32.2.7 Using low-power comparator . . . 862
32.3 Registers summary . . . 863

Revision history . . . 864

Disclaimer . . . 865

1 Introduction

This document provides information about Infineon's PSC3M/P6 MCU series of microcontrollers designed for industrial applications. These microcontrollers feature an Arm® Cortex®-M33 CPU and integrated programmable peripherals. They are specifically designed for real-time applications and are manufactured using an advanced 40-nm process. In addition, they offer multiple low-power modes, robust security features, a flexible clocking system, rich communication interfaces, and advanced digital and analog peripherals. They are also compatible with industry-standard development tools.

1.1 Detailed features

This device has the following features:

- CPU subsystem
  - Arm® Cortex®-M33 running up to 180 MHz
  - Digital signal processor (DSP), floating-point unit (FPU), memory protection unit (MPU), 16 KB I-cache
  - 2x 16-channel direct memory access (DMA) controllers with configurable cyclic redundancy check (CRC)
- Memory
  - On-chip flash with ECC support
    - Up to 512-KB flash with read while write (RWW) capability, 64 KB ROM for boot code, and bootloader functions
    - Built-in device firmware upgrade (DFU) support in boot ROM via serial interface (UART/I2C/SPI)
  - Up to 128 KB SRAM with ECC support
    - Full SRAM available in Deep Sleep
    - SRAM data path is protected with a hardware mechanism (ECC) for soft error detection and correction
- Security
  - Platform security architecture level 3 (PSA L3) ready
  - TrustZone framework that establishes an isolated device root of trust (RoT) for trust attestation and software management
  - Step-wise authentication of execution images until the control is handed over to the user code
  - Image authentication and integrity check
  - Cryptography accelerator
    - Hardware acceleration for symmetric (AES-128, AES-256) and asymmetric cryptographic algorithms (RSA and elliptic curve cryptography (ECC)) supported by vector unit (VU) and hash functions (SHA-256, SHA-384, SHA-512)
    - Signature verification: ECDSA NIST P-256, ECDSA NIST P-384 (Default), ECDSA NIST P-521 , LMS SHA256_M32_H10, XMSS-SHA2_10_256
    - True random number generator (TRNG) function
  - Post-quantum cryptography (PQC) algorithm support for ML_DSA and ML_KEM using wolfSSL crypto library
  - DICE ready: Device identifier composition engine (DICE) mechanism support
- Clocking subsystem
  - 8 MHz IMO with Deep Sleep operation offering ±2% accuracy
  - 48 MHz internal high-frequency oscillator (IHO) offering ±1% accuracy
  - 32 kHz internal low frequency oscillator (ILO) offering ±10% accuracy

- 4 to 36 MHz external crystal oscillator (ECO) support
- 32.768 kHz external watch crystal oscillator (WCO) usable for real-time clock (RTC)
- External clock (EXTCLK): Maximum frequency 80 MHz
- One frequency lock loop (FLL) with 24-100 MHz output range
- One digital phase-locked loops, DPLL#0, with 25-240 MHz output range

• Low power (1.71 V to 3.63 V) operation
- Six power modes (Active, Sleep, Deep Sleep, Deep Sleep-RAM, Deep Sleep-OFF, and Hibernate) for fine-grained power management
- Four power profiles (Ultra Low Power, Medium Frequency, Low Power, Overdrive) to optimize speed vs power consumption in Active mode

• Communication peripherals
- Serial communication blocks (SCBs)
  - 6x independent run-time-reconfigurable SCBs; each software-configurable as I2C, SPI, or UART in master or slave mode
  - 1x SCB supports Deep Sleep operation and wake-up from Deep Sleep in I2C slave and SPI slave modes
  - 1x SCB supports fast SPI operation at up to 50 MHz
  - All SCBs support single-wire, half-duplex UART mode
  - 1x SCB additionally supports PMBus protocol:
    - Baud rates of up to 1 Mbps
    - Timeout event generation (TGS) for wrong or stuck commands
    - Hardware address screening with support for a minimum of five slave address masks (MSAs) to reject or NACK incoming address commands
    - Compliance with PMBus (v1.4) and SMBus (v3.2) specifications
- CAN FD
  - Up to 2x CAN FD; 1 instance supports 8 Mbps operation
- Up to 1x Universal serial bus full-speed host and device with USB PHY
- 1x I3C interface with over-voltage-tolerant (OVT) I/O support

• High-performance, programmable analog subsystem (HPPASS)
- Analog-to-digital converter (ADC)
  - 1x 12-bit, 9-Msps SAR ADC with support up to 24 analog inputs
  - 16x parallel sample/hold (S/H) stages supporting simultaneous idle sampling of up to 16 inputs; 12x direct samplers and 4x multiplexed samplers
  - Up to 8 additional GPIOs can be used as ADC inputs
  - 4x multiplexed samplers can be connected to:
    - Dedicated analog pads
    - Analog-capable GPIOs
    - OpAmp outputs
    - Internal references and internal temperature sensor
  - Configurable input gain of 1×, 3×, 6×, and 12× on all 16 S/H circuits
  - Up to 4x OpAmps for signal amplification and conditioning
  - 2x 12-bit R2R DACs. DAC output can be connected to OpAmp inputs or to dedicated analog pin

        - Digital comparator at the output to compare the ADC result against programmed boundary values
        - Digital comparator outputs can be connected to timer/counter pulse-width modulator (TCPWM) (low latency between the modules)
    - Comparator with slope generator (CSG)
        - Up to 9x Active comparators without Deep Sleep functionality, each with a 10-bit DAC to generate the comparator reference
        - Each comparator supports an external reference/threshold through pins
        - Active comparator can be used with the built-in DAC in Hysteresis mode
        - Comparator outputs can be brought to pins for control loop applications
        - Comparator outputs can be connected to TCPWM (low latency between the modules)
        - Logical OR of multiple comparator trigger outputs connected as an input trigger to TCPWM via triggerMUX
- Low-power comparator (LPComp):
    - 2x comparators in LPComp are available in Active, Deep Sleep, and Hibernate modes
    - Comparator inputs and outputs can be connected to external pins; outputs can also be routed internally to TCPWM
    - Option to internally route same signal to multiple comparator inputs
    - Internal fixed reference voltage of 0.6 V ± 0.2 V can be connected to inverting input of the comparators
- Real-time control peripherals
    - Coordinate rotation digital computer (CORDIC)
        - Supports all CORDIC operating modes for solving circular (trigonometric), hyperbolic functions, and integrated independent lookup tables to accelerate calculation
    - Timer/counter pulse-width modulator (TCPWM)
        - 16 x 16-bit TCPWM channels
        - 8 x 32-bit TCPWM channels supporting high-resolution PWM generation (HRPWM) for PWM outputs
        - Center-aligned, edge, and pseudorandom modes
        - Comparator-based triggering of kill signals
        - Shadow update of duty, period, dead-time, output signal polarity, and dithering (pseudorandom mode)
        - Multichannel control: In a group of eight TCPWM channels, one channel within a group can trigger another channel
        - Ability to logically combine the outputs of multiple channels through Smart I/O
        - Dedicated output triggers mux in a group to allow flexibility to the PWM channel as a trigger and/or gate signals to the HPPASS
        - HRPWM feature for period, duty, and dead-time insertion
        - 2x motion interface (MOTIF) supporting:
            - Standalone multi-channel mode

- Hall sensor interface with autonomous BLDC block commutation support
- Quadrature encoder interface to decode motor speed and rotor position

- I/O subsystem
  - Programmable GPIO pins
    - Up to 86 functional pins (70 GPIOs; 8 out of 70 GPIOs can be used for analog inputs + 16 dedicated analog-only inputs)
    - Programmable drive modes, strengths, and slew rates
  - Programmable digital
    - Up to 7 Smart I/O-capable ports (I/Os, 56 LUTs) enable Boolean operations on I/O signals

1.2 Architecture

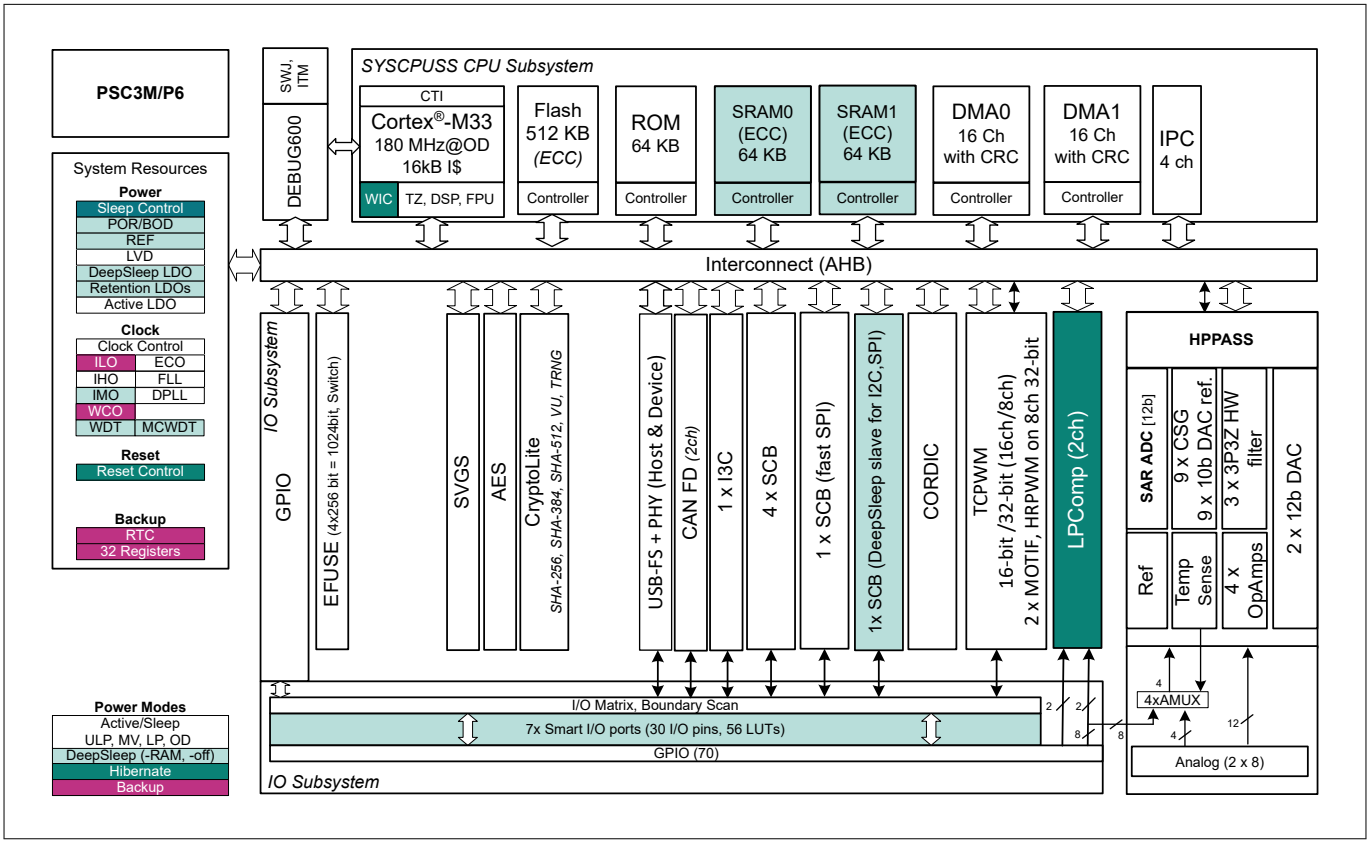


Figure 1 Functional block diagram

1.3 Document organization and conventions

1.3.1 Major sections

The information in this reference manual is organized as follows:

- **Chapter:** Presents the chapters specific to an individual aspect of the device. These are the detailed implementation and use information for some aspect of the integrated circuit
- **Registers summary:** Summary of registers associated with the chapter
- **Registers Reference Manual:** Supplies all device register details summarized in the reference manual. This is an additional document

1.3.2 Document conventions

1.3.2.1 Register conventions

Register conventions are detailed in the Registers Reference Manual.

1.3.2.2 Numeric naming

Hexadecimal numbers are represented with all letters in uppercase with an appended lowercase ‘h’ (for example, ‘14h’ or 3Ah) and hexadecimal numbers may also be represented by a ‘0x’ prefix, the C coding convention. Binary numbers have an appended lowercase ‘b’ (for example, 01010100b or 01000011b’). Numbers not indicated by an ‘h’ or ‘b’ are decimal.

1.3.2.3 Units of measurement

Table 1 lists the units of measurement used in this document.

Table 1 Units of measurement

Abbreviation	Units of measurement
bps	bits per second
°C	degrees celsius
dB	decibels
dBm	decibel-milliwatts
fF	femto farads
G	Giga
Hz	Hertz
k	kilo, 1000
K	kilo, 2^10
KB	1024 bytes, or approximately 1000 bytes
Kbit	1024 bits
kHz	kilohertz (32.000)
kΩ	kilo-ohms
MHz	megahertz
MΩ	megaohms
µA	microamperes
µF	microfarads
µs	microseconds
µV	microvolts
µVrms	microvolts root-mean-square
mA	milliamperes

(table continues...)

Table 1 (continued) Units of measurement

Abbreviation	Units of measurement
ms	milliseconds
mV	millivolts
nA	nanoamperes
ns	nanoseconds
nV	nanovolts
Ω	ohms
pF	picofarads
pp	peak-to-peak
ppm	parts per million
SPS	samples per second
σ	sigma: one standard deviation
V	Volts

1.3.2.4 Acronyms and abbreviations

Table 2 lists the acronyms and abbreviations used in this document.

Table 2 Acronyms and abbreviations

Acronym	Definition
AC	Autonomous controller
ACK	Acknowledge
ADC	Analog-to-digital converter
AES	Advanced Encryption Standard
AF	Analog filter
AHB	AMBA high-performance bus, an Arm® data transfer bus
AIRCR	Application Interrupt and Reset Control Register
AMBA	Advanced microcontroller bus architecture
ARB	Arbitration
BGR	Band gap reference
BOD	Brown-out detect
CAN	Controller area network
CAN-FD	Controller area network flexible data-rate
CC0	Capture/Compare0
CC1	Capture/Compare1

(table continues...)

Table 2 (continued) Acronyms and abbreviations

Acronym	Definition
CM-33	Cortex® M33
CORDIC	Coordinate Rotation Digital Computer
CPU	Central processing unit
CPUSS	CPU subsystem
CSG	Comparator and slope generator
CSV	Cock supervision
CTS	Clear to Send
CTI	Cross Trigger Interface
CTM	Cross Trigger Matrix
DAC	Digital-to-analog converter
DAP	Debug access port
DDR	Double Data Rate
DF	Digital filter
DMA	Direct memory access
DMAC	Direct memory access controller
DPLL	Digital phase-locked loop
DSI	Digital system interconnect
DSP	Digital signal processing
DU	Data unit
DW	Data wire
ECC	Error correcting code
ECO	External crystal oscillator
Em_EEPROM	Emulated Electrically Erasable Programmable Read-only Memory
EOC	End of conversion
EPPB	External private peripheral us
FIFO	First in first out
FIR	Finite impulse response
FLL	Frequency locked loop
FM	Flash macro
FPU	Floating point unit
FSM	Finite-state machine
GPIO	General-purpose input output
HPPASS	High-performance programmable analog subsystem

(table continues...)

Table 2 (continued) Acronyms and abbreviations

Acronym	Definition
HRPWM	High-resolution PWM
HSIOM	High-speed I/O matrix
I2C	Inter-integrated circuit
I3C	Improved Inter-Integrated Circuit
IBI	In-band Interrupts
I-Cache	Instruction cache
ICER	Interrupt clear enable register
IDAU	Implementation defined attribution unit
IHO	Internal high-speed oscillator
ILO	Internal low-speed oscillator
IMO	Internal main oscillator
INFRA	Infrastructure
IOSS	I/O subsystem
IPC	Inter processor communication
IRQ	Interrupt request
ISER	Interrupt set enable register
ISR	Interrupt service routine
ITNS	Interrupt target non-secure
LCS	Life cycle stage
LDO	Low dropout
LFCLK	Low frequency clock
LIN	Local interconnect network
LP	Low power
LPCOMP	Low-power comparator
LPM	Link Power Management
LUT	Look-up table
LVD	Low-voltage detect
MCU	Microcontroller unit
MCWDT	Multi-counter watchdog timer
MF	Medium frequency
MISO	Master in slave out
MMIO	Memory-mapped I/O
MOSI	Master out slave in

(table continues...)

Table 2 (continued) Acronyms and abbreviations

Acronym	Definition
MOTIF	Motion interface
MPC	Memory protection controller
MPS	Magnitude PreScaler
MPU	Memory protection unit
MSC	Master security controller
MSP	Main stack pointer
MTQ	Minimum time quantum
NACK	No acknowledgment
NMI	Non-maskable interrupt
NVIC	Nested vectored interrupt controller
NVM	Nonvolatile memory
OD	Overdrive
OTP	One-time programmable
OV	Over flow
PC	Protection context
PDL	Peripheral driver library
PendSV	Pending supervisory
PERI	Peripheral interconnect
PID	Protected identifier
PLL	Phase-locked loop
POR	Power-on reset
PPB	Private peripheral bus
PPC	Peripheral protection controller
PPU	Power policy unit
PSP	Process stack pointer
PSR	Program and status register
PWM	Pulse width modulation
PWMDT	Pulse width modulation with dead time insertion
RAM	Random access memory
RMA	Return merchandise authorization
RTC	Real time clock
RTS	Request to send
RX	Receive

(table continues...)

Table 2 (continued) Acronyms and abbreviations

Acronym	Definition
SAR	Successive approximation register
SAU	Secure attribution unit
SCB	Serial communication block
SCL	Serial clock
SDA	Serial data
SE0	Single-ended zer
SFR	Special function register
SHA	Secure hash algorithm
SIE	Serial interface engine
SoC	System on-chip
SPI	Serial peripheral interface
SRAM	Static random access memory
SRAMC	Static random access memory controller
SRSS	System resource subsystem
SVCall	Supervisory call
SWD	Single-wire debug
TC	Terminal count
TCPWM	Timer counter pulse width modulation
TQ	Time quantum
TRNG	True random number generator
TX	Transmit
UART	Universal asynchronous receiver/transmitter
UART_MP	UART multi-processor
ULP	Ultra low power
UN	Underflow
USB	Universal serial bus
VTOR	Vector table offset register
VU	Vector unit
WCO	Watch crystal oscillator
WDT	Watchdog timer
WFE	Wait for event
WFI	Wait for interrupt
WIC	Wake-up interrupt controller

(table continues...)

Table 2 (continued) Acronyms and abbreviations

Acronym	Definition
WS	Wait states
XRES	External reset

2 CPU subsystem (CPUSS)

The CPU subsystem uses the 32-bit Arm® Cortex®-M33 (CM33). It provides security, control, communication, short interrupt-response time, high code density, and high 32-bit throughput while maintaining a strict cost and power-consumption budget. This section provides an overview of the Arm® Cortex® CPU in the PSC3M/P6 MCU. For details, see the Arm® Cortex®-M33 documentation set.

2.1 Features

The PSC3M/P6 MCU Arm® Cortex®-M33 (CM33) CPU has the following features:

- Maximum clock frequency of 180 MHz in Overdrive (OD) mode
- Single-precision floating-point unit (FPU) and DSP extension
- Memory protection unit (MPU) with up to 8 regions per security state (secure and non-secure)
- Nested vectored interrupt controller (NVIC) integrated with the CPU core, yielding low interrupt latency
- Wakeup interrupt controller (WIC) that allows the CPU to wake from the Deep Sleep power mode through Deep Sleep interrupts
- 16 KB instruction cache (I-cache) on the Cortex®-M33 C-AHB bus with four-way set associativity
- TrustZone-M memory mapping support for secure/non-secure aliasing based on the programmable Secure Attribution Unit (SAU) and a fixed Implementation Defined Attribution Unit (IDAU)

2.2 Architecture

CM33 is a 32-bit processor with a 32-bit data path, 32-bit registers, and a 32-bit memory interface. The CPU supports a wide range of instructions in the Thumb instruction set. The CM33 CPU supports Thread and Handler operating modes (see Operating modes and privilege levels). The CM33 CPU also supports secure and non-secure security states based on the security extension.

The CM33 instruction set includes:

- Signed and unsigned 32 × 32 multiplies with 32-bit or 64-bit results, and multiply-accumulate operations, all single cycle
- Signed and unsigned 32-bit division that takes 2 to 11 cycles depending on operand values
- DSP instructions
- Complex load and store memory access
- Complex bit manipulation

CM33 with FPU supports single-precision arithmetic as defined by the FPv5 architecture. It complies with ANSI/IEEE standard 754-2008, the IEEE standard for binary floating-point arithmetic.

2.2.1 CPUSS

The PSC3M/P6 CPUSS is a security, control, and communication CPU based on a Cortex®-M33 (CM33) core operating up to 180 MHz. Built on the Armv8-M architecture, the CM33 includes a 3-stage pipeline (3PIP) and a low-power domain with rapid clock switching. The CPUSS provides Arm® TrustZone-M compatible security and protection infrastructure and functions as the initial boot source for security and protection mechanisms.

2.2.1.1 Key security features

CM33 is a TrustZone-M CPU with an integrated security extension. It includes a secure attribution unit (SAU) inside the CPU. An external implementation defined attribution unit (IDAU) interfaces with the CPU per TrustZone-M specifications. For non-CPU masters in CPUSS, a master security controller (MSC) works with an

IDAU to differentiate secure and non-secure transfers. For additional details, refer to the TrustZone section of this manual.

2.2.1.2 Memory and bus architecture

The CPUSS includes a 16 KB, four-way set-associative instruction cache on the C-AHB bus. It supports on-chip SRAM controllers and memories, providing C-AHB bus access from SRAM for code execution. It features a 128-bit AHB5 bus interface for instruction fetch and data access from flash (NVM) and internal SRAM, which are accessible at aliased addresses. A 32-bit AHB5 bus interface provides access to memory-mapped flash and internal SRAM within the system memory map.

2.2.1.3 Peripherals and interconnect

The CPUSS includes two DMA controllers, each with 16 channels, and supports Arm® Q-channel for use with SCB, CORDIC, HPPASS, USB, and TCPWM peripherals. Four inter-processor communication (IPC) blocks enable secure and non-secure applications to communicate and synchronize. A centralized fault-reporting infrastructure requires a single fault interrupt handler and captures MPC and PPC violations, peripheral errors, memory controller errors such as ECC, and time-out errors.

2.2.1.4 Power and clocking

The CPUSS operates in the `clk_hf0` domain, which runs up to 180 MHz at 1.2 V. This domain powers most hardware modules, including the I-cache, AHB infrastructure, DMA, MSC and IDAU and MPC, and on-chip memory controllers operating at a common synchronous frequency.

2.2.1.5 Power-on behavior

Upon power-on reset (POR), the active power domain initializes first, as described in the device power modes section.

2.2.2 Address and memory maps

The CPU uses a fixed address map. The default memory map covers the range 0x00000000 to 0xFFFFFFFF. For details on the default memory map, see the memory model section of the Arm® Cortex®-M33 Processor Technical Reference Manual. This 32-bit (4 GB) address space is partitioned into non-secure and secure memory regions, as shown in Table 3. Based on Arm® TrustZone, bit [28] of the address defines whether a memory region is secure or non-secure, resulting in the following memory map.

Table 3 PSC3M/P6 MCU address and memory map

Access type	Address range	Region name	Bank mode	Size	Interface
Non-secure	0x22000000 - 0x2207FFFF	FLASH	Single bank	512 KB	S-bus
Secure	0x32000000 - 0x3207FFFF			512 KB	
Non-secure	0x02000000 - 0x0207FFFF		Single bank	512 KB	C-bus
Secure	0x12000000 - 0x1207FFFF			512 KB	
Non-secure	0x22000000 - 0x2203FFFF		Dual bank (main bank)	256 KB	S-bus

(table continues...)

Table 3 (continued) PSC3M/P6 MCU address and memory map

Access type	Address range	Region name	Bank mode	Size	Interface
	0x22800000 - 0x2283FFFF		Dual bank (alternate bank)	256 KB	
Secure	0x32000000 - 0x3203FFFF		Dual bank (main bank)	256 KB	
	0x32800000 - 0x3283FFFF		Dual bank (alternate bank)	256 KB	
Non-secure	0x02000000 - 0x0203FFFF		Dual bank (main bank)	256 KB	C-bus
	0x02800000 - 0x0283FFFF		Dual bank (alternate bank)	256 KB	
Secure	0x12000000 - 0x1203FFFF		Dual bank (main bank)	256 KB	
	0x12800000 - 0x1283FFFF		Dual bank (alternate bank)	256 KB	
Non-secure	0x04000000 - 0x0400FFFF	SRAM0	N/A	64 KB	C-bus
Secure	0x14000000 - 0x1400FFFF		N/A	64 KB	
Non-secure	0x24000000 - 0x2400FFFF		N/A	64 KB	S-bus
Secure	0x34000000 - 0x3400FFFF		N/A	64 KB	
Non-secure	0x04010000 - 0x0401FFFF	SRAM1	N/A	64 KB	S-bus
Secure	0x14010000 - 0x1401FFFF				
Non-secure	0x24010000 - 0x2401FFFF				C-bus
Secure	0x34010000 - 0x3401FFFF				
Non-secure	0x23400000 - 0x23407FFF	SFLASH	N/A	32 KB	S-bus
Secure	0x33400000 - 0x33407FFF		N/A	32 KB	
Non-secure	0x03400000 - 0x03407FFF		N/A	32 KB	C-bus
Secure	0x13400000 - 0x13407FFF		N/A	32 KB	
Non-secure	0x00800000 - 0x0080FFFF	Reserved	N/A	64 KB	
Secure	0x10800000 - 0x1080FFFF		N/A	64 KB	
Non-secure	0x42000000 - 0x43FFFFFF	SYSCPUSS peripheral (SYS MMIO)	N/A	32 MB	
Secure	0x52000000 - 0x53FFFFFF		N/A	32 MB	

***Note**: Secure and non-secure memory pairs use the same physical memory but are mapped differently based on their security configuration.*

Gaps in the address space are reserved and must not be accessed; accessing these gaps can cause hard faults or BUS ERROR events depending on the bus segment or peripheral associated with the attempted address. Cache coherency during protection context (PC) switches is maintained by software. For the CM33, all accessible address space in FLASH is cacheable.

Peripheral address spaces in the `0x4000_0000` and `0x5000_0000` regions are not cacheable. Execute in Place instruction fetches from these regions generate a MemFault. Address aliasing applies only to SRAM when accessed by the CM33 through the code bus to support code execution from SRAM.

***Note**: The `CM33_S_VECTOR_TABLE_BASE` and `CM33_NS_VECTOR_TABLE_BASE` registers define the secure and non-secure vector-table locations for the CM33 CPU.*

2.2.2.1 Wait state lookup tables

Table 4 specifies the required wait states for different memory modules.

Table 4 Memory wait state

Memory access		Number of wait states		Note or test condition
From	To	Min. cycle in ULP/MF mode	Min. cycle in LP/OD mode	
CM33	I-cache	0	0	-
CM33	SRAM	0	0	Additional 3 cycle with ECC enabled
CM33	Flash via C-bus (cached) Flash via S-bus (up-sizer)	4	9	Depends on cache hit/ miss, prefetch state, and access type.

2.3 Operating modes and privilege levels

The CM33 CPU supports secure and non-secure states. The CPU operates in thread and handler modes and can run in either Thumb or Debug state. The processor can also restrict or prevent access to selected resources by executing code in privileged or unprivileged mode. For details on operation and execution modes, see the Armv8-M Architecture Reference Manual.

2.3.1 Security states

When the Armv8-M security extension is included in the processor, the programmer model supports two orthogonal security states: secure and non-secure. With the security extension enabled, the CM33 CPU always resets into the secure state. Registers in the system-control space are banked across secure and non-secure states, and the non-secure register view is available at an address aliased to the secure state.

2.3.2 Operating modes

For each security state, the processor operates in Thread or Handler mode.

- **Thread mode**: Used to execute application software. The processor enters Thread mode on reset or after an exception return to Thread mode. Thread mode supports privileged and unprivileged code
- **Handler mode**: Used to handle exceptions. The processor enters Handler mode as a result of an exception. Handler mode supports only privileged code

The processor can change the security state when taking an exception. For example, when a secure exception is taken from the non-secure state, Thread mode transitions to the secure-state Handler mode. The processor can

call secure functions from the non-secure state and non-secure functions from the secure state. The security extension defines requirements for these calls to prevent secure data from being accessed in the non-secure state.

2.3.3 Operating modes

The processor operates in Thread or Handler mode.

- **Thread mode**: Used to execute application software. The processor enters Thread mode on reset or after an exception return to Thread mode. Thread mode supports privileged and unprivileged code
- **Handler mode**: Used to service exceptions. The processor enters Handler mode as a result of an exception. Handler mode supports only privileged code

2.3.4 Operating states

The processor operates in Thumb or Debug state:

- **Thumb state**: Normal execution state running 16-bit and 32-bit half-word-aligned Thumb instructions
- **Debug state**: State entered when the processor is in halting debug

2.3.5 Privilege levels

Privilege levels define the resources and instructions that software can access.

- **Unprivileged**: Software has limited access to MSR and MRS instructions and cannot use CPSID or CPSIE. It cannot access the system timer, NVIC, or system control block. It also has restricted access to memory or peripherals
- **Privileged**: Software can use all instructions and access all system resources

In Thread mode, the CONTROL register determines whether software executes in privileged or unprivileged mode. In Handler mode, software always executes in privileged mode. Only privileged software can write to the CONTROL register to change the privilege level. Unprivileged software can use the SVC instruction to transfer control to privileged software. In Handler mode, the MSP is always used. Exception entry and return mechanisms automatically update the CONTROL register, depending on whether MSP or PSP is in use.

In Thread mode, use the MSR instruction to set the stack-pointer bit in the CONTROL register. After changing the stack pointer, use an ISB instruction immediately after the MSR instruction to ensure that subsequent instructions execute using the updated stack pointer.

2.4 Instruction set

The CM33 CPU supports all 16-bit Thumb instructions defined by the Armv8-M architecture. The processor also implements the T32 instruction set based on Thumb-2 technology, providing high code density and reduced program-memory requirements.

The CM33 CPU implements the following Armv8-M instruction categories:

1. All base instructions
2. All instructions in the main extension
3. All instructions in the security extension when the security extension is enabled
4. All instructions in the DSP extension (optional)
5. All single-precision instructions and double-precision load and store instructions in the floating-point extension when the FPU is enabled

For details on these instructions, see Section 3, The Cortex®-M33 Instruction Set, in the Arm® Cortex®-M33 Devices Generic User Guide.

The processor also implements custom datapath extension (CDE) instructions. The CDE introduces 2 × 3 classes of instructions in the coprocessor instruction space:

1. Three classes operating on the general-purpose register file
2. Three classes operating on the floating-point register file

For information on the specific CDE instructions implemented in this processor, see Section 9, Arm® Custom Instructions, in the Arm® Cortex®-M33 Processor Reference Manual.

2.5 Registers summary

The CM33 CPU has the following 32-bit core registers. When the security extension is included, some of the registers are banked. The secure view of these registers is available when the CPU is in the secure state and the non-secure view when the CPU is in non-secure state.

Table 5 CM33 core register set summary

Name	Type[1]	Required privilege[2]	Reset value	Description
R0-R12	RW	Either	UNKNOWN	R0-R12 are general-purpose registers for data operations.
MSP (R13)	RW	Either	-[3]	The stack pointer (SP) is register R13. In Thread mode, the CONTROL register selects the stack pointer in use, either the main stack pointer (MSP) or the process stack pointer (PSP). When the Armv8-M security extension is included, the Cortex®-M33 processor implements two MSP and two PSP registers: • MSP_NS for the non-secure state • MSP_S for the secure state • PSP_NS for the non-secure state • PSP_S for the secure state
PSP (R13)	RW	Either	UNKNOWN	
MSPLIM	RW	Privileged	0x00000000	The stack-limit registers define how far the MSP and PSP can descend. When the Armv8-M security extension is included, the Cortex®-M33 processor provides two MSPLIM registers and two PSPLIM registers: • MSPLIM_NS for the non-secure state • MSPLIM_S for the secure state • PSPLIM_NS for the non-secure state • PSPLIM_S for the secure state
PSPLIM	RW	Privileged	0x00000000	
LR (R14)	RW	Either	UNKNOWN	The Link Register (LR) is register R14. It stores return information for subroutines, function calls, and exceptions.
PC (R15)	RW	Either	-[3]	The Program Counter (PC) is register R15. It holds the current program address.

(table continues...)

Table 5 (continued) CM33 core register set summary

Name	Type[1]	Required privilege[2]	Reset value	Description
xPSR	RW	Either	-[4]	The Program Status Register (PSR) combines the following registers: • Application Program Status Register (APSR) • Interrupt Program Status Register (IPSR) • Execution Program Status Register (EPSR) These registers provide different views of the PSR.
APSR	RW	Either	UNKNOWN	The APSR holds the current condition-flag state resulting from previous instruction executions.
IPSR	RO	Privileged	0x00000000	The IPSR holds the exception number of the current interrupt service routine.
EPSR	RO	Privileged	-[4]	The EPSR holds the Thumb-state bit, the execution-state bits for the If-Then (IT) instruction, and the interruptible-continuable instruction (ICI) field for an interrupted load-multiple or store-multiple instruction.
PRIMASK	RW	Privileged	0x00000000	The PRIMASK register prevents activation of exceptions that have configurable priority. When the Armv8-M security extension is included, the Cortex®-M33 processor provides two PRIMASK registers: • PRIMASK_NS for the non-secure state • PRIMASK_S for the secure state
BASEPRI	RW	Privileged	0x00000000	The BASEPRI register defines the minimum priority level used for exception processing. When the Armv8-M security extension is included, the Cortex®-M33 processor provides two BASEPRI registers: • BASEPRI_NS for the non-secure state • BASEPRI_S for the secure state

(table continues...)

Table 5 (continued) CM33 core register set summary

Name	Type[1]	Required privilege[2]	Reset value	Description
FAULTMASK	RW	Privileged	0x00000000	The FAULTMASK register prevents activation of all exceptions except the non-maskable interrupt (NMI) and, optionally, the secure HardFault. When the Armv8-M security extension is included, the Cortex®-M33 processor provides two FAULTMASK registers: • FAULTMASK_NS for the non-secure state • FAULTMASK_S for the secure state
CONTROL	RW	Privileged	0x00000000	The CONTROL register selects the stack pointer in use and, optionally, the privilege level when the processor operates in Thread mode. When the Armv8-M security extension is included, the Cortex®-M33 processor provides two CONTROL registers: • CONTROL_NS for the non-secure state • CONTROL_S for the secure state

1) Describes the access type during program execution in the thread mode and the handler mode. Debug access can differ.
2) An entry of either means privileged and unprivileged software can access the register.
3) Soft reset to the value retrieved by the reset handler.
4) Bit[24] is the T-bit and is loaded from bit[0] of the reset vector. All other bits are reset to 0.

The CM33 floating-point unit (FPU) includes the following registers:

- Thirty-two 32-bit single-precision registers (S0 to S31). These registers can also be accessed as sixteen 64-bit double-precision registers (D0 to D15
- The FPU also provides four control and status registers:

1. FPCCR – Floating-point Context Control Register
2. FPCAR – Floating-point Context Address Register
3. FPSCR – Floating-point Status Control Register
4. FPDSCR – Floating-point Default Status Control Register

For details on FPU register operation, see the Arm® Cortex®-M33 Reference Manual.

Use the MSR and MRS instructions to access the PSR, PRIMASK, CONTROL, FAULTMASK, and BASEPRI registers. Table 6 to Table 8 define the PSR bit assignments. For detailed information, see the Combined Program Status Register section in the Arm® Cortex®-M33 Reference Manual.

Table 6 CM33 APSR bit assignments

Bit	Name	Usage
31	N	Negative flag
30	Z	Zero flag
29	C	Carry or borrow a flag

(table continues...)

Table 6 (continued) CM33 APSR bit assignments

Bit	Name	Usage
28	V	Overflow flag
27	Q	DSP overflow and saturation flag
26:20	-	Reserved
19:16	GE[3:0]	Greater than or equal flags
15:0	-	Reserved

Table 7 CM33 IPSR bit assignments

Bit	Name	Usage
31:9	-	Reserved
8:0	Exception number	This is the number of the current exceptions: • 0 = Thread mode • 1 = Reset • 2 = NMI • 3 = HardFault • 4 = MemManage • 5 = BusFault • 6 = UsageFault • 7 = SecureFault • 8-10 = Reserved • 7-10 = Reserved • 11 = SVCall • 12 = DebugMonitor • 13 = Reserved • 14 = PendSV • 15 =SysTick • 16 = IRQ0 • 495 = IRQ479

Table 8 CM33 EPSR bit assignments

Bit	Name	Usage
31:27	-	Reserved, RES0
26:25, 15:10	ICI	Interruptible-continuable instruction bits
26:25, 15:10	IT	Indicates the execution state bits of the IT instruction
26:25, 11:10, 15:12	ECI	Exception continuation flags for beat-wise vector instructions. This field encodes which beats of the in-flight instructions have completed.
24	T	Thumb state bit
23:16	-	-

3 Bus infrastructure

The bus infrastructure in a system-on-chip (SoC) forms the backbone of communication between various components within the chip. It is a bus network that connects different IP blocks, such as processors, memory, peripherals, and accelerators. The bus infrastructure provides a standardized interface for data and control signals, enabling seamless interaction and data exchange among these components. It typically consists of different types of buses, such as system buses, interconnects, and peripheral buses, each serving a specific purpose. System buses and interconnects, such as the advanced high-performance bus (AHB), facilitate high-bandwidth communication between processors, memory, and other critical components.

The PSC3M/P6 MCU family features a SYSCPUSS, which consists of AHB5-based bus infrastructure that interconnects multiple bus masters with slaves. The bus interconnect uses bridges, upsizers, and downsizers to connect to other interconnects of varying bus widths, allowing access from multiple masters to slaves and enabling concurrent access and efficient operation even when several high-speed peripherals are working at the same time. This bus infrastructure for PSC3M/P6 devices is shown in Figure 2

Peripheral registers are accessed through the peripheral interconnect. The worst-case access latency depends on the transaction path and the clock domains of both the bus master and the peripheral. For example, the CPUSS, TCPWM, and HPPASS are all on CLK_HF0. CPUSS may require up to six CPU cycles to access the TCPWM and HPPASS registers.

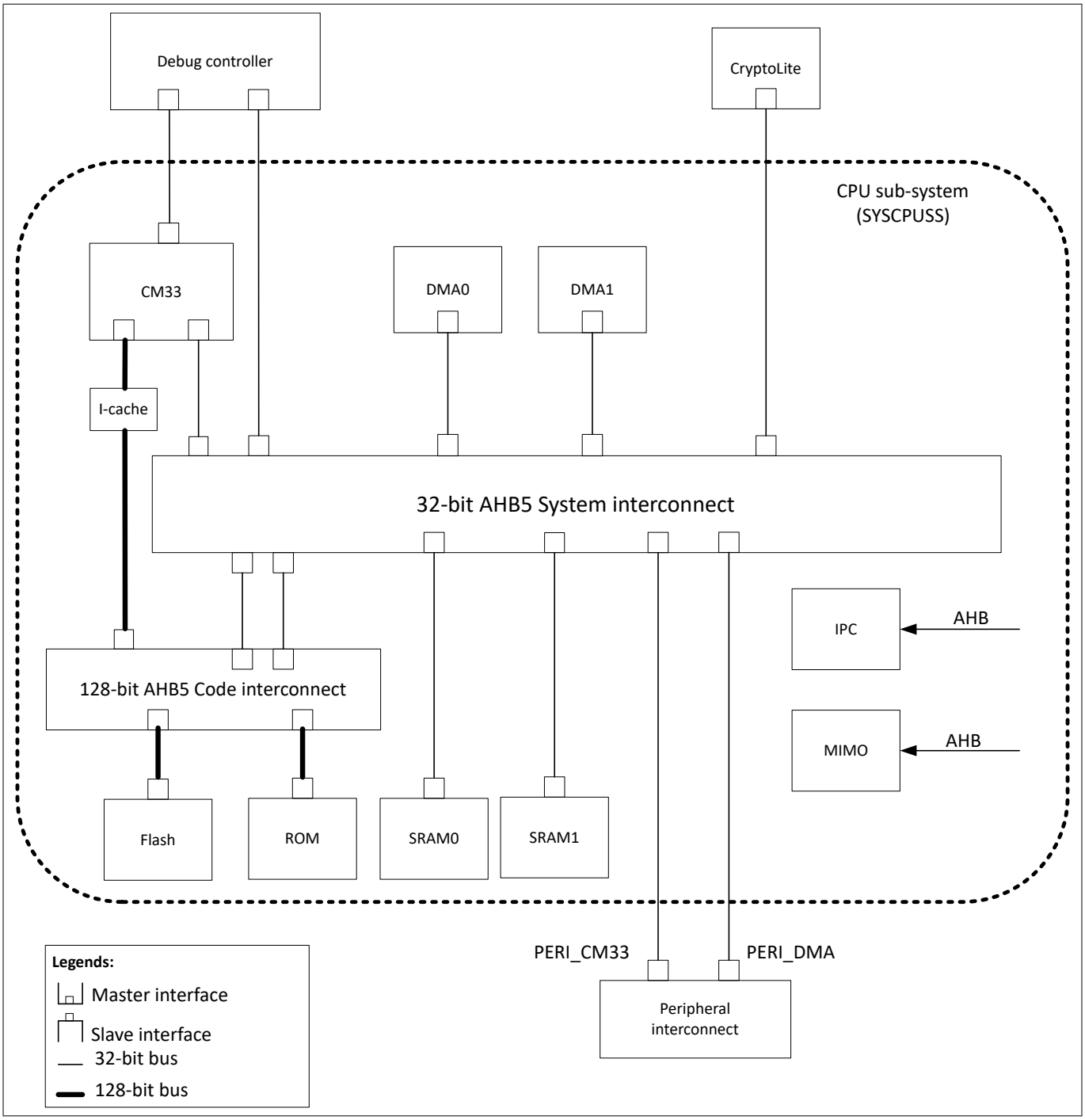


Figure 2 Bus infrastructure for PSC3M/P6 devices

3.1 SYSCPUSS AHB5 interconnect

The SYSCPUSS (CM33 CPU) uses a 128-bit and 32-bit multilayer AHB5 bus matrix interconnect. The AHB5 bus masters may reside in the SYSCPUSS or outside of the SYSCPUSS (the SYSCPUSS external bus masters). Similarly, the AHB5 bus slaves may reside inside the SYSCPUSS (for example, the on-chip memory controllers) or outside of the SYSCPUSS (for example, slaves connected to the peripheral interconnect (PERI) or to the external slave interfaces).

The SYSCPUSS AHB5 bus matrix interconnects the following masters and slaves:

- **Masters:**
  - Cortex®-M33 code AHB (C-AHB) bus via instruction cache (I-cache)
  - Cortex®-M33 slave AHB (S-AHB) bus
  - Two direct memory access – Datawire (DMA0, DMA1)
  - One cryptographic accelerator (CryptoLite)
- **Slaves:**
  - Flash controller inside SYSCPUSS
  - Two SRAM controller inside SYSCPUSS (SRAM0, SRAM1)
  - One ROM inside SYSCPUSS
  - CM33 to peripheral interconnect (PERI_CM33)
  - One AHB DMA to peripheral interconnect (PERI_DMA)

3.1.1 AHB5 bus matrix interconnect

The SYSCPUSS uses a round-robin-with latency scheme to implement bus arbitration in the AHB5 interconnect.AHB5 bus matrix interconnect outlines the SYSCPUSS AHB5 bus matrix interconnect.

Table 9 AHB5 bus matrix interconnect

Bus masters	Bus slaves					
	Flash	ROM	SRAM0	SRAM1	PERI_CM33	PERI_DMA
CM33_C-AHB	**X**	**X**	x	x	-	
CM33_S-AHB	**x**	-	X	X	X	-
DMA0	**X**	-	X	X	-	X
DMA1	**X**	-	X	X	-	X
CryptoLite	**X**	-	X	X	-	X

Where,

"**X**" = connection, bold font type denotes 128-bit bus

"-" = no connection

"x" = connection, aliased address path

"X" = connection

"**x**" = connection, aliased address path, bold font type denotes 128-bit bus

***Notes**:*

1. *Memory aliasing allows the CM33 C-AHB bus (I-cache) to access system SRAM. The CM33 S-AHB bus utilizes the aliased address path to write into flash memory, which is located in the C-AHB bus address space. Data can be read from the aliased address of flash memory, but this is not preferred because it is not cached by the CM33 I-cache*
2. *All CM33 C-AHB bus accesses are cacheable*
3. *DMA writes to aliased addresses occur using the S-AHB bus address space (above 0x20000000)*
4. *Writes to aliased addresses by the DMA or CM33 S-AHB bus must be followed by a cache-invalidation sequence to ensure that stale data does not remain in the cache*
5. *The PERI_CM33 and PERI_DMA interfaces provide access to peripherals connected via the peripheral interconnect (PERI)*

3.1.2 SYSCPU buses

3.1.2.1 Cortex®-M33 C-AHB bus

This bus connects the C-AHB bus of the Cortex®-M33 core to the bus matrix through the instruction cache. This bus is used for instruction fetch and data access to the internal and external memories mapped in the code region. The targets of this bus are the flash, internal SRAM, and ROM.

3.1.2.2 Cortex®-M33 S-AHB bus

This bus connects the S-AHB bus of the Cortex®-M33 core to the bus matrix. This bus is used by the core to access data located in a peripheral or SRAM area. The target of this bus are the aliased flash, internal SRAM, and peripherals connected via PERI_CM33.

3.1.2.3 DMA bus

This bus connects the bus matrix to the 32-bit AHB master interface of the DMA, which shares the clock with the CM33 CPU. The targets of this bus are the flash, internal SRAM, and peripherals connected via PERI_DMA.

3.2 AHB5 MMIO

The memory-mapped I/O registers for CM33, CryptoLite, DMA0/1, CPUSS, and all other peripheral configurations are accessible via the AHB5 MMIO bus interface.

3.3 Peripheral interconnect (PERI)

The peripheral interconnect (PERI) implements the AHB5 bus matrix infrastructure, connecting AHB masters to groups of peripheral devices. It enables internal and external masters of the SYSCPUSS to interface with peripherals operating in the low-power domain.

These peripheral devices include all memory-mapped registers (MMIO registers) of PERI, PCLK, and external peripheral devices, as well as synchronous and asynchronous peripherals mapped to the peripheral address space:

- 0x4200_0000 ~ 0x43FF_FFFF (Non-secure address map)
- 0x5200_0000 ~ 0x53FF_FFFF (Secure address map)

In the PSC3M/P6 MCU family, there is one PERI instance. As detailed in Table 10, PERI consists of six peripheral groups. These groups connect slave peripherals within their respective group to the SYSCPUSS and other AHB masters.

Table 10 Peripheral groups and slave peripherals

Peripheral group	Slave peripheral	Peripheral type	Description
PERI GROUP 0	Peripheral interconnect MMIO registers	Internal	SYSCPUSS platform component without I/O subsystem, is accessible at speeds of up to 180 MHz
	Peripheral protection control (PPC)		
	PERI_PCLK registers		

(table continues...)

Table 10 (continued) Peripheral groups and slave peripherals

Peripheral group	Slave peripheral	Peripheral type	Description
	APB slave group-0, SRAM0, SRAM1, PROM Controller, FLASH controller, MXCM33-0, DW-0/1, SYSCPUSS, IPC, Fault structures MMIO registers		
	SRSS core	Peripheral	
	SRSS power mode		
	SRSS backup		
	Cryptographic accelerator (CryptoLite)		
	Coordinate rotation digital computer (CORDIC)		
	DEBUG600		
	Supply voltage glitch sensor (SVGS)		
	Advanced encryption standard (AES)		
PERI GROUP 1	High-Speed I/O matrix (HSIOM)	Peripheral	Asynchronous peripherals are accessible at speeds of up to 180 MHz
	GPIO		
	Smart I/O		
	Low-power comparator (LPComp)		
PERI GROUP 2	DFT	Peripheral	
	Efuse		
PERI GROUP 3	Controller area network flexible data-rate (CAN FD)	Peripheral	Asynchronous peripherals
	Serial communication block (SPI/UART/I2C) (SCB[0:4])		
PERI GROUP 4	Timer/Counter/PWM (TCPWM)	Peripheral	1:1 peripherals are accessible at speeds of up to 180 MHz
	High-performance programmable analog subsystem (HPPASS)		
PERI GROUP 5	Serial communication block (SPI/UART/I2C) (SCB[5])	Peripheral	Asynchronous peripherals
	I3C[0]		

(table continues...)

Table 10 (continued) Peripheral groups and slave peripherals

Peripheral group	Slave peripheral	Peripheral type	Description
	USBFS		

The AHB bus interfaces of the peripheral slaves within a peripheral group all operate at the same frequency, referred to as `clk_group[i]`. To reduce power consumption, slaves can be disabled using `SL_CTL[i].ENABLED[j]`.

The trigger component handles the multiplexing of trigger signals. For more information, see Trigger multiplexer (Trigger MUX).

3.3.1 AHB bus error handling

The following bus errors are possible in the bus infrastructure:

1. AHB error response detected at the master interface
2. AHB error response from the peripheral decoder (peripheral decoder gives error response in any of the scenario's: "access to the respective slave is disabled/reset by using SL_* registers" (refer to the PSC3M/P6 MCU registers reference manual for more information on these registers) or "access to an unmapped slave address" or "error response from the respective peripheral itself")
3. No response for the AHB transaction from the peripheral for a higher number of cycles than expected

PERI uses the fault interface to report these errors.

3.4 Bus master peripherals

3.4.1 Direct memory access (DMA-DW)

The DMA0/1 can be used as an AHB master of the CPU subsystem. The DMA-DW is ideal for small data size and transactional DMA, which is typically used to send bytes between peripherals, for example, from ADC to RAM. The DMA-DW uses a 32-bit AHB bus.

3.4.2 CryptoLite

CryptoLite offers an AHB-Lite master interface, which supports 8/16/32-bit AHB-Lite transfers. The AHB-Lite master interface enables the subsystem to access operation operand data from system memories (for example, flash or SRAM memories). Memory buffer accesses can be 8/16/32-bit accesses.

3.4.3 USBFS

The USBFS peripheral interfaces with the APPCPUSS through its 32-bit AHB5 master for data transfer. It uses the AHB5 master interface to access system SRAM for DMA transfers.

4 Inter processor communication (IPC)

Inter-processor communication (IPC) provides the functionality for secure applications and non-secure applications to communicate and synchronize their activities. IPC hardware is implemented using two register structures.

- IPC Channel: Communication and synchronization between processors are achieved using this structure
- IPC Interrupt: Each interrupt structure configures an interrupt line, which can be triggered by a ‘notify’ or ‘release’ event of any IPC channel

The channel and interrupt structures are independent and have no correlation to each other, as shown in Figure 3. This allows for building varying models of interfaces, as shown in the Typical usage models.

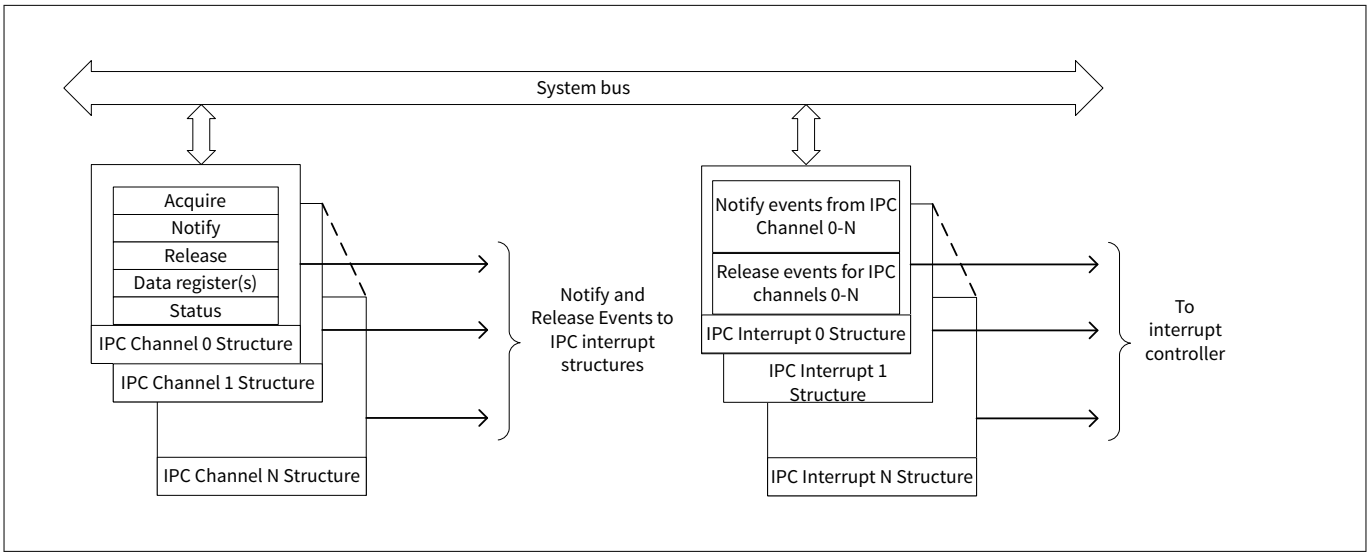


Figure 3 IPC register architecture

4.1 Features

The features of IPC are as follows:

- Implements locks for mutual exclusion and synchronization between secure and non-secure applications
- Allows sending messages between secure and non-secure applications
- Supports up to 4 channels for communication
- Supports up to 2 interrupts, which can be triggered using notify or release events from the channels

4.2 Architecture

4.2.1 IPC channel

An IPC channel is implemented as six hardware registers, as shown in Figure 4. The IPC channel registers are accessible to both secure and non-secure applications in the system.

- IPC_STRUCTx_ACQUIRE: This register determines the lock feature of the IPC. The IPC channel is acquired by reading this register. If the SUCCESS field returns a value of '1', it means that the read acquired the lock. If the SUCCESS field returns a value of '0', it means that the read operation failed to acquire the lock. The P, NS, PC, and MS fields represent the access attributes of the transaction that previously obtained the lock successfully, and the current access does not affect these fields.

Note that a single read access performs two functions:

- The attempt to acquire a lock
- Return the result of the acquisition attempt (SUCCESS field)

The atomicity of these two functions is essential in a CPU with multiple tasks that can preempt each other. When acquired, this register is released by writing any value into the IPC_STRUCTx_RELEASE register. If the register is already in an acquired state, another attempt to read it will not be able to acquire it

- IPC_STRUCTx_NOTIFY: This register is used to generate an IPC notify event. Each bit in this register corresponds to an IPC interrupt structure. The notify event generated from an IPC channel can trigger any or multiple interrupt structures
- IPC_STRUCTx_RELEASE: This field unlocks and triggers release events for the IPC interrupt structures, but only when the lock is obtained (LOCK_STATUS.ACQUIRED is '1'). The release event generated by an IPC channel can trigger any or multiple interrupt structures. To only release the IPC channel and not generate an interrupt, you can write a zero into the IPC release register. Masking of the interrupt can also be achieved by the IPC_INTR_STRUCTx_INTR_MASK register
- IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1: These registers are designed to hold data, and they are both 32-bit in size. These registers can be considered as the shared data memory for the channel. Typically, these registers will hold messages that need to be communicated between secure and non-secure applications. If the messages are larger than the combined 64-bit size, place pointers in one or both of these registers
- IPC_STRUCTx_LOCK_STATUS: This register provides the instantaneous lock status for the IPC channel. If the channel is acquired (LOCK_STATUS.ACQUIRED is '1'), it shows the protection context, security, and other information in the respective bit fields. However, if the lock is not acquired (LOCK_STATUS.ACQUIRED is '0'), the values in fields P, NS, PC, and MS are not valid. The lock status reading only gives an immediate status, which can change in the next cycle

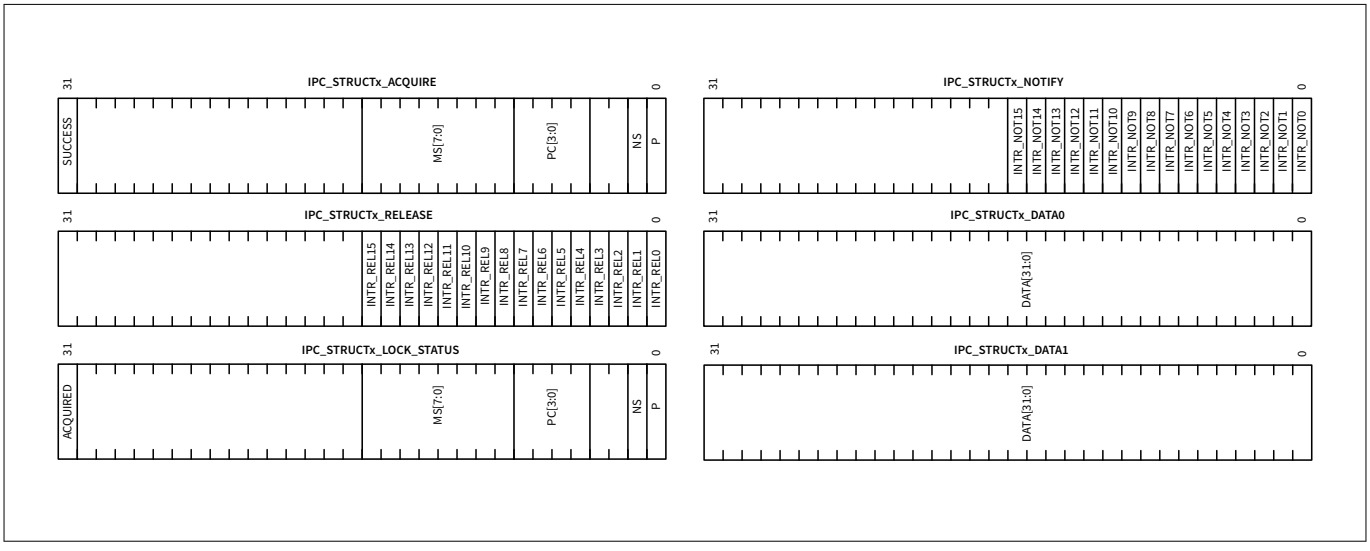


Figure 4 IPC channel structure

4.2.2 IPC interrupt

Each IPC interrupt line in the system has a corresponding IPC interrupt structure. An IPC interrupt can be triggered by a notify or a release event from any of the IPC channels in the system. You can choose to mask any of the sources of these events using the IPC interrupt registers. Figure 5 shows the registers in an IPC interrupt structure

- IPC_INTR_STRUCTx_INTR: This register provides the instantaneous status of the interrupt sources. Note that there are 16 notify and 16 release event bits in this register. These are the notify and release events

corresponding to the 16 IPC channels. When a notify event is triggered in IPC channel 0, the corresponding Notify 0 bit is activated in the interrupt registers. A write of 1 to a bit will clear the interrupt

- IPC_INTR_STRUCTx_INTR_MASK: The bit in this register masks the interrupt sources. Only the interrupt sources with their masks enabled can trigger the interrupt
- IPC_INTR_STRUCTx_INTR_SET: A write of 1 into this register sets the interrupt
- IPC_INTR_STRUCTx_INTR_MASKED: This register provides the instantaneous value of the interrupts after they are masked. The values in this register are IPC_INTR_STRUCTx_INTR and IPC_INTR_STRUCTx_INTR_MASK

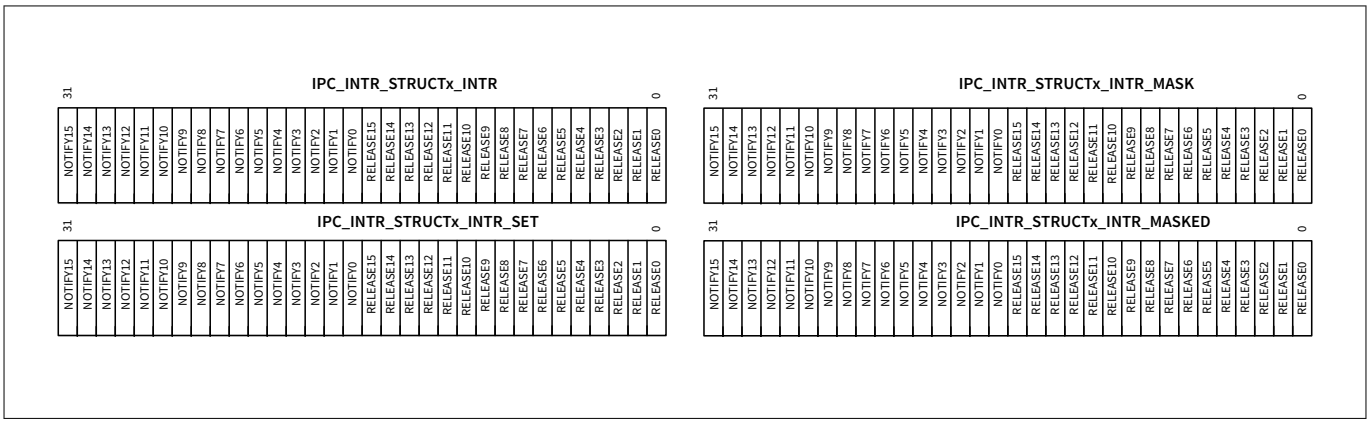


Figure 5 IPC interrupt structure

4.2.3 IPC channels and interrupts

The IPC block has a set of associated IPC interrupts. Each IPC interrupt register structure corresponds to an IPC interrupt line. This interrupt can trigger an interrupt on any of the processors in the system. The interrupt routing for processors is dependent on the device architecture.

Each IPC channel has a release and notify register, which can drive events to any of the IPC interrupts. An illustration of this relationship between the IPC channels and the IPC interrupt structure is shown in Figure 6.

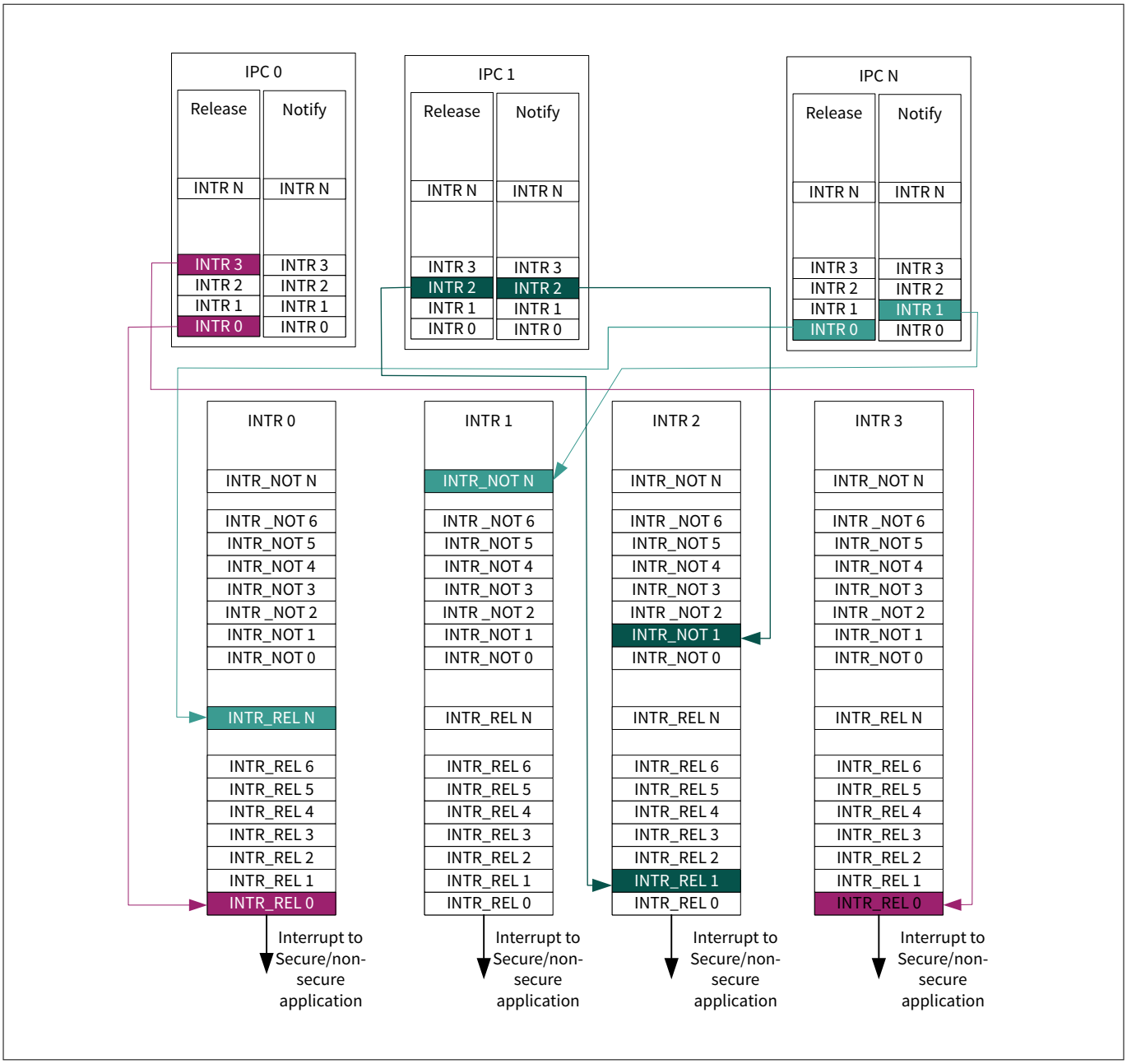


Figure 6 IPC channels and interrupts

4.3 Implementing locks

The IPC channels can be used to implement locks, which are typically used between secure and non-secure applications to enable some form of mutually exclusive access to a shared resource. When secure and non-secure applications share a resource, the secure or non-secure application can acquire and release the IPC channel. So secure and non-secure applications can assume an IPC channel as a lock. The semantics of this code are that access to the shared resource is gated by the secure and non-secure application ownership of the channel. So the secure or non-secure application must acquire the IPC channel before they can access the shared resource.

A failure to acquire the IPC channel by the secure or non-secure application signifies a lock on the shared resource. Note that the IPC channel does not enforce which application acquires or releases the channel. Secure or non-secure applications can both acquire or release the IPC channel.

4.4 Message passing

IPC channels can be used to communicate messages between secure and non-secure applications. In this use case, the channel is used in conjunction with the interrupt structures. The IPC channel is used to lock access to the data registers. The IPC channel is acquired by the sender and used to populate the message. The receiver reads the message and then releases the channel. Thus, between the sender transmitting data into the channel and the receiver reading it, the channel is locked for all other task access. The sender uses a notify event on the receiver's IPC interrupt to denote a send operation. The receiver acts on this interrupt and reads the data from the data registers. After the reception is complete, the receiver releases the channel and can also generate a release event for the sender's IPC interrupt. Note that the action of locking the channel does not restrict access to the data registers in hardware. This is a semantic that should be enforced by software.

Figure 7 shows an example of a sender (a non-secure application) sending data to a receiver (a secure application). IPC interrupt A is configured to interrupt non-secure application. IPC interrupt B is configured to interrupt secure application.

1. The sender will attempt to acquire the IPC channel by reading the IPC_STRUCTx_ACQUIRE register. If the channel is acquired, the sender has ownership of the channel for data transmission. This also changes the status of the channel and its corresponding IPC_STRUCTx_LOCK_STATUS register. If the channel is not acquired, the sender should wait until the channel is free for acquisition. This can be done by polling the IPC channel's IPC_STRUCTx_LOCK_STATUS register
2. After the IPC channel is acquired, the sender has control of the channel for communication and places the data in the IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1 registers
3. Now that the message is placed in the IPC channel, the sender generates a notify event on the receiver's interrupt line. It achieves this by setting the corresponding bit in the IPC channel's IPC_STRUCTx_NOTIFY register. This event creates a notify event at IPC interrupt B (IPC_INTR_STRUCTx_INTR). If the IPC channel's notify event is enabled by setting the mask bit (IPC_INTR_STRUCTx_INTR_MASK [31:16]) in the IPC interrupt B, this will generate an interrupt in the receiver
4. When it receives IPC interrupt B, the receiver can poll the IPC_INTR_STRUCTx_INTR_MASKED register to understand which IPC channel triggered the notify event. Based on this, the receiver identifies the channel to read and reads from the IPC channel's IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1 registers. The receiver has now received the data sent by the sender. It needs to release the channel so that other processes can use it
5. The receiver releases the channel. It also optionally generates a release event on the sender's IPC interrupt A. This generates a release event interrupt on the sender if the corresponding channel release event is masked

On receiving the release interrupt, the sender can act on the event based on the application requirement. It can either try to reacquire the channel for further transmission or go to other tasks because the transmission is complete.

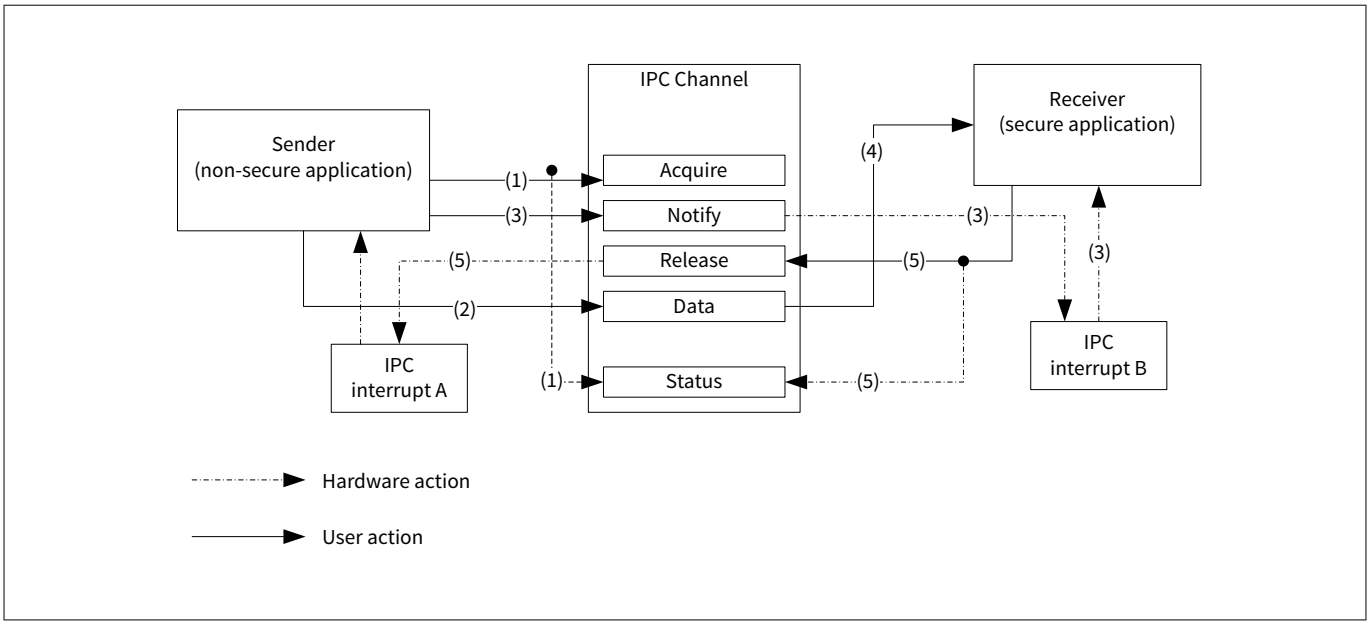


Figure 7 Sending messages using IPC

In the previous example, the size of the data being transmitted was only 64 bits. Larger messages can be sent as pointers. The sender can allocate a larger message structure in memory and pass the pointers to the data registers, see Figure 8 for usage. Note that the user code must implement the synchronization of the message read process.

- The implementation can stall the channel until the receiver has used all the data in the message packet and the message packet can be rewritten. This is worthless because it will stall other inter-process communications as the number of IPC channels is limited
- The receiver can release the channel as soon as it receives the pointer to the message packet. It implements the synchronization logic in the message packet as a flag, which the sender sets on write complete and the receiver clears on read complete

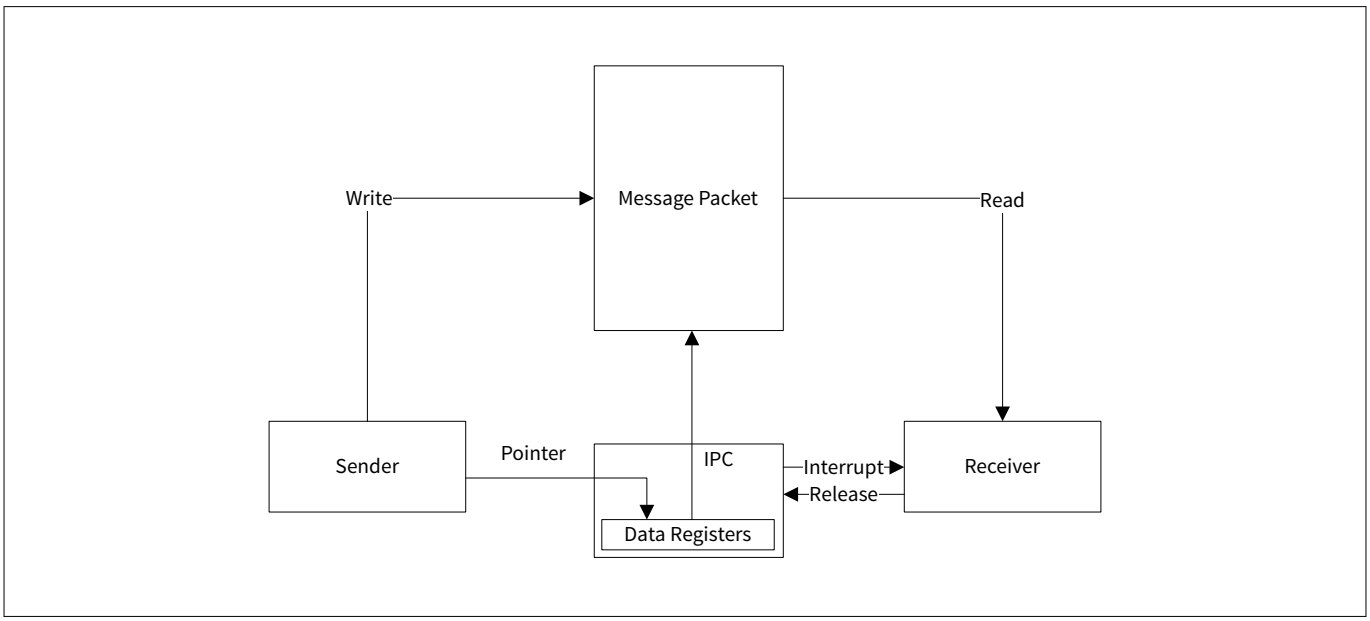


Figure 8 Communicating larger messages

4.5 Typical usage models

The unique channel and interrupt architecture of the PSC3M/P6 MCU IPC allows for a range of usage models for communication between secure and non-secure applications. Some of these are listed here as an example. Note that the communication models possible, based on the IPC architecture, are not restricted to the ones listed in this document. Also, this document only provides a high-level usage model and does not provide details of data management in the communication. This must be determined based on the specific application use case.

4.5.1 Full-duplex communication

In this usage model, an IPC channel is used according to the direction of communication between secure and non-secure applications. In this example, the IPC channel X is dedicated to data communication from non-secure application to secure application and IPC channel Y is for data communication from secure application to non-secure application. The IPC interrupt X will signal events on secure application. Therefore, its interrupt output is connected to secure application's system interrupts. The events are triggered by writing into the IPC interrupt register structure over the system bus. Similarly, IPC interrupt Y is dedicated to non-secure application.

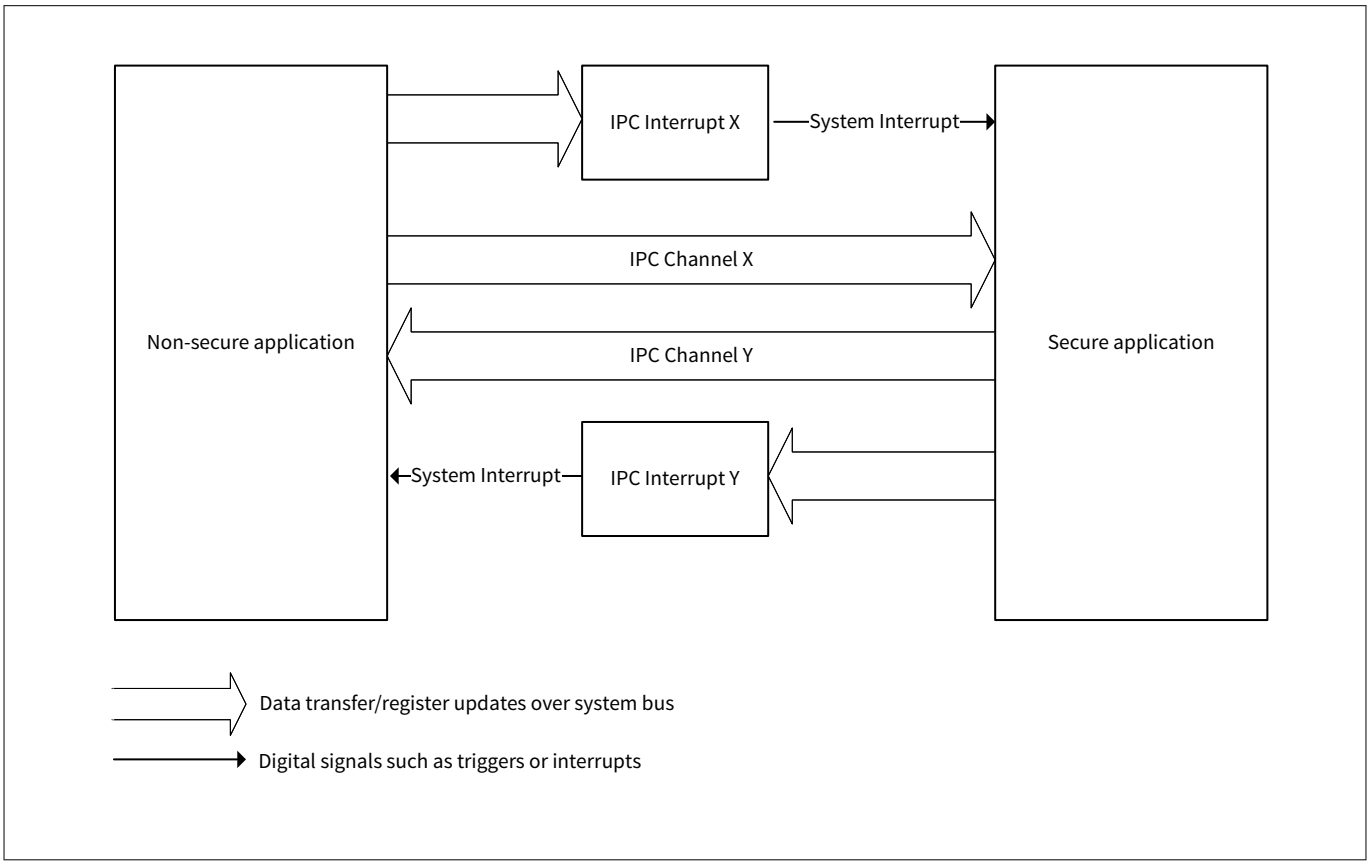


Figure 9 Full-duplex IPC between secure and non-secure application

4.5.2 Half-duplex with independent event handling

In this case, only one IPC channel is used to set up the transfer between the secure and non-secure applications. This means that only one side controls data transfer at a time. The channel's lock register must be used to avoid contention of the one shared IPC channel. Two independent events are supported since two IPC interrupt structures are used. This model is shown in Figure 10.

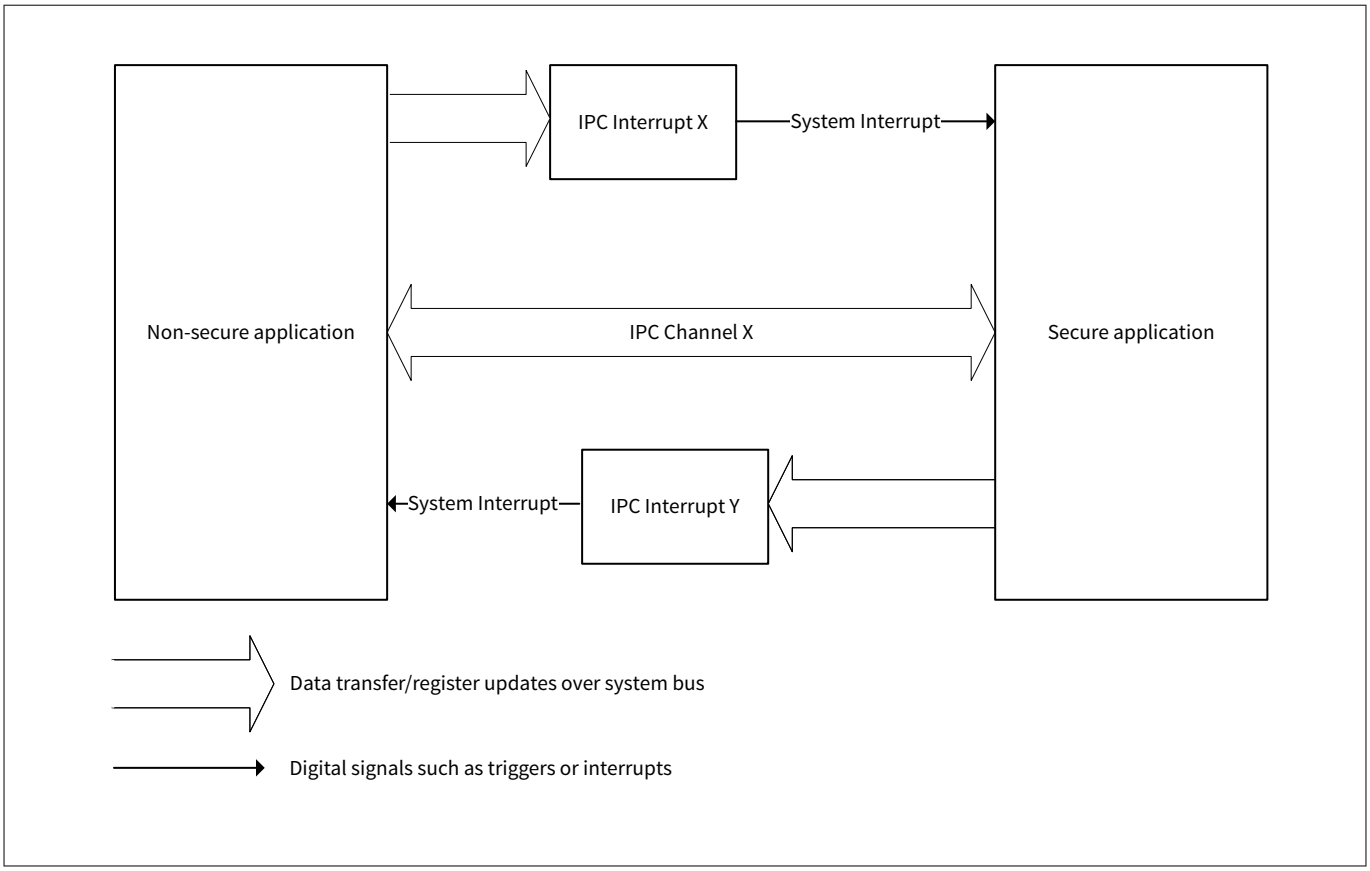


Figure 10 Half-duplex with independent event handling

4.5.3 Half-duplex with shared event handling

In this model, both the IPC channel and interrupt are shared between the secure and non-secure applications. Since the interrupt is also shared, access to the interrupt registers must be managed using the IPC lock of the channel. As shown in Figure 11, the IPC interrupt is set up to trigger interrupts in both secure and non-secure applications. Therefore, secure or non-secure application interrupts should have logic in their ISR to check which application is in control of the IPC and determine if the message and event were for that application.

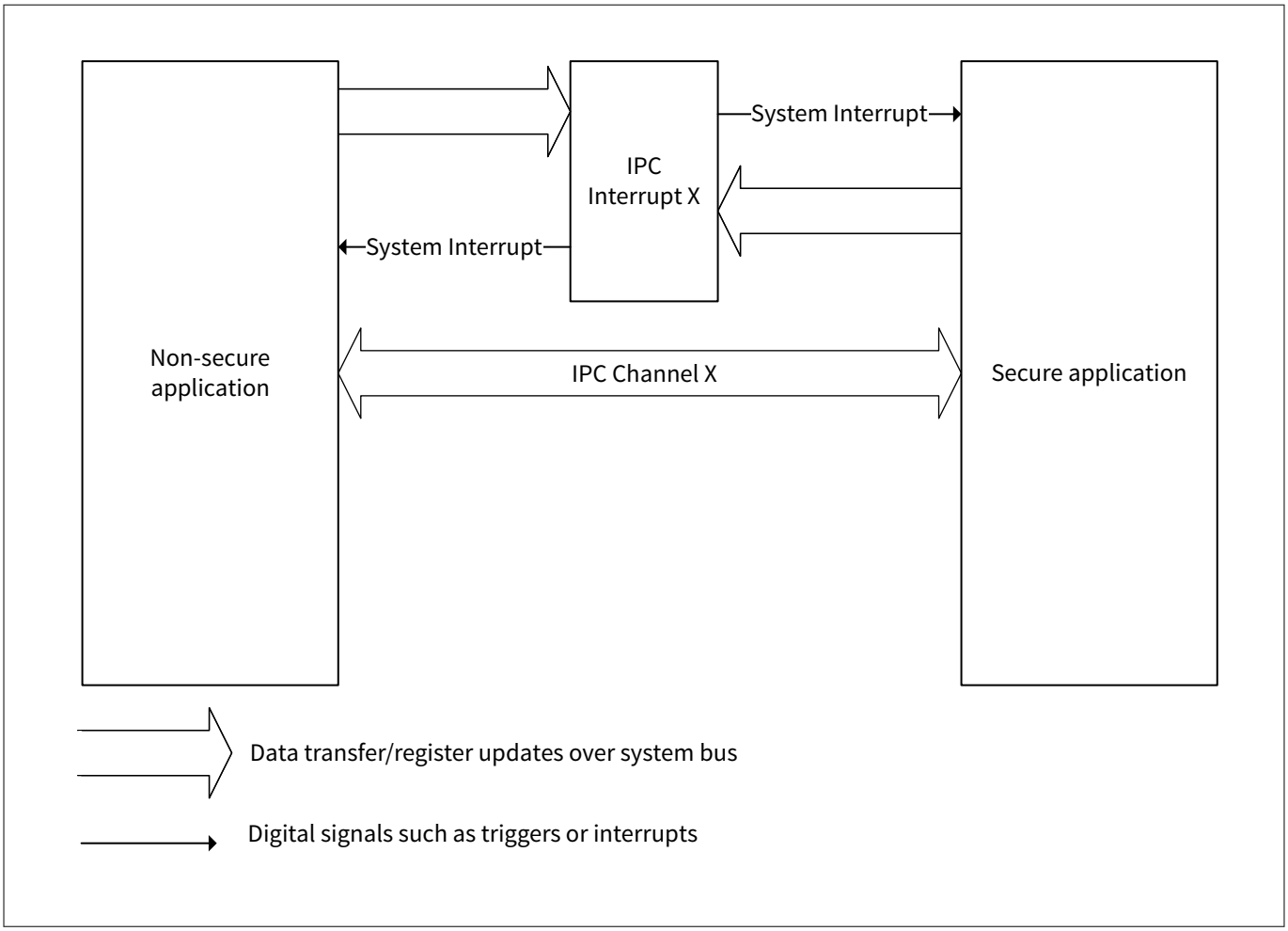


Figure 11 Half-duplex with shared event handling

5 Flash

The PSC3M/P6 flash offers high bulk program performance and supports ultra-low-power operation. Flash is typically used to store CPU instructions and data when the device's power is off. Flash may be written, but the process is much slower and more restrictive than for SRAM. This chapter explains the geometry and capability of flash memory.

5.1 Features

Flash has the following features:

- Flash memory size of up to 512 KB
- 512-byte row size; minimum programmable unit, containing 2 sectors and each sector with 512 rows
- 9-bit ECC for single error correction and dual error detection (SECDED)
- Supports the Read While Write feature
- Flash memory protection using the memory protection controller (MPC)
- Supports a programmable number of wait states
- Arm® Q-Channel LPI for low-power mode management
- 10 - 20 year retention
- Endurance of 100k program cycles

5.2 Configuration

5.2.1 FLASHC

This section explains the major components within the Flash controller (FLASHC).

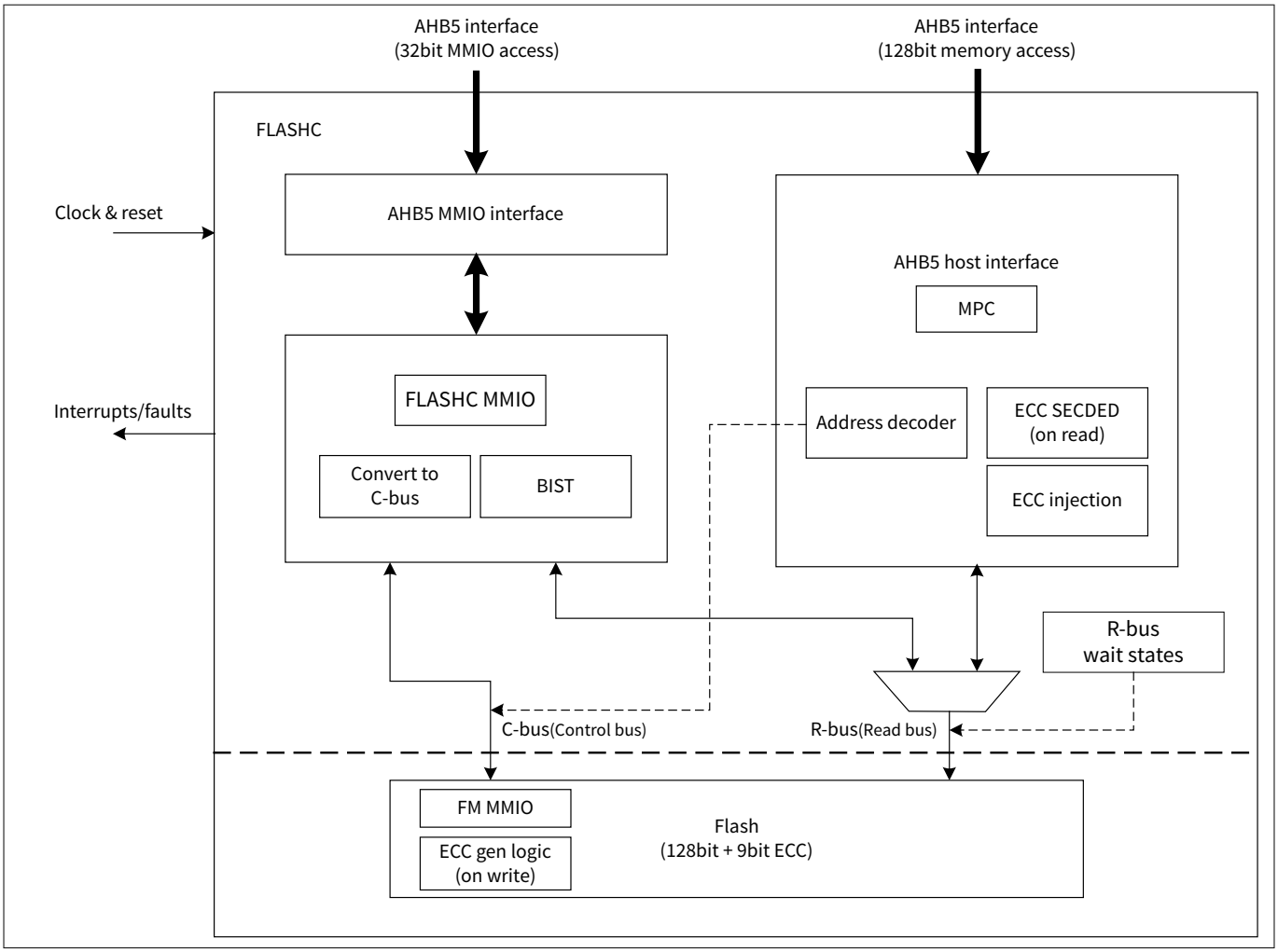


Figure 12 FLASHC high-level block diagram

Flash macro

The flash macro (FM) is a flash memory macro used in the system-on-chip. It supports read access at low-power (LP) and ultra-low-power (ULP) voltage ranges and allows writes only at LP levels. A read operation returns 137 bits per access (128 bits of user data plus 9 bits of ECC). The ECC is included in the read stream and not counted as user data.

The flash controller (FLASHC) provides the following FM platform controller functionality:

- It provides read from FM on the 128-bit AHB5 host interface (with MPC protection) connected to the 128-bit CODE interconnect. A write sequence can be initiated to FM (write to FM_ADDR/FM_REFRESH_ADDR FM register) on the same 128-bit AHB5 host interface or on the 32-bit AHB5 MMIO interface
- It provides MAIN, WORK, SFLASH, 33rd column and refresh rows regions. It can be operated in single bank or dual bank mode
- It supports three modes of operation: full R-bus, full C-bus and RWW (read while write) modes
- BIST support for at-speed read test
- Design time configurable ECC functionality
- A Fault interface connected to the CPUSS central fault reporting infrastructure

***Note**: The flash macro (FM) can be partitioned into multiple regions main flash, work flash, SFLASH, 33rd column, and refresh rows. In case of PSC3M/P6, main and work flash pointing to one region (user flash). In dual bank mode, the main flash refers to the sector where code is running, and the other sector becomes work region where CPU can easily erase and program data.*

Protection using MPC

FLASHC 128-bit AHB5 host interface is protected using MPC. MPC implements secure and non- secure block-based protection. It provides block-based protection for each protection context (PC) as well. There are separate entries for User Flash, SFLASH, and Refresh Rows. All entries are aggregated and implemented by one MPC. The 33rd column is protected by the same MPC entry as the memory block to which the 33rd column belongs.

BIST

FLASHC provides hardware based read BIST for implementing at-speed FM read BIST patterns in test/DFT modes. It covers the 33rd column, refresh rows and redundancy of the FM, including ECC bits. The BIST controller performs a single swipe through the FLASH address space defined by MMIO registers. It performs back-to-back read from FM and compares against BIST_DATA register or complement of BIST_DATA based on BIST_CTL.OPCODE register.

5.2.2 Flash geometry

The following figure shows the geometry of flash macro (FM).

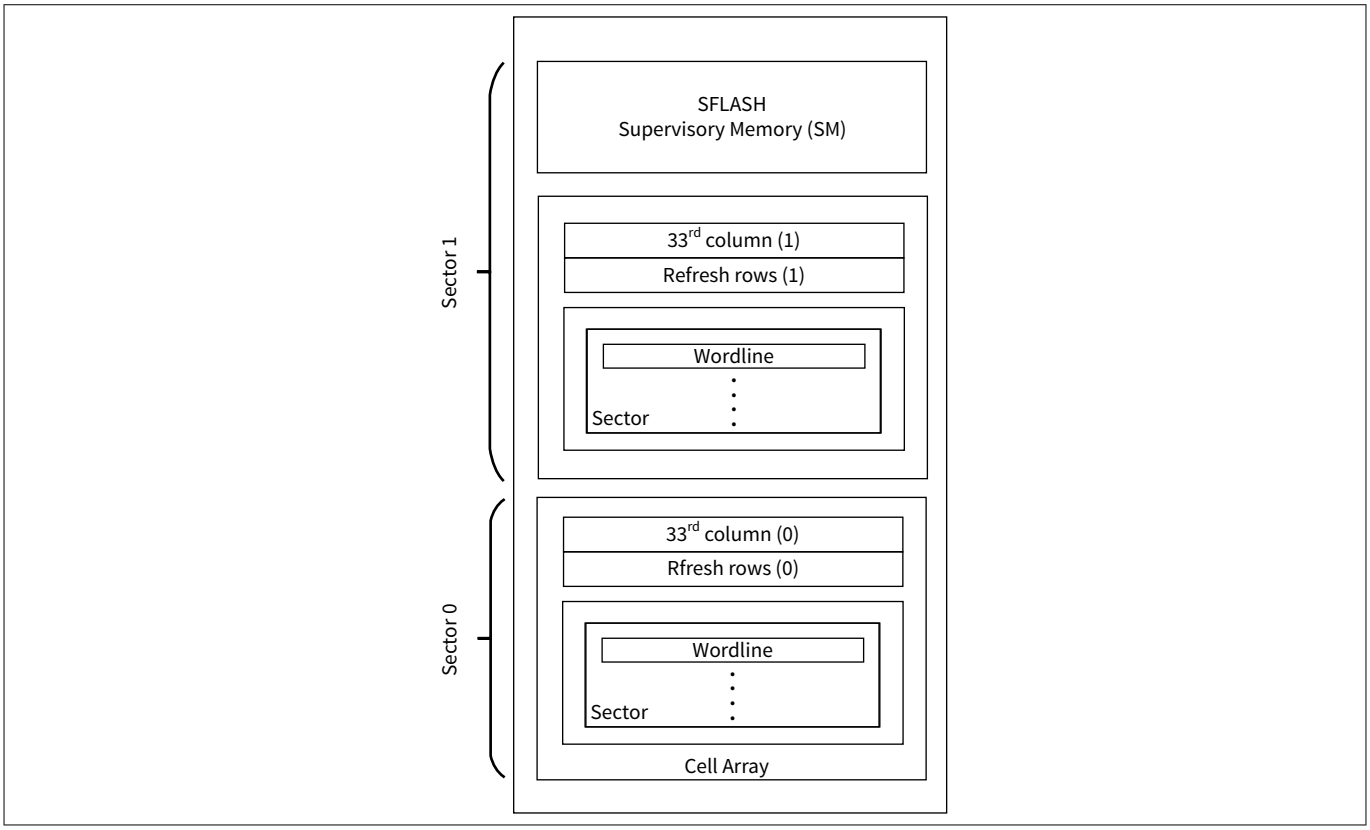


Figure 13 Flash geometry

The flash macro (FM) has a well-defined hierarchical structure and read/write behavior as described in the following sections.

- **Read word**

Flash macro read word consists of 128 data bits plus 9 bits of ECC. A read word is returned on every read command

- **Rows (word line)**

  A row corresponds to a word line in the flash macro (FM) and can be programmed or erased only in its entirety. A row is composed of 32 read words, yielding a row size of (32 RW ×128 bits = 4096 bits) 512 bytes

- **Sector**

  A sector is made up of multiple rows. The size of a sector is several kilobytes. In PSC3M/P6, a sector comprises 512 rows. Each sector occupies 256 KB of storage

- **33rd column (oldest/youngest row counter)**

  In a sector, the 33rd column serves as a counter to track the aging of rows during erase/program cycles. When erase/program cycles occur on rows within the same sector, unselected rows may experience program disturb, degrading endurance/retention. The 33rd column keeps count values that identify the oldest and youngest rows in the sector. The data from 33rd column is used to initiate a refresh operation. A refresh operation involves erasing and original data before reprogramming it. Any interruptions during this sequence may leave critical pages of data in a corrupted or erased state. Therefore, data from the pages to be refreshed is copied to the refresh rows which are additional pages in each sector only to support refresh operations. The page to be refreshed is then erased and reprogrammed with the same data with a new counter value.

- **Refresh rows**

  Each sector contains dedicated refresh rows with a fixed system address to support refresh operations. A refresh operation involves copying the data from the target rows to the refresh rows, erasing the target rows, and reprogramming the target rows with the data from the refresh rows while assigning a new counter value

  Important notes:

  - Refresh is not allowed in sector 1 because SFLASH is shared with this sector and SFLASH cannot be refreshed by user code. Therefore, any EEPROM emulation memory should be allocated within physical sector 0
  - Access to refresh rows must be configured with a protection context (PC) of 0, 1, or 2 to prevent corruption of the recovery process
  - It is the OEM's responsibility to test for refresh needs and to initiate the flash refresh routine when required

  Refresh is needed when the flash is written more than 100 K times. Flash writes to part of the flash will degrade the contents in other parts of the flash sector. The refresh function is provided to restore the rest of the flash sector after many writes. This is only required when the flash is used for data logging or other EEPROM emulation purposes. It is not required if the flash is only used for program storage. SROM APIs provide blocking operations to write, program, and erase flash rows. Refer to theSROM APIsfor details.

5.2.3 Bank modes and address mapping

This flash memory controller has a dual-bank mode feature. When using dual-bank mode, the flash memory region is split into two half-banks. One is called Bank 0, and the other is called the Bank 1.

FM supports a Read While Write operation on the same flash. This means that a write can be done on one logical bank and a read can be done from the other bank. If the write and read is done from the same logical bank, it will result in an error.

The main flash region supports dual-bank mode. The user can select the mode through FLASH_CTL.BANK_MODE.

This is to support Firmware updates of the software image in flash memory.

These address maps are configurable to support bank swapping as follows:

- When configuring single bank mode, the entire main flash regions are mapped as a single contiguous address region
- When configuring dual bank mode, the main flash region is split into two halves, and each half is presented as a separate address region. Furthermore, these halves can be swapped to support same-location firmware upgrades
  - Choosing mapping A will present the Bank 0 in the lower region and the Bank 1 in the upper region
  - Choosing mapping B will present the Bank 0 in the upper region and the Bank 1 in the lower region

When switching flash banks, the following rules must be followed:

- The BANK_MODE LSB must be changed for the banks to switch correctly. Modifying BANK_MAPPING without modifying the BANK_MODE LSB may result in errors

Refer "Address and memory maps" section of the CPU subsystem for flash address mapping.

5.3 Power modes

The PSC3M/P6 MCU supports system power modes including OFF, HIBERNATE, DEEPSLEEP_OFF, DEEPSLEEP_RAM, DEEPSLEEP, and ACTIVE. The following table shows system and flash power mode mapping.

Table 11 Flash power mode mapping

System power modes	FM power modes	FM time to wake	Condition
Hibernate	Hibernate	< 10 us	All analog circuits Off
DEEPSLEEP_OFF	Hibernate	< 10 us	All analog circuits Off
DEEPSLEEP_RAM	Hibernate	< 10 us	All analog circuits Off
DEEPSLEEP	Hibernate	< 10 us	All analog circuits Off
SLEEP	Standby	-	All analog circuits On, no clock
ACTIVE	Active	-	All analog circuits On, clock present

5.4 Read While Write support

The PSC3M/P6 MCU supports read operations on one sector while programming or erasing in another area. The FM contains two sectors, each 256 KB in size. The RWW feature is available between sectors. The application can read or execute from one sector while there is an ongoing write or erase operation in another sector. However, when the code execution or read is in the last 16 bytes of a given sector (say sector 0) and the flash write/erase operation is in the next sector (sector 1), an RWW violation may occur if cache pre-fetch is enabled. This is because pre-fetch will fetch the next 16 bytes of data, which is part of sector 1, while a write operation is underway in the same sector. This will result in a fault and should be considered during firmware design. Firmware can be designed to place dead code or constant data in the last 16 bytes of every sector, making sure the last 16 bytes of a sector never accessed, or it can disable cache pre-fetch during a flash write or erase operation.

The RWW FMs can be used as a traditional one that is Read Operations through Read-bus, Read or Write Operations through Control-bus, or in RWW mode, for which Read Operations occur through Read-bus while the part is written through Control-bus. The Read-bus and Control-bus have separate sector and row address registers and pre-decoders, their outputs being multiplexed at sector level. During the write operation, the concurrent read cannot happen in the same sector.

During RWW, the following operations can be performed on the Write Sector:

- Row erase or program
- Sector erase or pre-program

During RWW, the following are not available:

- Data read in the Write Sector
- Bulk Erase or Pre-Program
- Test Modes

5.5 ECC

The flash controller supports Error Correcting Code (ECC). The FM uses an additional 9 bits for every word of 128 bits to perform Single Error Correction (SEC) and Dual Error Detection (DED). Due to ECC, the internal read width is extended from 128 bits to 137 bits.

The ECC block can be tested by injecting incorrect ECC values. See ECC (error) injection for details.

5.5.1 ECC generation/encoding

The ECC Encoder is located in the FM, while the ECC Decoder is located in the flash controller within CPUSS. While in full C-bus or RWW mode, it is software configuration that either FM hardware-generated ECC or software-specified ECC written into FM follow FM C-bus ECC generation/encoding on the write sequence.

- When FM ANA_CTL0.ECC_ENC_DIS == 1'b0 implies the ECC encoder is enabled for the FM with the ECC feature. The PL ECC bits are loaded automatically
- When FM ANA_CTL0. ECC_ENC_DIS == 1'b1 implies the ECC encoder is disabled. FM will use the ECC parity from FM_PL_ECC.DATA9

5.5.2 ECC (error) injection

ECC injection can be enabled or disabled based on the register ECC_INJ_EN.ECC_INJ_ENABLE. In full R-bus or RWW mode, ECC parity can be injected based on ECC_INJ_EN.ECC_INJ_ENABLE to replace ECC parity delivered by Flash Macro. The ECC_CTL.WORD_ADDR and ECC_CTL.PARITY registers specify the address and the parity to be injected. ECC_CTL.PARITY is injected when ECC_INJ_EN.ECC_INJ_PC and ECC_CTL.WORD_ADDR both match the AHB5 read transaction over the 128-bit interface.

While in full R-bus or RWW mode, ECC parity (to replace ECC parity delivered by FM) can be injected prior to the ECC correction or decoding logic during the read. The flash MMIO ECC_INJ_EN/ECC_INJ_CTL can be enabled and configured to inject 9-bit ECC based on the Program Counter and address of the 128-bit long-word.

When ECC_INJ_EN.ECC_INJ_ENABLE is LOW, ECC_INJ_EN.ECC_INJ_PC inherits Program Counter (PC) from the master who enables/configures ECC_INJ_ENABLE HIGH.

- When ECC_INJ_ENABLE is HIGH, only the PC specified by ECC_INJ_PC can access (read/write) ECC_INJ_EN (ENABLE, ECC_ERROR, and ECC_INJ_PC bits)
- PC violation results in a bus error, and there is no additional fault/interrupt triggered. The only exception is PC0, which can access (read/write) ECC_INJ_EN at any time to break the potential deadlock of ECC_INJ_EN

ECC_INJ_CTL.WORD_ADDR specifies address (based on module-internal offset address) and ECC_INJ_CTL.PARITY, parity, to be injected.

- When ECC_INJ_ENABLE is HIGH, only the PC specified by ECC_INJ_PC can access (read/write) ECC_INJ_CTL
- PC violation results in a bus error, and there is no additional fault/interrupt triggered

PARITY is injected when ECC_INJ_PC and WORD_ADDR both match the AHB read transaction over the 128-bit interface.

- ECC_INJ_CTL.PARITY replaces ECC parity from FM and is used by the ECC correction/decoding logic
- ECC_INJ_EN.ECC_ERROR:
  - LOW (1'b0): If the injected ECC does not trigger any non-recoverable error (ECC errors <= 1)
  - HIGH (1'b1): If the injected ECC triggers a non-recoverable error (ECC errors >= 2). The AHB read transaction results in a bus error. There is no additional fault/interrupt triggered

ECC injection is subject to MPC protection. Each PC is allowed only to inject ECC parity into its allocated address blocks/ranges.

To disable ECC injection, the PC specified by ECC_INJ_PC should clear ECC_INJ_EN (reset ECC_INJ_ENABLE and ECC_ERROR bits). PC0 can also clear ECC_INJ_EN at any time.

FLASHC hardware clears ECC_INJ_CTL autonomously while ECC_INJ_EN is being cleared.

5.6 Security and protection

The FLASHC 128-bit AHB5 host interface is protected using MPC. The MPC implements Secure/Non-Secure block-based protection similar to the Arm® SIE-200 MPC. However, the functionality is extended to provide block-based protection for each protection context (PC) as well.

Refer to Memory Protection Controller (MPC) for details on MPC.

***Note**: Flash write can only be done from a secure application.*

***Note**: FLASHC APIs can be called only from a secure application.*

5.7 SROM APIs

SROM APIs provide blocking operations to write, program, and erase flash rows. Data is flashed in 512-byte rows (one row per operation). Larger data requires iterating over rows.

The main SROM APIs:

- Performs write operation

```
uint32_t cyboot_flash_write_row(uint32_t address, uint8_t* data, flash_context_t* ctx);
```

- Performs program operation

```
uint32_t cyboot_flash_program_row(uint32_t address, uint8_t* data, flash_context_t* ctx);
```

- Performs erase operation

```
uint32_t cyboot_flash_erase_row(uint32_t address, flash_context_t* ctx);
```

Context and parameters of SROM APIs:

- *** ctx**: Points to a RAM memory where flash context (flash_context_t) is located. For blocking mode, all fields must be zero
- **address**: Flash address to erase/program/write
- **data**: Pointer to RAM data for a single 512-byte row to be written/programmed

Data structure details (flash_context_t):

```
typedef struct
{
    uint32_t min_count;
    uint32_t max_count;
    uint32_t min_page_addr;
    uint32_t scratch_row_idx;
} cyboot_flash_refresh_t;
```

```
typedef struct
{
    /*
     * [0] 0 -BLOCKING, 1-NON-BLOCKING
     * [1] RWW or STALL_READ, for BLOCKING
     * [2] REFRESH_CTL, determines who provides the data for Column 33.
     * 0- Flash API computes the data for column 33.
     * 1- column 33 is programmed as is. Up to a caller to provide the data.
     * [3] A mode for Program Sector and Program Bulk operations.
     * 0- ALL
     * 1- Even/Odd
     */
    uint32_t flags;
    uint32_t hv_params_addr;
    cyboot_flash_refresh_t *refresh;
    /* params for non-blocking operations */
    cyboot_flash_callback_t callback_pre_irq;
    cyboot_flash_callback_t callback_post_irq;
    cyboot_flash_callback_t callback_complete;
    uint32_t callback_param;
    uint32_t state; /* A state machine - EMPTY, ERASE_0, PROGRAM_0, WRITE_0,1 */
    uint32_t flash_addr; /* Non-blocking WriteRow */
    uint32_t data_addr; /* Non-blocking WriteRow */
    uint32_t reserved[2]; /* size to be 8 bytes aligned */
} flash_context_t;
```

Important notes

- Each flash operation works on 512-byte rows; smaller data cannot be flashed or erased
- In blocking mode, the CPU is halted until the operation completes
- Flash context is written to RAM prior to the operation; breakpoints and stack considerations are part of the flow
- The SROM API Table 12 maps function pointers to specific flash operations

Table 12 SROM APIs for writing/erasing PSOC™ Control C3 flash memory

Address	Function pointer
0x1080FFE8u	cyboot_flash_write_row

(table continues...)

Table 12 (continued) SROM APIs for writing/erasing PSOC™ Control C3 flash memory

Address	Function pointer
0x1080FFE4u	cyboot_flash_program_row
0x1080FFE0u	cyboot_flash_erase_row

6 SRAM controller

This section explains the PSC3M/P6 MCU SRAM controller, its features, architecture, and wait states. The SRAM controller enables the CPU to read and write the PSC3M/P6 MCU's SRAM.

In the PSC3M/P6 MCU, the size of the SRAM is 128 KB. To know more about SRAM instances and their start addresses, see the Address and memory maps section of the CPU subsystem.

6.1 Features

The SRAM controller has the following features:

- Supports 8, 16, and 32-bit access
- Provides exclusive access support using Exclusive Access Monitor (EAM) from Arm® SIE-200
- SRAM memory protection using the MPC (Memory Protection Controller)
- SRAM power mode control using Arm® PPU (Power Policy Unit)
- The SRAM controller supports the Error Correction Code (ECC) for the SRAM

See the device datasheet for electrical specifications.

6.2 Architecture

The SRAM controller has an AHB5 interface that connects to the AHB5 infrastructure. The AHB5 transfers are the origin for all SRAM accesses. The SRAM controller differentiates between the following three types of AHB5 transfers:

- AHB5 read transfers
- 32-bit AHB5 write transfers
- 8-bit and 16-bit AHB5 write transfers (also referred to as Partial AHB5 write transfers)

6.2.1 AHB5 read transfers

AHB5 reads transfers, and a read transfer is translated into SRAM read access.

6.2.2 32-bit AHB5 write transfers

A 32-bit AHB5 write transfer is translated into SRAM write access.

6.2.3 Partial AHB5 write transfers

Partial writes (8-bits and 16-bits) are supported through read, modify, and write by applying the associated mask during write.

6.2.4 Power partition control

The PWR_MACRO_CTL register controls the power state of each SRAM power partition independently. Each bit of PWR_MACRO_CTL.OFF[31:0] represents the individual state of each SRAM power partition. Enabling the bit corresponding to the particular power partition turns OFF the partitions, and disabling the bit turns ON the respective partitions. It is mandatory to ensure that STATUS.PWR_DONE becomes 1 after the PWR_MACRO_CTL register contents are changed, indicating that the SRAM mode transition has completed successfully. Each power partition contains a (RAM[ ]_MACRO_NR/RAM[ ]_PWR_GROUP_NR) number of base SRAM units. In PSC3M/P6 MCU, RAMC0/1_MACRO_NR = 1 and RAMC0/1_PWR_GROUP_NR = 1 create 1 power partitions with 1 macro in each power partition.

The PWR_MACRO_CTL_LOCK register prevents accidental writes to the PWR_MACRO_CTL register by prohibiting read/write access when this (PWR_MACRO_CTL_LOCK[1:0]) field is not equal to zero.

6.3 Error Correcting Code (ECC)

The SRAM controller supports the Error Correcting Code (ECC) for the SRAM.

The 7-bit SECDED parity covers the 32-bit data word as a single entity, there is no dedicated parity for the four 8-bit bytes within the data. Therefore, any partial 8-bit or 16-bit AHB5 write bus transfer requires a SRAM read access and a SRAM write access.

- 32-bit AHB5 write bus transfers require only a single SRAM write access
- 8-bit and 16-bit AHB5 write bus transfers require an additional SRAM read access that precedes the SRAM write access to retrieve the missing data bytes

When a non-recoverable error that is error of more than 1 bit is detected, the data word that is used as the result for an AHB5 bus transfer is incorrect, but NO AHB5 bus error is generated. Instead, the error is communicated through the CPUSS fault reporting infrastructure.

6.3.1 ECC Error injection

The Error Injection mechanism is supported by 32-bit data word write into an address defined in ECC_MATCH. Reading from the address defined in ECC_MATCH.WORD_ADDR triggers a correctable or uncorrectable fault indication, if the value in ECC_CTL.PARITY has 1, 2, or more errors compared to the expected ECC value.

6.3.2 ECC error report functionality

The ECC syndrome logic reports recoverable and non-recoverable errors to the CPUSS fault reporting infrastructure. The ECC syndrome logic corrects recoverable errors. Errors up to one bit are recoverable.

6.3.2.1 Partial AHB5 write transfers

Error reported during partial write transfer involving SRAM read are listed as below.

- **Correctable Error (1-bit error):** This corrected 32-bit data word is then merged with the write data with appropriate mask and stored in the write buffer. A future write-buffer request results in a SRAM write access with the merged write data
- **Multiple-bit error:** the error is communicated through the CPUSS fault reporting infrastructure

6.3.2.2 AHB5 read transfers

During the AHB read transfers below are possible scenarios.

- **Correctable Error (1-bit error):** The corrected data is returned to the AHB5 interface. If ECC auto-correct functionality is enabled, then the corrected data is also stored in write-buffer. A future write-buffer request results in a SRAM write access with the corrected data
- **Multiple-bit error:** Corrupted-data (multiple-bit error data) is returned to the AHB master on hrdata[31:0]

6.3.2.3 Correctable ECC error reporting

When the auto-correct functionality is enabled for 1-bit correctable ECC errors, the corrected data and merged data during partial write along with the corresponding AHB address is stored in the WRITE-BUFFER. Until the data for this AHB address is written back from the WRITE_BUFFER to the SRAM macro, all future accesses to this AHB address result in the data being read from the WRITE_BUFFER.

6.3.2.4 Non-correctable ECC error reporting

When a non-recoverable error is detected, this is reported only via the fault reporting structure and no AHB bus error is generated. Further, the data word that is used as the result for an AHB5 bus transfer is incorrect.

6.4 Wait states

Under standard operating conditions, no wait states are introduced, as only AHB5 requests are issued. This behavior remains consistent even when ECC is enabled. However, a single wait state is inserted under the following conditions:

- Read-after-Write Dependency: A read operation immediately follows a write operation
- ECC Correction Overhead: An ECC correction was triggered during the preceding memory access

7 Interrupts

7.1 Features

The PSC3M/P6 MCU supports the following interrupt features:

- Supports 167 system interrupts
  - Four interrupts can be mapped to the CPU's non-maskable interrupt (NMI)
- All interrupt sources are capable of waking the CPU from CPU Sleep power mode
  - 38 interrupt sources are capable of waking the Cortex®-M33 from CPU DeepSleep power mode
- Nested vectored interrupt controller (NVIC) integrated with CPU core, yielding low interrupt latency
- Wakeup interrupt controller (WIC) enabling interrupt detection (CPU wakeup) in CPU DeepSleep power mode
- A vector table may be placed in either volatile or non-volatile memory
- Separate vector tables for Cortex®-M33 secure and non-secure states
- Configurable priority levels for each interrupt, eight levels on the Cortex®-M33
- Level-triggered interrupt signals

7.2 Architecture

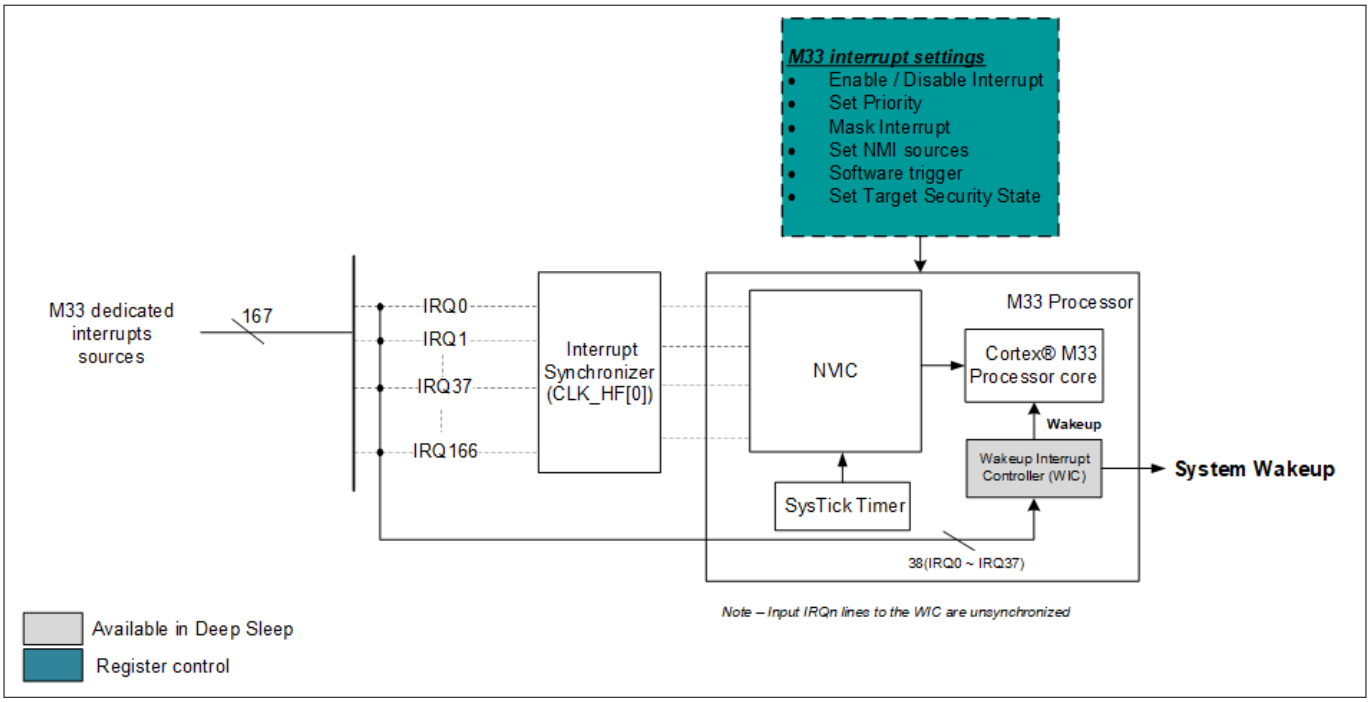


Figure 14 PSC3M/P6 interrupt architecture block diagram

Figure 14 shows the PSC3M/P6 MCU interrupt architecture. The Cortex®-M33 has system interrupt sources directly connected to the IRQn of the NVIC. The NVIC handles enabling/disabling individual interrupt IRQs, priority resolution, and communication with the CPU core. The other exceptions, such as NMI and hard faults, are not shown in Figure 14 because they are part of CPU core-generated events, unlike interrupts, which are generated by peripherals external to the CPU.

In addition to the NVIC, the PSC3M/P6 supports wakeup interrupt controllers (WIC) for the CPU and interrupt synchronization blocks that synchronize the interrupts to the CPU's CLK_HF0 domain (adds two CLK_HF0

cycles of delay for synchronization). The WIC provides detection of interrupts in the CPU DeepSleep power mode.

7.3 Interrupts and exceptions operation

7.3.1 Interrupt/exception handling

The following sequence of events occurs when an interrupt or exception event is triggered:

- The assumption is that all the interrupt and exception signals are initially low (idle or inactive state) and the processor executes the main code. Any interrupt signal that is held asserted is registered by the NVIC, if the interrupt or exception is enabled to be serviced by the CPU. The signal is now in a pending state, waiting to be serviced by the CPU
- On detecting the signal from the NVIC, the CPU stores its current context by pushing the contents of the CPU registers onto the stack
- The CPU also receives the exception number of the triggered interrupt from the NVIC. All interrupts and exceptions have a unique exception number. By using this exception number, the CPU fetches the address of the specific exception handler from the vector table. Each interrupt can be configured to target the secure or non-secure vector table
- The CPU then branches to this address and executes the exception handler that follows
- After completion of the exception handler, the CPU registers are restored to their original state using stack pop operations, and the CPU resumes the main code execution

When the NVIC receives an interrupt request while another interrupt is being serviced or receives multiple interrupt requests at the same time, it evaluates the priority of all these interrupts, sending the exception number of the highest priority interrupt to the CPU. Thus, a higher-priority interrupt can block the execution of a lower-priority ISR at any time. The Armv8-M CPU also supports tail-chaining. This means that on completion of an exception handler, if the next pending exception has a higher priority than all stacked exceptions, the stack pop operation is skipped and control is transferred to the new exception handler. Exceptions are handled in the same way that interrupts are handled. Each exception event has a unique exception number, which is used by the CPU to execute the appropriate exception handler.

Security is of importance; it is advised to mask non-secure interrupts during secure code execution. This is because a non-secure interrupt can inherit the protection context of the secure code if a non-secure interrupt were to trigger during secure code execution.

7.3.2 Level interrupts

PSC3M/P6 MCU only supports level signals on the interrupt lines (IRQn).

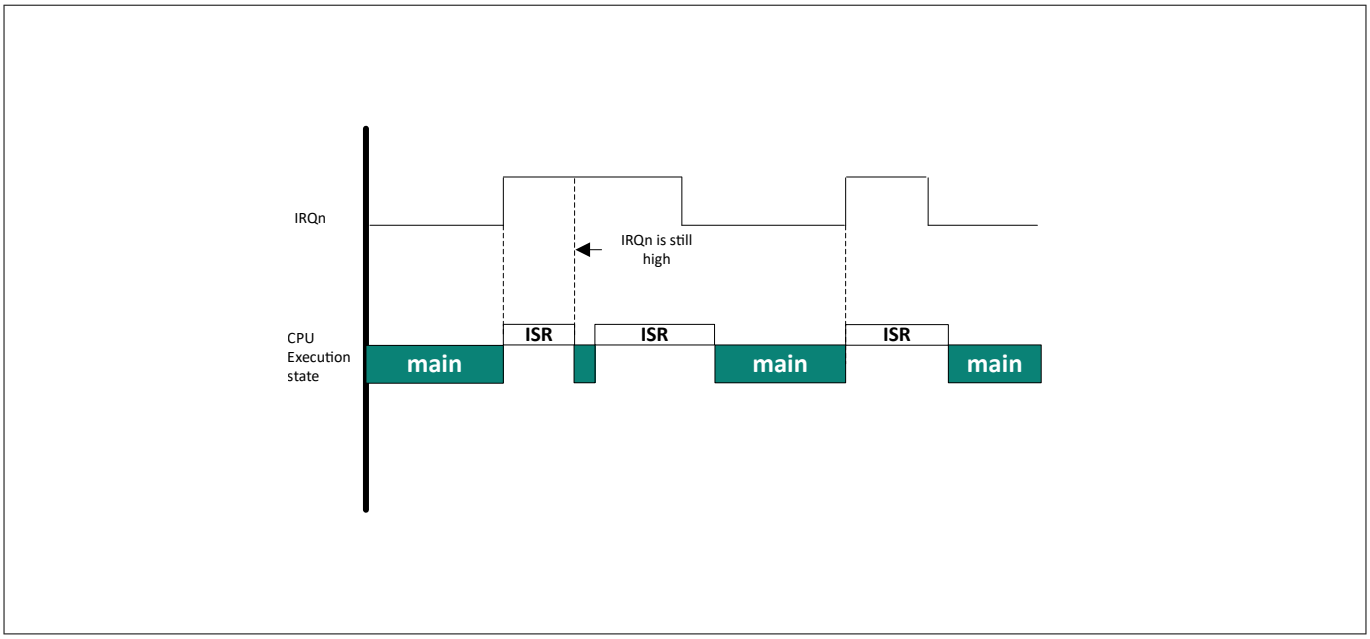


Figure 15 Level interrupts

***Note**: Pulse interrupts are not supported by this device.*

Figure 15 shows how a level interrupt is handled. Assuming the interrupt signal is initially inactive (logic low), the following sequence of events explains the handling of the level:

- For level interrupts, the interrupt signal remains asserted until the NVIC registers the interrupt request. The interrupt is now in the pending state, which means the CPU has not serviced the interrupt requests
- The NVIC then sends the exception number along with the interrupt request signal to the CPU. When the CPU starts executing the ISR, the pending state of the interrupt is cleared
- If the interrupt signal remains high after completing the ISR, it will be pending, and the ISR will be executed again. Figure 15 shows this for level-triggered interrupts, where the ISR is executed as long as the interrupt signal is high

7.3.3 Exception vector table

PSC3M/P6 MCU has two exception vector tables that store the entry point addresses for all exception handlers for the Cortex®-M33 secure state and the Cortex®-M33 non-secure state. The CPU fetches the appropriate address based on the exception number.

Table 13 Cortex®-M33 secure exception vector

Exception number	Secure exceptions	Exception priority	Vector address
-	Initial stack pointer value	-	Start_Address =CM33_S_VECTOR_TABLE _BASE or CM33's VTOR_S
1	Reset	-4	Start_Address + 0x0004
2	Secure non-maskable interrupt (NMI_S)	-2	Start_Address + 0x0008
3	Secure HardFault	(-1 or -3)	Start_Address + 0x000C

(table continues...)

Table 13 (continued) Cortex®-M33 secure exception vector

Exception number	Secure exceptions	Exception priority	Vector address
4	Secure Memory Management	Configurable (0-7)	Start_Address + 0x0010
5	Secure Bus Fault	Configurable (0-7)	Start_Address + 0x0014
6	Secure Usage fault	Configurable (0-7)	Start_Address + 0x0018
7	Secure Fault	Configurable (0-7)	Start_Address + 0x001C
8-10	Reserved	-	...
11	Secure Supervisory Call (SVCall_S)	Configurable (0-7)	Start_Address + 0x002C
12	Secure Debug Monitor	Configurable (0-7)	Start_Address + 0x0030
13	Reserved	-	...
14	Secure Pending Supervisory (PendSV_S)	Configurable (0-7)	Start_Address + 0x0038
15	Secure Sys Tick Timer (SysTick_S)	Configurable (0-7)	Start_Address + 0x003C
16	Secure IRQ(0)	Configurable (0-7)	Start_Address + 0x0040
...	...	...	...
181	Secure IRQ(165)	Configurable (0-7)	Start_Address + 0x02D4
182	Secure IRQ(166)	Configurable (0-7)	Start_Address + 0x02D8

***Note**: Start address = CM33_S_VECTOR_TABLE_BASE on reset and can be later modified by firmware by updating the CM33's Armv8-M VTOR_S register.*

Secure HardFault can change its priority based on the BFHFNMINS bit in the Armv8-M AIRCR register.

Table 14 Cortex®-M33 non-secure exception vector

Exception number	Secure exceptions	Exception priority	Vector address
-	Initial stack pointer value	-	Start_Address =CM33_S_VECTOR_TABLE_BASE or CM33's VTOR_NS
1	Reset	-4	Start_Address + 0x0004
2	non-maskable interrupt (NMI_NS)	-2	Start_Address + 0x0008
3	HardFault	(-1 or -3)	Start_Address + 0x000C
4	Memory Management	Configurable (0-7)	Start_Address + 0x0010
5	Bus Fault	Configurable (0-7)	Start_Address + 0x0014
6	Usage Fault	Configurable (0-7)	Start_Address + 0x0018
7	Secure Fault	Configurable (0-7)	Start_Address + 0x001C

(table continues...)

Table 14 (continued) Cortex®-M33 non-secure exception vector

Exception number	Secure exceptions	Exception priority	Vector address
8-10	Reserved	-	...
11	Supervisory Call (SVCall_NS)	Configurable (0-7)	Start_Address + 0x002C
12	Debug Monitor	Configurable (0-7)	Start_Address + 0x0030
13	Reserved	-	...
14	Pending Supervisory (PendSV_NS)	Configurable (0-7)	Start_Address + 0x0038
15	Sys Tick Timer (SysTick_NS)	Configurable (0-7)	Start_Address + 0x003C
16	IRQ(0)	Configurable (0-7)	Start_Address + 0x0040
...	...	...	...
181	IRQ(165)	Configurable (0-7)	Start_Address + 0x02D4
182	IRQ(166)	Configurable (0-7)	Start_Address + 0x02D8

***Note**: Start address = CM33_NS_VECTOR_TABLE_BASE on reset and can be later modified by firmware by updating the CM33's Armv8-M VTOR_NS register.*

The first word (4 bytes) in each vector table is not marked as exception number zero. This is because the first word in the exception table is used to initialize the main stack pointer (MSP) value on device reset; it is not considered as an exception. The vector table offset register (VTOR) present as part of the Armv8-M System Control Block registers (SCB) configures the vector table offset from the base address (0x0000). The VTOR value determines the location of the vector table, which can be placed in volatile and non-volatile memory. Note that the VTOR registers can be updated only in privilege CPU mode. The advantage of moving the vector table to volatile memory, such as SRAM, is that the exception handler addresses can be dynamically changed by modifying the vector table contents. However, a non-volatile memory vector table must be modified by a non-volatile memory write. Note that the exception table must be 1024 byte-aligned for the Cortex®-M33. The exception sources (exception numbers 1 to 15) are explained in the Exception sources section. The exceptions marked as reserved are not used, although they have addresses reserved for them in the vector table. The interrupt sources (exception numbers 16 and higher) are explained in the Interrupt sources section.

7.4 Exception sources

This section explains the different exception sources listed in Table 13 and Table 14 (exception numbers 1 to 15). This section also discusses a TrustZone concept called "banking". See the Interrupt/exception target security state and banking section for more information about banking.

7.4.1 Reset exception

A device reset is treated as an exception. The reset exception is always enabled with a fixed priority of –4, the highest priority exception.

A device reset can occur for multiple reasons, such as power-on-reset (POR), an external reset signal on the XRES pin, or a watchdog reset. For PSC3M/P6 MCU, after a device reset, the Cortex®-M33 core is available, which begins the secure boot process. A Cortex®-M33 secure reset exception is triggered. The Cortex®-M33 reset handler then begins the Cortex®-M33 boot code. Once the Cortex®-M33 boot code is completed successfully, the Cortex®-M33 enters the user application code.

7.4.2 Non-maskable interrupt exception

Non-maskable interrupt (NMI) is always enabled with a fixed priority of –2.

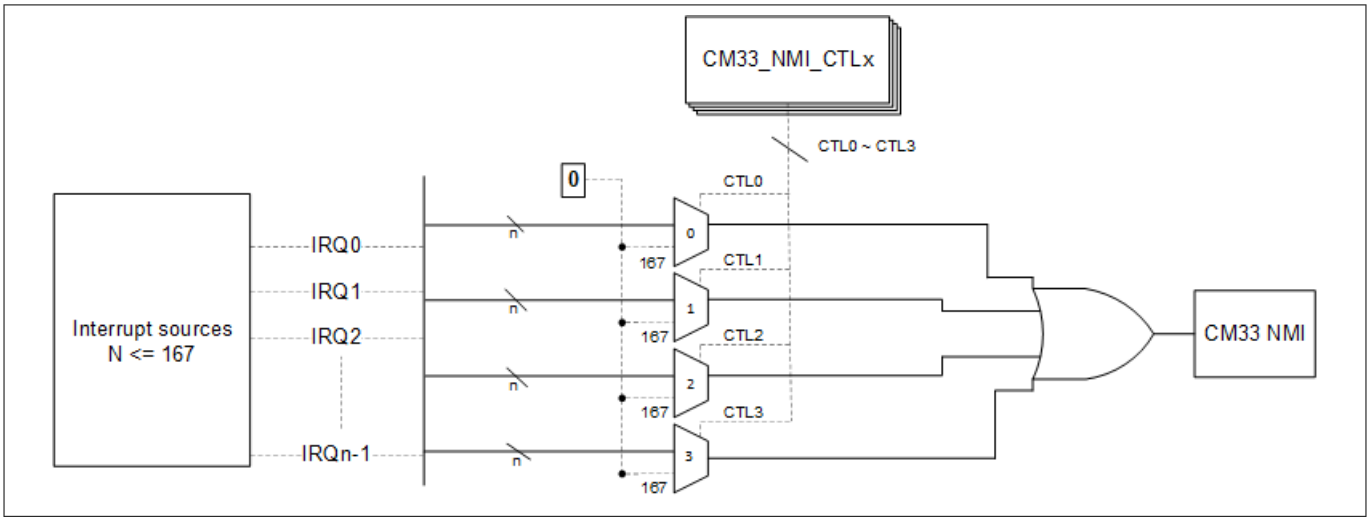


Figure 16 NMI trigger

The Cortex®-M33 provides an option to trigger an NMI exception from up to four of the system interrupts the CPU has access to (IRQ0—IRQn). The NMI exception triggered by the interrupt executes the NMI handler pointed to by the active exception vector table. The CM33_NMI_CTLx.SYSTEM_INT_IDX[9:0] specifies the system interrupt source that triggers the NMI from hardware. The registers have a default value of 1023; that is, if the register is set to 1023, then that particular register does not map any interrupt source to the NMI. There are four such registers, and each can map one interrupt vector to the NMI. NMI is triggered when any of the four interrupts are triggered; that is, the interrupts are logically ORed.

For the CPU, the exception is not banked between security states.

- If the BFHFNMINS bit in the Armv8-M Application Interrupt and Reset Control Register (AIRCR) is set to 0, the NMI exception targets the Secure NMI handler
- If the BFHFNMINS bit in the Armv8-M AIRCR register is set to 1, the NMI exception targets the non-Secure NMI handler

7.4.3 HardFault exception

PSC3M/P6 MCU Armv8-M CPU supports the HardFault exception. HardFault is an always-enabled exception that occurs because of an error during normal or exception processing. HardFault has a priority of at least –1, meaning it has a higher priority than any exception with configurable priority. A HardFault exception is a catch-all exception for different types of fault conditions, which include executing an undefined instruction and accessing an invalid memory address. The CPU does not provide fault status information to the HardFault exception handler, but it does permit the handler to perform an exception return and continue execution in cases where software can recover from the fault situation.

For CPU, the exception can be banked or not, depending on the BFHFNMINS in the AIRCR register.

- If the BFHFNMINS bit in the Armv8-M AIRCR register is set to 0, only Secure HardFault is enabled with a priority of –1. The Secure HardFault then handles all faults that are unable to preempt the current execution. In this scenario, the exception is not banked
- If the BFHFNMINS bit in the Armv8-M AIRCR register is set to 1, secure and non-secure HardFaults are enabled. Secure HardFault has a priority of –3, and non-secure HardFaults will have a priority of –1. The HardFault handler used depends on whether the fault targets a secure or non-secure state. In this scenario, the exception is banked

7.4.4 MemManage fault exception

A MemManage fault is an exception that occurs because of a memory protection-related fault. The fixed memory protection constraints determine this fault for both instruction and data memory transactions. This fault is always used to abort instruction access to Execute Never (XN) memory regions. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

The exception is banked between security states.

7.4.5 BusFault exception

A BusFault is an exception that occurs because of a memory-related fault for an instruction or data memory transaction. This may be from an error detected on a bus in the memory system. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

For the CPU, the exception is not banked between security states.

- If the BFHFNMINS bit in the ARMv8-M AIRCR register is set to 0, the BusFault exception targets the Secure BusFault handler
- If the BFHFNMINS bit in the ARMv8-M AIRCR register is set to 1, the BusFault exception targets the Non-Secure BusFault handler

7.4.6 UsageFault exception

A UsageFault is an exception that occurs because of a fault related to instruction execution. This includes:

- An undefined instruction
- An illegal unaligned access
- Invalid state on instruction execution
- An error on exception return

The following can cause a usage fault when the core is configured to report them:

- An unaligned address on word and half-word memory access
- Division by zero

The priority of the exception is configurable from 0 (highest) to 7 (lowest).

The exception is banked between security states.

7.4.7 SecureFault exception

The SecureFault is an extra exception on Armv8-M CPUs with the security extension, otherwise known as TrustZone. The SecureFault can be triggered by the various security checks that are performed. For example, jumping from a non-secure code to an address in the secure code that is not marked as a valid entry point will trigger the SecureFault. Use caution when handling the SecureFault and make sure that the handler does not inadvertently introduce a security vulnerability. The SecureFault will always target the secure state. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

7.4.8 Supervisor Call (SVCall) exception

SVCall is an always-enabled exception caused when the CPU executes the SVC instruction as part of the application code. Application software uses the SVC instruction to make a call to an underlying operating system and provide a service. This is known as a supervisor call. The SVC instruction enables the application to issue an SVCall that requires privileged access to the system. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

The exception is banked between security states.

7.4.9 DebugMonitor exception

Armv8-M CPUs support the DebugMonitor exception. DebugMonitor is only available if halting debug is disabled and debug monitoring is enabled. This can be done by setting the C_DEBUGEN bit to disabled in the Armv8-M Debug Halting Control and Status Register (DHCSR) and the MON_EN bit to enabled in the Armv8-M Debug Exception and Monitor Control Register (DEMCR). The DebugMonitor exception is handled by dedicated debug monitoring software running on the same core. A debug event causes a DebugMonitor exception only when the priority of the DebugMonitor exception is greater than the current exception priority. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

The exception is not banked between security states. DebugMonitor will target the secure handler if the Secure Debug Monitor Enable (SDME) bit in the Armv8-M DEMCR register is set to 1. If SDME is set to 0, the DebugMonitor exception targets the non-secure handler.

7.4.10 PendSupervisory (PendSV) exception

Armv8-M CPUs support the PendSV exception. PendSV is another supervisor call-related exception similar to SVCall, normally being software-generated. PendSV is always enabled, and its priority is configurable, similar to SVCall. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

The exception is banked between security states.

7.4.11 System Tick (SysTick) exception

SysTick provides a simple, 24-bit decrementing counter for various timekeeping purposes, such as an RTOS tick timer, high-speed alarm timer, or simple counter. The SysTick timer can be configured to generate an interrupt when its count value reaches zero, which is referred to as a SysTick exception. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

The exception is banked between security states. This means there are two SysTick timers, one for each security state.

7.5 Interrupt sources

The PSC3M/P6 MCU supports 167 interrupts from peripherals. The source of each interrupt is listed in Table 15. These system interrupts are mapped directly to the Cortex®-M33 NVIC.

The interrupts include standard interrupts from the on-chip peripherals such as TCPWM, SCB, watchdog, ADC, and so on. The interrupt generated is usually the logical OR of the different peripheral states. The peripheral interrupt status register should be read in the ISR to detect which condition generated the interrupt. These interrupts are level interrupts. The appropriate interrupt registers should be cleared in the ISR to deassert the interrupt. Usually, a write '1' is required to clear the registers. If the interrupt register is not cleared in the ISR, the interrupt remains asserted, and the ISR is executed continuously.

For the Cortex®-M33 CPU, all interrupts are not banked between security states.

As seen from Table 15, 38 interrupts (IRQ0 to IRQ37) can wake up the Cortex®-M33 from CPU DeepSleep power mode. All interrupts can wake up the CPU from CPU Sleep power mode.

Table 15 List of PSC3M/P6 MCU Cortex®-M33 interrupt sources

Cortex®-M33 interrupt	Cortex®-M33 exception number	CPU DeepSleep capable	Interrupt source
NMI	-2	-	Logical OR of four of the following 140 interrupt sources
IRQ0	16	Yes	GPIO Port Non-Secure #0

(table continues...)

Table 15 (continued) List of PSC3M/P6 MCU Cortex®-M33 interrupt sources

Cortex®-M33 interrupt	Cortex®-M33 exception number	CPU DeepSleep capable	Interrupt source
IRQ1	17	Yes	GPIO Port Non-Secure #1
IRQ2	18	Yes	GPIO Port Non-Secure #2
IRQ3	19	Yes	GPIO Port Non-Secure #3
IRQ4	20	Yes	GPIO Port Non-Secure #4
IRQ5	21	Yes	GPIO Port Non-Secure #5
IRQ6	22	Yes	GPIO Port Non-Secure #6
IRQ7	23	Yes	GPIO Port Non-Secure #7
IRQ8	24	Yes	GPIO Port Non-Secure #8
IRQ9	25	Yes	GPIO Port Non-Secure #9
IRQ10	26	Yes	GPIO Port Non-Secure #10
IRQ11	27	Yes	GPIO Port Non-Secure #11
IRQ12	28	Yes	GPIO Port Non-Secure #12
IRQ13	29	Yes	GPIO Port Non-Secure #13
IRQ14	30	Yes	GPIO Port Secure #0
IRQ15	31	Yes	GPIO Port Secure #1
IRQ16	32	Yes	GPIO Port Secure #2
IRQ17	33	Yes	GPIO Port Secure #3
IRQ18	34	Yes	GPIO Port Secure #4
IRQ19	35	Yes	GPIO Port Secure #5
IRQ20	36	Yes	GPIO Port Secure #6
IRQ21	37	Yes	GPIO Port Secure #7
IRQ22	38	Yes	GPIO Port Secure #8
IRQ23	39	Yes	GPIO Port Secure #9
IRQ24	40	Yes	GPIO Port Secure #10
IRQ25	41	Yes	GPIO Port Secure #11
IRQ26	42	Yes	GPIO Port Secure #12
IRQ27	43	Yes	GPIO Port Secure #13
IRQ28	44	Yes	GPIO Supply Detect
IRQ29	45	Yes	GPIO Non-Secure - All ports
IRQ30	46	Yes	GPIO Secure – All ports
IRQ31	47	Yes	SCB#0
IRQ32	48	Yes	MCWDT
IRQ33	49	Yes	Backup domain

(table continues...)

Table 15 (continued) List of PSC3M/P6 MCU Cortex®-M33 interrupt sources

Cortex®-M33 interrupt	Cortex®-M33 exception number	CPU DeepSleep capable	Interrupt source
IRQ34	50	Yes	Inter Process Communication #0
IRQ35	51	Yes	Inter Process Communication #1
IRQ36	52	Yes	WDT
IRQ37	53	Yes	LPCOMP
IRQ38	54	-	SRSS (LVD, CLKCAL)
IRQ39	55	-	SCB#1
IRQ40	56	-	SCB#2
IRQ41	57	-	SCB#3
IRQ42	58	-	SCB#4
IRQ43	59	-	SCB#5
IRQ44	60	-	I3C #0
IRQ45	61	-	FLASH Macro
IRQ46	62	-	CPUSS DMA#0, Channel#0
IRQ47	63	-	CPUSS DMA#0, Channel#1
IRQ48	64	-	CPUSS DMA#0, Channel#2
IRQ49	65	-	CPUSS DMA#0, Channel#3
IRQ50	66	-	CPUSS DMA#0, Channel#4
IRQ51	67	-	CPUSS DMA#0, Channel#5
IRQ52	68	-	CPUSS DMA#0, Channel#6
IRQ53	69	-	CPUSS DMA#0, Channel#7
IRQ54	70	-	CPUSS DMA#0, Channel#8
IRQ55	71	-	CPUSS DMA#0, Channel#9
IRQ56	72	-	CPUSS DMA#0, Channel#10
IRQ57	73	-	CPUSS DMA#0, Channel#11
IRQ58	74	-	CPUSS DMA#0, Channel#12
IRQ59	75	-	CPUSS DMA#0, Channel#13
IRQ60	76	-	CPUSS DMA#0, Channel#14
IRQ61	77	-	CPUSS DMA#0, Channel#15
IRQ62	78	-	CPUSS DMA#1, Channel#0
IRQ63	79	-	CPUSS DMA#1, Channel#1
IRQ64	80	-	CPUSS DMA#1, Channel#2
IRQ65	81	-	CPUSS DMA#1, Channel#3
IRQ66	82	-	CPUSS DMA#1, Channel#4

(table continues...)

Table 15 (continued) List of PSC3M/P6 MCU Cortex®-M33 interrupt sources

Cortex®-M33 interrupt	Cortex®-M33 exception number	CPU DeepSleep capable	Interrupt source
IRQ67	83	-	CPUSS DMA#1, Channel#5
IRQ68	84	-	CPUSS DMA#1, Channel#6
IRQ69	85	-	CPUSS DMA#1, Channel#7
IRQ70	86	-	CPUSS DMA#1, Channel#8
IRQ71	87	-	CPUSS DMA#1, Channel#9
IRQ72	88	-	CPUSS DMA#1, Channel#10
IRQ73	89	-	CPUSS DMA#1, Channel#11
IRQ74	90	-	CPUSS DMA#1, Channel#12
IRQ75	91	-	CPUSS DMA#1, Channel#13
IRQ76	92	-	CPUSS DMA#1, Channel#14
IRQ77	93	-	CPUSS DMA#1, Channel#15
IRQ78	94	-	SRAM#0 PPU
IRQ79	95	-	SRAM#1 PPU
IRQ80	96	-	Cortex®-M33 Floating Point Unit
IRQ81	97	-	Cortex®-M33 CTI #0
IRQ82	98	-	Cortex®-M33 CTI #1
IRQ83	99	-	CPUSS Faults
IRQ84	100	-	CPUSS PPU
IRQ85	101	-	CPUSS Master Security Controller
IRQ86	102	-	TCPWM #0, Group 0, Counter #0
IRQ87	103	-	TCPWM #0, Group 0, Counter #1
IRQ88	104	-	TCPWM #0, Group 0, Counter #2
IRQ89	105	-	TCPWM #0, Group 0, Counter #3
IRQ90	106	-	TCPWM #0, Group 0, Counter #4
IRQ91	107	-	TCPWM #0, Group 0, Counter #5
IRQ92	108	-	TCPWM #0, Group 0, Counter #6
IRQ93	109	-	TCPWM #0, Group 0, Counter #7
IRQ94	110	-	TCPWM #0, Group 1, Counter #256
IRQ95	111	-	TCPWM #0, Group 1, Counter #257
IRQ96	112	-	TCPWM #0, Group 1, Counter #258
IRQ97	113	-	TCPWM #0, Group 1, Counter #259
IRQ98	114	-	TCPWM #0, Group 1, Counter #260
IRQ99	115	-	TCPWM #0, Group 1, Counter #261

(table continues...)

Table 15 (continued) List of PSC3M/P6 MCU Cortex®-M33 interrupt sources

Cortex®-M33 interrupt	Cortex®-M33 exception number	CPU DeepSleep capable	Interrupt source
IRQ100	116	-	TCPWM #0, Group 1, Counter #262
IRQ101	117	-	TCPWM #0, Group 1, Counter #263
IRQ102	118	-	TCPWM #0, Group 2, Counter #512
IRQ103	119	-	TCPWM #0, Group 2, Counter #513
IRQ104	120	-	TCPWM #0, Group 2, Counter #514
IRQ105	121	-	TCPWM #0, Group 2, Counter #515
IRQ106	122	-	TCPWM #0, Group 2, Counter #516
IRQ107	123	-	TCPWM #0, Group 2, Counter #517
IRQ108	124	-	TCPWM #0, Group 2, Counter #518
IRQ109	125	-	TCPWM #0, Group 2, Counter #519
IRQ110	126	-	SRSS Main PPU
IRQ111	127	-	Crypto
IRQ112	128	-	TRNG
IRQ113	129	-	CAN#0, Interrupt#0, Channel#0
IRQ114	130	-	CAN#0, Interrupt#1, Channel#0
IRQ115	131	-	CAN#0, Interrupt#0, Channel#1
IRQ116	132	-	CAN#0, Interrupt#1, Channel#1
IRQ117	133	-	CAN#0, Consolidated interrupt #0
IRQ118	134	-	CAN#0, Consolidated interrupt #1
IRQ119	135	-	CORDIC
IRQ120	136	-	TCPWM #0, MOTIF #1, GRP #1, SR0
IRQ121	137	-	TCPWM #0, MOTIF #1, GRP #1, SR1
IRQ122	138	-	TCPWM #0, MOTIF #1, GRP #2, SR0
IRQ123	139	-	TCPWM #0, MOTIF #1, GRP #2, SR1
IRQ124	140	-	Combined HPPASS (AC, Error conditions)
IRQ125	141	-	Combined SAR Entry and FIR results
IRQ126	142	-	Individual SAR Entry result [0]
IRQ127	143	-	Individual SAR Entry result [1]
IRQ128	144	-	Individual SAR Entry result [2]
IRQ129	145	-	Individual SAR Entry result [3]
IRQ130	146	-	Individual SAR Entry result [4]
IRQ131	147	-	Individual SAR Entry result [5]
IRQ132	148	-	Individual SAR Entry result [6]

(table continues...)

Table 15 (continued) List of PSC3M/P6 MCU Cortex®-M33 interrupt sources

Cortex®-M33 interrupt	Cortex®-M33 exception number	CPU DeepSleep capable	Interrupt source
IRQ133	149	-	Individual SAR Entry result [7]
IRQ134	150	-	FIR result [0]
IRQ135	151	-	FIR result [1]
IRQ136	152	-	Combined SAR range
IRQ137	153	-	Individual SAR range [0]
IRQ138	154	-	Individual SAR range [1]
IRQ139	155	-	Individual SAR range [2]
IRQ140	156	-	Individual SAR range [3]
IRQ141	157	-	Individual SAR range [4]
IRQ142	158	-	Individual SAR range [5]
IRQ143	159	-	Individual SAR range [6]
IRQ144	160	-	Individual SAR range [7]
IRQ145	161	-	Combined CSG DAC
IRQ146	162	-	Individual CSG DAC [0]
IRQ147	163	-	Individual CSG DAC [1]
IRQ148	164	-	Individual CSG DAC [2]
IRQ149	165	-	Individual CSG DAC [3]
IRQ150	166	-	Individual CSG DAC [4]
IRQ151	167	-	Individual CSG DAC [5]
IRQ152	168	-	Individual CSG DAC [6]
IRQ153	169	-	Individual CSG DAC [7]
IRQ154	170	-	Individual CSG DAC [8]
IRQ155	171	-	Combined CSG CMP
IRQ156	172	-	Combined FIFO
IRQ157	173	-	Individual 3P3Z result interrupt [0]
IRQ158	174	-	Individual 3P3Z result interrupt [1]
IRQ159	175	-	Individual 3P3Z result interrupt [2]
IRQ160	176	-	Individual DAC buffer interrupt [0]
IRQ161	177	-	Individual DAC buffer interrupt [1]
IRQ162	178	-	SVGS interrupt
IRQ163	179	-	MXAES
IRQ164	180	-	USBFS HI
IRQ165	181	-	USBFS MED

(table continues...)

Table 15 (continued) List of PSC3M/P6 MCU Cortex®-M33 interrupt sources

Cortex®-M33 interrupt	Cortex®-M33 exception number	CPU DeepSleep capable	Interrupt source
IRQ166	182	-	USBFS LOW

7.6 Interrupt/exception target security state and banking

For the Cortex®-M33 CPU, exceptions and interrupts can target the secure or non-secure ISR handler. If an exception is said to be banked, this means that the resources for the exception are duplicated and both have a secure and non-secure version. If an exception is said to not be banked, this means the exception can be configured to only target the secure or non-secure handler.

The following are included in the banked exceptions:

- Banked enabled, pending, and active bits
- A banked SHPRn.PRI field
- A banked exception vector
- A handler for each security state

Table 16 Banked exception

Exception	Banked
Reset	No
HardFault	Yes
NMI	No
MemManagment Fault	Yes
BusFault	No
UsageFault	Yes
SecureFault	No
SVCall	Yes
DebugMonitor	No
PendSV	Yes
SysTick	Yes
IRQ(0)	No
…	...
IRQ(n)	No

***Note**: HardFault is a banked exception when the BFHFNMINS bit is set to 1. When the BFHFNMINS bit is set to 0, all HardFaults go to the secure handler.*

BusFault, NMI, and HardFault can be configured using the BFHFNMINS bit in the Armv8-M Application Interrupt and Reset Control Register (AIRCR). When the BFHFNMINS is set to 0, BusFault, NMI, and HardFault target their respective secure handlers. When the BFHFNMIS is set to 1, BusFault and NMI target their respective non-secure handlers, while HardFault becomes a banked exception with both secure and non-secure handlers.

SecureFault always targets the secure handler.

DebugMonitor targets the secure handler if the Secure Debug Monitor Enable (SDME) bit in the Armv8-M DEMCR register is set to 1. If SDME is set to 0, the DebugMonitor exception targets the non-secure handler.

7.6.1 Interrupt target security state

On the Cortex®-M33 CPU, each interrupt can be configured to target a secure or non-secure state. As a result, an interrupt can be configured to service either the Interrupt Service Routine (ISR) in the secure vector table or the ISR in the non-secure vector table. In the Armv8-M architecture, there are 16 Interrupt Target non-secure registers (ITNSn), which are used to configure the specific vector table that an interrupt uses. Each ITNSn register has 32 bits. The interrupt number is represented as 32n + m, where m is the bit number in the register and n is the register instance. Setting bit m to 0 configures the interrupt target for the secure state, while setting bit m to 1 configures the interrupt target for the non-secure state.

7.6.2 GPIO secure and non-secure interrupts

For the CPU, each GPIO pin can be configured to be secure or non-secure. A single GPIO port can have a combination of secure and non-secure pins. As a result, there are two interrupt sources going into the NVIC for each GPIO port, one for secure and one for non-secure GPIO interrupts. These two interrupt sources are present in Table 15. Configuring a pin's security state also configures the GPIO port interrupt source that will be triggered. As a result, a secure GPIO pin can only trigger the secure GPIO port interrupt, and a non-secure GPIO pin can only trigger the non-secure GPIO port interrupt.

7.7 Interrupt/exception priority

Exception priority is useful for exception arbitration when there are multiple exceptions that need to be serviced by the CPU. All exceptions other than Reset, NMI, and HardFault can be assigned a configurable priority level. The Reset and NMI have a fixed priority of –4 and –2, respectively. HardFaults on the CPU can have priority of –1 and –3. For more information on HardFault configurations with TrustZone, see the HardFault exception section. Lower priority numbers represent higher priorities. This means that the Reset, NMI, and HardFault exceptions have the highest priorities. The other exceptions can be assigned a configurable priority level between 0 and 7.

The Armv8-M CPU supports nested exceptions, in which a higher-priority exception can obstruct (interrupt) the currently active exception handler. This pre-emption does not happen if the incoming exception priority is the same as or lower than the active exception. The CPU resumes execution of the lower-priority exception handler after servicing the higher-priority exception. With eight priority levels supported, this allows up to eight nested exceptions. When the CPU receives two or more exception requests of the same priority, the lowest exception number is serviced first.

The priority of the exception numbers 4 to 15 can be configured by writing to the three System Handler Priority Registers (SHPR1, SHPR2, SHPR3). Each SHPRn stores the priority of four exceptions, PRI_N. Each priority number corresponds to an exception number. SHPR1 stores the priority for exceptions PRI_4 to PRI_7. SHPR2 stores the priority for exceptions PRI_8 to PRI_11. SHPR3 stores the priority for exceptions PRI_12 to PRI_15.

The priority of the interrupts can be configured by writing to the respective Interrupt Priority registers (NVIC_IPRn). This is a group of 124 32-bit registers, with each register storing the priority values of four interrupts, as specified in Table 17. The interrupt number corresponding to each field is represented as 4n + m, where m is the field number in the register and n is the register instance.

Table 17 Interrupt priority register bit definitions for NVIC_IPRn

Bits	Name	Description
7:5	PRI_N0	Priority of interrupt number 4N + 0

(table continues...)

Table 17 (continued) Interrupt priority register bit definitions for NVIC_IPRn

Bits	Name	Description
15:13	PRI_N1	Priority of interrupt number 4N + 1
23:21	PRI_N2	Priority of interrupt number 4N + 2
31:29	PRI_N3	Priority of interrupt number 4N + 3

7.8 Enabling and disabling interrupts

The NVICs of the Armv8-M CPUs provide registers to individually enable and disable the interrupts in software. If an interrupt is not enabled, the NVIC does not process the interrupt requests on that interrupt line. The Armv8-M Interrupt Set Enable Register (ISERn) and the Armv8-M Interrupt Clear-Enable Register (ICERn) are used to enable and disable the interrupts, respectively. There are 16 instances of the 32-bit ISER/ICER registers in the Armv8-M architecture. The interrupt number corresponding to each bit is represented as 32n+m, where m is the bit number in the register and n is the register instance. These registers can also be read in software to get the enable status of the interrupts. Enabling and disabling interrupts/exceptions shows the register access properties for these two registers. Note that writing zero to these registers has no effect. For CPU, ISER and ICER are not banked between security states.

Table 18 Interrupt enable/disable registers

Register	Operation	Bit value	Comment
Interrupt Set Enable Register (ISERn)	Write	1	To enable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled
	Write	1	To disable the interrupt
		0	No Effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled

The ISER and ICER registers are applicable only for interrupts. These registers cannot be used to enable or disable the exception numbers 1 to 15.

The Armv8-M SHCSR register can be configured to enable and disable SecureFault, UsageFault, BusFault, and MemFault. The SHCSR register has the SECUREFAULTENA, USGFAULTENA, BUSFAULTENA, and MEMFAULTENA bits, where setting them enables the exception while clearing it disables the exception. For the CPU, the register is banked between security states on a bit-by-bit basis. The SECUREFAULTENA and BUSFAULTENA bits are not banked between security states. The USGFAULTENA and MEMFAULTENA bits are banked between security states.

SysTick can be configured with the ENABLE bit in the Armv8-M SysTick Control and Status Register (SYST_CSR). For the CPU, this register is banked between security states.

DebugMonitor can be enabled with the MON_EN bit in the Armv8-M Debug Exception and Monitor Control Register (DEMCR). Note that the C_DEBUGEN bit needs to be disabled in the ARMv8-M Debug Halting Control and Status Register (DHCSR) for the DebugMonitor to function. For the CPU, the registers are not banked between security states.

Note that HardFault, NMI, SVCall, and PendSV are always enabled and therefore do not have an exception enable bit.

Interrupts and exceptions can also be disabled using the Armv8-M special purpose mask register (PRIMASK, FAULTMASK, and BASEPRI). The PRIMASK register can be used as a global exception enable register to mask all the configurable priority exceptions, irrespective of whether they are enabled. Configurable priority exceptions include all the exceptions except Reset, NMI, and HardFault listed in Table 15. The FAULTMASK register can be used to disable all faults and interrupts except for the NMI exception. The BASEPRI register can be used to disable interrupts based on priority level. Only interrupts with a priority value lower (higher priority) than what is in the BASEPRI register can trigger. When PRIMASK, FAULTMASK, or BASEPRI are set, none of the masked exceptions can be serviced by the CPU. The masked exceptions remain in a pending state until the mask is cleared. For the CPU, the registers are banked between security states.

7.9 Interrupt/exception states

Each exception can be in one of the following states:

Table 19 Interrupt/exception states

Exception state	Meaning
Inactive	The exception is not active or pending. Either the exception is disabled or the enabled exception has not been triggered.
Pending	The exception request has been received by the CPU/NVIC, and the exception is waiting to be serviced by the CPU.
Active	An exception request has been serviced by the CPU but whose exception handler execution is not yet complete. A high-priority exception can interrupt the execution of a lower priority exception. In this case, both the exceptions are in the active state.
Active and Pending	The exception is being serviced by the processor, and there is a pending request from the same source during its exception handler execution.

The Armv8-M Interrupt Control State Register (ICSR) contains status bits describing the various exception states.

- The VECTACTIVE bits ([8:0]) in the ICSR store the exception number for the current executing exception. This value is zero if the CPU does not execute any exception handler (the CPU is in thread mode). Note that the value in the VECTACTIVE bitfields is the same as the value in bits [8:0] of the Interrupt Program Status Register (IPSR), which is also used to store the active exception number
- The VECTPENDING bits ([20:12]) in the ICSR store the exception number of the highest priority pending exception. This value is zero if there are no pending exceptions
- The ISRPENDING bit (bit 22) in the ICSR indicates if a NVIC-generated interrupt is in a pending state

7.9.1 Pending interrupts/exception

When a peripheral generates an interrupt request signal to the NVIC or an exception event occurs, the corresponding exception enters the pending state. When the CPU starts executing the corresponding exception handler routine, the exception is changed from the pending state to the active state. The NVIC allows software pending of all interrupt lines by providing separate register bits for setting and clearing the pending states of the interrupts. The Interrupt Set-Pending register (ISPRn) and the Interrupt Clear-Pending register (ICPRn) are used to set and clear the pending status of the interrupt lines. There are 16 instances of the 32-bit ISPR/ICPR registers, where each bit corresponds to an interrupt. The interrupt number corresponding to each bit is represented as 32n + m, where m is the bit number in the register and n is the register instance. Table 20 shows the register access properties for these two registers. Note that writing zero to these registers has no effect. For the CPU, ISPR and ICPR are not banked between security states.

Table 20 Interrupt set pending/clear pending registers

Register	Operation	Bit value	Comment
Interrupt Set-Pending Register (ISPRn)	Write	1	To put an interrupt to pending state
		0	No effect
	Read	1	Interrupt is pending
		0	Interrupt is not pending
Interrupt Clear-Pending Register (ICPRn)	Write	1	To clear a pending interrupt
		0	No effect
	Read	1	Interrupt is pending
		0	Interrupt is not pending

Setting the pending bit when the same bit is already set results in only one execution of the ISR. The pending bit can be updated regardless of whether the corresponding interrupt is enabled. If the interrupt is not enabled, the interrupt line does not move to the pending state until it is enabled by writing to the ISER register. Note that the ISPR and ICPR registers are used only for peripheral interrupts. These registers cannot be used for pending exception numbers 1 to 15.

The pending bits for exceptions 1 to 15 can be found in the Armv8-M SHCSR, ICSR, and DEMCR registers.

The SHCSR has the HARDFAULTPENDED, SECUREFAULTPENDED, SVCALLPENDED, BUSFAULTPENDED, MEMFAULTPENDED, and USGFAULTPENDED bits, which are the pending bits for the exceptions with the same name. There are no clear pending registers for these exceptions. Writing and clearing these bits sets and clears the pending state of the exception.

The ICSR has the PENDNMISET, PENDSVSET, and PENDSTSET bits, which set the NMI, PendSV, and SysTick exceptions to the pending state when writing one. The ICSR also has the PENDNMICLR, PENDSVCLR, and PENDSTCLR bits, which clear the NMI, PendSV, and SysTick exceptions when writing one.

The DEMCR register has the MON_PEND bits, which are used to set or clear the pending state of the DebugMonitor.

7.10 Stack usage for interrupts/exceptions

When the CPU executes the main code (in thread mode) and an exception request occurs, the CPU stores the state of its general-purpose registers in the stack. It then starts executing the corresponding exception handler (in handler mode). The CPU pushes the contents of the eight 32-bit internal registers into the stack. These registers are the Program and Status Register (PSR), ReturnAddress, Link Register (LR or R14), R12, R3, R2, R1, and R0. Armv8-M CPUs without TrustZone have two stack pointers: the Main Stack Pointer (MSP) and the Process Stack Pointer (PSP). The Armv8-M CPUs with TrustZone enabled have four stack pointers to separate between secure and non-secure states: MSP_NS, MSP_S, PSP_NS, and PSP_S. Only one of the stack pointers can be active at a time. When in thread mode, the Active Stack Pointer bit in the Control register is used to define the current active stack pointer. When in handler mode, the MSP is always used as the stack pointer. The stack pointer always grows downward and points to the address that has the last pushed data.

When the CPU is in thread mode and an exception request comes, the CPU uses the stack pointer defined in the control register to store the general-purpose register contents. After the stack push operations, the CPU enters handler mode to execute the exception handler. When another higher-priority exception occurs while executing the current exception, the MSP is used for stack push/pop operations because the CPU is already in handler mode. See the CPU subsystem (CPUSS) section for details.

7.11 Interrupts and low-power modes

The CPU wake up from low-power modes when certain peripheral interrupt requests are generated. The Wakeup Interrupt Controller (WIC) block generates a wakeup signal that causes the CPU to enter the Active mode when one or more wakeup sources generate an interrupt signal. After entering Active mode, the ISR of the peripheral interrupt is executed.

The Wait For Interrupt (WFI) or Wait For Event (WFE) instruction, executed by the CPU, triggers the transition into CPU Sleep and CPU DeepSleep modes. Both the WFI and WFE instructions can wake up on interrupts. However, the WFE requires the interrupts to be unmasked in the CPU's Exception Mask register (PRIMASK). In addition, the WFE instruction puts the CPU to sleep based on the status of an event bit and wakes up from an event signal, typically sent by the other CPU. WFI does not require PRIMASK unmasking and can wake up the CPU from any pending interrupt masked to the NVIC or WIC. However, WFI cannot wake up the CPU from event signals from other CPUs. The sequence of entering the different low-power modes is detailed in the Device power modes.

Chip low-power modes have two categories of interrupt sources:

- Interrupt sources that are available in the CPU Active, CPU Sleep, and CPU DeepSleep modes
- Interrupt sources that are available only in the CPU Active and CPU Sleep modes

Refer to the Interrupt sources for interrupts that support DeepSleep.

When using the WFE instruction, make sure to call the WFE instruction twice to enter and exit Sleep/DeepSleep modes. This behavior comes from the event register implementation in the Armv8-M architecture.

According to the Armv8-M architecture reference manual (Section B2.1.1):

- A reset clears the event register
- Any WFE wakeup event, or the execution of an exception return instruction, sets the event register
- A WFE instruction clears the event register
- Software cannot read or write the value of the event register directly

Therefore, the first WFE instruction puts the CPU to sleep, and the second WFE clears the event register after a WFE wakeup, which sets the event register. So the next WFE will put the core to sleep.

7.12 Interrupt/exception - initialization/configuration

This section covers the different steps involved in initializing and configuring exceptions in the PSC3M/P6 MCU family.

1. Configuring the Exception Vector Table Location: The first step in using exceptions is to configure the vector table location as required, either in volatile or non-volatile memory. This configuration is done as described in the Exception vector table section. The vector table should be placed in volatile memory if the application must change the vector addresses dynamically. If the table is located in non-volatile memory, such as flash, then a flash write operation is required to modify the vector table contents

***Note**: Only secure applications can do the flash write operation*

2. Configuring individual exceptions: The next step is to configure individual exceptions required in an application, as explained in the earlier sections:

a. Configure the exception or interrupt source: This includes setting up the interrupt generation conditions. The register configuration depends on the specific exception required. Refer to the respective peripheral section for more information about the interrupt configuration supported by them

b. For the Cortex®-M33 CPU, configure the NVIC to use the secure or non-secure vector table when an interrupt source is triggered, as described in the interrupt target security state

c. Define the exception handler function and write the address of the function to the exception vector table. Table 15 gives the exception vector table format; write the exception handler address to the appropriate exception number entry in the table

d. Set up the exception priority, as explained in the Interrupt/exception priority section

e. Enable the exception, as explained in Enabling and disabling interrupts/exceptions

8 Fault monitoring

Fault monitoring allows you to monitor various faults generated within the device and take actions based on the fault reported. In addition to reporting faults, the fault structures in PSC3M/P6 MCU provide a mechanism to log data from the fault sources and optionally perform a soft reset.

The PSC3M/P6 MCU family supports one centralized fault reporting/monitoring structure that monitors faults generated within the device. The fault report structure can monitor and report faults from up to 35 sources. PSC3M/P6 MCU supports 35 fault sources.

8.1 Features

PSC3M/P6 MCU fault report structure supports the following features:

- Monitoring protection unit violations, peripheral-specific errors, memory controller-specific errors, and so on
- Four 32-bit data registers for recording fault information
- Soft reset on fault detection while retaining the fault information
- Interrupt generation on fault detection
- Trigger output to the DW for fault data transfer
- Fault-detected output to a pin for external fault handling

8.2 Architecture

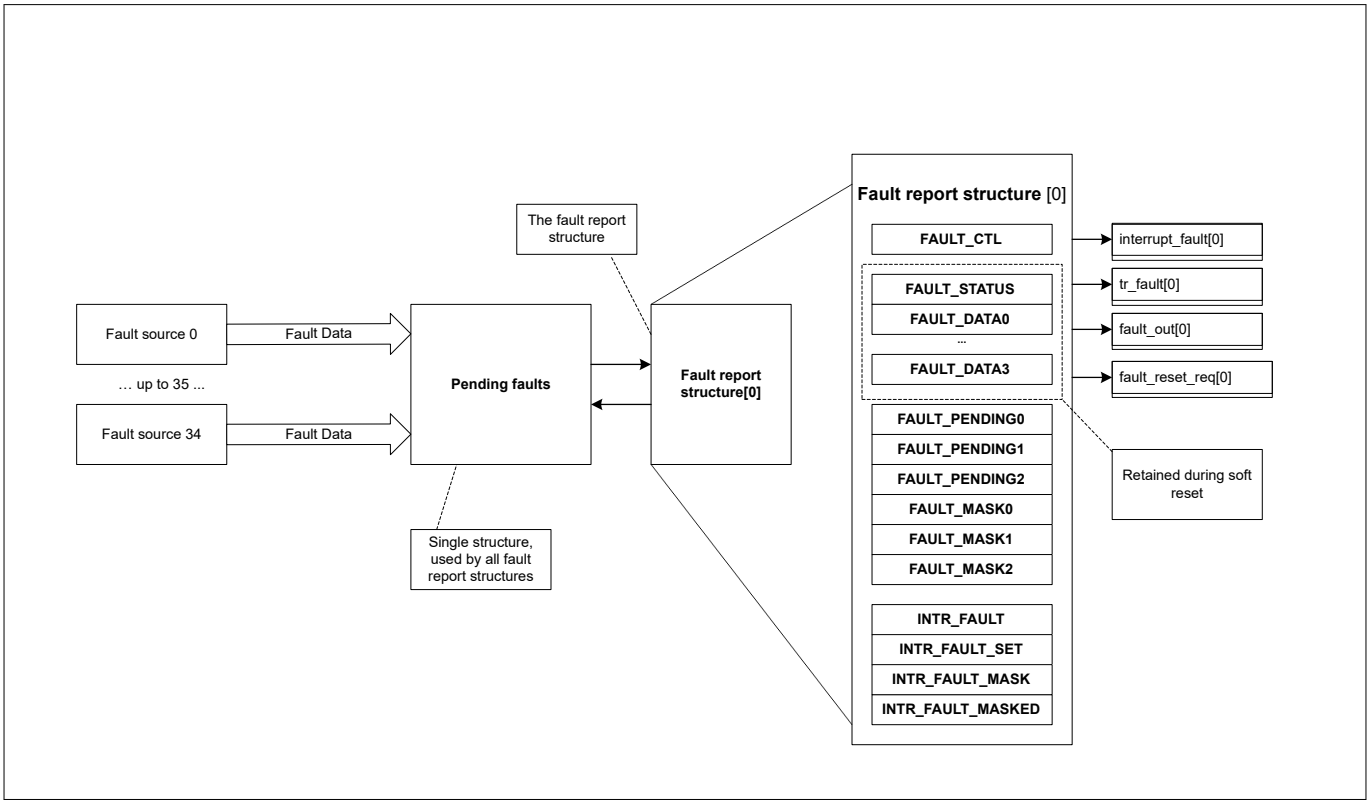


Figure 17 Fault report architecture

The PSC3M/P6 MCU family employs a centralized fault report structure to enable systemwide, consistent fault handling and simplify software development. The fault reporting hardware can be configured so that a single fault interrupt handler services all faults. This structure provides the fault source and additional fault-specific

information through a single set of MMIO registers, eliminating the need for an iterative search. All pending faults can be accessed from the same set of MMIO registers.

The fault structure captures faults associated with the following:

- MPC/PPC protection violations
- Peripheral-specific errors
- Memory controller-specific errors
- ECC errors
- Time-out errors

The fault structure can be configured to capture one or more specific faults as shown in Table 21. When the fault structure is configured to capture a fault, the occurrence is recorded as a pending fault. If all other faults are processed or if no other faults are pending, the fault data is stored in the fault structure registers. A successful capture can optionally trigger an interrupt for further processing by the Cortex® -M33, depending on the application requirement.

In addition to capturing fault data, the fault structure can perform a soft reset while retaining fault information.

This reset generates the RESET_ACT_FAULT reset cause in the SRSS_RES_CAUSE register.

8.2.1 Fault report

The PSC3M/P6 MCU family supports a single fault report structure, which includes a dedicated set of control and status registers. This fault report structure captures one fault at a time. The captured fault information includes the following:

- A fault validity bit (`FAULT_STRUCT0_STATUS.VALID`) that indicates whether a fault has been captured. This bit is set whenever a fault is captured and must be cleared after processing the fault information. New faults are captured only when this bit is set to '0'
- A 7-bit index, as shown in Table 21, that identifies the fault source (FAULT_STRUCT0_STATUS.IDX)
- Additional fault information (`FAULT_STRUCT0_DATA0` through `FAULT_STRUCT0_DATA3`) that describes fault-specific details. This information is fault-source-specific, and the data field encodings are provided in the Fault encodings section
- A writable fault register (`FAULT_STRUCT0_STATUS`) that allows software to trigger a fault

8.2.2 Signaling interface

In addition to capturing fault information, the fault report structure includes a signaling interface to notify the rest of the system about the captured fault. The interface supports the following functionalities:

- **Fault interrupt (`interrupt_fault[0]`)**:
  - Use the `FAULT_STRUCT0_INTR`, `FAULT_STRUCT0_INTR_SET`, `FAULT_STRUCT0_INTR_MASK`, and `FAULT_STRUCT0_INTR_MASKED` registers to monitor, set, and mask the fault structure's interrupt
  - When the interrupt is configured, an interrupt request is sent to the NVIC when `FAULT_STRUCT0_STATUS.VALID` is set
  - Only one cause exists for the interrupt, indicating that a fault has been detected
  - The fault report structure can be read within the interrupt handler to determine the fault source
- **DMA trigger (`tr_fault[0]`)**:
  - Enable the trigger by setting `FAULT_STRUCT0_CTL.TR_EN`
  - When the trigger is enabled, it activates the DMA trigger when `FAULT_STRUCT0_STATUS.VALID` is set
  - The trigger can connect to a DMA controller, which transfers the captured fault information from the fault report structure to memory while clearing the `FAULT_STRUCT0_STATUS.VALID` field
  - For failure analysis, use a memory location that is retained during a warm or soft reset

- **Chip output signal (`fault_out[0]`)**:
  - Enable the output signal by setting `FAULT_STRUCT0_CTL.OUT_EN`
  - When enabled, the fault structure outputs a signal when `FAULT_STRUCT0_STATUS.VALID` is set
  - This signal can be routed out of the device using the HSIOM to communicate non-recoverable faults to off-chip components, potentially resulting in a chip reset
- **Fault reset request (`fault_reset_req[0]`)**:
  - Enable the reset capability by setting `FAULT_STRUCT0_CTL.RESET_REQ_EN`
  - When enabled, `FAULT_STRUCT0_STATUS.VALID` triggers a soft reset, which is recorded as `RESET_ACT_FAULT` in the `SRSS_RES_CAUSE` register
  - The `FAULT_STRUCT0_STATUS` and `FAULT_STRUCT0_DATA` registers are retained through this reset

The fault report structure offers four distinct signaling interfaces, each with its own enabling functionality. Any enabled interface is activated when `FAULT_STRUCT0_STATUS.VALID` is set.

Since the device has a single `fault_reset_req` signal, the individual `fault_reset_req[x]` signals from multiple fault report structures are combined into a central `fault_reset_req` signal, as shown in Figure 18.

***Note**: 'x' represents different fault report structures sharing the same centralized structure.*

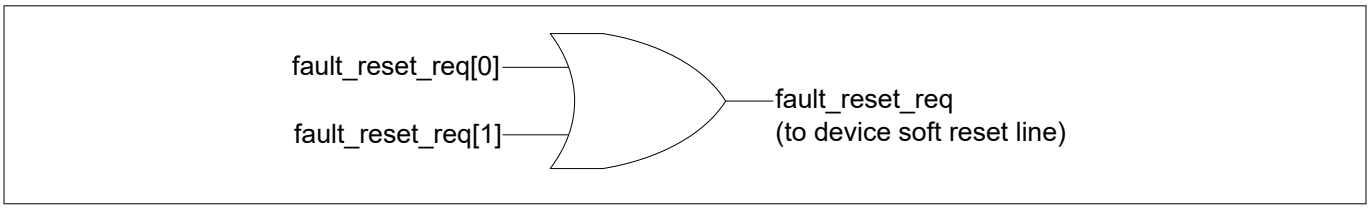


Figure 18 Fault reset

8.2.3 Monitoring

The fault report structure keeps track of all pending faults in the system. The `FAULT_STRUCT0_PENDING0`, `FAULT_STRUCT0_PENDING1`, and `FAULT_STRUCT0_PENDING2` registers show the fault sources that are currently pending. Each register provides a single pending bit for up to 35 fault sources. These `FAULT_STRUCT0_PENDINGx` registers are mirrored in the fault report structure. The fault source numbering scheme matches the numbering scheme of `FAULT_STRUCT0_STATUS.IDX` (see Table 21 for details).

Pending fault sources are not automatically captured by the fault structure. When a pending fault is captured by the fault structure, the associated pending bit is cleared to `0`.

The fault report structure can selectively capture faults. The `FAULT_STRUCT0_MASK0`, `FAULT_STRUCT0_MASK1`, and `FAULT_STRUCT0_MASK2` registers indicate which pending fault sources are enabled for capture by the fault structure. These faults are referred to as "enabled" faults. The fault structure includes its own set of `FAULT_STRUCT0_MASK` registers.

The fault structure only captures "enabled" faults when `FAULT_STRUCT0_STATUS.VALID` is `0`. Once a fault is captured, hardware sets `FAULT_STRUCT0_STATUS.VALID` to `1` and clears the corresponding pending bit to `0`. After the fault structure is processed, firmware or a DMA transfer should clear the `FAULT_STRUCT0_STATUS.VALID` bit to `0`.

Note that fault capturing is independent of the `FAULT_STRUCT0_INTR.FAULT` bit. If an interrupt is enabled, firmware should clear this bit after servicing the interrupt.

8.2.4 Low-Power mode operation

The fault report structure functionality is available only in CPU active and CPU sleep power modes. The interfaces between the fault sources and the fault report structures are reset during the CPU Deep Sleep power

mode. Since the fault report structure is an active functionality, pending faults (stored in the FAULT_STRUCT0_PENDING registers) are not retained when transitioning to the CPU Deep Sleep power mode.

The fault structure's registers can be categorized based on their reset domain and retention capabilities, as follows:

- **Active reset domain**:
  - Includes the FAULT_STRUCT0_PENDING, FAULT_STRUCT0_INTR, FAULT_STRUCT0_INTR_SET, and FAULT_STRUCT0_INTR_MASKED registers
  - These registers are not retained in CPU Deep Sleep power mode
- **Deep Sleep reset domain**:
  - Includes the FAULT_STRUCT0_CTL, FAULT_STRUCT0_MASK, and FAULT_STRUCT0_INTR_MASK registers
  - These registers are retained during CPU Deep Sleep power mode
  - Any system reset will reset these registers to their default state
- **Soft reset domain**:
  - Includes the FAULT_STRUCT0_STATUS and FAULT_STRUCT0_DATA registers
  - These registers are retained through soft resets (detectable in the SRSS_RES_CAUSE register)
  - Hard resets, such as XRES, POR, or BOD, will reset these registers

8.2.5 Using a fault structure

Follow these steps to configure and use a fault structure:

- Identify the faults from Table 21 to be monitored in the system
- For firmware-driven fault handling through interrupts:
  1. Set the FAULT_STRUCT0_INTR_MASK.FAULT bit to enable the fault interrupt
  2. Clear any pending interrupt by setting the FAULT_STRUCT0_INTR.FAULT
  3. Enable the fault interrupt on the CPU by configuring the appropriate ISER register. See the interrupts section for details
- For fault handling using the DW:
  1. Set the FAULT_STRUCT0_CTL.TR_EN bit to enable the DMA trigger
  2. Route the tr_fault[0] signal to the trigger input of the DMA controller
  3. Configure and enable the DMA controller to transfer the FAULT_STRUCT0_STATUS and FAULT_STRUCT0_DATA registers to memory. After the transfer is complete, write back 0 to the FAULT_STRUCT0_STATUS register to clear its status
- For fault handling outside the device:
  1. Set the FAULT_STRUCT0_CTL.OUT_EN bit to enable the chip output signal
  2. Route the fault_out[0] signal to a pin using the HSIOM
  3. Use the signal externally to process the fault. This may include generating an external reset, initiating a power cycle, or logging fault information
- If a soft reset is required for any detected faults:
  1. Set the FAULT_STRUCT0_CTL.RESET_REQ_EN bit to enable reset-on-fault capability
  2. Clear the FAULT_STRUCT0_STATUS.VALID register to remove any captured fault
- To define specific faults for capture by the fault structure: Set the appropriate fault index bits in the FAULT_STRUCT0_MASK registers to enable those faults for monitoring

8.2.6 CPU exceptions versus fault monitoring

Some faults captured in Table 21 also result in bus errors or CPU exceptions, such as Bus, Usage, Memory, or Hard faults. These faults can be communicated in two ways:

- **As a bus error**:
  - The fault is communicated to the master performing the faulting bus transfer
  - This causes exceptions in the CPU, such as Bus, Usage, Memory, or Hard faults
- **As a fault in a fault report structure**:
  - The fault is recorded in the fault report structure and can be communicated as a fault interrupt to any processor in the system
  - This enables fault handling on a processor other than the master of the faulting bus transfer
  - This method is particularly useful for faults that render the master of the faulting transfer unresponsive or unreliable

8.3 Fault sources

The fault sources can vary between device families. Table 21 provides the list of fault sources available in PSC3M/P6 MCU.

Table 21 Fault sources

Fault index	Source	Descriptions	Fault encodings
0	peri.peri_ms0_ppc_vio	Fault interface in the master interface. It is used to report PPC violations (Secure/Non-secure access mismatch, Privilege/User access mismatch, and PC access mismatch) and bus error.	PERI Master0 interface PPC violation
1	peri.peri_ms1_ppc_vio	Fault interface in the master1 interface. It is used to report PPC violations (Secure/Non-secure access mismatch, Privilege/User access mismatch, and PC access mismatch) and bus error.	PERI Master1 interface PPC violation
2	peri.peri_ppc_pc_mask_vio	Fault interface for PPC MMIO access violation. This fault interface is used to report PPC violation when a locked PC tries to write to the PC_MASK register.	PERI peripheral PPC PC mask violation
3	peri.peri_gp1_timeout_vio	Fault interface for AHB timeout in peripheral Group1	PERI peripheral Group1 AHB timeout
4	peri.peri_gp2_timeout_vio	Fault interface for AHB timeout in peripheral Group2	PERI peripheral Group2 AHB timeout

(table continues...)

Table 21 (continued) Fault sources

Fault index	Source	Descriptions	Fault encodings
5	peri.peri_gp3_timeout_vio	Fault interface for AHB timeout in peripheral Group3	PERI peripheral Group3 AHB timeout
6	peri.peri_gp4_timeout_vio	Fault interface for AHB timeout in peripheral Group4	PERI peripheral Group4 AHB timeout
7	peri.peri_gp5_timeout_vio	Fault interface for AHB timeout in peripheral Group5	PERI peripheral Group5 AHB timeout
8	peri.peri_gp0_ahb_vio	Fault interface for AHB ERROR in peripheral Group0	PERI peripheral Group0 AHB violation
9	peri.peri_gp1_ahb_vio	Fault interface for AHB ERROR in peripheral Group1	PERI peripheral Group1 AHB violation
10	peri.peri_gp2_ahb_vio	Fault interface for AHB ERROR in peripheral Group2	PERI Peripheral Group2 AHB violation
11	peri.peri_gp3_ahb_vio	Fault interface for AHB ERROR in peripheral Group3	PERI Peripheral Group3 AHB violation
12	peri.peri_gp4_ahb_vio	Fault interface for AHB ERROR in peripheral Group4	PERI Peripheral Group4 AHB violation
13	peri.peri_gp5_ahb_vio	Fault interface for AHB ERROR in peripheral Group5	PERI Peripheral Group5 AHB violation
14	cpuss.ramc0_mpc_fault_mmio	RAMC-0 MPC fault	RAMC-0 MPC violation
15	cpuss.ramc1_mpc_fault_mmio	RAMC-1 MPC fault	RAMC-1 MPC violation
16	cpuss.exp_ms_ahb_error_mmio	Fault generated due to an AHB error on the EXP bridge	AHB bridge error fault
17	cpuss.promc_mpc_fault_mmio	PROMC MPC fault	PROMC MPC violation
18	cpuss.ramc0_c_ecc_mmio	RAMC-0 correctable ECC fault	RAMC-0_C ECC mmio
19	cpuss.ramc0_nc_ecc_mmio	RAMC-0 non-correctable ECC fault	RAMC-0_NC ECC mmio
20	cpuss.ramc1_c_ecc_mmio	RAMC-1 correctable ECC fault	RAMC-1_C ECC mmio
21	cpuss.ramc1_nc_ecc_mmio	RAMC-1 non-correctable ECC fault	RAMC-1_NC ECC mmio
22	cpuss.flashc_mpc_fault	FLASHC MPC fault	FLASHC MPC fault
23	cpuss.flashc_bus_access_fault	FLASHC Bus Error fault	FLASHC bus access fault

(table continues...)

Table 21 (continued) Fault sources

Fault index	Source	Descriptions	Fault encodings
24	cpuss.flashc_main_c_ecc_mmio	FLASHC ECC correctable fault	FLASHC main_c ECC mmio fault
25	cpuss.flashc_main_nc_ecc_mmio	FLASHC ECC non-correctable fault	FLASHC main_nc ECC mmio fault
26	cpuss.flashc_fm_ctl_fault	FLASHC FM CTRL fault	FLASHC FM CTL fault
27	cpuss.flashc_fm_r_q_fault	FLASHC FM r_q fault	FLASHC FM r_q fault
28	cpuss.flashc_flash_ctl_fault	FLASHC FLASH CTL fault	FLASHC flash CTL fault
29	cpuss.flashc_addr_hole_fault	FLASHC ADDR HOLE fault	FLASHC addr hole fault
30	cpuss.flashc_lock_fault	FLASHC Lock fault	FLASHC lock fault
31	srss.fault_csv	Clock supervision fault interface	SRSS clock supervisor fault
32	srss.fault_ssv	Supply supervision fault interface	SRSS supply supervisor fault
33	svgs.fault	Supply voltage glitch sensor fault	Supply voltage glitch sensor fault
34	mxaes.fault	Advanced encryption standard fault	Advanced encryption standard fault

8.4 Fault encodings

This section provides the complete encodings of the data fields for each fault.

For data fields with an "H" prefix, such as HMASTER, HWRITE, and HPROT, see the Arm® AHB protocol specification for more details.

8.4.1 PERI Master0 interface PPC violation

This fault interface is used to report PPC violations and bus errors.

The data field 'type of violation' logs only the highest-priority violation if multiple violations occur simultaneously.

The peripheral region index value is not valid when the 'type of violation' indicates an AHB error.

Table 22 PERI Master Interface PPU violation data fields

DATA0	DATA1
[31:0]: Violating address	[9:0]: Peripheral region index [15:12]: Protection context of master interface transaction [18:16]: Type of violation • 0: No violation • 1: Secure/Non-secure access mismatch (2nd priority) • 2: Privileged/User access mismatch (3rd priority) • 3: PC access mismatch (1st priority) • 4: Reserved • 5: AHB Error (4th priority) • 6, 7: Reserved [31:24]: HMASTER

8.4.2 PERI Master1 interface PPC violation

This fault interface is used to report PPC violations and bus errors.

The data field 'type of violation' logs only the highest-priority violation if multiple violations occur simultaneously.

The peripheral region index value is not valid when the 'type of violation' indicates an AHB error.

Table 23 PERI Master Interface PPU violation data fields

DATA0	DATA1
[31:0]: Violating address	[9:0]: Peripheral region index [15:12]: Protection context of master interface transaction [18:16]: Type of violation • 0: No violation • 1: Secure/Non-secure access mismatch (2nd priority) • 2: Privileged/User access mismatch (3rd priority) • 3: PC access mismatch (1st priority) • 4: Reserved • 5: AHB Error (4th priority) • 6, 7: Reserved [31:24]: HMASTER

8.4.3 PERI peripheral PPC PC mask violation

This fault interface is used to report a PPC violation that occurs when a locked PC attempts to write to a PC_MASK

register.

Table 24 Peripheral PPC PC mask violation data fields

DATA0	DATA1
[31:0]: Violating address	[9:0]: Peripheral region index [11:10]: Reserved [15:12]: Protection context of master interface transaction [18:16]: Type of violation • 4: write to PC_MASK registers from a locked PC • 0, 1, 2, 3, 5, 6, 7: Reserved [23:19]: Reserved [31:24]: HMASTER

8.4.4 PERI peripheral Group1 AHB timeout

This fault interface reports timeout errors in peripheral Group1. The peripheral group has its own fault interface.

Table 25 Peripheral Group AHB timeout data fields

DATA0
[4:0]: Peripheral group slave number [31:5]: Reserved

8.4.5 PERI peripheral Group2 AHB timeout

This fault interface reports timeout errors in peripheral Group2. The peripheral group has its own fault interface.

Table 26 Peripheral Group AHB timeout data fields

DATA0
[4:0]: Peripheral group slave number [31:5]: Reserved

8.4.6 PERI peripheral Group3 AHB timeout

This fault interface reports timeout errors in peripheral Group3. The peripheral group has its own fault interface.

Table 27 Peripheral Group AHB timeout data fields

DATA0
[4:0]: Peripheral group slave number [31:5]: Reserved

8.4.7 PERI peripheral Group4 AHB timeout

This fault interface reports timeout errors in peripheral Group4. The peripheral group has its own fault interface.

Table 28 Peripheral Group AHB timeout data fields

DATA0
[4:0]: Peripheral group slave number [31:5]: Reserved

8.4.8 PERI peripheral Group5 AHB timeout

This fault interface reports timeout errors in peripheral Group5. The peripheral group has its own fault interface.

Table 29 Peripheral Group AHB timeout data fields

DATA0
[4:0]: Peripheral group slave number [31:5]: Reserved

8.4.9 PERI peripheral Group0 AHB violation

This fault interface reports bus errors on the peripheral group decoder (side facing the bridge).
The peripheral group has its own fault interface.

Table 30 Peripheral Group AHB violation data fields

DATA0	DATA1
[31:0]: Violating address	[0]: Privilege mode • 0: User mode • 1: Privilege mode [1]: HNONSEC [2]: HWRITE [3]: Reserved [7:4]: Protection context of master interface transaction [23:8]: HMASTER [31:30]: Error type • 0: AHB error • 1, 2, 3: Reserved

8.4.10 PERI peripheral Group1 AHB violation

This fault interface reports bus errors on the peripheral group decoder (side facing the bridge).
The peripheral group has its own fault interface.

Table 31 Peripheral Group AHB violation data fields

DATA0	DATA1
[31:0]: Violating address	[0]: Privilege mode • 0: User mode • 1: Privilege mode [1]: HNONSEC [2]: HWRITE [3]: Reserved [7:4]: Protection context of master interface transaction [23:8]: HMASTER [31:30]: Error type • 0: AHB error • 1, 2, 3: Reserved

8.4.11 PERI Peripheral Group2 AHB violation

This fault interface reports bus errors on the peripheral group decoder (side facing the bridge).
The peripheral group has its own fault interface.

Table 32 Peripheral Group AHB violation data fields

DATA0	DATA1
[31:0]: Violating address	[0]: Privilege mode • 0: User mode • 1: Privilege mode [1]: HNONSEC [2]: HWRITE [3]: Reserved [7:4]: Protection context of master interface transaction [23:8]: HMASTER [31:30]: Error type • 0: AHB error • 1, 2, 3: Reserved

8.4.12 PERI Peripheral Group3 AHB violation

This fault interface reports bus errors on the peripheral group decoder (side facing the bridge).
The peripheral group has its own fault interface.

Table 33 Peripheral Group AHB violation data fields

DATA0	DATA1
[31:0]: Violating address	[0]: Privilege mode • 0: User mode • 1: Privilege mode [1]: HNONSEC [2]: HWRITE [3]: Reserved [7:4]: Protection context of master interface transaction [23:8]: HMASTER [31:30]: Error type • 0: AHB error • 1, 2, 3: Reserved

8.4.13 PERI Peripheral Group4 AHB violation

This fault interface reports bus errors on the peripheral group decoder (side facing the bridge).
The peripheral group has its own fault interface.

Table 34 Peripheral Group AHB violation data fields

DATA0	DATA1
[31:0]: Violating address	[0]: Privilege mode • 0: User mode • 1: Privilege mode [1]: HNONSEC [2]: HWRITE [3]: Reserved [7:4]: Protection context of master interface transaction [23:8]: HMASTER [31:30]: Error type • 0: AHB error • 1, 2, 3: Reserved

8.4.14 PERI Peripheral Group5 AHB violation

This fault interface reports bus errors on the peripheral group decoder (side facing the bridge).
The peripheral group has its own fault interface.

Table 35 Peripheral Group AHB violation data fields

DATA0	DATA1
[31:0]: Violating address	[0]: Privilege mode • 0: User mode • 1: Privilege mode [1]: HNONSEC [2]: HWRITE [3]: Reserved [7:4]: Protection context of master interface transaction [23:8]: HMASTER [31:30]: Error type • 0: AHB error • 1, 2, 3: Reserved

8.4.15 RAMC-0 MPC violation

This fault is triggered when an access violation occurs, and MPC_CFG.RESPONSE is set. Access violations include:

- Reads from non-readable memory
- Writes to non-writable memory
- Non-secure access to secure memory

If MPC_CFG.RESPONSE is not set, no fault is generated when an access violation occurs. Instead:

- Read operations return zero
- Write operations are ignored

Table 36 MPC violation data fields

DATA0	DATA1
[31:0]: Violating address	[5:0]: HMASTER [15:6]: Reserved [16]: HNONSEC [17]: Secure or non-secure configuration of a violated block • 0: Secure • 1: non-secure [18]: HWRITE [23:19]: Reserved [27:24]: Protection context of master interface transaction [29:28]: Unused [30]: Security access violation [31]: PC access violation

8.4.16 RAMC-1 MPC violation

This fault is triggered when an access violation occurs, and MPC_CFG.RESPONSE is set. Access violations include:

- Reads from non-readable memory
- Writes to non-writable memory
- Non-secure access to secure memory

If MPC_CFG.RESPONSE is not set, no fault is generated when an access violation occurs. Instead:

- Read operations return zero
- Write operations are ignored

Table 37 MPC violation data fields

DATA0	DATA1
[31:0]: Violating address	[5:0]: HMASTER [15:6]: Reserved [16]: HNONSEC [17]: Secure or non-secure configuration of a violated block • 0: Secure • 1: non-secure [18]: HWRITE [23:19]: Reserved [27:24]: Protection context of master interface transaction [29:28]: Unused [30]: Security access violation [31]: PC access violation

8.4.17 AHB bridge error fault

This fault interface reports an AHB error on an AHB bridge.

Table 38 AHB bridge error data fields

DATA0	DATA1
[31:0]: Violation address	[0]: HPROT [1]: HNONSEC [2]: HWRITE [3]: Reserved [7:4]: Protection context of master interface transaction [15:8]: HMASTER [31:16]: Reserved

8.4.18 PROMC MPC violation

Table 39 PROMC MPC violation data fields

DATA0	DATA1
[31:0]: Violating address	[5:0]: MASTER [15:6]: Reserved [16]: NONSEC [17]: Reserved [18]: WRITE [23:19]: Reserved [27:24]: Protection context of master interface transaction [29:28]: Unused [30]: Security access violation [31]: PC access violation

8.4.19 RAMC-0_C ECC mmio

This fault interface reports correctable ECC error detection faults in RAMC-0. The fault is masked if it occurs alongside an MPC access fault.

Table 40 RAMC0_C ECC mmio data fields

DATA0	DATA1
[31:0]: Violating address	[6:0]: SYNDROME

8.4.20 RAMC-0_NC ECC mmio

This fault interface reports non-correctable ECC error detection faults in RAMC. The fault is masked if it occurs alongside an MPC access fault.

Table 41 RAMC0_NC ECC mmio data fields

DATA0	DATA1
[31:0]: Violating address	[6:0]: SYNDROME

8.4.21 RAMC-1_C ECC mmio

This fault interface reports correctable ECC error detection faults in RAMC-1. The fault is masked if it occurs alongside an MPC access fault.

Table 42 RAMC1_C ECC mmio data fields

DATA0	DATA1
[31:0]: Violating address	[6:0]: SYNDROME

8.4.22 RAMC-1_NC ECC mmio

This fault interface reports non-correctable ECC error detection faults in RAMC. The fault is masked if it occurs alongside an MPC access fault.

Table 43 RAMC1_NC ECC mmio data fields

DATA0	DATA1
[31:0]: Violating address	[6:0]: SYNDROME

8.4.23 FLASHC MPC fault

This is the FLASHC MPC violation fault interface.

Table 44 FLASHC MPC violation data fields

DATA0	DATA1
[31:0]: Violating address	[15:0] MASTER [16] Non-secure of transaction [17] Secure or non-secure configuration of a violated block [18] WR • 1: Write • 0: Read [23:19] Reserved [27:24] Transaction PC of access violation [29:28] Reserved [30] Security access violation [31] PC access violation

8.4.24 FLASHC bus access fault

This fault interface reports the following faults:

- C-bus access to FM registers FM_ADDR or FM_REFRESH_ADDR while in Full R-bus mode
- R-bus access while in Full C-bus mode
- AHB5 write transaction on the 128-bit AHB5 host interface while in Full C-bus/RWW mode with ENFORCE_PC_LOCK = 0
- Write to the FM_ADDR register in Full R-bus mode through a 128-bit AHB5 bus

Table 45 FLASHC bus access data fields

DATA0	DATA1
[31:0]: Violating address	[0] Full R-bus mode [1] Full C-bus mode [2] RWW mode [3] ENFORCE_PC_LOCK [4] Indicates transaction is from which AHB5 interface [6:5] Type of fault: • 00: C-bus access/write from the 128-bit AHB5 host interface while in full R-bus mode • 01: R-bus access while in full C-bus mode • 10: AHB5 write transaction on the 128-bit AHB5 host interface while in full C-bus/RWW mode, and ENFORCE_PC_LOCK = 0 • 11: Reserved [31:7] Reserved

8.4.25 FLASHC main_c ECC mmio fault

This fault interface reports a fault when parameter the ECC_PRESENT parameter is high and the ECC recoverable error count exceeds the threshold defined by ECC_LOGIR.ECCTHRESHOLD. This fault is not caused by ECC error injection.

Table 46 flashc_main_c_ecc_mmio data fields

DATA0	DATA1
[31:0] Violating address	[8:0] Syndrome [31:9] Reserved

8.4.26 FLASHC main_nc ECC mmio fault

This fault interface reports a fault when any non-recoverable ECC error occurs. This fault is not caused by ECC error injection.

Table 47 flashc_main_nc_ecc_mmio data fields

DATA0	DATA1
[31:0] Violating address	[8:0] Syndrome [31:9] Reserved

8.4.27 FLASHC FM CTL fault

This fault interface reports the following types of faults:

- Writing to the FLASH_MACRO_CTL register with invalid mode (IF_SEL = 1 and WR_EN = 1)
- Updating both FLASH_MACRO_CTL.IF_SEL and FLASH_MACRO_CTL.WR_EN simultaneously
- Writing a non-zero value to the TEST_CTL register when FLASH_CTL.ENFORCE_PC_LOCK is high

Table 48 FLASHC FM CTL data fields

DATA0	DATA1
[31:0] Violating address	[0] Value to be written to FLASH_MACRO_CTL.IF_SEL [1] Value to be written to FLASH_MACRO_CTL.WR_EN [2] ENFORCE_PC_LOCK [4:3] Type of fault • 00: Write to FLASH_MACRO_CTL register with FLASH_MACRO_CTL.IF_SEL and FLASH_MACRO_CTL.WR_EN 1 • 01: Updating FLASH_MACRO_CTL.IF_SEL and FLASH_MACRO_CTL.WR_EN at the same time • 10: Writing non-zero values in the TEST_CTL register when FLASH_CTL.ENFORCE_PC_LOCK is high • 11: Reserved [31:5] Reserved

8.4.28 FLASHC FM r_q fault

This fault interface reports faults caused by R-bus address conflicts (r_q_error from FM). The fault occurs when the read address on the R-bus conflicts with the address on the C-bus.

Table 49 FLASHC FM r_q data fields

DATA0	DATA1
[4:0] Word address [log2(FLASHC_ROWS_PER_SECTOR-1)+4:5] Row address [FM_ADDR_WIDTH-4:FM_ADDR_WIDTH-4-log2(FLASHC_SECTOR_M-1)+1] Sector address [FM_ADDR_WIDTH-3] axa (01 is for SFLASH) [FM_ADDR_WIDTH-2] bxa (01 is for Refresh row) [FM_ADDR_WIDTH-1] cxa (01 is for column 33) [32-FM_ADDR_WIDTH] Reserved	[7:0] Reserved [MASTER_WIDTH+8-1:8] Master ID [31:MASTER_WIDTH+8] Reserved

8.4.29 FLASHC flash CTL fault

This fault interface is designed to report the following faults related to invalid configurations of FLASH_MACRO_CTL.FM_MODE[3:0] based on the status of specific control parameters:

- Setting FLASH_MACRO_CTL.FM_MODE[3:0] to modes 9, 12, and 13 when FLASH_CTL.BLOCK_SECTOR_OPERATIONS is high
- Setting FLASH_MACRO_CTL.FM_MODE[3:0] to modes 6 and 8 when FLASH_CTL.BLOCK_SUBSECTOR_OPERATIONS is high
- Setting FLASH_MACRO_CTL.FM_MODE[3:0] to modes 10 and 15 when FLASH_CTL.BLOCK_BULK_OPERATIONS is high
- Setting FLASH_MACRO_CTL.FM_MODE[3:0] to mode 3 when FLASH_CTL.ENFORCE_PC_LOCK is high

Table 50 FLASHC flash CTL data fields

DATA0
[3:0] FM_MODE[3:0]
[4] ENFORCE_PC_LOCK
[5] BLOCK_SECTOR_OPERATIONS
[6] BLOCK_SUBSECTOR_OPERATIONS
[7] BLOCK_BULK_OPERATIONS
[9:8] Type of fault • 00: FLASH_CTL.BLOCK_SECTOR_OPERATIONS high forbids software setting FLASH_MACRO_CTL.FM_MODE[3:0] to modes 9, 12, and 13 • 01: FLASH_CTL.BLOCK_SUBSECTOR_OPERATIONS high forbids software setting FLASH_MACRO_CTL.FM_MODE[3:0] to modes 6 and 8 • 10: FLASH_CTL.BLOCK_BULK_OPERATIONS high forbids software setting FLASH_MACRO_CTL.FM_MODE[3:0] to modes 10 and 15 • 11: Setting FLASH_MACRO_CTL.FM_MODE[3:0] to mode 3 when FLASH_CTL.ENFORCE_PC_LOCK is high
[31:10] Reserved

8.4.30 FLASHC addr hole fault

This fault interface reports faults caused by transactions to an invalid address via the 128-bit AHB5 interface.

Such transactions may occur due to:

- Wounding
- Targeting FM address holes

Table 51 FLASHC address hole data fields

DATA0
[31:0] Violating address

8.4.31 FLASHC lock fault

This fault interface detects faults related to attempts to write or release FLASH_LOCK.LOCKED under the following conditions:

- By a non-acquired PC
- By the acquired PC with a non-acquired HMASTER_ID

Table 52 FLASHC lock data fields

DATA0
[0] FLASH_LOCK.LOCKED
[4:1] Transaction PC
[8:5] Acquired PC
[MASTER_WIDTH+8:9] Master ID of the violating transfer
[2*MASTER_WIDTH+8:MASTER_WIDTH+9] Acquired Master ID
[31:2*MASTER_WIDTH+9] Reserved

8.4.32 SRSS clock supervisor fault

This fault is generated when the SRSS detects an error in one of the clocks.

Table 53 SRSS clock supervisor (CSV) data fields

DATA0
[6:0]: CSV_HF
[24]: CSV_REF
[25]: CSV_LF
[26]: CSV_ILO

8.4.33 SRSS supply supervisor fault

This fault is generated when the SRSS detects an error in one of the power supplies.

Table 54 SRSS supply supervision (SSV) data field

DATA0
[0]: BOD_VDDA
[16]: HVLVD1
[17]: HVLVD2

8.4.34 Supply voltage glitch sensor fault

This fault interface is used to report SVGS glitch-detection faults.

Table 55 Supply voltage glitch sensor (SVGS) fault data fields

DATA0
[0] `UVC_GLITCH`: UVC Glitch detected
[1] `OVC_GLITCH`: OVC Glitch detected
[2] `UVFC_GLITCH`: UVFC Glitch detected
[3] `NVC_GLITCH`: NVC Glitch detected
[4] `HVC_GLITCH`: HVC Glitch detected

8.4.35 Advanced encryption standard fault

This fault interface is used to report faults in the advanced encryption standard (AES) module.

- When `alarm = 1` in the fault structure, it indicates that the MXAES module has been attacked, and the module has detected this alarm
- The alarm is reported to the SOC using the fault structure field `data[0]`
- Once in this state, the MXAES module becomes inaccessible until a reset (`rst_act_n = 0`) is executed by the SOC
- Internally, the module enters an alarm state and clears its entire content
- During the alarm state, any AHB access to the MXAES module results in a bus-fault response

Table 56	Advanced encryption standard (AES) data fields
DATA0	
[0]: alarm	

8.5 Registers summary

Table 57 fault registers summary

Registers	Function
FAULT_STRUCT0_CTL	fault control registers
FAULT_STRUCT0_STATUS	fault status registers
FAULT_STRUCT0_DATA	fault data registers capture fault information
FAULT_STRUCT0_PENDING0	fault pending0 registers specify pending (not captured) fault sources
FAULT_STRUCT0_PENDING1	fault pending1 registers specify pending (not captured) fault sources
FAULT_STRUCT0_PENDING2	fault pending2 registers specify pending (not captured) fault sources
FAULT_STRUCT0_MASK0	fault mask0 registers specify "enables" for fault sources
FAULT_STRUCT0_MASK1	fault mask1 registers specify "enables" for fault sources
FAULT_STRUCT0_MASK2	fault mask2 registers specify "enables" for fault sources
FAULT_STRUCT0_INT	fault interrupt cause registers
FAULT_STRUCT0_INT_SET	fault interrupt set registers
FAULT_STRUCT0_INT_MASK	fault interrupt mask registers
FAULT_STRUCT0_INT_MASKED	fault interrupt masked registers

9 Direct memory access (DMA - DW)

Direct Memory Access (DMA) transfers data to and from memory, peripherals, and registers using the 32-bit Advanced High-Performance Bus (AHB). These transfers occur independently from the CPU. The DMA can be configured to perform multiple independent data transfers. All data transfers are managed by a channel. There are 16 channels in the DMA. Refer to the MCU datasheet for more device details.

The CPU includes a DMA controller that can be used to transfer data to and from memory, peripherals, and registers. This allows for autonomous, deterministic control of peripherals such as the PWM or enables large data transfers from a peripheral such as the ADC to memory.

The DMA controllers are bus masters in the M33 domain:

- Two DMAs with 16 channels. It has a single transfer engine for all channels that arbitrates for bus master access. This DMA uses the 32-bit AHB bus that shares a clock with the M33. The DMA is optimal for small data transfers; transactional DMA is typically used to transfer data from memory to peripheral or peripheral to memory

***Note**: The DMA is referenced as the Data Wire (DW) in both register names and for triggers in the Trigger multiplexer (Trigger MUX) chapter.*

9.1 Features

Each DMA controller has the following features:

- Supports 16 data channels
- Four levels of priority for each channel
- Descriptors are defined in memory and referenced to the respective channels
- Supports single, 1D, or 2D transfer modes for a descriptor
- Configurable source and destination address increments
- Supports 8-bit, 16-bit, and 32-bit data widths at both source and destination
- Configurable input trigger behavior for each channel
- Configurable interrupt generation in each channel
- Configurable output trigger generation for each channel
- Descriptors can be chained to other descriptors for more complex transfers
- Configurable cyclical redundancy check (CRC)

9.2 Architecture

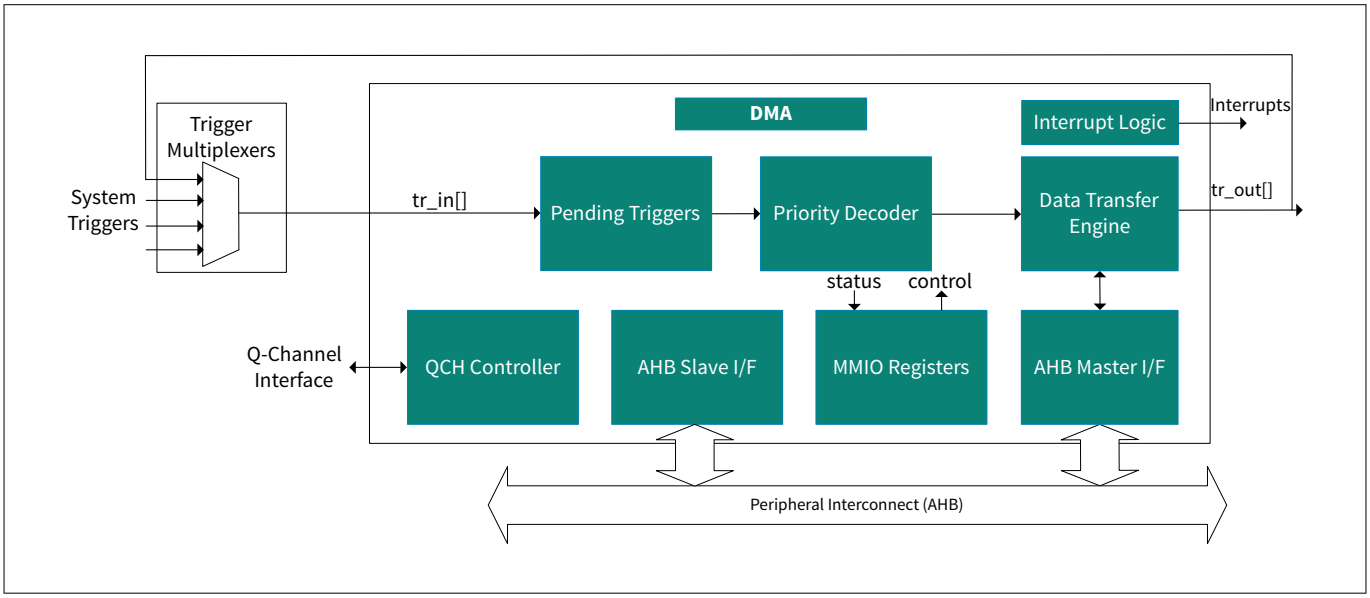


Figure 19 DMA architecture

A data transfer is initiated by an input trigger. This trigger may originate from the source of the transfer, the destination of the transfer, the CPU SW, or from another SoC component. Triggers provide Active/Sleep functionality and are not available in CPU/System DeepSleep and Hibernate power modes.

Once a transfer is triggered, the DMA completes the transfer as configured by the channel descriptor. Channels can be configured to perform a 1D or 2D memory copy. At the end of a transfer, the descriptor can initiate a new transfer by pointing to a new descriptor or generate a completion interrupt.

The DMA is broken up into the following blocks:

- Pending Triggers: Keeps track of activated triggers by locally storing them in pending bits. This is essential because multiple channel triggers may be activated simultaneously, whereas only one channel can be served by the data transfer engine at a time
- Priority Decoder: Determines the highest priority channel with an active trigger. Within a priority group, triggers are decoded on a round-robin basis
- Data Transfer Engine: Transfers data from a source location to a destination location. When idle, the data transfer engine is ready to accept the highest-priority activated channel. It is also responsible for reading the channel descriptor from memory
- AHB Slave I/F: AHB bus slave, which allows the main CPU to access controller MMIO control/status registers
- AHB Master I/F: AHB bus master, which allows the controller to initiate AHB data transfers to the source and destination locations as well as to read the descriptor from memory
- Interrupt Logic: Includes interrupt status for each of the channels
- MMIO Registers: Stores each channel's descriptor
- QCH (Q-Channel) Controller: Enables the Power Policy Unit (PPU) to check the current state of the DMA before changing the power state of the device

9.2.1 Standard flow

A typical DMA data transfer follows these steps:

1. The main CPU programs the descriptor chain (in memory) and associates it with a specific channel. Further, it programs the channel registers to set the desired attributes for the channel. It also programs the MMIO register, which selects a specific system trigger for the channel

2. The channel's system trigger is activated
3. Priority decoding determines the highest-priority pending channel
4. The data transfer engine accepts the activated channel and uses the channel identifier to load the channel's descriptor structure from memory using the master I/F. The descriptor structure specifies the channel's data transfer(s)
5. The data transfer engine uses the master I/F to load data from the source location
6. The data transfer engine uses the master I/F to store data at the destination location. The amount of data transferred depends on the TR_IN_TYPE
7. The data transfer engine updates the active descriptor registers and channel structure registers to reflect the data transfers. There is no update to the descriptors in memory
8. Output trigger generation and interrupt generation are determined by TR_OUT_TYPE and INTR_TYPE, respectively. DW can generate an error interrupt if it encounters an error. A channel gets disabled if there is an error

9.3 Channels

The DMA controller supports multiple independent data transfers that are managed by a channel. Each channel connects to a specific system trigger through a trigger multiplexer that is outside the DMA controller.

9.3.1 Channel priority

Only one DMA channel may be moving data at a time. Each channel has a corresponding priority to determine which channel gets bus access if multiple channels are triggered.

Each channel is assigned a priority (CHi_CTL.PRIO) between 0 and 3, with 0 being the highest priority and 3 being the lowest priority. Channels with the same priority constitute a priority group. Priority decoding determines the highest-priority pending channel. This channel is determined as follows:

- The highest priority group with pending channels is identified first
- Within this priority group, round-robin arbitration is applied. Round-robin arbitration (within a priority group) gives the highest priority to the lower channel indices (within the priority group)

Each channel can be marked as pre-emptable by setting CHi_CTL.PREEMPTABLE to '1'. When a higher-priority channel goes into a pending state, it will interrupt a channel marked pre-emptable in between a 1D transfer. After the completion of a higher-priority channel transfer, the channel marked pre-emptable is re-activated by the controller state machine.

9.3.2 Channel security attributes

Each channel has three access control attributes that are used by the memory protection controller (MPC) and peripheral protection controller (PPC) for access control. The following three fields are inherited from the access attributes of the MMIO write transaction that configures the CHi_CTL register.

- "Privileged Mode" (CHi_CTL.P) attribute can be inherited as privileged or user
- "Non-secure" (CHi_CTL.NS) attribute can be inherited as secure or non-secure
- "PC" (CHi_CTL.PC) inherits the protection context from the write transaction

9.3.3 Channel state

At any given time, there is at most one channel that is actively performing a data transfer. This channel is called the active channel. A channel can be in one of the four channel states described in the following table.

Table 58 Channel states

Channel state	Registers	Description
Disabled	CHi_CTL.ENABLED = 0	The channel has been disabled by setting DW_CHi_CTL.ENABLED to 0. The channel trigger is ignored in this state. If an active channel is disabled by software, there should be no assumptions made about the state of the channel (the current "position" of the transfer as reflected by the MMIO registers or descriptor(s)). A software channel re-enable should prepare the new descriptor(s) and reconfigure the channel.
Blocked	CHi_CTL.ENABLED = 1 CHi_STATUS.PENDING = 0	The channel is enabled and is waiting for a trigger to initiate a data transfer.
Pending	CHi_CTL.ENABLED = 1 CHi_STATUS.PENDING = 1	The channel is enabled and has received an active trigger. In this state, the channel is ready to initiate a data transfer but is waiting for it to be scheduled.
Active	CHi_CTL.ENABLED = 1 CHi_STATUS.PENDING = 1 DW_STATUS.CH_IDX = Active Channel DW_STATUS.ACTIVE = 1	The channel is enabled, has received an active trigger, and has been scheduled. It is actively performing data transfer(s). If there are multiple channels pending, the highest-priority pending channel gets scheduled.

Once a channel is in an active state, more information can be found using DW_STATUS.STATE:

- Val 0: Default/inactive state
- Val 1: Loading the channel descriptor from memory
- Val 2: Loading the data element from the source location
- Val 3: Storing the data element at the destination location
- Val 4: CRC functionality (only used when DESCR_CTL.DESCR_TYPE is set to CRC)
- Val 5: Update active control information (source and destination addresses) and wait for trigger deactivation
- Val 6: Error

9.3.4 Channel interrupt generation

Each DMA channel has an interrupt line associated with it. The DMA allows interrupt generation on the completion of a transfer as well as on error conditions. These interrupts can then wake the CPU up at the end of a transfer or signal that the CPU needs to fix a DMA error. Each DMA channel has six registers to control/interact with interrupts. For more information on transfer types, see the Descriptor memory transfer types section.

DESCR_CTL.INTR_TYPE selects what completion interrupt (CHi_STATUS.INTR_CAUSE is set to COMPLETION) is generated by the DMA channel:

- Set 0: An interrupt is generated after a single transfer

- Set 1: An interrupt is generated after a single 1D transfer. If the descriptor type is "single", the interrupt is generated after a single transfer. If the descriptor type is "1D", "CRC", or "2D", the interrupt is generated after the execution of a 1D transfer
- Set 2: An interrupt is generated after the execution of the current descriptor. Independent of the value of DESCR_NEXT_PTR.ADDR of the current descriptor
- Set 3: An interrupt is generated after the execution of the current descriptor. The value of DESCR_NEXT_PTR.ADDR of the current descriptor must be "0'

CHi_STATUS.INTR_CAUSE specifies the source of the interrupt cause when CHi_INTR.CH is set to '1'. The following interrupts can be generated in CHi_STATUS.INTR_CAUSE:

- Value 0: No interrupt detected (NO_INTR)
- Value 1: Interrupt on transfer completion configured based on the DESCR_CTL.INTR_TYPE field in the descriptor (COMPLETION)
- Value 2: Source bus error (SRC_BUS_ERROR)
- Value 3: Destination bus error (DST_BUS_ERROR)
- Value 4: Misaligned source address (SRC_MISAL)
- Value 5: Misaligned destination address (DST_MISAL)
- Value 6: Current descriptor pointer is null (CURR_PTR_NULL)
- Value 7: Active channel is in a disabled state (ACTIVE_CH_DISABLED)
- Value 8: Descriptor bus error (DESCR_BUS_ERROR)

CHi_INTR is set to '1' by hardware when an interrupt (as specified by CHi_STATUS.INTR_CAUSE) is detected. The register is cleared by writing a '1'.

CHi_INTR_SET is set to '1' to set the corresponding CHi_INTR field.

CHi_INTR_MASK is used to set the interrupt mask for the specified DMA channel.

CHi_INTR_MASKED when read, gives the bitwise and between CHi_INTR and CHi_INTR_MASK.

For error-related interrupt causes (CHi_STATUS.INTR_CAUSE 2 through 8), the hardware disables the channel (HW sets CHi_CTL.ENABLED to '0'). The interrupt should be cleared before the channel is re-enabled.

The DMA controller is a CPU Active power mode peripheral. Therefore, INTR and INTR_SET are not retained in CPU/System DeepSleep power modes. The INTR_MASK is retained.

9.4 Descriptors

The data transfer between a source and destination in a channel is configured using a collection of 32-bit words called a descriptor. A descriptor is a set of up to six 32-bit words that can be configured to support four different memory transfer types.

The descriptor pointer is specified in the DMA channel registers. CHi_CURR_PTR stores the address of the current descriptor and is updated to DESCR_NEXT_PTR after the execution of the current descriptor. The DMA controller does not modify the descriptor and treats it as read-only.

Each channel includes a CHi_IDX register, which includes the X_IDX and Y_IDX bit fields. These two fields contain the current descriptor X and Y indices as the transfer completes. These are set to zero when CHi_CURR_PTR is updated and are incremented after each X and Y loop.

9.4.1 Descriptor memory transfer types

The DMA supports four different types of memory transfers: single, 1D, 2D, and CRC (cyclic redundancy check). The memory transfer type is configured in DESCR_CTL.DESCR_TYPE but can require up to six configured 32-bit words to support the specified transfer type. The following sections describe the three memory transfer types.

9.4.1.1 Single transfer

The descriptor configuration performs a single data element. The data element size can be configured with DESCR_CTL.DATA_SIZE as 8-bit, 16-bit, or 32-bit. A single transfer requires the configuration of four 32-bit words. The following pseudocode describes the functionality for a single transfer.

Required configurations: DESCR_CTL, DESCR_SRC, DESCR_DST, and DESCR_NEXT_PTR.

```
// DST_ADDR is a pointer to an object of type defined by DESCR_DST.DST_ADDR
            // SRC_ADDR is a pointer to an object of type defined by DESCR_SRC.SRC_ADDR
            // t_DATA_SIZE is the type associated with the DESCR_CTL.DATA_SIZE
            DST_ADDR[0] = (t_DATA_SIZE) SRC_ADDR[0];
```

The following figure illustrates how to move 32 bits in memory using a single transfer.

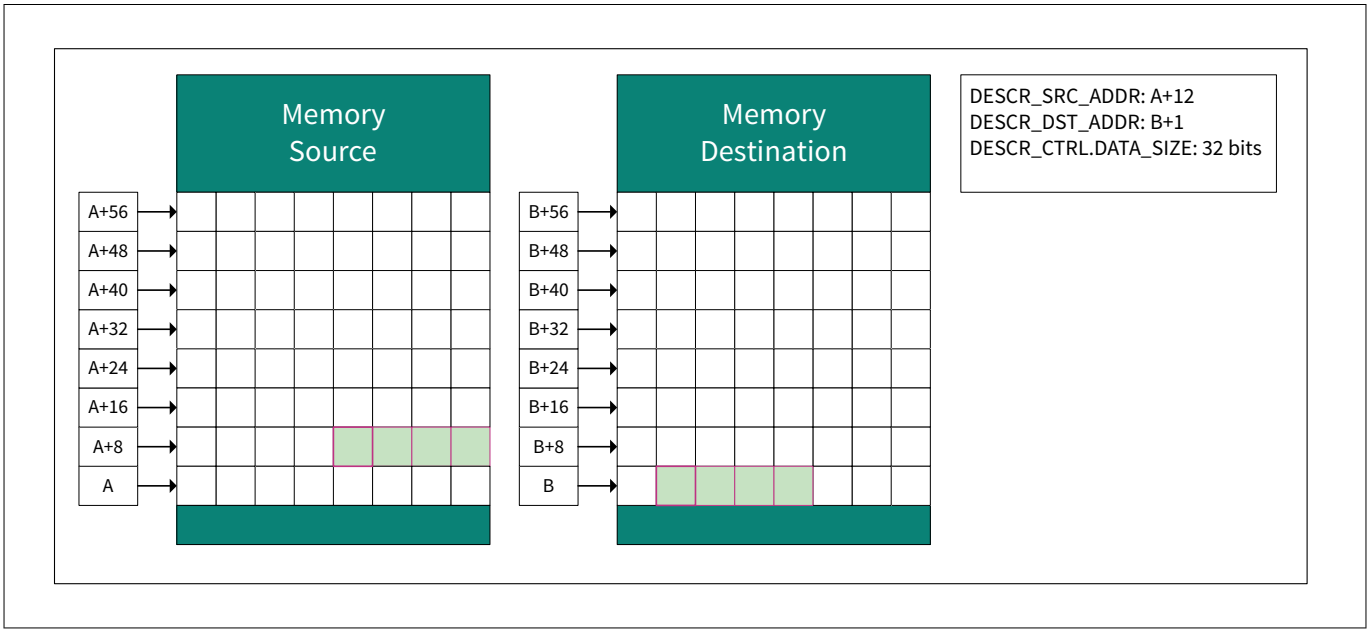


Figure 20 Single transfer

9.4.1.2 1D memory transfer

The descriptor configuration performs a one-dimensional “for loop”, where a channel copies X_COUNT+1 bytes from DESCR_SRC to DESCR_DST. A 1D memory transfer requires the configuration of five 32-bit words. The following pseudocode describes the functionality for a 1D memory transfer.

Required configurations: DESCR_CTL, DESCR_SRC, DESCR_DST, DESCR_X_CTL, and DESCR_NEXT_PTR.

```
// DST_ADDR is a pointer to an object of type defined by DESCR_DST.DST_ADDR
            // DST_X_INCR specifies the source increment between single transfer defined by
DESCR_X_CTL.DST_X_INCR
            // SRC_ADDR is a pointer to an object of type defined by DESCR_SRC.SRC_ADDR
            // SRC_X_INCR specifies the source increment between single transfer defined by
DESCR_X_CTL.SRC_X_INCR
            // t_DATA_SIZE is the type associated with the DESCR_CTL.DATA_SIZE
            // X_COUNT is the number of single transfers defined by DESCR_X_CTL.X_COUNT
            for (X_IDX = 0; X_IDX <= X_COUNT; X_ID++){
            DST_ADDR[X_IDX * DST_X_INCR] = (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR};
            }
```

The following figure illustrates how to move three 32-bit data elements in memory using a 1D memory transfer.

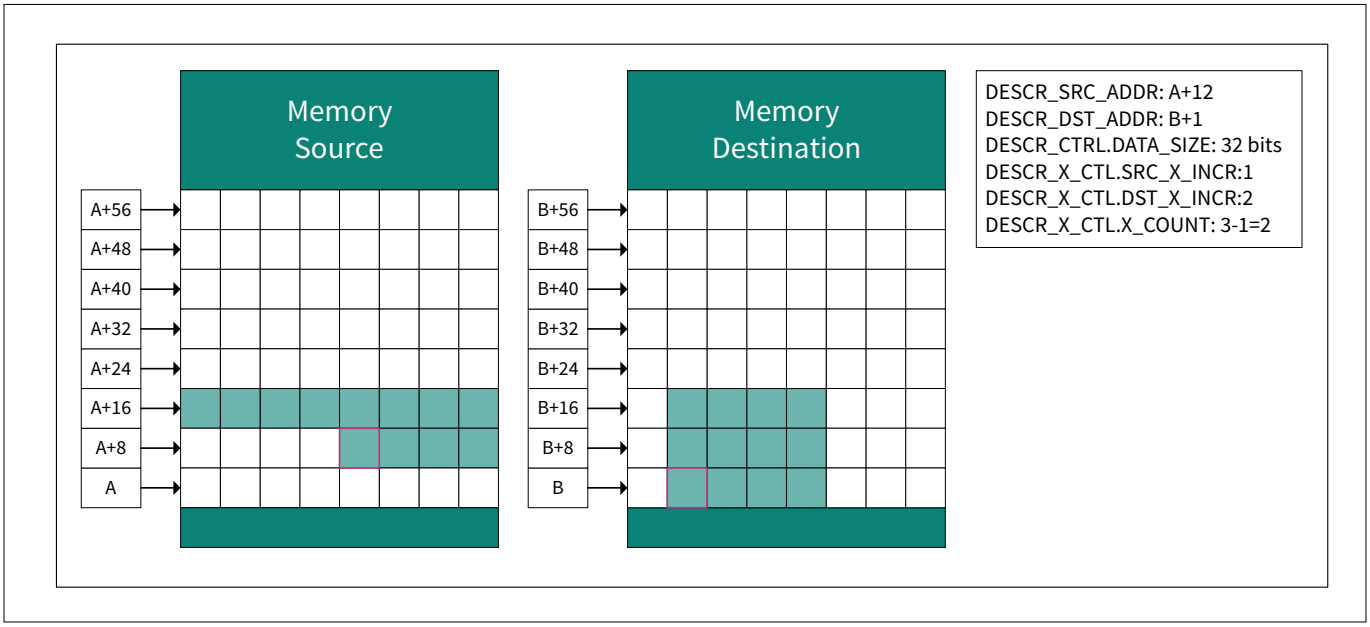


Figure 21 1D memory transfer

9.4.1.3 2D memory transfer

The descriptor configuration performs a two-dimensional “for loop”, where a channel copies (Y_COUNT+1)*(X_COUNT+1) bytes from DESCR_SRC to DESCR_DST. A 2D memory transfer requires the configuration of six 32-bit words. The following pseudocode describes the functionality for a 2D memory transfer.

Required configurations: DESCR_CTL, DESCR_SRC, DESCR_DST, DESCR_X_CTL, DESCR_Y_CTL, and DESCR_NEXT_PTR.

```
// DST_ADDR is a pointer to an object of type defined by DESCR_DST.DST_ADDR
            // SRC_ADDR is a pointer to an object of type defined by DESCR_SRC.SRC_ADDR
            // SRC_X_INCR/DST_X_INCR specifies the source and destination increment between
single transfers defined by DESCR_X_CTL
            // SRC_Y_INCR/DST_Y_INCR specifies the source and destination increment between 1D
transfers defined by DESCR_Y_CTL
            // Y_COUNT is the number of 1D transfers defined by DESCR_Y_CTL.Y_COUNT
            // t_DATA_SIZE is the type associated with the DESCR_CTL.DATA_SIZE
            // X_COUNT is the number of single transfers defined by DESCR_X_CTL.X_COUNT
            for (Y_IDX = 0; Y_IDX <= Y_COUNT; Y_IDX++) {
            for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
            DST_ADDR[X_IDX * DST_X_INCR + Y_IDX * DST_Y_INCR] = (t_DATA_SIZE) SRC_ADDR[X_IDX *
SRC_X_INCR + Y_IDX * SRC_Y_INCR];
            }
            }
```

The following figure illustrates how to move six 32-bit data elements in memory using a 2D memory transfer.

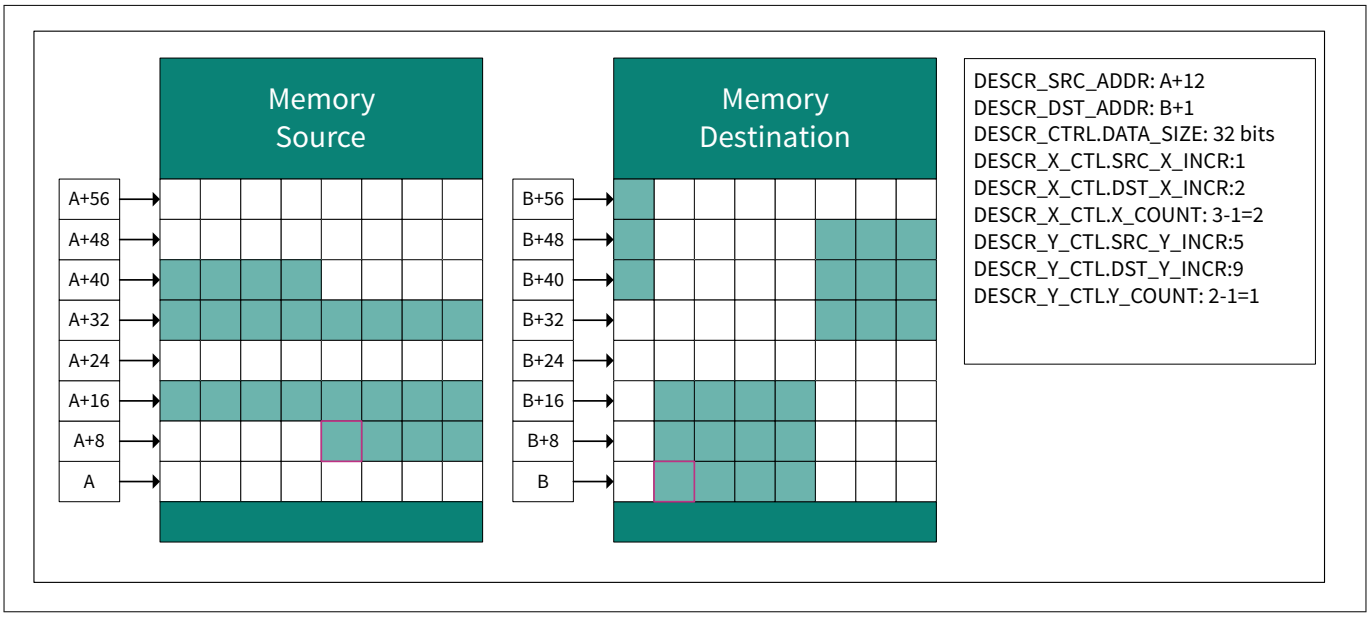


Figure 22 2D memory transfer

9.4.1.4 Cyclic Redundancy Check (CRC) transfer

The descriptor configuration performs a one-dimensional “for loop” similar to the 1D memory transfer. However, the source data is not transferred to a destination. Instead, a CRC checksum is calculated over the source data. A CRC transfer requires the configuration of five 32-bit words. The CRC configuration is provided through a set of registers that is shared by all DW channels, and the assumption is that the DW channels use the CRC functionality, which is mutually exclusive in time. The following pseudocode describes the functionality of a CRC transfer.

Required configurations: DESCR_CTL, DESCR_SRC, DESCR_DST, DESCR_X_CTL, and DESRC_NEXT_PTR.

Required registers: CRC_CTL, CRC_DATA_CTL, CRC_POL_CTL, CRC_LFSR_CTL, CRC_REM_CTL, and CRC_REM_RESULT.

```
// DST_ADDR is a pointer to an address location where the calculated CRC is stored.
            // SRC_ADDR is a pointer to an object of type defined by DESCR_SRC
            // t_DATA_SIZE is the type associated with the DATA_SIZE
            CRC_STATE = CRC_LFSR_CTL;
            for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
            Update_CRC (CRC_STATE, (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR];
            }
            DST_ADDR = CRC_STATE;
```

9.4.2 Descriptor size

The size of a descriptor depends on its type. Only relevant parameters are stored. For example, a single transfer descriptor does not contain the parameters DESCR_X_CTL and DESCR_Y_CTL. When a channel gets triggered, the descriptor is pulled from memory into the channel's active descriptor registers: CHi_ACT_DESCR_CTL, CHi_ACT_DESCR_SRC, CHi_ACT_DESCR_DST, CHi_ACT_DESCR_X_CTL, CHi_ACT_DESCR_Y_CTL, and CHi_ACT_DESCR_NEXT_PTR. The DMA only pulls the required 32-bit descriptor words for the specified DESCR_TYPE.

9.4.3 Descriptor configuration

There are six 32-bit words that make up a channel descriptor. Not all configurations require all six 32-bit words. When reading the descriptor from memory, only the required descriptor size will be read. This means that for a single transfer, only four 32-bit words are read from memory. The descriptor layout is shown in Figure 23.

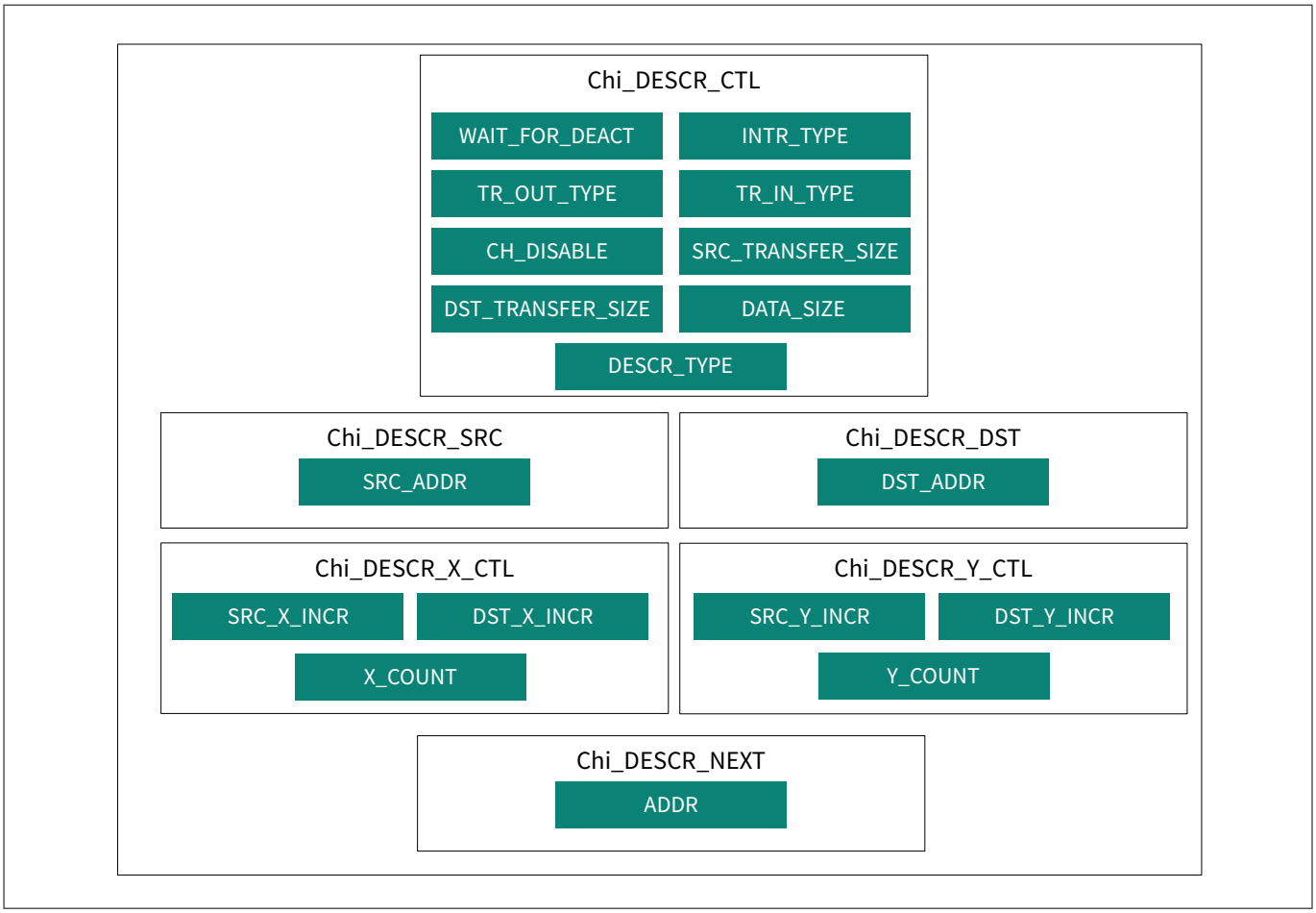


Figure 23 Descriptor layout

9.4.3.1 Descriptor control (DESCR_CTL)

This descriptor control register is broken into seven different controls required by all transfer types.

- WAIT_FOR_DEACT: Specifies whether the DMA controller should wait for the input trigger to be deactivated after it has completed the data transfer corresponding to the current trigger. This field is used for level-sensitive triggers to give sufficient time for the triggering agent to deactivate the trigger. The wait specified can be no wait (pulse-sensitive triggers), up to 4 cycles, up to 16 cycles, or indefinite. For more information, see the Trigger multiplexer (Trigger MUX) chapter
- INTR_TYPE: This defines when a completion interrupt is generated. The interrupt is generated in CHi_INTR. For more information, see the Channel interrupt generation section
- TR_OUT_TYPE: This defines when an output trigger is generated
- TR_IN_TYPE: An input trigger initiates a data transfer, and the TR_IN_TYPE defines the action on the trigger
- CH_DISABLE: Specifies whether the channel is disabled or not after completion of the current descriptor; this is independent of the value of the DESCR_NEXT_PTR value
- SRC_TRANSFER_SIZE: Specifies the bus transfer size to the source location. This can be set as the data size specified by DATA_SIZE or set to 32 bits
- DST_TRANSFER_SIZE: Specifies the bus transfer size to the destination location. This can be set as the data size specified by DATA_SIZE or set to 32 bits

- DATA_SIZE: Specifies the data element size if SRC_TRANSFER_SIZE or DST_TRANSFER_SIZE uses DATA_SIZE. This can set the data element size to 8-bit, 16-bit, or 32-bit
- DESCR_TYPE: Specifies the transfer type of the descriptor. DMA can be configured for single, 1D, 2D, or CRC transfers. See the Descriptor memory transfer types section for more information

9.4.3.2 Source and destination address (DESCR_SRC/DESCR_DST)

The source and destination addresses are set in the respective registers in the descriptor. These set the base addresses for the source and destination locations for the transfer. In case the descriptor is configured to transfer a single element, this descriptor holds the source/destination address of the data element. If the descriptor is configured to transfer multiple elements with a source address or destination address or both in an incremental mode, this field will hold the address of the first element that is transferred.

When the transfer type is set to CRC, this field should be programmed with the address of the DW_CRC_LSFR_CTL register. The calculated CRC LFSR state is written to this address with the AHB interface.

9.4.3.3 Descriptor X loop control (DESCR_X_CTL)

Used if DESCR_CTL.DESCR_TYPE is configured for 1D or 2D transfers.

- SRC_X_INCR: Specifies the increment of the source address for each X loop iteration (in multiples of SRC_TRANSFER_SIZE). This field is a signed number in the range [-2048, 2047]. If this field is "0", the source address is not incremented. This is useful for reading from RX FIFO structures
- DST_X_INCR: Specifies the increment of the destination address for each X loop iteration (in multiples of DST_TRANSFER_SIZE). This field is a signed number in the range [-2048, 2047]. If this field is "0", the destination address is not incremented. This is useful for writing to TX FIFO structures. This field is not used if the DESCR_TYPE is set to CRC
- X_COUNT: Number of iterations (minus 1) of the "X loop" (X_COUNT+1 is the number of single transfers in a 1D transfer). This field is an unsigned number in the range [0, 255], representing 1 through 256 iterations

9.4.3.4 Descriptor Y loop control (DESCR_Y_CTL)

Used if DESCR_CTL.DESCR_TYPE is configured for 2D transfers.

- SRC_Y_INCR: Specifies the increment of the source address for each Y loop iteration (in multiples of SRC_TRANSFER_SIZE). This field is a signed number in the range [-2048, 2047]
- DST_Y_INCR: Specifies the increment of the destination address for each Y loop iteration (in multiples of DST_TRANSFER_SIZE). This field is a signed number in the range [-2048, 2047]
- Y_COUNT: Number of iterations (minus 1) of the "Y loop" (X_COUNT+1)*(Y_COUNT+1) is the number of single transfers in a 2D transfer. This field is an unsigned number in the range [0, 255], representing 1 through 256 iterations

9.4.3.5 Descriptor next pointer (DESCR_NEXT_PTR)

Address of the next descriptor in the descriptor list. When this field is "0", this is the last descriptor in the descriptor list.

9.4.4 Descriptor transfer Size

The word width for a transfer can be configured using the transfer/data size parameter in the descriptor. The settings are diversified into source transfer size, destination transfer size, and data size. The data size parameter (DESCR_CTL.DATA_SIZE) sets the width of the bus for the transfer. The source and destination transfer sizes set

by DESCR_CTL.SCR_TRANSFER_SIZE and DESCR_CTL.DST_TRANSFER_SIZE can have a value of either DATA_SIZE or 32 bits. DESCR_CTL.DATA_SIZE can have a 32-bit, 16-bit, or 8-bit setting.

The source and destination transfer size for the DMA must match the addressable width of the source and destination, regardless of the width of data that must be moved. The DATA_SIZE parameter corresponds to the width of the actual data. For example, if a 16-bit PWM is used as a destination for DMA data, the DST_TRANSFER_SIZE must be set to 32 bits to match the width of the PWM register, because the peripheral register width for the TCPWM block (and most peripherals) is always 32 bits wide. However, in this example, the DATA_SIZE for the destination may still be set to 16-bit because the 16-bit PWM only uses two bytes of data.

Table 59 summarizes the possible combinations of the transfer size.

Table 59 Data transfer width

DATA_SIZE	SCR_TRANSFER_SIZE	DST_TRANSFER_SIZE	Typical usage	Description
8-bit	8-bit	8-bit	Memory to memory	No data manipulation
8-bit	32-bit	8-bit	Peripheral to memory	Higher 24 bits from the source dropped
8-bit	8-bit	32-bit	Memory to peripheral	Higher 24 bits zero padded at destination
8-bit	32-bit	32-bit	Peripheral to peripheral	Higher 24 bits from the source dropped and higher 24 bits zero padded at destination
16-bit	16-bit	16-bit	Memory to memory	No data manipulation
16-bit	32-bit	16-bit	Peripheral to memory	Higher 16 bits from the source dropped
16-bit	16-bit	32-bit	Memory to peripheral	Higher 16 bits zero padded at destination
16-bit	32-bit	32-bit	Peripheral to peripheral	Higher 16 bits from the source dropped and higher 16-bit zero padded at destination
32-bit	32-bit	32-bit	Peripheral to peripheral	No data manipulation

9.4.5 Descriptors chaining

When descriptors are chained together, it is known as a descriptor list. The DESCR_NEXT_PTR field contains a pointer to the next descriptor in the chain. Align this pointer to a multiple of 8 bytes. A channel executes the next descriptor in the chain when it completes executing the current descriptor. In the last descriptor in the chain, DESCR_NEXT_PTR is set to “0” (a NULL pointer). It is possible to have a circular list, in which case the execution continues indefinitely until there is an error or the channel or the controller is disabled by SW. A circular list includes descriptors that point to itself.

9.5 Triggers

Each DMA channel has one input trigger and one output trigger. These triggers are used to start a data transfer or to signal another peripheral at the end of the transfer. Both input and output triggers are configured per descriptor.

9.5.1 Input triggers

An input trigger initiates a data transfer. This trigger may originate from the source of the transfer, the destination of the transfer, the CPU SW, or from another peripheral. Triggers provide CPU Active/Sleep functionality and are not available in CPU/System DeepSleep and Hibernate power modes.

Each channel connects to a specific system trigger through a trigger multiplexer that is outside the DMA controller. The trigger multiplexer block routes trigger signals to the specified destinations. The DMA is one such destination. Refer to the Trigger multiplexer (Trigger MUX) chapter for details on the trigger multiplexer block.

The DMA controls what happens on trigger inputs using the TR_IN_TYPE and WAIT_FOR_DEACT fields. These fields are both part of the DESCR_CTL register.

TR_IN_TYPE: Each DMA channel's input triggers are configured using DESCR_CTL.TR_IN_TYPE, which defines the action of each channel when triggered. Table 60 describes the different trigger types.

Table 60 Input trigger types

Trigger type	Description
Type 0	A trigger results in the execution of a single transfer. The descriptor type can be single, 1D, or 2D.
Type 1	A trigger results in the execution of a single 1D transfer. If the descriptor type is "single", the trigger results in the execution of a single transfer. If the descriptor type is "1D" or "2D", the trigger results in the execution of a 1D transfer.
Type 2	A trigger results in the execution of the current descriptor.
Type 3	A trigger results in the execution of the current descriptor and continues (without requiring another input trigger) with the execution of the next descriptor using the next descriptor's information.

WAIT_FOR_DEACT: The DMA has controls for input trigger deactivation. When the DMA transfer based on the TR_IN_TYPE is completed, the data transfer engine checks the state of trigger deactivation. The data transfer on the second trigger is initiated only after the deactivation of the first. The DESCR_CTL.WAIT_FOR_DEACT parameter will determine when the trigger signal is considered deactivated. The first DMA transfer is activated when the trigger is activated, but the transfer is not considered complete until the trigger is deactivated. This field is used to synchronize the controller's data transfers with the agent that generated the trigger. The different trigger deactivations are listed in Table 61.

Table 61 Trigger deactivation types

Deactivation types	Description
Type 0	Pulse trigger: Does not wait for deactivation. When a trigger is detected, the transfer is initiated. After completing the transfer, if the trigger is still active, it is considered another trigger, and the subsequent transfer is initiated immediately.

(table continues...)

Table 61 (continued) Trigger deactivation types

Deactivation types	Description
Type 1	Level-sensitive: Waits for four slow clock cycles after the transfer to be considered as a deactivation. When a trigger is detected, the transfer is initiated. After completing the transfer, if the trigger is still active, then it is considered another trigger after waiting for four cycles. Then, a subsequent transfer is initiated. The transfer corresponding to the trigger is considered complete only at the end of four additional cycles. Even trigger output events will be affected based on this delay. This parameter adds up to a 4-cycle delay in each trigger transaction and hence affects throughput.
Type 2	Level-sensitive: Waits for 16 slow clock cycles after the transfer to be considered as a deactivation. When a trigger is detected, the transfer is initiated. After completing the transfer, if the trigger is still active, then it is considered another trigger after waiting for 16 cycles. Then, a subsequent transfer is initiated. The transfer corresponding to the trigger is considered complete only at the end of the 16 additional cycles. Even trigger output events will be affected based on this delay. This parameter adds up to a 16-cycle delay in each trigger transaction and hence affects throughput.
Type 3	Pulse trigger: Waits indefinitely for deactivation. The DMA transfer is initiated after the trigger signal deactivates. The next transfer is initiated only if the trigger goes low and then high again. A trigger signal that remains active or does not transition to zero between two transactions will simply stall the DMA channel.

A DMA transfer can also be triggered by software. Writing a '1' to CHi_TR_CMD generates a trigger, which sets a trigger pending on the channel. The channel must be enabled.

9.5.2 Pending triggers

A channel is considered pending when an enabled channel is triggered (CHi_STATUS.PEDNING set to 1). Once triggered, the DMA controller initiates a transfer if no other channel is in a pending state.

If there are multiple channels with a pending state, then the DMA will use priority decoding to determine the highest-priority pending channel. This channel is determined as follows:

1. First, the highest-priority group with pending channels is identified
2. Second, within this priority group, round-robin arbitration is applied, which gives the highest priority to the lower channel indices (within the priority group). The channel priority can be set by using CHi_CTL.PRIO

9.5.3 Output triggers

Each channel has an output trigger, which is high for at least two slow clock cycles. This trigger is generated on the completion of a data transfer. At the system level, these output triggers can be connected to the trigger multiplexer component. The trigger multiplexer block is outside the DMA block (refer to the Trigger Multiplexer Block section). This connection allows a DMA controller output trigger to be connected to a DMA controller input trigger. In other words, the completion of a transfer in one channel can activate another channel or even reactivate the same channel.

TR_OUT_TYPE: The DMA channel output triggers are configured using CHi_DESCR_CTL.TR_OUT_TYPE defines when an output trigger is generated. The different trigger types are listed in the following table.

Table 62 Output trigger types

Trigger type	Description
Type 0	An output trigger is generated after a single transfer.
Type 1	An output trigger is generated after a single 1D transfer. If the descriptor type is "single", the output trigger is generated after a single transfer. If the descriptor type is "1D", "CRC", or "2D", the output trigger is generated after the execution of a 1D transfer.
Type 2	An output trigger is generated after the execution of the current descriptor.
Type 3	An output trigger is generated after the execution of a descriptor list: after the execution of the current descriptor AND the current descriptor DESCR_NEXT_PTR.ADDR is “0”.

9.6 Performance

A clock cycle estimate can be calculated for a DMA transfer. This estimate depends on the transfer type specified in DESCR_CTL.DESCR_TYPE.

9.6.1 Single transfer

The DMA controller takes 14 clock cycles (of “clk”) for a single data transfer if there are no wait states on the bus infrastructures for loading descriptors or for loading/storing the data elements. The following is a breakdown of the clock cycles:

- Two cycles for trigger synchronization
- One cycle for detecting, priority decoding, and making the channel pending
- One cycle for the controller state machine to detect the pending channel and start
- Two cycles for descriptor channel control information to be loaded from DW channel configuration memory to DW control flip-flops
- Four cycles to load the descriptors (DESCR_CTL, DESCR_SRC, and DESCR_DST)
- One cycle to load the next pointer (DESCT_NEXT_PTR)
- Three cycles for loading the data from the source and for storing the data at the destination

9.6.2 1D transfer

The following is the formula for calculating the number of cycles for completing a 1D transfer of “n” data elements:

Number of cycles = 12 + n*3 + m, where “m” is the total number of wait states seen by the DW controller for loading/storing descriptors/data.

The following is the split of the 12 fixed cycles in the above expression:

- Two cycles for trigger synchronization
- One cycle for detecting, priority decoding, and making the channel pending
- One cycle for the controller state machine to detect the pending channel and start
- Two cycles for descriptor channel control information to be loaded from the DW channel configuration memory to the DW control flip-flops
- Four cycles to load the descriptors (DESCR_CTL, DESCR_SRC, and DESCR_DST)
- One cycle to load descriptor X control (DESCR_X_CTL)
- One cycle to load the next pointer (DESCT_NEXT_PTR)

Each data element transfer from source to destination takes three cycles, which is represented by n*3 in the formula.

9.6.3 2D transfer

The following is the formula for calculating the number of cycles for completing a 2D transfer of “n” data elements:

Number of cycles = 13 + n*3 + m, where “m” is the total number of wait states seen by the DW controller for loading/storing descriptors/data.

The following is the split of the 13 fixed cycles in the above expression:

- Two cycles for trigger synchronization
- One cycle for detecting, priority decoding, and making the channel pending
- One cycle for the controller state machine to detect the pending channel and start
- Two cycles for descriptor channel control information to be loaded from DW channel configuration memory to DW control flip-flops
- Four cycles to load the descriptors (DESCR_CTL, DESCR_SRC, and DESCR_DST)
- One cycle to load descriptor X control (DESCR_X_CTL)
- One cycle to load descriptor Y control (DESCR_Y_CTL)
- One cycle to load the next pointer (DESCT_NEXT_PTR)

Each data element transfer from source to destination takes three cycles, which is represented by n*3 in the formula.

9.6.4 CRC transfer

The following is the formula for calculating the number of cycles for completing the CRC computation of “n” data words.

Number of cycles = 17 + n*9 + m, where “m” is the total number of wait states seen by the DW controller for loading/storing descriptors/data.

The CRC computation for each data word takes ‘9’ clock cycles, which is represented by n*9 in the formula (assuming the CRC descriptor configuration uses word transfers). The extra 17 cycles include trigger synchronization, detection, loading of the descriptor, and storing the computed CRC back at the destination.

10 Trigger multiplexer (Trigger MUX)

Every peripheral in this MCU device is interconnected using trigger signals. Trigger signals are used by peripherals to indicate the occurrence of an event or transition to a state.

These triggers are used to effect or initiate an action in other peripherals. The trigger multiplexers help the user to route triggers from a source peripheral to a destination. Triggers are produced by one peripheral and consumed by another. Unlike interrupts, triggers are used to synchronize between peripherals rather than between a peripheral and the Arm® CPU core.

10.1 Features

Trigger multiplexer has the following features:

- Ability to connect any trigger signal from one peripheral to another
- Provides multiplexer-based trigger groups and 1-to-1 trigger groups
- Supports a software trigger that can trigger any signal in the block
- Ability to configure a trigger multiplexer with trigger manipulation features in hardware, such as inversion and edge/level detection
- Ability to block triggers in debug mode

10.2 Architecture

Triggers are used to synchronize the functionality of peripherals in hardware (as opposed to software-based synchronization). Peripheral DMA uses triggers to initiate the transfer of a data element from one address location to another. For example, an “ADC conversion done” event may initiate the transfer of an ADC sample from an ADC module to an SRAM memory location.

Triggers are digital signals generated by peripheral blocks to denote a state, such as FIFO level, or an event, such as the completion of an action. These trigger signals typically serve as initiators of other actions in other peripheral blocks. An example is an ADC peripheral block sampling three channels. After the conversion is complete, a trigger signal will be generated, which in turn triggers a DMA channel that transfers the ADC data to a memory buffer. This example is shown in Figure 24.

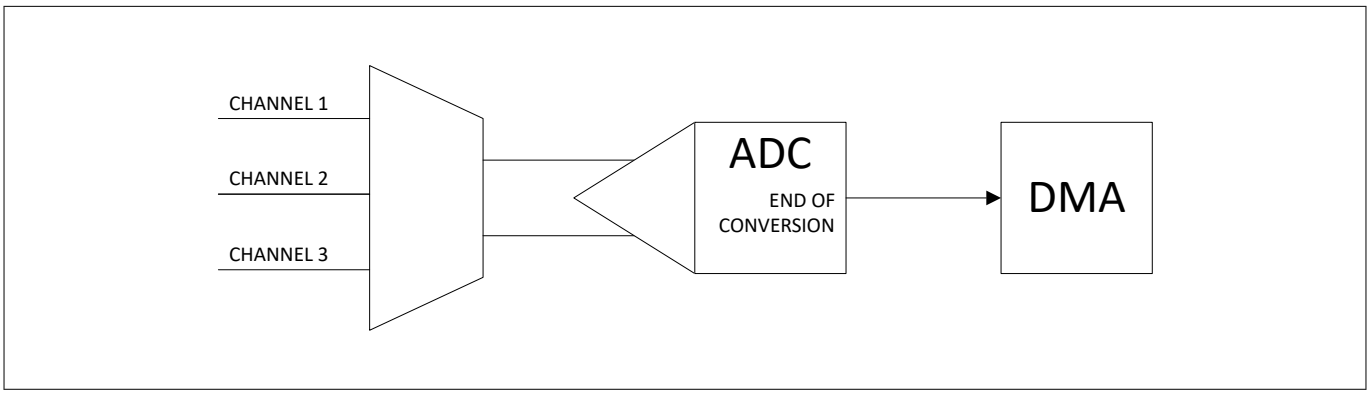


Figure 24 Trigger signal example

The PSC3M/P6 MCU has multiple peripheral blocks; each of these blocks can be connected to other blocks through trigger signals, based on the system implementation. To support this, there is hardware, which is a series of multiplexers used to route the trigger signals from potential sources to destinations. This hardware is called the trigger multiplexer block. The trigger multiplexer can connect to any trigger signal emanating out of any peripheral block in the MCU and route it to any other peripheral to initiate or affect an operation at the destination peripheral block.

Triggers come in two types:

- High-active, level-sensitive triggers: This type typically reflects a peripheral state. For example, tr_fifo_empty indicates that a FIFO is empty. The trigger remains '1' as long as the FIFO is empty. This type requires an action by the consumer of the trigger for the producer to deactivate the trigger. For example, tr_fifo_empty is deactivated by writing a data element to the associated FIFO. This trigger type can be produced on any clock
- Rising edge, pulse triggers: This type typically reflects the occurrence of an event. For example, tr_adc_done indicates that a SAR ADC has converted a sample. This trigger type is produced on the peripheral system interface clock: the trigger remains '1' for a two-cycle pulse on the peripheral system interface clock and returns to '0' by itself

If the consumer of the trigger cannot immediately react to the trigger, it needs to be able to remember that the trigger occurred. If the consumer of the trigger is confronted with multiple triggers in short succession, it may need to remember multiple triggers or decide to miss triggers.

Triggers should be treated as asynchronous signals between the producer and consumer peripheral; the consumer peripheral should synchronize input triggers. In addition, for pulse triggers, the consumer peripheral may perform rising edge detection and memorize the occurrence of the trigger.

At a high-level of abstraction, a trigger is a wire connection between a producer and a consumer peripheral. However, at a more detailed level, several processing steps are distinguished.

In general, a trigger input signal indicates the completion of a peripheral action or a peripheral event. In general, a trigger output signal initiates a peripheral action.

The decision for standard multiplexer components (as part of peripheral groups) rather than multiplexer components integrated as part of peripherals has the following rationale:

- A standard multiplexer component enforces a consistent user interface. The selection of a trigger input signal for a specific trigger output signal (peripheral input signal) is the same for all components.
- Any additional functionality provided by the multiplexer components, such as software control over output trigger signal activation, benefits all components.

The trigger module provides multiplexing functionality. It may be required to perform any of the following functions on a peripheral that uses the trigger output signals:

- Synchronization of the trigger signal to the peripheral clock domain
- Edge detection on the trigger signal
- Storing/remembering the trigger signal

A trigger component consists of multiple trigger groups. A trigger group can be of two types:

- A 1-to-1 based connectivity group: This group type connects a peripheral input trigger to one specific peripheral output trigger
- A multiplexer-based connectivity group: This type connects a peripheral input trigger to multiple peripheral output triggers. The selection is under software control: PERI_TR_GR_TR_CTL.TR_SEL

The trigger component for PSC3M/P6 MCU provides fifteen multiplexer-based trigger groups and one 1-to-1 trigger groups.

10.3 Functional description

The trigger component multiplexes trigger signals. The trigger input source signals are typically peripheral output signals. The trigger output destination signals are typically peripheral input signals.

Examples of trigger input source signals are:

- TCPWM output signals; for example, a counter reaches a pre-programmed limit
- ADC output signals; for example, an ADC conversion has completed
- I/O input signals
- DMA controller output signals that indicate the completion of a transfer

Examples of trigger output destination signals are:

- TCPWM input signals; for example, “start a counter”
- ADC input signals; for example, “start an ADC conversion”
- I/O output signals
- DMA controller input signals to start a transfer

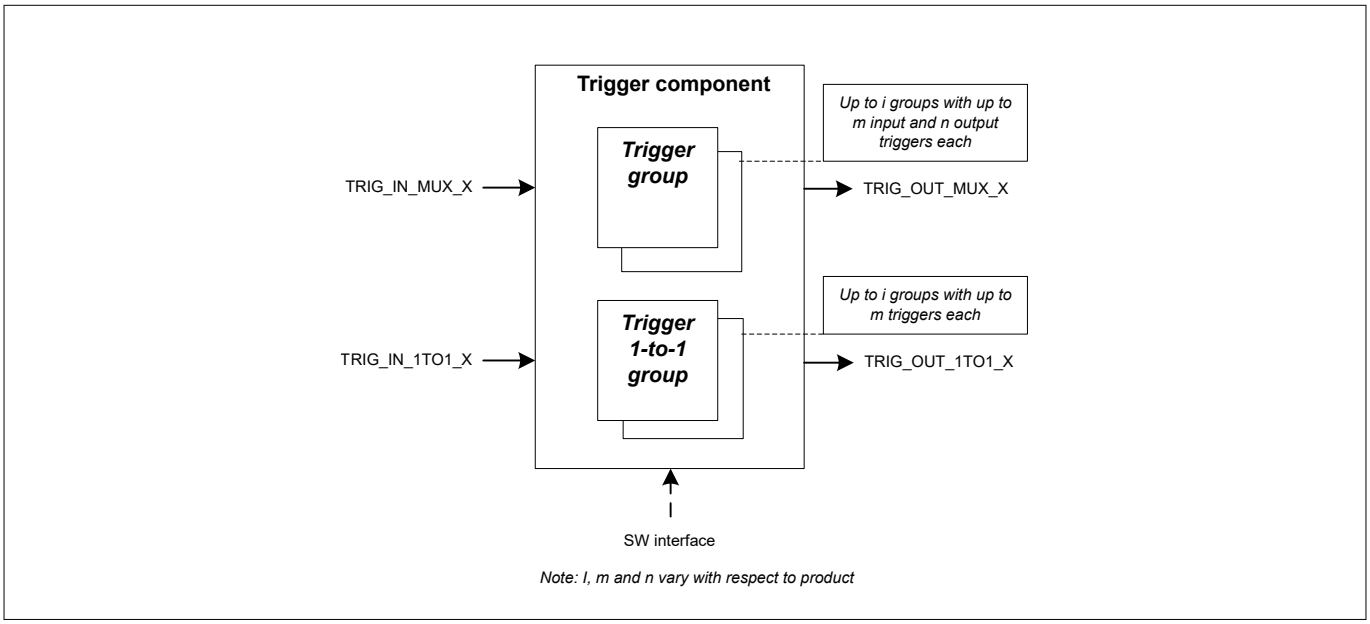


Figure 25 Trigger configuration parameters

Each group is associated with the trigger inputs of a specific peripheral. Figure 25 gives an overview.

Trigger destination peripherals are connected to trigger source peripheral components TRIG_IN_MUX_x and TRIG_IN_1TO1_x. Trigger source peripherals are connected to trigger destination peripheral components TRIG_OUT_MUX_x and TRIG_OUT_1TO1_x. It is important to distinguish between the functionality that is provided by the trigger component in PERI and the functionality that is provided by the peripheral.

The trigger component provides the following functionality (on TRIG_IN_MUX_x and TRIG_IN_1TO1_x):

- For a multiplexer-based connectivity group, an input trigger TRIG_IN_MUX_x is selected for each output trigger TRIG_OUT_MUX_x. Note that all output triggers in a group[i] share the same input triggers. For a 1-to-1 based group, an input trigger TRIG_IN_1TO1_x is connected to an output trigger TRIG_OUT_1TO1_x
- Software control is provided for trigger activation. This control allows for the activation of a specific trigger. For level-sensitive triggers, the trigger activation is completely under software control. For edge-sensitive triggers, the trigger activation is two high/'1' cycles on CLK_PERI
- Trigger propagation can be blocked in debug mode (typically when a CPU is halted). This allows it to isolate the trigger consumer peripheral from getting input triggers during a debug halt. The debug mode is indicated by the level trigger input x_DEBUG_FREEZE_TR_IN1, which is connected to a CPUSS CTI trigger output
- Hardware edge detection is provided to allow pulse triggers that transfer to the output trigger clock domain. This hardware performs asynchronous edge detection to support input triggers that operate on a higher clock frequency than the output trigger clock domain. This functionality is intended for pulse triggers (level triggers typically bypass the edge detection functionality)
- Hardware trigger signal level inversion
- Most trigger multiplexers have all output signals connected to a common peripheral. The manipulation logic is tied to the clock for that peripheral. However, the debug multiplexer has outputs in many clock domains. This is valid because most destinations are level-sensitive signals, I/Os, or some other debug destinations that might not need any clock manipulations

A peripheral may provide the following functionality:

- Synchronization of the output triggers TRIG_OUT_MUX_x and TRIG_OUT_1TO1_x
- Edge detection of the synchronized output triggers
- Storing/remembering the trigger signal

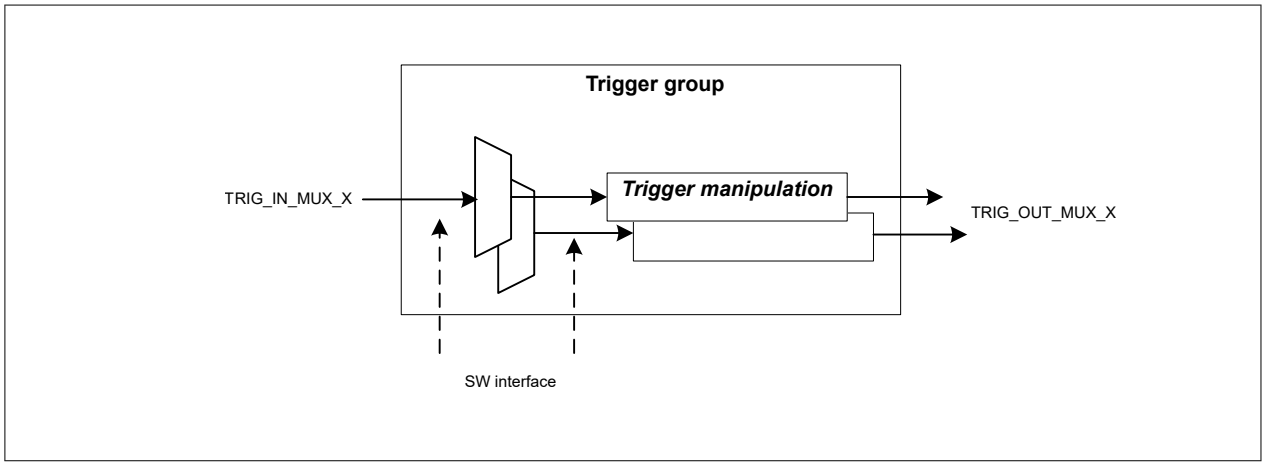


Figure 26 Trigger group

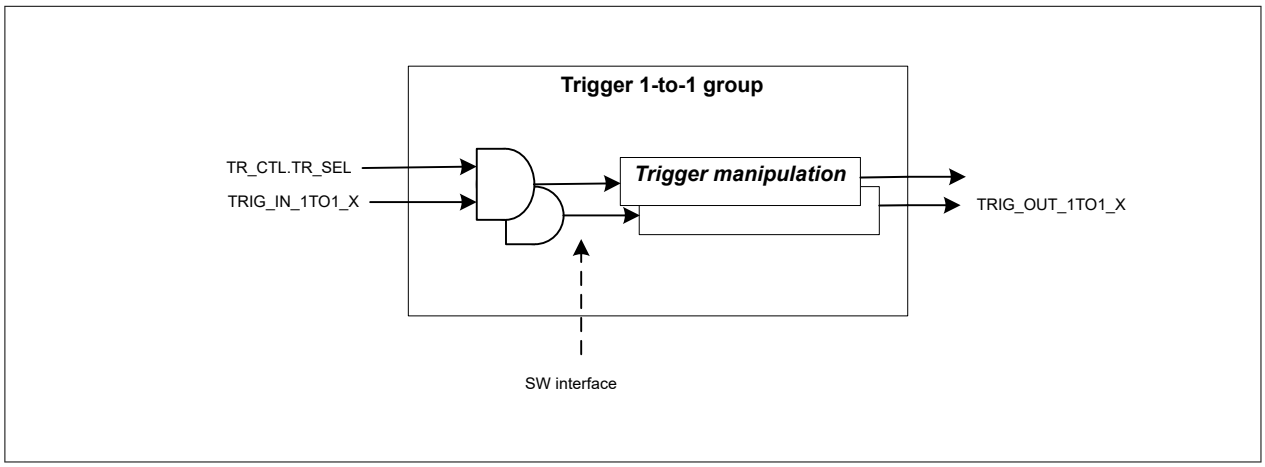


Figure 27 Trigger 1-to-1 group

***Note**: A 1-to-1 group has AND-gate functionality to disable an input trigger*

An overview of the Trigger multiplexer connections for PSC3M/P6 MCU is shown in Table 63.

Table 63 Trigger MUX connections overview

In	Trigger MUX	Out
PDMA0[0:3]	PDMA0	DW0 - 16 channels
TCPWM[all OUT0]		
IOSS[0:67]		
SCB[all]		
CTI_TR_OUT[0:1]		
MXCODIC_DONE		

(table continues...)

Table 63 (continued) Trigger MUX connections overview

In	Trigger MUX	Out
CAN_TT_TR_OUT[0:1]		
LPCOMP[comp0, comp1]		
CPUSS_TR_FAULT[0]		
TRNG_BITSTREAM		
HPPASS[all]		
I3C[tx,rx]		
USB_DMA_REQ[0:7]		
Same "in" as PDMA0	PDMA1	DW1 - 16 channels
PDMA0[all]	HSIOM0 Trigger	IOSS.peri_tr_io_output_out[0:67]
PDMA1[all]		
TCPWM[all out1]		
TCPWM0_MOTIF [TR_OUT all]		
TCPWM tr_line[all]		
TCPWM tr_line_compl[all]		
CTI_TR_OUT[0:1]		
MXCODIC_DONE		
CAN_TT_TR_OUT[0:1]		
LPCOMP[comp0, comp1]		
CPUSS_TR_FAULT[0]		
TRNG_BITSTREAM		
HPPASS[all]		
I3C[tx,rx]		
IOSS[0:67]	HSIOMLOOP Trigger	IOSS[68:135]
TCPWM[line][0:3]	HSIOM1 Trigger	IOSS[136:143]
TCPWM[line_compl][0:3]		
TCPWM[line][256:259]	HSIOM2 Trigger	IOSS[144:151]
TCPWM[line_compl][256:259]		
TCPWM[line][512:515]	HSIOM3 Trigger	IOSS[152:159]
TCPWM[line_compl][512:515]		
PDMA0[all]	CPUSS_CTI	CPUSS.CTI_TR_IN[0:1]
PDMA1[all]		
TCPWM[all out1]		
TCPWM0_MOTIF [TR_OUT all]		

(table continues...)

Table 63 (continued) Trigger MUX connections overview

In	Trigger MUX	Out
IOSS[0:67]		
SCB[all]		
CTI_TR_OUT[0:1]		
MXCODIC_DONE		
CAN_TT_TR_OUT[0:1]		
LPCOMP[comp0, comp1]		
CPUSS_TR_FAULT[0]		
HPPASS[all]		
I3C[tx,rx]		
USB_DMA_REQ[0:7]		
CAN_TT_TR_OUT[0:1]	CANTT	CAN_TT_TR_IN[0:1]
CPUSS_TR_FAULT[0]	PERI_FREEZE	PERI_DEBUG_FREEZE_TR_IN
CTI_TR_OUT[0:1]		TCPWM_DEBUG_FREEZE_TR_IN
LPCOMP[comp0, comp1]	TCPWM0	TCPWM0_TR_IN[0:25]
HPPASS[all]		
PDMA0[all]		
PDMA1[all]		TCPWM0[TR2_IN all]
TCPWM[all out1]		
TCPWM0_MOTIF [TR_OUT all]		
IOSS[0:67]		
SCB[all]		TCPWM0[TR3_IN all]
CANFD[all]		
CPUSS_TR_FAULT[0]		
I3C[tx,rx]		
TCPWM0[OUT0 all]	TCPWM1	TCPWM0[TR4 all]
TCPWM0[OUT0, OUT1 all]	TCPWM_MOTIF	TCPWM0_MOTIF [TR_IN all]
IOSS[0:67]		
HPPASS[TR_LEVEL all]		
TCPWM0[OUT1 all]	HPPASS	HPPASS_TR_A_IN[all]
TCPWM0_MOTIF [TR_OUT all]		HPPASS_TR_B_IN[all]
IOSS[0:67]		
PDMA0[all]	USBFS	USB_DMA_BURSTEND[0:7]
PDMA1[all]		

***Note**: PDMA and Data Wire (DW) is used interchangeably for register names and for triggers in this section.*

The detailed Trigger MUX connections for PSC3M/P6 MCU can be found in the following tables:

Table 64 PERI Trigger source group 0 (PDMA0 Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1:4	PDMA0_TR_OUT[0:3]	Allow P-DMA0 to chain to PDMA0. Channels 0 - 3 are for chaining
5:12	TCPWM0_GRP0_OUT0_[0:7]	32-bit TCPWM counters to PDMA0
13:20	TCPWM0_GRP1_OUT0_[0:7]	
21:28	TCPWM0_GRP2_OUT0_[0:7]	
29:96	HSIOM_TR_OUT[0:67]	Triggers connected to pins in HSIOM (peri.tr_in)
97	SCB_I2C_SCL[0]	
98	SCB_TX[0]	SCB TX/RX DMA requests
99	SCB_RX[0]	
100	SCB_TGS_RELOAD[0][0]	
101	SCB_TGS_STOP[0][0]	
102	SCB_TGS_GATED[0][0]	
103	SCB_I2C_SCL[1]	
104	SCB_TX[1]	
105	SCB_RX[1]	
106	SCB_TGS_RELOAD[1][0]	
107	SCB_TGS_STOP[1][0]	
108	SCB_TGS_GATED[1][0]	
109	SCB_I2C_SCL[2]	
110	SCB_TX[2]	
111	SCB_RX[2]	
112	SCB_TGS_RELOAD[2][0]	
113	SCB_TGS_STOP[2][0]	
114	SCB_TGS_GATED[2][0]	
115	SCB_I2C_SCL[3]	
116	SCB_TX[3]	
117	SCB_RX[3]	
118	SCB_TGS_RELOAD[3][0]	
119	SCB_TGS_STOP[3][0]	
120	SCB_TGS_GATED[3][0]	

(table continues...)

Table 64 (continued) PERI Trigger source group 0 (PDMA0 Trigger MUX)

Trigger MUX input	Trigger source label	Description
121	SCB_I2C_SCL[4]	
122	SCB_TX[4]	
123	SCB_RX[4]	
124	SCB_TGS_RELOAD[4][0]	
125	SCB_TGS_STOP[4][0]	
126	SCB_TGS_GATED[4][0]	
127	SCB_I2C_SCL[5]	
128	SCB_TX[5]	
129	SCB_RX[5]	
130	SCB_TGS_RELOAD[5][0]	
131	SCB_TGS_STOP[5][0]	
132	SCB_TGS_GATED[5][0]	
133:134	CTI_TR_OUT[0:1]	CPUSS Debug events
135	MXCORDIC_DONE	MXCORDIC done trigger
136	CANFD_TT_TR_OUT[0]	CAN TT Sync Outputs
137	CANFD_TT_TR_OUT[1]	CAN TT Sync Outputs
138	CAN_DBG0	
139	CAN_DBG1	
140	CAN_FIFO0_CH0	
141	CAN_FIFO0_CH1	
142	CAN_FIFO1_CH0	
143	CAN_FIFO1_CH1	
144	LPCOMP_DSI_COMP0	LPCOMP Comparator#0
145	LPCOMP_DSI_COMP1	LPCOMP Comparator#1
146	CPUSS_TR_FAULT[0]	
147	TRNG_BITSTREAM	TRNG_BITSTREAM
148:151	PASS_FIFO[0:3]	Triggers (LEVEL) for FIFO (DW)
152:167	PASS_PULSE[0:15]	Triggers (PULSE) for sar_result (DW) and AC (misc usage)
168:183	PASS_LEVEL[0:15]	Triggers (LEVEL) for MC algo
184	I3C_TX[0]	
185	I3C_RX[0]	
186:193	USB_DMA_REQ[0:7]	

Table 65 PERI Trigger destination group 0 (PDMA0 Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:15	PDMA0_TR_IN[0:15]	Triggers to P-DMA0

Table 66 PERI Trigger source group 1 (PDMA1 Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1:4	PDMA1_TR_OUT[0:3]	Allow P-DMA1 to chain to PDMA1. Channels 0 - 3 are for chaining
5:12	TCPWM0_GRP0_OUT0_[0:7]	32-bit TCPWM counters to PDMA1
13:20	TCPWM0_GRP1_OUT0_[0:7]	
21:28	TCPWM0_GRP2_OUT0_[0:7]	
29:96	HSIOM_TR_OUT[0:67]	Triggers connected to pins in HSIOM (peri.tr_in)
97	SCB_I2C_SCL[0]	
98	SCB_TX[0]	SCB TX/RX DMA requests
99	SCB_RX[0]	
100	SCB_TGS_RELOAD[0][0]	
101	SCB_TGS_STOP[0][0]	
102	SCB_TGS_GATED[0][0]	
103	SCB_I2C_SCL[1]	
104	SCB_TX[1]	
105	SCB_RX[1]	
106	SCB_TGS_RELOAD[1][0]	
107	SCB_TGS_STOP[1][0]	
108	SCB_TGS_GATED[1][0]	
109	SCB_I2C_SCL[2]	
110	SCB_TX[2]	
111	SCB_RX[2]	
112	SCB_TGS_RELOAD[2][0]	
113	SCB_TGS_STOP[2][0]	
114	SCB_TGS_GATED[2][0]	
115	SCB_I2C_SCL[3]	
116	SCB_TX[3]	
117	SCB_RX[3]	
118	SCB_TGS_RELOAD[3][0]	
119	SCB_TGS_STOP[3][0]	

(table continues...)

Table 66 (continued) PERI Trigger source group 1 (PDMA1 Trigger MUX)

Trigger MUX input	Trigger source label	Description
120	SCB_TGS_GATED[3][0]	
121	SCB_I2C_SCL[4]	
122	SCB_TX[4]	
123	SCB_RX[4]	
124	SCB_TGS_RELOAD[4][0]	
125	SCB_TGS_STOP[4][0]	
126	SCB_TGS_GATED[4][0]	
127	SCB_I2C_SCL[5]	
128	SCB_TX[5]	
129	SCB_RX[5]	
130	SCB_TGS_RELOAD[5][0]	
131	SCB_TGS_STOP[5][0]	
132	SCB_TGS_GATED[5][0]	
133:134	CTI_TR_OUT[0:1]	CPUSS Debug events
135	MXCORDIC_DONE	MXCORDIC done trigger
136	CANFD_TT_TR_OUT[0]	CAN TT Sync Outputs
137	CANFD_TT_TR_OUT[1]	CAN TT Sync Outputs
138	CAN_DBG0	
139	CAN_DBG1	
140	CAN_FIFO0_CH0	
141	CAN_FIFO0_CH1	
142	CAN_FIFO1_CH0	
143	CAN_FIFO1_CH1	
144	LPCOMP_DSI_COMP0	LPCOMP Comparator#0
145	LPCOMP_DSI_COMP1	LPCOMP Comparator#1
146	CPUSS_TR_FAULT[0]	
147	TRNG_BITSTREAM	TRNG_BITSTREAM
148:151	PASS_FIFO[0:3]	Triggers (LEVEL) for FIFO (DW)
152:167	PASS_PULSE[0:15]	Triggers (PULSE) for sar_result (DW) and AC (misc usage)
168:183	PASS_LEVEL[0:15]	Triggers (LEVEL) for MC algo
184	I3C_TX[0]	
185	I3C_RX[0]	
186:193	USB_DMA_REQ[0:7]	

Table 67 PERI Trigger destination group 1 (PDMA1 Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:15	PDMA1_TR_IN[0:15]	Triggers to P-DMA1

Table 68 PERI Trigger source group 2 (HSIOM0 Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1:16	PDMA0_TR_OUT[0:15]	P-DMA0 tr_out
17:32	PDMA1_TR_OUT[0:15]	P-DMA1 tr_out
33:40	TCPWM0_GRP0_OUT1_[0:7]	32-bit TCPWM counters to PDMA1
41:48	TCPWM0_GRP1_OUT1_[0:7]	
49:56	TCPWM0_GRP2_OUT1_[0:7]	
57:64	TCPWM0_MOTIF_GRP1_TR_OUT[0:7]	MOTIF Trigger Group #1 MOTIF #0
65:72	TCPWM0_MOTIF_GRP2_TR_OUT[0:7]	MOTIF Trigger Group #2 MOTIF #0
73:80	TCPWM0_GRP0_LINE_[0:7]	TCPWM trigger line
81:88	TCPWM0_GRP0_LINE_COMPL_[0:7]	TCPWM trigger line_compl
89:96	TCPWM0_GRP1_LINE_[256:263]	TCPWM trigger line
97:104	TCPWM0_GRP1_LINE_COMPL_[256:263]	TCPWM trigger line_compl
105:112	TCPWM0_GRP2_LINE_[512:519]	TCPWM trigger line
113:120	TCPWM0_GRP2_LINE_COMPL_[512:519]	TCPWM trigger line_compl
121	SCB_I2C_SCL[0]	
122	SCB_TX[0]	SCB TX/RX DMA requests
123	SCB_RX[0]	
124	SCB_TGS_RELOAD[0][0]	
125	SCB_TGS_STOP[0][0]	
126	SCB_TGS_GATED[0][0]	
127	SCB_I2C_SCL[1]	
128	SCB_TX[1]	
129	SCB_RX[1]	
130	SCB_TGS_RELOAD[1][0]	
131	SCB_TGS_STOP[1][0]	
132	SCB_TGS_GATED[1][0]	
133	SCB_I2C_SCL[2]	
134	SCB_TX[2]	

(table continues...)

Table 68 (continued) PERI Trigger source group 2 (HSIOM0 Trigger MUX)

Trigger MUX input	Trigger source label	Description
135	SCB_RX[2]	
136	SCB_TGS_RELOAD[2][0]	
137	SCB_TGS_STOP[2][0]	
138	SCB_TGS_GATED[2][0]	
139	SCB_I2C_SCL[3]	
140	SCB_TX[3]	
141	SCB_RX[3]	
142	SCB_TGS_RELOAD[3][0]	
143	SCB_TGS_STOP[3][0]	
144	SCB_TGS_GATED[3][0]	
145	SCB_I2C_SCL[4]	
146	SCB_TX[4]	
147	SCB_RX[4]	
148	SCB_TGS_RELOAD[4][0]	
149	SCB_TGS_STOP[4][0]	
150	SCB_TGS_GATED[4][0]	
151	SCB_I2C_SCL[5]	
152	SCB_TX[5]	
153	SCB_RX[5]	
154	SCB_TGS_RELOAD[5][0]	
155	SCB_TGS_STOP[5][0]	
156	SCB_TGS_GATED[5][0]	
157:158	CTI_TR_OUT[0:1]	Debug events
159	MXCORDIC_DONE	MXCORDIC done trigger
160	CANFD_TT_TR_OUT[0]	CAN TT Sync Outputs
161	CANFD_TT_TR_OUT[1]	CAN TT Sync Outputs
162	LPCOMP_DSI_COMP0	LPCOMP Comparator#0
163	LPCOMP_DSI_COMP1	LPCOMP Comparator#1
164	CPUSS_TR_FAULT[0]	
165	TRNG_BITSTREAM	TRNG_BITSTREAM
166:169	PASS_FIFO[0:3]	Triggers (LEVEL) for FIFO (DW)

(table continues...)

Table 68 (continued) PERI Trigger source group 2 (HSIOM0 Trigger MUX)

Trigger MUX input	Trigger source label	Description
170:185	PASS_PULSE[0:15]	Triggers (PULSE) for sar_result (DW) and AC (misc usage)
186:201	PASS_LEVEL[0:15]	Triggers (LEVEL) for MC algo
202	I3C_TX[0]	
203	I3C_RX[0]	

Table 69 PERI Trigger destination group 2 (HSIOM0 Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:67	HSIOM_TR_IO_OUTPUT[0:67]	Triggers to HSIOM

Table 70 PERI Trigger source group 3 (HSIOMLOOP Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1:68	HSIOM_TR_IO_OUTPUT[0:67]	Triggers connected from HSIOM (peri.tr_in)

Table 71 PERI Trigger destination group 3 (HSIOMLOOP Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:67	HSIOM_TR_IO_OUTPUT[68:135]	Triggers from HSIOM to HSIOM

Table 72 PERI Trigger source group 4 (HSIOM1 Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1:4	TCPWM0_GRP0_LINE[0:3]	
5:8	TCPWM0_GRP0_LINE_COMPL[0:3]	

Table 73 PERI Trigger destination group 4 (HSIOM1 Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:7	HSIOM_TR_IO_OUTPUT[136:143]	Triggers from TCPWM GRP0 to HSIOM

Table 74 PERI Trigger source group 5 (HSIOM2 Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1:4	TCPWM0_GRP1_LINE[0:3]	
5:8	TCPWM0_GRP1_LINE_COMPL[0:3]	

Table 75 PERI Trigger destination group 4 (HSIOM2 Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:7	HSIOM_TR_IO_OUTPUT[144:151]	Triggers from TCPWM GRP1 to HSIOM

Table 76 PERI Trigger source group 6 (HSIOM3 Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1:4	TCPWM0_GRP2_LINE[0:3]	
5:8	TCPWM0_GRP2_LINE_COMPL[0:3]	

Table 77 PERI Trigger destination group 6 (HSIOM3 Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:7	HSIOM_TR_IO_OUTPUT[152:159]	Triggers from TCPWM GRP2 to HSIOM

Table 78 PERI Trigger source group 7 (CPUSS_CTI Trigger MUX)

Trigger MUX input	Trigger source label	Description
151	SCB_TX[2]	
152	SCB_RX[2]	
153	SCB_I2C_SCL[3]	
154	SCB_TX[3]	
155	SCB_RX[3]	
156	SCB_I2C_SCL[4]	
157	SCB_TX[4]	
158	SCB_RX[4]	
159	SCB_I2C_SCL[5]	
160	SCB_TX[5]	
161	SCB_RX[5]	
162	SCB_TGS_RELOAD[5][0]	
163	SCB_TGS_STOP[5][0]	
164	SCB_TGS_GATED[5][0]	
165:166	CTI_TR_OUT[0:1]	Debug events
167	MXCORDIC_DONE	MXCORDIC done trigger
168	CANFD_TT_TR_OUT[0]	CAN TT Sync Outputs
169	CANFD_TT_TR_OUT[1]	CAN TT Sync Outputs
170	LPCOMP_DSI_COMP0	LPCOMP Comparator#0
171	LPCOMP_DSI_COMP1	LPCOMP Comparator#1

(table continues...)

Table 78 (continued) PERI Trigger source group 7 (CPUSS_CTI Trigger MUX)

Trigger MUX input	Trigger source label	Description
172	CPUSS_TR_FAULT[0]	
173:176	PASS_FIFO[0:3]	Triggers (LEVEL) for FIFO (DW)
177:192	PASS_PULSE[0:15]	Triggers (PULSE) for sar_result (DW) and AC (misc usage)
193:208	PASS_LEVEL[0:15]	Triggers (LEVEL) for MC algo
209	I3C_TX[0]	
210	I3C_RX[0]	
211:218	USB_DMA_REQ[0:7]	

Table 79 PERI Trigger destination group 7 (CPUSS_CTI Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:1	CPUSS_CTI_TR_IN[0:1]	Triggers to debug600.cti

Table 80 PERI Trigger source group 8 (CANTT Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1	CAN_TT_TR_OUT[0]	CAN TT sync outputs
2	CAN_TT_TR_OUT[1]	CAN TT sync outputs

Table 81 PERI Trigger destination group 8 (CANTT Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0	CAN_TT_TR_IN[0]	CAN TT sync inputs
1	CAN_TT_TR_IN[1]	CAN TT sync inputs

Table 82 PERI Trigger source group 9 (PERI_FREEZE Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1	CPUSS_TR_FAULT[0]	
2:3	CTI_TR_OUT[0:1]	Debug events

Table 83 PERI Trigger destination group 9 (PERI_FREEZE Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0	PERI_DEBUG_FREEZE_TR_IN	Trigger to freeze operation
1	TCPWM_DEBUG_FREEZE_TR_IN	

Table 84 PERI Trigger source group 10 (TCPWM0 Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1	LPCOMP_DSI_COMP0	LPCOMP Comparator#0
2	LPCOMP_DSI_COMP1	LPCOMP Comparator#1
3:6	PASS_FIFO[0:3]	Trigger to PERI for FIFO level
7:22	PASS_PULSE[0:15]	Trigger to PERI for MCPASS pulse triggers
23:38	PASS_LEVEL[0:15]	Trigger to PERI for range level
39:54	PDMA0_TR_OUT[0:15]	P-DMA0 tr_out
55:70	PDMA1_TR_OUT[0:15]	P-DMA1 tr_out
71:78	TCPWM0_GRP0_OUT1_[0:7]	
79:86	TCPWM0_GRP1_OUT1_[0:7]	
87:94	TCPWM0_GRP2_OUT1_[0:7]	
95:102	TCPWM0_MOTIF_GRP1_TR_OUT[0:7]	MOTIF Trigger Group #1 MOTIF #0
103:110	TCPWM0_MOTIF_GRP2_TR_OUT[0:7]	MOTIF Trigger Group #2 MOTIF #0
111:178	HSIOM_TR_OUT[0:67]	Triggers connected to pins in HSIOM (peri.tr_in)
179	SCB_I2C_SCL[0]	
180	SCB_TX[0]	SCB TX/RX DMA requests
181	SCB_RX[0]	
182	SCB_TGS_RELOAD[0][0]	
183	SCB_TGS_STOP[0][0]	
184	SCB_TGS_GATED[0][0]	
185	SCB_I2C_SCL[1]	
186	SCB_TX[1]	
187	SCB_RX[1]	
188	SCB_I2C_SCL[2]	
189	SCB_TX[2]	
190	SCB_RX[2]	
191	SCB_I2C_SCL[3]	
192	SCB_TX[3]	
193	SCB_RX[3]	
194	SCB_I2C_SCL[4]	
195	SCB_TX[4]	

(table continues...)

Table 84 (continued) PERI Trigger source group 10 (TCPWM0 Trigger MUX)

Trigger MUX input	Trigger source label	Description
196	SCB_RX[4]	
197	SCB_I2C_SCL[5]	
198	SCB_TX[5]	
199	SCB_RX[5]	
200	SCB_TGS_RELOAD[5][0]	
201	SCB_TGS_STOP[5][0]	
202	SCB_TGS_GATED[5][0]	
203	CANFD_TT_TR_OUT[0]	CAN TT Sync Outputs
204	CANFD_TT_TR_OUT[1]	CAN TT Sync Outputs
205	CPUSS_TR_FAULT[0]	
206	I3C_TX[0]	
207	I3C_RX[0]	

Table 85 PERI Trigger destination group 10 (TCPWM0 Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:25	TCPWM0_TR_IN[0:25]	Triggers to TCPWM0
26	TCPWM0_GRP0_CNT0_TR2_IN	Group [0], Counter[0], Trigger[2]
27	TCPWM0_GRP0_CNT1_TR2_IN	Group [0], Counter[1], Trigger[2]
28	TCPWM0_GRP0_CNT2_TR2_IN	Group [0], Counter[2], Trigger[2]
29	TCPWM0_GRP0_CNT3_TR2_IN	Group [0], Counter[3], Trigger[2]
30	TCPWM0_GRP0_CNT4_TR2_IN	Group [0], Counter[4], Trigger[2]
31	TCPWM0_GRP0_CNT5_TR2_IN	Group [0], Counter[5], Trigger[2]
32	TCPWM0_GRP0_CNT6_TR2_IN	Group [0], Counter[6], Trigger[2]
33	TCPWM0_GRP0_CNT7_TR2_IN	Group [0], Counter[7], Trigger[2]
34	TCPWM0_GRP1_CNT0_TR2_IN	Group [1], Counter[0], Trigger[2]
35	TCPWM0_GRP1_CNT1_TR2_IN	Group [1], Counter[1], Trigger[2]
36	TCPWM0_GRP1_CNT2_TR2_IN	Group [1], Counter[2], Trigger[2]
37	TCPWM0_GRP1_CNT3_TR2_IN	Group [1], Counter[3], Trigger[2]
38	TCPWM0_GRP1_CNT4_TR2_IN	Group [1], Counter[4], Trigger[2]
39	TCPWM0_GRP1_CNT5_TR2_IN	Group [1], Counter[5], Trigger[2]
40	TCPWM0_GRP1_CNT6_TR2_IN	Group [1], Counter[6], Trigger[2]
41	TCPWM0_GRP1_CNT7_TR2_IN	Group [1], Counter[7], Trigger[2]
42	TCPWM0_GRP2_CNT0_TR2_IN	Group [2], Counter[0], Trigger[2]
43	TCPWM0_GRP2_CNT1_TR2_IN	Group [2], Counter[1], Trigger[2]

(table continues...)

Table 85 (continued) PERI Trigger destination group 10 (TCPWM0 Trigger MUX)

Trigger MUX output	Trigger destination label	Description
44	TCPWM0_GRP2_CNT2_TR2_IN	Group [2], Counter[2], Trigger[2]
45	TCPWM0_GRP2_CNT3_TR2_IN	Group [2], Counter[3], Trigger[2]
46	TCPWM0_GRP2_CNT4_TR2_IN	Group [2], Counter[4], Trigger[2]
47	TCPWM0_GRP2_CNT5_TR2_IN	Group [2], Counter[5], Trigger[2]
48	TCPWM0_GRP2_CNT6_TR2_IN	Group [2], Counter[6], Trigger[2]
49	TCPWM0_GRP2_CNT7_TR2_IN	Group [2], Counter[7], Trigger[2]
50	TCPWM0_GRP0_CNT0_TR3_IN	Group [0], Counter[0], Trigger[3]
51	TCPWM0_GRP0_CNT1_TR3_IN	Group [0], Counter[1], Trigger[3]
52	TCPWM0_GRP0_CNT2_TR3_IN	Group [0], Counter[2], Trigger[3]
53	TCPWM0_GRP0_CNT3_TR3_IN	Group [0], Counter[3], Trigger[3]
54	TCPWM0_GRP0_CNT4_TR3_IN	Group [0], Counter[4], Trigger[3]
55	TCPWM0_GRP0_CNT5_TR3_IN	Group [0], Counter[5], Trigger[3]
56	TCPWM0_GRP0_CNT6_TR3_IN	Group [0], Counter[6], Trigger[3]
57	TCPWM0_GRP0_CNT7_TR3_IN	Group [0], Counter[7], Trigger[3]
58	TCPWM0_GRP1_CNT0_TR3_IN	Group [1], Counter[0], Trigger[3]
59	TCPWM0_GRP1_CNT1_TR3_IN	Group [1], Counter[1], Trigger[3]
60	TCPWM0_GRP1_CNT2_TR3_IN	Group [1], Counter[2], Trigger[3]
61	TCPWM0_GRP1_CNT3_TR3_IN	Group [1], Counter[3], Trigger[3]
62	TCPWM0_GRP1_CNT4_TR3_IN	Group [1], Counter[4], Trigger[3]
63	TCPWM0_GRP1_CNT5_TR3_IN	Group [1], Counter[5], Trigger[3]
64	TCPWM0_GRP1_CNT6_TR3_IN	Group [1], Counter[6], Trigger[3]
65	TCPWM0_GRP1_CNT7_TR3_IN	Group [1], Counter[7], Trigger[3]
66	TCPWM0_GRP2_CNT0_TR3_IN	Group [2], Counter[0], Trigger[3]
67	TCPWM0_GRP2_CNT1_TR3_IN	Group [2], Counter[1], Trigger[3]
68	TCPWM0_GRP2_CNT2_TR3_IN	Group [2], Counter[2], Trigger[3]
69	TCPWM0_GRP2_CNT3_TR3_IN	Group [2], Counter[3], Trigger[3]
70	TCPWM0_GRP2_CNT4_TR3_IN	Group [2], Counter[4], Trigger[3]
71	TCPWM0_GRP2_CNT5_TR3_IN	Group [2], Counter[5], Trigger[3]
72	TCPWM0_GRP2_CNT6_TR3_IN	Group [2], Counter[6], Trigger[3]
73	TCPWM0_GRP2_CNT7_TR3_IN	Group [2], Counter[7], Trigger[3]

Table 86 PERI Trigger source group 11 (TCPWM1 Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	

(table continues...)

Table 86 (continued) PERI Trigger source group 11 (TCPWM1 Trigger MUX)

Trigger MUX input	Trigger source label	Description
1:8	TCPWM0_GRP0_OUT0_[0:7]	
9:16	TCPWM0_GRP1_OUT0_[0:7]	
17:24	TCPWM0_GRP2_OUT0_[0:7]	

Table 87 PERI Trigger destination group 11 (TCPWM1 Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0	TCPWM0_GRP0_CNT0_TR4_IN	Group [0], Counter[0], Trigger[4]
1	TCPWM0_GRP0_CNT1_TR4_IN	Group [0], Counter[1], Trigger[4]
2	TCPWM0_GRP0_CNT2_TR4_IN	Group [0], Counter[2], Trigger[4]
3	TCPWM0_GRP0_CNT3_TR4_IN	Group [0], Counter[3], Trigger[4]
4	TCPWM0_GRP0_CNT4_TR4_IN	Group [0], Counter[4], Trigger[4]
5	TCPWM0_GRP0_CNT5_TR4_IN	Group [0], Counter[5], Trigger[4]
6	TCPWM0_GRP0_CNT6_TR4_IN	Group [0], Counter[6], Trigger[4]
7	TCPWM0_GRP0_CNT7_TR4_IN	Group [0], Counter[7], Trigger[4]
8	TCPWM0_GRP1_CNT0_TR4_IN	Group [1], Counter[0], Trigger[4]
9	TCPWM0_GRP1_CNT1_TR4_IN	Group [1], Counter[1], Trigger[4]
10	TCPWM0_GRP1_CNT2_TR4_IN	Group [1], Counter[2], Trigger[4]
11	TCPWM0_GRP1_CNT3_TR4_IN	Group [1], Counter[3], Trigger[4]
12	TCPWM0_GRP1_CNT4_TR4_IN	Group [1], Counter[4], Trigger[4]
13	TCPWM0_GRP1_CNT5_TR4_IN	Group [1], Counter[5], Trigger[4]
14	TCPWM0_GRP1_CNT6_TR4_IN	Group [1], Counter[6], Trigger[4]
15	TCPWM0_GRP1_CNT7_TR4_IN	Group [1], Counter[7], Trigger[4]
16	TCPWM0_GRP2_CNT0_TR4_IN	Group [2], Counter[0], Trigger[4]
17	TCPWM0_GRP2_CNT1_TR4_IN	Group [2], Counter[1], Trigger[4]
18	TCPWM0_GRP2_CNT2_TR4_IN	Group [2], Counter[2], Trigger[4]
19	TCPWM0_GRP2_CNT3_TR4_IN	Group [2], Counter[3], Trigger[4]
20	TCPWM0_GRP2_CNT4_TR4_IN	Group [2], Counter[4], Trigger[4]
21	TCPWM0_GRP2_CNT5_TR4_IN	Group [2], Counter[5], Trigger[4]
22	TCPWM0_GRP2_CNT6_TR4_IN	Group [2], Counter[6], Trigger[4]
23	TCPWM0_GRP2_CNT7_TR4_IN	Group [2], Counter[7], Trigger[4]

Table 88 PERI Trigger source group 12 (TCPWM MOTIF Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	

(table continues...)

Table 88 (continued) PERI Trigger source group 12 (TCPWM MOTIF Trigger MUX)

Trigger MUX input	Trigger source label	Description
1	TCPWM0_GRP0_OUT0_[0]	
2	TCPWM0_GRP0_OUT1_[0]	
3	TCPWM0_GRP0_OUT0_[1]	
4	TCPWM0_GRP0_OUT1_[1]	
5	TCPWM0_GRP0_OUT0_[2]	
6	TCPWM0_GRP0_OUT1_[2]	
7	TCPWM0_GRP0_OUT0_[3]	
8	TCPWM0_GRP0_OUT1_[3]	
9	TCPWM0_GRP0_OUT0_[4]	
10	TCPWM0_GRP0_OUT1_[4]	
11	TCPWM0_GRP0_OUT0_[5]	
12	TCPWM0_GRP0_OUT1_[5]	
13	TCPWM0_GRP0_OUT0_[6]	
14	TCPWM0_GRP0_OUT1_[6]	
15	TCPWM0_GRP0_OUT0_[7]	
16	TCPWM0_GRP0_OUT1_[7]	
17	TCPWM0_GRP1_OUT0_[0]	
18	TCPWM0_GRP1_OUT1_[0]	
19	TCPWM0_GRP1_OUT0_[1]	
20	TCPWM0_GRP1_OUT1_[1]	
21	TCPWM0_GRP1_OUT0_[2]	
22	TCPWM0_GRP1_OUT1_[2]	
23	TCPWM0_GRP1_OUT0_[3]	
24	TCPWM0_GRP1_OUT1_[3]	
25	TCPWM0_GRP1_OUT0_[4]	
26	TCPWM0_GRP1_OUT1_[4]	
27	TCPWM0_GRP1_OUT0_[5]	
28	TCPWM0_GRP1_OUT1_[5]	
29	TCPWM0_GRP1_OUT0_[6]	
30	TCPWM0_GRP1_OUT1_[6]	
31	TCPWM0_GRP1_OUT0_[7]	
32	TCPWM0_GRP1_OUT1_[7]	
33	TCPWM0_GRP2_OUT0_[0]	

(table continues...)

Table 88 (continued) PERI Trigger source group 12 (TCPWM MOTIF Trigger MUX)

Trigger MUX input	Trigger source label	Description
34	TCPWM0_GRP2_OUT1_[0]	
35	TCPWM0_GRP2_OUT0_[1]	
36	TCPWM0_GRP2_OUT1_[1]	
37	TCPWM0_GRP2_OUT0_[2]	
38	TCPWM0_GRP2_OUT1_[2]	
39	TCPWM0_GRP2_OUT0_[3]	
40	TCPWM0_GRP2_OUT1_[3]	
41	TCPWM0_GRP2_OUT0_[4]	
42	TCPWM0_GRP2_OUT1_[4]	
43	TCPWM0_GRP2_OUT0_[5]	
44	TCPWM0_GRP2_OUT1_[5]	
45	TCPWM0_GRP2_OUT0_[6]	
46	TCPWM0_GRP2_OUT1_[6]	
47	TCPWM0_GRP2_OUT0_[7]	
48	TCPWM0_GRP2_OUT1_[7]	
49:116	HSIOM_TR_OUT[0:67]	Triggers connected to pins in HSIOM (peri.tr_in)
117:132	PASS_TR_LEVEL[0:15]	Triggers (LEVEL) for MC algo

Table 89 PERI Trigger destination group 12 (TCPWM MOTIF Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:7	TCPWM0_MOTIF_GRP1_TR_IN[0:7]	MOTIF Trigger Group #1 MOTIF #0
8:15	TCPWM0_MOTIF_GRP2_TR_IN[0:7]	MOTIF Trigger Group #2 MOTIF #0

Table 90 PERI Trigger source group 13 (HPPASS Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1:8	TCPWM0_GRP0_OUT1_[0:7]	32-bit TCPWM counters
9:16	TCPWM0_GRP1_OUT1_[0:7]	
17:24	TCPWM0_GRP2_OUT1_[0:7]	
25:32	TCPWM0_MOTIF_GRP1_TR_OUT[0:7]	MOTIF Trigger Group #1 MOTIF #0
33:40	TCPWM0_MOTIF_GRP2_TR_OUT[0:7]	MOTIF Trigger Group #2 MOTIF #0
41:108	HSIOM_TR_OUT[0:67]	Triggers connected to pins in HSIOM (peri.tr_in)

Table 91 PERI Trigger destination group 13 (HPPASS Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:15	PASS_TR_A_IN[0:15]	PASS trigger muxed inputs
16:31	PASS_TR_B_IN[0:15]	PASS trigger muxed inputs

Table 92 PERI Trigger source group 14 (USBFS Trigger MUX)

Trigger MUX input	Trigger source label	Description
0	ZERO	
1:16	PDMA0_TR_OUT[0:15]	P-DMA0 tr_out
17:32	PDMA1_TR_OUT[0:15]	P-DMA1 tr_out

Table 93 PERI Trigger destination group 14 (USBFS Trigger MUX)

Trigger MUX output	Trigger destination label	Description
0:7	USB_DMA_BURSTEND[0:7]	USB trigger muxed inputs

The Trigger 1-to-1 connections for PSC3M/P6 MCU can be found in the tables below:

Table 94 PERI Trigger 1-to-1 group 0 (CAN DW0 triggers - from DW back to CAN)

Input	Label	Description
0	PDMA0_TR_OUT[0]_ACK_TO_CAN_0	CAN DW acknowledge
1	PDMA0_TR_OUT[0]_ACK_TO_CAN_1	CAN DW acknowledge

10.4 Software triggers

All input and output signals to a trigger multiplexer can be triggered by software. This is accomplished by writing into the PERI_TR_CMD register. This register allows you to trigger the corresponding signal for a number of peripheral clock cycles.

10.5 Trigger functionality

Figure 28 gives an overview of a multiplexer trigger group.

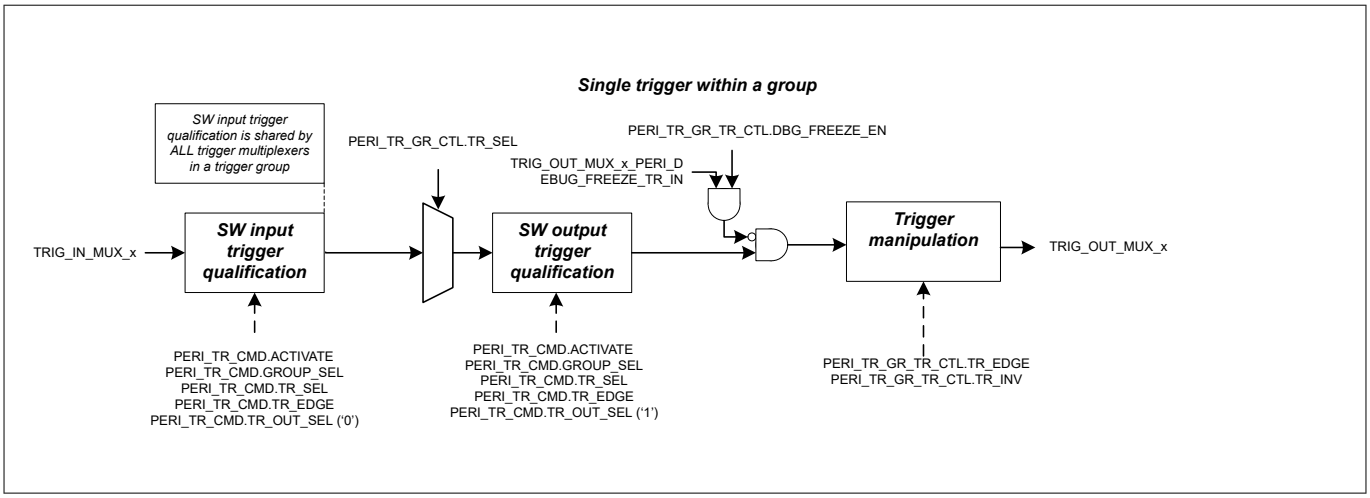


Figure 28 Multiplexer Trigger group

Group trigger configuration:

PERI[Peri Group Number]_TR_GR[Group Number].PERI_TR_GR_TR_CTL[Trigger Number].TR_SEL = Input trigger to be connected to

PERI[Peri Group Number]_TR_GR[Group Number].PERI_TR_GR_TR_CTL[Trigger Number].TR_INV = Invert the trigger or not

PERI[Peri Group Number]_TR_GR[Group Number].PERI_TR_GR_TR_CTL[Trigger Number].TR_EDGE = Edge or level-sensitive trigger

Activating the group trigger via software

PERI[Peri Group Number]_TR_CMD.TR_SEL = Input of Trigger Number

PERI[Peri Group Number]_TR_CMD.GROUP_SEL = Trigger Number Group_Nr

PERI[Peri Group Number]_TR_CMD.TR_EDGE = Edge or level-sensitive trigger

PERI[Peri Group Number]_TR_CMD.OUT_SEL = 1 for output trigger and 0 for input trigger

PERI[Peri Group Number]_TR_CMD.ACTIVATE = 1 signifies configured trigger activation

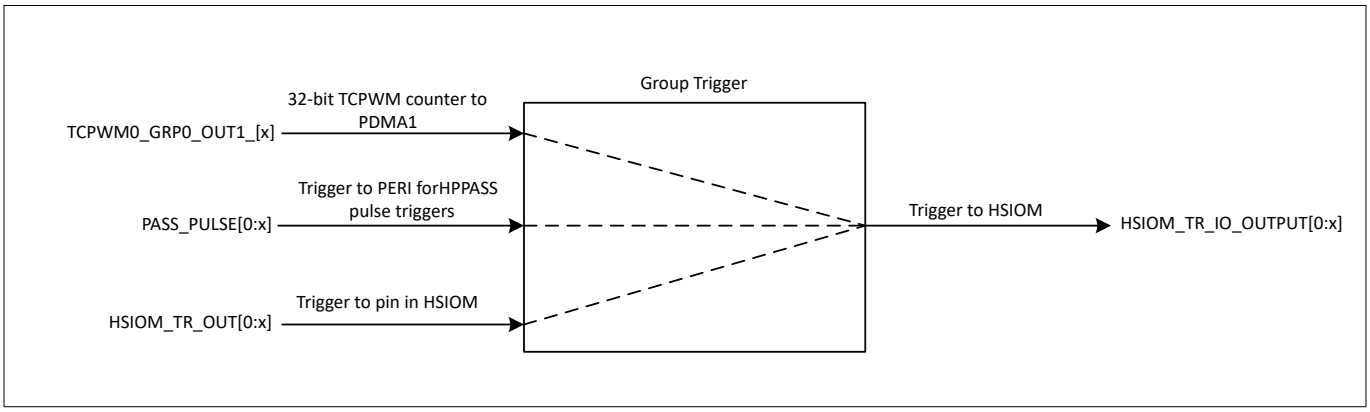


Figure 29 Example of Group Trigger

Figure 30 gives an overview of a 1-to-1 group.

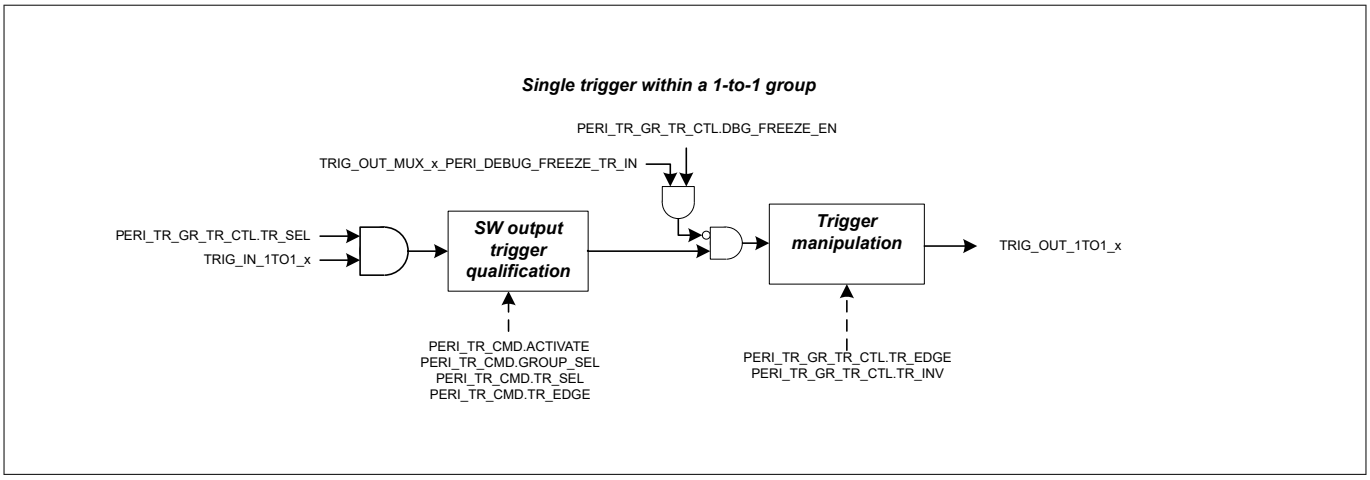


Figure 30 1-to-1 Trigger group

1TO1 trigger configuration:

PERI[Peri Group Number]_TR_1TO1_GR[Group Number]. PERITR_1TO1_GR_TR_CTL[Trigger Number].TR_SEL = True (input trigger) or False (constant signal level '0')

PERI[Peri Group Number]_TR_1TO1_GR[Group Number]. PERI_TR_1TO1_GR_TR_CTL[Trigger Number].TR_INV = Invert the output trigger or not

PERI[Peri Group Number]_TR_1TO1_GR[Group Number]. PERI_TR_1TO1_GR_TR_CTL[Trigger Number].TR_EDGE = Edge or level-sensitive trigger

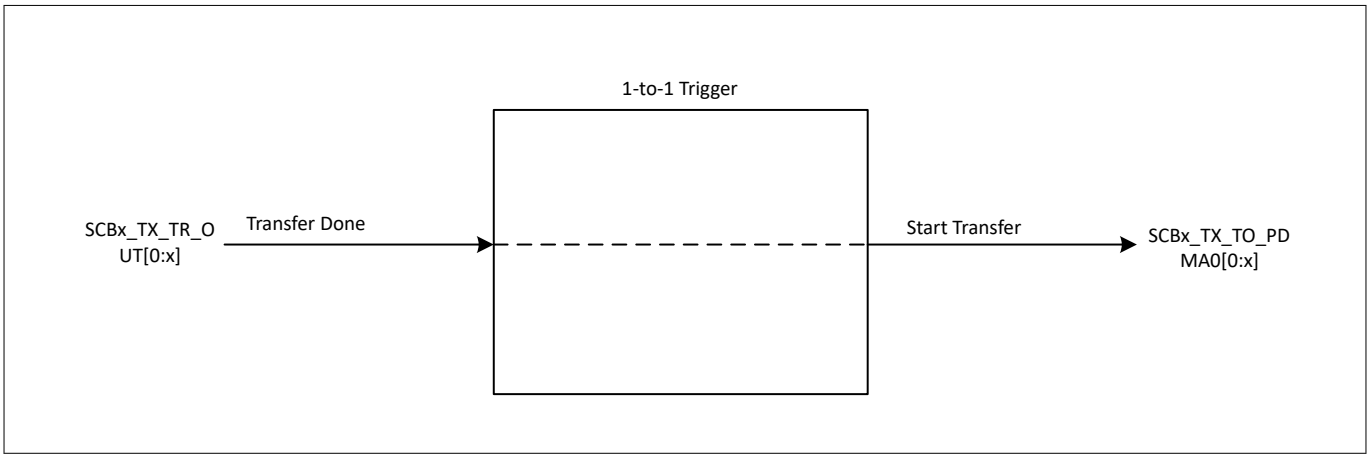


Figure 31 Example of 1-to-1 Trigger

As mentioned, the trigger manipulation logic provides asynchronous edge detection logic. The trigger manipulation is detailed in Figure 32.

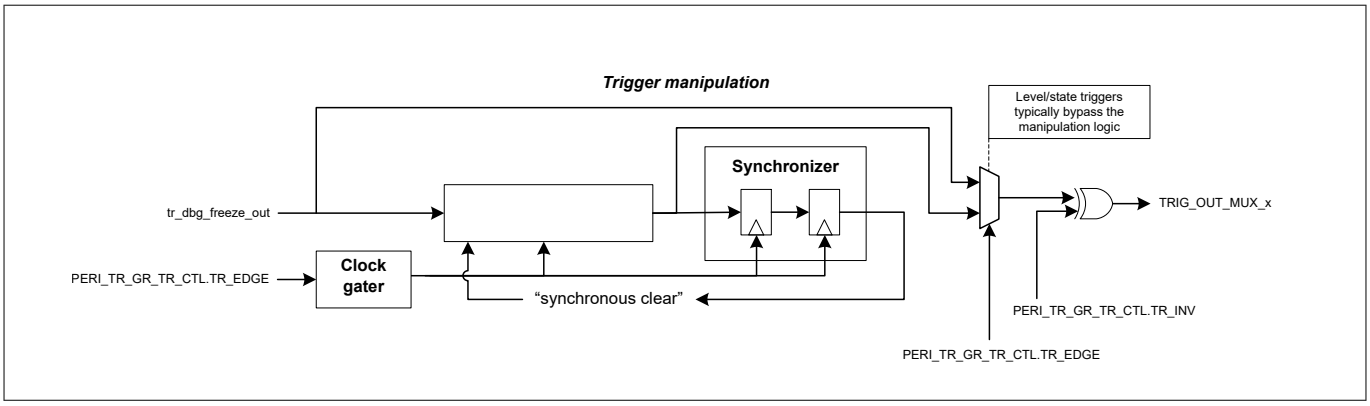


Figure 32 Trigger manipulation

In addition, there are I/Os (TRIG_IN[0:x]) that can be used to generate triggers. These inputs can be used to trigger TCPWM, SAR ADC, PERI, and CPU-CTI. For device-specific Trigger MUX assignments, refer to the device datasheet.

***Note**: There may be multiple combinations of triggers that may lead to a collision in the event that both triggers are enabled simultaneously.*

10.6 Registers summary

Table 95 Trigger multiplexer registers

Symbol	Name	Description
PERI[i]_TR_CMD	Trigger command Register	Control enables the software activation of a specific input trigger or output trigger of the trigger multiplexer structure
PERI[i]_TR_GR[x]_TR_CTL[y]	Trigger Group Control Register	Controls group trigger actions and connects a peripheral input trigger to multiple peripheral output triggers It can also invert the signal and specify if the output signal should be treated as a level-sensitive or edge-sensitive trigger. Specifies if the output trigger is blocked in debug mode. When set high, tr_dbg_freeze (bit 12) will block the output trigger generation. In the register format, 'i' is the peri group, 'x' is the trigger group, and 'y' is the output trigger line number from the multiplexer.

(table continues...)

Table 95 (continued) Trigger multiplexer registers

PERI[i]_TR_1TO1_GR[x]_TR_CTL[y]	Trigger 1-to-1 Group Control Register	Controls the 1-to-1 trigger actions and connects a peripheral input trigger to a specific peripheral output trigger. It also includes trigger manipulation that can invert the signal and specify if the output signal should be treated as a level-sensitive or edge-sensitive trigger. Specifies if the output trigger is blocked in debug mode. When set high, tr_dbg_freeze (bit 12) will block the output trigger generation. In the register format, 'i' is the peri group, 'x' is the trigger group, and 'y' is the output trigger line number from the multiplexer.

11 Debug and Trace system

The PSC3M/P6 MCU program and debug interface provides a communication gateway for an external device to perform programming or debugging. The external device can be an Infineon-supplied programmer and debugger, or a third-party device that supports programming and debugging. The serial wire debug (SWD) or the Joint Test Action Group (JTAG) interface can be used as the communication protocol between the external device and the PSC3M/P6 MCU.

11.1 Features

The program and debug interface has the following features:

- Supports programming and debug through the JTAG or SWD interface
- CM33 supports 8-bit Instrumentation Trace Macrocell (ITM) and 4-bit Embedded Trace Macrocell (ETM) tracing
- Supports Cross Triggering Interface (CTI) and Cross Triggering Matrix (CTM)
- CM33 supports eight hardware breakpoints and four watch-points
- Provides read and write access to all memory and registers in the system while debugging, including the CM33 register banks when the core is running or halted

11.2 Architecture

Figure 33 shows the block diagram of the program and debug interface in the PSC3M/P6 MCU. The debug and access port (DAP) acts as the program and debug interface. The external programmer or debugger, also known as the “host”, communicates with the DAP of the PSC3M/P6 MCU “target” using either the Serial Wire Debug (SWD) or Joint Test Action Group (JTAG) interfaces. The debug physical port pins communicate with the DAP through the high-speed I/O matrix (HSIOM). See the I/O subsystem chapter for details on the HSIOM.

The debug infrastructure is organized in the following groups:

- DAP (provides pin interfaces through which the debug host can connect to the target)
- Cortex®-M33 core debug components
- Other debug infrastructure (includes CTM, the TPIU, and the system ROM table)

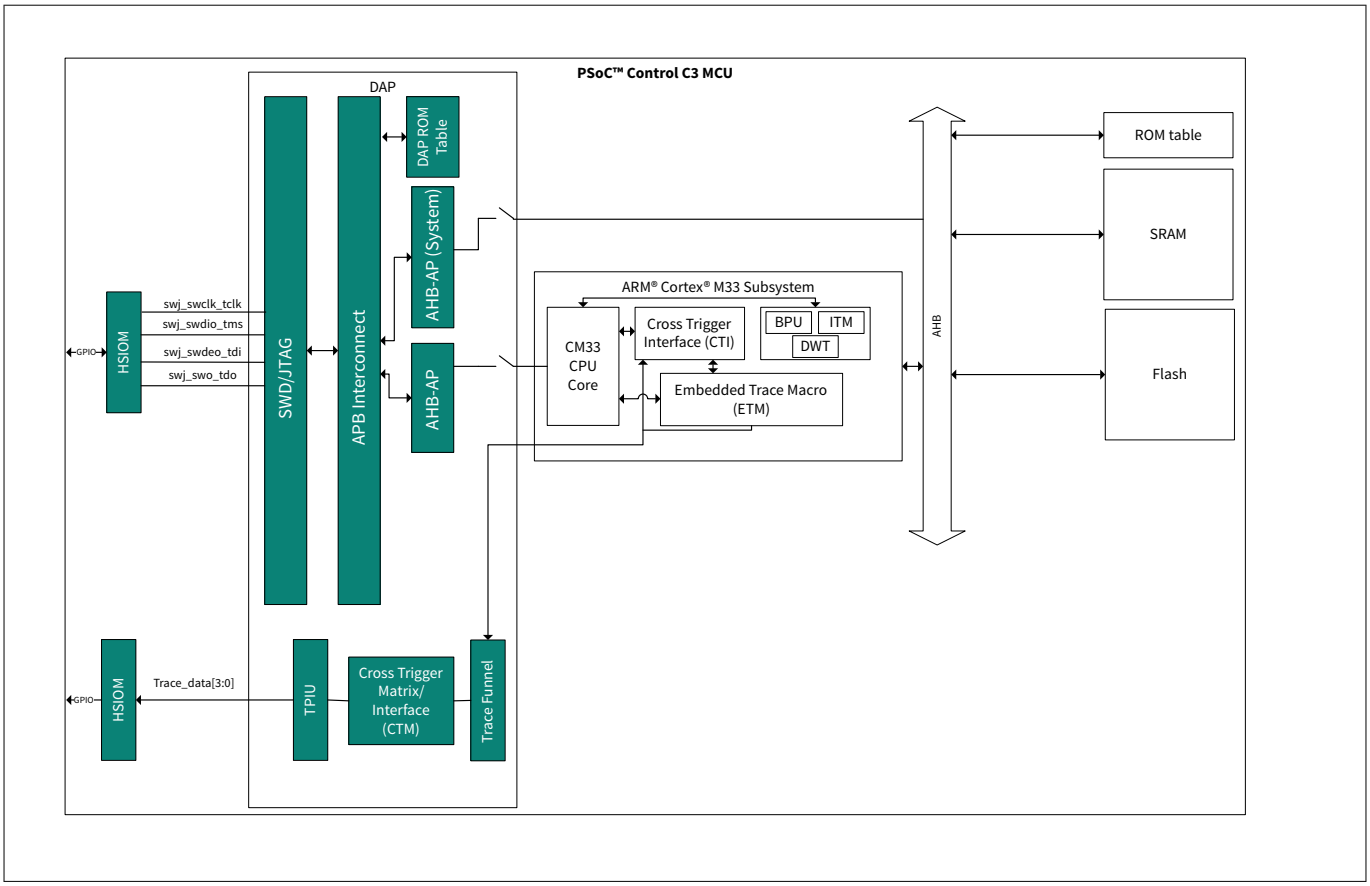


Figure 33 Program and debug architecture

The DAP communicates with the core using the Arm®-specified advanced high-performance bus (AHB) interface. The external host can effectively take control of the entire device through the DAP to perform both programming and debugging operations, the DAP can be disabled in production devices for security.

The following are the various debug and trace components:

- Debug components
  - JTAG and SWD for debug control and access
- Trace source components
- Trace sink components
- Trace port interface unit (TPIU) to drive the trace information out of the chip to an external trace port analyzer
- Cross-triggering components
  - Cross-trigger interface (CTI)
  - Cross-trigger matrix (CTM)

11.2.1 Debug access port (DAP)

The DAP consists of a combined SWD/JTAG interface (SWJ). The SWJ includes a SWD listener that decides whether the JTAG interface (default after reset) or SWD interface is active. The JTAG and SWD interfaces are mutually exclusive because they share pins.

The debug port (DP) communicates with debug components using the Advanced Peripheral Bus Interconnect (APBIC) infrastructure, which then connects to one of the two AHB Access ports:

- The Cortex® -M33 AP connects directly to the AHB debug port and gives access to the Cortex® -M33 internal debug components. This allows access to the rest of the system through the Cortex® -M33 AHB master

interface. The AP provides the debug host the same view as an application running on Cortex® -M33. This includes access to the MMIO of other debug components in the Cortex® -M33 subsystem. These debug components can be accessed by the Cortex® -M33 CPU, but cannot be reached through the other APs or by the Cortex® -M33 core.

- The System-AP, gives access to the rest of the system, such as user SRAM and Flash, through the AHB mux. This allows access to the System ROM table, which cannot be reached any other way. The system ROM table provides the chip ID, but is otherwise empty.

***Note**: Both the debug and access port registers are Arm® IPs, which can be found in the Arm® Debug Interface Architecture Specification ADIv6.0.*

***Note**: The DAP runs asynchronously with the Cortex® M33.*

11.2.1.1 DAP power domain

The entire debugger uses the same power domain as the Cortex® -M33 subsystem. Almost all debug components are part of the Active power domain. The only exception is the SWD as it uses two pins, SWCLKTCK and SWDIOTMS, which are operational in Deep Sleep. This allows for the SWD debug host to connect during Deep Sleep, while the application is running, or while powered down.

After the debugger is connected to the chip, it must bring the chip to the Active state before any operation. For this, the SWD/JTAG-DP has a register (DP_CTL_STAT) with two power request bits: CDBGPWRUPREQ and CSYSPWRUPREQ. These bits request for debug power and system power, respectively. These bits must remain set for the duration of the debug session.

The JTAG pins are only operational in Active mode, which means that the JTAG debug and JTAG boundary scan are not available when the system is in Deep Sleep. JTAG functionality is available only after a chip power-on reset.

Move the chip to the Active state before any operation. When the debugger attaches, it switches the CM33 into the active power domain. All other resources must be turned to an Active state by authenticated or debugging software.

11.2.1.2 DAP security

The DAP pins are controlled by the policy file. A device from the factory with a standard policy file sets up the DAP pins so that a SWD or JTAG interface can be connected without configuration. When using a secure policy file, configure the DAP pins and enable them in the secure policy file. For more information on the secure policy file, see the Secure boot section.

Cortex® -M33 has its own CPUSS_AP_CTL register where the debug features, such as invasive/non-invasive debug and secure privileged invasive/non-invasive debug, can be configured. These debug features are configured in the policy file, which is used by the secure enclave to set the AP_CTL registers in the protected region of Flash. The SYS_AP is controlled by the Cortex® M33 AP_CTL register and is enabled by setting AP_CTL.SYS_DISABLE to ‘00’ in the policy file.

All AP_CTL policy field names that end with _DISABLE are two bits, with ‘00’ being enable, ‘01’ being disable, and ‘1x’ being permanently disabled. For any secure/non-secure debug features to be enabled, AP_CTL.CM33_0_DISABLE must be set to ‘00’.

The following table shows the register configuration in the policy file for Cortex® -M33 for non-secure debugging.

Table 96 Non-secure debug

Debug mode	AP_CTL.CM33_0_DBG_DISABLE	AP_CTL.CM33_0_NID_DISABLE
Invasive	00	01
Non-Invasive	01	00

For secure debug features, set the following in the policy file for Cortex® -M33

Table 97 Cortex® -M33 Secure Debug

Debug mode	AP_CTL.CM33_0_SPID_DISABLE	AP_CTL.CM33_0_SPNID_DISABLE	AP_CTL.CM33_0_SECURE_DISABLE
Secure Invasive	00	01	00
Secure Non-Invasive	01	00	00

***Note**: To enable Secure Invasive Debug, set CM33_0_DBG_DISABLE to '00'. To enable Secure Non-Invasive debug, set CM33_0_NID_DISABLE to '00'*

11.2.2 ROM tables

The CoreSight architecture allows for discoverable debug topology. This means that a debug host can find all debug components available on a chip without knowing the identity of the chip. To discover the chip identity and the debug topology, use the CoreSight ROM tables.

The ROM tables are organized in a tree hierarchy. Each AP has a register that contains a 32-bit address pointing to the base of the root ROM table for that AP. The root ROM table in the debugger is the 'Dap ROM Table', which points to the different APs. The Coresight DP component has a register called BASEPTR that points to the DAP ROM table. For PSC3M/P6 MCU there are three such root ROM tables (DAP, M33, and System).

Each ROM table contains 32-bit entries that have an address pointer that either points to the base of the next level ROM table or the base of a leaf debug component. Each ROM table also contains a set of ID registers that hold JEDEC compliant identifiers to identify the manufacturer, part number, and major and minor revision number. For all ROM tables in PSC3M/P6 MCU, these IDs are the same. Each ROM table and CoreSight compliant component also contains component identification registers.

11.2.3 Trace

The Cortex® -M33 include Instrumentation Trace Macrocell (ITM) and Embedded Trace Macrocell (ETM). These components capture the program execution flow from the CPUs and generate trace outputs on the Advanced Trace Bus (ATB). These trace outputs are passed to the core-sight trace funnel (CSTF), which combines multiple trace streams on to a single bus.

Built into the debugger is a 4-bit Trace Port Interface Unit (TPIU). This is used to drive external pins through the IOSS interface, so that the trace can be captured by an external trace port analyzer (TPA). For more information, refer to the Arm® Debug Interface Architecture Specification ADIv6.0.

11.2.4 Embedded cross triggering

Arm® CoreSight includes Embedded Cross Triggering (ECT) that provides the ability to trigger (send or receive) events between different debug components. For example, you can use trigger events to:

- Start or stop CPU
- Start or stop instruction tracing based on the trace buffer being full, not full, or based on other events

CoreSight uses two main components to support ECT: the Cross-Trigger Interface (CTI) and Cross Trigger Matrix (CTM). The CTI interfaces with other debug components, sending triggers back and forth and synchronizing them as needed. The CTM connects several CTIs, which allows events to be communicated from one CTI to another.

The debug infrastructure provides a CTI that primarily connects events to the TPIU and system triggers.

Figure 34 shows overall cross trigger functionality.

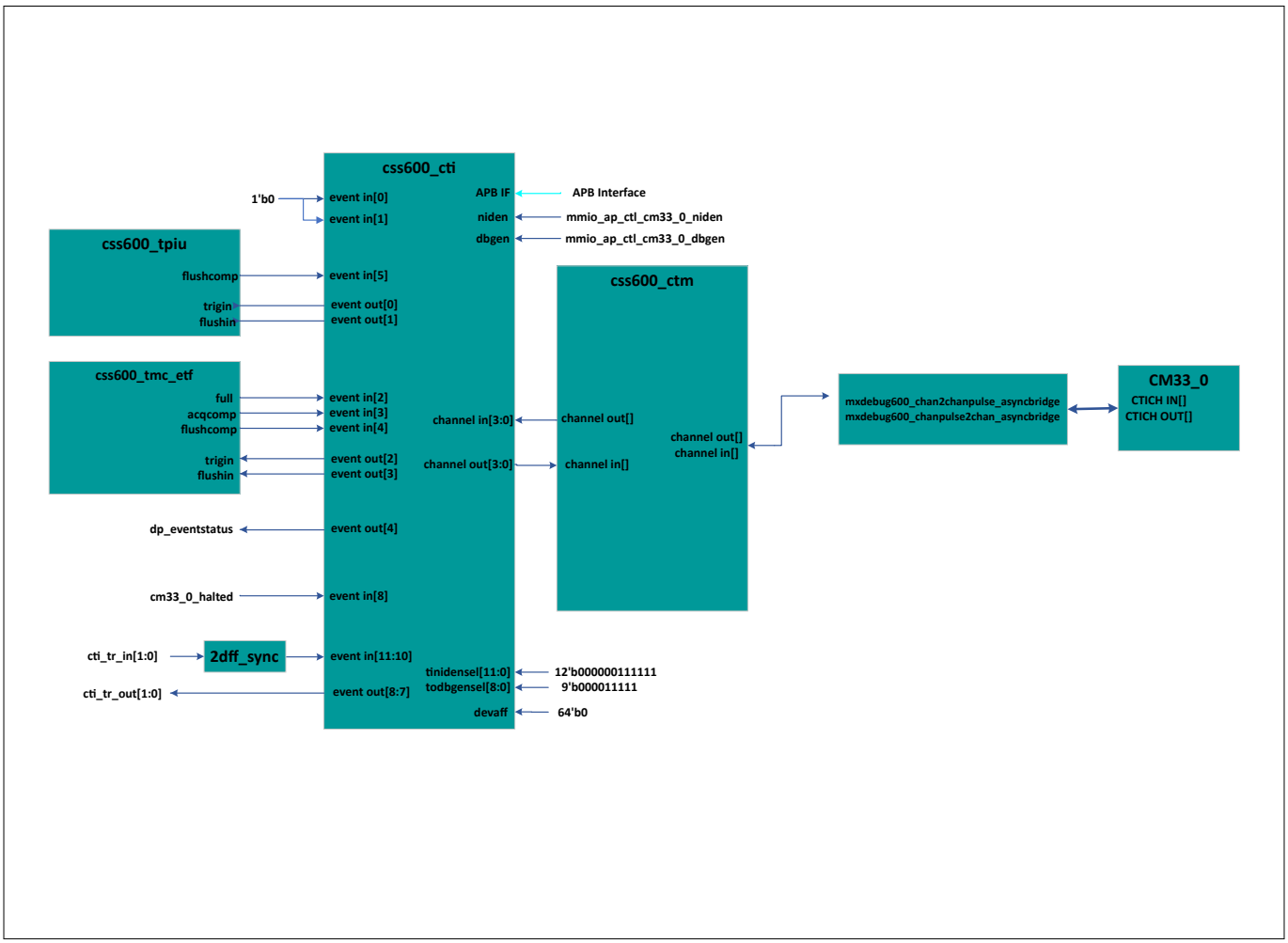


Figure 34 Cross Trigger Functionality

The EVENT_IN_LEVEL, SW_HANDSHAKE, “tinidensel[]” and “todbgensel[]” constants corresponding to “dp_eventstatus”, TPIU and ETF triggers are set in accordance with the CoreSight SoC-600 configuration and integration manual.

The trigger inputs “cm33_0_halted” are treated as level trigger inputs and the EVENT_IN_LEVEL parameter for these triggers is set as ‘1’. The level halted signals of this CTI are used to implement debug freeze functionality in peripherals (like TCPWMs etc).

The “cti_tr_in[]” and “cti_tr_out[]” triggers treatment requires system level verification. SW_HANDSHAKE[] constant corresponding to “cti_tr_out[]” is set to ‘0’ and this must be verified at system level, with potential adjustments made based on feedback from the verification process.

Refer Cortex®-CM33 TRM for the details about CM33 CTI connections to the respective debug components within CM33.

11.2.4.1 Freeze Peripherals in Debug Mode

The level halted signals of the CTI are used to implement debug freeze functionality in peripherals (like TCPWMs etc).

To configure the system for freezing TCPWM in debug mode, follow the steps given below:

- Make sure that the debug session is active by checking the DEBUG_SESSION bit of the PWR_CTL register before enabling the freeze and suspend function

  ***Note**: The freeze and suspend function can only be enabled when a debug session is active.*

- To implement the stop function using the CM33_0 HALT action, perform the following steps:

  1. Set up EVENT(8) for the CM33_0 HALT action
  2. Enable the cross triggering mapping logic functionality using the CTI control register
  3. Enable a cross trigger event to the corresponding channel using the CTI trigger to channel enable register
  4. Enable a channel event for the corresponding channel to generate a CTITRIGOUT output using the CTI channel to trigger enable register

  ***Note**: Direct connection of CM33_0 HALT to cti_tr_out[1:0] or TCPWM is not supported.*

- Use the trigger mux function to connect the CTITRIGOUT output to the tr_debug_freeze input of the TCPWM, as described in the TriggersMux chapter. This configuration enables the routing of cti_tr_out[0:1] to TCPWM for trigger-to-freeze operation

The following code snippet can be referred:

```
#define TRC_CTICONTROL (*((volatile uint32_t*)(0xE0080000)))
#define TRC_CTIINEN(n) (*((volatile uint32_t*)(0xE0080020 + 4*n)))
#define TRC_CTIOUTEN(n) (*((volatile uint32_t*)(0xE00800A0 + 4*n)))

/*Check if debug session active throug*/
if(1U == _FLD2VAL(SRSS_PWR_CTL_DEBUG_SESSION, SRSS_PWR_CTL))
{

  /* CTI Configuration */
  TRC_CTICONTROL = 0x1;
  TRC_CTIINEN(8) = 0x1;
  TRC_CTIOUTEN(7) = 0x1;

  /*Connect the debug-freeze signal*/
  Cy_TrigMux_Connect(TRIG_IN_MUX_8_CTI_TR_OUT0, TRIG_OUT_MUX_8_TCPWM_DEBUG_FREEZE_TR_IN, false,
TRIGGER_TYPE_LEVEL);
}
```

11.3 Serial wire debug (SWD) interface

The PSC3M/P6 MCU supports programming and debugging through the SWD interface. The SWD protocol is a packet-based serial transaction protocol. At the pin level, it uses a single bidirectional data signal (SWDIO) and an unidirectional clock signal (SWDCK). The host programmer always drives the clock line, whereas either the host or the target drives the data line. A complete data transfer (one SWD packet) consists of three phases:

- Host packet request phase – The host issues a request to the PSC3M/P6 MCU target

- Target acknowledge response phase – The PSC3M/P6 MCU target sends an acknowledgement to the host
- Data transfer phase – The host or target writes data to the bus, depending on the direction of the transfer

For more information on the SWD implementation, refer to Arm® Debug Interface Architecture Specification ADIv6.0.

11.4 JTAG interface

In response to higher pin densities on ICs, the Joint Test Action Group (JTAG) proposed a method to test circuit boards by controlling the pins on the ICs (and reading their values) via a separate test interface. The solution, later formalized as IEEE Standard 1149.1-2001, is based on the concept of a serial shift register routed across all of the pins of the IC – hence the name “boundary scan.” The circuitry at each pin is supplemented with a multipurpose element called a boundary scan cell. In thePSC3M/P6 MCU, most GPIO port pins have a boundary scan cell associated with them (see the GPIO block diagrams in the I/O subsystem section). The interface used to control the values in the boundary scan cells is called the Test Access Port (TAP) and is commonly known as the JTAG interface. It consists of three signals: Test Data In (TDI), Test Data Out (TDO), and Test Mode Select (TMS). A clock signal (TCK) that clocks the other signals is also included and a debug TAP reset (TRST) is also present which is an optional input signal. TDI, TMS, and TCK are all inputs to the device and TDO is the output from the device. For more information on the JTAG interface, refer to Arm® Debug Interface Architecture Specification ADIv6.0.

11.5 Programming the MCU

The PSC3M/P6 MCU is programmed using the following sequence:

1. Acquire the SWD port in the PSC3M/P6 MCU
2. Enter the programming mode
3. Execute the device programming routines such as Silicon ID Check, Flash Programming, Flash Verification, and Check-sum Verification

***Note**: For detail, see PSOC™ Control C3 MCU Programming specification.*

11.5.1 SWD port acquisition

When the device is reset by asserting the external reset (XRES) pin, the DP defaults to the JTAG operation. To switch to the SWD protocol, reset the device using XRES to put it into its reset state. The host must send the following SWD connect sequence:

1. Set SWDIO HIGH for at least five SWCLK cycles. This places the JTAG TAP state machine into the Test Logic Reset state and selects the IDCODE instruction
2. Change the device from the JTAG to dormant state by writing the 31-bit sequence (0x2EEEEEE6, MSB first) on SWDIO. The device is now in a dormant state
3. Set SWDIO HIGH for at least eight SWCLK cycles. This ensures that the device is not in the middle of a Selection Alert sequence
4. On SWDIO, send the 128-bit Selection Alert sequence (0x49CF9046 A9B4A161 97F5BBC7 45703D98, MSB first)
5. Set SWDIO LOW for four SWCLK cycles
6. On SWDIO, send the required activation code sequence (0b0101_1000, MSB first)
7. Place the target into a known state by setting SWDIO HIGH for at least 50 SWCLK cycles. This makes sure the SWD interface is in the line reset state. The device is now in SWD configuration. The PSC3M/P6 MCU uses Dormant operation to move between JTAG to Dormant, and then Dormant to SWD

For more information about this operation, refer to the Arm® Debug Interface Architecture Specification ADIv6.0.

11.5.2 SWD programming mode entry

After the SWD port is acquired, the host must enter the device programming mode within a specific time window. This is done by setting the TEST_MODE bit (bit 31) in the test mode control register (MODE register). Make sure to configure the debug port before entering the device programming mode.

11.5.3 SWD programming routine executions

When the device is in programming mode, the external programmer can start sending the SWD packet sequence for performing programming operations such as flash erase, flash program, checksum verification, and so on.

11.6 PSC3M/P6 MCU SWD debug interface

The Cortex® -M33 DAP debugging features are classified into two types: invasive debugging and noninvasive debugging. Invasive debugging includes program halting and stepping, breakpoints, and data watch-points. Noninvasive debugging includes instruction address profiling and device memory access, which includes the flash memory, SRAM, and other peripheral registers.

The DAP has three major debug subsystems:

- Debug Control and Configuration registers
- Breakpoint Unit (BPU) – Provides breakpoint support
- Debug Watchpoint (DWT) – Provides watchpoint support

See the Armv8-M architecture reference manual for complete details on the debug architecture.

***Note**: The BPU is part of the Flashpoint and Breakpoint (FPB) unit in Arm® Debug Interface Architecture Specification ADIv6.0.*

11.6.1 Debug control and configuration registers

The debug control and configuration registers are used to execute firmware debugging. The registers and their key functions are as follows (see the Armv8-M architecture reference manual for the complete bit-level definitions of these registers):

- **Debug Halting Control and Status Register (DHCSR):** This register contains the control bits to enable debug, halt the CPU, and perform a single-step operation. It also includes status bits for the debug state of the processor.
- **Debug Fault Status Register (DFSR):** This register describes the reason a debug event occurs and includes debug events, which are caused by a CPU halt, breakpoint event, or watchpoint event.
- **Debug Core Register Data Register (DCRDR):** This register is used to store the data to write to or read.
- **Debug Exception and Monitor Control Register (DEMCR):** This register contains the enable bits for global debug watchpoint (DWT) block enable, reset vector catch, and hard fault exception catch.

11.6.2 Break point units (BPU)

The BPU provides breakpoint functionality on instruction fetches. The Cortex® -M33 in the PSC3M/P6 MCU supports eight hardware breakpoints. Along with the hardware breakpoints, any number of software breakpoints can be created by using the BKPT instruction. The BPU has the following two types of registers:

- The breakpoint control register (FPB_CTRL) is used to enable the BPU and store the number of hardware breakpoints supported by the debug system.
- Each hardware breakpoint has a Breakpoint Compare Register (FPB_COMPx). It contains the enable bit for the breakpoint and the compare address value. The typical use case is that when an instruction fetch address matches the compare address of a breakpoint, a breakpoint event is generated and the processor is halted.

11.6.3 Data watchpoints (DWT)

The DWT provides watchpoint support on a data address access or a program counter (PC) instruction address.

The DWT supports two watchpoints for the Cortex® -M33. It also provides an external program counter sampling using a PC sample register, which can be used for noninvasive coarse profiling of the program counter.

- The watchpoint compare (DWT_COMPx) registers store the compare values that are used by the watchpoint comparator for the generation of watchpoint events. Each watchpoint has an associated DWT_COMPx register.
- The watchpoint function (DWT_FUNCTIONx) registers store the conditions that trigger the watchpoint events. They may be program counter watchpoint event or data address read/write access watchpoint events. A status bit is also set when the associated watchpoint event occurs.
- The watchpoint comparator PC sample register (DWT_PCSR) stores the current value of the program counter. This register is used for coarse, non-invasive profiling of the program counter register.

11.6.4 Debugging the PSC3M/P6 MCU device

The host debugs the target by accessing the debug control and configuration registers, BPU registers, and DWT registers. All registers are accessed through the SWD interface; in the Cortex® -M33, DAP converts the SWD packets to appropriate register access through the DAP-AHB interface.

The first step in debugging the target is to acquire the SWD port. The acquire sequence can be found in the SWD port acquisition section. The SWD port is acquired when the correct Cortex® -M33 DAP SWDID is read from the target device. For the debug transactions to occur on the SWD interface, do not use the corresponding pins for any other purpose. See the I/O subsystem section to understand how to configure the SWD port pins, allowing them to be used only for the SWD interface or for other functions such as LCD and GPIO. If debugging is required, do not use the SWD port pins for other purposes.

If only programming support is needed, the SWD pins can be used for other purposes. When the SWD port is acquired, the external debugger sets the C_DEBUGEN bit in the DHCSR register to enable debugging. Then, the different debugging operations such as stepping, halting, breakpoint configuration, and watchpoint configuration are carried out by writing to the appropriate registers in the debug system.

11.7 Registers summary

Table 98 Debug Features Registers

Name	Description
CPUSS_AP_CTL	Configure debug features

12 Cryptographic accelerator

The cryptographic accelerator block provides hardware implementation and acceleration of cryptographic functions. Implementation in hardware takes less time and energy than the equivalent firmware implementation. In addition, the block provides True Random Number Generation (TRNG) functionality. The PSC3M/P6 MCU cryptographic accelerator block name is CryptoLite.

12.1 Features

The following are the features of the cryptographic accelerator:

- Advanced Encryption Standard (AES)-128[1)] to accelerate block cipher functionality and only supports the encryption functionality of a single 128-bit block with a fixed 128-bit key
- Secure Hash Algorithm (SHA)-256/384/512 to accelerate hash functionality
- True random number generator (TRNG)
- Vector unit (VU) to support asymmetric key cryptography, such as RSA and Elliptic-Curve Cryptography (ECC)

12.2 Architecture

The following figure gives an overview of the CryptoLite block.

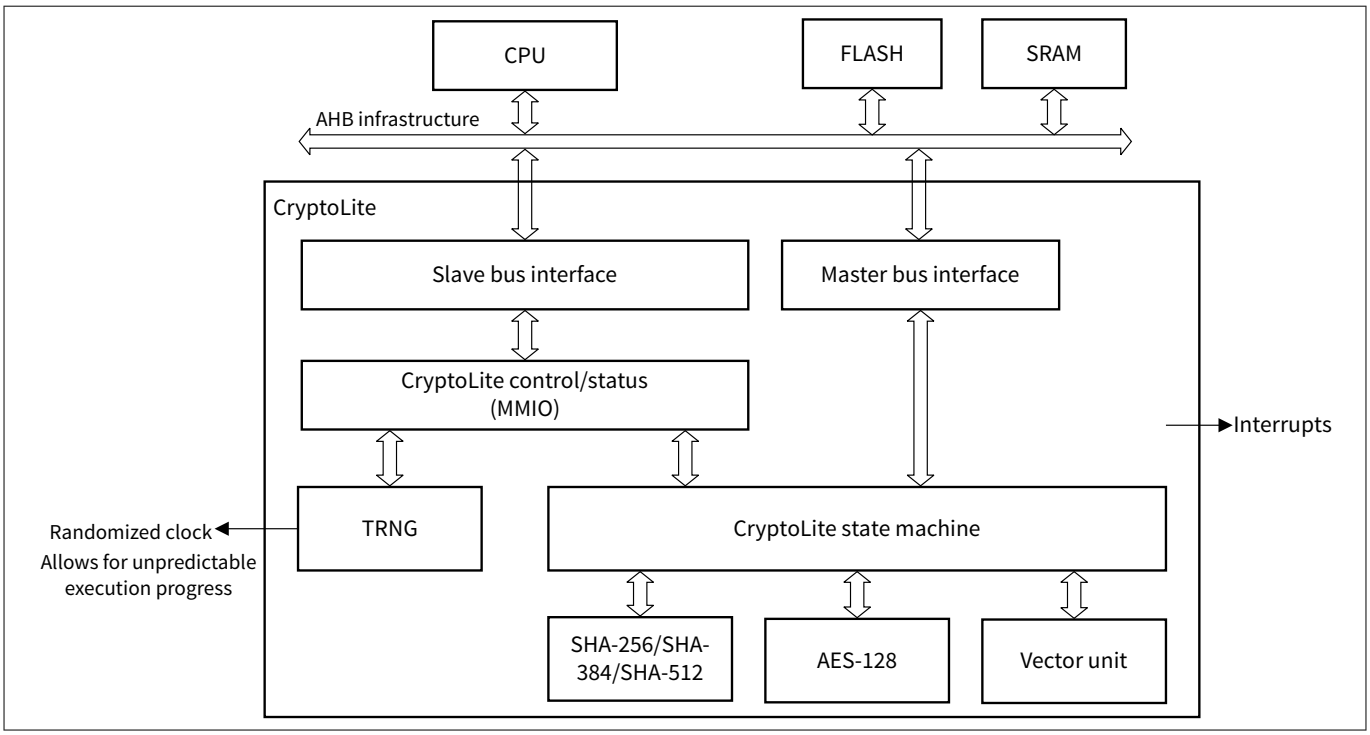


Figure 35 CryptoLite block diagram

The CryptoLite block has the following interfaces:

- AHB slave interface: External bus masters use this interface to access the block MMIO registers
- AHB master interface: The block uses this interface to access hardware function data structures and operands in memories (FLASH and SRAM)

1 The device also includes a dedicated AES block supporting both 128-bit and 256-bit AES, with additional security features. See Advanced Encryption Standard (AES) chapter for the details.

- Two interrupts: One interrupt for regular behavior, for example, TRNG is initialized. One interrupt for irregular/erroneous behavior, for example, when the AHB master interface encounters a bus error
- Randomized clock: This is the TRNG digitized analog signal (DAS) bit-stream sequence
- A clock and reset signal interface connects to the System Resources subsystem (SRSS)

The CryptoLite block provides hardware acceleration for a minimal set of cryptographic functions. The functionality is implemented by four components:

- AES-128 component: To accelerate block cipher functionality. The component supports:
  - Forward encryption of a single 128-bit block with a 128-bit key
- SHA component: To accelerate hash functionality. The component supports:
  - For SHA-256: Message schedule calculation and processing for a 512-bit message chunk
  - For SHA-384/SHA-512: Message schedule calculation and processing for a 1024-bit message chunk
- Vector unit (VU) component: To accelerate asymmetric key cryptography (for example, RSA and ECC). The component supports:
  - Large integer multiplication, addition, and so on
- TRNG component: Based on a set of ring oscillators, the TRNG includes a hardware health monitor. This component generates true random numbers using up to six ring oscillators

The component provides speeds up, for example, secure boot Digital Signature Authentication (DSA or ECDSA) through its SHA and VU hardware acceleration. The digital signature provides message authentication, integrity, and non-repudiation.

12.2.1 Randomized clock

The CryptoLite exposes the TRNG digitized analog signal (DAS) bit-stream sequence at its interface signal "*tr_trng_bitstream*". The sequence may be used to randomize the timing of functionality. This signal can be routed to GPIOs and DMA using the trigger multiplexer. See the Trigger multiplexer (Trigger MUX) chapter for more details.

The default value of the "*tr_trng_bitstream*" signal is set to constant '0' when oscillators are not enabled.

The TRNG functionality may be stopped by the monitor on repetition count (RC) or adaptive proportion (AP) error detection. If the TRNG functionality is stopped, both the DAS bit-stream and the randomized clock signal are set to constant '0'. If the "stop functionality" is not enabled, the DAS bit-stream and the randomized clock will continue to produce random bit sequences. See the True random number generator (TRNG) section for more details.

12.2.2 Hardware/software interface

The SHA, AES-128, and VU components have a similar hardware/software interface. The following components have a dedicated MMIO register that provides a pointer to a data structure in memory:

- The SHA component uses the CRYPTOLITE_SHA_DESCR pointer register
- The AES-128 component uses the CRYPTOLITE_AES_DESCR pointer register
- The VU component uses the CRYPTOLITE_VU_DESCR pointer register

A write to one of these MMIO registers starts a hardware function. The block will read the data structure from memory, and the data structure specifies the desired hardware function and its operands. The data structures are hardware-function-specific. The block performs the specified hardware function. From start to finish, the hardware function is considered busy.

- While a hardware function (SHA, AES-128, VU) is busy, the CryptoLite block cannot accept a new MMIO register write transfer; a new MMIO register write transfer will be made pending, which blocks any other (write or read) MMIO register transfers
- While a hardware function is busy and the CryptoLite block is not blocked, the block can accept a new MMIO register read transfer. For example, a read transfer may read the CRYPTOLITE_STATUS register, which indicates if the CryptoLite block is busy

The above behavior has the following implications:

- Only one hardware function (SHA, AES-128, VU) can be busy/active at the same time; that is, they are mutually exclusive. This is desired as the implementation shares hardware resources
- The data structure pointer MMIO registers cannot be overwritten by software while a hardware function is busy
- Software may prepare a next data structure in memory and write its pointer to a data structure pointer MMIO register while a hardware function is not busy. The write transfer of the next data structure's pointer will block the cryptographic block. This prevents the data structure pointer MMIO register from being overwritten

***Note**: Software should ensure that the current (busy) and next data structures in memory do not overlap*

***Note**: The TRNG and a hardware function (SHA, AES-128, VU) can be active at the same time. However, a TRNG MMIO register write transfer will be made pending when a hardware function is busy. Therefore, it is advised that the TRNG functionality be setup/started before the hardware functions are used. A TRNG MMIO register read transfer (CRYPTOLITE_INTR_TRNG and CRYPTOLITE_TRNG_RESULT) is possible when a hardware function is busy.*

As mentioned, a data structure in memory specifies the hardware function and its operands. The operands are typically pointers to 8-bit bytes or 32-bit word sequences in memory. These sequences in memory may represent a secret key (AES-128), a message chunk (SHA), or a long unsigned integer number (VU). The following figure shows a high-level block diagram of the descriptor, data structure, and operands.

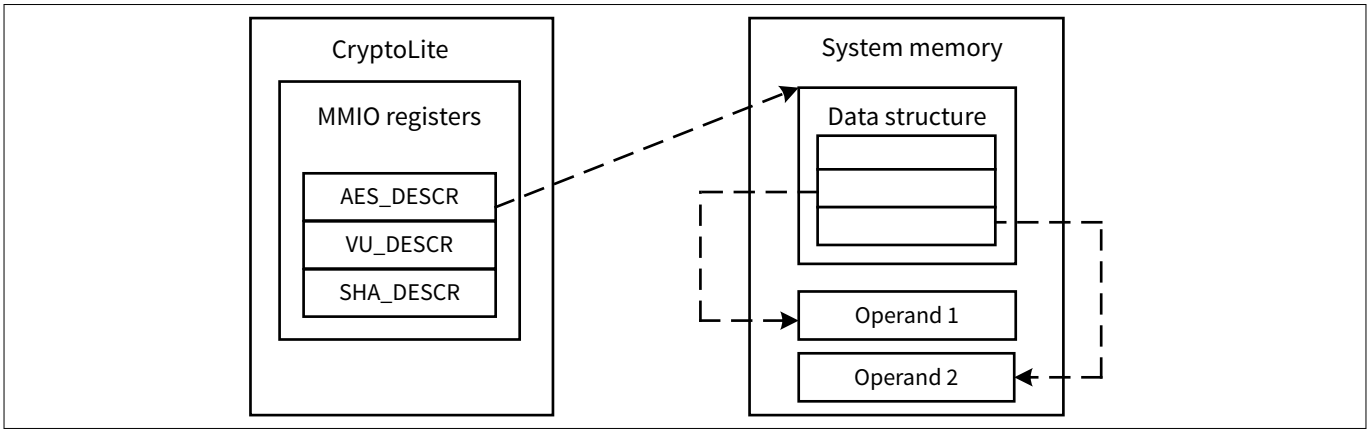


Figure 36 CryptoLite hardware/software interface

Most cryptographic functions are implemented using a combination of CPU software and hardware acceleration.

A hardware function (SHA, AES-128, VU) uses an AHB master interface to access memory. A bus error is reported (CRYPTOLITE_INTR_ERROR.BUS_ERROR), and the hardware function is terminated under an AHB bus error.

12.3 Block functions

12.3.1 AES-128

The AES-128 block cipher function is a member of the AES block cipher family. It encrypts a 128-bit block with a 128-bit key.

A block cipher mode of operation uses the AES-128 block cipher function in a specific configuration to provide message confidentiality and authenticity. The CryptoLite component provides AES-128 block cipher function hardware acceleration. A specific block cipher mode of operation is implemented using software and hardware acceleration. Typically, a message is broken up into multiple 128-bit blocks. The common modes of operation are:

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter (CTR)

The AES-128 block cipher function performs 10 rounds. Each round updates the internal cipher state. Each round has a dedicated 128-bit round key, which is derived from the provided 128-bit key.

The CryptoLite AES-128 block cipher function hardware acceleration consists of a single hardware function:

- **Encrypt**: Encrypt a 128-bit block P (plain-text) with a 128-bit SK (secret key) into a 128-bit block C (cipher-text)

A write to the CryptoLite block's CRYPTOLITE_AES_DESCR MMIO register starts a 128-bit AES encryption in the ECB mode. The CRYPTOLITE_AES_DESCR register expects a pointer to a three 32-bit word structure in memory. The pointer must be 32-bit word-aligned. The CryptoLite component retrieves the three control words that specify the hardware function.

- **Word 0**: Pointer to a 16 * 8-bit secret key SK. The secret key is a hardware function input
- **Word 1**: Pointer to a 16 * 8-bit block P. The block is a hardware function input
- **Word 2**: Pointer to a 16 * 8-bit block C. The block is a hardware function output

The following figure shows AES-128 data flow.

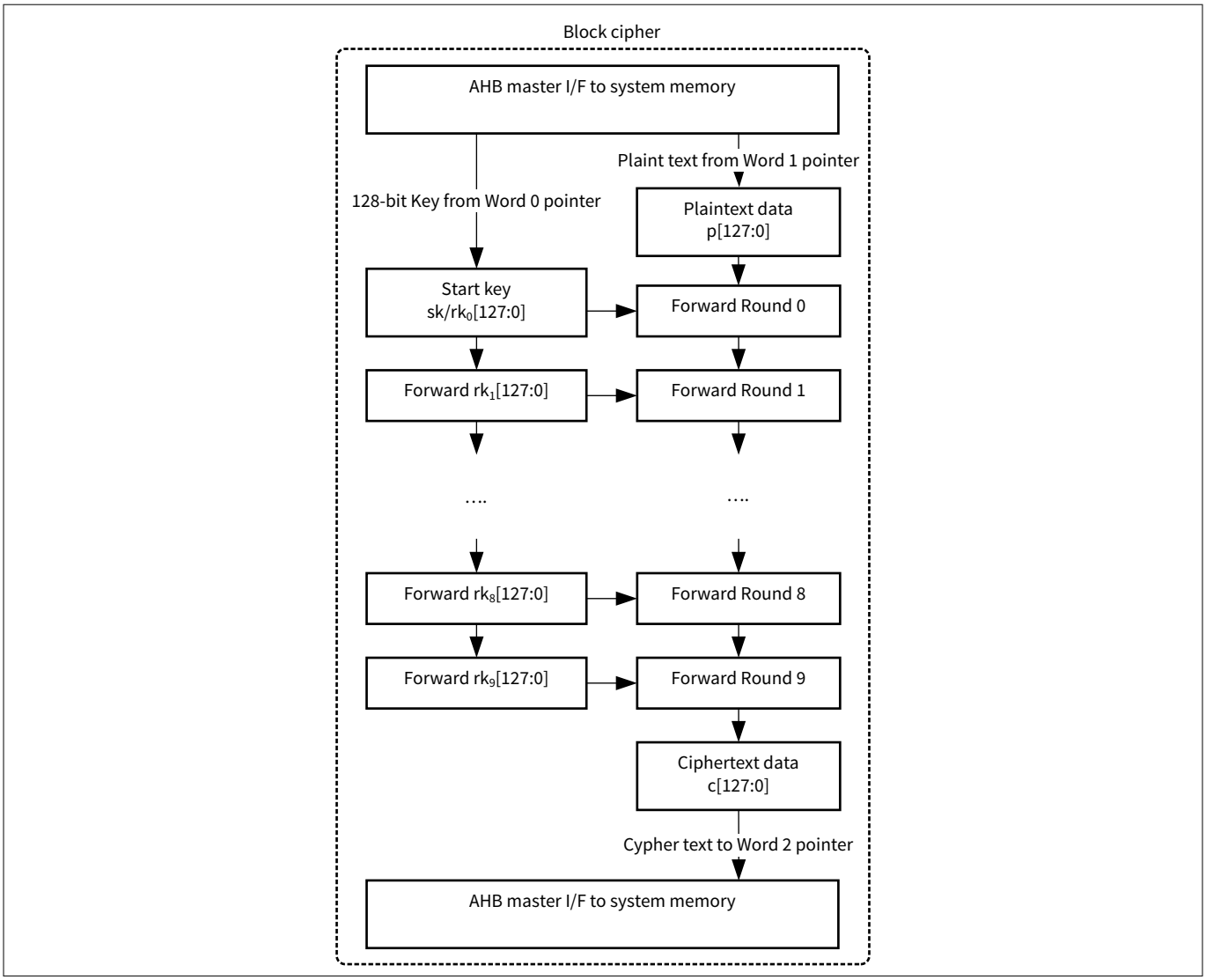


Figure 37 AES-128 data flow

***Note**: The AES-128 block cipher function operates on 8-bit bytes. As a result, no special endianness considerations are applicable from a hardware point of view.*

12.3.2 SHA

The CryptoLite supports the SHA-256, SHA-384, and SHA-512 algorithms acceleration. Before performing hash function, set the CRYPTOLITE_CTL.SHA_SEL register to select which the SHA operation enabled.

12.3.2.1 SHA-256/384/512

The SHA-256, SHA-384 and SHA-512 hash functions are a member of the SHA-2 hash family. To calculate a message's hash value (also known as message digest), a message is broken up into multiple 512-bit (SHA-256) or 1024-bit (SHA-384/SHA-512) message chunks. The SHA-256, SHA-384 and SHA-512 hash function process a single message chunk.

The SHA-256 hash function has an internal 256-bit (eight 32-bit words) hash state H[j] (with j = 0, …, 7), as well as the SHA-512 and SHA-384 hash function have an internal 512-bit (eight 64-bit words) hash state H[j] (with j = 0, ..., 7). For the first message chunk, the start hash state is initialized with SHA-256/SHA-384/SHA-512

(whichever is implemented based on CRYPTOLITE_CTL.SHA_SEL register configuration) pre-defined values as specified in NIST.FIPS.180-4. For each successive message chunk, the start hash state is the previous message chunk's end hash state. The final message chunk's end hash state is the message's hash value.

The SHA-256 hash function performs 64 rounds of a compression function, as well as the SHA-384 and SHA-512 hash function perform 80 rounds of a compression function. Each compression function round updates the internal hash state. For SHA-256, each round has a dedicated 32-bit round constant K[i] and a dedicated 32-bit message schedule value W[i]; for SHA-384 and SHA-512, each round has a dedicated 64-bit round constant K[i] and a dedicated 64-bit message schedule value W[i].

The round constants have SHA pre-defined values as specified in NIST.FIPS.180-4; they are fixed.

The message schedule values are created from the 512-bit (SHA-256) or 1024-bit (SHA-384/SHA-512) message chunk; they are message-specific.

The CryptoLite SHA-256 hash function hardware acceleration consists of two hardware functions:

- Message schedule function
  - For SHA-256, creation of 64 * 32-bit message schedule values W[i] (with = 0, …, 63) from a 512-bit message chunk M
  - For SHA-384 and SHA-512, creation of 80 * 64-bit message schedule values W[i] (with = 0, ..., 79) from a 1024-bit message chunk M
- Process function
  - For SHA-256, processing of a 256-bit hash state by performing 64 rounds of a compression function. The process inputs are a 256-bit hash state and a message schedule W (as created by the message schedule function). The process output is an updated 256-bit hash state. Note that the round constants K are an implicit rather than explicit process input (they are hardwired in the process function)
  - For SHA-384 and SHA-512, processing of 384-bit and 512-bit hash state respectively is done by performing 80 rounds of a compression function. The process inputs are a 512-bit hash state and a message schedule W (as created by the message schedule function). The process output is an updated 512-bit and 384-bit hash state. Note that the round constants K are an implicit, rather than explicit process input (they are hardwired in the process function)

To process a single message chunk, one message schedule hardware function is performed, followed by one process hardware function.

A write to the CRYPTOLITE_SHA_DESCR MMIO register starts a hardware function. The CRYPTOLITE_SHA_DESCR register expects a pointer to a three 32-bit word structure in memory.

***Note**: The pointer must be 32-bit word-aligned.*

The CryptoLite component retrieves the three control words that specify the hardware function. The first word (word 0) is generic; the other two words (words 1 and 2) are hardware function-specific.

- **Word 0** bit position 28 is used to specify the hardware function
  - If WORD0[28] is '0', a message schedule hardware function is performed
  - If WORD0[28] is '1', a process hardware function is performed

For a message schedule function, words 1 and 2 are defined as follows:

- For SHA-256
  - **Word 1**: Pointer to a 16 * 32-bit (512 bit) message chunk M. The message chunk is a hardware function input. The pointer must be 32-bit word-aligned
  - **Word 2**: Pointer to a 64 * 32-bit (2048 bit) message schedule array W. The message schedule is a hardware function output. The pointer must be 32-bit word-aligned
- For SHA-384 and SHA-512
  - **Word 1**: Pointer to a 16 * 64-bit (1024 bit) message chunk M. The message chunk is a hardware function input. The pointer must be 64-bit word-aligned
  - **Word 2**: Pointer to an 80 * 64-bit (5120 bit) message schedule array W. The message schedule is a hardware function output. The pointer must be 64-bit word-aligned

For a process function, words 1 and 2 are defined as follows:

- For SHA-256
  - **Word 1**: Pointer to an 8 * 32-bit (256 bit) hash state H. The hash state is both a hardware function input and a hardware function output (the input is overwritten by the output). The pointer must be 32-bit word-aligned
  - **Word 2**: Pointer to a 64 * 32-bit (2048 bit) message schedule array W. The message schedule is a hardware function input. The pointer must be 32-bit word-aligned
- For SHA-384 and SHA-512
  - **Word 1**: Pointer to an 8 * 64-bit (512 bit) hash state H. The hash state is both a hardware function input and a hardware function output (the input is overwritten by the output). The pointer must be 64-bit word-aligned
  - **Word 2**: Pointer to a 80 * 64-bit (5120 bit) message schedule array W. The message schedule is a hardware function input. The pointer must be 64-bit word-aligned

***Note**: The SHA-256 hash function operates on 32-bit words and assumes a big-endian memory layout of these words in memory. The PSC3M/P6 MCU uses a little-endian memory layout. This has the following implications:*

- *When a message chunk's 8-bit bytes are used to construct the message schedule's 32-bit words W[i], a little-to-big-endian conversion ("LE2BE") is processed by hardware*
- *When a final hash value's 32-bit words H[j] are used as a message digest (assuming the message digest is considered a byte stream), a big-to-little-endian conversion ("BE2LE") is required by software*

***Note**: The SHA-384 and SHA-512 hash function operate on 64-bit words and assumes a big-endian memory layout of these words in memory. The PSC3M/P6 MCU uses a little-endian memory layout. This has the following implications:*

- *When a message chunk's 8-bit bytes are used to construct the message schedule's 64-bit words W[i], a little-to-big-endian conversion ("LE2BE") is processed by hardware*
- *When a final hash value's 64-bit words H[j] are used as a message digest (assuming the message digest is considered a byte stream), a big-to-little-endian conversion ("BE2LE") is required by software*
- *SHA-384 writes eight 64-bit final hash digest to the memory, firmware is required to obtain by truncating the final hash value to its left-most 384 bits*

12.3.3 Vector unit (VU)

The vector unit (VU) supports long integer operations that are useful for asymmetric key cryptography such as RSA or Elliptic-curve cryptography (ECC).

The operations have a single destination operand (DST) and one or two source operands (SRC0 and SRC1). Each operand has a specific bit size. This bit size is specified in 32-bit word multiples. A value of x specifies a bit size of (x+1) * 32 bits.

The VU supports the following long integer operations:

- MUL: This operation is a regular long integer multiplication operation.
  - DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0] * SRC1[src1_bit_size-1:0]
- XMUL: This operation is a carry-less long integer multiplication operation.
  - DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0] xmul SRC1[src1_bit_size-1:0]
- ADD: This operation is a long integer addition operation.
  - DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0] + SRC1[src1_bit_size-1:0]
- SUB: This operation is a long integer subtraction operation.
  - DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0] - SRC1[src1_bit_size-1:0]
- XOR: operation. This operation is a long integer exclusive-or operation.
  - DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0] ^ SRC1[src1_bit_size-1:0]
- MOV: This operation moves/copies a long integer.
  - DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0]
- LSR1: This operation shifts a long integer 1-bit position to the right.
  - DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0] >> 1
- LSL1: This operation shifts a long integer 1-bit position to the left.
  - DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0] << 1
- LSR: This operation shifts a long integer a specific number of bit positions to the right.
  - DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0] >> “bit shift amount”
- COND_SUB: This operation is a conditional long integer subtraction operation.
  - DST[dst_bit_size-1:0] = (SRC0[src0_bit_size-1:0] >= SRC1[src1_bit_size-1:0]) ? (SRC0[src0_bit_size-1:0] - SRC1[src1_bit_size-1:0]) : SRC0[src0_bit_size-1:0]

  ***Note**: The operation’s “greater than equal” operator uses the source bit size (not the destination bit size), and the operation’s “subtraction” operator uses the destination bit size*

A write to the CRYPTOLITE_VU_DESCR register starts a VU operation. The CRYPTOLITE_VU_DESCR register expects a pointer to a four 32-bit word structure in memory. The pointer must be 32-bit word-aligned. The CryptoLite component retrieves the four control words that specify the hardware function. The first word (word 0) is generic; the other three words (words 1, 2, and 3) are hardware function-specific.

- **Word 0**:
  - Bit positions 31:28 specify the operation

Table 99 Vector unit opcodes

Opcode	Operation
0_H	Multiplication (MUL)
1_H	Addition (ADD)
2_H	Subtraction (SUB)
3_H	Exclusive or (XOR)
4_H	Binary multiplication (XMUL)
5_H	Logical shift right by 1 (LSR1)

(table continues...)

Table 99 (continued) Vector unit opcodes

Opcode	Operation
6_H	Logical shift left by 1 (LSL1)
7_H	Logical shift right (LSR)
8_H	Conditional subtraction (COND_SUB)
9_H	Move (MOV)
A_H to F_H	Reserved

- Bit positions 27:25 are reserved
- Bit positions 24:16 specify the word length (minus 1) of the destination operand. A 9-bit word length allows for up to 512 32-bit word (16384-bit) destination operands
- Bit positions 15:8 specify the word length (minus 1) of source operand 1 (if present)
- Bit positions 7: 0 specify the word length (minus 1) of source operand 0 (if present). An 8-bit word length allows for up to 256 32-bit word (8192-bit) source operands

***Note**: Note that for reserved opcodes (>9), there is no change in hardware state and only a few cycles STATUS.BUSY will be high. This is because CryptoLite hardware does not know if the VU opcode is undefined at the point of start initiated by MMIO write, so CryptoLite will be busy until it fetches the opcode from the descriptor location and becomes idle (STATUS.BUSY is cleared to '0' when an undefined opcode is detected)*

For MUL, XMUL, ADD, SUB, XOR, and COND_SUB operations, words 1, 2, and 3 are defined as follows:

- **Word 1**: Pointer to word 0 (the least significant word) of source operand 0. The pointer must be 32-bit word-aligned
- **Word 2**: Pointer to word 0 (the least significant word) of source operand 1. The pointer must be 32-bit word-aligned
- **Word 3**: Pointer to word 0 (the least significant word) of the destination operand. The pointer must be 32-bit word-aligned

For LSR1, LSL1, and MOV operations, words 1, 2, and 3 are defined as follows:

- **Word 1**: Pointer to word 0 (the least significant word) of the source operand. The pointer must be 32-bit word-aligned
- **Word 2**: Not used
- **Word 3**: Pointer to word 0 (the least significant word) of the destination operand. The pointer must be 32-bit word-aligned

For LSR operations, words 1, 2, and 3 are defined as follows:

- **Word 1**: Pointer to word 0 (the least significant word) of the source operand. The pointer must be 32-bit word-aligned
- **Word 2**: Shift amount. Bit positions 12:0 specify the shift amount
- **Word 3**: Pointer to word 0 (the least significant word) of the destination operand. The pointer must be 32-bit word-aligned

Note on operand size:

- The source and destination operands have dedicated size specifications. The destination operand may be larger than the source operands. Typically, this is useful for multiplication operations
- When a destination operand size is smaller than the operation result size, the more significant bits of the operation result are ignored. When a destination operand size is larger than the operation result size, the operation result '0' is extended

Typically, the source operand and destination operand occupy different memory locations. However, to limit memory allocation, it may be beneficial to:

- Use the same memory locations for the two source operands. This can be done without any restrictions. For example, the two source operands of an ADD operation are the same to double the source operand
- Use the same memory locations for the source and the destination operands. This comes with restrictions that are operation dependent. Specifically, the following operations allow for the use of the same memory locations for the source operand and the destination operand:
  - ADD, SUB, XOR, MOV, LSR1, and LSL1 operations

  The following operations do not allow for the use of the same memory locations for the source and the destination operands:
  - MUL, XMUL, LSR, and COND_SUB operations

***Note**: The VU operates on 32-bit word multiples and assumes a little-endian memory layout of these words in memory. The PSC3M/P6 MCU uses a little-endian memory layout. As a result, no endianness conversion is required.*

12.3.4 True random number generator (TRNG)

The true random number generator component (TRNG) generates true random numbers. The bit size of these generated numbers is fixed at 32 bits.

The TRNG relies on up to six ring oscillators to provide physical noise sources. A ring oscillator consists of a series of inverters connected in a feedback loop to form a ring. Due to the temperature sensitivity of the inverter delays, jitter is introduced to a ring's oscillating signal. The jittered oscillating signal is sampled to produce a "digitized analog signal" (DAS). This is done for all multiple-ring oscillators.

To increase entropy and reduce bias in DAS bits, the DAS bits are further post-processed. Post-processing involves two steps:

- An optional reduction step (over up to six ring oscillator DAS bits and over one or multiple DAS bit periods) to increase entropy
- An optional “von Neumann correction” step to reduce a ‘0’ or ‘1’ bias

This correction step processes pairs of reduction bits as produced by the previous step. Given two reduced bits r0 and r1 (with r0 being produced before r1), the correction step is defined as follows:

{r0, r1} = {0, 0}: no bit is produced

{r0, r1} = {0, 1}: a ‘0’ bit is produced (bit r0)

{r0, r1} = {1, 0}: a ‘1’ bit is produced (bit r0)

{r0, r1} = {1, 1}: no bit is produced

That is, the correction step only produces a bit on a ‘0’ to ‘1’ or ‘1’ to ‘0’ transition. Note that for a random input bit sequence, the correction step produces an output bit sequence of close to one-quarter the frequency of the input bit sequence (the input reduction bits are processed in non-overlapping pairs, and only half of the pair encodings result in an output bit).

Post-processing produces bit samples that are considered true random bit samples. The true random bit samples are shifted into a register to provide random values of up to 32 bits.

As a result of high switching activity, ring oscillators consume a significant amount of power. Therefore, when the TRNG functionality is disabled, the ring is "broken" to prevent switching. When TRNG functionality is enabled, a ring oscillator initially has predictable behavior. However, over time, infinitesimal environmental (temperature) changes cause an increasing deviation from this predictable behavior.

- During the initial delay, the ring oscillator is not a reliable physical noise source
- After an initial delay, the same ring oscillator will show different oscillation behavior and provide a reliable physical noise source

Therefore, the DAS bits can be dropped during an initialization period (CRYPTOLITE_TRNG_CTL0.INIT_DELAY).

The following figure gives an overview of the TRNG component.

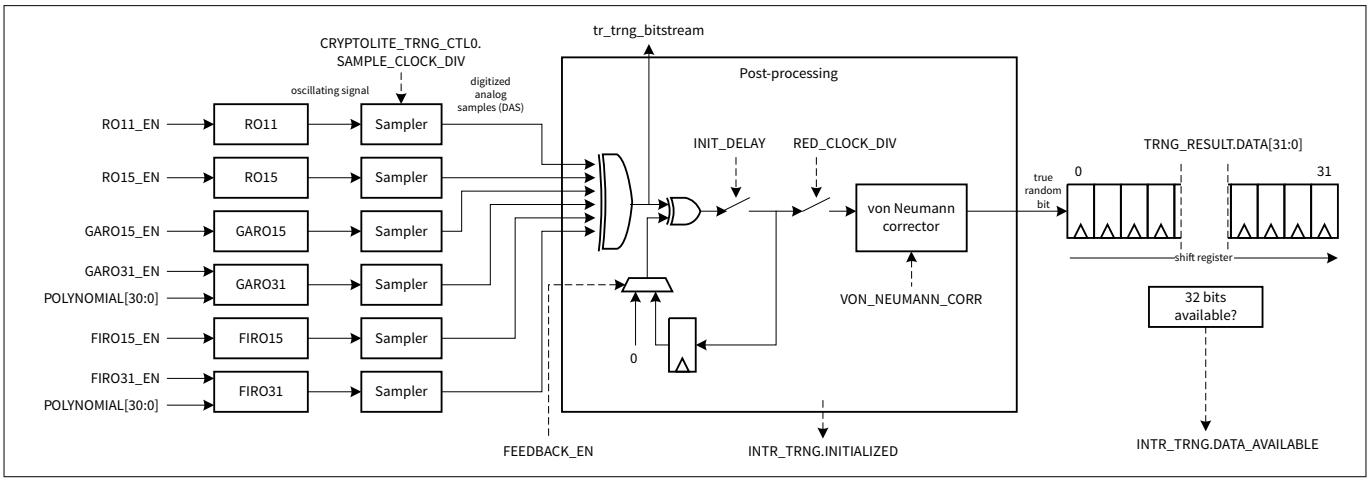


Figure 38 TRNG overview

The FEEDBACK_EN bitfield of the CRYPTOLITE_TRNG_CTL0 register controls the feedback. When '0', there is no feedback. This functionality is present, as it was found that the "von Neumann" corrector may be less effective with feedback enabled.

The "Sampler" logic digitizes an oscillating signal. The following figure illustrates the functionality.

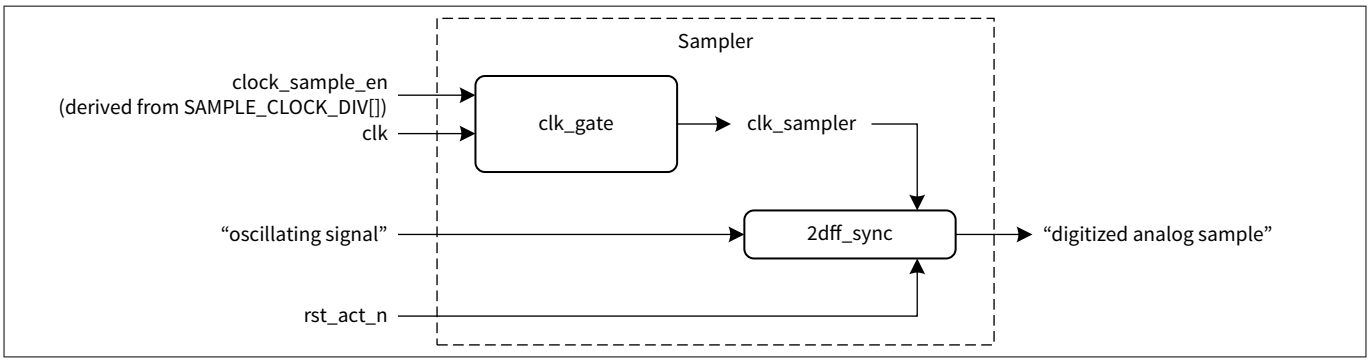


Figure 39 Sampler logic

***Note**: When a ring oscillator is disabled, the synchronization logic is reset. Therefore, the ring oscillator contributes a constant '0' to the reduction step of the post-processing.*

As mentioned, the TRNG relies on up to six ring oscillators:

- RO11: A fixed-ring oscillator consisting of 11 inverters
- RO15: A fixed-ring oscillator consisting of 15 inverters
- GARO15: A fixed Galois-based ring oscillator of 15 inverters
- GARO31: A flexible Galois-based ring oscillator of up to 31 inverters. A programmable polynomial of up to order 31 provides flexibility in the oscillator feedback
- FIRO15: A fixed Fibonacci-based ring oscillator of 15 inverters
- FIRO31: A flexible Fibonacci-based ring oscillator of up to 31 inverters. A programmable polynomial of up to order 31 provides flexibility in the oscillator feedback

Each ring oscillator can be enabled or disabled. When disabled, the ring is "broken" to prevent switching. The XXX_EN enable signals originate from the CRYPTOLITE_TRNG_CTL1 register.

***Note**: When using TRNG, all oscillators shall be active. The recommend polynomial of GARO31 is 0xB2D1AB70, and the recommend polynomial of FIRO31 is 0xE6B8C3B3.*

12.3.4.1 Fixed ring oscillators: RO11, RO15, GARO15, FIRO15

The following figure illustrates the schematics of the fixed-ring oscillators.

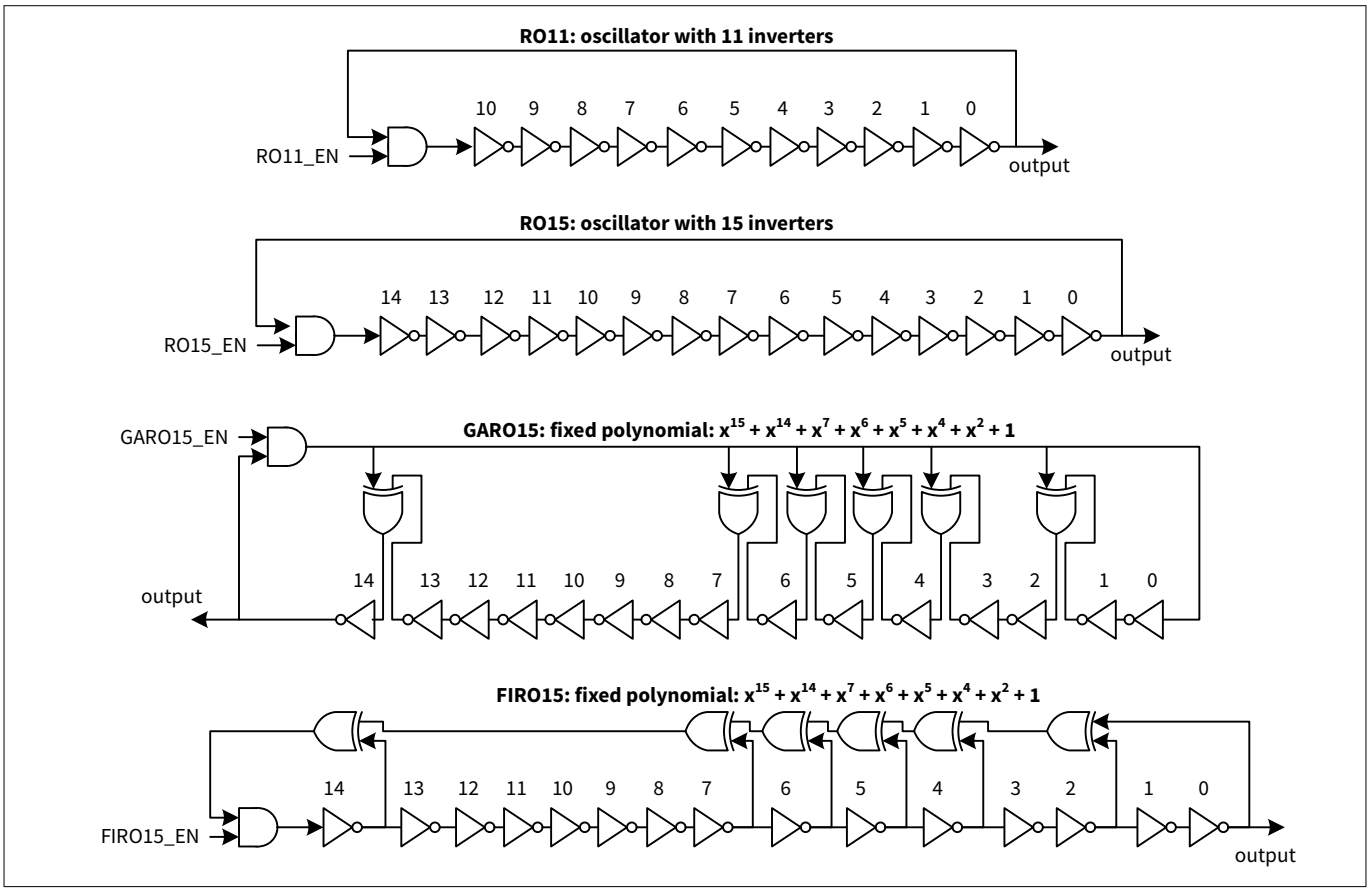


Figure 40 Four fixed-ring oscillators: RO11, RO15, GARO15, and FIRO15

12.3.4.2 Ring oscillator: GARO31

The flexible Galois based-ring oscillators rely on programmable polynomials to specify the oscillator feedback. This allows for rings of 1, 3, 5,…, 31 inverters (an odd number is required to generate an oscillating signal). The following figure gives an overview of the Galois-based ring oscillator.

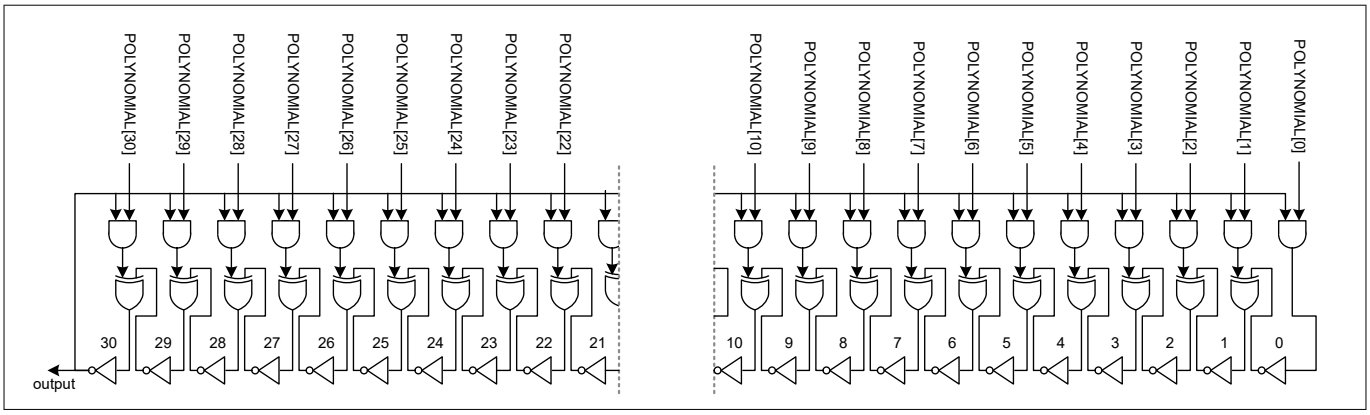


Figure 41 Flexible Galois-based ring oscillator: GARO31

When the ring oscillator is disabled (GARO31_EN bit of the CRYPTOLITE_TRNG_CTL1 register is '0'), the polynomial is forced to "0" and the ring is broken, as illustrated in the following figure.

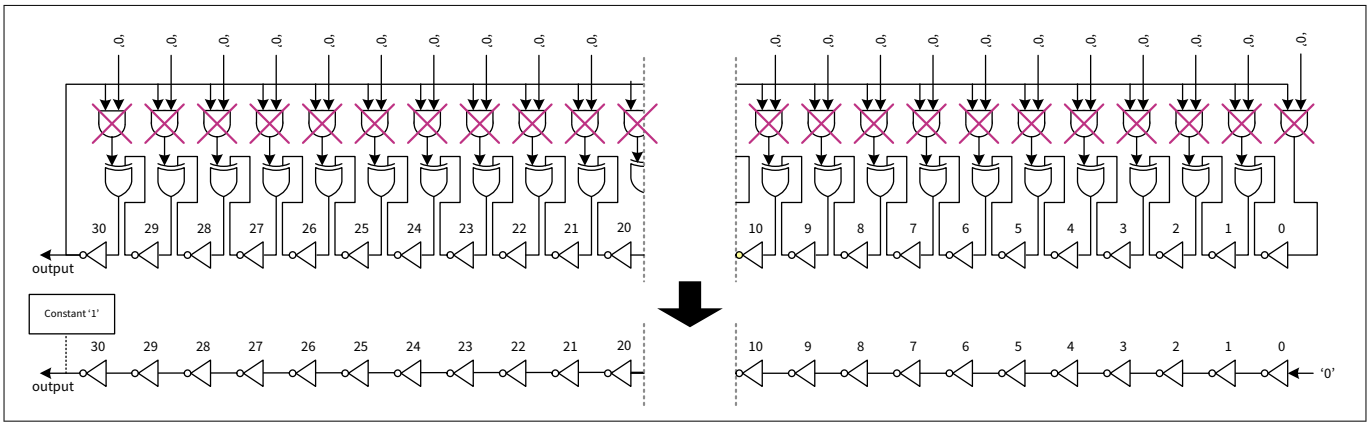


Figure 42 GARO31 stopped

The programmable polynomial specifies the oscillator feedback. The following figure illustrates two examples.

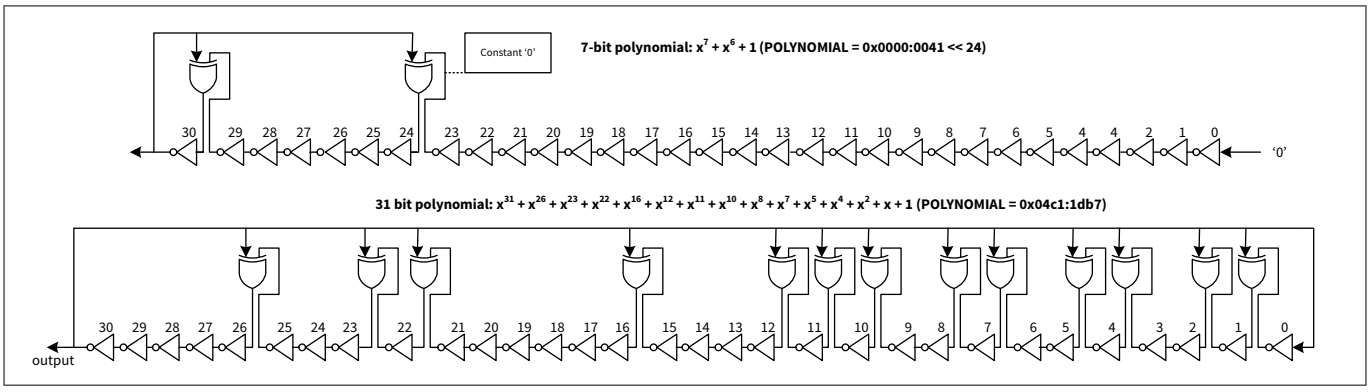


Figure 43 GARO31 examples

12.3.4.3 Ring oscillator: FIRO31

The flexible Fibonacci-based ring oscillators rely on programmable polynomials to specify the oscillator feedback. This allows for rings of 1, 3, 5,…, 31 inverters (an odd number is required to generate an oscillating signal). The following figure gives an overview of the Fibonacci-based ring oscillator.

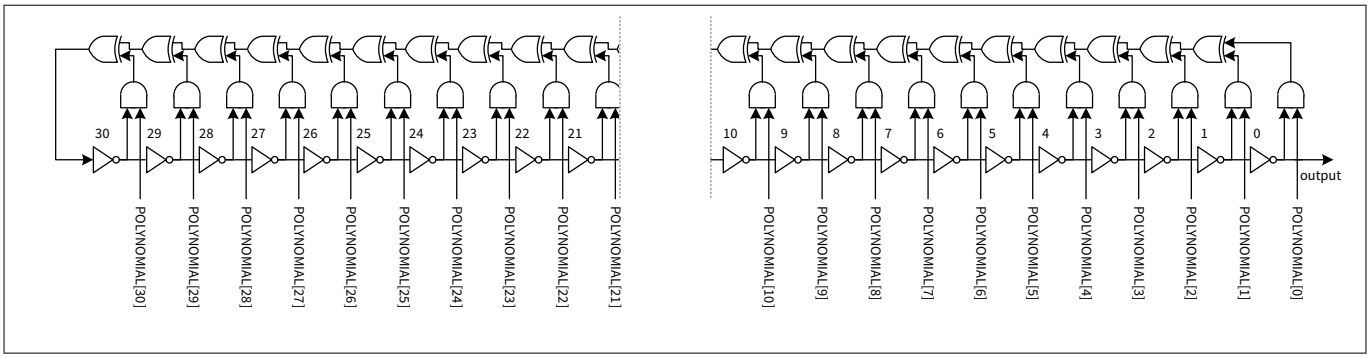


Figure 44 Flexible Fibonacci-based ring oscillator: FIRO31

When the ring oscillator is disabled (FIRO31_EN bit of the CRYPTOLITE_TRNG_CTL1 register is '0'), the polynomial is forced to "0" and the ring is broken, as illustrated in the following figure.

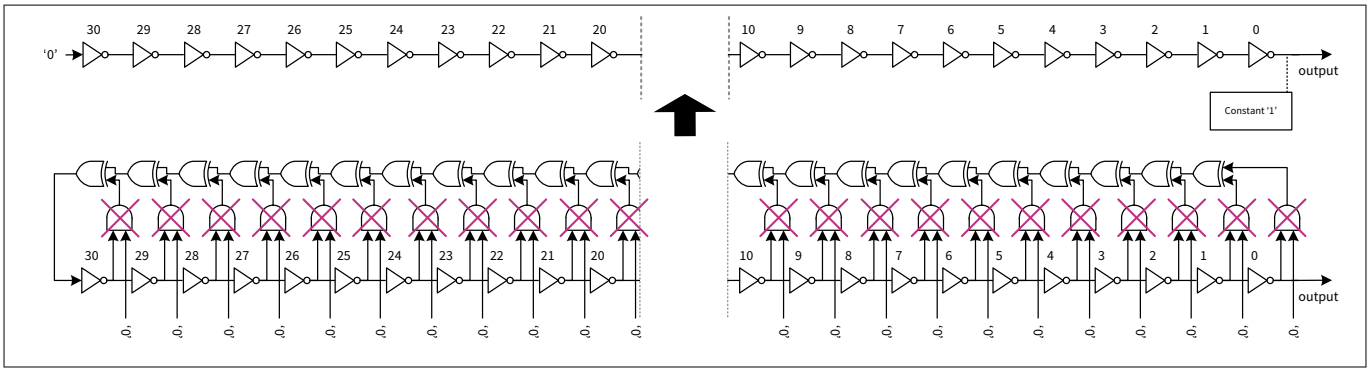


Figure 45 FIRO31 stopped

The programmable polynomial specifies the oscillator feedback. The following figure illustrates two examples.

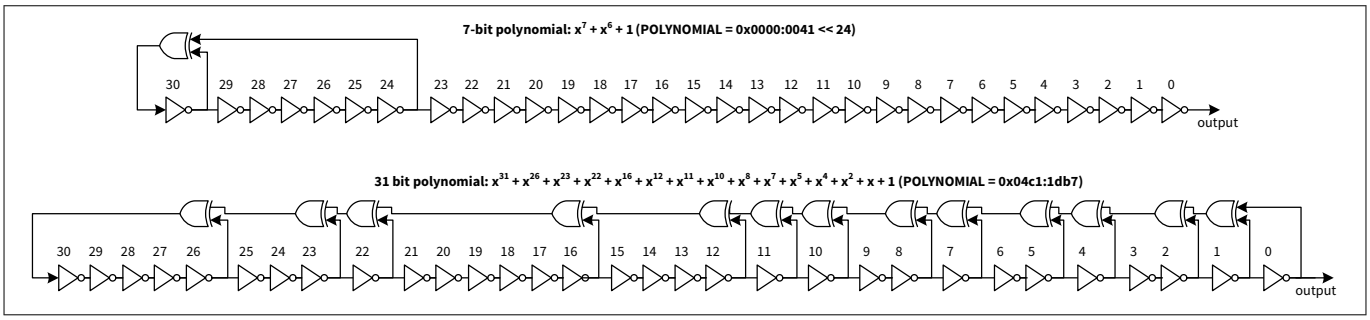


Figure 46 FIRO31 examples

12.3.4.4 TRNG result

A 32-bit TRNG value is provided through a read from the CRYPTOLITE_TRNG_RESULT register. This read is a blocking read; when 32-bit is not available, the read is blocked. A read from the CRYPTOLITE_TRNG_RESULT register returns "0" when the TRNG is not enabled (all XXX_EN fields of the CRYPTOLITE_TRNG_CTL1 register are '0').

12.3.4.5 TRNG health monitor

The TRNG has a built-in health monitor that performs tests on the digitized noise source to detect deviations from the intended behavior. For example, the health monitor detects "stuck at" faults in the digitized analog samples. The following figure illustrates the functionality of the health monitor.

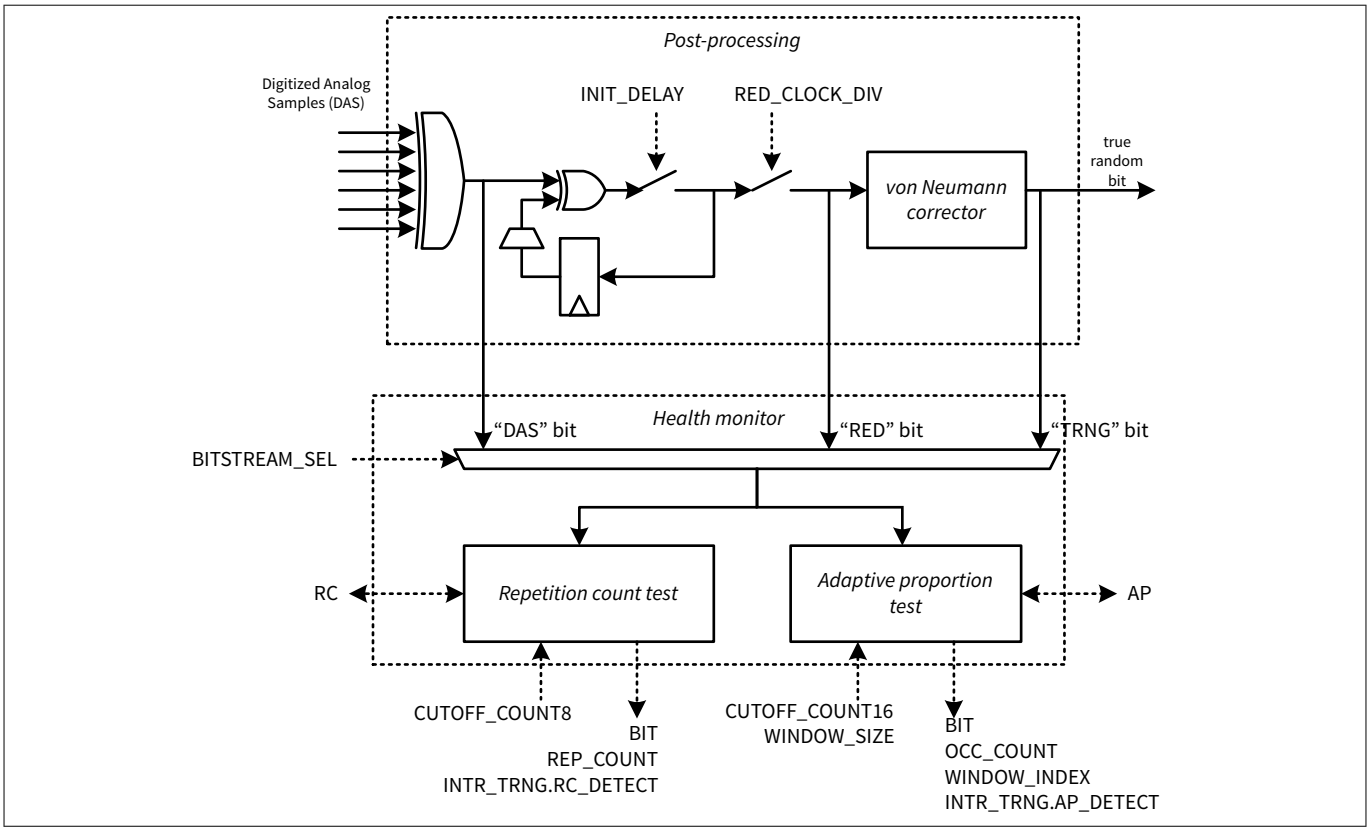


Figure 47 TRNG health monitor

The health monitor tests one out of three selected digitized bit streams:

- DAS bit-stream: This is the XOR of the digitized analog samples
- RED bit-stream: This is the bit-stream of reduction bits. Note that each reduction bit may be calculated over multiple DAS bits
- TR bit-stream: This is the bit-stream of true random bits (after the "von Neumann reduction" step)

The health monitor performs two different tests:

- **The repetition count test**

  This test checks for the repetition of the same bit value ('0' or '1') in a bit-stream. A detection indicates that a specific active bit value (specified by a status field BIT) has repeated for a pre-programmed number of bits (specified by a control field CUTOFF_COUNT8[7:0] of the CRYPTOLITE_RNG_MON_RC_CTL register). The test uses a counter to maintain the number of repetitions of the active bit value (specified by the status field REP_COUNT[7:0] of the CRYPTOLITE_TRNG_MON_RC_STATUS1 register).

  If the test is started (specified by the RC bitfield of the CRYPTOLITE_TRNG_MON_CTL register) and a change in the bit-stream value is observed, the active bit value BIT is set to the new bit value, and the repetition counter REP_COUNT[] is set to "1". If the bit-stream value is unchanged, the repetition counter REP_COUNT[] is incremented by "1".

  A detection stops the repetition count test (hardware sets the RC bit of the CRYPTOLITE_TRNG_MON_CTL register to '0'), sets the associated interrupt status field to '1', and ensures that hardware does not modify the status fields. When the test is stopped, REP_COUNT[] equals CUTOFF_COUNT8[].

  A detection stops the TRNG functionality (all XXX_EN fields of the CRYPTOLITE_TRNG_CTL1 register are set to '0' by hardware) if the STOP_ON_RC_DETECT bit of the CRYPTOLITE_TRNG_CTL0 register is set to '1'

- **The adaptive proportion test**

  This test checks for a disproportionate occurrence of a specific bit value ('0' or '1') in a bit stream. A detection indicates that a specific active bit value (specified by a status field BIT) has occurred a pre-

programmed number of times (specified by a control field CUTOFF_COUNT16[15:0] of the CRYPTOLITE_TRNG_MON_AP_CTL register) in a bit sequence of a specific bit window size (specified by a control field WINDOW_SIZE[31:16] of CRYPTOLITE_TRNG_MON_AP_CTL register). The test uses a counter to maintain an index in the current window (specified by WINDOW_INDEX[15:0] of the CRYPTOLITE_TRNG_MON_AP_STATUS1 register) and a counter to maintain the number of occurrences of the active bit value (specified by the status field OCC_COUNT[31:16] of the CRYPTOLITE_TRNG_MON_AP_STATUS1 register).

If the test is started (specified by the AP bit of the CRYPTOLITE_TRNG_MON_CTL register), the bit-stream is partitioned into bit sequences of a specific window size. At the first bit of a bit sequence, the active bit value BIT is set to the first bit value, the counter WINDOW_INDEX is set to “0”, and the counter OCC_COUNT is set to “1”. For all other bits of a bit sequence, the counter WINDOW_INDEX is incremented by “1”. If the new bit value equals the active bit value BIT, the counter OCC_COUNT[] is incremented by “1”. Note that the active bit value BIT is only set at the first bit of a bit sequence.

A detection stops the adaptive proportion test (hardware sets the AP bit of the CRYPTOLITE_TRNG_MON_CTL register to ‘0’), sets the associated interrupt status field to ‘1’, and ensures that hardware does not modify the status fields. When the test is stopped, OCC_COUNT[] equals CUTOFF_COUNT16[] and the WINDOW_INDEX identifies the bit sequence index on which the detection occurred.

A detection stops the TRNG functionality (all XXX_EN fields of the CRYPTOLITE_TRNG_CTL1 register are set to ‘0’ by hardware) if the STOP_ON_AP_DETECT bit of the CRYPTOLITE_TRNG_CTL0 register is set to ‘1’

12.4 Interrupts

The CryptoLite provides two interrupts:

- TRNG interrupt
- Error interrupt

12.4.1 TRNG interrupt

This interrupt is used to report TRNG behavior, and this interrupt is generated for the below scenarios:

- TRNG is initialized
- TRNG data is available in the CRYPTOLITE_TRNG_RESULT register
- TRNG monitor detects a "repetition count" error
- TRNG monitor detects an "adaptive proportion" error

The TRNG interrupt has an interrupt mask in the CRYPTOLITE_INTR_TRNG_MASK register. By making the interrupt mask low, the corresponding interrupt source is ignored. The TRNG interrupt is generated if the interrupt mask bit is high and the corresponding interrupt source is pending.

When servicing a TRNG interrupt, the interrupt service routine (ISR) can clear the interrupt source by writing a ‘1’ to the corresponding interrupt bit in the CRYPTOLITE_INTR_TRNG register.

The CRYPTOLITE_INTR_TRNG_MASKED register is the logical AND between the interrupt sources and the interrupt mask. This register provides a convenient way for the firmware to determine the source of the interrupt.

For verification and debug purposes, a set bit (such as DATA_AVAILABLE in the CRYPTOLITE_INTR_TRNG_SET register) is used to trigger each interrupt. This action allows the firmware to generate an interrupt without the actual event occurring.

12.4.2 Error interrupt

This interrupt is generated for an AHB bus error on the AHB master interface.

The error interrupt is enabled by setting the BUS_ERROR bit of the CRYPTOLITE_INTER_ERROR_MASK register and reads the CRYPTOLITE_INTR_ERROR_MASKED register to know the interrupt masked status forwarded to the CPU interrupt controller.

When servicing the error interrupt, the interrupt service routine (ISR) can clear the interrupt by writing a "1" to the BUS_ERROR bit of the CRYPTOLITE_INTR_ERROR register.

The BUS_ERROR bit of the CRYPTOLITE_INTR_ERROR_SET register can be used to assert an interrupt for firmware debugging.

***Note**: The cryptographic block terminates its AES, SHA, or VU functionality when it detects an error. The error is sticky; this allows software to check for an error after a series of operations rather than checking after every individual operation.*

12.5 Registers summary

Table 100 CryptoLite register summary

Register	Function
CRYPTOLITE_CTL	CryptoLite control register
CRYPTOLITE_STATUS	CryptoLite status register
CRYPTOLITE_AES_DESCR	AES descriptor pointer register
CRYPTOLITE_VU_DESCR	VU descriptor pointer register
CRYPTOLITE_SHA_DESCR	SHA descriptor pointer register
CRYPTOLITE_INTR_ERROR	CryptoLite error interrupt register
CRYPTOLITE_INTR_ERROR_SET	CryptoLite error interrupt set register
CRYPTOLITE_INTR_ERROR_MASK	CryptoLite error interrupt mask register
CRYPTOLITE_INTR_ERROR_MASKED	CryptoLite masked error interrupt output register
CRYPTOLITE_TRNG_CTL0	TRNG control 0 register
CRYPTOLITE_TRNG_CTL1	TRNG control 1 register
CRYPTOLITE_TRNG_STATUS	TRNG status register
CRYPTOLITE_TRNG_RESULT	TRNG result register
CRYPTOLITE_TRNG_GARO_CTL	TRNG Galois-based ring oscillator (GARO) control register
CRYPTOLITE_TRNG_FIRO_CTL	TRNG Fibonacci-based ring oscillator (FIRO) control register
CRYPTOLITE_TRNG_MON_CTL	TRNG health monitor control register
CRYPTOLITE_TRNG_MON_RC_CTL	TRNG health monitor repetition count test control register
CRYPTOLITE_TRNG_MON_RC_STATUS0	TRNG health monitor repetition count test status 0 register
CRYPTOLITE_TRNG_MON_RC_STATUS1	TRNG health monitor repetition count test status 1 register
CRYPTOLITE_TRNG_MON_AP_CTL	TRNG health monitor adaptive proportion test control register
CRYPTOLITE_TRNG_MON_AP_STATUS0	TRNG health monitor adaptive proportion test status 0 register
CRYPTOLITE_TRNG_MON_AP_STATUS1	TRNG health monitor adaptive proportion test status 0 register
CRYPTOLITE_INTR_TRNG	TRNG interrupt register

(table continues...)

Table 100 (continued) CryptoLite register summary

Register	Function
CRYPTOLITE_INTR_TRNG_SET	TRNG interrupt set register
CRYPTOLITE_INTR_TRNG_MASK	TRNG interrupt request mask register
CRYPTOLITE_INTR_TRNG_MASKED	TRNG masked interrupt output register

13 TrustZone

13.1 Features

- Arm® TrustZone enabled Cortex®-M33 CPU
- Secure and non-secure CPU states
- Attribution units - Secure Attribution Unit (SAU) and Implementation Defined Attribution Unit (IDAU)
- Secure and non-secure aliasing of the memory map
- Secure and non-secure interrupts
- Secure and non-secure peripherals
- Secure and non-secure debug

13.2 Architecture

Arm® TrustZone technology is an optional security extension designed for Armv8-M. The security extension is enabled on PSC3M/P6 MCU devices for improved security.

TrustZone implements system-wide security by partitioning the resources (memory and peripherals) into secure or non-secure worlds. TrustZone introduces two more states to the CPU - secure and Non-secure. These security states are orthogonal to the existing Thread and Handler modes, resulting in four modes: Secure Handler, Secure Thread, Non-secure Handler, and Non-secure Thread modes. In the PSC3M/P6 MCU device, with TrustZone extension enabled, the M33 CPU starts in a secure state. In the secure state the software has access to both secure as well as non-secure resources, while in the non-secure state, software has access only to non-secure resources. This allows execution of secure and non-secure software on the same CPU with an isolation of resources between the two worlds and which is based on the memory map. The partitioning of memory map is achieved with a programmable Secure Attribution Unit (SAU) and a fixed Implementation Defined Attribution Unit (IDAU).

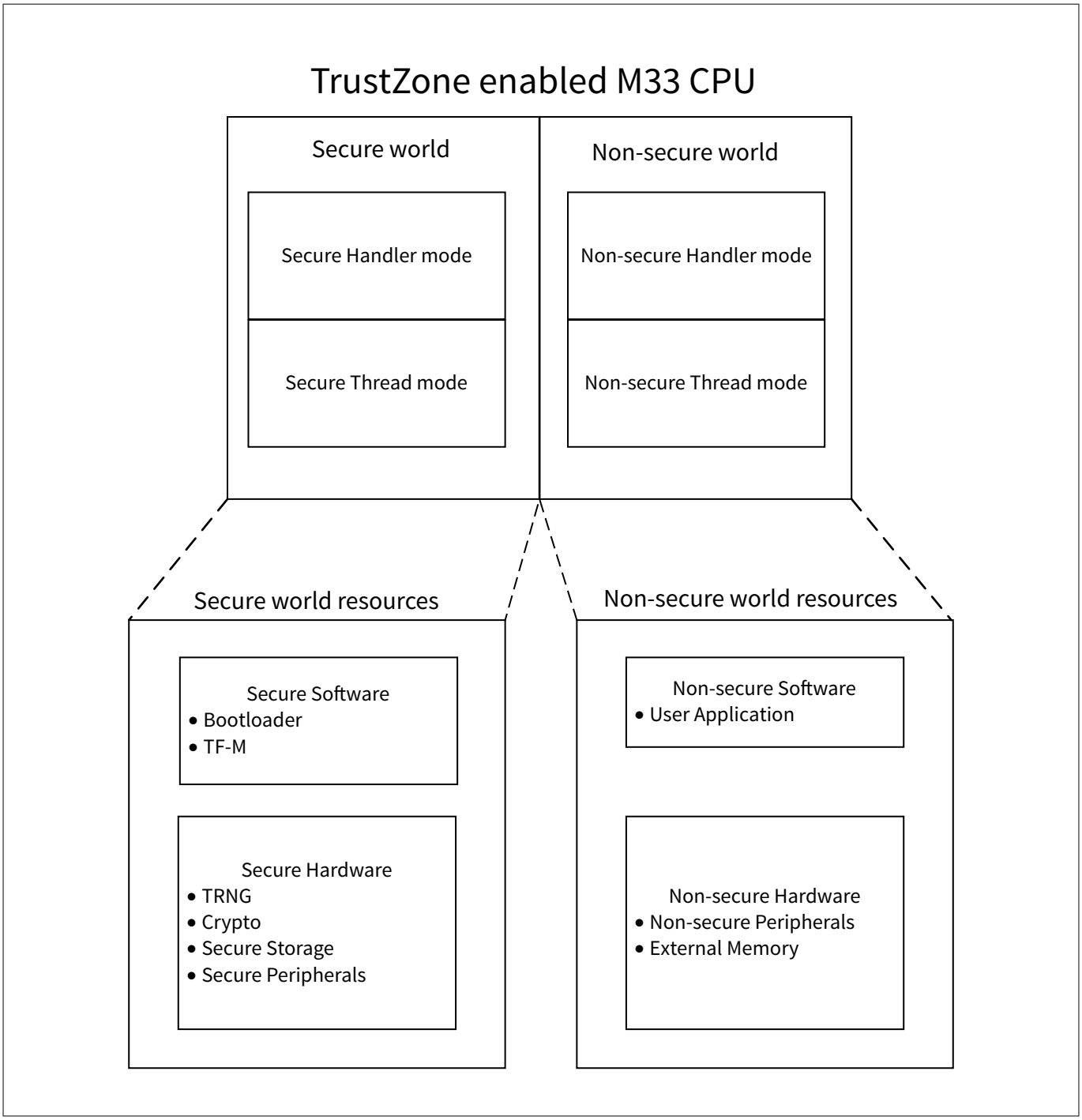


Figure 48 Arm® TrustZone enabled system overview

The following key internal resources are banked between secure and non-secure states:

- Stack pointers
- Interrupt Masking registers
- MPU
- SysTick timer
- VTOR
- Some other registers of System Control Block

Some registers are banked on a bit-by-bit basis. See Arm® Cortex®-M33 Devices Generic User Guide for full list of banked registers. The M33 CPU has separate vector tables for secure and non-secure software. With the security extension enabled, a new SecureFault exception is added in the system.

13.3 Memory regions

The complete 4 GB address space is partitioned into secure and non-secure regions. These regions are aliased in the address space, meaning that one memory location is mapped to two different locations in the address map, one in the secure region and the other in the non-secure region. All memories and peripherals are aliased and placed in both regions.

The address space is partitioned into three regions:

- Secure (S):
  - Secure addresses are used for accessing memory and peripherals which are accessible only by Secure software
  - A CPU in secure state can only execute instructions from the secure region
  - Data in secure addresses is accessible by Secure software
- Non-secure callable (NSC):
  - Special type of secure region which holds the secure gateway (SG) instructions, allows the state transition from non-secure to secure state
- Non-secure (NS) :
  - Non-Secure addresses are used for accessing memory and peripherals which are accessible by both Secure and Non-secure software
  - A CPU in the Non-Secure state execute instructions only from the non-secure region
  - Data in Non-Secure addresses is accessible by secure and non-secure software

The attribution units, SAU, and IDAU assign security attributes to memory locations and partition the memory space. Any access violation originating from the security attribute checks generates a SecureFault exception.

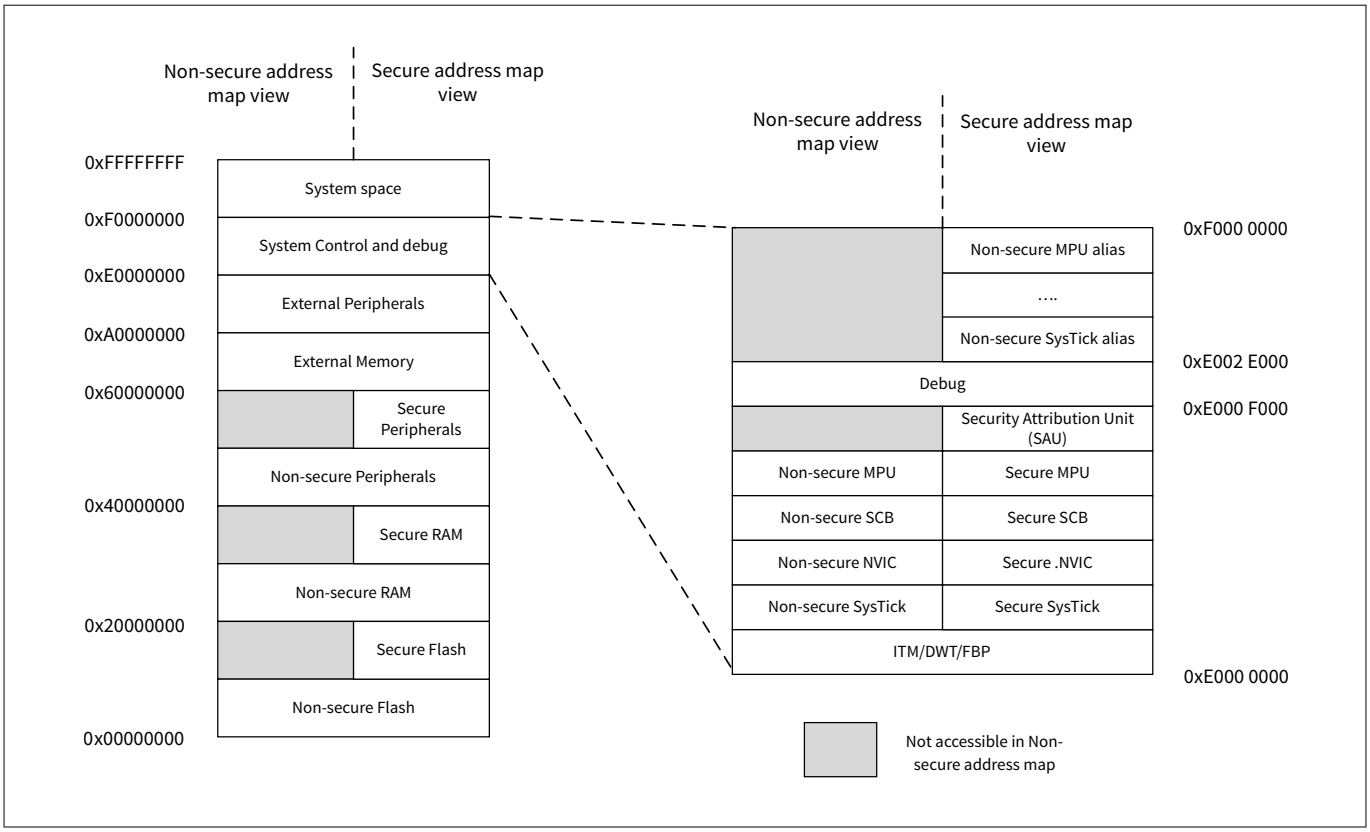


Figure 49 Memory space of TrustZone enabled system

13.4 Attribution units

The memory space is partitioned with Secure Attribution Unit (SAU) and Implementation defined Attribution Unit (IDAU). The SAU and IDAU work in conjunction to determine the security attribute of an address. The higher security level amongst the two will be applied. The HIGH to LOW security levels are as follows: Secure (S) > non-secure callable (NSC) > non-secure (NS)

Table 101 summarizes the resultant security attribute.

Table 101 Resultant security level

SAU	IDAU	Resultant security attribute
S	X	S
X	S	S
NSC	NSC or NS	NSC
NSC or NS	NSC	NSC
NS	NS	NS

***Note**: X = Don't Care*

13.4.1 Implementation Defined Attribution Unit (IDAU)

- IDAU provides an address aliasing scheme that allows a memory space to be partitioned into non-secure, non-secure callable and secure regions

- It has a fixed implementation, meaning that the address aliasing scheme is not configurable
- It uses the address bit 28 to segregate secure and non-secure memory regions. A single bit can represent only two states—0 for NS and 1 for S. To indicate NSC regions, the IDAU reuses the S indication. It marks the address ranges 0x1000_0000–0x1FFF_FFFF (CODE S) and 0x3000_0000–0x3FFF_FFFF (RAM S) as NSC by reusing the S indication
- Private peripheral bus (PPB) and external private peripheral bus (EPPB) regions are exempt from security attribute check
- Address map partitioning can be overridden only with a SAU configuration of a higher security level
- Implementation is attached externally to M33 CPU and all non-CPU masters on their AHB buses
- Returns region number for all addresses, required for Test Target (TT) instruction

Table 102 shows the memory space partitioning defined by the fixed IDAU implementation:

Table 102 IDAU memory map partitioning

Region	Start address	End address	Size	Security attribute	Region number
Code NS	0x0000 0000	0x0FFF FFFF	256 MB	NS	0x0
Code S	0x1000 0000	0x1FFF FFFF	256 MB	NSC	0x1
RAM NS	0x2000 0000	0x2FFF FFFF	256 MB	NS	0x2
RAM S	0x3000 0000	0x3FFF FFFF	256 MB	NSC	0x3
Peripherals NS	0x4000 0000	0x4FFF FFFFF	256 MB	NS	0x4
Peripherals S	0x5000 0000	0x5FFF FFFF	256 MB	S	0x5
External memory NS	0x6000 0000	0x6FFF FFFF	256 MB	NS	0x6
External memory S	0x7000 0000	0x7FFF FFFF	256 MB	S	0x7
External memory reserved	0x8000 0000	0x8FFF FFFF	256 MB	NS	0x8
	0x9000 0000	0x9FFF FFFF	256 MB	S	0x9
External device reserved	0xA000 0000	0xAFFF FFFF	256 MB	NS	0xA
	0xB000 0000	0xBFFF FFFF	256 MB	S	0xB
	0xC000 0000	0xCFFF FFFF	256 MB	NS	0xC
	0xD000 0000	0xDFFF FFFF	256 MB	S	0xD
System address space	0xE000 0000	0xE00F FFFF	1 MB	Exempt	0xE
	0xE010 0000	0xEFFF FFFF	255 MB	NS	0xE
	0xF000 0000	0xF00F 0000	1 MB	Exempt	0xF
	0xF010 0000	0xFFFF FFFF	256 MB	S	0xF

13.4.2 Secure Attribution Unit (SAU)

- SAU is a programmable unit and is a part of M33 CPU
- Configuration is only applicable to M33 CPU
- Supports eight programmable regions
- SAU is disabled at reset

- When the SAU is disabled, the address space attribution is controlled by SAU_CTRL.ALLNS bit field
- When the SAU is enabled, the eight programmable regions are used for marking memory as NS or NSC. Memory regions that are not covered by SAU are marked as Secure
- It is the job of secure software to configure SAU and to create NS and NSC regions that enables non-secure software execution

Table 103 shows an example SAU configuration.

Table 103 Example SAU configuration

Region	Purpose	Start address	End address	SAU security attribute	IDAU security attribute	Resultant security attribute
Code NS	Flash	0x02000000	0x0201FFFF	-	NS	S
		0x02020000	0x0207FFFF	NS		NS
	SRAM	0x04000000	0x04007FFF	-	NS	S
		0x04008000	0x0401FFFF	NS		NS
Code S	Flash	0x12000000	0x1201FEFF	-	NSC	S
		0x1201FF00	0x1201FFFF	NSC		NSC
		0x12020000	0x1207FFFF	-		S
	SRAM	0x14000000	0x1401FFFF	-	NSC	S
RAM NS	Flash	0x22000000	0x2201FFFF	-	NS	S
		0x22020000	0x2207FFFF	NS		NS
	SRAM	0x24000000	0x24007FFF	-	NS	S
		0x24008000	0x2401FFFF	NS		NS
RAM S	Flash	0x32000000	0x3207FFFF	-	NSC	S
	SRAM	0x34000000	0x3401FFFF	-	NSC	S
Peripheral NS	Peripherals	0x42000000	0x43FFFFFF	NS	NS	NS
Peripheral S	Peripherals	0x52000000	0x53FFFFFF		S	S
External memory NS	-	0x60000000	0x6FFFFFFF		NS	NS
External memory S	-	0x70000000	0x7FFFFFFF		S	S
External memory reserved	-	0x80000000	0x8FFFFFFF		NS	NS
		0x90000000	0x9FFFFFFF		S	S
External device reserved	-	0xA0000000	0xAFFFFFFF		NS	NS
		0xB0000000	0xBFFFFFFF		S	S
		0xC0000000	0xCFFFFFFF		NS	NS
		0xD0000000	0xDFFFFFFF		S	S
System address space	System control and debug	0xE0000000	0xE00FFFFF		Exempt	Exempt

(table continues...)

Table 103 (continued) Example SAU configuration

Region	Purpose	Start address	End address	SAU security attribute	IDAU security attribute	Resultant security attribute
		0xE0100000	0xEFFFFFFF		NS	NS
		0xF0000000	0xF00F0000		Exempt	Exempt
		0xF0100000	0xFFFFFFFF		S	S

13.5 State transition

Three instructions are defined for M33, to support security state transition:

- BXNS: Branch with exchange to non-secure
- BLXNS: Branch with link and exchange to non-secure
- SG: Secure gateway

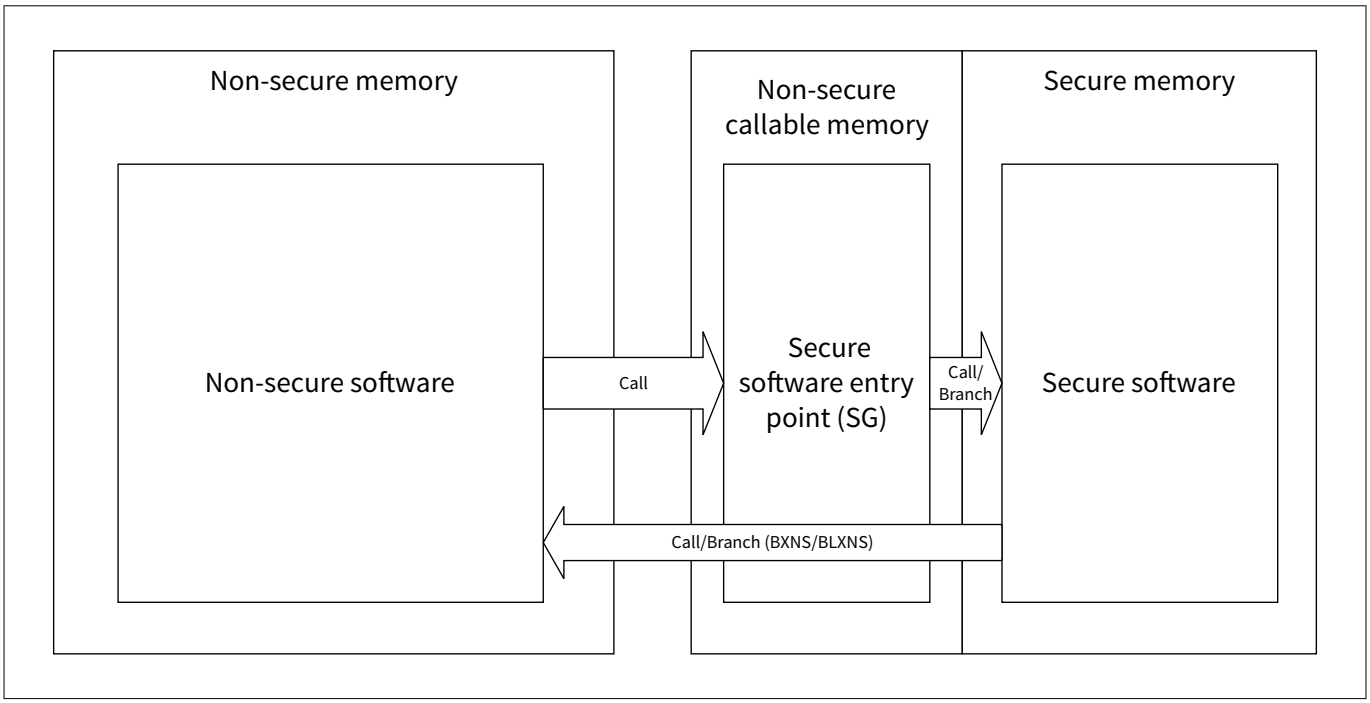


Figure 50 Security state transition

13.5.1 Secure to non-secure state transition

After a reset, the M33 CPU starts in the secure state. In the secure state, the CPU can only execute instructions from the secure memory. For the secure software to call non-secure software, the CPU state transition can be initiated with either BXNS or BLXNS instruction. Calling the BXNS instruction performs state transition and branches to non-secure address. BXLNS performs the state transition and calls a subroutine at a non-secure address.

These instructions are executed only when the CPU is in the secure state. BXNS/BLXNS are undefined if called from non-secure state.

13.5.2 Non-secure to Secure state transition

Non-secure software cannot directly access the Secure memory. For state transition SG instruction should be the first instruction executed from Non-secure Callable region. Successful execution of SG instruction allows CPU to transition to Secure state and now a call/branch to Secure software can be performed. Typically, Non-secure to Secure state transition is implemented to call a subroutine from Secure Software.

SG instruction should only be executed from Non-secure Callable region. If executed from Non-secure region the instruction behaves as a NOP.

13.5.3 State transition from exception or interrupt

The state transition can also take place due to an exception or interrupt. The exceptions and interrupts can be configured to the target either secure or non-secure state.

- If the CPU is in the secure state and an exception or interrupt targeting the non-secure state is triggered, this results in transition from the secure to the non-secure state
- If the CPU is in the non-secure state and an exception or interrupt targeting the secure state is triggered, this results in transition from the non-secure state to the secure state
- If the arriving interrupt has same state as the current CPU state, no state transition occurs

14 Protection units

The PSC3M/P6 MCU implements protection units to enforce security by allowing or restricting bus transfers based on specific properties of a transfer. The rules that determine protection are implemented in MMIO registers of protection units, which define the protected address space and the protection attributes. The hardware that evaluates these protection attributes to restrict or permit access is the protection unit. The PSC3M/P6 MCU devices have different types of protection units such as the Security Attribution Unit (SAU), Implementation Defined Attribution Unit (IDAU), Memory Protection Unit (MPU), Memory Protection Controller (MPC), Peripheral Protection Controller (PPC), and Master Security Controller (MSC). Each type of protection unit has a distinct set of MMIO registers that help define different protection regions and their attributes.

The PSC3M/P6 MCU platform is based on the Armv8-M architecture, and TrustZone security functions are part of this architecture. In a TrustZone-enabled system, software is divided into secure and non-secure domains. The processor handles the partitioning of the address space using the programmable SAU in combination with a device-specific logic block IDAU. For more details, see TrustZone.

While SAU and IDAU are essential, they may not be sufficient for partitioning the memory space into a large number of regions in security-driven applications. In such cases, it may be necessary to partition the address space into many more regions, such as at the page granularity. The MPC and PPC enable the partitioning of the memory space and the peripheral space into several regions. These units control the accessibility of each memory and peripheral region based on non-secure attributes and protection context. The MPU, a programmable unit part of the CPU, allows privileged software to define memory access permissions. Additionally, for bus masters without security extension, the MSC adds TrustZone capability.

The protection units allow or restrict bus transfers on the bus infrastructure based on access attributes such as:

- Read or Write attribute
- Execute attribute to distinguish a code access from a data access
- User or Privileged attribute to distinguish, for example, OS or kernel access from a task or thread access
- Secure or Non-secure attribute to distinguish a ‘Secure’ access from a 'Non-secure' access
- Protection context attribute to distinguish access from different protection contexts

The functioning of a secure system is based on the following components:

- **Bus masters**: This term refers to the bus masters in the architecture. In the PSC3M/P6 MCU device, examples of bus masters include a Cortex®-M core, DW, or the test controller
- **Protection units**: These are hardware blocks that implement the protection scheme to protect memory, peripherals, and shared resources from bus masters. They enforce the protection defined in the protection unit's MMIO registers
- **Protection MMIO registers**: MMIO registers form a register structure in memory that establishes rules based on which each protection unit evaluates a transfer. Each protection unit is associated with a register structure. An MMIO register structure describes the following:
  - A memory region on which the rule is applied. A memory region can be defined with either Start/End address or Start/End block index
  - A set of protection attributes, which are properties based on which a transfer is evaluated. These include:
    - R/W/X
    - User or Privileged
    - Secure or Non-secure
    - Protection context

The set of protection attributes that can be configured in MMIO register depends on the protection unit it is associated with.

Each bus master is associated with its own access attributes, which define the bus master's access privileges. A bus master is associated with attributes such as secure/non-secure, protection context, and user/privileged

attribute. These are dynamic attributes that change based on the bus master's context and state. However, for some bus masters, attributes such as secure/non-secure and user/privileged attribute are fixed and cannot be changed.

Protection attributes appear in two places:

- Protection MMIO registers
- Bus master's protection attribute

The protection units evaluate these access attributes of masters against those of memory/peripheral access attributes. If the access attributes match (bus master and memory/peripheral), the protection units allow the transfer; otherwise, the transfer is blocked.

Protection units address the following:

- **Security requirements**: This includes preventing malicious attacks from accessing secure memory or peripherals
- **Safety requirements**: This includes detecting accidental (non-malicious) software errors and random hardware errors. It is important to enable failure analysis to investigate the root cause of a safety violation

14.1 Architecture

When a bus master accesses a memory or peripheral, the access is evaluated by a protection unit against the protection attributes of the memory or peripheral location being accessed. If the bus master’s protection attributes satisfy the protection attributes of the memory or peripheral, the protection unit then allows the access. In the case of an access violation, a fault condition is triggered, resulting in a bus error. MPC and PPC allow configuring the error response to either a Bus Error or Read-Zero Write Ignore (RAZ/WI).

The MPU, SAU, and IDAU are part of the M33 CPU subsystem and protect the entire 4-GB address space. For more information on SAU and IDAU, see Attribution units. MPCs are also part of the M33 CPU subsystem and protect memories from illegal access. A separate protection unit type, PPC, is provided for peripheral protection within the peripheral address space. Additionally, the Master Security Controller (MSC) is implemented for non-CPU bus masters without security extension to add TrustZone capability.

- **MPU**: The M33 bus master has dedicated MPUs and are implemented as part of the CPU, under the control of privileged software. Typically, MPU access attributes are dynamically updated by privileged software such as RTOS/TF-M SPM
- **MPC**: MPCs are designed for implementing protection in scenarios with multiple bus masters. They implement protection based on a concept called Protection Context, allowing them to distinguish between different protection contexts and secure from non-secure accesses, thereby providing effective protection in a multi-master scenario
- **PPC**: PPCs are protection units provided in the peripheral register space for peripheral protection. Their attributes are similar to MPCs, except that they are intended to protect peripheral spaces. PPCs distinguish between different protection contexts and secure from non-secure accesses, as well as user mode accesses from privileged mode accesses. They implement protection for fixed address regions corresponding to specific peripherals
- **MSC**: MSCs are instantiated to provide security extension attributes for adding TrustZone capability. They set up the master’s security attributes and are paired with masters without security extension to protect the transactions initiated on the bus. Examples of masters include DW and Crypto

The PSC3M/P6 MCU protection architecture consists of the following components:

Figure 51 gives an overview of the PSC3M/P6 MCU protection architecture.

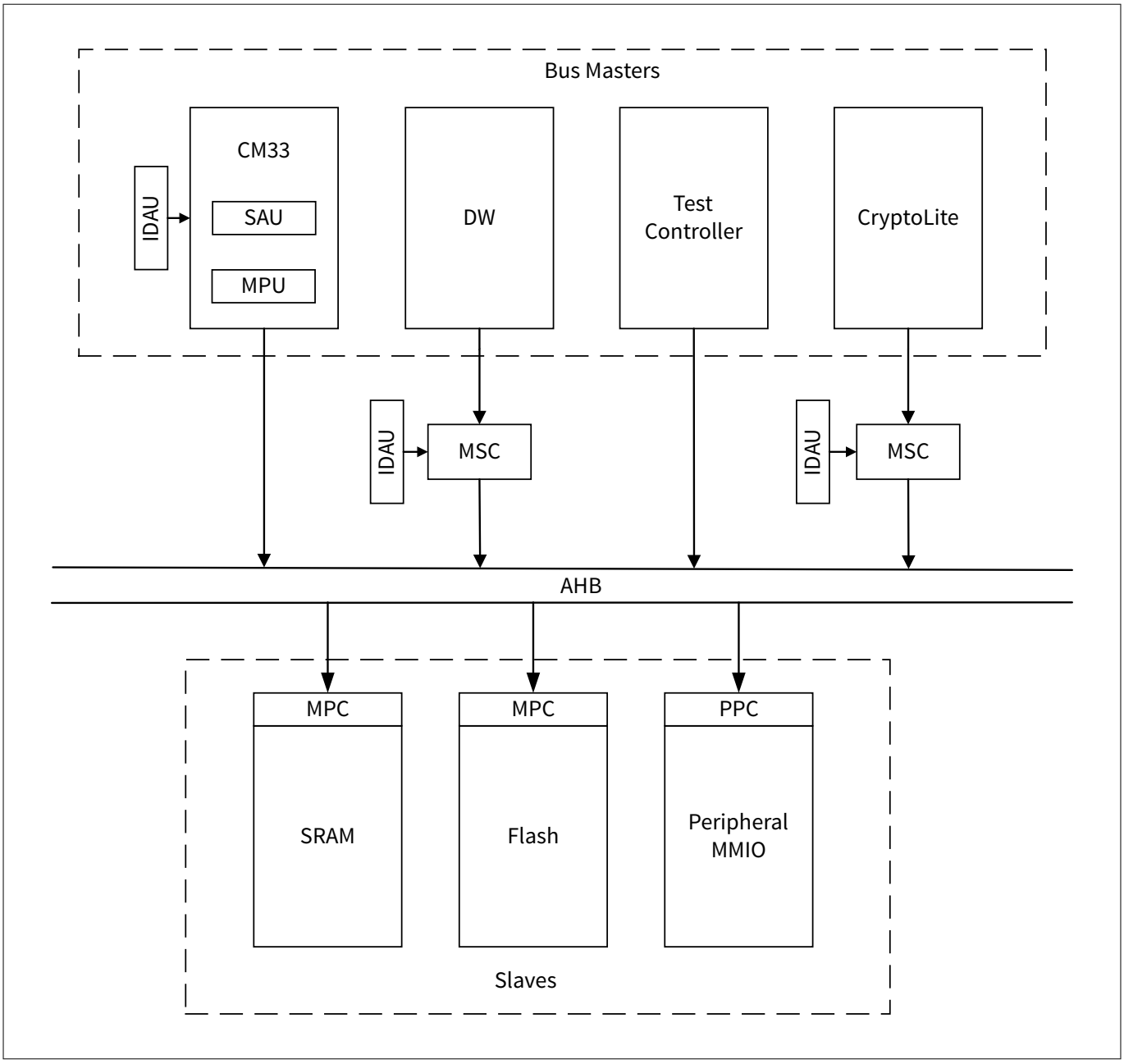


Figure 51 Protection architecture

A protection violation is caused by a mismatch between a bus master's access attributes and the protection attributes for the memory or peripheral region. These violations can be captured in the fault report structure to allow for failure analysis. The fault report structures are capable of generating an interrupt to indicate the occurrence of a fault. This is particularly useful if the violating bus master cannot resolve the bus error by itself but requires another CPU bus master to resolve the bus error on its behalf. A protection violation results in a bus error, causing the bus transfer to be halted before reaching its intended target.

Figure 52 shows the attribute check precedence in the PSC3M/P6 MCU device.

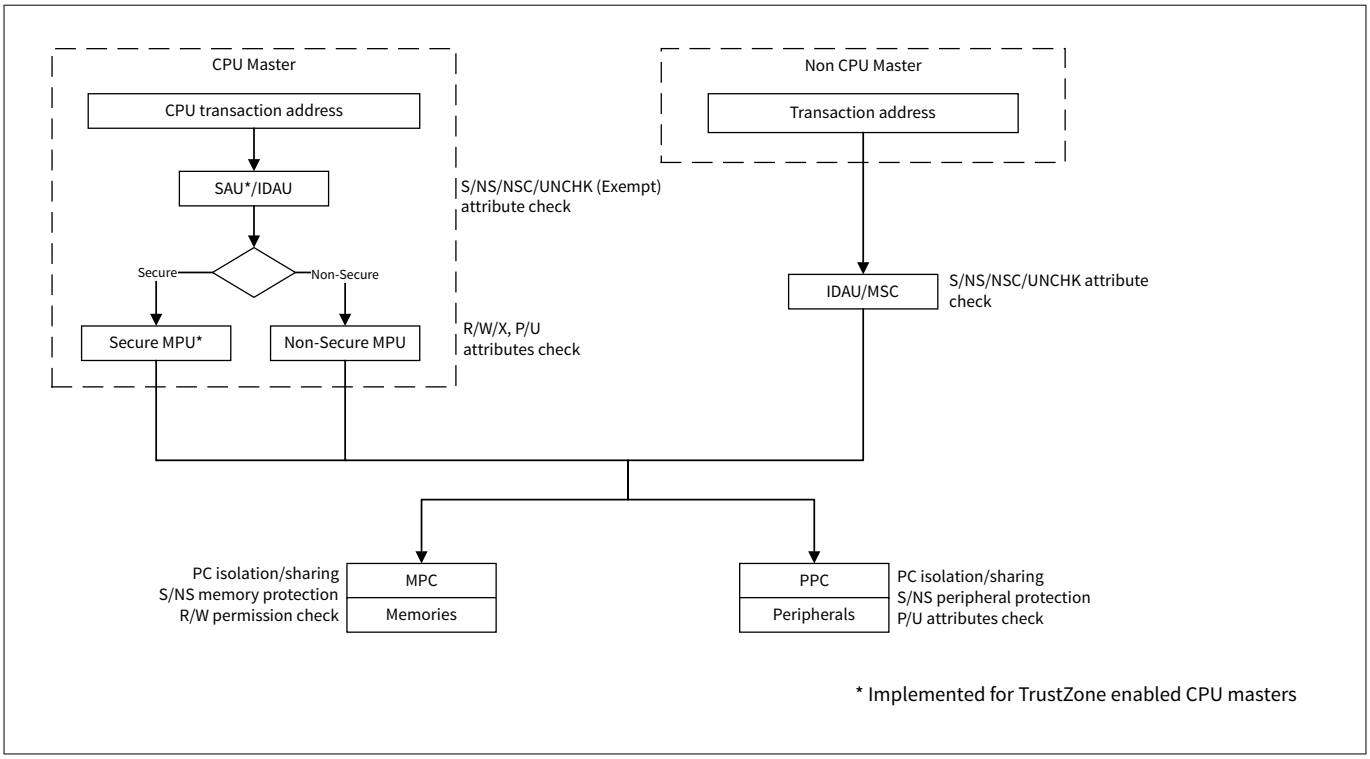


Figure 52 Attribute check precedence

The different types of protection units cater to different use cases, as described in the following table.

Table 104 PSC3M/P6 MCU protection units

Protection unit	Use	Number of instances
MPU	Used to protect memory between tasks within a single Arm® core. A task in one of the Arm® cores can protect its memory from access by another task in the same core.	M33 CPU: 1 Secure and 1 Non-secure MPU with 8 programmable regions for each MPU
MPC	Used to protect memory that is shared between multiple bus masters (with different protection contexts).	3
PPC	These units protect device-defined peripheral memory space. PPC has fixed regions and is used exclusively to protect the predefined peripheral address region (PPC region).	1
SAU	Used for partitioning the address space to isolate Secure and Non-Secure resources.	1
IDAU	Implements fixed partitioning of the address space to isolate Secure and Non-Secure resources.	3
MSC adapter	Used to add TrustZone capability to masters without security extension	2

14.2 Protection context

Protection context is an Infineon proprietary protection mechanism that enables the enforcement of access restrictions to memory and peripherals shared among multiple bus masters, as well as from the same bus master. Each bus master has a protection context field MS_PCx_PC.PC[3:0], which is utilized as the protection context attribute for all bus transfers initiated by the master. The MPCs and PPCs utilize this protection context attribute to allow or restrict bus transfers between a bus master and memory or peripheral.

Programming the master's MS_PCx_PC.PC[3:0] (PC[ ]) field allows for the setting of a bus master's protection context. Changes to the protection context are necessary for bus masters that may transition between multiple tasks, each catering to different protection contexts. As the protection context attribute enables or restricts bus transfers, changes to the protection context field should be carefully controlled to avoid compromising security. Each bus master has an MSx_CTL.PC_MASK[7:0] (PC_MASK[ ]) protection context mask field that identifies the allowed protection contexts for the bus master. This bit field is a 'one-hot' field specifying the allowed protection contexts for a bus master. Multiple bus masters can share a protection context

The protection context field PC[ ] indicates the active protection context (PC), and its modifications are restricted by the associated PC_MASK[ ] value. PC[ ] can only be set to 'i' if the corresponding mask bit PC_MASK[i] is '1'.

For example, when attempting to program MS_PC_STRUCT[x].PC[3:0] to '4':

- If PC_MASK[4] is '1', PC[ ] is set to '4'
- If PC_MASK[4] is '0', PC[ ] is not changed

The PSC3M/P6 MCU supports eight protection contexts, with each context predefined for a specific task. Table 105 shows the PC assignment for each task.

Table 105 Protection context assignment to firmware layers

PC	Firmware/software layer
0	ROM_BOOT/FLASH_BOOT
1	-
2	Protected firmware/OEM_BL/TF-M/OEM App
3-6	OEM App
7	SYS-AP (MXDEBUG600)

Table 106 specifies the bus master's protection attributes configured during device boot.

Table 106 Master security attributes

Master name	Master ID	MSx_CTL value			MS_PCx_PC value
		Privileged attribute	NS attribute	PC_MASK	Active PC
M33 CPU	0	-	-	PC2, PC3, PC4, PC5, PC6	PC2
DW	4, 5	Inherited			
CRYPTOLITE	7	Inherited			
Test controller	31	-	-	PC7	PC7

For masters that provide their own access attributes, the corresponding bit fields in the MSx_CTL register structure are not implemented. For instance, the M33 CPU's MS0_CTL does not have P and NS bit fields, as it provides its own P and NS attributes.

***Note**: The DW controller and Crypto accelerator inherit the access control attributes of the bus transfer(s) that programmed the channels or components.*

14.2.1 Special protection contexts

The active PC of the M33 CPU bus master can be changed by writing to the PC[ ] bit field. Access to this field is typically restricted to PC2 secure software, and the active PC cannot be directly modified by any other software. In cases where a switch to PC2/3 from PC4/5/6 is required, it can only be performed with a handler mechanism.

Out of the eight protection contexts PC2 and PC3 are treated as special: the entry to these special PCs is hardware-controlled. For special PCx (where x refers to the special PC number), a programmable exception handler address is provided at MXCM33_CM33_PCx_HANDLER.ADDR[31:0]. The bit field MXCM33_CM33_PC_CTL.VALID[x] indicates whether the handler for PCx, provided in MXCM33_CM33_PCx_HANDLER.ADDR[31:0], is valid. If a CPU exception handler fetch returns a handler address that matches the programmed MXCM33_CM33_PCx_HANDLER.ADDR[31:0] address value, it results in the CM33 PC being changed to PCx by the hardware. If a PC's MXCM33_CM33_PC_CTL.VALID[x] is not set, the PC is treated as an ordinary PC.

Writing to MXCM33_CM33_PCx_HANDLER.ADDR[31:0] and MXCM33_CM33_PC_CTL.VALID[x] bit fields from an OEM application is prohibited. These registers can only be programmed via the OEM provisioning process. The OEM policy provides a JSON object for programming these registers.

For the M33 bus master, the MS_PC0_PC register includes an additional MS_PC0_PC.PC_SAVED[19:16] bit field that holds the saved PC. The current PC and a saved PC implement a two-entry stack. Upon entry of a special exception handler, the hardware pushes the current PC to the stack. It is the responsibility of the exception handler to restore the active PC and saved PC before exiting from the handler.

When PCx is controlled by hardware (indicated by the bit MXCM33_CM33_PC_CTL.VALID[x] being ‘1’), the protection context cannot be selected by the software. In this case, the corresponding bit of the data written to PC_MASK[x] is forced to ‘0’. Note that PC switching with a handler is implemented only for the M33 CPU and not for other bus masters.

14.3 Master protection attributes

The protection attributes in protection unit's MMIO registers establish rules for different memory regions and their access attributes. The protection units use the bus master's own protection attributes to regulate access based on the rules set in the protection unit's MMIO registers. Not all bus masters provide all protection attributes associated with a bus transfer. For example, none of the bus masters have a native protection context attribute, which should be dynamically set based on the task executed by the bus master.

To ensure system-wide restricted access, the missing attributes are provided in the bus master's MMIO register fields. Some of these fields are set during the ROM boot process and cannot be altered by the user code.

- The PC_MASK[] and PC[] register fields provide protection context functionality
- The MSx_CTL.P register field provides the unprivileged or privileged attribute for masters that do not provide their own attribute
- The MSx_CTL.NS register field provides the secure or non-secure attribute for masters that do not provide their own attribute
- The execute attribute of masters that do not provide the execute attribute is set to '0'

See Table 106 for the bus master protection attribute set by the ROM boot code.

14.3.1 Master Security Controller (MSC)

The AHB5 TrustZone Master Security Controller (MSC) serves as an adapter for connecting a legacy AHB-lite master or an AHB5 master without the security extension to an AHB5 system and adding TrustZone for Armv8-M capability. A master without a security extension can be configured as secure or non-secure by either programming the NS bit field, which is a part of the MSx_CTL register, or by inheriting the attribute from the master that programmed it. The MSC evaluates the NS attribute of a security-unaware master against IDAU. If the master has the appropriate NS attribute, access is allowed; otherwise, it is blocked. The MSC can be programmed to behave RZWI or generate an MSC interrupt in case of access violation.

For additional details about the MSC, see the CoreLink SIE-200 technical reference manual from Arm®.

14.3.1.1 Memory Protection Unit (MPU)

The Armv8-M memory protection units (MPUs) are programmable units and are part of the CPU. They allow the privileged software to define memory access permissions (R/W/X) and memory attributes for normal memory or device memory, and also monitors instruction fetch and data access operations from the CPU. In the PSC3M/P6 MCU platform, the M33 CPU is equipped with one secure MPU and one non-secure MPU with eight programmable regions for each MPU. Any access violation observed by the MPU triggers a MemManage fault.

For more information on the architecture and registers, see the Armv8-M memory model and memory protection unit (MPU) user guide from Arm®.

14.4 Slave protection attributes

The MPU, MPC, and PPC evaluate the access attributes of memory and peripheral slaves before every transaction with a master.

14.4.1 Memory Protection Controller (MPC)

The memory protection controller (MPC) is located on the AHB bus. The SRAM and Flash are equipped with the MPC.

The MPC provides the following functionality:

- Allocates protection context (PC) based on run-time memory separation or isolation and sharing
- Acts as an Armv8-M TrustZone-M compliant memory protection controller between the secure and non-secure world
- Makes the memory accessible and visible either in its secure address alias or its non-secure address alias
- Divides the memory space into fixed-sized blocks/pages, with SRAM blocks being 2 KB and Flash blocks being 2 KB
- Configures the NS/R/W attributes of each block for all PCs
- Blocks any non-secure access targeting the secure region
- Blocks any secure access targeting the non-secure region
- Offers configurable error responses:
  - MPCx_CFG.RESPONSE=1’b0: hardware performs Read Zero Write Ignore (RZWI)
  - MPCx_CFG.RESPONSE=1’b1: hardware provides bus error

The MPC comprises MMIO registers that program the LUT_SRAM. The LUT_SRAM contains a look-up-table of the security attributes and is connected to an attribution check logic block, which allows or blocks the transaction.

Figure 53 shows the architecture of the MPC.

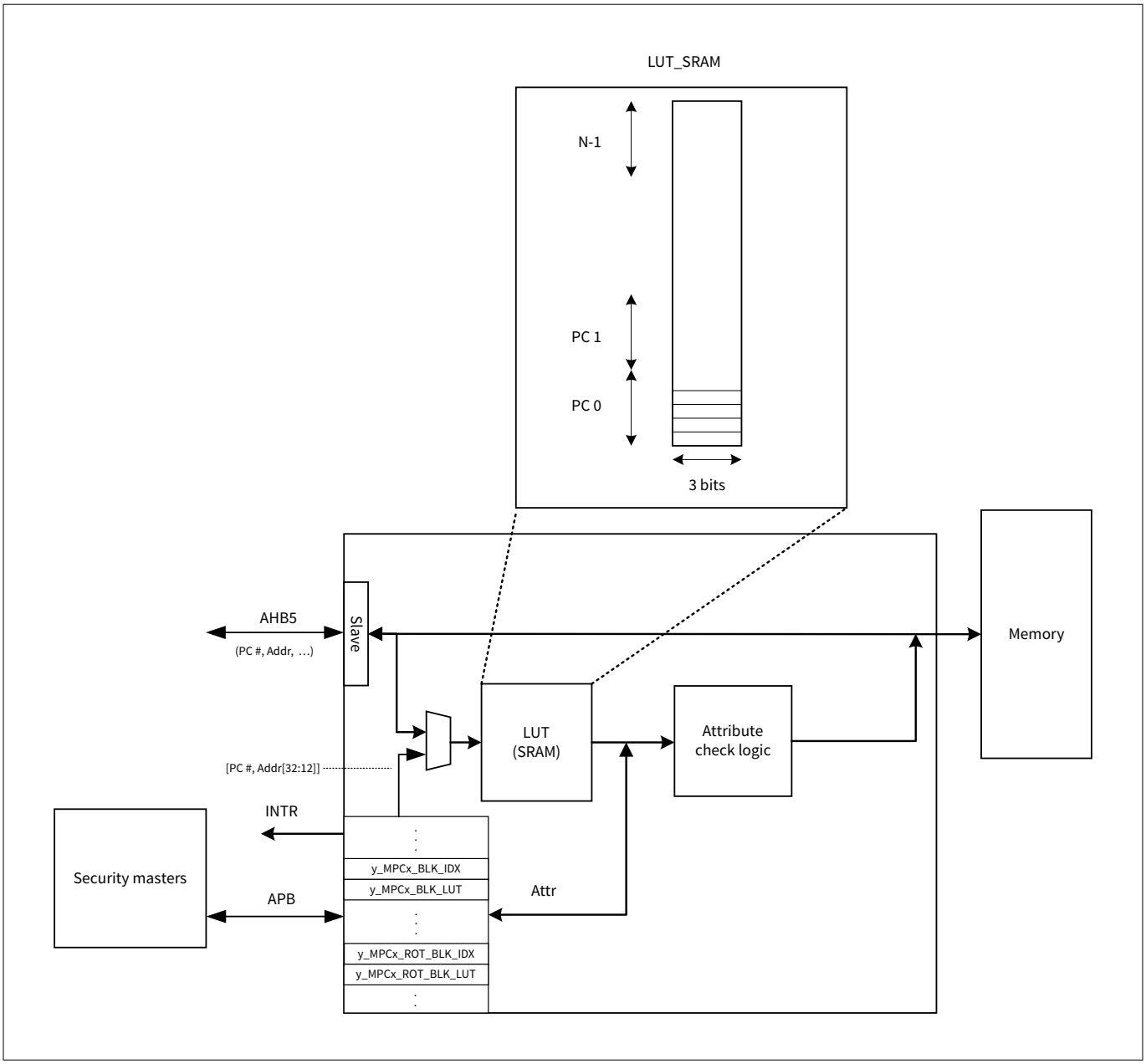


Figure 53 MPC architecture

***Note**: The Secure/Non-secure (NS) attribute distinguishing a 'secure' access from a 'non-secure' access, is a reflection of the resultant security attribute of the location being accessed. For a data access, if a CPU in the S state makes an access to an NS location, the access is deemed as NS. Conversely, if the transaction is made to an S location, the access is deemed as S. Similarly, a CPU in the NS state making a data access to an NS location is deemed as NS. Likewise, for an instruction access, the Secure/Non-secure (NS) access attribute represents the security attribute of the memory location being accessed.*

14.4.1.1 Software view

The MPC includes a look-up table comprised of SRAM (SRAM_LUT) containing 3-bit NS/R/W attributes per memory block for all PCs. For each PC, NSx1, Rx1, Wx1 attribute bits per memory block are reserved and stored

in the LUT_SRAM. The total number of LUT_SRAM bits is 3 * PC_NR * the number of blocks for a memory module, with the maximum supported PC_NR being 8.

The FLASH MPC configuration is part of the OEM policy and is applied by the Boot ROM. At runtime, the OEM application can modify only the NS attribute. The R/W attributes are configured by the secure Boot ROM software and are locked.

The SRAM MPC configuration is not part of the OEM policy. Boot ROM does not apply any configuration, and the OEM application can fully configure all NS/R/W attributes at runtime.

Upon reset, the hardware state machine initializes the LUT_SRAM into secure memory with R/W access for all PCs. All registers, as well as LUT_SRAM, are not retained in DEEPSLEEP_RAM/OFF power modes and need to be restored after a wake-up event.

Figure 54 shows the software view of the MPC.

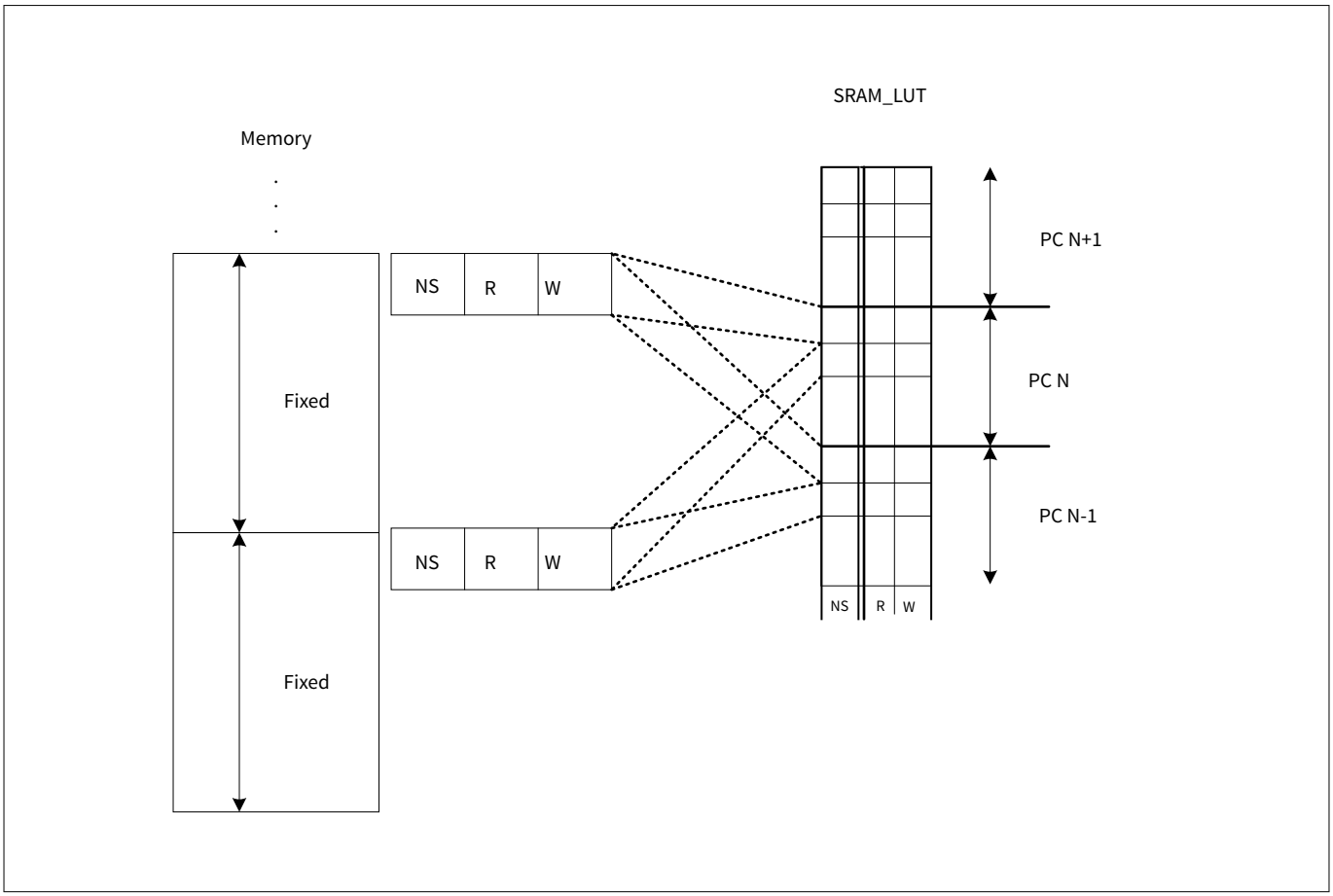


Figure 54 MPC software view

Figure 55 shows the MPC register structure.

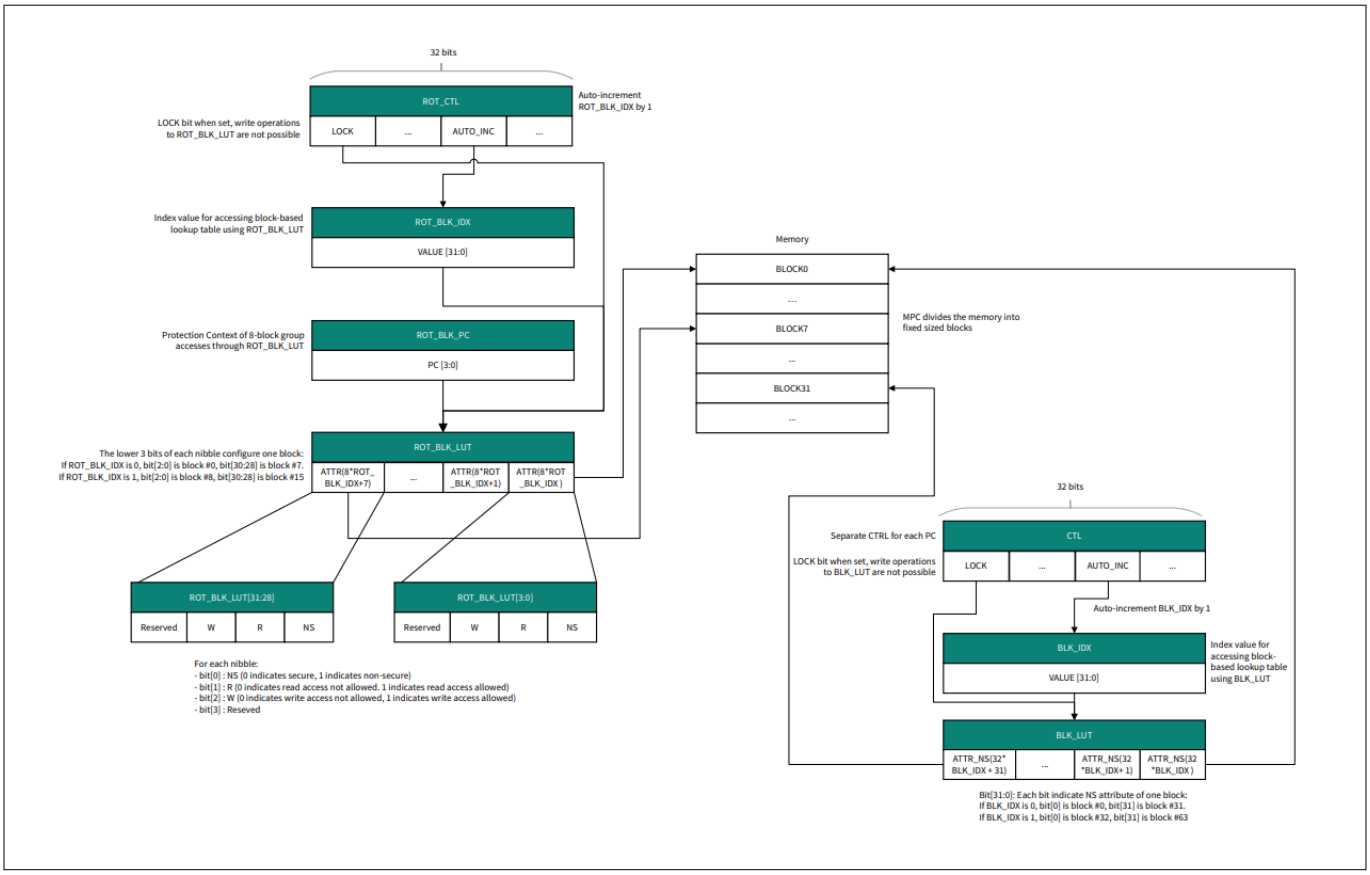


Figure 55 MPC MMIO registers

14.4.1.2 MPC MMIO register protection

The MPC MMIO registers are protected by the PPC, allowing only trusted software to make modifications to the MPC configuration. The x_PPC_MPC_MAIN, x_PPC_MPC_PC, and x_PPC_MPC_ROT PPC regions control access to the MPC MMIO registers. In the PSOC™ C3 P6/M6 platform, subset of these MPC PPC regions are configured by ROM in a manner that restricts modification of the MPC configuration to only the ROM code executing in PC0, while the remaining regions can be configured by application code executing in PC2/3/4/5/6/7. See Table 108 for the list of regions only accessible by PC0.

14.4.2 Peripheral Protection Controller (PPC)

The PPC provides peripheral access control based on Arm® TrustZone-M secure or non-secure (NS), privileged or unprivileged (P), and protection context (PC) attributes. Situated between bus masters and peripherals, the PPC groups the MMIO registers of a peripheral that typically require the same security permissions into ‘regions.’ For example, the registers of SCB0 are grouped together in one region. Each PPC region can be configured with PC/NS/P attributes, and a peripheral can have multiple PPC regions depending on its access control needs. The PPC checks the security attribute of a peripheral region against the transaction attribute from the masters for a security violation. If the transaction has a security violation, it is then blocked.

The PPC provides the following functionality:

- TrustZone-compatible peripheral protection with AHB5 bus interface
- Blocking non-secure transfers to secure regions and secure transfers to non-secure regions are considered as violations

- Blocking non-privileged transfers if the peripheral region is configured as privileged
- Configurable error responses:
  - Read Zero Write Ignore (RZWI)
  - Bus fault

14.4.3 PPC access control rules

The PPC configuration is primarily controlled by the following registers:

Table 107 PPC registers

Register name	Function
LOCK_MASK	This register provides a lock mechanism for each PC.
PC_MASK	This register indicates the PCs that have access to the peripheral region, with one PC_MASK for each peripheral region.
NS_ATT	This register indicates whether a peripheral region is non-secure or secure. Each bit corresponds to a peripheral region.
S_P_ATT	This register indicates the secure privileged or unprivileged status for a peripheral region, with each bit corresponding to a peripheral region.
NS_P_ATT	This register indicates the non-secure privileged or unprivileged status for a peripheral region, with each bit corresponding to a peripheral region.

The following convention is used in the subsequent section to describe the PPC restrictions, where "i" stands for a PC value and "j" stands for a peripheral region.

- After a reset or power cycle, PC_MASK bits are by default set to high, S_P_ATT and NS_ATT bits set to low, meaning secure privileged access is provided to all PCs
- If PC_MASK[j].PC_MASK[i]=0x0 and if LOCK_MASK[i]=0, PC "i" can have Read and Write access to PC_MASK[j], NS_ATT[j], NS_P_ATT[j], S_P_ATT[j]
- LOCK_MASK[i] can only be written from the corresponding PC "i". The LOCK_MASK bits that are accessed by PC "i" and are not associated with PC "i" return RZWI
- PC "i" has Read or Write access to the peripheral region "j" if PC_MASK[j].PC_MASK[i] = 1’b1
- If (LOCK_MASK[i] & PC_MASK[j].PC_MASK[i]) == 1’b1:
  - The peripheral region "j" security attributes cannot be changed (with exception on the NS_P_ATT registers). This means that PC "i" is “locked” as its corresponding LOCK_MASK bit is set to 1’b1
  - PC "i" that is 'locked' cannot write to any PC_MASK
  - PC "i" that is 'locked' cannot be given additional access to any peripheral region nor be removed from the peripheral region
  - Additionally, for any PC "k", if PC_MASK[j].PC_MASK[k] is set to 1’b1, PC "k" can only clear the PC_MASK that is not 'locked', that is, remove access from other PCs to the peripheral region "j". PC "k" cannot provide additional access to other PCs as PC "i" has a 'locked' peripheral region "j"
- If LOCK_MASK[i]==1’b1 and if PC_MASK[j].PC_MASK[i]=0x0:
  - PC "i" is subjected to the 'locked' condition specified in the requirements above and does not have Read or Write access to the peripheral region "j". PC "i" cannot change the security attributes for the peripheral region "j" but PC "i" has read access to the security attributes for the peripheral region "j"

Figure 56 shows the PPC MMIO register structure.

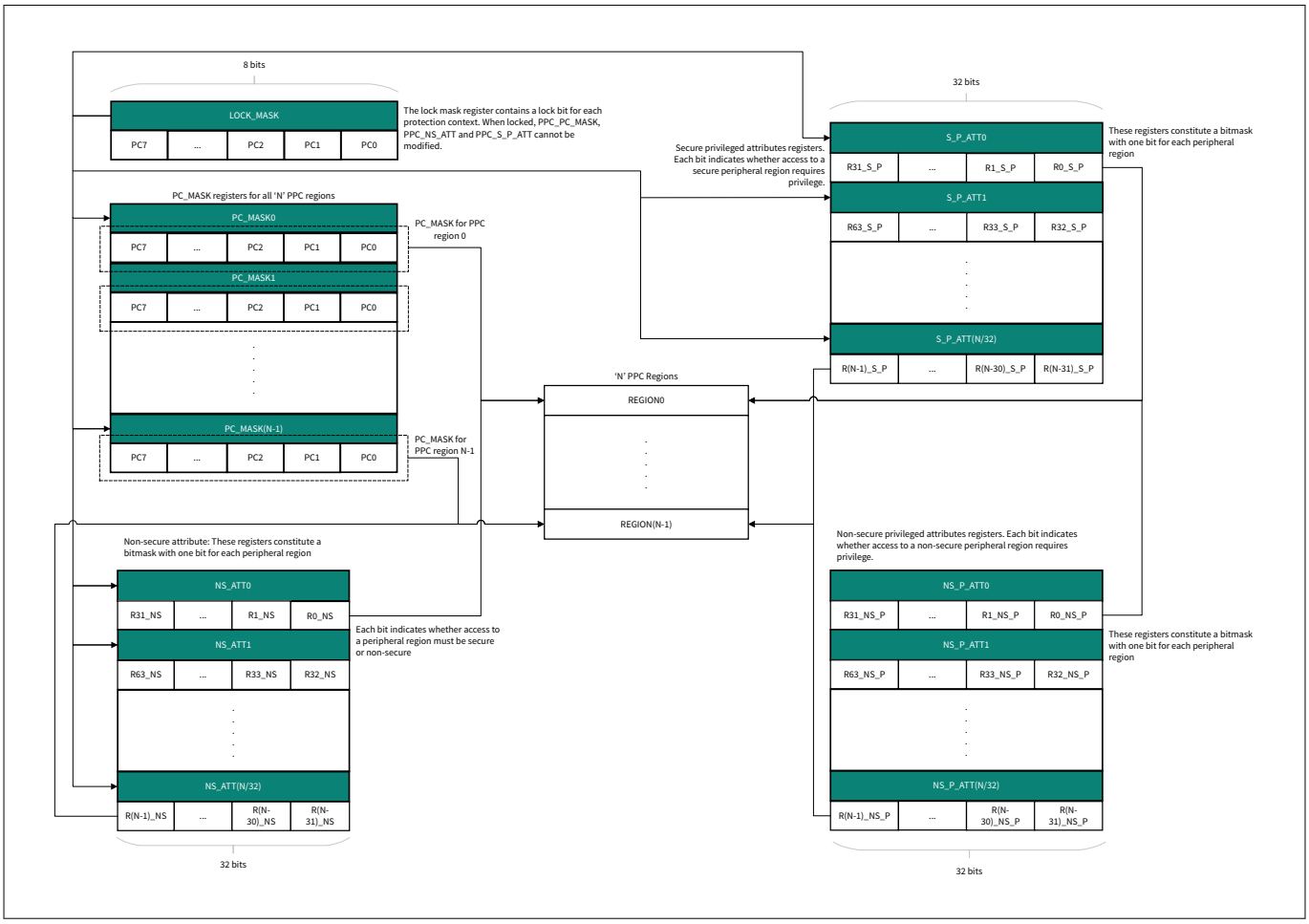


Figure 56 PPC MMIO structure

The ROM boot code configures and locks some of the PPC regions, which are shared with PC0. See Table 108 for a list of regions shared with PC0.

For the PPC register definition and programming model, see the PSC3M/P6 MCU register reference manual.

14.4.3.1 Security aware peripherals

In PSC3M/P6 MCU, peripherals are categorized as security aware or security unaware. The security unaware PPC peripheral region (SECURITY_AWARE = 0) adheres to the following PPC rules:

- Secure transactions can access secure regions only, and non-secure transactions can access non-secure regions only

The security aware PPC peripheral region (SECURITY_AWARE = 1) adheres to the following PPC rules:

- NS = 0: This region allows only secure transactions to access secure resources
- NS = 1: This region allows secure transactions to access both secure and non-secure resources, and non-secure transactions can access only non-secure resources. The expectation is that the peripheral itself can perform the security check, and hence the PPC allows both secure and non-secure transactions to pass through if other attributes match

For a list of security aware peripherals, see Table 108

14.4.3.2 PPC regions

The PPC not only protects the peripherals but also the MMIO registers of other blocks, such as MPC, peripheral interconnect, SRSS, and so on. There are a total of 267 regions in the PPC. The ROM boot code configures the NS_ATT, S_P_ATT, NS_P_ATT, and PC_MASK registers.

The ROM boot code configures the attributes of all regions of PPC in the following manner:

- NS_ATT = 0 - PPC region is secure
- S_P_ATT= 1 - Privilege and non-privilege access set for all but four regions. S_P_ATT is set to 0 for following four regions: PPC0_PPC_PPC_SECURE, PPC0_PPC_PPC_NONSECURE, CPUSS_AP, MS_PC0_PRIV
- NS_P_ATT = 1 for all regions
- Sets lock bit for PC0. PC2 software is responsible for configurations of the PPC access permissions and attributes for the rest of the PCs

Table 108 shows the list of all PPC0 regions and their PC_MASK configured by the boot code.

Table 108 PPC regions

PPC region number	PPC region name	Start address (hex)	Size (hex)	SECURITY_AWARE	Accessible by (PCx)
0	PERI0_MAIN	42000000	4000	0	2, 3, 4, 5, 6, 7
1	PERI0_GR0_GROUP	42004010	8	0	2, 3, 4, 5, 6, 7
2	PERI0_GR1_GROUP	42004040	20	0	2, 3, 4, 5, 6, 7
3	PERI0_GR2_GROUP	42004080	20	0	2, 3, 4, 5, 6, 7
4	PERI0_GR3_GROUP	420040c0	20	0	2, 3, 4, 5, 6, 7
5	PERI0_GR4_GROUP	42004110	8	0	2, 3, 4, 5, 6, 7
6	PERI0_GR5_GROUP	42004140	20	0	2, 3, 4, 5, 6, 7
7	PERI0_GR0_BOOT	42004020	4	0	0
8	PERI0_GR1_BOOT	42004060	4	0	0
9	PERI0_GR2_BOOT	420040a0	4	0	0
10	PERI0_GR3_BOOT	4.20E+05	4	0	0
11	PERI0_GR4_BOOT	42004120	4	0	0
12	PERI0_GR5_BOOT	42004160	4	0	0
13	PERI0_TR	42008000	8000	0	2, 3, 4, 5, 6, 7
14	PPC0_PPC_PPC_SECURE	42020000	4000	0	0, 2, 3
15	PPC0_PPC_PPC_NONSECURE	42024000	4000	0	0, 2, 3
16	PERI_PCLK0_MAIN	42040000	20000	0	2, 3, 4, 5, 6, 7
17	CPUSS	42100000	10000	0	2, 3, 4, 5, 6, 7
18	RAMC0_CM33	42110000	40	0	2, 3, 4, 5, 6, 7
19	RAMC1_CM33	42120000	40	0	2, 3, 4, 5, 6, 7
20	RAMC0_BOOT	42110100	8	0	0
21	RAMC1_BOOT	42120100	8	0	0
22	RAMC0_RAM_PWR	42110200	100	0	2, 3, 4, 5, 6, 7

(table continues...)

Table 108 (continued) PPC regions

PPC region number	PPC region name	Start address (hex)	Size (hex)	SECURITY_AWARE	Accessible by (PCx)
23	RAMC1_RAM_PWR	42120200	100	0	2, 3, 4, 5, 6, 7
24	RAMC0_MPC0_PPC_MPC_MAIN	42114000	4	0	2, 3, 4, 5, 6, 7
25	RAMC1_MPC0_PPC_MPC_MAIN	42124000	4	0	2, 3, 4, 5, 6, 7
26	RAMC0_MPC0_PPC_MPC_PC	42114100	20	0	2, 3, 4, 5, 6, 7
27	RAMC1_MPC0_PPC_MPC_PC	42124100	20	0	2, 3, 4, 5, 6, 7
28	RAMC0_MPC0_PPC_MPC_ROT	42114200	20	0	2, 3, 4, 5, 6, 7
29	RAMC1_MPC0_PPC_MPC_ROT	42124200	20	0	2, 3, 4, 5, 6, 7
30	PROMC_CM33	42140000	4	0	2, 3, 4, 5, 6, 7
31	PROMC_MPC0_PPC_MPC_MAIN	42141000	4	0	0
32	PROMC_MPC0_PPC_MPC_PC	42141100	20	0	0
33	PROMC_MPC0_PPC_MPC_ROT	42141200	20	0	0
34	FLASHC_BOOT	42150000	8	0	2, 3, 4, 5, 6
35	FLASHC_BOOT1	42150100	20	0	2, 3, 4, 5, 6, 7
36	FLASHC_MAIN	42150200	10	0	2, 3, 4, 5, 6
37	FLASHC_DFT	42150400	80	0	0
38	FLASHC_ECC	42150800	10	0	2, 3, 4, 5, 6
39	FLASHC_MPC0_PPC_MPC_MAIN	42151000	4	0	2, 3, 4, 5, 6, 7
40	FLASHC_MPC0_PPC_MPC_PC	42151100	20	0	2, 3, 4, 5, 6, 7
41	FLASHC_MPC0_PPC_MPC_ROT	42151200	20	0	0
42	FLASHC_FM_CTL_FM_DFT	42152000	4	0	0
43	FLASHC_FM_CTL_FM_BOOT	42152040	8	0	0
44	FLASHC_FM_CTL_FM_MAIN	42152800	800	0	2, 3, 4, 5, 6
45	MXCM33_CM33	42160000	100	0	2, 3, 4, 5, 6, 7
46	MXCM33_CM33_S	42161000	4	0	2, 3, 4, 5, 6, 7
47	MXCM33_CM33_NS	42161004	4	0	2, 3, 4, 5, 6, 7
48	MXCM33_BOOT_PC0	42162000	80	0	0
49	MXCM33_BOOT_PC1	42162100	4	0	0
50	MXCM33_BOOT_PC2	42162140	4	0	0
51	MXCM33_BOOT_PC3	42162180	4	0	0

(table continues...)

Table 108 (continued) PPC regions

PPC region number	PPC region name	Start address (hex)	Size (hex)	SECURITY_AWARE	Accessible by (PCx)
52	MXCM33_BOOT	421621c0	4	0	0
53	MXCM33_CM33_INT	42168000	400	0	2, 3, 4, 5, 6, 7
54	DW0_DW	42180000	80	0	2, 3, 4, 5, 6, 7
55	DW1_DW	42190000	80	0	2, 3, 4, 5, 6, 7
56	DW0_DW_CRC	42180100	80	0	2, 3, 4, 5, 6, 7
57	DW1_DW_CRC	42190100	80	0	2, 3, 4, 5, 6, 7
58	DW0_CH_STRUCT0_CH	42188000	40	0	2, 3, 4, 5, 6, 7
59	DW0_CH_STRUCT1_CH	42188040	40	0	2, 3, 4, 5, 6, 7
60	DW0_CH_STRUCT2_CH	42188080	40	0	2, 3, 4, 5, 6, 7
61	DW0_CH_STRUCT3_CH	421880c0	40	0	2, 3, 4, 5, 6, 7
62	DW0_CH_STRUCT4_CH	42188100	40	0	2, 3, 4, 5, 6, 7
63	DW0_CH_STRUCT5_CH	42188140	40	0	2, 3, 4, 5, 6, 7
64	DW0_CH_STRUCT6_CH	42188180	40	0	2, 3, 4, 5, 6, 7
65	DW0_CH_STRUCT7_CH	421881c0	40	0	2, 3, 4, 5, 6, 7
66	DW0_CH_STRUCT8_CH	42188200	40	0	2, 3, 4, 5, 6, 7
67	DW0_CH_STRUCT9_CH	42188240	40	0	2, 3, 4, 5, 6, 7
68	DW0_CH_STRUCT10_CH	42188280	40	0	2, 3, 4, 5, 6, 7
69	DW0_CH_STRUCT11_CH	421882c0	40	0	2, 3, 4, 5, 6, 7
70	DW0_CH_STRUCT12_CH	42188300	40	0	2, 3, 4, 5, 6, 7
71	DW0_CH_STRUCT13_CH	42188340	40	0	2, 3, 4, 5, 6, 7
72	DW0_CH_STRUCT14_CH	42188380	40	0	2, 3, 4, 5, 6, 7
73	DW0_CH_STRUCT15_CH	421883c0	40	0	2, 3, 4, 5, 6, 7
74	DW1_CH_STRUCT0_CH	42198000	40	0	2, 3, 4, 5, 6, 7
75	DW1_CH_STRUCT1_CH	42198040	40	0	2, 3, 4, 5, 6, 7
76	DW1_CH_STRUCT2_CH	42198080	40	0	2, 3, 4, 5, 6, 7
77	DW1_CH_STRUCT3_CH	421980c0	40	0	2, 3, 4, 5, 6, 7
78	DW1_CH_STRUCT4_CH	42198100	40	0	2, 3, 4, 5, 6, 7
79	DW1_CH_STRUCT5_CH	42198140	40	0	2, 3, 4, 5, 6, 7
80	DW1_CH_STRUCT6_CH	42198180	40	0	2, 3, 4, 5, 6, 7
81	DW1_CH_STRUCT7_CH	421981c0	40	0	2, 3, 4, 5, 6, 7
82	DW1_CH_STRUCT8_CH	42198200	40	0	2, 3, 4, 5, 6, 7
83	DW1_CH_STRUCT9_CH	42198240	40	0	2, 3, 4, 5, 6, 7
84	DW1_CH_STRUCT10_CH	42198280	40	0	2, 3, 4, 5, 6, 7

(table continues...)

Table 108 (continued) PPC regions

PPC region number	PPC region name	Start address (hex)	Size (hex)	SECURITY_AWARE	Accessible by (PCx)
85	DW1_CH_STRUCT11_CH	421982c0	40	0	2, 3, 4, 5, 6, 7
86	DW1_CH_STRUCT12_CH	42198300	40	0	2, 3, 4, 5, 6, 7
87	DW1_CH_STRUCT13_CH	42198340	40	0	2, 3, 4, 5, 6, 7
88	DW1_CH_STRUCT14_CH	42198380	40	0	2, 3, 4, 5, 6, 7
89	DW1_CH_STRUCT15_CH	421983c0	40	0	2, 3, 4, 5, 6, 7
90	CPUSS_ALL_PC	421c0000	80	0	2, 3, 4, 5, 6, 7
91	CPUSS_DDFT	421c0080	4	0	0
92	CPUSS_CM33_S	421c0100	4	0	2, 3, 4, 5, 6, 7
93	CPUSS_CM33_NS	421c0120	4	0	2, 3, 4, 5, 6, 7
94	CPUSS_MSC_INT	421c0200	10	0	2, 3, 4, 5, 6, 7
95	CPUSS_AP	421c1000	4	0	0, 2, 3
96	CPUSS_BOOT	421c2000	8	0	0
97	MS0_MAIN	421c4000	4	0	0
98	MS4_MAIN	421c4040	4	0	0
99	MS5_MAIN	421c4050	4	0	0
100	MS7_MAIN	421c4070	4	0	0
101	MS31_MAIN	421c41f0	4	0	0
102	MS_PC0_PRIV	421c5000	4	0	0, 2, 3,
103	MS_PC31_PRIV	421c51f0	4	0	2, 3
104	MS_PC0_PRIV_MIR	421c5004	4	0	2, 3, 4, 5, 6, 7
105	MS_PC31_PRIV_MIR	421c51f4	4	0	2, 3, 4, 5, 6, 7
106	MSC_ACG	421c6000	40	0	2, 3, 4, 5, 6, 7
107	CPUSS_SL_CTL_GROUP	421c8000	8	0	2, 3, 4, 5, 6, 7
108	IPC_STRUCT0_IPC	421d0000	20	1	2, 3, 4, 5, 6, 7
109	IPC_STRUCT1_IPC	421d0020	20	1	2, 3, 4, 5, 6, 7
110	IPC_STRUCT2_IPC	421d0040	20	1	2, 3, 4, 5, 6, 7
111	IPC_STRUCT3_IPC	421d0060	20	1	2, 3, 4, 5, 6, 7
112	IPC_INTR_STRUCT0_INTR	421d1000	10	1	2, 3, 4, 5, 6, 7
113	IPC_INTR_STRUCT1_INTR	421d1020	10	1	2, 3, 4, 5, 6, 7
114	FAULT_STRUCT0_MAIN	4.21E+02	100	0	2, 3, 4, 5, 6, 7
115	SRSS_GENERAL	42200000	400	0	2, 3, 4, 5, 6, 7
116	SRSS_GENERAL2	42200400	40	0	2, 3, 4, 5, 6, 7
117	SRSS_HIB_DATA	422008a0	10	0	2, 3, 4, 5, 6, 7

(table continues...)

Table 108 (continued) PPC regions

PPC region number	PPC region name	Start address (hex)	Size (hex)	SECURITY_AWARE	Accessible by (PCx)
118	SRSS_MAIN	42201000	1000	0	2, 3, 4, 5, 6, 7
119	SRSS_SECURE	42202000	2000	0	2, 3, 4, 5, 6, 7
120	RAM_TRIM_SRSS_SRAM	42204000	8	0	2, 3, 4, 5, 6, 7
121	SRSS_DPLL	42204200	20	0	2, 3, 4, 5, 6, 7
122	SRSS_WDT	4220c000	10	0	2, 3, 4, 5, 6, 7
123	MAIN	4220d000	40	0	2, 3, 4, 5, 6, 7
124	PWRMODE_PWRMODE	42210000	4000	0	2, 3, 4, 5, 6, 7
125	BACKUP_BACKUP	42220000	100	0	2, 3, 4, 5, 6, 7
126	BACKUP_B_BREG0	42221000	10	0	2, 3, 4, 5, 6, 7
127	BACKUP_B_BREG1	42221010	10	0	2, 3, 4, 5, 6, 7
128	BACKUP_B_BREG2	42221020	20	0	2, 3, 4, 5, 6, 7
129	BACKUP_B_BREG3	42221080	40	0	2, 3, 4, 5, 6, 7
130	BACKUP_BACKUP_SECURE	4222ff00	4	0	0
131	CRYPTOLITE_MAIN	42230000	100	0	2, 3, 4, 5, 6, 7
132	CRYPTOLITE_TRNG	42230100	100	0	2, 3, 4, 5, 6, 7
133	MXCORDIC_1_1	42240000	10000	0	2, 3, 4, 5, 6, 7
134	DEBUG600_DEBUG600	42250000	4	0	2, 3, 4, 5, 6, 7
135	SVGS_SVGS_MAIN	42260000	40	0	2, 3, 4, 5, 6, 7
136	SVGS_SVGS_AUX	42260040	40	0	0
137	MXAES_MAIN1	42270000	800	0	2, 3, 4, 5, 6, 7
138	MXAES_BOOT1	42270800	800	0	2, 3, 4, 5, 6, 7
139	MXAES_MAIN2	42271000	800	0	2, 3, 4, 5, 6, 7
140	MXAES_BOOT2	42271800	800	0	2, 3, 4, 5, 6, 7
141	MXWOUND_MAIN	42280000	4	0	0
142	MXWOUND_MXWOUND_PROP0_PROP	42280100	4	0	0
143	MXWOUND_MXWOUND_PROP1_PROP	42280140	4	0	0
144	HSIOM_PRT0_PRT	42400000	8	1	2, 3, 4, 5, 6, 7
145	HSIOM_PRT1_PRT	42400010	8	1	2, 3, 4, 5, 6, 7
146	HSIOM_PRT2_PRT	42400020	8	1	2, 3, 4, 5, 6, 7
147	HSIOM_PRT3_PRT	42400030	8	1	2, 3, 4, 5, 6, 7
148	HSIOM_PRT4_PRT	42400040	8	1	2, 3, 4, 5, 6, 7

(table continues...)

Table 108 (continued) PPC regions

PPC region number	PPC region name	Start address (hex)	Size (hex)	SECURITY_AWARE	Accessible by (PCx)
149	HSIOM_PRT5_PRT	42400050	8	1	2, 3, 4, 5, 6, 7
150	HSIOM_PRT6_PRT	42400060	8	1	2, 3, 4, 5, 6, 7
151	HSIOM_PRT7_PRT	42400070	8	1	2, 3, 4, 5, 6, 7
152	HSIOM_PRT8_PRT	42400080	8	1	2, 3, 4, 5, 6, 7
153	HSIOM_PRT9_PRT	42400090	8	1	2, 3, 4, 5, 6, 7
154	HSIOM_PRT10_PRT	424000a0	8	1	2, 3, 4, 5, 6, 7
155	HSIOM_PRT11_PRT	424000b0	8	1	2, 3, 4, 5, 6, 7
156	HSIOM_PRT12_PRT	424000c0	8	1	2, 3, 4, 5, 6, 7
157	HSIOM_PRT13_PRT	424000d0	8	1	2, 3, 4, 5, 6, 7
158	HSIOM_SECURE_PRT0_SECURE_PRT	42401000	4	1	2, 3, 4, 5, 6, 7
159	HSIOM_SECURE_PRT1_SECURE_PRT	42401010	4	1	2, 3, 4, 5, 6, 7
160	HSIOM_SECURE_PRT2_SECURE_PRT	42401020	4	1	2, 3, 4, 5, 6, 7
161	HSIOM_SECURE_PRT3_SECURE_PRT	42401030	4	1	2, 3, 4, 5, 6, 7
162	HSIOM_SECURE_PRT4_SECURE_PRT	42401040	4	1	2, 3, 4, 5, 6, 7
163	HSIOM_SECURE_PRT5_SECURE_PRT	42401050	4	1	2, 3, 4, 5, 6, 7
164	HSIOM_SECURE_PRT6_SECURE_PRT	42401060	4	1	2, 3, 4, 5, 6, 7
165	HSIOM_SECURE_PRT7_SECURE_PRT	42401070	4	1	2, 3, 4, 5, 6, 7
166	HSIOM_SECURE_PRT8_SECURE_PRT	42401080	4	1	2, 3, 4, 5, 6, 7
167	HSIOM_SECURE_PRT9_SECURE_PRT	42401090	4	1	2, 3, 4, 5, 6, 7
168	HSIOM_SECURE_PRT10_SECURE_PRT	424010a0	4	1	2, 3, 4, 5, 6, 7
169	HSIOM_SECURE_PRT11_SECURE_PRT	424010b0	4	1	2, 3, 4, 5, 6, 7
170	HSIOM_SECURE_PRT12_SECURE_PRT	424010c0	4	1	2, 3, 4, 5, 6, 7
171	HSIOM_SECURE_PRT13_SECURE_PRT	424010d0	4	1	2, 3, 4, 5, 6, 7

(table continues...)

Table 108 (continued) PPC regions

PPC region number	PPC region name	Start address (hex)	Size (hex)	SECURITY_AWARE	Accessible by (PCx)
172	HSIOM_AMUX	42402000	20	1	2, 3, 4, 5, 6, 7
173	HSIOM_MON	42402200	10	1	2, 3, 4, 5, 6, 7
174	GPIO_PRT0_PRT	42410000	40	1	2, 3, 4, 5, 6, 7
175	GPIO_PRT1_PRT	42410080	40	1	2, 3, 4, 5, 6, 7
176	GPIO_PRT2_PRT	42410100	40	1	2, 3, 4, 5, 6, 7
177	GPIO_PRT3_PRT	42410180	40	1	2, 3, 4, 5, 6, 7
178	GPIO_PRT4_PRT	42410200	40	1	2, 3, 4, 5, 6, 7
179	GPIO_PRT5_PRT	42410280	40	1	2, 3, 4, 5, 6, 7
180	GPIO_PRT6_PRT	42410300	40	1	2, 3, 4, 5, 6, 7
181	GPIO_PRT7_PRT	42410380	40	1	2, 3, 4, 5, 6, 7
182	GPIO_PRT8_PRT	42410400	40	1	2, 3, 4, 5, 6, 7
183	GPIO_PRT9_PRT	42410480	40	1	2, 3, 4, 5, 6, 7
184	GPIO_PRT10_PRT	42410500	40	1	2, 3, 4, 5, 6, 7
185	GPIO_PRT11_PRT	42410580	40	1	2, 3, 4, 5, 6, 7
186	GPIO_PRT12_PRT	42410600	40	1	2, 3, 4, 5, 6, 7
187	GPIO_PRT13_PRT	42410680	40	1	2, 3, 4, 5, 6, 7
188	GPIO_PRT0_CFG	42410040	40	1	2, 3, 4, 5, 6, 7
189	GPIO_PRT1_CFG	424100c0	40	1	2, 3, 4, 5, 6, 7
190	GPIO_PRT2_CFG	42410140	40	1	2, 3, 4, 5, 6, 7
191	GPIO_PRT3_CFG	424101c0	40	1	2, 3, 4, 5, 6, 7
192	GPIO_PRT4_CFG	42410240	40	1	2, 3, 4, 5, 6, 7
193	GPIO_PRT5_CFG	424102c0	40	1	2, 3, 4, 5, 6, 7
194	GPIO_PRT6_CFG	42410340	40	1	2, 3, 4, 5, 6, 7
195	GPIO_PRT7_CFG	424103c0	40	1	2, 3, 4, 5, 6, 7
196	GPIO_PRT8_CFG	42410440	40	1	2, 3, 4, 5, 6, 7
197	GPIO_PRT9_CFG	424104c0	40	1	2, 3, 4, 5, 6, 7
198	GPIO_PRT10_CFG	42410540	40	1	2, 3, 4, 5, 6, 7
199	GPIO_PRT11_CFG	424105c0	40	1	2, 3, 4, 5, 6, 7
200	GPIO_PRT12_CFG	42410640	40	1	2, 3, 4, 5, 6, 7
201	GPIO_PRT13_CFG	424106c0	40	1	2, 3, 4, 5, 6, 7
202	GPIO_SEC_GPIO	42417000	4	1	2, 3, 4, 5, 6, 7
203	GPIO_GPIO	42418000	40	1	2, 3, 4, 5, 6, 7
204	GPIO_TEST	42419000	8	1	2, 3, 4, 5, 6, 7

(table continues...)

Table 108 (continued) PPC regions

PPC region number	PPC region name	Start address (hex)	Size (hex)	SECURITY_AWARE	Accessible by (PCx)
205	SMARTIO_PRT0_PRT	42420000	100	0	2, 3, 4, 5, 6, 7
206	SMARTIO_PRT1_PRT	42420100	100	0	2, 3, 4, 5, 6, 7
207	SMARTIO_PRT2_PRT	42420200	100	0	2, 3, 4, 5, 6, 7
208	SMARTIO_PRT3_PRT	42420300	100	0	2, 3, 4, 5, 6, 7
209	SMARTIO_PRT5_PRT	42420500	100	0	2, 3, 4, 5, 6, 7
210	SMARTIO_PRT6_PRT	42420600	100	0	2, 3, 4, 5, 6, 7
211	SMARTIO_PRT9_PRT	42420900	100	0	2, 3, 4, 5, 6, 7
212	LPCOMP	42430000	10000	0	2, 3, 4, 5, 6, 7
213	DFT	42600000	1000	0	0
214	EFUSE_CTL1	42610000	4	0	2, 3, 4, 5, 6, 7
215	EFUSE_CTL2	42610100	80	0	0
216	EFUSE_CTL3	42610180	4	0	2, 3, 4, 5, 6, 7
217	EFUSE_DATA_BOOT1	42610800	80	0	2, 3, 4, 5, 6, 7
218	CANFD0_CH0_CH	42800000	200	0	2, 3, 4, 5, 6, 7
219	CANFD0_CH1_CH	42800200	200	0	2, 3, 4, 5, 6, 7
220	CANFD0_MAIN	42801000	40	0	2, 3, 4, 5, 6, 7
221	CANFD0_BUF	42810000	10000	0	2, 3, 4, 5, 6, 7
222	SCB0	42820000	10000	0	2, 3, 4, 5, 6, 7
223	SCB1	42840000	10000	0	2, 3, 4, 5, 6, 7
224	SCB2	42850000	10000	0	2, 3, 4, 5, 6, 7
225	SCB3	42860000	10000	0	2, 3, 4, 5, 6, 7
226	SCB4	42870000	10000	0	2, 3, 4, 5, 6, 7
227	SCB5	42c00000	10000	0	2, 3, 4, 5, 6, 7
228	TCPWM0_GRP0_CNT0_CNT	42a00000	100	0	2, 3, 4, 5, 6, 7
229	TCPWM0_GRP0_CNT1_CNT	42a00100	100	0	2, 3, 4, 5, 6, 7
230	TCPWM0_GRP0_CNT2_CNT	42a00200	100	0	2, 3, 4, 5, 6, 7
231	TCPWM0_GRP0_CNT3_CNT	42a00300	100	0	2, 3, 4, 5, 6, 7
232	TCPWM0_GRP0_CNT4_CNT	42a00400	100	0	2, 3, 4, 5, 6, 7
233	TCPWM0_GRP0_CNT5_CNT	42a00500	100	0	2, 3, 4, 5, 6, 7
234	TCPWM0_GRP0_CNT6_CNT	42a00600	100	0	2, 3, 4, 5, 6, 7
235	TCPWM0_GRP0_CNT7_CNT	42a00700	100	0	2, 3, 4, 5, 6, 7
236	TCPWM0_GRP1_CNT0_CNT	42a10000	100	0	2, 3, 4, 5, 6, 7
237	TCPWM0_GRP1_CNT1_CNT	42a10100	100	0	2, 3, 4, 5, 6, 7

(table continues...)

Table 108 (continued) PPC regions

PPC region number	PPC region name	Start address (hex)	Size (hex)	SECURITY_AWARE	Accessible by (PCx)
238	TCPWM0_GRP1_CNT2_CNT	42a10200	100	0	2, 3, 4, 5, 6, 7
239	TCPWM0_GRP1_CNT3_CNT	42a10300	100	0	2, 3, 4, 5, 6, 7
240	TCPWM0_GRP1_CNT4_CNT	42a10400	100	0	2, 3, 4, 5, 6, 7
241	TCPWM0_GRP1_CNT5_CNT	42a10500	100	0	2, 3, 4, 5, 6, 7
242	TCPWM0_GRP1_CNT6_CNT	42a10600	100	0	2, 3, 4, 5, 6, 7
243	TCPWM0_GRP1_CNT7_CNT	42a10700	100	0	2, 3, 4, 5, 6, 7
244	TCPWM0_GRP2_CNT0_CNT	42a20000	100	0	2, 3, 4, 5, 6, 7
245	TCPWM0_GRP2_CNT1_CNT	42a20100	100	0	2, 3, 4, 5, 6, 7
246	TCPWM0_GRP2_CNT2_CNT	42a20200	100	0	2, 3, 4, 5, 6, 7
247	TCPWM0_GRP2_CNT3_CNT	42a20300	100	0	2, 3, 4, 5, 6, 7
248	TCPWM0_GRP2_CNT4_CNT	42a20400	100	0	2, 3, 4, 5, 6, 7
249	TCPWM0_GRP2_CNT5_CNT	42a20500	100	0	2, 3, 4, 5, 6, 7
250	TCPWM0_GRP2_CNT6_CNT	42a20600	100	0	2, 3, 4, 5, 6, 7
251	TCPWM0_GRP2_CNT7_CNT	42a20700	100	0	2, 3, 4, 5, 6, 7
252	TCPWM0_TR_ALL_GF_TR_ALL_GF	42a80000	80	0	2, 3, 4, 5, 6, 7
253	TCPWM0_TR_ALL_SYNC_BYPASS_TR_ALL_SYN_BYPASS	42a90000	4	0	2, 3, 4, 5, 6, 7
254	TCPWM0_MOTIF_GRP1_MOTIF0_MOTIF	42aa4000	200	0	2, 3, 4, 5, 6, 7
255	TCPWM0_MOTIF_GRP2_MOTIF0_MOTIF	42aa8000	200	0	2, 3, 4, 5, 6, 7
256	MCPASS_AFE	42b60000	10000	0	2, 3, 4, 5, 6, 7
257	MCPASS_SARADC	42b70000	10000	0	2, 3, 4, 5, 6, 7
258	MCPASS_SAR	42b80000	10000	0	2, 3, 4, 5, 6, 7
259	MCPASS_DAC_BUFFER	42b90000	10000	0	2, 3, 4, 5, 6, 7
260	MCPASS_HWFILT3P3Z	42ba0000	1000	0	2, 3, 4, 5, 6, 7
261	MCPASS_CSG	42bb0000	10000	0	2, 3, 4, 5, 6, 7
262	MCPASS_ACTRLR	42bd0000	10000	0	2, 3, 4, 5, 6, 7
263	MCPASS_INFRA	42be0000	10000	0	2, 3, 4, 5, 6, 7
264	MCPASS_MMIO	42bf0000	10000	0	2, 3, 4, 5, 6, 7
265	I3C0	42c10000	10000	0	2, 3, 4, 5, 6, 7
266	USBFS0	42c20000	10000	0	2, 3, 4, 5, 6, 7

***Note**: Access to the TCPWM0_MOTIF_GRP1_MOTIF0_MOTIF and TCPWM0_MOTIF_GRP2_MOTIF0_MOTIF regions is allowed from the PCs listed only if the part supports MOTIF. Refer to the part's datasheet to determine if the part supports MOTIF. If the part does not support MOTIF, access to this regions are restricted to PC0.*

15 Secure boot

Secure boot is defined as the process of obtaining, validating, and launching the product firmware. PSC3M/P6 MCU BootROM consists of ROM_BOOT and FLASH_BOOT. After reset, the immutable ROM_BOOT always begins and verifies the integrity of SFLASH memory, which consists of FLASH_BOOT, Infineon and OEM policy assets. After ROM_BOOT transfer control to FLASH_BOOT, the pre-programmed FLASH_BOOT extends the functionality of ROM_BOOT and applies full trims as needed before authenticating the next stage image (OEM application). FLASH_BOOT authenticates and launches the next stage application or edge protect bootloader. The following chapter describes each stage of the secure boot process in detail.

15.1 Basic definitions

The following terms and definitions will help you to understand the terms used in this document. Many of these terms will be discussed in more detail within this section.

- **BootROM**: The PSC3M/P6 MCU consists of two firmware components: ROM_BOOT and FLASH_BOOT. Collectively, these firmware components are referred to as the immutable BootROM
- **Chain of Trust (CoT)**: The root of trust begins with the Infineon code residing in ROM, which cannot be altered. Chain of Trust is established by validating the blocks of software starting from the root of trust (RoT) ROM_BOOT
- **Debug Access Port (DAP)**: Interface between an external debugger/programmer and PSC3M/P6 MCU for programming and debugging.
- **Digest**: A digest is computed by applying a hash function on a data set, such as the binary of an application
- **Digital signature**: Encrypting of the digest (hash of a data set). For example, the encrypted hash of the user application
- **Device integrity exam**: A test used prior to provisioning to verify that this is a genuine device and is in a known state
- **Hash:** A crypto algorithm that generates a repeatable but unique signature for a given block of data. This function is not reversible
- **Hardware security module (HSM)**: This is a physical computing or programming device that protects and manages digital keys, it can perform digital software signing and encryption as well as other cryptographic functions
- **Hardware unique key (HUK):** A 256-bit device root key for deriving all device-specific keys
- **Inter-process communication (IPC):** IPC hardware used to facilitate communication between secure and non-secure memory
- **Life cycle stage (LCS):** LCS is the security mode in which the device is operating
- **Memory protection controller (MPC):** Enables to isolate the parts of memory by means of PCs
- **Master security controller (MSC):** Helps to protect the transactions initiated on the bus
- **Non-volatile counter (NV counter):** Security counter to offer anti-rollback protection
- **OTP**: One-time programmable memory
- **Protection context (PC):** This parameter allows each bus master a security state level from '0' to '7'. A bus master can be assigned a PC value that stays static or that is changed during application execution. PC provides a more precise way of applying memory restrictions. The PC state works together with protection units
- **Public-key cryptography (PKC):** Otherwise known as asymmetrical cryptography. Public-key cryptography is an encryption technique that uses a paired public and private key (or asymmetric key)

algorithm for secure data. It is used to secure a message or block of data. The private key is used to encrypt data and must be kept secure, and the public key is used to decrypt but can be disseminated widely
  - **Public key:** The public key can be shared, but it should be authenticated or secure so it cannot be modified
  - **Private key:** The private key must be kept in a secure location, so it cannot be viewed or stolen. It is used to encrypt a block of data that will be decrypted using an associated public key
- **Policy:** Policies are a collection of predefined (name, value) pairs that describe what is and is not allowed on the device to which the policy applies
- **Peripheral protection controller (PPC):** Allows configuring the peripherals such that they are accessible only in pre-configured PCs assigned by the RoT
- **Provisioning:** The process by which keys, policies and secrets are injected into the device. Once provisioned, the device can be accessed or modified only with the keys injected adhering to the relevant policies
- **Platform security architecture (PSA):** Firmware security specifications and open-source reference implementations, to help building best-practice security into products
- **RMA:** Return merchandise authorization
- **SESIP:** Security Evaluation Standard for IoT (Internet of Things) Platforms
- **SHA-256:** A cryptographic hash algorithm is used to create a digest for a block of data or code. This hash algorithm produces a 256-bit unique digest of the data, no matter the size of the data block

15.2 Features

The PSC3M/P6 MCU provides the following secure boot features:

- Each boot stage verifies the next stage image before launching them; therefore, offers a chain of trust until launches the first user application
- Implements a hardware root of trust with immutable trust anchor with the help of pre-programmed multistage BootROM (consists of ROM_BOOT and FLASH_BOOT firmware)
- Provides a conventional development LCS to get started with and transition to production LCS to productize the device
- Built-in immutable key storage area that can be loaded with required OEM keys and other user keys during provisioning
- Supports dual bank. The device user flash is divided into two equal-sized banks (256 KB each), known as bank 0 and bank 1. This feature is enabled/disabled through provisioning
- Supports protected firmware. This feature enables customer to protect their firmware IP
- Supports secure debug and secure provisioning from Infineon to the users
- Device provisioning and firmware upgrade can be performed through either SWD interface or BootROM DFU Serial Interface

15.3 Architecture

15.3.1 Policies and keys

15.3.1.1 Policies

- **IFX_POLICY:** Infineon factory provisions the IFX_POLICY. It consists of pre-programmed Infineon assets and a securely bootable and upgradable piece of pre-programmed software. These are factory-programmed and locked by Infineon
- **PROT_FW_POLICY:** PROT_FW_POLICY consists of PROT_FW_ROT_KEYS and protected firmware attributes. PROT_FW_POLICY is provisioned in development LCS
- **OEM_POLICY:** OEM provisions the OEM_POLICY in development LCS. Policy consists of device life cycle state (LCS), debug policy, boot policy, dfu policy, OEM_ROT_KEYS and other assets. Policy can be changed several times until "prov_oem_complete" flag is set to "true"

15.3.1.2 Keys

The keys that are described in this section are asymmetric ECC key pairs. Each pair includes both a PRIVATE and a PUBLIC Key.

- **IFX_ROT_KEY:** IFX_ROT_KEY is an asymmetric ECC 521-bit ECDSA key pair used to validate the assets signed and installed by Infineon during the initial manufacturing. Infineon stores the IFX_ROT_PRIVATE_KEY securely in internally controlled HSM and stores the IFX_ROT_PUBLIC_KEY on the device OTP. RAM applications are signed using the stored IFX_ROT_PRIVATE_KEY and validated using the corresponding IFX_ROT_PUBLIC_KEY stored on the device.
- **PROT_FW_ROT_KEY:** PROT_FW_ROT_KEY is used to authenticate the protected firmware. To revoke this key, use PROT_FW_ROT_KEY_REVOCATION.
- **OEM_ROT_KEY:** OEM_ROT_KEY is used to authenticate the OEM application images. The device supports separate OEM RoT keys for Development and Production LCS. In Development LCS, one OEM RoT key is available, while in Production, two keys are available. In Production LCS, by default, OEM_ROT_KEY0 is used to authenticate the images. If needed, the OEM can revoke the key to use OEM_ROT_KEY1. The OEM can revoke the key only once through provisioning, and it cannot be reverted. During the provisioning process, the OEM can configure the OEM_ROT_KEY pair to use either ECDSA-521, ECDSA-384, or ECDSA-256 bit ECC. After the OEM has made this selection, the corresponding public key should be used to sign the OEM image. By default, the policy is configured to use ECDSA-521.

15.3.2 Life cycle stages (LCS)

PSC3M/P6 MCUs have configurable, non-volatile life cycle stages that play a critical role in device security. These stages follow a strict, one-way progression, controlled by blowing an eFuse, which ensures that once a stage is completed, it cannot be reversed. For instance, once a device transitions from the development stage to production, it cannot be reverted back to development. Instead, it can remain in production or move forward to the return merchandise authorization (RMA) stage, as determined by the OEM.

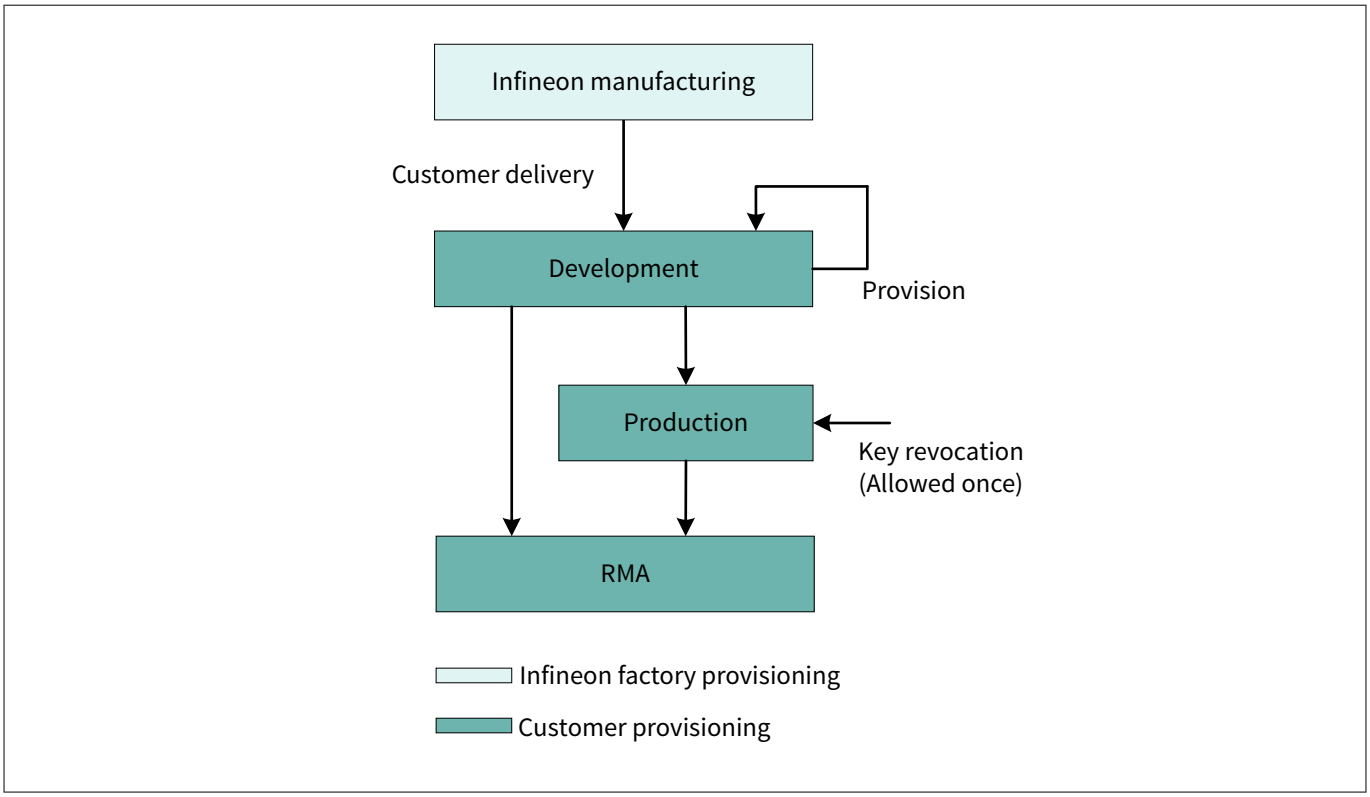


Figure 57 Life cycle stages

15.3.2.1 IFX_MFG

In Infineon Fab, parts are produced, tailored, and programmed in the IFX_MFG LCS. All the necessary testing and trimming takes place in this LCS and Infineon assets are loaded to the device. Device is transitioned to development LCS before shipping from the Infineon factory.

15.3.2.2 Development LCS

PSC3M/P6 MCU devices are shipped in development LCS to the customer. Before the devices enter this state, all the trimming and factory testing is completed, and the devices are shipped out of Infineon in this LCS. Customers must provision OEM assets during development LCS and develop their end application. At this stage, the device can be re-provisioned any number of times, and the OEM_ROT_KEY and assets can be replaced as needed through OEM provisioning.

15.3.2.3 Production LCS

This is the in-field operational LCS of a secure device. Before moving onto this phase, signed valid application code must have been programmed and the HASH of the OEM keys must be provisioned (OEM_ROT_KEYS). In this LCS, device will boot only if application is authenticated successfully. If authentication fails at any stage, next application will not boot, and device boot process terminates here.

During transition of production LCS, the provisioning RAM Application performs the following:

- Provisions OEM_ROT_KEYS
- Verifies all policies are provisioned with the authorized production keys

- Writes SECURE_HASH to the OTP. A 32-byte SHA-512 SECURE_HASH consists of hash of OEM_ROT_KEYs and OEM_POLICY
- Writes the OTP bit to move irreversibly to the production state

  The debug policy can be set to permanently disable the debug in this LCS. Alternately, the debug policy can be set to disable if debug capabilities are required in this LCS and the device can be debugged with a secure debug certificate. Infineon recommends keeping the debug ports closed in production LCS.

15.3.2.4 RMA

The device can transition to the RMA lifecycle when a customer require Infineon to perform the failure analysis (FA). The device can be moved to RMA mode by invoking `TransitionToRMA RAMApp` with a certificate signed by OEM_ROT_KEY. Customer should erase all OEM-owned assets on the device before invoking the `RAMApp` and transition to the RMA LCS state. Once the device is transitioned to the RMA life cycle stage, the device can no longer be provisioned or programmed again. After the device is moved into RMA mode and when it boots, it does not attempt to execute any code in the user flash; instead, it waits for the `OpenRMA` token through the debug port, signed by Infineon, to enable further internal debug and analysis of the device.

15.3.3 Boot sequence

Secure boot authenticates the application images through the use of a public key cryptography security protocol defined by an industry-specific standard. ROM_BOOT trims and initializes the hardware. It validates the FLASH_BOOT before it jumps to execute FLASH_BOOT. FLASH_BOOT extends the functionality of ROM_BOOT, applies IFX and OEM policies, validates, and launches the OEM application. The BootROM uses hashing extensively to authenticate data and code stored in the device. The OEM can have multiple images based on BOOT_APP_LAYOUT[] and BOOT_CFG_ID defined in the OEM policy, as mentioned in Table 109. Figure 58 shows the secure boot workflow.

Table 109 Boot configuration

BOOT_CFG_ID	Image options
Simple application configuration	
`SIMPLE_APP`	• By default, the device is configured for BOOT_SIMLPE_APP configuration • The user flash should have an unsigned raw image • The image should be placed at the start of the flash address
`DUAL_BANK_SIMPLE_APP`	• In this configuration, the user flash memory is split in to two equal size (256 KB each) banks: bank0 and bank1 • BootROM launches the application image from the active bank, which is determined by the dual bank counter value located at BOOT_BANK_CTR_OFFSET • Rest of the features are the same as SIMPLE_APP
Secure boot configuration	
The following configurations are supported for secure boot: • `SECURE_APP` • `DUAL_BANK_SECURE_APP` • `PROT_FW`	

(table continues...)

Table 109 (continued) Boot configuration

BOOT_CFG_ID	Image options
SECURE_APP	• The BootROM shall authenticate multiple OEM images as specified in BOOT_APP_LAYOUT[] and launch the first CM33 secure application • The BOOT_APP_LAYOUT[] address list determines the number of images that the BootROM will authenticate • The BootROM shares boot status information, providing the pass/fail status for image authentication
DUAL_BANK_SECURE_APP	• This configuration is the same as DUAL_BANK_SIMPLE_APP, and additionally BootROM expects the OEM image should be signed with OEM_ROT_KEY and aligned on MCUBoot format • The BootROM authenticates the active bank OEM images and launches it after successful authentication
PROT_FW	• This configuration is meant for protected firmware, which enforces read and write restrictions and allows only execution permission • The protected firmware should be signed with PROT_ROT_KEY and aligned with the MCUboot format • The BootROM launches the protected firmware in the PC2 context and allows debugging only with a debug token

15.3.3.1 Secure boot

The BootROM provides the fundamental security features required to provide a secure system. When device is shipped from Infineon in development LCS, "secure boot" is disabled in the policy and BootROM boots to the OEM Application, without validating it. "Secure boot" can be enabled in development LCS by provisioning OEM assets and programming OEM Application image in MCUboot format.

When "secure boot" is enabled in the policy, BootROM validates the first application using OEM_ROT_KEY_PUBLIC and creates a chain of trust between BootROM and the first OEM application to provide the highest order of security.

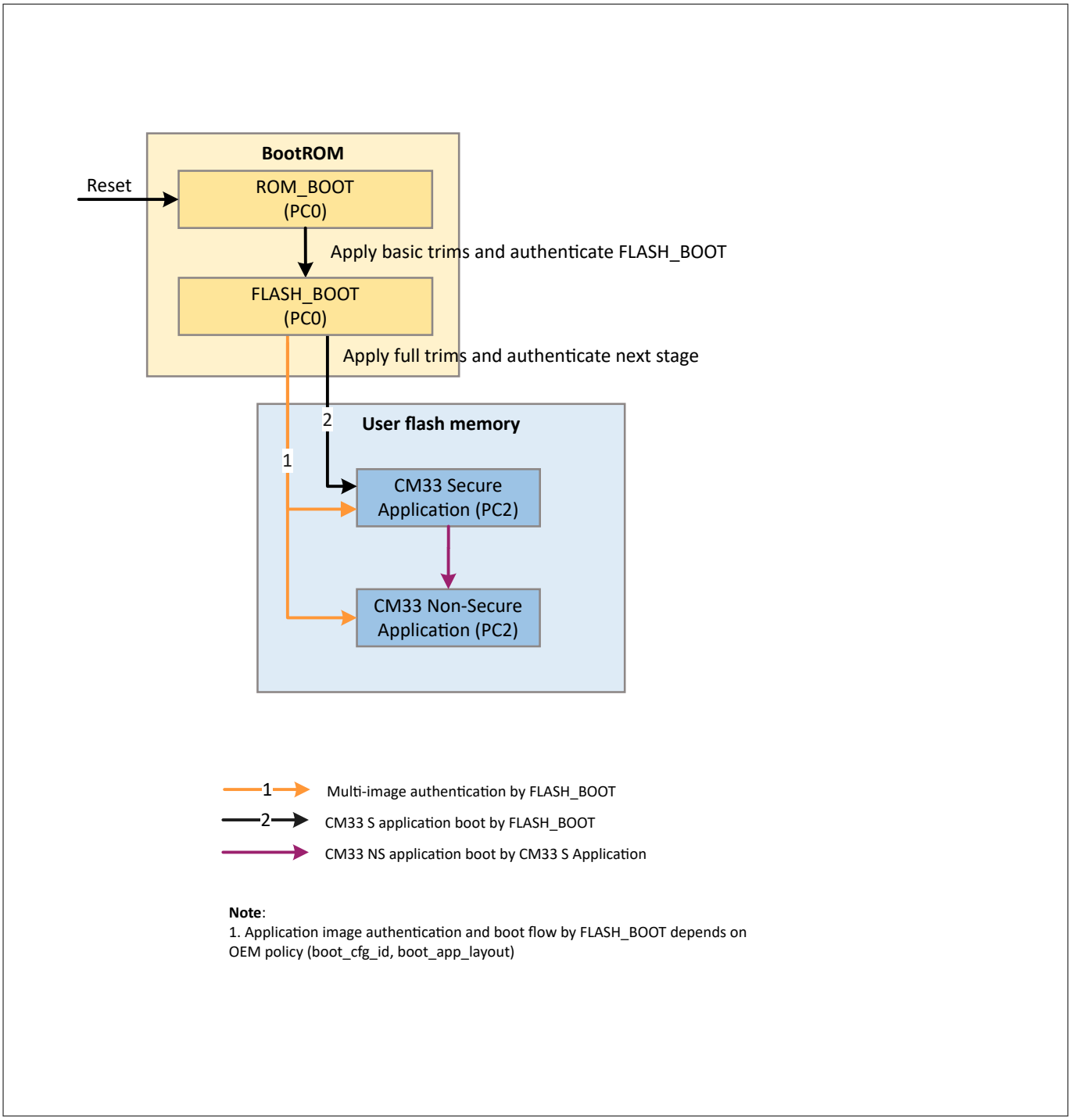


Figure 58 Secure bootflow

15.3.3.2 Protection states

A life cycle stage is a strict irreversible progression dictated by writing one time program (OTP) eFuse bits. Every time a boot occurs, protecting settings (MPC and PPC) of the device are applied by boot process. If a critical error, such as a hardware initialization error or any Infineon provisioned asset corruption or mismatch detection is found by Boot, the device enters the corrupted state.

In the event that secure boot is enabled and the authentication of the first user application image fails, the device will enter a 'dead branch' state. To recover from this state, the user will need to re-provision the device with the correct OEM assets in the development life cycle stage. When two user application images are present,

if only the second user application image authentication fails, the device will still boot the first user application image with a failure indication.

When a device encounters a debug token verification failure, it will not grant access to its debug port. To configure the debug policy, you need to modify the debug section of the OEM provisioning policy.

15.3.4 BootROM serial interface

BootROM provides simple serial interfaces for performing a number of operations during manufacturing including provisioning and downloading the protected firmware as well as customer applications, using Infineon's proprietary simple device firmware update (DFU) over a serial protocol for communication.

By default, the DFU feature is not enabled in the original equipment manufacturer's (OEM) policy. To enable the feature, users must configure the DFU-related OEM policy fields listed in Table 110 and provision the device using Serial Wire Debug (SWD) interface.

Table 110 DFU Configuration in OEM Policy

DFU related OEM Policy Field	Description
DFU_ENABLE	This field enables the DFU feature. It should be set to true to enable it.
DFU_IF	This field configures the serial interface to be used for DFU. The following serial interfaces are supported: • UART • SPI • I^2C
DFU_PINS_CTL	This field configures the SCB port(s) to be used for DFU: • primary_port - Primary DFU interface port number (0, 1, ..., 13) • alternate_port - Alternate DFU interface port number (0, 1, ..., 13); set to 0 for single-port SCBs
DFU_EN_PIN	This field configures the GPIO pin checked during each boot-up to determine whether to enter DFU mode or proceed with the normal boot flow. Format: Px_y, where x is the port number (0, 1, ..., 13) and y is the pin number (0, 1, ..., 7).

(table continues...)

Table 110 (continued) DFU Configuration in OEM Policy

DFU related OEM Policy Field	Description
DFU_CONFIG	This field configures the selected serial interface. The pin fields specify num (pin number 0-7) and alternate_port (false = primary port, true = alternate port): UART - Baud = 115200, Bits = 8, Stop Bits = 1, RTS/CTS = No, Parity = None (fixed, cannot be changed) • scb - SCB instance number (0-5) • rx - RX pin • tx - TX pin SPI • scb - SCB instance number (0-5) • spi_clk_mode - SPI SCLK mode (0-3): - 0 - CPOL=0, CPHA=0 - 1 - CPOL=0, CPHA=1 - 2 - CPOL=1, CPHA=0 - 3 - CPOL=1, CPHA=1 • mosi - MOSI pin • miso - MISO pin • clk - CLK pin • nss - NSS pin I2C • scb - SCB instance number (0-5) • address - I2C slave address, 7 bits (0-127) • sda - SDA pin • scl - SCL pin

***Note**: To set the DFU_EN_PIN high or low, use external pull-up or pull-down resistors of 1kΩ or less.*

15.3.5 Secure provisioning and re-provisioning

Provisioning is the act of configuring a device with an authorized set of keys, certificates, credentials, and firmware images. Provisioning is performed using the provisioning applications (RAM application) that are built to run from the device SRAM. These applications program the Infineon and OEM assets into OTP, SFLASH and advance the device to development or production LCS. The device is ready to launch the user application in these LCSs.

Development LCS allows you to re-provision the OEM assets during the development phase any number of times. Production LCS will not allow to re-provision the OEM assets rather allows only once revoking the OEM_ROT_KEY and PROT_FW_ROT_KEY keys.

15.3.6 Protection context used in secure boot

Protection context (PC) is not one specific block of the hardware. However, it is used for all the protection units, such as MPU, MPCs, and PPCs. Multiple protection contexts are used to isolate the different security levels within the PSC3M/P6 MCU device. Secure boot process uses PC0 and PC2 as shown in Figure 58 and Table 105.

16 Advanced Encryption Standard (AES)

This chapter describes the advanced encryption standard (AES) accelerator module, this module provides high-performance AES encryption or decryption with key lengths of 128/256 bits.

16.1 Introduction

The AES cryptographic peripheral is a dedicated hardware that implements a full algorithm according to `AES_FIPS_PUB_197` and `ISO_IEC_18033_3` standards.

The hardware can be easily programmed to perform an AES encryption or decryption with key lengths 128 and 256 bits. It has built-in security features, such as masking and redundancy mechanisms to protect the key as well as the input and output data against power analysis, fault, and probing attacks. The special function registers (SFRs) are specified below in the registers section. Functional software behavior of the module is described below.

16.2 Overview

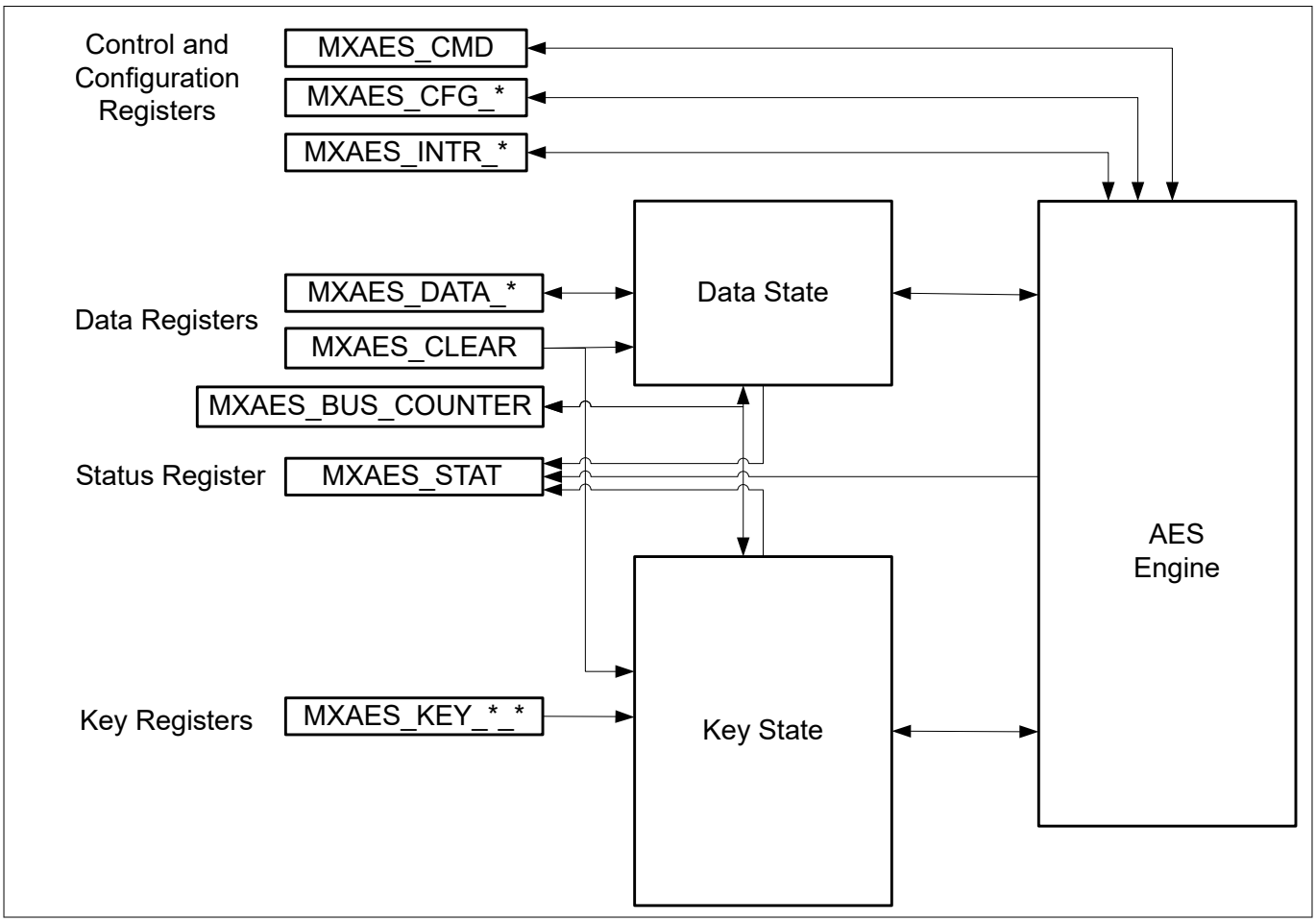


Figure 59 AES user interface

The AES SFR interface has the following types of registers:

- Configuration and command registers: These registers are used to let the software select the AES calculation mode, key length and, for example, set the security level required. The command register is used to start a calculation

- Status register: It provides general status information, for example, whether the AES engine is busy with a calculation and whether the result is ready or not
- Key registers: These registers are used to load directly to key shares into the AES key state
- Data registers: These are the input and output registers for the plain and cipher texts

16.3 Programming model

The programming sequence given in the below flow chart guides the user with the steps mandatory to execute to perform an AES encryption with this hardware accelerator module.

***Note**: The order of writing of MXAES_CFG_2 and MXAES_KEY_* and MXAES_DATA_* does not matter.*

***Note**: Clearing data is needed for starting a new calculation otherwise a DATA_X_Y word write is XORed with current DATA_X_Y content.*

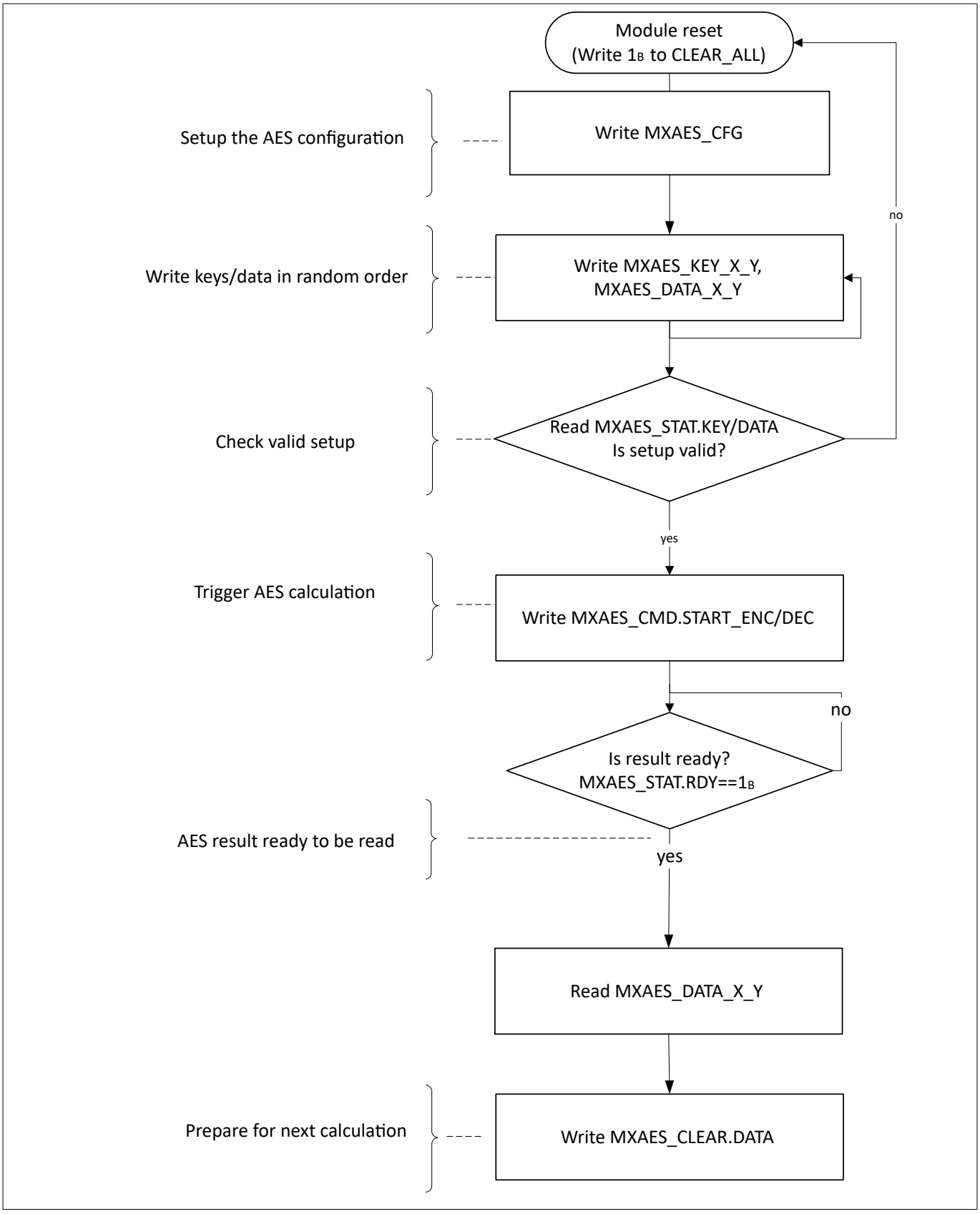


Figure 60 AES programming sequence

16.4 Functional description

16.4.1 Module reset

Through the module, reset the AES can be reset separately without the need to reset the whole system. The module reset can be triggered from software by setting the CLEAR_ALL bit to 1. This clears all SFRs, data, keys, and masks internal states except the internal seed.

16.4.2 Endianess

When writing a 128-bit plaintext block 0x1234ABCD9876EEFF1122334455667788, 0xCDAB3412 has to be written to AES_DATA_x_0, 0xFFEE7698 to AES_DATA_x_2, 0x44332211 to AES_DATA_x_3, and 0x88776655 to AES_DATA_x_3. However, the chronological order of reading and writing the data shares does not matter.

16.4.3 BusFault scenarios

To reduce the verification effort and the module complexity, the AES engine reacts to some scenarios with BusFaults such as not allowed read or write accesses or not allowed read or write accesses when the engine is not idle or specific not allowed subsequent accesses, non-allowed combinations of some modes and useless subsequent operations. The not allowed read or write accesses are given in the access modes tables below each register's definition but the rest of cases to which the engine reacts with a BusFault are listed below:

- Writing to or reading from unmapped registers
- Writing KEY_X_Y while the key state contains a KEY_X_Y decryption key

16.4.4 Handling of key state clearing

The clear register gives the user the possibility to clear keys. Clearing a key also clears the corresponding valid flag. The table below lists the keys which have been cleared according to which CLEAR.AES_KEY_* field has been set. Reading the STAT status bits after having written the CLEAR bit allows the user to check if the key-clearing settings through CLEAR write access – have been executed correctly.

Table 111 Key location cleared by CLEAR register

Field	List of keys implicitly cleared	List of key registers to reload	List of CRYPT_MODE keys not affected	List of keys not affected
AES_KEY_128LO	AES_KEY_256	KEY_0 KEY_1 KEY_2 KEY_3	AES128HI	AES_KEY_128HI
AES_KEY_128HI	AES_KEY_256	KEY_4 KEY_5 KEY_6 KEY_7	AES128LO	AES_KEY_128LO
AES_KEY_256	AES_KEY_128LO AES_KEY_128HI	AES_KEY_*	-	-

16.4.5 Fault structure

This module is a fault generating component. It acknowledges a fault in case the AES hardware has encountered a malicious behavior. It implements a fault interface: A fault request output signal, a fault acknowledge input signal and a fault data output information to support the handshake protocol as specified by the CPUSS fault report structure.

16.4.6 Timing

The total execution time for AES consists of the I/O overhead and the actual computation time. The I/O overhead strongly depends on the software that controls the AES module. The table below shows the actual hardware computation time. It gives first the time during which the engine is busy with a computation and the second the time elapsed for the AES result to be ready depending on the SEC_LEVEL configuration. For SEC_LEVEL=0, the result of an AES 128 operation is ready in 469 clock cycles.

Table 112 AES calculations timing

Engine Operation	AES-128	AES-256
Engine busy [clock cycles]	$352+192*2^{SEC_LEVEL}$	$472+192*2^{SEC_LEVEL}$
Result ready [clock cycles]	$277+192*2^{SEC_LEVEL}$	$373+192*2^{SEC_LEVEL}$

16.4.7 Debug support

This section describes the behavior of the AES module for debug accesses and during Debug state. The debugger has access to the module even if disabled via the System/Product PAU. However, there are no extra access rights for debug purposes and the module does not support debug registers. For debug accesses the AES operates as in normal mode of operation. It does not stop or suspend its current operation when a Debug state is entered.

***Note**: The differences in CPU and debug accesses are specified in the access rights table in cases where register access rights are different from CPU to debug accesses. Additionally, no side effects occur with debug accesses.*

16.5 AES registers interface

This section defines the programmer's view of the AES accelerator. The AES memory-mapped address region is 8 KB large. AES address space is partitioned into two address spaces: non protected (AES_R0 and AES_R1 regions) and protected (AES_R2 and AES_R3 regions). Protected AES registers can be identified through the value 1B of the thirteenth bit of the offset address. Second, each of these two times 4 K bytes address spaces is in turn partitioned into two address spaces: One for software higher access rights protected, so-called protection control (PC0/PC1) and a second for the rest of the software. The AES registers mapped to the latter two times 2 KB regions (AES_R1 and AES_R3) can be identified through the value 1B of the twelfth bit of the offset address.

Both regions sizes are power of two integers:

- R1=OFFSET_END_R1-OFFSET_START_R1+1=2^{11} Bytes
- R3=OFFSET_END_R3-OFFSET_START_R3+1=2^{11} Bytes

The following figure gives an overview of these four AES address-mapped regions.

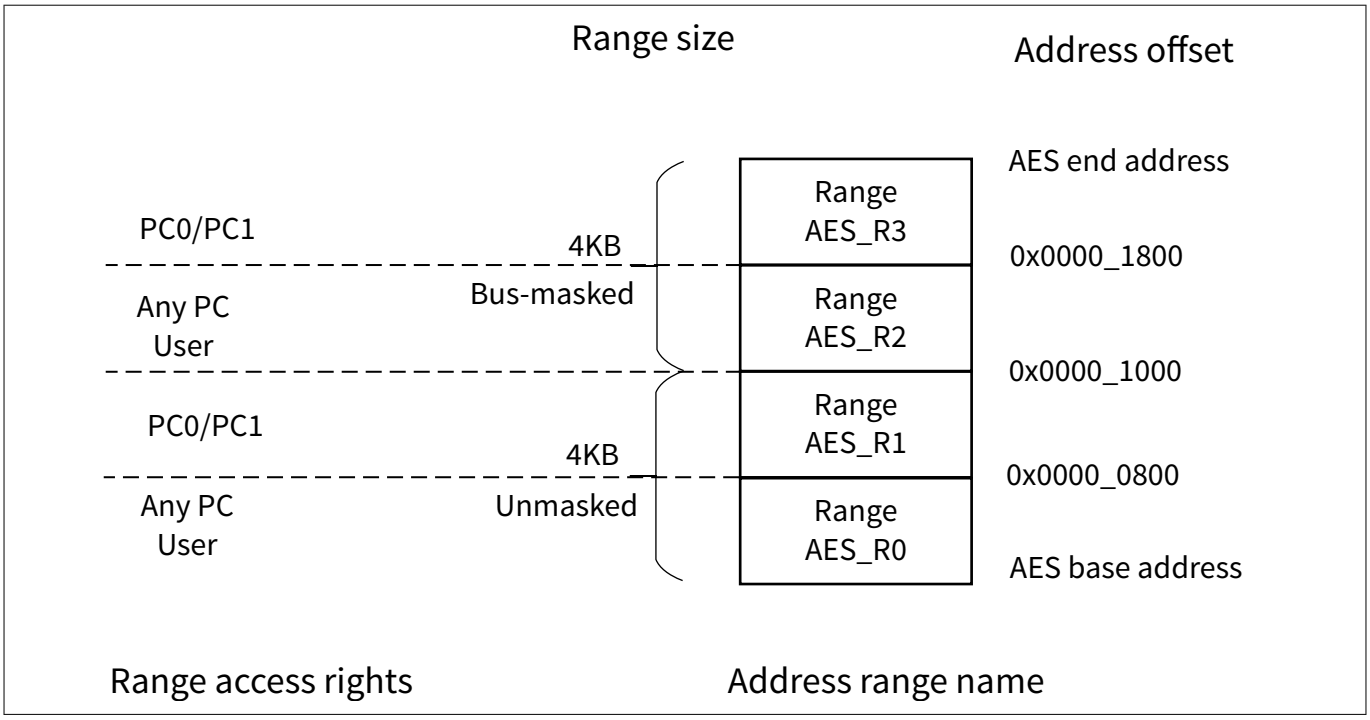


Figure 61 AES address space

16.5.1 Registers overview

Table 113 Registers overview

Register	Description	Address
MXAES_CFG	AES configuration register	0x4227_0000
MXAES_CMD	AES command register	0x4227_000C
MXAES_STAT	AES status register	0x4227_0014
MXAES_CFG_2	AES configuration register 2	0x4227_0060
MXAES_INTR	AES interrupt cause register	0x4227_0080
MXAES_INTR_SET	AES interrupt set register	0x4227_0084
MXAES_INTR_MASK	AES interrupt mask register	0x4227_008C
MXAES_INTR_MASKED	AES masked interrupt register	0x4227_0090
MXAES_CLEAR	AES clear register	0x4227_00C0
MXAES_BUS_COUNTER	AES bus counter register	0x4227_00DC
MXAES_QCHAN_CTL	AES Q-channel response register	0x4227_00E0
MXAES_SEED_U	AES user seeding register	0x4227_1000
MXAES_KEY_x_y (x=0-1;y=0-7)	AES key register (x for index of shares; y for index of word)	0x4227_1100 + x * 0x60 + y * 12
MXAES_DATA_x_y (x=0-1;y=0-3)	AES data register (x for index of shares; y for index of word)	0x4227_1400 + x * 0x30 + y * 12

(table continues...)

Table 113 (continued) Registers overview

Register	Description	Address
MXAES_DATA_RR_x_ y (x=0-1;y=0-3)	AES data reread register (x for index of shares; y for index of word)	0x4227_1490 + x * 0x30 + y * 12

In tables where register bits or bit fields are defined, the following conventions are used to indicate the access types.

Table 114 Bit function terminology

Abbreviation	Description
r	The bit or bit field can only be read (read-only).
w	The bit or bit field can only be written (write-only). A read to this register will always give a default value back that is described in the register documentation.
rw	The bit or bit field can be read and written.
rw1ch	The bit or bit field can be read and written, but the bit or bit field can be cleared only by hardware.
wh	The bit or bit field can only be written (write-only), but the bit or bit field can be also set or reset by hardware.
rwh	As rw, but the bit or bit field can also be set or reset by hardware.
rh	This bit or bit field can be modified by hardware (read-hardware, typical example: status flags). A read of this bit or bit field gives the actual status of this bit or bit field back. Writing to this bit or bit field has no effect on the setting of this bit or bit field.
rw1s	As rw, only the software can set it by writing 1.

16.5.2 Access conditions

Table 115 Access conditions of AES

Condition	Description
AES_BUSY	The AES engine is busy. This condition defines a register's access behavior when the AES is busy doing a cryptographic operation; no key can be written. This mode is indicated by: AES_STAT.BSY = BSY
AES_IDLE	The AES engine is not busy and not ready simultaneously. It is the so-called idle state in the AES module finite state machine. This condition defines a register's access behavior indicated by: AES_STAT.RDY = NRDY and AES_STAT.BSY = NBSY
AES_NBSY	The AES engine is not busy with a calculation. AES_STAT.BSY = NBSY

(table continues...)

Table 115 (continued) Access conditions of AES

Condition	Description
AES_NRDY	The data is being loaded or data result is not ready. This condition defines a register's access behavior indicated by: AES_STAT.RDY = NRDY
AES_RDY	The AES result is ready. This condition defines a register's access behavior when the AES result is ready; data can be read but not written. Note that the ready access mode is not the complementary of the busy access mode. This mode is indicated by: AES_STAT.RDY = RDY
CPU	Access mode for CPU.
DEBUG	Access mode for debugger.
User	In the User mode, all SFRs are accessible.

16.5.3 Resets

Table 116 Resets of AES

Reset	Description
Module reset	A software synchronous local clearing of AES module is triggered when CLEAR_ALL is set.
Power-on reset	Reset through cold reset
System reset	System reset

16.5.4 AES configuration register

The configuration register is used for the two main settings of AES, which are the selection of the algorithm and the selection of the mode of calculation. Writing operations with undefined values for the fields CRYPT_MODE or CALC_MODE are ignored (i.e. the register keeps its previous value). NOMODE setting and any combination with NOMODE are illegal writes. Any legal write to MXAES_CFG clears all the AES data registers and the AES_STAT.RDY flag. A switch CFG.CRYPT_MODE from AES128LO/HI to AES256 or vice versa clears KEY_X_Y and the key state content.

MXAES_CFG Offset address: 0x0000

AES configuration register Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res			CRYPT_MODE					Res			CALC_MODE				
r			rw					r			rw				

Field	Bits	Type	Description
CALC_MODE	4:0	rw	**Mode of AES calculation** This field selects the mode of operation. • 0x00 **NOMODE**: No calculation mode selected • 0x03 **CIPHER**: Set the cipher mode This mode corresponds to a encrypt or decrypt calculation with masked key and data. To select the type of operation and start a calculation set MXAES_CMD.START_ENC or START_DEC
CRYPT_MODE	12:8	rw	**Cryptographic mode** This field selects the key's length and key state location of the AES cryptographic engine. • 0x00 **NOMODE**: No key size selection • 0x10 **AES128LO**: Set AES mode using lower half 128-bit key • 0x13 **AES128HI**: Set AES mode using higher half 128-bit key • 0x16 **AES256**: Set AES mode using 256-bit key

Table 117 Access modes of MXAES_CFG

Condition	Access mode	Description
AES_BUSY and Debug	read/write	This register is read only. Writes are ignored.
AES_BUSY	read only	This register is read only in this mode. Writes to this register trigger a BusFault.
(default)	read/write	This register has no access restriction.

16.5.5 AES command register

This command register of AES can be used to launch an operation. Note that setting START_ENC, START_DEC at the same time is ignored and has no impact. The setting START_ENC=START_DEC=1 is ignored. Only 0b10 and 0b01 cause an operation (either START_DEC or START_ENC bit is set). Setting more than one bit will be ignored. Writing 0b00 does not cause any action in the module. Writing 0b11 is ignored.

MXAES_CMD Offset address: 0x000C

AES command register Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res														START_DEC	START_ENC
r														rw1ch	rw1ch

Field	Bits	Type	Description
CALC_ENC	0	rw1ch	**Launch an AES encryption operation** Setting MXAES_CMD.START_ENC launches an encryption. Any write to this bit is ignored if data is not valid and key for the selected key length in MXAES_STAT is not valid. The START_ENC is cleared only by hardware as soon as the flag BSY is cleared. • 0x0 **NENC**: No encryption operation is launched • 0x1 **ENC**: An encryption operation is launched
CRYPT_DEC	1	rw1ch	**Launch an AES decryption operation** Setting MXAES_CMD.START_DEC launches a decryption. Any write to this bit is ignored if data is not valid and key for the selected key length in MXAES_STAT is not valid. The START_DEC is cleared only by hardware as soon as the flag BSY is cleared. • 0x0 **NDEC**: No decryption operation is launched • 0x1 **DEC**: A decryption operation is launched

Table 118 Access modes of MXAES_CMD

Condition	Access mode	Description
AES_BUSY and Debug	read/write	This register is read only. Writes are ignored.
AES_BUSY	read only	This register is read only in this mode. Writes to this register trigger a BusFault.
(default)	read/write	This register has no access restriction.

16.5.6 AES status register

The status register provides information about the general status of the AES engine and the key state. Key flags are a feedback information for the software to check the keys status given that key registers are not readable. In particular, the below flags indicate which keys are loaded and whether AES is busy/ready or not. The AES_STAT.xxx_KEY_xxx flags are set only after all 2 key shares are loaded. Only the AES_STAT.xxx_KEY_xxx flags which correspond to the set CRYPT_MODE are reliable. Changing the CRYPT_MODE sets the AES_STAT.xxx_KEY_xxx flags corresponding to that mode if enough key portions have been written for this selected mode. This avoids reloading the key when switching to a different CRYPT_MODE. For example, only when AES_CFG={AES128HI, CIPHER} and all key words AES_KEY_x_y (for x=0 to 1 and y=4 to 7) have been loaded will the AES_STAT.AES_KEY_128HI flag be set to 1B. No AES operation can be started until the relevant key status flag is set.

MXAES_STAT Offset address: 0x0014

AES status register Reset value: Table 120

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res	START_DEC_RR	START_ENC_RR	CRYPT_MODE_RR					Res			CALC_MODE_RR				
r	rh	rh	rh					r			rh				

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BSY	RDY	SEED	RES	SEC_LEVEL_RR				Res			AES_KEY_256	Res	AES_KEY_128HI	AES_KEY_128LO	DATA
rh	rh	rh	r	rh				r			rh	r	rh	rh	rh

Field	Bits	Type	Description
DATA	0	rh	**AES data valid** This data validation flag becomes valid when a complete 128-bit data block is written to the registers DATA_0_0, DATA_0_1, DATA_0_2, DATA_0_3, DATA_1_0, DATA_1_1, DATA_1_2 and DATA_1_3 registers. • 0x0 **DNV**: Data has not been completely loaded • 0x1 **DV**: Data is valid
AES_KEY_128LO	1	rh	**AES 128 low key valid** • 0x0 **ALIV**: 128-bit AES key low has not been completely loaded • 0x1 **ALV**: 128-bit AES key low is valid
AES_KEY_128HI	2	rh	**AES 128 high key valid** • 0x0 **AHIV**: 128-bit AES key high has not been completely loaded • 0x1 **AHV**: 128-bit AES key high is valid
AES_KEY_256	4	rh	**AES 256 Key valid** • 0x0 **A2IV**: 256-bit AES key has not been completely loaded • 0x1 **A2V**: 256-bit AES key is valid
SEC_LEVEL_RR	11:8	rh	**SEC_LEVEL reread bits**
SEED	13	rh	**Seeding status bit** Reading this bit gives the programmer the information if a seeding has been performed. This bit is permanently set to '1'. • 0x0 **NO_SEED**: Seeding has not been performed yet • 0x1 **SEED**: Seeding already done
RDY	14	rh	**Ready flag** The ready flag indicates whether the result of an operation is available. It is asserted as soon as a valid result is available. This flag is deasserted in one of the following cases: MXAES_CLEAR.DATA or CLEAR_ALL bit is set, any DATA_X_Y register is written or any legal write to CFG register. • 0x0 **AES_NRDY**: Result is not ready to be read • 0x1 **AES_RDY**: Result is valid and ready to be read

(table continues...)

(continued)

Field	Bits	Type	Description
BSY	15	rh	**Busy flag** The busy flag indicates whether the AES engine is busy or not. Attention this flag indicates whether an encryption or decryption is undergoing or not. • 0x0 **NBSY**: Module is not busy with an AES operation. In this status keys can be written • 0x1 **BSY**: : Module is busy doing a cryptographic operation: neither data nor keys can be written
CALC_MODE_RR	20:16	rh	**AES CALC_MODE reread bits**
CRYPT_MODE_RR	28:24	rh	**AES CRYPT_MODE reread bits**
START_ENC_RR	29	rh	**AES START_ENC reread bit** This bit is set by hardware as soon as CMD.START_ENC is set by software. This bit is cleared by hardware as soon as STAT.BSY is cleared.
START_DEC_RR	30	rh	**AES START_DEC reread bit** This bit is set by hardware as soon as CMD.START_DEC is set by software. This bit is cleared by hardware as soon as STAT.BSY is cleared.

Table 119 Access modes of MXAES_STAT

Condition	Access mode	Description
Debug	read/write	This register is read only. Writes are ignored.
(default)	ready only	This register is read only. Writes trigger a BusFault.

Table 120 Reset values of MXAES_STAT

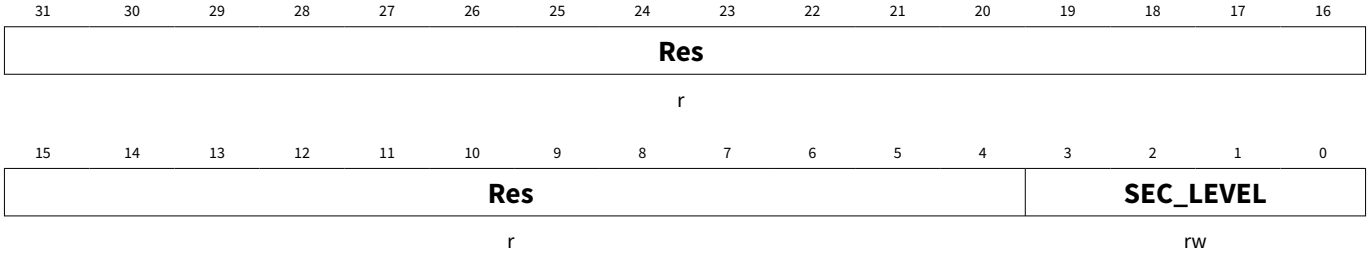
Condition	Access mode	Description
Power-on reset	0000 0000 0000 0000 00X0 0000 0000 0000b	Value set by power-on reset. This value depends on the NRNGA_INTEGRATION parameter.
System reset	0000 0000 0000 0000 00X0 0000 0000 0000b	Value set by system reset. SEED is not changed.
Module reset	0000 0000 0000 0000 00â€“0 0000 0000 0000b	Value set by CLEAR_ALL. SEED is not changed.

16.5.7 AES configuration register 2

The second configuration register allows to configure the overall security level of AES. Modifications of MXAES_CFG_2 are allowed as long as the module is not busy with an operation. Otherwise it triggers a BusFault. Note that it is not necessary to write this register if the default reset value is convenient for the application.

MXAES_CFG_2 Offset address: 0x0060

AES configuration register 2 Reset value: 0x0000 0000



Field	Bits	Type	Description
SEC_LEVEL	3:0	rw	**Security level** This field defines the overall security level. This feature consists of randomizing the AES execution and increases the resistance against side channel attacks targeting a key extraction as well as the strength of the backward calculation against differential fault analysis. This feature has a significant impact on the AES performance. The default value 0x0 gives the least randomization level with least performance drop impact. The security level can be increased by setting any valid values from 0x1 to 0xA to this field. Any value greater than 0xA is evaluated as 0xA.

Table 121 Access modes of MXAES_CFG_2

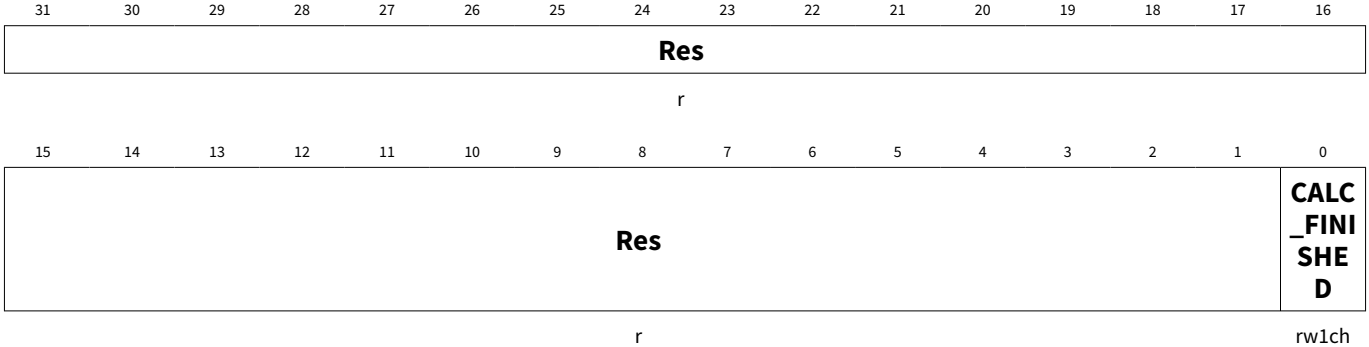
Condition	Access mode	Description
AES_BUSY and Debug	read/write	This register is read only. Writes are ignored.
AES_BUSY	read only	This register is read only in this mode. Writes to this register trigger a BusFault.
(default)	read/write	This register has no access restriction.

16.5.8 AES interrupt cause register

This register holds the interrupt causes of the module. It is set to 1 by hardware and set to 0 by software (RW1C attribute in the memory map (Readable and write one to clear)).

MXAES_INTR Offset address: 0x0080

AES interrupt cause register Reset value: 0x0000 0000



Field	Bits	Type	Description
CALC_FINISHED	0	rw1ch	**Interrupt cause calculation finished** This cause bit is set to 1 by hardware when the AES engine is not busy anymore with a calculation (also notified by the STAT.BSY flag). This is the unique interrupt cause of the module. This interrupt is cleared by software by either writing a 1 or by module reset or with the CLEAR_ALL bit. The software clear overrules the hardware set in case both happen simultaneously. • 0x0 **CLEARED**: No interrupt or interrupt cleared by software or module reset • 0x1 **CLEAR**: Clear interrupt

Table 122 Access modes of MXAES_INTR

Condition	Access mode	Description
AES_BUSY and Debug	read/write	This register is read only. Writes are ignored.
AES_BUSY	read only	This register is read only in this mode. Writes to this register trigger a BusFault.
(default)	read/write	This register has no access restriction.

16.5.9 AES interrupt set register

This register provides software with the capability to set a specific interrupt cause bit in MXAES_INTR register. The cause bit is set to 1 by software (RW1S attribute in the memory map). This functionality is useful for debug purposes. This register does not require dedicated storage elements, but reuses the MXAES_INTR register.

MXAES_INTR_SET Offset address: 0x0084

AES interrupt set register Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res															CALC_FINISHED
r															rw1s

Field	Bits	Type	Description
CALC_FINISHED	0	rw1s	**Interrupt set calculation finished** Interrupt set bit corresponding to interrupt cause CALC_FINISHED bit. This bit is readable. Only the software can set it by writing 1.

Table 123 Access modes of MXAES_INTR_SET

Condition	Access mode	Description
AES_BUSY and Debug	read/write	This register is read only. Writes are ignored.
AES_BUSY	read only	This register is read only in this mode. Writes to this register trigger a BusFault.
(default)	read/write	This register has no access restriction.

16.5.10 AES interrupt mask register

This register holds the interrupt mask bits of co-located interrupt cause bits in INTR register.

MXAES_INTR_MASK Offset address: 0x008C

AES interrupt mask register Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res															CALC_FINISHED
r															rw

Field	Bits	Type	Description
CALC_FINISHED	0	rw	**Interrupt mask calculation finished** The interrupt cause bit will only cause an interrupt when its mask bit is set.

Table 124 Access modes of MXAES_INTR_MASK

Condition	Access mode	Description
AES_BUSY and Debug	read/write	This register is read only. Writes are ignored.
AES_BUSY	ready only	This register is read only in this mode. Writes to this register trigger a BusFault.
(default)	read/write	This register has no access restriction.

16.5.11 AES masked interrupt register

This register is the logical AND of the MXAES_INTR and MXAES_INTR_MASK registers. Typically, an Interrupt Service Routine (ISR) starts by identifying the cause bit that caused the interrupt. Without this register, two MMIO register reads are required: one read for MXAES_INTR and one read for MXAES_INTR_MASK. With the

register, only a single MMIO register read is required. This gain seems small, but ISRs are typically latency sensitive, and the MMIO register reads may travel a long way over the chip's bus infrastructure. Furthermore, this register comes cheap as it does not require dedicated storage elements, but uses the existing AES_INTR and AES_INTR_MASK registers.

MXAES_INTR_MASKED Offset address: 0x0090

AES masked interrupt register Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res															
r															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res															CALC_FINISHED
r															rh

Field	Bits	Type	Description
CALC_FINISHED	0	rh	**Masked interrupt calculation finished** Logical AND of the MXAES_INTR.CALC_FINISHED and MXAES_INTR_MASK.CALC_FINISHED

Table 125 Access modes of MXAES_INTR_MASKED

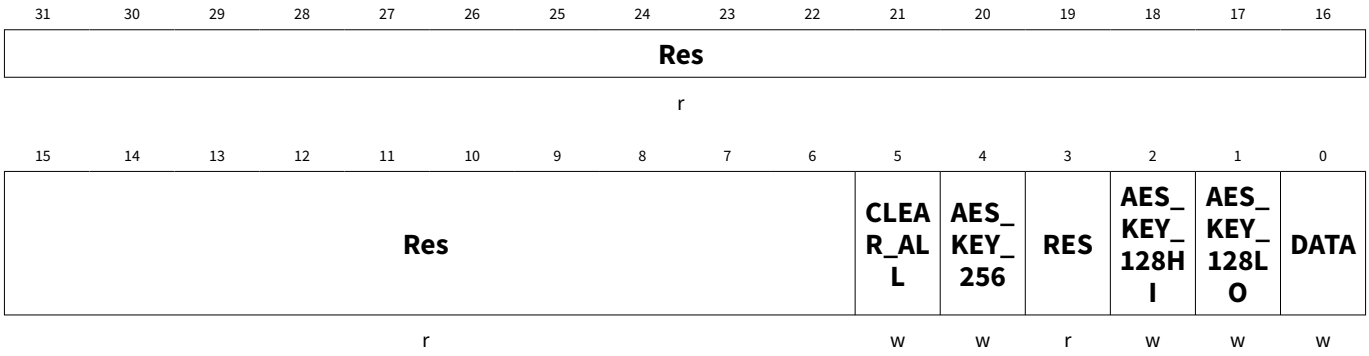
Condition	Access mode	Description
Debug	read/write	This register is read only. Writes are ignored.
(default)	read only	This register is read only. Writes trigger a BusFault.

16.5.12 AES clear register

The clear register gives the user the possibility to clear keys from the key state or clear data status flags (valid or not). Clearing a status flag also clears all words and shares of the corresponding data or key state keys. This is a write-only register. Reading the MXAES_STAT status bits after having written the MXAES_CLEAR allows the user to check if the flag clearing settings - through MXAES_CLEAR write access - have been executed correctly. A CLEAR_ALL set overrules when combined with any other bit set.

MXAES_CLEAR Offset address: 0x00C0

AES clear register Reset value: 0x0000 0000



Field	Bits	Type	Description
DATA	0	w	**Clear data** • 0x0 **NCLR**: Data keeps its value • 0x1 **CLR**: Data clearing triggered
AES_KEY_128LO	1	w	**Clear AES 128 low key** • 0x0 **NCLR**: 128-bit AES key low keeps its value • 0x1 **CLR**: 128-bit AES key is cleared
AES_KEY_128HI	2	w	**Clear AES 128 high Key** • 0x0 **NCLR**: 128-bit AES key high keeps its value • 0x1 **CLR**: 128-bit AES key high is cleared
AES_KEY_256	4	w	**Clear AES 256 key** • 0x0 **NCLR**: 256-bit AES key keeps its value • 0x1 **CLR**: 256-bit AES key is cleared if it is set in AES_STAT. This clears AES_KEY_128HI and LO keys
CLEAR_ALL	5	w	**Clear all SFRs and AES internal registers** Setting this bit behaves as a module reset. It clears all AES keys, data, all SFRs and all AES internal registers. • 0x0 **NCLR**: Clear is disabled • 0x1 **CLR**: A clear all is triggered

Table 126 Access modes of MXAES_CLEAR

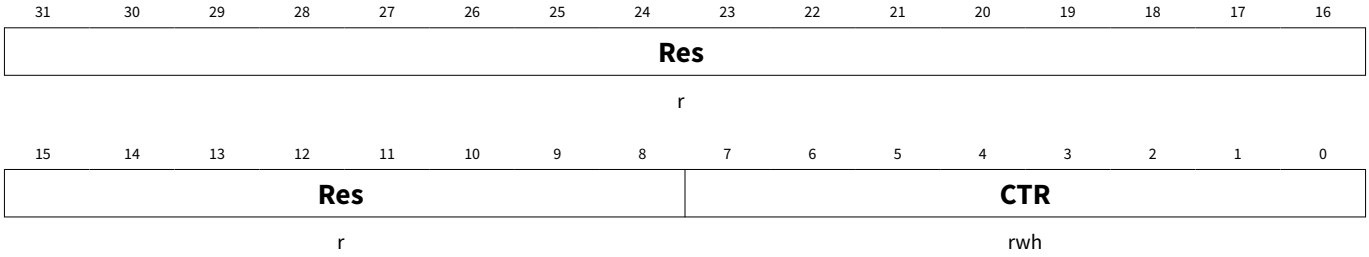
Condition	Access mode	Description
AES_BUSY and Debug	read/write	This register is write only. Writes are ignored and reads return 0xDEADDEAD.
AES_BUSY	none	Any access to this register triggers a BusFault.
Debug	read/write	No restrictions
(default)	write only	This register is write only. Reads trigger a BusFault.

16.5.13 AES bus counter register

This register supports the software in keeping track of the number of data and key shares written.

MXAES_BUS_COUNTER Offset address: 0x00DC

AES bus counter register Reset value: 0x0000 0000



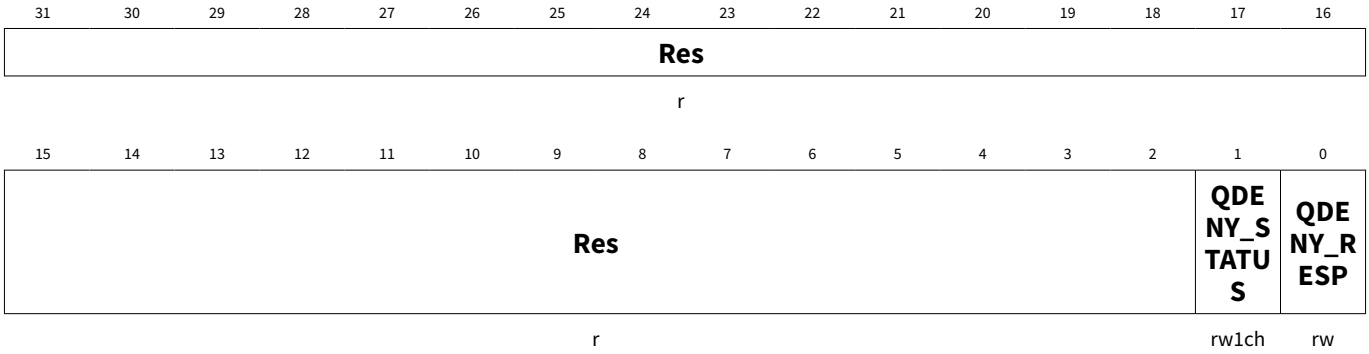
Field	Bits	Type	Description
CTR	7:0	rwh	**Key and data write counter** The counter is incremented on every CPU bus write access of data or key. On Debug write access of key or data registers the counter is not incremented. It is reset on module reset but never reset otherwise. When it reaches 0xFF and a key or data bus write happens, it is incremented to 0x00 while the overflow is ignored.

16.5.14 AES Q-channel response register

This register stores the Q-CHANNEL response bit. It allows the software to determine whether to assert qdeny or qaccept when qreq is asserted. If the product does not use Q-Channel, this register has no effect and can be ignored.

MXAES_QCHAN_CTL Offset address: 0x00E0

AES Q-channel response register Reset value: 0x0000 0000



Field	Bits	Type	Description
QDENY_RESP	0	rw	**Q-Channel QDENY Response Policy** This policy bit provides software flexibility to determine whether to assert qdeny or qaccept when qreq is asserted but the IP is not quiescent. Note: the AES always accepts the request. • 0x0 **ACCEPT**: A quiescence request is always accepted • 0x1 **DENY**: A quiescence request may be denied

(table continues...)

(continued)

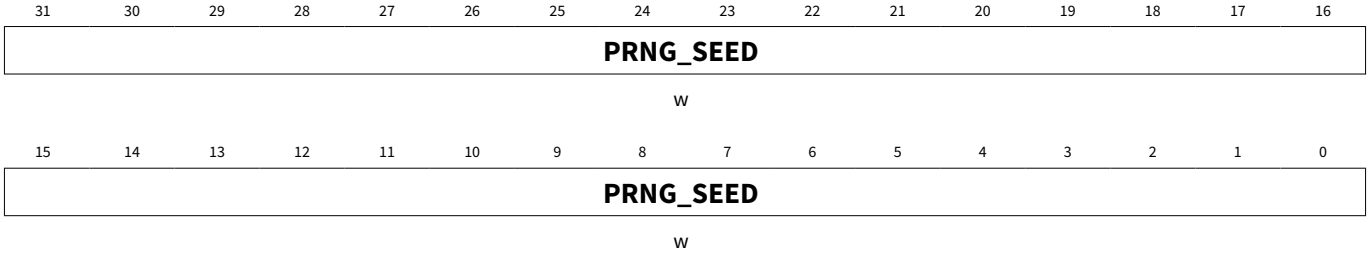
Field	Bits	Type	Description
QDENY_STATUS	1	rw1ch	**Q-Channel QDENY Status** This status bit allows dynamic debug when a qdeny occurs. • 0x0 **NODENY**: No quiescence request was denied • 0x1 **DENIED**: Q quiescence request was denied

16.5.15 AES user seeding register

The SEED_U register can be used to trigger the internal seeding of AES that brings the internal mask units into a random state. This register provides a random number for the mask initialization. This register is write only.

MXAES_SEED_U Offset address: 0x1000

AES user seeding register Reset value: 0x0000 0000



Field	Bits	Type	Description
PRNG_SEED	31:0	w	**Seed for internal masking** Writing to this register triggers the internal seeding of the PRNG mask generator of AES. The content of this register is used to seed the AES mask generator in addition to the NRNGA internal noise source. It does not reset the PRNG, it actually adds up entropy to the entropy of the NRNGA internal noise source. This register is not readable however writable at any time also during STAT.BSY status.

Table 127 Access modes of MXAES_SEED_U

Condition	Access mode	Description
Debug	read/write	This register is write only. Reads return 0xDEADDEAD.
(default)	write only	This register is write only. Reads trigger a BusFault.

16.5.16 AES Key registers

This section describes the SFR interface of the key state. 'x' refers to the index of shares and y for the index of the word. For security reasons, key registers cannot be read. Note that any SFR write is XORed with the previous register content. The chronological order of writing of the key words and shares does not matter.

AES_KEY_0_0 corresponds to the bits 31 down to 0 bits in the key state and AES_KEY_0_7 to the bits 255 down to 224 of the first share.

AES_KEY_1_0 corresponds to the bits 31 down to 0 bits of the second share and AES_KEY_1_7 to the bits 255 down to 224 of the second share.

Note that writing two key shares into the words KEY_0-1_y is mandatory. A first complete key block must be written into the registers KEY_0_0 to KEY_0_* and a second into AES_KEY_1_0 to KEY_1_*. For example, for AES256 all key words KEY_x_y(x=0-1, y=0-7) has to be written at least once. This sets the key valid flag corresponding to its CRYPT_MODE. Writing at least once KEY_0_4, KEY_1_4, KEY_0_5, KEY_1_5, KEY_1_6, KEY_1_6, KEY_0_7, KEY_1_7 sets the key valid flag of AES128HI. Note that updating an AES key after a decryption operation has been performed leads to an unpredictable behavior or key state's content.

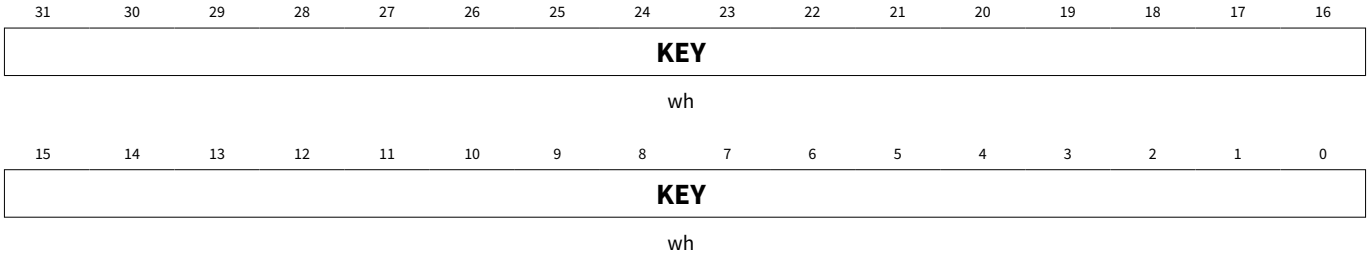
All MXAES_KEY_X_Y registers of the key state are cleared in these cases:

- Change CRYPTO_MODE from AES128LO/HI to AES256 and conversely
- Perform a clear with the clear register bit fields AES_256_KEY or CLEAR_ALL

16.5.16.1 AES key register (x for index of shares; y for index of word)

MXAES_KEY_x_y (x=0-1;y=0-7) Offset address: 0x1100+x*0x60+y*12

AES key register [x for index of shares; y for index of word) Reset value: 0x0000 0000



Field	Bits	Type	Description
KEY	31:0	wh	**Key value**

Table 128 Access modes of MXAES_KEY_x_y (x=0-1;y=0-7)

Condition	Access mode	Description
AES_IDLE and Debug	read/write	This register is write only. Reads return 0xDEADDEA.
AES_BUSY and Debug	read/write	Writes are ignored and reads return 0xDEADDEAD.
AES_IDLE and CPU	write only	Reads trigger a BusFault.
AES_BUSY and CPU (default)	none	Any access to this register triggers a BusFault.

16.5.17 AES Data registers

This section describes AES data shares and how and when to read or write the yth word of the xth data share.

After Reset, once the module has been configured by setting the crypto mode and loading its corresponding key words in shares as appropriate into the key state, the module is ready to receive data. The chronological order of writing/reading of the data words and shares does not matter. DATA_X_Y and KEY_X_Y words and shares writes can be interleaved as wished.

Note that any data SFR write is XORed with the data register content. A first complete data block must be written into the registers AES_DATA_0_0 to AES_DATA_0_3 and a second into AES_DATA_1_0 to AES_DATA_1_3. This sets the data valid flag. The encrypted/decrypted data result can be read from this register if AES_STAT.RDY=1B. DATA_0_0 corresponds to the bits 31 down to 0 bits of the 128-bit first share of data to load and DATA_0_3 to the bits 127 down to 96. DATA_1_0 corresponds to the bits 31 down to 0 bits of the 128-bit second share of data to load and DATA_1_3 to the bits 127 down to 96. Note that writing two data shares is mandatory and as soon as the corresponding eight words are written the AES_STAT.DATA flag is set.

This DATA_RR_x_y registers allow the user to check that data transfers to and from the AES_DATA_x_y registers have been performed correctly by means of a verification transfer after a AES_DATA_x_y data transfer. Write or read the yth word of the xth data share. DATA_RR_0_0 corresponds to the bits 31 down to 0 bits of the 128-bit first share of data to load and DATA_RR_0_3 to the bits 127 down to 96. DATA_RR_1_0 corresponds to the bits 31 down to 0 bits of the 128-bit second share of data to load and DATA_RR_1_3 to the bits 127 down to 96. The reread registers DATA_RR_* support the address integrity protection of the DATA_* data exchanges between the processor and the AES AHB module via the peripheral bus through a special register address encoding of the AES_DATA_RR_*. These registers are specially encoded to protect the integrity of the data shares transfer against single needle forcing and single laser shot fault injection attacks.

16.5.17.1 AES data register (x for index of shares; y for index of word)

AES data to encrypt/decrypt can be load from/to this register. When ready, the result can be read from this register.

MXAES_DATA_x_y (x=0-1;y=0-3) Offset address: 0x1400+x*0x30+y*12

AES data register [x for index of shares; y for index of word) Reset value: 0x0000_0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA															
rwh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA															
rwh															

Field	Bits	Type	Description
DATA	31:0	rwh	**Data value**

Table 129 Access modes of MXAES_DATA_x_y (x=0-1;y=0-3)

Condition	Access mode	Description
AES_RDY and AES_NBSY	read/write	No restrictions
AES_NRDY and AES_NBSY and Debug	read/write	This register is writable in this mode. Reads return 0xDEADDEAD.
AES_RDY and AES_BUSY and Debug	read/write	This register is readable in this mode. Writes are ignored.
AES_NRDY and AES_BUSY and Debug	read/write	Reads return 0xDEADDEAD. Writes are ignored.
CPU and AES_NRDY and AES_BUSY (default)	none	Any access triggers a BusFault.

(table continues...)

Table 129 (continued) Access modes of MXAES_DATA_x_y (x=0-1;y=0-3)

Condition	Access mode	Description
CPU and AES_NRDY and AES_NBSY	write only	This register is writable in this mode. Reads return a BusFault.
CPU and AES_RDY and AES_BUSY	ready only	This register is readable in this mode. Writes return a BusFault.

16.5.17.2 AES data reread register (x for index of shares; y for index of word)

This register content is used to check the integrity of the corresponding yth word of the xth data share in DATA_X_Y register.

MXAES_DATA_RR_x_y (x=0-1;y=0-3) Offset address: 0x1490+x*0x30+y*12

AES data reread register [x for index of shares; y for index of word) Reset value: 0x0000_0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DATA															
rh															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA															
rh															

Field	Bits	Type	Description
DATA	31:0	rh	**Data value**

Table 130 Access modes of MXAES_DATA_RR_x_y (x=0-1;y=0-3)

Condition	Access mode	Description
CPU and AES_RDY	read only	This register is read only
CPU and AES_NBSY	read only	DATA_RR_x_y register can be read only if DATA_x_y has been written, otherwise BusFault.
AES_NRDY and AES_BUSY and CPU (default)	none	Any access triggers a BusFault.
AES_NRDY and AES_BUSY and Debug	read/write	Writes are ignored and reads return 0xDEADDEAD.

17 Power supply and monitoring

The PSC3M/P6 MCU operates using a single regulated VDDD supply within the range of 1.71 V to 3.63 V. In addition, there is an optional VBACKUP supply that can be used, which has a range of 1.4 V to 3.63 V. It integrates multiple LDO regulators to power the blocks within the device in various power modes.

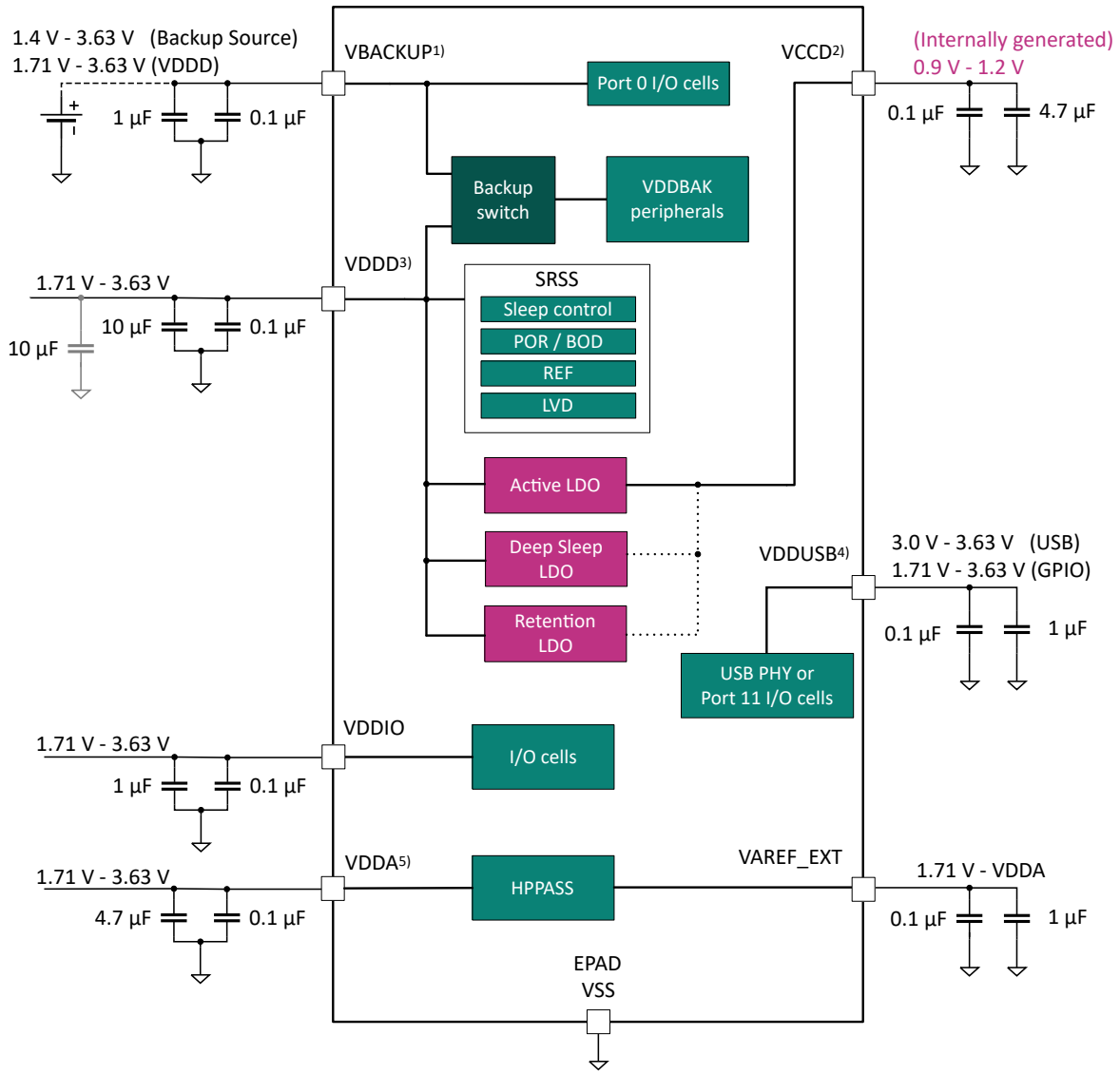
The device supports multiple power supply rails, such as VDDD, VDDA, VDDIOs, and VBACKUP, which enable the application to use a separate supply for different blocks within the device. For example, VDDA powers analog peripherals such as ADC and CSG. The PSC3M/P6 MCU family supports power-on-reset (POR), brown-out-detect (BOD), and low voltage-detect (LVD) circuits for power supply monitoring and failure protection purposes.

17.1 Features

The power supply and monitoring subsystem of the PSC3M/P6 MCU supports the following features:

- Operating voltage range:
  - VDDIO: 1.71 V to 3.63 V
  - VDDA: 1.71 V to 3.63 V
  - VDDD: 1.71 V to 3.63 V
  - VBACKUP: 1.4 V to 3.63 V
- User-selectable core logic operation:
  - Ultra Low Power (ULP): Core logic runs at 0.9 V
  - Medium Frequency (MF): Core logic runs at 1.0 V
  - Low Power (LP): Core logic runs at 1.1 V
  - Overdrive (OD): Core logic runs at 1.2 V
- Multiple independent supply rails:
  - VDDD
  - VDDA
  - VDDIOs
  - VBACKUP
- Multiple on-chip linear regulators:
  - One Active LDO
  - Multiple low-power regulators to power the peripherals operating in different power modes
- Two POR/BOD circuits to monitor VDDD and VCCD
- One low voltage detect (LVD) circuit to monitor VDDD

17.2 Architecture



1) VBACKUP can be connected to a coin cell (1.4 V to 3.63 V) or a supercapacitor, depending on the application. If not used, it should be connected to VDDD.
2) VCCD is internally generated. It is only used to connect external bypass capacitors for the internal LDO. Do not use it to power external circuits.
3) A 10 µF capacitor on VDDD supports full operating voltage range for Active/Hibernate modes and Deep Sleep with VDDD ≥ 2.3 V. Use 2x10 µF capacitors or a single 22 µF capacitor for Deep Sleep operation down to 1.71 V.
4) VDDUSB powers the USB PHY (DP and DM lines) or the Port 11 I/O cells for devices without USB support.
5) VDDA should be ≥ 2.7 V to run HPPASS at the highest supported clock speed of 180 MHz.

Figure 62 Power supply and monitoring system block diagram

Figure 62 shows the typical powering configuration with a single regulated supply used for all supply pins and VBACKUP connected to a coin cell. Isolated sources can be used; for example, to isolate the analog circuits, VDDA can be connected to an independent regulated supply between 1.71 V to 3.63 V.

The regulators and supply pins/rails shown in Figure 62 power various blocks inside the device. The availability of various supply rails/pins for an application depends on the device package selected. Refer to the device datasheet for details.

All core regulators draw their input power from the VDDD supply pin. The VCCD supply generated by the Active LDO provides power to all active domain peripherals in Active/Sleep mode. Deep Sleep LDO powers the Deep

Sleep peripherals, while Retention LDO powers the retention logic for SRAM in Deep Sleep and Deep Sleep-RAM power modes. In Hibernate mode, all internal regulators are switched off, and Hibernate peripherals such as LPComp and Sleep Control are powered directly from the VDDD supply.

The I/O cells operate from various VDDIOx pins, depending on the port where they are located. The VCCD supply drives the logic inside the I/O cells from core logic peripherals. VDDA powers the HPPASS (SAR ADC and CSG) and the analog logic, such as analog mux switches inside the I/O cells. To know which I/Os operate from which supply, refer to the device datasheet.

In addition to the power rails and regulators, the device provides options to monitor supply rails and provide protection against potential supply failures. These include POR/BOD circuits and an LVD circuit. The Sleep Control block provides the circuits required for entry and exit from low power modes.

17.3 Power supply

The device includes the following core regulators to power peripherals in various power modes:

17.3.1 Active regulator

The device includes a linear LDO regulator to power the Active and Sleep mode peripherals. This active linear regulator generates the core voltage (VCCD) required for Active mode operation of the peripherals from VDDD. The regulator is capable of providing 0.9 V, 1.0 V, 1.1 V, and 1.2 V[2)] for core operation. When changing the active regulator output voltage, it takes some time to stabilize, especially when changing to a higher voltage. This should be taken into account if the core voltage is changed by directly writing to the configuration registers. The PDL function includes the stabilization delay.

The regulator is available in Active and Sleep power modes. This regulator implements two sub-modes:

- High-current mode
- Low-current mode

The high-current mode is the normal operating mode; that is, the device operates to its full capacity in Active power mode. In the low-current mode, the current output from the regulator as well as its quiescent current is limited. This mode implements the LPActive, Sleep and LPSleep power modes. The low-current mode sets limitations on the capabilities and availability of resources in the Sleep mode. For example, in case of LPActive and LPSleep sub-modes, the maximum supported clock frequency is limited. For details, see Device power modes. The device always starts in Active mode; therefore, the Active LDO is always turned on at power on or when going to Active mode from any other operating mode. In power modes other than Active and Sleep, the active linear regulator is turned off.

17.3.2 Deep Sleep regulator

The Deep Sleep regulator powers the peripherals available in Deep Sleep mode. The regulator is optimized for low quiescent current operation to reduce the device's operating current in low power modes.

17.3.3 Retention regulator

The retention LDO reduces the Deep Sleep current by supplying a lower voltage to the retention flops and SRAM core in Deep Sleep mode. The lower supply voltage ensures the minimum retention voltage requirements are met while minimizing the leakage current.

2 Refer to the device datasheet for available core voltages.

17.3.4 Backup domain

A backup domain is an independent power domain. During normal operation, the VDDD supply provides power to the backup domain peripherals. In the absence of VDDD supply, the backup switch goes to the normally closed position, thereby utilizing the VBACKUP supply input to power the backup domain peripherals such as RTC. The user can either connect a backup source, such as a coin cell or supercapacitor, to the VBACKUP pin or connect it externally to the VDDD if backup functionality is not required.

17.4 SRSS reference block

The SRSS reference block contains two instances of 0.8 V band gap reference (BGR) voltage generators and a 250 nA current reference. These are used by the internal regulators, voltage monitors, PLLs, and so on. The Active BGR available during Active/Sleep modes consumes 20 uA of current, while the Deep Sleep BGR available during Deep Sleep mode consumes only 0.2 uA of current.

17.5 Voltage monitoring

The PSOC™ Control series MCU offers multiple voltage monitoring and supply failure protection options. This includes POR, BOD, and LVD.

17.5.1 Power-on-reset (POR)

POR circuits provide a reset pulse during the initial power ramp. POR circuits monitor VDDD and VCCD voltages. POR circuits are used during the initial chip power-up and then disabled. Refer to the device datasheet for details on the POR trip-point levels.

17.5.2 Brown-out detection (BOD)

The BOD circuit protects the operating or retention logic from possibly unsafe supply conditions by applying a reset to the device. The BOD monitors the VDDD voltage and generates a reset if the voltage excursion dips below the minimum VDDD voltage required for safe operation (see the device datasheet for details). The system will not come out of RESET until the supply is detected to be valid again. The BOD circuit guarantees a reset in Active, Sleep, Deep Sleep-RAM, and Deep Sleep-Off power modes before the system crashes.

There is no on-chip BOD support in Hibernate mode. Applications that require BOD support during Hibernate need to use external supervisor circuits. There is no configuration register for BOD. For details on the BOD trip points, supported supply ramp rate, and brown-out detection response time, refer to the device datasheet.

17.5.3 Low voltage detect (LVD)

An LVD circuit monitors the external supply voltage and generates an interrupt or fault while the device is still within its operating limits. Firmware can utilize the LVD interrupt to implement proactive measures for prevention. Use the PWR_LVD_CTL register to customize and adjust the trip point of the LVD. The PSC3M/P6 MCU incorporates a single LVD on the VDDD rail. It is active and functional during Active/Sleep modes. LVD is not available during Deep Sleep and Hibernate modes. By default, the LVD is disabled to conserve power since its operation consumes energy. It is important to note that LVD does not serve as a reset source; instead, the system reset responsibility lies with the BOD circuits. However, if desired, the firmware can initiate a system reset through the LVD interrupt.

The HVLVD1_TRIPSEL [3:0] bit-field is used to select one of the sixteen supported voltage thresholds. See the PWR_LVD_CTL register description in the PSC3M/P6 MCU Register TRM for threshold value details. The HVLVD_SRCSEL[6:4] bit-field only supports VDDD monitoring by using the default value of 0x0 or by using 0x1 and 0x4 to select the VDDD branch through AMUXBUSA or AMUXBUSB, respectively. Other values are reserved.

The HVLVD1_EN [7] bit is used to enable the LVD. Ensure that the LVD is disabled before changing any other configuration bits in the PWR_LVD_CTL register. The HVLVD1_EDGE_SEL [17:16] bit-field is used to select which edge (s) will trigger an action when the threshold is crossed. This can be configured to disabled, rising edge, falling edge, or both rising and falling edges.

17.6 Sleep control

Sleep control contains the required logic and circuits to enable power mode transitions from Active/Sleep to Deep Sleep and Hibernate and wake up to Active/Sleep mode. There is a possibility that co-processors and peripherals are operating without CPU intervention. To ensure that these operations are not impacted when entering Deep Sleep, sleep control implements a quiescence request and response scheme. When the Deep Sleep entry is requested, the CPU sends a quiescence request to PERI, which expands this request to the relevant peripherals. These requests are sent on dedicated interfaces called Q-channels. If any peripheral denies the request, the Deep Sleep entry is aborted.

The Wake-up Interrupt Controller (WIC) is used for sensing the inputs that can wake up the system from Deep Sleep and Hibernate mode. The WIC forwards the wake-up request to the main Power Policy Unit (PPU), and the PPU in turn executes the wake-up sequence to bring the device into Active mode. For further details, refer to Device power modes.

17.7 Registers summary

Table 131 Power Supply and Monitoring register summary

Register	Function
PWR_LVD_CTL	High Voltage/Low Voltage Detector (HVLVD) configuration register
PWR_LVD_STATUS	High Voltage/Low Voltage Detector (HVLVD) status register
PWR_CTL2	POR/BOD, voltage/current reference, LDO control register

18 Device power modes

The PSC3M/P6 MCU device supports multiple power modes intended to minimize average power consumption in an application. PSC3M/P6 MCU supports six low-power modes to achieve the best compromise between low-power consumption, short start-up time, available peripherals, and available wake-up sources.

The Peripheral Driver Library (PDL) supports all device power mode transitions and is the recommended method for transitioning and configuring the PSC3M/P6 MCU power resources.

18.1 Features

The PSC3M/P6 MCU power modes provide the following features:

- Software can utilize power modes to optimize power consumption in an application
- DeepSleep mode supports multiple wake-up sources and configurable amounts of SRAM retention
- Hibernate mode allows wake-up from I/O, LPComp, RTC, and timer alarms

Power consumption in different power modes is further controlled by using the following methods:

- Enabling and disabling clocks to peripherals
- Powering on/off clock sources
- Powering on/off peripherals and parts inside PSC3M/P6 MCU

18.2 Architecture

The PSC3M/P6 MCU supports the following power modes, in the order of decreasing power consumption:

- **Active/Sleep**: In Active mode, all peripherals and the CPU are available. In Sleep mode, all peripherals except the CPU are available
  - **Low-Power Active (LPACTIVE)/Low-Power Sleep (LPSLEEP)**: These modes are low-power profiles implemented as register configurations within Active/Sleep modes
- **Deep Sleep**: In this mode, active logic is frozen, but low-speed functions are available. Most configurations and memory states are retained. High-speed clock sources are turned off. Wakeup returns the processor to where it left off
- **Deep Sleep RAM**: Active mode logic is not retained, which necessitates a processor reset upon wakeup. The entire system RAM can be retained. Wakeup causes the processor to ‘warm’ reboot
- **Deep Sleep OFF**: Active mode logic and memories are not retained. Wakeup causes the processor to ‘cold’ reboot
- **Hibernate**: The lowest power mode where VDDD is still present. The device and I/O states are frozen, and the device resets on wake up

Two embedded linear voltage regulators supply power to most of the digital circuitry in the device: the linear active regulator and the Deep Sleep regulator. The linear active regulator is utilized in the Active and Sleep modes, while the Deep Sleep regulator, along with the retention and NWELL regulators, is used in the Deep Sleep, Deep Sleep RAM, and Deep Sleep OFF modes.

The device supports multiple voltage operating points, allowing for dynamic voltage scaling to optimize power consumption in the Active mode. The voltage from the linear active regulator, which powers the logic core (VCCD), can be adjusted based on the system’s maximum operating frequency.

The power management unit system operates in the following mode selections:

- **Ultra Low Power (ULP) mode**: 0.9 V setting with CPU running at 50 MHz
- **Mid Frequency (MF) mode**: 1.0 V setting with CPU running at 70 MHz
- **Low Power (LP) mode**: 1.1 V setting with CPU running at 150 MHz
- **Over Drive (OD) mode**: 1.2 V setting with CPU running at 180 MHz

The PSC3M/P6 MCU device operates with a single external supply (VDDD) in the range of 1.71 V to 3.63 V and features six power modes, with transitions managed by the power subsystem. The backup power supply (VBACKUP) operates within the range of 1.4 V to 3.63 V. When the external VBACKUP is not connected, the supply on VDDD is used as the backup power supply.

18.2.1 Power modes

The PSC3M/P6 MCU device supports the following power modes:

Table 132 PSC3M/P6 MCU power modes

Power mode	Description	Entry condition	Wake-up source	Wake-up action
Active	• Primary mode of operation with all peripherals and CPU available	Wake up from Sleep, all Deep Sleep modes, Hibernate reset, or any other reset	Not applicable	Not applicable
Sleep	• CPU is in Sleep mode; all other peripherals are available	Register write from Active mode or wake up from Deep Sleep through debugger	Any interrupt to CPU	Interrupt
Deep Sleep	• All high-frequency clocks and peripherals are turned off • Low-frequency clocks and low-power analog and digital peripherals are available for operation and as wake-up sources	Register write from Active [1)]	GPIO interrupt, Low-power comparator, SCB-instance, watchdog timer, RTC alarms, and debugger	Interrupt or debug
Deep Sleep RAM	• A low-power mode, similar to Deep Sleep, except most configuration states are not retained • Wakeup causes the processors to reboot • The software can use the retained SRAM contents (programmable) to more quickly recover the application state, enabling a 'warm' reboot	Register write from Active	GPIO interrupt, Low-power comparator, SCB-instance, watchdog timer, and RTC alarms	Warm reboot

(table continues...)

Table 132 (continued) PSC3M/P6 MCU power modes

Power mode	Description	Entry condition	Wake-up source	Wake-up action
Deep Sleep OFF	• A low-power mode, similar to Deep Sleep, except most configuration states and memories are not retained • Wakeup causes the processors to perform a 'cold' reboot	Register write from Active	GPIO interrupt, Low-power comparator, SCB-instance, watchdog timer, and RTC alarms	Cold reboot
Hibernate	• GPIO states are frozen; all high-frequency clocks and peripherals are switched off • Low-frequency clocks and WCO can function • Device resets on wake-up event	Register write from Active	WAKEUP pins, Low-power comparator, RTC alarms, and watchdog timer	Hibernate reset

1) For Deep Sleep entry considerations, refer notes in Deep Sleep

18.2.1.1 Active and Sleep

Active mode: The device enters Active mode upon any reset, which is the normal operating power mode where all device resources used by the system are active and clocked. In this mode, the CPU executes code along with all logic and is memory-powered. The firmware may decide to enable or disable specific peripherals and power domains based on the application and power requirements. All peripherals are available for use in Active mode.

Sleep mode: In Sleep mode, the CPU clock is turned off, and the CPU halts code execution to enter sleep. All peripherals available in Active mode are also available in Sleep mode. Peripheral activity can continue; if the CPU is required, it can be activated via clock gating instantaneously. Any unmasked interrupt can wake up the CPU to Active mode.

LPACTIVE/LPSLEEP: They are low-power profiles implemented as register configurations within Active/Sleep modes to reduce power consumption during Active or Sleep mode. These low-power profiles are similar to Active/Sleep, except the current is limited, clocks may run at a lower frequency, and some functions are not available or are limited.

- **LPACTIVE**:
  - Clock for clk_hf0 (main logic clock) is selected as IMO, which runs at 8 MHz nominal
  - Optional peripherals are turned off, including reference voltage buffer, low voltage detection (LVD), and other clock sources
  - Code runs from flash
  - POR/BOD and references are put in their low-power mode, limiting the maximum external ramp rate on VDDD
- **LPSLEEP**: LPSLEEP relates to LPACTIVE in the same way that Sleep relates to Active

18.2.1.2 Deep Sleep

Deep Sleep mode is a low-power mode in which the active logic is frozen while low-speed functions remain available. Most configuration and memory states are retained, and GPIO states are frozen. In this mode, high-speed clock sources such as the internal main oscillator (IMO), external crystal oscillator (ECO), and phase-locked loop (PLL) are turned off, rendering the CPU, SRAM, high-speed logic, and high-speed peripherals unusable. However, low-speed clock sources like the watch crystal oscillator (WCO) and internal low-speed oscillator (ILO), as well as peripherals configured and enabled by the firmware, continue to operate. The ILO can be turned off to conserve more power. Wake-up returns the processor to where they left off.

This mode allows the operation to resume at the same program counter value, with the state allowing the resumption of operation fully retained. Stack and critical parameters can also be retained, and interrupts from low-speed asynchronous or low-power analog peripherals can cause a CPU to wake up from Deep Sleep mode. Upon a normal wake-up from Deep Sleep, the processor returns to either Active or Sleep, depending on the programmed behavior for the particular wake-up interrupt, while a debug wake-up from Deep Sleep returns to Sleep.

This mode is used in applications where the processor does not need to run the code, but peripherals are required to function at a speed lower than the system clock speed.

***Notes**:*
Important considerations before entering Deep Sleep mode:

1. *Enable IHO*
2. *Set device Active mode power profile to ULP mode (VCCD = 0.9 V)*
3. *Set BOD to low power mode*
4. *Bypass the retention regulator by setting PWR_CTL2.RET_REG_DIS = 1*
5. *Update the bit-fields PWR_TRIM_PWRSYS_CTL2.RET_REG_TRIM = 0x2*
6. *Initiate Deep Sleep entry*

18.2.1.3 Deep Sleep RAM

In Active mode, logic is not retained, requiring a processor reset upon wake-up. However, in Deep Sleep RAM mode, the entire SRAM is retained. The software can store crucial information in the system RAM, enabling a ‘warm’ reboot that returns the application to its previous state more quickly than a full ‘cold’ reboot.

While in Deep Sleep mode, logic is operational, but only the SRAM is retained. When a wake-up occurs, it results in a reset, known as a warm reboot.

***Note**:*

- *The SRAM region (SRAM size minus 8 KB) is utilized by Infineon firmware during the boot operation. Although this region is available to the user, data retention across resets, including return from Deep Sleep RAM, is not guaranteed in this area, as it can be overwritten by Infineon boot firmware*
- *On warm boot (wakeup from Deep Sleep RAM/Deep Sleep OFF), the CPU will be in secure mode with the Protection Context set to 2. The value stored in SRSS BOOT ENTRY must be accessible to Secure/PC2 programs*
- *Peripherals that are inactive during Deep Sleep mode will be reset upon waking from Deep Sleep RAM or Deep Sleep OFF, requiring re-initialization after a warm boot*

18.2.1.4 Deep Sleep OFF

In Active mode, logic and memories are not retained at all, necessitating a full 'cold' reboot to restore the application to its previous state.

During Deep Sleep, logic is functional, but the Active logic and SRAM are powered off. When a wakeup occurs, it results in a reset (cold reboot).

In Deep Sleep OFF mode, SRAM retention is not available.

Table 133 outlines the PSC3M/P6 MCU wake-up behavior after a specific Deep Sleep state.

Table 133 Deep Sleep states and wake-up behavior

Deep Sleep state	After wakeup
Deep Sleep	Wake up from WFI/WFE
Deep Sleep RAM	Warm reboot
Deep Sleep OFF	Cold reboot

18.2.1.5 Hibernate

Hibernate mode is the lowest power mode of the device, designed for applications in a dormant state while external supplies are still present and XRES is de-asserted. In this mode, both the Active and Deep Sleep regulators are turned off, and all clocks are stopped. Additionally, GPIO states are frozen, and only asynchronous events, such as transitions on designated I/O pins and possible reset events, are supported. It is possible to use the comparators in this mode, at the expense of low-speed comparator current. This mode requires the device to undergo a reset.

Upon emerging from Hibernate mode, the Boot ROM code detects the chip's arrival at that state and executes the minimum required code to resume operations, taking advantage of retained data as appropriate. The GPIO state is retained in this mode. Wake-up from Hibernate mode is facilitated through dedicated wake-up pins (P2.0 and P9.0) and a low-power comparator output. The low-power comparator operation in Hibernate mode requires external voltages for wake-up comparison. Additionally, an RTC alarm and the watchdog timer (WDT) interrupt can generate a Hibernate wake-up signal. The configuration of the wake-up source from Hibernate mode can be done using the PWR_HIB_WAKE_CTL register, for example, by setting the HIB_WAKE_RTC of the PWR_HIB_WAKE_CTL register to enable the RTC alarm wake-up from Hibernate mode.

To prevent accidental entry into Hibernate mode in applications that cannot meet its requirements, an option to disable the Hibernate mode is provided. This can be achieved by setting the HIBERNATE_DISABLE bit [30] of the PWR_HIBERNATE register to disable Hibernate mode in the device. Note that this bit is a write-once bit during execution and is cleared only on reset. When entering Hibernate mode, debug functionality is lost, and the debugger disconnects.

The primary difference between Hibernate and Deep Sleep OFF modes is that in the latter, the Deep Sleep logic and regulators are operational, while in Hibernate, only VDDD-supplied logic and backup domain are available.

18.2.2 Other operation states

In addition to the power modes discussed in the previous sections, the device can also be in two other states – Reset and Off states. These states are determined by the external power supply and XRES connections. Entering these modes does not require any firmware action, and exiting them does not necessitate an interrupt or wake-up event. The inclusion of the discussion on the Reset and Off states aims to cover all potential modes and states that the device can enter. These states can be utilized in a system to further optimize power consumption.

18.2.2.1 Reset state (XRES)

Reset is the device state when an external reset (XRES pin pulled low) is applied or when power-on-reset/brownout detection is asserted. Reset does not function as a power mode. During the reset state, all the

components in the device are powered down, and I/Os are tristated, keeping the power consumption to a minimum.

18.2.2.2 Off state (OFF)

The off-state simply represents the device state with no power applied.

18.3 Power mode transitions

Figure 63 shows the various states that the device can be in, along with possible power mode transition paths.

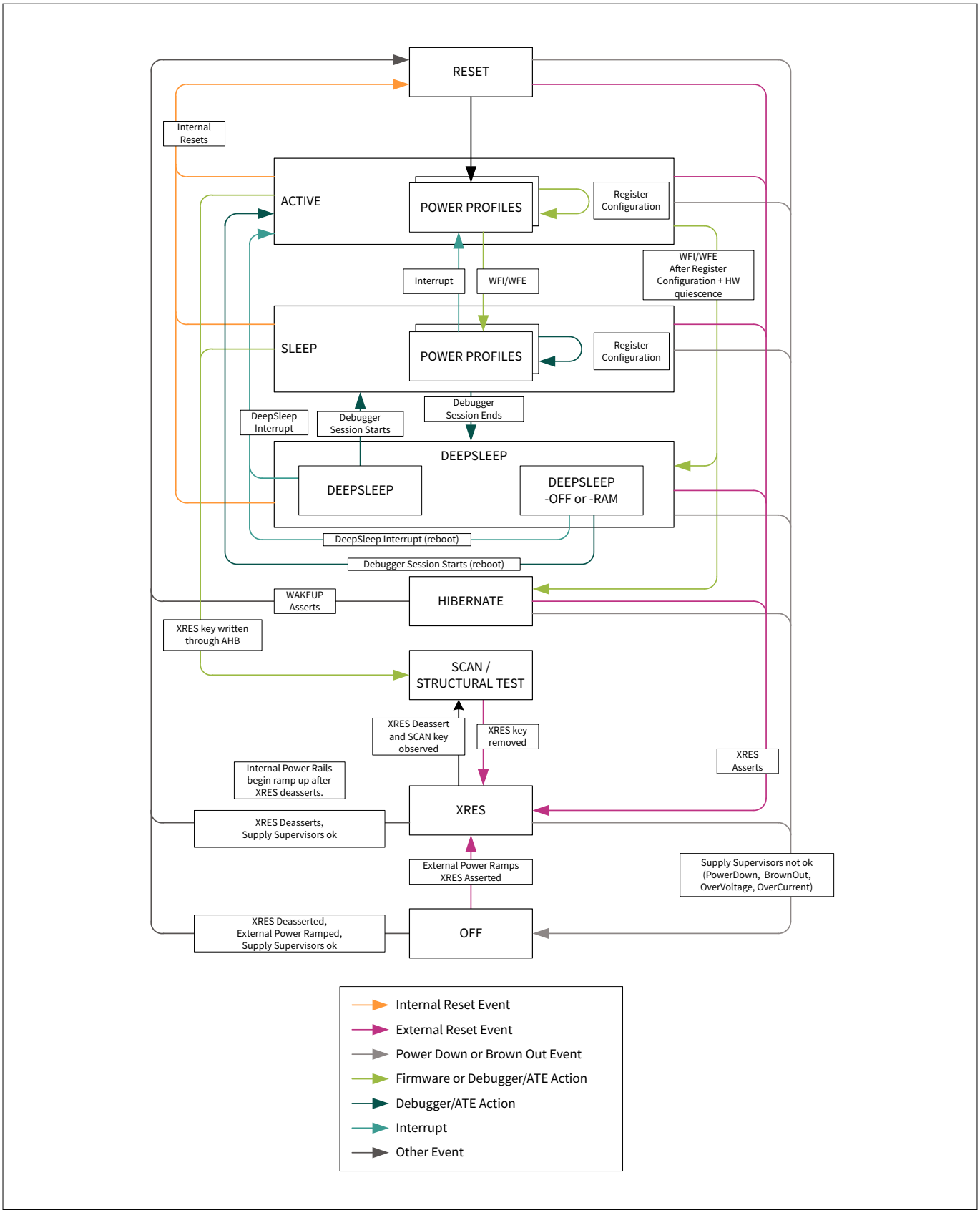


Figure 63 Power mode transitions

***Note**: For Deep Sleep entry considerations, refer notes in Deep Sleep*

18.4 Summary

Table 134 shows the available resources that are active in certain power modes and their corresponding wake-up parameters.

Table 134 Power modes and wake-up parameters

	Active/Sleep	Deep Sleep	Deep Sleep-RAM	Deep Sleep-OFF	Hibernate	Off
Parameters						
Wake source[1)]	Any interrupt	DS peripherals	DS peripherals	DS peripherals	RTC/HIB peripherals	Power on
Wake action	Resume	Resume	Warm boot	Reset/cold boot	Reset	Reset
Wake time	One CPU cycle	<20 µs	Deep Sleep + warm boot	Deep Sleep + cold boot	POR + cold boot < 1 ms	
Resources						
ECO	On/Off	Off	Off	Off	Off	Off
IHO	On	Off	Off	Off	Off	Off
IMO	On	On/Off	Off	Off	Off	Off
ILO	On/Off	On/Off	On/Off	On/Off	On/Off	Off
WCO	On/Off	On/Off	On/Off	On/Off	On/Off	Off
CPU	On/Sleep	Retention	Off	Off	Off	Off
SRAM	On	On	On/Off	Off	Off	Off

1) See Table 135 for the list of peripherals available in DS (Deep Sleep) and HIB (Hibernate) power modes.

The operational states available in low-power modes are generally limited in functionality and parametric performance compared to their capabilities in the Active modes. Additionally, blocks that do not support low-power modes, such as Deep Sleep and Hibernate, cannot wake up the CPUs from these modes.

Table 135 outlines the availability of various device components during the device power modes/states.

Table 135 Resources available in different power modes

Block	Power mode				
	Active	Sleep	Deep Sleep	Hibernate	Backup
CPUSS					
CPU	Y	N	N	N	N
NVIC	Y	Y	N	N	N
WIC	Y	Y	Y	Y	N
FLASH	Y	Y	N	N	N
SRAM	Y	Y	Y	N	N
DMA	Y	Y	N	N	N
Programmable digital					

(table continues...)

Table 135 (continued) Resources available in different power modes

Block	Power mode				
	Active	Sleep	Deep Sleep	Hibernate	Backup
SMART I/O	Y	Y	Y	N	N
Fixed function digital					
TCPWM	Y	Y	N	N	N
SCB	Y	Y	Y[1)]	N	N
CAN FD	Y	Y	N	N	N
Special function					
CORDIC	Y	N	N	N	N
Analog					
HPPASS (SAR, CSG)	Y	Y	N	N	N
LPComp	Y	Y	Y	Y	N
I/O					
GPIO	Y	Y	Y	Y[2)]	N
Backup					
RTC	Y	Y	Y	Y	Y
Registers	Y	Y	Y	Y	Y

1) Only SCB 0 (I2C, SPI)
2) Only hibernate_wakeup pins (P2.0 and P9.0) are operational and capable of waking up the device from Hibernate mode.

18.5 Registers summary

Table 136 Register list

Name	Description
PWR_CTL	Controls the device's power mode options and enables observation of the current state.
PWR_CTL2	
PWR_HIBERNATE	Controls various Hibernate mode entry/exit related options.
PWR_HIB_WAKE_CTL	The Hibernate Wake-up Mask register sets the hibernate assignment wake-up and triggers recording.
PWR_HIB_WAKE_CTL2	The Hibernate Wake-up Polarity register selects the polarity for the corresponding hibernate wake-up sources.
PWR_HIB_WAKE_CAUSE	The Hibernate Wake-up Cause register records the hibernate unmasked wake-up causes and retains them during the Hibernate mode.

19 Clocking system

The PSC3M/P6 MCU family provides flexible clocking options with on-chip oscillators, phase-lock loop (PLL), and supports multiple external clock sources.

19.1 Features

The PSC3M/P6 MCU clock system includes the following resources:

- Three internal clock sources:
  - 8-MHz internal main oscillator (IMO)
  - 48-MHz internal high-speed oscillator (IHO)
  - 32-kHz internal low-speed oscillator (ILO)
- Three external clock sources:
  - Up to 80-MHz external clock (EXT_CLK) input from an I/O pin
  - 4- to 36-MHz external crystal oscillator (ECO)
  - 32.768-kHz external watch crystal oscillator (WCO)
- One frequency lock loop (FLL) with 24- to 100-MHz output range
- One digital phase-locked loop (DPLL0) with 25- to 240-MHz output range
- Each clock source has a clock supervisor (CSV)

19.2 Architecture

Figure 64 shows the generic view of the clocking system in the PSC3M/P6 MCU family.

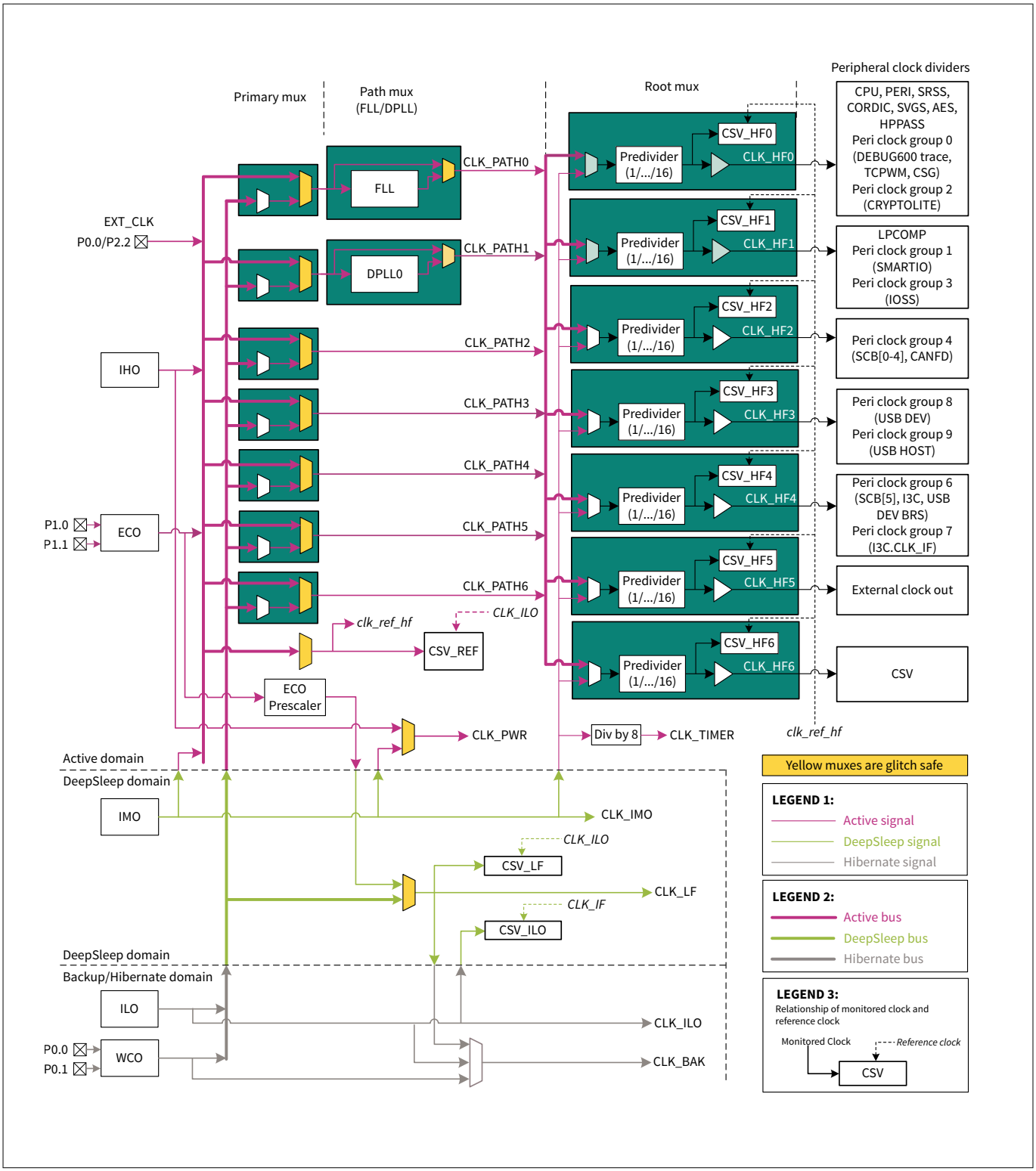


Figure 64 Clocking system block diagram

***Note**: See Peripheral clock dividers section for more details about the peripheral clock dividers.*

19.3 Clock sources

19.3.1 Internal main oscillator (IMO)

The IMO is a high-speed internal (crystal-less) main oscillator with fast wake-up and low jitter that produces a fixed frequency of 8-MHz, and has a tolerance of ±2%. The DPLL or FLL use the IMO to generate a wide range of higher frequency clocks or the high-frequency root clocks use the IMO directly. The IMO is available in Active, Sleep, and DeepSleep modes.

When USB is present the USB Start-of-Frame (SOF) signal is used to trim the IMO to ensure that the IMO matches the accuracy of the USB SOF. The ENABLE_LOCK bitfield in the USBDEV_CR1 register of the USB block needs to be set for this feature to work.

19.3.2 Internal high-frequency oscillator (IHO)

The IHO is an accurate, high-speed internal (crystal-less) oscillator that produces a fixed frequency 48 MHz without any external components, and has a tolerance of ±1%. The IHO is the default clock source after power-on-reset (POR) or any other reset. A high-speed clock can be derived using the IHO plus a DPLL. The IHO is only available in Active and Sleep modes.

19.3.3 Internal low-speed oscillator (ILO)

The ILO is low-speed internal oscillator that operates with no external components, and outputs a stable clock at 32-kHz nominal frequency. The ILO is relatively low power and low accuracy. It is available in all power modes. If the ILO is to remain active in Hibernate mode and across power-on-reset (POR) or brown-out detection (BOD), the ILO_BACKUP bit must be set in the CLK_ILO_CONFIG register.

The ILO can be used as the clock source for CLK_LF as well as a source for the backup domain (CLK_BAK). CLK_BAK runs the real-time clock (RTC). Although the ILO is not suitable as an RTC due to its poor accuracy, it can be used as a HIBERNATE wake-up source using the wake-up alarm facility in the RTC. In this case, the VDDD rail must be supplied during hibernate for the ILO to run, and the ILO_BACKUP bit must be set in the CLK_ILO_CONFIG register.

The ILO can be used for the source of the watchdog timer (WDT). The ILO is enabled and disabled with the ENABLE bit of the CLK_ILO_CONFIG register. Leave the ILO enabled if it is the source of the WDT. If the WDT is enabled, the only way to disable the ILO is to first clear the WDT_LOCK bit in the WDT_CTL register and then clear the ENABLE bit in the CLK_ILO_CONFIG register. If the WDT_LOCK bit is set, any register write to disable the ILO is ignored. Enabling the WDT automatically enables the ILO.

The calibration counters described in the Clock calibration counters chapter can be used to measure the ILO against a high-accuracy clock such as the ECO. This result can then be used to determine how the ILO must be adjusted. The ILO can be trimmed using the CLK_TRIM_ILO_CTL register.

19.3.4 External clock (EXT_CLK)

The external clock is a 0.001 to 80-MHz range clock that can be sourced from a signal on a designated I/O pin. This clock can be used as the source clock for either the DPLL or FLL, or can be used directly by the high-frequency clocks.

When manually configuring a pin as an input to EXT_CLK, set the drive mode of the pin to high-impedance digital to enable the digital input buffer. See the I/O subsystem chapter for more details. Consult the device datasheet to determine the specific pin used for EXT_CLK.

19.3.5 External crystal oscillator (ECO)

The PSC3M/P6 MCU contains an oscillator to drive an external 4-MHz to 36-MHz crystal. This clock source is built using an oscillator circuit. The circuit employs an external crystal that needs to be populated on the external crystal pins of the PSC3M/P6 MCU. The hardware automatically disables the ECO during Hibernate, DeepSleep, and XRES. ECO operates only in Active mode.

The ECO can be enabled by using the CLK_ECO_CONFIG.ECO_EN register bit-fields. After the ECO is enabled, check the CLK_ECO_STATUS register to ensure it is ready. The ECO clock can be a choice for CLK_HF and CLK_LF (prescaled).

19.3.6 Watch crystal oscillator (WCO)

The WCO is a highly accurate 32.768-kHz clock source. It is the primary clock source for the RTC. The WCO can also be used as a source for CLK_LF and CLK_BAK. The WCO can be enabled and disabled by setting the WCO_EN bit in the BACKUP_CTL register for the backup domain. The WCO can also be bypassed and an external 32.768-kHz clock can be routed on a WCO output pin. This is done by setting the WCO_BYPASS bit in the BACKUP_CTL register for the backup domain.

The WCO can also operate from external clock/sine wave inputs. In these modes, bypass the WCO by setting the WCO_BYPASS bit in the BACKUP_CTL register before enabling the WCO. Also, configure the GPIOs for the WCO_OUT and WCO_IN signals in analog mode.

19.3.7 ECO prescaler

The ECO prescaler divides the ECO, and creates a clock that can be used with a CLK_LF clock. This feature is available only during Active and Sleep modes. It cannot be used during DeepSleep or Hibernate modes.

The division function has a 10-bit integer divider and an 8-bit fractional divider. This function is configured using the CLK_ECO_PRESCALE register.

19.4 Clock generation

The PSC3M/P6 MCU has fully integrated clocks and provides clocks to all blocks that require clocks. It switches between different clock sources without glitches, and ensures that there are no metastable conditions.

19.4.1 Digital phase-locked loop (DPLL)

DPLL is a digital PLL for the PSC3M/P6 MCU that supports output clock frequencies of 25 to 250 MHz. The PLL uses a digital filter and an internal 250- to 500-MHz digital controlled oscillator (DCO) with an output duty cycle of 48–52%. It also contains an LDO to reject supply noise and improve jitter performance.

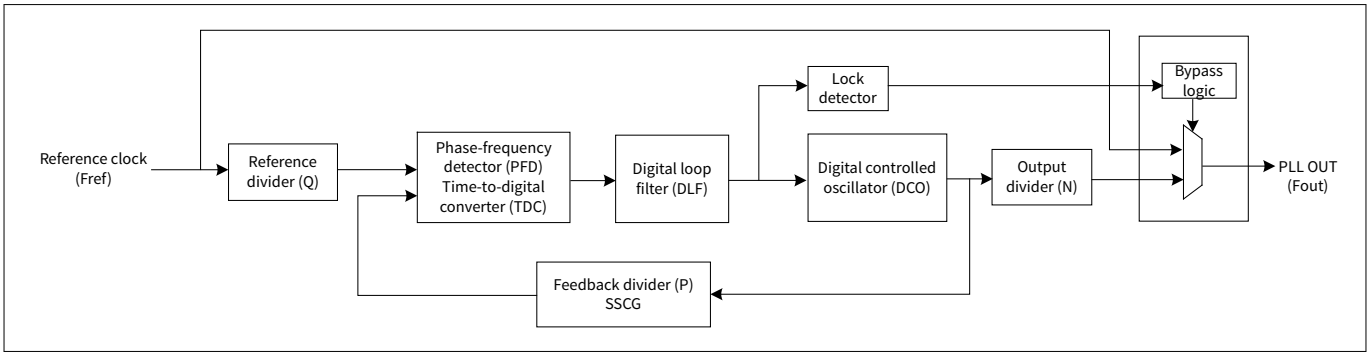


Figure 65 DPLL block diagram

Figure 65 shows the DPLL block diagram. The time to digital converter (TDC) block translates the difference between the reference clock and feedback clock into the code, which will be input to the digital loop filter (DLF). The DCO has an input of digital code and generates the output frequency accordingly.

The PLL provides multiple frequency-synthesis configurations, using programmable input and output integer dividers (not only binary) and a feedback divider (integer and fractional divider). The PLL has three programmable dividers:

- P (product): This divider is in the feedback path between the DCO output and the phase-frequency detector (PFD) input. It sets the multiplication factor of the DCO
- Q (quotient): This divider is between the input (reference) clock and the PFD. It divides the input clock before passing the same to the PFD
- N (output/post divider): This divider accepts the output clock of the DCO and divides it down to provide the desired frequency and output of the PLL

All dividers (P, Q, and N) are digitally programmable to divide their incoming clock by an integer number, while the feedback clock also supports division by a fractional number. The PLL generates a specific output frequency based on the setting of P, Q, and N dividers, and application of an appropriate reference clock, .

The PLL supports the following operation modes:

- **Integer operation mode**: The PLL generates integer modification of the input frequency by configuring P, Q, and N - it sets the PLL output frequency.

  PLL output frequency equation (Integer mode):

$$F_{out} = F_{ref} \times \left(P/Q/N\right) \tag{1}$$

  ***Note**: Fref is the input frequency of the PLL, that is, the frequency of the high-frequency clock source selected using the PATH_MUX (for example, 8 MHz from the IMO)*

- **Fractional operation mode**: The fractional frequency is defined by 24 input bits. The frequency resolution is $1/2^{24}$.

  PLL output frequency equation (Fractional mode):

$$F_{out} = \left(F_{ref}/Q\right) \times \left(P + Frac_div\right)/N \tag{2}$$

  ***Note**: The configuration of fractional operation uses the CLK_DPLL_LPx_CONFIG2 register. Frac_div is the value set by CLK_DPLL_LPx_CONFIG2.FRAC_DIV divided by 2^{24} . Frac_div = CLK_DPLL_LPx_CONFIG2.FRAC_DIV / 2^{24}.*

  For example: Non-standard frequency 40.456 MHz generated from Fref = 8 MHz

  P = 40, Q = 1, N = 8, Frac_div =7650410

  Fout = (8 * (40 + 7650410/16777216)) / 8 = 40.456 MHz

Follow these steps to configure the PLL:

1. Determine the desired reference clock frequency (Fref) and the desired output frequency (Fout). Calculate the reference (Q), feedback (P), and output (N) dividers subject to the following constraints:
   - **a.** PFD frequency (frequency constraints: 4 - 8 MHz). Fpfd = Fref/Q. There may be multiple reference divider values that meet this constraint
   - **b.** DCO frequency (frequency constraints: 250 - 500 MHz). DCO = Fpfd × P. There may be multiple feedback divider values that meet this constraint with different REFERENCE_DIV choices

   c. Output frequency. PLL_OUT = DCO/N. It may not be possible to get the exact desired frequency due to granularity; therefore, consider the frequency error of the two closest choices
   d. Choose the best combination of divider parameters depending on the application
2. Program the divider settings in the CLK_DPLL_LPx_CONFIG register. Do not enable the PLL during the same register write as configuring the dividers. Do not change the divider settings while the PLL is enabled
3. Configure the fractional divider in CLK_DPLL_LPx_CONFIG2 register and SSCG in CLK_DPLL_LPx_CONFIG3 register if needed
4. Enable the PLL (CLK_PLL_CONFIGx.ENABLE = 1). Wait at least 1 µs for PLL circuits to start
5. Wait until the PLL is locked before using the output. By default, the PLL output is bypassed to its reference clock and will automatically switch to the PLL output when it is locked. This behavior can be changed using CLK_DPLL_LPx_CONFIG.BYPASS_SEL. The status of the PLL can be checked by reading CLK_DPLL_LPx_STATUS. This register contains a bit indicating the PLL has locked. It also contains a bit indicating if the PLL lost the lock status

Spread spectrum clock generation (SSCG)

Spread spectrum clock generation (SSCG) is a method by which the energy contained in the narrow band of a clock source is spread over a wider band in a controlled manner, so reducing the peak spectral amplitude of the fundamental and the harmonics to lower the radiated emission from the clock source. This is achieved by modulating the clock frequency with a waveform. The configuration of the SSCG uses the CLK_DPLL_LPx_CONFIG3 register.

SSCG has the following parameters:

- **Modulation Rate**

  Modulation rate (MR) is the rate (in Hz) at which the energy of the clock source is distributed over the band of frequencies around the output clock frequency. Modulation rate must be much lower than the source clock frequency, and must be above the audio frequency range
- **Modulation Depth**

  Modulation depth (also known as deviation) is the frequency range over which the clock changes while varying at the modulation rate. It is specified as a percentage (%), which is the ratio of the bandwidth of frequency excursion (ΔF) to the source clock frequency. This determines the amount of peak EMI reduction achievable. Generally, the larger the modulation depth, the greater the EMI reduction. However, the system constraints on the frequency variation must be evaluated while selecting the Modulation Depth
- **Modulation Type**

  Modulation type (or spreading mode) specifies the relationship of the frequency deviation of the modulated clock relative to the non-modulated clock. For general SSCG, center, up, and down spread are available, but this PLL only supports down spread

  Down spreading is where the maximum frequency of the spread spectrum clock is the same as that of the non-modulated clock. In a down-spread system, the output clock varies between (Fc − ΔF) and Fc at the modulation rate and following the modulation profile. It can be represented as "Fout = Fc − ΔF"

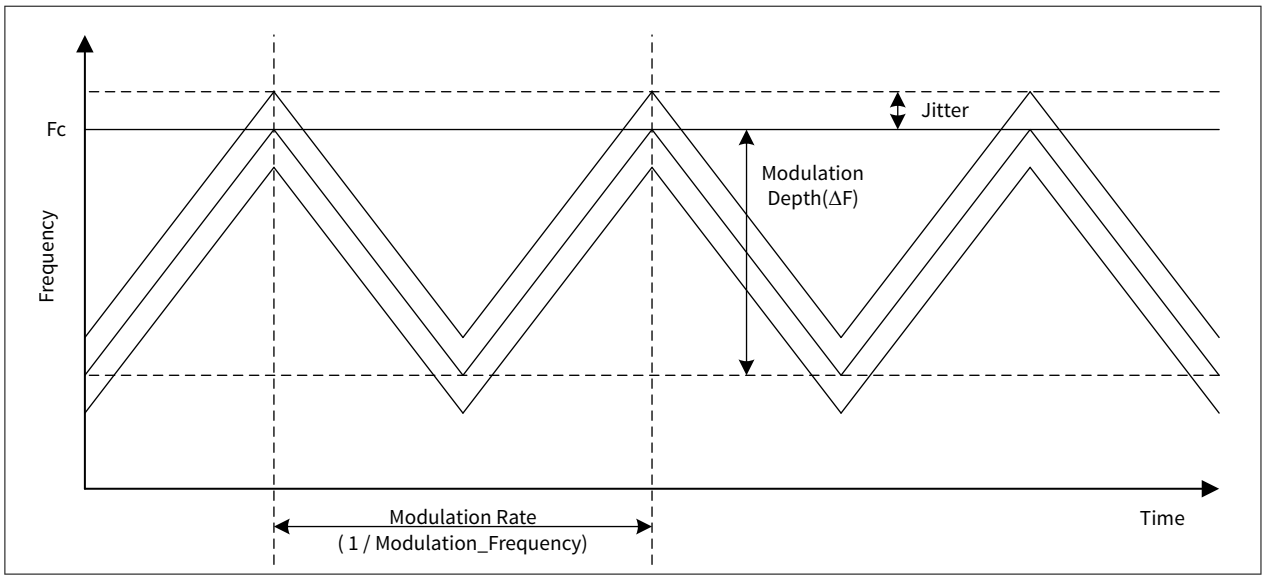


Figure 66 Down spreading example

Fout = the modulated output clock frequency
Fc = the source or carrier frequency
ΔF = total frequency deviation (min to max)

19.4.2 Frequency lock loop (FLL)

The PSC3M/P6 MCU contains one frequency lock loop (FLL), which resides on CLK_PATH0. The FLL is capable of generating a clock output in the range 24 MHz to 100 MHz. The input frequency must be between 0.008 to 100 MHz, and must be at least 2.5 times less than the current-controlled oscillator (CCO) frequency. This makes it possible to use the IMO to generate much higher clock frequencies for the rest of the system.

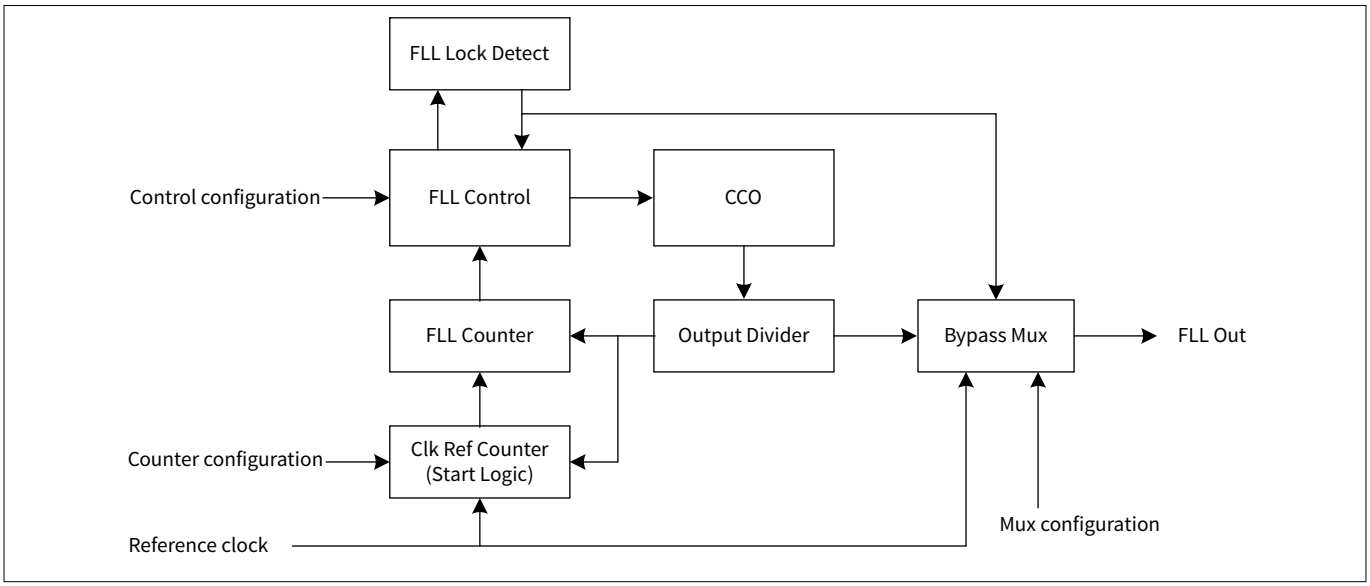


Figure 67 FLL block diagram

The FLL is similar in purpose to a PLL but is not equivalent; there are some differences:

- It consumes less current than the PLL

- FLL does not lock the phase. At the heart of the FLL is a current-controlled oscillator (CCO). The output frequency of this CCO is controlled by adjusting the trim of the CCO; this is done in hardware and is explained in detail later in this section
- FLL can produce up to 100-MHz clock with good duty cycle through its divided clock output
- FLL reference clock can be the WCO (32.768 kHz), IMO (8 MHz), or any other periodic clock source

The CCO can output a stable frequency in the 48 MHz to 200 MHz range. This range is divided into five sub-ranges as shown by Table 137.

Table 137 CCO frequency ranges

CCO rang	0	1	2	3	4
f_{min}	48 MHz	64 MHz	85 MHz	113 MHz	150 MHz
f_{max}	64 MHz	85 MHz	113 MHz	150 MHz	200 MHz

***Note**: The output of the CCO has an option to enable a divide by two or not. For this device, the divide by two must always be enabled.*

Within each range, the CCO output is controlled via a 9-bit trim field. This trim field is updated via hardware based on the control algorithm described below.

A reference clock must be provided to the FLL. This reference clock is typically the IMO, but can be many different clock sources. The FLL compares the reference clock against the CCO clock to determine how to adjust the CCO trim. Specifically, the FLL counts the number of CCO clock cycles inside a specified window of reference clock cycles. The number of reference clock cycles to count is set by the FLL_REF_DIV field in the CLK_FLL_CONFIG2 register.

After the CCO clocks are counted, they are compared against an ideal value and an error is calculated. The ideal value is programmed into the FLL_MULT field of the CLK_FLL_CONFIG register.

As an example, the reference clock is the IMO (8 MHz), the desired CCO frequency is 100 MHz, the value for FLL_REF_DIV is set to 146. This means that the FLL counts the number of CCO clocks within 146 clock periods of the reference clock. In one clock cycle of the reference clock (IMO), there should be 100/8 = 12.5 clock cycles of the CCO. Multiply this number by 146 and the value of FLL_MULT should be 1825.

If the FLL counts a value different from 1825, it attempts to adjust the CCO such that it achieves 1825 the next time it counts. This is done by scaling the error term with FLL_LF_IGAIN and FLL_LF_PGAIN found in CLK_FLL_CONFIG3. Figure 68 shows how the error (err) term is multiplied by FLL_LF_IGAIN and FLL_LF_PGAIN and then summed with the current trim to produce a new trim value for the CCO. The CCO_LIMIT field in the CLK_FLL_CONFIG4 can be used to put an upper limit on the trim adjustment; this is not needed for most situations.

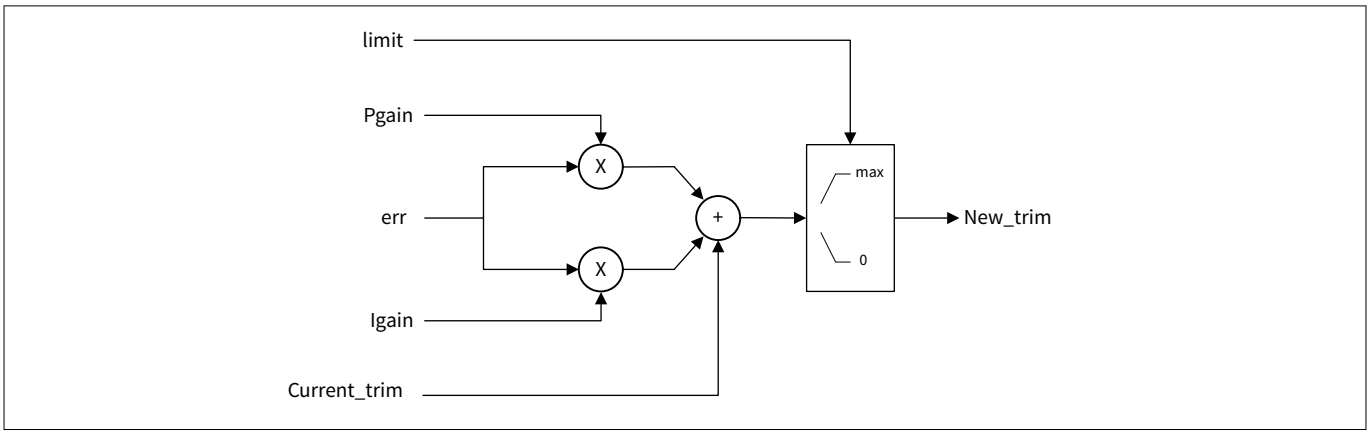


Figure 68 FLL error correction diagram

The FLL determines whether it is “locked” by comparing the error term with the LOCK_TOL field in the CLK_FLL_CONFIG2 register. When the error is less than LOCK_TOL, the FLL is considered locked.

After each adjustment to the trim, the FLL can be programmed to wait a certain number of reference clocks before doing a new measurement. The number of reference clocks to wait is set in the SETTLING_COUNT field of CLK_FLL_CONFIG3. Set this such that the FLL waits ~1 µs before a new count. Therefore, if the 8-MHz IMO is used as the reference, this field should be programmed to ‘8’.

When configuring the FLL, there are two important factors that must be considered: lock time and accuracy. Accuracy is the closeness to the intended output frequency. These two numbers are inversely related to each other via the value of REF_DIV. Higher REF_DIV values lead to higher accuracy, whereas lower REF_DIV values lead to faster lock times.

In the example used previously, the 8-MHz IMO was used as the reference, and the desired FLL output was 100 MHz. For that example, there are 12.5 CCO clocks in one reference clock. If the value for REF_DIV is set to ‘1’ then FLL_MULT must be set to either ‘13’ or ‘12’. This results in a CCO output of either 96 MHz or 104 MHz, and an error of 4 percent from the desired 100 MHz. Therefore, the best way to improve this is to increase REF_DIV. However, the larger REF_DIV is, the longer each measurement cycle takes, so increasing the lock time. In this example, REF_DIV was set to 146. This means each measurement cycle takes 146 * (1/8 MHz) = 18.25 µs, whereas when REF_DIV is set to 1, each measurement cycle takes 1 * (1/8 MHz) 0.125 µs.

Another issue with lower REF_DIV values is that the minimum LOCK_TOL is 1, so the output of the CCO can have an error of ±1. In the example where REF_DIV = 1 and FLL_MULT = 13, the MULT value can really be 12, 13, or 14 and still be locked. This means the output of the FLL may vary between 96 and 112 MHz, which may not be desirable.

Thus, a choice must be made between faster lock times and more accurate FLL outputs. The biggest change to make for this is the value of REF_DIV. The CLK_FLL_STATUS register checks the status of FLL.

Enabling and disabling the FLL

The FLL requires firmware sequencing when enabling, disabling, and entering/exiting DeepSleep.

Follow these steps to enable FLL:

1. Enable the CCO by writing CLK_FLL_CONFIG4.CCO_ENABLE = 1 and wait until CLK_FLL_STATUS.CCO_READY == 1
2. Ensure the reference clock has stabilized and CLK_FLL_CONFIG3.BYPASS_SEL = FLL_REF
3. Write FLL_ENABLE = 1 and wait until CLK_FLL_STATUS.LOCKED == 1
4. Write CLK_FLL_CONFIG3.BYPASS_SEL = FLL_OUT to switch to the FLL output

Follow these steps to disable FLL:

1. Ensure the processor is operating from a different clock than Clock Path 0. If the muxes are changed, wait four FLL output clock cycles for it to complete
2. Write CLK_FLL_CONFIG3.BYPASS_SEL = FLL_REF and read the same register to ensure the write completes
3. Wait at least six FLL reference clock cycles and disable it with FLL_ENABLE = 0
4. Disable the CCO by writing CLK_FLL_CONFIG4.CCO_ENABLE = 0

Before entering DeepSleep, either disable the FLL using the above sequence or use the following procedure to deselect/select it before/after DeepSleep. Before entering DeepSleep, write CLK_FLL_CONFIG3.BYPASS_SEL =FLL_REF to change the FLL to use its reference clock. After DEEPSLEEP wake-up, wait until CLK_FLL_STATUS.LOCKED==1 and then write CLK_FLL_CONFIG3.BYPASS_SEL = FLL_OUT to switch to the FLL output.

***Note**: It is not recommended to use the FLL_AUTO option in the BYPASS_SEL field.*

19.5 Clock trees

The PSC3M/P6 MCU clocks are distributed throughout the device, as shown in Figure 64. The clock trees are as follows:

- Path clocks
- High-frequency root clocks (CLK_HF)
- Low-frequency clock (CLK_LF)
- Backup clock (CLK_BAK)
- Timer clock (CLK_TIMER)
- Power clock (CLK_PWR)

19.5.1 Clock path

The PSC3M/P6 MCU has seven clock paths: CLK_PATH0 contains the FLL, CLK_PATH1 contains DPLL0, CLK_PATH2, CLK_PATH3, CLK_PATH4, and CLK_PATH5 and CLK_PATH6 are a direct connection to the high-frequency root clocks. Note that the FLL and DPLL can be bypassed if they are not needed. These paths are the input sources for the high-frequency clock roots.

Each clock path has a mux to determine which source clock will clock that path. This configuration is done in the CLK_PATH_SELECTx register.

Table 138 Clock path source selections

Name	Description
PATH_MUX[2:0]	Selects the source for CLK_PATHx 0: IMO - Internal main oscillator 1: EXT_CLK - External clock pin 2: ECO - External-crystal oscillator 3: Reserved. Do not use 4: DSI_MUX 5: Reserved. Do not use 6: IHO - Internal high-speed oscillator 7: Reserved. Do not use

***Note**: Switching away from the originally selected clock requires four cycles of it. Do not disable the original clock during this time.*

The DSI mux is configured through the CLK_DSI_SELECTx register. Table 139 shows the DSI mux source selections.

Table 139 DSI mux source selections

Name	Description
DSI_MUX[4:0]	Selects the source for the DSI_MUXx 0-15: Reserved. Do not use 16: ILO - Internal low-speed oscillator 17: WCO - Watch crystal oscillator 18-31: Reserved. Do not use

19.5.2 High-frequency root clocks (CLK_HF)

The PSC3M/P6 MCU has seven high-frequency root clocks (CLK_HF[0 - 6]). Each CLK_HF has a particular destination on the device, as shown in Table 140. See the Peripheral clock divider groups chapter for more details on the Peri clock group.

Table 140 CLK_HF destinations

Name	Description
CLK_HF[0]	CPU, PERI, AHB infrastructure, SRSS, Peri clock group 0 (DEBUG600 trace, TCPWM, HPPASS CSG), Peri clock group 2 (CRYPTOLITE), CORDIC, SVGS, AES, HPPASS
CLK_HF[1]	LPCOMP, Peri clock group 1 (SMART IO), Peri clock group 3 (IOSS)
CLK_HF[2]	Peri clock group 4 (CAN, SCB [0 - 4])
CLK_HF[3]	Peri clock group 8 (USB DEV), Peri clock group 9 (USB HOST)
CLK_HF[4]	Peri clock group 6 (SCB[5], I3C, USB), Peri clock group 7 (I3C.CLK_IF)
CLK_HF[5]	Clock output on clk_ext pin (when used as an output)
CLK_HF[6]	CSV

Each high-frequency root clock has a mux to determine its source and predivider. The root source can be IMO or ROOT_MUX by the DIRECT_MUX bit of CLK_DIRECT_SELECT[i] register selection show in Table 141.

Table 141 Clock root direct select register

Name	Description
DIRECT_MUX[8]	CLK_HF the direct select mux configuration: 0: IMO 1: ROOT_MUX, see CLK_ROOT_SELECT register

The ROOT_MUX and predivider configuration is done in the CLK_ROOT_SELECT[i] register shown in Table 142.

Table 142 Clock root select register

Name	Description
ROOT_MUX[3 : 0]	CLK_HF input clock path selection: 0: Select CLK_PATH0 1: Select CLK_PATH1 2: Select CLK_PATH2 3: Select CLK_PATH3 4: Select CLK_PATH4 5: Select CLK_PATH5 6: Select CLK_PATH6 [7 : 15]: Reserved

(table continues...)

Table 142 (continued) Clock root select register

Name	Description
ROOT_DIV[11 : 8]	Selects predivider value for the clock root: 0: No Divider 1: Divide selected clock source by 2 2: Divide selected clock source by 3 3: Divide selected clock source by 4 4: Divide selected clock source by 5 5: Divide selected clock source by 6 6: Divide selected clock source by 7 7: Divide selected clock source by 8 8: Divide selected clock source by 9 9: Divide selected clock source by 10 10: Divide selected clock source by 11 11: Divide selected clock source by 12 12: Divide selected clock source by 13 13: Divide selected clock source by 14 14: Divide selected clock source by 15 15: Divide selected clock source by 16
ENABLE	Enable for this clock root. All clock roots default to disabled (ENABLE==0) except HFCLK0, which cannot be disabled.

***Note**: CLK_HF[1-6] can be enabled and disabled. CLK_HF[0] is always enabled as it is the source of the CPU. To enable and disable CLK_HF[1-6] set the ENABLE bit in the CLK_ROOT_SELECT register.*

19.5.3 Low-frequency clock (CLK_LF)

The low-frequency clock (CLK_LF) in the PSC3M/P6 MCU has three input options: ILO, WCO, and ECO prescaler. CLK_LF is the source for the multi-counter watchdog timers (MCWDTs) and the RTC.

The source of CLK_LF is set in the LFCLK_SEL bits of the CLK_SELECT register. Writes to this field are ignored unless the WDT is unlocked using the WDT_LOCK register; see the Watchdog timer chapter for the details. It takes four cycles of the originally selected clock to switch away from it. Do not disable the original clock during this time.

Table 143 LFCLK input selection bits LFCLK_SEL

Name	Description
LFCLK_SEL[2:0]	LFCLK input clock selection: 0: ILO - Internal low-speed oscillator 1: WCO - Watch-crystal oscillator. Requires Backup domain to be present and properly configured (including external watch crystal, if used) 2 - 4: Reserved 5: ECO_PRESCALER - External-crystal oscillator after prescaling. Does not work in DEEPSLEEP or HIBERNATE modes. Intended for applications that operate in ACTIVE/SLEEP modes only. This option is only valid when ECO is present in the product 6: Reserved

19.5.4 Backup clock (CLK_BAK)

The backup clock is used to clock the backup domain, specifically the RTC. For more information see Backup chapter.

19.5.5 Timer clock (CLK_TIMER)

The timer clock (CLK_TIMER) can be used as a clock source for the CPU SYSTICK timer. The timer clock is standardized as a 1-MHz (fixed) clock derived from IMO, which is always running in Active/Sleep modes. It turns off during other modes, such as DeepSleep. In the PSC3M/P6 MCU, there is no pre-divider and it only provides a fixed 1-MHz frequency, which is also a divided-by-8 clock from CLK_IMO. The SRSS constraints automatically create the relevant internal clocks.

19.5.6 Power clock (CLK_PWR)

The power clock (CLK_PWR) is used to clock the Arm® power architecture components, Power Police Unit (PPU) and Power Dependency Control Matrix (PDCM). The CLK_PWR is always running in Active/Sleep modes. It turns off during other modes, such as DeepSleep mode.

The source of power clock can be the IMO or IHO; it is set in the CLK_PWR_MUX bits of the PWRMODE_CLK_SELECT register. The divide value is set by the CLK_PWR_DIV bits of this register.

Table 144 Power mode clock select register

Name	Description
CLK_PWR_DIV[7 : 0]	CLK_PWR is generated by dividing the CLK_PWR_MUX selection by (CLK_PWR_DIV+1)
CLK_PWR_MUX[17 : 16]	Selects the source for the power clock:: 0: IMO 1: IHO 2: Reserved

19.6 CLK_HF distribution

The PSC3M/P6 MCU has several CLK_HFs, which connect the CPU subsystem, SRSS, and peripheral functions. Also, a part of CLK_HF connects to CSV. See the High-frequency root clocks (CLK_HF) chapter for more details. Some peripherals connect to CLK_HF by the peripheral clock divider (see the Peripheral clock divider groups chapter for the details).

The following table lists the details of the CLK_HF distribution network, and the CPU and peripheral maximum frequency under the different active modes. The PSC3M/P6 MCU supports four Active modes: Ultra Low Power (ULP) mode (0.9 V), Medium Frequency (MF) mode (1.0 V), Low Power (LP) mode (1.1 V), and Over Driver (OD) mode (1.2 V). See the Device power modes chapter for more the details of power modes.

Table 145 Clock distribution and Active mode performance parameters

CLK_HF	Peripherals	Source	Max permitted clock frequency [1]			
			OD (1.2 V)	LP (1.1 V)	MF (1.0 V)	ULP (0.9 V)
CLK_HF0	CPU, PERI, SRSS, CORDIC, DEBUG600, SVGS, AES, HPPASS[2], Peri clock group 0 (TCPWM, CSG), Peri clock group 2 (CRYPTOLITE)	IMO, IHO, DPPL0, FLL	180 MHz	150 MHz	70 MHz	50 MHz
CLK_HF1	LPCOMP, Peri clock group 1 (SMARTIO), Peri clock group 3 (IOSS)	IHO, DPLL0, FLL	180 MHz	150 MHz	70 MHz	50 MHz
CLK_HF2	Peri clock group 4 (CAN, SCB[0:4])	IHO, FLL	100 MHz	100 MHz	25 MHz	25 MHz
CLK_HF3	Peri clock group 8 (USB DEV), Peri clock group 9 (USB HOST)	ECO, FLL	50 MHz	50 MHz	50 MHz	50 MHz
CLK_HF4	Peri clock group 6 (SCB[5], I3C, USB), Peri clock group 7 (I3C.CLK_IF)	IHO, DPLL0	180 MHz	150 MHz	70 MHz	50 MHz
CLK_HF5	EXT_CLK	IHO, DPLL0	80 MHz	80 MHz	80 MHz	50 MHz
CLK_HF6	CSV	IMO	8 MHz	8 MHz	8 MHz	8 MHz

1) Maximum clock frequency after the corresponding clock source (DPLL/FLL + dividers). All internal tolerances and affects are covered by these frequencies.
2) The minimum permitted clock frequency of HPPASS is 60 MHz.

19.7 Peripheral clock dividers

The PSC3M/P6 MCU peripherals, such as SCBs and TCPWMs, require a clock. These peripherals can be clocked only by a peripheral clock divider. The PSC3M/P6 MCU has 35 peripheral clock (PCLK) dividers. The output of any of these dividers can be routed to any peripheral within the group.

Clocks can be divided down to generate synchronous clocks for the analog and digital peripherals. The following integer and fractional clock dividers are provided:

- Eighteen 8-bit integer clock dividers
- Seven 16-bit integer clock dividers
- Eight 16.5 fractional clock dividers (16 integer bits, 5 fractional bits)
- Two 24.5 fractional clock divider (24 integer bits, 5 fractional bits)

19.7.1 Fractional clock dividers

Fractional clock dividers allow the clock divisor to include a fraction of 0..31/32. For example, a 16.5-bit divider with an integer divide value of 3 generates a 16-MHz clock from a 48-MHz CLK_PERI. A 16.5-bit divider with an integer divide value of 4 generates a 12-MHz clock from a 48-MHz CLK_PERI. A 16.5-bit divider with an integer divide value of 3 and a fractional divider of 16 generates a 48 / (3 + 16/32) = 48/3.5 = 13.7-MHz clock from a 48-MHz CLK_PERI. Not all 13.7-MHz clock periods are equal in size; some will have a 16-MHz period and others will have a 12-MHz period, such that the average is 13.7 MHz.

Fractional dividers are useful when a high-precision clock is required (for example, for a UART/SPI serial interface). Fractional dividers are not used when a low jitter clock is required, because the clock periods have a jitter of one CLK_PERI cycle.

19.7.2 Peripheral clock divider configuration

The peripheral clock dividers are configured using registers from the peripheral block; specifically DIV_CMD, DIV_8_CTL, DIV_16_CTL, DIV_16_5_CTL, DIV_24_5_CTL, and CLOCK_CTL registers.

First, configure the clock divider via the DIV_8_CTL, DIV_16_CTL, DIV_16_5_CTL, and DIV_24_5_CTL registers. There is one register for each divider; for example, there are eight DIV_8_CTL registers as there are eight 8-bit dividers. In these registers, set the value of the integer divider; if it is a fractional divider, then set the fraction portion as well.

After the divider is configured use the DIV_CMD register to enable the divider. This is done by setting the DIV_SEL to the divider number you want to enable, and setting the TYPE_SEL to the divider type. For example, if you want to enable the 0th 16.5-bit divider, write ‘0’ to DIV_SEL and ‘2’ to TYPE_SEL. If you want to enable the tenth 16-bit divider, write ‘10’ to DIV_SEL and ‘1’ to TYPE_SEL. See the registers reference manual for more details.

To connect a peripheral to a specific divider, the CLOCK_CTL register is used. There is a CLOCK_CTL register corresponding to each PCLK of clock divider group mentioned in the Peripheral clock divider groups chapter. For example, to select the second 8-bit divider for PCLK with group 5 and output 3, configure DIV_SEL to ‘2’ and TYPE_SEL to ‘1’ in the PERI_PCLK_GR5_CLOCK_CTL3 register. Also, the ‘Output’ order matches the order of y of the PERI_PCLK_GRx_CLOCK_CTLy register.

Phase-aligning dividers

For specific use cases, generate clocks that are phase-aligned. For example, consider the generation of two gated clocks at 24 and 12 MHz, both of which are derived from a 48-MHz CLK_PERI. If phase alignment is not considered, the generated gated clocks appear as follows.

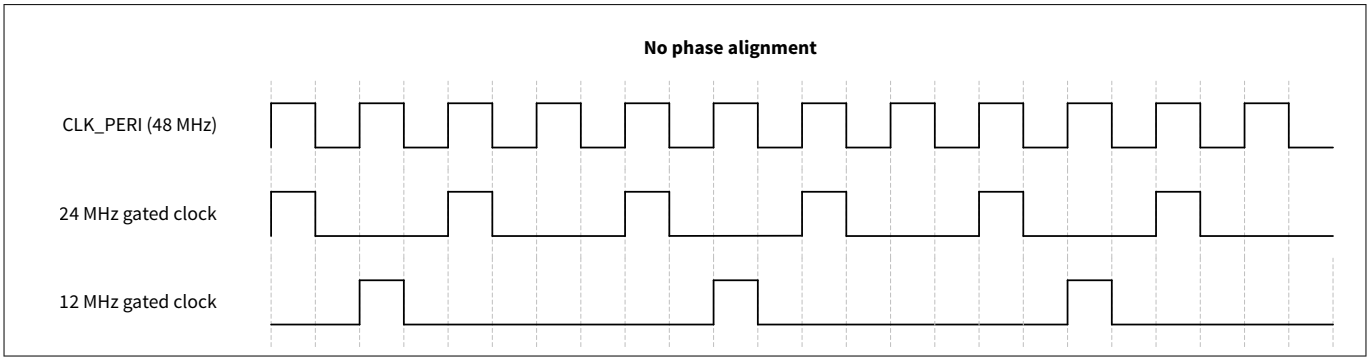


Figure 69 Non phase-aligned clock dividers

These clock signals may or may not be acceptable, depending on the logic functionality implemented on these two clocks. If the two clock domains communicate with each other, and the slower clock domain (12 MHz) assumes that each high/‘1’ pulse on it s clock coincides with a high/‘1’ phase pulse in the higher clock domain (24 MHz), the phase misalignment is not acceptable. To address this, it is possible to have dividers produce clock signals that are phase-aligned with an y of the other (enabled) clock dividers. Therefore, if (enabled) divider x is used to generate the 24-MHz clock, divider y can be phase-aligned to divider x and used to generate the 12-MHz clock. The aligned clocks appear as follows.

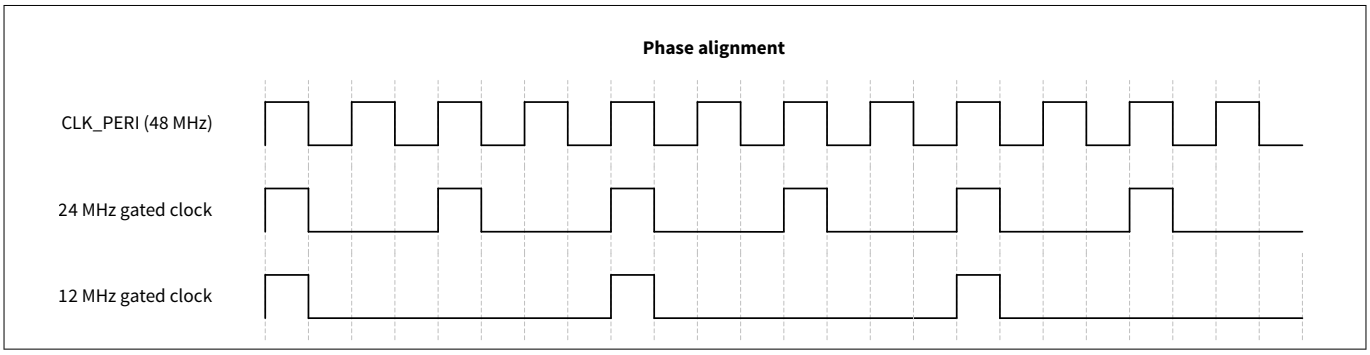


Figure 70 Phase-aligned clock dividers

Phase alignment also works for fractional divider values. If (enabled) divider x is used to generate the 38.4-MHz clock (divide by 1 8/32), divider y can be phase-aligned to divider x and used to generate the 19.2-MHz clock (divide by 2 16/32). The generated gated clocks appear as follows.

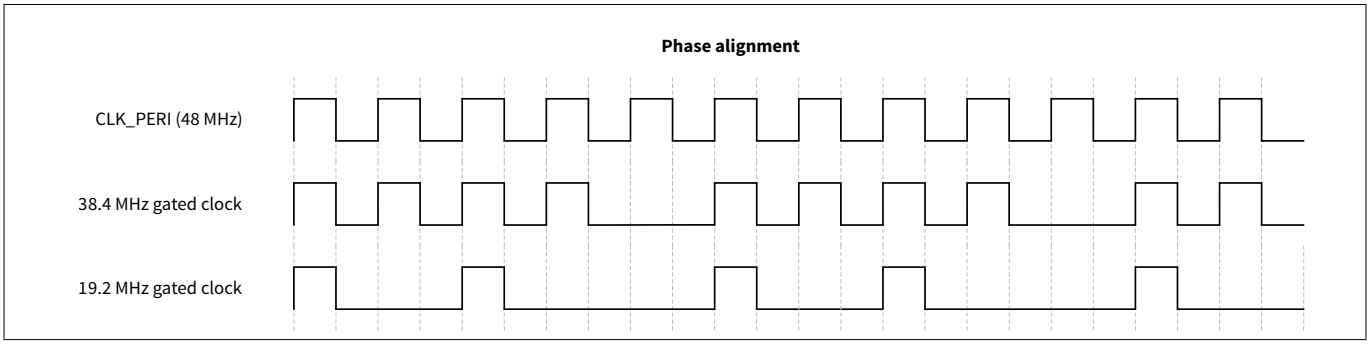


Figure 71 Phase-aligned fractional dividers

Divider phase alignment requires that the divider to which it is phase-aligned is already enabled. This requires the dividers to be enabled in a specific order.

Phase alignment is implemented by controlling the start moment of the divider counters in hardware. When a divider is enabled, the divider counters are set to '0'. The divider counters will only start incrementing from '0' to the programmed integer and fractional divider values when the divider to which it is phase-aligned has an integer counter value of '0'.

Note that the divider and clock multiplexer control register fields are all retained during the DeepSleep power mode. However, the divider counters that are used to implement the integer and fractional clock dividers are not. These counters are set to '0' during the DeepSleep mode. Therefore, when transitioning from DeepSleep to Active mode, all dividers (and clock signals) are enabled and phase-aligned by design.

Phase alignment is accomplished by setting the PA_DIV_SEL and PA_DIV _TYPE bits in the DIV_CMD register before enabling the clock. For example, to align the fourth 8-bit divider to the third 16-bit divider, set DIV_SEL to '4', TYPE_SEL to '0', PA_DIV_SEL to '3', and PA_TYPE_SEL to '1'.

19.7.3 Peripheral clock divider groups

The PSC3M/P6 MCU has nine PCLK clock divider groups, the outputs of each group are connected to a specific peripheral. For a complete list of peripherals, refer to the following clock groups table.

Peri clock group 0

- Root clock: CLK_HF[0]
- Divider count: 21
- Output count: 1
- 8 bits integer divider count: 12
- 16 bits integer divider count: 6
- 16.5 bits fractional divider count: 2
- 24.5 bits fractional divider count: 1

Table 146 Peri clock group 0: divided clock outputs

Output Number	Destination
0	Debug trace clock (debug600.clock_trace_in)
1	TCPWM group 0 32-bit counter 0 (tcpwm[0].clock_counter_en[0])
2	TCPWM group 0 32-bit counter 1 (tcpwm[0].clock_counter_en[1])
3	TCPWM group 0 32-bit counter 2 (tcpwm[0].clock_counter_en[2])
4	TCPWM group 0 32-bit counter 3 (tcpwm[0].clock_counter_en[3])
5	TCPWM group 0 32-bit counter 4 (tcpwm[0].clock_counter_en[4])
6	TCPWM group 0 32-bit counter 5 (tcpwm[0].clock_counter_en[5])
7	TCPWM group 0 32-bit counter 6 (tcpwm[0].clock_counter_en[6])
8	TCPWM group 0 32-bit counter 7 (tcpwm[0].clock_counter_en[7])
9	TCPWM group 1 16-bit counter 0 (tcpwm[0].clock_counter_en[256])
10	TCPWM group 1 16-bit counter 1 (tcpwm[0].clock_counter_en[257])
11	TCPWM group 1 16-bit counter 2 (tcpwm[0].clock_counter_en[258])
12	TCPWM group 1 16-bit counter 3 (tcpwm[0].clock_counter_en[259])
13	TCPWM group 1 16-bit counter 4 (tcpwm[0].clock_counter_en[260])
14	TCPWM group 1 16-bit counter 5 (tcpwm[0].clock_counter_en[261])
15	TCPWM group 1 16-bit counter 6 (tcpwm[0].clock_counter_en[262])

(table continues...)

Table 146 (continued) Peri clock group 0: divided clock outputs

Output Number	Destination
16	TCPWM group 1 16-bit counter 7 (tcpwm[0].clock_counter_en[263])
17	TCPWM group 2 16-bit counter 0 (tcpwm[0].clock_counter_en[512])
18	TCPWM group 2 16-bit counter 1 (tcpwm[0].clock_counter_en[513])
19	TCPWM group 2 16-bit counter 2 (tcpwm[0].clock_counter_en[514])
20	TCPWM group 2 16-bit counter 3 (tcpwm[0].clock_counter_en[515])
21	TCPWM group 2 16-bit counter 4 (tcpwm[0].clock_counter_en[516])
22	TCPWM group 2 16-bit counter 5 (tcpwm[0].clock_counter_en[517])
23	TCPWM group 2 16-bit counter 6 (tcpwm[0].clock_counter_en[518])
24	TCPWM group 2 16-bit counter 7 (tcpwm[0].clock_counter_en[519])
25	TCPWM group 1 MOTIF 0 (tcpwm[0].clock_motif_en[32])
26	TCPWM group2 MOTIF 0 (tcpwm[0].clock_motif_en[64])
27	HPPASS CSG clock (pass.clock_csg)

Peri clock group 1

- Root clock: HF[1]
- Divider count: 1
- Output count: 7
- 8-bit integer divider count: 1
- 16-bit integer divider count: 0
- 16.5-bit fractional divider count: 0
- 24.5-bit fractional divider count: 0

Table 147 Peri clock group 1: divided clock outputs

Output number	Destination
0	SMARTIO (ioss.clock_smartio_pclk_pos_en[0])
1	SMARTIO (ioss.clock_smartio_pclk_pos_en[1])
2	SMARTIO (ioss.clock_smartio_pclk_pos_en[2])
3	SMARTIO (ioss.clock_smartio_pclk_pos_en[3])
4	SMARTIO (ioss.clock_smartio_pclk_pos_en[5])
5	SMARTIO (ioss.clock_smartio_pclk_pos_en[6])
6	SMARTIO (ioss.clock_smartio_pclk_pos_en[9])

Peri clock group 2

- Root clock: HF[0]
- Divider count: 0
- Output count: 1
- 8-bit integer divider count: 0
- 16-bit integer divider count: 0
- 16.5-bit fractional divider count: 0
- 24.5-bit fractional divider count: 0

Table 148 Peri clock group 2: divided clock outputs

Output number	Destination
0	CRYPTOLITE (cryptolite.clk_hf)

***Note**: CRYPTOLITE peripheral directly connect to CLK_HF[0].*

Peri clock group 3

- Root clock: HF[1]
- Divider count: 0
- Output count: 1
- 8-bit integer divider count: 0
- 16-bit integer divider count: 0
- 16.5-bit fractional divider count: 0
- 24.5-bit fractional divider count: 0

Table 149 Peri clock group 3: divided clock outputs

Output number	Destination
0	IOSS (ioss.clk_hf)

***Note**: The I/O subsystem directly connects to CLK_HF[1].*

Peri clock group 4

- Root clock: HF[2]
- Divider count: 9
- Output count: 12
- 8-bit integer divider count: 4
- 16-bit integer divider count: 1
- 16.5-bit fractional divider count: 3
- 24.5-bit fractional divider count: 1

Table 150 Peri clock group 4: divided clock outputs

Output number	Destination
0	SCB0 (scb[0].clock_scb_en])
1	SCB0 (scb[0].clock_tgs_en)
2	SCB1 (scb[1].clock_scb_en])
3	SCB1 (scb[1].clock_tgs_en)
4	SCB2 (scb[2].clock_scb_en])
5	SCB2 (scb[2].clock_tgs_en)
6	SCB3 (scb[3].clock_scb_en])
7	SCB3 (scb[3].clock_tgs_en)
8	SCB4 (scb[4].clock_scb_en])
9	SCB4 (scb[4].clock_tgs_en)

(table continues...)

Table 150 (continued) Peri clock group 4: divided clock outputs

Output number	Destination
10	CAN 0 channel 0 (canfd[0].clock_can_en[0])
11	CAN 0 channel 1 (canfd[0].clock_can_en[1])

Peri clock group 6

- Root clock: HF[4]
- Divider count: 4
- Output count: 4
- 8-bit integer divider count: 1
- 16-bit integer divider count: 0
- 16.5-bit fractional divider count: 3
- 24.5-bit fractional divider count: 0

Table 151 Peri clock group 6: divided clock outputs

Output number	Destination
0	SCB5 (scb[5].clock_scb_en])
1	SCB5 (scb[5].clock_tgs_en)
2	I3C (i3c[0].clock_i3c_en)
3	USB (usb[0].clock_dev_brs)

Peri clock group 7

- Root clock: CLK_HF[4]
- Divider count: 0
- Output count: 1
- 8 bits integer divider count: 0
- 16 bits integer divider count: 0
- 16.5 bits fractional divider count: 0
- 24.5 bits fractional divider count: 0

Table 152 Peri clock group 7: divided clock outputs

Output Number	Destination
0	I3C (i3c[0].clk_if)

***Note**: The I3C CLK_IF directly connects to CLK_HF[4].*

Peri clock group 8

- Root clock: CLK_HF[3]
- Divider count: 0
- Output count: 1
- 8 bits integer divider count: 0
- 16 bits integer divider count: 0
- 16.5 bits fractional divider count: 0
- 24.5 bits fractional divider count: 0

Table 153 Peri clock group 8: divided clock outputs

Output Number	Destination
0	USB device (usb[0].clk_usb_dev)

***Note**: The USB device clock directly connects to CLK_HF[3].*

Peri clock group 9

- Root clock: CLK_HF[3]
- Divider count: 0
- Output count: 1
- 8 bits integer divider count: 0
- 16 bits integer divider count: 0
- 16.5 bits fractional divider count: 0
- 24.5 bits fractional divider count: 0

Table 154 Peri clock group 9: divided clock outputs

Output Number	Destination
0	USB host (usb[0].clk_usb_host)

***Note**: The USB host clock directly connects to CLK_HF[3].*

19.8 Clock calibration counters

The clocking system in the PSC3M/P6 MCU has built-in hardware calibration counters. Use these counters to compare the frequency of two clock sources against one another. The primary use case is to take a higher accuracy clock, such as the ECO, and use it to measure a lower accuracy clock such as the ILO. Use the result of this measurement to trim the ILO.

There are two counters:

1. Calibration counter 1 is clocked off of calibration clock 1 (generally the high-accuracy clock) and it counts down
2. Calibration counter 2 is clocked off of calibration clock 2 and it counts up

When calibration counter 1 reaches 0, calibration counter 2 stops counting up and its value can be read. From that value, determine the frequency of calibration clock 2 with the following equation:

$$Calibration\ clock\ 2\ frequency = \frac{Counter\ 2\ final\ value}{Counter\ 1\ final\ value} \times Calibration\ clock\ 1\ requency \quad (3)$$

For example, if calibration clock 1 = 8 MHz, Counter 1 = 1000, and Counter 2 = 5

$$Calibration\ clock\ 2\ frequency = \frac{5}{1000} \times 8\ MHz = 40\ KHz \quad (4)$$

Calibration clock 1 and calibration clock 2 are selected with the CLK_OUTPUT_FAST register. All clock sources are available as a source for these two clocks. CLK_OUTPUT_SLOW is also used to select the clock source.

Calibration counter 1 is programmed in CLK_CAL_CNT1 register. Calibration counter 2 can be read in CLK_CAL_CNT2 register.

When calibration counter 1 reaches 0, the CAL_COUNTER_DONE bit is set in the CLK_CAL_CNT1 register.

19.9 Clock supervision (CSV)

The clock supervision (CSV) circuit checks whether the frequency of the monitored clock is within the allowed frequency window. If the monitored clock stops, or fails to start, it is detected as low frequency.

The PSC3M/P6 MCU clocking system has CSV_HFs, CSV_REF, CSV_LF, and CSV_ILO to monitor all high-frequency root clocks (CLK_HFs), reference clock (CLK_REF_HF), low-frequency clock (CLK_LF), and internal low-speed oscillator(ILO) (see Figure 64 for the CSV connection). The CSV has the following features:

- All CLK_HFs have the CSV. All CLK_HF CSVs use the same reference clock (CLK_REF_HF). The reference clock is a selection of one of the Active clock sources (Register: CSV_REF_SEL). Typically, the IMO is selected (default)
- ILO is supervised both by CLK_LF and CLK_REF_HF; CLK_LF is needed for DeepSleep supervision and CLK_REF_HF is needed for accuracy while Active
- CSV_REF monitor CLK_REF_HF with CLK_ILO
- All CSV_HFs and CSV_REF are in the Active domain
- The DeepSleep domain has two CSVs (CSV_LF and CSV_ILO). CSV_LF is used to monitor the selected CLK_LF clock with ILO. CSV_ILO is used to monitor ILO with CLK_LF

CSV operation

The basic operation principle of the CSV circuit is as follows:

- The monitored clock generates a Monitor event (Period), and the reference clock generates a Lower and Upper limit
- The Monitor event is compared against a Lower or Upper limit
- An error is reported if a Monitor event ≤ Lower limit, or Monitor event > Upper limit

***Note**: Period is the monitored clock count, while Target is the reference clock count. Their time periods are the same in an ideal situation.*

The Figure 72 shows an example of the signal of CSV operation.

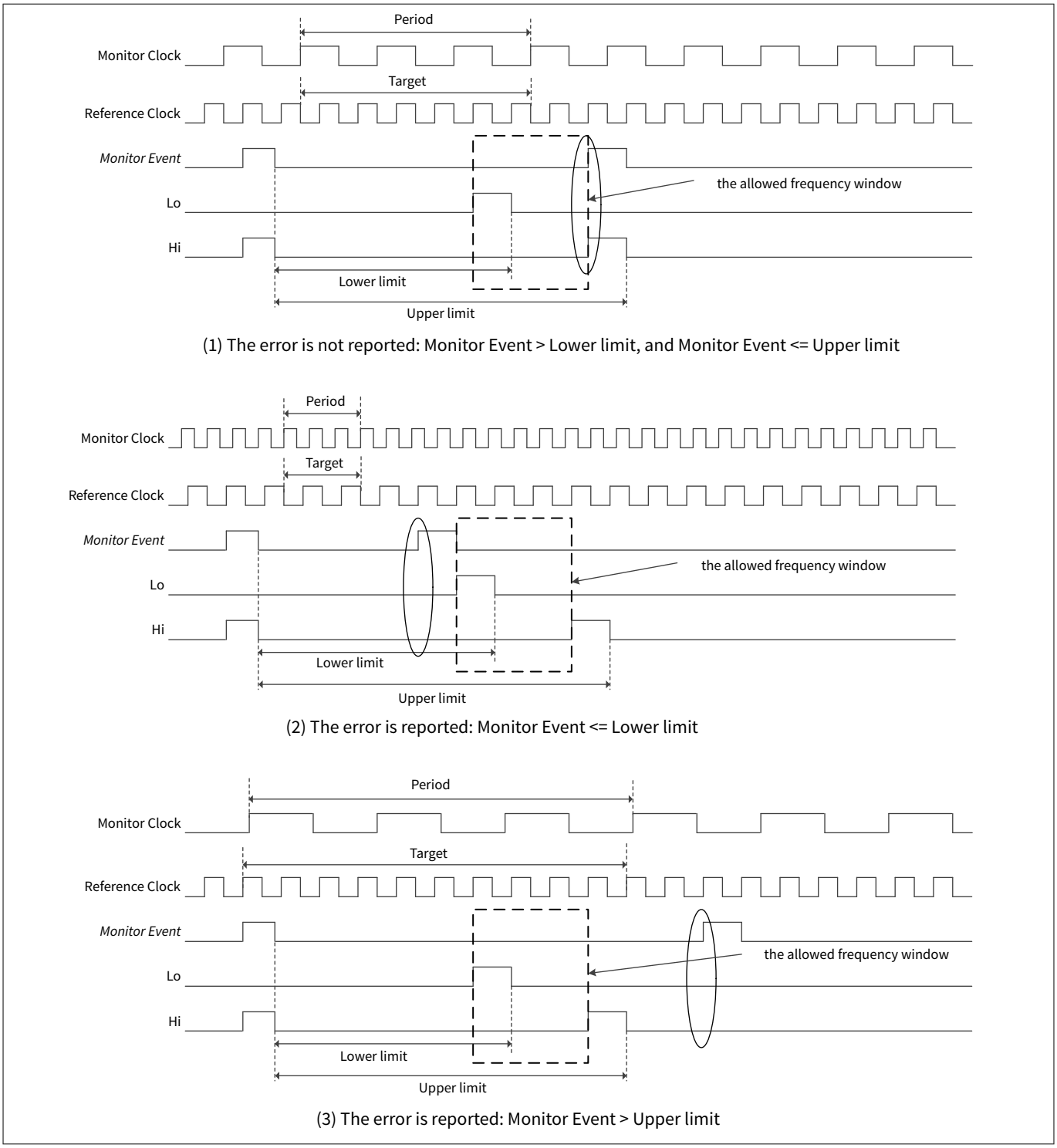


Figure 72 CSV operation signal example

The key points for CSV operation are as follows:

- Check the parameters:
  - Reference clock frequency and tolerance (integer%)
  - Monitor clock frequency and the required tolerance (integer%)

  The required monitor clock tolerance should be equal to or larger than the reference clock tolerance
- Determine “Target” with the following formula:
  Minimum target = 200/Reference_clock_tolerance

For example, for a tolerance of one percent, the target must be at least 200. Increasing the target increases CSV accuracy and latency

- Determine "Period" with the following formula:
  Period = Target / (Reference_clock_frequency/Monitor_clock_frequency)
- Determine Lower_limit and Upper_limit with the following formula:
  - Lower_limit = Period * ((Reference_clock_frequency * (1 – tolerance/100)) / (Monitor_clock_frequency * (1 + tolerance/100)))
  - Upper_limit = Period * ((Reference_clock_frequency * (1 + tolerance/100)) / (Monitor_clock_frequency * (1 – tolerance/100)))
- If the two clocks are asynchronous (typical), then there is a one-cycle variation of Monitor Event periods
- The frequency window needs to account for the maximum clock tolerance on both clocks
- Lower_limit must be at least one less than Upper_limit
- All CSVs are initially off and require configuration before enabling
- The Active domain CSVs are automatically stopped during DeepSleep mode
  - After wakeup, they automatically restart
  - Each CSV has a software programmable startup time. In the case of WCO, the CSV startup time is unused (STARTUP = 0). CSV should be enabled after WCO is started (BACKUP_STATUS.WCO_OK = 1)
- All Active domain CSVs can either generate a Reset or Fault report
  - The CLK_HF0 CSV must use reset because the fault structure runs on CLK_HF0
  - All other CSVs should use a fault report to allow software to shutdown
  - A fault report results in an interrupt
- The DeepSleep domain CSVs operate during Active and DeepSleep, and can only report faults (no reset option)
  - A CSV error detection wakes up the system (if needed), which enables fault reporting
  - The fault report results in an interrupt (no direct interrupt from CSV)
- All CSVs are enabled independent of the monitored clock, therefore:
  - Software should disable the CSV before stopping or reconfiguring the monitored clock (to avoid a false error detection)
  - The CSV needs to be reconfigured accordingly and re-started after the monitored clock is restarted
- The CSV_REF_SEL register selects a source to be used as the reference clock for CSV in the Active domain

  The registers to configure the CSV function are as follows:
  - These registers can enable CSV, and can configure an action when CSV is activated
    CSV_HF_CSVx_REF_CTL, CSV_REF_CSV_REF_CTL, CSV_LF_CSV_REF_CTL, CSV_ILO_CSV_REF_CTL
  - The following registers can configure upper limit and lower limit. Set the Lower_limit and Upper_limit as -1
    CSV_HF_CSVx_REF_LIMIT, CSV_REF_CSV_REF_LIMIT, CSV_LF_CSV_REF_LIMIT, CSV_ILO_CSV_REF_LIMIT
  - The following registers can configure PERIOD (time period). Set the Period as -1
    CSV_HF_CSVx_MON_CTL, CSV_REF_CSV_MON_CTL, CSV_LF_CSV_MON_CTL, CSV_ILO_CSV_MON_CTL
- When CSV violation occurs, SRSS_FAULT_CSV is indicated in the IDX bit of the FAULT_STRUCT_STATUS register. The CSV fault report is captured in FAULT_STRUCT_DATA register, see the Fault monitoring chapter for more details

19.10 Registers summary

Registers related to the clock system are show in Table 155, Table 156, Table 157, and Table 158.

Table 155 Clock system registers summary in the PERI

Register	Function
PERI_PCLK_GRx_DIV_CMD	Divider command register
PERI_PCLK_GRx_CLOCK_CTLy	Divider clock control register
PERI_PCLK_GRx_DIV_8_CTLy	8-bit divider control register
PERI_PCLK_GRx_DIV_16_CTLy	16-bit divider control register
PERI_PCLK_GRx_DIV_16_5_CTLy	16.5-bit divider control register
PERI_PCLK_GRx_DIV_24_5_CTLy	24.5-bit divider control register

Table 156 Clock system registers summary in the SRSS

Register	Function
CLK_DSI_SELECTx	Clock DSI select register
CLK_PATH_SELECTx	Clock path select register
CLK_OUTPUT_FAST	Fast clock output select register
CLK_OUTPUT_SLOW	Slow clock output select register
CLK_CAL_CNT1	Clock calibration counter 1 register
CLK_CAL_CNT2	Clock calibration counter 2 register
CLK_DIRECT_SELECTx	Clock root direct select register
CLK_ROOT_SELECTx	Clock root select register
CLK_SELECT	Clock select register
CLK_IMO_CONFIG	IMO configuration register
CLK_ECO_PRESCALE	ECO prescaler configuration register
CLK_ECO_CONFIG	ECO configuration register
CLK_ECO_CONFIG2	ECO configuration register 2
CLK_ECO_STATUS	ECO status register
CLK_FLL_CONFIG	FLL configuration register
CLK_FLL_CONFIG1	FLL configuration register 1
CLK_FLL_CONFIG2	FLL configuration register 2
CLK_FLL_CONFIG3	FLL configuration register 3
CLK_FLL_CONFIG4	FLL configuration register 4
CLK_FLL_STATUS	FLL status register
CLK_ILO_CONFIG	ILO configuration register
CLK_IHO_CONFIG	IHO configuration register
CLK_DPLL_LPx_CONFIG	DPLL configuration register
CLK_DPLL_LPx_CONFIG2	DPLL configuration register 2
CLK_DPLL_LPx_CONFIG3	DPLL configuration register 3

(table continues...)

Table 156 (continued) Clock system registers summary in the SRSS

Register	Function
CLK_DPLL_LPx_CONFIG4	DPLL configuration register 4
CLK_DPLL_LPx_CONFIG5	DPLL configuration register 5
CLK_DPLL_LPx_CONFIG6	DPLL configuration register 6
CLK_DPLL_LPx_CONFIG7	DPLL configuration register 7
CLK_DPLL_LPx_STATUS	DPLL status register
CSV_HF_CSVx_REF_CTL	Clock supervision reference control register for CLK_HFx
CSV_HF_CSVx_REF_LIMIT	Clock supervision reference limits register for CLK_HFx
CSV_HF_CSVx_MON_CTL	Clock supervision monitor control register for CLK_HFx
CSV_REF_SEL	Select CSV reference clock for Active domain
CSV_REF_CSV_REF_CTL	Clock supervision reference control register for CLK_REF_HF
CSV_REF_CSV_REF_LIMIT	Clock supervision reference limits register for CLK_REF_HF
CSV_REF_CSV_MON_CTL	Clock supervision monitor control register for CLK_REF_HF
CSV_LF_CSV_REF_CTL	Clock supervision reference control register for CLK_LF
CSV_LF_CSV_REF_LIMIT	Clock supervision reference limits register for CLK_LF
CSV_LF_CSV_MON_CTL	Clock supervision monitor control register for CLK_LF
CSV_ILO_CSV_REF_CTL	Clock supervision reference control register for CLK_ILO
CSV_ILO_CSV_REF_LIMIT	Clock supervision reference limits register for CLK_ILO
CSV_ILO_CSV_MON_CTL	Clock supervision monitor control register for CLK_ILO

Table 157 Clock system registers summary in the BACKUP

Register	Function
BACKUP_CTL	BACKUP control register

Table 158 Clock system registers summary in the PWRMODE

Register	Function
PWRMODE_CLK_SELECT	Clock selection register for power mode components

20 Watchdog timer

The watchdog timer (WDT) is a hardware timer that automatically resets the device in the event of an unexpected firmware execution path. If enabled, the WDT must be serviced periodically in the firmware to avoid a reset. Otherwise, the timer elapses and generates a device reset. In addition, the WDT can be used as an interrupt source or a wake-up source in low-power modes.

The PSC3M/P6 MCU family includes one free-running WDT and one multi-counter WDT (MCWDT). The watchdog system has four counters: the free-running WDT has one 32-bit counter, and the MCWDT has two 16-bit counters and one 32-bit counter. All 16-bit counters and the 32-bit counter can generate a watchdog device reset. All four counters can generate an interrupt on a match event.

20.1 Features

The PSC3M/P6 MCU WDT supports these features:

One 32-bit free-running WDT with:

- ILO as the input clock source by default. The clock source can be selected from ILO and CLK_BAK. For more details on clock sources, refer to the Clocking systemsection
- Device reset generation if not serviced within a configurable interval
- Periodic Interrupt/wakeup generation in Active, Sleep, DeepSleep, and Hibernate power modes

Multi-counter watchdog timer (MCWDT) with:

- Device reset generation, if not serviced within a configurable interval
- LFCLK (ILO , WCO) as the input clock source
- Periodic interrupt/wake up generation in Active, Sleep, and DeepSleep power modes
- Two 16-bit and one 32-bit independent counters, which can be configured as a single 64-bit or 48-bit (with one 16-bit independent counter), or two 32-bit cascaded counters
- Window mode operation supported for 16-bit counters

20.2 Architecture

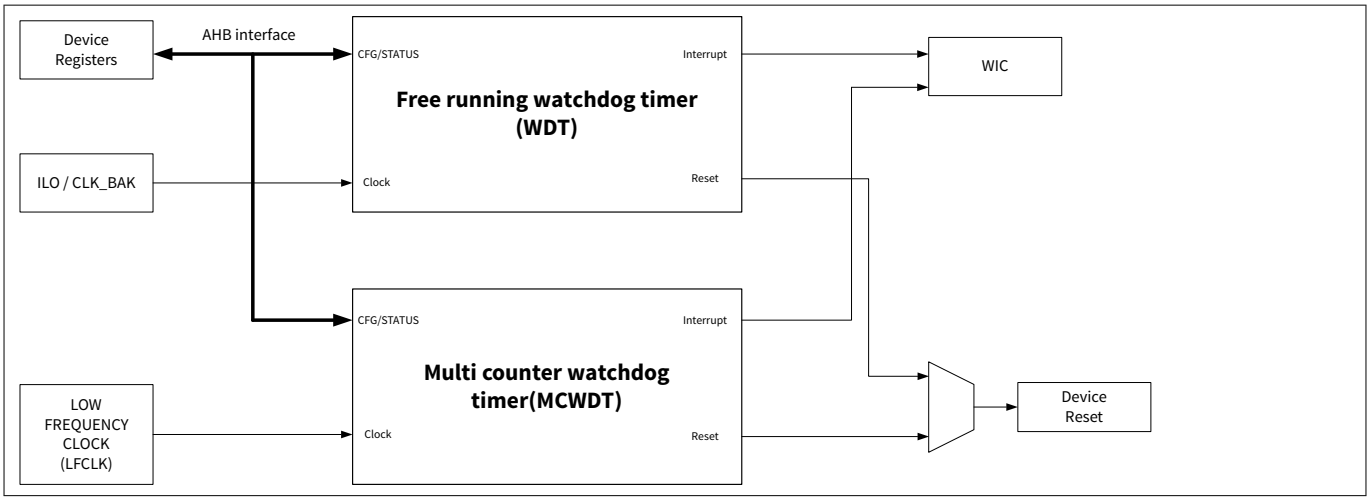


Figure 73 Watchdog timer block diagram

20.3 Basic watchdog timer

20.3.1 Overview

Figure 74 shows the functional overview of the free-running WDT. The WDT has a free-running wraparound up-counter with a maximum of 32-bit resolution. The counter is clocked by the ILO by default. The clock source can be changed using WDT_CTL.WDT_CLK_SEL bits to CLK_BAK. The timer can generate an interrupt on match and a reset event on the third unserviced interrupt. The number of bits used for a match comparison is configurable as depicted in Figure 74.

***Note**: BAK - Selected CLK_BAK source, if present. See BACKUP_CTL. This choice is not recommended for applications that rely upon the watchdog timer for safety or security. Generation of CLK_BAK is not protected by WDT_LOCK and is in a different memory region with potentially different security attributes.*

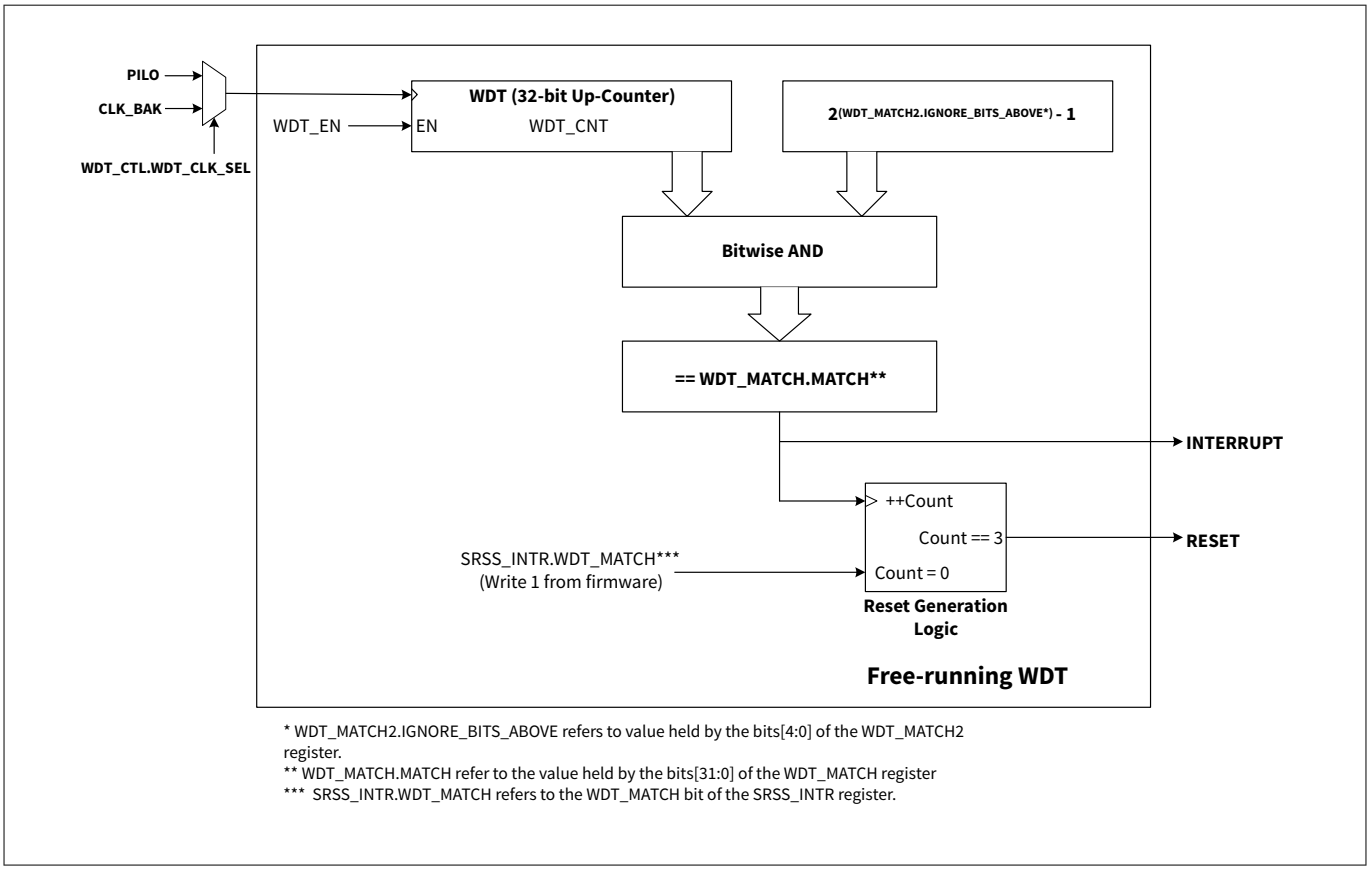


Figure 74 Free-running WDT functional diagram

When enabled, the WDT counts up on each rising edge of the ILO or CLK_BAK. When the counter value (WDT_CNT register) equals the match value stored in MATCH bits [31:0] of the WDT_MATCH register, an interrupt is generated. The match event does not reset the WDT counter and the WDT keeps counting until it reaches the 32-bit boundary (0xFFFF FFFF) at which point, it wraps around to 0 and counts up. The match interrupt is generated every time the counter value equals the match value.

The WDT_MATCH bit of the SRSS_INTR register is set whenever a WDT match interrupt occurs. This interrupt must be cleared by writing a '1' to the same bit. Clearing the interrupt resets the watchdog. If the firmware does not clear the interrupt for two consecutive occasions, the third interrupt generates a device reset.

In addition, the WDT provides an option to set the number of bits to be used for comparison. The IGNORE_BITS_ABOVE (bits [4:0] of the WDT_MATCH2 register) is used for this purpose. These bits configure the

number of LSBs to ignore from the 32-bit count value while performing the match. For example, when the value of these bits equals 4, bit 5 to bit 31 are ignored and bit 0 to bit 4 are considered for match and the WDT counter behaves similar to a 5-bit counter. Note that these bits do not reduce the counter size – the WDT_CNT register still counts from 0 to (0xFFFF FFFF) (32-bit).

The WDT can be enabled or disabled using the WDT_EN bit [0] of the WDT_CTL register. WDT_CTL[5:4] adds the ability to select the WDT clock source among the ILOs (default) and the CLK_BAK. The WDT_CTL register provides a mechanism to lock the WDT configuration registers. The WDT_LOCK bits [31:30] control the lock status of the WDT registers. These bits are special bits, which can enable the lock in a single write; to release the lock, two different writes are required. The WDT_LOCK bits protect the WDT_EN bit, WDT_MATCH register, CLK_ILO_CONFIG register, and LFCLK_SEL bits [2:0] of the CLK_SELECT register. Note that the WDT_LOCK bits are not retained in DeepSleep mode and Hibernate mode and reset to their default (LOCK) state after a wake-up. Therefore, to update any register protected by the WDT_LOCK bits after a DeepSleep wake-up or Hibernate wake-up, issue a WDT UNLOCK sequence before the register update.

Table 159 explains various registers and bitfields used to configure and use the WDT.

Table 159 Free-running WDT configuration options

Register [Bit_Pos]	Bit_Name	Description
WDT_CTL[0]	WDT_EN	Enable or disable the watchdog reset
		0 : WDT reset disabled
		1 : WDT reset enabled
WDT_CTL[5:4]	WDT_CLK_SEL	Select source for WDT
		0: ILO
		2 : CLK_BAK - Selected clk_bak source, if present. See BACKUP_CTL. This choice is not recommended for applications that rely on the watchdog timer for safety or security, Generation of clk_bak is not protected by WDT_LOCK and is in a different memory region with potentially different security attributes.
WDT_CTL[31:30]	WDT_LOCK	Lock or unlock write access to the watchdog configuration and clock-related registers. When the bits are set, the lock is enabled.
		0 : No effect
		1 : Clear bit 0
		2 : Clear bit 1
		3 : Set both bit 0 and 1 (lock enabled)

(table continues...)

Table 159 (continued) Free-running WDT configuration options

Register [Bit_Pos]	Bit_Name	Description
		WDT locks on a reset. Since this field is not retained in DeepSleep or Hibernate mode, the WDT is locked after wake-up from these modes.
WDT_CNT[31:0]	COUNTER	Current value of WDT counter
WDT_MATCH[31:0]	MATCH	Match value to generate watchdog match event
WDT_MATCH2[4:0]	IGNORE_BITS_ABOVE	Number of LSBs to consider for comparison with the MATCH value. This bit-field has a minimum setting of 3.
SRSS_INTR[0]	WDT_MATCH	WDT interrupt request This bit is set whenever a watchdog match event happens. The WDT interrupt is cleared by writing a '1' to this bit. Software can write 1 to this bit to feed the watchdog.
SRSS_INTR_MASK[0]	WDT_MATCH	Mask for the WDT interrupt
		0: WDT interrupt is blocked
		1: WDT interrupt is forwarded to CPU

20.3.2 Watchdog reset

A watchdog is typically used to protect the device against firmware/system crashes or faults. When the WDT is used to protect against system crashes, clear the WDT interrupt bit by a portion of the code that is not directly associated with the WDT interrupt. Otherwise, even if the main function of the firmware crashes or is in an endless loop, the WDT interrupt vector can still be intact and feed the WDT periodically. Note that when the debug probe is connected, the device reset is blocked and an interrupt is generated instead.

The safest way to use the WDT against system crashes is to:

- Configure the watchdog reset period such that the firmware can reset the watchdog at least once during the period, even along the longest firmware delay path
- Reset (feed) the watchdog by clearing the interrupt bit regularly in the main body of the firmware code by writing a '1' to the WDT_MATCH bit in the SRSS_INTR register. Note that this does not reset the watchdog counter; it feeds only the watchdog so that it does not cause a reset for the next two match events

Do not reset the watchdog (clear interrupt) in the WDT interrupt service routine (ISR) if WDT is used as a reset source to protect the system against crashes. Therefore, do not use the WDT reset feature and ISR together.

The recommended steps to use WDT as a reset source are as follows:

1. Make sure the WDT configuration is unlocked by clearing the WDT_LOCK bits[31:30] of the WDT_CTL register. Note that clearing the bits requires two writes to the register with each write clearing one bit as explained in Table 159
2. Write the desired IGNORE_BITS_ABOVE in the WDT_MATCH2 register to set the counter resolution to be used for the match

3. Write any match value to the WDT_MATCH register. The match value does not control the period of watchdog reset as the counter is not reset on a match event. This value provides an option to control only the first interrupt interval; after that the successive interrupts' period is defined by the IGNORE_BITS_ABOVE. The approximate watchdog period (in seconds) is given by the following equation:

$$\text{Watchdog reset period} = ILO_{period} \times (2 \times 2^{(IGNORE_BITS_ABOVE)} + WDT_MATCH) \quad (5)$$

4. Set the WDT_MATCH bit in the SRSS_INTR register to clear any pending WDT interrupt
5. Enable ILO by setting the ENABLE bit [31] of the CLK_ILO_CONFIG register
6. Enable the WDT by setting the WDT_EN bit in the WDT_CTL register
7. Lock the WDT and ILO configuration by writing '3' to the WDT_LOCK bits. This also locks the LFCLK_SEL bits of the CLK_SELECT register
8. In the firmware, write '1' to the WDT_MATCH bit in the SRSS_INT register to feed (clear interrupt) the watchdog

20.3.3 Watchdog interrupt

In addition to generating a device reset, the WDT can be used to generate interrupts. Note that interrupt servicing and watchdog reset cannot be used simultaneously using the free-running WDT.

The watchdog counter can send interrupt requests to the CPU in CPU Active power modes and to the wake-up interrupt controller (WIC) in CPU Sleep and DeepSleep power modes. In addition, the watchdog can wake up the device from Hibernate power mode. It works as follows:

- CPU Active mode: In Active power mode, the WDT can send the interrupt to the CPU. The CPU acknowledges the interrupt request and executes the ISR. Clear the interrupt in the ISR
- CPU Sleep or DeepSleep mode: In this mode, the CPU is powered down. Therefore, the interrupt request from the WDT is directly sent to the WIC, which then wakes up the CPU. The CPU acknowledges the interrupt request and executes the ISR. Clear the interrupt in the ISR
- Hibernate mode: In this mode, the entire device except a few peripherals (such as WDT and LPCOMP) are powered down. Any interrupt to wake up the device in this mode results in a device reset. Hence, there is no interrupt service routine or mechanism associated with this mode

  For more details on device power modes, see the Device power modes section

Because of its free-running nature, the WDT should not be used for periodic interrupt generation. Use the Multi-counter WDT (MCWDT), instead, to generate periodic interrupts. If absolutely required, follow these steps to use the WDT as a periodic interrupt generator:

1. Unlock the WDT if this is the first update to the WDT registers after a DeepSleep or Hibernate wake-up, or a device reset
2. Write the desired IGNORE_BITS_ABOVE in the WDT_MATCH register to set the counter resolution to be used for the match
3. Write the desired match value to the WDT_MATCH register
4. Set the WDT_MATCH bit in the SRSS_INTR register to clear any pending WDT interrupt
5. Enable the WDT interrupt to CPU by setting the WDT_MATCH bit in SRSS_INTR_MASK
6. Enable the SRSS interrupt to the CPU by configuring the appropriate ISER register (see the Interrupts section for details)
7. In the ISR, unlock the WDT; clear the WDT interrupt and add the desired match value to the existing match value. By doing so, another interrupt is generated when the counter reaches the new match value (period)

20.4 Multi-counter WDT

20.4.1 Overview

The PSC3M/P6 MCU has one MCWDT block. The MCWDT block includes two 16-bit counters (MCWDT0_WDT0 and MCWDT0_WDT1) and one 32-bit counter (MCWDT0_WDT2). You can configure these counters to work independently or in cascade (up to 64-bit). The 16-bit counters can generate an interrupt or reset the device. The 32-bit counter can only generate an interrupt. All the counters are clocked by LFCLK.

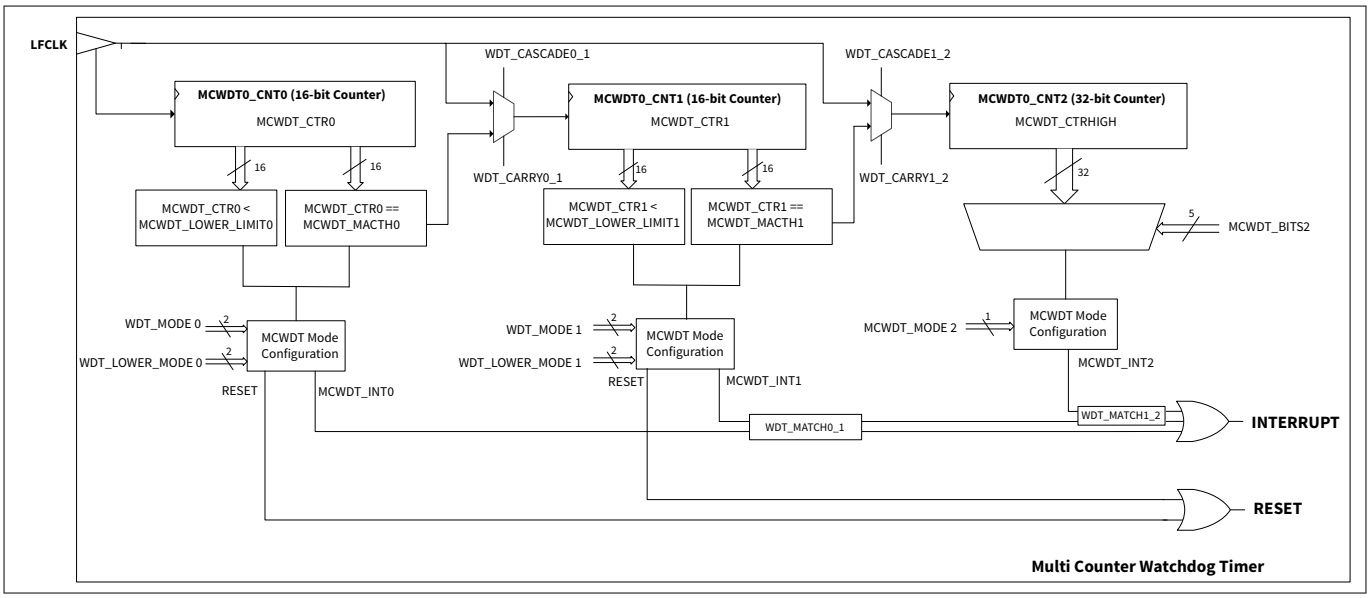


Figure 75 Multi-Counter WDT functional diagram

20.4.1.1 MCWDT0_WDT0 and MCWDT0_WDT1 counters operation

MCWDT0_WDT0 and MCWDT0_WDT1 are 16-bit up counters, which can be configured to be a 16-bit free-running counter or a counter with any 16-bit period. Use these counters to generate an interrupt or reset.

The WDT_CTR0 bits [15:0] and WDT_CTR1 bits [31:16] of the MCWDT0_CNTLOW register hold the current counter values of MCWDT0_WDT0 and MCWDT0_WDT1 respectively. The WDT_MATCH0 bits [15:0] and WDT_MATCH1 bits [31:16] of the MCWDT0_MATCH register store the match value for MCWDT0_WDT0 and MCWDT0_WDT1 respectively. The WDT_MODEx bits of the MCWDT0_CONFIG register configure the action the watchdog counter takes on a match event (WDT_MATCHx == WDT_CTRx). The MCWDT0_WDT0/WDT1 counters perform the following actions:

- Assert interrupt (WDT_INTx) on match
- Assert a device reset on match
- Assert an interrupt on match and a device reset on the third unhandled interrupt

In addition to generating reset and interrupt, configure the match event to clear the counters. This is done by setting the WDT_CLEARx bit of the MCWDT0_CONFIG register. The MCWDT0_WDT0/WDT1 counter operation is shown in Figure 76

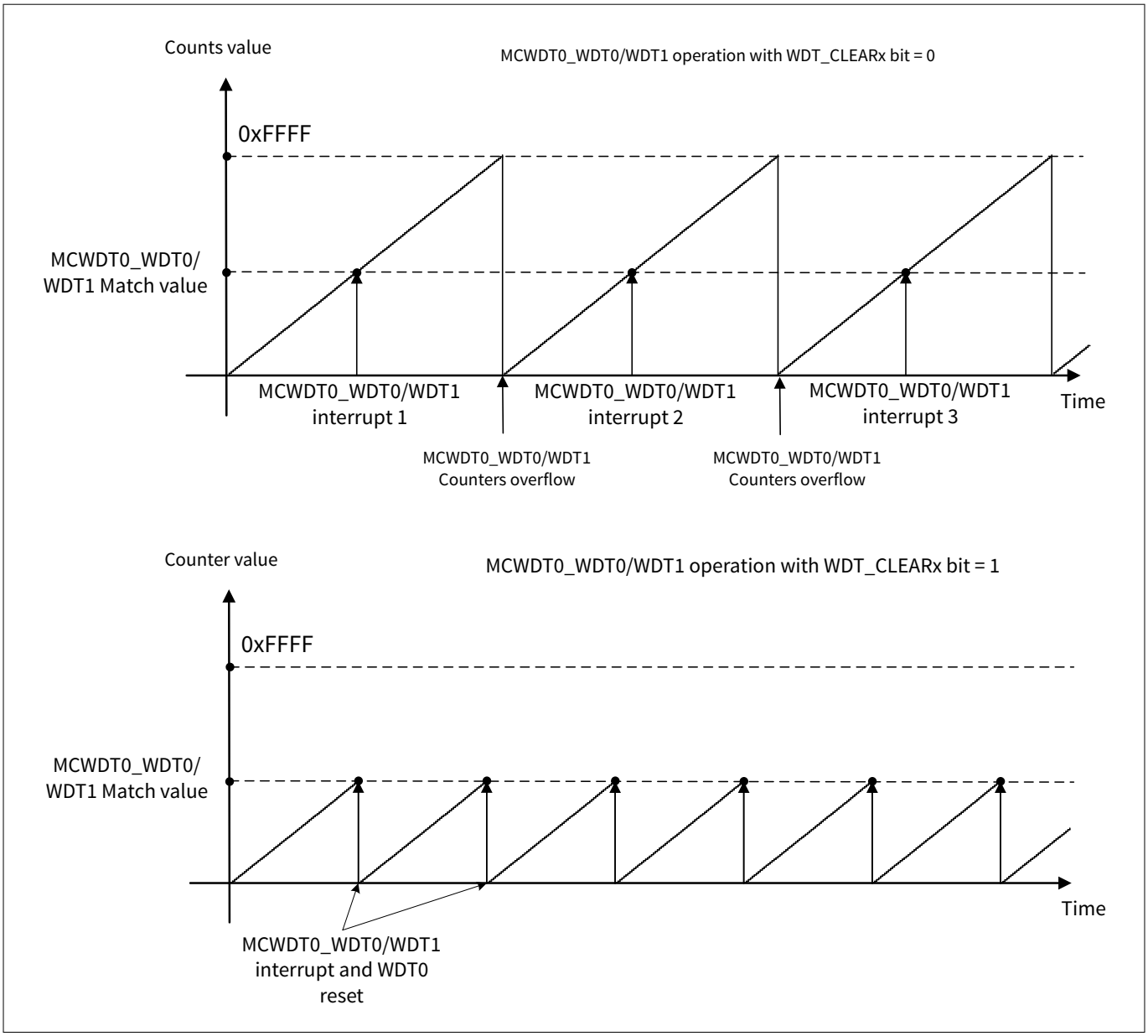


Figure 76 MCWDT0_WDT0/WDT1 operation

The PSC3M/P6 MCU supports the window mode operation for the MCWDT0_CTR0/1 sub-counters as shown in Figure 77. In window mode, an early threshold value stored in WDT_LOWER_LIMIT0 bits [15:0] and WDT_LOWER_LIMIT1 bits [31:16] of the MCWDT0_LOWER_LIMIT register for MCWDT0_WDT0 and MCWDT0_WDT1 respectively. It can be used if the WDT_LOWER_MODE0 bit [5:4] and WDT_LOWER_MODE1 bit [13:12] is set to '2' in the MCWDT0_CONFIG register to generate a reset if the counter is serviced too early.

The Window capability only works when the counter does not overflow and wrap around. As a practical matter, it means that only WDT_MODE0/1== RESET is useful for window operation. Other WDT_MODE0/1 settings are not supported for window operation.

The Window capability works with sub-counters 0 and 1. Cascading them is also supported, in which case the lower limit registers are treated as a combined limit. The Window capability is not supported for sub-counter 2.

It must be possible to dynamically change the lower and upper limits immediately after servicing. This must not trigger any action unless the original service violated the allowed window.

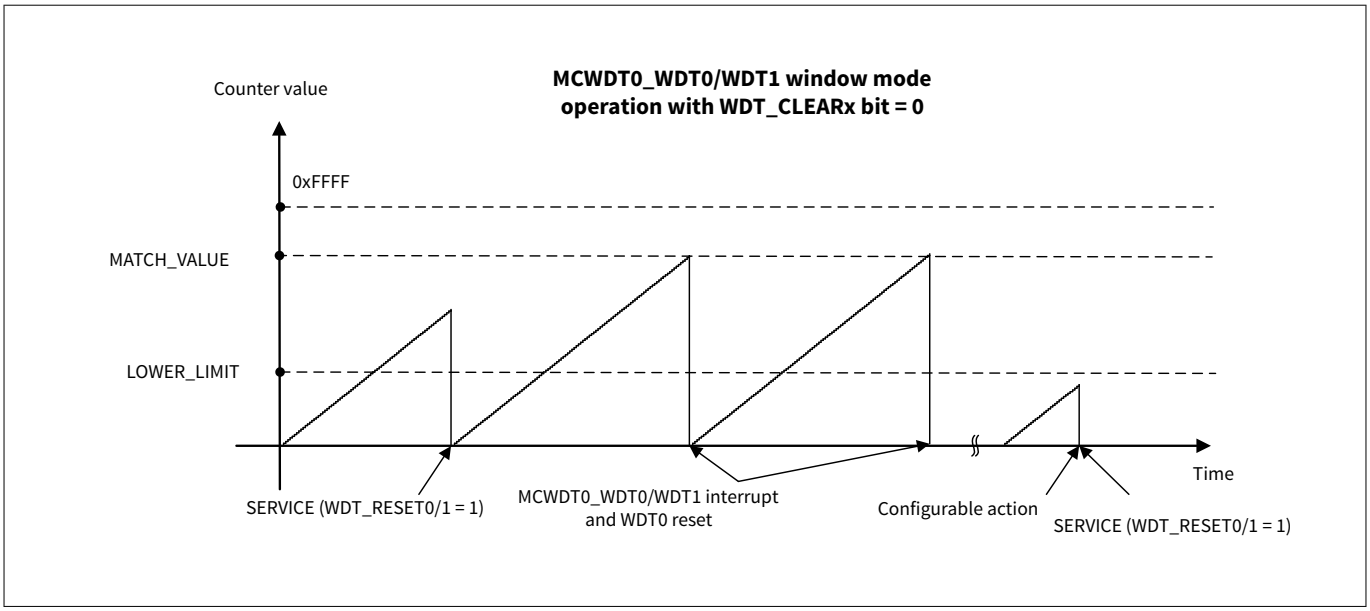


Figure 77 MCWDT0_WDT0/WDT1 window mode operation

Table 160 MCWDT0_WDT0 and MCWDT0_WDT1 configuration options

Register [Bit_Pos]	Bit_Name	Description
MCWDT0_CONFIG[1:0] MCWDT0_CONFIG[9:8]	WDT_MODE0 WDT_MODE1	Watchdog Counter Action on Match. Action is taken on the next increment after the values match (WDT_CTR0=WDT_MATCH0).
		0:Do nothing
		1: Assert interrupt (WDT_INTx)
		2: Assert device Reset
		3: Assert interrupt on match and a device reset after 3rd unhandled interrupt
MCWDT0_CONFIG[5:4] MCWDT0_CONFIG[13:12]	WDT_LOWER_MODE0 WDT_LOWER_MODE1	Watchdog Counter Action on service before lower limit. 0: Do nothing 1: Assert interrupt (MCWDT_INTx) 2: Assert WDT reset
MCWDT0_CONFIG[2] MCWDT0_CONFIG[10]	WDT_CLEAR0 WDT_CLEAR1	Clear the MCWDT0_WDT0/WDT1 counter on match. In other words, (WDT_MATCHx + 1) acts similar to a period for the MCWDT0_WDT0/ WDT1 counter. 0: Free running counter 1: Clear WDT_CTRx bits on match.

(table continues...)

Table 160 (continued) MCWDT0_WDT0 and MCWDT0_WDT1 configuration options

Register [Bit_Pos]	Bit_Name	Description
MCWDT0_CNTLOW[15:0] MCWDT0_CNTLOW[31:16]	WDT_CTR0 WDT_CTR1	Current counter values. Bits[15:0] contains the current value of the MCWDT0_WDT0 counter. Bits[31:16] contains the current value of the MCWDT0_WDT1 counter.
MCWDT0_MATCH[15:0] MCWDT0_MATCH[31:16]	WDT_MATCH0 WDT_MATCH1	Match values Changing WDT_MATCHx requires 1.5 CLK_LF cycles to come into effect. After changing WDT_MATCHx, do not enter the DeepSleep mode for at least one CLK_LF cycle to ensure theWDT updates to the new setting.
MCWDT0_LOWER_LIMIT[15:0] MCWDT0_LOWER_LIMIT[31:16]	WDT_LOWER_LIMIT0 WDT_LOWER_LIMIT1	Provides lower limit values for counters 0 and 1.

20.4.1.2 MCWDT0_WDT2 counter operation

The MCWDT0_WDT2 is a 32-bit free-running counter, which can be configured to generate an interrupt. The MCWDT0_CNTHIGH register holds the current value of the WDT2 counter. WDT2 does not support a match feature. However, it can be configured to generate an interrupt when one of the counter bits toggle. The WDT_BITS2 bits [28:24] of the MCWDT0_CONFIG register selects the bit on which the MCWDT0_WDT2 interrupt is asserted. WDT_MODE2 bit [16] of the MCWDT0_CONFIG register decides whether to assert an interrupt on bit toggle or not. Figure 78 shows the MCWDT0_WDT2 counter operation.

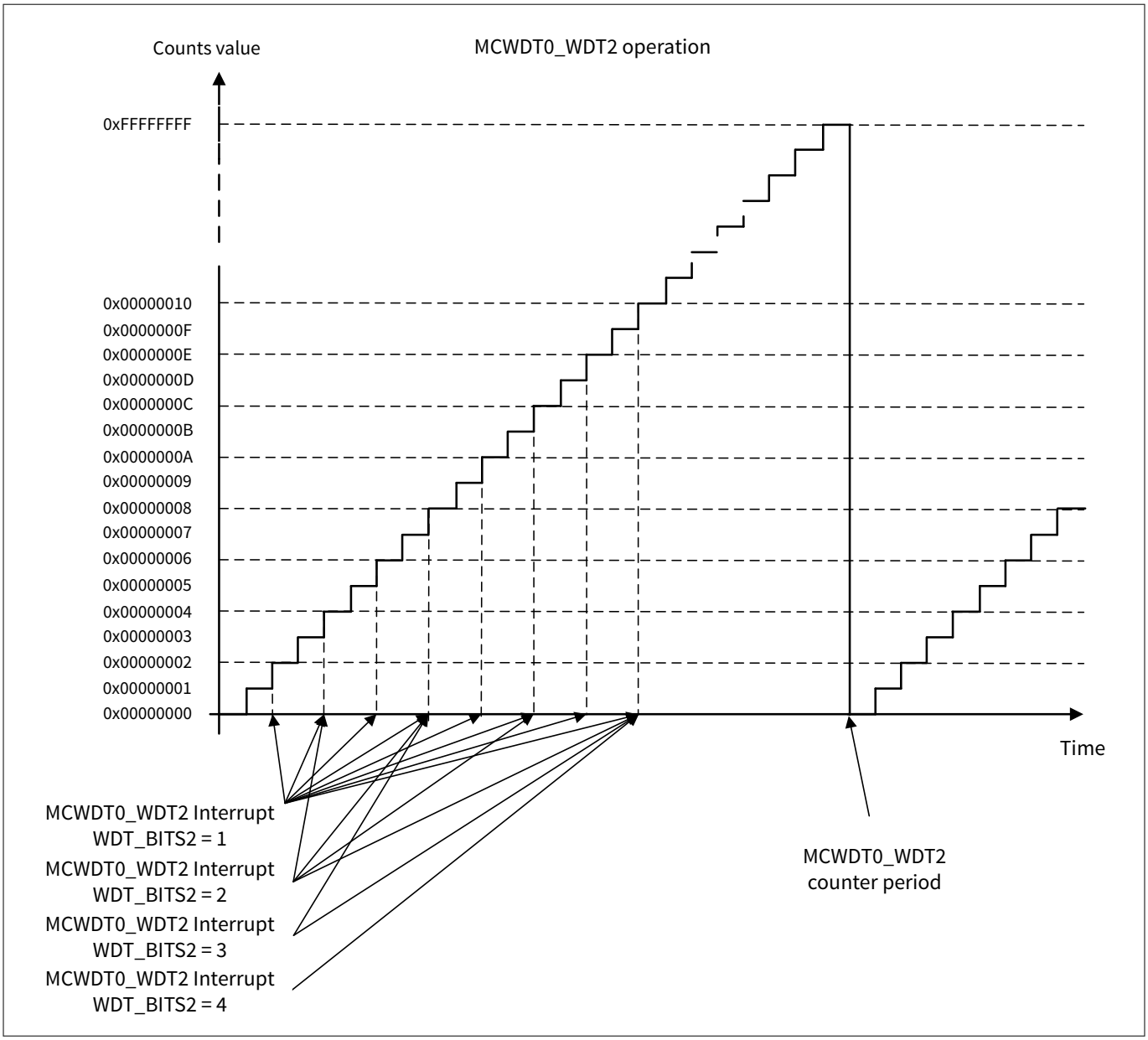


Figure 78 MCWDT0_WDT2 operation

Table 161 MCWDT0_WDT2 Configuration Options

Register [Bit_Pos]	Bit_Name	Description
MCWDT0_CONFIG[16]	WDT_MODE2	MCWDT0_WDT2 mode 0: Free-running counter 1: Free-running counter with interrupt (WDT_INTx) request generation. The interrupts are generated when the bit position specified in WDT_BITS2 toggles in the counter value held by MCWDT0_CNTHIGH register.

(table continues...)

Table 161 (continued) MCWDT0_WDT2 Configuration Options

Register [Bit_Pos]	Bit_Name	Description
MCWDT0_CONFIG[28:24]	WDT_BITS2	Bit to monitor for MCWDT0_WDT2 interrupt assertion 0: Asserts when bit [0] of MCWDT0_CNTHIGH register toggles (interrupt every tick) …… 31: Asserts when bit [31] of MCWDT0_CNTHIGH register toggles (interrupt every 2^(31) ticks)
MCWDT0_CNTHIGH[31:0]	WDT_CTR2	Current counter value of MCWDT0_WDT2

20.4.2 Enabling and disabling MCWDT

The MCWDT counters are enabled by setting the WDT_ENABLEx bit in the MCWDT0_CTL register and are disabled by clearing it. Enabling or disabling a MCWDT requires up to two LFCLK cycles to come into effect. Therefore, the WDT_ENABLEx bit value must not be changed more than once in that period and the WDT_ENABLEDx bit of the MCWDT0_CTL register can be used to monitor the enabled/disabled state of the counter.

The WDT_RESETx bit of the MCWDT0_CTL register clears the corresponding MCWDT0 counter when set in firmware. The hardware clears the bit after the counter resets. This option is useful when the MCWDT0_WDT0 or MCWDT0_WDT1 is configured to generate a device reset on a match event. In such cases, the device resets when the counter reaches the match value. Thus, setting the WDT_RESET0 or WDT_RESET1 bit resets MCWDT0_WDT0 or MCWDT0_WDT1 respectively, preventing device reset.

After the MCWDT is enabled, do not write to the MCWDT configuration (MCWDT0_CONFIG) and control (MCWDT0_CTL) registers. Accidental corruption of these registers can be prevented by setting the MCWDT_LOCK bits [31:30] of the MCWDT0_LOCK register. If the application requires updating the match value (WDT_MATCH) when the MCWDT is running, the MCWDT_LOCK bits must be cleared. The MCWDT_LOCK bits require two different writes to clear both the bits. Writing a ‘1’ to the bits clears bit 0. Writing a ‘2’ clears bit 1. Writing a ‘3’ sets both the bits and writing ‘0’ does not have any effect. Note that the MCWDT_LOCK bit protects only the MCWDT0_CTL, MCWDT0_CONFIG, and MCWDT0_MATCH registers. The LFCLK select registers are protected by the free-running WDT lock bits.

Table 162 Watchdog configuration options

Register [Bit_pos]	Bit_Name	Description
MCWDT0_CTL[0] MCWDT0_CTL[8] MCWDT0_CTL[16]	WDT_ENABLE0 WDT_ENABLE1 WDT_ENABLE2	Enable MCWDT counter. May take up to two LFCLK cycles to take effect. 0: Counter is disabled (not clocked) 1: Counter is enabled (counting up)

(table continues...)

Table 162 (continued) Watchdog configuration options

Register [Bit_pos]	Bit_Name	Description
MCWDT0_CTL[1] MCWDT0_CTL[9] MCWDT0_CTL[17]	WDT_ENABLED0 WDT_ENABLED1 WDT_ENABLED2	Indicates actual state of counter x. Monitor this bit after changing the WDT_ENABLEx bit, to receive an acknowledgment of the changes.
MCWDT0_CTL[3] MCWDT0_CTL[11] MCWDT0_CTL[19]	WDT_RESET0 WDT_RESET1 WDT_RESET2	Reset the MCWDT counter x to 0. Hardware clears the bit when reset is complete. 0: Software - No action 1: Software - Resets the counter
MCWDT0_LOCK[31:30]	MCWDT_LOCK	Locks or unlocks write access to the MCWDT0_CTL, MCWDT0- _CONFIG, and MCWDT0_MATCH registers. When the bits are set, the lock is enabled. 0: No effect 1: Clears bit 0 2: Clears bit 1 3: Sets both bit 0 and 1 (lock enabled)

20.4.3 Watchdog cascade options

The cascade configuration shown in Figure 75 provides an option to increase the MCWDT counter range or maximum timeout. The WDT_CASCADE0_1 bit [3] of the MCWDT0_CONFIG register cascades MCWDT0_WDT0 and MCWDT0_WDT1 and the WDT_CASCADE1_2 bit [11] of the MCWDT0_CONFIG register cascades MCWDT0_WDT1 and MCWDT0_WDT2. Note that cascading two 16-bit counters does not provide a 32-bit counter; instead, you get a 16-bit period counter with a 16-bit prescaler. For example, when cascading MCWDT0_WDT0 and MCWDT0_WDT1, MCWDT0_WDT0 acts as a prescaler for MCWDT0_WDT1 and the prescaler value is defined by the WDT_MATCH0 bits [15:0] in the MCWDT0_MATCH register. The MCWDT0_WDT1 has a period defined by WDT_MATCH1 bits [31:16] in the MCWDT0_MATCH register. The same logic applies to MCWDT0_WDT1 and MCWDT0_WDT2 cascading.

Additionally, the carry operation of the cascaded counters can be configured on a match or roll over using the WDT_CARRY0_1 and WDT_CARRY1_2 bits of the MCWDT0_CONFIG register. The WDT_MATCH0_1 and WDT_MATCH1_2 bits can be configured to define the matching behavior of the cascaded counters. The interrupt on match can be generated when only the first counter matches or when both the cascaded counters match simultaneously.

The summary of the cascade configuration bits is given in Table 163

Table 163 Watchdog cascade options

Register [Bit_Pos]	Bit_name	Description
MCWDT0_CONFIG[3]	WDT_CASCADE0_1	Cascade Watchdog Counters 0,1. Counter 1 increments the cycle after WDT_CTR0=WDT_MATCH0. 0: Independent counters 1: Cascaded counters
MCWDT0_CONFIG[11]	WDT_CASCADE1_2	Cascade Watchdog Counters 1,2. Counter 2 increments the cycle after WDT_CTR1=WDT_MATCH1. It is allowed to cascade all three WDT counters. 0: Independent counters 1: Cascaded counters. When cascading all three counters, WDT_CLEAR1 must be 1.
MCWDT0_CONFIG[6]	WDT_CARRY0_1	Carry out behavior that applies when WDT_CASCADE0_1==1. This bit is not used when WDT_CASCADE0_1==0. 0: carry out on counter 0 match. 1: carry out on counter 0 roll-over.
MCWDT0_CONFIG[14]	WDT_CARRY1_2	Carry out behavior that applies when WDT_CASCADE1_2==1. This bit is not used when WDT_CASCADE1_2==0. 0: carry out on counter 1 match. 1: carry out on counter 1 roll-over.
MCWDT0_CONFIG[7]	WDT_MATCH1_0	Specifies matching behavior when WDT_CASCADE0_1==1. When WDT_CASCADE0_1==0, this bit is not used and match is based on counter 1 alone. 0: Match based on counter 1 alone. 1: Match based on counter 1 and counter 0 matching simultaneously.

(table continues...)

Table 163 (continued) Watchdog cascade options

Register [Bit_Pos]	Bit_name	Description
MCWDT0_CONFIG[15]	WDT_MATCH1_2	Specifies matching behavior when WDT_CASCADE1_2==1. When WDT_CASCADE1_2==0, this bit is not used and match is based on counter 2 alone. 0: Match based on counter 2 alone. 1: Match based on counter 2 and counter 1 matching simultaneously.

When using cascade (WDT_CASCADE0_1 or WDT_CASCADE1_2 set), resetting the counters when the prescaler or lower counter is at its match value with the counter configured to clear on match, results in the upper counter incrementing to 1 instead of remaining at 0. Correct this behavior by issuing a second reset to the upper counter after approximately 100 µs from the first reset. Note that the second reset is required only when the first reset is issued while the prescaler counter value is at its match value. Figure 79 illustrates the behavior when MCWDT0_WDT0 and MCWDT0_WDT1 are cascaded along with the second reset timing.

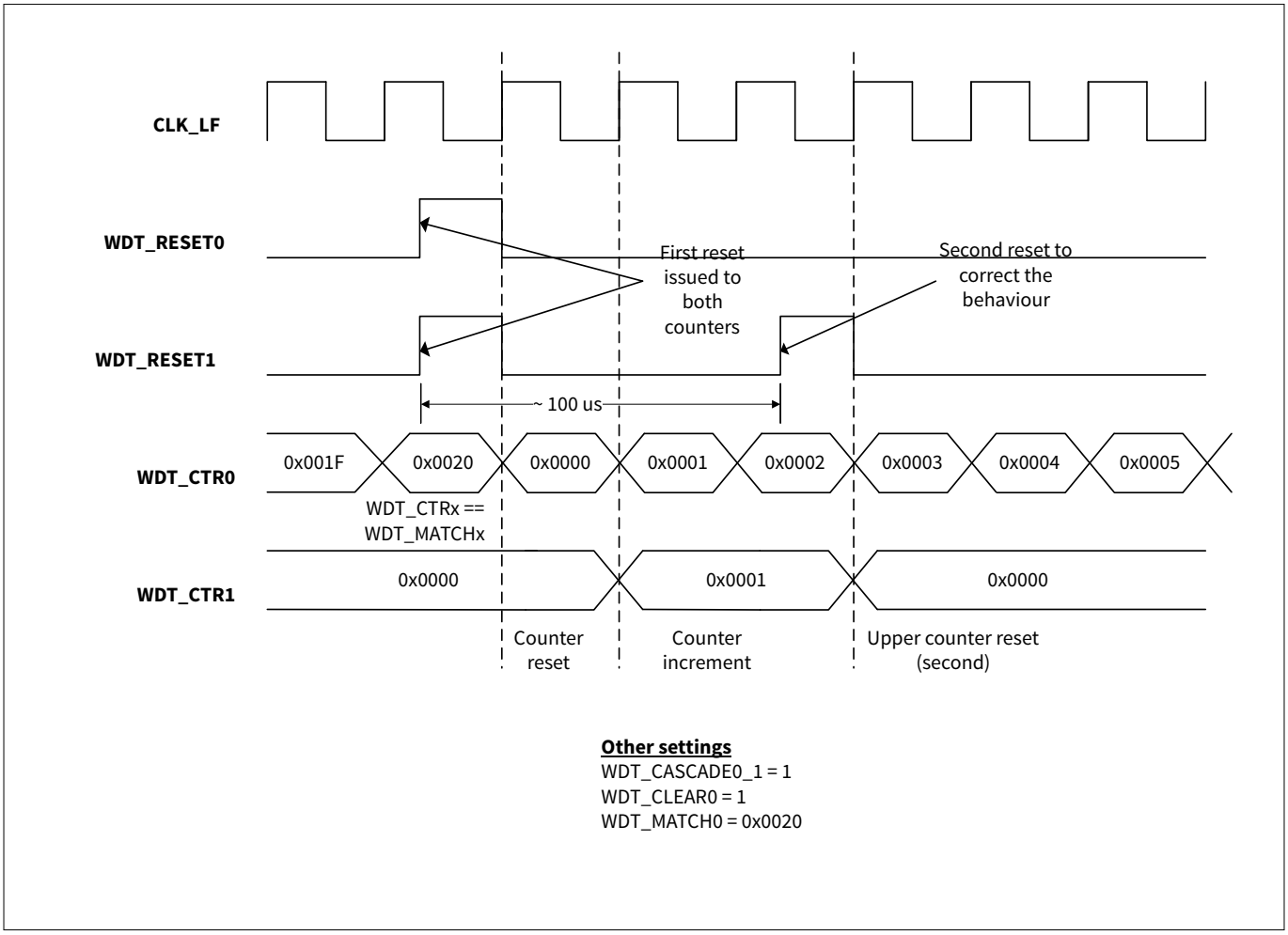


Figure 79 MCWDT reset behavior in Cascaded mode

In addition, the counters exhibit non-monotonicity in the following cascaded conditions:

- If WDT_CASCADE0_1 is set, then WDT_CTR1 does not increment the cycle after WDT_CTR0 = WDT_MATCH0

- If WDT_CASCADE1_2 is set, then WDT_CTR2 does not increment the cycle after WDT_CTR1 = WDT_MATCH1
- If both WDT_CASCADE0_1 and WDT_CASCADE1_2 are set, then WDT_CTR2 does not increment the cycle after WDT_CTR1 = WDT_MATCH1 and WDT_CTR1 does not increment the cycle after WDT_CTR0 = WDT_MATCH0

When cascading is enabled, always read the WDT_CTR1 or WDT_CTR2 counter value only when the prescaler counter (WDT_CTR0 or WDT_CTR1) value is not 0. This makes sure the upper counter is incremented after a match event in the periodic counter.

20.4.4 MCWDT reset

MCWDTx_WDT0 and MCWDTx_WDT1 can be configured to generate a device reset similar to the free-running WDT reset. Note that when the debug probe is connected, the device reset is blocked but an interrupt is generated if configured. Follow these steps to use the MCWDTx_WDT0 or MCWDTx_WDT1 counter of a MCWDTx block to generate a system reset:

1. Configure the MCWDT to generate a reset using the WDT_MODEx bits in MCWDTx_CONFIG. Configure the WDT_MODE0 or WDT_MODE1 bits in MCWDTx_CONFIG to '2' (reset on match) or '3' (interrupt on match and reset on the third unhandled interrupt)
2. Optionally, set the WDT_CLEAR0 or WDT_CLEAR1 bit in the MCWDTx_CONFIG register for MCWDTx_WDT0 or MCWDTx_WDT1 to reset the corresponding watchdog counter to '0' on a match event. Otherwise, the counters are free running. See MCWDT0_WDT0 and MCWDT0_WDT1 counters operation for details
3. Calculate the watchdog reset period such that the firmware can reset the watchdog at least once during the period, even along the longest firmware delay path. If WDT_MODEx == 2, the match value is the same as the watchdog period. If WDT_MODEx == 3, the match value is one-third of the watchdog period. Write the calculated match value to the WDT_MATCH register for MCWDTx_WDT0 or MCWDTx_WDT1. Optionally, enable cascading to increase the interval

   ***Note**: The legal value for the WDT_MATCH field is 1 to 65535*

4. If WDT_MODEx == 2, set the WDT_RESETx bit in the MCWDTx_CONFIG register to reset the WDTx counter to 0. If WDT_MODEx == 3, set the WDT_INTx bit in MCWDTx_INTR to clear any pending interrupts
5. Enable WDTx by setting the WDT_ENABLEx bit in the MCWDTx_CTL register. Wait until the WDT_ENABLEDx bit is set
6. Lock the MCWDTx configuration by setting the MCWDT_LOCK bits of the MCWDTx_CTL register
7. In the firmware, feed (reset) the watchdog as explained in step 4

Do not reset the watchdog in the WDT ISR. It is also not recommended to use the same watchdog counter to generate a system reset and interrupt. For example, if MCWDTx_WDT0 is used to generate system reset against crashes, then MCWDTx_WDT1 or MCWDTx_WDT2 should be used for periodic interrupt generation

20.4.5 MCWDT interrupt

When configured to generate an interrupt, the WDT_INTx bits of the MCWDTx_INTR register provide the status of any pending watchdog interrupts. The firmware must clear the interrupt by setting the WDT_INTx. The WDT_INTx bits of the MCWDTx_INTR_MASK register mask the corresponding WDTx interrupt of the MCWDTx block to the CPU.

Follow these steps to use WDT as a periodic interrupt generator:

1. Write the desired match value to the MCWDT_MATCH register for MCWDTx_WDT0/WDT1 or the WDT_BITS2 value to the MCWDTx_CONFIG register for MCWDTx_WDT2

   ***Note**: The legal value for the WDT_MATCH field is 1 to 65535*

2. Configure the WDTx to generate an interrupt using the WDT_MODEx bits in MCWDTx_CONFIG. Configure the WDT_MODE0 or WDT_MODE1 bits in MCWDTx_CONFIG for MCWDTx_WDT0 or MCWDTx_WDT1 to '1' (interrupt on match) or '3' (interrupt on match and reset on third unhandled interrupt). For MCWDTx_WDT2, set the WDT_MODE2 bit in the MCWDTx_CONFIG register
3. Set the WDT_INT bit in MCWDTx_INTR to clear any pending interrupt
4. Set the WDT_CLEAR0 or WDT_CLEAR1 bit in the MCWDTx_CONFIG register for MCWDTx_WDT0 or MCWDTx_WDT1 to reset the corresponding watchdog counter to '0' on a match event
5. Mask the WDTx interrupt to the CPU by setting the WDT_INTx bit in the MCWDTx_INTR_MASK register
6. Enable WDTx by setting the WDT_ENABLEx bit in the MCWDTx_CTL register. Wait until the WDT_ENABLEDx bit is set
7. Enable the MCWDTx interrupt to the CPU by configuring the appropriate ISER register. Refer to the Interrupts section
8. In the ISR, clear the WDTx interrupt by setting the WDT_INTx bit in the MCWDTx_INTR register

Note that interrupts from all three WDTx counters of the MCWDT block are mapped as a single interrupt to the CPU. In the interrupt service routine, the WDT_INTx bits of the MCWDTx_INTR register can be read to identify the interrupt source. However, each MCWDT block has its own interrupt to the CPU. For details on interrupts, see the Interrupts section. The MCWDT block can send interrupt requests to the CPU in Active power mode and to the WIC in Sleep and DeepSleep power modes. It works similar to the free-running WDT.

20.5 Reset cause detection

The RESET_WDT bit [0] in the RES_CAUSE register indicates the reset generated by the free-running WDT. The RESET_MCWDT0 bit in the RES_CAUSE register indicates the reset generated by the MCWDT0 block. These bits remain set until cleared or until a Power-On Reset (POR), Brown-Out Detection (BOD), or External Reset (XRES) occurs. All other resets leave this bit unaltered. For more details, see the Reset system chapter on TRM.

20.6 Registers summary

Table 164 WDT registers

Name	Description
WDT_CTL	Watchdog Counter Control Register
WDT_CNT	Watchdog Counter Count Register
WDT_MATCH	Watchdog Counter Match Register
WDT_MATCH2	Watchdog Counter Match Register2
MCWDT0_MCWDT_CNTLOW	Multi-counter WDT Sub-counters 0/1
MCWDT0_MCWDT_CNTHIGH	Multi-counter WDT Sub-counter 2
MCWDT0_MCWDT_MATCH	Multi-counter WDT Counter Match Register for counters 0 and 1
MCWDT0_MCWDT_CONFIG	Multi-counter WDT Counter Configuration, including bit toggle interrupt generation for counter 2
MCWDT0_MCWDT_CTL	Multi-counter WDT Counter Control
MCWDT0_MCWDT_LOCK	Multi-counter WDT Counter Lock Register
MCWDT0_MCWDT_LOWER_LIMIT	Multi-Counter WDT Lower Limit Register
MCWDT0_MCWDT_INTR	Multi-counter WDT Counter Interrupt Register

(table continues...)

Table 164 (continued) WDT registers

Name	Description
MCWDT0_MCWDT_INTR_SET	Multi-counter WDT Counter Interrupt Set Register
MCWDT0_MCWDT_INTR_MASK	Multi-counter WDT Counter Interrupt Mask Register
MCWDT0_MCWDT_INTR_MASKED	Multi-counter WDT Counter Interrupt Masked Register
CLK_SELECT	Clock Selection Register
CLK_ILO_CONFIG	ILO Configuration
SRSS_INTR	SRSS Interrupt Register
SRSS_INTR_SET	SRSS Interrupt Set Register
SRSS_INTR_MASK	SRSS Interrupt Mask Register
SRSS_INTR_MASKED	SRSS Interrupt Masked Register
RES_CAUSE	Reset Cause Observation Register

21 Backup

The Backup domain adds an “always on” functionality to PSC3M/P6 MCUs using a separate power domain supplied by a backup supply (VBACKUP) such as a battery or supercapacitor. It contains a real-time clock (RTC) with an alarm feature, supported by a 32768-Hz watch crystal oscillator (WCO).

The Backup is not a power mode; it is a power domain with its own power supply, which can be Active during any of the device power modes.

21.1 Features

- Fully-featured RTC
  - Year/Month/Date, Day-of-Week, Hour : Minute : Second fields
  - RTC uses binary instead of Binary Coded Decimal
  - Supports both 12-hour and 24-hour formats
  - Automatic leap year correction
- Configurable alarm function
  - Alarm on Month/Date, Day-of-Week, Hour : Minute : Second fields
  - Two independent alarms
- 32768 Hz WCO with calibration
- Automatic backup power switching
- Built-in supercapacitor charger
- 32 backup registers

21.2 Architecture

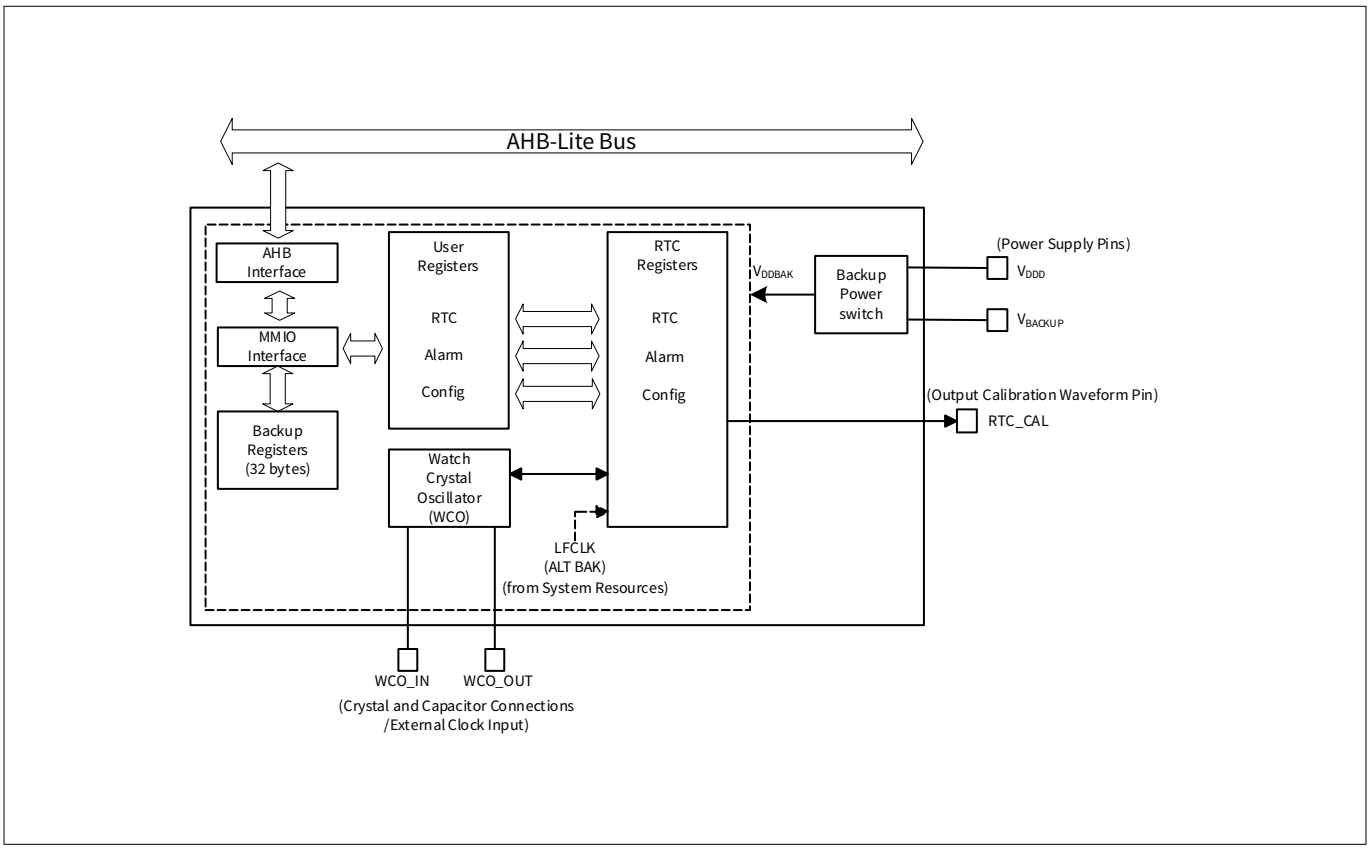


Figure 80 Block diagram

The Backup system includes an accurate WCO that can generate the clock required for the RTC with the help of an external crystal or an external clock inputs. The RTC has a programmable alarm feature, that generate interrupts to the CPU. The AHB-Lite interface provides firmware access to MMIO registers in the Backup domain.

An automatic backup power switch selects the VDDBAK supply required to run the blocks in the Backup domain either VDDD (main power) or VBACKUP (backup battery/supercapacitor power).

The domain also has Backup registers that store 32 bytes of data and retain their contents even when the main supply (VDDD) is OFF, if the backup supply (VBACKUP) is present.

21.3 Power supply

Power to the Backup system (VDDBAK) is automatically switched between VDDD (main supply) and VBACKUP (Backup domain supply). VBACKUP is typically connected to an independent supply derived from a battery or supercapacitor.

There are no VBACKUP versus VDDD sequencing restrictions for the power selector switch. Either VBACKUP or VDDD may be removed during the normal operation, and the Backup system will remain powered.

The VDDBAK_CTL bit field in the BACKUP_CTL register controls the behavior of the power selector switch. See the registers reference manual for more details on this register. The different mode options are as follows:

- VDDBAK_CTL = 0 (Default mode): Selects VDDD when the brownout detector in the system resources is enabled and no brownout situation is detected (see the Power supply and monitoring chapter for more details). Otherwise, it selects the highest supply among VDDD and VBACKUP
- VDDBAK_CTL = 1, 2, or 3: Always selects VBACKUP for debug purposes

If a supercapacitor is connected to VBACKUP, the PSC3M/P6 MCU can charge the supercapacitor while VDDD is available. The supercapacitor charging can be enabled by writing “3C” to the EN_CHARGE_KEY bit field in the BACKUP_CTL register. Note that this feature charges the supercapacitors only and cannot safely charge a battery. Do not write this key when VBACKUP is connected to a battery. The battery charging is handled at the board level using an external circuitry.

***Note**: If VDDD and VBACKUP are connected on the PCB, the Backup domain may require an explicit reset triggered by firmware using the RESET bit field in the BACKUP_RESET register. This firmware reset is required if the VBACKUP supply was invalid during a previous power supply ramp-up or brownout event. It is not necessary to reset the Backup domain if the RES_CAUSE register indicates a non-power related reset as the cause for the reset, or if the PSC3M/P6 MCU just woke from the Hibernate mode and the supply is assumed to be valid the entire time.*

21.4 Clocking

The RTC primarily runs from a 32768 Hz clock, after it is scaled down to one second ticks. This clock signal can come from either of these internal sources:

- Watch crystal oscillator (WCO). This is a high accuracy clock generator that is suitable for RTC applications and requires a 32768 Hz external crystal populated on the application board. The WCO can also operate without crystal, using external clock/sine wave inputs. These additional operating modes are explained later in this section. The WCO is supplied by the Backup domain and can therefore run without VDDD present
- Alternate backup clock (ALTBAK): This option allows the use of LFCLK generated by the System Resources Subsystem (SRSS) as the Backup domain clock. Note that LFCLK is not available in all device power modes or when the VDDD is removed

The clock glitches can propagate into the Backup system when LFCLK is enabled or disabled by the SRSS. In addition, LFCLK may not be as accurate as WCO depending on the actual source of LFCLK. Due to this LFCLK is not recommend for RTC applications. Also, if the WCO is intended to be the clock source then choose it directly instead of routing through LFCLK.

For more details on clocks and calibration, see the Clocking system in the reference manual.

The RTC clock source can be selected using the CLK_SEL bit field in the BACKUP_CTL register. The WCO_EN bit in the BACKUP_CTL register is used to enable or disable the WCO. If the WCO operates with an external crystal, make sure the WCO_BYPASS bit in the BACKUP_CTL register is cleared before enabling the WCO. In addition, the PRESCALER bit-field in BACKUP_CTL is configured for a prescaler value of 32768.

***Note**: The external crystal and load capacitors of proper values must be connected to WCO_IN and WCO_OUT pins. See the device datasheet for details on the component values and the electrical connections. In addition, GPIOs must be configured for WCO_OUT and WCO_IN signals. See the I/O subsystem chapter on reference manual to configure the GPIOs.*

21.4.1 WCO with external clock/sine wave input

The WCO can also operate from external clock/sine wave inputs. In these modes, WCO must be bypassed by setting the WCO_BYPASS bit in the BACKUP_CTL register before enabling the WCO. Also, the GPIOs must be configured for WCO_OUT and WCO_IN signals (in the Analog mode). The external clock/sine wave input modes, prescaler settings, and electrical connections are as follows:

- 32768 Hz External clock mode: In this mode, WCO_IN is floating and WCO_OUT is externally driven by a 32768- Hz square wave clock toggling between ground and VDDD supply levels. In this configuration, the WCO_OUT pin functions as a digital input pin for the external clock. The PRESCALER bit field in BACKUP_CTL is configured to a prescaler value 32768
- 60 Hz External clock mode: This mode can be used for deriving a clock from the 60 Hz AC mains supply. In this mode, WCO_OUT is floating and WCO_IN is driven with an external sine wave of zero DC offset, derived from the 60 Hz/120 V mains through a 100:1 capacitive divider. For example, a suitable capacitive divider is formed by connecting a 220 pF/6 V capacitor between WCO_IN and ground, and a 2.2 pF/200 V capacitor between WCO_IN and the 60 Hz/120 V mains input. The PRESCALER bit field in BACKUP_CTL is configured to a prescaler value of 60
- 50 Hz External clock mode: This mode is similar to the 60-Hz mode, and is used for 50 Hz/220 V mains standard. The capacitive divider explained previously is modified to fit this type of supply by having a 1 pF/250 V capacitor between WCO_IN and the mains input. The PRESCALER bit field in BACKUP_CTL is configured to a prescaler value of 50

21.4.2 Calibration

The absolute frequency of the clock input can be calibrated using the BACKUP_CAL_CTL register. Calibration only works when the PRESCALER bit field in BACKUP_CTL is set to 32768.

CALIB_VAL is a 6 bit field in the BACKUP_CAL_CTL register that holds the calibration value for absolute frequency (at a fixed temperature). One bit of this field translates into 128 ticks added or removed from the clock count. Therefore, each bit translates to a change of 1.085 ppm (128/(60*60*32768)), when CAL_COMP_PER_MIN is set to a default value. The positive values 0x01-0x3c (1..60) add pulses, negative values remove pulses, so giving a range of +/-65.1 ppm (limited by 60 minutes per hour, not the range of this field) when CAL_COMP_PER_MIN is set at default value. Calibration is performed hourly, starting at 59 minutes and 59 seconds, applied as 64 ticks every 30 seconds until there are 2*CALIB_VAL adjustments when CAL_COMP_PER_MIN is set to a default value.

CALIB_SIGN field in the BACKUP_CAL_CTL register controls whether the ticks are added (takes fewer clock ticks to count one second) or subtracted (takes more clock ticks to count one second).

Table 165 rrss

21.5 Reset

The PSC3M/P6 MCU reset sources monitor the device power supply such as the power-on reset (POR) and Brownout detection (BOD) that cannot reset the Backup system as long as the backup supply (VDDBAK) is present. Moreover, the internal and external resets: watchdog timer (WDT) reset and XRES cannot reset the Backup system. The following conditions are needed for reset.

- All the power supplies are removed from the Backup domain, also known as a “cold-start”
- The firmware triggers a Backup domain reset using the RESET bit field in the BACKUP_RESET register

21.6 Real-time clock

Table 166 RTC fields

Bit field name	Number of bits	Description
RTC_SEC	6	Calendar seconds, 0-59
RTC_MIN	6	Calendar minutes, 0-59
RTC_HOUR	5	Calendar hours, value depending on 12/24 hour mode In 24 hour mode, bits RTC_HOURS[20:16] = 0-23 In 12 hour mode, bit RTC_HOURS[20]:0= AM, 1= PM, RTC_HOURS[19:16]=1-12
CTRL_12HR	1	Select 12/24 hour mode: 1=12 hr, 0=24 hr
RTC_DAY	3	Calendar day of the week, 1-7 The user defines the meaning of the values, assign 1= Monday is recommended
RTC_DATE	5	Calendar day of the Month, 1-31 Automatic leap year correction
RTC_MON	4	Calendar month, 1-12
RTC_YEAR	7	Calendar year, 0-99

The RTC supports both 12-hour format with AM/PM flag, and 24-hour format for “hours” field. The RTC also includes a “day of the week” field, which counts from 1 to 7. Programmer defines weekday to be represented by a value of ‘1’.

The RTC implements automatic leap year correction for the Date field (day of the month). If the year is divisible by 4, the month of February (Month = 2) has 29 days instead of 28. When the year reaches 2100 - the year field

rolls over from 99 to 00 - the leap year correction will be wrong (2100 is flagged as a leap year which it is not); therefore, an interrupt is raised to allow the firmware to take appropriate actions. This interrupt is called the century interrupt.

The user registers containing these bit fields: BACKUP_RTC_TIME and BACKUP_RTC_DATE. See the corresponding register descriptions in the registers reference for details. The user registers are in the high-frequency bus clock domain and the actual RTC registers run from the low-frequency 32768-Hz clock, reading and writing RTC registers require special care. These processes are explained in the following sections.

21.6.1 Reading RTC User registers

To start a read transaction, the firmware sets the READ bit in the BACKUP_RTC_RW register. The RTC registers is copied to user registers and frozen so that the firmware reads a coherent RTC value safely. Clear the READ bit to complete the transaction.

The READ bit cannot be set if:

- The RTC is still busy with a previous operation (i.e., the RTC_BUSY bit in the BACKUP_STATUS register is set)
- WRITE bit in the BACKUP_RTC_RW register is set

The firmware verifies the above bits are not set before the READ bit is set.

21.6.2 Writing to RTC User registers

When the WRITE bit in the BACKUP_RTC_RW register is set, data is written into the RTC user registers; otherwise, the RTC user registers are ignored. When all the RTC writes complete, the firmware clears the WRITE bit for the RTC update to take effect. After the WRITE bit is cleared, the hardware copies all the new data on one single WCO clock edge to ensure coherency to the actual RTC registers.

The WRITE bit cannot be set if:

- RTC is still busy with a previous operation (ixs.e., the RTC_BUSY bit in the BACKUP_STATUS register is set)
- READ bit in the BACKUP_RTC_RW register is set

The firmware updates the values written to the RTC fields to form a coherent legal set. The hardware does not check the validity of the written values. Writing illegal values results in an undefined behavior of the RTC.

In the middle of an RTC update with the WRITE bit set, and a brownout, reset, or entry to Deep Sleep or Hibernate mode occurs, the write operation will not complete. This is because the WRITE bit is cleared by a reset, and the RTC update is triggered only when this bit is cleared by a WRITE transaction. If the write operation is in progress (RTC_BUSY), data corruption occurs. Otherwise, the system is reset or enters Deep Sleep or Hibernate mode.

To update only one or a few of the RTC fields, for example, when the RTC is adjusted for daylight saving time (DST), then update only the hour field, although the seconds and minutes fields are disturbed and continues to run. In this case, an ‘Update’ flag is maintained for each RTC field. The updated fields are copied to the actual RTC when the BACKUP_RTC_RW.WRITE bit is cleared.

21.7 Alarm feature

The alarm feature allows the RTC to generate an interrupt, which is used to wake the system from Sleep, Deep Sleep, and Hibernate power modes.

The Alarm feature consists of six fields corresponding to the fields of the RTC: Month/Date, Day-of-Week, and Hour : Minute : Second. Each Alarm field has an enable bit that is set to enable matching; if the bit is cleared, then the field will be ignored for matching.

The alarm bit fields are as follows:

Table 167 Alarm bit fields

Bit field name	Bit_Pos	Register_name	Description
ALM_SEC	5:0	BACKUP_ALMx_TIME	Alarm seconds, 0-59
ALM_SEC_EN	7	BACKUP_ALMx_TIME	Alarm second enable: 0=ignore, 1=match
ALM_MIN	13:8	BACKUP_ALMx_TIME	Alarm minutes, 0-59
ALM_MIN_EN	15	BACKUP_ALMx_TIME	Alarm minutes enable: 0=ignore, 1=match
ALM_HOUR	20:16	BACKUP_ALMx_TIME	Alarm hours, value depends on 12/24 HR mode 24 HR: [4:0]=0-23 12 HR: [4]:0=AM, 1=PM, [3:0]=1-12
ALM_HOUR_EN	23	BACKUP_ALMx_TIME	Alarm hour enable: 0=ignore, 1=match
ALM_DAY	26:24	BACKUP_ALMx_TIME	Alarm day of the week, 1-7 It is up to the user to define the meaning of the values, but 1=Monday is recommended
ALM_DAY_EN	31	BACKUP_ALMx_TIME	Alarm day of the Week enable: 0=ignore, 1=match
ALM_DATE	4:0	BACKUP_ALMx_DATE	Alarm day of the Month, 1-31. Leap year corrected
ALM_DATE_EN	7	BACKUP_ALMx_DATE	Alarm day of the month enable: 0=ignore, 1=match
ALM_MON	11:8	BACKUP_ALMx_DATE	Alarm month, 1-12
ALM_MON_EN	15	BACKUP_ALMx_DATE	Alarm month enable: 0=ignore, 1=match

(table continues...)

Table 167 (continued) Alarm bit fields

Bit field name	Bit_Pos	Register_name	Description
ALM_EN	31	BACKUP_ALMx_DATE	Master enable for alarm 1. 0: alarm 1 is disabled. Fields for date and time are ignored. 1: alarm 1 is enabled. The alarm triggers when the new date and time matches all the enabled date and time fields, which can happen more than once depending on configuration. If none of the date and time fields are enabled, then this alarm triggers once every second.

If the master enable (ALM_EN) is set, but all the alarm fields for date and time are disabled, an alarm interrupt is generated once every second. Note that there is no alarm field for year.

The PSC3M/P6 MCU has two independent alarms. See the BACKUP_ALM1_TIME, BACKUP_ALM1_DATE, BACKUP_ALM2_TIME, and BACKUP_ALM2_DATE registers in the registers reference manual for more details.

***Note**: The alarm user registers, similar to RTC user registers, require certain special steps before read/write operations, as explained in the Reading RTC User registers and Writing to RTC User registers in the reference manual.*

Interrupts must be properly configured for the RTC to generate interrupts/wake up events. Also, to enable RTC interrupts to wake up the device from Hibernate mode, the HIB_WAKE_RTC bit in the PWR_HIB_WAKE_CTL register must be set. See the Device power modes chapter and Interrupts chapters for more details.

The BACKUP_INTR_MASK register is used to disable certain interrupts from the Backup system.

Table 168 Interrupt Mask Bits

Bit Name	Description
ALARM1	Mask bit for interrupt generated by ALARM1
ALARM2	Mask bit for interrupt generated by ALARM2
CENTURY	Mask bit for century interrupt (interrupt generated when the year field rolls over from 99 to 00)

21.8 Backup registers

The Backup domain has thirty two registers (BACKUP_BREG_SET0 to BACKUP_BREG_SET3). Each of these register can be used to store 1 byte of important user information/flags. These registers can retain their contents even when the main supply (VDDD) is off as long as backup supply (VBACKUP) is present. These registers can also be used to store information that must be retained when the device enters Hibernate mode.

21.9 Registers summary

Table 169 Backup registers

Register name	Description
BACKUP_CTL	Main control register (including power and clock)
BACKUP_RTC_RW	RTC read/write control register
BACKUP_STATUS	Status register
BACKUP_RTC_TIME	Calendar seconds, minutes, hours, and day of week
BACKUP_RTC_DATE	Calendar day of month, month, and year
BACKUP_ALM1_TIME	Alarm 1 seconds, minute, hours, and day of week
BACKUP_ALM1_DATE	Alarm 1 day of month, and month
BACKUP_ALM2_TIME	Alarm 2 seconds, minute, hours, and day of week
BACKUP_ALM2_DATE	Alarm 2 day of month, and month
BACKUP_INTR	Interrupt Request register
BACKUP_INTR_MASK	Interrupt Mask register
BACKUP_INTR_MASKED	Interrupt Masked Request register
BACKUP_BREG_SET0 - BACKUP_BREG_SET3	Backup registers
BACKUP_RESET	Reset Register for the Backup domain

22 Reset system

PSC3M/P6 MCU supports several types of resets that ensure error-free operation during power-up and allow the device to reset based on user-supplied external hardware or internal software reset signals. The resets, which have a broad scope and are generally aligned with power domains and global power modes, cause a reboot that ends in active mode. Some blocks may have local resets that are described in their respective chapters. Reset assertion is asynchronously propagated, and reset deassertion is synchronized to each clock domain where it is used. Additionally, PSC3M/P6 MCU contains the hardware that has the capability to report the reset that occurs.

Furthermore, the fault manager converts a fault into a high-priority interrupt, such as NMI, to give the processor an opportunity to return to a safe state, such as halting memory writes and releasing peripherals. The processor can then trigger its own local reset or a system reset. This allows recovery from the faults generated by the safety circuits, such as clock supervision, supply supervision, and multi-counter watchdog timer.

22.1 Features

The PSC3M/P6 MCU has the following reset sources:

- Brownout detection (BOD) to reset the device if the power supply falls below the device specifications during normal operation
- External reset (XRES) to reset the device using an external input
- Watchdog timer (WDT/MCWDT) reset to reset the device if the firmware execution fails to periodically service the watchdog timer
- Software-initiated reset(SYS) to reset the device on demand using the firmware
- Test controller or debugger asserted reset(TC_DEBUG)
- Fault detection reset to reset the device if certain faults occur(FAULT_ACT)
- Clock supervision triggered reset for high-frequency clock-related errors (CSV_HF)
- Clock supervision triggered reset for reference clock-related errors (CSV_REF)
- Hibernate wakeup reset to bring the device out of the Hibernate low-power mode(HIB WAKEUP)
- Programmable XRES (PXRES) to trigger a reset equivalent to XRES from the software

22.2 Reset sources

The following sections describe the reset sources available in the PSC3M/P6 MCU family. Table 170 includes a list of all the reset sources.

Table 170 PSC3M/P6 MCU reset sources

Reset source	Reset condition	Availability in power modes	Observable	SRAM retention
External reset (XRES)	External reset pin asserted low	All	No	No
Brownout detect (BOD)	Brownout detected for VDDD, VCCD, and backup supply	All except Off, XRES, Hibernate	No	No
WDT reset	Watchdog timer triggered reset action	Active, Sleep, Deep Sleep	Yes	Yes[1)]

(table continues...)

Table 170 (continued) PSC3M/P6 MCU reset sources

Reset source	Reset condition	Availability in power modes	Observable	SRAM retention
MCWDT reset	Multi-counter WDT triggered reset action	Active, Sleep, Deep Sleep	Yes	Yes[1)]
Hibernate wakeup (HIB_WAKEUP)	Exit from Hibernate mode	Hibernate	Yes (on PWR_HIBERNATE.TOKEN field)	No
Internal system reset (SYS)	System reset (generated internally by CPUSS using SYSRESETREQ)	Active, Sleep	Yes	Yes[1)]
Test/Debug reset (TC_DEBUG)	Test controller or debugger asserted reset	Active, Sleep	Yes	Yes[1)]
Fault detection active logic (FAULT_ACT)	Fault manager triggered reset	Active, Sleep	Yes	Yes[1)]
Clock supervision (CSV_HF)	Loss or frequency violation of a high-frequency clock HF0 – HF6.	Active, Sleep	Yes	CSV_HF0: NO[2)] CSV_HF1..6: YES[1)]
Clock supervision (CSV_REF)	Loss or frequency violation of the reference clock source that is used to monitor the other HF clock sources.	Active, Sleep	Yes	No
Programmable XRES (PXRES)	CPU-triggered reset with scope similar to XRES (triggered using RES_PXRES_CTL.PXRES_TRIGGER).	Active, Sleep	No	No

1) Assumes orderly shutdown handled by the processor before the reset is triggered.
2) Since system RAM is clocked by clk_hf0, a reset triggered by CSV_HF0 could lead to an unstable RAM clock and corrupted contents.

***Note**: None of these sources can reset the backup system. The backup domain is reset only when all the power supplies are removed from it (also known as a cold start) or if the firmware triggers a reset using the BACKUP_RESET register. For more details, see Backup.*

The Table 171 shows the mapping of reset sources to the destinations affected by the reset. An 'X' indicates that the block has at least some register/bits that are affected by the corresponding reset.

Table 171 PSC3M/P6 MCU reset cause distribution

Reset type/ Source	Reset Cause bits	Data registers in FAULT structure	Backup registers	Debug logic	CM33
XRES	X	X		X	X
BOD	X	X	X[1)]	X	X
WDT		X		X[2)]	X
MCWDT		X		X[2)]	X
HIB_WAKEUP	X	X		X	X
SYS		X			X
TC_DEBUG		X		X	X
FAULT_ACT		X			X
CSV_HF		X			X
CSV_REF		X			X

1) If the backup power supply is not present.
2) If the reset source is triggered during Deep Sleep.

***Note**: Watchdog and clock supervision faults are special cases because it is possible that the fault handling system has crashed. For example, a clock supervision fault may mean that a clock glitch occurred when the clock failed. Although unlikely, this glitch could lock up the fault handling system. In this case, the fault logging system may not log the fault, but the reset can be observed.*

22.2.1 Brownout reset

Brownout detection circuits monitor the device's digital voltage supply (VDDD) and internally generated supply voltage (VCCD) and generate a reset if they fall below their voltage threshold. The device stays in reset until all brownout detectors are released. This also occurs during an initial power ramp. See the Power supply and monitoring chapter for more details.

22.2.2 WDT/MCWDT reset

The watchdog timer reset causes a reset if the WDT or multi-counter WDT (MCWDT) are not serviced by the firmware within a specified time limit. In the window mode, MCWDT generates a reset if it is serviced earlier than the specified time. See the Watchdog timer chapter for more details.

22.2.3 Software reset

Software reset is a mechanism that allows the software or a connected debugger to request a system reset. Cortex®-M33 can request a reset by writing a '1' to the TRIGGER_SOFT bit of the RES_SOFT_CTL register. This can also be done via a debugger probe.

In addition to the software reset using the RES_SOFT_CTL register, this MCU also has an option to trigger XRES from the Arm® Cortex®-M33. The XRES interrupt can be triggered by writing a '1' to the PXRES_TRIGGER bit of the RES_PXRES_CTL register. RES_PXRES_CTL is a software-accessible register that triggers a full scope reset equivalent to XRES.

22.2.4 External reset

The external reset (XRES) is a reset triggered by an external signal, causing an immediate system reset when asserted. The XRES pin is active low – a logic '1' on the pin has no effect, while a logic '0' causes a reset. The pin is pulled to logic '1' inside the device and is available as a dedicated pin. For detailed pinout information, refer to the Pinout section of the device datasheet.

The XRES pin holds the device in reset as long as the pin input is '0'. When the pin is released (changed to logic '1'), the device goes through a normal boot sequence. The logical thresholds for XRES and other electrical characteristics are listed in the Electrical Specifications section of the device datasheet. XRES is available in all power modes but cannot reset the backup system.

22.2.5 Debugger reset

The debugger reset is asserted by the test controller or debugger. The DP_CTL_STAT register in the SWD/JTAG-DP has a reset request bit (CDBGRSTREQ), and this request is sent to the reset controller to create the debug reset signal.

The debugger reset can be identified from the RESET_TC_DBGRESET bit field of the RES_CAUSE register.

22.2.6 Fault detection reset

The fault reporting structure in PSC3M/P6 MCU can be configured to request a reset for user-configurable faults.

Faults that occur during Deep Sleep do not wake the system but are pending on wake. If a condition requires a wake from Deep Sleep, it should be configured as an interrupt to the WIC.

Faults generated by clock supervision and MCWDTs can indicate that the fault system may have also failed. These circuits can be configured to directly cause a reset.

22.2.7 High frequency clock error reset

The clock supervision logic initiates a reset due to the loss of a high-frequency clock or watch-crystal clock, or due to a high-frequency clock error. The RESET_CSV_HF is a 7-bit field in the RES_CAUSE2 register that can be used to identify resets caused by the loss or frequency violation of a high-frequency clock.

For more information on clocks, see the Clocking system chapter.

22.2.8 Reference clock error reset

The clock supervision logic requests a reset due to the loss or frequency violation of the reference clock source used to monitor other high-frequency clock sources. The RESET_CSV_REF field in the RES_CAUSE2 register can be used to identify resets caused by the loss or frequency violation of the reference clock source used to monitor the other high-frequency clock sources.

For more information on clocks, see the Clocking system chapter.

22.2.9 Hibernate wakeup reset

The Hibernate wakeup reset occurs when one of the Hibernate wakeup sources performs a device reset to return to the Active power mode. See the Device power modes chapter for details on Hibernate mode and available wakeup sources.

TOKEN is an 8-bit field in the PWR_HIBERNATE register that is retained through a Hibernate wakeup sequence. The firmware can use this bitfield to differentiate hibernate wakeup from a general reset event.

22.3 Identifying reset sources

When the device comes out of reset, it is often useful to know the cause of the reset. Reset causes are recorded in the RES_CAUSE and RES_CAUSE2 registers. These registers have specific status bits allocated for some of the reset sources. They record the occurrences of WDT reset, MCWDT reset, software reset, logic-protection fault reset, debugger reset, and clock-supervision resets. However, they do not record the occurrences of BOD, XRES, PXRES, or Hibernate wakeup resets. The bits in these registers are set on the occurrence of the corresponding reset and remain set after the reset until cleared by the firmware or a loss of retention such as XRES or BOD resets.

Hibernate wakeup resets can be detected by examining the TOKEN field in the PWR_HIBERNATE register as described previously. However, Hibernate wakeup resets that occur as a result of an XRES cannot be detected. The reset causes in the RES_CAUSE and RES_CAUSE2 registers are shown in Table 172.

After identifying and evaluating the reset cause, it is necessary to clear the RESET_CAUSE and RESET_CAUSE2 registers. This procedure is required to capture the next reset.

Table 172 Reset cause bits to detect reset source

Register [bit pos]	Bitfield	Number of bits	Description
RES_CAUSE[0]	RESET_WDT	1	A hardware WDT reset has occurred since the last power cycle.
RES_CAUSE[1]	RESET_ACT_FAULT	1	Fault logging system requested a reset from its Active logic.
RES_CAUSE[3]	RESET_TC_DBGRESET	1	Test controller or debugger asserted reset. Only resets debug domain.
RES_CAUSE[4]	RESET_SOFT	1	CPU requested a system reset through its TRIGGER_SOFT. This can be done via a debugger probe or from firmware.
RES_CAUSE[5]	RESET_MCWDT0	1	Multi-counter WDT reset has occurred since the last power cycle.
RES_CAUSE2[6:0]	RESET_CSV_HF	7	Clock supervision logic requested a reset due to loss or frequency violation of a high-frequency clock. Each bit index K corresponds to a HFCLK<K>.
RES_CAUSE2[16]	RESET_CSV_REF	1	Clock supervision logic requested a reset due to loss or frequency violation of the reference clock source that is used to monitor the other HF clock sources.

For more information, see the RES_CAUSE and RES_CAUSE2 registers in the register's reference manual.

If these methods cannot detect the cause of the reset, then it may be an unrecorded and a non-retention reset. These resets cannot be distinguished using on-chip resources.

22.4 Registers summary

Table 173 Reset system registers list

Register	Description
RES_CAUSE	Reset cause observation register
RES_CAUSE2	Reset cause observation register 2
PWR_HIBERNATE	Hibernate power mode control register. Contains a TOKEN field that can be used to detect the Hibernate wakeup reset
RES_SOFT_CTL	Soft reset trigger register
RES_PXRES_CTL	Control register for programmable external reset (PXRES).

23 SVGS (Supply Voltage Glitch Sensor)

23.1 Introduction

The Supply Voltage Glitch Sensor (SVGS) is a peripheral that detects supply-voltage manipulations by attackers attempting to disrupt device operation. The SVGS module provides a robust and reliable way to monitor the supply voltage and detect any potential glitches or attacks. The block implements high-speed comparators that supervise the microcontroller's core supply (VCCD) and provide protection against disturbances on operational supply (VDDD), ensuring immunity of status retention against noise events up to 2 µs.

The voltage glitch sensors are widely deployed in power systems, electronic devices, and industrial control systems to detect abnormal voltage transients. Upon detecting a glitch, the sensor can trigger alarms or initiate corrective actions to protect the system from potential damage.

23.2 Features

The SVGS module consists of four continuous-time comparators that monitor the supply voltage (VCCD) and detect any overshoot, undershoot, noise, or other anomalies. The module provides a range of features, including:

- VCCD overshoot monitor with a configurable threshold
- VCCD undershoot monitor with a configurable threshold, using a filtered VCCD signal
- VCCD noise monitor on VCCD with a configurable threshold
- VDDD undershoot monitor, detecting voltage levels below the brownout threshold at HV supply under which comparators operate

The application software can enable/disable the block and set up interrupts (svgs_interrupt_IRQ - IRQ141) for notification through the registers, allowing it to decide the action based on the reported event.

The SVGS operates only in Active mode and is disabled in other than Active mode to save power.

23.3 Architecture

The SVGS module consists of four comparators (with independent enable, independent mask):

- UVC (Under-Voltage Comparison) function: detects undershoot events of VCCD
- OVC (Over-Voltage Comparison) function: detects overshoot events of VCCD
- NVC (Noise Voltage Comparison) function: compares noise amplitudes of VCCD with respect to its defined noise threshold
- UVFC (Under-Voltage Comparison with Filtered supply) function: detects undershoot events of filtered VCCD

Each comparator has a configurable reference voltage and can be enabled or disabled independently. And it can operate in half power (lower power, slower detect) or full power (higher power, faster detect).

The SVGS is intended for deployment by Boot ROM and the Secure application on CM33-Main CPU, and should be configured as secure-only accessible.

***Note**: It is recommended to reset (reboot) the SVGS block after a detected event for reliable continued operation.*

23.4 Functional description

The SVGS_MAIN register block contains control, status, interrupt, and configuration registers. Comparator trip/trim values are stored in SFLASH, from where BootROM loads and applies them during startup.

The application firmware may enable/disable the SVGS module and configure fault or interrupt handling through svgs_interrupt_IRQ (IRQ141).

The SVGS module has several registers that control its operation and provide status information:

The SVGS_CTL register controls the operation of the SVGS module.

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_CTL[3:0]	ENABLE	RW	"0101" = Disable the SVGS, all other values will enable SVGS. (Note: It will be disabled during DEEPSLEEP for power savings.)
SVGS_CTL[8]	FAULT_EN	RW	0 = Disable Fault generation 1 = A fault will be generated to the CPUSS fault handler for any enabled comparator. (Note: This fault will not be registered if the reset is not masked for the triggered comparator. Reset shall take precedence.)

The SVGS_STATUS register indicates captured events and internal state of the block.

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_STATUS[0]	UVC_CAPTURED	R	1 = UVC (Under Voltage Comparison function) occurred since the last reset
SVGS_STATUS[1]	OVC_CAPTURED	R	1 = OVC (Over Voltage Comparison function) occurred since the last reset
SVGS_STATUS[2]	UVFC_CAPTURED	R	1 = UVFC (Under Voltage (Filtered) Comparison function) occurred since the last reset
SVGS_STATUS[3]	NVC_CAPTURED	R	1 = NVC (Noise Voltage Comparison function) occurred since the last reset
SVGS_STATUS[4]	HVC_CAPTURED	R	1 = HVC (High Voltage Comparison function) occurred since the last reset
SVGS_STATUS[7:5]	-	-	Reserved
SVGS_STATUS[8]	EN_SEQ_ACTIVE	R	1 = HW enable sequence is active. SW should check that this sequence has completed, meaning SVGS is fully enabled before security-critical operations.
SVGS_STATUS[9]	SVGS_ENABLED	R	1 = SVGS block enable sequence is complete, and SVGS is functioning per loaded settings. 0 = Block is disabled, or in the process of enable sequence.

SVGS_CMP_CFG_RO is a read-only register that reports the enable, power, and mask configuration currently applied to all comparators.

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_CMP_CFG_RO[0]	UVC_EN	R	1 = Enable UVC (Under Voltage Comparison function)

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_CMP_CFG_RO[1]	OVC_EN	R	1 = Enable OVC (Over Voltage Comparison function)
SVGS_CMP_CFG_RO[2]	UVFC_EN	R	1 = Enable UVFC (Under Voltage (Filtered) Comparison function)
SVGS_CMP_CFG_RO[3]	NVC_EN	R	1 = Enable NVC (Noise Voltage Comparison function)
SVGS_CMP_CFG_RO[8]	UVC_PWR	R	1 = UVC in full power, else half power mode
SVGS_CMP_CFG_RO[9]	OVC_PWR	R	1 = OVC in full power, else half power mode
SVGS_CMP_CFG_RO[10]	UVFC_PWR	R	1 = UVFC in full power, else half power mode
SVGS_CMP_CFG_RO[11]	NVC_PWR	R	1 = NVC in full power, else half power mode
SVGS_CMP_CFG_RO[16]	UVC_MASK	R	1 = UVC masked from affecting svgs_glitch_event
SVGS_CMP_CFG_RO[17]	OVC_MASK	R	1 = OVC masked from affecting svgs_glitch_event
SVGS_CMP_CFG_RO[18]	UVFC_MASK	R	1 = UVFC masked from affecting svgs_glitch_event
SVGS_CMP_CFG_RO[19]	NVC_MASK	R	1 = NVC masked from affecting svgs_glitch_event
SVGS_CMP_CFG_RO[20]	HVC_MASK	R	1 = HVC masked from affecting svgs_glitch_event

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_CMP_CFG_RO[27:24]	SVGS_UVFC_TC	R	The time constant of the UVFC functionality. Each increment in the configured setting sets the increment (logarithmically equi-spaced) in the time constant 0'h sets time constant 20 ns 1'h sets time constant 24.4 ns 2'h sets time constant 29.8 ns 3'h sets time constant 36.4 ns 4'h sets time constant 44.4 ns 5'h sets time constant 54.3 ns 6'h sets time constant 66.2 ns 7'h sets time constant 81 ns 8'h sets time constant 98.8 ns A'h sets time constant 147.3 ns B'h sets time constant 179.9 ns C'h sets time constant 219.7 ns D'h sets time constant 268.3 ns E'h sets time constant 327.5 ns F'h sets time constant 400 ns

The SVGS_CMP_TRIP_RO register defines trip thresholds for UVC, OVC, UVFC, and NVC comparators, with voltage steps and ranges

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_CMP_TRIP_RO[5:0]	SVGS_UVC_TRIP	R	Configured threshold settings of UVC functionality in steps of 6.2 mV. 000000 = 760.00 mV 000001 = 766.2 mV …. 111111 = 1150 mV
SVGS_CMP_TRIP_RO[9:8]	SVGS_OVC_TRIP	R	Configured threshold settings of OVC functionality in steps of 25 mV. 00 = 1250 mV 01 = 1275 mV 10 = 1300 mV 11 = 1325 mV
SVGS_CMP_TRIP_RO[21:16]	SVGS_UVFC_TRIP	R	Configured threshold settings of UVFC functionality in steps of 6.2 mV. 000000 = 760.00 mV 000001 = 766.2 mV …. 111111 = 1150 mV

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_CMP_TRIP_RO[27:24]	SVGS_NVC_TRIP	R	Configured threshold settings of NVC functionality in steps of 10 mV. 0000 = 30 mV 0001 = 40 mV …. 1111 = 180 mV

The SVGS_INTR register provides interrupt status information.

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_INTR[0]	UVC_GLITCH	RW1C	UVC Glitch detected interrupt
SVGS_INTR[1]	OVC_GLITCH	RW1C	OVC Glitch detected interrupt
SVGS_INTR[2]	UVFC_GLITCH	RW1C	UVFC Glitch detected interrupt
SVGS_INTR[3]	NVC_GLITCH	RW1C	NVC Glitch detected interrupt
SVGS_INTR[4]	HVC_GLITCH	RW1C	HVC Glitch detected interrupt

***Note**: A field marked RW1C can be read normally, but on write:*

- *Writing 1 → clears the bit (hardware sets it to 0)*
- *Writing 0 → has no effect*

The SVGS block must be reset (disabled and enabled) to clear the interrupt. Otherwise, the event status remains set, and the interrupt will continue to trigger

The SVGS_INTR_SET register sets the interrupt status.

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_INTR_SET[0]	UVC_GLITCH	RW1S	Set interrupt for UVC_GLITCH interrupt.
SVGS_INTR_SET[1]	OVC_GLITCH	RW1S	Set interrupt for OVC_GLITCH interrupt.
SVGS_INTR_SET[2]	UVFC_GLITCH	RW1S	Set interrupt for UVFC_GLITCH interrupt.
SVGS_INTR_SET[3]	NVC_GLITCH	RW1S	Set interrupt for NVC_GLITCH interrupt.
SVGS_INTR_SET[4]	HVC_GLITCH	RW1S	Set interrupt for HVC_GLITCH interrupt.

***Note**: A field marked RW1S can be read normally, but on write:*

- *Writing 1 → sets the bit (hardware writes 1)*
- *Writing 0 → has no effect*

The SVGS_INTR_MASK register masks the interrupts.

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_INTR_MASK[0]	UVC_GLITCH	RW	Mask for UVC_GLITCH interrupt.
SVGS_INTR_MASK[1]	OVC_GLITCH	RW	Mask for OVC_GLITCH interrupt.
SVGS_INTR_MASK[2]	UVFC_GLITCH	RW	Mask for UVFC_GLITCH interrupt.
SVGS_INTR_MASK[3]	NVC_GLITCH	RW	Mask for NVC_GLITCH interrupt.
SVGS_INTR_MASK[4]	HVC_GLITCH	RW	Mask for HVC_GLITCH interrupt.

The SVGS_INTR_MASKED register provides the masked interrupt status.

Register [Bit_Pos]	Bit_Name	Access	Description
SVGS_INTR_MASKED[0]	UVC_GLITCH	R	Logical and of corresponding request and mask bits.
SVGS_INTR_MASKED[1]	OVC_GLITCH	R	Logical and of corresponding request and mask bits.
SVGS_INTR_MASKED[2]	UVFC_GLITCH	R	Logical and of corresponding request and mask bits.
SVGS_INTR_MASKED[3]	NVC_GLITCH	R	Logical and of corresponding request and mask bits.
SVGS_INTR_MASKED[4]	HVC_GLITCH	R	Logical and of corresponding request and mask bits.

Initialization Sequence

During the bootup and initialization, the SVGS trims are loaded by the BootROM. The user application can make the SVGS block operational by enabling it.

- Enable the SVGS module by writing to the SVGS_CTL register
- Wait for the SVGS module to become enabled, indicated by the SVGS_STATUS register being set to 0x200
- Check the SVGS_STATUS register to ensure that the SVGS module is enabled and functioning correctly
- SVGS_STATUS.EN_SEQ_ACTIVE is 1 while the SVGS module is performing an initialization. When SVGS is enabled, it sets SVGS_ENABLED=1 and EN_SEQ_ACTIVE=0
- Beginning normal operation once SVGS is fully active

Interrupt Handling

To handle interrupts generated by the SVGS module, follow these steps:

- Enable the SVGS interrupt (svgs_interrupt_IRQ - IRQ141) by setting the corresponding bit in the SVGS_INTR_MASK register

- Set up an interrupt handler to handle the SVGS interrupt
- In the interrupt handler, check the SVGS_INTR register to determine which comparator triggered the interrupt
- Take the necessary action to handle the interrupt, such as reset/reboot the system or taking corrective action

Fault Handling

To handle faults generated by the SVGS module, follow these steps:

- Enable fault generation by setting the FAULT_EN bit in the SVGS_CTL register
- Set up a fault handler to handle the SVGS fault
- In the fault handler, check the SVGS_STATUS register to determine which comparator triggered the fault
- Take the necessary corrective action to handle the fault

Independent of reset/fault/IRQ, event status is latched in SVGS_STATUS and SVGS_INTR/INTR_MASKED.

23.5 Registers summary

Register list

The SVGS protected run time registers (non-privileged software accessible)

Name	Access	Function
SVGS_CTL	RW	SVGS Control Register
SVGS_STATUS	R	SVGS Status Register
SVGS_CMP_CFG_RO	R	SVGS Comparator Configuration
SVGS_CMP_TRIP_RO	R	SVGS Comparator trip-point configuration
SVGS_TRIM_RO	R	SVGS Trim Register
SVGS_INTR	RW1C	SVGS Interrupt Register
SVGS_INTR_SET	RW1S	SVGS Interrupt Set Register
SVGS_INTR_MASK	RW	SVGS Interrupt Mask Register
SVGS_INTR_MASKED	R	SVGS Interrupt Masked Register

24 I/O subsystem

This section explains the I/O subsystem of PSC3M/P6 MCU, its features, architecture, operating modes, and interrupts. The I/O subsystem provides the interface between the CPU core and peripheral components to the outside world. The capabilities of thePSC3M/P6 MCU enable efficient signal routing to most pins, simplifying circuit design and board layout. The GPIO pins in this MCU family are grouped into ports; a port can have a maximum of eight GPIO pins.

This section includes the following subsections:

- Features
- Architecture
- Functional description
- Programmable digital (Smart I/O)
- Registers summary

24.1 Features

The PSC3M/P6 MCU GPIOs have the following features:

- Drive modes
  - Strong drive
  - Resistive pull-up
  - Resistive pull-down
  - Open drain, drives low
  - Open drain, drives high
  - Resistive pull-up or pull-down
  - High impedance
- Four drive strength options
- Input threshold selection (CMOS or LVTTL)
- Individual control of input and output disables
- Hold mode for latching the previous state, which retains the I/O state during Deep Sleep and Hibernate modes
- Selectable slew rates for dV/dt-related noise control
- Support for one I3C signal outputs with GPIOs featuring overvoltage tolerance (OVT) functionality
- Up to eight GPIOs multiplexed with analog inputs to provide additional analog input connections

24.2 Architecture

The PSC3M/P6 MCU is equipped with analog and digital peripherals. Figure 81 shows an overview of the routing between the peripherals and pins.

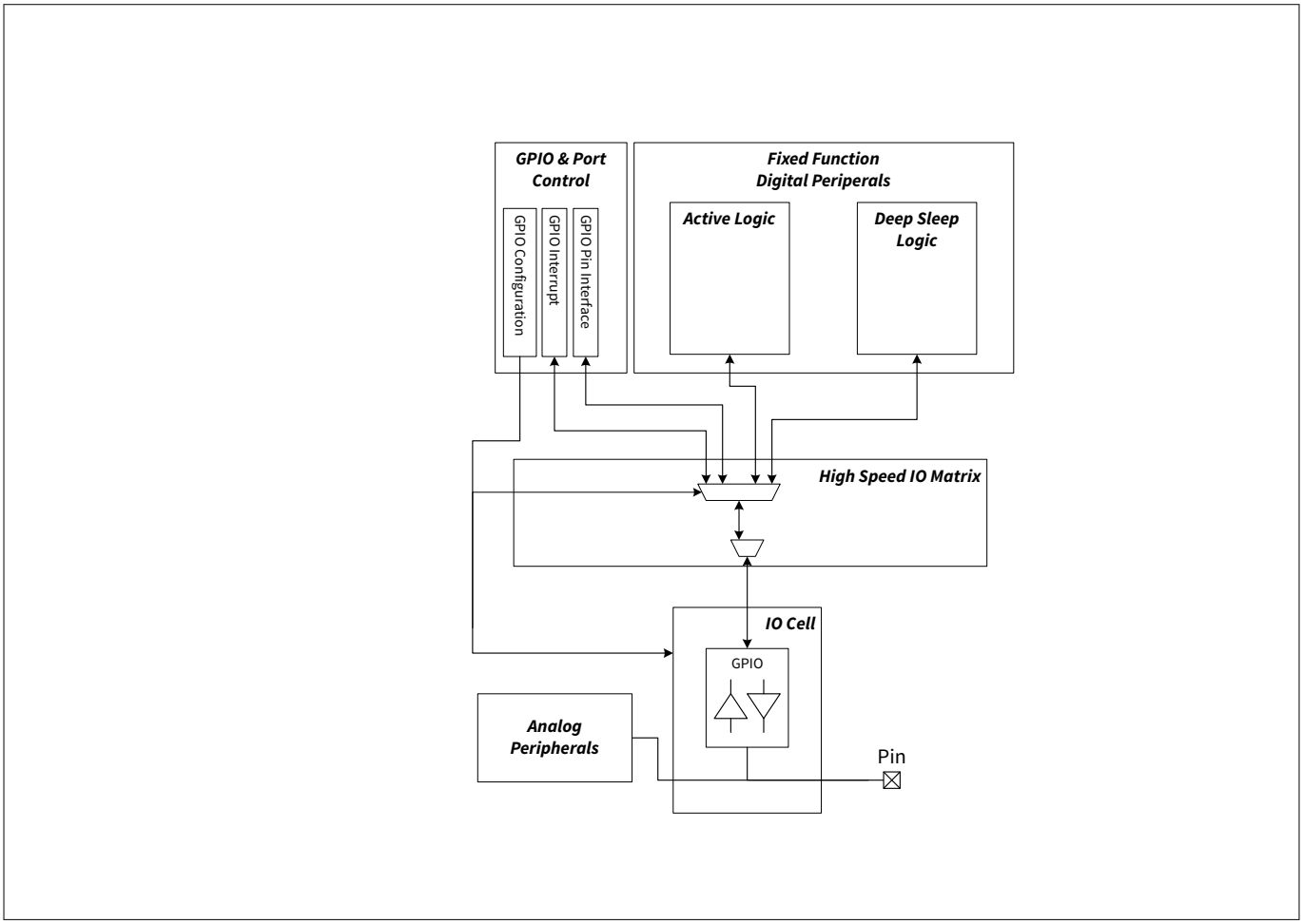


Figure 81 GPIO interface overview

GPIO pins are connected to I/O cells. These cells are equipped with an input buffer for the digital input, providing high-input impedance and a driver for the digital output signals. The MCU also includes up to 16 dedicated analog pins, as described in the High-performance programmable analog subsystem (HPPASS) section.

The digital peripherals connect to the I/O cells via the high-speed I/O matrix (HSIOM). The HSIOM for each pin contains multiplexers to connect between the selected peripheral and the pin. Additionally, HSIOM bridges the connection between the digital system interconnect (DSI) and the pins.

24.2.1 I/O cell architecture

Figure 82 shows the I/O cell architecture present in every GPIO cell. It comprises an input buffer and an output driver that connect to the HSIOM multiplexers for digital input and output signals. Analog peripherals connect directly to the pin for point-to-point connections.

The GPIO component provides the I/O cell configuration information through the registers. These registers are retained in Deep Sleep power mode, but are reset to their default value in Hibernate power mode. To allow for Hibernate Interrupt functionality, the I/O cells hold or freeze their configuration information when entering either Deep Sleep or Hibernate power mode. As a result, the configuration signals can be routed in the Active power domain.

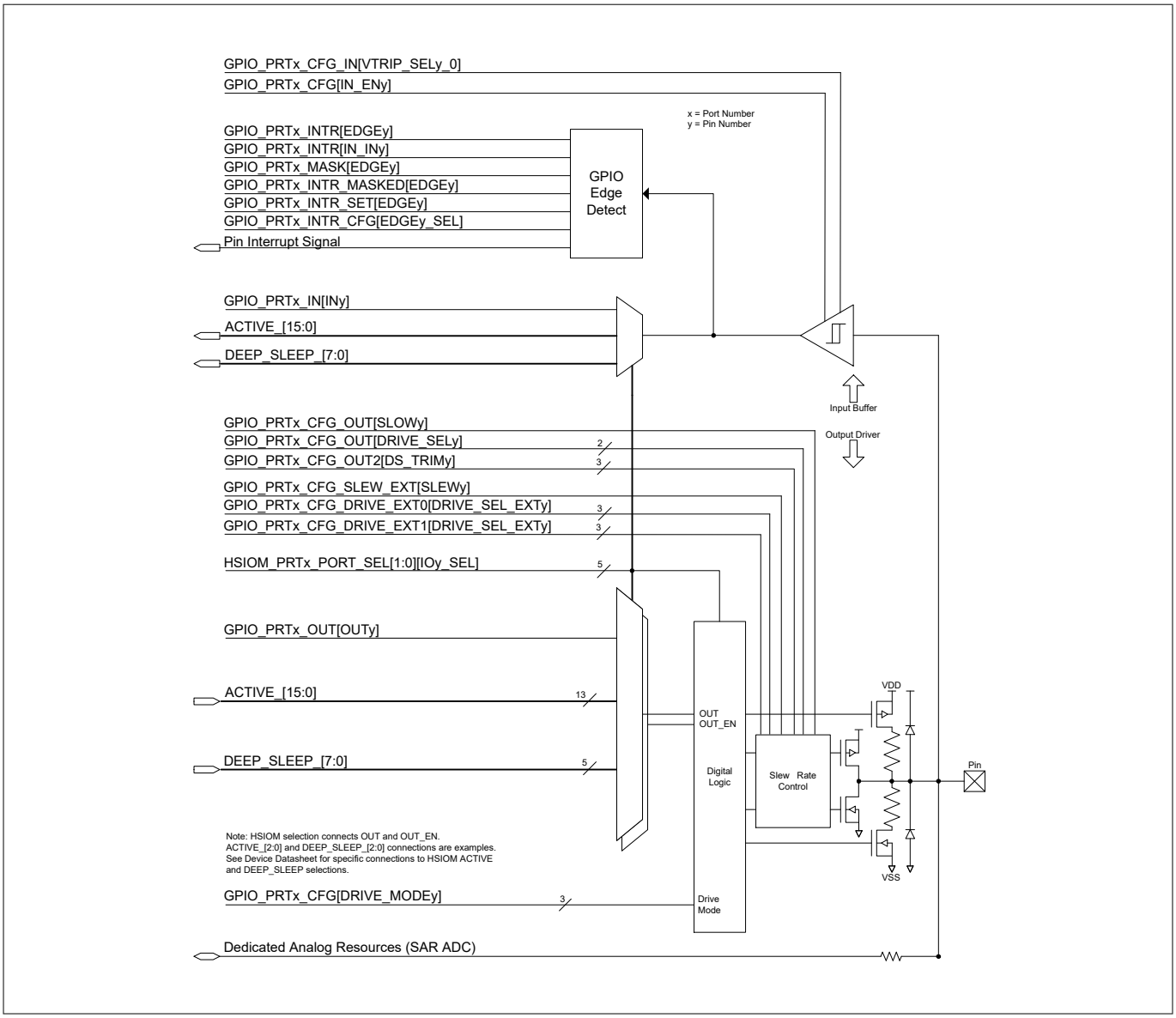


Figure 82 GPIO cell architecture

If the HSIOM makes a functional connection to an I/O cell, the GPIO provides the configuration information. If the HSIOM makes a test connection (scan, parallel test mode or JTAG) to an I/O cell, the GPIO configuration information is ignored. And, the HSIOM provides the required configuration information.

I/O cell configuration includes information such as drive mode (pull-up/pull-down) and drive strength. Configuration information may be for a specific I/O pad: drive mode, drive strength, fast versus slow slew control transitioning, input buffer mode, and so on.

The I/Os in an I/O port are accessible by software to provide controllability of the I/O output signals and observability of the I/O input signals. Combined, controllability, and observability provide software bit banging functionality.

The GPIO data input and data output/data output enable signals for I/O cells are on the HSIOM functional connections. The specific connection is under the control of the HSIOM register fields.

24.2.2 HSIO to map alternate functions on GPIO

The high-speed I/O matrix (HSIOM) is a set of high-speed multiplexers that route internal CPU and peripheral signals to and from GPIOs. HSIOM allows GPIOs to be shared with multiple functions and multiplexes the pin

connection to a user-selected peripheral. The HSIOM_PRTx_PORT_SEL[1:0] registers allow a single selection from up to 32 different connections to each pin as listed in Table 174.

Table 174 HSIOM connections

Alternate function 5-bit[4:0]	Name	Digital driver signal source		Digital signal destination	Description
		OUT	OUT_EN		
0	GPIO	OUT register	1	IN register	GPIO_PRTx_OUT register controls “out”
1	Reserved	-	-	-	-
2	Reserved	-	-	-	-
3	Reserved	-	-	-	-
4	Reserved	-	-	-	-
5	Reserved	-	-	-	-
6	Reserved	-	-	-	-
7	Reserved	-	-	-	-
8	ACT_0	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 0 - See the datasheet for specific pin connectivity
9	ACT_1	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 1 - See the datasheet for specific pin connectivity
10	ACT_2	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 2 - See the datasheet for specific pin connectivity
11	ACT_3	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 3 - See the datasheet for specific pin connectivity

(table continues...)

Table 174 (continued) HSIOM connections

Alternate function 5-bit[4:0]	Name	Digital driver signal source		Digital signal destination	Description
		OUT	OUT_EN		
12	DS_0	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	Deep Sleep functionality 0 - See the datasheet for specific pin connectivity
13	DS_1	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	Deep Sleep functionality 1 - See the datasheet for specific pin connectivity
14	DS_2	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	Deep Sleep functionality 2 - See the datasheet for specific pin connectivity
15	DS_3	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	Deep Sleep functionality 3 - See the datasheet for specific pin connectivity
16	ACT_4	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 4 - See the datasheet for specific pin connectivity
17	ACT_5	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 5 - See the datasheet for specific pin connectivity
18	ACT_6	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 6- See the datasheet for specific pin connectivity

(table continues...)

Table 174 (continued) HSIOM connections

Alternate function 5-bit[4:0]	Name	Digital driver signal source		Digital signal destination	Description
		OUT	OUT_EN		
19	ACT_7	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 7 - See the datasheet for specific pin connectivity
20	ACT_8	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 8 - See the datasheet for specific pin connectivity
21	ACT_9	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 9 - See the datasheet for specific pin connectivity
22	ACT_10	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 10 - See the datasheet for specific pin connectivity
23	ACT_11	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 11 - See the datasheet for specific pin connectivity
24	ACT_12	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 12 - See the datasheet for specific pin connectivity
25	ACT_13	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 13 - See the datasheet for specific pin connectivity

(table continues...)

Table 174 (continued) HSIOM connections

Alternate function 5-bit[4:0]	Name	Digital driver signal source		Digital signal destination	Description
		OUT	OUT_EN		
26	ACT_14	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 14 - See the datasheet for specific pin connectivity
27	ACT_15	Active Source OUT	Active Source OUT_EN	Active Source IN	Active functionality 15 - See the datasheet for specific pin connectivity
28	DS_4	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	Deep Sleep functionality 4 - See the datasheet for specific pin connectivity
29	DS_5	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	Deep Sleep functionality 5 - See the datasheet for specific pin connectivity
30	DS_6	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	Deep Sleep functionality 6 - See the datasheet for specific pin connectivity
31	DS_7	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	Deep Sleep functionality 7 - See the datasheet for specific pin connectivity

***Note**: The Active and Deep Sleep sources are pin-dependent. See the "Pinouts" section of the device datasheet for more details on the features supported by each pin.*

24.2.3 I/O state on power-up

At power-up, all GPIOs are in a high-impedance analog state, and the input buffers are disabled. During run-time, you can configure GPIOs by writing to the associated registers. However, the pins that support the debug

access port (DAP) connections (SWD lines) are always enabled as SWD lines during power-up. If these pins are left floating, there is a risk of crowbar current as the DAP connection does not provide pull-up or pull-down resistors. To avoid this, you can disable or reconfigure the DAP connection for general-purpose use through the HSIOM, but only after the device has booted and started executing code.

24.2.4 Behavior in low-power modes

Table 175 shows the status of GPIOs in low-power modes.

Table 175 Behavior in low-power modes

Low-power mode	State
Sleep	Standard GPIO, GPIO-OVT, and SIO pins are active and can be driven by most peripherals such as TCPWMs, and SCBs, which can operate in Sleep mode. Inputs buffers are active; so an interrupt on any I/O can be used to wake the CPU.
Deep Sleep	GPIO, GPIO-OVT, and SIO pins, connected to Deep Sleep domain peripherals, are functional. All other pins maintain the last output driver state and configuration. Pin interrupts are functional on all I/Os and can be used to wake the device.
Hibernate	Pin output states and configuration are latched and remain in the frozen state. Pin interrupts are functional only on select IOs and can be used to wake the device. See the device datasheet for specific hibernate pin connectivity.

24.3 Functional description

24.3.1 Digital input buffer

The digital input buffer provides a high-impedance buffer for the external digital input. The buffer is enabled or disabled by the `GPIO_PRTx_CFG.IN_ENy` bit (where 'x' is the port number and 'y' is the pin number).

The input buffer is connected to the HSIOM for routing to the CPU Port registers and selected peripherals. Writing to the HSIOM Port Select register (`HSIOM_PRTx_PORT_SEL`) selects the pin connection. See the device datasheet for the specific connections available for each pin. A port pin can be used as an input and output at the same time. If a pin is only connected to an analog signal, the input buffer should be disabled to avoid crowbar currents. Each pin's input buffer trip point and hysteresis are configurable for the following modes:

- CMOS + I2C
- TTL

Use the `GPIO_PRTx_CFG_IN.VTRIP_SELy_0` bit to select CMOS and TTL buffer modes. The trip levels of CMOS and TTL are shown in Figure 83.

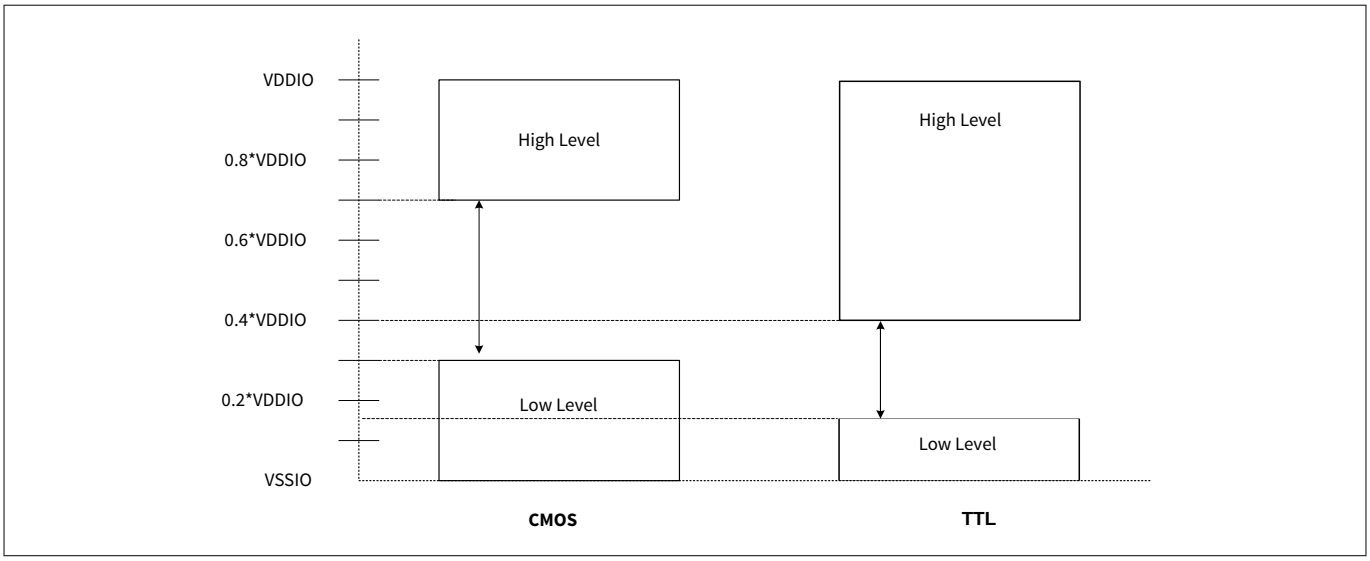


Figure 83 Input buffer mode's tripping levels

24.3.2 Digital output driver

The digital output driver drives the pins. It consists of circuitry to implement different drive modes and slew rate control for the digital output signals. The HSIOM selects the control source for the output driver. The two primary types of control sources are Port Configuration registers and Fixed Function Digital peripherals. A particular HSIOM connection is selected by writing to the HSIOM Port Select register (`HSIOM_PRTx_PORT_SEL`).

Each GPIO pin has ESD diodes to clamp the pin voltage to the I/O supply source. Ensure that the voltage at the pin does not exceed the I/O supply voltage VDDIO/VDDD/VDDA or drop below VSSIO/VSSD/VSSA. For the absolute maximum and minimum GPIO voltage, see the device datasheet.

The Output Data register (`GPIO_PRTx_OUT`) associated with the output pin enables or disables the digital output driver in the hardware. Peripherals other than the GPIO port, directly control both the output and the output enable signals of the output buffer.

24.3.2.1 GPIO-OVT pins

Select device pins are designed to be over voltage tolerant (OVT) and can be used to interface with busses or signals that may exceed the VDDIO supply of the pin or the entire device, or when the VDDIO supply may not always be present. GPIO-OVT pins are similar to regular GPIOs but have the added feature of being overvoltage tolerant. These pins include hardware that compares the voltage on the pin to the VDDIO voltage. If the voltage on the GPIO-OVT pin exceeds VDDIO, the output driver is disabled, and the pin driver is tri-stated. This results in negligible current sinking at the pin. See the device datasheet to know which pins are over voltage tolerant.

***Note**: In over voltage conditions, the input buffer data is not valid if the external source's specification of VOH and VOL do not match the trip points of the input buffer defined by the current VDDIO voltage.*

24.3.2.2 Drive strength

The drive strength of an I/O specifies how much current I/O pin can drive and sink while maintaining the minimum VOH and VOL levels. When the output is configured for fast slew rate, the port drive strength can be set to one of four levels using the DRIVE_SEL field of the port output configuration register (`GPIO_PRTx_CFG_OUT`). The drive strength field determines the active portion of the output driver used and can affect the slew rate of output signals. Drive strength options are full drive strength (default), one-half strength, one-quarter strength,

and one-eighth strength. Drive strength must be set to full drive strength when the slow slew rate bit (SLOW) is set.

The I/O port provides 2 advanced registers called as Port output Drive Select Extension Register (GPIO_PRTx_CFG_DRIVE_EXT[0-1]) to individually configure the drive strength of each pin of the respective port.

Table 176 Drive select for GPIO_STD

Drive select	DRIVE_SEL[0:1]	Description
DRIVE_SEL_THREE	11	1/8 drive strength: GPIO drives current at one-eighth of its maximum rated specification.
DRIVE_SEL_TWO	10	1/4 drive strength: GPIO drives current at one-quarter of its maximum rated specification.
DRIVE_SEL_ONE	01	1/2 drive strength: GPIO drives current at one-half of its maximum rated specification
DRIVE_SEL_ZERO	00	Full drive strength: GPIO drives current at its maximum rated specification.

24.3.2.3 Slew rate control

Some GPIO pins have fast and slow output slew rate options for the strong drivers configured using the SLOW bit of the Port Output Configuration register (GPIO_PRTx_CFG_SLEW_EXT). By default, this bit is cleared and the port works in fast slew mode. This bit can be set if a slow slew rate is required. Slower slew rate results in reduced EMI and crosstalk and is recommended for low-frequency signals or signals without strict timing constraints.

When configured for a fast slew rate, the drive strength can be set to one of four levels defined in Drive strength.

24.3.2.4 Drive mode

Each I/O is individually configurable to one of eight drive modes by the DRIVE_MODE[7:0] field of the Port Configuration register (GPIO_PRTx_CFG).

Table 177 lists the drive modes using GPIO_PRTx_CFG register. It provides a comprehensive breakdown of the different modes in two distinct columns: GPIO functionality and fixed peripheral functionality.

***Note**: OUT_EN is not user-configurable; its value is set according to the pin mode. For example, in GPIO mode OUT_EN = 1. See Table 174*

Table 177 Drive mode settings

Drive mode	Value	GPIO port configuration register, AMUXBUS				Fixed-function digital peripheral			
		OUT_EN = 1		OUT_EN = 0		OUT_EN = 1		OUT_EN = 0	
		OUT = 1	OUT = 0	OUT = 1	OUT = 0	OUT = 1	OUT = 0	OUT = 1	OUT = 0
High impedance	0	High-Z	High-Z	High-Z	High-Z	High-Z	High-Z	High-Z	High-Z
Reserved	1	-	-	-	-	-	-	-	-
Resistive pull up	2	Weak 1	Strong 0	High-Z	High-Z	Strong 1	Strong 0	Weak 1	Weak 1
Resistive pull down	3	Strong 1	Weak 0	High-Z	High-Z	Strong 1	Strong 0	Weak 0	Weak 0
Open drain, drives low	4	High-Z	Strong 0	High-Z	High-Z	Strong 1	Strong 0	High-Z	High-Z
Open drain, drives high	5	Strong 1	High-Z	High-Z	High-Z	Strong 1	Strong 0	High-Z	High-Z
Strong	6	Strong 1	Strong 0	High-Z	High-Z	Strong 1	Strong 0	High-Z	High-Z
Resistive pull up or pull down	7	Weak 1	Weak 0	High-Z	High-Z	Strong 1	Strong 0	Weak 1	Weak 0

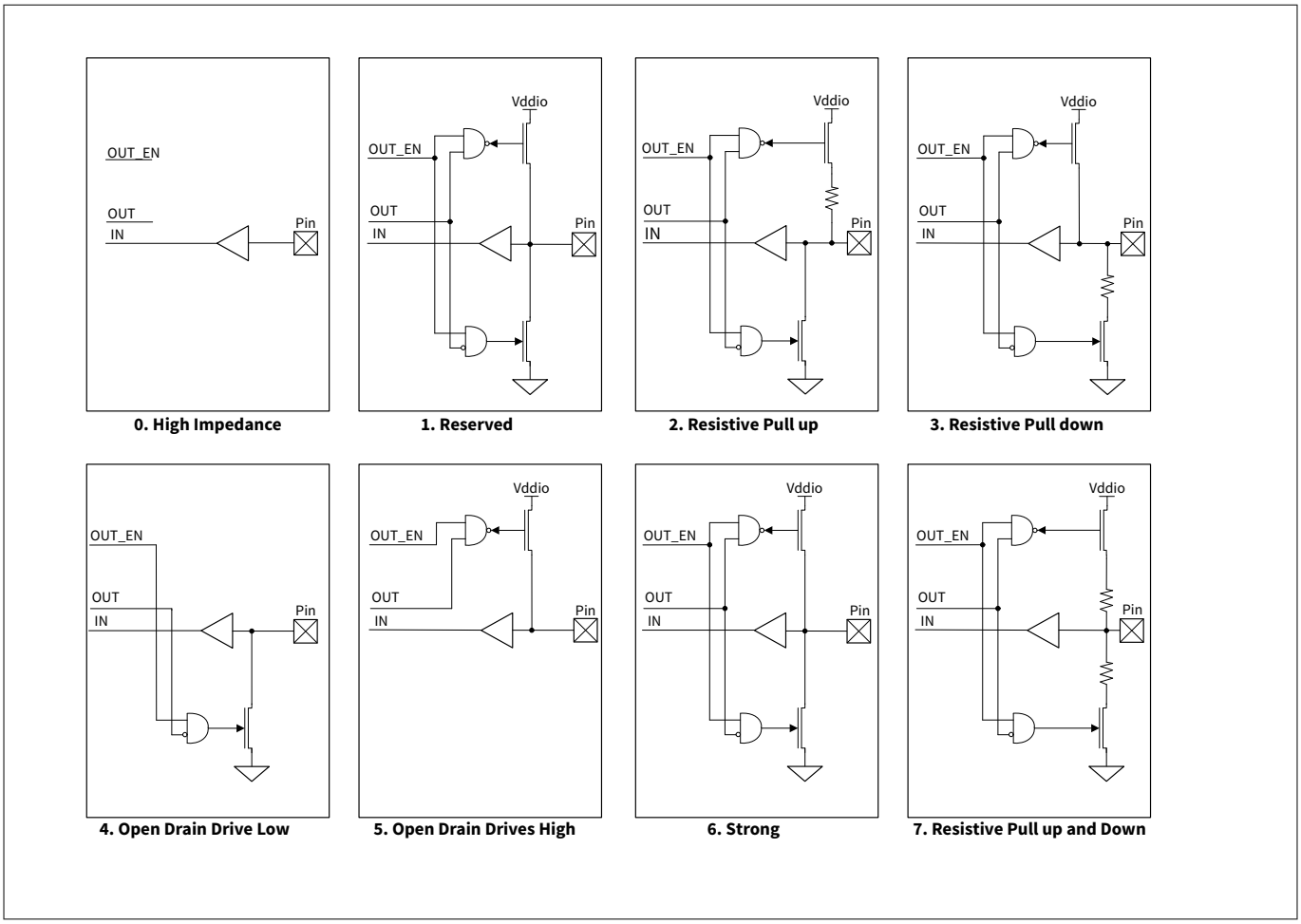


Figure 84 GPIO port, drive mode block diagrams

Figure 85 is an output driver diagram that shows the pin view for fixed-function-based peripherals in all available drive modes.

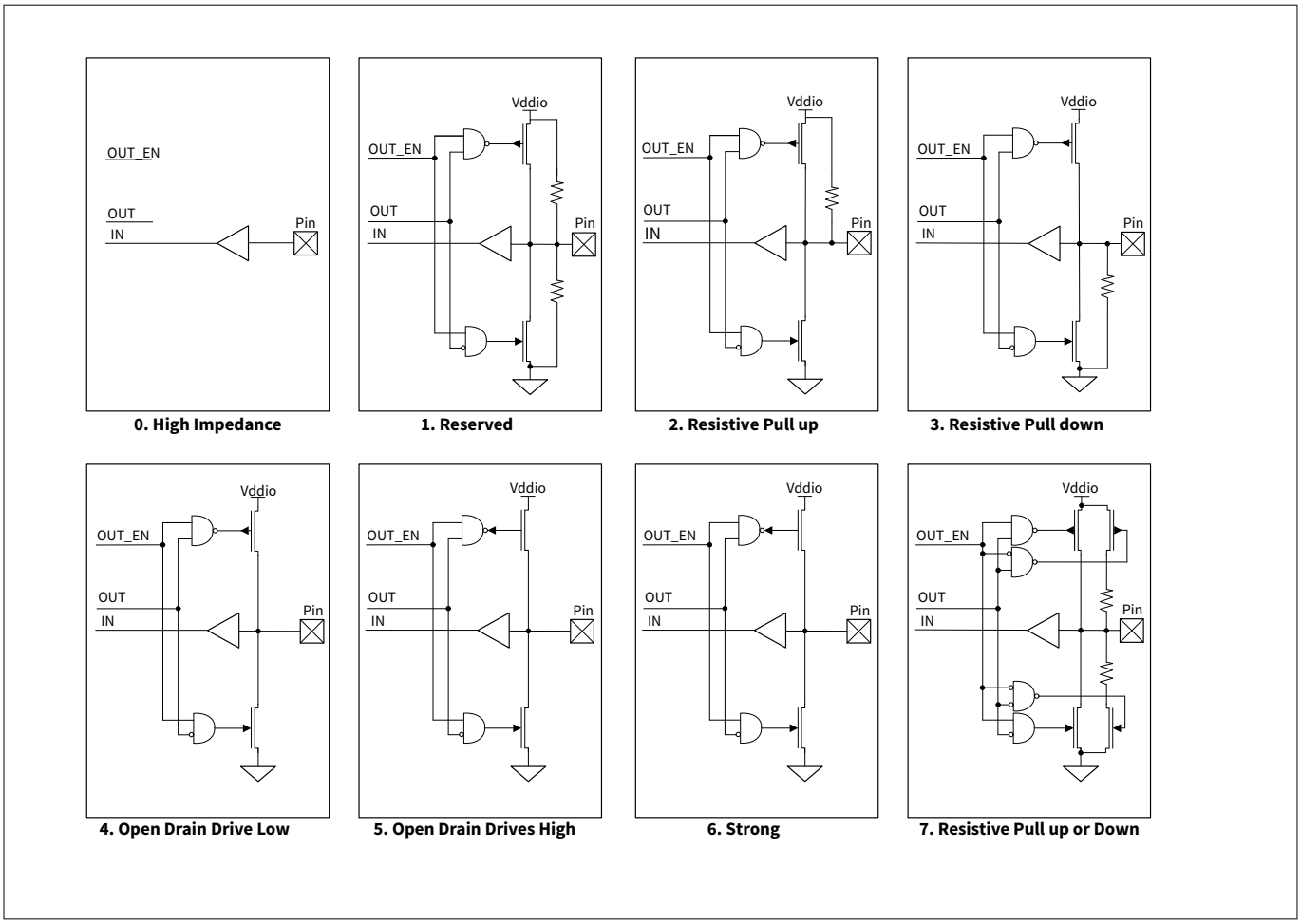


Figure 85 Fixed Function Peripheral I/O drive mode block diagrams

1. High-impedance(0)

   High-impedance mode is the standard high-impedance (High-Z) state recommended for analog and digital inputs. For digital signals, the input buffer is enabled; for analog signals, the input buffer is typically disabled to reduce crowbar current and leakage in low-power designs. To achieve the lowest device current, unused GPIOs must be configured to the high-impedance drive mode with input buffer disabled. High-impedance drive mode with input buffer disabled is also the default pin reset state
2. Resistive pull-up mode(2) and resistive pull-down mode(3)

   Resistive modes provide a series resistance in one of the data states and strong drive in the other. Pins can be used for either digital input or digital output in these modes. If resistive pull-up is required, a '1' must be written to that pin's data register bit. If resistive pull-down is required, a '0' must be written to that pin's data register. Interfacing mechanical switches is a common application of these drive modes. The resistive modes are also used to interface the MCU with open drain drive lines. Resistive pull-up is used when the input is open drain low and resistive pull-down is used when the input is open drain HIGH
3. Open drain drives low(4) and open drain drives high(5)

   Open drain modes provide high impedance in one of the data states and strong drive in the other. Pins are useful as digital inputs or outputs in these modes. Therefore, these modes are widely used in bidirectional digital communication. Open drain drive high mode is used when the signal is externally pulled down and open drain drive low is used when the signal is externally pulled high. A common application for the open drain drive low mode is driving I2C bus signal lines
4. Strong drive(6)

The strong drive mode is the standard digital output mode for pins; it provides a strong CMOS output drive in both high and low states. Strong drive mode pins should not be used as inputs under normal circumstances. This mode is often used for digital output signals or to drive external devices

5. Resistive pull-up or resistive pull-down(7)

   In the resistive pull-up or pull-down mode, the GPIO will have a series resistance in both logic 1 and logic 0 output states. The high data state is pulled up while the low data state is pulled down. This mode is useful when the pin is driven by other signals that may cause shorts

For I3C capable ports, GPIO_PRTx_CFG_OUT3 control register, extends the drive mode control of the GPIO. The corresponding GPIO_PRTx_CFG bits must be set to 0 for this register settings to take effect. This register controls the drive mode along with the three pull-up resistor options available for the I3C functionality. Refer to the device register technical reference manual for further details.

24.3.3 Firmware-controlled GPIO

For standard firmware-controlled GPIO using registers, the GPIO mode must be selected in the `HSIOM_PRTx_PORT_SEL` register.

The `GPIO_PRTx_OUT` register is used to read and write the output buffer state for GPIOs. A write operation to this register changes the GPIO's output driver state to the written value. A read operation reflects the output data written to this register and the resulting output driver state. It does not return the current logic level present on GPIO pins, which may be different. Using the `GPIO_PRTx_OUT` register, read-modify-write sequences can be safely performed on a port that has both input and output GPIOs.

In addition to the Data register, the following three registers are provided to set, clear, and invert the output data respectively on specific pins in a port without affecting other pins: `GPIO_PRTx_OUT_SET`, `GPIO_PRTx_OUT_CLR`, and `GPIO_PRTx_OUT_INV` . This avoids the need for read-modify- write operations in most use cases. Writing '1' to these register bit fields will set, clear, or invert the respective pin; writing '0' will have no effect on the pin state.

`GPIO_PRTx_IN` is the Port I/O Pad register that provides the actual logic level present on the GPIO pin when read. Writes to this register have no effect.

24.3.4 Interrupt

All port pins have the capability to generate interrupts. There are two routing possibilities for pin signals to generate interrupts, as shown in Figure 86.

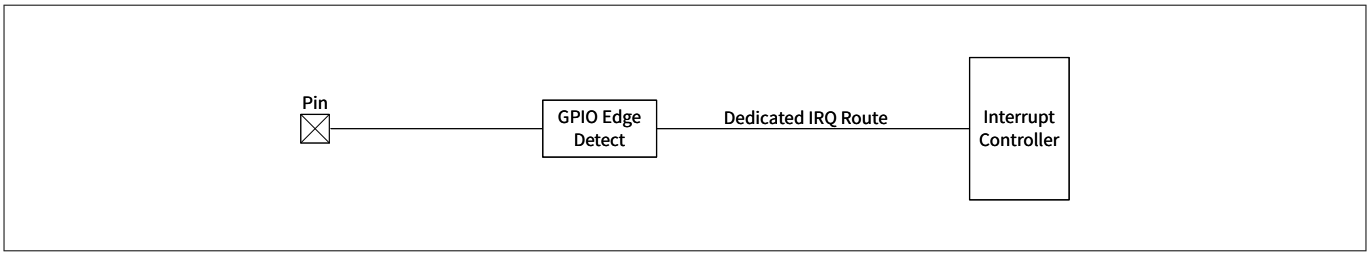


Figure 86 Interrupt signal routing

Pin signals through the ‘GPIO edge detect’ block with direct connection to the CPU interrupt controller. Interrupt generation is independent of HSIOM configuration, consider disabling it (`GPIO_PRTx_INTR_CFG.EDGEy_SEL` = 00) to avoid unwanted GPIO interrupt.Figure 87 shows the block diagram of the GPIO Edge Detect block.

Each GPIO pin interrupt can be activated either on GPIO pin rising or GPIO pin falling edge changes based on `GPIO_PRTx_INTR_CFG` register selection. Each of the up to eight GPIO pads in an I/O port has interrupt cause fields that are set to ‘1’ on a GPIO pin input signal rising and falling edge respectively (`GPIO_PRTx_INTR` register). The `GPIO_PRTx_INTR_MASK` register of the I/O port specifies which interrupt is propagated to its interrupt output. This propagated GPIO pin interrupt is a deep sleep functionality interrupt and this allows for a GPIO input signal

change to wake up the CPU from deep sleep power mode after combined with other given GPIO port “i” pin interrupts as shown in Figure 87 to form a port interrupt interrupts_gpio[i].

In addition, a register field specifies one specific I/O input signal that is routed to a 50-ns glitch filter (`GPIO_PRTx_INTR_CFG.FLT_SEL` register). The glitch filter output has a dedicated detection circuitry and has dedicated detection control fields (such as `GPIO_PRTx_INTR.FLT_EDGE`). Each I/O port has a dedicated interrupt associated to it (interrupts_gpio[i] for I/O port i). Figure 86 illustrates the interrupt functionality. Figure 87 shows the GPIO edge detect block architecture.

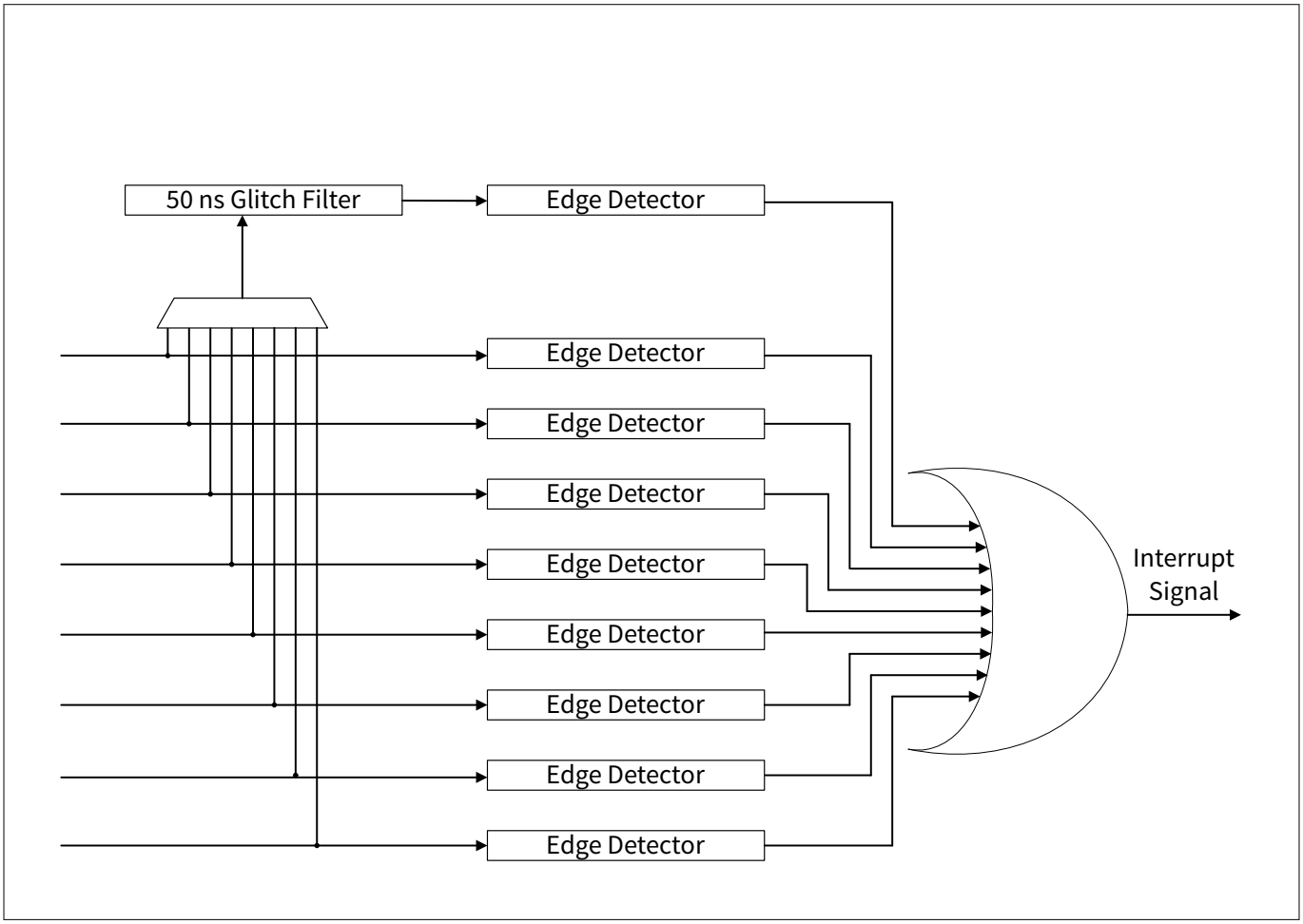


Figure 87 GPIO edge detect block architecture

The software ISR can read the 8+1 interrupt cause fields to determine the I/O or glitch filter signal(s) that caused the interrupt activation. The ISR needs to clear the interrupt cause fields to deactivate the interrupt.

An edge detector is present at each pin. It is capable of detecting rising edge, falling edge, and both edges without any reconfiguration. The edge detector is configured by writing into the `GPIO_PRTx_INTR_CFG.EDGEy_SEL` field, as shown in Table 178.

Table 178 Edge detector configuration

EDGE_SEL	Configuration
00	Interrupt is disabled
11	Interrupt on both edges
10	Interrupt on falling edge
01	Interrupt on rising edge

Writing '1' to the corresponding status bit clears the pin edge state. It is important to clear the edge state status bit; otherwise, an interrupt can occur repeatedly for a single trigger or respond only once for multiple triggers, which is explained later in this section. When the Port Interrupt Control Status register is read at the same time an edge is occurring on the corresponding port, it can result in the edge not being properly detected. Therefore, when using GPIO interrupts, it is recommended to read the Status register only inside the corresponding interrupt service routine and not in any other part of the code.

Firmware and the debug interface are able to trigger a hardware interrupt from any pin by setting the corresponding bit in the GPIO_PRTx_INTR_SET register.

In addition to the pins, each port provides a glitch filter connected to its own edge detector. This filter can be driven by one of the pins of a port. The selection of the driving pin is done by writing to the GPIO_PRTx_INTR_CFG.FLT_SEL field as shown in Table 179.

Table 179 Glitch filter input section

FLT_SEL	Selected pin
000	Pin 0 is selected
001	Pin 1 is selected
010	Pin 2 is selected
011	Pin 3 is selected
100	Pin 4 is selected
101	Pin 5 is selected
110	Pin 6 is selected
111	Pin 7 is selected

When a port pin edge occurs, it is required to know which pin caused the edge. This is done by reading the port Interrupt Status register (GPIO_PRTx_INTR). This register includes both the latched information on which pin detected an edge and the current pin status. This allows the CPU to read both information in a single read operation. This register has an additional use - to clear the latched edge state.

The GPIO_PRTx_INTR_MASK register enables forwarding of the GPIO_PRTx_INTR edge detect signal to the interrupt controller when a '1' is written to a pin's corresponding bit field. The GPIO_PRTx_INTR_MASKED register can then be read to determine the specific pin that generated the interrupt signal forwarded to the interrupt controller. The masked edge detector outputs of a port are then ORed together and routed to the interrupt controller (NVIC in the CPU subsystem). Thus, there is only one interrupt vector per port.

The masked and ORed edge detector block output is routed to the interrupt source multiplexer, which gives an option of level and rising edge detection. If the Level option is selected, an interrupt is triggered repeatedly as long as the port interrupt status register bit is set. If the rising edge detect option is selected, an interrupt is triggered only once if the Port Interrupt Status register is not cleared. Thus, it is important to clear the interrupt status bit if the edge detect block is used.

There is a dedicated interrupt vector for each port when the interrupt signal is routed through the fixed-function route.

All the port interrupt vectors are also ORed together into a single interrupt vector for use on devices with more ports than there are interrupt vectors available. To determine the port that triggered the interrupt, the GPIO_INTR_CAUSEx registers can be read. A '1' present in a bit location indicates that the corresponding port has a pending interrupt. The indicated GPIO_PRTx_INTR register can then be read to determine the pin source.

The GPIO_VDD_ACTIVE register provides the capability to read the state of the external power supplies. It indicates the absence or presence of VDDIO supplies, VDDA, and VDDD. In addition, power supply interrupts can be configured to generate interrupts on supply state change. The GPIO_VDD_INTR_MASK register is used to mask or

enable the forwarding of an interrupt to the CPUs. The status of an interrupt can be checked with the GPIO_VDD_INTR register.

24.3.5 Analog I/O

Analog resources, such as LPCOMP and external crystal pins for ECO and WCO, which require low-impedance routing paths, have fixed pins. These pins provide direct connections to specific analog blocks using the AMUXBUS. They help improve performance and should be given priority over other pins when using these analog resources. See the device datasheet for details on these dedicated pins. To configure a GPIO as an analog pin connecting to AMUXBUS, it should be configured in high-impedance analog mode with the input buffer disabled and then routed to the correct AMUXBUS using the HSIOM_PORT_SELx register.

24.3.6 Secure GPIO

Secure GPIO provides secure configuration per IO pin. All IO pins default to Non-secure. All AHB accesses associated with the pin configuration are subjected to the security attribute check as below:

1. Secure transactions can only access MMIO registers for pins that are configured as Secure
2. Non-secure transactions can only access MMIO registers for pins that are configured as Non-secure

Violation of the above rules results in Read Zero Write Ignore (RZWI) behavior. Therefore, software needs to ensure that the function/peripheral selected (via HSIOM_PRTx.PORT_SEL0.IOy_SEL) to access the IO pin has a matching security profile.

For MMIO registers common to all ports, and registers that are organized per port are exempted from this check. If a security check is required to access these MMIO registers, it must be placed in a secure region that is protected by a PPC.

24.4 Programmable digital (Smart I/O)

The Smart I/O block adds programmable logic to an I/O port. This programmable logic integrates board-level Boolean logic functionality such as AND, OR, and XOR into the port.

The Smart I/O block has the following features:

- Integrate board-level Boolean logic functionality into a port
- Ability to preprocess HSIOM input signals from the GPIO port pins
- Ability to postprocess HSIOM output signals to the GPIO port pins
- Support in all device power modes except Hibernate
- Integrate closely to the I/O pads, providing the shortest signal paths with programmability

24.4.1 Overview

The Smart I/O block is positioned in the signal path between the HSIOM and the I/O port. The HSIOM multiplexes the output signals from fixed-function peripherals and CPU to a specific port pin and conversely. The position of the Smart I/O block in this signal path enables it to act as a bridge that can process signals between port pins and HSIOM, as shown in Figure 88.

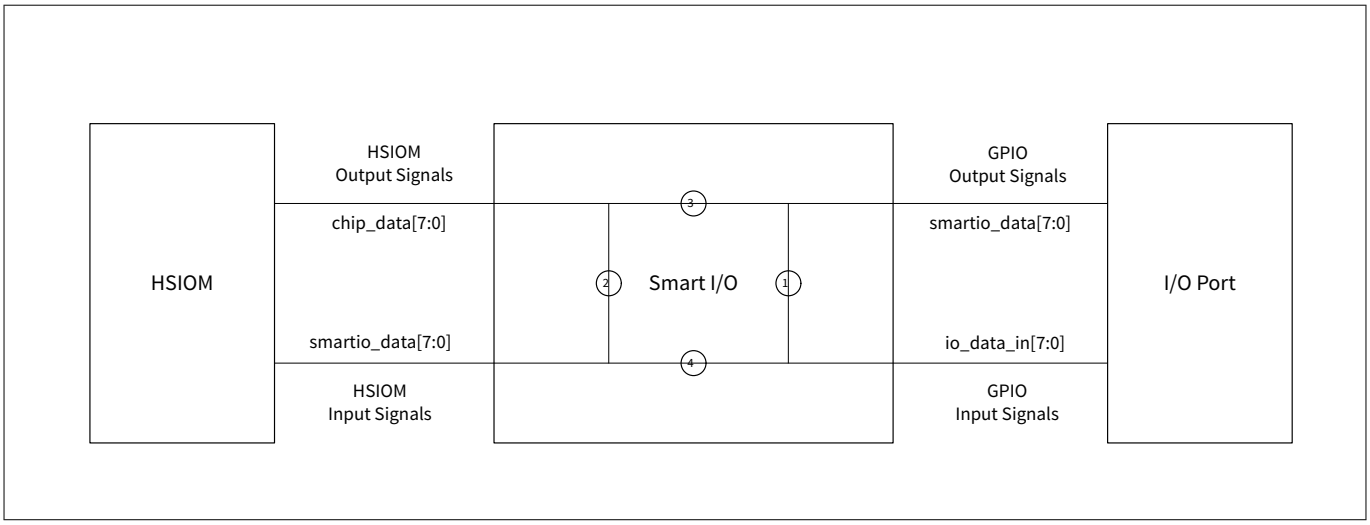


Figure 88 Smart I/O interface

The signal paths supported through the Smart I/O block as shown in Figure 88 are as follows:

1. Implement self-contained logic functions that directly operate on port I/O signals
2. Implement self-contained logic functions that operate on HSIOM signals
3. Operate on and modify HSIOM output signals and route the modified signals to port I/O signals
4. Operate on and modify port I/O signals and route the modified signals to HSIOM input signals

The following sections discuss the Smart I/O block components, routing, and configuration in detail. In these sections, GPIO signals (io_data) refer to the input/output signals connected to the I/O port; device or chip (chip_data) signals refer to the input/output signals connected to HSIOM.

24.4.2 Block components

The internal logic of the Smart I/O includes the following components:

- Clock and reset
- Synchronizers (Sync)
- Three-input look-up table (LUT)
- Data unit (DU)

Figure 89 shows the Smart I/O internal routing.

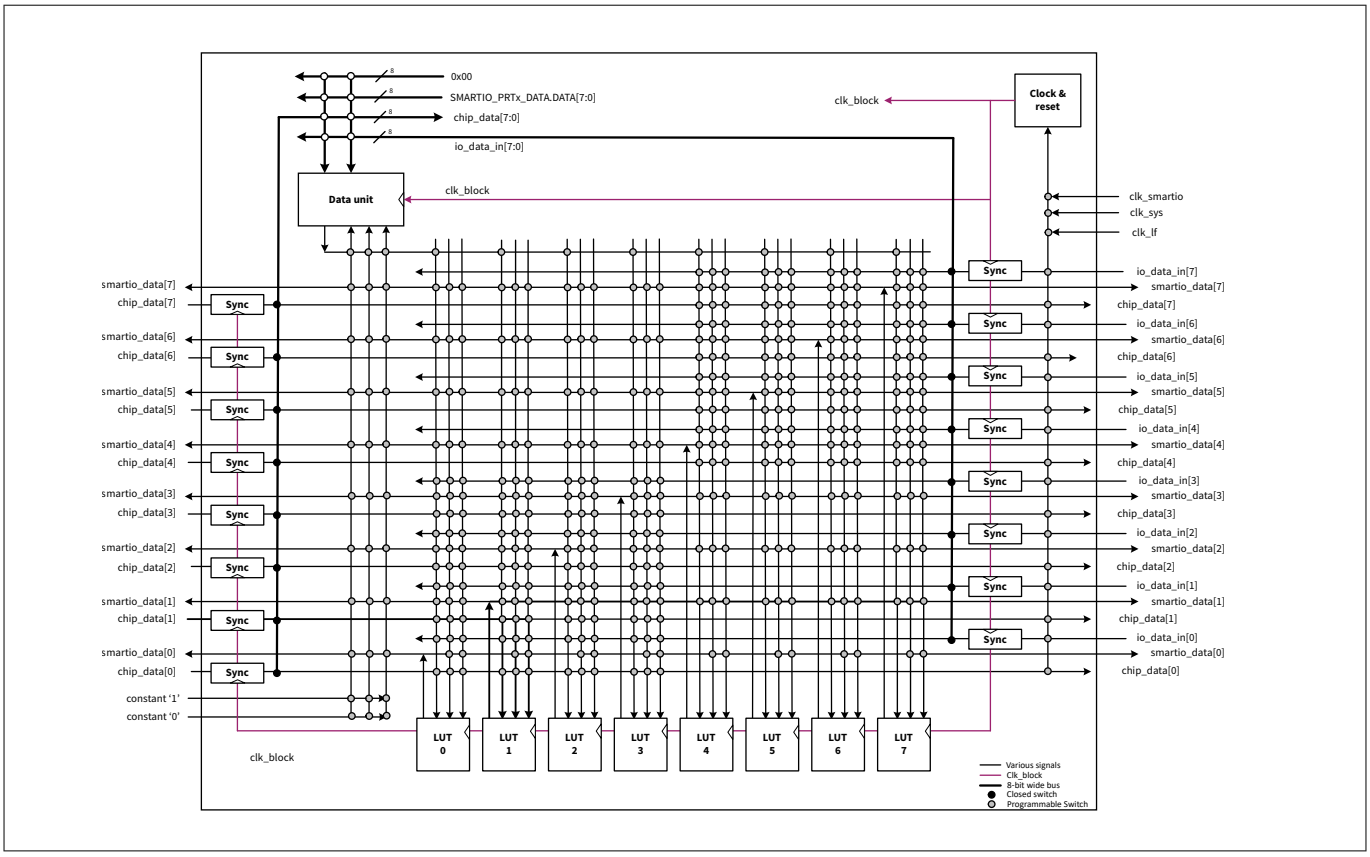


Figure 89 Smart I/O internal routing

24.4.2.1 Clock and reset

The clock and reset component selects the Smart I/O block's clock (clk_block) and reset signal (rst_block_n). A single clock and reset signal is used for all components in the block. The clock and reset sources are determined by the CLOCK_SRC[4:0] bitfield of the SMARTIO_PRTx_CTL register. The selected clock is used for the synchronous logic in the block components, which includes the I/O input synchronizers, LUT, and data unit components. The selected reset is used to asynchronously reset the synchronous logic in the LUT and data unit components.

***Note**: The selected clock (clk_block) for the block's synchronous logic is not phase-aligned with other synchronous logic in the device, operating on the same clock. Therefore, communication between Smart I/O and other synchronous logic should be treated as asynchronous.*

The following clock sources are available for selection:

- GPIO input signals "io_data_in[7:0]". These clock sources have no associated reset
- HSIOM output signals "chip_data[7:0]". These clock sources have no associated reset
- Smart I/O clock (clk_smartio). This is derived from the system clock (clk_sys) using a peripheral clock divider. See the clock section for details on peripheral clock dividers. This clock is available only in Active power mode. The clock can have one out of two associated resets: rst_sys_act_n and rst_sys_dpslp_n

  These resets determine in which power modes the block synchronous state is reset; for example, rst_sys_act_n is intended for Smart I/O synchronous functionality in Active power mode
- Low-frequency system clock (clk_lf). This clock is available in Deep Sleep power mode. This clock has an associated reset, rst_lf_dpslp_n. Reset is activated if the device enters Hibernate mode, or is at POR

When the block is enabled, the selected clock (clk_block) and associated reset (rst_block_n) are provided to the fabric components. When the fabric is disabled, no clock is released to the fabric components and the reset is activated (the LUT and data unit components are set to the reset value of ‘0’).

The I/O input synchronizers introduce a delay of two clk_block cycles (when synchronizers are enabled). As a result, in the first two cycles, the block may be exposed to stale data from the synchronizer output. Hence, during the first two clock cycles, the reset is activated and the block is in bypass mode.

Table 180 Clock and reset register control

Register[BIT_POS]	Bit Name	Description
SMARTIO_PRTn_CTL[12:8]	CLOCK_SRC[4:0]	Clock (clk_fabric)/reset (rst_fabric_n) source selection: 0: io_data_in[0]/1 ... 7: io_data_in[7]/1 8: chip_data[0]/1 ... 15: chip_data[7]/1 16: : clk_smartio/rst_sys_act_n. Used for Active functionality synchronous logic on "clk_smartio". This selection is intended for synchronous operation on a PCLK specified clock frequency ("clock_smartio_pos_en"). 17: clk_smartio/rst_sys_dpslp_n. Used for Deep Sleep functionality synchronous logic on "clk_smartio" (note that "clk_smartio" is NOT available in DeepSleep and Hibernate power modes). This selection is intended for synchronous operation on a PCLK specified clock frequency 18: same as 17 19: clk_lf/rst_lf_dpslp_n. This selection is intended for synchronous operation on"clk_lf". 20-30: The clock source is a constant '0'. Any of these clock sources should be selected when Smart I/O is disabled to ensure low power consumption. 31: clk_sys/1. This selection is intended for asynchronous operation. The Smart I/O is fully functional (reset is deactivated) after two clock edges of the selected clock.

24.4.2.2 Synchronizer

Each GPIO input signal and device input signal (HSIOM input) can be used either asynchronously or synchronously. To use the signals synchronously, a double flip-flop synchronizer, as shown in Figure 90, is placed on both these signal paths to synchronize the signal to the Smart I/O clock (clk_block). The synchronization for each pin/input is enabled or disabled by setting or clearing the IO_SYNC_EN[i] bitfield for GPIO input signal and CHIP_SYNC_EN[i] for HSIOM signal in the SMARTIO_PRTx_SYNC_CTL register, where 'i' is the pin number.

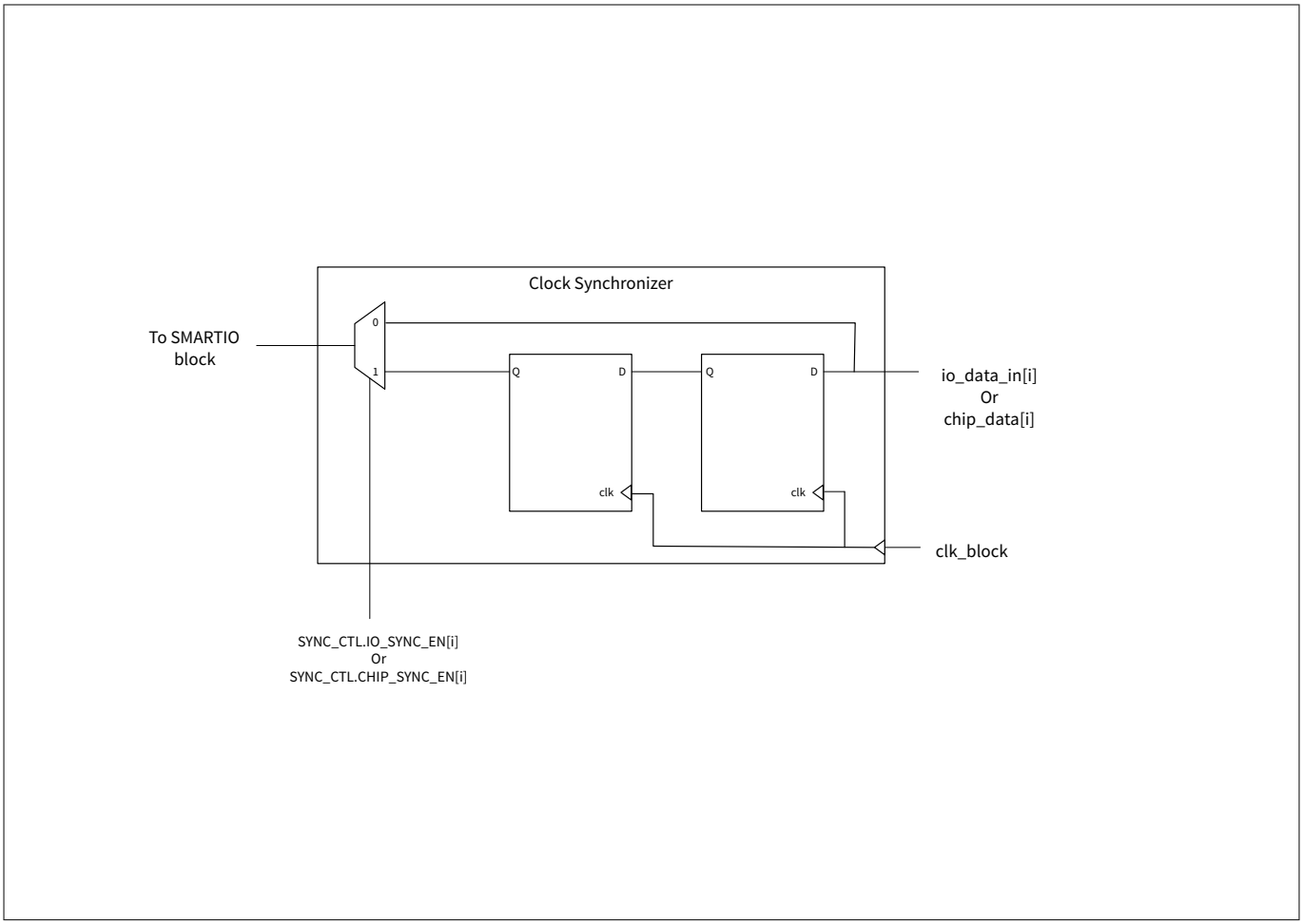


Figure 90 Smart I/O clock synchronizer

24.4.2.3 Lookup table (LUT)

Each Smart I/O block contains eight lookup table (LUT) components. The LUT component consists of a three-input LUT and a flip-flop. Each LUT block takes three input signals and generates an output based on the configuration set in the SMARTIO_PRTx_LUT_CTLy register (y denotes the LUT number). For each LUT, the configuration is determined by an 8-bit lookup vector LUT[7:0] and a 2-bit opcode OPC[1:0] in the SMARTIO_PRTx_LUT_CTLy register. The 8-bit vector is used as a lookup table for the three input signals. The 2-bit opcode determines the usage of the flip-flop. The LUT configuration for different opcodes is shown in Figure 91.

The SMARTIO_PRTx_LUT_SELy registers select the three input signals (tr0_in, tr1_in, and tr2_in) going into each LUT. The input can come from the following sources:

- Data unit output
- Other LUT output signals (tr_out)

- HSIOM output signals (chip_data[7:0])
- GPIO input signals (io_data[7:0])

LUT_TR0_SEL[3:0] bits of the SMARTIO_PRTx_LUT_SELy register selects the tr0_in signal for the yth LUT. Similarly, LUT_TR1_SEL[3:0] bits and LUT_TR2_SEL[3:0] bits select the tr1_in and tr2_in signals, respectively. For more details, see Table 181.

Table 181 LUT register control

Register[BIT_POS]	Bit Name	Description
SMARTIO_PRTx_LUT_CTLy[7:0]	LUT[7:0]	LUT configuration. Depending on the LUT opcode (LUT_OPC), the internal state, and the LUT input signals tr0_in, tr1_in, and tr2_in, the LUT configuration is used to determine the LUT output signal and the next sequential state.
SMARTIO_PRTx_LUT_CTLy[9:8]	LUT_OPC[1:0]	The LUT opcode specifies the LUT operation as illustrated in Figure 91
SMARTIO_PRTx_LUT_SELy[3:0]	LUT_TR0_SEL[3:0]	LUT input signal “tr0_in” source selection: 0: Data unit output 1: LUT 1 output 2: LUT 2 output 3: LUT 3 output 4: LUT 4 output 5: LUT 5 output 6: LUT 6 output 7: LUT 7 output 8: chip_data[0] (for LUTs 0, 1, 2, 3); chip_data[4] (for LUTs 4, 5, 6, 7) 9: chip_data[1] (for LUTs 0, 1, 2, 3); chip_data[5] (for LUTs 4, 5, 6, 7) 10: chip_data[2] (for LUTs 0, 1, 2, 3); chip_data[6] (for LUTs 4, 5, 6, 7) 11: chip_data[3] (for LUTs 0, 1, 2, 3); chip_data[7] (for LUTs 4, 5, 6, 7) 12: io_data_in[0] (for LUTs 0, 1, 2, 3); io_data[4] (for LUTs 4, 5, 6, 7) 13: io_data_in[1] (for LUTs 0, 1, 2, 3); io_data[5] (for LUTs 4, 5, 6, 7) 14: io_data_in[2] (for LUTs 0, 1, 2, 3); io_data[6] (for LUTs 4, 5, 6, 7) 15: io_data_in[3] (for LUTs 0, 1, 2, 3); io_data[7] (for LUTs 4, 5, 6, 7)

(table continues...)

Table 181 (continued) LUT register control

Register[BIT_POS]	Bit Name	Description
SMARTIO_PRTx_LUT_SELy[11:8]	LUT_TR1_SEL[3:0]	LUT input signal “tr1_in” source selection: 0: LUT 0 output 1: LUT 1 output 2: LUT 2 output 3: LUT 3 output 4: LUT 4 output 5: LUT 5 output 6: LUT 6 output 7: LUT 7 output 8: chip_data[0] (for LUTs 0, 1, 2, 3); chip_data[4] (for LUTs 4, 5, 6, 7) 9: chip_data[1] (for LUTs 0, 1, 2, 3); chip_data[5] (for LUTs 4, 5, 6, 7) 10: chip_data[2] (for LUTs 0, 1, 2, 3); chip_data[6] (for LUTs 4, 5, 6, 7) 11: chip_data[3] (for LUTs 0, 1, 2, 3); chip_data[7] (for LUTs 4, 5, 6, 7) 12: io_data_in[0] (for LUTs 0, 1, 2, 3); io_data[4] (for LUTs 4, 5, 6, 7) 13: io_data_in[1] (for LUTs 0, 1, 2, 3); io_data[5] (for LUTs 4, 5, 6, 7) 14: io_data_in[2] (for LUTs 0, 1, 2, 3); io_data[6] (for LUTs 4, 5, 6, 7) 15: io_data_in[3] (for LUTs 0, 1, 2, 3); io_data[7] (for LUTs 4, 5, 6, 7)
SMARTIO_PRTx_LUT_SELy[19:16]	LUT_TR2_SEL[3:0]	LUT input signal “tr2_in” source selection. The encoding is the same as for LUT_TR1_SEL.

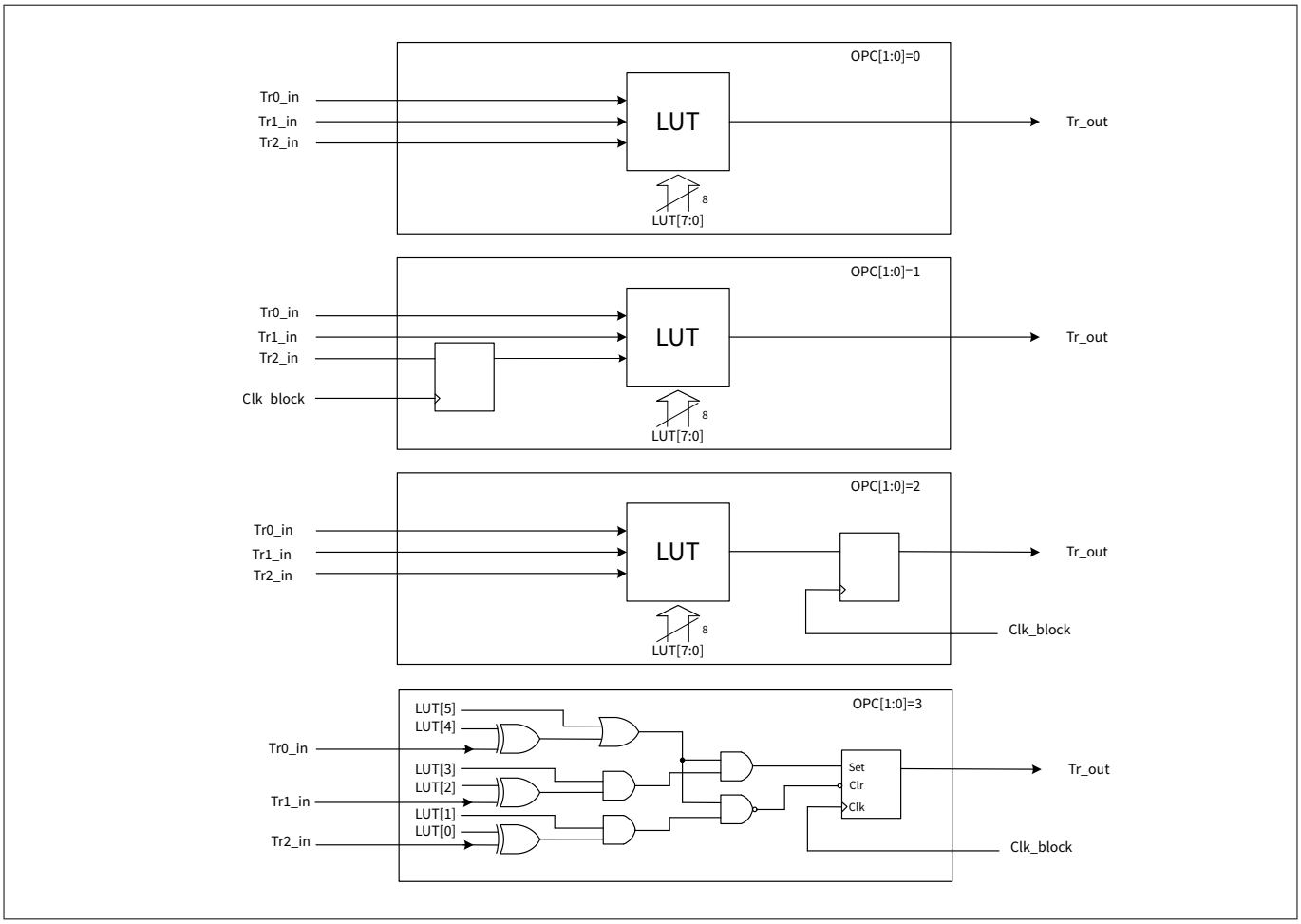


Figure 91 Smart I/O LUT configuration

24.4.2.4 Data unit (DU)

Each Smart I/O block includes a data unit (DU) component. The DU consists of a simple 8-bit datapath. It is capable of performing simple increment, decrement, increment/decrement, shift, and AND/OR operations. The operation performed by the DU is selected using a 4-bit opcode DU_OPC[3:0] bitfield in the SMARTIO_PRTx_DU_CTL register.

The DU component supports up to three input trigger signals (tr0_in, tr1_in, tr2_in) similar to the LUT component. These signals are used to initiate an operation defined by the DU opcode. In addition, the DU also includes two 8-bit data inputs (data0_in[7:0] and data1_in[7:0]) that are used to initialize the 8-bit internal state (data[7:0]) or to provide a reference. The 8-bit data input source is configured as:

- Constant ‘0x00’
- io_data_in[7:0]
- chip_data_in[7:0]
- DATA[7:0] bitfield of SMARTIO_PRTx_DATA register

The trigger signals are selected using the DU_TRx_SEL[3:0] bitfield of the SMARTIO_PRTx_DU_SEL register. The DUT_DATAx_SEL[1:0] bits of the SMARTIO_PRTx_DU_SEL register select the 8-bit input data source. The size of the DU (number of bits used by the datapath) is defined by the DU_SIZE[2:0] bits of the SMARTIO_PRTx_DU_CTL register. See the block diagram in Figure 92 for register control details.

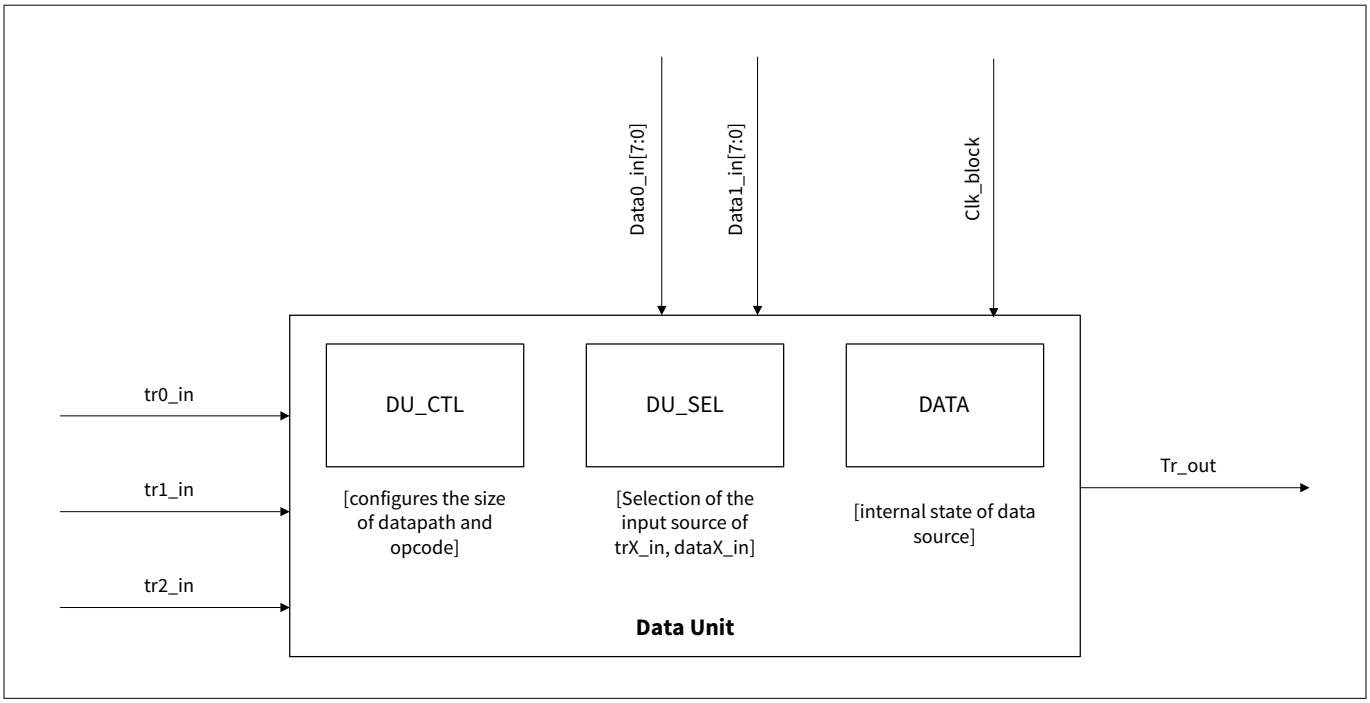


Figure 92 Data unit (DU)

See Table 182 for register control details.

Table 182 Data unit register control

Register[BIT_POS]	Bit name	Description
SMARTIO_PRTx_DU_CTL[2:0]	DU_SIZE[2:0]	Size/width of the data unit (in bits) is DU_SIZE+1. For example, if DU_SIZE is 7, the width is 8 bits.
SMARTIO_PRTx_DU_CTL[11:8]	DU_OPC[3:0]	The data unit opcode specifies the data unit operation: 1: INCR 2: DECR 3: INCR_WRAP 4: DECR_WRAP 5: INCR_DECR 6: INCR_DECR_WRAP 7: ROR 8: SHR 9: AND_OR 10: SHR_MAJ3 11: SHR_EQL Otherwise: Undefined.

(table continues...)

Table 182 (continued) Data unit register control

Register[BIT_POS]	Bit name	Description
SMARTIO_PRTx_DU_SEL[3:0]	DU_TR0_SEL[3:0]	Data unit input signal “tr0_in” source selection: 0: Constant '0'. 1: Constant '1'. Data unit output. 10–3: LUT 7–0 outputs. Otherwise: Undefined.
SMARTIO_PRTx_DU_SEL[11:8]	DU_TR1_SEL[3:0]	Data unit input signal “tr1_in” source selection. Encoding same as DU_TR0_SEL
SMARTIO_PRTx_DU_SEL[19:16]	DU_TR2_SEL[3:0]	Data unit input signal “tr2_in” source selection. Encoding same as DU_TR0_SEL
SMARTIO_PRTx_DU_SEL[25:24]	DU_DATA0_SEL[1:0]	Data unit input data “data0_in” source selection: 0: 0x00 1: chip_data[7:0] 2: io_data[7:0] 3: SMARTIO_PRTx_DATA.DATA[7:0] register field
SMARTIO_PRTx_DU_SEL[29:28]	DU_DATA1_SEL[1:0]	Data unit input data “data1_in” source selection. Encoding same as DU_DATA0_SEL.
SMARTIO_PRTx_DATA[7:0]	DATA[7:0]	Data unit input data source.

The DU generates a single output trigger signal (tr_out). The internal state (du_data[7:0]) is captured in flip-flops and requires clk_block.

The following pseudo code describes the various datapath operations supported by the DU opcode. Note that “Comb” describes the combinatorial functionality – that is, functions that operate independent of previous

output states. “Reg” describes the registered functionality – that is, functions that operate on inputs and previous output states (registered using flip-flops).

```
// The following is shared by all operations.
mask = (2 ^ (DU_SIZE + 1)- 1)
data_eql_data1_in = (data & mask) == (data1_in & mask));
data_eql_0 = (data & mask) == 0);
data_incr = (data + 1) & mask;
data_decr = (data - 1) & mask;
data0_masked = data_in0 & mask;
// INCR operation: increments data by 1 from an initial value (data0) until it reaches a
// final value (data1).
Comb: tr_out = data_eql_data1_in;
Reg: data <= data;
if (tr0_in) data <= data0_masked; //tr0_in is reload signal - loads masked data0
// into data
else if (tr1_in) data <= data_eql_data1_in ? data : data_incr; //increment data until
// it equals data1
// INCR_WRAP operation: operates similar to INCR but instead of stopping at data1, it
// wraps around to data0.
Comb: tr_out = data_eql_data1_in;
Reg: data <= data;
if (tr0_in) data <= data0_masked;
else if (tr1_in) data <= data_eql_data1_in ? data0_masked : data_incr;
// DECR operation: decrements data from an initial value (data0) until it reaches 0.
Comb: tr_out = data_eql_0;
Reg: data <= data;
if (tr0_in) data <= data0_masked;
else if (tr1_in) data <= data_eql_0 ? data : data_decr;
// DECR_WRAP operation: works similar to DECR. Instead of stopping at 0, it wraps around
// to data0.
Comb: tr_out = data_eql_0;
Reg: data <= data;
if (tr0_in) data <= data0_masked;
else if (tr1_in) data <= data_eql_0 ? data0_masked : data_decr;
// INCR_DECR operation: combination of INCR and DECR. Depending on trigger signals
// it either starts incrementing or decrementing. Increment stops at data1 and decrement
// stops at 0.
Comb: tr_out = data_eql_data1_in | data_eql_0;
Reg: data <= data;
if (tr0_in) data <= data0_masked; // Increment operation takes precedence over
// decrement when both signal are available
else if (tr1_in) data <= data_eql_data1_in ? data : data_incr;
else if (tr2_in) data <= data_eql_0 ? data : data_decr;
// INCR_DECR_WRAP operation: same functionality as INCR_DECR with wrap around to data0 on
// reaching the limits.
Comb: tr_out = data_eql_data1_in | data_eql_0;
Reg: data <= data;
if (tr0_in) data <= data0_masked;
else if (tr1_in) data <= data_eql_data1_in ? data0_masked : data_incr;
else if (tr2_in) data <= data_eql_0 ? data0_masked : data_decr;
// ROR operation: rotates data right and LSb is sent out. The data for rotation is taken
// from data0.
```

```
Comb: tr_out = data[0];
Reg: data <= data;
if (tr0_in) data <= data0_masked;
else if (tr1_in) {
data <= {
0,
data[7: 1]
} & mask; //Shift right operation
data[du_size] <= data[0]; //Move the data[0] (LSb) to MSb
}
// SHR operation: performs shift register operation. Initial data (data0) is shifted out
// and data on tr2_in is shifted in.
Comb: tr_out = data[0];
Reg: data <= data;
if (tr0_in) data <= data0_masked;
else if (tr1_in) {
data <= {
0,
data[7: 1]
} & mask; //Shift right operation
data[du_size] <= tr2_in; //tr2_in Shift in operation
}
// SHR_MAJ3 operation: performs the same functionality as SHR. Instead of sending out
// the shifted out value, it sends out a '1' if in the last three samples/shifted-out
// values (data[0]), the signal high in at least two samples. otherwise, sends a '0'.
// This function sends out the majority of the last three samples.
Comb: tr_out = (data == 0x03) |
(data == 0x05) |
(data == 0x06) |
(data == 0x07);
Reg: data <= data;
if (tr0_in) data <= data0_masked;
else if (tr1_in) {
data <= {
0,
data[7: 1]
} & mask;
data[du_size] <= tr2_in;
}
// SHR_EQL operation: performs the same operation as SHR. Instead of shift-out, the
// output is a comparison result (data0 == data1).
Comb: tr_out = data_eql_data1_in;
Reg: data <= data;
if (tr0_in) data <= data0_masked;
else if (tr1_in) {
data <= {
0,
data[7: 1]
} & mask;
data[du_size] <= tr2_in;
}
// AND_OR operation: ANDs data1 and data0 along with mask; then, ORs all the bits of
// the ANDed output.
```

```
Comb: tr_out = | (data & data1_in & mask);
Reg: data <= data;
if (tr0_in) data <= data0_masked;
```

24.4.3 Routing

The Smart I/O block includes many switches that are used to route the signals in and out of the block and also between various components present inside the block. The routing switches are handled through the PRTGIO_PRTx_LUT_SELy and SMARTIO_PRTx_DU_SEL registers. For more details, see the register reference manual. The Smart I/O internal routing is shown in . In the figure, note that LUT7 to LUT4 operate on io_data/chip_data[7] to io_data/chip_data[4] whereas LUT3 to LUT0 operate on io_data/chip_data[3] to io_data/chip_data[0].

24.4.4 Operation

The Smart I/O block should be configured and operated as follows:

1. Before enabling the block, all the components and routing should be configured as explained in
2. In addition to configuring the components and routing, some block level settings must be configured correctly for the desired operation.
   a. Bypass control: The Smart I/O path can be bypassed for a particular GPIO signal by setting the BYPASS[i] bitfield in the SMARTIO_PRTx_CTL register. When bit 'i' is set in the BYPASS[7:0] bitfield, the ith GPIO signal is bypassed to the HSIOM signal path directly – Smart I/O logic will not be present in that signal path. This is useful when the Smart I/O function is required only on select I/Os
   b. Pipelined trigger mode: The LUT input multiplexers and the LUT component itself do not include any combinatorial loops. Similarly, the data unit also does not include any combinatorial loops. However, when one LUT interacts with the other or to the data unit, inadvertent combinatorial loops are possible. To overcome this limitation, the PIPELINE_EN bitfield of the SMARTIO_PRTx_CTL register is used. When set, all the outputs (LUT and DU) are registered before branching out to other components
3. After the Smart I/O block is configured for the desired functionality, the block can be enabled by setting the ENABLED bitfield of the SMARTIO_PRTx_CTL register. If disabled, the Smart I/O block is put in bypass mode, where the GPIO signals are directly controlled by the HSIOM signals and conversely. The Smart I/O block must be configured; that is, all register settings must be updated before enabling the block to prevent glitches during register updates

Table 183 Smart I/O block controls

Register [BIT_POS]	Bit Name	Description
SMARTIO_PRTx_CTL[25]	PIPELINE_EN	Enable for pipeline register: 0: Disabled (register is bypassed). 1: Enabled

(table continues...)

Table 183 (continued) Smart I/O block controls

Register [BIT_POS]	Bit Name	Description
SMARTIO_PRTx_CTL[31]	ENABLED	Enable Smart I/O. Set to '1' only when the Smart I/O is completely configured: 0: Disabled (signals are bypassed; behavior as if BYPASS[7:0] is 0xFF). When disabled, the block (data unit and LUTs) reset is activated. If the block is disabled: - The PIPELINE_EN register field should be set to '1', to ensure low power consumption. - The CLOCK_SRC register field should be set to 20 to 30 (clock is constant '0'), to ensure low power consumption. 1: Enabled. When enabled, it takes three clk_block clock cycles until the block reset is deactivated and the block becomes fully functional. This action ensures that the I/O pins' input synchronizer states are flushed when the block is fully functional.
SMARTIO_PRTx_CTL[7:0]	BYPASS[7:0]	Bypass of the Smart I/O, 1 bit for each I/O pin: BYPASS[i] is for I/O pin i. When ENABLED is '1', this field is used. When ENABLED is '0', this field is not used and Smart I/O is always bypassed. 0: No bypass (Smart I/O is present in the signal path) 1: Bypass (Smart I/O is absent in the signal path)

24.5 Registers summary

Table 184 Registers summary

Registers	Name	Description
GPIO_PRTx_OUT	Port Output register	Reads and writes the output driver data for I/O pins in the port.
GPIO_PRTx_OUT_CLR	Port Output Clear register	Clears output data of specific I/O pins in the port.

(table continues...)

Table 184 (continued) Registers summary

Registers	Name	Description
GPIO_PRTx_OUT_SET	Port Output Set register	Sets output data of specific I/O pins in the port.
GPIO_PRTx_OUT_INV	Port Output Invert register	Inverts output data of specific I/O pins in the port.
GPIO_PRTx_IN	Port Input register	Reads the current pin state present on I/O pin inputs.
GPIO_PRTx_INTR	Port Interrupt Status register	Reads the current pin interrupt state.
GPIO_PRTx_INTR_MASK	Port Interrupt Mask register	Configures the mask that forwards pin interrupts to the CPU's interrupt controller. This register only masks forwarding of interrupts to the CPU's interrupt controller; it does not enable/ disable the logging of interrupts into the INTR register.
GPIO_PRTx_INTR_MASKED	Port Interrupt Masked Status register	Contains the AND-ed values of INTR and INTR_MASK registers forwarded to the CPU interrupt controller.
GPIO_PRTx_INTR_SET	Port Interrupt Set register	Allows firmware to set pin interrupts.
GPIO_PRTx_INTR_CFG	Port Interrupt Configuration register	Selects the edge detection type for each pin interrupt.
GPIO_PRTx_CFG	Port Configuration register	Selects the drive mode and input buffer enable for each pin.
GPIO_PRTx_CFG3	Enhanced port configuration register for I3C capable I/Os	Selects the enhanced drive mode including the additional pull-up resistor options for I3C
GPIO_PRTx_CFG_IN	Port Input Configuration register	Configures the input buffer mode (CMOS or TTL) for each pin. VTRIP_SEL[7:0]_0.
GPIO_PRTx_CFG_OUT	Port Output Configuration register	Configures the port output buffer drive strength.
GPIO_PRTx_CFG_SLEW_EXT	Port Output Configuration register	Configures the port output buffer slew rate.
HSIOM_PRTx_PORT_SELy	HSIOM Port Select Register	Selects the hardware peripheral connection to I/O pins.
GPIO_INTR_CAUSEz	Interrupt Port Cause Register	Provides interrupt status corresponding to ports (0 + z × 32) to (31 + z × 32). "z" can be from 0 to 3.
GPIO_VDD_ACTIVE	External Power Supply Detection Register	Provides external power supply status.

(table continues...)

Table 184 (continued) Registers summary

Registers	Name	Description
GPIO_VDD_INTR	Supply Detection Interrupt Register	This register is set whenever a supply ramp up or ramp down is detected. Some bits may be set after system power-up, depending on power supply sequencing.
GPIO_VDD_INTR_MASK	Supply Detection Interrupt Mask Register	Configures the supply detection interrupts for all supplies. It only masks the forwarding of interrupts to the CPUs and does not enable/disable the logging of interrupts into the VDD_INTR Register.
GPIO_VDD_INTR_MASKED	Supply Detection Interrupt Masked Register	Contains the AND-ed values of VDD_INTR and VDD_INTR_MASK Registers.
GPIO_VDD_INTR_SET	Supply Detection Interrupt Set Register	Allows a firmware or debugger to set interrupt bits in the VDD_INTR register by writing a '1' to the corresponding bit field. When read, it returns the same value as the VDD_INTR Register.
GPIO_VDD_PORSW_STATUS	Power switch status	Provides the status of the power switch.
GPIO_VDD_PORSW_CTL	Supply Power Switch control Register	Configures inrush current limit level select, threshold select, and level set for GPIO.
GPIO_SEC_INTR_CAUSE	Secure Interrupt port cause register	Provides interrupt status corresponding to ports.
SMARTIO_PRTx_CTL	SMARTIO Control Register	This is the control register for SMARTIO on the specific port. It controls enable, clock source, bypass, and so on
SMARTIO_PRTx_LUT_SELy	LUT Component Input Selection Register	LUT Input Selection Register. LUT_TR0_SEL, LUT_TR1_SEL, and LUT_TR2_SEL
SMARTIO_PRTx_LUT_CTLy	LUT Component Control Register	Provides the opcode for LUT
SMARTIO_PRTx_DU_SEL	Data Unit Component Input Selection Register	Data Unit Input Selection Register
SMARTIO_PRTx_DU_CTL	Data Unit Component Control Register	Data Unit Control Register
SMARTIO_PRTx_DATA	Data Register	Data unit input data source

25 Timer, counter, and pulse-width modulator (TCPWM)

The timer, counter, and pulse-width modulator (TCPWM) block in the PSC3M/P6 MCU provides multiple functions. It includes 16-bit and 32-bit timer and counter features. It also supports quadrature (QUAD), PWM, high-resolution PWM (HRPWM), and MOTIF (motion interface). The TCPWM block is implemented as three groups, with up to 8 counters per group. Groups 1 and 2 offer 16-bit resolution and Group 0 offers 32-bit resolution. The timer can measure the input signal period and pulse width. The counter can count event occurrences. The block can generate PWM signals in both active mode and Sleep mode. Group 0 includes HRPWM and Groups 1 and 2 include quadrature (QUAD) and MOTIF.

This chapter covers the TCPWM features, implementation details, and its operational modes.

25.1 Features

The TCPWM block has the following features:

- Supports up to 3 counter groups. Each counter group consists up to 8 counters
- Each counter can run in one of the following function modes:
  - Timer
  - Capture
  - Quadrature (QUAD)
  - PWM
  - PWM with dead time (PWM_DT)
- A configurable motion interface (MOTIF) block (included in counter group 1 and group 2)
- The MOTIF module can operate in four different modes: quadrature decoder, hall sensor, stand-alone multi-channel mode, and quadrature mode with stand-alone multi-channel
- Supports the High-Resolution PWM (HRPWM) feature for 32-bit TCPWM counters (Group 0), in both PWM and PWM_DT modes
- 16- or 32-bit counters , two compare/capture (CC0/CC1), and period registers
- Parallel data path support for CC0/CC1 register through dedicated data interface
- Counter compare, period, line_sel, and dead time registers are double-buffered
- Up, down, and up/down counting modes
- Clock pre-scaling (1x, 2x, 4x , 8x, 16x, 32x, 64x, 128x)
- Glitch filter with configurable depth supported on General-purpose triggers used by all counters and specific One-to-one triggers for each counter
- Rising edge, falling edge, combined rising/falling edge detection, or pass-through on all hardware input signals to derive counter events
- The start, reload, stop, capture0, and capture1 events can be generated by software
- Two output trigger signals for each counter to indicate underflow, overflow, cc0_match, cc1_match, cc0_cc1_match, LINE_OUT, and TC events
- Two PWM complementary output lines for each counter. Dead time insertion is 16 bits and independent for two PWM outputs
- Hardware lost bit-field status for missing captured events (CC0 and CC1 compare/capture registers)
- Independent control of clock pre-scaling and dead time
- Configurable PWM dithering on period as well as duty cycle in Group 1 and Group 2
- Configurable option to continue/pause with the ability to enter passive state through kill polarity in debug mode

- Output state control for PWM signal during the stop or kill operation synchronous to period match
- Independent line polarity setting for PWM signal during the kill operation

25.1.1 Supported features of TCPWM block

The following table lists the counter group supported features of TCPWM.

Table 185 Counter group features of TCPWM block

Features[1)]	TCWPM0		
	Group 0	**Group 1**	**Group 2**
Number of counters per TCPWM group	8	8	8
Counter width in number of bits per TCPWM group (GRP_CNT_WIDTH)	32-bit	16-bit	16-bit
Advanced motor control features (AMC) are present (GRP_AMC_PRESENT)	1	1	1
Stepper Motor Control features are present	1	1	1
HRPWM feature extension is available for counter group (GRP_HRPWM_PRESENT)	1	N/A	N/A
Number of micro ticks bits supported in HRPWM feature (GRP_HRPWM_WIDTH)	6	N/A	N/A
Number of TCPWM MOTIF instances units per counter group (GRP_MOTIF_NR)	N/A	1	1
Dithering Function is present (GRP_DITHERING_PRESENT)	N/A	1	1
Quadrature feature inside counter are present (QUAD_PRESENT)	N/A	1	1

1) The feature is present; N/A: the feature is not present.

25.2 Architecture

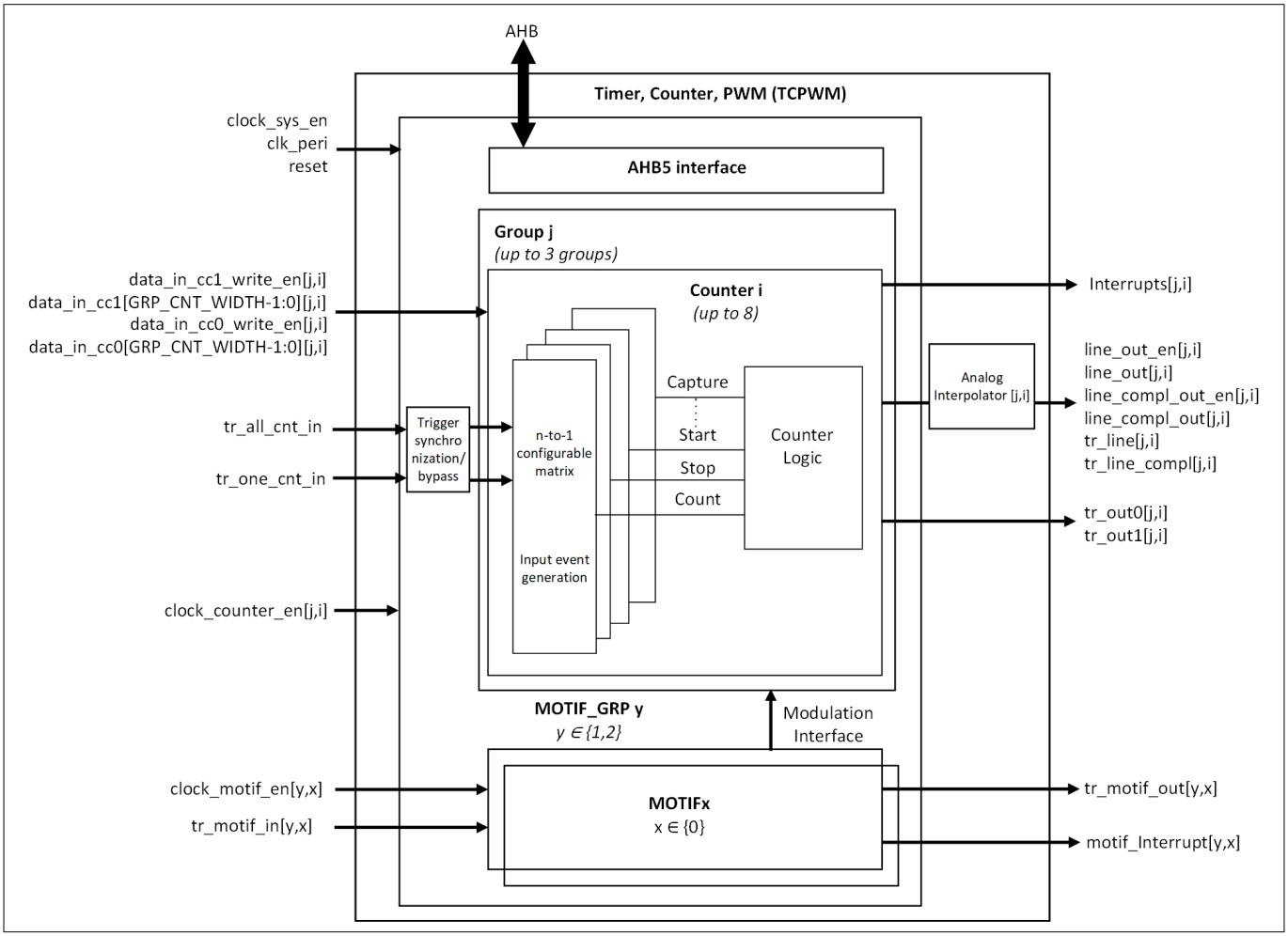


Figure 93 TCPWM block diagram

The three main registers that control the counters are:

- COUNTER: Holds the current counter value
- PERIOD: Stores the upper threshold for the counter. When the counter runs for n cycles, this field should be set to n − 1
- CC0/CC1: Used to capture the counter value in CAPTURE mode. In all other modes, these registers are compared to the counter value

Each counter supports up to 29 input trigger signals, plus two constant inputs, '0' and '1'. All inputs are synchronized to the CLK_PERI clock

TCPWM block interfaces:

- **Bus interface:** Connects the block to the CPU subsystem via the AHB-5 interface
- **I/O signal interface:**
  - Input triggers (examples: reload, start, stop, count, capture0, capture1) that initiate internal counter events
  - Output signals (examples: LINE_OUT, LINE_COMPL_OUT, TR_OUT0, TR_OUT1) that reflect internal events (such as reloads) and can be used to trigger other peripherals
  - Internal events (underflow, overflow, capture/compare) generate the corresponding outputs
- **Interrupts:** Provides interrupt request signals from each counter based on terminal count (TC), CC0_match, or CC1_match events

See the Registers summary for more information on all registers required for this block.

25.2.1 Enabling and disabling counters in a TCPWM block

A counter can be enabled by writing '1' to the corresponding ENABLE bit of the CTRL register; it can be disabled by writing '0' to the same bit.

***Note**: The counter must be configured before enabling it. Disabling the counter retains the values in the configuration registers.*

25.2.2 Clocking

The TCPWM counter can have its own clock. The only source for the clock is from the configurable peripheral clock dividers generated by the clocking system; see the Clocking system chapter for details. To select a clock divider for a particular counter inside a TCPWM, use the CLOCK_CTL register from the peripheral register space. In this section, the clock to the counter is called clk_counter. Event generation is performed on clk_counter. Another clock, system clock (clk_sys), is used for the pulse width of the output triggers. The system clock (clk_sys) is synchronous to clk_peri, but can be divided using CLOCK_CTL.

25.2.2.1 Clock pre-scaling

The clk_counter can be further divided inside each counter, with values of 1, 2, 4, 8, 16, 32, 64, and 128. This division is called pre-scaling. The pre-scaling is set in the PS_DIV [2:0] field of the PS register.

25.2.2.2 Count event

The counter functionality is performed on an “active count” pre-scaled clock, which is gated by a “count event” signal. For example, a counter increments or decrements by '1' every counter clock cycle in which a count event is detected.

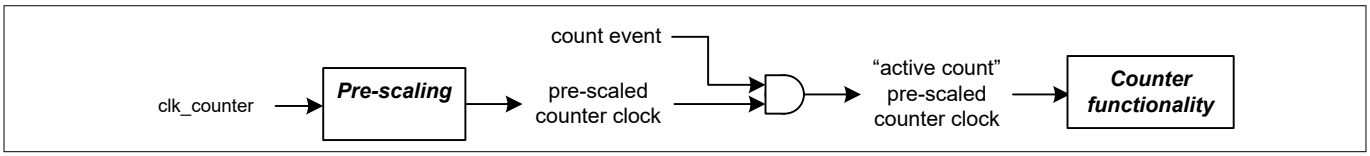


Figure 94 Counter clock generation

All status or output changes can only happen at the “active count” pre-scaled counter clock. In other words, if the count event is inactive, counter, status, interrupt, and all outputs do not change value. For example, if the count event is in pass through mode, and if the count event is low when the counter goes to “0” in down count mode, the terminal count (tc) event and underflow event are generated at the next pre-scaled counter clock after the count event goes high. The only exception is immediate kill mode. Kill input suppresses the PWM output immediately regardless of the “active count” pre-scaled counter clock.

***Note**: Not supported in all modes*

25.2.3 Trigger inputs

The TCPWM block has 29 Trigger_In signals and constant '0' and '1' signals, which come from other on-chip resources such as SCBs, DMA, HPPASS, and so on. The Trigger_In signals are shared with all counters inside the TCPWM block.

Two types of trigger signals are synchronized and can be used by the counters to generate events.

- **General-purpose triggers:** These can be used by all counters. These triggers are generated by different blocks in the system and are distributed by the trigger infrastructure (peripheral trigger multiplexers). See the Trigger multiplexer (Trigger MUX) chapter for more details
- **One-to-one triggers:** A separate set exists for each counter, that is only connected to that counter. These triggers are used for direct trigger connections from trigger sources (such as HPPASS level triggers) to associated TCPWM counters

Use the trigger mux registers, TR_IN_SEL0 and TR_IN_SEL1, to configure which signals get routed to the Trigger_In for the TCPWM block. See Table 186 for all possible multiplexer settings selecting an input trigger event for a TCPWM block. For each event, two constant trigger inputs are available. Input trigger 0 is always constant '0' and input trigger 1 is always constant '1'.

Each counter can select any of the 31 trigger signals to be the source for any of the following events:

- Capture 0 and Capture 1
- Count
- Reload
- Stop/Kill
- Start

***Note**: The PERI0_TR_CMD register can be used to trigger the Reload, Stop, Start, and Capture0/1 respectively from software.*

Table 186 Multiplexer selection for input trigger events

Input Trigger Selection Register	Bit field	Bits	Description
TR_IN_SEL0	CAPTURE0_SEL	4:0	Selects one of the 31 input triggers as a capture0 trigger. In the PWM, PWM_DT modes this trigger is used to switch the values, if the compare and period registers with their buffer counterparts
	COUNT_SEL	12:8	Selects one of the 31 input triggers as a count trigger. In QUAD mode, this is the first phase (phi A)
	RELOAD_SEL	20:16	Selects one of the 31 input triggers as a reload trigger. In QUAD mode, this is the index or revolution pulse
	STOP_SEL	28:24	Selects one of the 31 input triggers as a stop trigger. In PWM, PWM_DT modes, this is the kill trigger

(table continues...)

Table 186 (continued) Multiplexer selection for input trigger events

Input Trigger Selection Register	Bit field	Bits	Description
TR_IN_SEL1	START_SEL	4:0	Selects one of the 31 input triggers as a start trigger. In QUAD mode, this is the second phase (phi B)
	CAPTURE1_SEL	12:8	Selects one of the up to 31 input triggers as a capture1 trigger

The following sections describe each TCPWM mode and the function of each input event in detail.

A typical operation uses the reload event once to initialize and start the counter and the stop event to stop the counter. When the counter is stopped, the start event can be used to start the counter with its counter value unmodified from when it was stopped.

If stop, reload, and start events coincide, the following precedence relationship holds:

- A stop event has a higher priority than a reload event
- A reload event has a higher priority that a start event

As a result, when a reload or start event coincides with a stop event, the reload or start event has no effect.

Before going to the counter each Trigger_IN can pass through a positive edge detector, negative edge detector, both edge detector, or pass straight through to the counter. This is controlled using the TR_IN_EDGE_SEL register.

Multiple detected events are treated as follows:

- In the rising edge and falling edge modes, multiple events are effectively reduced to a single event. As a result, events may be lost
- In the rising/falling edge mode, an even number of events is not detected and an odd number of events is reduced to a single event. This is because the rising/falling edge mode is typically used for capture events to determine the width of a pulse. The current functionality ensures that the alternating pattern of rising and falling is maintained

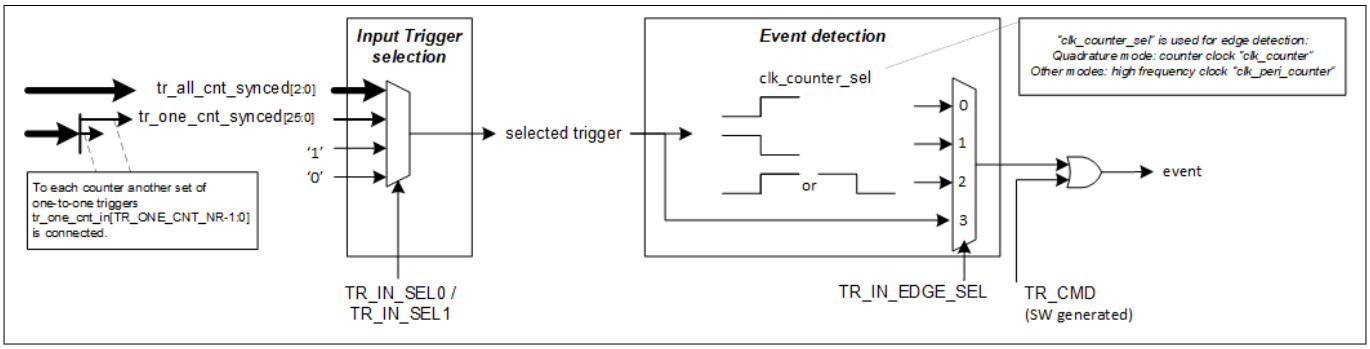


Figure 95 TCPWM input events

According to Table 186, a dedicated input trigger signal for trigger event generation can be defined by the TR_IN_SEL0 and TR_IN_SEL1 registers. The selection can be done between two constant signals (constant '0' and constant '1'), specific one-to-one trigger input signals, or general-purpose input trigger signals. Figure 95 shows how the input trigger source is selected.

The number of one-to-one (tr_one_cnt_synced) and general-purpose (tr_all_cnt_synced) input triggers are device-specific (refer to the PSC3M/P6 MCU datasheet).

Table 187 shows how the multiplexer should be handled for the input trigger event generation.

Table 187 Handling input trigger multiplexers

Input trigger selection	Input trigger	Input trigger source
0	constant ‘0’	constant '0'
1	constant ‘1’	constant '1'
2	tr_one_cnt_in[0]	Refer to the trigger mux block
3	tr_one_cnt_in[1]	Refer to the trigger mux block
4	tr_one_cnt_in[2]	Refer to the trigger mux block
5	tr_all_cnt_in[0]	Refer to the trigger mux block
...	...	...
30	tr_all_cnt_in[25]	Refer to the trigger mux block

***Note**: The input triggers can be generated by different sources. While the general-purpose trigger inputs (tr_all_cnt_in[0] to tr_all_cnt_in[25])] are only from the trigger multiplexer block (see the Trigger inputs chapter), the one-to-one input triggers can also be generated by external GPIO input pins.*

All trigger inputs are synchronized to a peripheral clock. When more than one event occurs in the same counter_clock period, one or more events may be missed. This can happen for high-frequency events (frequencies close to the counter frequency) and a timer configuration in which a pre-scaled (divided) counter_clock is used.

The following figure illustrates the timing on how input triggers are detected by the counter.

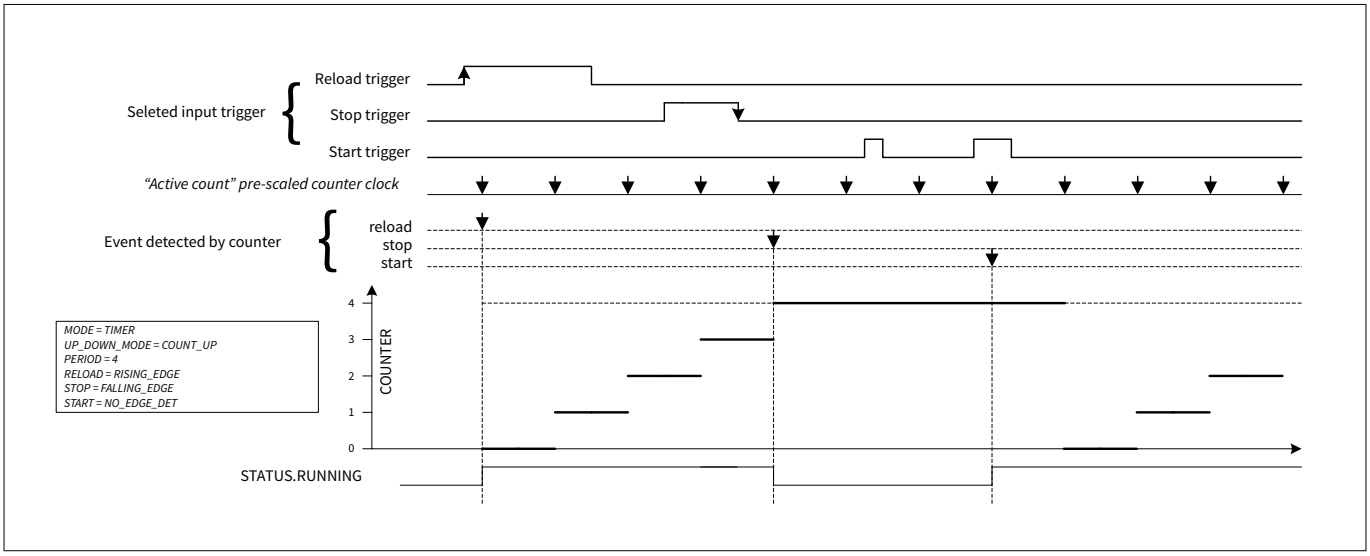


Figure 96 Input trigger detection by “active count” pre-scaled counter clock

***Note**: The event marked as an arrow in the figures in the following sections depicts the event that has been detected by the counter.*

Typically, the count event is a constant ‘1’ and pre-scaling is off. In this case, the “active count” pre-scaled counter clock is the same as the counter clock. In other cases, edge detection may detect multiple events (on

the counter clock) before the next “active count” pre-scaled counter clock on which the detected event is used. Multiple detected events are treated as follows:

- In the rising edge and falling edge modes, multiple events are effectively reduced to a single event. As a result, events may be lost
- In the rising/falling edge mode, an even number of events is not detected and an odd number of events is reduced to a single event. This is because the rising/falling edge mode is typically used for capture events to determine the width of a pulse. The current functionality ensures that the alternating pattern of rising, falling, rising, falling, and so on, is maintained

A pass-through event is not remembered by CLK_PERI; it affects the functionality if it lasts and can be detected by a counter-operation clock. If the pulse width of a pass-through event is less than a counter-operation clock cycle, it may get lost. Pass-through detection may result in an event that is active for multiple counter clocks. This may result in undesirable behavior of the counter and its associated trigger outputs. Use the pass-through event detection only for stop and count event types in most function modes. Pass-through mode can also be used in switch events in the PWM/PWM_DT mode, if it selects the constant high as the source.

25.2.4 Glitch filtering

Glitch filtering is the process of removing unwanted pulses from a digital input signal that is usually high or low. Glitch filter can be inserted to the general-purpose triggers and specific one-to-one triggers. The glitch filter runs on clk_peri.

The glitch filter can be configured through the ALL_GF register for the glitch filter that is used for general-purpose triggers and ONE_GF register for glitch filter that is used for the one-to-one triggers. The ALL_GF register and ONE_GF register is the same design in hardware and has the same register configuration setup. They are named differently to differentiate the register naming for the glitch filter that is connected to general-purpose trigger or one-to-one triggers. Each instance of the glitch filter has its own register for independent configuration.

Through the combination setting of GF_DEPTH and GFPS_DIV, user can select the depth of the glitch filter from 0 to 512.

Figure 97 illustrates how the output of the glitch filter changes when the duration of the input of the glitch filter is equal or greater than glitch filter depth. Otherwise, the output is unchanged from its current value.

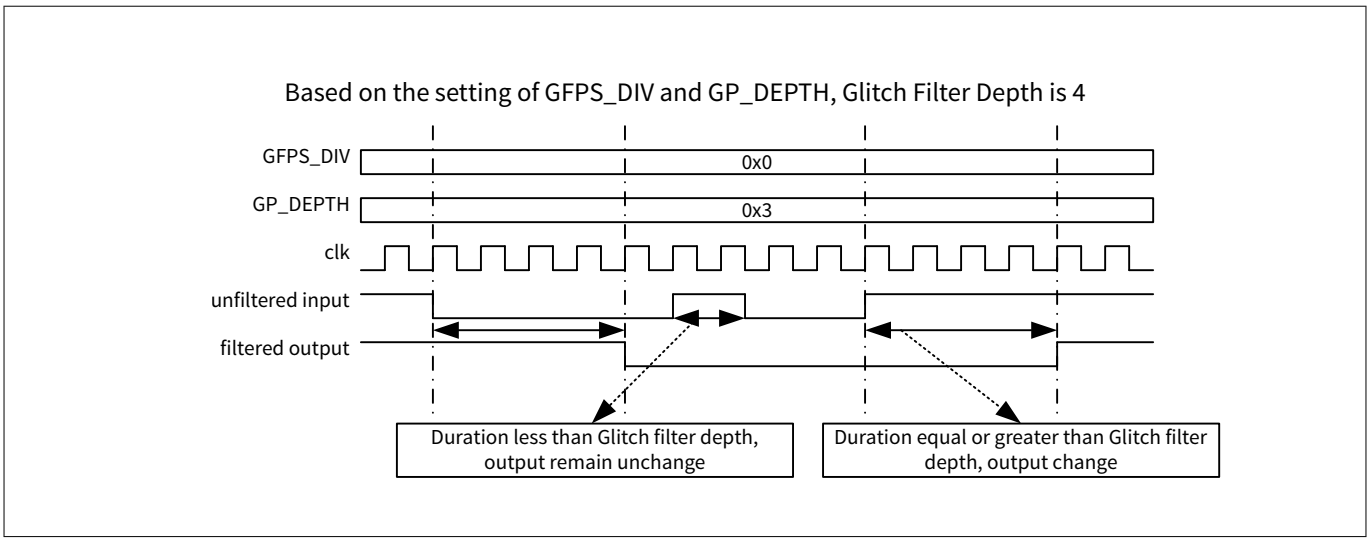


Figure 97 Glitch filter operation

Figure 219 illustrates an example on the connection of the glitch filter to general-purpose triggers and one-to-one triggers for two different TCPWM counter groups.

25.2.5 Synchronization of multiple counters

The previous sections described hardware-based event generation. In addition, software-based event generation is supported: the reload, start, stop, capture0, and capture1 events can be generated by writing to the PERI0_TR_CMD registers. These are counter-specific registers and allow software-based event generation only for a single counter. Synchronized software-based event generation (such as starting multiple counters synchronously) is possible by selecting the same trigger signal in all desired counters (via TR_IN_SEL0 and TR_IN_SEL1 registers) and generating a trigger by the PERI0_TR_CMD register in the PERI block.

The following figure shows an example of how to synchronously start multiple counters with TR_CMD of the PERI block.

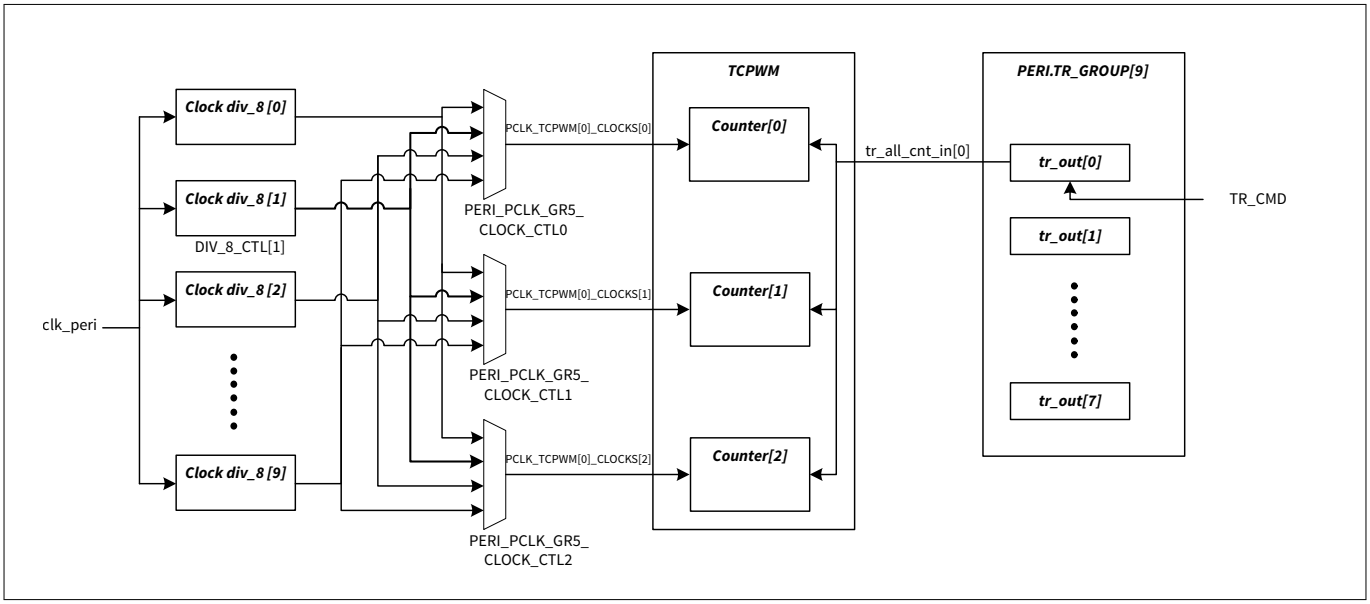


Figure 98 PERI TR_CMD synchronously starts counters

The following example describes the required steps to start counters synchronously:

- Configure PERI_PCLK_GR5_CLOCK_CTL0/1/2 to select the same clock divider clock_div_8[1]
- Configure DIV_8_CTL[1] and DIV_CMD to generate clock divider enable signal for TCPWM. Hence, three counters have the same clock divider enable signal; in other words, their clocks are synchronous
- Enable the three counters one by one - counter is enabled but will not run until the start or reload event is detected
- Configure the three counters start event, all selecting tr_all_cnt_in[0]; this means that the three counters will run synchronously when tr_all_cnt_in[0] asserts
- Use TR_CMD to generate a trigger pulse on TR_GROUP[9].TR_OUT[0]. All three counters will run synchronously

***Note**: All registers listed here belong to the PERI block (see clocking system chapter).*

- A software-based event is set after writing TR_CMD respective bit to '1', and cleared by hardware on the next "active count" pre-scaled counter clock. For some events in a specific mode, it is cleared on the next "active count" pre-scaled counter clocks that have a TC event. The event detection setting (TR_IN_EDGE_SEL) does not have an effect on a software-based event

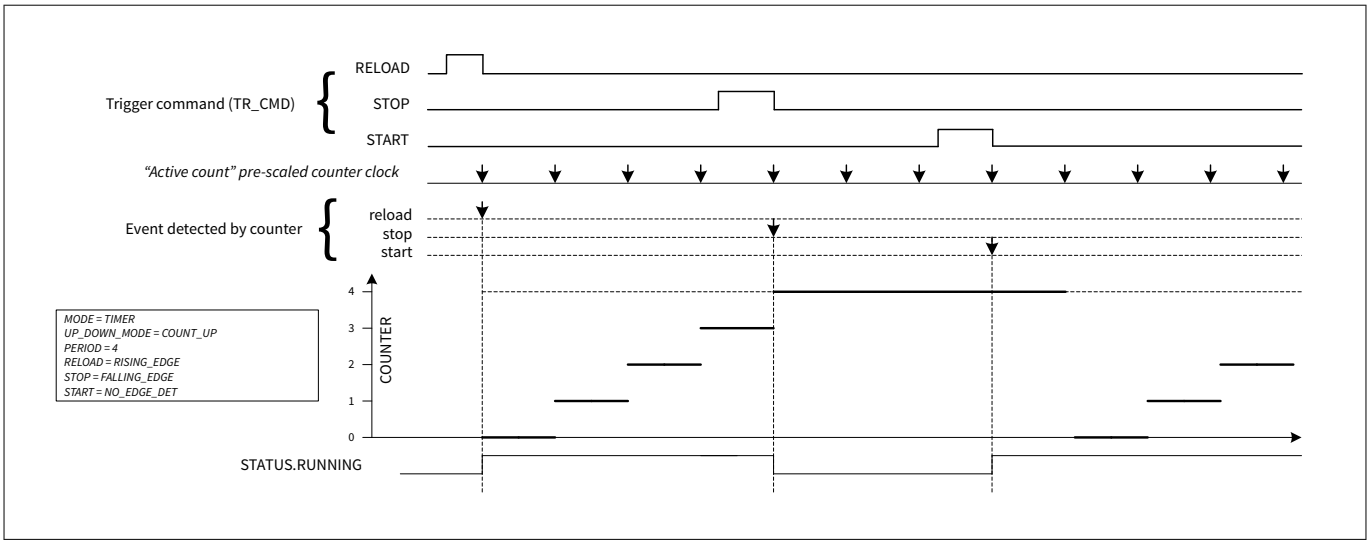


Figure 99 Software trigger command detection by active count pre-scaled counter clock

25.2.6 Trigger outputs

Each counter has two trigger output signals (TR_OUT0 and TR_OUT1) and they can be routed through the trigger mux to other peripherals on the device. The bit field OUT0 in the TR_OUT_SEL register is used to select one of the internal events to generate output trigger 0 (TR_OUT0), and the bit field OUT1 selects one of the internal events to generate TR_OUT1. Also, it allows disabling of the output triggers.

The following are the events indicated by the counter trigger output signal:

- Overflow (OV): An overflow event indicates that in up counting mode, COUNTER equals the PERIOD register, and is changed to a different value
- Underflow (UN): An underflow event indicates that in down counting mode, COUNTER equals 0, and changes to a different value
- TC (Terminal count): A TC event is the logical OR of the underflow and overflow events
- CC0/1_MATCH: This event is generated when the counter is running and one of the following conditions occurs:
  - Counter equals the compare value. This event is either generated when the match is about to occur (COUNTER does not equal CC0/1 and changes to CC0/1) or when the match is not about to occur (COUNTER equals CC0/1 and changes to a different value)
  - A capture event has occurred and the CC0 (CC1) and CC0_BUFF (CC1_BUFF) registers are updated

  ***Note**: CC1_MATCH event is only available when parameter GRP_CC1_PRESENT = 1*
- CC0_CC1_MATCH: This event is triggered when either CC0_MATCH or CC1_MATCH occurs
- LINE_OUT: A PWM output signal
- DISABLED: Output trigger is disabled

The selection of the events for the output trigger generation is done by the TR_OUT_SEL register. It also allows disabling the output triggers.

***Note**: These signals only remain high for two cycles of CLK_SYS. For reliable operation, the condition that causes this trigger should be a maximum of one quarter of the CLK_SYS. For example, if the CLK_SYS is running at 24 MHz, the condition causing the trigger should occur at a frequency equal to or less than 6 MHz.*

When LINE_OUT is selected for output triggers, the output trigger bypasses the two-cycle pulses generation logic and directly outputs LINE_OUT.

The generated triggers have two main uses:

- Initiating a DMA data transfer. For example, in the PWM mode with an up counting timer, the overflow can be used to transfer new period and compare values from memory to the counters' PERIOD_BUFF and CC0_BUFF registers
- Reconstruction of a PWM signal in a programmable digital component. As documented in Pulse-width modulation (PWM) mode, the PWM line output signal is derived from the cc0_match (cc1_match), underflow, and overflow internal events. By making these internal events available as output triggers, other components can reconstruct and potentially modify the PWM signal (note the mentioned frequency restrictions)

25.2.7 Internal events

25.2.7.1 Underflow event

An underflow event indicates that in down counting, the COUNTER equals zero, and changes to a different value. Reload also generates an underflow event in some specific mode. The following table summarizes the underflow generation of each function mode.

Table 188 Underflow generation

Mode	UP	DOWN	UPDN1	UPDN2
Timer	Counter is decremented and changes from a state in which COUNTER equals 0. Reload event in DOWN, UPDN1, and UPDN2 modes			
Capture	Counter is decremented and changes from a state in which COUNTER equals 0. Reload event in DOWN, UPDN1, and UPDN2 modes			
QUAD	• QUAD_RANGE0: Not used • QUAD_RANGE0_CMP: Not used • QUAD_RANGE1_CMP: Counter value COUNTER equals 0 and is decrementing • QUAD_RANGE1_CAPT: Counter value COUNTER equals 0 and is decrementing			
PWM	Counter is decremented and changes from a state in which COUNTER equals 0. Reload event in DOWN, UPDN1, and UPDN2 modes			
PWM_DT	Counter is decremented and changes from a state in which COUNTER equals 0. Reload event in DOWN, UPDN1, and UPDN2 modes			

25.2.7.2 Overflow event

An overflow event indicates that in up counting, the COUNTER equals PERIOD, and changes to a different value. Reload also generates an overflow event in some specific mode.

The following table summarizes the overflow generation of each function mode.

Table 189 Overflow generation

Mode	UP	DOWN	UPDN1	UPDN2
Timer	Counter is incrementing and changes from a state in which COUNTER equals PERIOD. Reload event in UP count mode			
Capture	Counter is incrementing and changes from a state in which COUNTER equals PERIOD. Reload event in UP count mode			
QUAD	• QUAD_RANGE0: Not used • QUAD_RANGE0_CMP: Not used • QUAD_RANGE1_CMP: Counter value COUNTER equals PERIOD and is incrementing • QUAD_RANGE1_CAPT: Counter value COUNTER equals PERIOD and is incrementing			
PWM	Counter is incrementing and changes from a state in which COUNTER equals PERIOD. Reload event in UP count mode			
PWM_DT	Counter is incrementing and changes from a state in which COUNTER equals PERIOD. Reload event in UP count mode			

25.2.7.3 Terminal count event

A terminal count (TC) event is the logical OR of the underflow and overflow events. An exception is that a reload event generates an underflow or overflow, but not a TC event. In quadrature mode, index will generate TC event.

The following table summarizes the TC generation of each function mode.

Table 190 TC generation

Mode	UP	DOWN	UPDN1	UPDN2
Timer	Overflow	Underflow	Underflow	Logic OR of overflow and underflow
Capture	Overflow	Underflow	Underflow	Logic OR of overflow and underflow
QUAD	QUAD_RANGE0: • Index event QUAD_RANGE0_CMP: • Counter value COUNTER equals 0 or 0xffff/0xffffffff in “wrap-around capture” mode • Index or capture0 event in “index capture” mode QUAD_RANGE1_CMP: • Counter value COUNTER equals 0 and is decrementing (underflow) or PERIOD and is incrementing (overflow) • Index event QUAD_RANGE1_CAPT: • Same as QUAD_RANGE1_CMP			
PWM	Overflow	Underflow	Underflow	Logic OR of overflow and underflow
PWM_DT	Overflow	Underflow	Underflow	Logic OR of overflow and underflow

25.2.7.4 cc0_match (cc1_match) event

A cc0_match event indicates that the COUNTER equals CC0. This event is either generated when the COUNTER is about to change to CC0, or when the COUNTER equals CC0 and is about to change to a different value. A special case is for 0 or 100 percent duty cycle generation in PWM mode; for more details, see the Pulse-width modulation (PWM) mode. In other specific operation modes, the event is used to indicate that the CC0/ CC0_BUFF registers are updated. cc1_match is generated based on the state of the COUNTER and CC1; other behaviors are the same as cc0_match.

The following tables summarize the cc0/1_match generation of each function mode.

Table 191 cc0_match generation

Mode	UP	DOWN	UPDN1	UPDN2
Timer	Counter changes from a state in which COUNTER equals CC0			
Capture	Capture0 event			
QUAD	QUAD_RANGE0: • Counter value COUNTER equals 0 or 0xffff • Index event QUAD_RANGE0_CMP: Counter changes to a state in which COUNTER equals CC0 QUAD_RANGE1_CMP: Same as QUAD_RANG0_CMP QUAD_RANGE1_CAPT: Capture0 event			
PWM	Counter changes to a state in which COUNTER equals CC0 or COUNTER passed the CCO Value Left Aligned Case: CC0 <= COUNTER Right Aligned Case: CC0>= COUNTER		COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC0 If advanced motor control features are present, CC0_MATCH_DOWN_EN/ CC0_MATCH_UP_EN will enable/disable cc0_match generation	
PWM_DT	Counter changes to a state in which COUNTER equals CC0 or COUNTER passed the CCO Value Left Aligned Case : CC0 <= COUNTER Right Aligned Case: CC0>= COUNTER		COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC0. If advanced motor control features are present in a counter group, CC0_MATCH_DOWN_EN/ CC0_MATCH_UP_EN will enable/disable cc0_match generation	

Table 192 cc1_match generation

Mode	UP	DOWN	UPDN1	UPDN2
Timer	Counter changes from a state in which COUNTER equals CC1			
Capture	Capture1 event			
QUAD	QUAD_RANGE0: Not used QUAD_RANGE0_CMP: Counter changes to a state in which COUNTER equals CC1 QUAD_RANGE1_CMP: Same as QUAD_RANG0_CMP QUAD_RANGE1_CAPT: Capture1 event			

(table continues...)

Table 192 (continued) cc1_match generation

Mode	UP	DOWN	UPDN1	UPDN2
PWM	Counter changes to a state in which COUNTER equals CC1 or COUNTER passed the CC1 Value. Left Aligned Case : CC1 <= COUNTER Right Aligned Case: CC1 >= COUNTER		COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC1. If advanced motor control features are present, CC1_MATCH_DOWN_EN/ CC1_MATCH_UP_EN will enable/disable cc1_match generation	
PWM_DT	Counter changes to a state in which COUNTER equals CC1 or COUNTER passed the CC1 Value. Left Aligned Case : CC1 <= COUNTER Right Aligned Case: CC1 >= COUNTER		COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC1. If advanced motor control features are present, CC1_MATCH_DOWN_EN/ CC1_MATCH_UP_EN will enable/disable cc1_match generation	

25.2.8 Interrupts

The TCPWM block provides a dedicated interrupt output for each counter. Interrupts are counter mode specific and can be generated for a terminal count (TC) or Compare/Capture0/1 (CC0/1) event. A TC is the logical OR of the OV and UN events.

Four registers are used to handle interrupts in this block, as shown in Table 193.

Table 193 Interrupt register

Interrupt registers	Bits	Name	Description
INTR(Interrupt request register)	0	TC	This bit is set to '1', when a terminal count is detected. Write '1' to clear this bit
	1	CC0_MATCH	This bit is set to '1' when the counter value matches the capture/ compare0 (CC0) register value. Write '1' to clear this bit
	2	CC1_MATCH	This bit is set to '1' when the counter value matches the capture/ compare1 (CC1) register value. Write '1' to clear this bit
INTR_SET(Interrupt set request register)	0	TC	Write '1' to set the corresponding bit in the interrupt request register. When read, this register reflects the interrupt request register status

(table continues...)

Table 193 (continued) Interrupt register

Interrupt registers	Bits	Name	Description
	1	CC0_MATCH	Write '1' to set the corresponding bit in the interrupt request register. When read, this register reflects the interrupt request register status
	2	CC1_MATCH	Write '1' to set the corresponding bit in the interrupt request register. When read, this register reflects the interrupt request register status
INTR_MASK(Interrupt mask register)	0	TC	Mask bit for the corresponding TC bit in the interrupt request register
	1	CC0_MATCH	Mask bit for the corresponding CC_MATCH0 bit in the interrupt request register
	2	CC1_MATCH	Mask bit for the corresponding CC_MATCH1 bit in the interrupt request register
INTR_MASKED(Interrupt masked request register)	0	TC	Logical AND of the corresponding TC request and mask bits
	1	CC0_MATCH	Logical AND of the corresponding CC_MATCH0 request and mask bits
	2	CC1_MATCH	Logical AND of the corresponding CC_MATCH1 request and mask bits

25.2.9 Debug mode

The TCPWM counters support debugging through CTRL.DBG_FREEZE_EN and CTRL.DBG_SUS_EN bits. It can be configured per counter if the counter operation continues or pauses in debug state (for example, after running to a break point). This feature is especially intended when using a TCPWM counter as an OS timer. It is realized by gating the counter clock “clk_counter” when entering debug state while DBG_FREEZE_EN = ‘1’.

The table tabulates the behavior of the debug mode based on CTRL.DBG_FREEZE_EN and CTRL.DBG_SUS_EN bits.

Table 194 Debug mode

	DBG_FREEZE_EN	DBG_SUS_EN
Counter operation continues	0	0
Counter operation pauses immediately	1	0
Counter operation pauses after tc event and enter passive state. Passive state is is controlled through CTRL.KILL_LINE_POLARITY	0	1
Counter operation pauses immediately and enters passive state. Passive state is is controlled through CTRL.KILL_LINE_POLARITY. STATUS.RUNNING bit set to 0	1	1

***Note**: When freeze TCPWM in debug mode, it is necessary to set not only DBG_FREEZE_EN and DBG_SUS_EN but also the signal (tr_debug_freeze). The tr_debug_freeze is from the DEBUG600 System Cross Trigger Interface (CTI).*

***Note**: The freeze and suspend functions can be enabled only in the debug session active; hence, check whether the debug session is active through the DEBUG_SESSION bit of the PWR_CTL register before enabling.*

25.2.10 PWM outputs

The PWM, PWM_DT operation modes produce two output signals:

- A PWM LINE_OUT output signal
- A complementary PWM LINE_COMPL_OUT output signal (inverted version of LINE_OUT)

***Note**: In PWM and PWM_DT modes, the CC0_match, CC1_match, underflow, and overflow internal event conditions are used to drive LINE_OUT and LINE_COMPL_OUT, by configuring the TR_PWM_CTRL register (see Table 195).*

Table 195 Configuring output line for OV, UN, and CC0/1 conditions

Field	Bit	Value	Event	Description
CC0_MATCH_MODE Default Value = 3	1:0	0	Set LINE_OUT to '1	Configures output line on a compare match (CC0) event
		1	Clear LINE_OUT to '0	
		2	Invert LINE_OUT	
		3	No change	
OVERFLOW_MODE Default Value = 3	3:2	0	Set LINE_OUT to '1	Configures output line on an overflow (OV) event
		1	Clear LINE_OUT to '0	
		2	Invert LINE_OUT	

(table continues...)

Table 195 (continued) Configuring output line for OV, UN, and CC0/1 conditions

Field	Bit	Value	Event	Description
		3	No change	
UNDERFLOW_MODE Default Value = 3	5:4	0	Set LINE_OUT to '1	Configures output line on an underflow (UN) event
		1	Clear LINE_OUT to '0	
		2	Invert LINE_OUT	
		3	No change	
CC1_MATCH_MODE Default Value = 3	7:6	0	Set LINE_OUT to '1	Configures output line on a compare match (CC1) event
		1	Clear LINE_OUT to '0	
		2	Invert LINE_OUT	
		3	No change	

The generation of PWM output signals is a multi-step process. Both LINE_OUT and LINE_COMPL_OUT are generated from the PWM signal line. The PWM signal line is generated as per the state of cc0_match, cc1_match, underflow, and overflow internal events, as specified by the counter's TR_PWM_CTRL register. For each internal event, the TR_PWM_CTRL register specifies how the event affects the output LINE_OUT.

- The output is set to '0'
- The output is set to '1'
- The output is inverted
- The output is not affected

If the internal cc0_match event generates at the same time when an internal underflow or overflow event generates, cc0_match takes effect after LINE_OUT changes state per settings of underflow/overflow. cc1_match takes effect after cc0_match. Table 196 lists some examples to show the mechanism.

Table 196 LINE_OUT construction example

Coincide case	Overflow	Underflow	CC0_match	CC1_match	LINE_OUT
CC0_match and overflow	CLEAR	Don't care	INVERT	Don't care	1 (SET)
CC0_match and underflow	Don't care	SET	INVERT	Don't care	0 (CLEAR)
CC0_match and CC1_match	Don't care	Don't care	SET	CLEAR	CLEAR
CC0_match and CC1_match and overflow	INVERT	Don't care	INVERT	INVERT	INVERT

25.2.11 Power modes

The TCPWM block works in ACTIVE, LPACTIVE, SLEEP, and LPSLEEP modes. The TCPWM block is connected to a digital core power supply. The retention registers will be still powered in DEEPSLEEP mode to keep state for the MMIO registers, which needs to retain the value.

In ACTIVE, LPACTIVE, SLEEP, and LPSLEEP power modes, clk_peri is provided. In DEEPSLEEP mode, there is no clk_peri running.

The vccact is cut off in HIBERNATE and DEEPSLEEP modes, but supplied in ACTIVE/LPACTIVE/SLEEP/LPSLEEP modes.

The vccret is supplied in DEEPSLEEP mode for retention registers and off in HIBERNATE mode.

There is no power switch integrated in the block.

Table 197 Power modes in TCPWM block

Power mode	Block status
ACTIVE	This block is fully operational in this mode with the clock running with power on.
LPACTIVE	LPACTIVE (LOWPOWERACTIVE) is similar to ACTIVE, with some tradeoffs applied to reduce current.
SLEEP	CPU is sleep in this mode but the function is still on. All counter clocks are ON.
LPSLEEP	LPSLEEP (LOWPOWERSLEEP) relates to LPACTIVE in the same way that SLEEP relates to ACTIVE.
DEEPSLEEP	In this mode, the vccact power to this block is off and no clk_peri is provided, hence, the logic is not functional. All retention registers will keep their state with power vccret.
HIBERNATE	In this mode, the entire block is powered off and no clk_peri is provided.

25.3 Counter operation modes

The counter block can function in four operational modes, as shown in Table 198. The MODE [26:24] field of the counter control register (CTRL) configures the counter in the specific operational mode.

Table 198 Operational mode configuration

Mode	MODE field [26:24]	Description
Timer	000	The counter increments or decrements by '1' at every counter clock cycle in which a count event is detected. The Compare/Capture register is used to compare the count.
Capture	010	The counter increments or decrements by '1' at every counter clock cycle in which a count event is detected. A capture event copies the counter value into the capture register.
PWM	100	Pulse width modulation (PWM) mode. HRPWM feature is supported in this mode through HRPWM_CTRL register.

(table continues...)

Table 198 (continued) Operational mode configuration

Mode	MODE field [26:24]	Description
PWM_DT	101	Pulse width modulation with dead time insertion. HRPWM feature is supported in this mode through HRPWM_CTRL register.

The counter can be configured to count up, down, and up/down by setting the UP_DOWN_MODE [17:16] field in the CTRL register, as shown in Table 199.

Table 199 Counting mode configuration

Counting modes	UP_DOWN_MODE[17:16]	Description
UP counting mode	00	Increments the counter until the period value is reached. A Terminal Count (TC) condition is generated when the counter changes from the period value.
DOWN counting mode	01	Decrements the counter from the period value until 0 is reached. A TC condition is generated when the counter changes from a value of '0'.
UP/DOWN counting mode 1	10	Increments the counter until the period value is reached, and then decrements the counter until '0' is reached. A TC condition is generated only when the counter changes from a value of '0'.
UP/DOWN counting mode 2	11	Similar to up/down counting mode 1 but a TC condition is generated when the counter changes from '0' and when the counter value changes from the period value.

25.3.1 Timer mode

The timer mode can be used to measure how long an event takes or the time difference between two events. The timer functionality increments/decrements a counter between 0 and the value stored in the PERIOD register. When the counter is running, the count value stored in the COUNTER register is compared with the compare/capture register (CC0 and CC1). When the COUNTER equals CC0, the cc0_match event is generated, even-handedly when the COUNTER equals CC1, the cc1_match event is generated.

The timer functionality is typically used for one of the following:

- Timing a specific delay - the count event is a constant '1'
- Counting the occurrence of a specific event - the event should be connected as an input trigger and selected for the count event

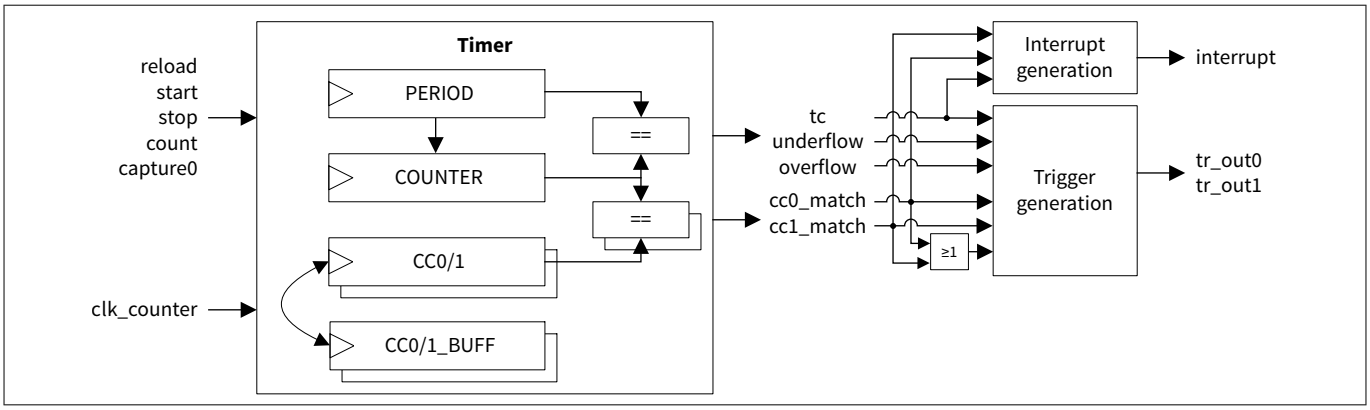


Figure 100 Timer mode function diagram

Note:
- *The triggers tr_out0 and tr_out1 are generated based on the internal events cc0_match, cc1_match, OR combination of cc0_match and cc1_match, underflow, overflow, and tc respectively (selection done by register TR_OUT_SEL)*
- *The timer functionality only uses the PERIOD register (and not PERIOD_BUFF)*
- *It is not recommended to write to COUNTER when the counter is running*
- *In the COUNT_UPDN2 mode,* ***software should only change PERIOD for a TC condition when COUNTER is "0"*** *(and not when COUNTER is PERIOD)*

Table 200 Timer mode trigger input description

Trigger inputs	Usage
Reload	Initializes and starts the counter. Behavior is dependent on UP_DOWN_MODE: • COUNT_UP: The counter is set to '0' and the count direction is set to 'up' • COUNT_DOWN: The counter is set to PERIOD and the count direction is set to 'down' • COUNT_UPDN1/2: The counter is set to '1' and the count direction is set to 'up' Can be used when the counter is running or not running.
Start	Starts the counter. The counter is not initialized by the hardware. The current counter value is used. Behavior is dependent on UP_DOWN_MODE. When the counter is not running: • COUNT_UP: The count direction is set to 'up' • COUNT_DOWN: The count direction is set to 'down' • COUNT_UPDN1/2: The count direction is not modified Note that when the counter is running, the start event has no effect and can be used when the counter is running or not running.
Stop	Stops the counter.

(table continues...)

Table 200 (continued) Timer mode trigger input description

Trigger inputs	Usage
Count	Count event increments/decrements the counter.
Capture0	This event is used as an external up/down counting event. External direction control mode is enabled via QUAD_ENCODING_MODE[21:20] field of CTRL register: • EXT_DIR_FALLING: Counter changes current count direction based on falling edge of capture0 input • EXT_DIR_RISING: Counter changes current count direction based on rising edge of capture0 input • EXT_DIR_LEVEL: Counter count increase when capture0 level equal to high, counter count decrease when capture0 level equals to low
Capture1	Not used.

Incrementing and decrementing the counter is controlled by the count event and the clk_counter. Typical operation uses a constant '1' count event and clk_counter without pre-scaling. Advanced operations are also possible; for example, the counter event configuration can decide to count the rising edges of a synchronized input trigger.

Table 201 Timer mode supported features

Supported features	Description
Clock pre-scaling	Pre-scales the clk_counter.
One shot	Counter is stopped by hardware on a tc event. In COUNT_UPDN2, the counter is stopped on tc event when underflow.
Auto reload CC	CC0 and CC0_BUFF are exchanged on a cc0_match event when CTRL.SWAP_ENABLE = “1” (when specified by CTRL.AUTO_RELOAD_CC, no input event is required). CC1 and CC1_BUFF are exchanged on a cc1_match event when CTRL.SWAP_ENABLE = “1” (when specified by CTRL.AUTO_RELOAD_CC1, no input event is required). CC0_BUFF are transferred to CC0 on a cc0_match event when CTRL.SWAP_ENABLE = “0” (when specified by CTRL.AUTO_RELOAD_CC0, no input event is required). CC1_BUFF are transferred to CC1 on a cc1_match event when CTRL.SWAP_ENABLE = “0” (when specified by CTRL.AUTO_RELOAD_CC1).

(table continues...)

Table 201 (continued) Timer mode supported features

Supported features	Description
Up/down modes	Specified by UP_DOWN_MODE: • COUNT_UP: The counter counts from 0 to PERIOD • COUNT_DOWN: The counter counts from PERIOD to 0 • COUNT_UPDN1/2: The counter counts from 1 to PERIOD and back to 0

Table 202 lists the trigger outputs and the conditions when they are triggered.

Table 202 Timer mode trigger outputs

Trigger outputs	Description
cc0_match	Counter changes from a state in which COUNTER equals CC0.
cc1_match	Counter changes from a state in which COUNTER equals CC1.
Underflow (UN)	Counter is decremented and changes from a state in which COUNTER equals 0. Reload event generates underflow in COUNT_DOWN, COUNT_UPDN1 or COUNT_UPDN2 mode.
Overflow (OV)	Counter is incrementing and changes from a state in which COUNTER equals PERIOD. Reload event generates overflow in COUNT_UP mode.
TC	Specified by UP_DOWN_MODE: • COUNT_UP: tc event is the same as the overflow event • COUNT_DOWN: tc event is the same as the underflow event • COUNT_UPDN1: tc event is the same as the underflow event • COUNT_UPDN2: tc event is the same as the logical OR of the overflow and underflow events Reload will generate underflow/overflow, but not generate tc.

Table 203 Timer mode PWM outputs

PWM outputs	Description
LINE_OUT	Not used.
LINE_COMPL_OUT	Not used.

25.3.1.1 Up/Down Count Modes

Up/down count modes control the counting direction (increment or decrement) while counter is running.

Figure 101 illustrates a timer in COUNT_UP mode. The counter is initialized (to 0) and started with a software-based reload event.

***Note**:*
- *When the counter changes from a state in which COUNTER is 4, an overflow and tc event are generated*
- *When the counter changes from a state in which COUNTER is 2, a cc0_match event generated*
- *PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 clk_counter periods*

A constant count event of '1' and clk_counter without pre-scaling is used in the following scenarios. If the count event is '0' and a reload event is triggered, the reload is only registered on the first clock edge when the count event is '1'. This means that the first clock edge when the count event is '1' is not used for counting. It is used for reload.

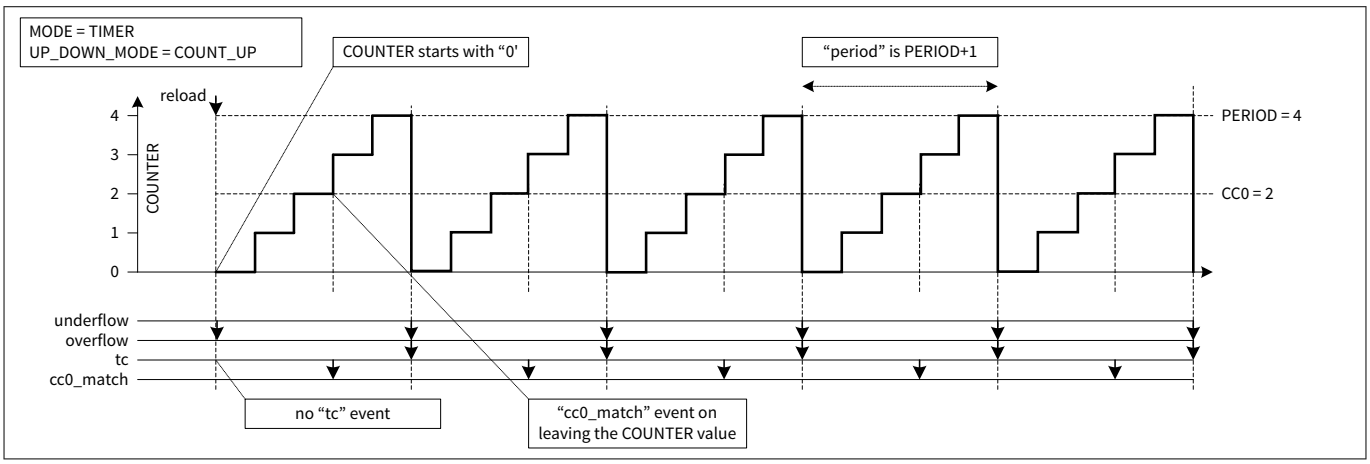


Figure 101 Timer in COUNT_UP mode

Figure 102 illustrates a timer in COUNT_DOWN mode. The counter is initialized (to PERIOD) and started with a software-based reload event.

***Note**:*
- *When the counter changes from a state in which COUNTER is 0, an underflow and tc event are generated*
- *When the counter changes from a state in which COUNTER equals 2, a cc0_match event is generated*
- *PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods*

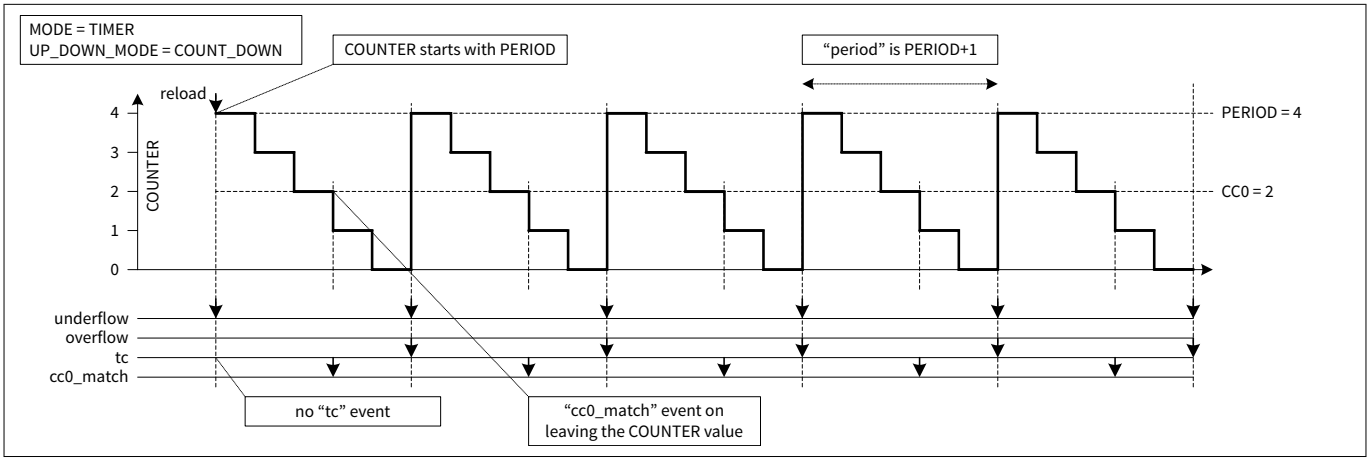


Figure 102 Timer in COUNT_DOWN mode

Figure 103 illustrates a timer in COUNT_UPDN1 mode. The counter is initialized (to 1) and started with a software-based reload event.

Note:
- *When the counter changes from a state in which COUNTER is 4, an overflow is generated*
- *When the counter changes from a state in which COUNTER is 0, an underflow and tc event are generated*
- *When the counter changes from a state in which COUNTER equals 2, a cc0_match event is generated*
- *PERIOD is 4, resulting in an effective repeating counter pattern of 2*4 = 8 counter clock periods*

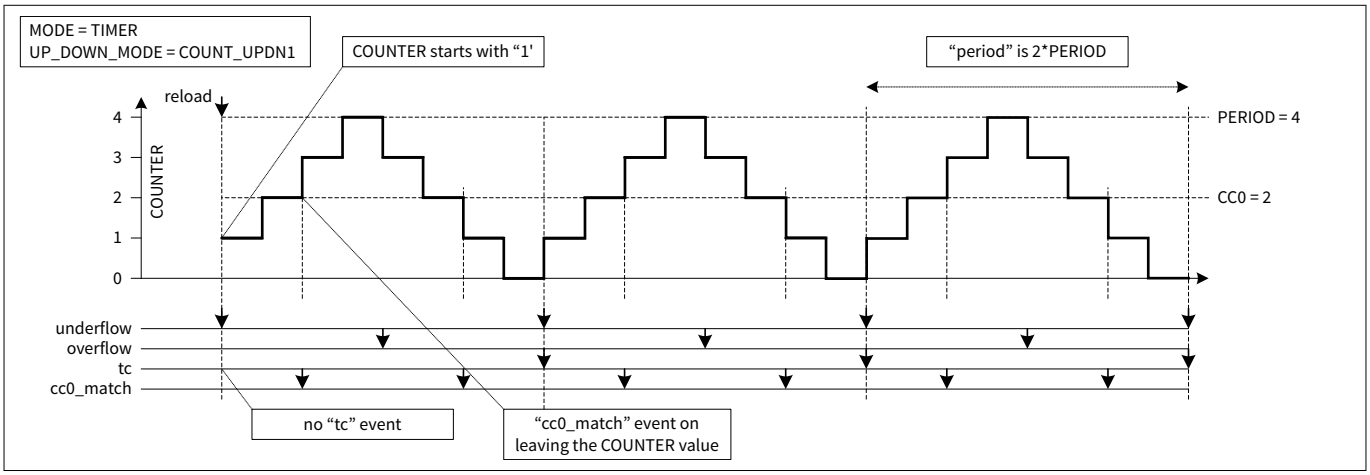


Figure 103 Timer in COUNT_UPDN1 mode

COUNT_UPDN2 has the same behavior with COUNT_UPDN1, except for the tc event that will generate with an overflow event.

25.3.1.2 One Shot Mode

Figure 104 illustrates a timer in "one-shot" operation mode.

Note:
- *The counter is stopped on a tc event*
- *In COUNT_UPDN2, the counter is stopped on tc event when underflow*
- *COUNTER will stay at the next value after tc. In the following figures, the counter stays at "0" after it stops running*

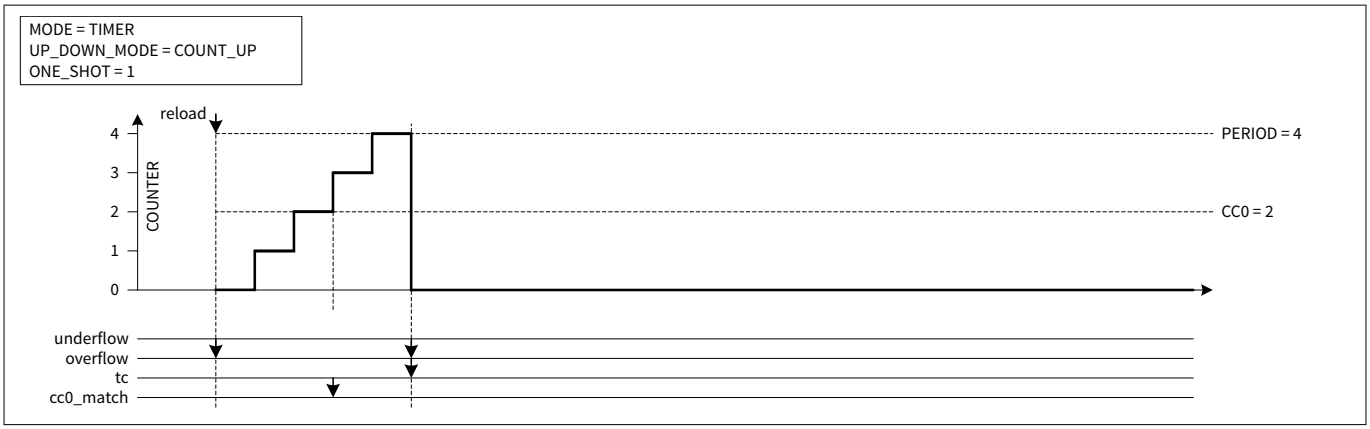


Figure 104 Timer in one-shot mode

25.3.1.3 Clock Pre-scaling

The supported pre-scaler values are: “/1”, “/2”, “/4”, “/8”, “/16”, “/32”, “/64”, “/128”, Figure 105 illustrates clock pre-scaling.

Note: • *The counter is only incremented every other counter cycle*

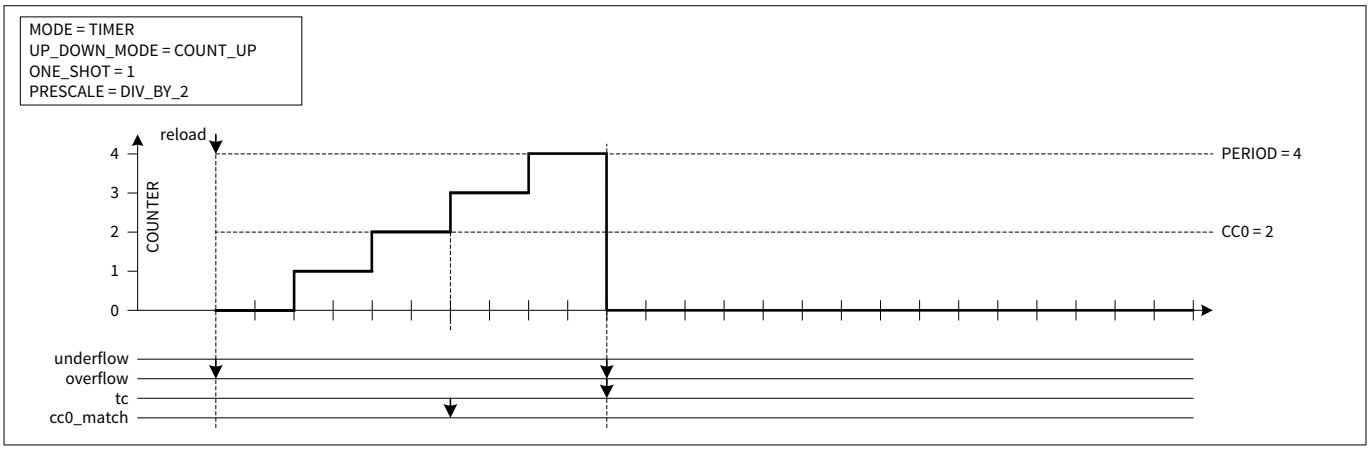


Figure 105 Timer clock pre-scaling

25.3.1.4 CC Auto Reload

When the cc0_match event is asserted, set the control bits CTRL.AUTO_RELOAD_CC0 and CTRL.SWAP_ENABLE to '1' to enable the hardware to automatically swap the CC0 and CC0_BUFF registers without requiring any additional input event.

Figure 106 illustrates a timer that uses both CC0 and CC0_BUFF registers.

Note: *CC0 and CC0_BUFF are exchanged on a cc0_match event.*

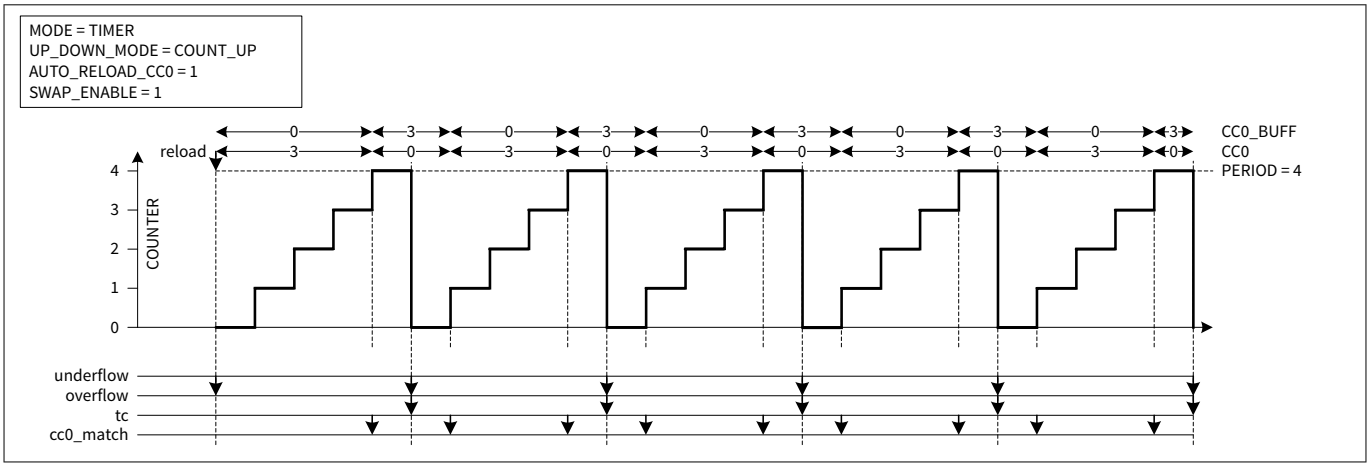


Figure 106 CC0/CC0_BUFF auto switch at cc0_match when SWAP_ENABLE = “1”

If CTRL.SWAP_ENABLE is “0”, the CC0_BUFF value is transferred to CC0 by HW once the cc0_match event is asserted (no input event is required to trigger the exchange).

Figure 107 illustrates a timer that uses both CC0 and CC0_BUFF registers when SWAP_ENABLE = "0".

Note: *CC0_BUFF is transferred to CC0 on a cc0_match event.*

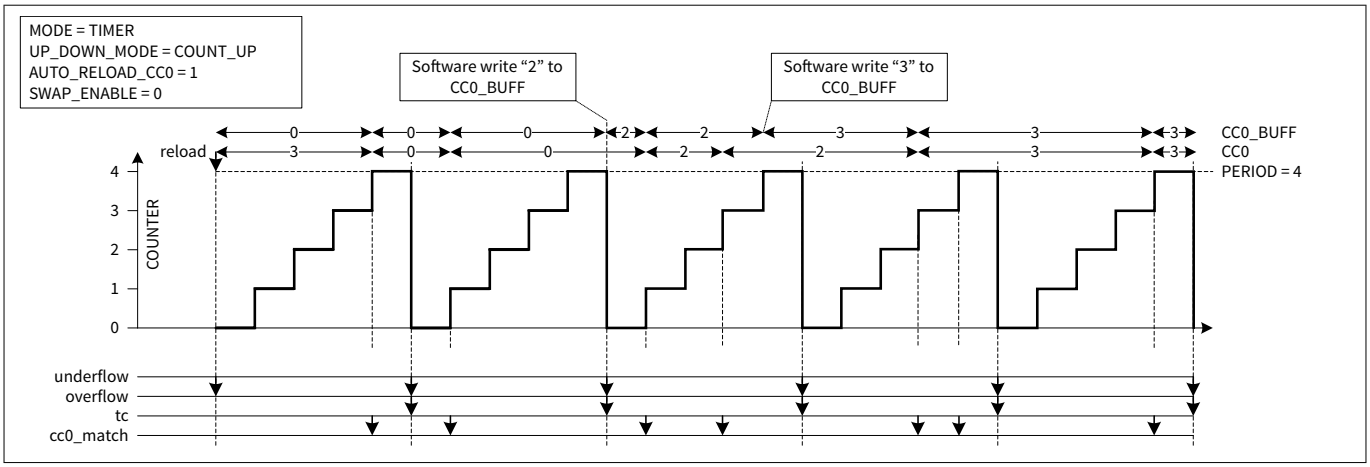


Figure 107 CC0_BUFF auto switch to CC0 at cc0_match with SWAP_ENABLE = “0”

The above description is applicable for CC1/CC1_BUFF.

25.3.1.5 External Input Event

Timer mode supports reload/start/stop/count input event.

Figure 108 illustrates a counter that is initialized and started (reload event), stopped (stop event) and continued/started (start event).

***Note**: The counter does not change value when it is NOT running (STATUS.RUNNING).*

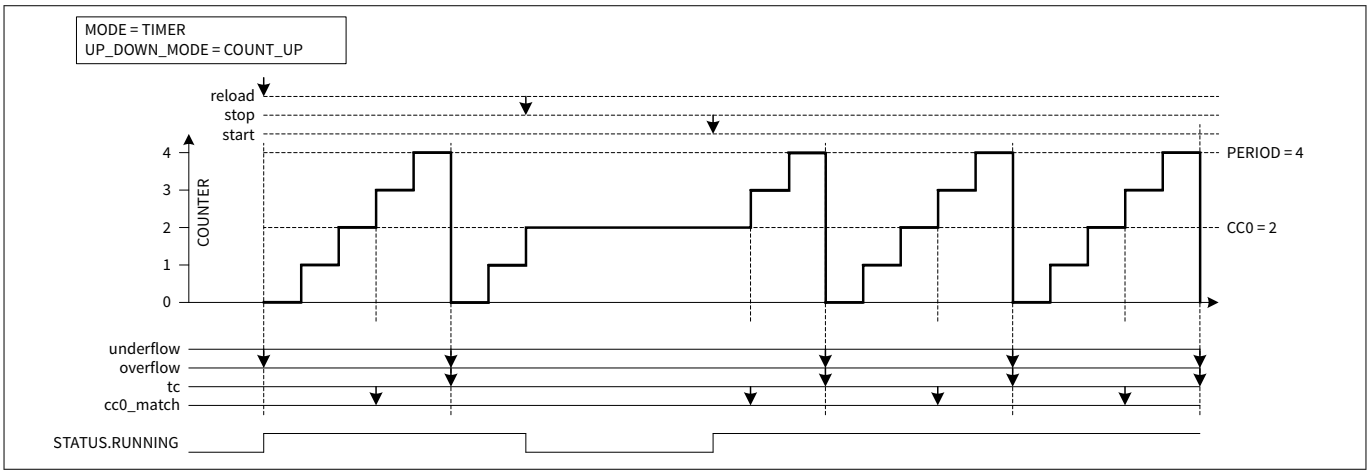


Figure 108 Counter start/stopped/continued

Reload can input when counter is running. The following description will clarify the behavior when reload coincides with tc event, cc0_match event or stop event.

- A stop event takes precedence over reload event
- Reload event generation takes precedence over “running counter” functionality (for underflow, overflow and tc event generation)
- Reload event determines:
- - underflow event generation

- overflow event generation
- tc event generation (always ‘0’)

• “Running counter” functionality determines:
- underflow event generation
- overflow event generation
- tc event generation
- cc0_match event generation
- CC0/CC0_BUFF exchange

Figure 109 illustrates a count event that becomes low while counter is running, and a rising edge detection stop event asserts when count event is low (the case is count event is used as pass through mode). The COUNTER will stop run and hold value when count event is low. The stop event will not change running status to low until the count event becomes high (“active count” pre-scaled counter clock).

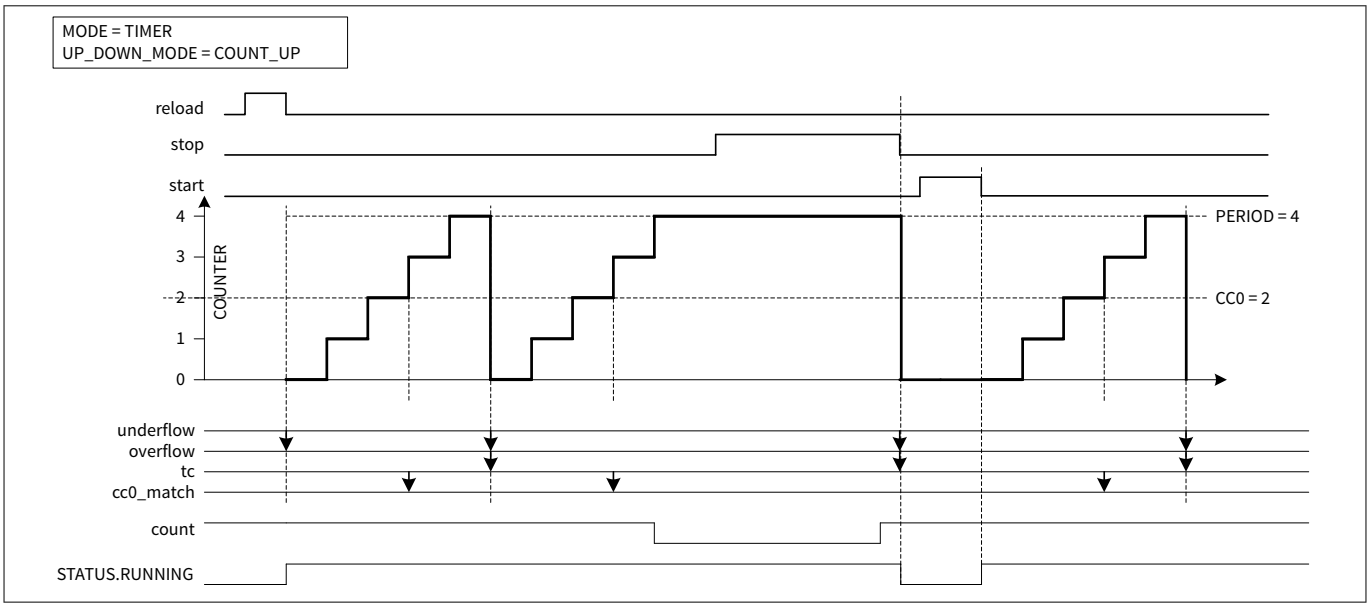


Figure 109 Function of count event

25.3.1.6 External Direction Control

Timer mode supports external direction control feature through the CTRL.QUAD_ENCODING_MODE bits. It uses the capture0 events to determine the up or down counting event. To disable this feature, set the CTRL.QUAD_ENCODING_MODE bits to '0'.

Configure the counter to change its current count direction based on the falling, rising, or level of the capture0 input. The following diagrams show the example how the counter changes current count direction based on falling edge, rising edge, and signal level in timer mode.

Note:
- *Set TR_IN_EDGE_SEL.CAPTURE0_EDGE to “3” (NO_EDGE_DET) to use this feature*
- *Use this feature only with the COUNT_UP/COUNT_DOWN mode in CTRL.UP_DOWN_MODE. Setting COUNT_UPDN1 or COUNT_UPDN2 in CTRL.UP_DOWN_MODE will result in unexpected behavior*

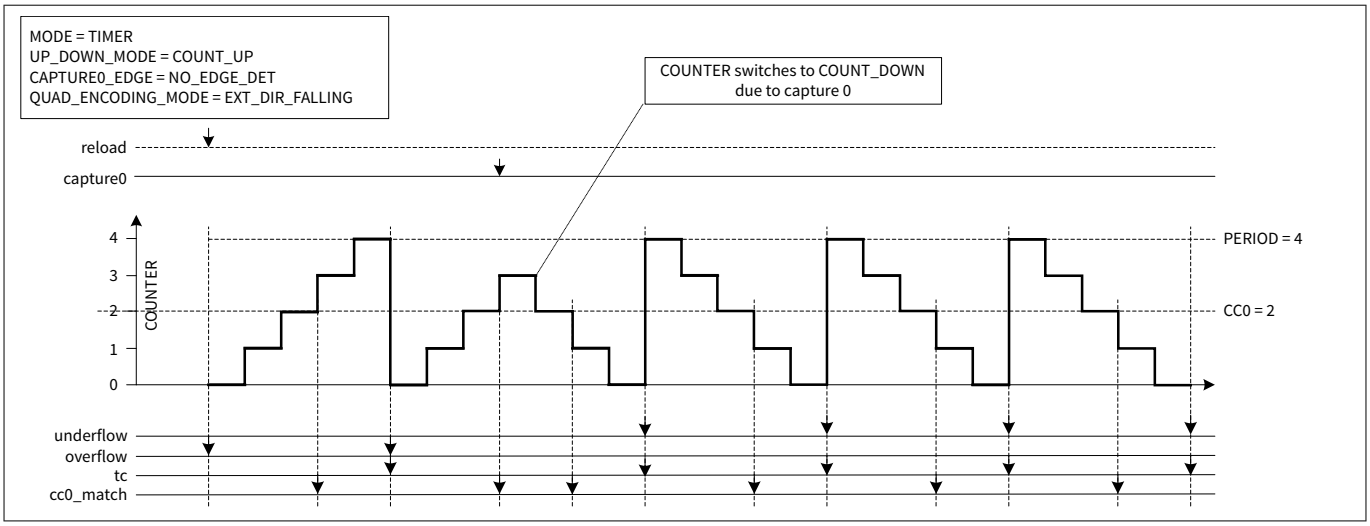


Figure 110 External direction control based on EXT_DIR_FALLING setting

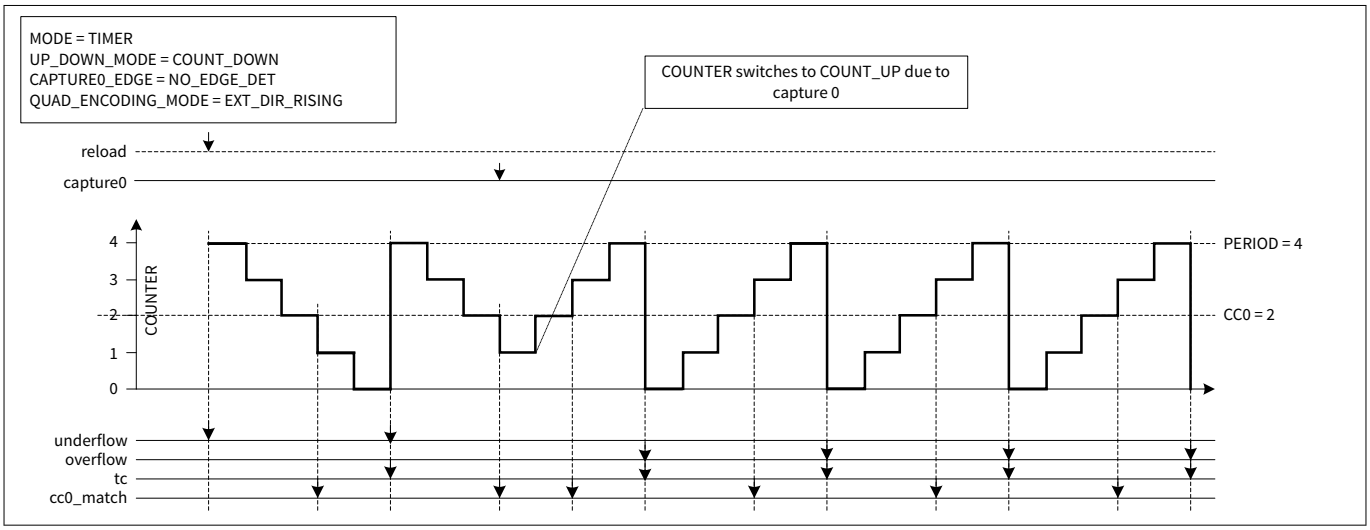


Figure 111 External direction control based on EXT_DIR_RISING setting

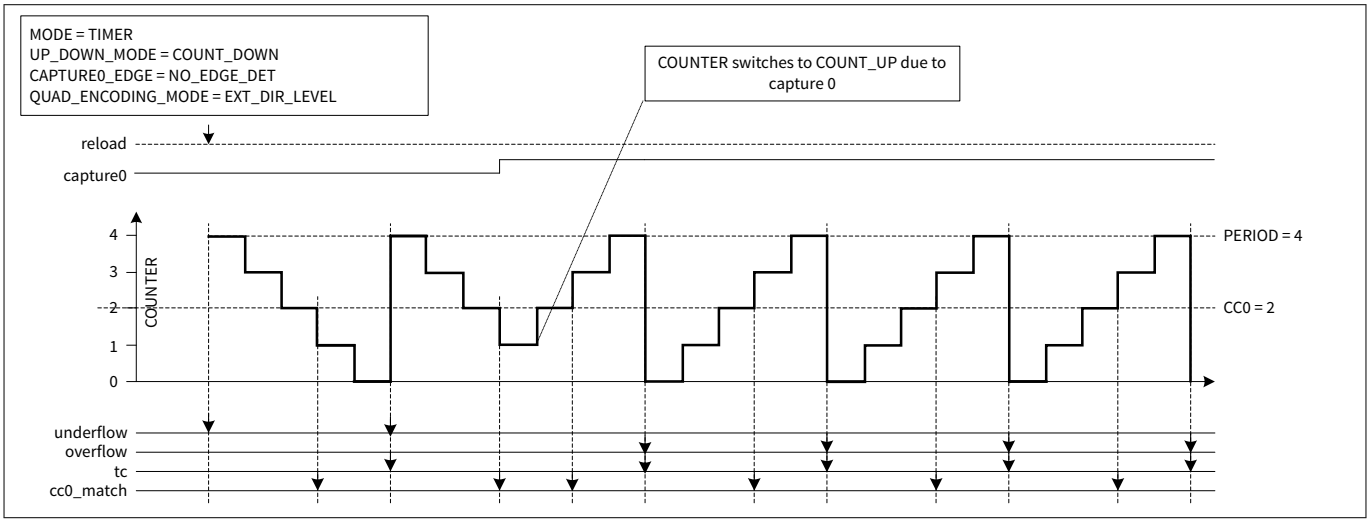


Figure 112 External direction control based on EXT_DIR_LEVEL setting

To trigger a direction change in the counter, use the software capture trigger through TR_CMD.CAPTURE0. When operating in Timer mode, set CTRL.QUAD_ENCODING_MODE to EXT_DIR_RISING. After the direction change takes effect, the hardware will automatically clear the TR_CMD.CAPTURE0 bit to '0'.

25.3.1.7 How to configure counter for timer mode

The steps to configure the counter for Timer mode of operation and the affected register bits are as follows:

- Disable the counter by writing '0' to the ENABLE bit of the CTRL register
- Select Timer mode by writing '000' to the MODE[26:24] field of the CTRL register
- Set the required 16- or 32-bit period in the PERIOD register
- Set the 16- or 32-bit compare value in the CC0 register and the buffer compare value in the CC0_BUFF register
- Set the AUTO_RELOAD_CC0 field of the CTRL register, if required to switch values at every CC condition
- Set clock pre-scaling by writing to the PS_DIV [2:0] field of the PS register
- Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the CTRL register
- The timer can be configured to run either in continuous mode or one-shot mode by writing 0 or 1, respectively, to the ONE_SHOT[18] field of the CTRL register
- Set the TR_IN_SEL0 or TR_IN_SEL1 register to select the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
- Set the TR_IN_EDGE_SEL register to select the edge of the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
- If required, set the interrupt upon TC or CC0_MATCH or CC1_MATCH condition
- Enable the counter by writing '1' to the ENABLED bit of the CTRL register. A reload trigger must be provided through the firmware(RELOAD bit in the PERI0_TR_CMD register) to start the counter if the hardware reload signal is not enabled

25.3.2 Capture mode

Capture functionality increments/decrements a counter between 0 and PERIOD. While counting, the capture0 event is observed. When the capture0 event is activated the counter value COUNTER is copied to CC0 (and CC0 is copied to CC0_BUFF). STATUS.CC0_READ_MISS bit will indicate whether data in the CC0 and CC0_BUFF are being overwritten before read by software. The STATUS.CC0_READ_MISS will be set to ‘1’, when past capture event has been overwritten by new data from COUNTER before software read CC0 and CC0_BUFF. STATUS.CCO_READ_MISS will be clear to ‘0’ once there is a read operation on either CC0 or CC0_BUFF.

The capture1 event and STATUS.CC1_READ_MISS bit are present. If so, when the capture1 event is activated the counter value COUNTER is copied to CC1 (and CC1 is copied to CC1_BUFF).

Capture functionality can be used to measure the width of a pulse (connected as one of the input triggers and used as capture0 or capture1 event).

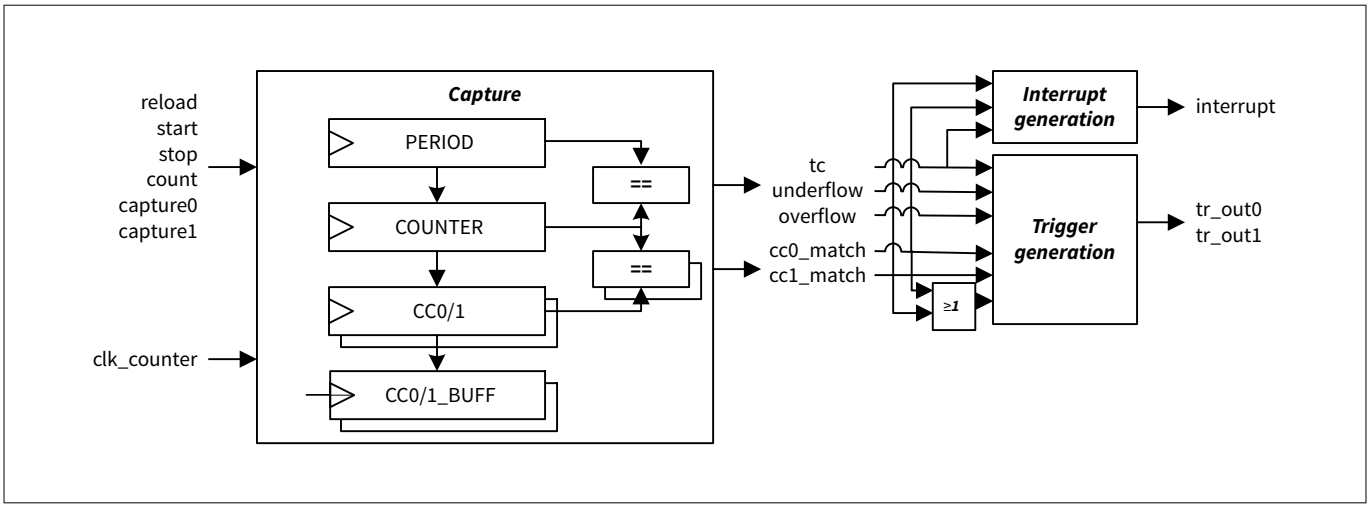


Figure 113 Capture mode function diagram

Table 204 Capture mode trigger input description

Generated events	Usage
Reload	Initializes and starts the counter. **Can be used while the counter is running or not running.** Initialization is dependent on UP_DOWN_MODE: • COUNT_UP: The counter is initialized to “0” • COUNT_DOWN: counter is initialized to PERIOD COUNT_UPDN1/2: The counter is initialized to “1”
Start	Starts the counter (thecounter is NOT initialized by HW). Can be used when the counter is running or not running. When the counter is running, the start event has no effect.
Stop	Stops the counter.
Count	Count event increments/decrements the counter per settings of up/down modes.
Capture0	Copies the counter value to CC0 and copies CC0 to CC0_BUFF.
Capture1	Copies the counter value to CC1 and copies CC1 to CC1_BUFF.

Table 205 Supported features of capture

Supported features	Description
Clock pre-scaling	Pre-scales the counter clock “clk_counter”.
One shot	Counter is stopped by HW, on a tc event. In COUNT_UPDN2, the counter is stopped on tc event when underflow.

(table continues...)

Table 205 (continued) Supported features of capture

Supported features	Description
Up/down modes	Specified by UP_DOWN_MODE: • COUNT_UP: counter counts from 0 to PERIOD • COUNT_DOWN: counter counts from PERIOD to 0 COUNT_UPDN1/2: counter counts from 1 to PERIOD and back to 0.

Table 206 Internal events of capture

Internal events	Description
cc0_match	CC0 is copied to CC0_BUFF and the counter value is copied to CC0 (cc0_match equals capture0 event).
cc1_match	CC1 is copied to CC1_BUFF and the counter value is copied to CC1 (cc1_match equals capture1 event).
Underflow	Counter is decremented and changes from a state in which COUNTER equals “0”. Reload event generates underflow in COUNT_DOWN, COUNT_UPDN1 or COUNT_UPDN2 mode.
Overflow	Counter is incrementing and changes from a state in which COUNTER equals PERIOD. Reload event generates overflow in COUNT_UP mode.
tc	Specified by UP_DOWN_MODE: • COUNT_UP: tc event is the same as the overflow event • COUNT_DOWN: tc event is the same as the underflow event • COUNT_UPDN1: tc event is the same as the underflow event • COUNT_UPDN2: tc event is the same as the logical OR of the overflow and underflow events Reload will generate underflow/overflow, but not generate tc.

Table 207 Capture mode line outputs

PWM outputs	Description
line_out	Not used.
line_compl_out	Not used.

25.3.2.1 Capture COUNTER value into CC0/CC0_BUFF

When the capture0 event is detected, the COUNTER value will be copied into the CC0 register, meanwhile the value of CC0 register will be copied into CC0_BUFF register. This feature can be used to measure the width of a pulse (connected as one of the input triggers and used as capture0 event with rising/falling edge detection mode). The following figure illustrates capture behavior in the COUNT_UP mode.

***Note**:*

- *Capture0 event detection uses rising edge detection. As a result, the capture0 event is remembered till the next "active count" pre-scaled counter clock*
- *When a capture0 event occurs, COUNTER is copied into CC0. CC0 is copied to CC0_BUFF*
- *A cc0_match event is generated when the counter value is captured*
- *CC0_READ_MISS will be set to '1' if software never performs a read before the third capture event. It will be cleared to '0' once software performs a read on either CC0 or CC0_BUFF*

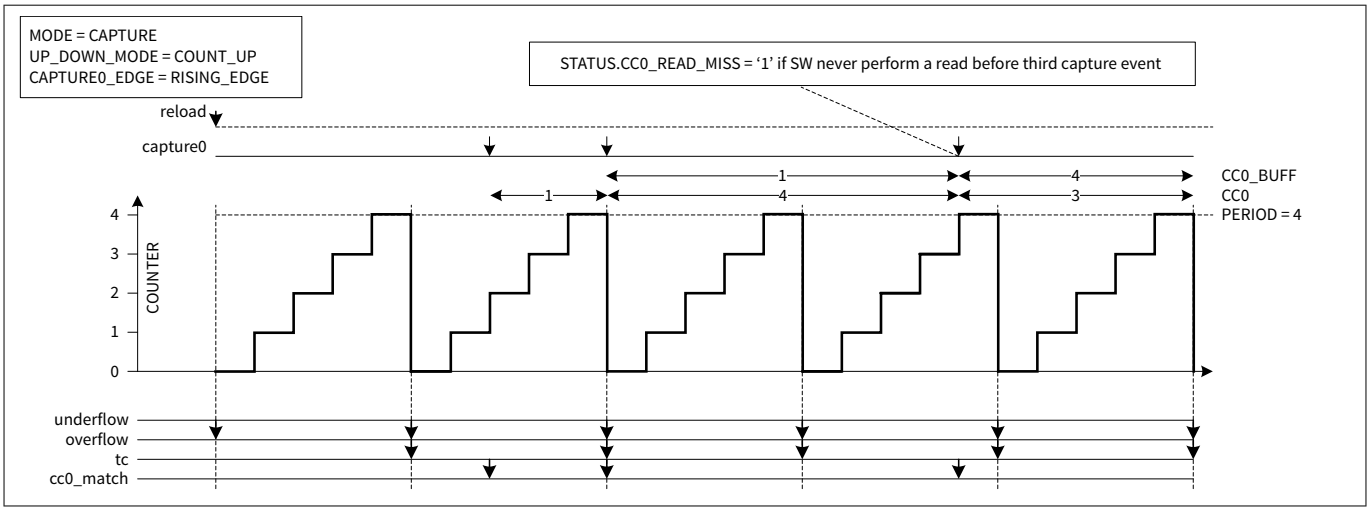


Figure 114 CC0/CC0_BUFF updated by capture0 event

When multiple capture events are detected before the next "active count" pre-scaled counter clock, capture events are treated as follows:

- In the rising edge or falling edge detection mode, multiple events are effectively reduced to a single event
- In the rising/falling both edge detection mode, an even number of events is not detected and an odd number of events are reduced to a single event

The above behavior is illustrated by the following figures, in which a pre-scaler by a factor of 4 is used. The first figure uses rising edge detection. The second figure uses rising/falling edge detection.

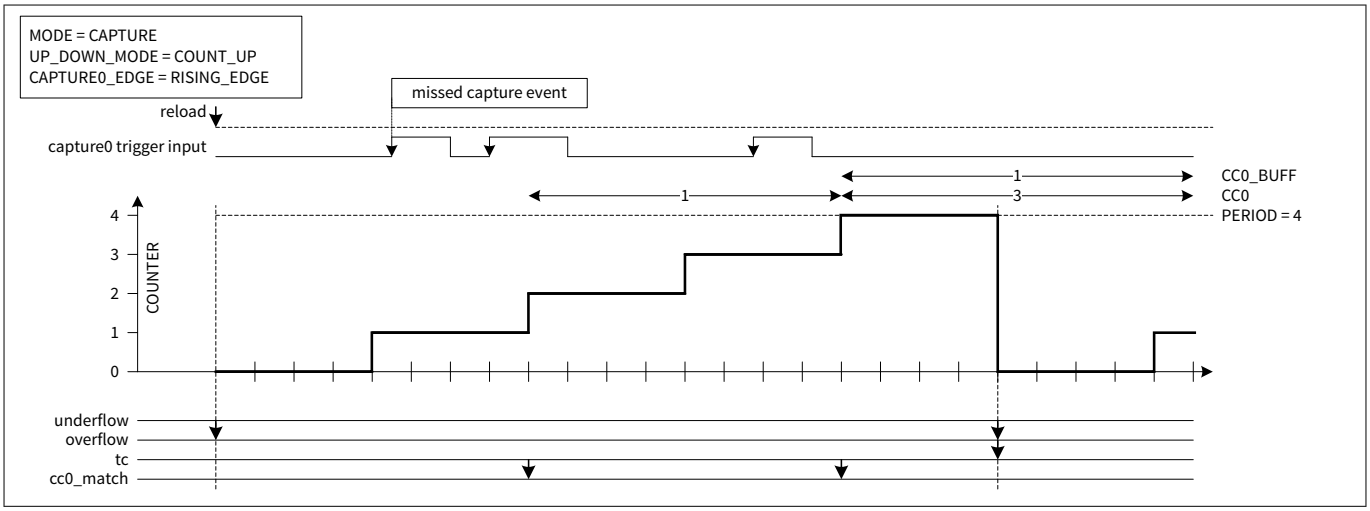


Figure 115 Multiple rising edge capture

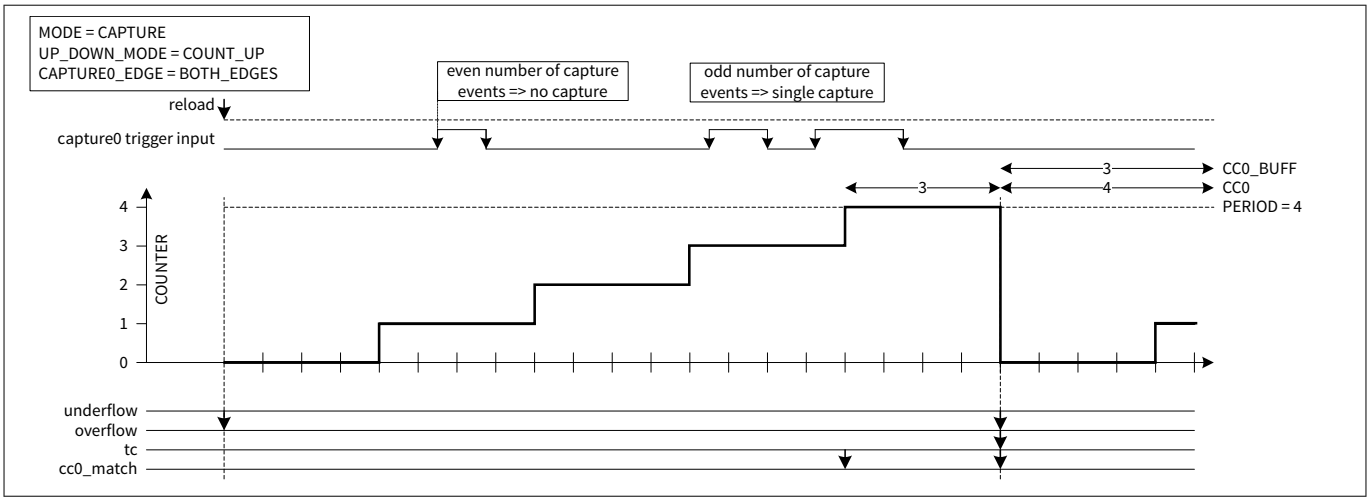


Figure 116 Multiple rising/falling both edge capture

25.3.2.2 Up/Down Count Modes

It is same as timer function mode.

25.3.2.3 One Shot Mode

It is same as timer function mode.

25.3.2.4 Clock Pre-scaling

It is same as timer function mode.

25.3.2.5 How to configure counter for capture mode

The steps to configure the counter for Capture mode operation and the affected register bits are as follows:

- Disable the counter by writing '0' to the ENABLE bit of the CTRL register
- Select Capture mode by writing '010' to the MODE[26:24] field of the CTRL register
- Set the required 16-bit or 32-bit period in the PERIOD register
- Set clock pre-scaling by writing to the PS_DIV [2:0] field of the PS register
- Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the CTRL register
- The counter can be configured to run either in continuous mode or one-shot mode by writing 0 or 1, respectively, to the ONE_SHOT[18] field of the CTRL register
- Set the TR_IN_SEL0 or TR_IN_SEL1 register to select the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
- Set the TR_IN_EDGE_SEL register to select the edge that causes the event (Reload, Start, Stop, Capture0/1, and Count)
- If required, set the interrupt upon TC or CC0_MATCH or CC1_MATCH condition
- Enable the counter by writing '1' to the ENABLED bit in the CTRL register. A reload trigger must be provided through firmware(RELOAD bit in PERI0_TR_CMD register) to start the counter if the hardware reload signal is not enabled

25.3.3 Quadrature mode

Quadrature functionality increments/decrements a counter between 0 and the maximum value 0xffff/0xffffffff or PERIOD , depending on QUAD_RANGE_MODE. Counter updates are under control of quadrature signal inputs: index, phiA and PhiB. The index input is used to indicate an absolute position. The phiA and phiB inputs are used to determine a change in position (the rate of change in position can be used to derive speed).

25.3.3.1 Quadrature mode functionality overview

Table 208 Range modes of QUAD

Supported range modes (QUAD_RANGE_MODE)\	Description
QUAD_RANGE0	Counter range is between 0x0000 and 0xffff/0xffffffff for timer resolution 16 and 32 bits respectively. It is 100% backward compatible with previous TCPWM quadrature behavior.
QUAD_RANGE0_CMP	Counter range is also between 0x0000 and 0xffff/ 0xffffffff. It is to a certain extend backward compatible with previous TCPWM quadrature behavior but allows a compare function during quadrature decoding using the CC0/CC0_BUFF registers and the cc0_match event (with GRP_CC1_PRESENT also a second compare function is available).
QUAD_RANGE1_CMP	Counter range is between 0x0000 and PERIOD, allows a compare function during quadrature decoding using the CC0/CC0_BUFF registers and the cc0_match event (with GRP_CC1_PRESENT also a second compare function is available).
QUAD_RANGE1_CAPT	Counter range is also between 0x0000 and PERIOD. Quadrature functionality in QUAD_RANGE1_CAPT mode provides the same functionality as the QUAD_RANGE1_CMP mode with the only difference that 1 or 2 capture functions (depending on CC1_PRESENT) are available instead of 1 or 2 compare functions.

Table 209 Encoding modes of QUAD

Supported encoding modes (QUAD_ENCODING_MODE)	Description
Quadrature encoding	4 encoding schemes for the phiA and phiB inputs are supported (as specified by CTRL.QUAD_ENCODING_MODE): • X1 encoding • X2 encoding • X4 encoding • Up/down rotary count mode

Note: • *Clock pre-scaling is NOT supported and the count event is used as a quadrature input phiA. As a result, the quadrature functionality operates on the counter clock ("clk_counter"), rather than on an "active count" pre-scaled counter clock*

Table 210 Input events of QUAD

Generated events	Usage
reload	This event acts as a quadrature index input. It initializes the counter and starts the quadrature functionality. Rising edge or falling edge event detection mode should be used. • QUAD_RANGE0: initialize counter to 0x8000/0x80000000, midpoint for resolution 16 and 32 bits respectively. • QUAD_RANGE0_CMP: initialize counter to 0x8000/0x80000000 • QUAD_RANGE1_CMP: initialize counter to 0 • QUAD_RANGE1_CAPT: initialize counter to 0
Start	This event acts as a quadrature phiB input. Pass through (no edge detection) event detection mode should be used for X1/X2/X4.
Stop	Stops the quadrature functionality. Once quadrature is stopped, reload must be used to start the quadrature.
Count	This event acts as a quadrature phiA input. Pass through (no edge detection) event detection mode should be used for X1/X2/X4.
Capture0	QUAD_RANGE0: Not used. QUAD_RANGE0_CMP: Second index. QUAD_RANGE1_CMP: Not used QUAD_RANGE1_CAPT: Capture event to copy COUNTER to CC0, CC0 to CC0_BUFF
Capture1	Only available when parameter GRP_CC1_PRESENT = 1 • QUAD_RANGE0: Not used • QUAD_RANGE0_CMP: Not used • QUAD_RANGE1_CMP: Not used • QUAD_RANGE1_CAPT: 2nd capture event to copy COUNTER to CC1, CC1 to CC1_BUFF

Table 211 Internal events of QUAD

Internal events	Description
cc0_match	QUAD_RANGE0: • Counter value COUNTER equals 0 or 0xffff/0xffffffff • Index event QUAD_RANGE0_CMP: Counter changes to a state in which COUNTER equals CC0. QUAD_RANGE1_CMP: Same as QUAD_RANG0_CMP. QUAD_RANGE1_CAPT: Capture0 event.
cc1_match	QUAD_RANGE0: Not used QUAD_RANGE0_CMP: Counter changes to a state in which COUNTER equals CC1 QUAD_RANGE1_CMP: Same as QUAD_RANG0_CMP QUAD_RANGE1_CAPT: Capture1 event
Underflow	QUAD_RANGE0: Not used. QUAD_RANGE0_CMP: Not used. QUAD_RANGE1_CMP: Counter value COUNTER equals 0 and is decrementing. QUAD_RANGE1_CAPT: Counter value COUNTER equals 0 and is decrementing.
Overflow	QUAD_RANGE0: Not used. QUAD_RANGE0_CMP: Not used. QUAD_RANGE1_CMP: Counter value COUNTER equals PERIOD and is incrementing. QUAD_RANGE1_CAPT: Counter value COUNTER equals PERIOD and is incrementing.
TC	QUAD_RANGE0: Index event. QUAD_RANGE0_CMP: • Counter value COUNTER equals 0 or 0xffff/0xffffffff when capture on wrap-around function is selected • Index or capture0 event when capture on index function is selected QUAD_RANGE1_CMP: • Counter value COUNTER equals 0 and is decrementing (underflow) or PERIOD and is incrementing (overflow) • Index event Counter value COUNTER equals PERIOD and is incrementing. QUAD_RANGE1_CAPT: Same as QUAD_RANGE1_CMP

Table 212 Line outputs of QUAD

PWM outputs	Description
line_out/tr_line	Not used
line_compl_out/tr_line_compl	Not used

QUAD_ENCODING_MODE specifies the quadrature encoding scheme. It can be X1, X2, X4 or up/down rotary mode.

Counter increments (incr1 event) and decrements (decr1 event) are determined by the quadrature encoding scheme as illustrated by the following Figure.

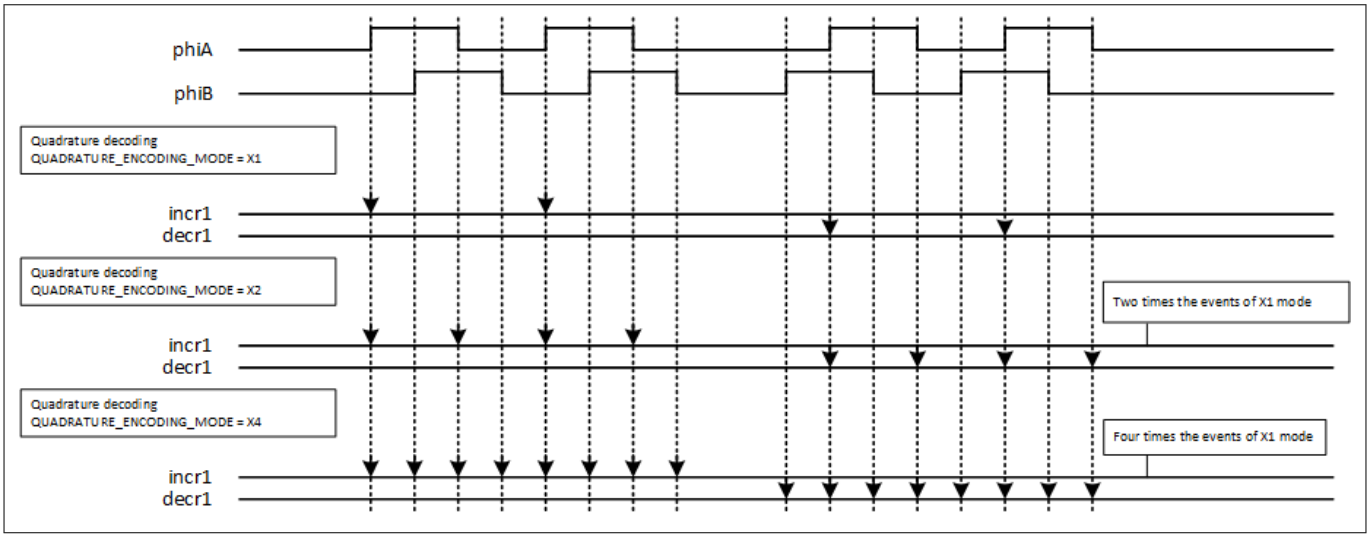


Figure 117 QUAD X1, X2 and X4 mode

***Note**: The x1 encoding scheme is identical with an up / down counting functionality of the following way: Rising edges of input phiA increment or decrement the counter depending on the state of input phiB (direction input).*

With UP_DOWN encoding (up/down rotary count mode) the counter is incremented by phiA and decremented by phiB as illustrated by the following Figure. In UP_DOWN encoding mode, phiA and phiB could be pass through or edge detection mode.

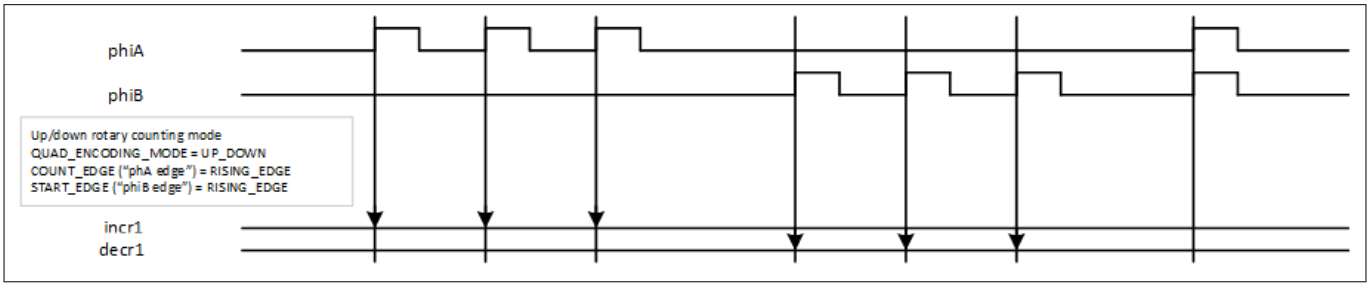


Figure 118 QUAD up down rotary mode

The state of phiA/phiB determines the increment/decrement per settings of encoding mode, and the phiA/phiB is detected by counter clock "clk_counter". The increment/decrement occurs when edge of phiA or phiB is detected by "clk_counter". To get correct quadrature encoding as above figure, only one edge can happen within one "clk_counter" cycle. Following figure illustrates phase inputs edge detection in X4 encoding mode.

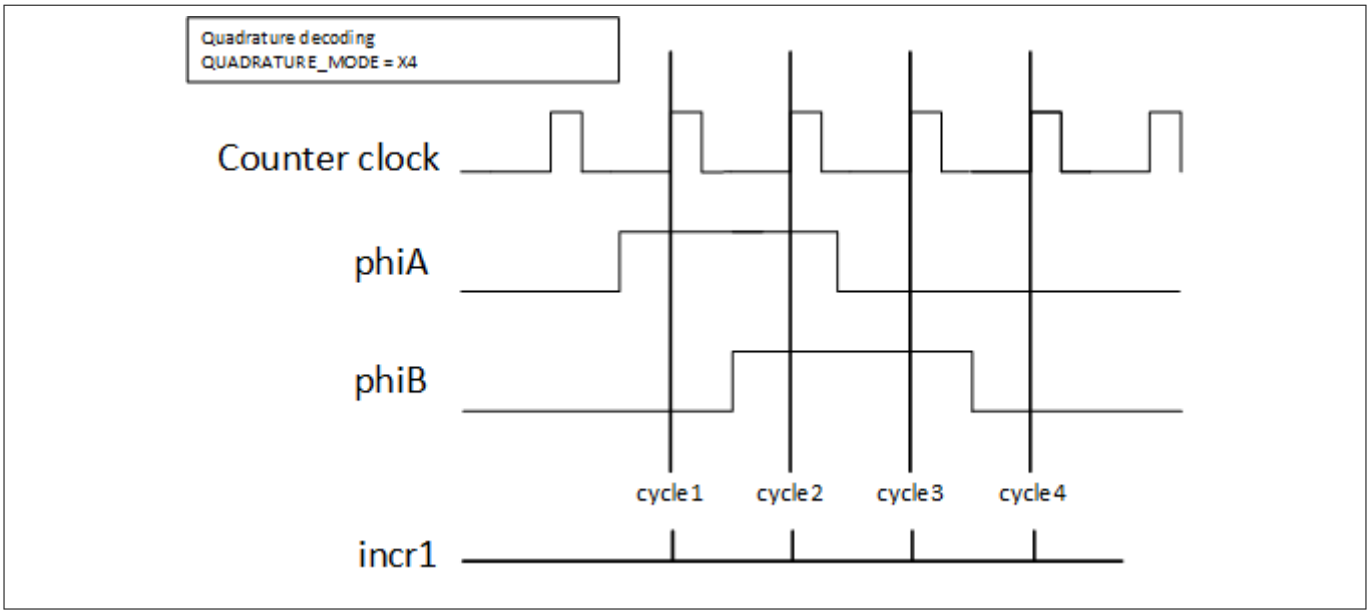


Figure 119 QUAD phiA-phiB edge detection on counter clock

25.3.3.2 QUAD_RANGE0

In this mode the counter range is between 0x0000 and 0xffff/0xffffffff (for 16/32 bits counter). It is 100% backward compatible with previous TCPWM quadrature behavior.

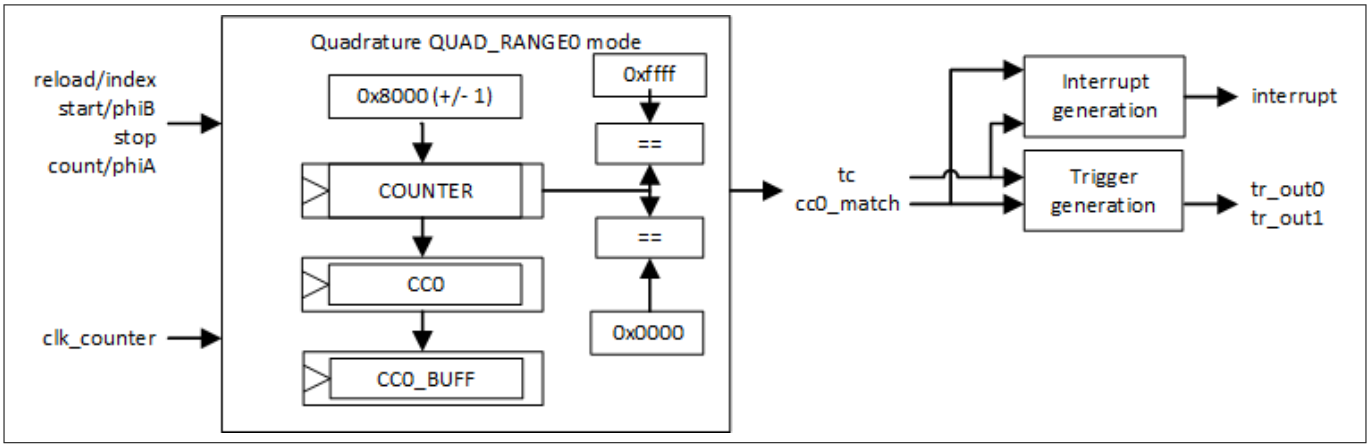


Figure 120 Quadrature (QUAD_RANG0) function diagram

Quadrature functionality in QUAD_RANGE0 mode is described as follows (for 16/32 bits counter):

- A SW generated reload event starts quadrature operation. As a result, COUNTER is set to 0x8000, which is the counter midpoint (the COUNTER is set to 0x7fff if the reload event coincides with a decrement event; the COUNTER is set to 0x8001 if the reload event coincides with an increment event). Note that a SW generated reload event is typically generated only once, when the counter is NOT running. All other reload/index events are HW generated reload events as a result of the quadrature index signal
- During quadrature operation:
  - The counter value COUNTER is incremented or decremented based on the specified quadrature encoding scheme
  - On a reload/index event, CC0 is copied to CC0_BUFF, COUNTER is copied to CC0, and COUNTER is set to 0x8000. In addition, the tc and cc0_match events are generated

- When the counter value COUNTER is 0x0000, CC0 is copied to CC0_BUFF, COUNTER (0x0000) is copied to CC0, and COUNTER is set to 0x8000. In addition, the cc0_match event is generated
- When the counter value COUNTER is 0xffff, CC0 is copied to CC0_BUFF, COUNTER (0xffff) is copied to CC0, and COUNTER is set to 0x8000. In addition, the cc0_match event is generated

The SW interrupt handler uses the tc and cc0_match interrupt cause fields to distinguish between a reload/index event and a situation in which a minimum/maximum counter value was reached (about to wrap around). The CC0 and CC0_BUFF registers are used to determine when the interrupt causing event occurred.

Note that a counter increment/decrement can coincide with a reload/index/tc event or with a situation cc0_match event. Under these circumstances, the counter value is set to either 0x8000+1 (increment) or 0x8000-1 (decrement).

The following Figure illustrates quadrature functionality as a function of the reload/index, incr1 and decr1 events.

The first reload/index event copies the counter value COUNTER to CC0.

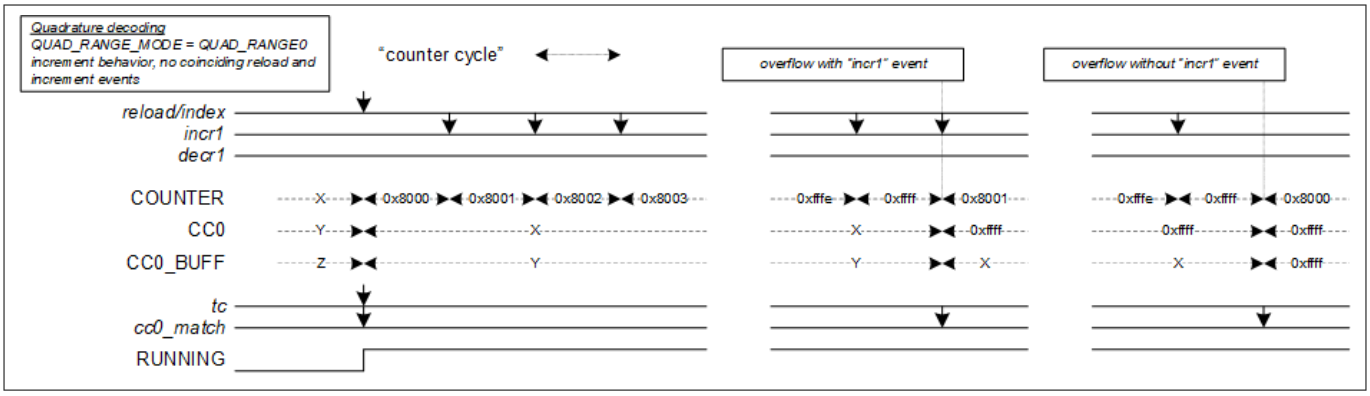


Figure 121 Overflow coincides with increment

The following figures illustrate quadrature functionality for different event scenarios (including scenarios with coinciding events). In all scenarios, the first reload/index event is generated by SW when the counter is not yet running.

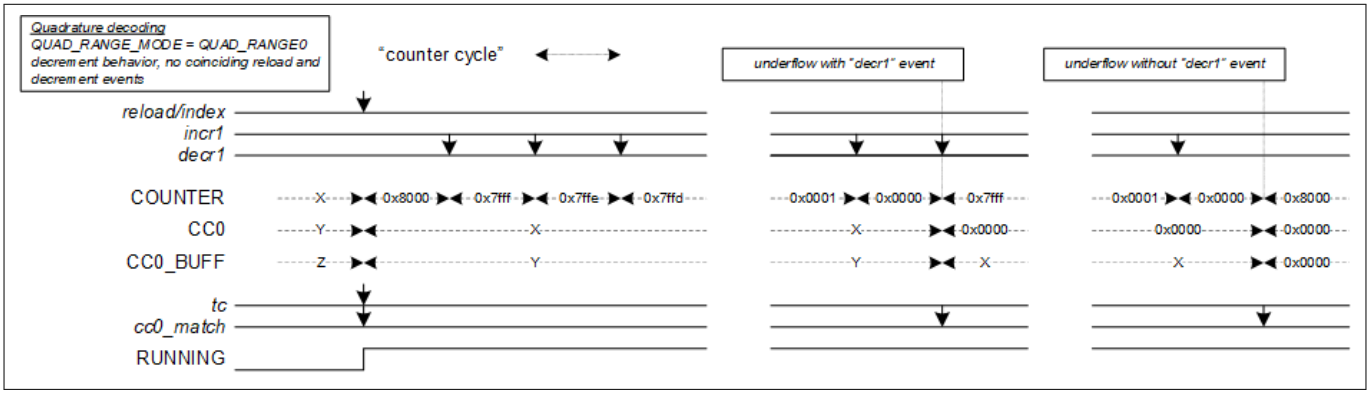


Figure 122 Underflow coincides with decrement

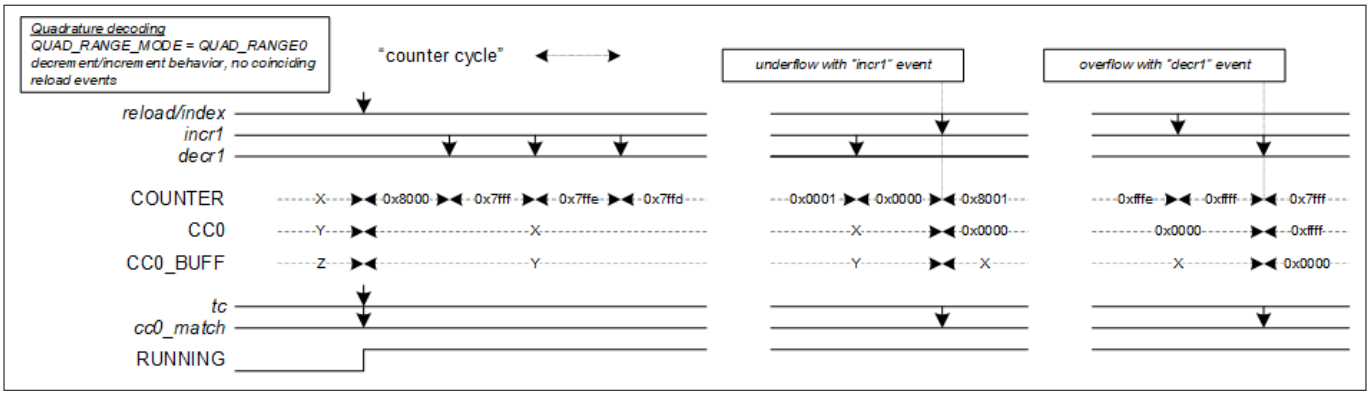


Figure 123 Underflow coincides with increment and overflow coincides with decrement

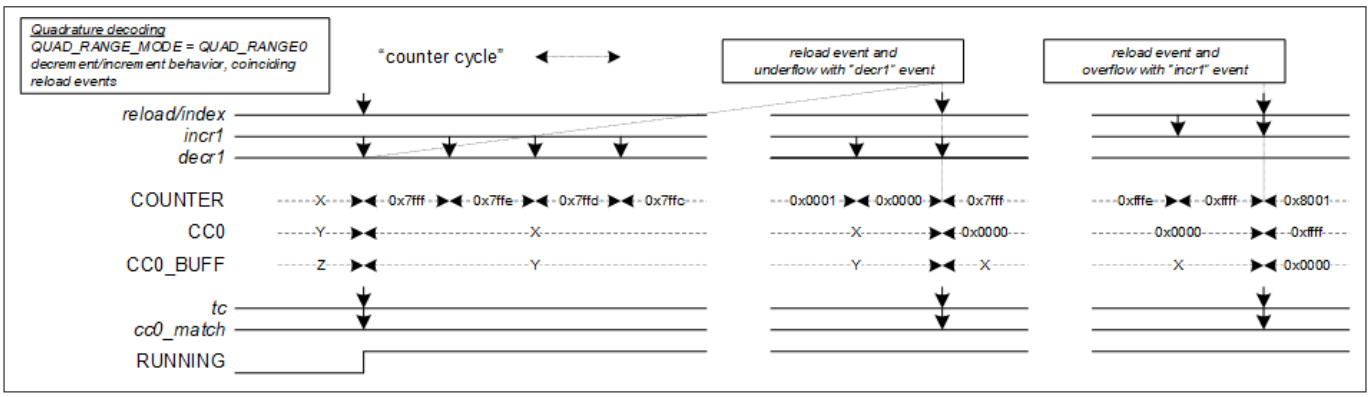


Figure 124 index, decrement (increment) and underflow (overflow) coincide

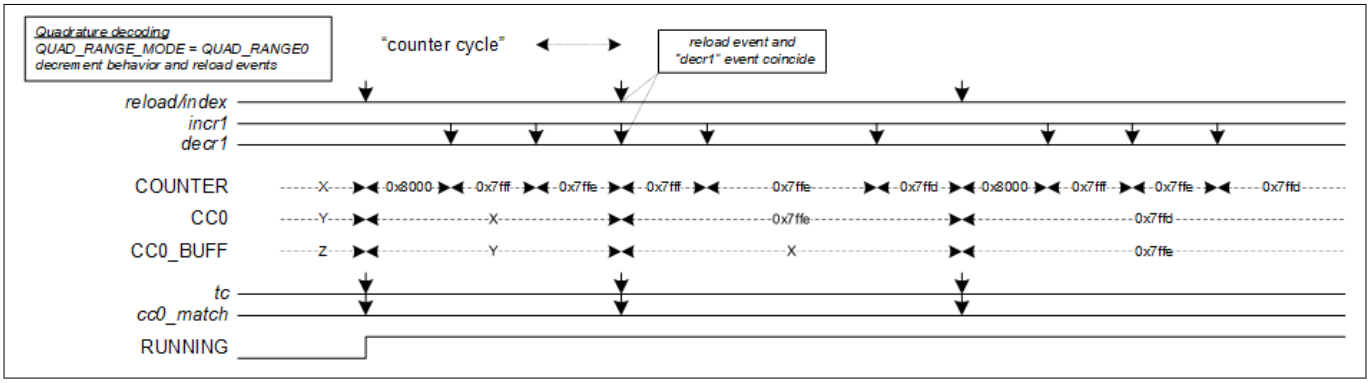


Figure 125 index coincides with and without decrement

The QUAD_RANGE0 functionality has the advantage that the interrupts when reaching minimum/maximum values are far apart in time, so such interrupts are unlikely to get lost.

This mode is preferred when interrupts are used for example to implement a higher range counter in SW.

However, care need to be taken for such SW counter since HW counter and SW counter are not updated in an atomic operation. This is not recommended for applications with real time requirements (e.g. motor control).

A disadvantage of this mode is that a physical angle position of the quadrature encoder can have multiple counter representations, so SW need to do modulo and subtract operations to calculate the absolute angle position.

```
x = COUNTER; // read COUNTER register
if (x>=0x8000)
pos = (x-0x8000) mod NR_COUNTS; // NR_COUNTS = encoders number of counts for one revolution
else
pos = NR_COUNTS – ((0x8000-x) mod NR_COUNTS);
```

25.3.3.3 QUAD_RANGE0_CMP

In this mode the counter range is also between 0x0000 and 0xffff/0xffffffff (for 16/32 bits counter). It is to a certain extend backward compatible with previous TCPWM quadrature behavior but allows a compare function during quadrature decoding using the CC0/CC0_BUFF registers and the cc0_match event (with GRP_CC1_PRESENT also a second compare function is available).

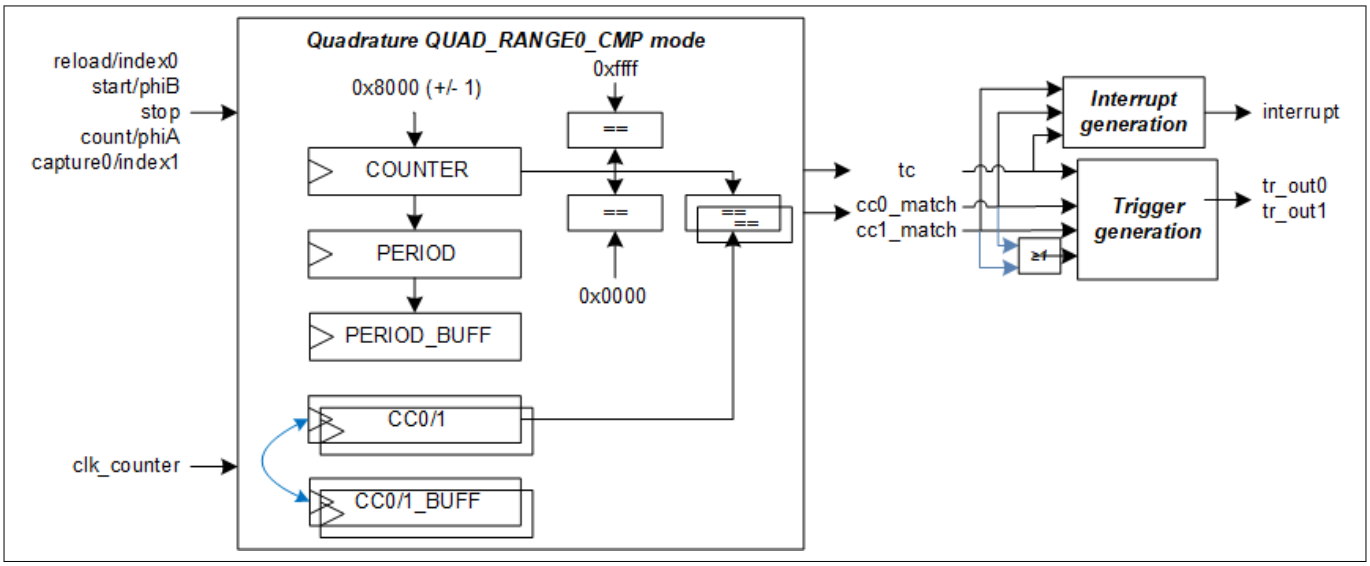


Figure 126 Quadrature (QUAD_RANG0_CMP) function diagram

Quadrature functionality in QUAD_RANGE0_CMP mode provides the same functionality as the QUAD_RANGE0 mode, except for the following differences (for 16 bits counter):

- PERIOD and PERIOD_BUFF are used instead of CC0 and CC0_BUFF to capture the counter value when a reload/index event occurs or the minimum/maximum value is reached (about to wrap around)
  - When the ‘capture on index’ function is selected (via overloaded AUTO_RELOAD_PERIOD bit) and a reload/index event occurs, PERIOD is copied to PERIOD_BUFF, COUNTER is copied to PERIOD, and COUNTER is set to 0x8000. In addition, the tc event is generated
  - When the ‘capture on wrap-around’ function is selected (via overloaded AUTO_RELOAD_PERIOD bit) and the counter value COUNTER is 0x0000 or 0xffff, PERIOD is copied to PERIOD_BUFF, COUNTER (0x0000 or 0xffff) is copied to PERIOD, and COUNTER is set to 0x8000. In addition, the tc event is generated
- Capture0 can be used as 2nd index event. This event acts as a second quadrature index input. It has the same function as the reload/index0 event. Both events are OR combined
- CC0 and CC0_BUFF are used for compare functionality
  - A cc0_match event is generated when the counter changes to a state in which COUNTER equals CC0

- CC0 and CC0_BUFF are exchanged on a cc0_match event (when CTRL.AUTO_RELOAD_CC0 = “1” and CTRL.SWAP_ENABLE = “1”)
- CC0_BUFF value is transferred to CC0 on a cc0_match event (when CTRL.AUTO_RELOAD_CC0 = “1” and CTRL.SWAP_ENABLE = “0”)

• When parameter GRP_CC1_PRESENT = 1, a second compare function is available
- A cc1_match event is generated when the counter changes to a state in which COUNTER equals CC1
- CC1 and CC1_BUFF are exchanged on a cc1_match event (when CTRL.AUTO_RELOAD_CC1 = “1” and CTRL.SWAP_ENABLE = “1”)
- CC1_BUFF value is transferred to CC1 on a cc1_match event (when CTRL.AUTO_RELOAD_CC1 = “1” and CTRL.SWAP_ENABLE = “0”)

Note that ‘capture on index’ and ‘capture on wrap-around’ functions are separated to prevent PERIOD and PERIOD_BUFF from being overwritten before SW has read them in case a wrap-around is followed by multiple index events in a short time (quadrature encoder is moved back and forth around its index point). If both functions are needed, two counters can be used synchronously in QUAD_RANGE0_CMP mode, one with ‘capture on index’, and the other with ‘capture on wrap-around’ behavior selected.

Note also that multiple compare values can be realized (even when parameter GRP_CC1_PRESENT = 0) by multiple synchronous counters in QUAD_RANGE0_CMP mode with different CC0 values.

Except the differences mentioned above the Quadrature QUAD_RANGE0_CMP mode behaves as the Quadrature QUAD_RANGE0 mode including behavior with coinciding events. The following Figure illustrates an example scenario with decrementing counter and additional compare functionality.

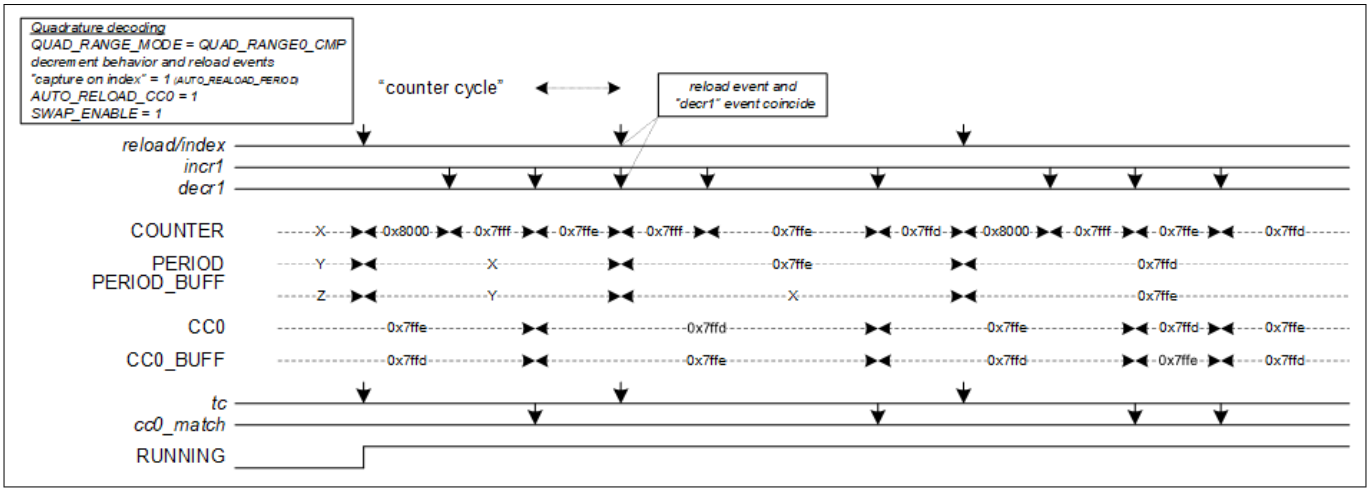


Figure 127 Quadrature (QUAD_RANG0_CMP) operation with AUTO_RELOAD_CC0=“1” and SWAP_ENABLE=“1”

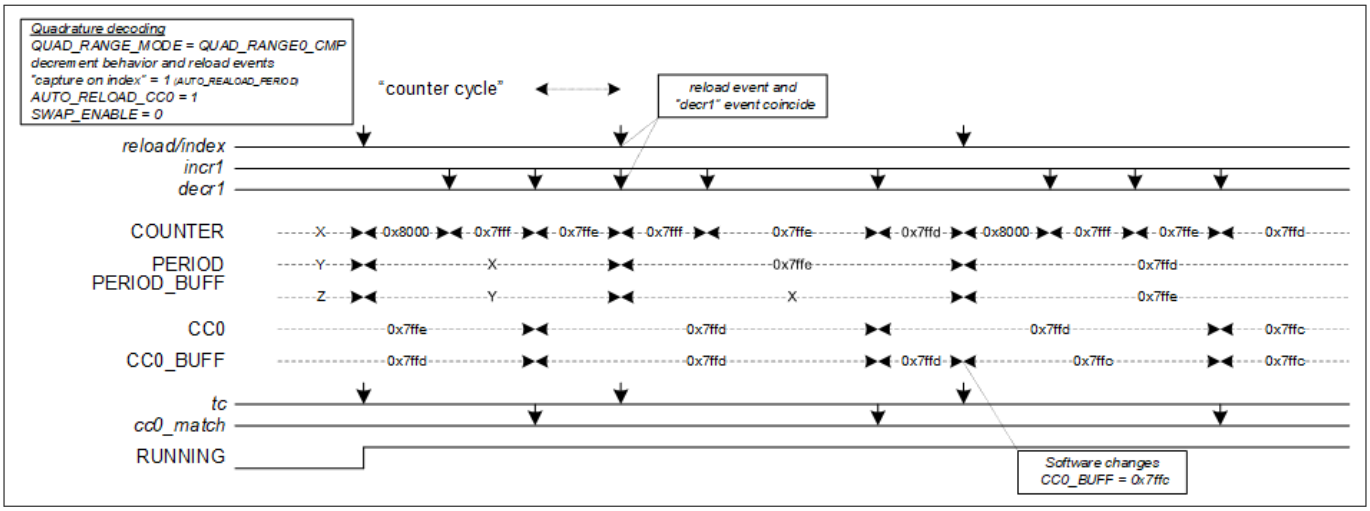


Figure 128 Quadrature (QUAD_RANG0_CMP) operation with AUTO_RELOAD_CC0="1" and SWAP_ENABLE="0"

Following figure illustrates the case that CC0 = 0x8000. The next value of counter and current value of CC0 determines cc0_match generation. The intermediate state of counter due to events coinciding has no effect. The CC0 swapping is only determined by cc0_match asserting and AUTO_RELOAD_CC0.

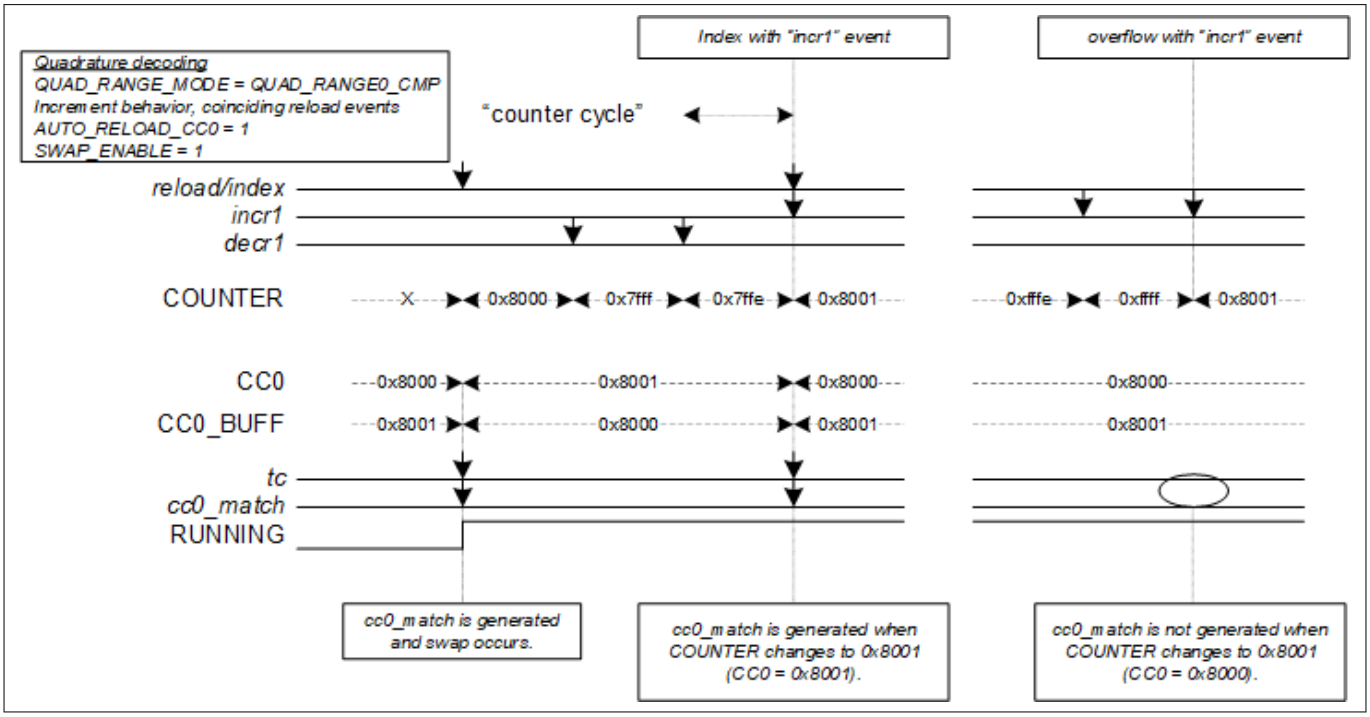


Figure 129 cc0_match generation and CC0/CC0_BUFF swap when CC0=0x8000

The QUAD_RANGE0_CMP functionality allows a similar interrupt usage as in QUAD_RANGE0 mode. Additionally it supports 1 or 2 compare functions. More compare functions can be reached with multiple synchronously running counters in QUAD_RANGE0_CMP mode. These compare functions can be for example used for a position compare.

As in QUAD_RANGE0 mode there is the disadvantage that a physical angle position of the quadrature encoder can have multiple counter representations, so SW need to do modulo and subtract operations to calculate the absolute angle position.

In QUAD_RANGE0_CMP mode this SW operation can be simplified using 2 compare functions, e.g. by setting CC0 = 0x8000 + NR_COUNTS and CC1 = 0x8000 – NR_COUNTS, and feeding back the cc0/1_match events back

into the TCPWM counter as index0/1 events using peripheral trigger multiplexers. This sets the counter back to its midpoint when reaching CC0 or CC1 value (after up to 3 clk_peri cycles for synchronization). This way the modulo operation can be saved in SW when calculating the absolute angle position:

```
x = COUNTER; // read COUNTER register
if (x>=0x8000)
pos = x-0x8000;
else
pos = x – CC1;
```

This is much less SW overhead than in QUAD_RANGE0 mode, however SW is still involved. A DMA copy of the absolute angle position e.g. to a send buffer of CAN / UART / field bus interface to synchronize with other devices is NOT possible.

This can only be supported when the counter represents the absolute angle position, as done in the following QUAD_RANGE1_CAPT / QUAD_RANGE1_CMP modes.

25.3.3.4 QUAD_RANGE1_CMP

In this mode the counter range is between 0x0000 and PERIOD. It is NOT backward compatible with previous TCPWM quadrature behavior.

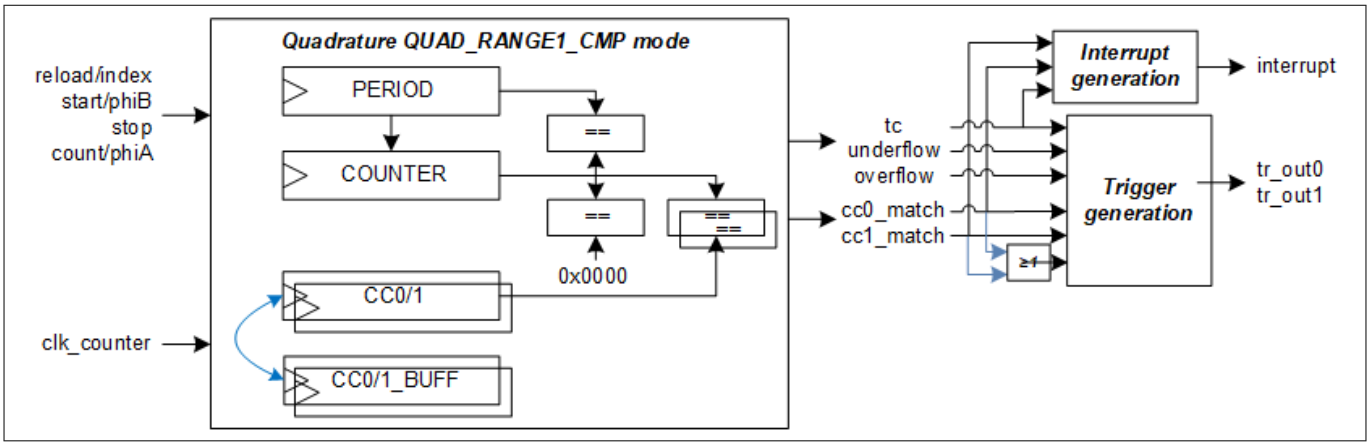


Figure 130 Quadrature (QUAD_RANG1_CMP) function diagram

Quadrature functionality in QUAD_RANGE1_CMP mode is described as follows:

- A SW generated reload event starts quadrature operation. As a result, COUNTER is set to 0x0000 (the COUNTER is set to PERIOD if the reload event coincides with a decrement event; the COUNTER is set to 0x0001 if the reload event coincides with an increment event). Note that a SW generated reload event is generated only once, when the counter is NOT running. All other reload/index events are HW generated reload events as a result of the quadrature index signal
- During quadrature operation:
  - The counter value COUNTER is incremented or decremented based on the specified quadrature encoding scheme
  - - On a reload/index event, COUNTER is set to 0x0000. In addition, the tc event is generated
    - When the counter value COUNTER is 0x0000 and decrementing, COUNTER is set to PERIOD. In addition, the tc event and underflow event are generated

- When the counter value COUNTER equals PERIOD and is incrementing, COUNTER is set to 0x0000. In addition, the tc event and overflow event are generated
  - CC0 and CC0_BUFF are used for compare functionality
- A cc0_match event is generated when the counter changes to a state in which COUNTER equals CC0
- CC0 and CC0_BUFF are exchanged on a cc0_match event (when specified by CTRL.AUTO_RELOAD_CC0 and CTRL.SWAP_ENABLE = “1”). CC0_BUFF are transferred to CC0 on a cc0_match event (when specified by CTRL.AUTO_RELOAD_CC0 and CTRL.SWAP_ENABLE = “1”)
  - When parameter GRP_CC1_PRESENT = 1 a second compare function is available
- A cc1_match event is generated when the counter changes to a state in which COUNTER equals CC1
- CC1 and CC1_BUFF are exchanged on a cc1_match event (when specified by CTRL.AUTO_RELOAD_CC1 and CTRL.SWAP_ENABLE = “1”)

Note that a counter increment/decrement can coincide with a reload/index event. In this case, the counter value is set to either 0x0000+1 (increment) or PERIOD (decrement).

The following Figures illustrates quadrature functionality as a function of the reload/index, incr1 and decr1 events when CTRL.SWAP_ENABLE = “1”.

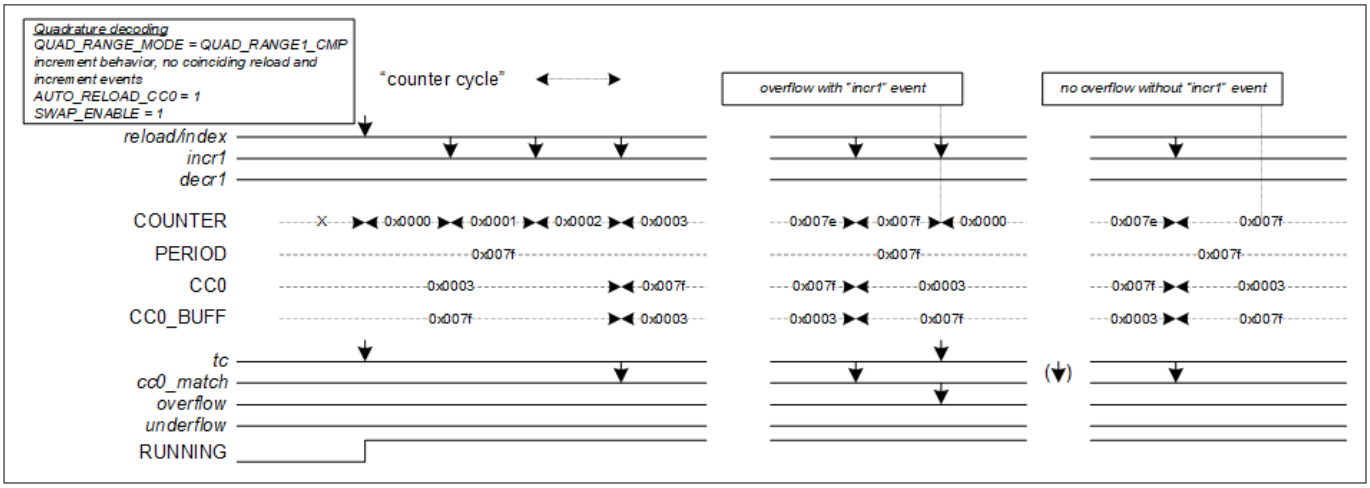


Figure 131 Quadrature index, incr1 and tc (overflow) generation

The following Figures illustrate quadrature functionality for different event scenarios (including scenarios with coinciding events). In all scenarios, the first reload/index event is generated by SW when the counter is not yet running.

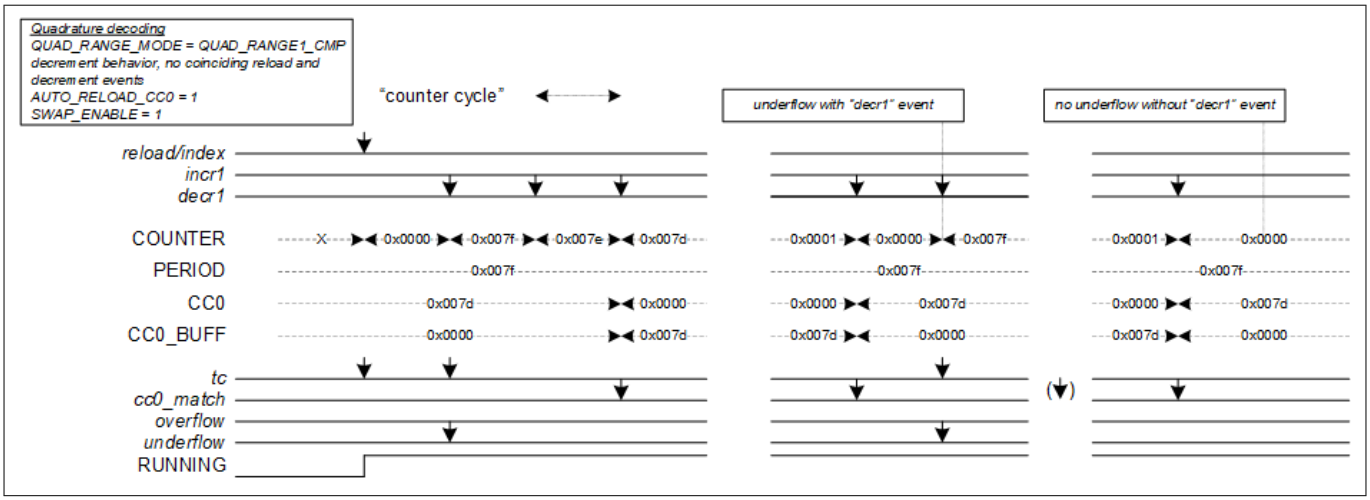


Figure 132 Quadrature index, decr1 and tc (underflow) generation

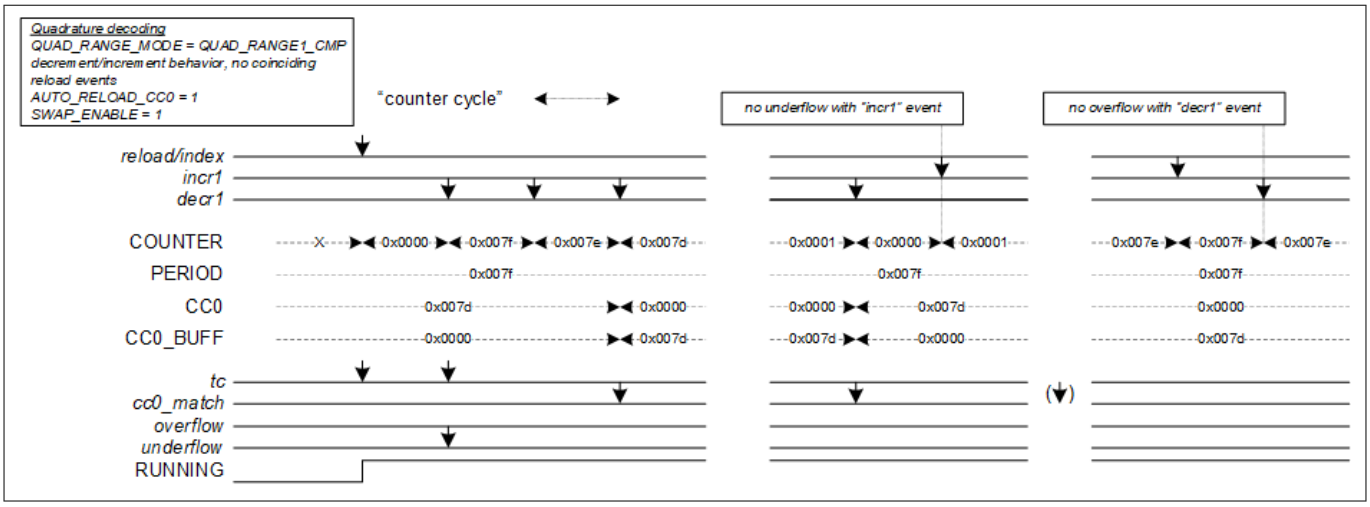


Figure 133 No TC(underflow) after COUNTER = "0x0000" and no TC(overflow) after COUNTER = "PERIOD"

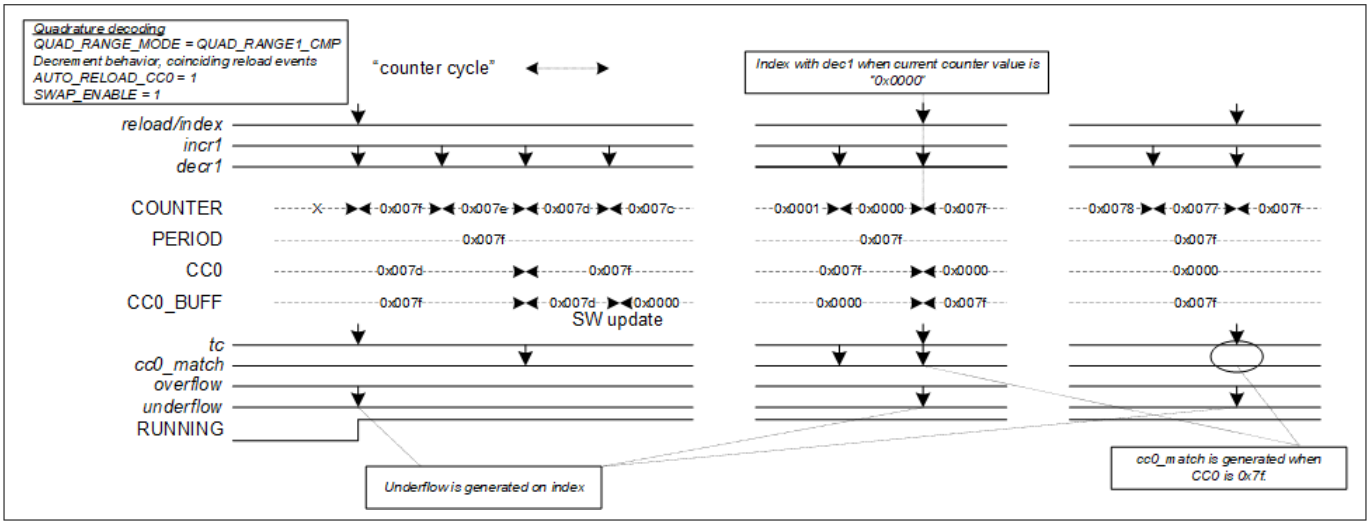


Figure 134 Index, decrement, underflow and cc0_match coincide

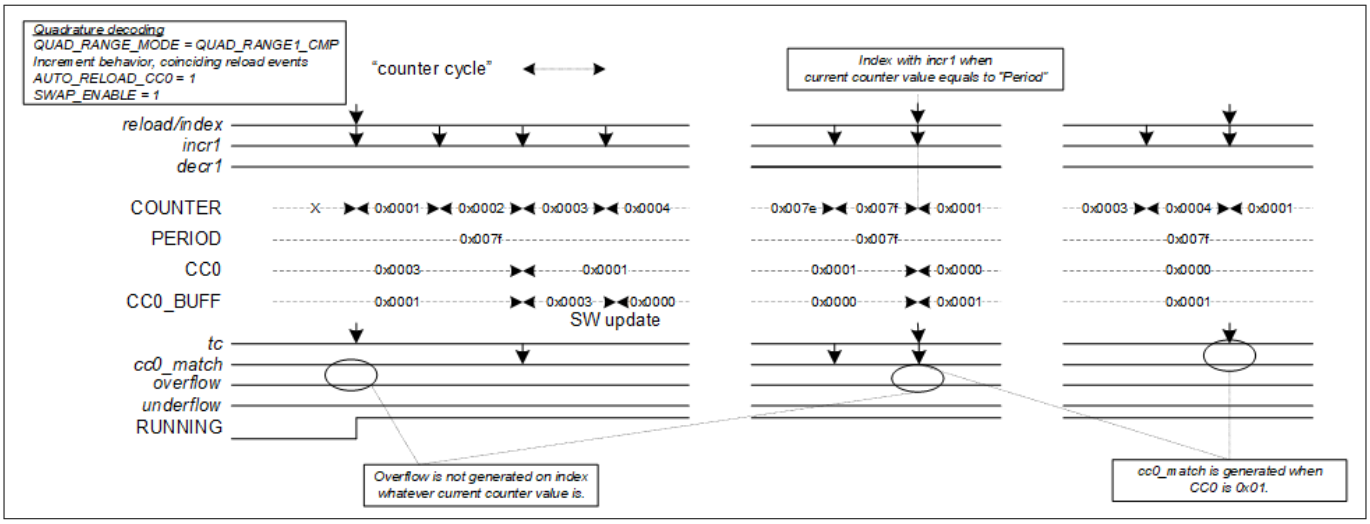


Figure 135 Index, increment, overflow and cc0_match coincide

The QUAD_RANGE1_CMP functionality allows that the COUNTER register reflects the current angle position of the rotary encoder, i.e. no MOD or SUB calculations need to be done in SW on the COUNTER value to get the current angle position. This allows a DMA copy of the current angle position from the COUNTER register, e.g. to a send buffer of CAN / UART / field bus interface to synchronize with other devices.

However, a disadvantage of this mode is that fast sequences of tc interrupts can occur (when encoder moves back and forth around start position). It is recommended to not use the tc interrupt in this mode.

25.3.3.5 QUAD_RANGE1_CAPT

In this mode the counter range is also between 0x0000 and PERIOD. Quadrature functionality in QUAD_RANGE1_CAPT mode provides the same functionality as the QUAD_RANGE1_CMP mode with the only difference that 1 or 2 capture functions are available instead of 1 or 2 compare functions.

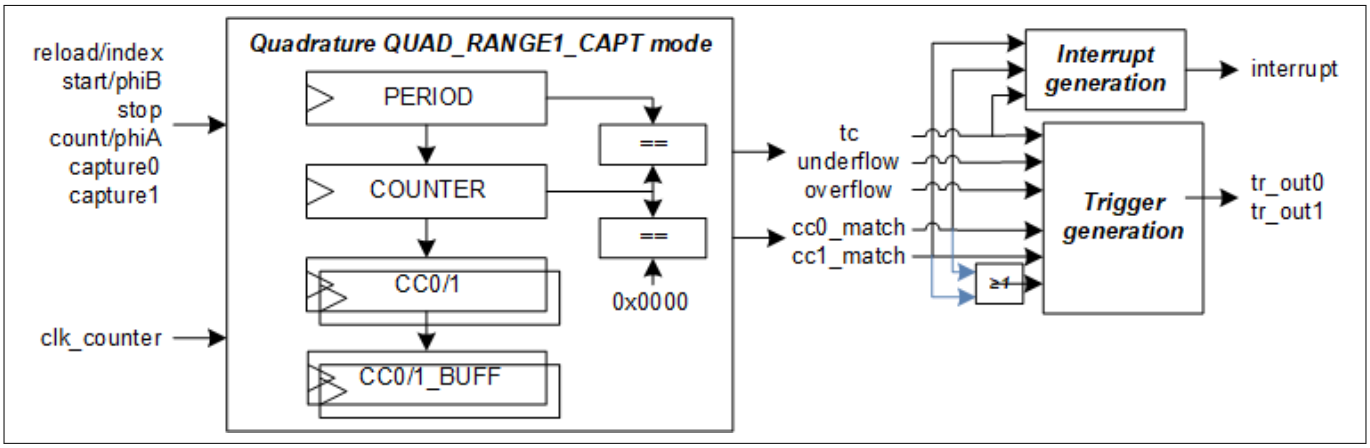


Figure 136 Quadrature (QUAD_RANG1_CAPT) function diagram

The following Figure illustrates an example scenario with decrementing counter and capture functionality.

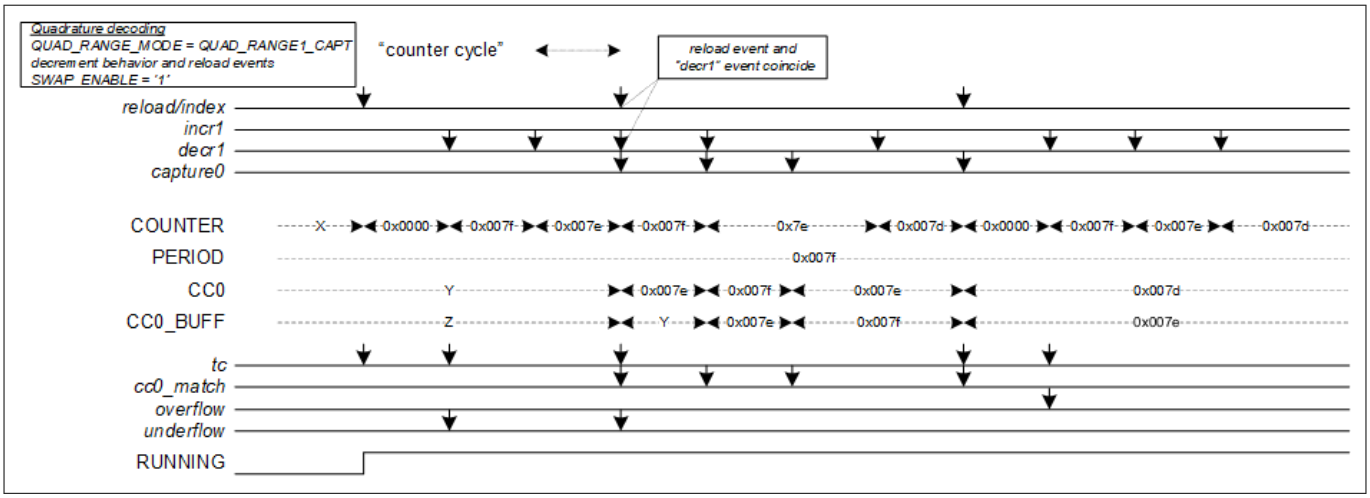


Figure 137 Quadrature (QUAD_RANG1_CAPT) capture operation

25.3.4 Pulse-width modulation (PWM) mode

The PWM functionality increments/decrements a counter between 0 and PERIOD. When the counter is running, the counter value COUNTER is compared with CC0 (CC1). When COUNTER equals CC0 (CC1), the cc0_match (cc1_match) event is generated. Additionally, on a counter overflow and counter underflow, the overflow and underflow events are generated. Combined, the cc0_match, cc1_match, overflow, and underflow events are used to generate a pulse-width modulated signal on the PWM LINE_OUT and LINE_COMPL_OUT output signals. Left-aligned, right-aligned, and center-aligned PWM signals can be generated. Asymmetric PWM signals can be generated using the COUNT_UPDN2 mode. The current PWM output level can be read. A special case of 0 or 100 percent duty cycle is supported. The PERIOD_BUFF register is used for duty cycle update and becomes active by a TC event.

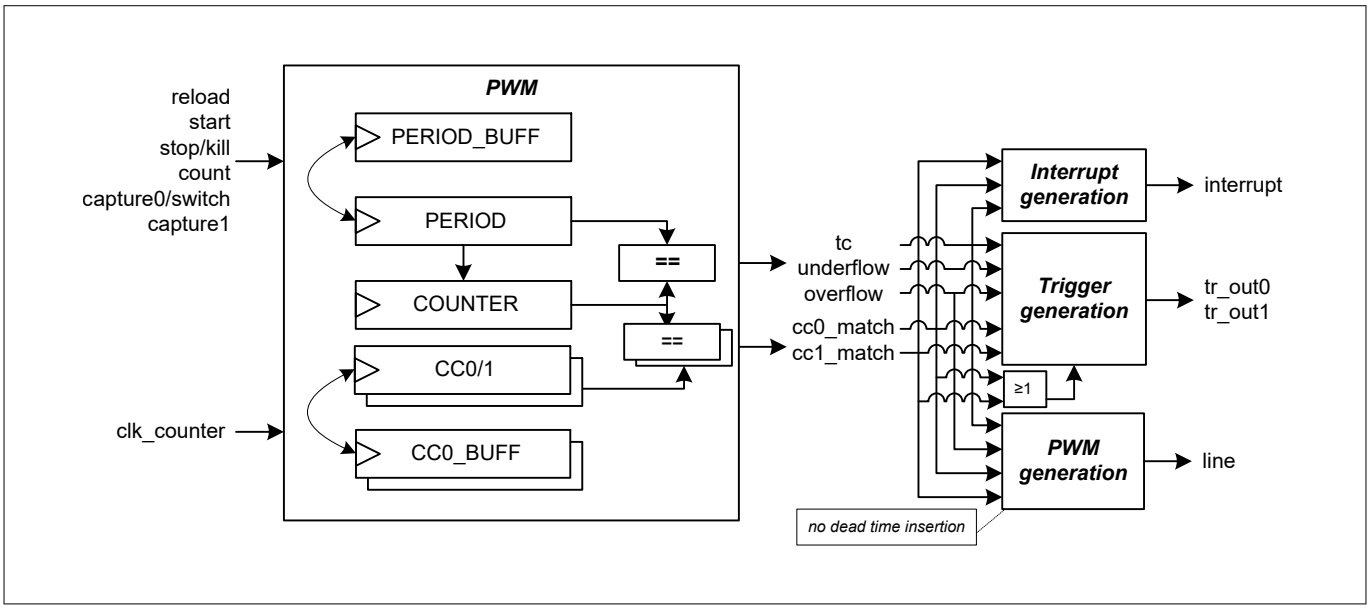


Figure 138 PWM mode functionality

Table 213 PWM mode trigger input description

Generated events	Usage
reload	Initializes and starts the counter. **Should only be used when the counter is not running.** Initialization is dependent on UP_DOWN_MODE: • COUNT_UP: The counter is initialized to “0” • COUNT_DOWN: counter is initialized to PERIOD COUNT_UPDN1/2: The counter is initialized to “1”
start	Starts the counter (thecounter is NOT initialized by HW). Can be used when the counter is running or not running. When the counter is running, the start event has no effect.
Stop	Stops the counter. Different stop/kill modes exist.
Count	Count event increments/decrements the counter per settings of up/down modes.
Capture0	This event acts as a switch event. When this event is active, the CC0/CC0_BUFF, CC1/CC1_BUFF, and PERIOD/PERIOD_BUFF and LINE_SEL/LINE_SEL_BUFF registers are exchanged on a tc event (when specified by CTRL.AUTO_RELOAD_CC0/1, CTRL.AUTO_RELOAD_PERIOD and CTRL.AUTO_RELOAD_LINE_SEL) when CTRL.SWAP_ENABLE = “1”. This event acts as a switch event. When this event is active, CC0_BUFF is transferred to CC0, CC1_BUFF is transferred to CC1 and PERIOD_BUFF and LINE_SEL_BUFF registers are transferred to PERIOD and LINE_SEL on a tc event (when specified by CTRL.AUTO_RELOAD_CC0/1, CTRL.AUTO_RELOAD_PERIOD and CTRL.AUTO_RELOAD_LINE_SEL) when CTRL.SWAP_ENABLE = “0”. When a switch event is detected, it is kept pending till the next tc event. **A switch event requires rising, falling, or rising/falling edge mode. Pass through mode is not supported, unless the selected event is a constant ‘0’ or ‘1’ The switch event will be cleared and has no effect if it is detected when the counter is not running.**

(table continues...)

Table 213 (continued) PWM mode trigger input description

Generated events	Usage
Capture1 (stop1/kill1)	This event acts as a second stop/kill event. It has the same function as the stop0/kill0 event. Both events are OR combined. **Note:** Having 2 stop/kill events for a PWM allows selecting for one stop/kill event a common trigger from a PERI trigger multiplexer (allowing synchronous stop/kill operation of multiple PWMs) while selecting for the other stop/kill event, for example, a dedicated ADC out-of-range trigger (for example, allowing real time hardware stop of a PWM when current of PWM driven signal is out of range).

Table 214 PWM mode supported features

Supported features	Description
Clock pre-scaling	Pre-scales the counter clock “clk_counter”.
One shot	Counter is stopped by HW, on a tc event. In COUNT_UPDN2, the counter is stopped on tc event when underflow.
Auto reload CC	CC0 and CC0_BUFF are exchanged on a switch event and tc event (when specified by CTRL.AUTO_RELOAD_CC0). CC1 and CC1_BUFF are exchanged on a switch event AND tc event (when specified by CTRL.AUTO_RELOAD_CC1).
Auto reload PERIOD	PERIOD and PERIOD_BUFF are exchanged on a switch event and tc event (when specified by CTRL.AUTO_RELOAD_PERIOD).) Note: when COUNT_UPDN2 mode exchanges PERIOD and PERIOD_BUFF at a tc event that coincides with an overflow event, software should ensure that the PERIOD and PERIOD_BUFF values are the same.
Auto reload LINE_SEL	LINE_SEL and LINE_SEL_BUFF are exchanged on a switch event AND tc event (when specified by CTRL.AUTO_RELOAD_LINE_SEL) when CTRL.SWAP_ENABLE = “1”. LINE_SEL_BUFF is transferred to LINE_SEL on a switch event AND tc event (when specified by CTRL.AUTO_RELOAD_LINE_SEL) when CTRL.SWAP_ENABLE = “0”.
Up/down modes	Specified by UP_DOWN_MODE: • COUNT_UP: counter counts from 0 to PERIOD • COUNT_DOWN: counter counts from PERIOD to 0 • COUNT_UPDN1/2: counter counts from 1 to PERIOD and back to 0

(table continues...)

Table 214 (continued) PWM mode supported features

Supported features	Description
Kill modes	Specified by PWM_IMM_KILL, PWM_SYNC_KILL, PWM_STOP_ON_KILL, PWM_TC_SYN_KILL_DT and PWM_SYN_KILL_DT. See the sections below for further details.

***Note**: The PWM mode does not support dead time insertion. This functionality is supported by the separate PWM_DT mode.*

Table 215 PWM mode trigger output description

Trigger output	Description
cc0_match	Specified by UP_DOWN_MODE: • COUNT_UP and COUNT_DOWN: counter changes to a state in which COUNTER equals CC0 • COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC0. CC0_MATCH_UP_EN/CC0_MATCH_DOWN_EN will enable or disable cc0_match generation when count is in up/down counting. (CC0=0 and CC0=PERIOD will be the special case for 0/100% duty cycle PWM generation
cc1_match	Specified by UP_DOWN_MODE: • COUNT_UP and COUNT_DOWN: counter changes to a state in which COUNTER equals CC1 • COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC1. CC1_MATCH_UP_EN/CC1_MATCH_DOWN_EN will enable or disable cc1_match generation when count is in up/down counting
Underflow	Counter is decremented and changes from a state in which COUNTER equals “0”. Reload event generates underflow in COUNT_DOWN, COUNT_UPDN1 or COUNT_UPDN2 mode.
Overflow	Counter is incrementing and changes from a state in which COUNTER equals PERIOD. Reload event generates overflow in COUNT_UP mode.

(table continues...)

Table 215 (continued) PWM mode trigger output description

Trigger output	Description
tc	Specified by UP_DOWN_MODE: • COUNT_UP: tc event is the same as the overflow event • COUNT_DOWN: tc event is the same as the underflow event • COUNT_UPDN1: tc event is the same as the underflow event • COUNT_UPDN2: tc event is the same as the logical OR of the overflow and underflow events Reload will generate underflow/overflow, but not generate tc.

Table 216 PWM mode internal events description

PWM output	Description
line_out	PWM line output
line_compl_out	Complementary PWM line output.

Figure 139 illustrates the process of “line_out” and “line_compl_out” generation. Generally, “line_out” output reflects the state of PWM signal “line” and “line_compl_out” output reflects the inverted version of “line”. The behavior of a line depends on different function modes. Furthermore, some other facts will impact “line_out” and “line_compl_out” per different function modes, they are “line selection”, “dead time insertion”, “kill function”, “line polarity” and “kill line polarity”.

The PWM output signals are generated by using the underflow, overflow, and cc_match events. Each of these events can be individually set to INVERT, SET, or CLEAR pwm_dt_input.

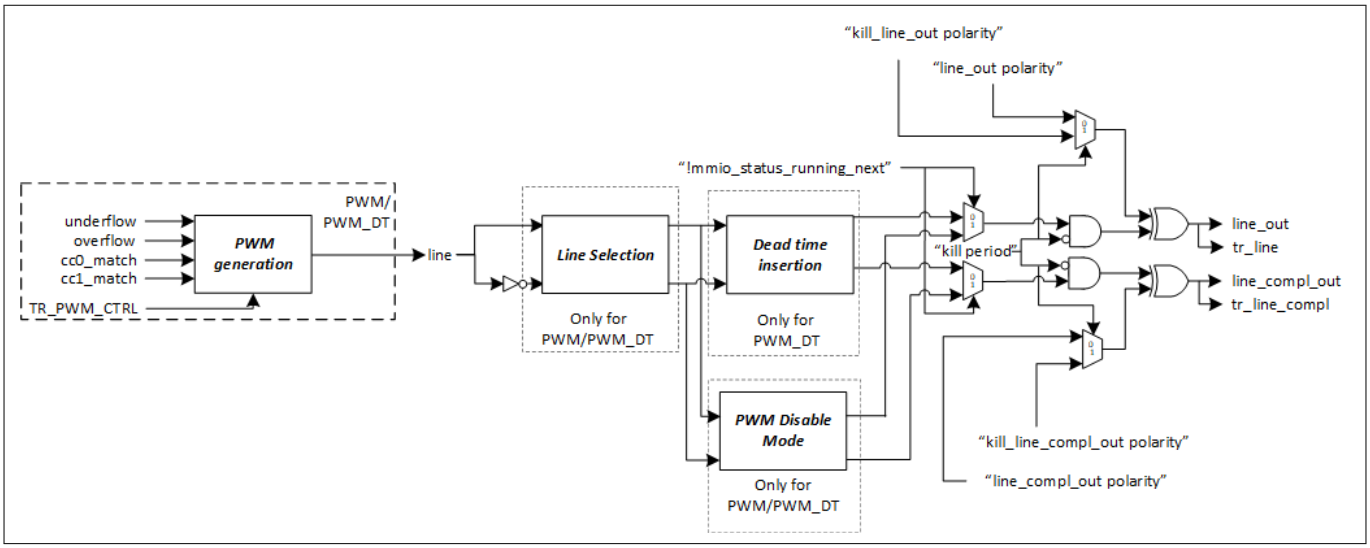


Figure 139 PWM output generation

***Note**: that the cc0_match event generating in COUNT_UP and COUNT_DOWN modes are different from the generation in other TIMER/CAPTURE modes. This is to ensure that 0% and 100% duty cycles can be generated.*

PWM behavior depends on the PERIOD and CC0 registers. Software can update the PERIOD_BUFF and CC0_BUFF registers, without affecting the PWM behavior. The switch/capture event can be used to switch the values of the compare and buffered compare registers. It also switches the values of the period and buffered period registers when CTRL.SWAP_ENABLE = “1”. When CTRL.SWAP_ENABLE = “0”, the switch/capture event can be used to transfer the values of the buffered compare register to compare register. It also transfers the values of the buffered period register to the period register.

Following table summarizes the kill mode supported in PWM mode. Any settings that are not defined in the following table are not supported or not recommended.

Table 217 Kill modes of PWM

Kill mode	Settings	Kill behavior
No-IMM-Async	PWM_IMM_KILL = 0; PWM_SYNC_KILL = 0; PWM_TC_SYNC_KILL_DT = 0; PWM_SYNC_KILL_DT = 0; PWM_STOP_ON_KILL = 0; STOP_EDGE = NO_EDGE_DET	• PWM output get suppressed at the next active count clock after kill input is active • PWM output suppress is removed at the next active count clock after kill input is inactive
IMM-Async	PWM_IMM_KILL = 1; PWM_SYNC_KILL = 0; PWM_TC_SYNC_KILL_DT = 0; PWM_SYNC_KILL_DT = 0; PWM_STOP_ON_KILL = 0; STOP_EDGE = NO_EDGE_DET	• PWM output is suppressed immediately after kill input is active • PWM output suppress is removed at the next active count clock after kill input is inactive
No-IMM-Sync	PWM_IMM_KILL = 0; PWM_SYNC_KILL = 1; PWM_TC_SYNC_KILL_DT = 0; PWM_SYNC_KILL_DT = 0; PWM_STOP_ON_KILL = 0; STOP_EDGE = RISING;	• PWM output get suppressed at the next active count clock after kill input is active • PWM output suppress is removed at the next tc event after kill input is inactive
IMM-Sync	PWM_IMM_KILL = 1; PWM_SYNC_KILL = 1; PWM_TC_SYNC_KILL_DT = 0; PWM_SYNC_KILL_DT = 0; PWM_STOP_ON_KILL = 0; STOP_EDGE = RISING;	• PWM output is suppressed immediately after kill input is active • PWM output suppress is removed at the next tc event after kill input is inactive

(table continues...)

Table 217 (continued) Kill modes of PWM

Kill mode	Settings	Kill behavior
TC-Sync-DT	PWM_IMM_KILL = 0; PWM_SYNC_KILL = 0; PWM_TC_SYNC_KILL_DT = 1 PWM_SYNC_KILL_DT = 0; PWM_STOP_ON_KILL = 0; STOP_EDGE = RISING;	• PWM output get suppressed at the next tc event after kill input is active. If deadtime is enabled, output gets suppressed after the tc event + deadtime • PWM output suppress is removed at the next tc event after kill input is inactive • This is only supported in PWM_DT mode
IMM-Sync-DT	PWM_IMM_KILL = 1; PWM_SYNC_KILL = 0; PWM_TC_SYNC_KILL_DT = 0; PWM_SYNC_KILL_DT = 1; PWM_STOP_ON_KILL = 0; STOP_EDGE = RISING;	• PWM output line_out get suppressed immediately whereas line_compl_out get suppressed after dead time when kill input is active • PWM output suppress is removed at the next tc event after kill input is inactive • This is only supported in PWM_DT mode
No-IMM-Stop	PWM_IMM_KILL = 0; PWM_SYNC_KILL = Don't care; PWM_TC_SYNC_KILL_DT= Don't care; PWM_SYNC_KILL_DT = Don't care; PWM_STOP_ON_KILL = 1; STOP_EDGE = RISING_EDGE/ FALLING_EDGE/BOTH_EDGES;	• PWM output get suppressed at the next active count clock after kill input is active • PWM output suppress is removed when counter restart after kill input is inactive
IMM-Stop	PWM_IMM_KILL = 1; PWM_SYNC_KILL = Don't care; PWM_TC_SYNC_KILL_DT = Don't care; PWM_SYNC_KILL_DT = Don't care PWM_STOP_ON_KILL = 1; STOP_EDGE = RISING_EDGE/ FALLING_EDGE/BOTH_EDGES;	• PWM output is suppressed immediately after kill input is active • PWM output suppress is removed when counter restart after kill input is inactive

25.3.4.1 Up/down Count Modes

Up/down count modes control the counting direction (increment or decrement) while the counter is running.

COUNT_UP mode

Figure 140 illustrates a PWM in COUNT_UP mode. The counter is initialized (to 0) and started with a software-based reload event.

Notes:

- When the counter changes from a state in which COUNTER is 4, an overflow and tc event are generated
- When the counter changes to a state in which COUNTER equals 2, a cc0_match event is generated
- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods

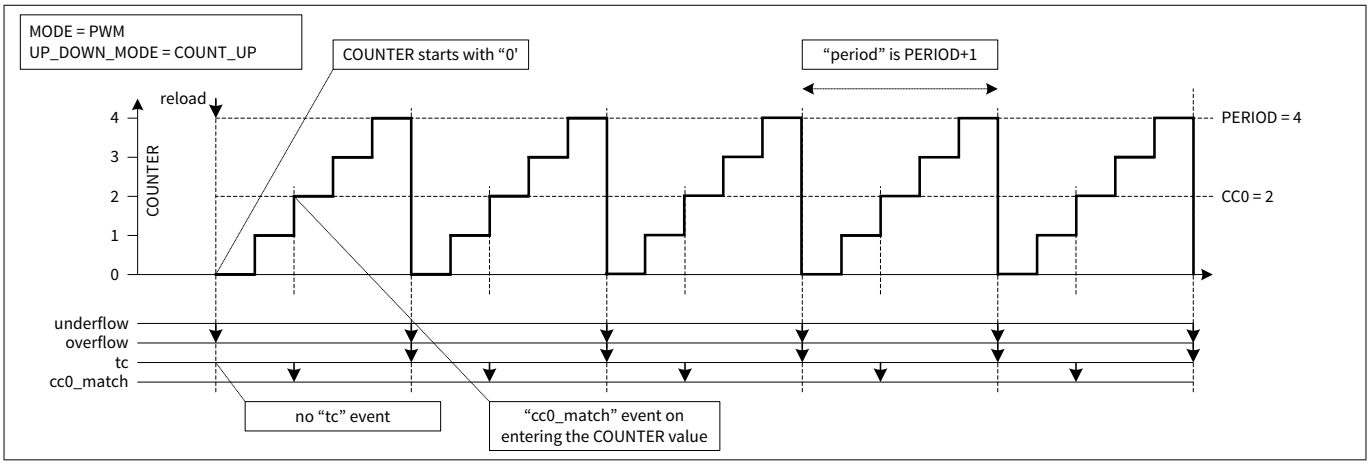


Figure 140 PWM COUNT_UP mode

COUNT_DOWN mode

Figure 141 illustrates a PWM in COUNT_DOWN mode. The counter is initialized (to PERIOD) and started with a software-based reload event.

Note:

- When the counter changes from a state in which COUNTER is 0, an underflow and tc event are generated
- When the counter changes to a state in which COUNTER equals 2, a cc0_match event is generated
- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods

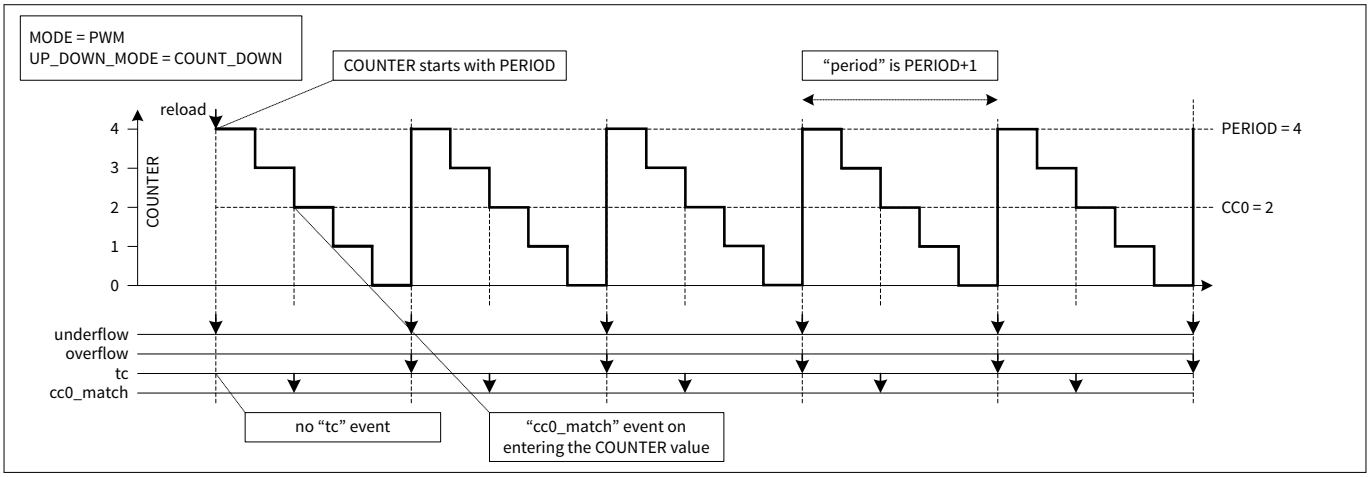


Figure 141 PWM in COUNT_DOWN mode

COUNT_UPDN1 mode

Figure 142 illustrates a PWM in COUNT_UPDN1 mode. The counter is initialized (to 1) and started with a software-based reload event.

Note:

- When the counter changes from a state in which COUNTER is 4, an overflow is generated
- When the counter changes from a state in which COUNTER is 0, an underflow and TC event are generated
- When the counter changes from a state in which COUNTER equals 2, a cc0_match event is generated

Note that the actual counter value COUNTER from before the reload event is not used, instead the
The counter value before the reload event is considered to be 0

- PERIOD is 4, resulting in an effective repeating counter pattern of 2*4 = 8 counter clock periods

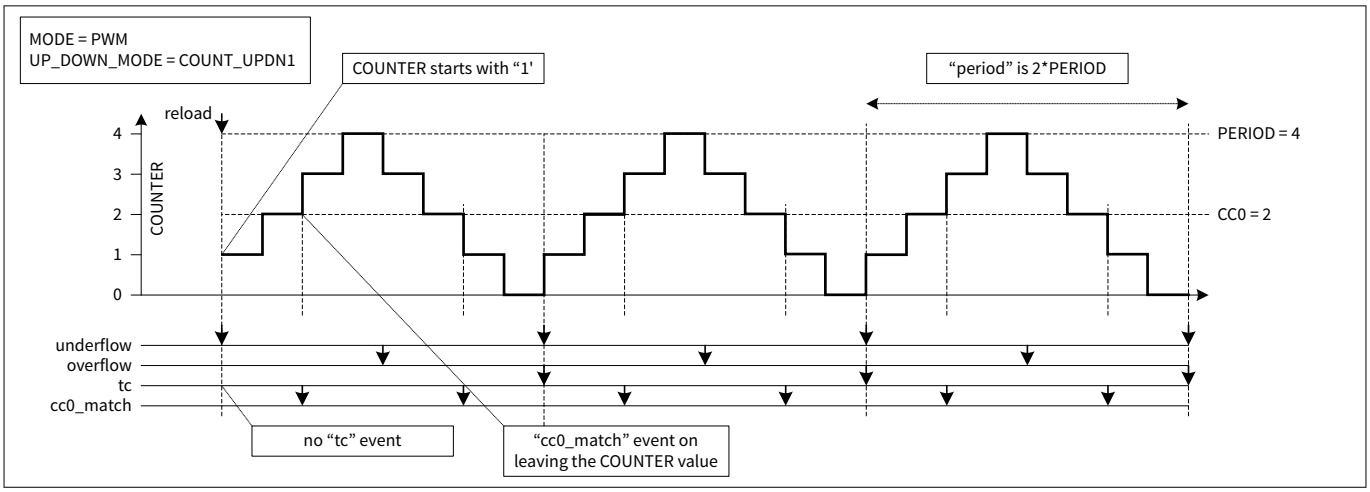


Figure 142 PWM in COUNT_UPDN1 mode

25.3.4.2 One Shot Mode

It is same as timer function mode.

25.3.4.3 Clock Pre-scaling

It is same as timer function mode.

25.3.4.4 CC and PERIOD auto reload with switch event

Auto CC reload and auto PERIOD reload will provide dynamic PWM duty cycle change and count period length change. The active switch event (capture0) is required for switch activity:

- At the TC event, if the switch event is active and the AUTO_RELOAD_CC bit of the CTRL register is set to '1', CC0/1 and CC0/1_BUFF will exchange value
- At the TC event, if the switch event is active and the AUTO_RELOAD_PERIOD bit of CTRL register is set to '1', PERIOD and PERIOD_BUFF will exchange value
- CC1 and CC1_BUFF will exchange value at the TC event if capture0 is active

The following figures illustrate the update of period value in COUNT_UP mode and COUNT_DOWN mode resulting in different period times after each switch event.

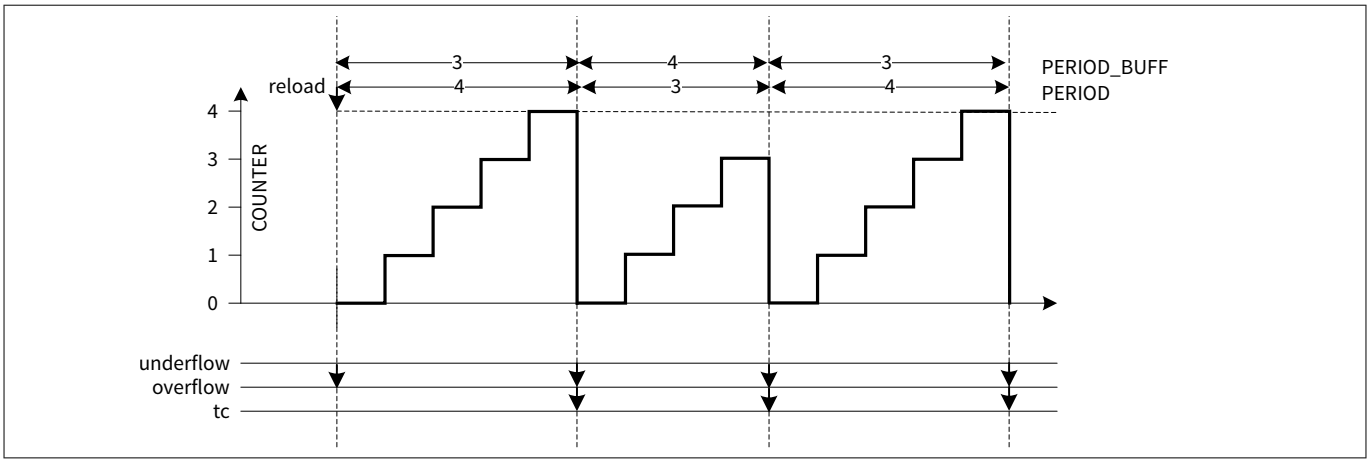


Figure 143 PERIOD/PERIOD_BUFF exchange in COUNT_UP mode by a switch event

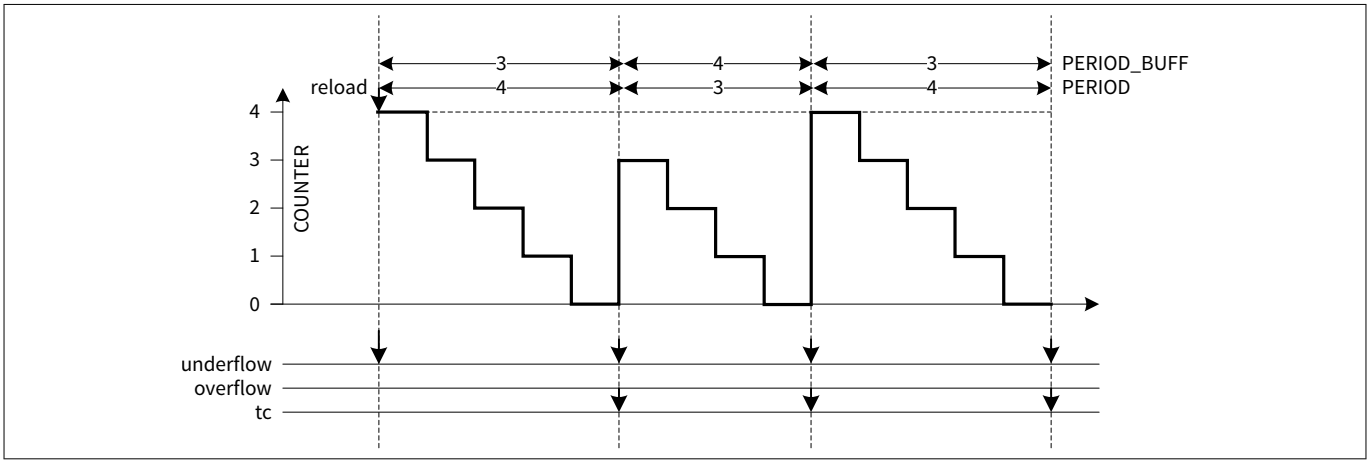


Figure 144 PERIOD/PERIOD_BUFF exchange in COUNT_DOWN mode by a switch event

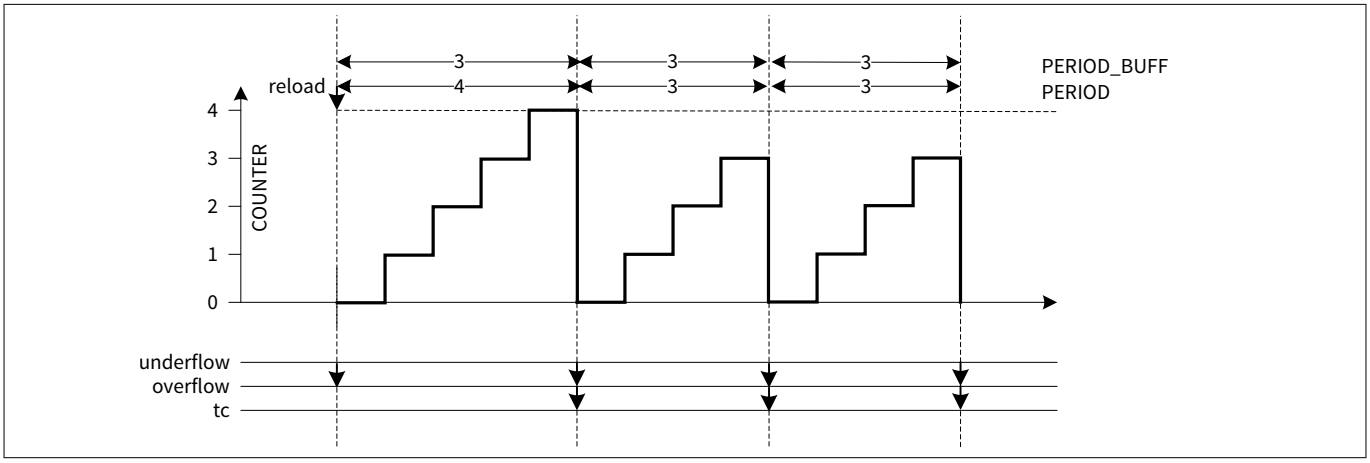


Figure 145 PERIOD/PERIOD_BUFF exchange in COUNT_UP mode (SWAP_ENABLE="0")

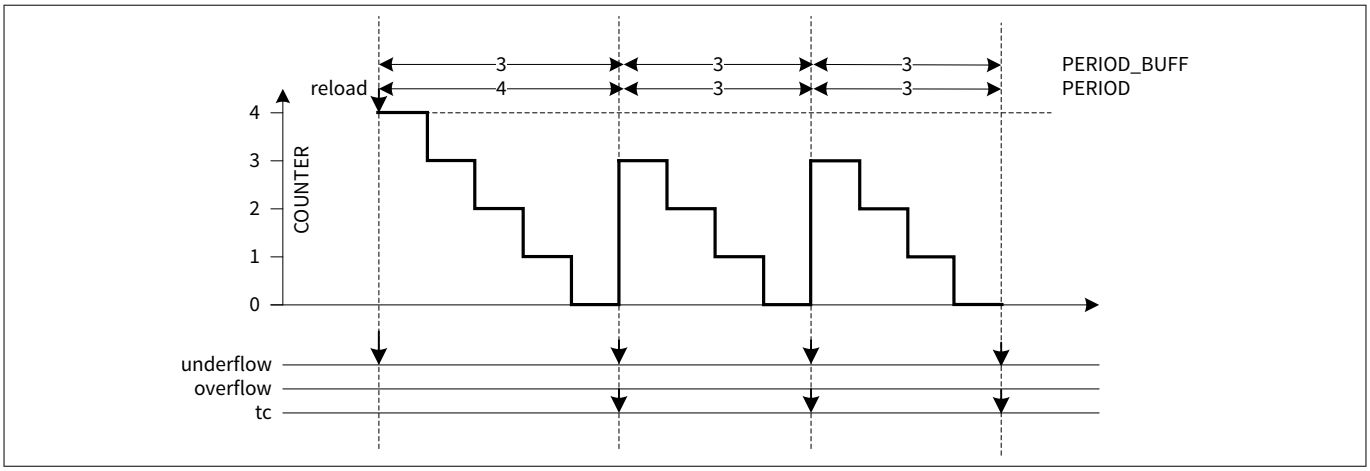


Figure 146 PERIOD/PERIOD_BUFF exchange in COUNT_DOWN mode (SWAP_ENABLE="0")

A potential problem arises when software updates are not completed before the next tc event with an active pending switch event. For example, if software updates PERIOD_BUFF before the tc event, and CC0_BUFF after the tc event, switching does not reflect the CC0_BUFF register update. To prevent this from happening, the switch event should be generated by software through an MMIO register write after both the PERIOD_BUFF and CC0_BUFF registers are updated. The switch event is kept pending by the hardware till the next tc event occurs.

The previous paragraph addressed synchronized updates of the CC0/CC0_BUFF and PERIOD/PERIOD_BUFF registers of a single PWM by means of a software-generated switch event. During motor control, three PWMs work in unison and updates to all period and compare register pairs should be synchronized. All three PWMs have synchronized periods and as a result have synchronized tc events. The switch event for all three PWMs is generated by the same hardware trigger input to synchronously update CC0 at the same TC, after the PERIOD_BUFF and CC0_BUFF registers of all three PWMs are updated by software.

25.3.4.5 Left/Right/Center align PWM with CC0/CC0_BUFF auto reload

PWM can generate left align, right align, and center align with the following features supported in PWM mode:

- Up/down count mode must be used to generate different phase aligned PWM
- Line state is changed per underflow/overflow/cc0_match/cc1_match internal event and can be configured in TR_PWM_CTRL register

The required settings for left align, right align, and center align PWM are:

- Left align:
  - Write the value '0' to the UP_DOWN_MODE [17:16] field in the CTRL register to set the counter direction to COUNT_UP mode
  - Write the value '0' (SET) to OVERFLOW_MODE [3:2] field of the TR_PWM_CTRL register to set the LINE_OUT signal to '1' when the COUNTER reaches PERIOD value
  - Write the value '1' (CLEAR) to the CC0_MATCH_MODE [1:0] field in the TR_PWM_CTRL register to set the LINE_OUT signal to '0' when the COUNTER equals CC0 value
- Right align:
  - Write the value '1' to the UP_DOWN_MODE [17:16] field in the CTRL register to set the counter direction to COUNT_DOWN mode

- Write the value '0' (SET) to CC0_MATCH_MODE [1:0] field in the TR_PWM_CTRL register to set the LINE_OUT signal to '1' when the COUNTER equals CC0 value
- Write the value '1' (CLEAR) to the UNDERFLOW_MODE [5:4] field in the TR_PWM_CTRL register to set the LINE_OUT signal to '0' when the COUNTER reaches '0'

- Center align:
  - Write the value '2' to the UP_DOWN_MODE [17:16] field in the CTRL register to set the counter direction to COUNT_UPDN1 mode
  - Write the value '0' (SET) to the OVERFLOW_MODE [3:2] field in the TR_PWM_CTRL register to set the LINE_OUT signal to '1' when the COUNTER reaches PERIOD value
  - Write the value '1' (CLEAR) to the UNDERFLOW_MODE [5:4] field in the TR_PWM_CTRL register to set the LINE_OUT signal to '0' when the COUNTER reaches '0'
  - Write the value '2' (INVERT) to the CC0_MATCH_MODE [1:0] field in the TR_PWM_CTRL register to invert the LINE_OUT signal when the COUNTER equals CC0 value

Left align PWM

Figure 147 illustrates a PWM in COUNT_UP mode with different CC0 values. The figure also illustrates how a left-aligned PWM can be creating using the PWM in COUNT_UP mode.

Note:

- CC0 is changed (to CC0_BUFF, which is not depicted) on a tc event
- Switch event is constant “1”

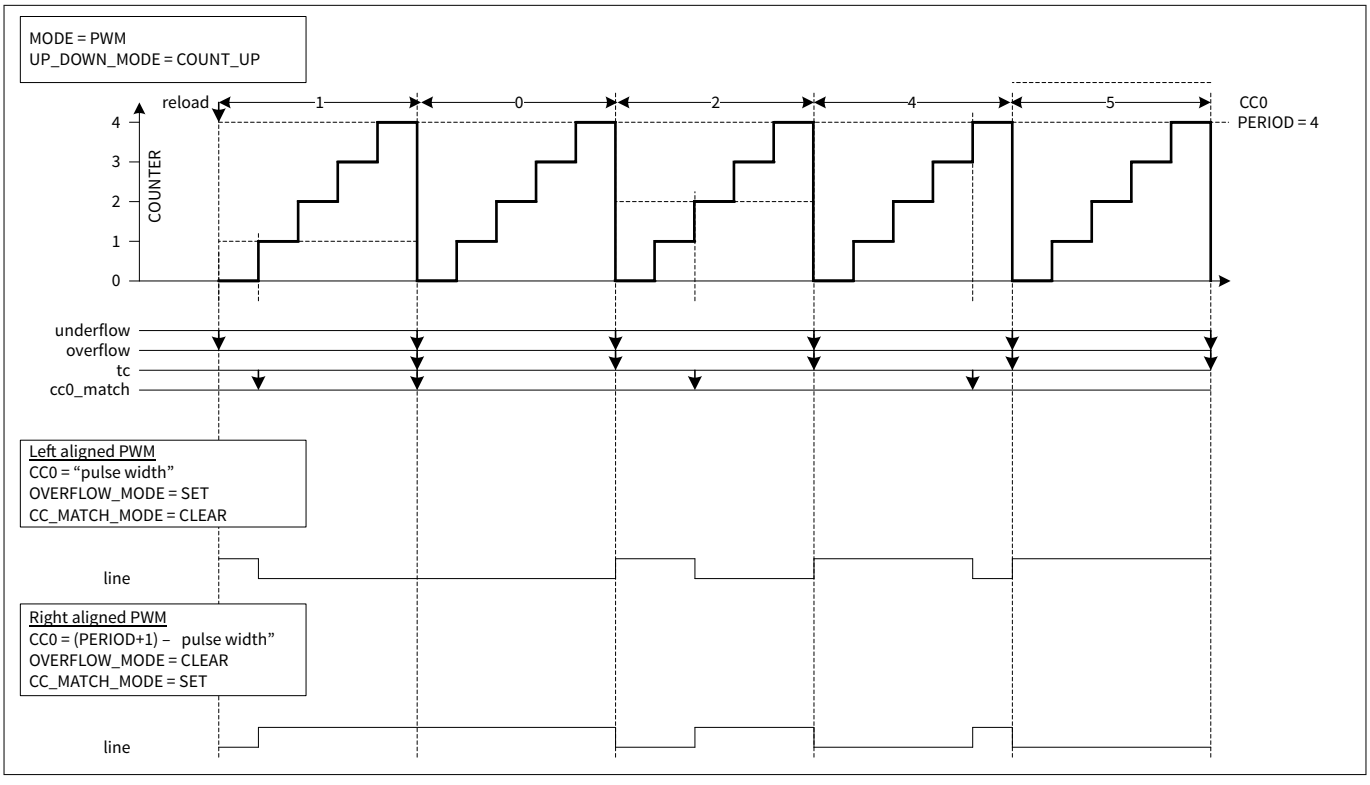


Figure 147 Left align PWM

Right align PWM

Figure 148 illustrates a PWM in COUNT_DOWN mode with different CC0 values. The figure also illustrates how a right-aligned PWM can be created using the PWM in COUNT_DOWN mode.

Note:

- CC0 is changed (to CC0_BUFF, which is not depicted) on a TC event
- Switch event is constant “1”

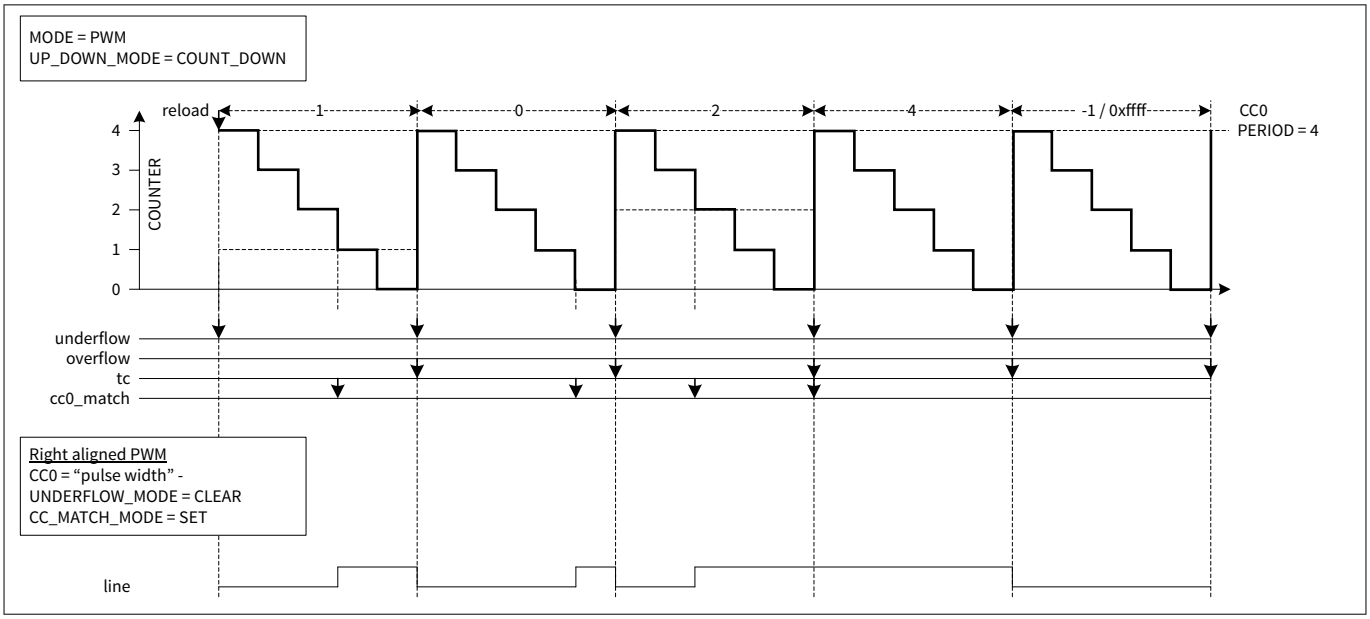


Figure 148 Right align PWM

Center align PWM

Figure 149 illustrates a PWM in COUNT_UPDN1 with different CC0 values. The figure also illustrates how a center-aligned PWM can be creating using the PWM in COUNT_UPDN1 mode.

Note:

- The switch event is generated by hardware trigger 1, which is a constant ‘1’ and therefore, always active at the TC condition
- CC0 is changed (to CC0_BUFF, which is not depicted) on a TC event
- cc0_match is generated at the beginning of the count period at the same time the CC0 switches to “0”. It is a special case that cc0_match generate without “COUNTER equals CC0” (CC0 changes to “0” at the same time COUNTER changes to “1” from “0”)

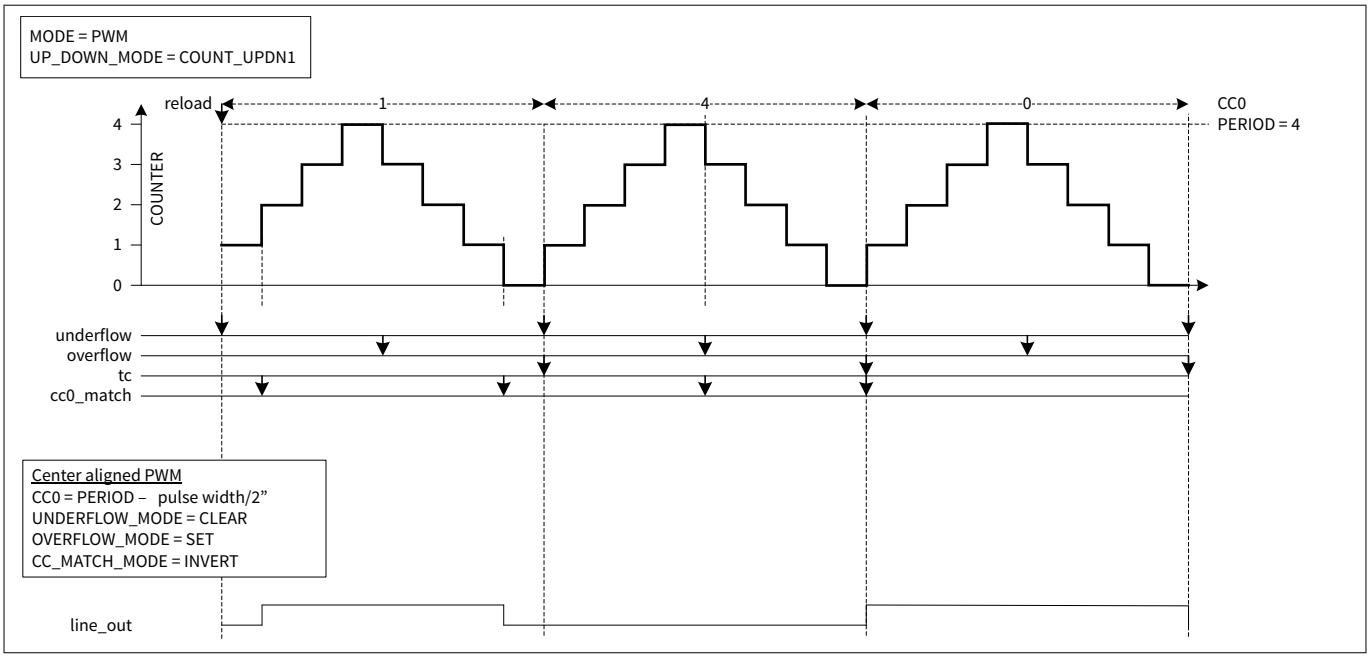


Figure 149 Center align PWM

Figure 150 illustrates another corner case that CC0 equals 0 when reload event comes. The actual counter value before the reload event is NOT used, instead the counter value before the reload event is considered to be 0. As a result, when the first CC0 value at the reload event is 0, a cc0_match event is generated.

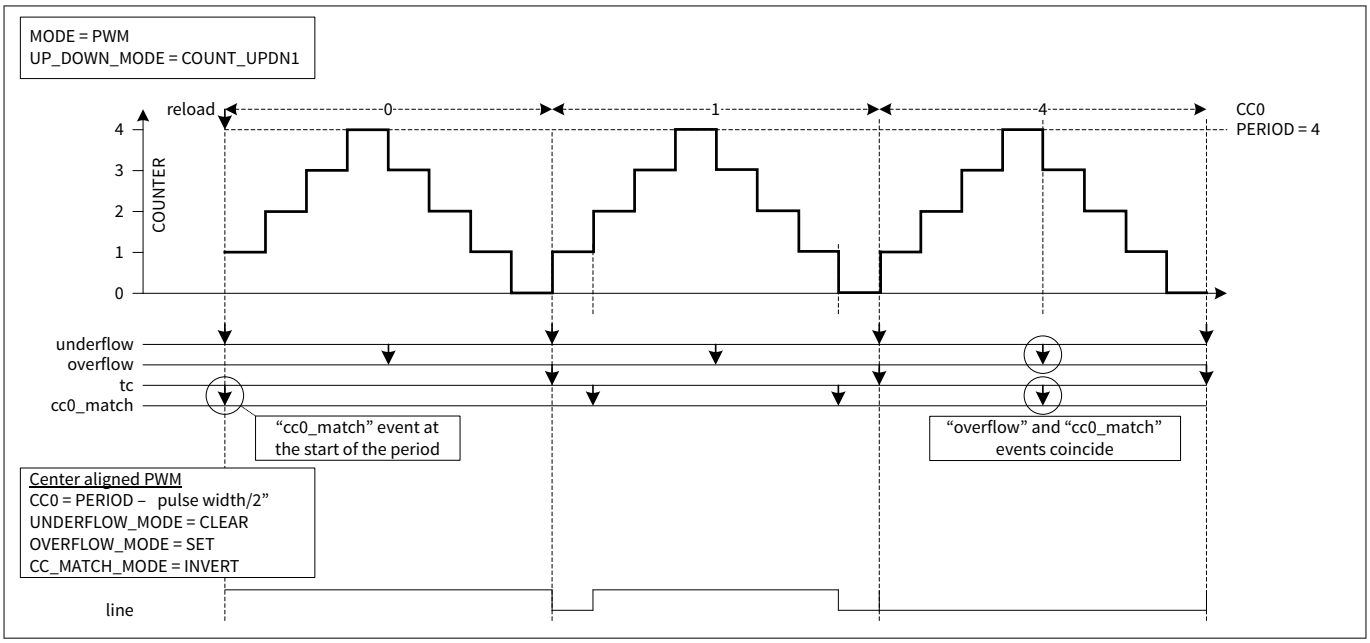


Figure 150 Center align PWM with CC0 = “0” after reload

Asymmetric PWM

The PWM mode supports the generation of an asymmetric PWM. For an asymmetric PWM, the “line” pulse is not necessarily centered in the middle of the period. This functionality is realized by having a different CC0 value when counting up and when counting down. The CC0 and CC0_BUFF values are exchanged on an overflow event. Note that this restricts the asymmetry of the generated “line” pulse.

Following are the settings required for asymmetric PWM:

- Asymmetric PWM: COUNT_UPDN2 mode with “overflow=SET, underflow=CLEAR, cc0_match=INVERT”

The asymmetric PWM typically uses the same period value when counting up and counting down. When PERIOD and PERIOD_BUFF are switched on a TC event (overflow or underflow event), ensure that:

- Within a PWM period, the period values are the same (an overflow switch of PERIOD and PERIOD_BUFF should NOT change the period value; that is, PERIOD_BUFF should be PERIOD)
- Between PWM periods, the period value can change (an underflow switch of PERIOD and PERIOD_BUFF may change the period value; that is, PERIOD_BUFF may be different from PERIOD)

Figure 151 illustrates how the COUNT_UPDN2 mode is used to generate an asymmetric PWM.

Note:

- A TC event is generated for both an underflow and overflow event. The TC event is used to exchange the CC0 and CC0_BUFF values

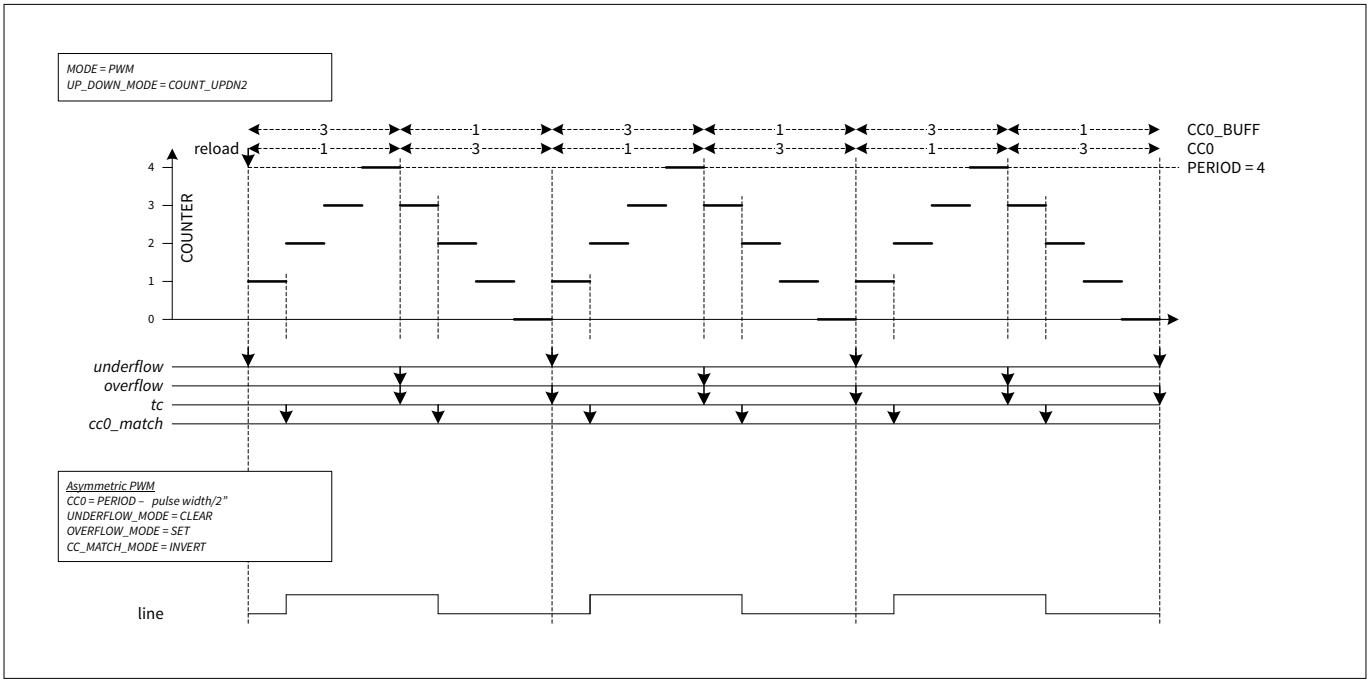


Figure 151 Asymmetric PWM

The previous waveform-illustrated functionality when the CC0 values are neither “0” not PERIOD. Corner case conditions in which the CC0 values equal “0” or PERIOD are illustrated below.

***Note**:*

- *When up counting, when the CC0 value at the underflow event is 0, a cc0_match event is generated*
- *When down counting, when the CC0 value at the overflow event is PERIOD, a cc0_match event is generated*
  - *A tc event is generated for both an underflow and overflow event. The tc event is used to exchange the CC0 and CC0_BUFF values*
- *Software updates CC0_BUFF and PERIOD_BUFF in an interrupt handler on the tc event (and overwrites the hardware-updated values from the CC0/CC0_BUFF and PERIOD/PERIOD_BUFF exchanges)*

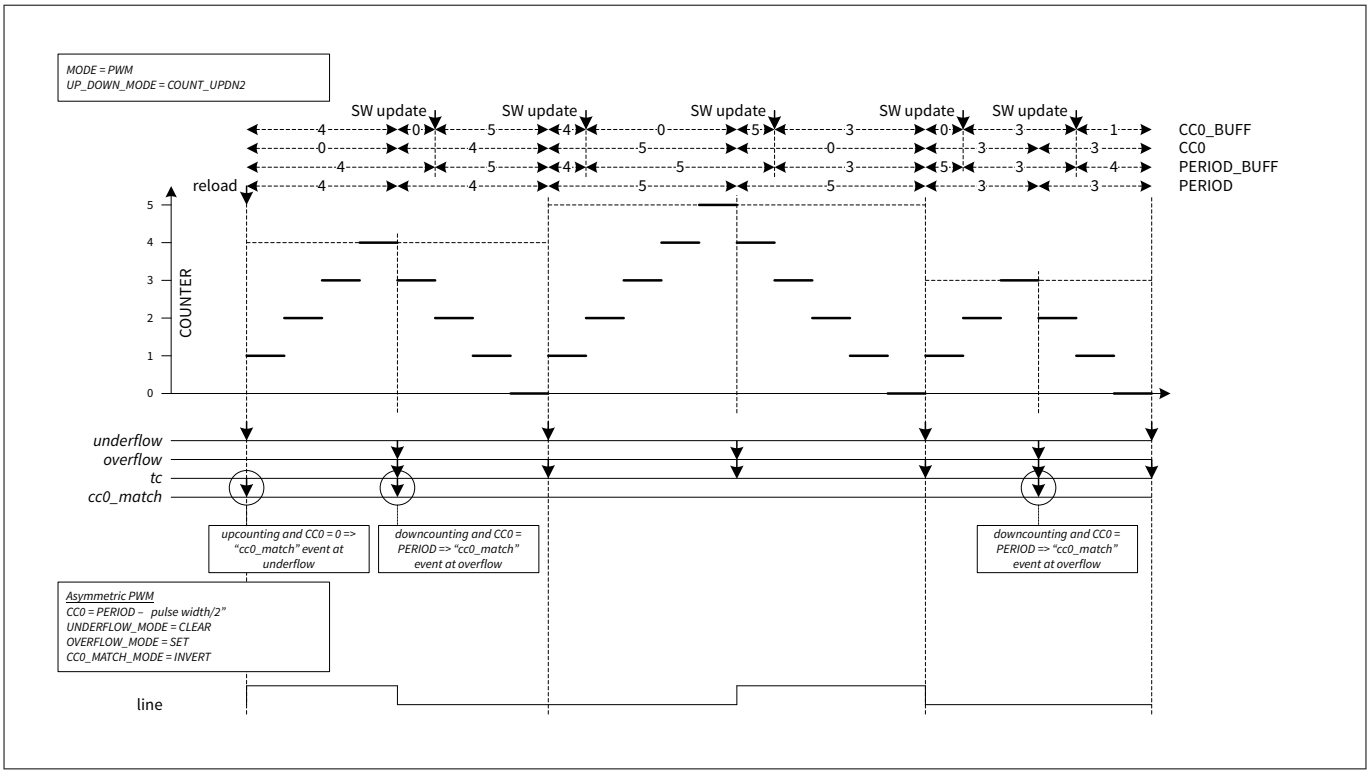


Figure 152 Asymmetric PWM with CC0=0 and CC0=PERIOD duty cycle

The compare 1 function using registers CC1 and CC1_BUFF generating the cc1_match event is the same as the compare function using registers CC0 and CC0_BUFF generating the cc0_match event.

The cc1_match event can also be used to generate the PWM output signals. Using both cc0_match and cc1_match events for PWM output signal generation provide another way to generate an asymmetric PWM as shown in the following figure.

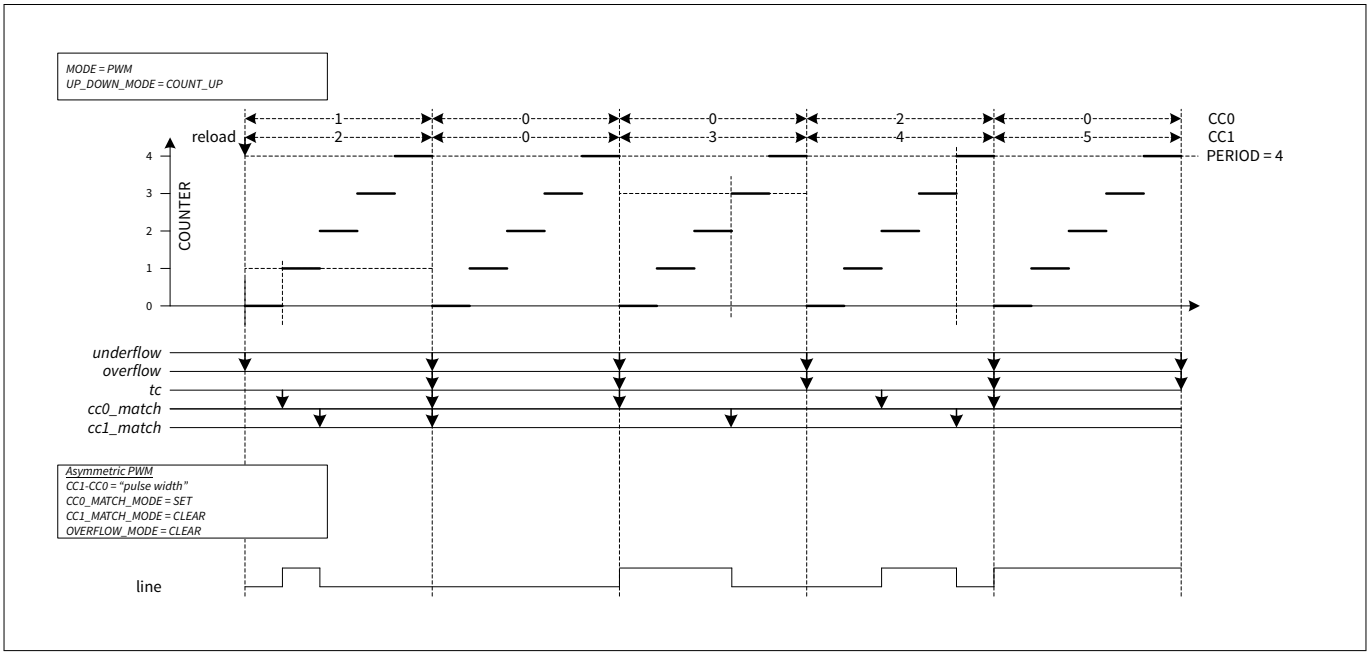


Figure 153 Asymmetric PWM by cc0_match and cc1_match in COUNT_UP mode

Such asymmetric PWM generation is more flexible than using only one compare function in the COUNT_UPDN2 mode. However, if another (3rd) compare function is needed, for example, to trigger an ADC, another synchronously running counter has to be used.

The generation of compare match 0 and compare match 1 events can be enabled or disabled individually for up and down counting (during COUNT_UPDN1/2 mode). That allows an asymmetric PWM generation in COUNT_UPDN1 mode where one compare match event modifies the PWM output only while counting up and the other compare match event modifies the PWM output only while counting down. This is illustrated in the Figure 154, which shows one of three center aligned PWM phases for motor control during a time where the duty cycle value is increased from one period to the next (rising part of sign wave modulated onto the PWM signal).

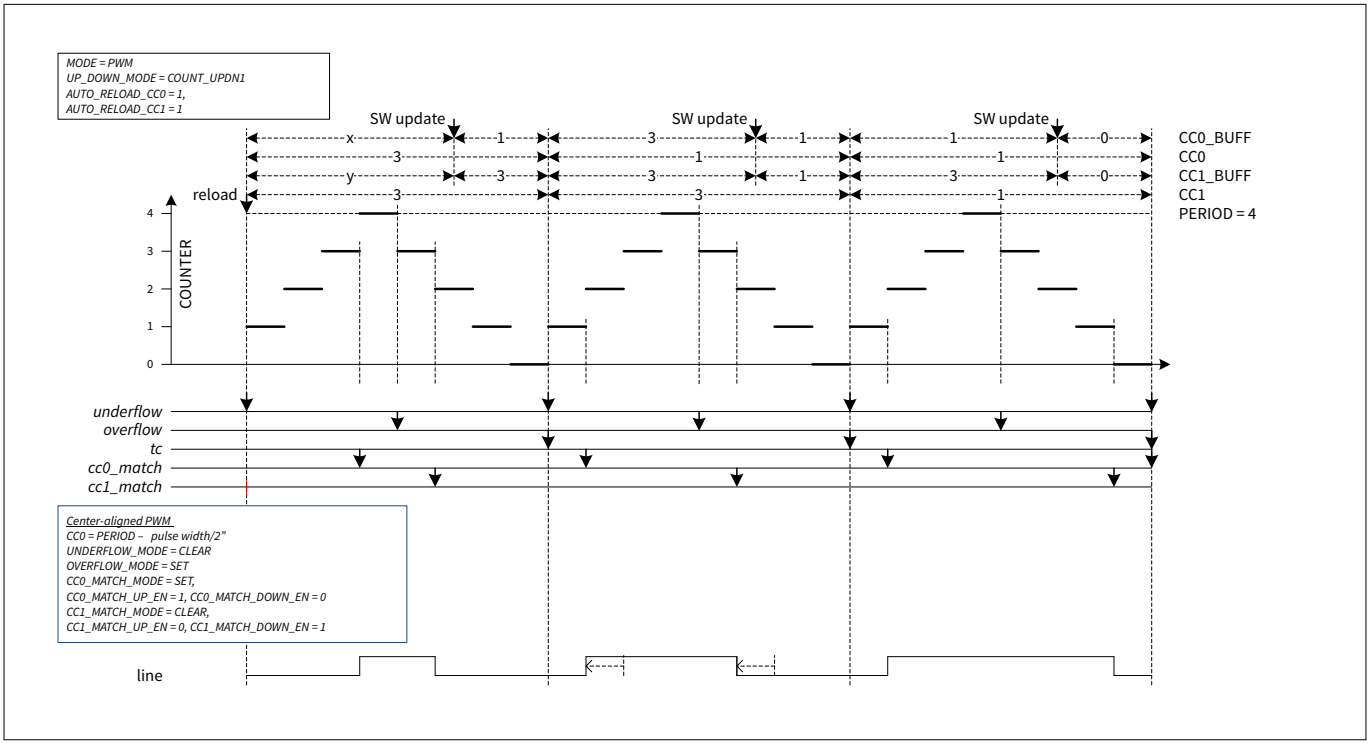


Figure 154 Asymmetric PWM by cc0_match and cc1_match in COUNT_UPDN1 mode

Instead of an always center-aligned PWM the phase of the PWM signal can be temporarily shifted to allow a single shunt current measurement (current measurement at two triggers with difference calculation in software) for motor control of a PMSM (permanent-magnet synchronous motor) when the current duty cycle values of the 3 phases do not allow that (too small window where 1 PWM channel is active, 2 others not). Compared to the asymmetric PWM realized with only one compare function in the COUNT_UPDN2 mode this solution uses two independent buffered compare values and generates less CPU load (less interrupts needed), that is, all updates can be done in the 'ADC done' interrupt service routine calculating the new duty cycle values and introducing a temporary phase shift for single shunt current measurement.

The ways to enable asymmetric PWM using the 2nd compare function:

- COUNT_UP/DOWN mode with "cc0_match =SET, cc1_match=CLEAR". Pulse width = CC1-CC0
- COUNT_UPDN1/2 mode with "overflow=SET, underflow=CLEAR, cc0_match=SET, cc1_match=CLEAR, CC0_MATCH_UP_EN=1, CC0_MATCH_DOWN_EN=0, CC1_MATCH_ UP_EN=0, CC1_MATCH_DOWN_EN=1". Pulse width = 2xPERIOD – CC1 – CC0

Kill mode

PWM mode has different stop/kill modes. The mode is specified by PWM_IMM_KILL, PWM_STOP_ON_KILL, PWM_SYNC_KILL, PWM_TC_SYNC_KILL_DT and PWM_SYNC_KILL_DT.

The configuration option PWM_IMM_KILL specifies when the PWM output signals are started to be suppressed.

- PWM_IMM_KILL is '1'. The PWM output signals DT_LINE_OUT and DT_LINE_COMPL_OUT are immediately suppressed when a kill event is detected
- PWM_IMM_KILL is '0'. The PWM output signals DT_LINE_OUT and DT_LINE_COMPL_OUT are suppressed synchronously with the next count clock after a kill event is detected

The following figures illustrate both configurations.

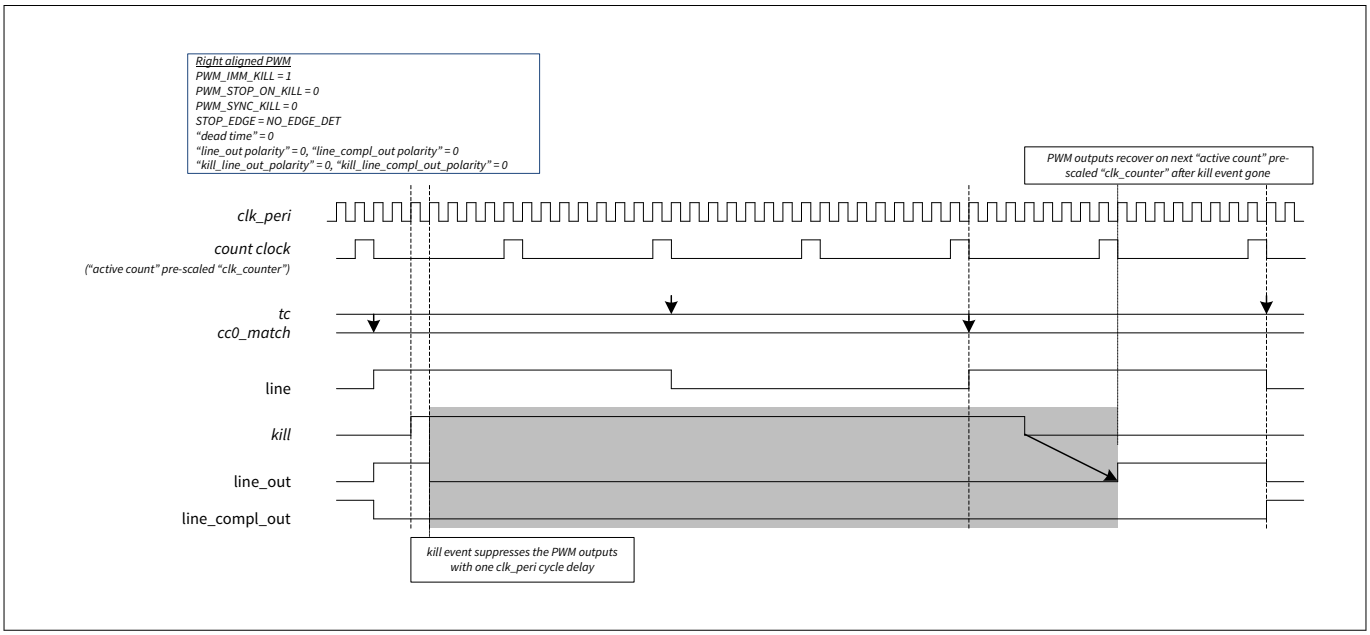


Figure 155 Kill suppresses line output immediately (PWM_IMM_KILL = 1)

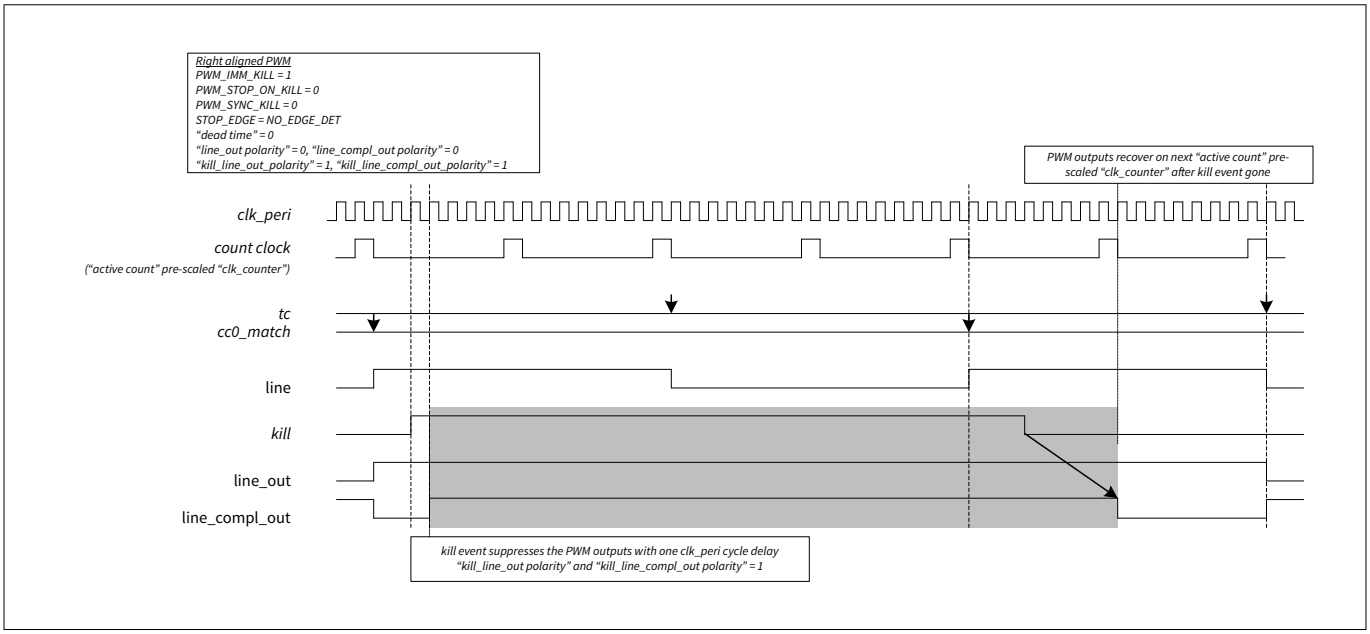


Figure 156 Kill suppresses line output immediately (PWM_IMM_KILL = 1) with "kill_line_out_polarity" and "kill_line_compl_out_polarity" = 1

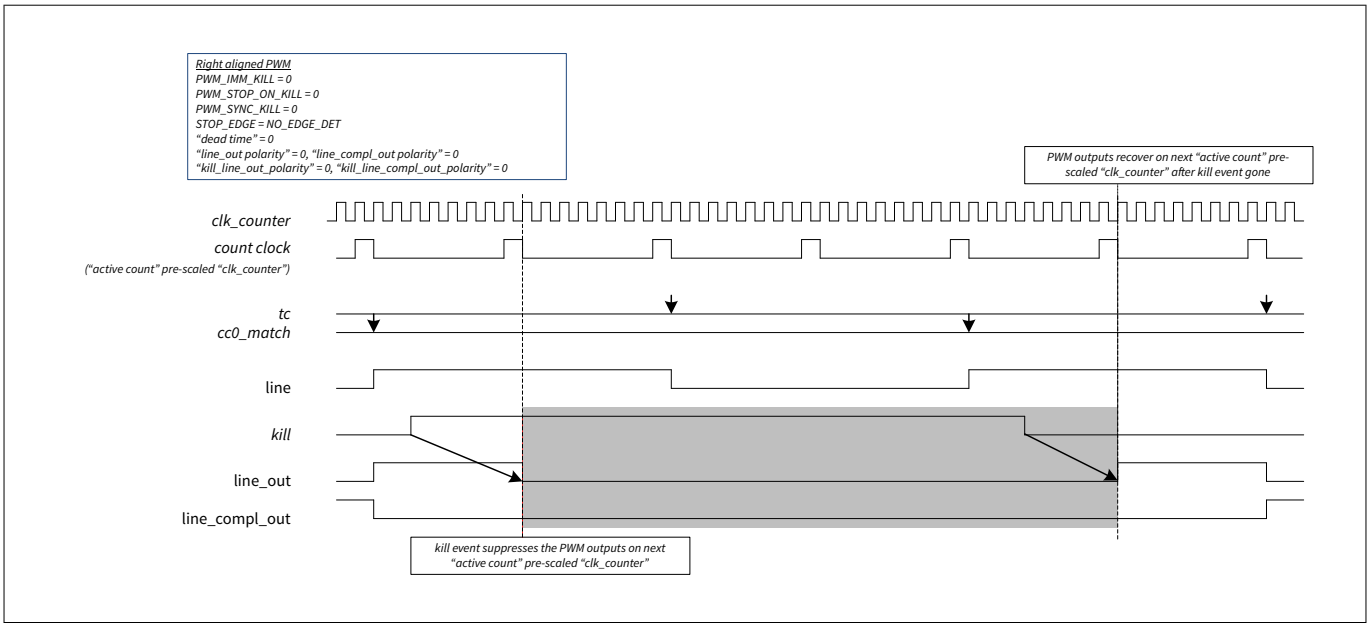


Figure 157 Kill suppresses line output by count clock (PWM_IMM_KILL = 0)

The following figures illustrate the 5 modes mentioned earlier, with "kill_line_out_polarity" and "kill_line_compl_out_polarity" set to 0 and 1.

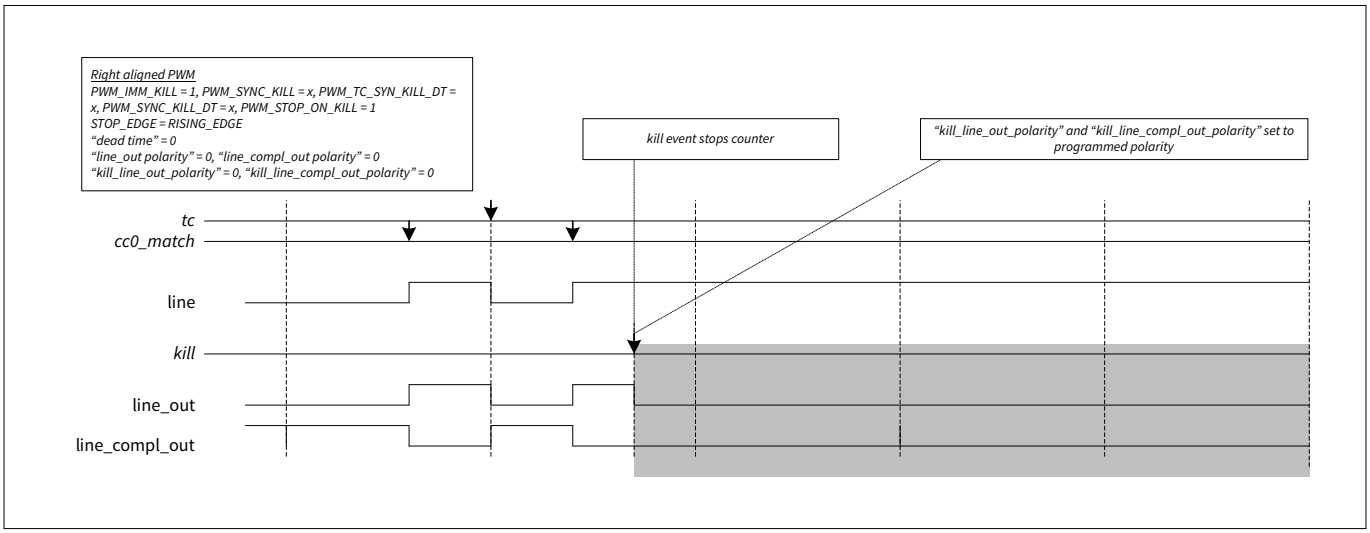


Figure 158 Stop on kill

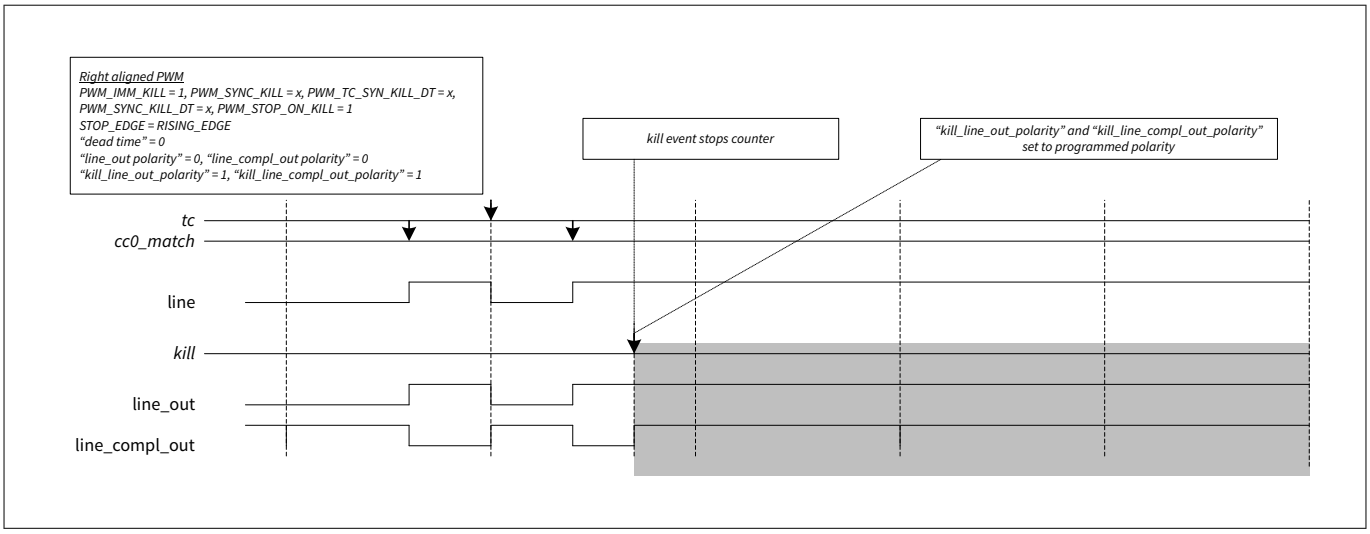


Figure 159 Stop on kill with “kill_line_out_polarity” and “kill_line_compl_out_polarity” = 1

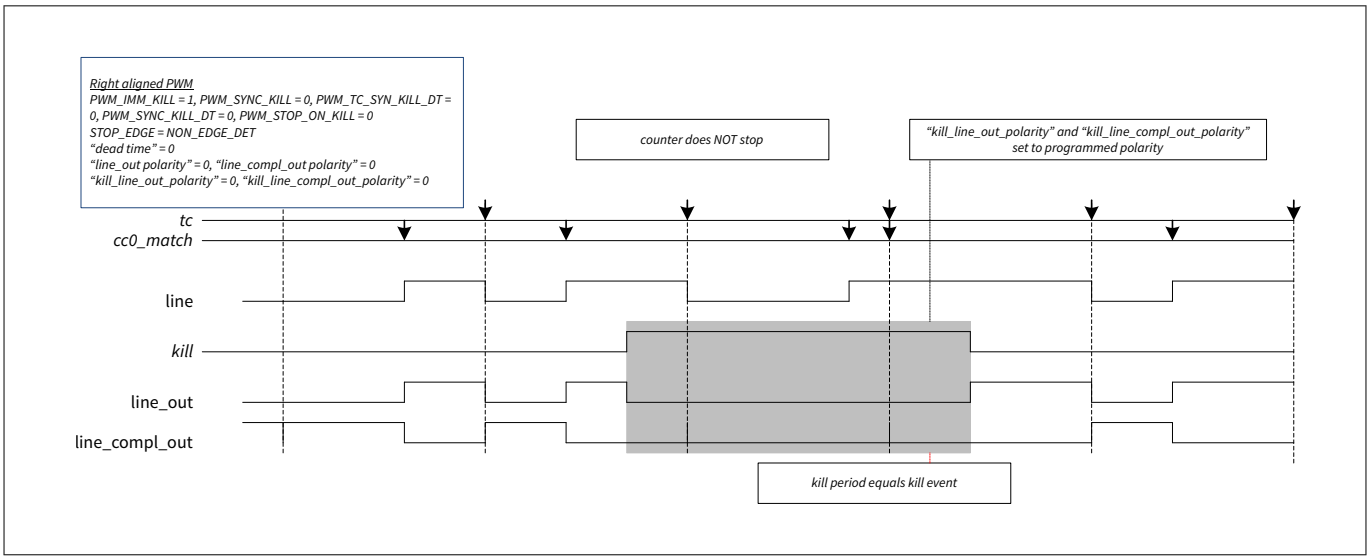


Figure 160 Asynchronous kill

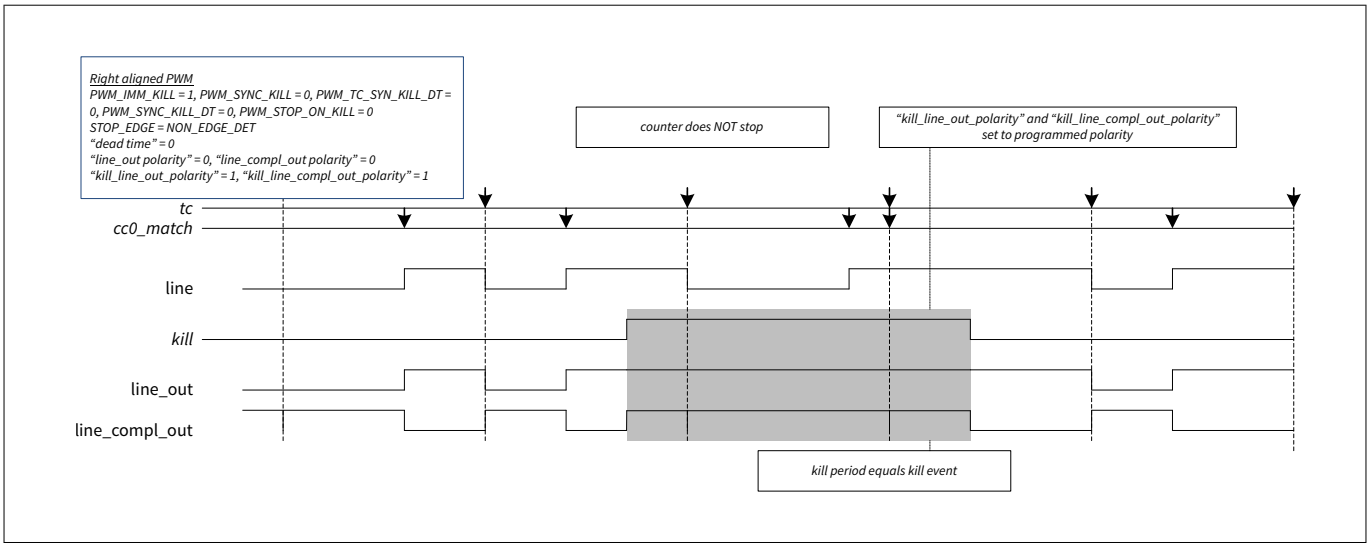


Figure 161 Asynchronous kill with "kill_line_out_polarity" and "kill_line_compl_out_polarity" = 1

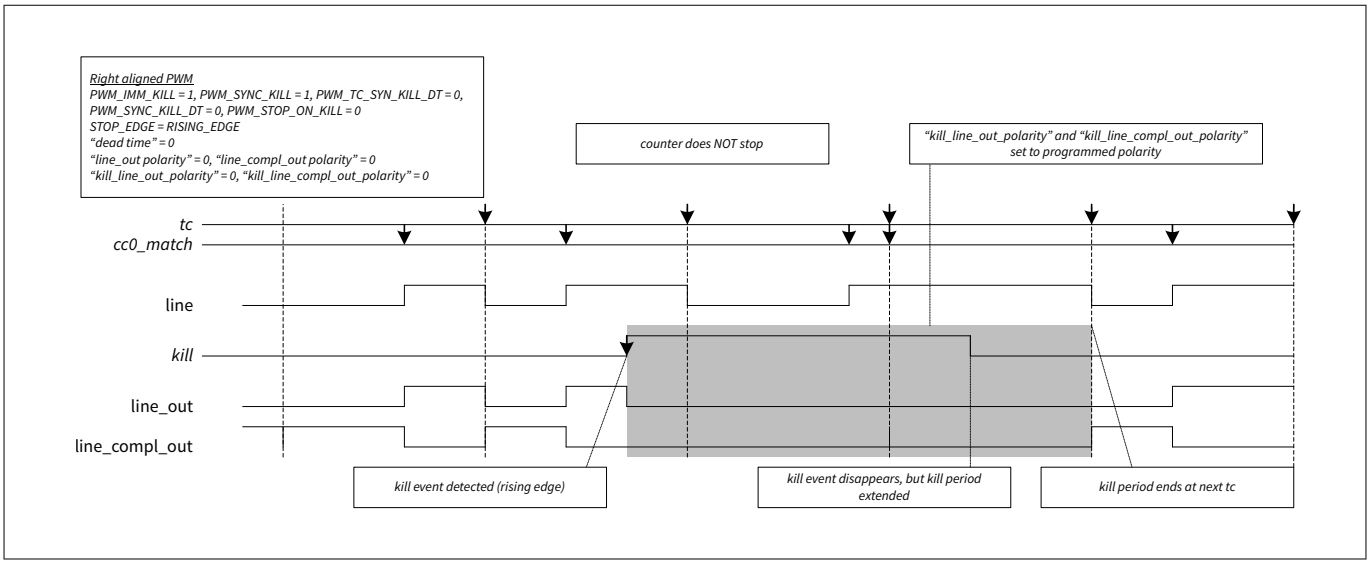


Figure 162 Synchronous kill

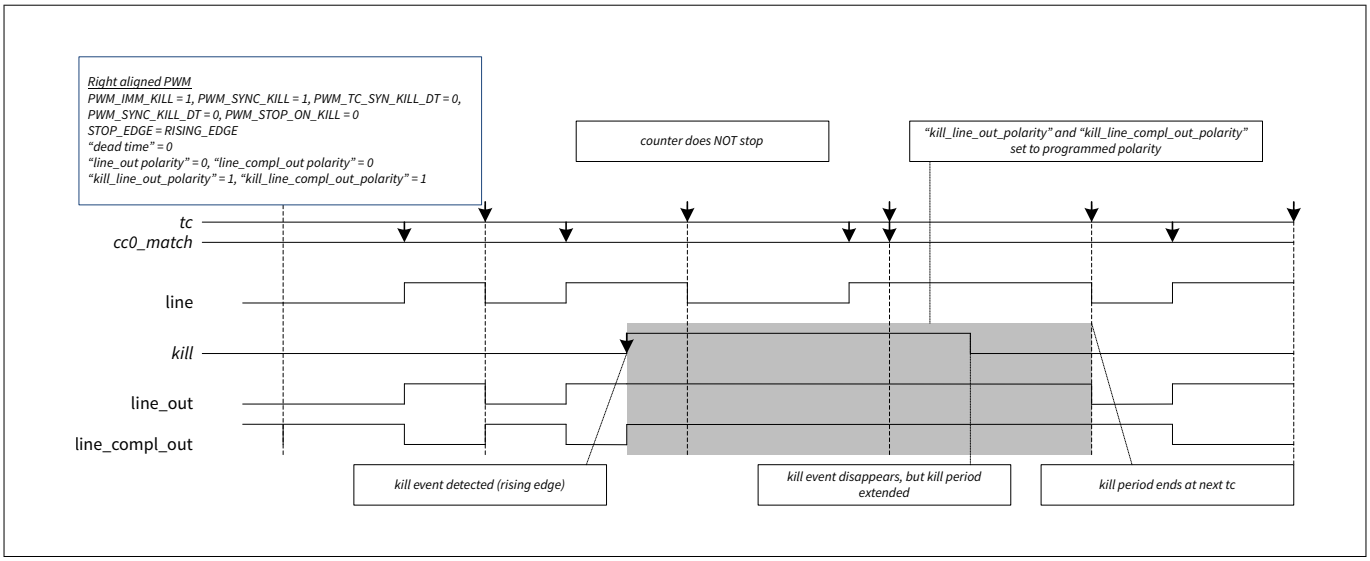


Figure 163 Synchronous kill with “kill_line_out_polarity” and “kill_line_compl_out_polarity” = 1

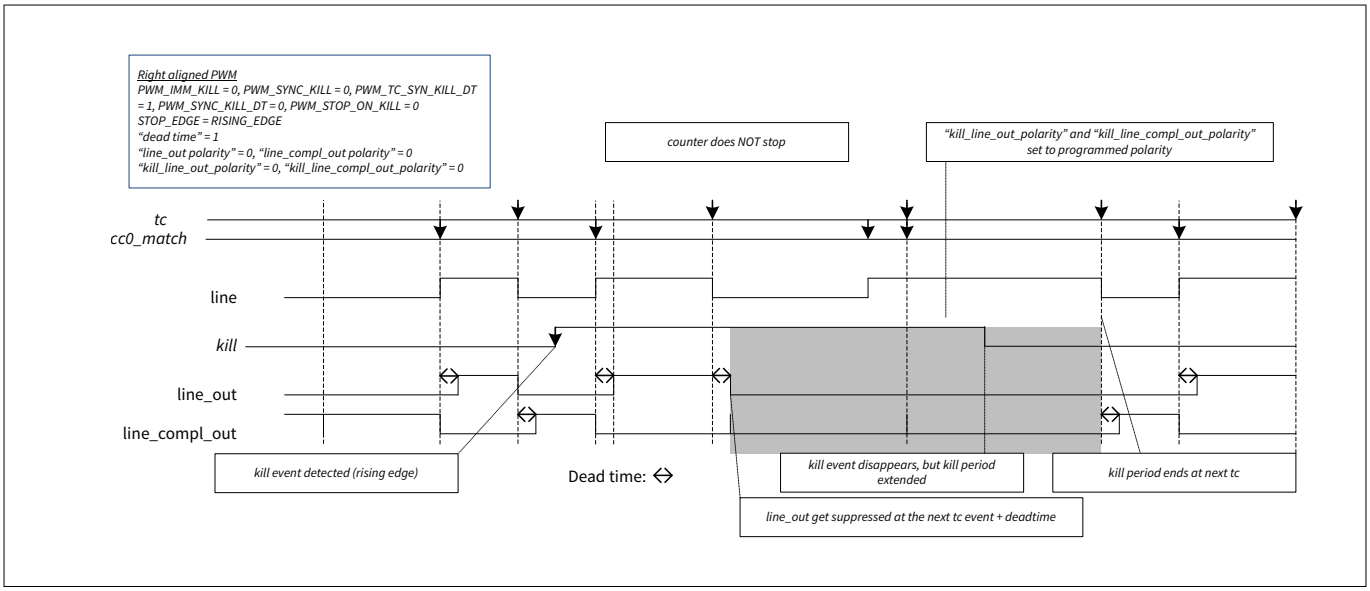


Figure 164 TC synchronous DT kill

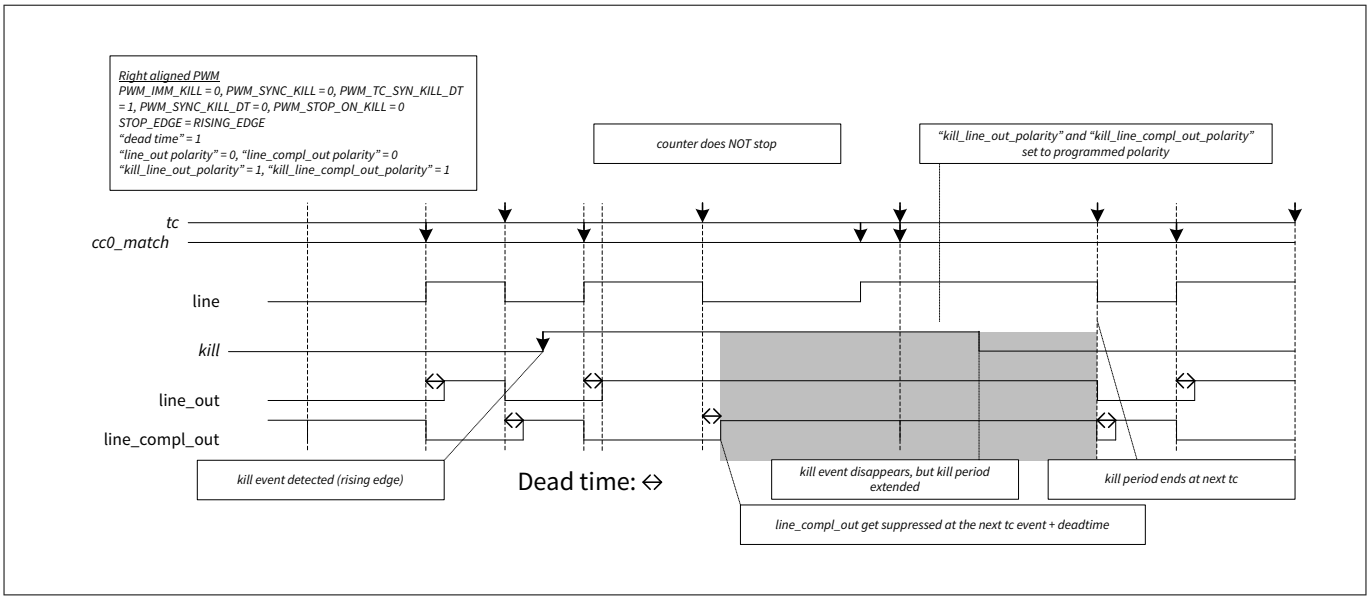


Figure 165 TC synchronous DT kill with "kill_line_out_polarity" and "kill_line_compl_out_polarity" = 1

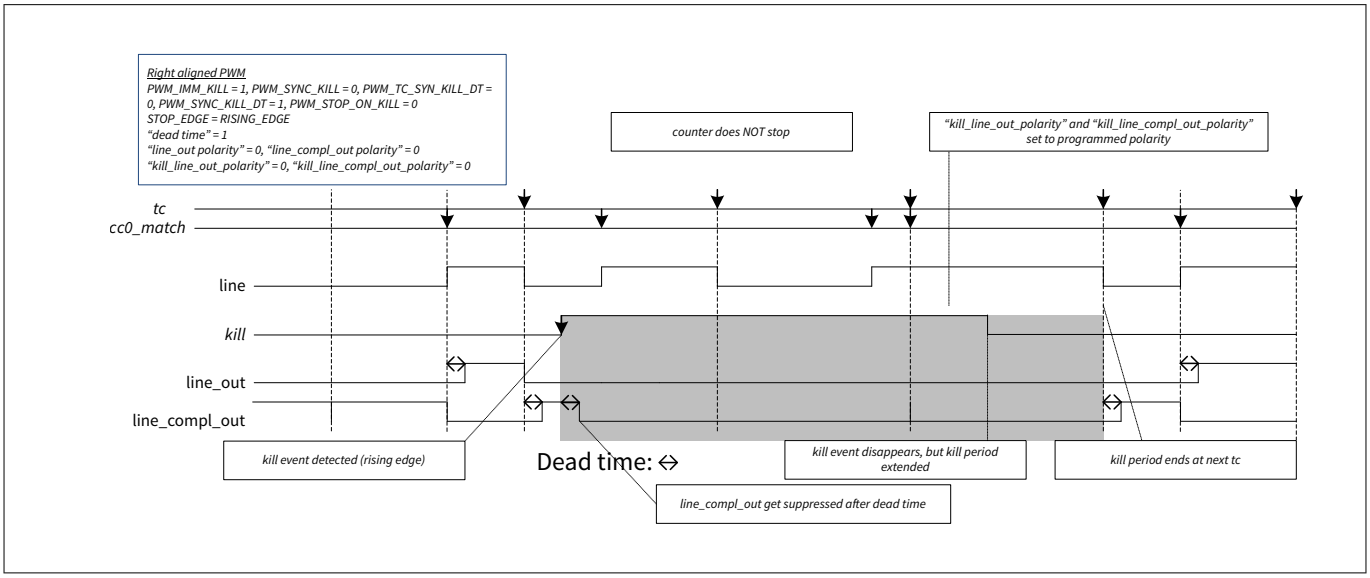


Figure 166 Synchronous DT kill

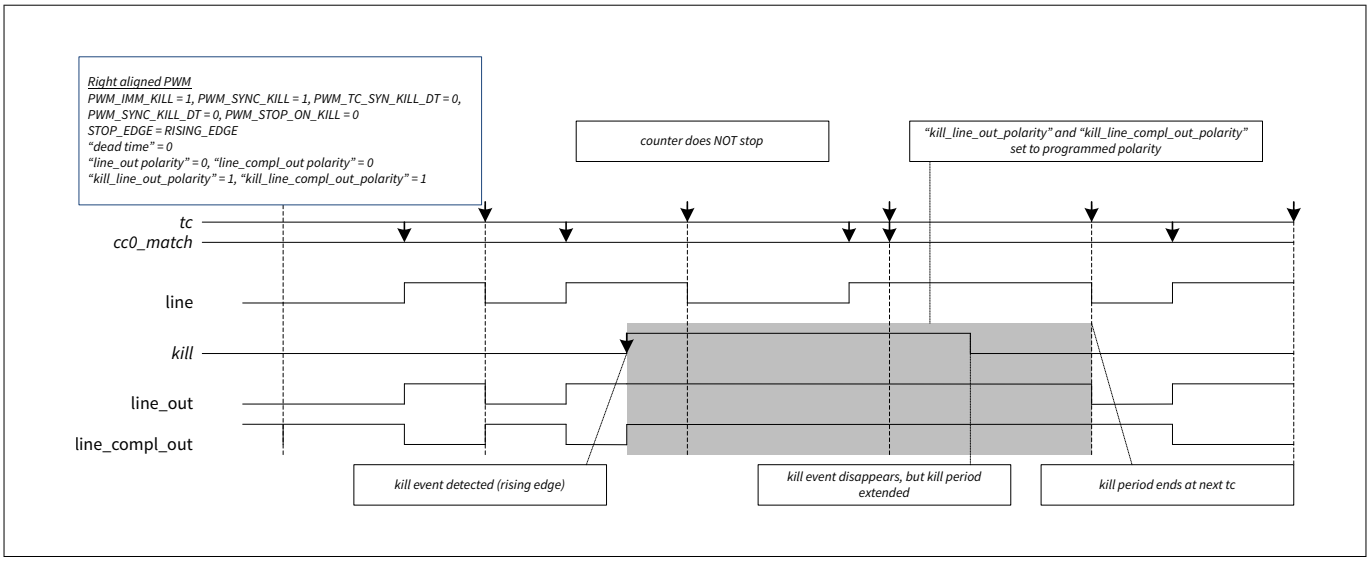


Figure 167 Synchronous DT kill with "kill_line_out_polarity" and "kill_line_compl_out_polarity" = 1

Different from asynchronous kill, synchronous kill will stop suppressing line output on tc event after kill is gone.

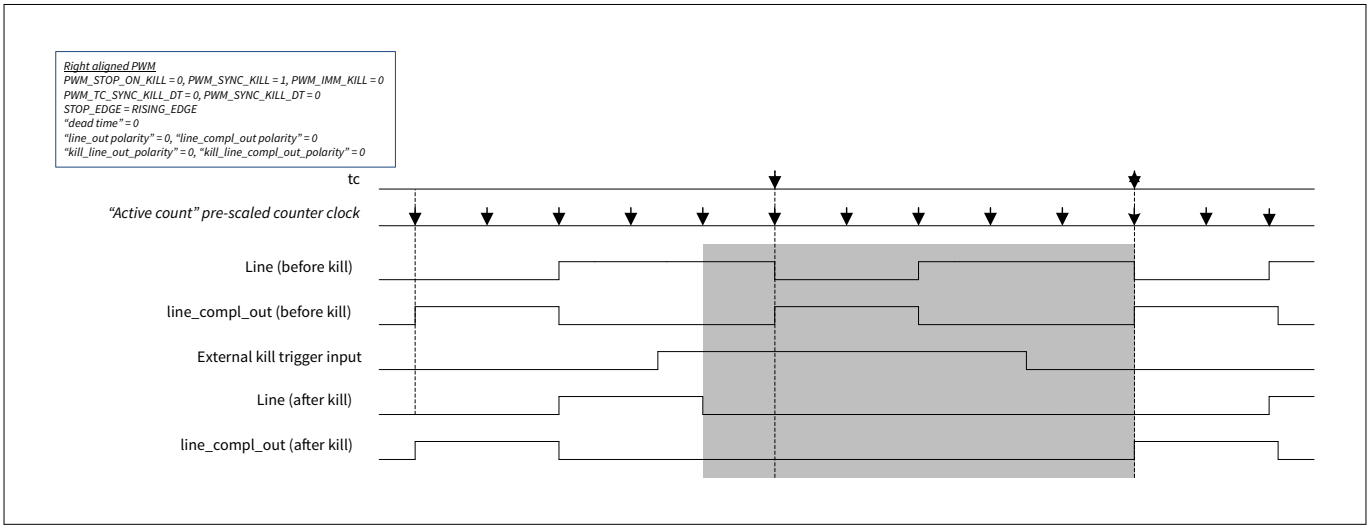


Figure 168 Synchronous kill timing (PWM_IMM_KILL = 0)

Whatever the value of PWM_IMM_KILL is, PWM recovers on the next "active count" pre-scaled counter clock (PWM_SYNC_KILL=0) or next TC (PWM_SYNC_KILL=1) after the kill event is inactive. An exception is that when PWM_SYNC_KILL= "0" and PWM_IMM_KILL= "1", pulse width of kill event is too narrow to be recognized by the "active count" pre-scaled counter clock. The PWM output recovers immediately as illustrated in the following figure.

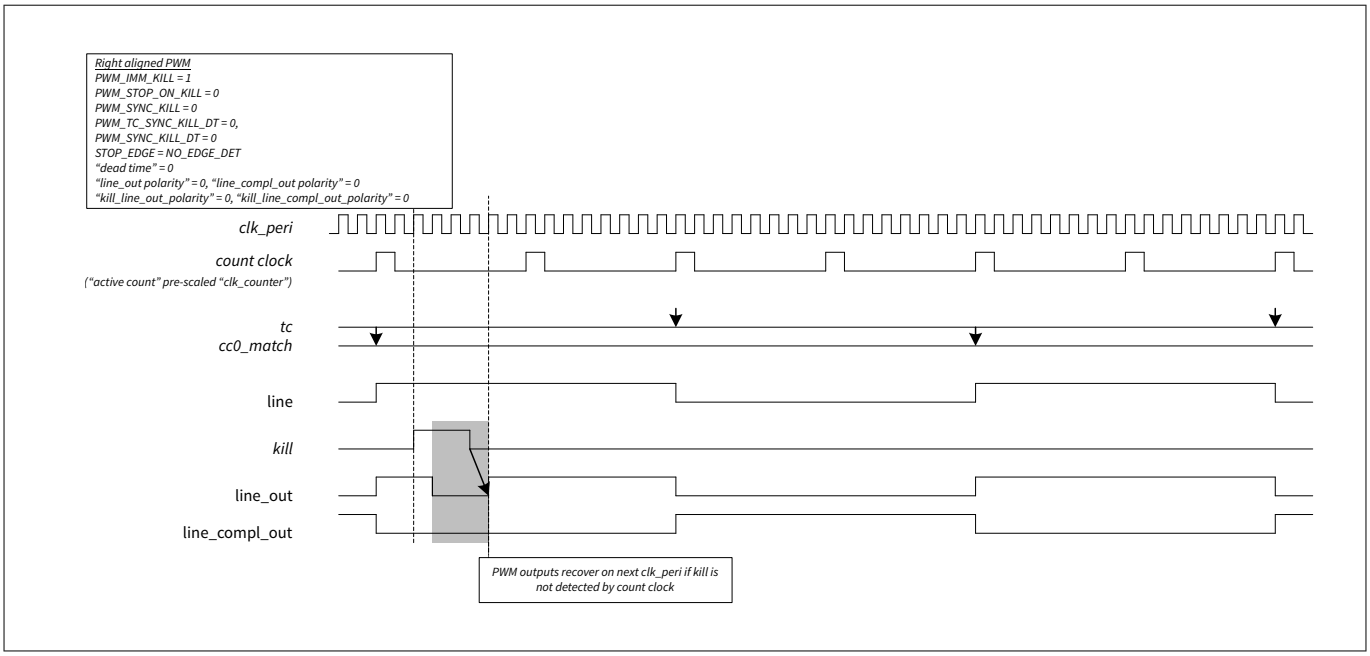


Figure 169 PWM outputs recover on next clk_peri if kill is not detected by count clock

Capture1 event also works as a kill input, just like the stop event. Both events are OR combined and share the same kill mode settings. Having two stop/kill events for a PWM allows selecting for one stop/kill event a common trigger from a PERI trigger multiplexer (allowing synchronous stop/kill operation of multiple PWMs) while selecting for the other stop/kill event, for example, a dedicated ADC out-of-range trigger (for example, allowing real-time hardware stop of a PWM when current of PWM driven signal is out of range).

25.3.4.6 PWM output selection for stepper motor control

In PWM mode, the TCPWM counters can be used for Stepper Motor Control including micro stepping. Therefore, the two PWM output signals “line_out” and “line_compl_out” (which are usually a pair of complementary PWM signals during normal PWM operation) can be set to the following options using an output select register (register LINE_SEL, fields OUT_SEL and COMPL_OUT_SEL):

- constant low (‘0’)
- constant high (‘1’)
- PWM signal “line”
- Inverted PWM signal “line”
- ‘Z’ (high impedance)

***Note**: This feature generates digital signals for stepper motor control. These signals cannot directly drive a stepper motor, as they lack sufficient current strength.*

Constant low or high values are used for full stepping while a PWM signal is used for micro stepping. With the option “Z” the TCPWM output signal “line_out_en” or “line_compl_out_en” can be set to ‘0’ which causes the connected IO cell to not drive the output. This allows it to measure the induced voltage of a currently not driven coil via ADC, which is used for zero point detection.

***Note**: For full-stepping, set the TCPWM counter to PWM mode. However, a PWM generation is not needed, that is, the compare function resources (CC0/CC1) are free to be used for other purposes (for example, trigger generation).*

For each coil of a stepper motor, a separate TCPWM counter is used. These counters are running synchronously (sharing the same reload/start/stop/count events and the same period). To achieve a synchronous update of the output signals across multiple TCPWM counters, the same double buffering and switching method is used

for updating the CC0/CC1 or PERIOD registers. Therefore, a LINE_SEL_BUFF register is used which is exchanged with the LINE_SEL register on a TC event with an actively pending switch event (when specified by CTRL.AUTO_RELOAD_LINE_SEL). This is shown in the following figure of an example driving the two coils of a stepper motor with two TCPWM counters when CTRL.SWAP_ENABLE = “1”.

It illustrates three full steps (180°, 270°, 0°(=360°) electrical angles), followed by three micro steps using a PWM (~30°, ~60°, ~120° electrical angles). It is just an illustration example showing one step per PWM counter PERIOD. In reality, the steps for micro stepping are significantly slower compared to the PWM period and therefore, the PWM duty cycles (CC0) and/or output select registers (LINE_SEL) are stable over many PWM periods before they get changed by a switch event.

The time base for the steps can be realized by an additional TPCWM counter in timer mode (running on a slower counter clock or with a higher PWM period or counting overflow events of the PWM counter). This counter can generate interrupts that trigger the CPU to update PWM duty cycles and/or output select registers and to finally generate the switch event using the PERI0_TR_CMD register in the MXPERI block.

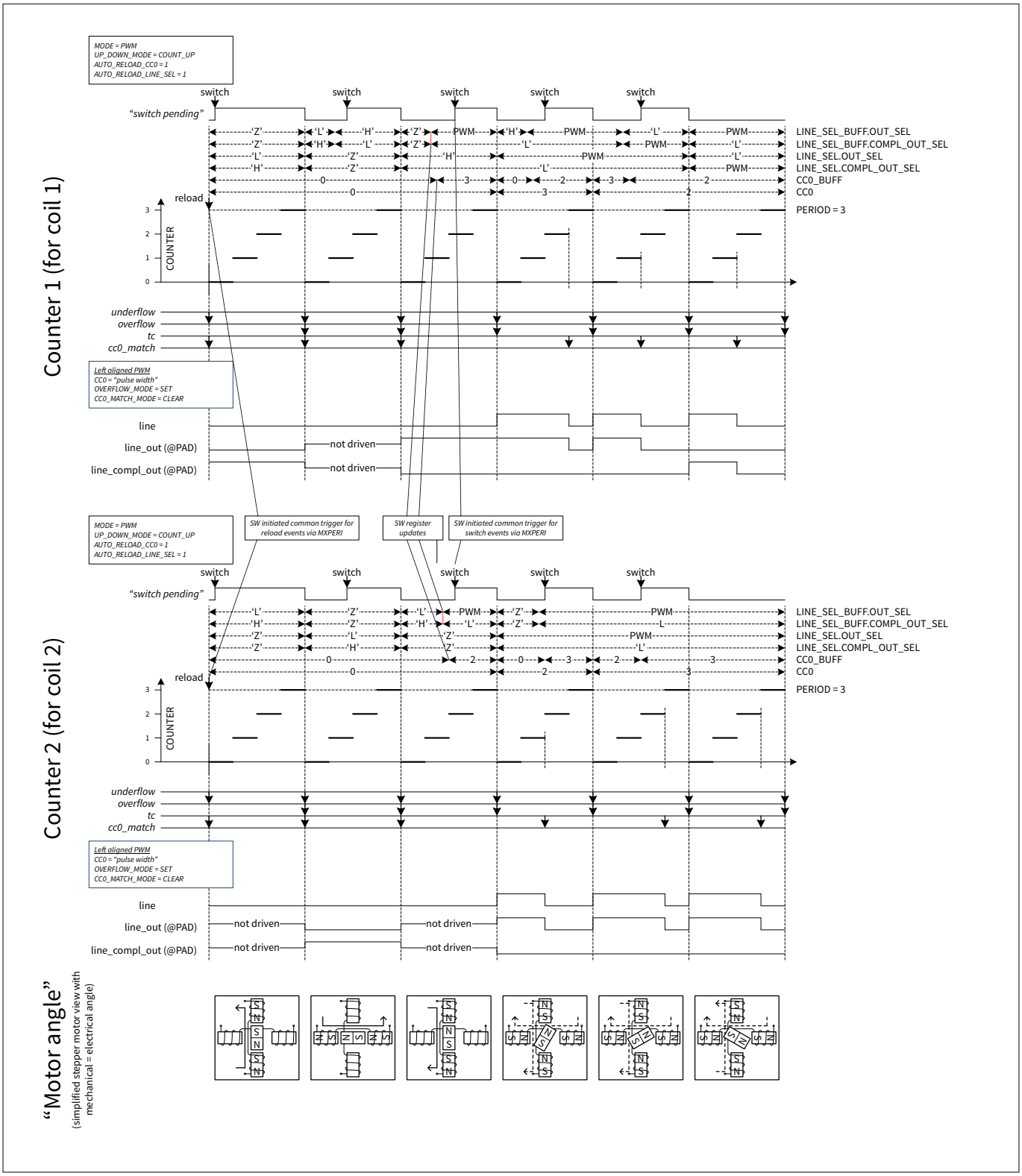


Figure 170 PWM output for stepper motor

The following figure shows an example of driving the two coils of a stepper motor with two TCPWM counters when CTRL.SWAP_ENABLE = "0". It illustrates how the software can update the register for the stepper motor control.

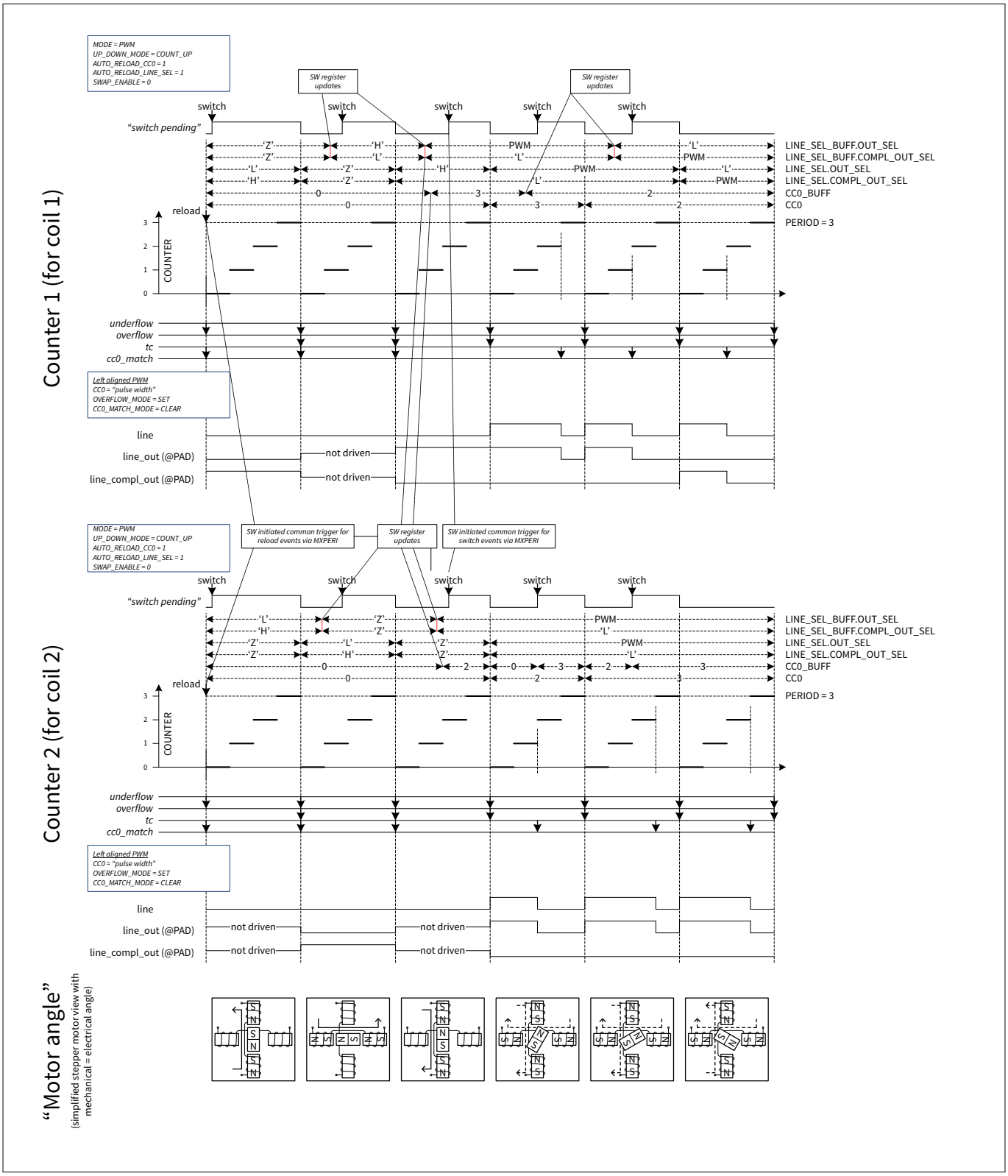


Figure 171 PWM output for stepper motor (SWAP_ENABLE=0 case)

25.3.4.7 How to configure counter for PWM mode

The steps to configure the counter for the PWM mode of operation and the affected register bits are as follows:

- Disable the counter by writing '0' to the ENABLE bit of the CTRL register

- Select PWM mode by writing '100' to the MODE[26:24] field of the CTRL register
- Set the required 16-bit/32-bit period in the PERIOD register and the buffer period value in the PERIOD_BUFF register to switch values, if required
- Set the 16-bit/32-bit compare value in the CC0/1 register and buffer compare value in the CC0/1_BUFF register to switch values, if required
- Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the CTRL register to configure left-aligned, right-aligned, or center-aligned PWM
- Set the PWM_IMM_KILL, PWM_STOP_ON_KILL, and PWM_SYNC_KILL fields of the CTRL register as required
- Set the TR_IN_SEL0 or TR_IN_SEL1 register to select the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
- Set the TR_IN_EDGE_SEL register to select the edge of the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
- LINE_OUT and LINE_COMPL_OUT can be controlled by the TR_PWM_CTRL register to set, reset, or invert upon cc0_match, cc1_match, overflow, and underflow conditions
- If required, set the interrupt upon TC or CC0/1 condition
- Enable the counter by writing '1' to the ENABLED bit of the CTRL register. A reload trigger must be provided through firmware (RELOAD bit in the PERI0_TR_CMD register) to start the counter if the hardware reload signal is not enabled

25.3.5 Pulse width modulation with dead time mode (PWM_DT) mode

The PWM_DT functionality is the same as PWM functionality, except for the following differences:

- PWM_DT supports dead time insertion. PWM does NOT support dead time insertion
- PWM_DT supports clock pre-scaling

Dead time insertion is a step that operates on a preliminary PWM output signal “line”.

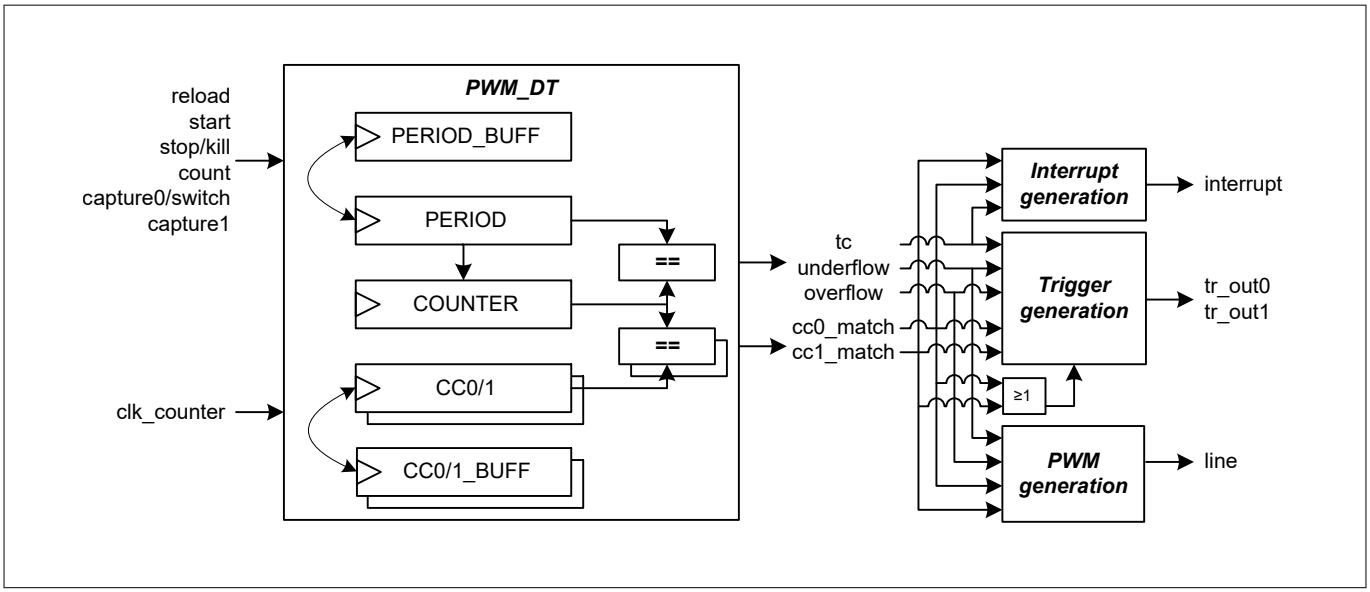


Figure 172 PWM with dead time functionality

- The dead time is a 16-bit value (max. dead time is 65535 counter cycles)
- Different dead times can be used before activating PWM output signals “line_out” and “line_compl_out”. The dead time for “line_out” is specified by the 16-bit register field DT_LINE_OUT (consists of DT_LINE_OUT_H and DT_LINE_OUT_L). The dead time for “line_compl_out” is specified by the 16-bit register field DT_LINE_COMPL_OUT

***Note**: To prevent undefined behavior, ensure that the dead time insertion (dead time value) does not exceed the period value.*

The following figure shows an example with different dead times for the PWM output signals “line_out” and “line_compl_out” when CTRL.SWAP_ENABLE = “1”.

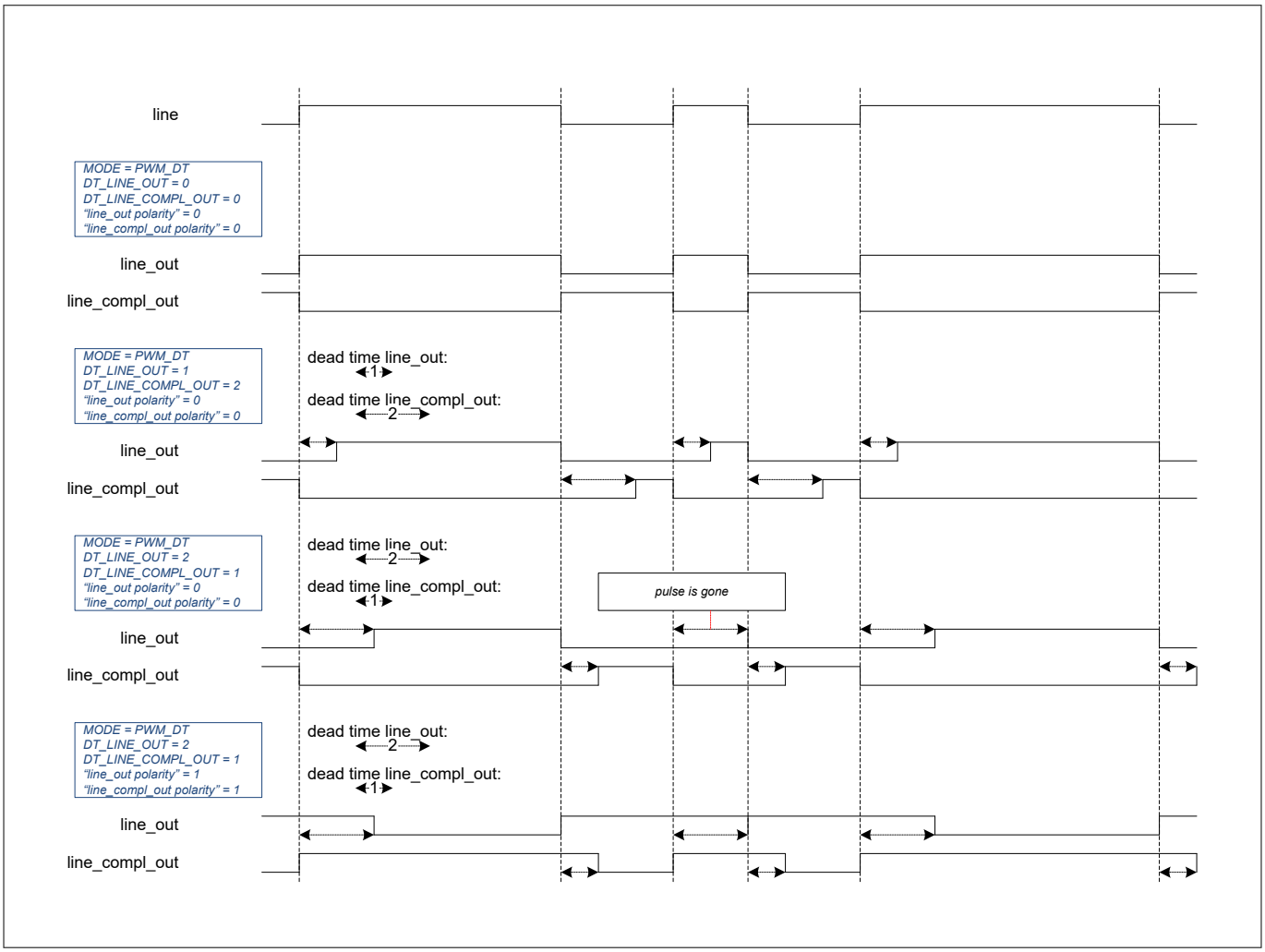


Figure 173 “line_out” and “line_compl_out” with different dead time

The following figures show an example driving the two coils of a stepper motor with two TCPWM counters when CTRL.SWAP_ENABLE = “0”. It illustrates how the software can update the register for the stepper motor control to incorporate dead times for the PWM output signals “line_out” and “line_compl_out”. Software can update the dead time in the DT_BUFF register and it will transfer to the DT register at the tc event. For COUNT_UPDN2 mode, the shadow transfer happens at the period match where TC and underflow coincides.

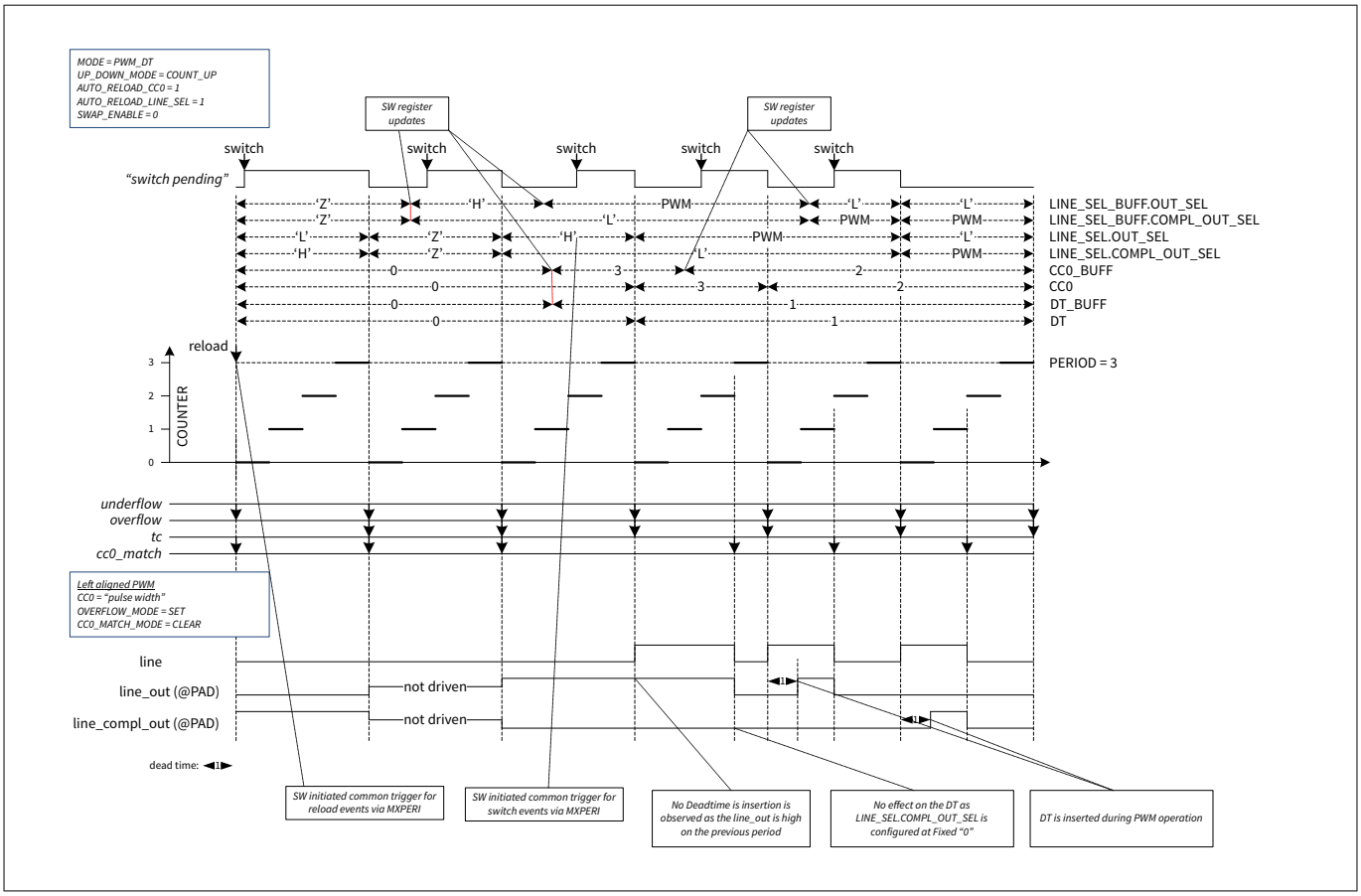


Figure 174 PWM_DT output case 1 scenario (SWAP_ENABLE=0 case)

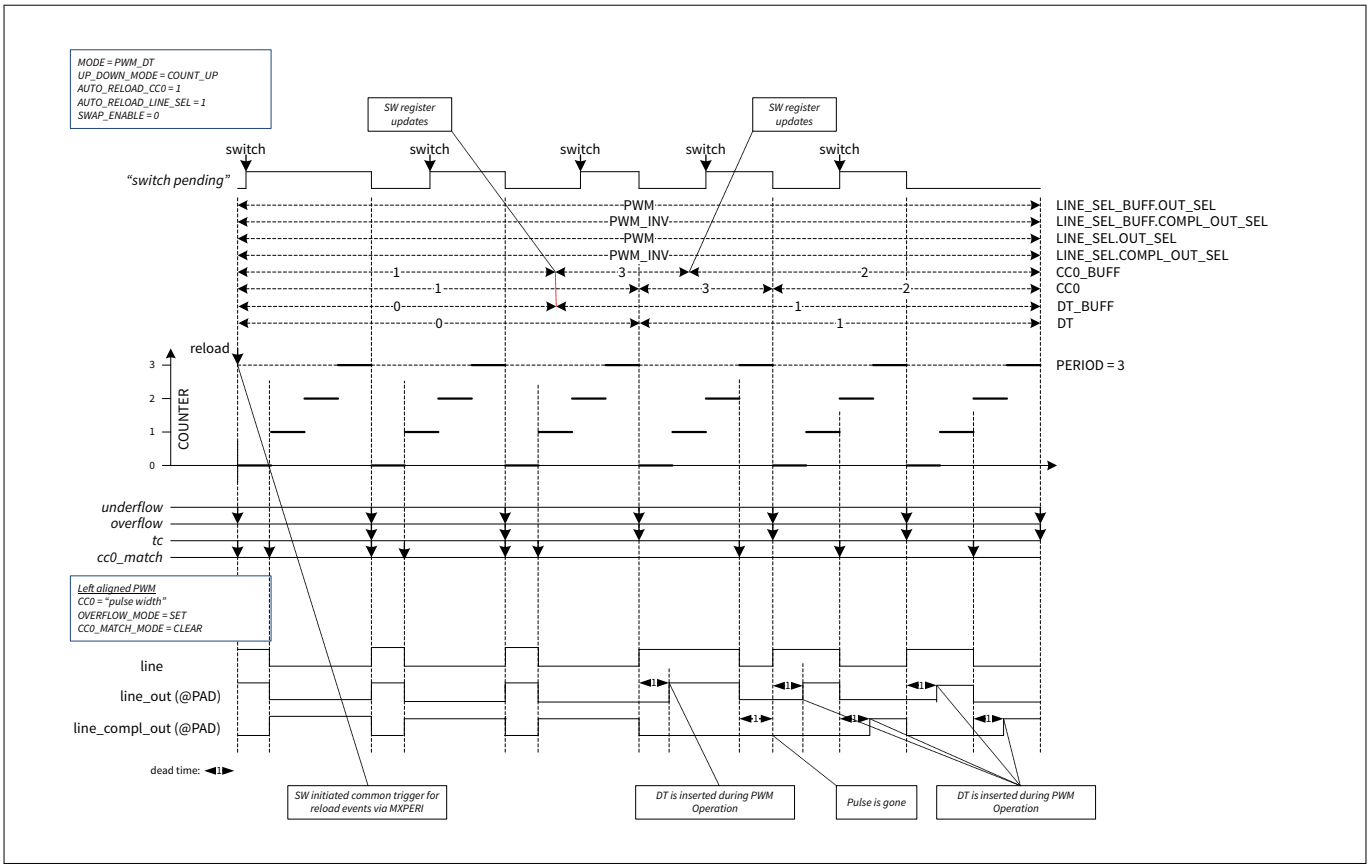


Figure 175 PWM_DT output case 2 scenario (SWAP_ENABLE=0 case)

Dead Time Generation

The switching transients are typically non-symmetric between on and off, which can create a short circuit if the on-time is shorter than the off-time.

In PWM_DT mode, the TCPWM module ensures a dead-time whenever the output selection changes to an active state.

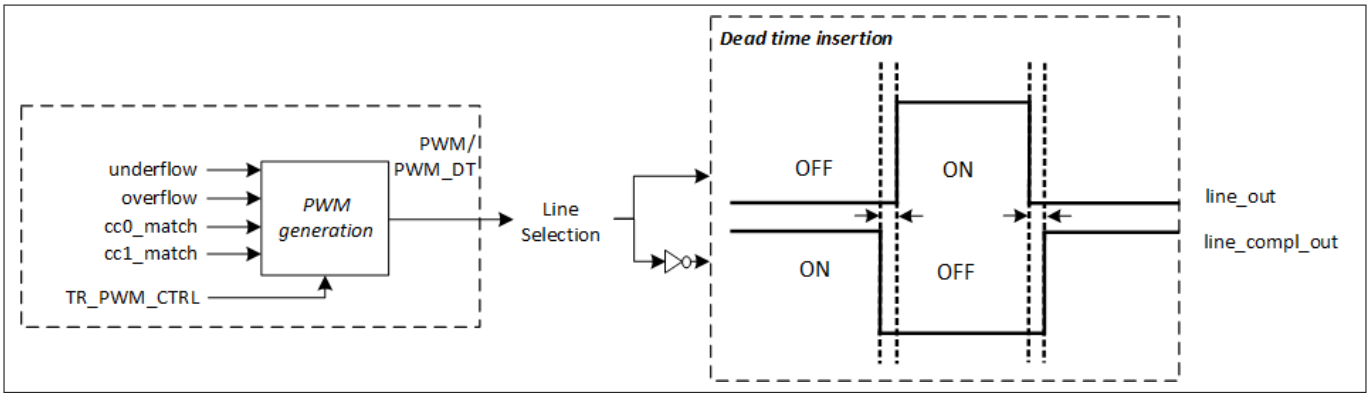


Figure 176 PWM Dead Time insertion

Specifically:

- A dead-time is generated when line_sel.out_sel transitions to 1 and a positive edge is detected on line_out
- A dead-time is generated when line_sel.compl_out_sel transitions to 1 and a positive edge is detected on line_compl_out

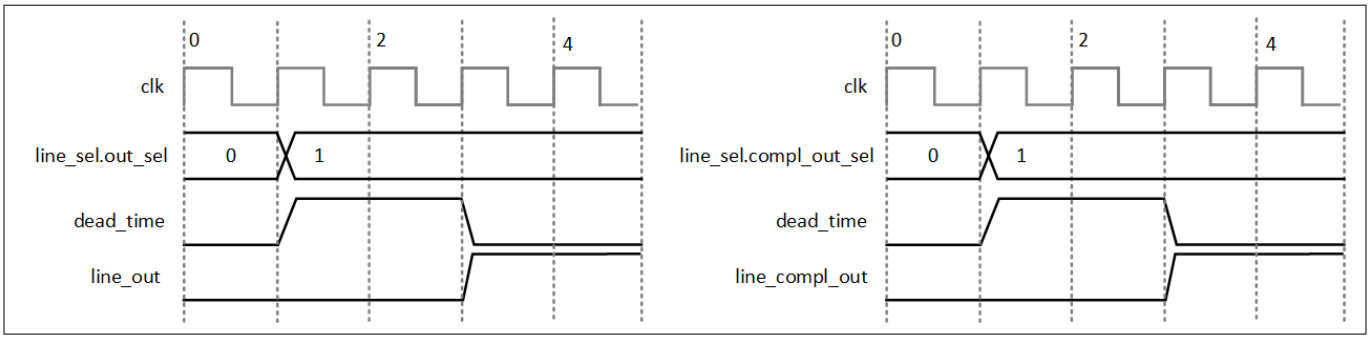


Figure 177 PWM dead time insertion with line_out and complementary

This mechanism mitigates the issue of non-symmetric switch-on and switch-off times, which can otherwise cause a short circuit.

25.3.5.1 How to configure counter for PWM with dead time mode

The steps to configure the counter for PWM with Dead Time mode of operation and the affected register bits are as follows:

- Disable the counter by writing '0' to the ENABLE bit of the CTRL register
- Select PWM with Dead Time mode by writing '101' to the MODE[26:24] field of the CTRL register
- Set the required dead time by writing to the DT_LINE_OUT_L [7:0] and DT_LINE_OUT_H [15:8] fields of the DT register
- Set the required 16-bit/32-bit period in the PERIOD register and the buffer period value in the PERIOD_BUFF register to switch values, if required
- Set the 16-bit/32-bit compare value in the CC0/1 register and buffer compare value in the CC0/1_BUFF register to switch values, if required
- Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the CTRL register to configure left-aligned, right-aligned, or center-aligned PWM
- Set the PWM_IMM_KILL, PWM_STOP_ON_KILL, PWM_SYNC_KILL, PWM_TC_SYNC_KILL and PWM_SYNC_KILL_DT fields of the CTRL register as required
- Set the TR_IN_SEL0 or TR_IN_SEL1 register to select the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
- Set the TR_IN_EDGE_SEL register to select the edge of the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
- LINE_OUT and LINE_COMPL_OUT can be controlled by the TR_PWM_CTRL register to set, reset, or invert upon cc0_match, cc1_match, overflow, and underflow conditions
- If required, set the interrupt upon TC or CC0/1 condition
- Enable the counter by writing '1' to the ENABLED bit of the CTRL register. A reload trigger must be provided through firmware (RELOAD bit in the PERI0_TR_CMD register) to start the counter if the hardware reload signal is not enabled

25.4 Dithering

The dithering feature is available on Group 1 and Group 2. The dithering feature is added in these counter groups for PWM and PWM_DT mode. Dithering can be turned ON through CTRL.DITHEREN for PWM and PWM_DT mode. Dithering can be turned ON for a period and duty cycle independently.

For period dithering, a pseudo random number generated from the Linear Feedback Shift Register (LFSR) is added or subtracted from the period value. A signal similar to PWM is created by comparing the counter value

with the period value. The generated signal has different frequency/noise characteristics than a regular PWM signal.

For duty cycle dithering, a pseudo random number generated from LFSR is added or subtracted from the CC0/CC1 value. The duty cycle of the PWM like signal is created by comparing the counter value with the CC0/CC1 value. The generated signal has a different random duty cycle characteristics than a regular PWM signal.

Dithering functionality is described as follows:

- The value in the PLFSR/DLFSR register is initialized to the non-0 value (seed value) at LFSR.PLFSR/ LFSR.DLFSR by software before dithering is enabled through CTRL.DITHEREN
- LFSR polynomial :
  - The value in PLFSR/DLFSR registers is change based on an 8-bit LFSR taps polynomial. The length of the LFSR is 8 bit that is fixed in design
  - When the counter value COUNTER equals PERIOD, a tc event is generated. for COUNT_UPDN1/ COUNT_UPDN2, tc event is generated where “PERIOD” is 2*PERIOD (one overflow and one underflow condition)
  - When the counter value COUNTER equals CC0, a cc0_match event is generated
  - When the counter value COUNTER equals CC1, a cc1_match event is generated
  - The software can control the magnitude of the LFSR value to be added or subtracted by programming LFSR.Limiter bits
  - The MSB bit of PLFSR/DLFSR will determine the polarity of the value to be added/subtracted from PERIOD/CC0/CC1
  - With signed number representation of PLFSR/DLFSR, the randomness added to period/duty is in the range of -2^(N-1) to 2^(N-1)-1 with N is the length of the PLFSR/SLFSR. In the design, N is set to 8
  - When software enables the dithering feature through CTRL.DITHEREN, it will not enable the dithering immediately on the line_out and line_comp_out. Dithering will only be introduced on the line_out and line_comp_out on the period match event
  - The dithering feature is disable through period match event and the end of the period of the sequence

***Note**: Note the following when using the dithering feature:*

- *There is a limiter option in the LSFR register that allows user to limit the magnitude of the value to be added to period/CC0/CC1*
- *The value of PERIOD, CC0, and CC1 shall be larger than the limiter value*
- *CC0 + MAX(Dither Value) < Period + Min (Dither value), which is controlled through the limiter*

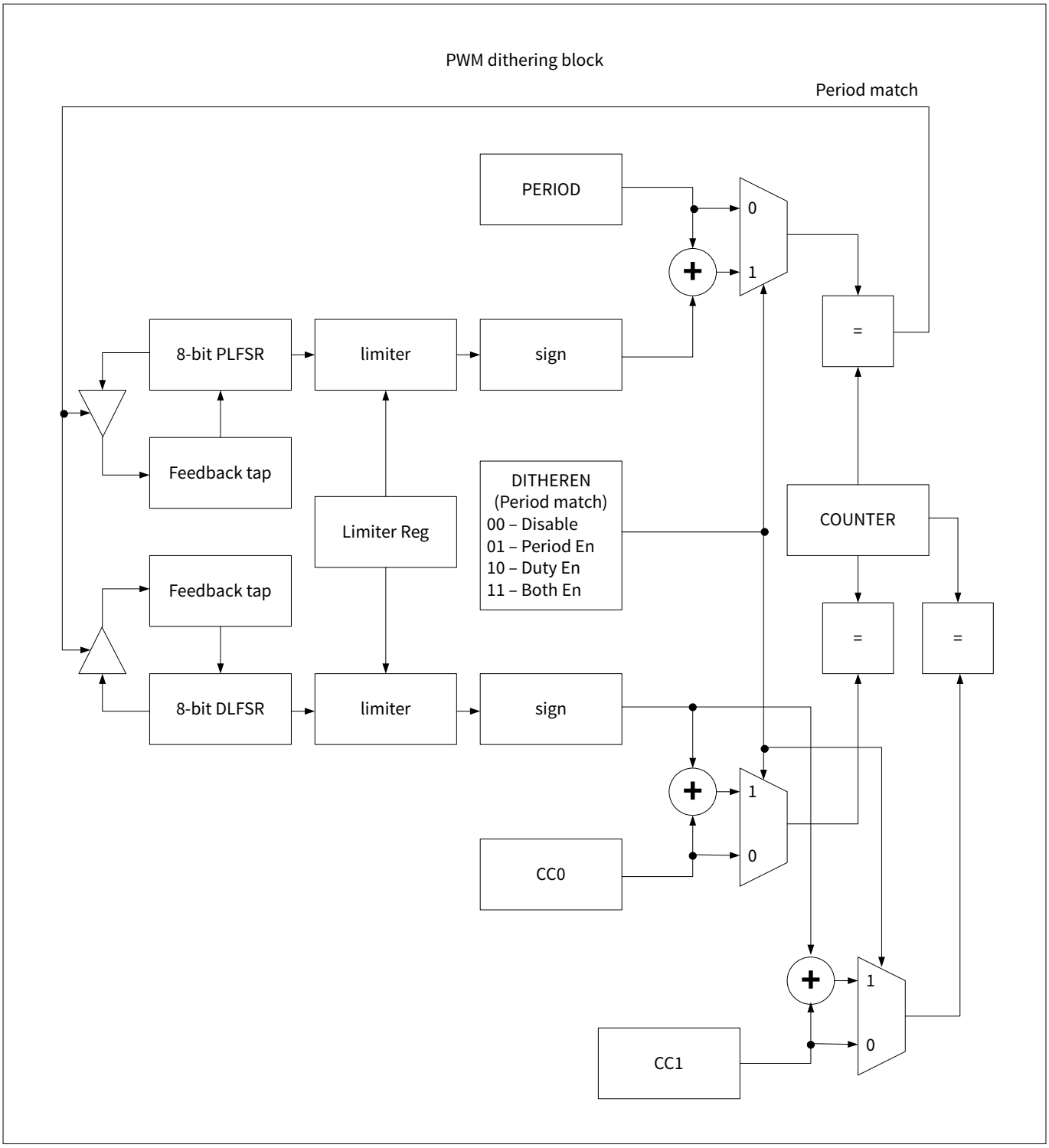


Figure 178 PWM dithering block

25.5 Motion interface (MOTIF)

PSC3M/P6 contains two MOTIF instances, each linked to a separate TCPWM counter group (Group 1 and Group 2).

25.5.1 Overview

The MOTIF module comprises three major sub units:

- Quadrature decoder unit
- Hall sensor control unit
- Multi-channel mode unit

The quadrature decoder unit is used for position control linked with a rotary incremental encoder. The hall sensor control unit is used for direct control of brushless DC motors. The multi-channel mode unit is used with the hall sensor mode to output the wanted motor control pattern but also can be used in a stand-alone manner to perform a simple multi-channel control of several control units.

25.5.1.1 Features

General features

- Quadrature decoder
  - Interface for position measurement
  - Interface for motor revolution measurement
  - Interface for velocity measurement
  - Interrupt sources for phase error, motor revolution, direction change and error on phase decoding
- Hall sensor
  - Simple build-in mode for brushless DC motor control
  - Shadow register for the multi-channel pattern
  - Complete synchronization with the PWM signals and the multi-channel pattern update
  - Interrupt sources for correct hall event detection, wrong hall event detection
- Multi-channel mode
  - Simple usage with hall sensor mode
  - Stand-alone multi-channel mode
  - Shadow register for the multi-channel pattern

25.5.1.2 Block diagram

The MOTIF module can operate in four different modes:

- Quadrature decoder
- Hall sensor
- Stand-alone multi-channel
- Quadrature mode with stand-alone multi-channel

To complete the control/measurement loop of all these modes, link the MOTIF with TCPWM counter(s). Each MOTIF module connects to a dedicated channel counter group. Each MOTIF module contains several inputs and outputs (including service requests) as shown in Figure 179 can be mapped to available module functions, depending on the configuration that was selected: quadrature decoder, hall sensor, or stand-alone multi-channel. The module also has two dedicated service request lines.

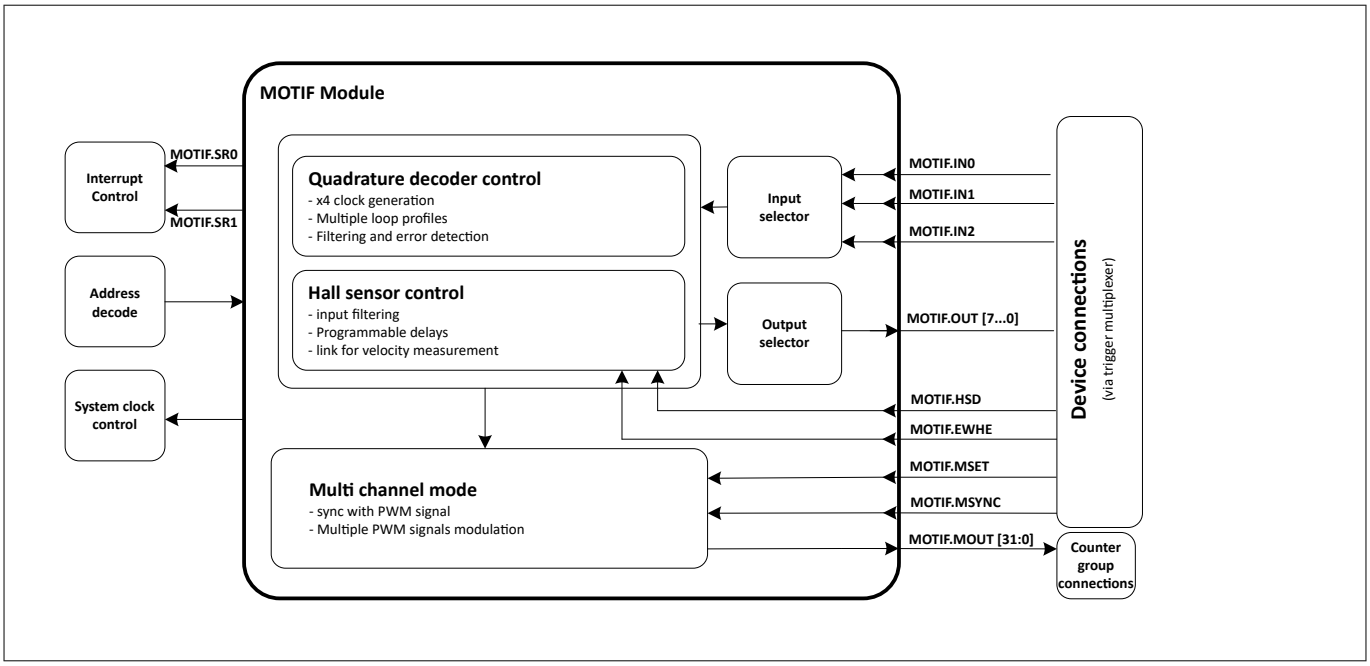


Figure 179 MOTIF block diagram

25.5.2 MOTIF functional description

The following sections describe the complete set of functions and usability of the MOTIF block. In each figure several registers may be depicted to indicate controllability or configurability. These registers follow the description given in Figure 180.

***Note**: The indexing in a register can be done via the non-capital y, x, or n.*

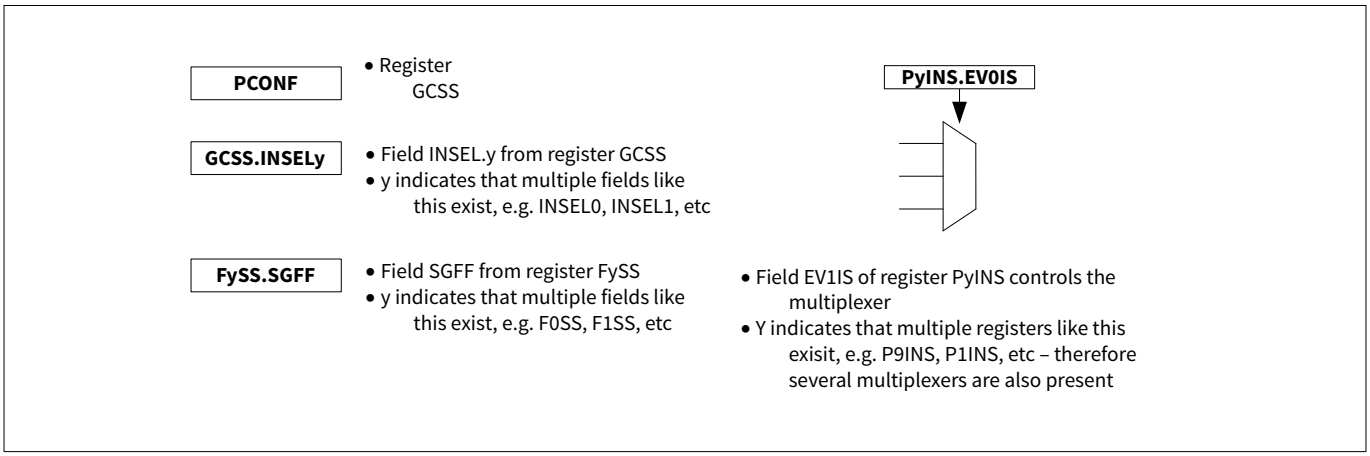


Figure 180 Register description in figures (example)

The MOTIF module contains a function selector unit that is used in parallel by both quadrature decoder and hall sensor control units. This block selects which input signals should be decoded for each control unit. On the output selector, one can select which functions need to be forwarded to the outputs, depending on the operation mode of the module. The MOTIF module also contains a subset of inputs that are only used in hall sensor/multi-channel mode. These inputs are connected to TCPWM counter(s) via trigger multiplexer and are used to control the delays between pattern sampling, multi-channel pattern update, and synchronization, and so on.

In the quadrature decoder mode, it is possible to define which of the signals is the leading phase and also the active level for each one. The quadrature decoder mode can also receive the clock and direction signals directly from an external source that can be used with simple encoders. The multi-channel offers a shadow register for the multi-channel pattern enabling this way an update on the fly of these parameters. A write access always addresses the shadow register while a read always returns the actual value of the multi-channel pattern.

25.5.2.1 MOTIF MMIO registers

A set of MOTIF specific MMIO registers support MOTIF functionality:

MOTIF Kernel Registers

- PCONF Global control register
- PSUS Suspend Configuration
- PRUNS MOTIF run bit set
- PRUN MOTIF run bit status
- MIDR Module Identification register

Hall Sensor Mode Registers

- HIST Hall Inputs Sample Trigger
- HMEC Hall Mode Extra Config
- HALP Hall Current and Expected patterns
- HALPS Hall Current and Expected shadow patterns
- HOSC Hall Sensor Output Config
- CLUT Hall Mode LUT Config
- SLUT Hall Mode LUT status

Multi-Channel Mode Registers

- MCM Multi-Channel Mode Pattern
- MCSM Multi-Channel Mode shadow Pattern
- MCMS Multi-Channel Mode Control set
- MCMC Multi-Channel Mode Control clear
- MCMF Multi-Channel Mode flag status
- MCPF Multi-Channel Pattern Fault
- MOSC Multi-Channel Output Config
- MCMEC Multi-Channel Extra Config
- MCSM1 Multi-Channel Mode shadow pattern LUT1
- MCSM2 Multi-Channel Mode shadow pattern LUT2
- MCSM3 Multi-Channel Mode shadow pattern LUT3
- MCSM4 Multi-Channel Mode shadow pattern LUT4
- MCSM5 Multi-Channel Mode shadow pattern LUT5

Quadrature Decoder Mode Registers

- QDC Quadrature Decoder Configuration
- QOSC Quadrature Output Config

Interrupt Registers

- PFLG MOTIF interrupt status
- PFLGE MOTIF interrupt enable
- SPFLG Interrupt set register
- RPFLG Interrupt clear register
- PDBG MOTIF Debug Register
- PLP0S MOTIF Low Pass 0 Status
- PLP1S MOTIF Low Pass 1 Status
- PLP2S MOTIF Low Pass 2 Status

25.5.2.2 MOTIF Pin mapping

The MOTIF global inputs and outputs are mapped as shown in the table. The formula in the table uses “y” and “x”. The mapping is defined as:

- “y”: Motif Group Index
- “x”: Motif index

In general Motif Group Index ranges from 0 to 7, and Motif index ranges from 0 to 31. For the PSOC™ Control C3 (M/P)6, the mapping assigns Motif Group Index values of 1 and 2, and Motif index equals to 0.

Table 218 MOTIF pin description

Pin (global)	I/O	Map to	Hall Sensor mode	Quadrature mode	Multi-channel Mode (stand-alone)
tr_motif_in[(y*256+0) + x*8]	I	MOTIF_GRP[y].MOTIF[x].IN0	Hall Input 1	Encoder Phase A or clock	Not used
tr_motif_in[(y*256+1) + x*8]	I	MOTIF_GRP[y].MOTIF[x].IN1	Hall Input 2	Encoder Phase B or clock	Not used
tr_motif_in[(y*256+2) + x*8]	I	MOTIF_GRP[y].MOTIF[x].IN2	Hall Input 3	Index/Zero marker	Not used
tr_motif_in[(y*256+3) + x*8]	I	MOTIF_GRP[y].MOTIF[x].HSD	Hall pattern sample delay	Not used	Not used
tr_motif_in[(y*256+4) + x*8]	I	MOTIF_GRP[y].MOTIF[x].EWHE	Fault event input	Not used	Fault event input
tr_motif_in[(y*256+5) + x*8]	I	MOTIF_GRP[y].MOTIF[x].MSET	Multi-channel next pattern update	Not used	Multi-channel next pattern update
tr_motif_in[(y*256+6) + x*8]	I	MOTIF_GRP[y].MOTIF[x].MSYNC	Multi-channel pattern update synchronization	Not used	Multi-channel pattern update synchronization

(table continues...)

Table 218 (continued) MOTIF pin description

Pin (global)	I/O	Map to	Hall Sensor mode	Quadrature mode	Multi-channel Mode (stand-alone)
tr_motif_in[(y*256+7) + x*8]	I	MOTIF_GRP[y].MOTIF[x].SPARE	Not used	Not used	Not used
tr_motif_out[(y*256+ {6:0}) + x*8]	O	MOTIF_GRP[y].MOTIF[x].OUT[6:0]	Programmable outputs	Programmable outputs	Programmable outputs
tr_motif_out[(y*256+7) + x*8]	O	MOTIF_GRP[y].MOTIF[x].OUT[7]	Not used	Not used	Not used
motif_interrupt [(y*64+0) + x*2	O	MOTIF_GRP[y].MOTIF[x].SR0	Service request line 0	Service request line 0	Service request line 0
motif_interrupt [(y*64+1) + x*2	O	MOTIF_GRP[y].MOTIF[x].SR1	Service request line 1	Service request line 1	Service request line 1

25.5.2.3 MOTIF clocks

The TCPWM receives a single clock “clk_peri”. Furthermore, it receives a system clock enable signal “clock_sys_en” to generate internal “clk_root”, if all counters and MOTIF modules are disabled via CTRL.ENABLE, PCONF.ENABLE then “clk_root” still gated.

Inside each MOTIF block, two clocks are derived from “clk_root”, these are “clk_ahb_motif” and “clk_motif”. “clk_ahb_motif” is used for MOTIF MMIO registers whereas “clk_motif” is used for MOTIF operational logic and input event synchronization logic.

MOTIF block receives a clock enable signal “clock_motif_en[i]” which can be used to gate all clocks inside a particular MOTIF instance.

The mapping of clock_motif_en[] to motif[] is shown below:

clock_motif_en [0] → group[0] motif [0]

clock_motif_en [1] → group[0] motif [1]

…

clock_motif_en [30] → group[0] motif [30]

clock_motif_en [31] → group[0] motif [31]

…

clock_motif_en [254] → group[7] motif [30]

clock_motif_en [255] → group[7] motif [31]

When MOTIF module is disabled via PCONF.ENABLE=0 register the “clk_motif” is gated whereas “clk_ahb_motif” is active, all non-retention flip flops inside MMIO register and Operational logic are held in reset whereas the retention register can still be accessed by SW.

When debug mode is active, “clk_ahb_motif” is still active, whereas “clk_motif” is gated. Refer section MOTIF debug behavior.

Event generation is performed on clk_motif.

All status or output change can only happen on clk_motif.

The TCPWM receives a single clock “clk_peri”. Furthermore, it receives a system clock enable signal “clock_sys_en” and a MOTIF clock enable signal “clock_motif_en[i]” for each MOTIF block (up to 256).

The “clock_sys_en” signal is used to derive clock “clk_root”.

The “clock_ahb_en” signal is generated by the AHB5 interface and used to derive “clk_ahb”.

The “clock_motif_en[i]” signal is used to derive MOTIF clock “clk_motif[i]”. “clock_motif_en[i]” is provided by the PERI/PCLK component. Note that the PERI/PCLK component allows for multiple clock gate enable signals to be connected to the same divider. This allows for synchronous operation of multiple MOTIF blocks.

MOTIF MMIO registers are accessible by both the hardware (“clk_motif[i]” clock domain) and the AHB5 interface (“clk_ahb” clock domain). These registers use clock “clk_ahb_motif[i]”, which is a gated version of the input clock “clk_peri”, using clock enable signals “clk_ahb_en” and “clock_motif_en[i]”.

***Note**: In application case, MOTIF is always running at the higher frequency.*

25.5.2.4 MOTIF reset

There are two global resets inputs to TCPWM, rst_sys_act_n and rst_sys_dpslp_n. The rst_sys_act_n is ANDed with PCONF.ENABLE register to produce the internal MOTIF reset signal called “rst_motif_act_n”.

MOTIF operational logic and non-retention MMIO registers can be reset using rst_motif_act_n. Synchronizer for tr_motif_in[] is also reset via “rst_motif_act_n” signal.

All retention MMIO MOTIF registers will be reset by rst_sys_dpslp_n.

25.5.2.5 Input trigger mapping

The mapping of input trigger “tr_motif_in[]” to motif[] is shown in the table Table 218

In general MOTIF group Index ranges from 0 to 7, and MOTIF index ranges from 0 to 31. For the PSOC™ control C3 (M/P)6, the mapping shown in table Table 219assigns MOTIF group Index values to 1 and 2, and MOTIF index equals to 0.

Table 219 Input trigger mapping

Input Trigger	MOTIF Instance
tr_motif_in[255:263]	MOTIF_GRP[1].MOTIF[0]
tr_motif_in[512:519]	MOTIF_GRP[2].MOTIF[0]
Others	Unused

25.5.2.6 Output trigger mapping

The mapping of output trigger “tr_motif_out[]” to motif[] is shown in the table Table 218

In PSOC™ control C3 (M/P)6, the mapping shown in table Table 220 assigns MOTIF group Index values to 1 and 2, and MOTIF index equals to 0.

Table 220 Output trigger mapping

Output Trigger	MOTIF Instance
tr_motif_out[255:263]	MOTIF_GRP[1].MOTIF[0]
tr_motif_out[512:519]	MOTIF_GRP[2].MOTIF[0]

(table continues...)

Table 220 (continued) Output trigger mapping

Output Trigger	MOTIF Instance
Others	Unused

25.5.2.7 Function selector connection scheme

The Function selector maps the input function signals to the selected operating mode, whether Quadrature or Hall Sensor Mode.

It is also possible to perform a low pass filtering on the MI0, MI1 and MI2 inputs, the field PCONF.LPC controls the low pass filters cut frequency. The low pass filter is counter based with a configurable depth, power of 2. The counter increments from zero to depth when an edge is detected at the input. The input has to be the same while the counter is counting up to the desired depth for the output to be updated. If the input changes before counter reaches depth, the counter is reset and the output filter is not changed. The low pass filter is active when MOTIF is stopped, this enables Software to trigger a Hall sample if required. If MOTIF is disabled, the Low Pass Filter logic is in reset. The Low pass filter has a valid output which gets asserted after the first correct sample. The low pass filter runs on clock_motif and is reset with rst_motif_act_n signal.

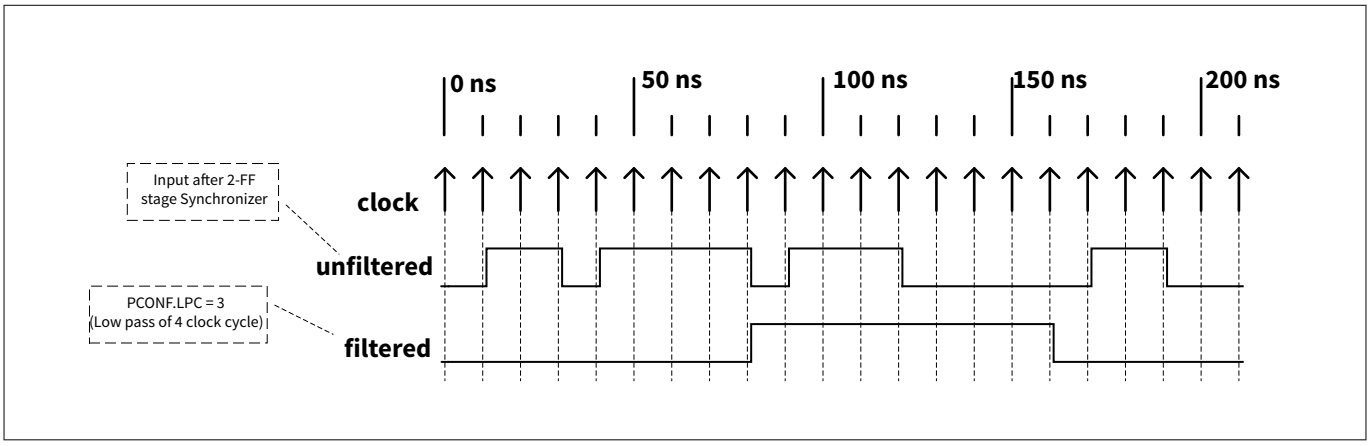


Figure 181 MOTIF Low pass filter

The Hall Sensor Mode is the default function, PCONF.FSEL = 00B. In this mode, the Function selector maps the MI0 to the Hall Input 1, the MI1 to the Hall input 2 and the MI2 to the Hall input 3. When the Quadrature Decoder mode is selected, PCONF.FSEL = 01B, MI0, MI1, and MI2 can be mapped to the following functions: Phase A or Clock, Phase B or Direction or Index signal coming from the rotary motor encoder - this is programmable on via the QDC register. It is also possible to enable in parallel the Quadrature Decoder and stand-alone Multi-Channel mode, by setting PCONF.FSEL = 11B (Hall Sensor Mode and Quadrature Mode can never be used in parallel). The Output Selector allows a very flexible mapping of the events generated by the three available modes, into a specific module output pin.

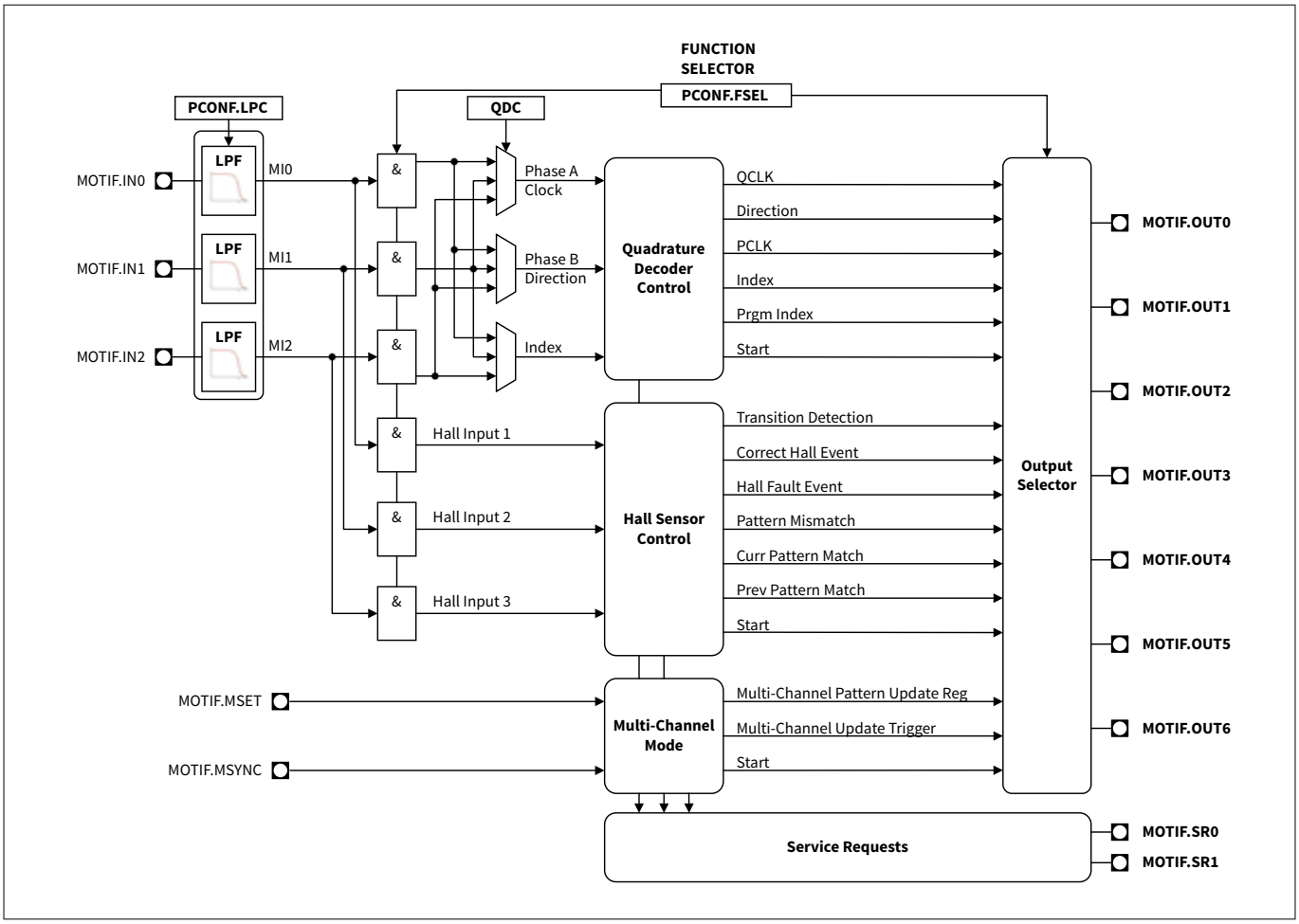


Figure 182 MOTIF connection scheme

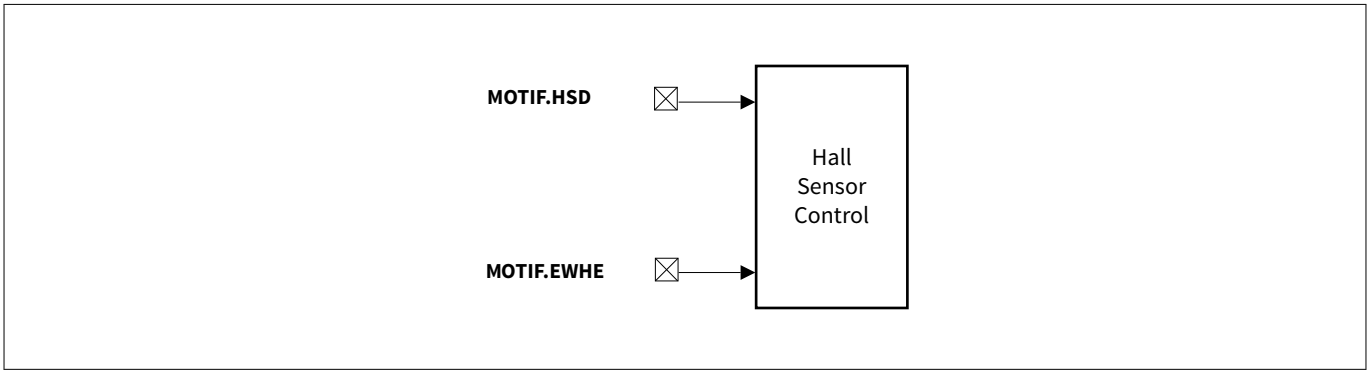


Figure 183 MOTIF additional connections

25.5.3 Hall sensor control

25.5.3.1 Hall Sensor standard operation

The Hall sensor mode is divided into three major loops:

- Detection of an update in the hall inputs

- Delay between the detection
- Sampling of the hall inputs for comparison against the expected hall pattern and the update of the multi-channel pattern

The hall inputs are directly connected to an edge detection circuitry and with any modification in any of these three inputs, an event is generated-transition detection, see Figure 184. This event can be connected to one TCPWM counter via a trigger multiplexer that is controlling the delay between the edge detection and the next step on the hall sensor mode, the sampling of the hall inputs. The signal used to trigger the sample of the Hall inputs can be active at the rising or falling edge (PCONF.SPES). The application software can also issue/request a sample of the hall inputs, by writing 1_B into the HIST.HIS field (normally used for start-up).

When the sampling trigger is sensed, the hall inputs are stored and compared against the current pattern, HALP.HCP, the expected hall pattern, HALP.HEP, and the previous pattern HALP.HPP (the previous pattern comparison can be disabled).

The edge detection circuit generates an enable signal for sampling the hall inputs, PIFHSP, and the sample logic generates a pulse every time that new values are captured, PIFHRDY, that is used inside the pattern compares logic.

The external sample trigger can also be bypass by setting the HMEC.HDBP bit field to 1_B. This will trigger an immediate sample of the hall inputs after the filtering stage if a hall transition is detected. The hall inputs are also sampled as soon as the MOTIF start bit sees a rising edge. The software has to make sure to start MOTIF after LPF output is valid.

When the sampled value matches the expected hall pattern, an event/pulse is generated - the correct hall event - to indicate that the sampled pattern matched the expected one. At the same time, the values programmed into the shadow registers are loaded. The HALP.HCP[LSB] is linked to the hall input 1, and the HALP.HCP[MSB] is linked to the hall input 3 (the same is applicable for the HALP.HEP register).

The update of the fields HALP.HCP, HALP.HEP, and HALP.HPP can come from software by writing 1_B to MCMS.STHR.

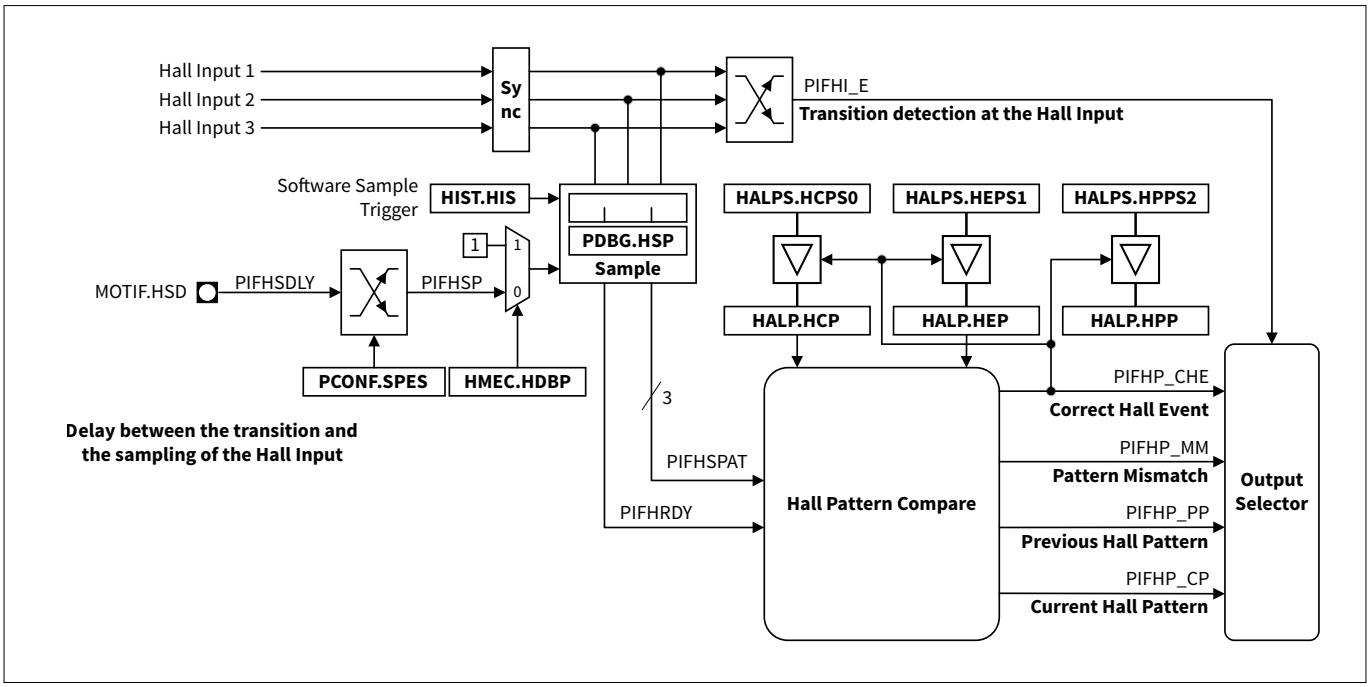


Figure 184 Hall sensor control diagram

When the sampled value matches the current hall pattern, a line glitch is deemed to have occurred and no action is taken. Besides the comparison against the current hall pattern it is also possible to use a third comparison rule - the previous hall pattern. This rule can be enabled by setting the HMEC.HPPE bit field to 1_B. With this comparison, a wrong hall event is not generated if the sampled pattern matches the value

programmed in the HALP.HPP. This is important when the motor is in a stand-still position, which may impose that with too less torque the previous hall pattern is sensed at the MOTIF inputs.

When the sampled value does not match any of the values (the current, expected, and previous pattern), a major error is deemed to have occurred and a wrong hall event signal is generated. Every time that a sampled pattern leads to a wrong hall event or when it matches the current pattern or previous pattern (this one can be enabled or disabled) a pattern mismatch event is generated and can be forward to one of the module output pins (via the output selector), see Figure 185

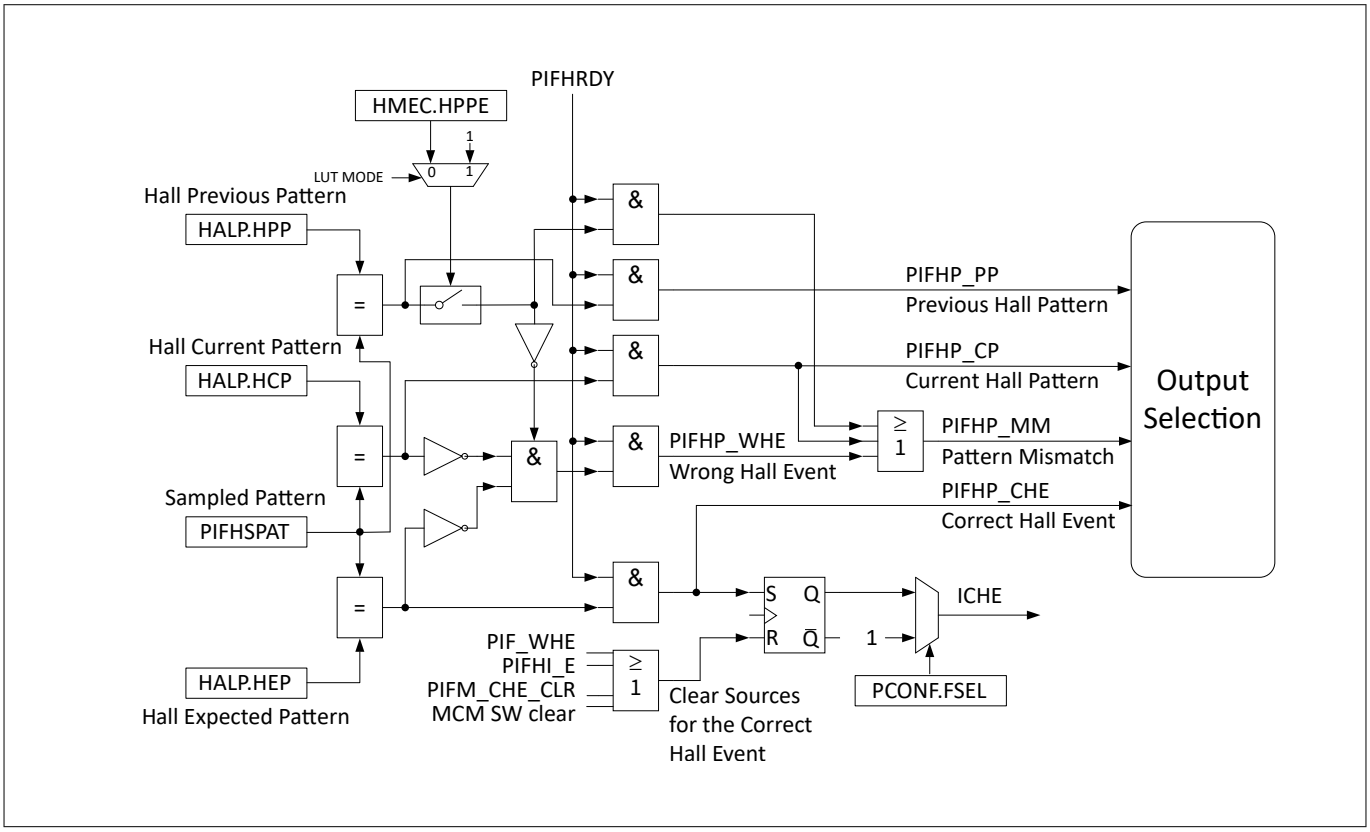


Figure 185 Hall sensor compare logic

A fault hall event can also be decoded by the MOTIF module. This fault hall event is a selectable combination of the wrong hall event coming from the hall compare logic and a programmable module input pin, see Figure 186.

This fault hall event can be forwarded to one of the MOTIFx.OUTy module pins via the output selector (for example, to perform a forced stop operation in the PWM units) and also can be used to clear the multi-channel pattern, see Figure 187.

When a fault event is detected in the hall sensor mode, the hardware shall stop MOTIF as a safety mechanism. This safety mechanism feature is only applicable in LUT Mode. The software needs to act and reconfigure and start MOTIF again after a fault event is detected.

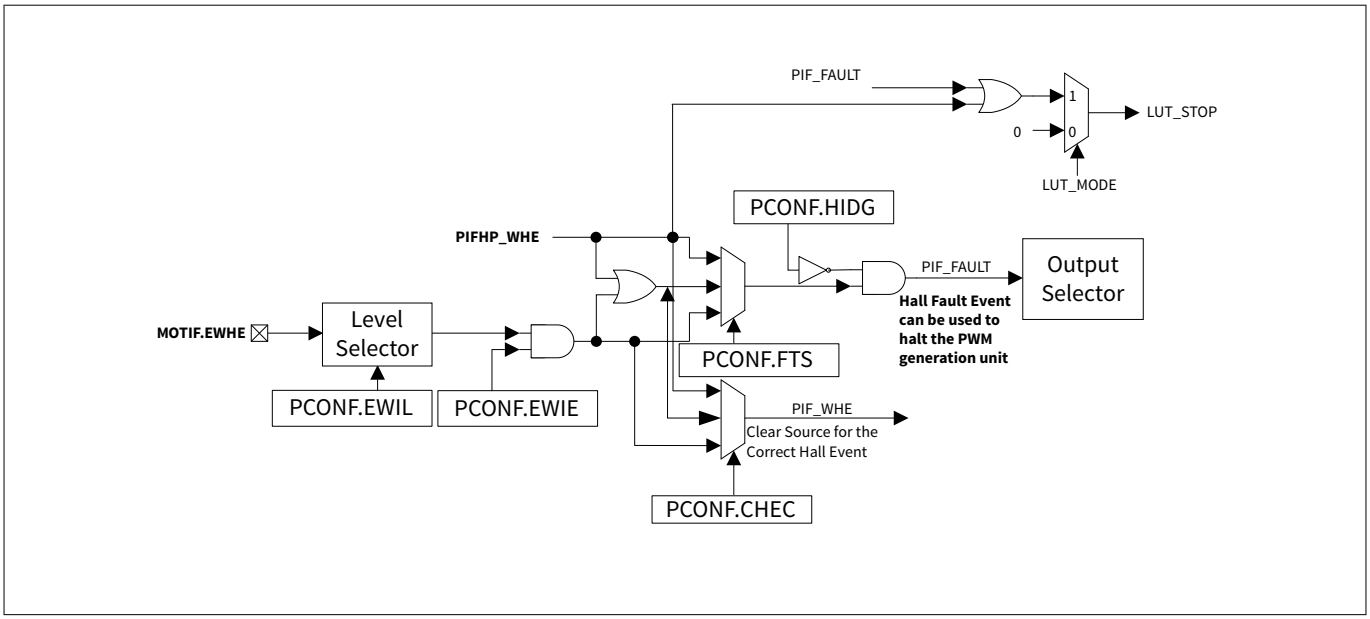


Figure 186 Hall fault event/Idle logic

After the correct hall event is detected, a delay can be generated between this detection and the update of the multi-channel pattern. In the Figure 187, it is demonstrated the control logic of the multi-channel mode.

The delay for the update of the multi-channel pattern can be controlled directly by a TCPWM counter. The trigger that indicates that the delay is finished, is mapped to input MOTIFx.MSET. One can also select the active edge for the trigger via PCONF.MSES. This delay insertion can be bypass by setting the MCMEC.MEBP bitfield to 1_B.

The PCONF.MCUE field selects the source that enables an update of the multi-channel pattern. When set to 1_B , the multi-channel pattern can only be updated after the software has written a 1_B into the MCMS.MNPS field.

After the update delay (if enabled), the multi-channel pattern still needs to be synchronized with the PWM signal. For this, the input MOTIFx.MSYNC is used. One can also select the active edge for the trigger via PCONF.MSYES. When the configured edge is detected in this signal, then the new multi pattern is applied at the MOTIFx.MOUT[31:0] outputs, with the MCM.MCMP[31] linked to the MOTIFx.MOUT[31] and MCM.MCMP[0] to MOTIFx.MOUT[0]. This synchronization stage can also be bypass by setting the MCMEC.MSBP to 1_B.

A multi-channel pattern update request trigger is connected to the MCMF.MSS register field and it can be forwarded to a module pin via the output selector. This register field (MSS) is enabling the multi-channel pattern update (that is done upon receiving the sync signal, PIFMSYNC). The multi-channel pattern is used with a TCWPM counter to perform a synchronous update of the multi-channel PWM output. The update of the MCM.MCMP can be triggered by the software by writing 1_B to MCMS.STMR.

It is also possible to use the hall fault event to clear/load the multi-channel pattern with a pre-configured value - MCPF.MCFV. This mechanism can be used whenever a wrong operation is detected and a specific safe pattern needs to be applied to the PWM timers that are controlling the output switches.

The 32-bit multi-channel pattern outputs can affect the PWM output of the associated counter in two different ways:

- AND gating the PWM - forcing the PWM to passive
- OR gating the PWM - forcing the PWM to active

Each counter will then receive four signals driven by the multi-channel pattern outputs to perform modulation in line_out and line_comp_out. One signal dictates if an OR or an AND is needed, and another signal is used as the second input for the function.

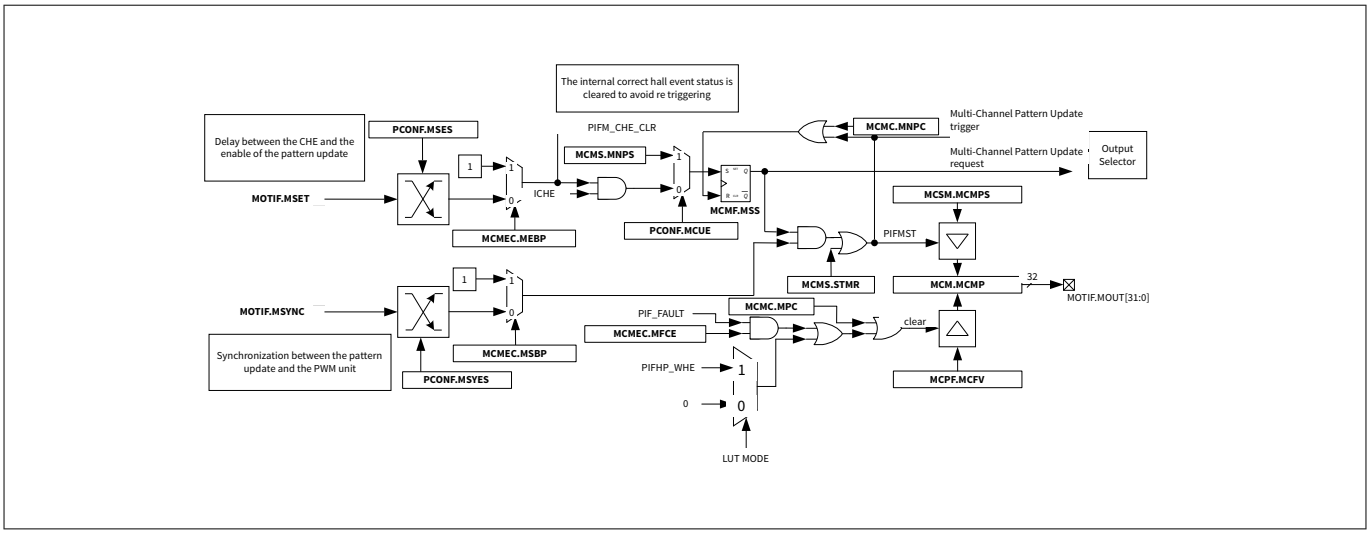


Figure 187 Multi-channel mode diagram

Figure 188 depicts an example of a state diagram using the Hall Sensor Mode for a Brushless DC Motor Control Application. Notice that in this example several states can be bypassed/not used (for example, is possible to avoid the PWM Sync stage by setting the MCMEC.MSBP bitfield to 1B), therefore one should consider that this example contains a super set of the Hall Sensor Mode Control functions.

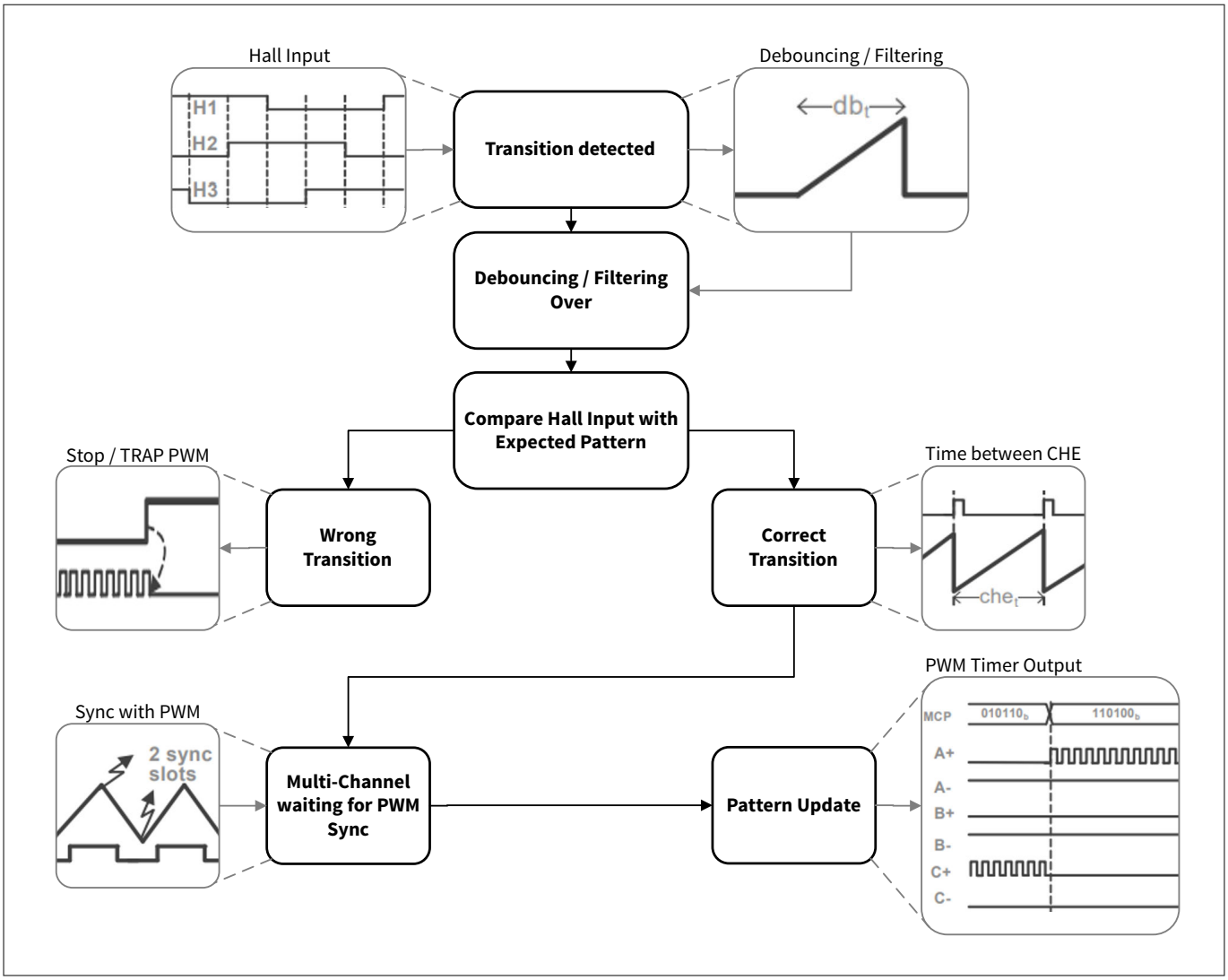


Figure 188 State diagram example for BLDC

Figure 189 shows all the previous described steps in the hall sensor mode description within the timer domain: every time that a transition on a hall input is detected, the signal - transition detection - is asserted. This signal is used to start the timing delay between the transition detection and the sampling of the hall inputs.

In this scenario, the status output (TR_OUT0 in Figure 189) of a TCPWM counter was used to control the timing 1 (delay between the transition and the sampling of the hall inputs). With the rising edge of the TR_OUT0 signal, the hall inputs are sampled and if they match the expected pattern, the signal - the correct hall event - is asserted.

A period match event output (TR_OUT1 signal) mapped to the same TCPWM counter is used to control the delay between a correct hall event and the update of the multi-channel pattern. This service request is asserted when a period match is detected.

Another TCPWM counter is used to measure the time between each correct hall event. This timer cell is represented by the timing 2 in the Figure 189 (the CR symbolizes the capture register of the counter block).

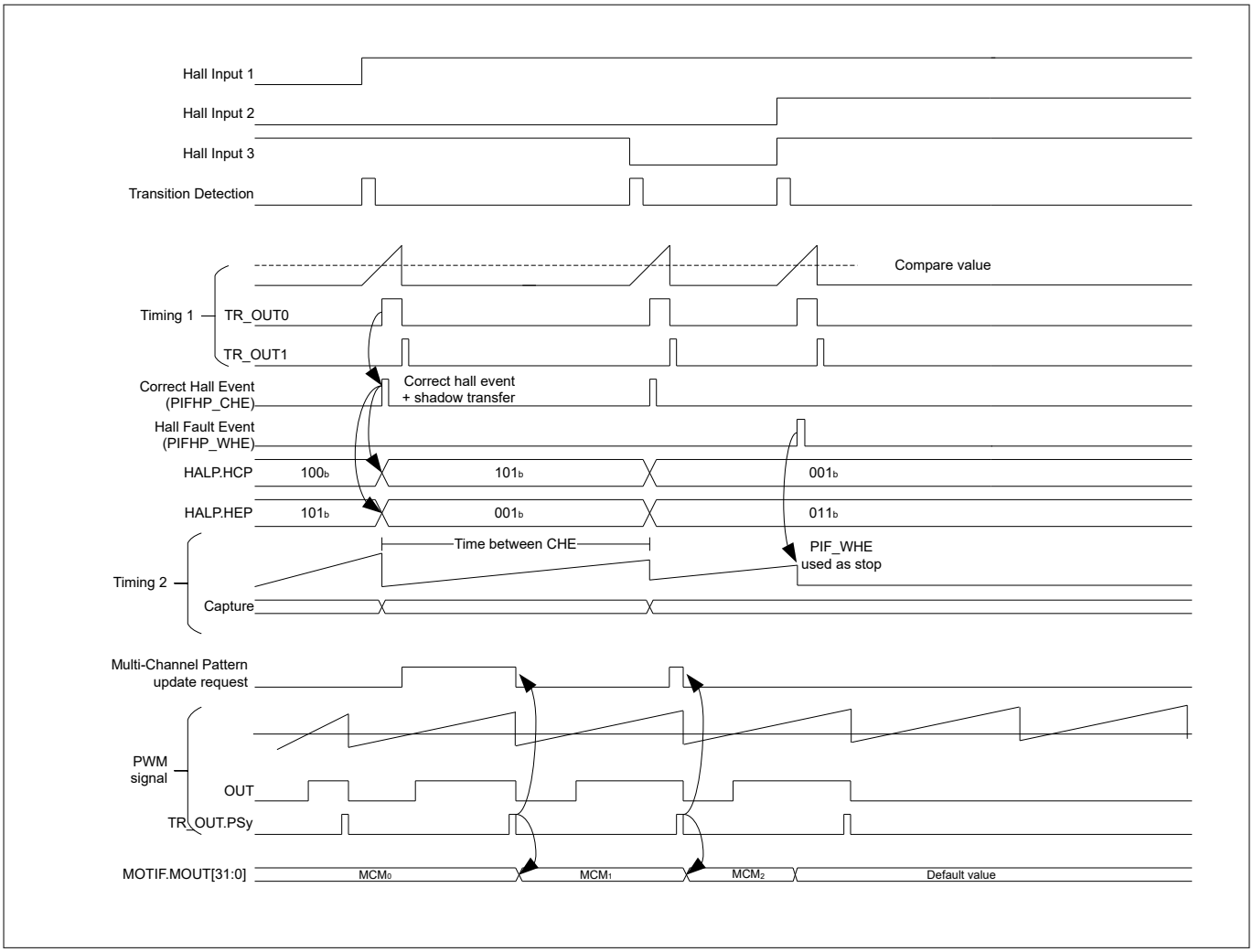


Figure 189 Hall sensor timing diagram

25.5.3.2 Hall Sensor operation with LUT

The hall sensor can operate in look-up table (LUT) mode. This mode aims to reduce CPU load for typical motor control phase commutation in a semi-autonomous way. The LUT mode is a fully hardware-based feedback control loop assisted by software. As long as the motor runs in the same direction, the hardware takes full control on multi-channel pattern output, MCM.MCMP. The Output pattern will be taken from a pre-loaded LUT by software. When the motor changes direction, MOTIF is stopped and the software needs to repopulate the LUT content and start MOTIF again.

Each MOTIF module contains two LUTs. Hall LUT (HLUT, also called “input LUT”) and Modulation LUT (MLUT, also called “output LUT). HLUT contains a hall input pattern, which gets loaded to HALP registers. MLUT contains an output PWM modulation pattern which gets loaded to the MCM.MCMP register.

Each MOTIF has a LUT control logic, which is used to assist the LUT operation. The following registers are associated with this mode:

- LUT control configuration

CLUT.LUTEN -> enable/disable LUT mode

- LUT status and debug register

SLUT.PTR -> internal pointer value status

- MLUT (output LUT)

MCSM, MCSM1, MCSM2, MCSM3, MCSM4, MCSM5

- HLUT (input LUT)

HALPS.HCPS0[2:0], HALPS.HEPS1[2:0], HALPS.HPPS2[2:0], HALPS.HLUT3[2:0], HALPS.HLUT4[2:0], HALPS.HLUT5[2:0]

The following figure shows a basic diagram of the main logic involved during LUT operation mode. If LUT mode is enabled, LUT control logic decides which value from HLUT and MLUT gets loaded to HALP and MCM registers respectively. If LUT mode is disabled the loading of HALP and MCM registers is done from a fixed register location as described in Hall Sensor standard operation section.

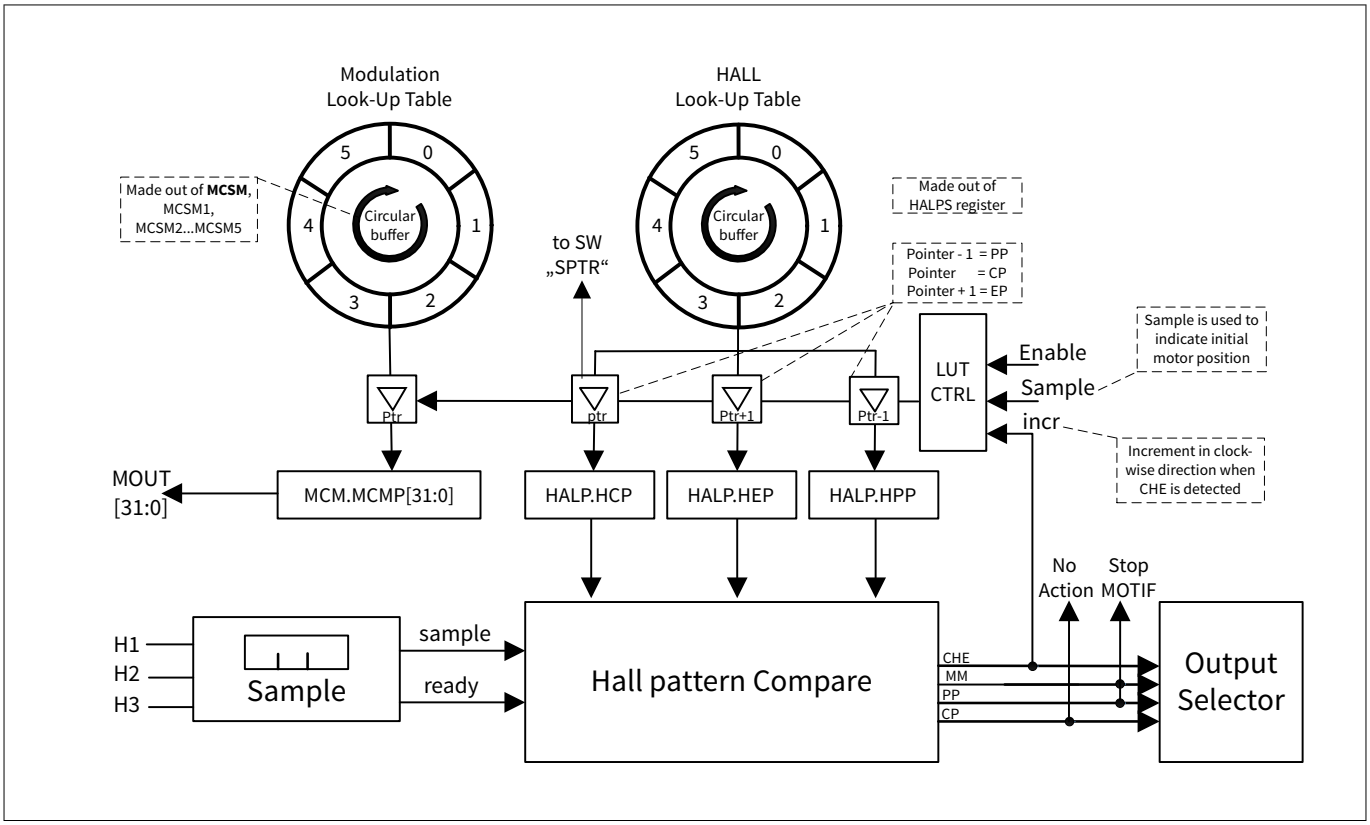


Figure 190 Hall sensor mode with LUT support

The LUT operation is a hardware-based loop, however it requires software assistance in some conditions as described below.

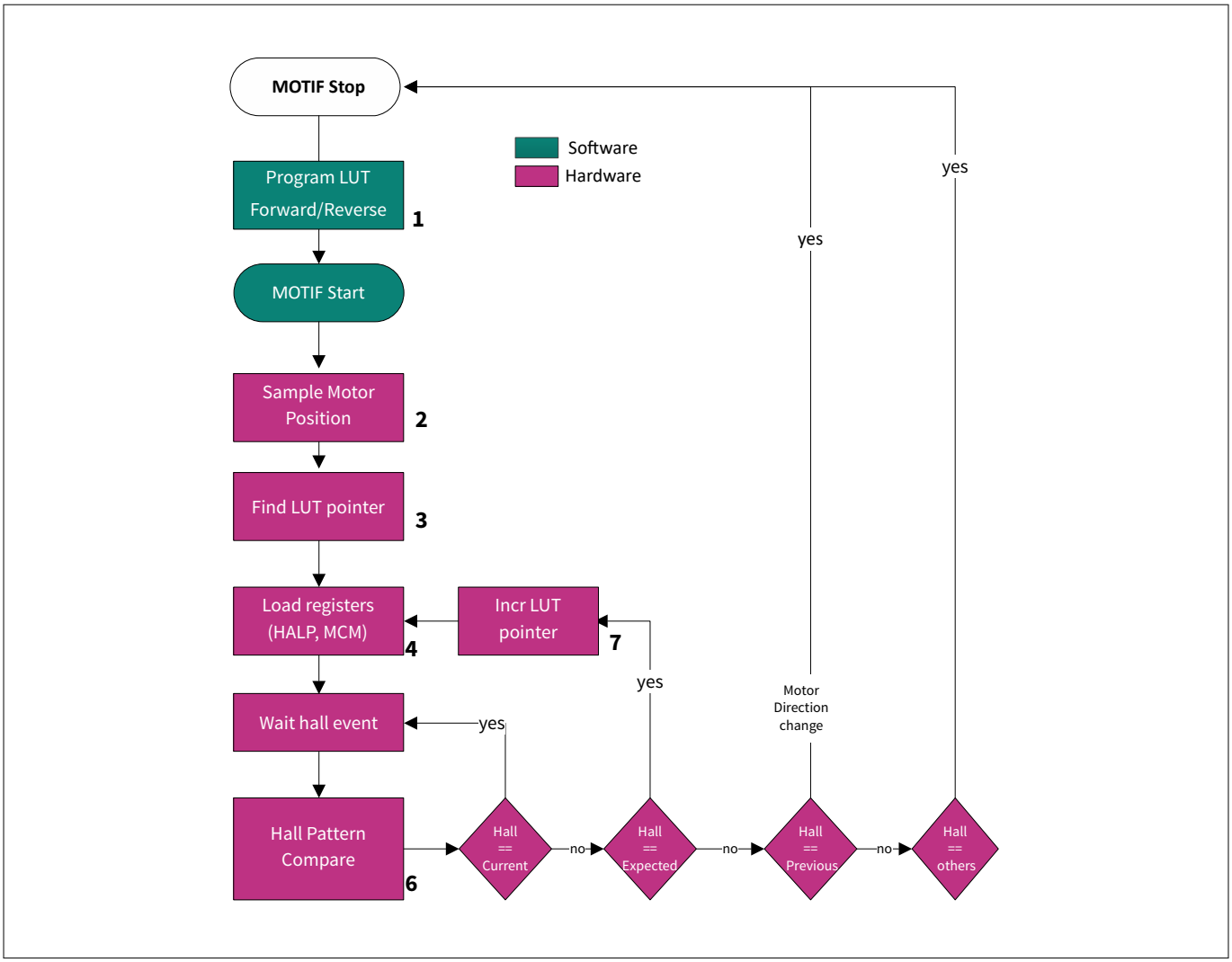


Figure 191 LUT mode - Sequence of events

STEP 1: Program LUT

Software writes HLUT and MLUT when MOTIF is stopped. Writing into both LUT while MOTIF is started might lead to incorrect behavior. The LUT is loaded with a forward or reverse pattern depending on motor desired direction. The pattern is stored in LUT independently from motor position. The hardware is responsible to find out the current motor position and to load the right hall and modulation pattern.

HLUT and MLUT share the same hardware pointer for data extraction, therefore software needs to take care to keep them aligned with respect to LUT address. A wrong programming might lead to incorrect behavior. For example, if the current motor position matches HLUT(x), the output modulation needs to be located in MLUT(x), the same index. Software must take care of keeping them aligned.

The LUT works as a circular buffer, going only in a clockwise direction over 6 addresses, 0 to 5. HLUT cannot have repeated values. All six values are expected to be unique. If the current pattern is stored in HLUT(x) then the expected pattern must be located in HLUT(x+1) and the previous pattern must be located in HLUT(x-1). The same applies for MLUT.

STEP 2: Motor position sample

The sample of HALL inputs is done automatically by hardware when the MOTIF module is started and HALL mode is enabled, this is done to obtain current motor position. Enabling LUT mode after MOTIF is started might lead to incorrect behavior. It shall be prevented by software. The motor position sample is available 2 clock cycles after the MOTIF module is started and with LUT mode enabled.

STEP 3: Find LUT pointer

After the hall sample, the hardware compares the current motor position PDBG.HSP (available from STEP 2) against all six HLUT values in a parallel way. The LUT pointer corresponds to the address which content matches the current motor position. Only one compare out of six is expected to be valid. This is because all HLUT values are expected to be unique.

If all six HALL LUT values differ from motor current position, a pattern mismatch event is generated. Hardware will stop MOTIF. The pattern mismatch event can be used by software to load a modulation fault safe pattern. An interrupt can be generated. The LUT address pointer “PTR”, which corresponds to the current pattern, is updated and can be accessible via software through the SLUT.PTR register. Valid values are 0, 1, 2, 3, 4, and 5. Invalid values 6, 7. This operation is executed only once at the starting phase after MOTIF is enabled (transit from stop to start). The LUT pointer will not be incremented at this stage as the purpose is to search and align the index of the LUT.

STEP 4: Load Registers (HALP, MCM).

Automatic shadow transfer takes place to load HALP and MCM registers with LUT pointer index. After the correct hall event is detected, a delay can be generated between this detection and the update of the multi-channel pattern. In the Hall Sensor standard operation, it is demonstrated the control logic of the multi-channel mode.

The delay for the update of the multi-channel pattern can be controlled directly by a TCPWM counter. The trigger that indicates that the delay is finished is mapped to input MOTIFx.MSET. One can also select the active edge for the trigger via PCONF.MSES. This delay insertion can be bypass by setting the MCMEC.MEBP bitfield to 1_B.

The PCONF.MCUE field selects the source that enables an update of the multi-channel pattern. When set to 1B, the multi-channel pattern can only be updated after the software has written a 1B into the MCMS.MNPS field.

After the update delay (if enabled), the multi-channel pattern still needs to be synchronized with the PWM signal. For this, the input MOTIFx.MSYNC is used. One can also select the active edge for the trigger via PCONF.MSYES. When the configured edge is detected in this signal, then the new multi-pattern is applied at the MOTIFx.MOUT[31:0] outputs from the output look-up table. This synchronization stage can also be bypass by setting the MCMEC.MSBP to 1_B.

STEP 5: Wait hall event

When the sampling trigger is sensed, the hall inputs are stored and compared against the current pattern, HALP.HCP, the expected hall pattern, HALP.HEP, and the previous pattern HALP.HPP.

Step 6: Hall pattern compares

If hall input == expected pattern:

- MCUE = 0 needs to be set by software
- MOTIF runs freely without software requiring to update LUT
- CHE is generated, next values from LUT are loaded to HALP, MCM
- Output modulation MCM.MCMP = MCSM
- Go to step 7

If hall input == current pattern:

- MOTIF runs freely without software requiring to update LUT
- No action is taken (line glitch deemed to have occurred)
- Output modulation MCM.MCMP = keep value
- Back to step 5

If hall input == previous pattern:

- MOTIF is stopped by hardware
- PHP is generated to software to indicate motor direction change
- Output modulation MCM.MCMP = keep value
- Back to step 0 (stop MOTIF)

If hall input != current, expected, previous patterns

- MOTIF is stopped by hardware
- WHE is generated to software
- Output modulation MCM.MCMP = MCPF.MCFV (safe pattern)
- Back to step 0 (stop MOTIF)

STEP 7: Increment LUT pointer

If the correct hall event is decoded, LUT pointers are incremented by 1. PTR always points to the current hall pattern.

- PTR = Current pattern
- PTR + 1 = Expected pattern
- PTR - 1 = Previous pattern

The LUT address pointer “PTR”, which corresponds to the current pattern, is updated and can be accessible via software through the SLUT.PTR register. Valid values are 0, 1, 2, 3, 4, and 5. Invalid values 6, 7. This operation is performed every time a correct hall event is detected.

The following table summarizes the following registers that have an impact or relationship with LUT Mode:

Table 221 Register relationship for LUT operation

Register	Description	LUT mode usage
PCONF.FTS	LUT mode can use a fault hall event signal for the operation	Can be used in LUT mode
PCONF.CHEC	LUT mode can use correct hall event clear selection	Can be used in LUT mode
PCONF.HIDG	LUT mode will only use the WHE signal for the operation when PCONF.HIDG = 1 LUT mode will use WHE and PIF_FAULT for the operation when PCONF.HIDG = 0	Can be used in LUT mode
PCONF.EWIE	External wrong hall event enable (Used with PCONF.FTS) 0 - External wrong hall event emulation signal, MOTIFx.EWHE, is disabled 1 - External wrong hall event emulation signal, MOTIFx.EWHE, is enabled.	Can be used in LUT mode

(table continues...)

Table 221 (continued) Register relationship for LUT operation

Register	Description	LUT mode usage
PCONF.MCUE	The PCONF.MCUE field selects the source that enables an update of the multi-channel pattern. When set to 1B, the multi-channel pattern can only be updated after the software has written a 1B into the MCMS.MNPS field. Multi-channel pattern software update enable 0 - Multi-channel pattern update is controlled via hardware 1 - Multi-channel pattern update is controlled via software	Can be used in LUT mode
HMEC.HPPE	Will not have an impact on the LUT mode, the hardware will use WHE signal for the operation	Cannot be used in LUT Mode
MCMEC.MNPS	Multi-channel pattern can only be updated after the software has written a 1 into the MCMS.MNPS field provided PCONF.MCUE is set to 1.	Can be used in LUT Mode

25.5.4 Quadrature decoder control

The quadrature decoder mode is selected by setting PCONF.FSEL = 01_B or PCONF.FSEL = 11_B (in this case the multi-channel mode is also enabled).

Inside the quadrature decoder mode, two different subsets are available:

- Standard quadrature mode
- Direction count mode

The standard mode is used when the external rotary encoder provides two-phase signals and additionally and index/marker signal that is generated once per shaft revolution. The direction count mode is used when the external encoder only provides a clock and a direction signal.

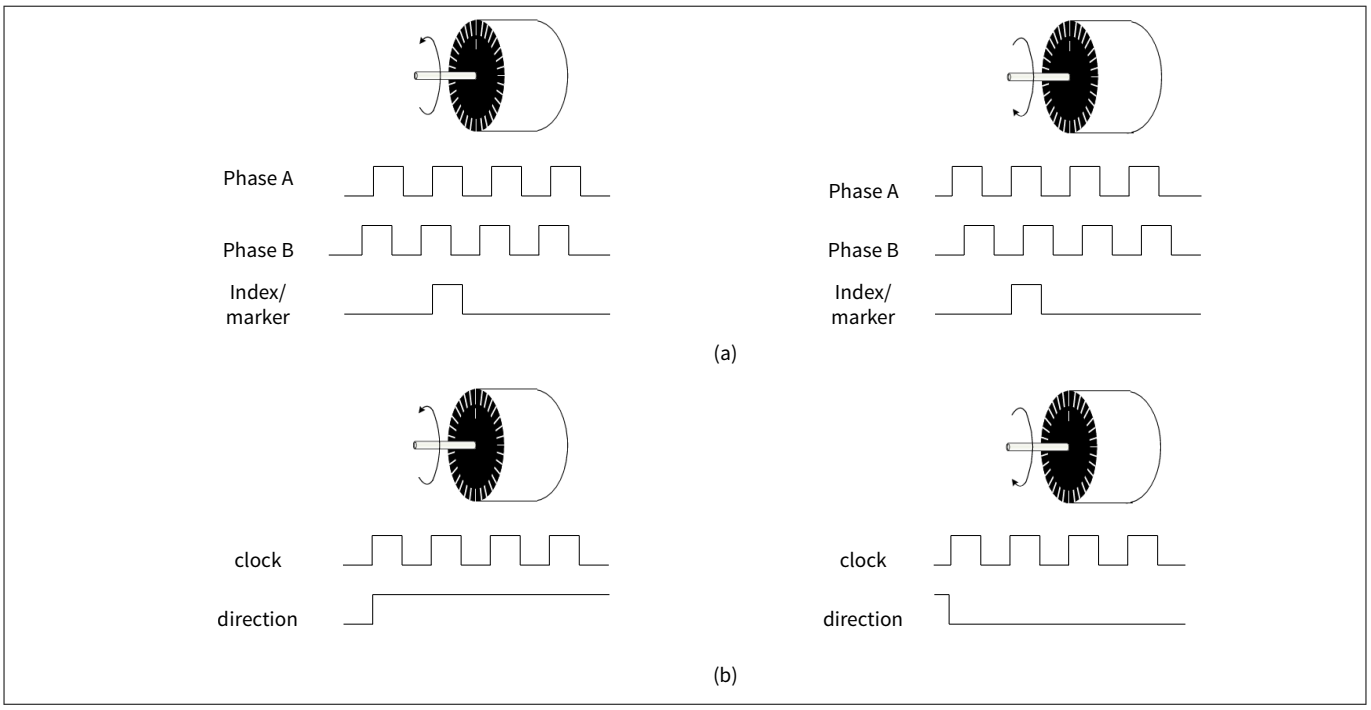


Figure 192 Rotary encoder types - a) standard two phase plus index signal; b) clock plus direction

Standard quadrature mode

The quadrature decoder unit offers a very flexible phaseA/phaseB configuration stage. Normally for a clockwise motor shaft rotation, phase A should precede phase B but nevertheless, you can configure the leading phase as well as the specific active state for each signal, see Figure 193

The selection of the phase A, phase B and Index signals are done via the QDC.PACS, QDC.PBDS, and QDC.IDS control fields. There are two major blocks that build the quadrature decoder control unit: the block that decodes the quadrature clock and motor shaft direction and the block that handles the index (motor revolution) control.

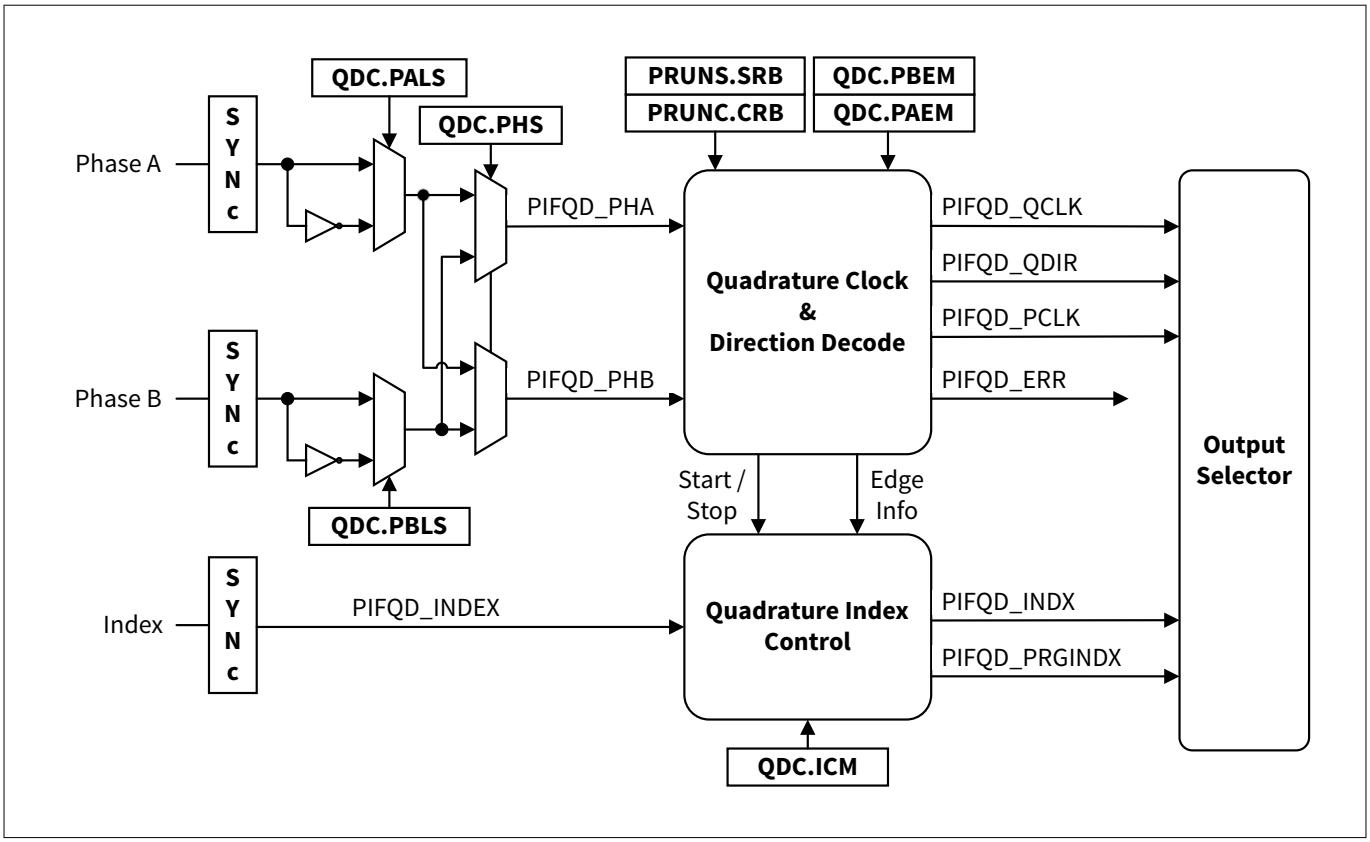


Figure 193 Quadrature decoder control overview

The quadrature clock can be used for position measurement and it can be forwarded to any of the MOTIFx.OUTy outputs. This clock can be decoded from every edge of the phase signals and therefore there can be 4 clock pulses per phase period. The edges (from the Phase A and Phase B signals) that are used to decode the quadrature clock signal can be configured by the application software.

The following figures show several examples for the clock decoding from the phase A and phase B signals. The configuration of which edges, rising, falling, both or none, is done via the QDC.PAEM and QDC.PBEM (for phase A and phase B accordingly). Please take note that when QDC.PHS is set to '1', QDC.PAEM is used to configure Phase B and QDC.PBEM is used to configure phase A.

The following diagram shows the X1 clock generation behavior.

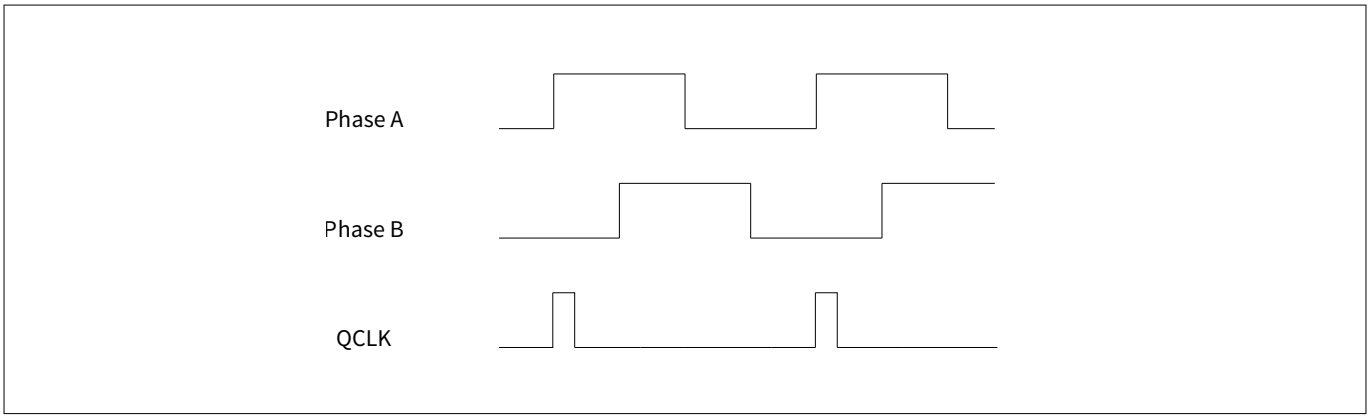


Figure 194 X1 clock generation PAEM=RISE/PBEM=NONE

The following diagram shows the X2 clock generation behavior.

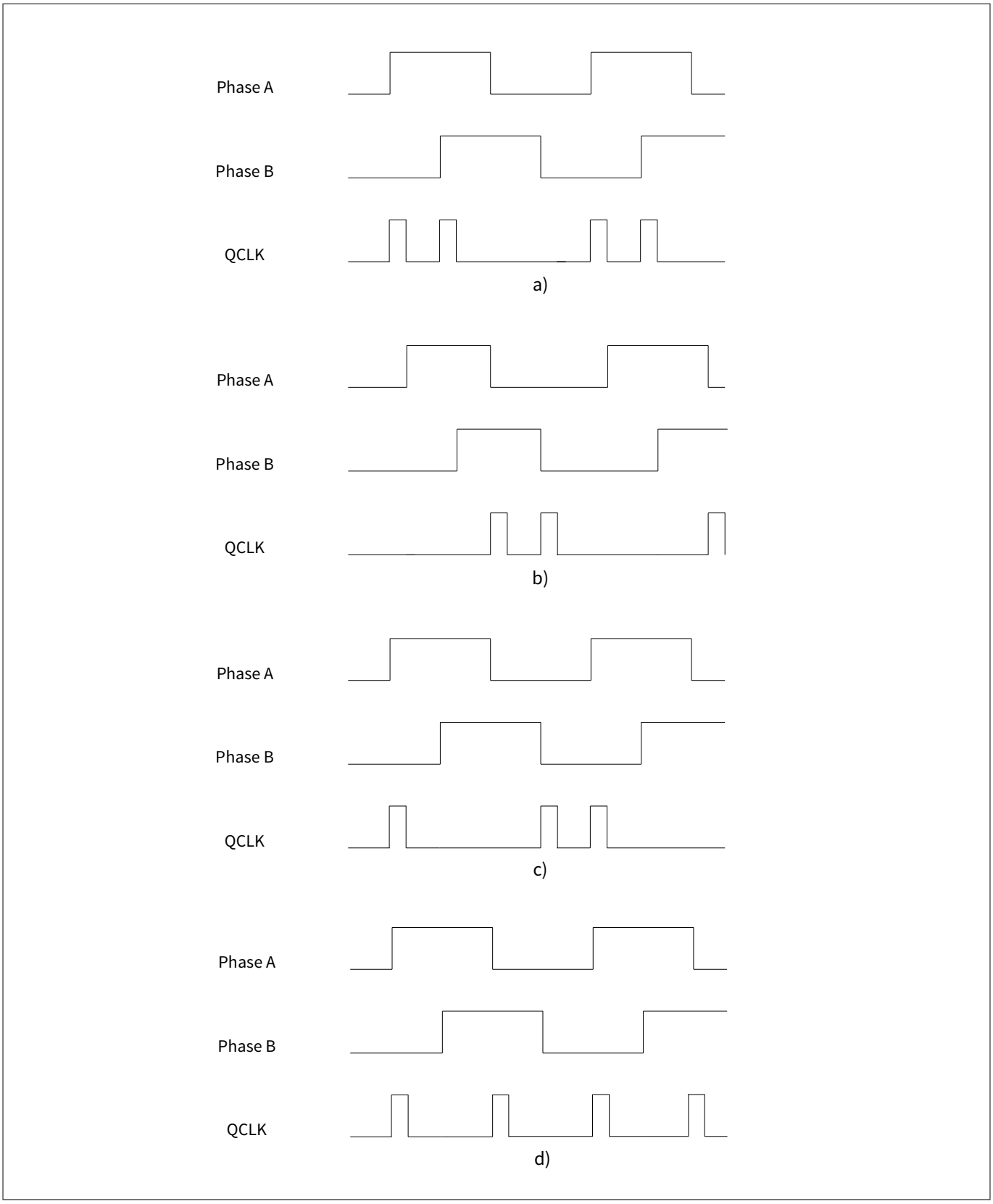


Figure 195 X2 clock generation: a) PAEM=RISE/PBEM=RISE; b) PAEM=FALL/PBEM=FALL; c) PAEM=RISE/PBEM=FALL; d) PAEM=BOTH/PBEM=NONE

The following diagram shows the X4 clock generation behavior.

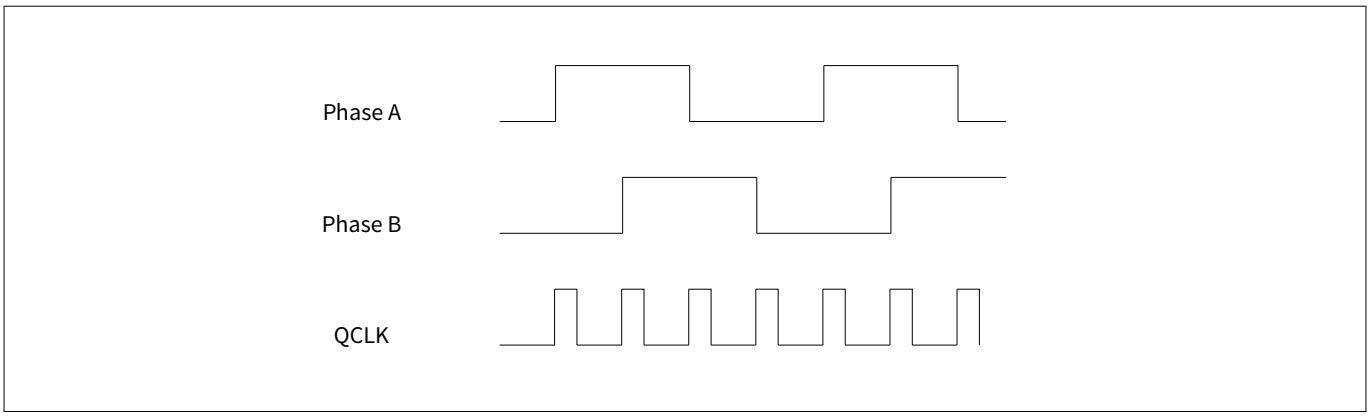


Figure 196 X4 clock generation PAEM=BOTH/PBEM=BOTH

A period clock is also generated for velocity measurement operations that can be forwarded to any of the MOTIFx.OUTy outputs. The decoding of this clock is also affected by the QDC.PAEM and QDC.PBEM fields. The direction of the motor rotation is decoded from the Phase A and Phase B signals - QDIR - and it can also be forwarded to any of the MOTIFx.OUTy outputs. The direction signal is asserted HIGH when the motor is rotating clockwise and LOW when it is turning in the counterclockwise direction.

The index control logic memorizes which was the first edge after the assertion of the index signal, so the same quadrature transition is used for index event operations. It also memorizes the direction of the first index, so that it can control when the revolution increment signal should be asserted. Two index events can be forwarded to the MOTIFx.OUTy outputs - the INDX event and the PRGINDX event - Index Control for a complete description.

An error signal, connected to a flag (and if enable an interrupt can be generated) can also be generated when a wrong phase pair is detected (this event is only available as an interrupt and it cannot be forward to the MOTIFx.OUTy outputs).

The quadrature decoder control uses the information of the current and previous phase pair to decode the direction and clocks. Both phase signals pass through an edge detection logic from which the outputs are going to be used against the valid/invalid transitions stages, see Figure 197. There is the possibility to reset the decoder state machine (but not the flags and retention mmio registers) by writing a 1_B into the PCONF.ENABLE field. The quadrature decoder current state, previous state, and current index value can be read via the PDBG register.

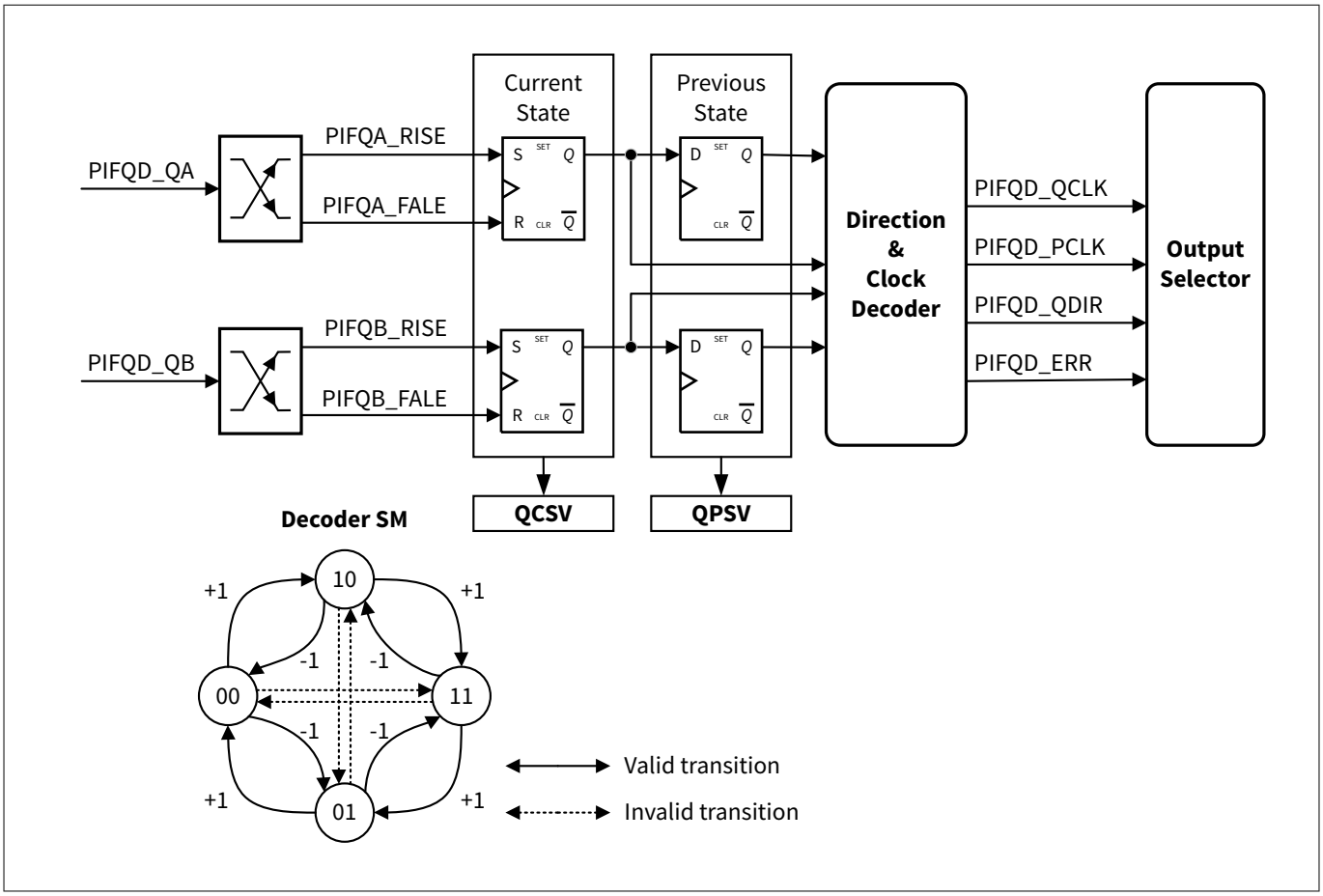


Figure 197 Quadrature decoder states

Direction count mode

Some position encoders do not have the phase signals as outputs, instead, they provide two signals that contain the clock and direction information. This is known as the Direct Count Mode.

When using a position encoder of this type, the user should set the PCONF.QDCM bit field to 1_B (enabling the direction count mode). In this case, the user must select to which of the input groups - MOTIFx.INy - the clock and direction signals are connected. This is done via the QDC.PACS and QDC.PBDS control fields. The input signals - direction and clock - are synchronized with the MOTIF module clock and shall be low pass filtered if enabled before they are forwarded to any of the MOTIFx.OUTy outputs, via the output selector. Notice that the direction input is not obligatory. One can disable it by setting the QDC.PBDS to 00_B. The output signals PCLK, INDX, PRGINDX, ERR are set to constant zero in this mode.

25.5.4.1 Quadrature clock and direction decoding

The quadrature decoder unit outputs two clocks. One clock is used for position control and therefore is generated in each edge transition of the phase signal - QCLK. The second clock is used for velocity measurements and is immune to possible glitches on the phase signals that can be present at very slow rotation speeds causing motor vibrations - PCLK.

The glitches due to motor vibrations are not necessarily electrical noise glitches. This type of vibration glitch is slow enough that it cannot be de-glitched by input filters, refer Figure 199.

The PCLK is not generated when motor direction changes (transition on QDIR). Exception is that at the initial start of motor operation. PCLK is generated for the first transition of phase A or phase B.

The decoding of the direction signal is done following the rules in Figure 197. Figure 198 shows all the valid transitions of phase A and phase B signals. Figure 200 shows the difference between the two clock signals in the case that some glitches are present in the phase signals.

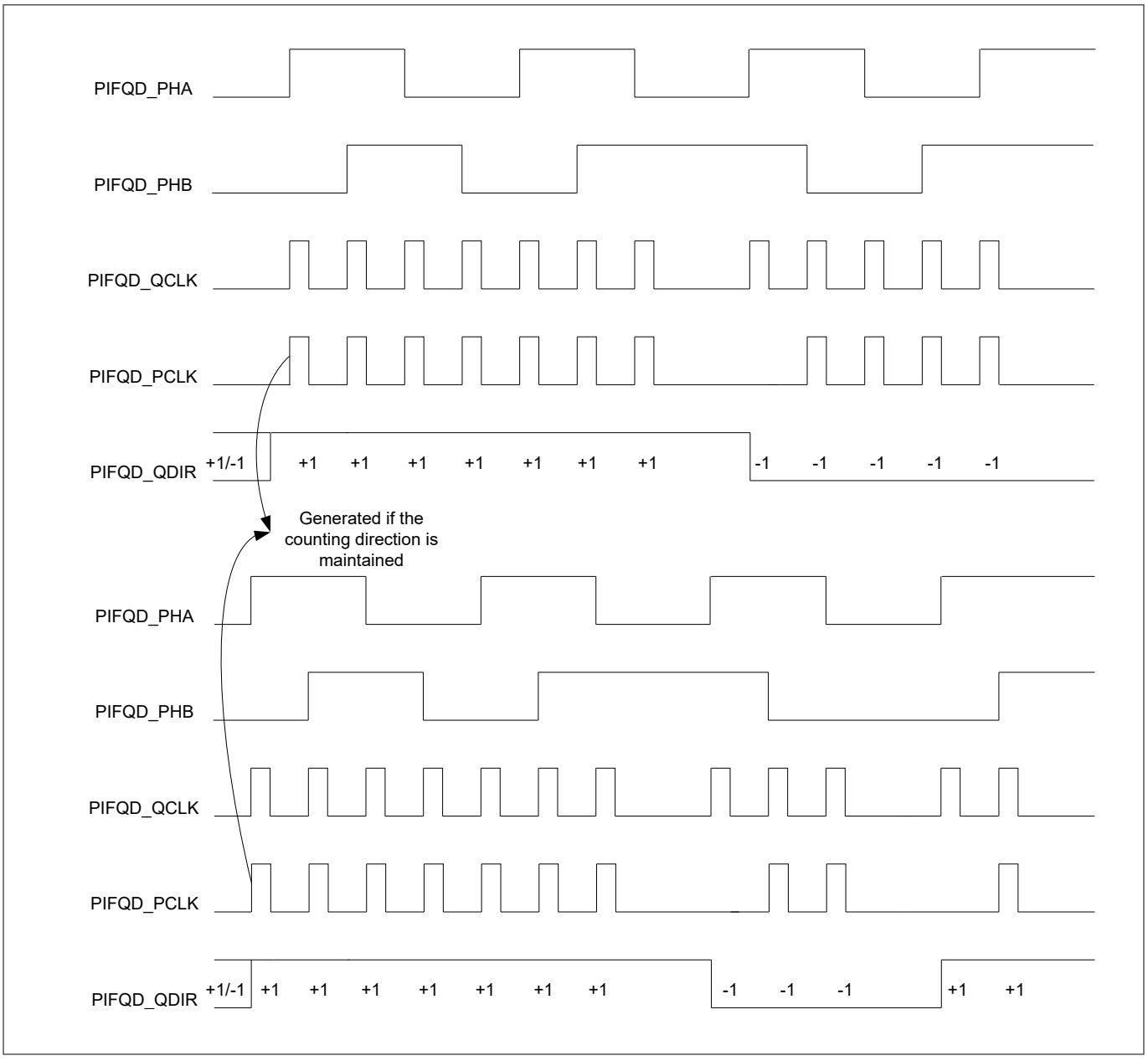


Figure 198 Quadrature clock and direction timings

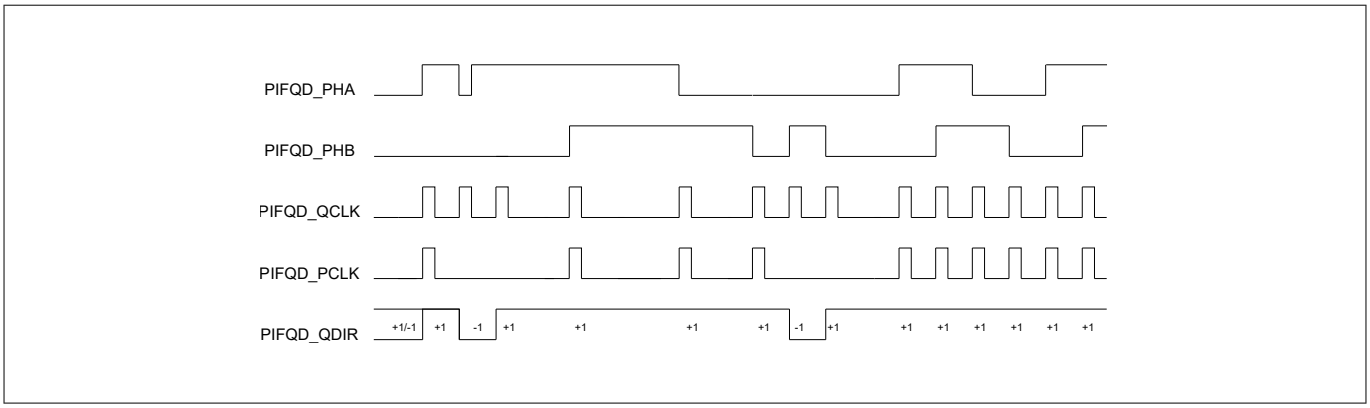


Figure 199 Quadrature clock with jitter

25.5.4.2 Index control

The index control logic has two different output events. One is asserted every time that an index signal is detected (and the motor shaft rotation is the same) - INDX – and therefore it can be used in a revolution counter. The software can monitor not only the position of the shaft but also the total number of revolutions that have occurred.

The activity of the other event - PRGINDX - can be programmed via the QDC.ICM field. Depending on the value set in the QDC.ICM, this signal can be generated:

- In every index signal occurrence
- Only on the first index signal occurrence
- Disabled - this output is never asserted

This is useful for applications that need to reset the counters on every index event or only at the first one.

The Index control logic memorizes the immediate next phase edge that follows an index so the generated signals have always the same reference. If the first-phase edge after the index is the rising edge of the phase B signal, then the index signals are going to be generated in the next index event with the rising edge of the phase B signal if the direction is kept or with the falling edge of the phase B signal if the direction has changed.

***Note**: If a x2 clock generation (for example, where only the rising edge of Phase A or Phase B is used to generate the quadrature clock) index decoding follows the same behavior as mentioned above, no index signal pulse is lost.*

Figure 200 shows the timing diagram for the generation of the index signals. In this case the QDC.ICM = 10_B, which means that the PRGINDX event is going to be generated in every occurrence of the input index.

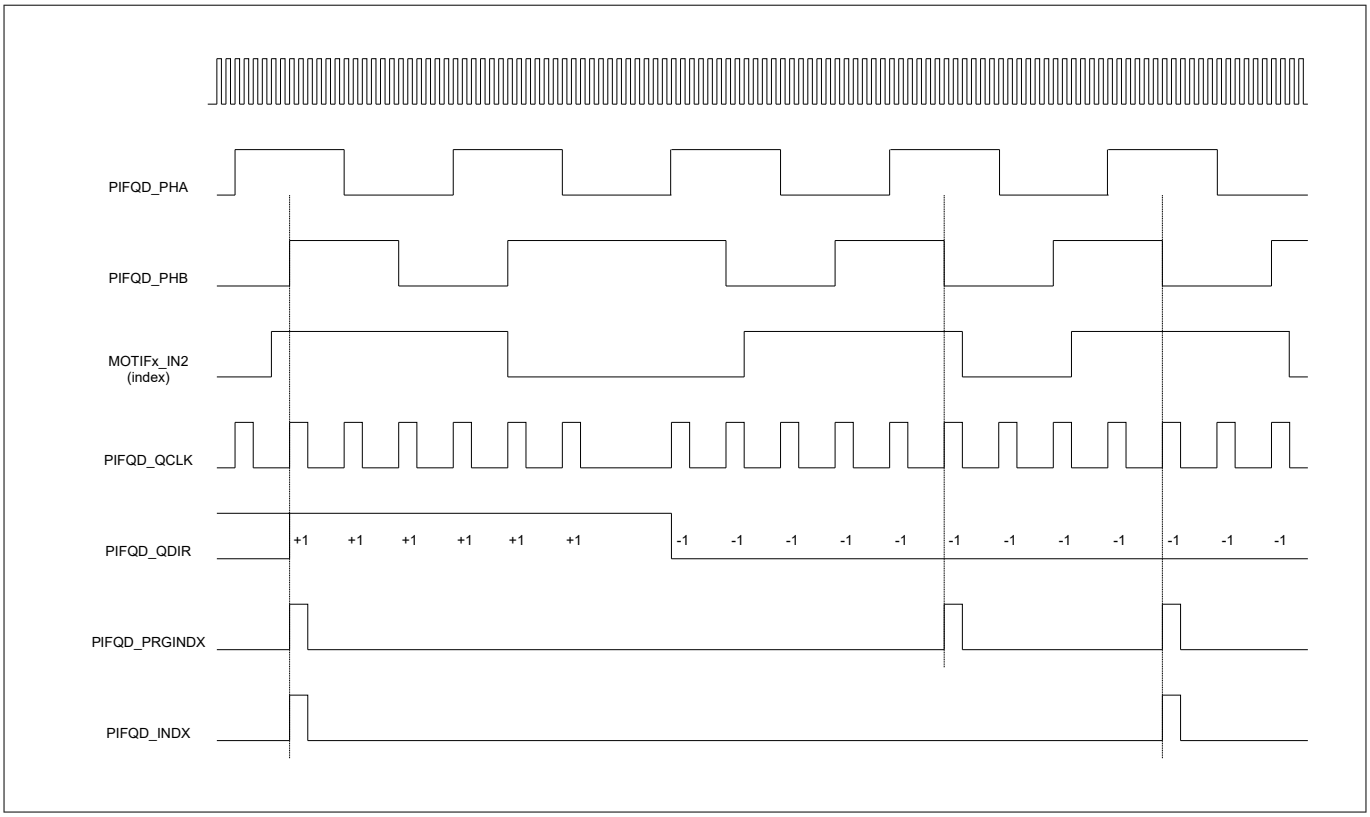


Figure 200 Index signals timing

25.5.5 Standalone multi-channel mode

The multi-channel mode can be used without the hall sensor control by setting the PCONF.FSEL = 10_B. The stand-alone multi-channel mode shall support two shadow transfer mechanisms for updating the multi-channel pattern output, one is called single shadow transfer, and the other is called Look-up table shadow transfer.

Single shadow transfer

The single shadow transfer mechanism is the same as the one described in hall sensor control section. The trigger for a pattern update can come from the software, by writing 1_B to MCMS.MNPS, or it can come from an external signal. This external signal should be mapped to the PIFMSET function by selecting the appropriate value in the PCONF.MSETS field, see Figure 187.

The synchronization between the update of the new multi-channel pattern and control signal may be also enabled by setting the MCMEC.MSBP to 0_B – the input signal MOTIFx.MSYNC is used to this function, see Figure 187.

The multi-channel pattern can be clear/load with the value preprogrammed in the MCPF.MCFV register into the MCM.MCMP - whenever the software writes a 1_B into the MCMC.MPC bitfield or if enabled when a fault is detected in the MOTIFx.EWHE module pin, see Figure 186. The value preprogrammed in the MCPF.MCFV is then applied to the MOTIFx.MOUT[31:0] outputs.

Look-up Table shadow transfer

The stand-alone multi-channel mode shall support an update of the output pattern modulation via look-up table (output LUT). To enable this operation, the LUTEN register shall be programmed to ‘1’.

The output LUT can be programmed by software, and can store up to 6 different modulation patterns, each of them consists of 32 bits. The hardware applies one modulation pattern at a time on MOTIF.MOUT[31:0] pin, this happens whenever a trigger for pattern update is received.

The following figure illustrates how the LUT is linked to the MOUT pin.

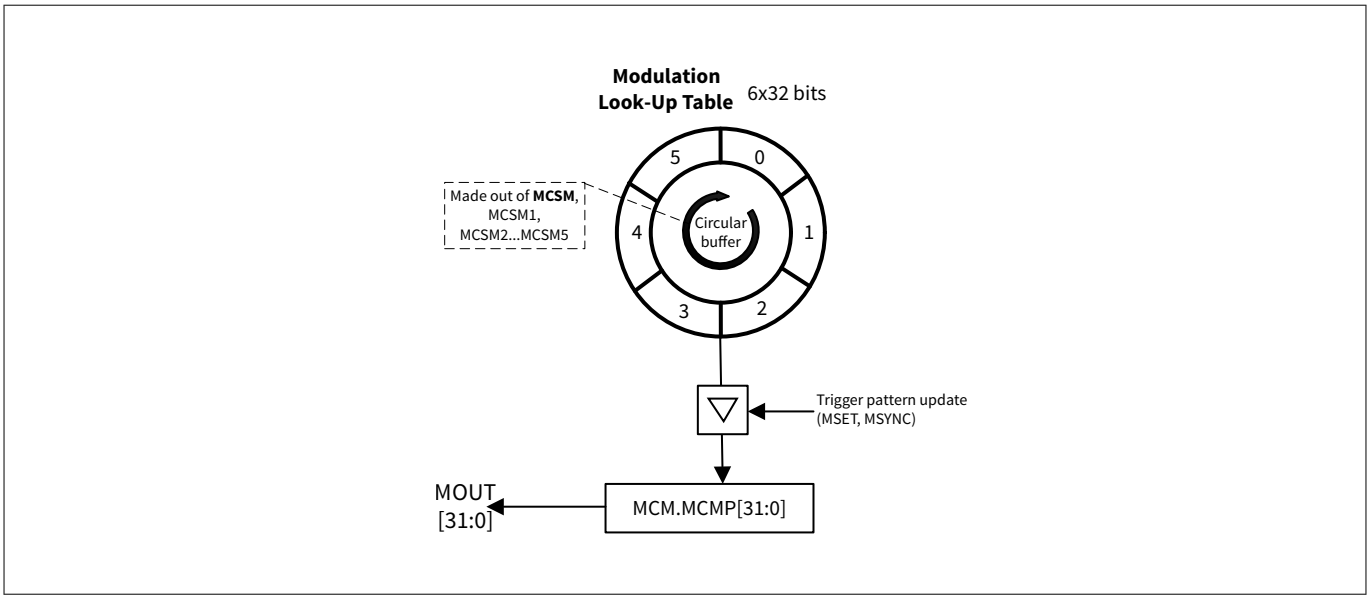


Figure 201 Stand-alone multi-channel with LUT support

The trigger for a pattern update can come from software, by writing 1_B to MCMS.MNPS, or it can come from an external signal. This external signal should then be mapped to the PIFMSET function by selecting the appropriate value in the PCONF.MSETS field. The synchronization between the update of the new multi-channel pattern and control signal may be also enabled by setting the MCMEC.MSBP to 0_B – the input signal MOTIFx.MSYNC is used to this function.

The hardware shall access the LUT content in a circular way. It starts from position 0 up to 5 and repeat again from 0. Every time a trigger for a pattern update is received the LUT address is incremented by 1.

The mapping between LUT addresses and LUT data registers is the following:

- LUT [0] corresponds to MCSM
- LUT [1] corresponds to MCSM1
- LUT [2] corresponds to MCSM2
- LUT [3] corresponds to MCSM3
- LUT [4] corresponds to MCSM4
- LUT [5] corresponds to MCSM5

The following timing diagram illustrates the operation of stand-alone multi-channel with LUT enabled. The MSET, MSYNC condition is used to increment by 1 the LUT address and also to transfer the content from LUT[x] to MOUT[31:0] output. The multi-channel pattern can be clear/load with the value preprogrammed in the MCPF.MCFV register into the MCM.MCMP - whenever the software writes a 1_B into the MCMC.MPC bitfield. Whenever the MOTIFx.EWHE input event is received, the MOTIF module is stopped and the value preprogrammed in the MCPF.MCFV is applied to MOUT[31:0] outputs. Software needs to set PCONF.HIDG=0, PCONF.FTS=0/2, MCMEC.MFCE=1 for this behavior.

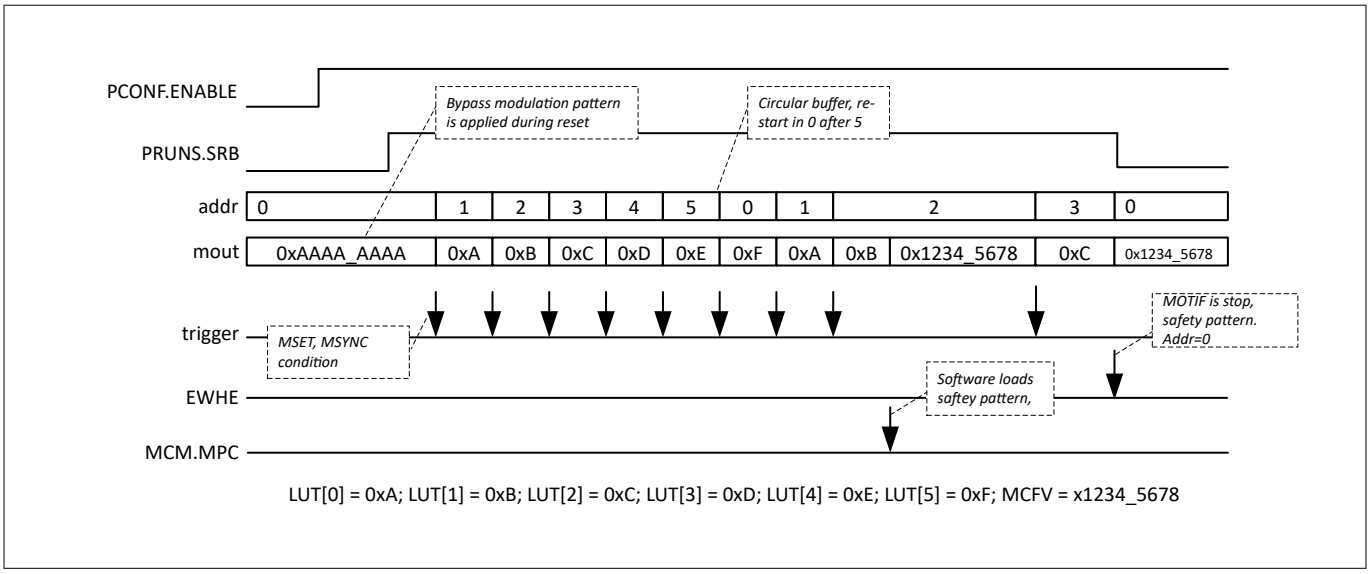


Figure 202 Timing stand-alone modulation with LUT support

***Note**: In single shadow transfer mode, the shadow transfer to MOTIF.MOUT pin always takes place from the MCSM register. Software must ensure to re-write a new pattern into this register after shadow transfer.*

In LUT mode, the software shall program the output LUT only once and the hardware shall loop through the 6-address starting from address 0. The process is stopped when MOTIF is stopped.

25.5.6 Modulation interface from MOTIF

MOTIF module provides a 32-bit register that drives multi-channel PWM patterns. This register supports several operating modes including hall sensor mode with multichannel, stand-alone multichannel mode, and quadrature mode with stand-alone multichannel. Each modulation output register can be connected to up to 8 TCPWM counters. For all connected counters, both the line_out and the line_compl_out signals can be modulated synchronously by the MOTIF module.

The 32-bit modulation output register MOTIFx.MOUT[31:0] connects to GRPx.COUNTER[7:0]. 4 bits required to modulate each TCPWM counter, 2 bits each for line_out and line_compl_out as described in the table Table 222

Table 222 MOTIFx.MOUT[31:0] modulation output mapping to GRPx.COUNTER[7:0]

MOTIF Modulation Output bits	Connected counters	Modulated line out
MOTIF_GRPx.MOTIF0.MOUT[1:0]	GRPx.COUNTER [0]	Line_out
MOTIF_GRPx.MOTIF0.MOUT[3:2]		Line_compl_out
MOTIF_GRPx.MOTIF0.MOUT[5:4]	GRPx.COUNTER [1]	Line_out
MOTIF_GRPx.MOTIF0.MOUT[7:6]		Line_compl_out
MOTIF_GRPx.MOTIF0.MOUT[9:8]	GRPx.COUNTER [2]	Line_out
MOTIF_GRPx.MOTIF0.MOUT[11:10]		Line_compl_out
MOTIF_GRPx.MOTIF0.MOUT[13:12]	GRPx.COUNTER [3]	Line_out
MOTIF_GRPx.MOTIF0.MOUT[15:14]		Line_compl_out
MOTIF_GRPx.MOTIF0.MOUT[17:16]	GRPx.COUNTER [4]	Line_out
MOTIF_GRPx.MOTIF0.MOUT[19:18]		Line_compl_out

(table continues...)

Table 222 (continued) MOTIFx.MOUT[31:0] modulation output mapping to GRPx.COUNTER[7:0]

MOTIF Modulation Output bits	Connected counters	Modulated line out
MOTIF_GRPx.MOTIF0.MOUT[21:20]	GRPx.COUNTER [5]	Line_out
MOTIF_GRPx.MOTIF0.MOUT[23:22]		Line_compl_out
MOTIF_GRPx.MOTIF0.MOUT[25:24]	GRPx.COUNTER [6]	Line_out
MOTIF_GRPx.MOTIF0.MOUT[27:26]		Line_compl_out
MOTIF_GRPx.MOTIF0.MOUT[29:28]	GRPx.COUNTER [7]	Line_out
MOTIF_GRPx.MOTIF0.MOUT[31:30]		Line_compl_out

The modulation logic is available on all counter groups, if enabled.

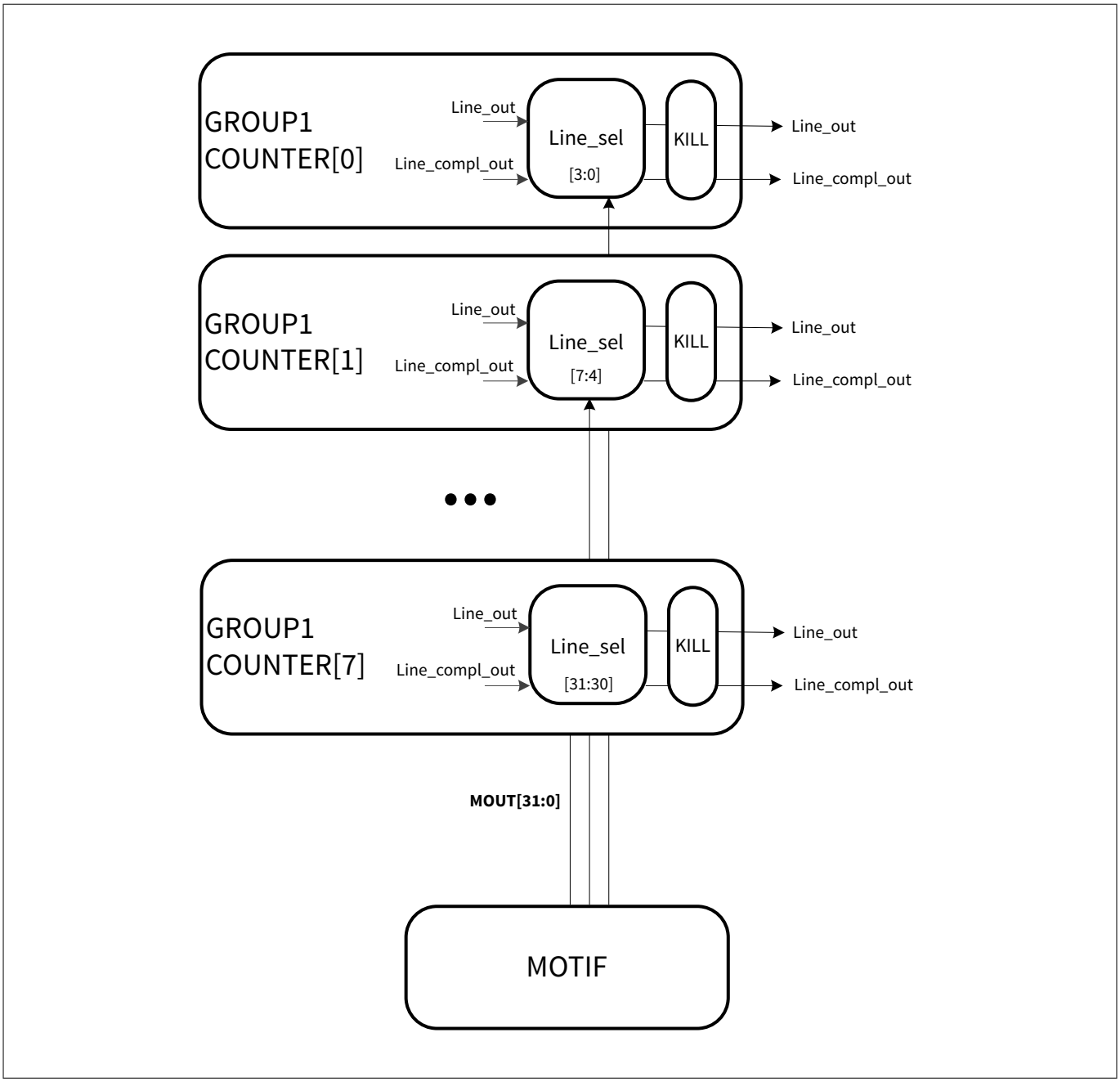


Figure 203 MOTIF0 modulation connection to group 1 counters

The MOTIF_GRPx_MOTIF0.MOUT[3:0] mapping to GRPx.COUNTER[0] is described in the tableTable 223.The same mapping applies to all remaining counters in the group when enabled

Table 223 Multi-channel pattern influence on PWM output

Multi-channel pattern	Value	PWM output behavior on the GRPx.COUNTER[0]
MOTIF_GRPx_MOTIF0.MOUT[1:0]	00	0
	01	PWM
	10	!PWM
	11	1

(table continues...)

Table 223 (continued) Multi-channel pattern influence on PWM output

Multi-channel pattern	Value	PWM output behavior on the GRPx.COUNTER[0]
MOTIF_GRPx_MOTIF0.MOUT[3:2]	00	0
	01	PWM
	10	!PWM
	11	1

***Note**:*

If MOTIF gets disabled via PCONF.ENABLE=0, the modulation output from MOTIF gets reset to a default value. The reset value bypasses the PWM inside the counter and has not effect on the PWM signal.

MOTIF0.MOUT = 0x9999_9999

If MOTIF gets stopped via PRUNS.SRB=0, the modulation output from MOTIF will keep the existing value.

25.5.6.1 Modulation inside counter logic

The PWM signal conditioning can be done via MOTIF. The counter MMIO LINE_SEL register needs to be configured to select MOTIF as the input source for modulation.

Figure 204 illustrates the line out generation logic of PWM, PWM_DT modes.

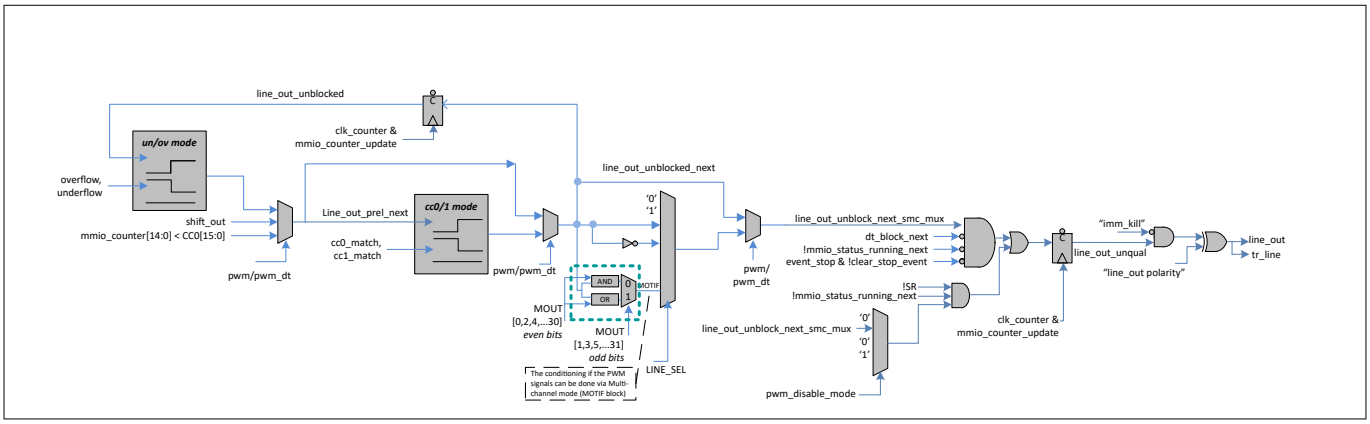


Figure 204 line_out generation logic

MOTIF interface supports 0, 1, PWM and inverse PWM . The MOTIF output mapping for line_out and line_compl_out would be:

Table 224 MOTIF output mapping for line_out

Motif output mapping	Line_out
mout[1],mout[0]	
00	0
01	PWM
10	!PWM
11	1

Similarly line_compl_out would also be updated as:

Table 225 MOTIF output mapping for line_compl_out

Motif output mapping	Line_compl_out
mout[3],mout[2]	
00	0
01	PWM
10	!PWM
11	1

25.5.7 Synchronous start

The MOTIF module has a synchronous start event that can be forwarded to any of the MOTIFx.OUTy outputs via the output selector. This event can be used together with the TCPWM counters for a complete synchronous start of both modules. The synchronous start is linked with the run bit of the MOTIF module, which means that every time that the run bit is set a transition is generated that can be forward to one of the outputs.

By using the synchronous start output, the software does not perform two independent accesses to start the MOTIF and the TCWPM counters and therefore is guaranteed that both modules start their operation at the exact same time.

25.5.8 Output selector

The output selector receives the signals generated by the three main sub units comprised inside the MOTIF:

- signals from the quadrature (or direction count) control logic
- signals from the hall sensor control logic
- signals from the multi-channel control logic

The first stage of the output selector, as seen in Figure 205 will gate the signals that are not being used depending on the function that has been selected by the PCONF.FSEL field (this field dictates which MOTIF sub unit is being currently used - quadrature mode, hall sensor mode, multi-channel mode). If a specific sub unit is not being used, t. Generate signals coming from this control logic will be disabled.

On the second stage of the output selector, one can use the node pointers to forward the specific signal/event, generate by a sub unit to the wanted MOTIF output pin – MOTIFx.OUT[0...6]. The node pointer configurations are handled via the QOSC, HOSC and MOSC registers, for the quadrature decoder, hall sensor and multi-channel control logic respectively.

An OR stage will group all the signals/events that have been forward to the same MOTIF output pin - for example, if the QCLK and Start (from the quadrature control logic) have been configured in their node pointer, to be forward to MOTIFx.OUT0. Then at this specific pin, one should expect to see both events ORed (an event is going to be seen whenever a QCLK is generated or when a Start is detected).

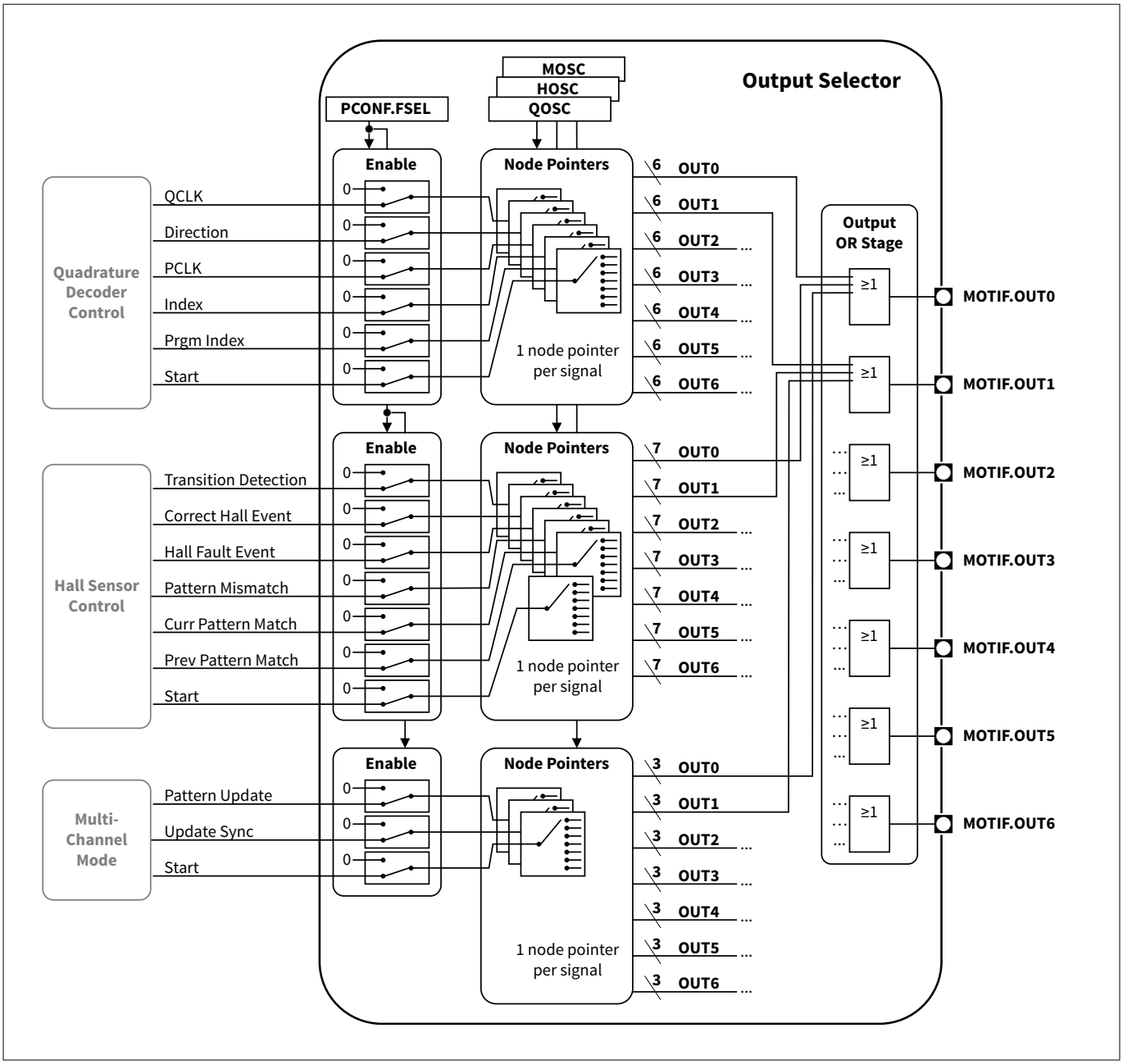


Figure 205 Output selector logic diagram

25.5.9 Using the MOTIF

The MOTIF module needs to be linked with a TCPWM counters module to perform the full set of functions in each of the possible modes (because it does not contain built-in counters/timers).

To operate the MOTIF in the quadrature decoder mode (at least) five counters are needed. The hall sensor mode needs at least 3 counters to control a brushless DC motor and also (at least) two counters to control delay. The stand-alone multi-channel mode linking configuration depends heavily on the use cases and therefore the number of counters used is freely chosen by the user.

Hall sensor mode usage:

When using the hall sensor mode of the MOTIF, the multi-channel mode is also working to provide the multi-channel pattern to TCWPM counters. The counter module performs the multi-channel PWM Modulation. For

this purpose, the counter needs to be configured via LINE_SEL register to select MOTIF as the signal conditioning source for PWM signal.

In Figure 206, the hall sensor mode is used with 2 groups of counters with five counters in total. Group 1 is configured with advance motor control features (three counters). Group 0 is configured as a standard General-purpose counter (two counters). Group 0 counters is being are used to control the delays between the hall transition detection of the hall inputs and the actual sampling, and also to control the delay between a correct hall event and the multi-channel pattern update enable.

The rising edge of Group 0 counter 0 tr_out0 is used as the finish trigger for the first delay. It connects to MOTIF input HSD. While Group 0 counter 0 tr_out1 is used as the delay between the hall input sampling and the modulation pattern update, it connects to MOTIF input MSET. Group 0 Counter 0 is configured in single shot mode, so that the time delay can be re triggered every time that a request from the MOTIF occurs.

The Group 0 Counter 1 is being used in capture mode, to capture the time between the correct hall events (storing this way the motor speed between two correct hall events). The correct hall event of the MOTIF is used as the capture trigger for the counter while the pattern mismatch event is used as stop the Group 0 Counter 0 and Group 0 Counter 1. The capture and stop triggers are configured in the specific timer slices as active on the rising edge.

The Group 1 counters are used to generate the PWM signals to control the motor and therefore, the multi-channel pattern outputs MOTIFx.MOUT[31:0] are linked to this unit.

To close the multi-channel loop, an output of the Group 1 counters needs to be connected to the MOTIF module (tr_out0 period match), so that the multi-channel pattern is updated synchronously with the PWM period. All MOTIF input and output events are connected via trigger multiplexer, whereas MOTIF modulation output is connected internally via hardware connection to TCPWM counters Group 1.

Counter configuration:

- 2 Groups of counters
  - G0_CNTx standard counter (two counters)
  - G1_CNTx with advanced motor control (three counters)

G0_CNT0 is used to delay hall input sampling and to generate MSET signal to MOTIF (one-shot timer mode).

- G0_CNT1 is used capture mode
- G1_CNT[2:0] is used to generate 3-phase PWM (PWM right-aligned mode)

MOTIF configuration:

- OUT0 = edge detection
- OUT1 = correct hall event
- OUT2 = pattern mismatch
- OUT3 = wrong hall event
- OUT4 = start

MOTIF <> TCPWM connectivity

- OUT0 is used to reload G0_CNT0 via a 1-to-1 trigger
- OUT1 is used to capture G0_CNT1 via 1-to-1 trigger (it is also used to reload G0_CNT1 since **TCPWM does not have a capture clear feature**)
- OUT2 is used to stop G0_CNTx via group trigger
- OUT3 is used to stop G1_CNTx via group trigger
- OUT4 is used to reload G1_CNTx via group trigger
- G0_CNT0 TROUT0 (CC0 match) is routed to HSD
- G0_CNT0 TROUT1 (period match) is routed to MSET
- G1_CNT0 TROUT0 (period match) is routed to MSYNC

PWM and MOTIF are synchronously started by making use of the RUN bit via OUT4.

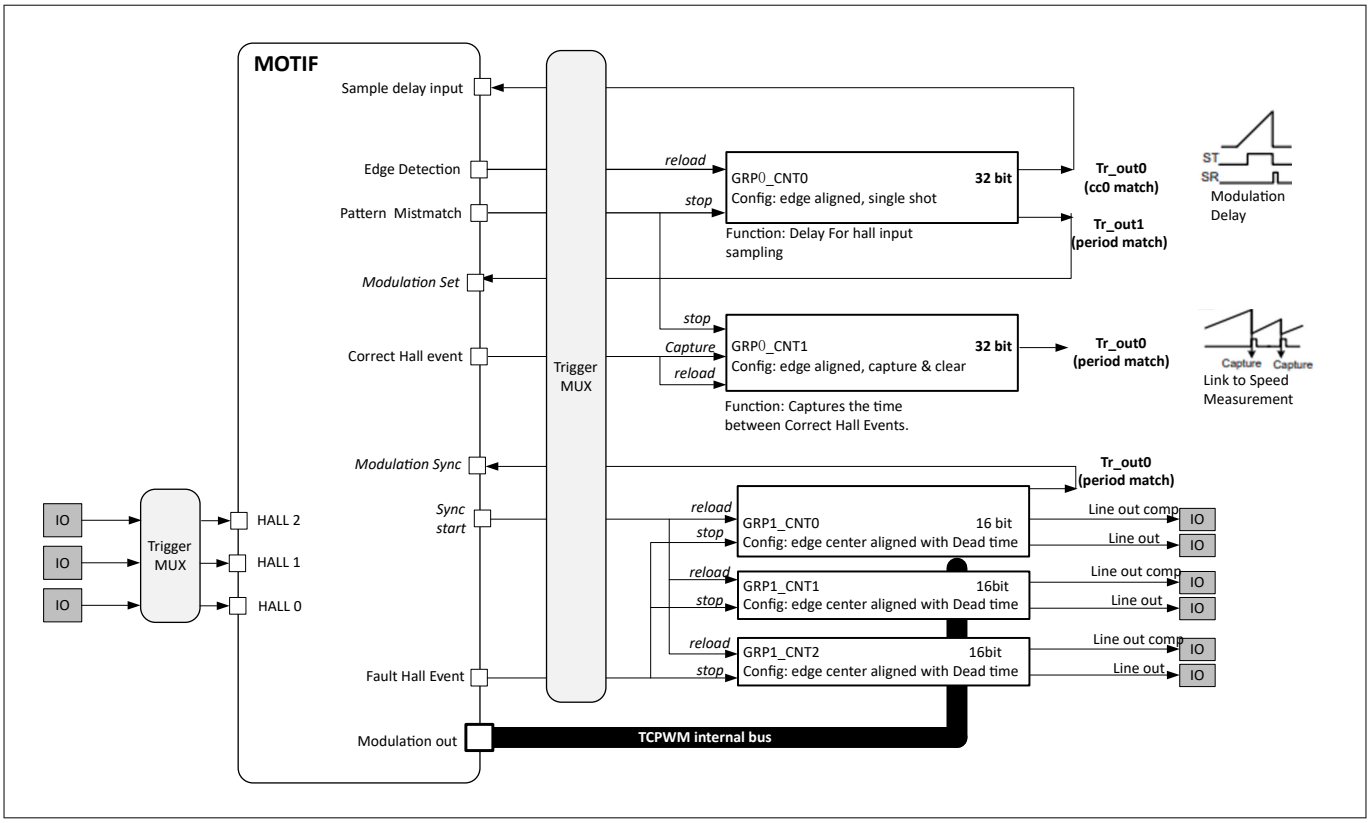


Figure 206 Hall sensor mode usage - profile 1

Quadrature decoder mode usage:

The quadrature decoder mode can be used in a very flexible way when connected with a TCPWM Counter module. The connection profile depends on the number of functions that the user wants to perform in parallel: position control, revolution control, and velocity measurement.

Quadrature decoder usage - Tick and revolution compare plus velocity between N ticks

In Figure 207, the MOTIF is linked with four TCPWM counters via a trigger multiplexer. In this profile, the user in quadrature decoder mode has a counter being used for position control (comparison), one counter for revolution counting, and two aggregated counters used for velocity measurement.

Counter 0 is connected to the QCLK and direction events. This imposes to configure the QCLK as counting functionality and direction as an up/down counting function, inside Counter 0. This counter is then used to track the actual position of the system and the compare channel can be configured to trigger the required actions, when the position reaches a certain value.

Counter 1 is connected to the INDX event (Index event). This event is used as a motor revolution counter. The compare channel can be programmed to trigger an interrupt every time that the motor performs N revolutions.

Counter 2 and Counter 3 are used to perform velocity measurements. Counter 2 uses the PCLK as a counting event. The compare channel of this slice is then used to trigger a capture event in Counter 3. The last one is using the module clock and therefore in every capture event, the actual system ticks are captured. This way the user knows how much time has elapsed between N phase periods and can calculate the corresponding velocity profile.

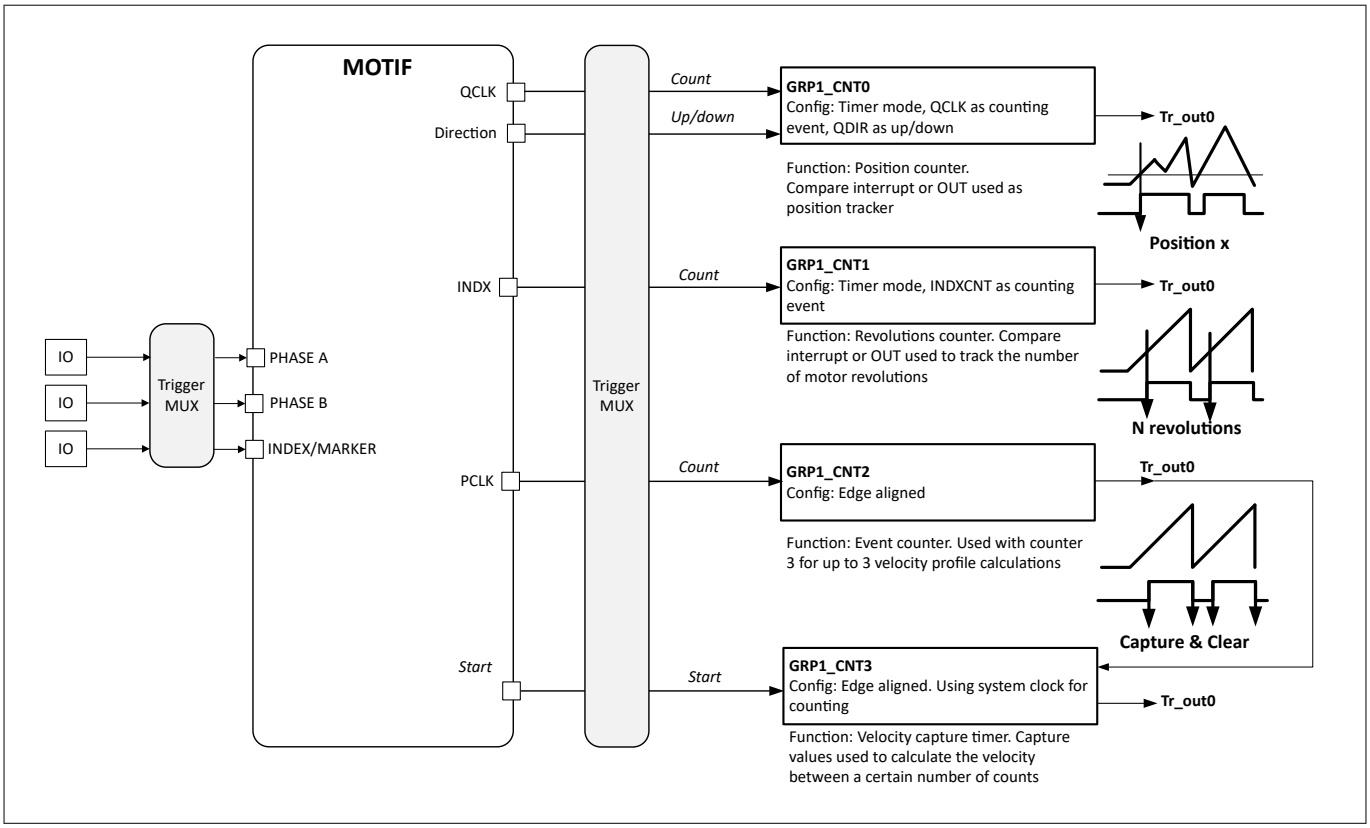


Figure 207 Quadrature decoder mode usage - profile 1

Quadrature decoder usage - Extended tick comparison plus velocity between N ticks

The profile demonstrated in the figure below enables the usage of 2 compare channels for the position control. This profile is especially useful for multi-operations during just one motor revolution.

Counter 0 and Counter 1 are using the QCLK event as a counting function and the direction as an up/down function. The comparing channels in each counter are then programmed with different compare values.

Counter 2 and Counter 3 are used in the same manner as profile 1, see Figure 207

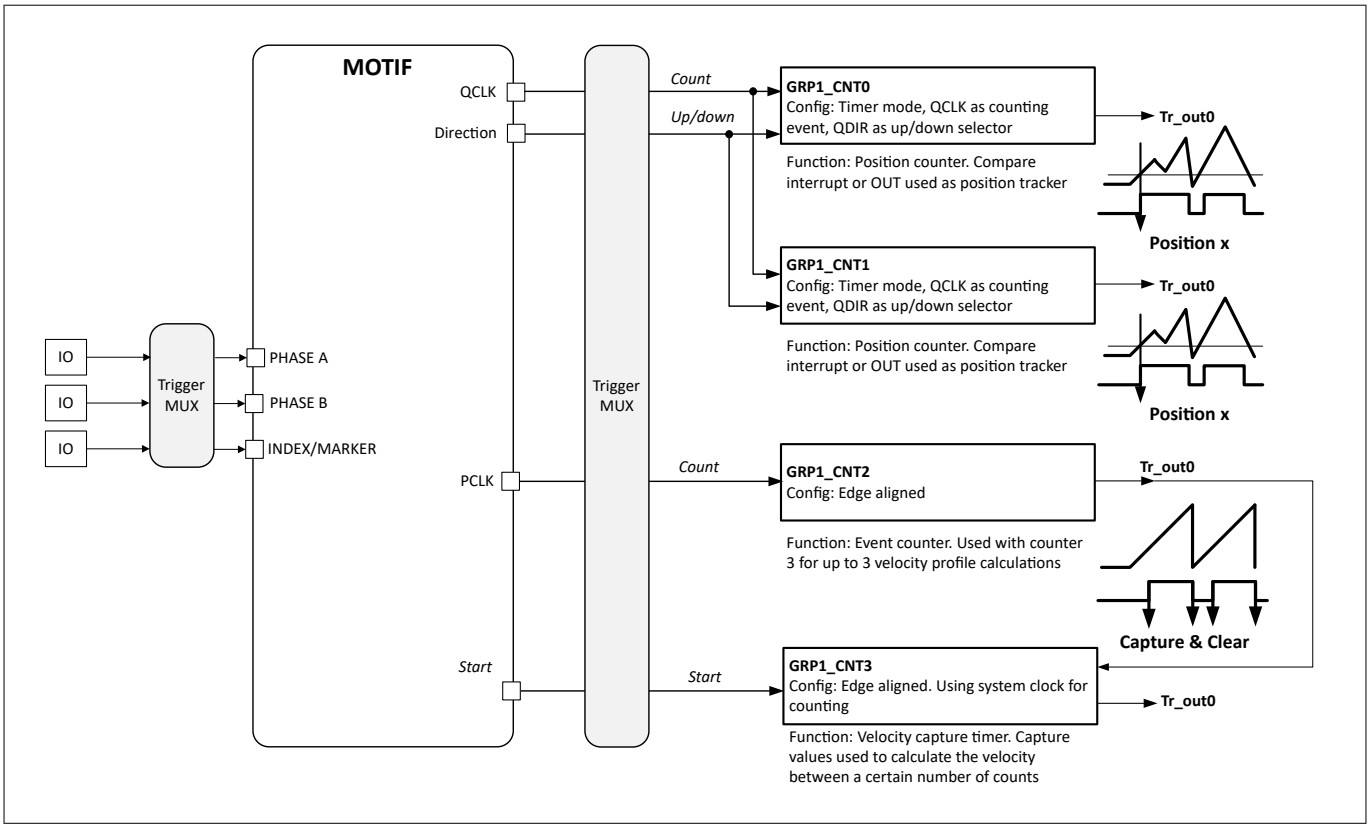


Figure 208 Quadrature decoder mode usage - profile 2

Quadrature decoder usage - Tick and revolution comparison with index clear plus velocity between N ticks

In some applications it is useful to use the index marker as a clear signal for the position and velocity control. The PRGINDX event can be configured to be generated only at the first index marker or at all maker hits. This PRGINDX event is then connected to the specific counter and used as a clear function. The following figure shows the adaptation of profile 1 with the PRGINDX event used as a clear function.

The same procedure can be used in profile 2, so that we can have the 2 compare channels plus the velocity measurement and the PRGINDX as clear signal.

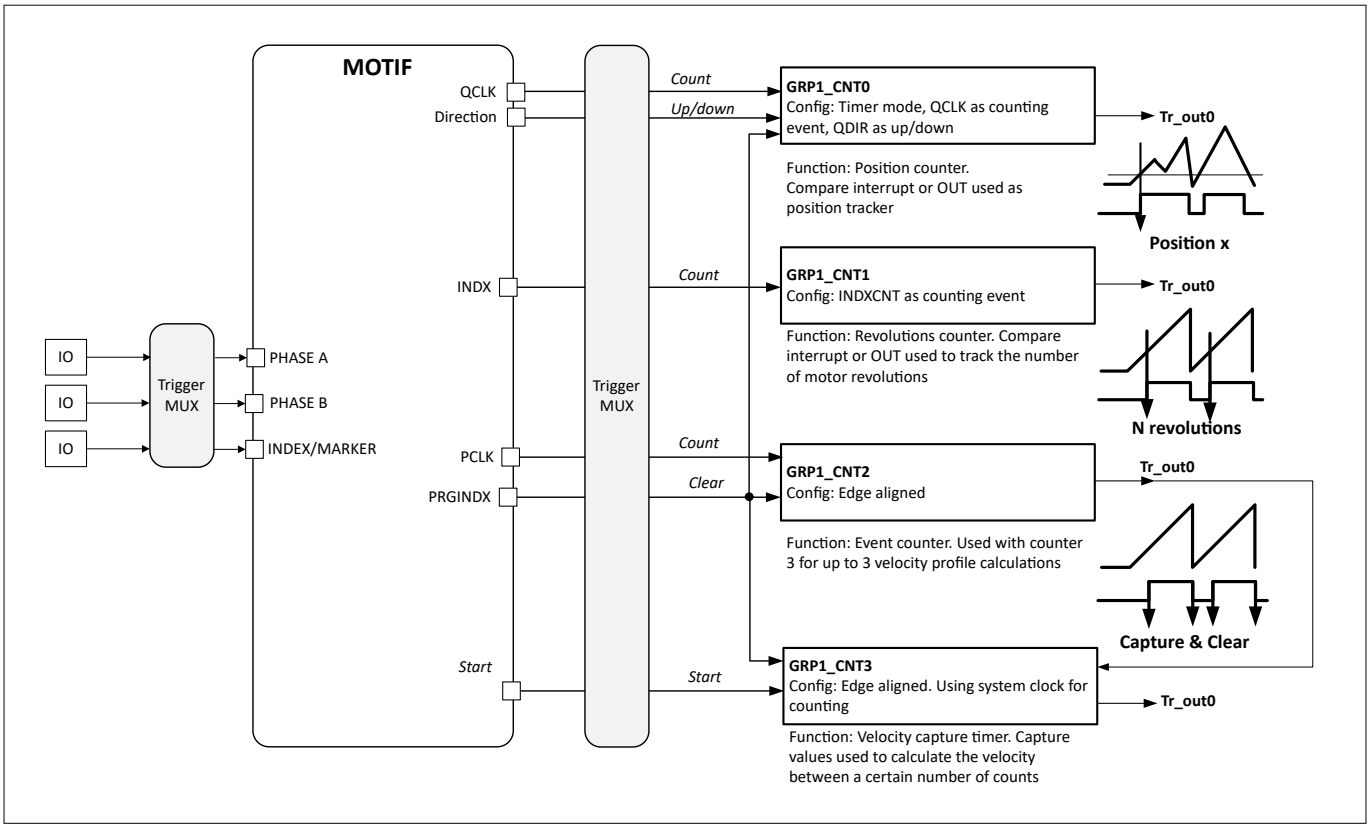


Figure 209 Quadrature decoder mode usage - profile 3

Quadrature decoder usage - Tick comparison plus micro tick velocity for slow rotating speeds

When the motor rotating speed is slow, it may not be suitable to discard the time between each occurrence of a rotary encoder tick.

In a fast-rotating system, the time between ticks is normally discarded and the velocity calculation can be done, by considering the number of ticks that have been elapsed since the last ISR. But in a slow rotating system, considering the number of ticks may not be enough (because of the associated error), and the software needs also to take into consideration, the time between the last tick and the actual ISR trigger.

The following timing diagram shows a slow rotating system, where the ISR for the velocity calculation is triggered in a periodic way. In this case, because of the small number of ticks between each ISR trigger, the software needs to know not only the number of elapsed ticks but also the elapsed time between the last tick and the ISR occurrence.

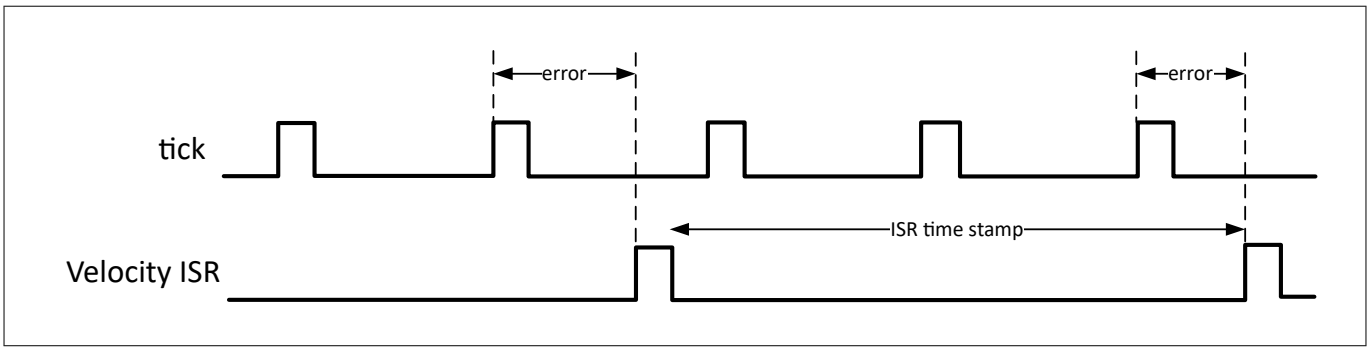


Figure 210 Example slow rotating system

It is possible to build a profile with the MOTIF and one counter module to perform a control loop that is immune to this slow velocity calculation pitfall. Counter 0, is used to monitor the current position of the motor shaft.

Three additional counters are needed to build the slow velocity calculation loop: Counter 1, Counter 2 and Counter 3.

Counter 1, is counting the number of ticks (PCLK) that are elapsed between each velocity ISR occurrence. Counter 2 is counting the time between each tick (PCLK) occurrence. This Counter is cleared and started within every tick and therefore it always keeps the timing info between the last and the current tick. Counter 3, is controlling the periodicity of the velocity ISR. Every time that a velocity ISR is triggered, Counter 3 also triggers a capture in Counter 2 and a capture & clear in Counter 1.

With this mechanism, every time that the software reads back the captured values from Counter 1 and Counter 2, it can calculate the speed based on:

- The number of ticks elapsed since the last ISR
- Plus the amount of time elapsed between the last tick and the ISR

This control loop therefore offers a very accurate way to calculate the motor speed within systems with low velocity.

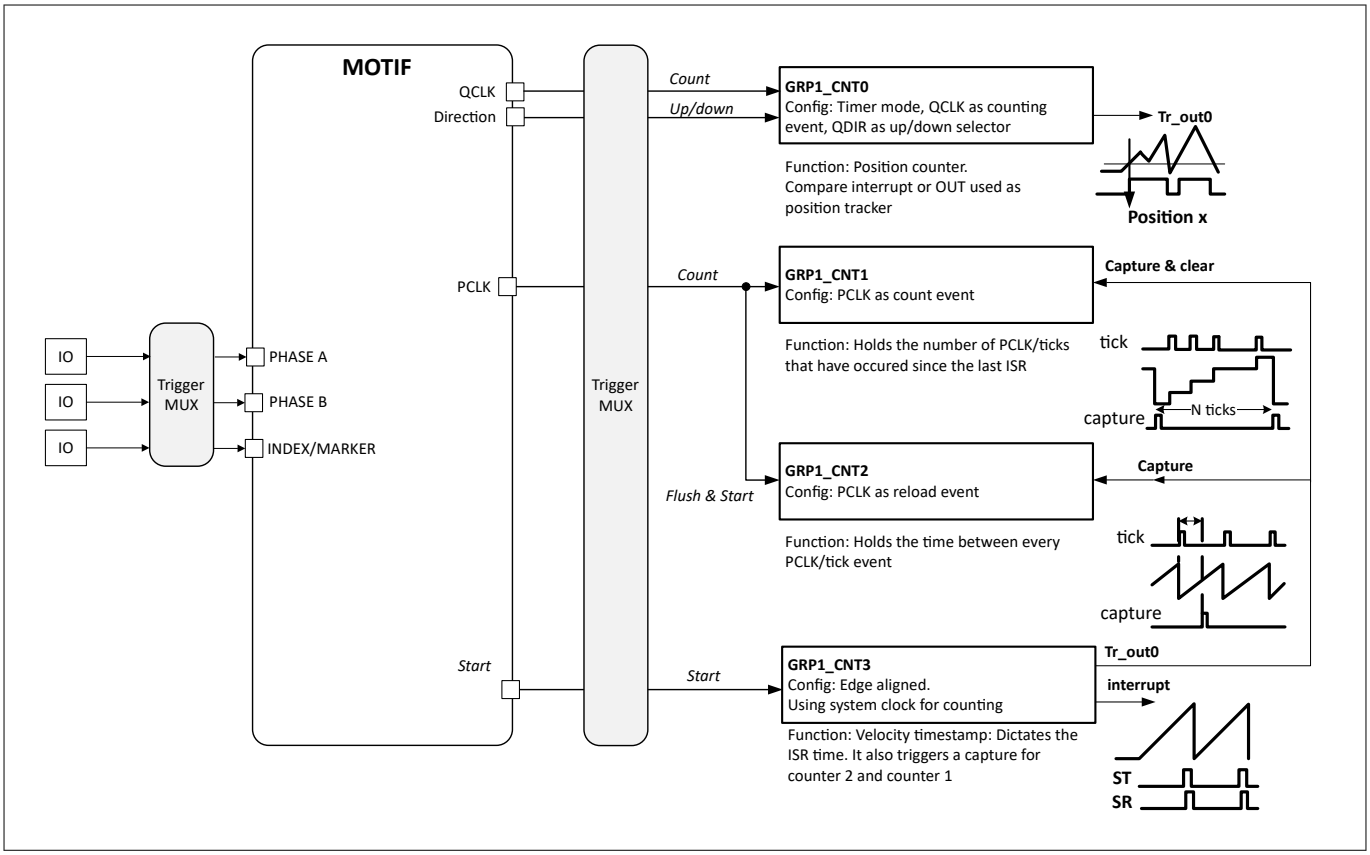


Figure 211 Quadrature decoder mode usage - profile 4

Stand-alone multi-channel mode usage

The multi-channel mode can be used in the MOTIF module without being linked with the hall sensor mode. This is especially useful for performing generic control loops that need to be perfectly synchronized.

The stand-alone multi-channel mode is very generic and depends very much on the control loop specified by the user. A generic profile for the multi-channel mode is to use a set of PWM counters to perform the signal generation and a General-purpose counter linked with an external pin that is used as a trigger for updating the multi-channel pattern.

On Figure 212 a General-purpose counter is used to control the delay between the update of an external signal (for example, an external sensor) and the update of the multi-channel pattern. Note that this external signal can also be connected to the MOTIF module if no timing delay is needed or it can be just controlled by software, by writing a one into the MCMS.MNPS field.

One of the PWM counter tr_out outputs can then be chosen to act as a synchronization trigger between the generated signal and the update of the multi-channel pattern.

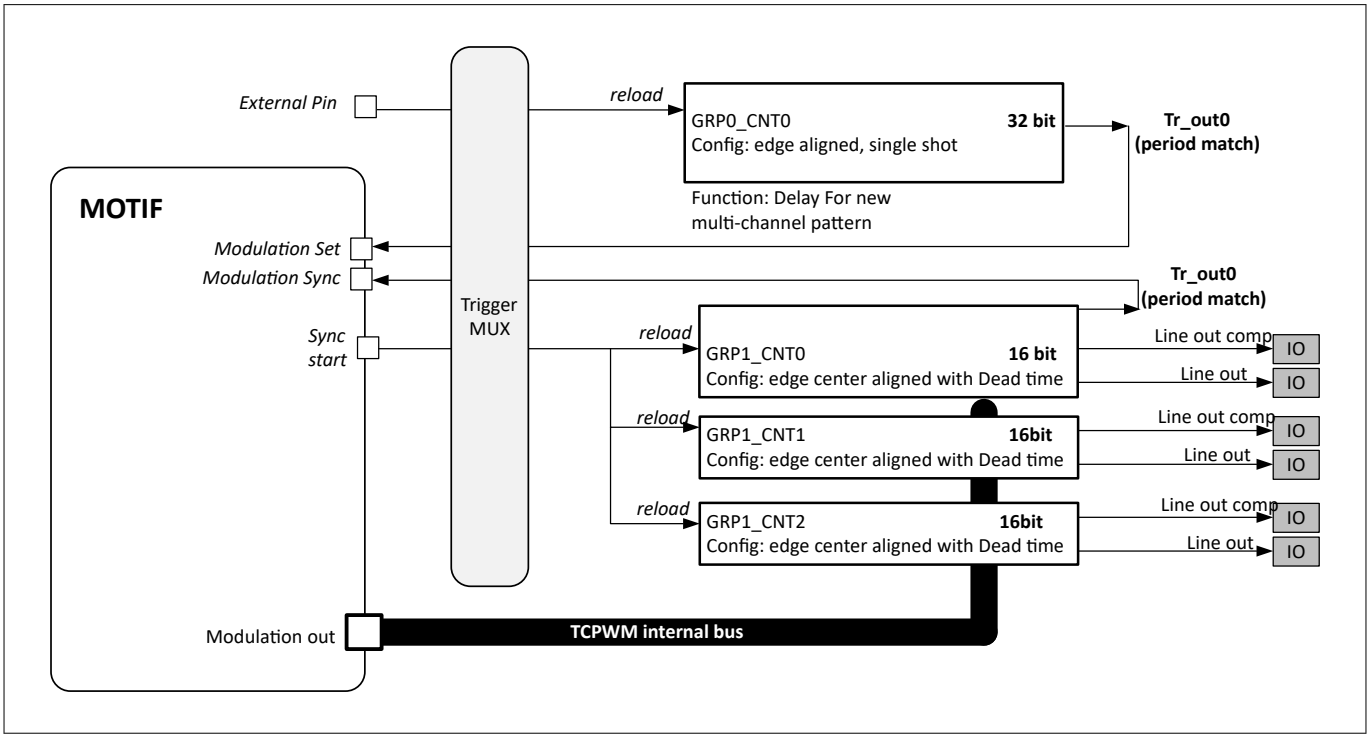


Figure 212 Stand-alone multi-channel mode usage

PWM output controlled by MOTIF modulation pattern

Figure 213 shows different types of PWM switching schemes supported by MOTIF together with the TCPWM counter.

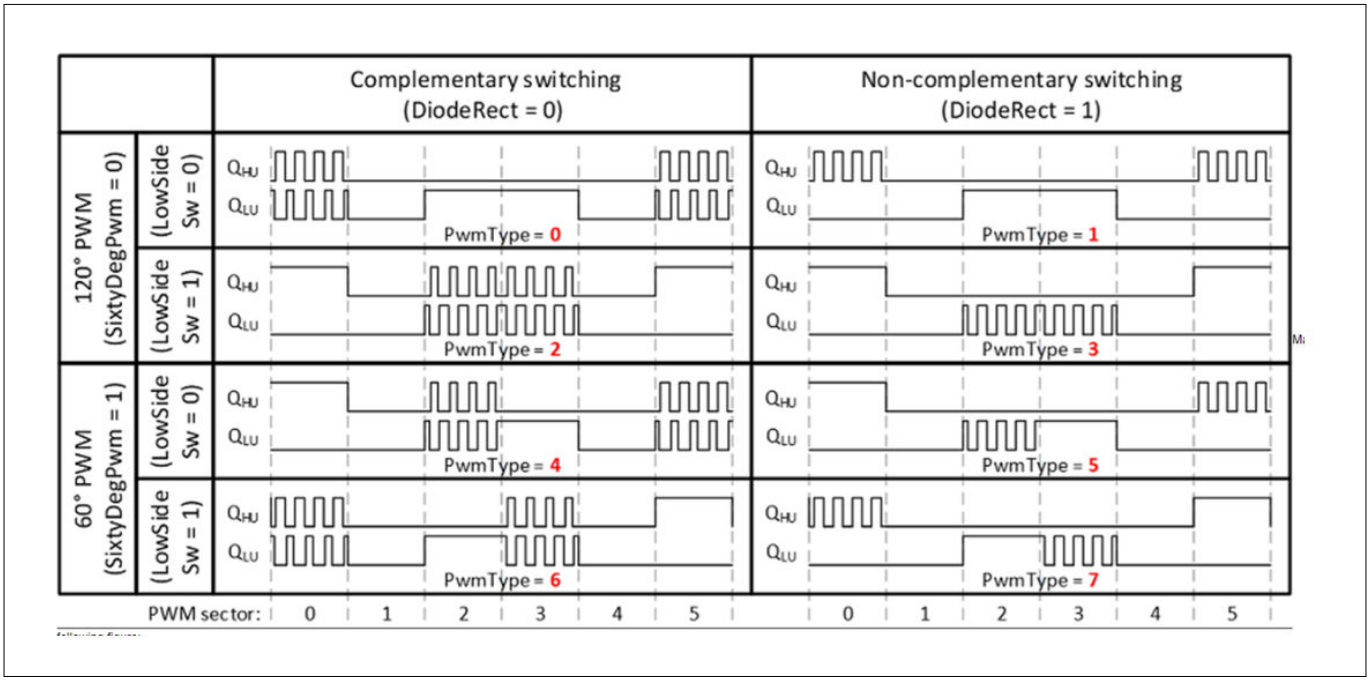


Figure 213 PWM switching modulation via MOTIF

25.5.10 Service request generation

The MOTIF has several interrupt sources that are linked to the different operation modes: hall sensor mode, quadrature decoder mode, and stand-alone multi-channel mode.

The interrupt flags for the hall sensor mode are described in Hall sensor mode flags while the interrupt flags of the Quadrature Decoder Mode are described in Quadrature decoder flags.

The flags associated with the multi-channel function are available in all the modes. This is because the multi-channel mode can operate in parallel with the quadrature decoder mode and is needed whenever the hall sensor mode is activated.

Each of the interrupt sources, can be routed to the MOTIFx.SR0 or MOTIFx.SR1 output, depending on the value programmed in the PFLGE register.

25.5.10.1 Hall sensor mode flags

The Hall Sensor Control contains six flags that can be configured to generate an interrupt request pulse, see Figure 214.

The six interrupt sources are:

- transition at the Hall Inputs (PFLG.HIES)
- occurrence of a correct hall event (PFLG.CHES)
- occurrence of a wrong hall event (PFLG.WHES)
- occurrence of a current pattern match event (PFLG.CPES)
- occurrence of a previous pattern match event (PFLG.PPES)
- shadow transfer of the Multi-Channel pattern (PFLG.MSTS)

The last one is triggered every time the Multi-Channel pattern is updated (PIFMST), which means that the MOTIFx.MOUT[31:0] output was updated with a new value.

Each of this interrupt sources can be enabled/disabled individually. The SW also has the possibility to set and clear the specific flags by writing a 1B into the specific fields of SPFLG and RPFLG registers. By enabling an interrupt source, an interrupt pulse is generated every time that a flag set operation occurs, independently if the flag is already set or not.

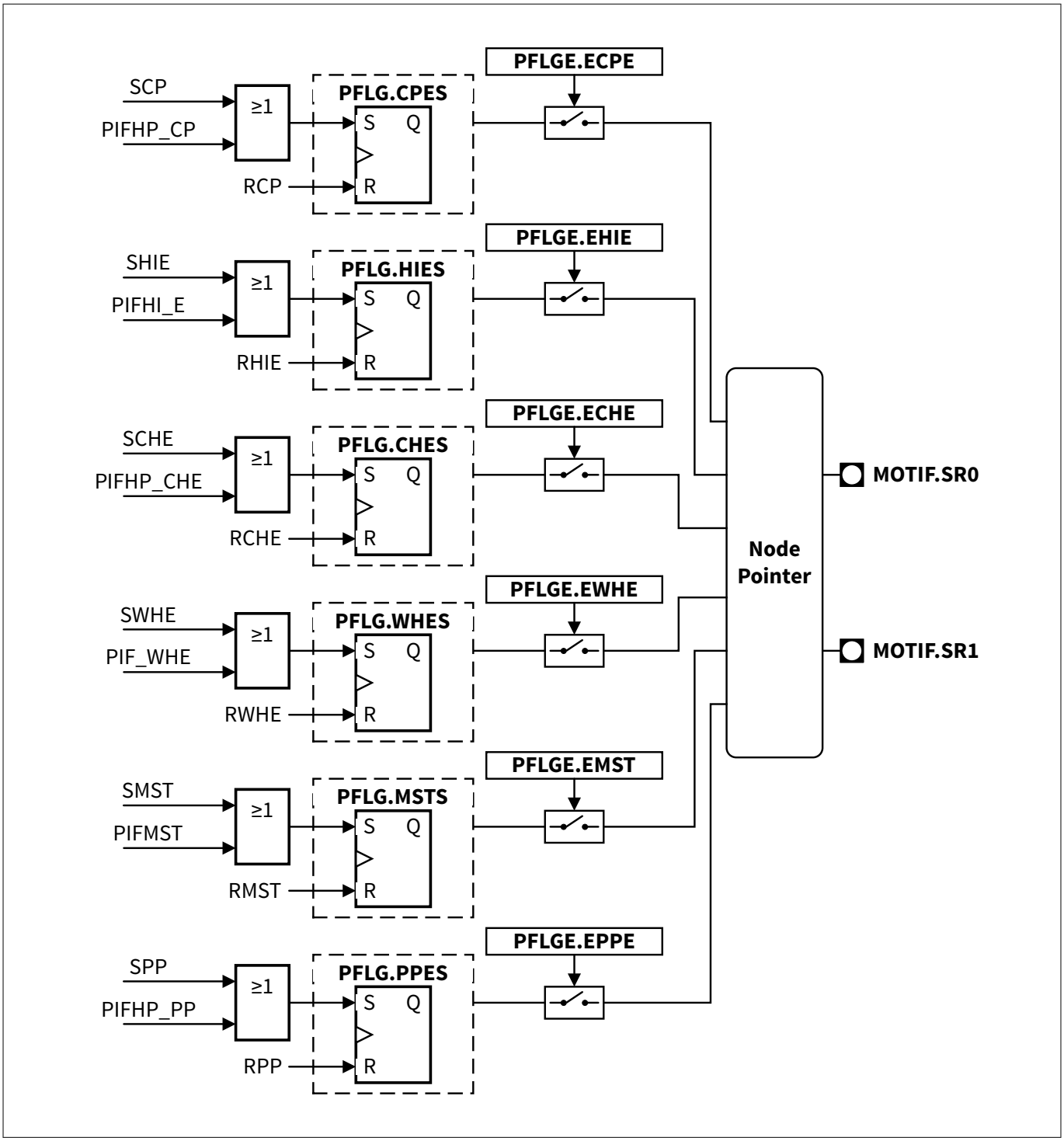


Figure 214 Hall sensor mode flags

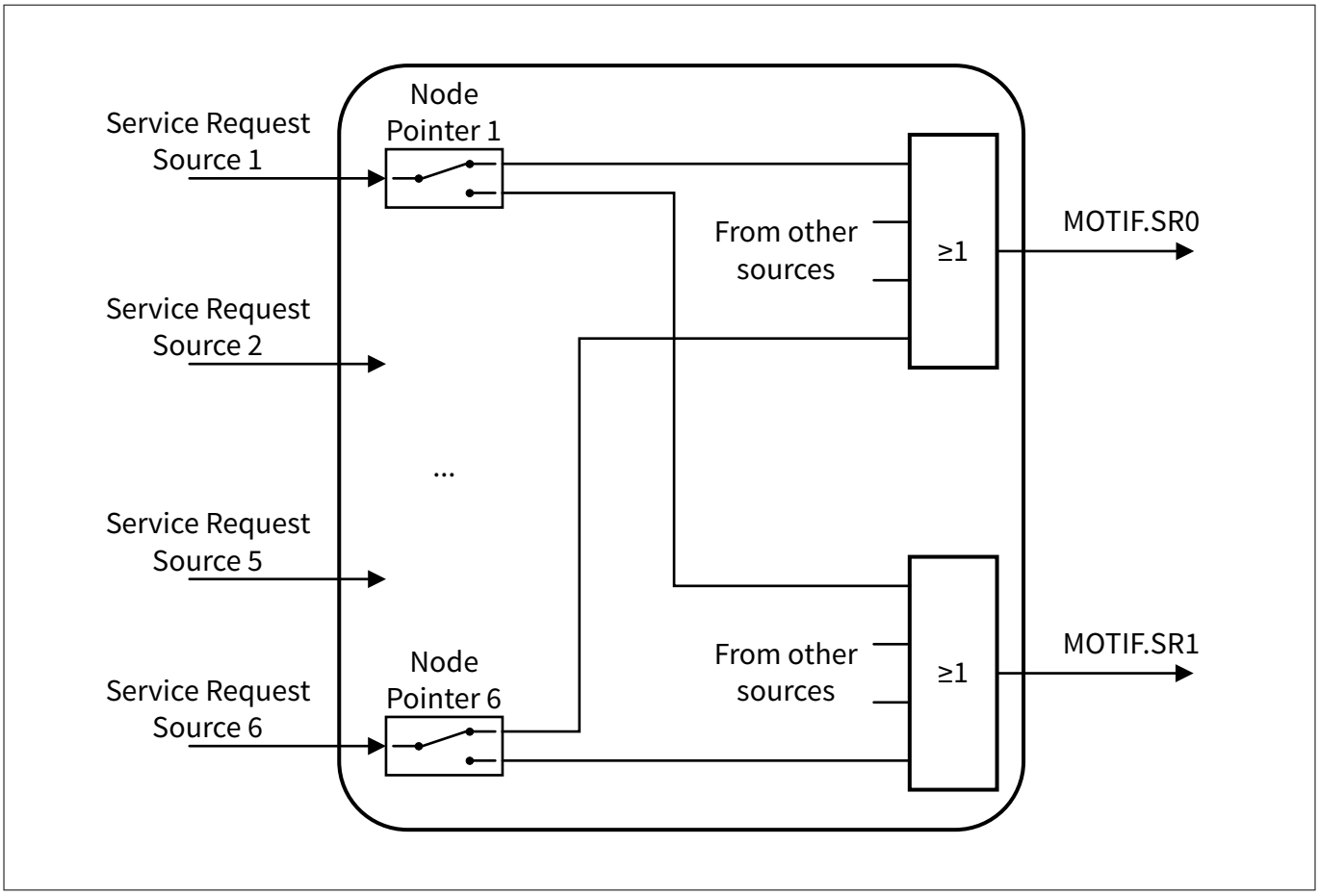


Figure 215 Interrupt node pointer overview - hall sensor mode

25.5.10.2 Quadrature decoder flags

The Quadrature Decoder Mode has five flags that can be enabled individually as interrupt sources (besides these five flags, the ones that are linked to the usage of Multi-Channel mode are also available: Multi-Channel Pattern update (PFLG.MSTS) and Wrong Hall event (PFLG.WHES). This can be useful if one selects the Quadrature Mode and the Multi-Channel stand-alone functionality.

These five flags are:

- index event detection (PFLG.INDXS)
- phase detection error (PFLG.ERRS)
- quadrature clock generation (PFLG.CNTS)
- period clock generation (PFLG.PCLKS)
- direction change (PFLG.DIRS)

By enabling an interrupt source, an interrupt pulse is generated every time that a flag set operation occurs, independently if the flag is already set or not.

The index event detection flag is set every time that an index is detected.

The phase error flag is set when one invalid transition on the phase signals is detected, see Figure 197

The quadrature clock and period clock flags are generated accordingly with the timings shown in Figure 198

The direction change flag is set, every time that an inversion of the motor rotation happens.

Each flag can be set/cleared individually by SW, by writing into the specific field on the registers SPFLG and RPFLG, see Figure 216

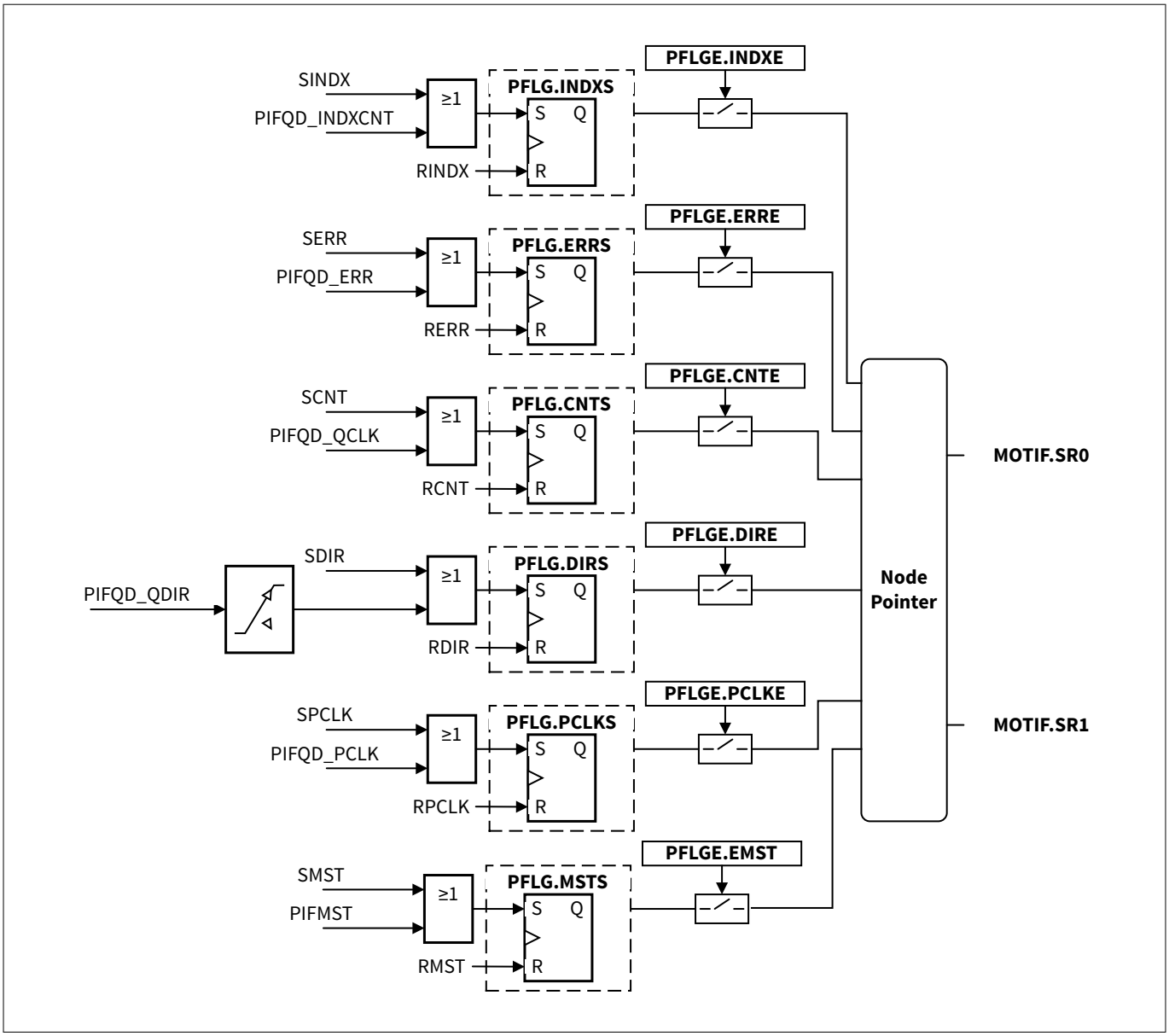


Figure 216 Quadrature decoder flags

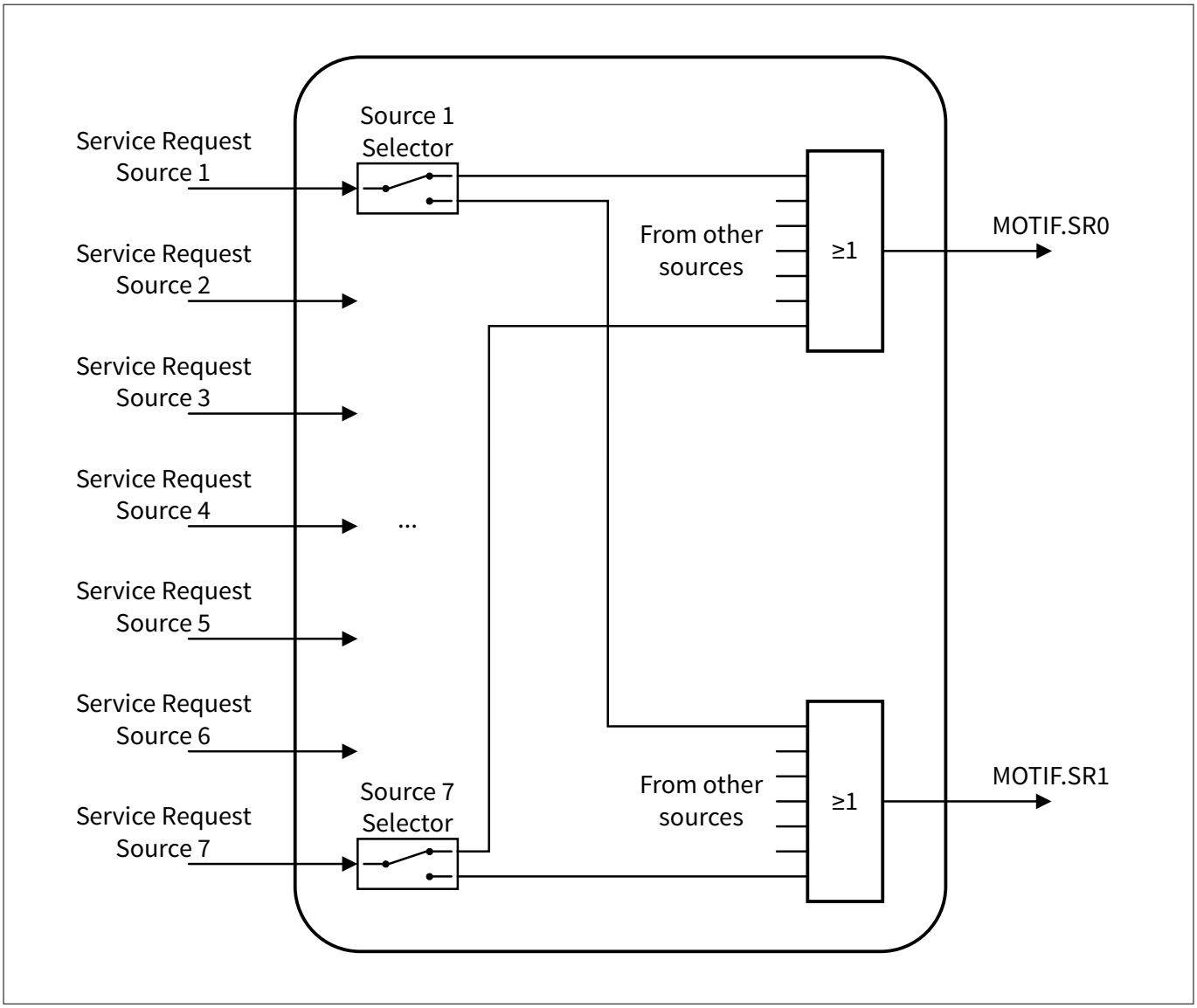


Figure 217 Interrupt node pointer overview - quadrature decoder mode

25.5.11 MOTIF debug behavior

The TCPWM MOTIF supports debugging. It can be configured per MOTIF if the MOTIF operation continues or pauses in a debug state (for example, after running to a break point), which can be realized by gating the counter clock “clk_motif” when entering the debug state defined by PSUS.SUS.

The MOTIF registers can still be accessed by the CPU. This mode is useful for debugging purposes, for example, where the current device status should be frozen to get a snapshot of the internal values. In debug mode, the entire MOTIF is halted. The debug mode is non-intrusive concerning the register bits. This means register bits are not modified by hardware when entering or leaving the suspended mode. Entry into debug mode can be configured at the kernel level by means of the register PSUS. The module is only functional after the debug signal becomes inactive.

The TCPWM has an input “tr_debug_freeze” which signalizes entering the debug state. It is connected via a PERI trigger multiplexer to the 2 cross-trigger interface outputs of the CPUSS. Depending on which CPU shall be debugged, the debug state output trigger of that CPU can be selected as “tr_debug_freeze” input of the TCPWM. Since this signal is routed through the trigger multiplexers, it needs to be synchronized (as all trigger signals are considered to be asynchronous due to potential long wires). For these MOTIF which are controlled

by the CPU to be debugged and which should be stopped when entering the debug mode defined by PSUS.SUS register.

***Note**: The MOTIF input trigger synchronizer logic also works with "clk_motif". During debug state input triggers are not synchronized.*

25.5.12 Initialization and System dependencies

25.5.12.1 Initialization

The initialization sequence for an application that is using the MOTIF should be the following:

1. Apply system reset
2. Release system reset
3. Enable the MOTIF via register PCONF.ENABLE=1

   ***Note**: Once enabled, the reset "rst_motif_act_n" gets released and motif operation logic receives "clk_motif". The LPF is active and running, so MOTIF inputs tr_motif_in[0], tr_motif_in[1], tr_motif_in[2] are filtered out if the filter is enabled.*

4. Configure MOTIF registers via AHB access
5. Read PDBG.LPFxV until tr_motif_in[0], tr_motif_in[1], tr_motif_in[2] inputs are equal to one
6. Start MOTIF via PRUNS.SRB=1

The internal state machines shall go out of idle state and run into operation mode.

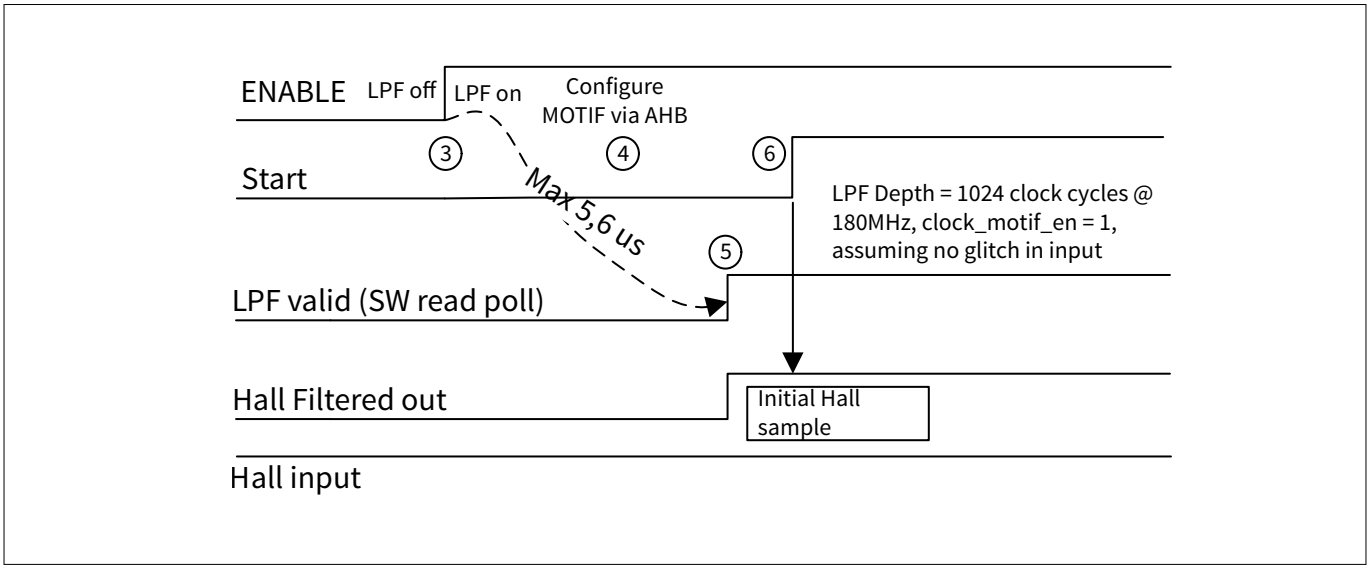


Figure 218 Initialization sequence for Hall mode

The initialization sequence is the same for hall mode and quad mode. In stand-alone multichannel mode, 5. can be omitted as LPF is not used.

Stop MOTIF is done via PRUNS.SRB=0, at this time, the state machines shall go to idle state, output events gets reset to default value, interrupt output get reset to default value and modulation output keep the existing value.

Disable MOTIF is done via PCONF.ENABLE=0, at this time, all non-retention registers get reset to the default value "rst_motif_act_n" is active. The MOTIF operational logic will not receive a clock, "clk_motif" is gated. The

MOTIF trigger synchronization will not receive a clock, “clk_peri_motif” is gated. The MOTIF MMIO registers get an active clock, “clk_ahb_motif”.

25.5.12.2 System dependencies

MOTIF may have different dependencies regarding module and bus clock frequencies. These dependencies should be addressed in the device clock unit and system architecture chapters.

Dependencies between several peripherals, regarding different clock operating frequencies may also exist. This should be addressed before configuring the connectivity between the MOTIF and some other peripheral.

The following topics must be taken into consideration for good MOTIF and system operation:

- Module input triggers for the MOTIF must not exceed the module clock frequency (if the triggers are generated internally in the device)
- Frequency of the MOTIF outputs used as triggers/functions on other modules must be crosschecked on the end point
- Applying and removing MOTIF from reset can cause unwanted operations in other modules. This can occur if the modules are using MOTIF outputs as triggers/functions

25.6 High-resolution PWM (HRPWM)

25.6.1 Overview

The high-resolution PWM (HRPWM) generation feature is supported in PWM and PWM_DT mode only. The HRPWM enhancement allows higher resolution control for power conversion application. The high-resolution generation support is only applicable to Group 0.

25.6.1.1 Features

- Supports HRPWM in PWM and PWM_DT mode only
- Supports HRPWM feature in 32-bit counter
- HRPWM feature is available only in the GROUP #0
- Supports period, duty cycle, and dead time with integer and factional features
  - Existing PERIOD, PERIOD_BUFF, CC0, CC0_BUFF, CC1, CC1_BUFF, DT, and DT_BUFF registers will be reused
  - PERIOD, CC0, CC1, DT registers are used for integer and fractional feature purposes
  - PERIOD_BUFF, CC0_BUFF, CC1_BUFF, DT_BUFF registers are used for shadow update purpose
- Supports 6 fractional bites for delay adjustment
- Supports the following PWM output modes: “right align, left align, center align, asymmetric with cc0, asymmetric with cc0 & cc1 and center align asymmetric with cc0 & cc1” for HRPWM feature

***Note**: For each TCPWM counter with HRPWM functionality; the correct order to disable the counter is:*

- *1) Disable the HRPWM function by setting HRPWM_EN =’0’*
- *2) Disable the counter by setting ENABLED =’0’*

***Note**: When the HRPWM feature is enabled, ensure that **PERIOD – CC0 >=3**. If **PERIOD - CC0 <= 2**, no CC0_match event will be generated, resulting in an abnormal PWM waveform output when operating in center-aligned PWM mode with UPDWN enabled.*

25.6.1.2 Block diagram

The Figure 219 shows an example of the connection of the HRPWM for a counter group.

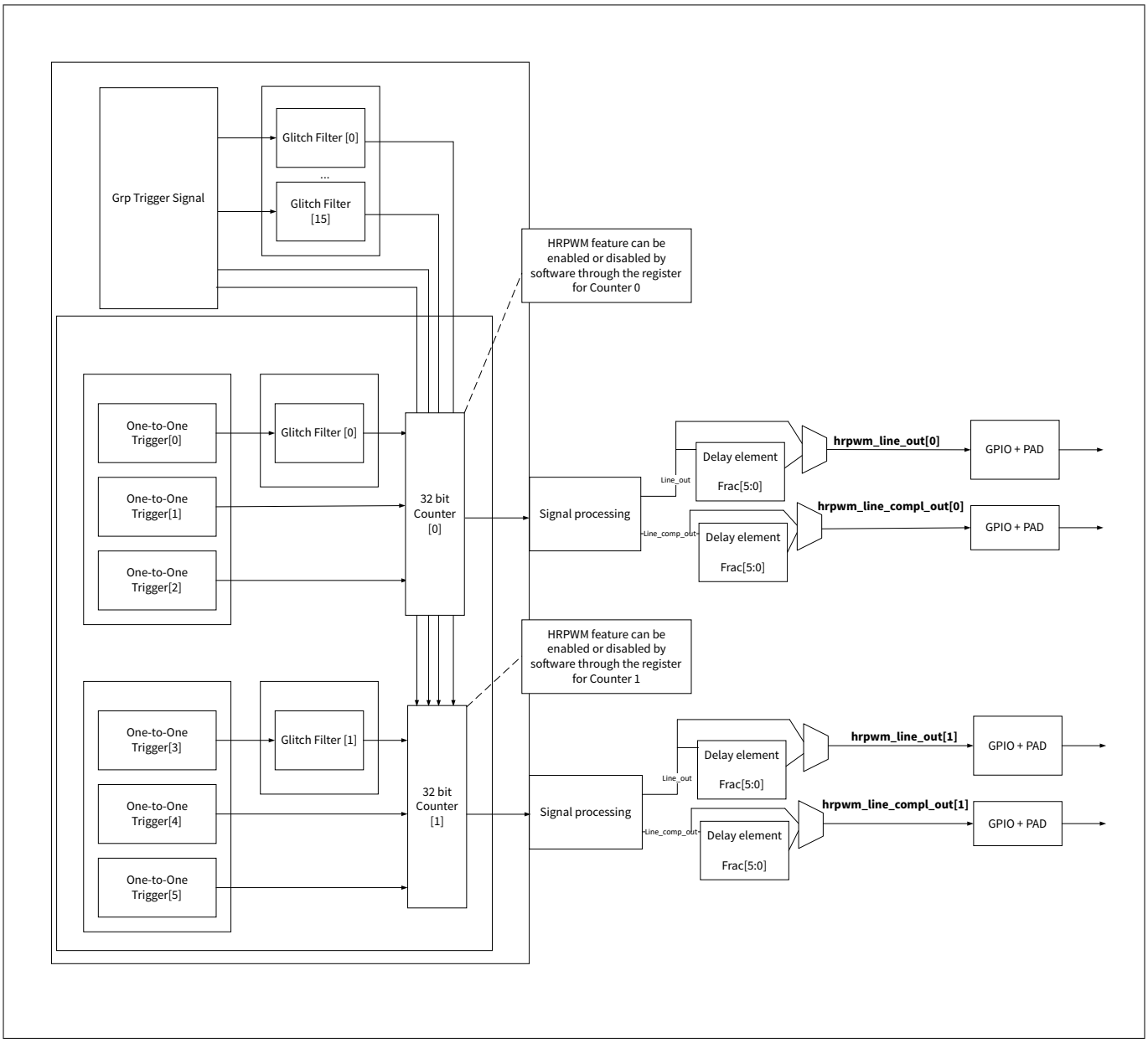


Figure 219 HRPWM block diagram

25.6.1.3 Clocks

Clocking has the following restrictions:

- When using the HRPWM feature with TCPWM, ensure that the clk_peri clock is not divided, as division is not allowed in this configuration
- To ensure proper operation, set the PS.PS_DIV to '0' when the HRPWM feature is enabled
- Configure the TCPWM to generate a clk_out signal for the delay element, which is used to generate micro ticks for shifting the edges of the PWM signal. Ensure that the clk_out signal has the same frequency as the counter clock

25.6.2 PWM

In the PWM mode, when the HRPWM feature is enabled, the LSBs are used for fractional or micro ticks for delaying the line_out and line_compl_out outputs for the PERIOD/PERIOD_BUFF, CC0/CC0_BUFF, and CC1/CC1_BUFF and DT/DT_BUFF.

Figure 220 illustrates the integer and fractional partition of the register.

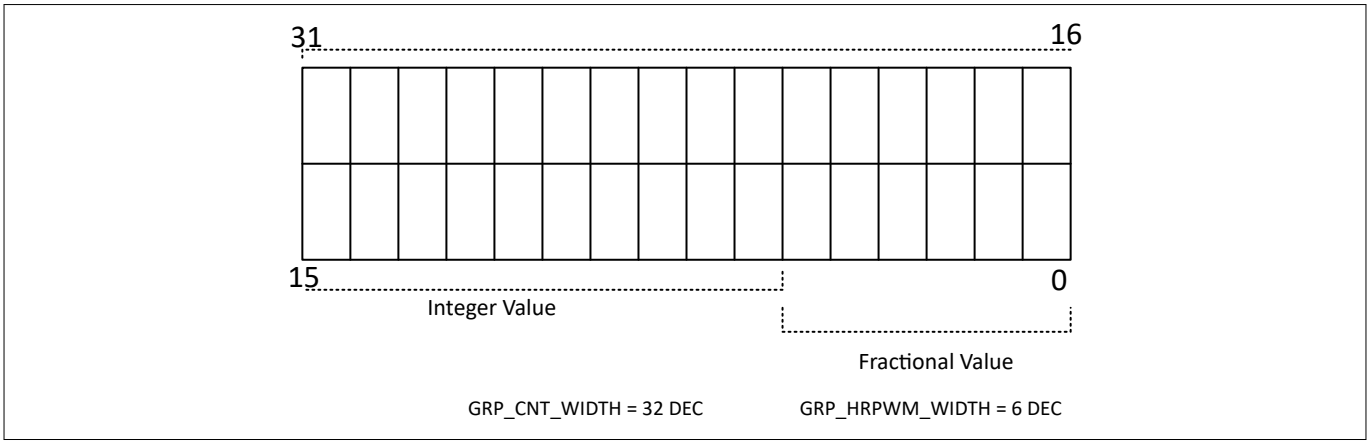


Figure 220 Example of register layout

***Note**: The maximum value for the integer part is 67108863. The maximum value for the fractional part is 63.*

Figure 221 illustrates how to program the integer and fractional value into the PERIOD register. Software will write 2374 DEC into the PERIOD register to set the 37 DEC for the integer and 6 DEC for the fractional value.

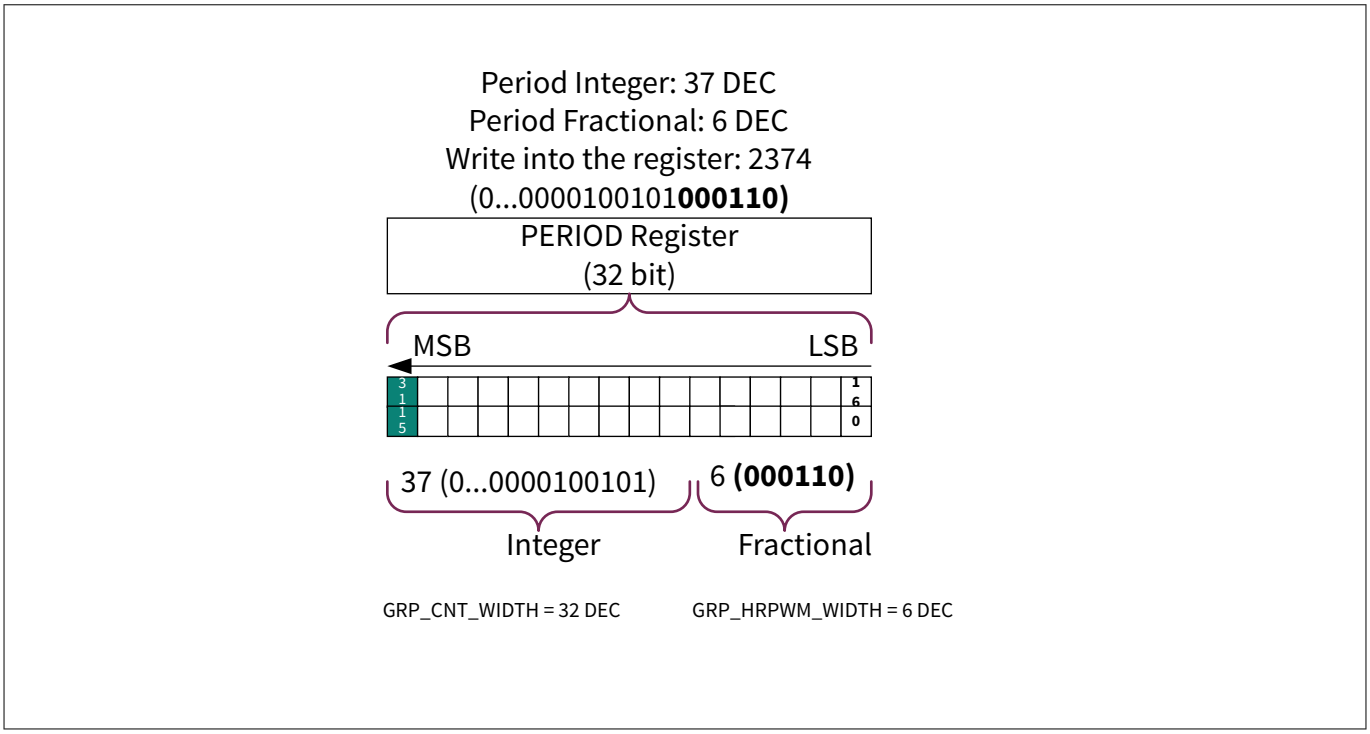


Figure 221 Example of programming integer and fractional values into the register

The fractional value programmed in the registers is used to generate data outputs:

- data_shift_line_out[5:0] – This is used by the delay element to shift the rising and falling edge of line_out by equivalent micro ticks. TCPWM generates a value in synchronous to the tc and cc0/cc1 match event for the rising and falling edge of line_out to control the micro ticks the corresponding edge must be delayed
- data_shift_line_compl_out[5:0] – This is used by the delay element to shift the rising and falling edge of line_compl_out by equivalent micro ticks. TCPWM generates a value in synchronous to the tc and cc0/cc1 match event for the rising and falling edge of line_compl_out to control the micro ticks the corresponding edge must be delayed

***Note**: For each TCPWM counter with HRPWM functionality, there are two delay elements: one for delaying the rising and falling edges of line_out and another one for delaying the rising and falling edge of the line_compl_out*

25.6.2.1 Computation concept

Figure 222 and Figure 223 shows an example of how data_shift_line_out and data_shift_line_compl_out are calculated in PWM mode and sent out with line_out and line_compl_out to the delay element.

***Note**: Fractional delay is inserted by delay element, resulting in hrpwm_line_out and hrpwm_line_compl_out out of the delay element.*

The hrpwm_* signals shown below are just for clarification and not actual outputs from TCPWM. hrpwm_line_out and hrpwm_line_compl_out are signal names used to differentiate between TCPWM output and delay element output. In the final, there will be two output signals with the name line_out and line_compl_out.

When the PERIOD register contains a fractional value >0, it is added into PF_ACC. When the PF_ACC value overflows and the carry is added into the integer value (PI) of the PERIOD register, the carry is cleared at the next TC event.

PF_ACC is also added fractional value configured into the CC0/CC1 register. When the added value CC0_ACC or CC1_ACC overflows as well, the carry is added to the corresponding CC0/CC1 register integer value for the interval of one TC event and then the carry is cleared.

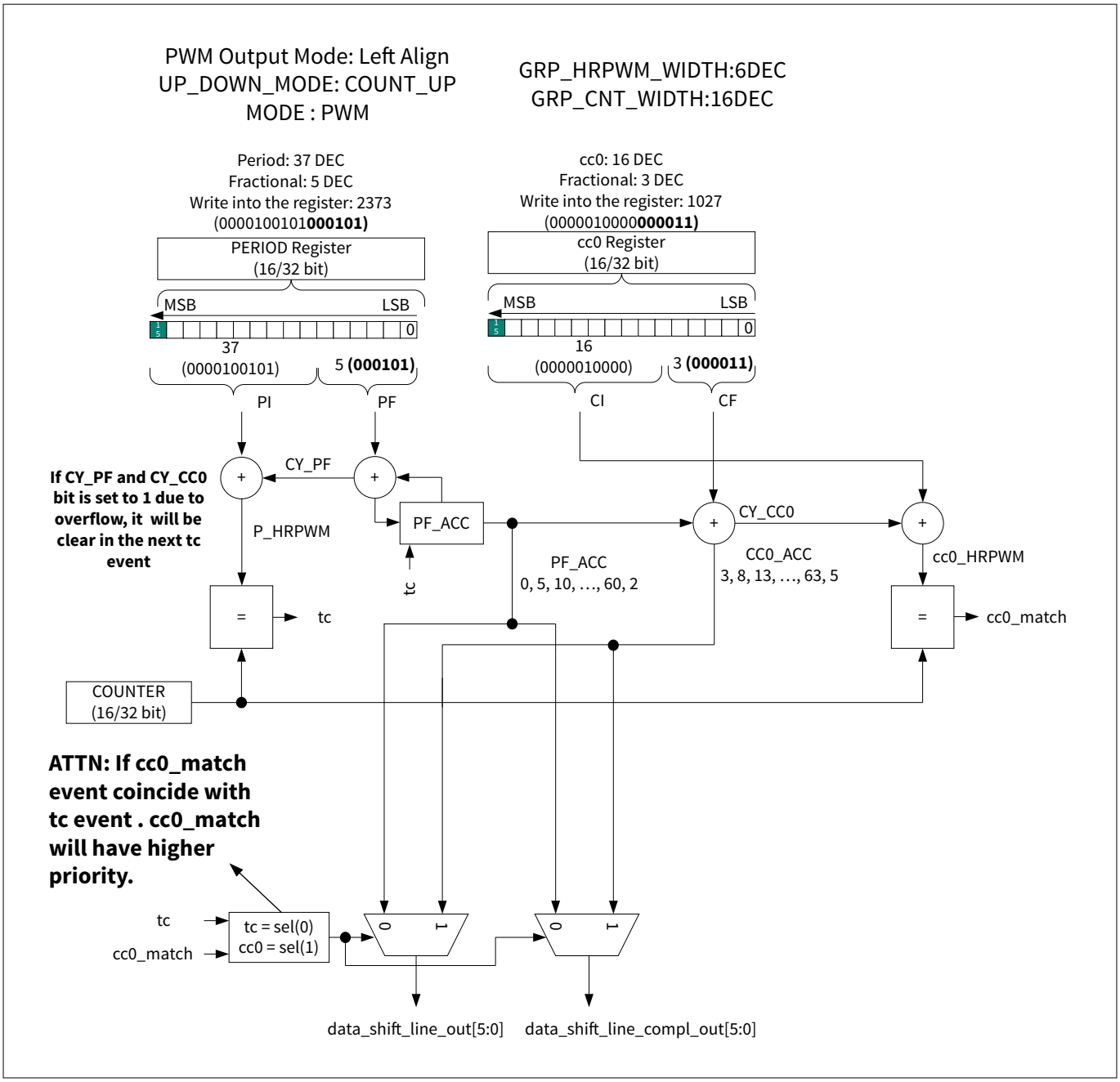


Figure 222 Calculation of data_shift_line_out and data_shift_line_compl_out in PWM mode

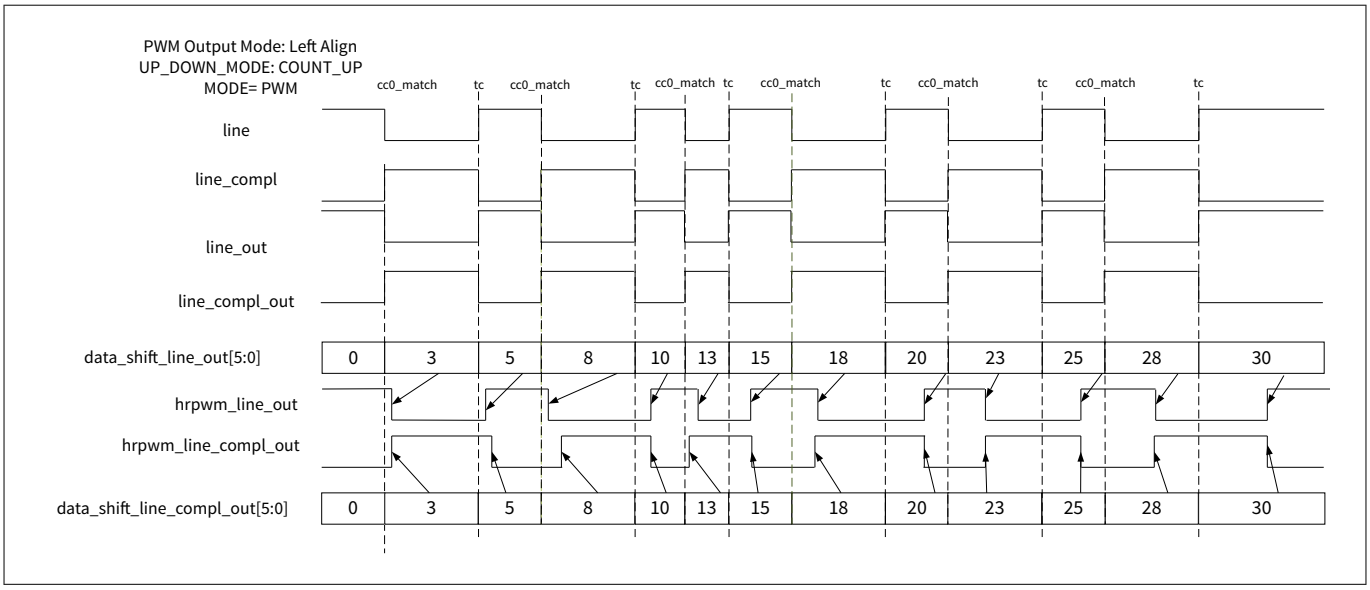


Figure 223 Example of TCPWM outputs in PWM mode

25.6.2.2 PWM output mode

The following output modes are supported in the HRPWM.

Table 226 PWM output mode in HRPWM

PWM output mode	UP_DOWN_MODE	TR_PWM_CRL	Other settings	"line" pulse width	Period
Right align	COUNT_DOWN	UNDERFLOW = CLEAR CC0_MATCH = SET	N/A	CC0+1	PERIOD + 1
Left align	COUNT_UP	OVERFLOW = SET CC0_MATCH = CLEAR	N/A	CC0	PERIOD + 1
Center align	COUNT_UPDN1	OVERFLOW = SET UNDERFLOW = CLEAR CC0_MATCH = INVERT	N/A	2xPERIOD – 2xCC0-1 (when CF=non-zero) 2xPERIOD – 2xCCO (when CF=zero)	PERIOD * 2

(table continues...)

Table 226 (continued) PWM output mode in HRPWM

PWM output mode	UP_DOWN_MODE	TR_PWM_CRL	Other settings	"line" pulse width	Period
Center align asymmetrically with cc0	COUNT_UPDN2	OVERFLOW = SET UNDERFLOW = CLEAR CC0_MATCH = INVERT	N/A	2xPERIOD – CC0 – CC0_BUFF-1 (when CF=non-zero) 2xPERIOD – CC0 – CC0_BUFF (when CF=zero)	PERIOD * 2
Asymmetric with cc0 & cc1	COUNT_DOWN COUNT_UP	CC0_MATCH = SET CC1_MATCH = CLEAR OVERFLOW = CLEAR (UP) UNDERFLOW = CLEAR (DOWN)	N/A	CC1-CC0	PERIOD + 1
Center align asymmetric with cc0 & cc1	COUNT_UPDN1 COUNT_UPDN2	OVERFLOW = SET UNDERFLOW = CLEAR CC0_MATCH = SET CC1_MATCH = CLEAR	CC0_MATCH_UP_EN = 1, CC0_MATCH_DOWN_EN = 0, CC1_MATCH_UP_EN = 0, CC1_MATCH_DOWN_EN = 1,	2xPERIOD – CC0 – CC1 – 1 (when CF=non-zero) 2xPERIOD – CC0 – CC1 (when CF=zero) (No swap at overflow in UPDN2)	PERIOD * 2

25.6.2.3 PWM output mode - Left align

The following diagrams illustrate a few examples with different CF setup of PWM output mode configured in left align with HRPWM feature enabled.

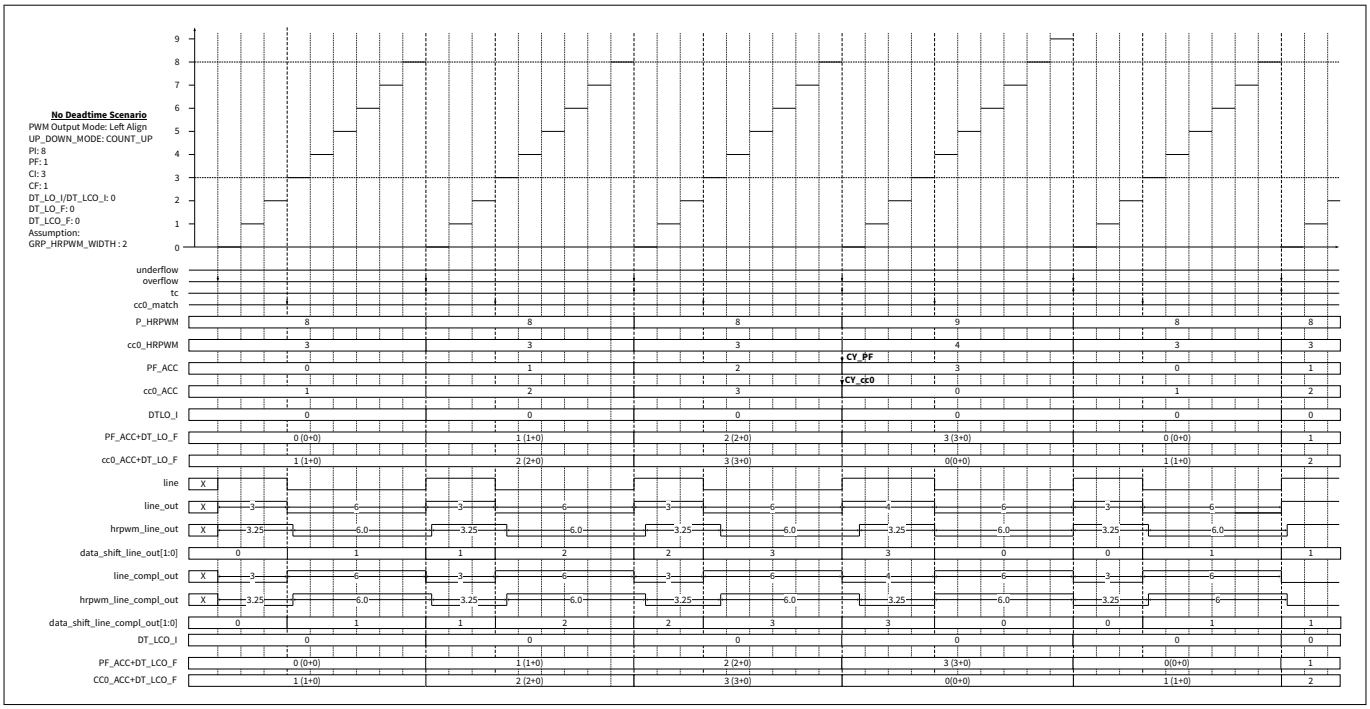


Figure 224 Example of Left Align in PWM mode with HRPWM Feature enabled - Scenario 1 (CF=1)

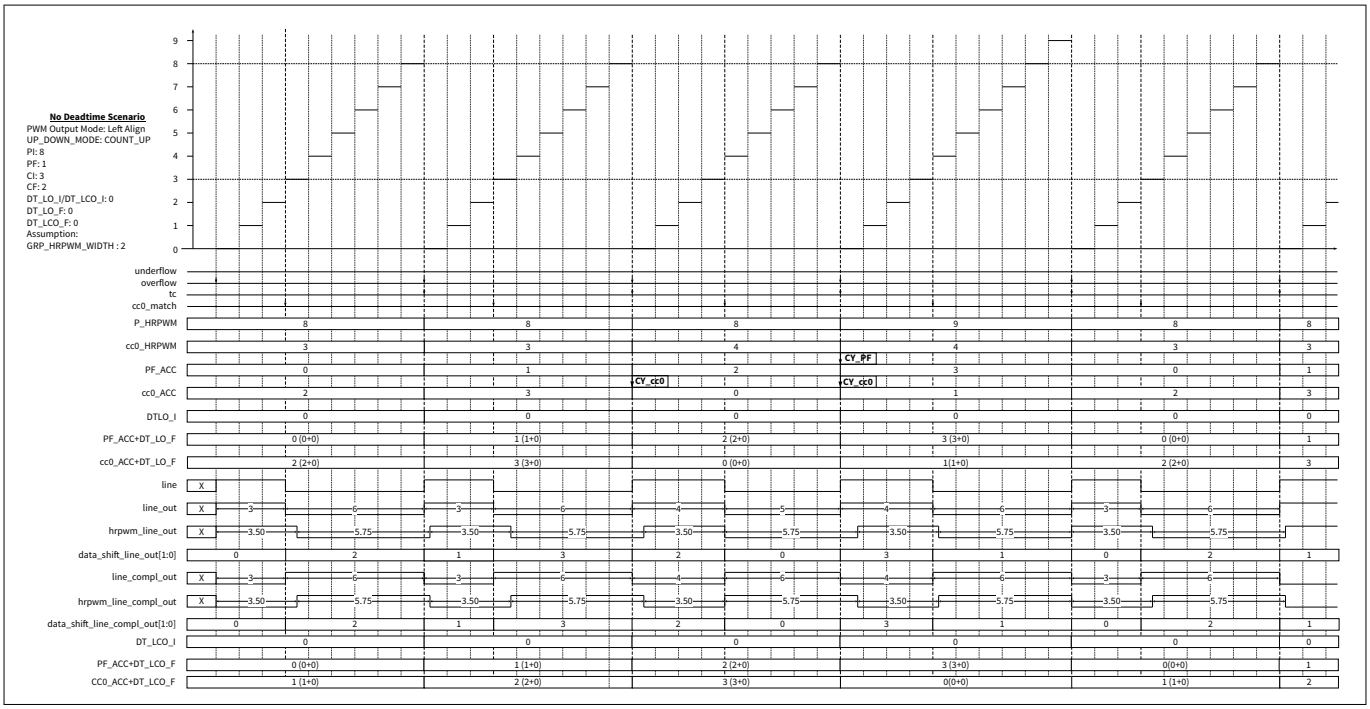


Figure 225 Example of Left Align in PWM mode with HRPWM Feature enabled - Scenario 2 (CF=2)

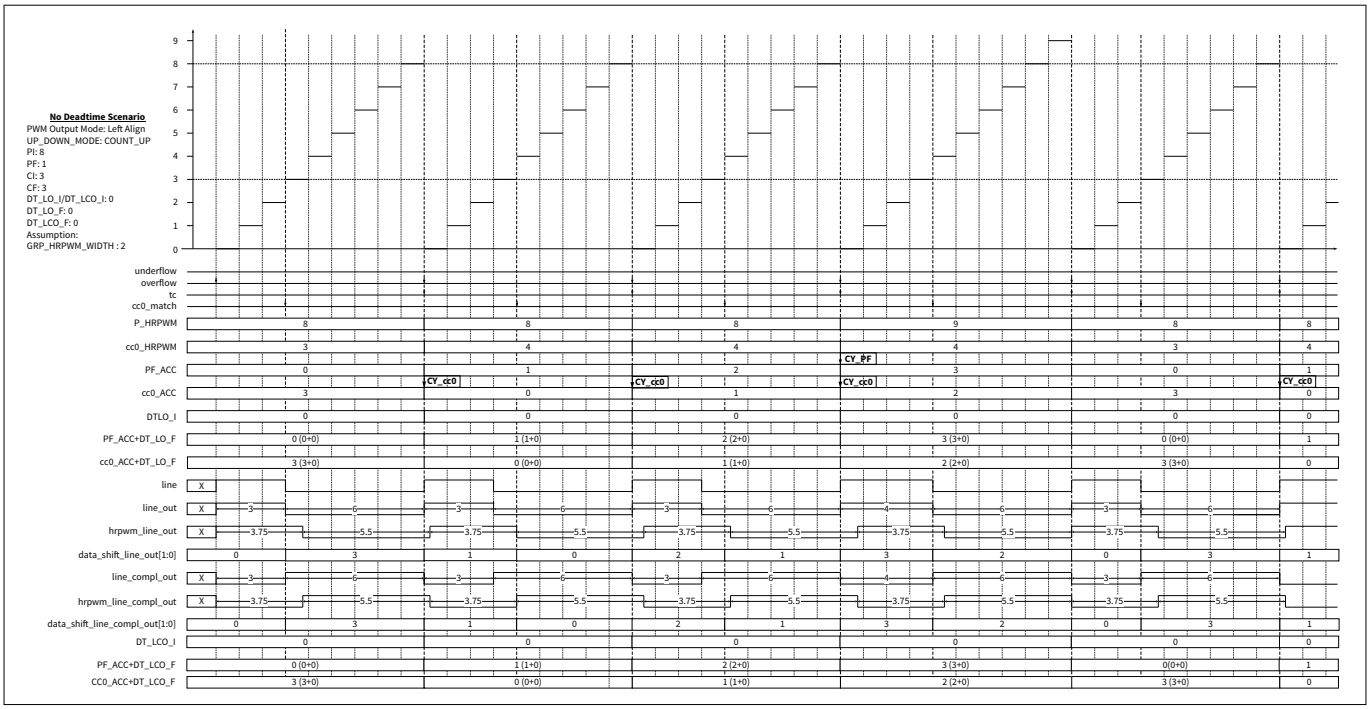


Figure 226 Example of Left Align in PWM mode with HRPWM Feature enabled - Scenario 3 (CF=3)

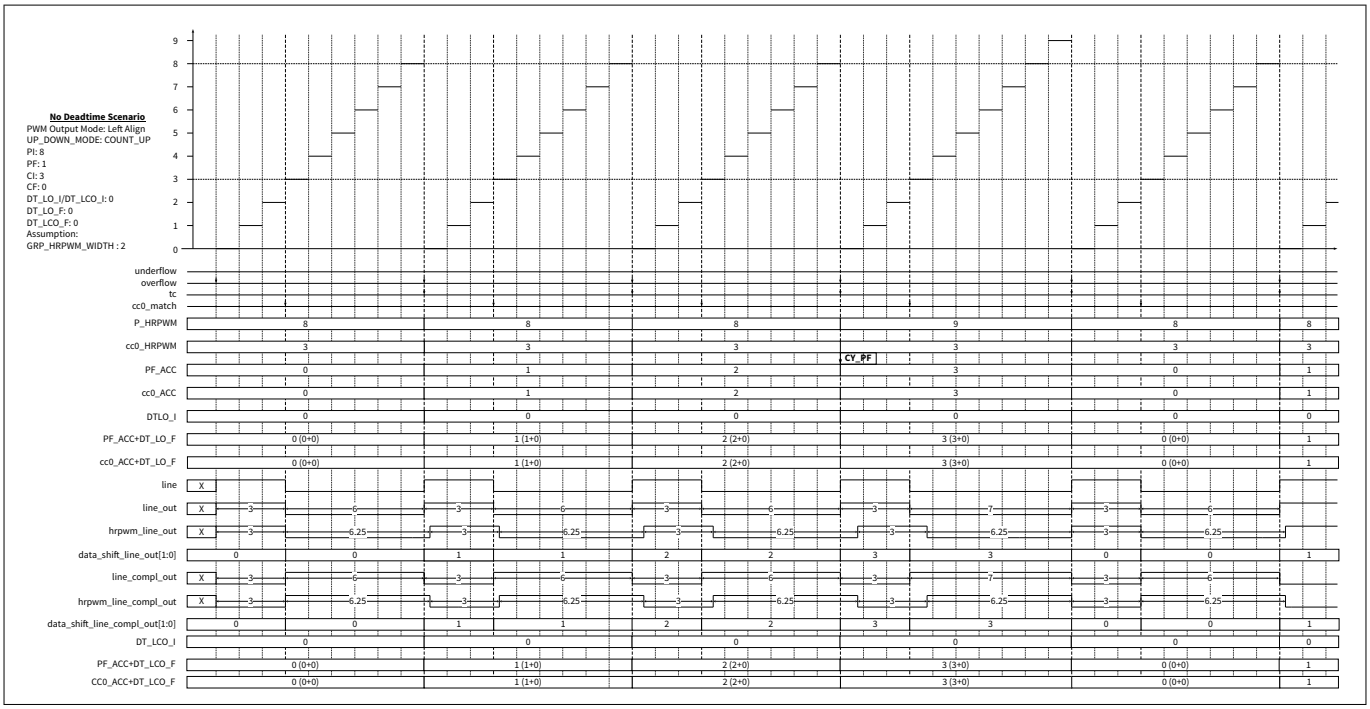


Figure 227 Example of Left Align in PWM mode with HRPWM Feature enabled - Scenario 4 (CF=0)

25.6.2.4 PWM output mode - Right align

For the right align case, cc0_ACC and cc0_HRPWM computation are as followed:

- signed_cc0_ACC = PF_ACC + PF – CF, if signed_cc0_ACC is negative, cc0_ACC will be two complements
- msb_cc0_acc will be set to 1 when signed_cc0_ACC value is greater equal than 2^GRP_HRPWM_WIDTH

- sn_cc0_acc will be set to 1 when signed_cc0_ACC is negative
- cc0_hrpwm = CI + CY_PF + sn_cc0_acc – signed_cy_cc0, where signed_cy_cc0 = msb_cc0_acc XOR sn_cc0_acc

The Table 227 shows a few examples of the computation for GRP_HRPWM_WIDTH set to 2:

Table 227 Examples of computation for right align PWM output mode

PF_ACC	PF	CF	CI	signed_cc0_ACC (PF_ACC+ PF-CF)	msb_cc0_acc	sn_cc0_acc	signed_cy_cc0	cc0_hrpwm
3	3	1	5	5 (101B)	1	0	1	4 (CY_PF= 0)
1	1	3	5	-1 (111B)	1	1	0	6 (CY_PF=0)
1	1	2	5	0 (000B)	0	0	0	5 (CY_PF=0)

The following diagrams show a few examples with different CF setup of PWM output mode configured in right align with HRPWM feature enabled.

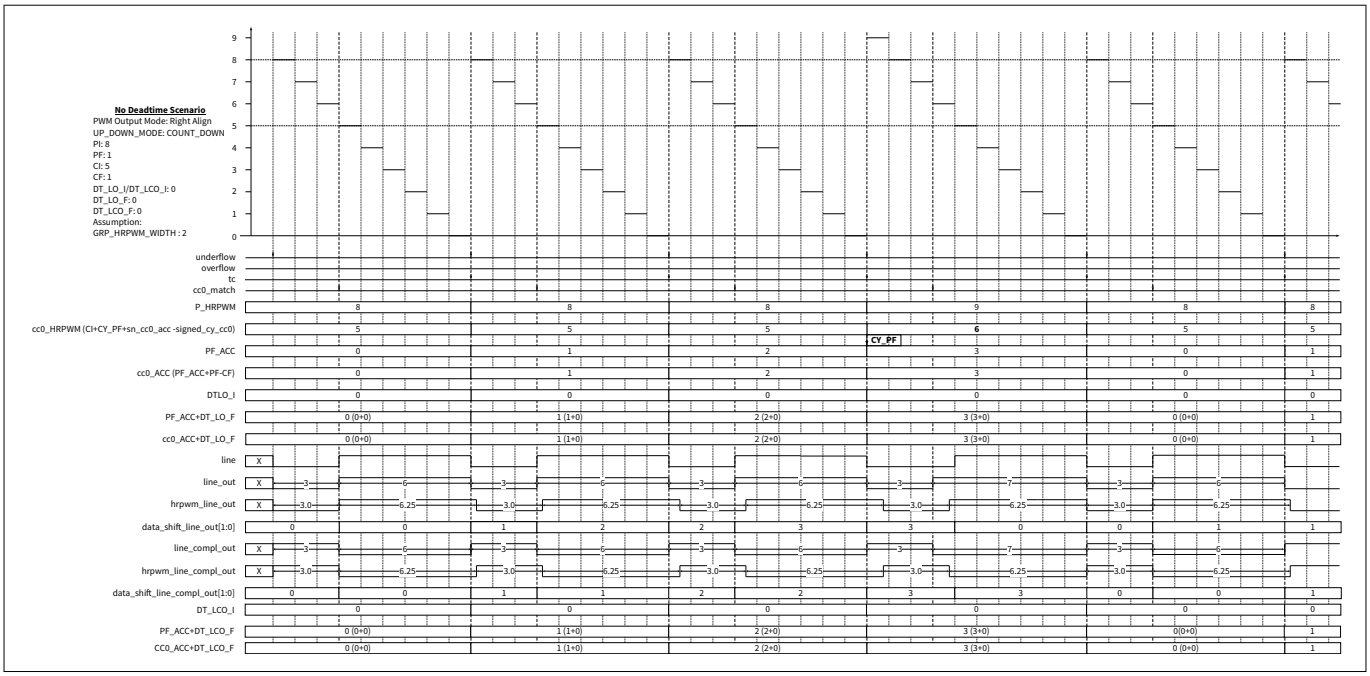


Figure 228 Example of right align in PWM mode with HRPWM feature enabled – scenario 1 (CF=1)

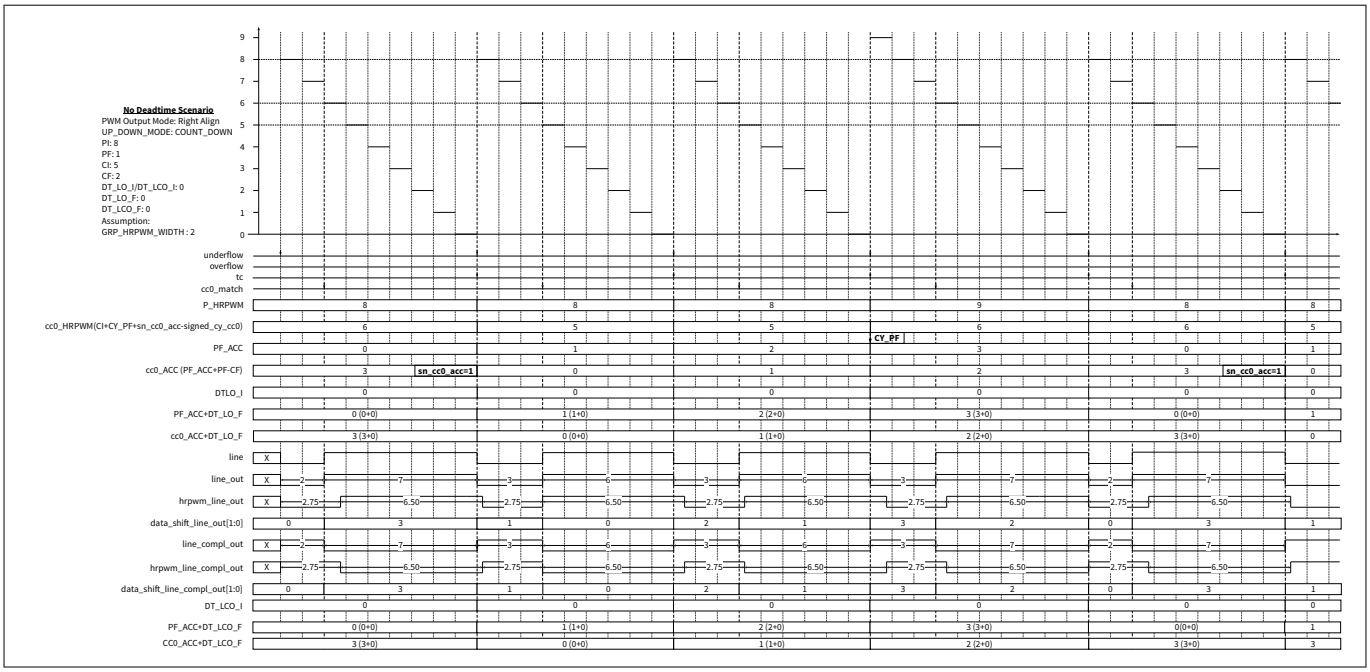


Figure 229 Example of right align in PWM mode with HRPWM feature enabled scenario 2 (CF=2)

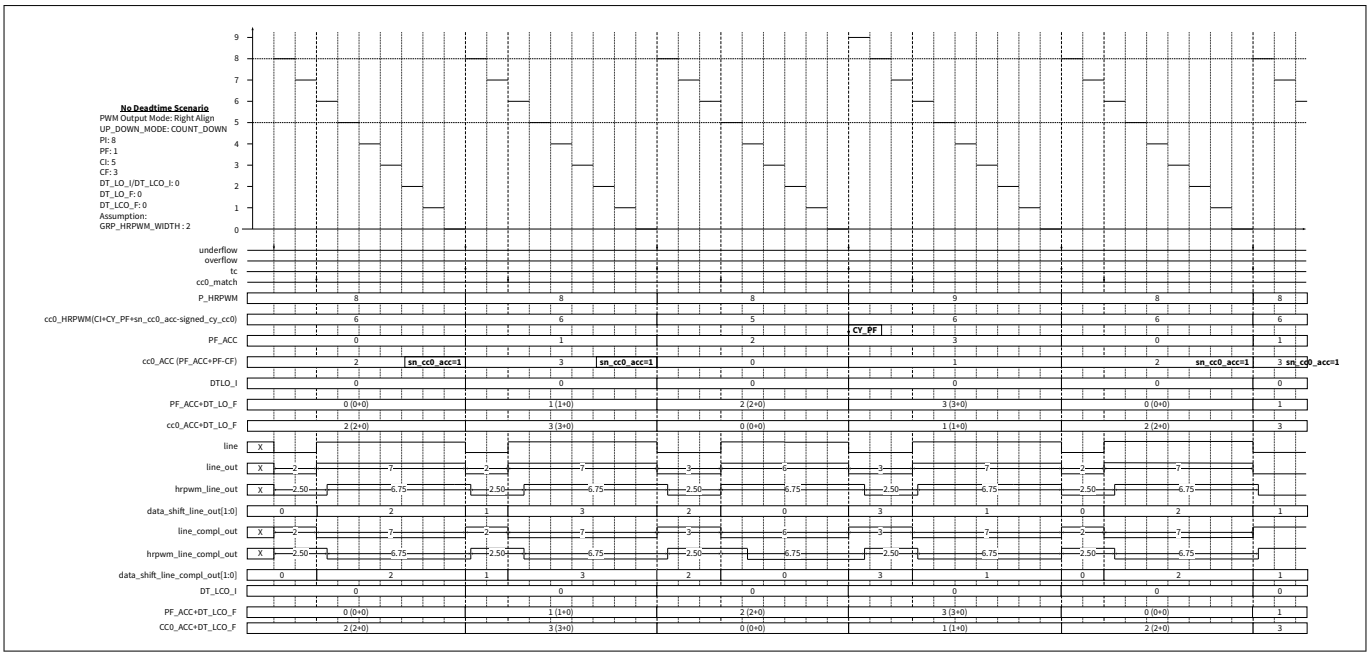


Figure 230 Example of right align in PWM mode with HRPWM feature enabled scenario 3 (CF=3)

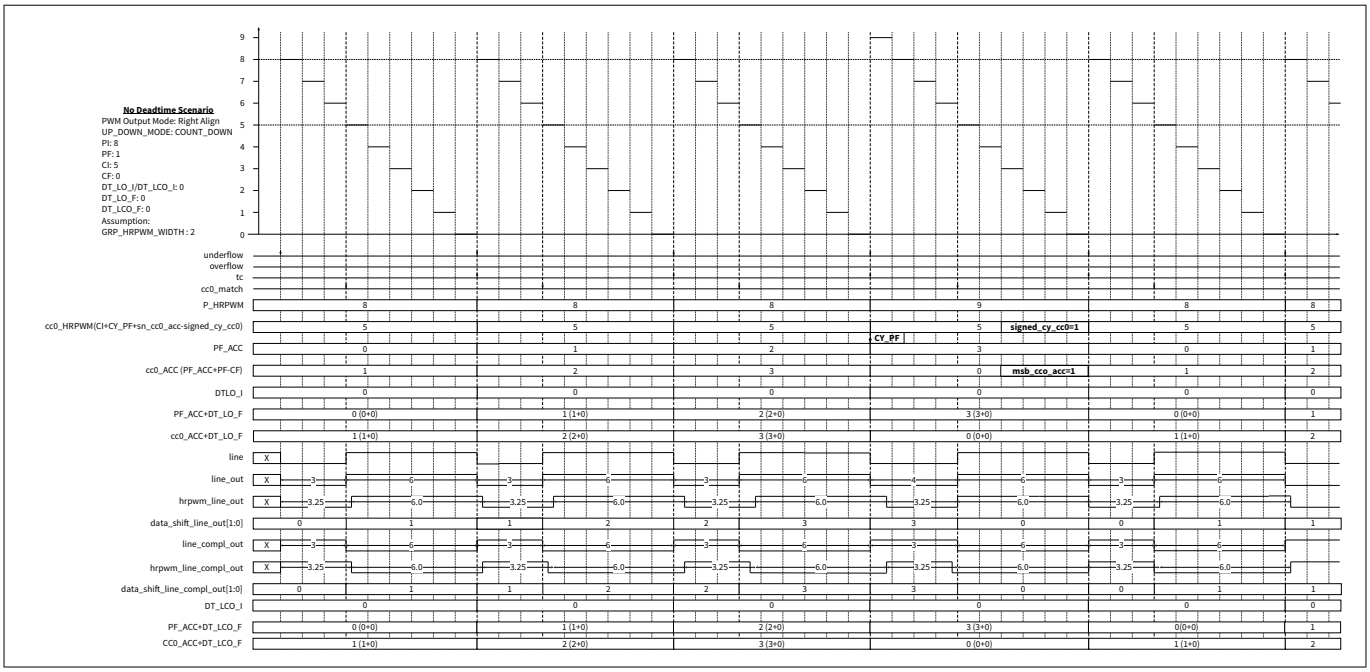


Figure 231 Example of right align in PWM mode with HRPWM feature enabled – scenario 4 (CF=0)

25.6.2.5 PWM output mode – Asymmetric with cc0 & cc1

The Figure 232 shows an example of PWM output mode configured in asymmetrical with cc0 & cc1 with HRPWM feature enabled.

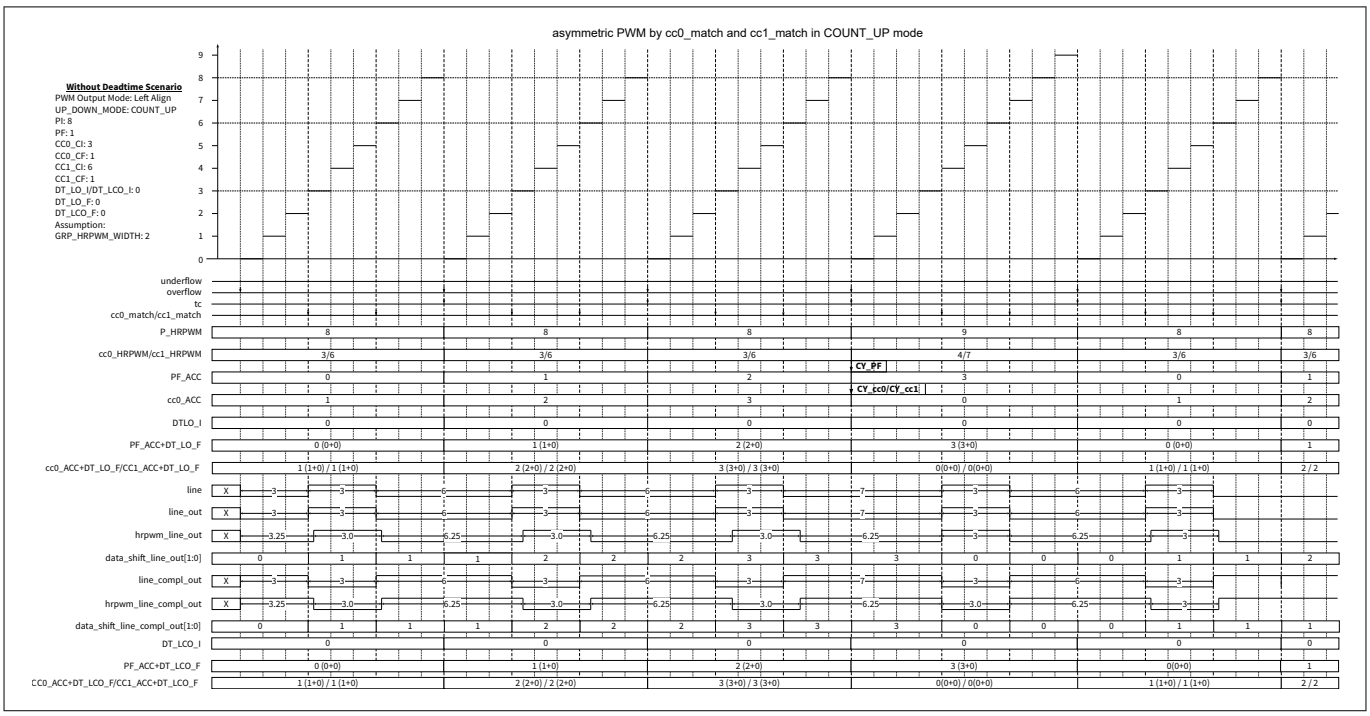


Figure 232 Example of asymmetric with cc0 & cc1 in PWM Mode with HRPWM feature enabled

25.6.2.6 PWM output mode – Center align

The Figure 233 shows the expectation of HRPWM for center align case. It shows the handling of the dead time and its fraction component in addition to the overall fractional shift value that will be forwarded to the delay elements.

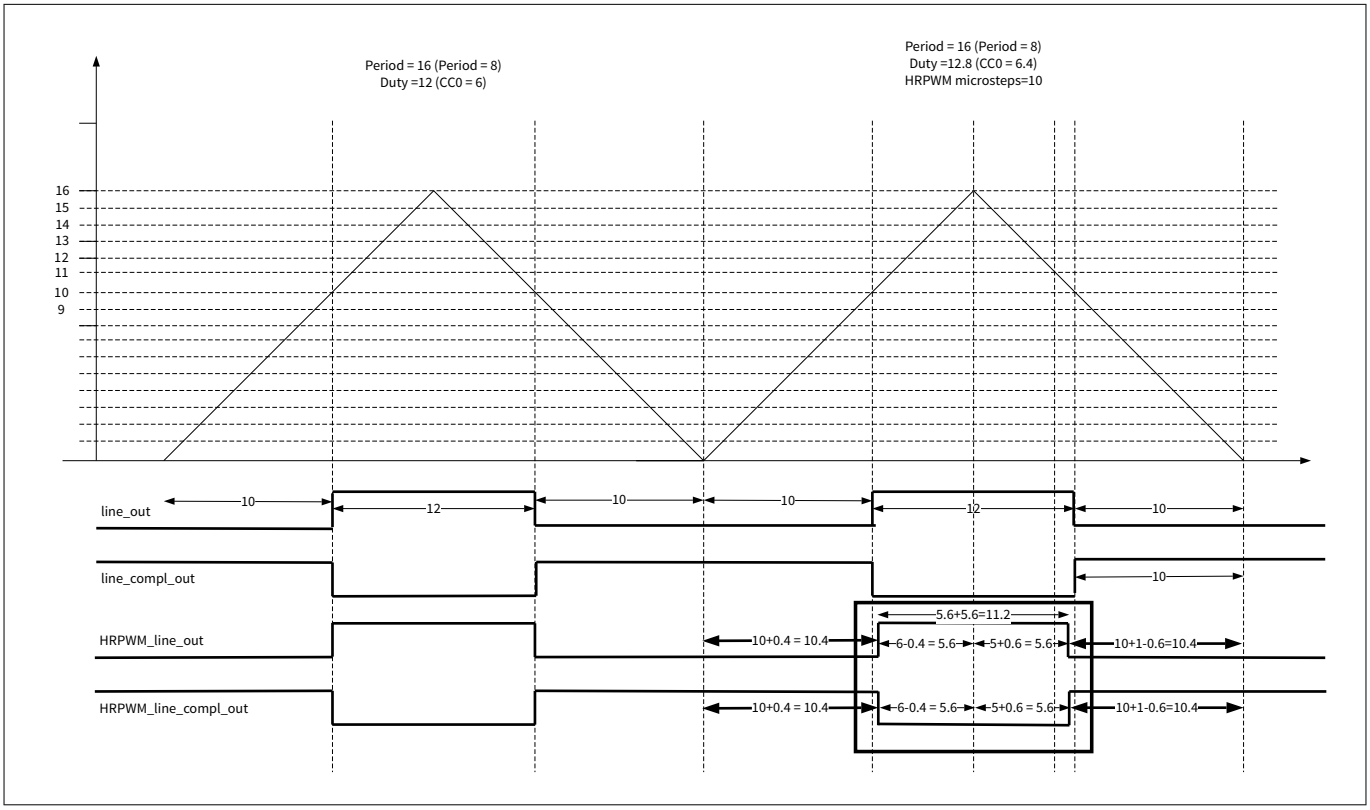


Figure 233 Center align case HRPWM expectation

The following diagrams show a few examples with different CF setup of PWM output mode configured in center align with HRPWM feature enabled. For the center align case, the design will change the value of cc0 to CI+1 upon an overflow event when CF is non-zero. The duty cycle duration will be equal for each period cycle when PF is set to zero.

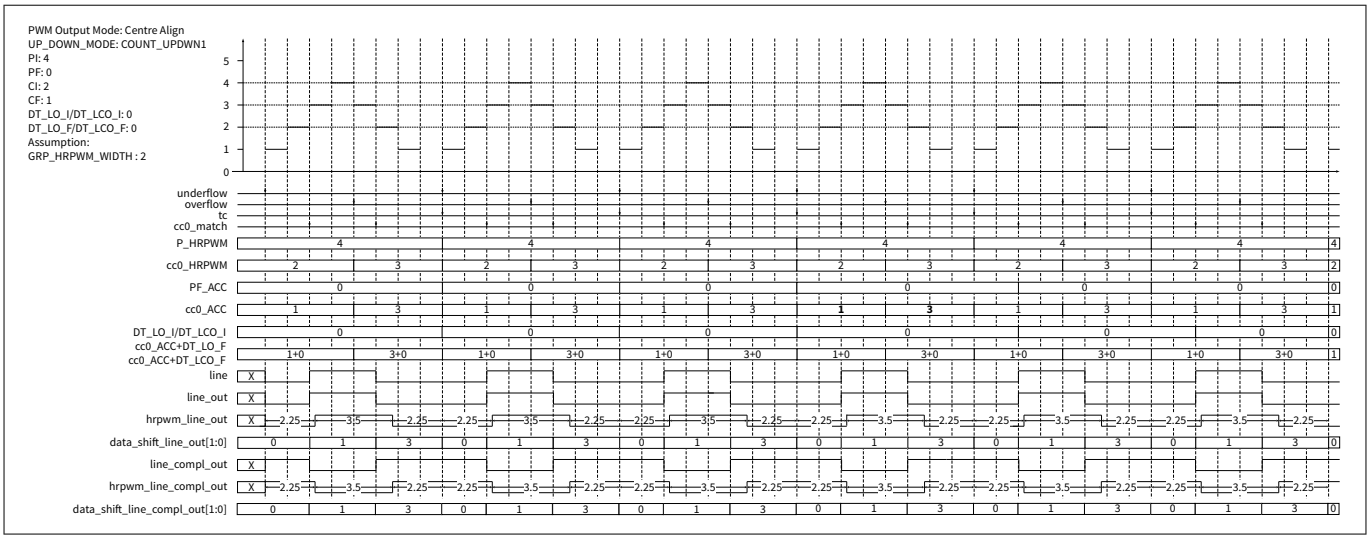


Figure 234 Example of center align – Scenario 1 (PF=0, CF=1) in PWM mode with HRPWM feature enabled

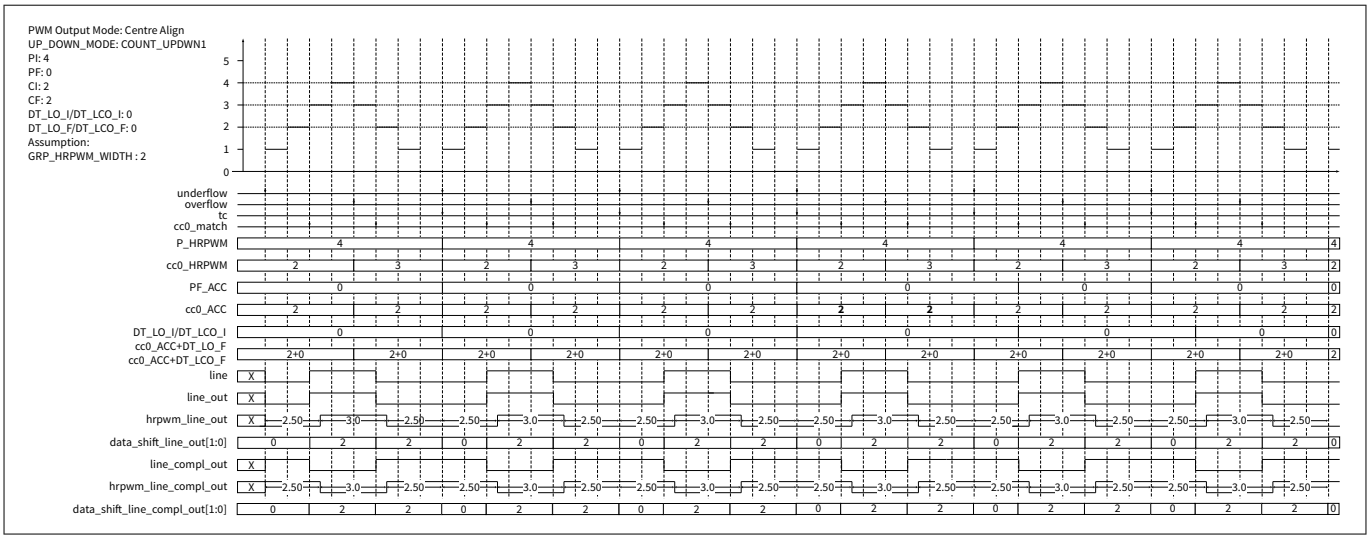


Figure 235 **Example of center align – Scenario 2 (PF=0, CF=2) in PWM mode with HRPWM feature enabled**

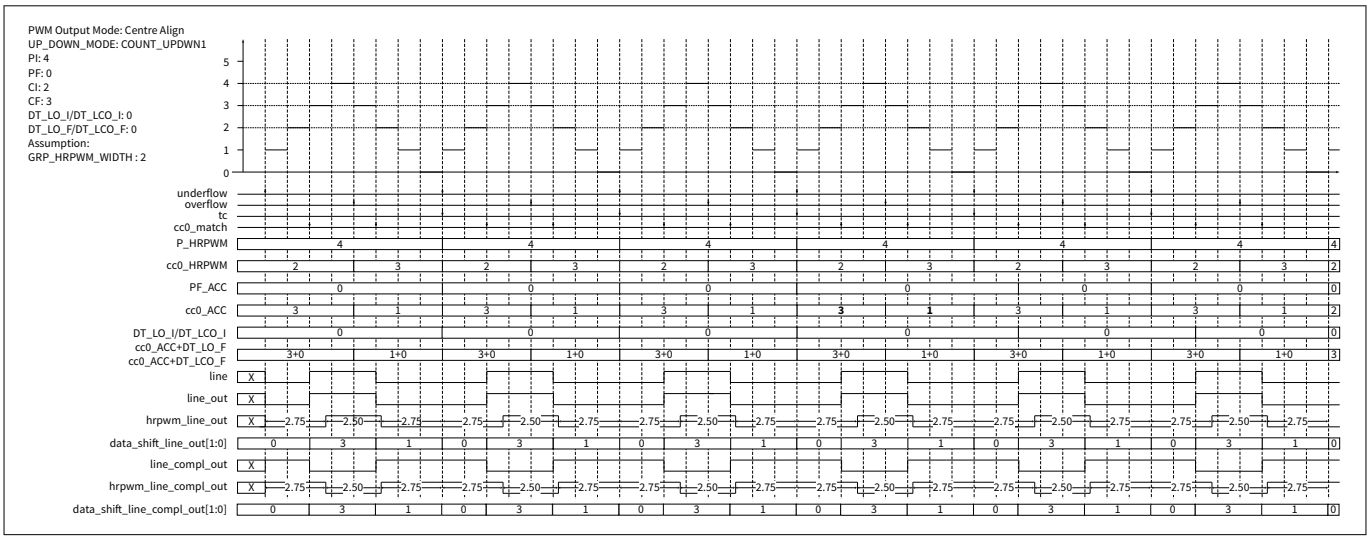


Figure 236 **Example of center align – Scenario 3 (PF=0, CF=3) in PWM mode with HRPWM feature enabled**

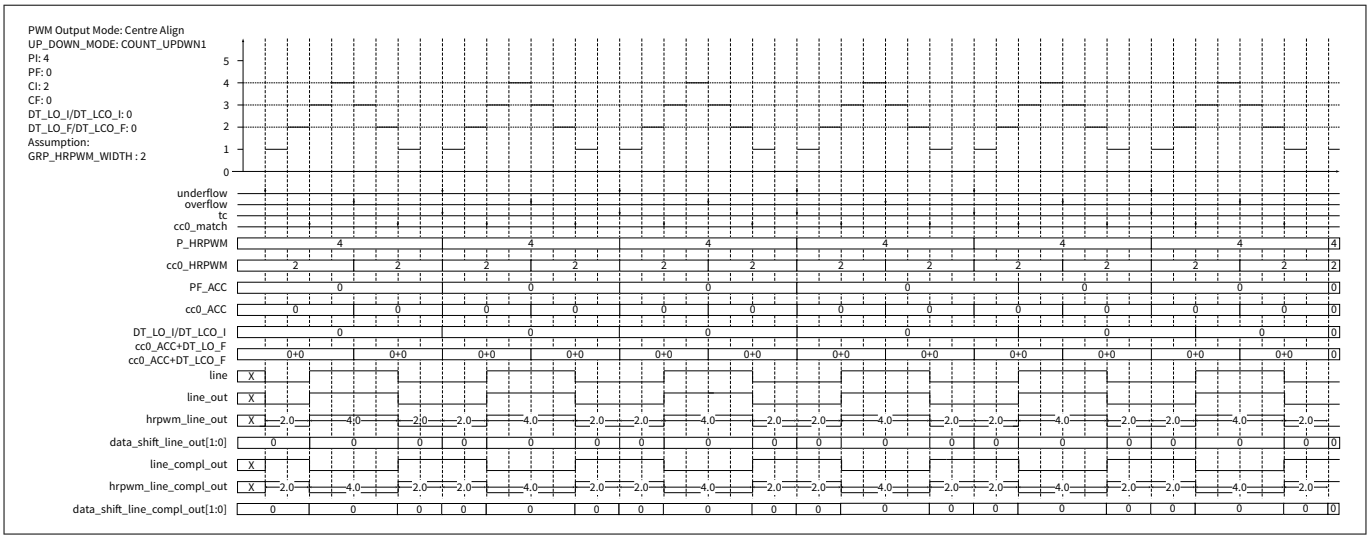


Figure 237 Example of center align – Scenario 4 (PF=0, CF=0) in PWM mode with HRPWM feature enabled

If PF is set to non-zero, the duty cycle will not be equal for each period cycle. For the center align case, the design will change the value of cc0 to CI+1 upon an overflow event when CF is non-zero. In addition, special handling needs to be handled at CI+1 to factor the subtraction for the count down portion when the overflow condition occurred. Please refer to the figure below.

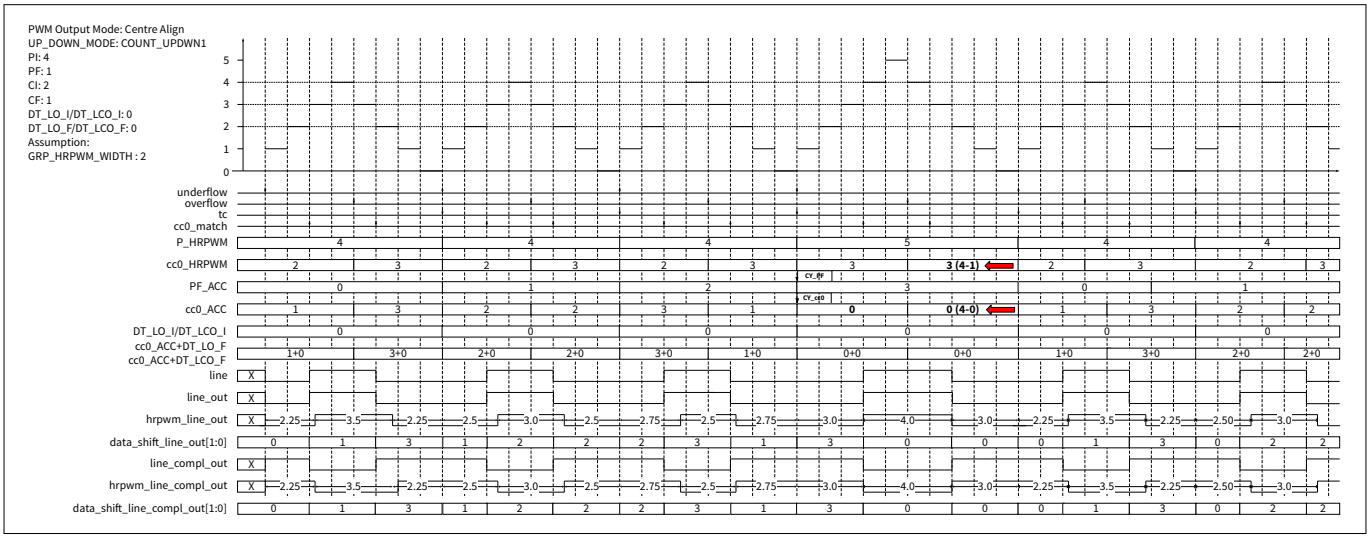


Figure 238 Example of center align – PF set to non-zero in PWM mode with HRPWM feature enabled

25.6.2.7 PWM output mode – Center align asymmetrical with cc0

The following diagram illustrate an example of PWM output mode configured in center align asymmetrical with cc0 with HRPWM feature enabled.

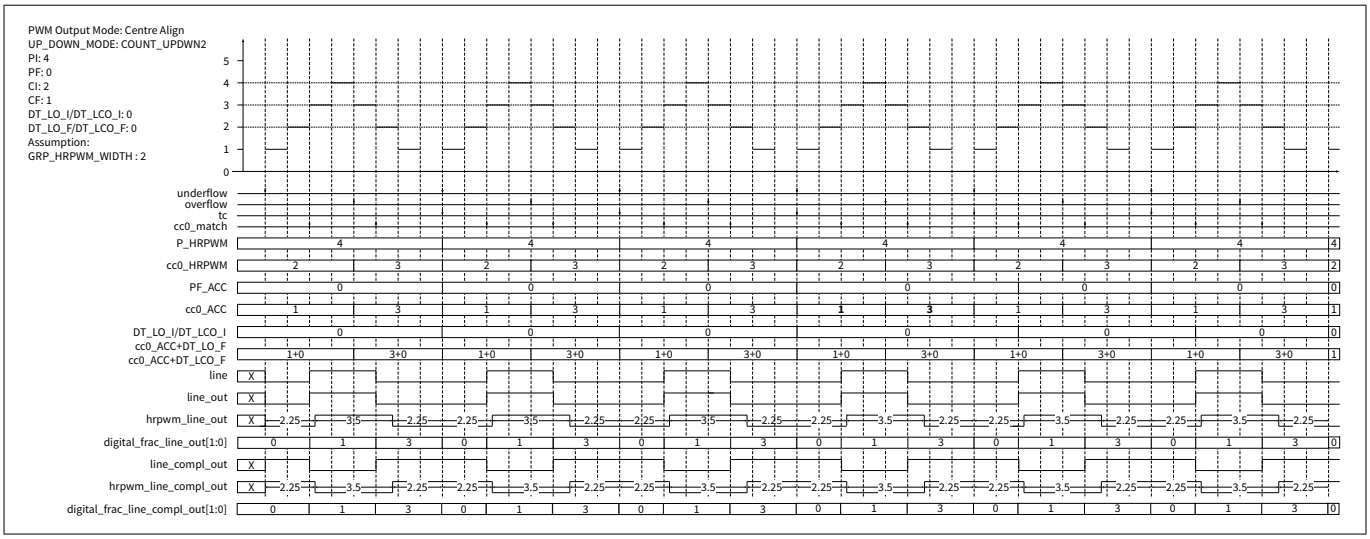


Figure 239 Example of Center Align asymmetric with cc0 in PWM Mode with HRPWM Feature enabled

25.6.2.8 PWM output mode – Center align asymmetric with cc0 and cc1

The Figure 240 shows an example of a PWM output mode configured in center align asymmetrical with cc0 and cc1 with HRPWM feature enabled.

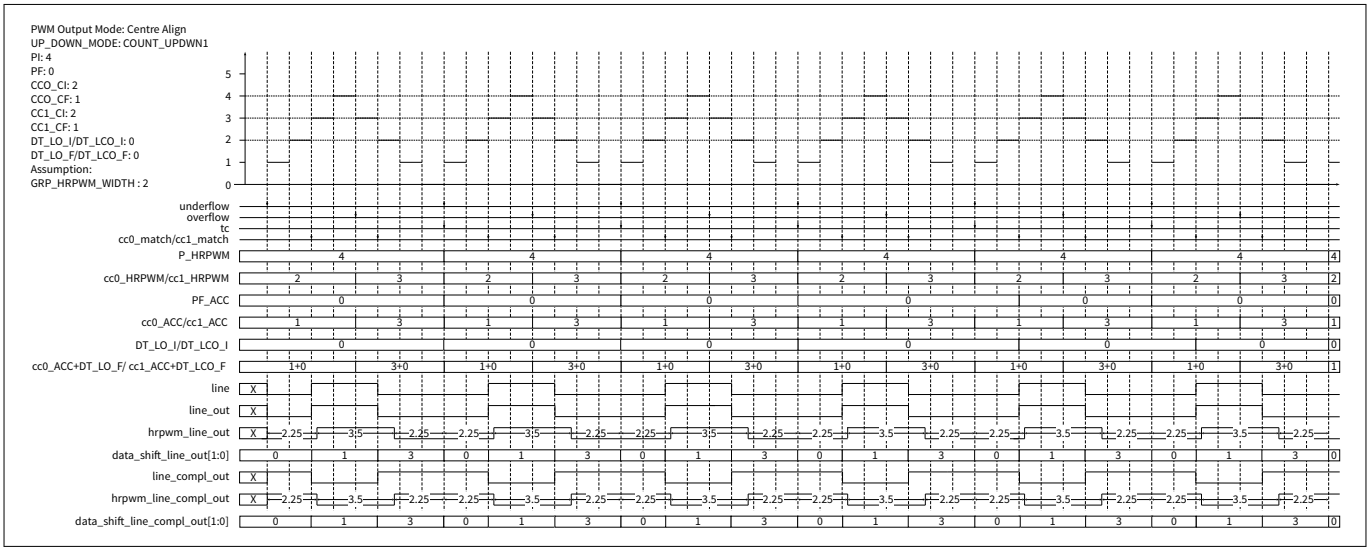


Figure 240 Example of center align asymmetric with cc0 and cc1 in PWM mode with HRPWM feature enabled

25.6.3 PWM_DT

In PWM_DT mode, when the HRPWM feature is enabled, the following are supported:

- The LSBs are used for fractional or micro ticks for delaying the line_out and line_compl_out outputs for the PERIOD/PERIOD_BUFF, CC0/CC0_BUFF, and CC1/CC1_BUFF and DT/DT_BUFF. The number of LSB bits used

for fractional control is 6. The remaining bits given by [GRP_CNT_WIDTH-1:GRP_HRPWM_WIDTH] are used by the TCPWM counter

- The LSBs are used for fractional or micro ticks for delaying the line_out and line_compl_out outputs in deadtime insertion for the following registers DT/DT_BUFF. The number of LSB bits used for fraction control is 6. The remaining 26 bits are used by the TCPWM counter

The same PWM output mode that is defined in PWM mode for the HRPWM feature is supported in PWM_DT Mode.

Refer to PWM output mode section for more information.

25.6.3.1 Computation concept

The following figures show an example of how data_shift_line_out and data_shift_line_compl_out are calculated in PWM_DT mode and sent out with line_out and line_compl_out to the delay element.

When the DT register contains a fractional value >0 for line_out, it is added into PF_ACC. Similarly, the DT fractional value for line_compl_out is added to CC0_ACC. In case the final value overflows, the carry is added to the corresponding DT integer value for one TC event period.

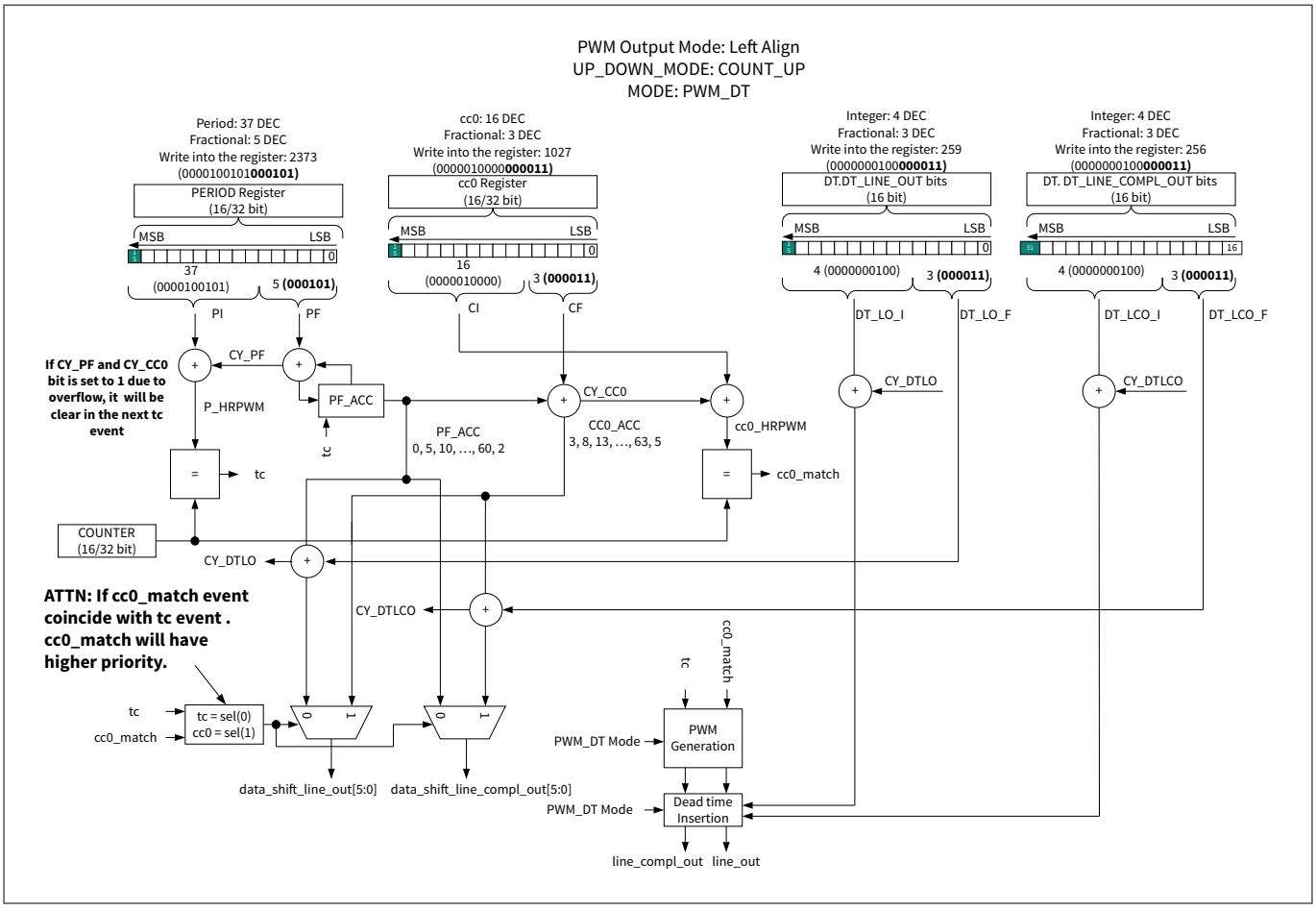


Figure 241 Calculation of data_shift_line_out and data_shift_line_compl_out in PWM_DT mode

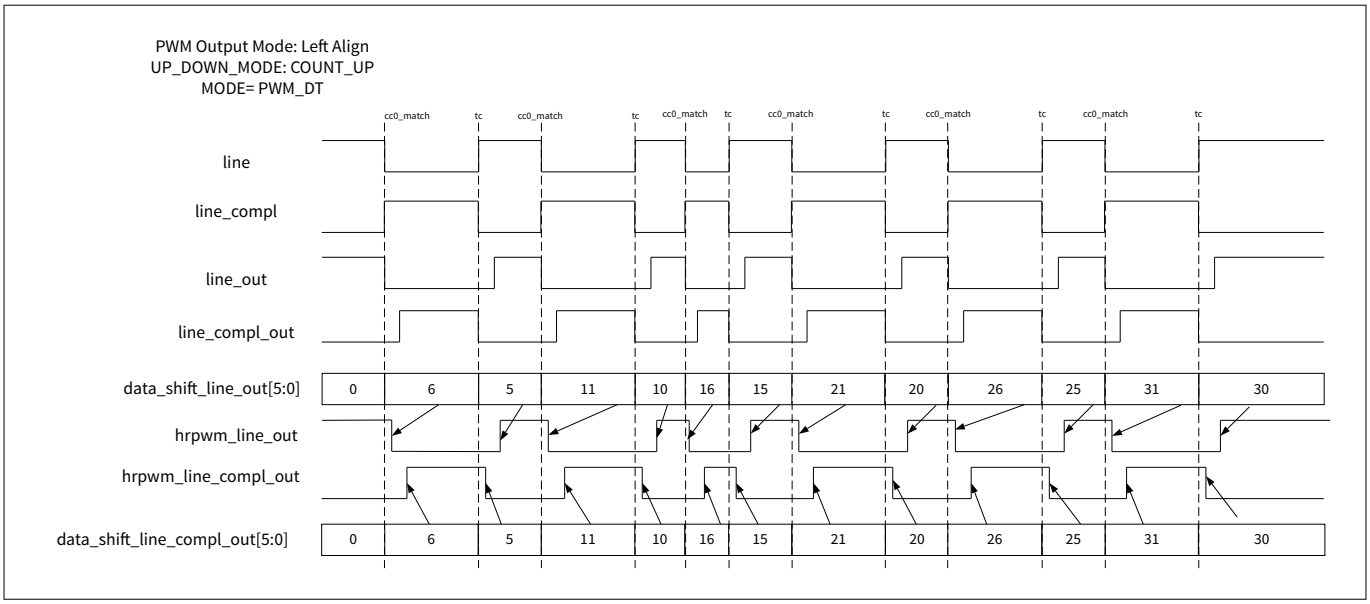


Figure 242 Example of TCPWM outputs in PWM_DT mode

25.6.3.2 PWM_DT – Left align

The Figure 243 shows an example of PWM output mode configured in left align with HRPWM feature enabled under PWM_DT mode. Note that the duty cycle duration will be equal for each period cycle. Carry event for dead time will occur due to DT_LCO_F + CCO_ACC and DT_LO_F + PF_ACC event.

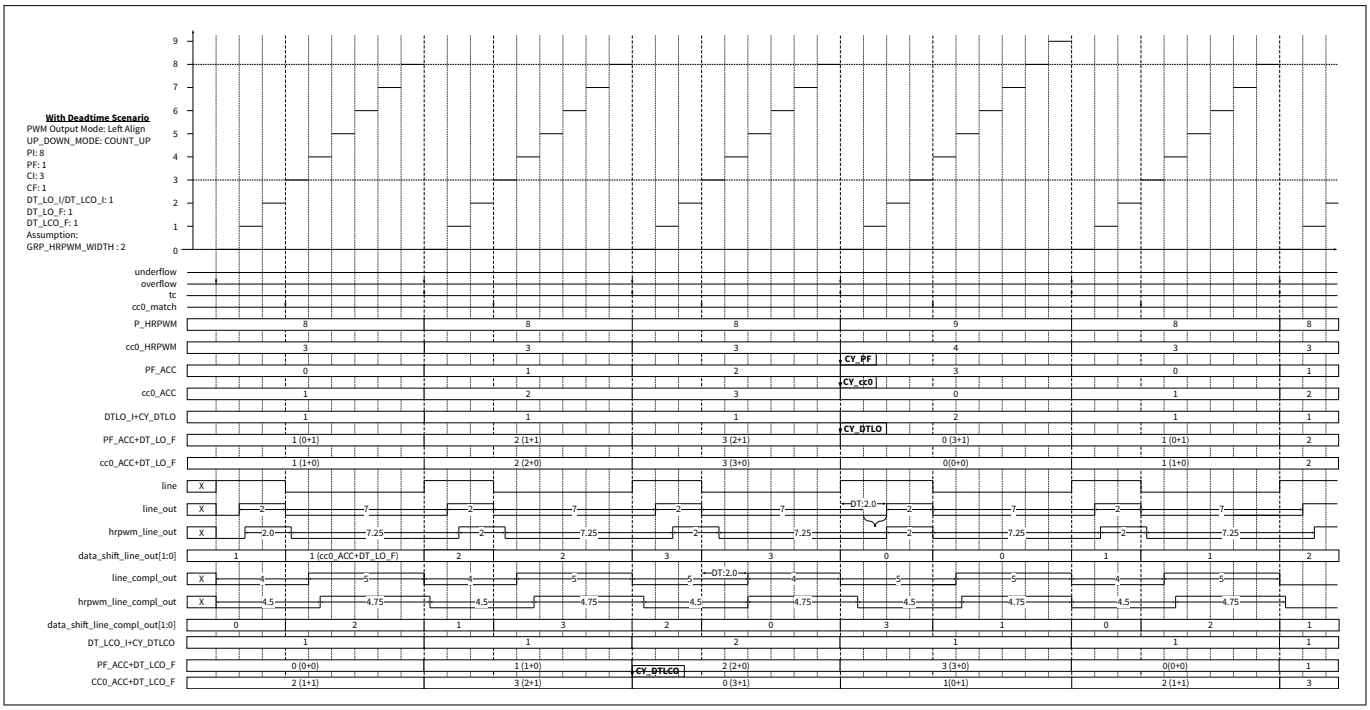


Figure 243 Example of left align in PWM_DT mode with HRPWM feature enabled

25.6.3.3 PWM_DT – Right align

The Figure 244 shows an example of PWM output mode configured in right align with HRPWM feature enabled under PWM_DT mode. Note that the duty cycle duration will be equal for each period cycle.

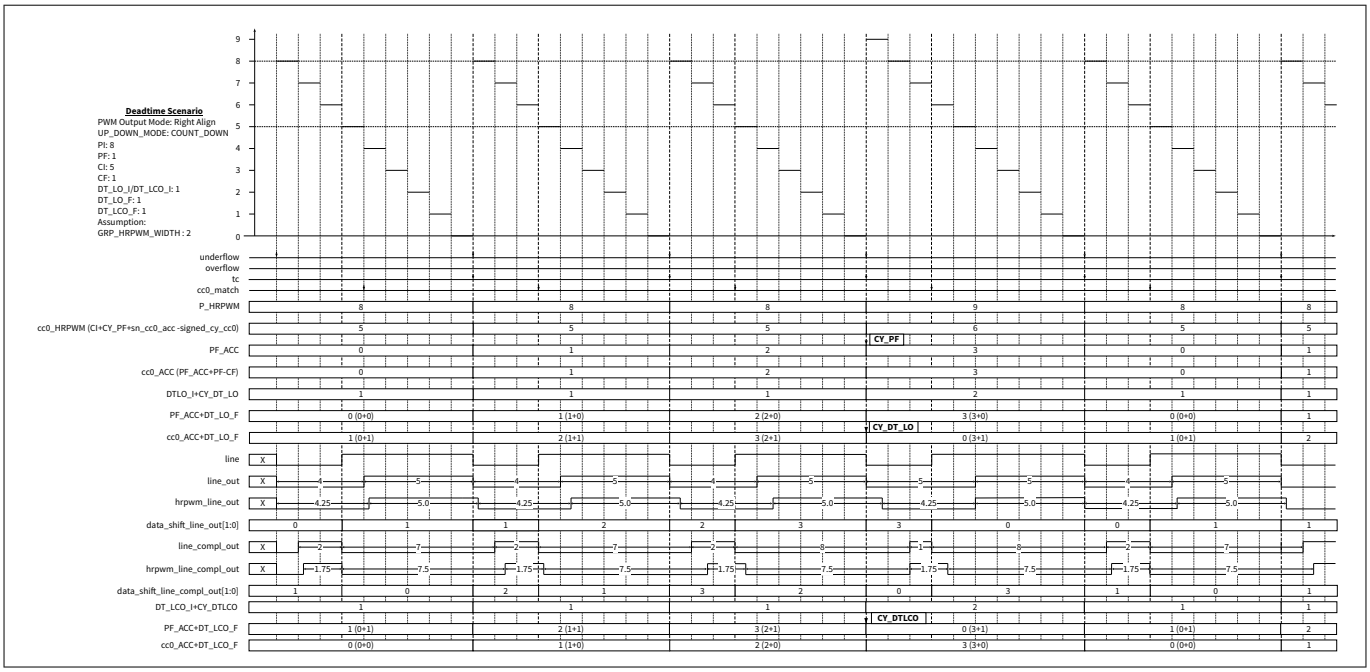


Figure 244 Example of right align in PWM_DT with HRPWM feature enabled

25.6.3.4 PWM_DT – Center align

The following diagramillustrates the expectation of HRPWM for center align case. It shows the handling of the dead time and its fraction component in addition to the overall fractional shift value that will be forwarded to the delay elements.

***Note**: When the fractional period is set to non-zero, the duty cycle will not be equal for each period cycle.*

- Edge at #1(pointed by blue dot) adds the fractional part of the DEADTIME_LINE_OUT to the overall fractional as shown with orange text
- Edge at #2(pointed by blue dot) shall not add the fractional part of the DEADTIME_LINE_OUT to the overall fractional
- Edge at #3(pointed by blue dot) adds the fractional part of the DEADTIME_LINE_COMPL_OUT to the overall fractional as shown with orange text
- Edge at #4(pointed by blue dot) shall not add the fractional part of the DEADTIME_LINE_ COMPL_OUT to the overall fractional

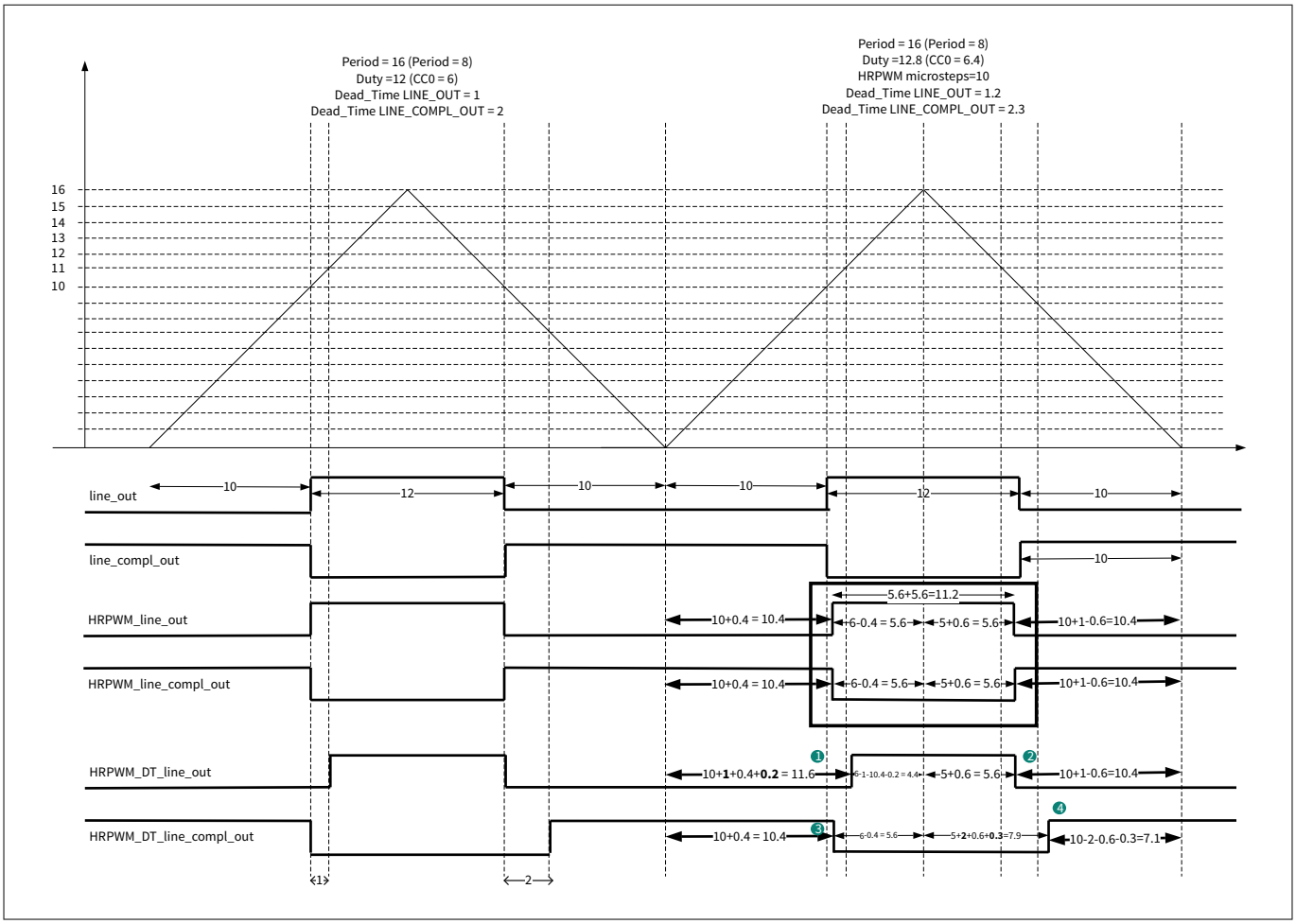


Figure 245 Center Align Case HRPWM Expectation for PWM_DT Mode

The following diagram illustrates an example of PWM output mode configured in center align with HRPWM feature enabled under PWM_DT mode. Please be aware that for center align case, design will change the value of cc0 to CI+1 upon overflow event. The duty cycle duration will be equal for each period cycle when PF is set to zero. Carry event for dead time will occurred due to cc0_ACC + DT_LCO_F

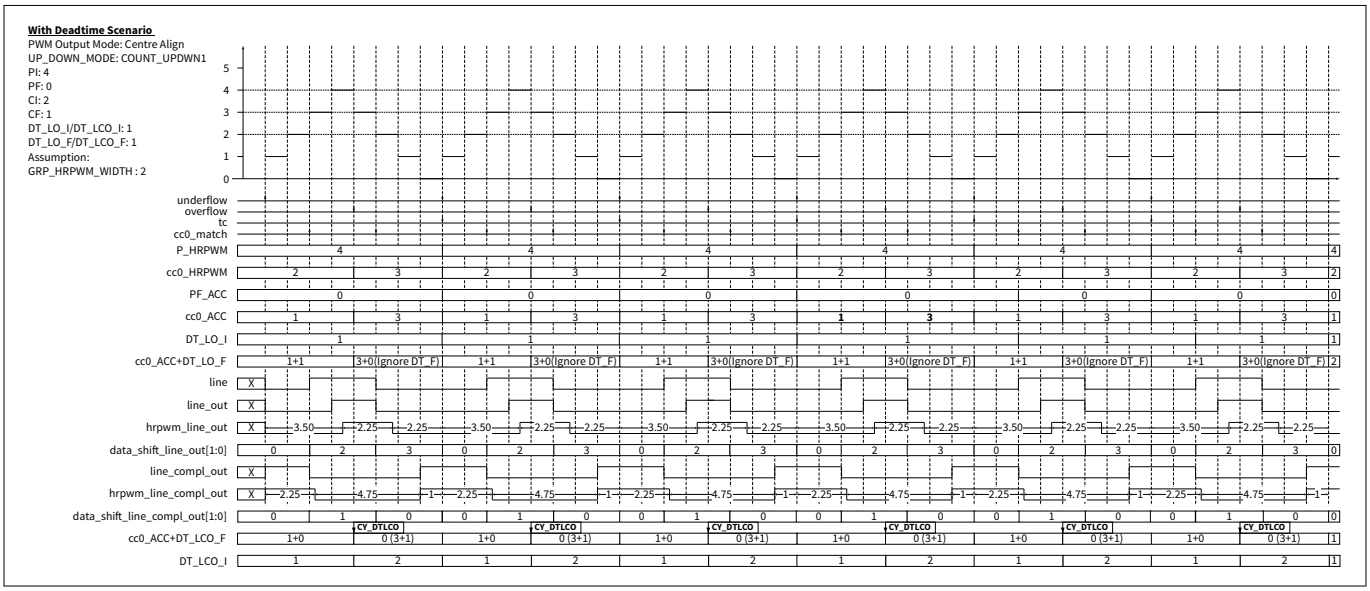


Figure 246 Example of Center Align – PF set to 0 in PWM_DT Mode with HRPWM Feature enabled

25.7 Recovery

TCPWM can be recovered with any active reset event, such as:

- Power-on reset (POR)
- External reset (XRES)
- Watchdog timer reset (MCWDT and WDT)
- Brownout detection reset
- Overvoltage and overcurrent detection reset

There is no unexpected state in which the TCPWM can enter.

25.8 Initialize

The initial state of TCPWM pins is Hi-Z. Some registers are reset on an active reset; some of the MMIO registers are retained in Deep Sleep. None of the registers are retained through Hibernate or other Low-Power modes. An active reset will reset the pin state back to Hi-Z unless CTRL.PWM_DISABLE_MODE is set to a non-zero value. Results vary in this scenario based on the configuration.

***Note**: On warm boot (wake-up from Deep Sleep), software must enable TCPWM once before resuming normal TCPWM operation, because PERI group 0 is not reinitialized automatically.*

25.9 Pin status

When TCPWM is unused, the status for TCPWM pins will be Hi-Z. To disable TCPWM, make sure the ENABLED and PWM_DISABLE_MODE bits in the CTRL register are set to '0', MOTIF_PCONF.ENABLE also is set to 0.

25.10 Registers summary

Table 228 List of TCPWM registers

Name	Description
TCPWM0_GRPx_CNTy_CTRL	Counter control register
TCPWM0_GRPx_CNTy_STATUS	Counter status register
TCPWM0_GRPx_CNTy_COUNTER	Counter count register
TCPWM0_GRPx_CNTy_CC0	Counter compare/capture 0 register
TCPWM0_GRPx_CNTy_CC0_BUFF	Counter buffered compare/capture 0 register
TCPWM0_GRPx_CNTy_CC1	Counter compare/capture 1 register
TCPWM0_GRPx_CNTy_CC1_BUFF	Counter buffered compare/capture 1 register
TCPWM0_GRPx_CNTy_PERIOD	Counter period register
TCPWM0_GRPx_CNTy_PERIOD_BUFF	Counter-buffered period register
TCPWM0_GRPx_CNTy_LINE_SEL	Counter line selection register
TCPWM0_GRPx_CNTy_LINE_SEL_BUFF	Counter-buffered line selection register
TCPWM0_GRPx_CNTy_DT	Counter PWM dead time register
TCPWM0_GRPx_CNTy_DT_BUFF	Counter buffered PWM dead time register

(table continues...)

Table 228 (continued) List of TCPWM registers

Name	Description
TCPWM0_GRPx_CNTy_PS	Counter prescalar register
TCPWM0_GRPx_CNTy_TR_CMD	Counter trigger command register
TCPWM0_GRPx_CNTy_TR_IN_SEL0	Counter input trigger selection register 0
TCPWM0_GRPx_CNTy_TR_IN_SEL1	Counter input trigger selection register 1
TCPWM0_GRPx_CNTy_TR_IN_EDGE_SEL	Counter input trigger edge selection register
TCPWM0_GRPx_CNTy_TR_PWM_CTRL	Counter trigger PWM control register
TCPWM0_GRPx_CNTy_TR_OUT_SEL	Counter output trigger selection register
TCPWM0_GRPx_CNTy_INTR	Interrupt request register
TCPWM0_GRPx_CNTy_INTR_SET	Interrupt set request register
TCPWM0_GRPx_CNTy_INTR_MASK	Interrupt mask register
TCPWM0_GRPx_CNTy_INTR_MASKED	Interrupt masked request register
TCPWM0_GRPx_CNTy_ONE_GF	Glitch filter register for one to one trigger
TCPWM0_GRPx_CNTy_LFSR	LFSR register. x ∈ {1, 2}
TCPWM0_GRPx_CNTy_TR_ONE_SYNC_BYPASS	Sync bypass register for one to one trigger
TCPWM0_GRPx_CNTy_HRPWM_CTRL	Counter control register for HRPWM feature. x ∈ {0}
TCPWM0_GRP0_CNTy_CTRL2	Control register for Counter
TCPWM0_TR_ALL_GF	Glitch filter for all group triggers
TCPWM0_TR_ALL_SYNC_BYPASS	Trigger Sync bypass for all group triggers
TCPWM0_MOTIF_GRPz_MOTIF0_PCONF	Global control register
TCPWM0_MOTIF_GRPz_MOTIF0_PSUS	Suspend configuration
TCPWM0_MOTIF_GRPz_MOTIF0_PRUNS	MOTIF runs bit set
TCPWM0_MOTIF_GRPz_MOTIF0_PRUN	MOTIF run bit status
TCPWM0_MOTIF_GRPz_MOTIF0_MIDR	Module Identification register
TCPWM0_MOTIF_GRPz_MOTIF0_HIST	Hall inputs sample trigger
TCPWM0_MOTIF_GRPz_MOTIF0_HMEC	Hall mode extra config
TCPWM0_MOTIF_GRPz_MOTIF0_HALP	Hall current and expected patterns
TCPWM0_MOTIF_GRPz_MOTIF0_HALPS	Hall current and expected shadow patterns
TCPWM0_MOTIF_GRPz_MOTIF0_HOSC	Hall sensor output config
TCPWM0_MOTIF_GRPz_MOTIF0_MCM	Multi-channel mode pattern
TCPWM0_MOTIF_GRPz_MOTIF0_MCSM	Multi-channel mode shadow pattern LUT0
TCPWM0_MOTIF_GRPz_MOTIF0_MCMS	Multi-channel mode control set
TCPWM0_MOTIF_GRPz_MOTIF0_MCMC	Multi-channel mode control clear
TCPWM0_MOTIF_GRPz_MOTIF0_MCMF	Multi-channel mode flag status

(table continues...)

Table 228 (continued) List of TCPWM registers

Name	Description
TCPWM0_MOTIF_GRPz_MOTIF0_MCPF	Multi-channel pattern fault
TCPWM0_MOTIF_GRPz_MOTIF0_MOSC	Multi-channel output config
TCPWM0_MOTIF_GRPz_MOTIF0_QDC	Quadrature decoder configuration
TCPWM0_MOTIF_GRPz_MOTIF0_QOSC	Quadrature output config
TCPWM0_MOTIF_GRPz_MOTIF0_MCMEC	Multi-channel extra config
TCPWM0_MOTIF_GRPz_MOTIF0_PFLG	MOTIF interrupt status
TCPWM0_MOTIF_GRPz_MOTIF0_PFLGE	MOTIF interrupt enables
TCPWM0_MOTIF_GRPz_MOTIF0_SPFLG	Interrupt set register
TCPWM0_MOTIF_GRPz_MOTIF0_RPFLG	Interrupt clear register
TCPWM0_MOTIF_GRPz_MOTIF0_MCSM1	Multi-channel mode shadow pattern LUT1
TCPWM0_MOTIF_GRPz_MOTIF0_MCSM2	Multi-channel mode shadow pattern LUT2
TCPWM0_MOTIF_GRPz_MOTIF0_MCSM3	Multi-channel mode shadow pattern LUT3
TCPWM0_MOTIF_GRPz_MOTIF0_MCSM4	Multi-channel mode shadow pattern LUT4
TCPWM0_MOTIF_GRPz_MOTIF0_MCSM5	Multi-channel mode shadow pattern LUT5
TCPWM0_MOTIF_GRPz_MOTIF0_CLUT	Hall mode LUT config
TCPWM0_MOTIF_GRPz_MOTIF0_SLUT	Hall mode LUT status
TCPWM0_MOTIF_GRPz_MOTIF0_PDBG	MOTIF debug register
TCPWM0_MOTIF_GRPz_MOTIF0_PLP0S	MOTIF low pass 0 status
TCPWM0_MOTIF_GRPz_MOTIF0_PLP1S	MOTIF low pass 1 status
TCPWM0_MOTIF_GRPz_MOTIF0_PLP2S	MOTIF low pass 2 status

***Note**: For PSC3M/P6:*

- *x denotes the group number. x ∈{0, 1, 2}*
- *y denotes the counter number within that group. y ∈{0, 1, 2, 3, 4, 5, 6, 7}*
- *z denotes the group number in respective TCWPM group. z ∈{1, 2}*

Note that overwriting the same value on each register has different effects and they are explained in the register map by the software access attributes. TCPWM registers have the following access restrictions:

- All status registers are not software-writable
- TR_CMD is set in software and cleared in hardware
- INTR is cleared in software and set in hardware (by writing '1' to INTR_SET)
- Read INTR_SET will return the value of INTR
- Other registers are normal and can be overwritten with the same value

26 Controller area network flexible data-rate (CAN FD)

26.1 Overview

The CAN FD controller complies with the ISO11898-1 (CAN specification rev. 2.0 parts A and B). In addition, it supports the Time-Triggered CAN (TTCAN) protocol defined in ISO 11898-4.

All message handling functions are implemented by the RX and TX handlers. The RX handler manages message acceptance filtering, transfer of received messages from the CAN core to a message RAM, and receives message status information. The TX handler transfers transmit messages from the message RAM to the CAN core and provides transmit status information.

Two separate clocks are provided to the CAN FD controller: CAN clock (CANFD[0].CLOCK_CAN_EN[0:1]) for CAN operation and system clock (SYSCPUSS/clk_hf0) for the internal block operation. The acceptance filtering is implemented by a combination of up to 192 filter elements, where each can be configured as a range, bit mask, or dedicated ID filter.

The CAN FD controller functions only in Active and Sleep power modes. In Deep Sleep mode, it is not functional but the configuration and message RAM content is fully retained.

26.1.1 Features

The CAN FD controller has the following features:

- Flexible data-rate (FD) (ISO 11898-1: 2015)
  - Up to 64 data bytes per message
  - Supports maximum 8 Mbps
- Time-Triggered (TT) communication on CAN (ISO 11898-4: 2004)
  - TTCAN protocol level 1 and level 2 completely in hardware
- AUTOSAR support
- Acceptance filtering
- Two channels; each channel includes the following:
  - Up to 64 dedicated receive buffers per channel
  - Two configurable receive FIFOs (up to 64 buffers each) with acceptance filters
  - Up to 32 dedicated transmit buffers
    - Configurable transmit FIFO
    - Configurable transmit queue
    - Configurable transmit event FIFO
- Programmable loop-back test mode
- Power-down support
- Shared message 4-KB RAM
- Receive FIFO top pointer logic
  - Enables DMA access on the FIFO
- DMA for debug message and received FIFOs

26.1.2 Features not supported

- Asynchronous serial communication (ASC)
- Interrupt of bit error corrected (IR.BEC) in M_TTCAN
  - This bit is fixed at '0'

26.2 Architecture

26.2.1 Block diagram

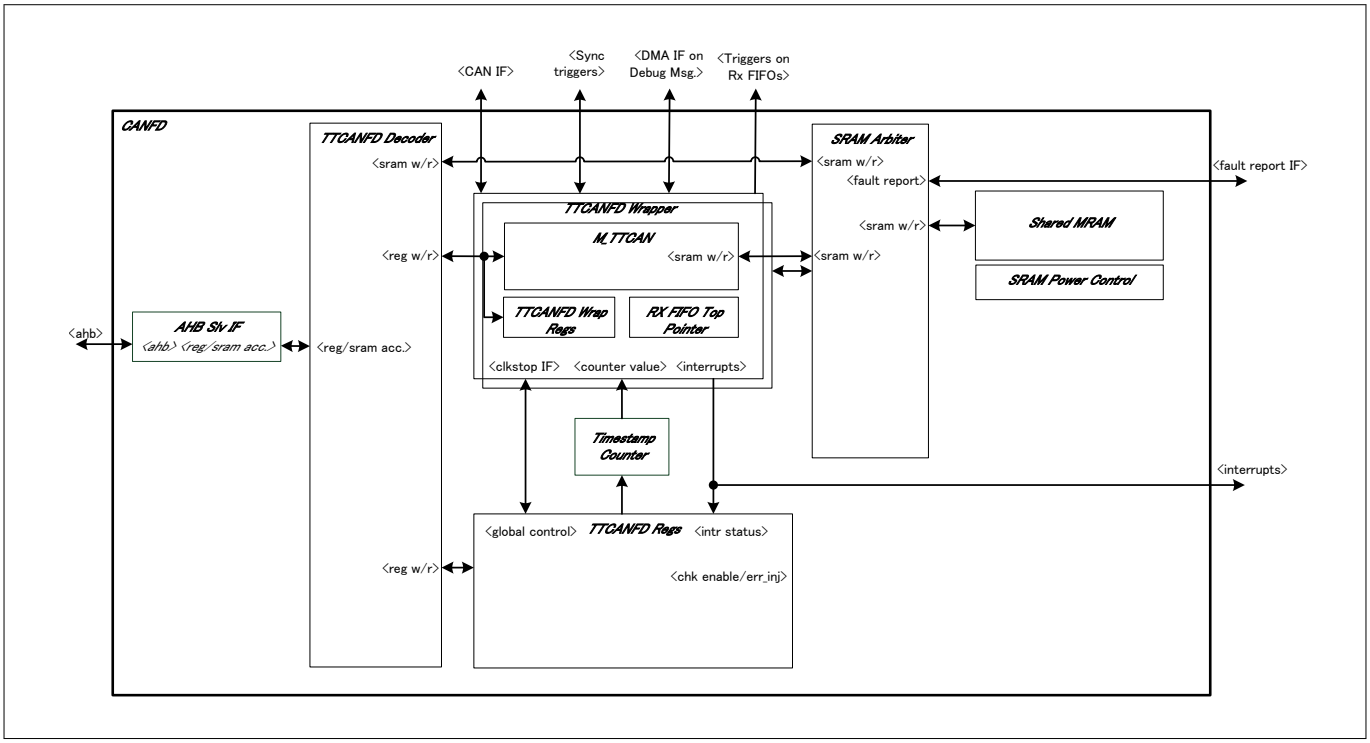


Figure 247 Block diagram

26.2.2 Dual clock sources

The TTCAN channel has two clock inputs: clk_can and clk_sys.

The `clk_can`, derived from the system peripheral clock dividers, is used for the CAN (or CAN FD) operation. It is recommended to use 20 MHz, 40 MHz, or 80 MHz.

The `clk_sys` is used for all operations such as, register accesses, SRAM accesses and so on, but in CAN operations. `clk_sys` must run equal to or faster than `clk_can`, and it is allowed to be derived from a spread-spectrum PLL to minimize EMC unlike `clk_can`.

See the Clocking system section for more details about the clock configuration.

26.2.3 Interrupt line

The M_TTCAN channel provides two interrupt lines: interrupt0 and interrupt1. Interrupts from any source within the M_TTCAN channel is routed either to interrupt0 or interrupt1 by using the ILS and TTILS registers. By default, all the interrupts are routed to interrupt0. By programming, enable interrupt line 0 (ILE.EINT0) and

enable interrupt line 1 (ILE.EINT1), the interrupt lines are enabled or disabled separately for each interrupt source.

26.3 Functional description

26.3.1 Modes of operation

26.3.1.1 Software initialization

This section describes to set or reset the initialization bit (CCCR.INIT). The CCCR.INIT bit is set as follows:

- Software or hardware reset
- Uncorrected bit error is detected in MRAM
- By choosing Bus Off

While the CCCR.INIT is set:

- Message transfer from and to the CAN bus is stopped
- The status of the CAN bus output canfd[0].ttcan_tx[0:1] is recessive (HIGH)
- The protocol error counters are unchanged

Setting CCCR.INIT does not change any Configuration register.

Resetting CCCR.INIT finishes the software initialization. The CAN FD controller then synchronizes itself to the data transfer on the CAN bus by waiting for the occurrence of a sequence of 11 consecutive recessive bits (Bus idle) before it can take part in bus activities and start the message transfer. Access/Set/Reset Properties of registers are affected by Configuration Change Enable (CCCR.CCE) access to the Configuration registers and enabled when both bits, CCCR.INIT and CCCR.CCE, are set (write protected). CCCR.CCE is set/reset while CCCR.INIT is '1'. CCCR.CCE is automatically reset when CCCR.INIT is reset.

The following registers are reset when CCCR.CCE is set

- HPMS - High priority message status
- RXF0S - RX FIFO 0 status
- RXF1S - RX FIFO 1 status
- TXFQS - TX FIFO/queue status
- TXBRP - TX buffer request pending
- TXBTO - TX buffer transmission occurred
- TXBCF - TX buffer cancellation finished
- TXEFS - TX event FIFO status
- TTOST - TT operation status
- TTLGT - TT local and global time, only global time TTLGT.GT is reset
- TTCTC - TT cycle time and count
- TTCSM - TT cycle sync mark

In addition

- The timeout counter value (TOCV.TOC[15:0]) is preset to the value configured by the timeout period(TOCC.TOP[15:0]) when CCCR.CCE is set
- The state machines of TX and RX handlers are held in idle state while CCCR.CCE is '1'

The following registers are only writable while CCCR.CCE is '0'.

- TXBAR - TX buffer add request
- TXBCR - TX buffer cancellation request

The test mode enable (CCCR.TEST) and bus monitoring mode(CCCR.MON) can only be set by the CPU while CCCR.INIT is '1' and CCCR.CCE is '1'. Both bits can be reset at any time.

The disable automatic retransmission (CCCR.DAR) can only be set/reset while CCCR.INIT is '1' and CCCR.CCE is '1'.

Message RAM initialization

Each MRAM word should be reset by writing '0x00000000' before the CAN FD controller configuration. This prevents MRAM bit errors when reading uninitialized words, and also avoids unexpected filter element configurations in MRAM.

26.3.1.2 Normal operation

The M_TTCAN's default operating mode after the hardware reset is event-driven CAN communication without time triggers (TTOCF.OM = 00). Both CCCR.INIT and CCCR.CCE is set before the TT operation mode is changed.

When M_TTCAN is initialized and CCCR.INIT is reset to '0', M_TTCAN synchronizes itself to the CAN bus and is ready for communication.

After passing the acceptance filtering, received messages including Message ID and DLC are stored into a dedicated RX buffer or into RX FIFO 0 or RX FIFO 1.

For messages to be transmitted, dedicated TX buffers and a TX FIFO/TX queue can be initialized or updated. The automated transmission on reception of remote frames is not implemented.

26.3.1.3 CAN FD operation

The two variants in CAN FD frame transmission are:

- CAN FD frame without bit rate switching
- CAN FD frame where the control, data, and CRC fields are transmitted with a higher bit rate than the beginning and end of the frame

The previously reserved bit in CAN frames with 11-bit identifiers and 29-bit identifiers are decoded as the FDF bit.

- FDF = recessive signifies a CAN FD frame
- FDF = dominant signifies a classic CAN frame

In a CAN FD frame, the two bits following FDF, reserved bits (RES), and bit rate switch (BRS) decide whether the bit rate inside the CAN FD frame is switched. A CAN FD bit rate switch signified by RES is dominant and BRS is recessive. The coding of RES as recessive is reserved for future expansion of the protocol. If the M_TTCAN receives a frame with FDF and RES as recessive, it will signal a protocol exception event by setting the PSR.PXE bit. When the protocol exception handling is enabled (CCCR.PXHD = 0), it causes the operation state to change from receiver (PSR.ACT =10) to integrating (PSR.ACT = 00) at the next sample point. If the protocol exception handling is disabled (CCCR.PXHD =1), the M_TTCAN treats a recessive RES bit as an error and responds with the error frame.

The CAN FD operation is enabled by programming CCCR.FDOE. If CCCR.FDOE is '1', transmission and reception of CAN FD frames is enabled. The transmission and reception of classic CAN frames is always possible. Whether a CAN FD frame or a classic CAN frame is transmitted can be configured via the FDF bit in the respective TX buffer element. With CCCR.FDOE as '0', the received frames are interpreted as classic CAN frames, which leads to the transmission of an error frame when receiving a CAN FD frame. When the CAN FD operation is disabled, no CAN FD frames are transmitted if the FDF bit of a TX buffer element is set. CCCR.FDOE and CCCR.BRSE can only be changed while CCCR.INIT and CCCR.CCE are both set.

CCCR.FDOE is '0', the setting of FDF and BRS is ignored and frames are transmitted in classic CAN format. When CCCR.FDOE = 1 and CCCR.BRSE = 0, only FDF of a TX buffer element is evaluated. When CCCR.FDOE = 1 and CCCR.BRSE = 1, transmission of CAN FD frames with bit rate switching is enabled. All the TX buffer elements with FDF and BRS bits set are transmitted in CAN FD format with the bit rate switching.

A mode change during the CAN operation is only recommended under the following conditions:

- The failure rate in the CAN FD data phase is significantly higher than in the CAN FD arbitration phase. In this case, disable the CAN FD bit rate switching option for transmissions
- During system startup, all the nodes are verified to transmit classic CAN messages that can communicate in CAN FD format and all the nodes switch to the CAN FD operation
- Wake-up messages in the CAN partial networking must be transmitted in classic CAN format
- End-of-line programming occurs if all the nodes are not CAN FD capable. The non-CAN FD nodes are held in Silent mode until the programming is completed. Then all the nodes are switchto classic CAN communication

In the CAN FD format, the coding of the DLC differs from the standard CAN format. The DLC codes 0 to 8 have the same coding as in standard CAN, codes 9 to 15, which in standard CAN have a data field of 8 bytes, are coded according to Table 229:

Table 229 Extended coding of DLC in CAN FD

DLC	9	10	11	12	13	14	15
Number of data bytes	12	16	20	24	32	48	64

In the CAN FD frames, the bit timing is switched inside the frame after the BRS bit, if this bit is recessive. Before the BRS bit, in the CAN FD arbitration phase, the nominal CAN bit timing is used as defined by the Nominal Bit Timing and Prescaler register (NBTP). In the following CAN FD data phase, the data phase bit timing is used as defined by the Data Bit Timing and Prescaler register (DBTP). The bit timing is switched from the data phase timing with a condition that occurs first: the CRC delimiter or when an error is detected.

The maximum configurable bit rate in the CAN FD data phase depends on the CAN clock frequency (clk_can). For example, with a CAN clock frequency of 20 MHz and the shortest configurable bit time of 4 tq, the bit rate in the data phase is 5 Mbps.

In both data frame formats, CAN FD and CAN FD with bit rate switching, the value of the bit error status indicator (ESI) is determined by the transmitter's error state at the start of the transmission. If the transmitter is error passive; ESI is transmitted recessive, otherwise it is transmitted dominant.

26.3.1.4 Transmitter delay compensation

During the data phase of a CAN FD transmission, only one node transmittes; and all the others receive. The length of the bus line has no impact. When transmitting via pin canfd[0].ttcan_tx[0:1], the M_TTCAN receives the transmitted data from its local CAN transceiver via pin canfd[0].ttcan_rx[0:1]. The received data is delayed by the transmitter delay. If the delay is greater than time segment before sample point (TSEG1), a bit error is detected. To enable a data phase bit time that is shorter than the transmitter delay, the delay compensation is introduced. Without the transmitter delay compensation, the bit rate in the data phase of a CAN FD frame is limited by the transmitter delay.

26.3.1.4.1 Description

The M_TTCAN's protocol unit has implemented a delay compensation mechanism to compensate the transmitter delay. This enables transmission with higher bit rates during the CAN FD data phase, independent of the delay of a specific CAN transceiver.

To check for bit errors during the data phase of transmitting nodes, the delayed transmit data is compared with the received data at the secondary sample point (SSP). If a bit error is detected, the transmitter reacts on the bit error at the next following regular sample point. During the arbitration phase the delay compensation is always disabled.

The transmitter delay compensation enables configurations where the data bit time is shorter than the transmitter delay. It is described in detail in the new ISO 11898-1:2015. It is enabled by setting the bit DBTP.TDC.

The received bit is compared with the transmitted bit at the SSP. The SSP position is defined as the sum of the measured delay from the M_TTCAN's transmit output canfd[0].ttcan_tx[0:1] through the transceiver to the receive input RX plus the transmitter delay compensation offset as configured by TDCR.TDCO. The transmitter delay compensation offset is used to adjust the position of the SSP inside the received bit (for example, half of the bit time in the data phase). The position of the secondary sample point is rounded to the next integer number of mtq (clk_can period).

PSR.TDCV shows the actual transmitter delay compensation value. PSR.TDCV is cleared when CCCR.INIT is set and updated at each transmission of an FD frame while DBTP.TDC is also set.

The following boundary conditions must be considered for the transmitter delay compensation implemented in the M_TTCAN:

- The sum of the measured delay from canfd[0].ttcan_tx[0:1] to canfd[0].ttcan_rx[0:1] and the configured transmitter delay compensation offset TDCR.TDCO is less than 6 bit times in the data phase
- The sum of the measured delay from canfd[0].ttcan_tx[0:1] to canfd[0].ttcan_rx[0:1] and the configured transmitter delay compensation offset TDCR.TDCO is less than or equal to 127 mtq. In case this sum exceeds 127 mtq, the maximum value of 127 mtq is used for transmitter delay compensation
- The data phase ends at the sample point of the CRC delimiter that stops checking of the receive bits at the SSPs

26.3.1.4.2 Transmitter delay compensation measurement

If transmitter delay compensation is enabled by programming DBTP.TDC = 1, the measurement is started within each transmitted CAN FD frame at the falling edge of bit FDF to bit RES. The measurement is stopped when this edge is seen at the receive input canfd[0].ttcan_rx[0:1] of the transmitter. The resolution of this measurement is one minimum time quanta (MTQ).

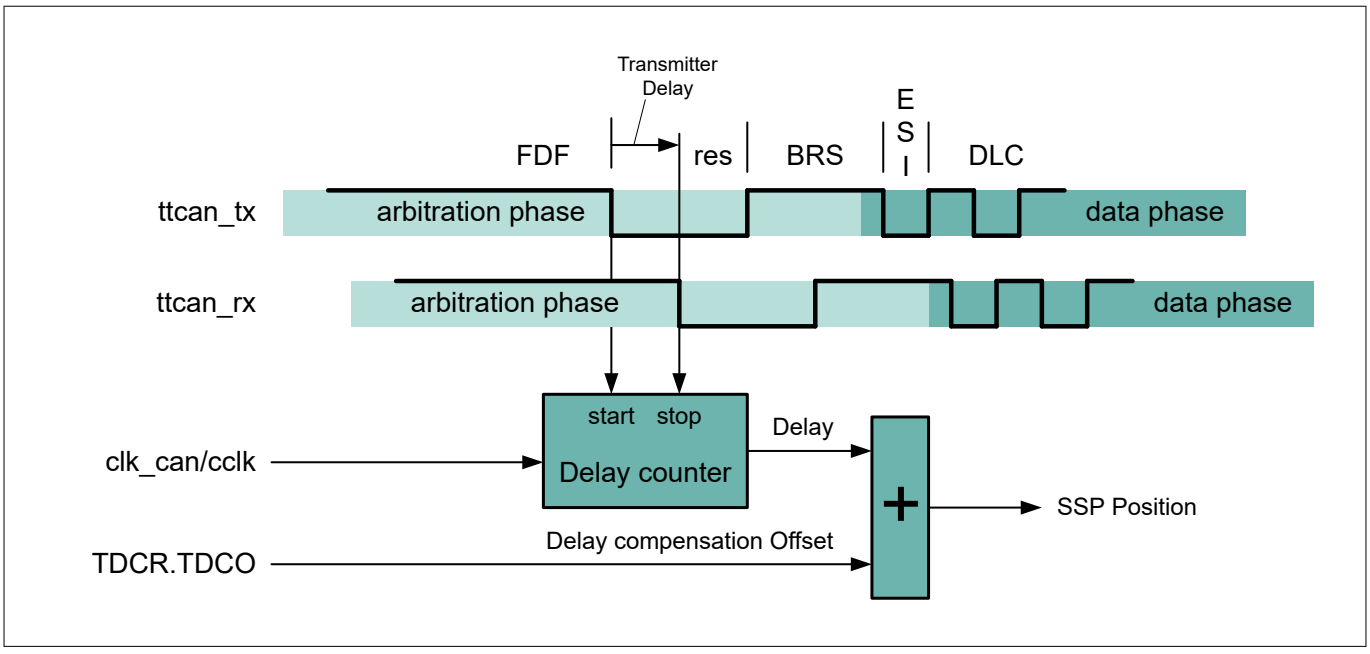


Figure 248 Transmitter delay measurements

A dominant glitch inside the received FDF bit ends the delay compensation measurement before the falling edge of the received RES bit, resulting in an early SSP position. The use of a transmitter delay compensation filter window is enabled by programming TDCR.TDCF. This defines a minimum value for the SSP position. Dominant edges on canfd[0].ttcan_rx[0:1], that results in an earlier SSP position are ignored for transmitter delay measurement. The measurement is stopped when the SSP position is at least TDCR.TDCF and canfd[0].ttcan_rx[0:1] is LOW.

26.3.1.5 Restricted Operation mode

In Restricted Operation mode, the node is able to receive data and remote frames and acknowledge valid frames, but it does not send data frames, remote frames, active error frames, or overload frames. In case of an error or overload condition, it does not send dominant bits; instead it waits for the occurrence of a bus idle condition to resynchronize itself to the CAN communication. The error counters (ECR.REC and ECR.TEC) are frozen while Error Logging (ECR.CEL) is active.

The CPU can set the CAN FD controller into Restricted Operation mode by setting the Restricted Operation mode bit (CCCR.ASM). CCCR.ASM can only be set by the CPU when both CCCR.CCE and CCCR.INIT are set to ‘1‘.CCCR.ASM can be reset by the CPU at any time.

The CAN FD controller enters Restricted Operation mode automatically when the TX handler is not able to read data from the MRAM in time. To leave this mode, the CPU should reset CCCR.ASM.

The Restricted Operation mode can be used in applications that adapt themselves to different CAN bit rates. In this case, the application tests different bit rates and leaves the mode after it has received a valid frame.

***Note**: The Restricted Operation mode must not be combined with the Loop Back mode (internal or external).*

26.3.1.6 Bus Monitoring mode

The M_TTCAN is set in Bus Monitoring mode by programming CCCR.MON to ‘1’ or when an error level S3 (TTOST.EL = 11) is entered. In this mode, the M_TTCAN is able to receive valid data frames and valid remote frames, but cannot start a transmission. In this mode, it sends only recessive bits on the CAN bus, if the M_TTCAN is required to send a dominant bit (ACK bit,overload flag, or active error flag), the bit is rerouted internally so that the M_TTCAN monitors this dominant bit, although the CAN bus may remain in the recessive state. In the Bus Monitoring mode, the TXBRP register is in the reset state.

The Bus Monitoring modeis used to analyze the traffic on a CAN bus without affecting it by the transmission of dominant bits. Figure 249 shows the connection of signals canfd[0].ttcan_tx[0:1] and canfd[0].ttcan_rx[0:1] to the M_TTCAN in Bus Monitoring mode.

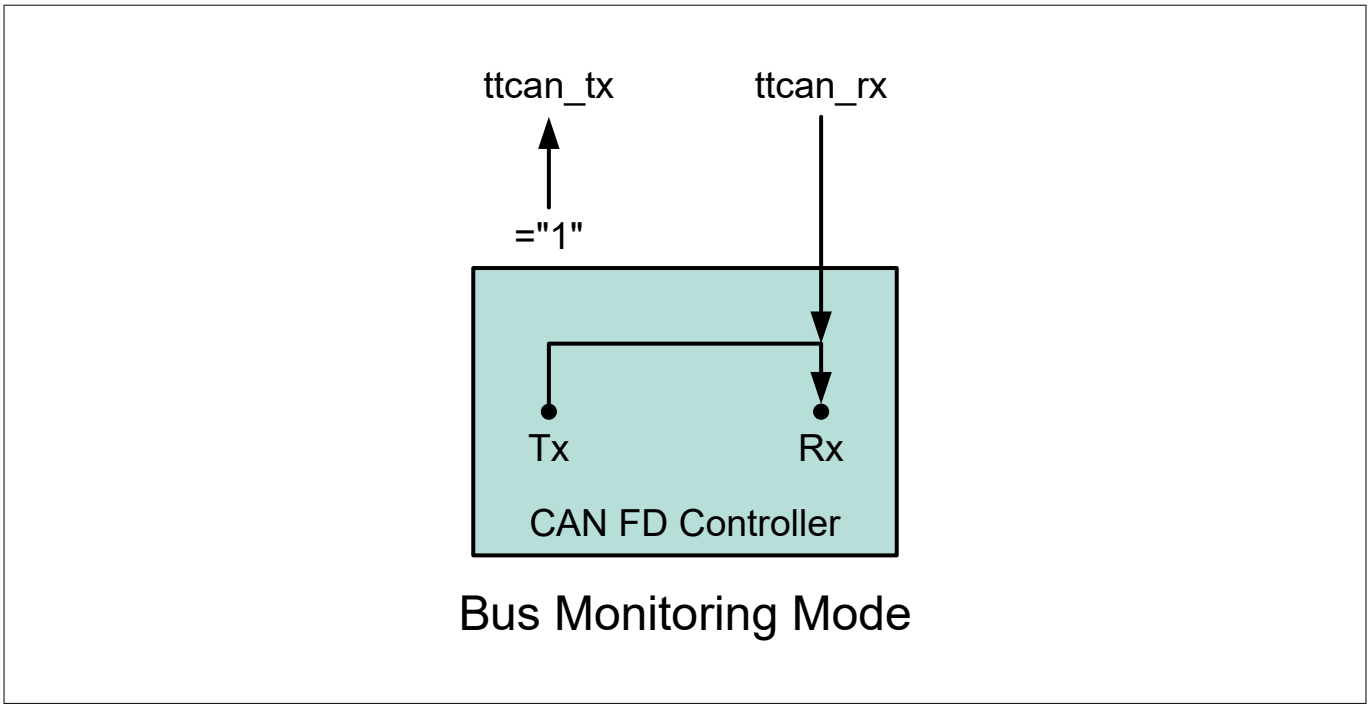


Figure 249 Pin control in Bus Monitoring mode

26.3.1.7 Disable automatic retransmission

M_TTCAN supports automatic retransmission of frames that have lost arbitration or disturbed by errors during the transmission. By default, automatic retransmission is enabled. To support time-triggered communication (as described in ISO 11898-1:2015, chapter 9.2), the automatic retransmission is disabled via CCCR.DAR.

In DAR mode, all the transmissions are automatically canceled after they start on the CAN bus. The TX Request Pending bit TXBRP.TRPx is reset after the successful transmission, when a transmission has not yet started at the point of cancellation, is aborted due to lost arbitration, or when an error occurred during frame transmission.

- Successful transmission: Corresponding TX Buffer Transmission Occurred bit TXBTO.TOx is set while the Corresponding TX Buffer Cancellation Finished bit TXBCF.CFx is not set
- Successful transmission in spite of cancellation: Both the Corresponding TX Buffer Transmission Occurred bit TXBTO.TOx and Corresponding TX Buffer Cancellation Finished bit TXBCF.CFx are set
- Arbitration lost or frame transmission disturbed: Corresponding TX Buffer Transmission Occurred bit TXBTO.TOx is not set while Corresponding TX Buffer Cancellation Finished bit TXBCF.CFx is set

In successful frame transmissions, and if storage of TX events is enabled, a TX Event FIFO element is written with Event Type ET = 10 (transmission despite cancellation).

26.3.1.8 Power Down (Sleep mode)

The M_TTCAN channel can be set to a power down mode via Clock Stop Request (CTL.STOP_REQ). As long as clock stop request is active, the STOP_REQ bit is '1'.

When all the pending transmission requests are complete, the M_TTCAN waits until the bus idle state is detected. Then, M_TTCAN sets CCCR.INIT to '1' to prevent any further CAN transfers. Now the M_TTCAN acknowledges that it is ready for power down by setting clock stop acknowledge(STATUS.STOP_ACK). Upon receiving the ACK from the channel, the hardware automatically switches off the clock to their respective channels.

To leave power down mode, the application must reset CTL.STOP_REQ. The M_TTCAN acknowledges this by resetting STATUS.STOP_ACK. Later, the application restarts the CAN communication by resetting the bit CCCR.INIT.

When the clock stop request is triggered through CTL.STOP_REQ, it must not be cleared before STATUS.STOP_ACK bit is set.

***Note**: Do not use the TTCAN CCCR.CSR register for the power down control; instead, use CTL.STOP_REQ. Similarly, use STATUS.STOP_ACK instead of CCCR.CSA.*

26.3.1.9 Test mode

To enable write access to the TEST register, Test Mode Enable bit (CCCR.TEST) is set to '1'. This allows the configuration of the test modes and test functions.

Four output functions are available for the CAN transmit pin, canfd[0].ttcan_tx[0:1], by programming TEST.TX. Apart from its default function of serial data output, it can drive the CAN sample point signal to monitor the M_TTCAN's bit timing; it can also drive constant dominant or recessive values. The actual value at the canfd[0].ttcan_rx[0:1] pin can be read from TEST.RX. Both functions can be used to check the CAN bus physical layer.

Due to the synchronization mechanism between the CAN clock and the host clock domain, there may be a delay of several host clock periods between writing to TEST.TX until the new configuration is visible at the output pin, canfd[0].ttcan_tx[0:1].This applies also when reading the input pin canfd[0].ttcan_rx[0:1] via TEST.RX.

***Note**: Test modes should be used for production tests or self-test only. The software control for the canfd[0].ttcan_tx[0] pin interferes with all CAN protocol functions. It is not recommended to use test modes for application.*

26.3.1.9.1 External Loop Back mode

M_TTCAN can be set in External Loop Back mode by programming TEST.LBCK to '1'. In the Loop Back mode, the M_TTCAN treats its transmitted messages as received messages and stores (if they pass acceptance filtering) into an RX buffer or an RX FIFO. Figure 250 shows the connection of signals canfd[0].ttcan_tx[0:1] and canfd[0].ttcan_rx[0:1] to the M_TTCAN in External Loop Back mode.

This mode is provided for the hardware self-test. To be independent from external stimulation, the M_TTCAN ignores acknowledge errors (recessive bit sampled in the acknowledge slot of a data/remote frame) in the Loop Back mode. In this mode, M_TTCAN performs an internal feedback from the TX output to the RX input. The actual value of the canfd[0].ttcan_rx[0:1] input pin is disregarded by the M_TTCAN. The transmitted messages can be monitored at the canfd[0].ttcan_tx[0:1] pin.

26.3.1.9.2 Internal Loop Back mode

To enter the Internal Loop Back mode by programming the TEST.LBCK and CCCR.MON bits to '1'. This mode can be used for a “Hot Self-test”, meaning M_TTCAN can be tested without affecting a running CAN system connected to the `canfd[0].ttcan_tx[0:1]` and `canfd[0].ttcan_rx[0:1]` pins. In this mode, the `canfd[0].ttcan_rx[0:1]` pin is disconnected from the M_TTCAN and `canfd[0].ttcan_tx[0:1]` pin is held recessive. Figure 250 shows the connection of `canfd[0].ttcan_tx[0:1]` and `canfd[0].ttcan_rx[0:1]` to the M_TTCAN in Internal Loop Back mode.

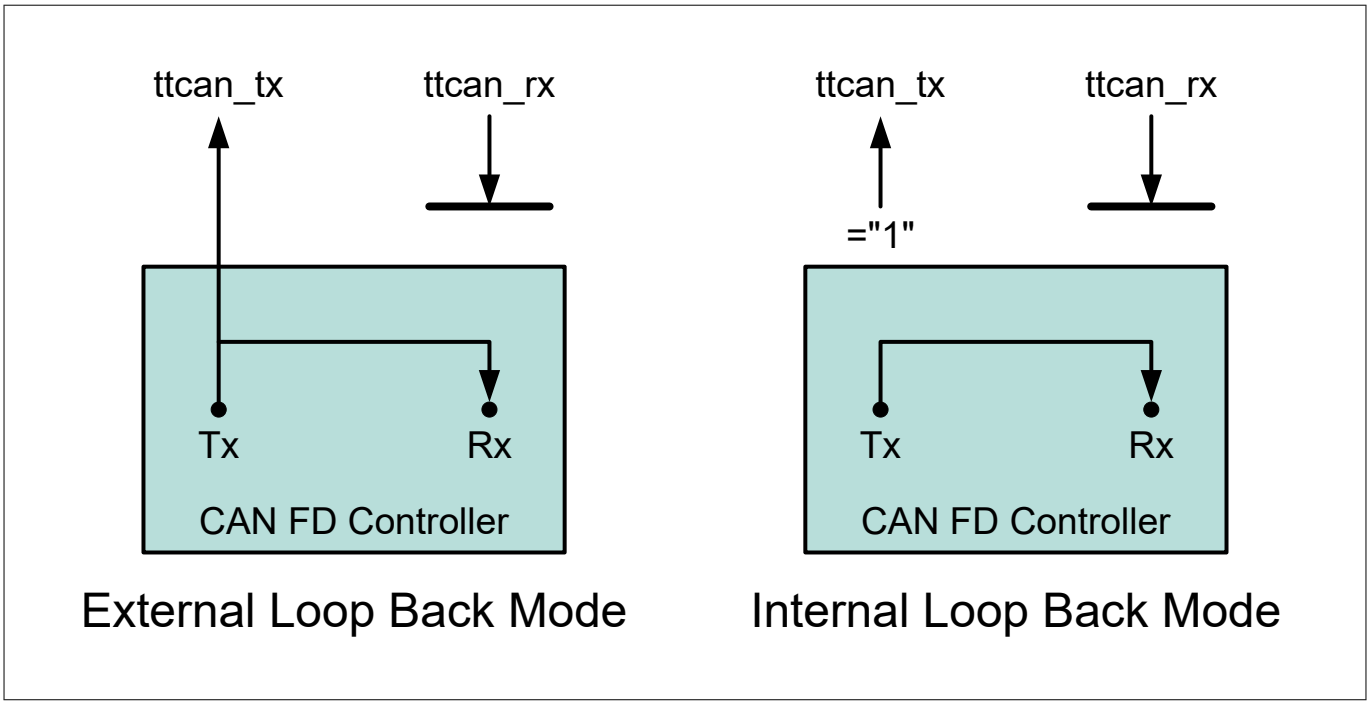


Figure 250 Pin control in Loop Back modes

26.3.1.10 Application watchdog

The application watchdog is served by reading the TTOST register. When the application watchdog is not served in time, the TTOST.AWE bit is set, all TTCAN communication is stopped, and the M_TTCAN is set into Bus Monitoring mode.

The TT application watchdog can be disabled by programming the Application Watchdog Limit TTOCF.AWL to '0x00'. The TT application watchdog should not be disabled in a TTCAN application program.

26.3.2 Timestamp generation

The M_TTCAN channel uses a 16-bit counter to record when messages are sent or received. This allows the application software to know the order in which events occurred.

The timestamp counter is configured through the TS_CTL register. The TS_CTL.ENABLED bit will enable the counter. Upon enabling, it will start incrementing according to the TS_CTL.PRESCALE [15:0]. The application can read the counter value through the TS_CNT register. Write access to the TS_CNT register will clear the TS_CNT.

When the timestamp counter is enabled, internal counter for prescaler counts with every cycle of clk_sys. When the counter value reaches the prescaler value, the timestamp counter increments by one and the internal prescaler counter is cleared. When TS_CTL.PRESCALE changes, TS_CNT should be written to reset them. This can make the internal prescaler counter follow a new value of TS_CTL.PRESCALE immediately.

The shared timestamp counter is a wrap-around counter. When the counter wraps around, IR.TSW for the M_TTCAN channel will be raised.

On start of frame reception/transmission, the timestamp counter value is captured and stored into the timestamp section of an RX buffer/RX FIFO (RXTS [15:0]) or TX Event FIFO (TXTS [15:0]) element.

***Note**: The counter value TS_CNT is not retained in Deep Sleep mode whereas the TS_CTL is retained.*

26.3.3 Timeout counter

To signal timeout conditions for RX FIFO 0, RX FIFO 1, and the TX Event FIFO, the M_TTCAN supplies a 16-bit timeout counter. It operates as a down-counter and uses the same prescaler controlled by TSCC. The TCP as the timestamp counter. A prescaler TSCC.TCP is configured to clock the timeout counter in multiples of CAN bit times (1…16). The timeout counter is configured via the register TOCC. The actual counter value can be read from TOCV.TOC.

The timeout counter can only be started while CCCR.INIT =0. It is stopped when CCCR.INIT = 1; for example, when the M_TTCAN enters Bus_Off state.

The operation mode is selected by TOCC.TOS. When operating in the Continuous mode, the counter starts when CCCR.INIT is reset. A write to TOCV presets the counter to the value configured by TOCC.TOP and continues down counting.

When the timeout counter is controlled by one of the FIFOs, an empty FIFO presets the counter to the value configured by TOCC.TOP. Down-counting is started when the first FIFO element is stored. Writing to TOCV has no effect.

When the counter reaches zero, the interrupt flag IR.TOO is set. In the Continuous mode, the counter is immediately restarted at TOCC.TOP.

***Note**: The clock signal for the timeout counter is derived from the CAN Core's sample point signal. Therefore, the time the timeout counter is decremented may vary due to the synchronization/resynchronization mechanism of the CAN Core. If the bit rate switch feature in CAN FD is used, the timeout counter is clocked differently in the arbitration and data fields.*

26.3.4 Receive handling

The RX handler controls the acceptance filtering, transfer of received messages to the RX buffers, or to one of the two RX FIFOs, as well as RX FIFO's put and get indices.

26.3.4.1 Acceptance filtering

The M_TTCAN offers the possibility to configure two sets of acceptance filters, one for standard identifiers and one for extended identifiers. These filters can be assigned to an RX buffer or to RX FIFO 0,1. For the acceptance filtering, each list of filters is executed from element #0 until the first match element. Acceptance filtering stops at the first matching element. The following filter elements are not evaluated for this message.

The main features are:

- Each filter element can be configured as
  - Range filter (from - to)
  - Filter for one or two dedicated IDs
  - Classic bit mask filter
- Each filter element is configurable for acceptance or rejection filtering
- Each filter element can be enabled/disabled individually
- Filters are checked sequentially; execution stops with the first matching filter element

Related configuration registers are:

- Global Filter Configuration (GFC)
- Standard ID Filter Configuration (SIDFC)
- Extended ID Filter Configuration (XIDFC)
- Extended ID AND Mask (XIDAM)

Depending on the configuration of the filter element (SFEC/EFEC) a match triggers one of the following actions:

- Store received frame in FIFO 0 or FIFO 1
- Store received frame in an RX buffer
- Store received frame in RX buffer and generate pulse at filter event pin
- Reject received frame
- Set High-Priority Message interrupt flag IR.HPM
- Set High-Priority Message interrupt flag IR.HPM and store the received frame in FIFO 0 or FIFO 1

Acceptance filtering is started after the complete identifier is received. After the acceptance filtering has completed, if a matching RX buffer or RX FIFO is found, the message handler starts to write the received message data in portions of 32 bits to the matching RX buffer or RX FIFO. If the CAN protocol controller has detected an error condition (such as CRC error), this message is discarded with the following impact on the affected RX buffer or RX FIFO:

RX buffer

The new data flag of the matching RX buffer is not set, but the RX buffer is (partly) overwritten with received data. For error types, see PSR.LEC and PSR.DLEC respectively.

RX FIFO

Put index of matching RX FIFO is not updated, but the related RX FIFO element is (partly) overwritten with received data. For error types, see PSR.LEC and PSR.DLEC, respectively. If the matching RX FIFO is operated in overwrite mode, the boundary conditions described in RX FIFO Overwrite Mode should be considered.

***Note**: When an accepted message is written to one of the two RX FIFOs, or into an RX buffer, the unmodified received identifier is stored independent of the filter(s) used. The result of the acceptance filter process depends on the sequence of configured filter elements.*

Range filter

The filter matches for all received frames with Message IDs in the range defined by SF1ID/SF2ID resp. EF1ID/EF2ID.

The two possibilities when range filtering is used with extended frames are:

- EFT = 00: The Message ID of received frames is ANDed with the XIDAM before the range filter is applied
- EFT = 11: The XIDAM is not used for range filtering

Filter for specific IDs

A filter element can be configured to filter one or two specific Message IDs. To filter a specific Message ID, the filter element should be configured with SF1ID = SF2ID andEF1ID = EF2ID, respectively.

Classic bit mask filter

Classic bit mask filtering is intended to filter groups of Message IDs by masking single bits of a received Message ID. With classic bit mask filtering, SF1ID/EF1ID is used as a Message ID filter, while SF2ID/EF2ID is used as a filter mask.

A zero-bit at the filter mask will mask the corresponding bit position of the configured ID filter; for example, the value of the received Message ID at that bit position is not relevant for acceptance filtering. Only those bits of the received Message ID where the corresponding mask bits are one are relevant for acceptance filtering.

If all mask bits are one, a match occurs only when the received Message ID and the Message ID filter are identical. If all mask bits are zero, all Message IDs match.

Standard message ID filtering

Figure 251 shows the flow for standard message ID (11-bit Identifier) filtering. The standard message ID Filter element is described in Standard message ID filter element.

It is controlled by the GFC and SIDFC message IDs, the Remote Transmission Request bit (RTR) and the Identifier Extension bit (IDE) of received frames are compared with the list of configured filter elements.

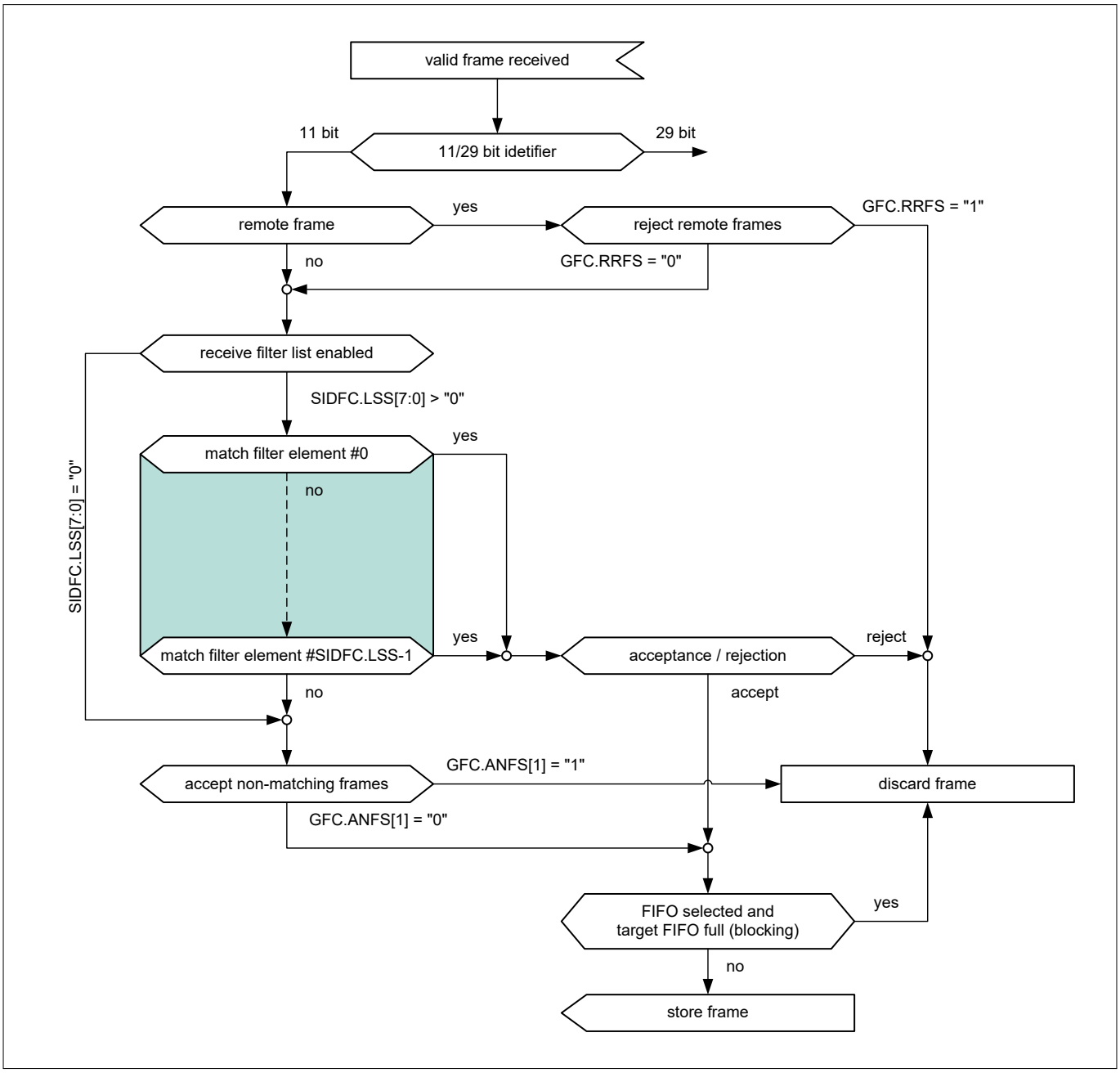


Figure 251 Standard message ID filter flow diagram

Extended message ID filtering

Figure 252 shows the flow for extended message ID (29-bit Identifier) filtering. The extended message ID filter element is described in Extended message ID filter element. It is controlled by the GFC and XIDFC Message IDs, the RTR bit, and IDE bit of received frames are compared against the list of configured filter elements. The Extended ID AND Mask XIDAM[28:0] is ANDed with the received identifier before the filter list is executed.

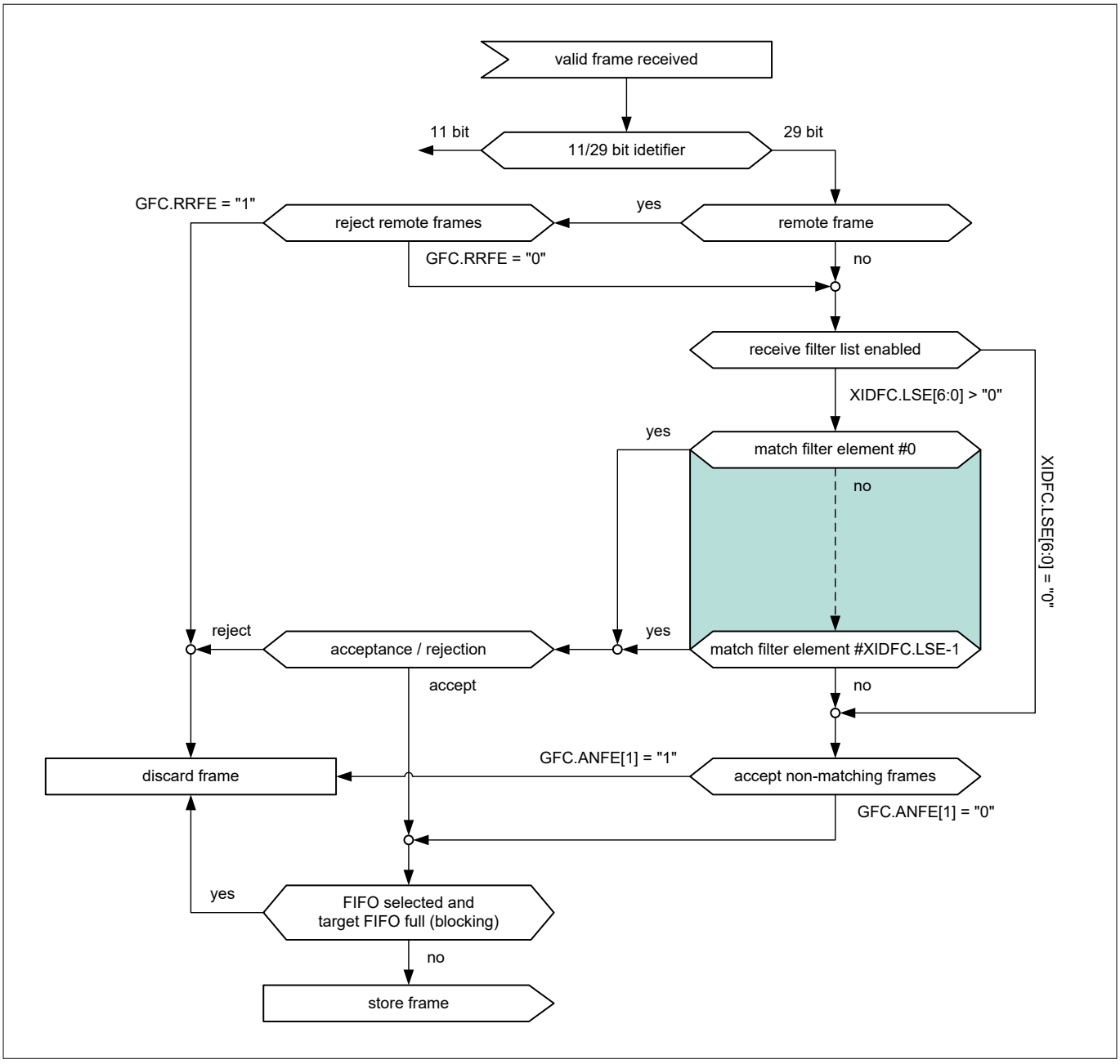


Figure 252 Extended message ID filter flow diagram

26.3.4.2 Receive FIFOs

RX FIFO 0 and RX FIFO 1 is configured to hold up to 64 elements each. The two RX FIFOs are configured via the RXF0C and RXF1C registers.

The received messages that pass acceptance filtering are transferred to the RX FIFO as configured by the matching filter element. For a description of the filter mechanisms available for RX FIFO 0 and RX FIFO 1, see Acceptance filtering. The RX FIFO element is described in RX buffer and FIFO element.

To avoid an RX FIFO overflow, the RX FIFO watermark can be used. When the RX FIFO fill level reaches the RX FIFO watermark configured by RXFnC.FnWM, interrupt flag IR.RFnW is set. When the RX FIFO Put Index reaches the RX FIFO Get Index, an RX FIFO Full condition is signaled by RXFnS.FnF. In addition, the interrupt flag IR.RFnF is set. The FIFO watermark interrupt flags can be used to trigger the DMA. DMA request for FIFO will remain set

until their respective trigger is cleared by software. Software can clear the trigger by clearing the watermark flag.

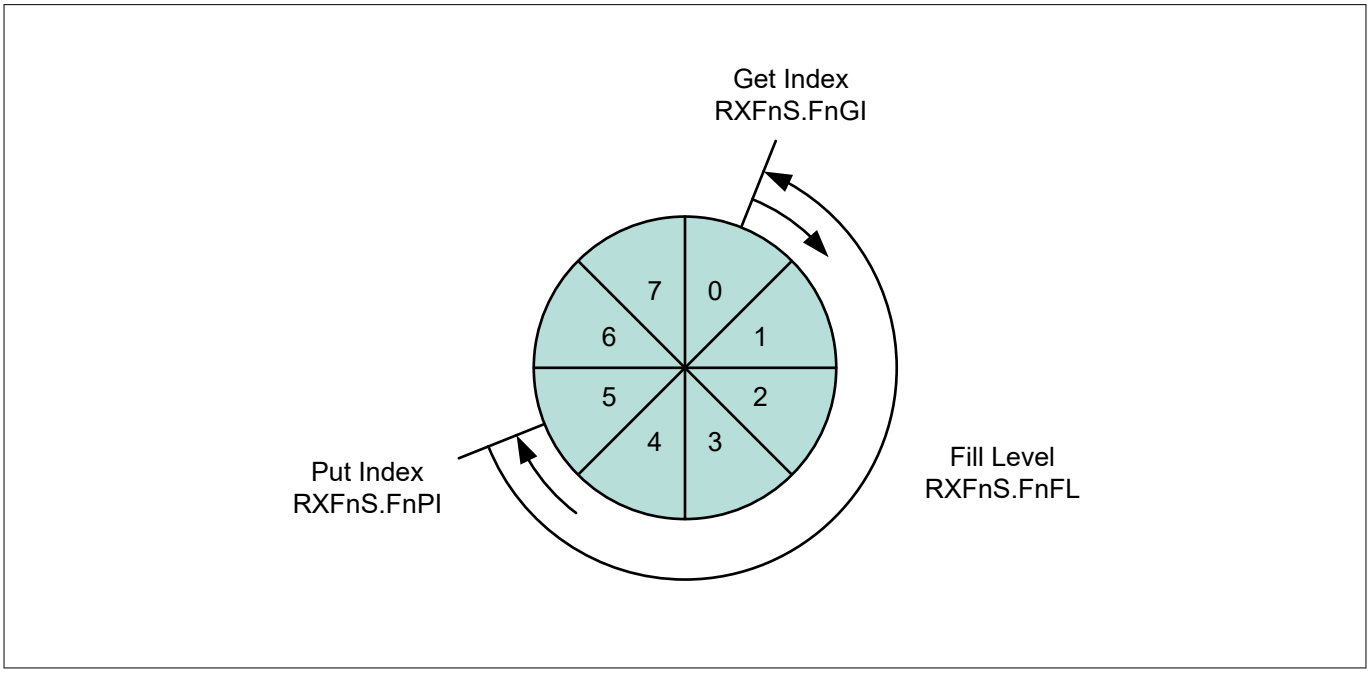


Figure 253 RX FIFO status

When reading from an RX FIFO, RX FIFO Get Index RXFnS.FnGI x FIFO element size has to be added to the corresponding RX FIFO start address RXFnC.FnSA. RXFIFO Top pointer logic is added to the CAN FD controller to make reading faster. See RX FIFO top pointer on RX FIFO top pointer.

Table 230 RX buffer/FIFO element size

RXESC.RBDS[2:0] RXESC.FnDS[2:0]	Data field [bytes]	FIFO element size [RAM words]
000	8	4
001	12	5
010	16	6
011	20	7
100	24	8
101	32	10
110	48	14
111	64	18

26.3.4.2.1 RX FIFO blocking mode

The RX FIFO blocking mode is configured by RXFnC.FnOM= 0. This is the default operation mode for RX FIFOs.

When an RX FIFO full condition is reached (RXFnS.FnPI = RXFnS.FnGI), no further messages are written to the corresponding RX FIFO until at least one message is read and the RX FIFO get index is incremented. An RX FIFO full condition is signaled by RXFnS.FnF = 1. In addition, the interrupt flag IR.RFnF is set.

If a message is received while the corresponding RX FIFO is full, this message is discarded and the message lost condition is signaled by RXFnS.RFnL = 1. In addition, the interrupt flag IR.RFnL is set.

26.3.4.2.2 RX FIFO overwrite mode

The RX FIFO overwrite mode is configured by RXFnC.FnOM = 1. When an RX FIFO full condition (RXFnS.FnPI =RXFnS.FnGI) is signaled by RXFnS.FnF = 1, the next message accepted for the FIFO will overwrite the oldest FIFO message. Put and Get Indices are both incremented by one.

When an RX FIFO is operated in the overwrite mode and an RXFIFO full condition is signaled, reading of the RX FIFO elements should start at least at Get Index + 1. This is because a received message may be written to the MRAM (Put Index) while the CPU is reading from the MRAM (GetIndex). In this case, inconsistent data may be read from the respective RX FIFO element. Adding an offset to the GetIndex when reading from the RX FIFO avoids this problem. The offset depends on how fast the CPU accesses the RXFIFO. Figure 254 shows an offset of two with respect to the get index when reading the RX FIFO. In this case, the two messages stored in element 1 and 2 are lost.

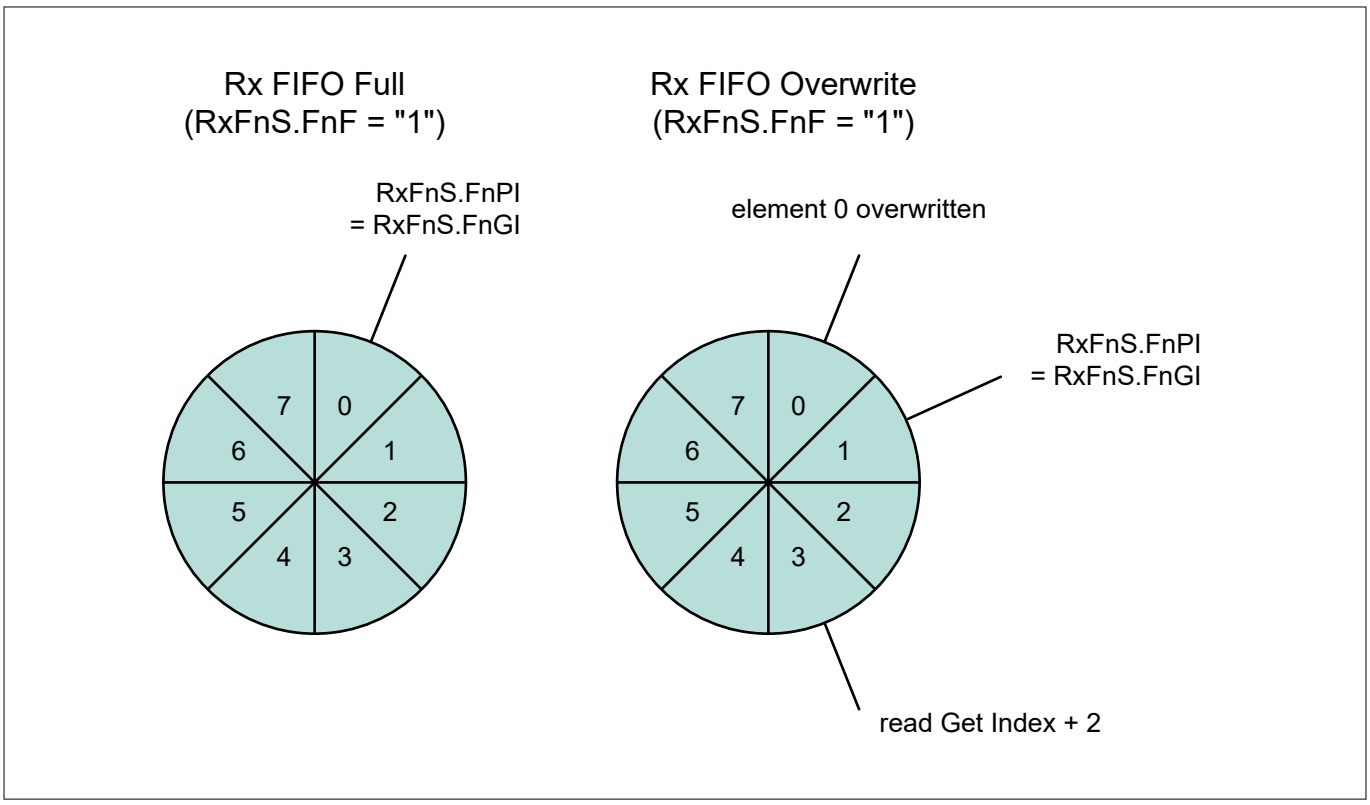


Figure 254 RX FIFO overflow handling

After reading from the RX FIFO, the number of the last element read must be written to the RX FIFO Acknowledge Index RXFnA.FnA. This increments the Get Index to that element number. If the Put Index is not incremented to this RX FIFO element, the RX FIFO full condition is reset (RXFnS.FnF = 0).

26.3.4.2.3 RX FIFO top pointer

M_TTCAN supports two receive FIFOs. Reading from these FIFOs requires the following steps:

- Retrieve read pointer
- Calculate correct MRAM address
- Read the data from MRAM
- Update the read pointer

To avoid these steps, RX FIFO top pointer logic is integrated in the CAN FD controller. It provides a single MMIO location (RXFTOPn_DATA; n = 0,1) to read the data from. Using such hardware logic has the following benefits:

- Higher performance data access
- Less bus traffic
- Reduced CPU load
- Reduced power
- Enables DMA access to FIFO

This logic is enabled when RXFTOP_CTL.FnTPE is set. Setting this bit enables the logic to set the FIFO top address (FnTA) and internal message word counter. The receive FIFO in the Top Status register (RXFTOPn_STAT) shows the respective FIFO top address and RXFTOPn_DATA provides the data located at the top address. See the register definitions for more details on both registers.

If RXFTOPn_DATA is read, the top pointer logic also updates the RX FIFO Acknowledge Index (RXFnA.FnA) in TTCAN channel.

***Note**: Top pointer logic is disabled when the channel is being configured (CCCR.CCE = 1). Reading RXFTOPn_DATA while the logic is disabled will return the invalid data.*

***Note**: When RX FIFO size is set to '1', do not use the RXFIFO top pointer function. Instead, read the received messages from MRAM directly.*

26.3.4.3 Dedicated RX buffers

The M_TTCAN supports up to 64 dedicated RX buffers. The start address of the dedicated RX buffer section is configured via RXBC.RBSA.

For each RX buffer, a standard or extended message ID Filter Element with SFEC/EFEC = 111 and SFID2/ EFID2[10:9] = 00 must be configured (see Standard message ID filter element and Extended message ID filter element).

After a received message is accepted by a filter element, the message is stored into the RX buffer in the message RAM referenced by the filter element. The format is the same as for an RX FIFO element. In addition, the flag IR.DRX (message stored in a dedicated RX buffer) in the interrupt register is set.

Table 231 Example filter configuration for RX buffers

Filter element	SFID1[10:0] EFID1[28:0]	SFID2[10:9] EFID2[10:9]	SFID2[5:0] EFID2[5:0]
0	ID message 1	00	00 0000
1	ID message 1	00	00 0001
2	ID message 1	00	00 0010

After the last word of a matching received message is written to the message RAM, the respective New data flag in NDAT1 and two registers are set. As long as the New data flag is set, the respective RX buffer is locked against updates from received matching frames. The New data flags is reset by the host by writing a '1' to the respective bit position.

While an RX buffer's New data flag is set, a message ID filter element referencing the specific RX buffer will not match, causing the acceptance filtering to resume. The following message ID filter elements causes the received message to be stored into another RX buffer, or into an RX FIFO, or the message may be rejected, depending on the filter configuration.

- Reset interrupt flag IR.DRX
- Read New Data registers
- Read messages from message RAM
- Reset New Data flags of processed messages

26.3.4.4 Debug on CAN support

The debug messages are stored into the RX buffers; three consecutive RX buffers (for example, #61, #62, and #63) are used to store debug messages A, B, and C. The format is the same for RX buffer and RX FIFO elements.

To filter the debug messages Standard/Extended Filter Elements with SFEC/EFEC = 111 is set. The messages that match these filter elements are stored into the RX buffers addressed by SFID2/EFID2[5:0].

After message C is stored, the DMA request is activated and the three messages can be read from the message RAM under DMA control. The RAM words holding the debug messages will not be changed by the M_TTCAN while the DMA request is activated. The behavior is similar to that of an RX buffer with its New data flag set.

After the DMA transfer is completed, an acknowledge from DMA resets the DMA request. Now the M_TTCAN is prepared to receive the next set of debug messages.

26.3.4.4.1 Filtering debug messages

The debug messages are filtered by configuring one Standard/Extended Message ID filter element for each of the three debug messages. To enable a filter element to filter debug messages, SFEC/EFEC should be programmed to “111”. Int his case the SFID1/SFID2 and EFID1/EFID2 fields have a different meaning (see Figure 251 Element and Figure 252 Element). While SFID2/EFID2[10:9] controls the debug message handling state machine, SFID2/EFID2[5:0] controls the storage location of a received debug message.

When a debug message is stored, neither the respective New data flag nor IR.DRX are set. The reception of debug messages can be monitored via RXF1S.DMS.

Table 232 Example filter configuration for debug message

Filter element	SFID1[10:0] EFID1[28:0]	SFID2[10:9] EFID2[10:9]	SFID2[5:0] EFID2[5:0]
0	ID debug message A	01	11 1101
1	ID debug message B	10	11 1110
2	ID debug message C	11	11 1111

26.3.4.4.2 Debug message handling

The debug message handling state machine assures that the debug messages are stored to three consecutive RX buffers in the correct sequence. If there are missing messages, the process is restarted. The DMA request is activated only when all three debug messages A, B, and C are received in an incorrect order.

The status of the debug message handling state machine is signaled via RXF1S.DMS.

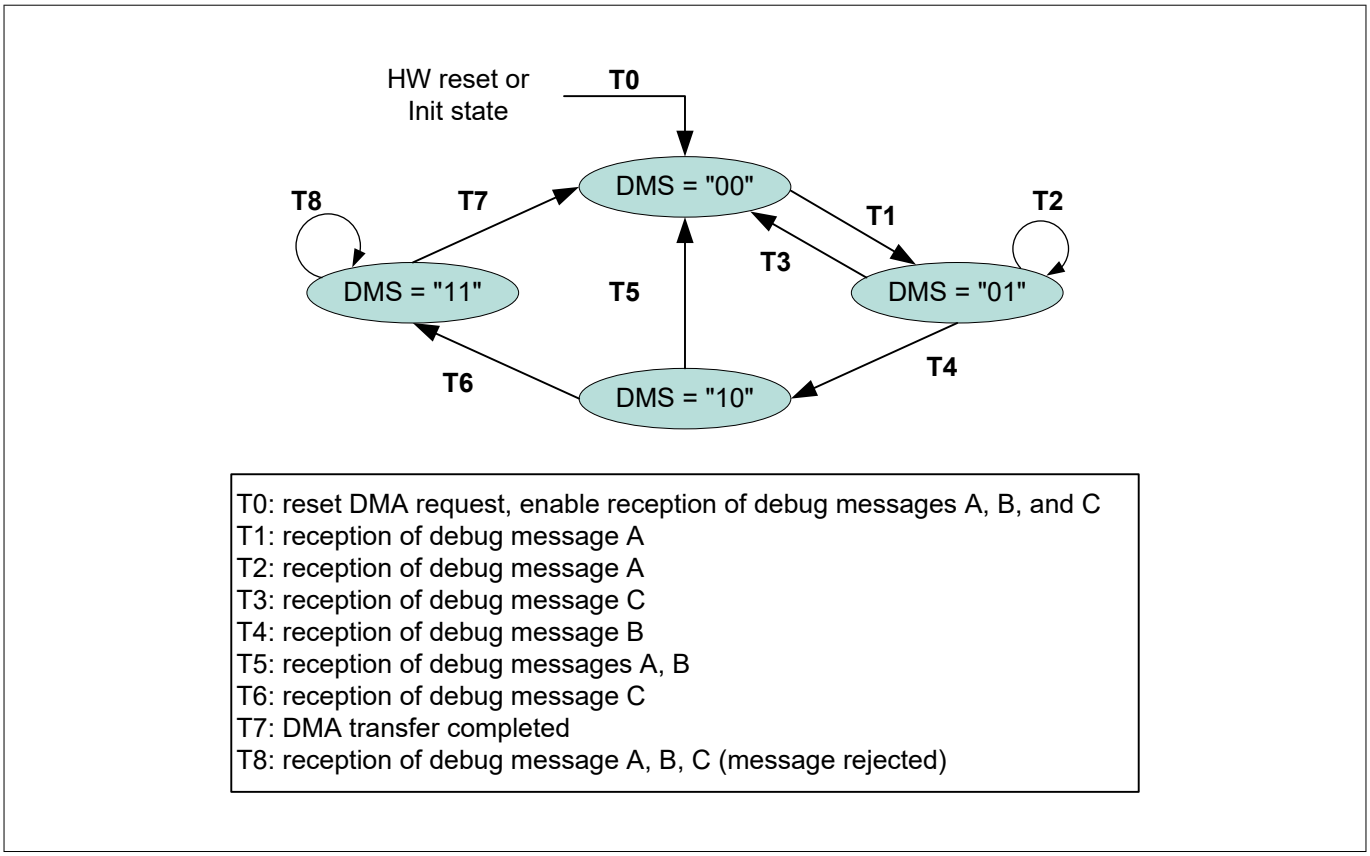


Figure 255 Debug message handling state machine

26.3.5 Transmission handling

The TX handler handles the transmission requests for the dedicated TX buffers, TX FIFO, and TX queue. It controls the transfer of transmit messages to the CAN core, the Put and Get Indices, and the TX Event FIFO. Up to 32 TX buffers can be set up for message transmission. The CAN mode for transmission (classic CAN or CAN FD) is configured separately for each TX buffer element. The TX buffer element is described in TX buffer element. Table 233 describes the possible configurations for frame transmission.

Table 233 Possible configuration for frame transmission

CCR		TX buffer element		Frame transmission
BRSE	FDOE	FDF	BRS	
ignored	0	ignored	ignored	Classic CAN
0	1	0	ignored	Classic CAN
0	1	1	ignored	FD without bit rate switching
1	1	0	ignored	Classic CAN
1	1	1	0	FD without bit rate switching
1	1	1	1	FD with bit rate switching

The TX handler starts a TX scan to check for the highest priority pending TX request (TX buffer with lowest message ID) when the TX Buffer Request Pending register (TXBRP) is updated, or when a transmission is started.

26.3.5.1 Transmit pause

The transmit pause feature is intended for use in CAN systems where the CAN message identifiers are (permanently) assigned to specific values and cannot be changed easily. These message identifiers may have a higher CAN arbitration priority than other defined messages, while in a specific application, their relative arbitration priority should be inverse. This may lead to a case where one Electronic Control Unit (ECU) sends a burst of CAN messages that cause another ECUs CAN messages to be delayed because the other messages have a lower CAN arbitration priority.

For example, if CAN ECU-1 has the transmit pause feature enabled and is requested by the application software to transmit four messages, it will, after the first successful message transmission, wait for two nominal bit times of bus idle before it is allowed to start the next requested message. If there are other ECUs with pending messages, those messages are started in the idle time, they will not need to arbitrate with the next message of ECU-1. After having received a message, ECU-1 is allowed to start its next transmission as soon as the received message releases the CAN bus.

The transmit pause feature is controlled by the Transmit Pause bit (CCCR.TXP). If the bit is set, the M_TTCAN controller will, each time it has successfully transmitted a message, pause for two nominal bit times before starting the next transmission. This enables other CAN nodes in the network to transmit messages even if their messages have lower prior identifiers. Default is transmitted pause disabled(CCCR.TXP = 0).

This feature loses burst transmissions coming from a single node and protects against “babbling idiot” scenarios where the application program erroneously requests too many transmissions.

26.3.5.2 Dedicated TX buffers

The dedicated TX buffers are intended for message transmission under complete control of the CPU. Each dedicated TX buffer is configured with a specific message ID. If multiple TX buffers are configured with the same message ID, the TX buffer with the lowest buffer number is transmitted .

If the data section is updated, a transmission is requested by an Add Request via TXBAR.ARn. The requested messages arbitrate internally with messages from an optional TX FIFO or TX queue and externally with messages on the CAN bus, and are sent out according to their message ID.

26.3.5.2.1 Addressing dedicated TX buffers

A dedicated TX buffer allocates an element size of 32-bit words in the message RAM as shown in the Table 234. Therefore, the start address of a dedicated TX buffer in the message RAM is calculated by (transmit buffer index (0 to 31) × Element Size) + TX Buffers Start Address (TXBC.TBSA[15:2]).

Table 234 TX Buffer/FIFO/queue element size

TXESC.TBDS [2:0]	Data field [bytes]	Element size [RAM words]
000	8	4
001	12	5
010	16	6
011	20	7

(table continues...)

Table 234 (continued) TX Buffer/FIFO/queue element size

TXESC.TBDS [2:0]	Data field [bytes]	Element size [RAM words]
100	24	8
101	32	10
110	48	14
111	64	18

26.3.5.3 Transmit FIFO

The TX FIFO operation is configured by programming TXBC.TFQM to '0'. Messages stored in the TX FIFO are transmitted starting with the message referenced by the Get Index TXFQS.TFGI. After each transmission, the Get Index is incremented cyclically until the TX FIFO is empty. The TXFIFO enables transmission of messages with the same message ID from different TX buffers in the order that these messages are written to the TX FIFO. The M_TTCAN calculates the TX FIFO Free Level TXFQS.TFFL as the difference between Get and Put Index. It indicates the number of available (free) TX FIFO elements.

New transmit messages must be written to the TX FIFO starting with the TX buffer referenced by the Put Index TXFQS.TFQPI. An Add Request increments the Put Index to the next free TX FIFO element. When the Put Index reaches the Get Index, TX FIFO Full (TXFQS.TFQF = 1) is signaled. In this case, no further messages should be written to the TX FIFO until the next message is transmitted and the Get Index is incremented.

When a single message is added to the TX FIFO, the transmission is requested by writing '1' to the TXBAR bit related to the TX buffer referenced by the TX FIFO's Put Index.

When multiple (n) messages are added to the TX FIFO, they are written to 'n' consecutive TX buffers starting with the Put Index. The transmissions are then requested via TXBAR. The Put Index is then cyclically incremented by n. The number of requested TX buffers should not exceed the number of free TX buffers as indicated by the TX FIFO Free Level.

When a transmission request for the TX buffer referenced by the Get Index is canceled, the Get Index is incremented to the next TX buffer with pending transmission request and the TX FIFO Free Level is recalculated. When transmission cancellation is applied to any other TX buffer, the Get Index and the FIFO Free Level remain unchanged.

A TX FIFO element allocates element size 32-bit words in the message RAM as shown in Table 234. Therefore, the start address of the next available (free) TX FIFO buffer is calculated by adding TX FIFO/Queue Put Index TXFQS.TFQPI (0…31) × Element Size to the TX buffer Start Address TXBC.TBSA.

26.3.5.4 TX queue

The TX queue operation is configured by programming TXBC.TFQM to '1'. Messages stored in the TX queue are transmitted starting with the message with the lowest message ID (highest priority). If multiple queue buffers are configured with the same message ID, the queue buffer with the lowest buffer number is transmitted first.

New messages must be written to the TX buffer referenced by the Put Index TXFQS.TFQPI. An Add Request cyclically increments the Put Index to the next free TX buffer. If the TX queue is full (TXFQS.TFQF = 1), the Put Index is not valid to write any further message till one of the requested messages is sent or a pending transmission request is cancelled.

The application may use the TXBRP register instead of the Put Index and may place messages to any TX buffer without pending transmission requests.

A TX queue buffer allocates element size of 32-bit words in the message RAM as shown in Table 234. Therefore, the start address of the next available (free) TX queue buffer is calculated by adding TX FIFO/queue Put Index TXFQS.TFQPI (0…31) × Element Size to the TX buffer start address TXBC.TBSA.

26.3.5.5 Mixed dedicated TX buffers/TX FIFO

In this case, the TX buffers section in the MRAM is subdivided into a set of dedicated TX buffers and a TX FIFO. The number of dedicated TX buffers is configured by TXBC.NDTB. The number of TX buffers assigned to the TX FIFO is configured by TXBC.TFQS. If TXBC.TFQS is programmed to '0', only the dedicated TX buffers are used.

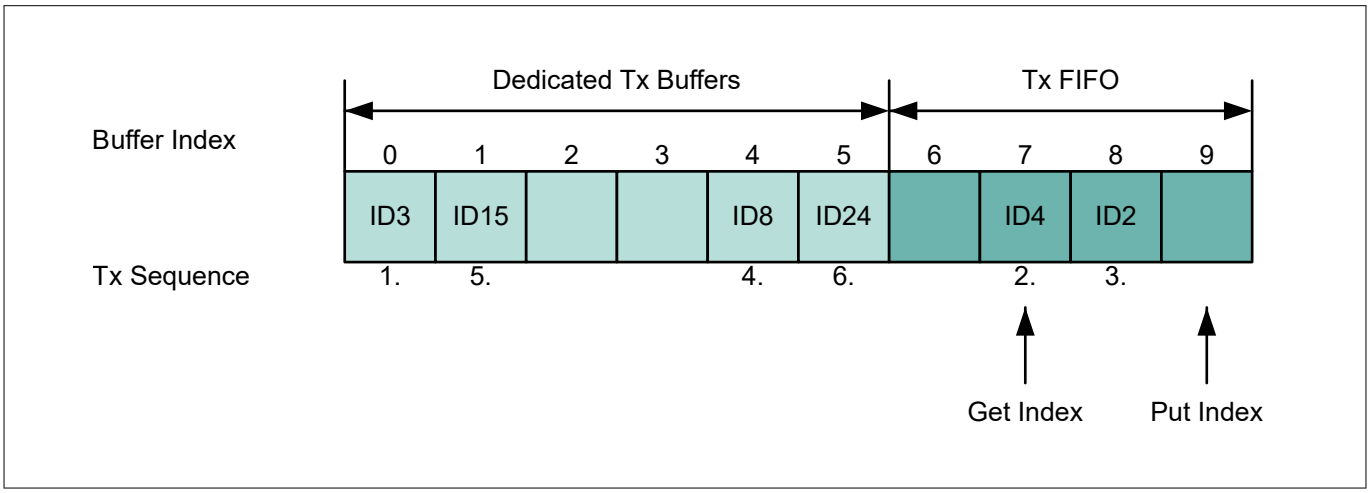


Figure 256 Example of mixed configuration dedicated TX buffers/TX FIFO

TX prioritization:

- Scan dedicated TX buffers and oldest pending TX FIFO buffer (referenced by TXFS.TFGI)
- Scan dedicated TX buffers and oldest pending TX FIFO buffer (referenced by TXFS.TFGI)

26.3.5.6 Mixed dedicated TX buffers/TX queue

In this case, the TX buffers section in the MRAM is subdivided into a set of dedicated TX buffers and a TX queue. The number of dedicated TX buffers is configured by TXBC.NDTB. The number of TX queue buffers is configured by TXBC.TFQS. In the case TXBC.TFQS is programmed to'0', only dedicated TX buffers are used.

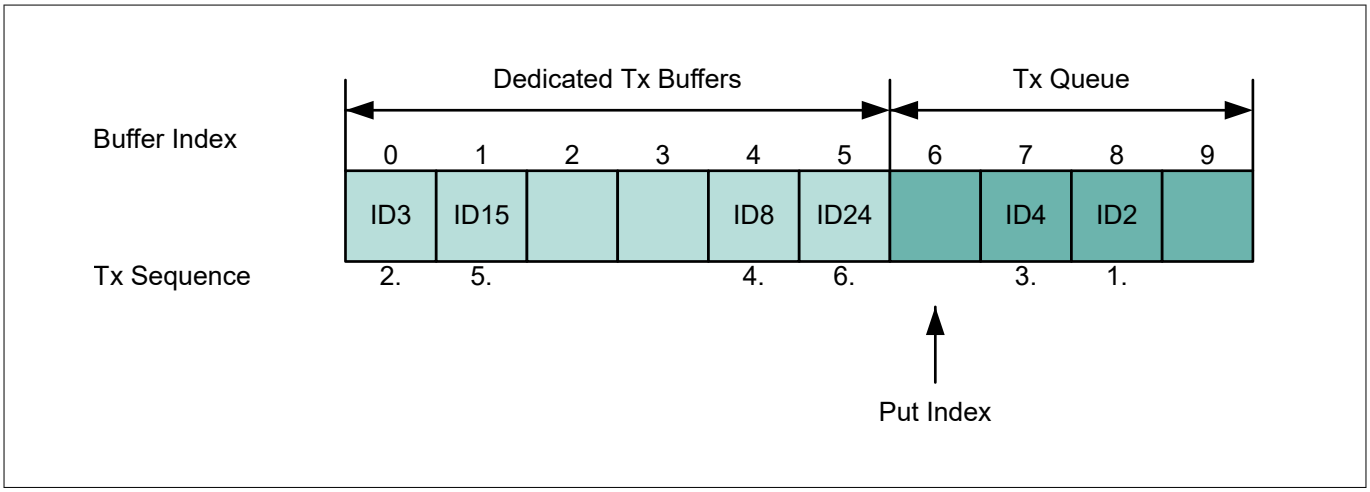


Figure 257 Example of mixed configuration dedicated TX buffers/TX queue

TX prioritization:

- Scan all TX buffers with the activated transmission request
- TX buffer with the lowest message ID gets the highest priority and is transmitted next

26.3.5.7 Transmit cancellation

M_TTCAN supports transmit cancellation. This feature is especially intended for gateway applications and AUTOSAR-based applications. To cancel a requested transmission from a dedicated TX buffer or a TX queue buffer the host must write '1' to the corresponding bit position (number of TX buffers) of the TXBCR register. The transmit cancellation is not intended for TX FIFO operation.

Successful cancellation is signaled by setting the corresponding bit of the TXBCF register to '1'.

In case the transmit cancellation is requested while transmission from a TX buffer is ongoing, the corresponding TXBRP bit remains set as long as the transmission is in progress. If the transmission was successful, the corresponding TXBTO and TXBCF bits are set. Otherwise, it is not repeated and only the corresponding TXBCF bit is set.

***Note**: If a pending transmission is canceled immediately before this transmission is started, there follows a short time window where no transmission is started even if another message is also pending in this node. This may enable another node to transmit a message, which may have a lower priority than the second message in this node.*

26.3.5.8 Transmit event handling

To support TX event handling, M_TTCAN has implemented a TX event FIFO. After M_TTCAN has transmits a message on the CAN bus, the message ID and time stamp are stored in a TX event FIFO element. To link a TX event to the TX event FIFO element, the message marker from the transmitted TX buffer is copied into the TX event FIFO element.

The TX event FIFO can be configured to a maximum of 32 elements. The TX event FIFO element is described in TX event FIFO element.

The purpose of the TX event FIFO is to decouple handling transmit status information from transmit message handling; i.e., a TX buffer holds only the message to be transmitted, while the transmit status is stored separately in the TX event FIFO. This has a advantage, especially when operating a dynamically managed transmit queue, that a TX buffer can be used for a new message immediately after successful transmission. There is no need to save transmit status information from a TX buffer before overwriting that TX buffer.

When a TX event FIFO full condition is signaled by IR.TEFF, no further elements are written to the TX event FIFO until one element is read and the TX event FIFO Get Index is incremented. In case the TX event occurs while the TX event FIFO is full, it is discarded and interrupt flag IR.TEFL is set.

To avoid the TX event FIFO overflow, use the TX event FIFO watermark. When the TX event FIFO fill level reaches the TX event FIFO watermark configured by TXEFC.EFWM, interrupt flag IR.TEFW is set.

When reading from the TX event FIFO, the TX event FIFO Get Index TXEFS.EFGI must be added twice to the TX event FIFO start address TXEFC.EFSA.

26.3.6 FIFO acknowledge handling

The Get Indices of RX FIFO 0, RX FIFO 1, and the TX event FIFO are controlled by the corresponding FIFO acknowledge index.

When RX FIFO top pointer hardware logic is used, it updates the RX FIFO acknowledge index. After RXFTOPn_DATA is read, the Acknowledge Index (RXFnA.FnA) is updates automatically, which will eventually set the FIFO get index to the FIFO acknowledge index plus one and thereby updates the FIFO fill level.

When the application does not use RX FIFO top pointer logic, update the Acknowledge Index uses one of the following two use cases:

- When only a single element is read from the FIFO (pointed by the Get Index), the Get Index value is written to the FIFO acknowledge index
- When a sequence of elements is read from the FIFO, it is sufficient to write the FIFO acknowledge index only once at the end of that read sequence (value is the index of the last element read), to update the FIFOs GetIndex

Note that the high-priority message from one of the two RX FIFOs, since the CPU has free access to the M_TTCAN's message RAM. In this case, the FIFO acknowledge index should is not written because this will set the Get Index to a wrong position and alters the FIFOs fill level with a loss in older FIFO elements.

***Note**: The application must ensure that a valid value is written to the FIFO acknowledge index. M_TTCAN does not check for erroneous values.*

26.3.7 Configuring the CAN bit timing

Each node in the CAN network has its own clock generator (usually a Quartz oscillator). The time parameter of the bit time can be configured individually for each CAN node. Even if each CAN node's oscillator has a different period, a common bit rate can be generated.

The oscillator frequencies vary slightly because of changes in temperature or voltage, or deterioration of components. As long as the frequencies vary only within the tolerance range of the oscillators, the CAN nodes can compensate for the different bit rates by resynchronizing to the bit stream.

26.3.7.1 CAN bit timing

The CAN FD operation defines two bit times – nominal bit time and data bit time. The nominal bit time is for the arbitration phase. The data bit time has an equal or shorter length and can be used to accelerate the data phase (see CAN FD operation).

The basic construction of a bit time is shared with both the nominal and data bit times. The bit time can be divided into four segments according to the CAN specifications (see Figure 258: the synchronization segment (Sync_Seg), the propagation time segment (Prop_Seg), the phase buffer segment 1 (Phase_Seg1), and the phase buffer segment 2(Phase_Seg2). The sample point at which the bus level is read and interpreted as the value of that respective bit, is located at the end of Phase_Seg1.

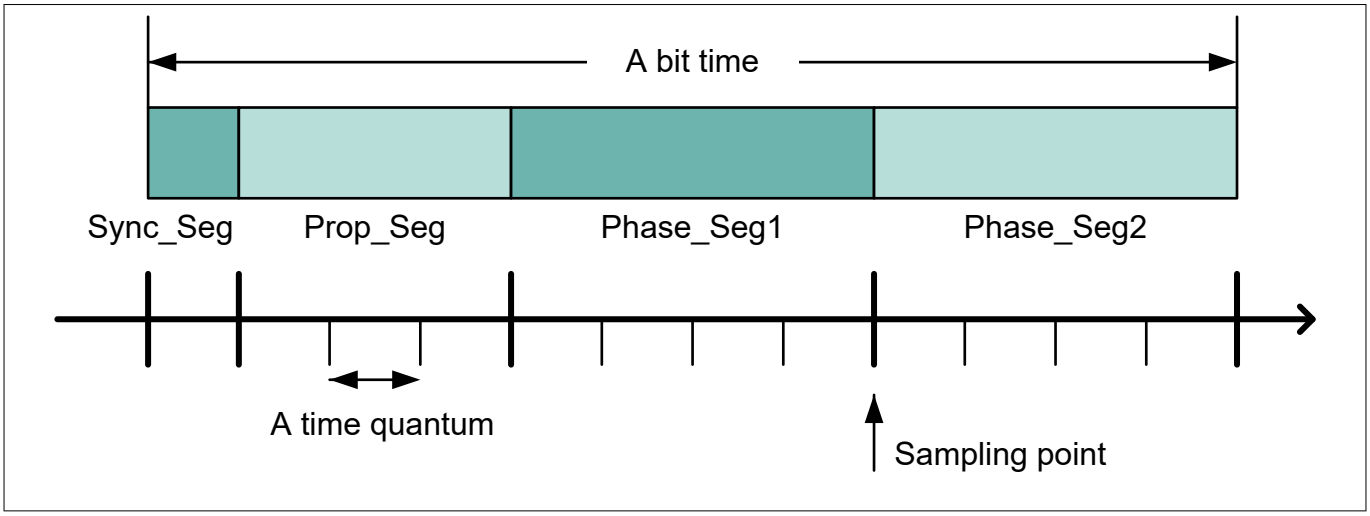


Figure 258 Bit time construction

Each segment consists of a programmable number of time quanta, which is a multiple of the time quantum that is defined by clk_can and a prescaler. The values and prescalers used to define these parameters differ for the nominal and data bit times, and are configured by Nominal Bit Timing and Prescaler register (NBTP) and Data Bit Timing and Prescaler register (DBTP) as shown in Table 235.

Table 235 Bit time parameters

Parameter	Description
Time quantum tq (nominal) and tqd (data)	Time quantum is derived by multiplying the basic unit time quanta (the clk_can period) with the respective prescaler. The time quantum is configured by the CAN FD controller as nominal: tq = (NBTP.NBRP[8:0] +1) × clk_can period data: tqd = (DBTP.DBRP[4:0] + 1) × clk_can period
Sync_Seg	Sync_Seg is fixed to one time quantum as defined by the CAN specifications and is not configurable (inherently built into the CAN FD controller). nominal: 1 tq data: 1 tqd
Prop_Seg	Prop_Seg is the part of the bit time that is used to compensate for the physical delay times within the network. The CAN FD controller configures the sum of Prop_Seg and Phase_Seg1 with a single parameter: nominal: Prop_Seg + Phase_Seg1 = NBTP.NTSEG1[7:0] + 1 data: Prop_Seg + Phase_Seg1 = DBTP.DTSEG1[4:0] + 1
Phase_Seg1	Phase_Seg1 is used to compensate for edge phase errors before the sampling point. It is lengthened by the resynchronization jump width. The sum of Prop_Seg and Phase_Seg1 is configured by the CAN FD controller as nominal: NBTP.NTSEG1[7:0] + 1 data: DBTP.DTSEG1[4:0] + 1
Phase_Seg2	Phase_Seg2 is used to compensate for edge phase errors after the sampling point. It is shortened by the resynchronization jump width. Phase_Seg2 is configured by the CAN FD controller as nominal: NBTP.NTSEG2[6:0] + 1 data: DBTP.DTSEG2[3:0] + 1
SJW	Resynchronization Jump Width. Used to adjust the length of Phase_Seg1 and Phase_Seg2. SJW will not be longer than either Phase_Seg1 or Phase_Seg2. SJW is configured by the CAN FD controller as nominal: NBTP.NSJW[6:0] + 1 data: DBTP.DSJW[3:0] + 1

These relations result in the following equations for the nominal and data bit times:

Nominal bit time

= [Sync_Seg + Prop_Seg + Phase_Seg1 + Phase_Seg2] × tq

= [1 + (NBTP.NTSEG1[7:0] + 1) + (NBTP.NTSEG2[6:0] + 1)] × [(NBTP.NBRP[8:0] +1) × clk_can period]

Data bit time

= [1 + (DBTP.DTSEG1[4:0] + 1) + (DBTP.DTSEG2[3:0] + 1)] × [(DBTP.DBRP[4:0] +1) × clk_can period]

***Note**: The information processing time (IPT) of the CAN FD controller is '0'; means that the data for the next bit is available at the first CAN clock edge after the sample point. Therefore, the IPT does not have to be accounted for when configuring Phase_Seg2, which is the maximum of Phase_Seg1 and the IPT.*

26.3.7.2 CAN bit rate

The bit rate is the inverse of bit time; therefore, the nominal bit rate is

1 / [{1 + (NBTP.NTSEG1[7:0] + 1) + (NBTP.NTSEG2[6:0] + 1)} × {(NBTP.NBRP[8:0] +1) × clk_can period}]

and the data bit rate is

1 / [{1 + (DBTP.DTSEG1[4:0] + 1) + (DBTP.DTSEG2[3:0] + 1)} × {(DBTP.DBRP[4:0] +1) × clk_can period}]

From these formulas, see that the bit rates of the CAN FD controller depends on the CAN clock (clk_can) period, and the range each parameter can be configured. The following tables list examples of the configurable bit rates at varying CAN clock frequencies. Empty boxes indicate that the desired bit rate are not configured at the specified input CAN clock frequency.

CAN clock frequency	8MHz		10MHz		16MHz		20MHz		32MHz		40MHz	
configuration / nominal bit rate	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1
125Kbps	64tq 32tq 16tq 8tq	1 2 4 8	80tq 40tq 20tq 10tq	1 2 4 8	128tq 64tq 32tq 16tq 8tq	1 2 4 8 16	160tq 80tq 40tq 20tq 10tq	1 2 4 8 16	256tq 128tq 64tq 32tq 16tq 8tq	1 2 4 8 16 32	320tq 160tq 80tq 40tq 20tq 10tq	1 2 4 8 16 32
250Kbps	32tq 16tq 8tq	1 2 4	40tq 20tq 10tq	1 2 4	64tq 32tq 16tq 8tq	1 2 4 8	80tq 40tq 20tq 10tq	1 2 4 8	128tq 64tq 32tq 16tq 8tq	1 2 4 8 16	160tq 80tq 40tq 20tq 10tq	1 2 4 8 16
500Kbps	16tq 8tq	1 2	20tq 10tq	1 2	32tq 16tq 8tq	1 2 4	40tq 20tq 10tq	1 2 4	64tq 32tq 16tq 8tq	1 2 4 8	80tq 40tq 20tq 10tq	1 2 4 8
1Mbps	8tq	1	10tq	1	16tq 8tq	1 2	20tq 10tq	1 2	32tq 16tq 8tq	1 2 4	40tq 20tq 10tq	1 2 4

Figure 259 Example configuration for nominal bit rates

CAN clock frequency	8MHz		10MHz		16MHz		20MHz		32MHz		40MHz	
configuration / data bit rate	Number of tqds per bit time	DBTP.DBRP + 1	Number of tqds per bit time	DBTP.DBRP + 1	Number of tqds per bit time	DBTP.DBRP + 1	Number of tqds per bit time	DBTP.DBRP + 1	Number of tqds per bit time	DBTP.DBRP + 1	Number of tqds per bit time	DBTP.DBRP + 1
500Kbps	16tqd 8tqd	1 2	20tqd 10tqd	1 2	32tqd 16tqd 8tqd	1 2 4	40tqd 20tqd 10tqd	1 2 4	32tqd 16tqd 8tqd	2 4 8	40tqd 20tqd 10tqd	2 4 8
1Mbps	8tqd	1	10tqd	1	16tqd 8tqd	1 2	20tqd 10tqd	1 2	32tqd 16tqd 8tqd	1 2 4	40tqd 20tqd 10tqd	1 2 4
2Mbps	-	-	-	-	8tqd	1	10tqd	1	16tqd 8tqd	1 2	20tqd 10tqd	1 2
4Mbps	-	-	-	-	-	-	-	-	8tqd	1	10tqd	1
5Mbps	-	-	-	-	-	-	-	-	-	-	8tqd	1
8Mbps	-	-	-	-	-	-	-	-	-	-	5tqd	1

Figure 260 Example configuration for data bit rates

***Note**: The user must configure the CAN bit timings to comply with the corresponding CAN standards to ensure proper communication on the CAN bus.*

26.4 Message RAM (MRAM)

The MRAM stores RX/TX messages and filter configurations.

***Note**:*
- *The MRAM should be '0' before configuration of the CAN FD controller to prevent bit errors when reading uninitialized words and to avoid unexpected filter element configurations in the MRAM*
- *Unused MRAM is not used for general purposes*

26.4.1 Message RAM configuration

The message RAM has a width of 32 bits. The CAN FD controller can be configured to allocate up to 4480 words in the MRAM (note that the number of words that can be used is limited by the size of the actual MRAM). It is not necessary to configure each of the sections listed in Figure 261, nor is there any restriction with respect to the sequence of the sections.

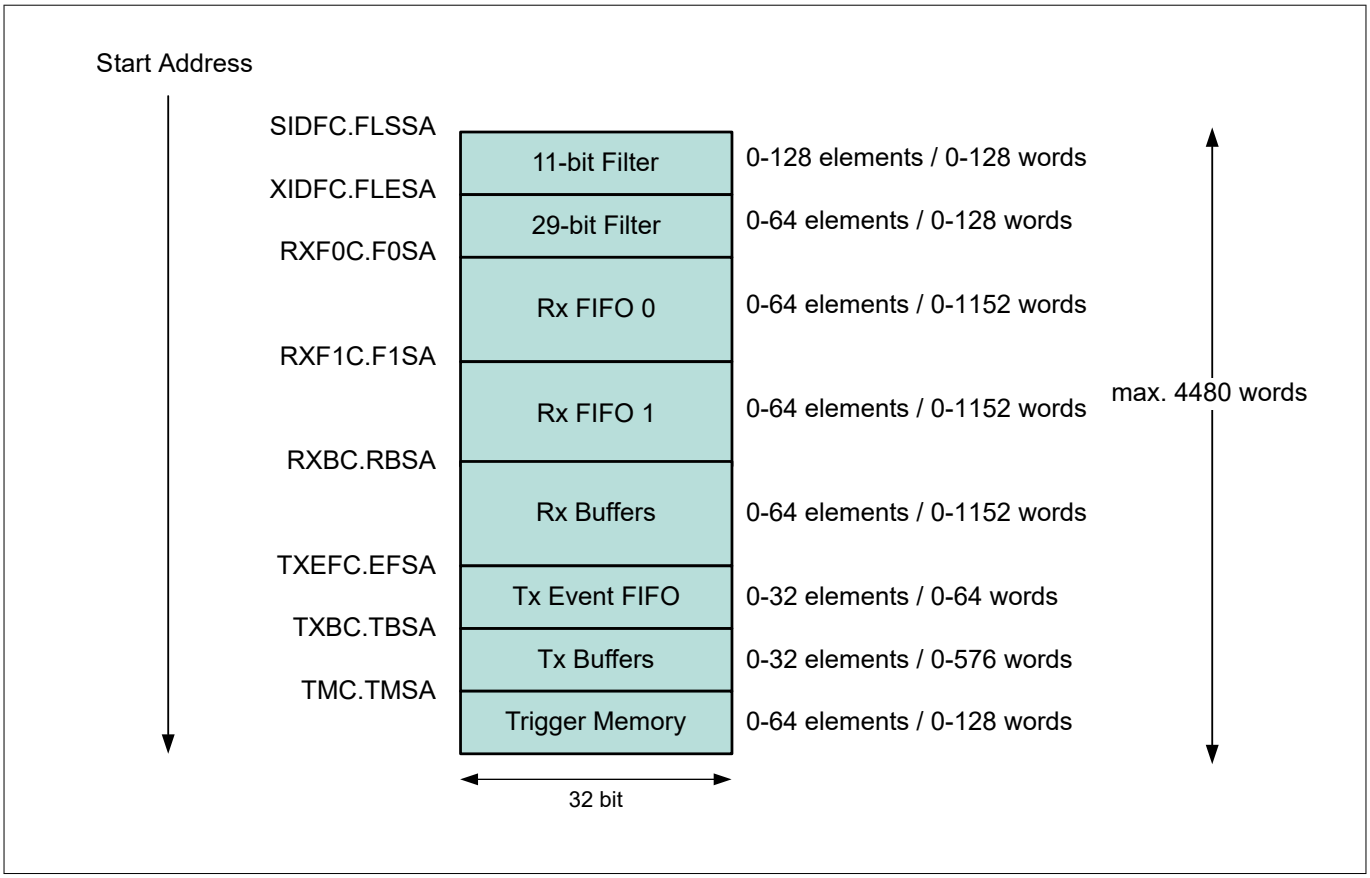


Figure 261 Message RAM configuration

The CAN FD controller addresses the message RAM in 32-bit words, not in single bytes. The configurable start addresses are 32-bit word addresses – only bits 15 to 2 are evaluated, the two least significant bits are ignored.

Note:
- *The CAN FD controller does not check for erroneous configuration of the message RAM. The configuration of the start addresses of different sections and the number of elements of each section should be done carefully to avoid falsification or loss of data*
- *The MRAM is accessible by both M_TTCAN and CPU. Dynamic round-robin scheme is implemented to allocate access*

26.4.2 RX buffer and FIFO element

An RX buffer and FIFO element is a block of 32-bit words, which holds the data and status of a received frame that was stored in the message RAM.

Up to 64 RX buffers and two RX FIFOs is configured in the message RAM. Each RX FIFO section is configured to store up to 64 received messages. The structure of an RX buffer and FIFO element is shown in Figure 262. The element size is configured to store CAN FD messages with up to 64 bytes data field via register RXESC (RX buffer/ FIFO element size configuration).

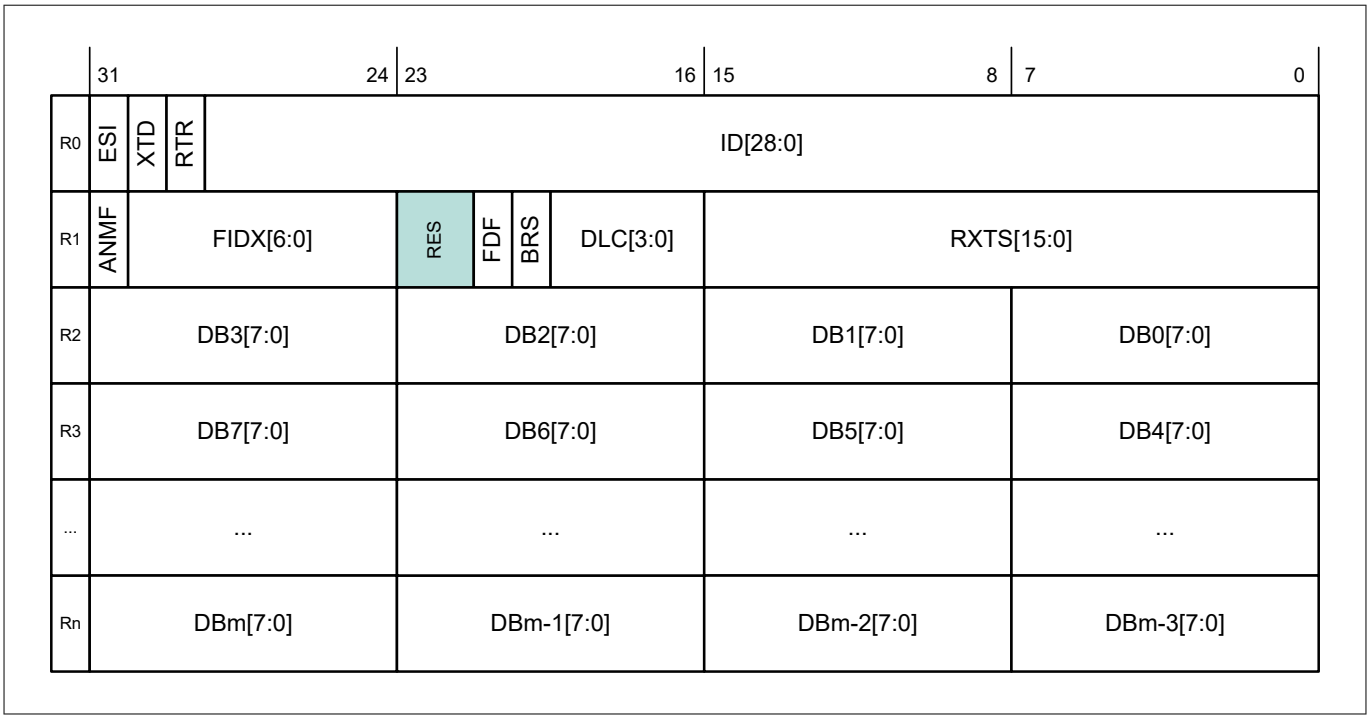


Figure 262 RX buffer and FIFO

Table 236 R0 [bit31] ESI: error state indicator

Bit	Description
0	Transmitting node is error active
1	Transmitting node is error passive

Signals to the CPU whether the received frame has a standard or extended identifier.

Table 237 R0 [bit30] XTD: extended identifier

Bit	Description
0	11-bit standard identifier
1	29-bit extended identifier

Signals to the CPU whether the received frame is a data frame or a remote frame.

Table 238 R0 [bit29] RTR: remote transmission request

Bit	Description
0	Received frame is a data frame
1	Received frame is a remote frame

***Note**: There are no remote frames in CAN FD format. In CAN FD frames (FDF = 1), the dominant remote request substitution (RRS) bit replaces bit remote transmission request (RTR).*

R0 [bit28:0] ID[28:0]: Identifier

A standard or extended identifier depending on bit XTD. A standard identifier is stored into ID[28:18].

Table 239 R1 [bit31] ANMF: accepted non-matching frame

Bit	Description
0	Received frame matching filter index FIDX
1	Received frame did not match any RX filter element

Acceptance of non-matching frames may be enabled via GFC.ANFS[1:0] (Accept non-matching frames standard) and GFC.ANFE[1:0] (Accept non-matching frames extended).

Table 240 R1 [bit30:24] FIDX[6:0]: filter index

FIDX[6:0]	Description
0-127	Index of matching RX acceptance filter element (invalid if ANMF = 1). Range is 0 to List Size Standard/ Extended minus 1(SIDFC.LSS - 1 resp. XIDFC.LSE - 1)

R1 [bit23:22] Reserved: Reserved bits

When writing, always write '0'. The read value is undefined.

Table 241 R1 [bit21] FDF: extended data length

Bit	Description
0	Classic CAN frame format
1	CAN FD frame format

Table 242 R1 [bit20] BRS: bit rate switch

Bit	Description
0	Frame received without bit rate switching
1	Frame received with bit rate switching

Table 243 R1 [bit19:16] DLC[3:0]: data length code

DLC[3:0]	Description
0-8	Classic CAN + CAN FD: received frame has 0-8 data bytes
9-15	Classic CAN: received frame has 8 data bytes. CAN FD: received frame has 12/16/20/24/32/48/64 data bytes. See Table 229 for details

R1 [bit15:0] RXTS[15:0]: RX timestamp

The timestamp counter value captured on start of frame reception. Resolution depending on configuration of the Shared Timestamp Counter Pre-scaler TS_CTL.PRESCALE[15:0].

Register name	Data bits	Data byte
R2 [bit31:24]	DB3[7:0] :	Data Byte 3
R2 [bit23:16]	DB2[7:0] :	Data Byte 2
R2 [bit15:8]	DB1[7:0] :	Data Byte 1
R2 [bit7:0]	DB0[7:0] :	Data Byte 0

Register name	Data bits	Data byte
R3 [bit31:24]	DB7[7:0] :	Data Byte 3
R3 [bit23:16]	DB6[7:0] :	Data Byte 2
R3 [bit15:8]	DB5[7:0] :	Data Byte 1
R3 [bit7:0]	DB4[7:0] :	Data Byte 0
- - -	- - -	- - -
Rn [bit31:24]	DBm[7:0] :	Data Byte m
Rn [bit23:16]	DBm-1[7:0] :	Data Byte m-1
Rn [bit15:8]	DBm-2[7:0] :	Data Byte m-2
Rn [bit7:0]	DBm-3[7:0] :	Data Byte m3

***Note**:*
- *Depending on the configuration of the element size (defined by RX buffer/FIFO element size configuration (RXESC)), Rn will vary from n = 3 to 17*
- *m is a function of n, m = (n - 1) × 4 - 1*
- *The number of valid data bytes are defined by the Data Length Code*

26.4.3 TX buffer element

A TX buffer element is a block of 32-bit words stored in the message RAM that holds data and control information of a frame to be transmitted by the CAN FD controller.

The TX buffers section is configured to hold dedicated TX buffers and a TX FIFO/TX queue. If the TX buffers section is shared by dedicated TX buffers and a TX FIFO/TX queue, the dedicated TX buffers start at the beginning of the TX buffers section followed by the buffers assigned to the TX FIFO or TX queue. The TX handler distinguishes between dedicated TX buffers and TX FIFO/TX queue by evaluating the TX buffer configuration TXBC.TFQS[5:0] (Transmit FIFO/queue size) and TXBC.NDTB[5:0] (Number of dedicated transmit buffer). The element size is configured to store CAN FD messages with up to 64 bytes data field via register TX buffer element size configuration (TXESC).

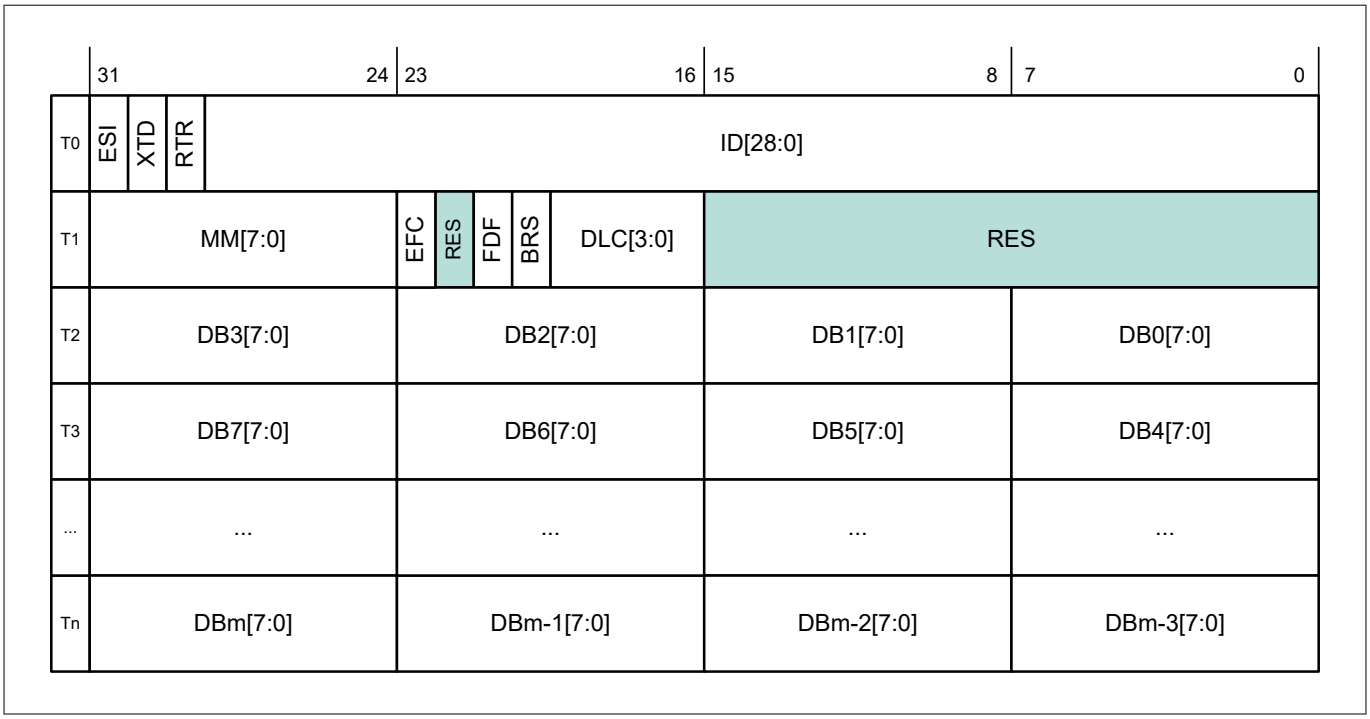


Figure 263 TX buffer element

Table 244 R0 [bit31] ESI: Error state indicator

Bit	Description
0	ESI bit in CAN FD format depends only on error passive flag
1	ESI bit in CAN FD format transmitted recessive

***Note**: The ESI bit of the transmit buffer is ORed with the error passive flag to decide the value of the ESI bit in the transmitted FD frame. As required by the CAN FD protocol specification, an error active node may optionally transmit the ESI bit recessive, but an error passive node will always transmit the ESI bit recessive.*

Table 245 R0 [bit30] XTD: extended identifier

Bit	Description
0	11-bit standard identifier
1	29-bit extended identifier

Signals to the CPU whether the received frame has a standard or extended identifier.

Table 246 R0 [bit29] RTR: remote transmission request

Bit	Description
0	Transmit data frame
1	Transmit remote frame

***Note**: When RTR = 1, the CAN FD controller transmits a remote frame according to ISO11898-1, even if FD Operation Enable (CCCR.FDOE) enables the transmission in CAN FD format.*

T0 [bit28:0] ID[28:0]: Identifier

Standard or extended identifier depending on bit XTD. A standard identifier has to be written to ID[28:18]

T1 [bit31:24] MM[7:0]: message marker

Written by CPU during TX buffer configuration. Copied into TX event FIFO element for identification of TX message status

Table 247 T1 [bit23] EFC: Event FIFO control

Bit	Description
0	Does not store TX events
1	Store TX events

T1 [bit22] Reserved: Reserved bit

When writing, always write '0'. The read value is undefined.

Table 248 T1 [bit21] FDF: FD format

Bit	Description
0	Frame transmitted in classic CAN format
1	Frame transmitted in CAN FD format

Table 249 T1 [bit20] BRS: bit rate switching

Bit	Description
0	CAN FD frames transmitted without bit rate switching
1	CAN FD frames transmitted with bit rate switching

***Note**: Bits ESI, FDF, and BRS are only evaluated when CAN FD operation is enabled CCCR.FDOE = 1. Bit BRS is only evaluated when in addition CCCR.BRSE = 1. See Table 229 for details on bits FDF and BRS.*

Table 250 T1 [bit19:16] DLC[3:0]: data length code

DLC[3:0]	Description
0-8	Classic CAN + CAN FD: transmit frame has0-8 data bytes
9-15	Classic CAN: transmit frame has 8 data bytes

CAN FD: transmit frame has 12/16/20/24/32/48/64 data bytes.

T1 [bit15:0] Reserved: Reserved bits

When writing, always write '0'. The read value is undefined.

T2 [bit31:24]	DB3[7:0] :	Data Byte 3
T2 [bit23:16]	DB2[7:0] :	Data Byte 2
T2 [bit15:8]	DB1[7:0] :	Data Byte 1
T2 [bit7:0]	DB0[7:0] :	Data Byte 0

T3 [bit31:24]	DB7[7:0] :	Data Byte 3
T3 [bit23:16]	DB6[7:0] :	Data Byte 2
T3 [bit15:8]	DB5[7:0] :	Data Byte 1
T3 [bit7:0]	DB4[7:0] :	Data Byte 0
- - -	- - -	- - -
Tn [bit31:24]	DBm[7:0] :	Data Byte m
Tn [bit23:16]	DBm-1[7:0] :	Data Byte m-1
Tn [bit15:8]	DBm-2[7:0] :	Data Byte m-2
Tn [bit7:0]	DBm-3[7:0] :	Data Byte m3

***Note**:*
- *Depending on the configuration of the element size (TXESC), Tn will vary from n = 3 to 17*
- *m is a function of n: m = (n – 1) × 4 – 1*

26.4.4 TX event FIFO element

Each TX event FIFO Element stores information about transmitted messages. By reading the TX event FIFO, the CPU gets this information in the order the messages were transmitted. The status information about the TX event FIFO can be obtained from register TX event FIFO status (TXEFS).

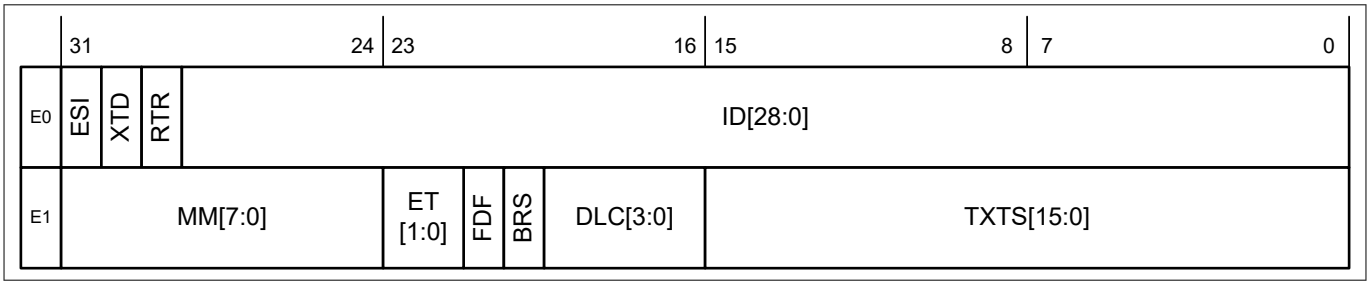


Figure 264 TX event FIFO element

Table 251 E0 [bit31] ESI: error state indicator

Bit	Description
0	Transmitting node is error active
1	Transmitting node is error passive

Table 252 E0 [bit30] XTD: Extended identifier

Bit	Description
0	11-bit standard identifier
1	29-bit extended identifier

Table 253 E0 [bit29] RTR: eemote transmission request

Bit	Description
0	Data frame transmitted
1	Remote frame transmitted

E0 [bit28:0] ID[28:0]: Identifier

Standard or extended identifier depending on bit XTD. A standard identifier is stored into ID[28:18]

E1 [bit31:24] MM[7:0]: message marker

Copied from TX buffer into TX Event FIFO element for identification of TX message status

Table 254 E1 [bit23:22] ET[1:0]: Event type

ET[1:0]	Description
00	Reserved
01	TX event
10	Transmission in spite of cancellation Always set for transmissions in DAR mode(Disable Automatic Retransmission mode)
11	Reserved

Table 255 E1 [bit21] FDF: FD format

Bit	Description
0	Classic CAN frame format
1	CAN FD frame format (new DLC-coding and CRC)

Table 256 E1 [bit20] BRS: bit rate switching

Bit	Description
0	Frame transmitted without bit rate switching
1	Frame transmitted with bit rate switching

Table 257 E1 [bit19:16] DLC[3:0]: data length code

DLC[3:0]	Description
0-8	Classic CAN + CAN FD: frame with 0-8 data bytes transmitted
9-15	Classic CAN: frame with 8 data bytes transmitted. CAN FD: frame with 12/16/20/24/32/48/64data bytes transmitted. See Table 229 for details

E1 [bit15:0] TXTS[15:0]: TX timestamp

The timestamp counter value captured on start-of-frame transmission. The resolution depends on the configuration of the Shared Timestamp Counter Pre-scaler TS_CTL.PRESCALE[15:0].

26.4.5 Standard message ID filter element

A standard message ID filter element consists of a single 32-bit word, and can be configured as a range filter, dual filter, classic bit mask filter, or filter for a single dedicated ID, for messages with 11-bit standard IDs.

Up to 128 filter elements can be configured for 11-bit standard IDs. When accessing a Standard Message ID Filter element, its address is the Filter List Standard Start Address (SIDFC.FLSSA[15:2]) +index of the filter element (0 to 127).

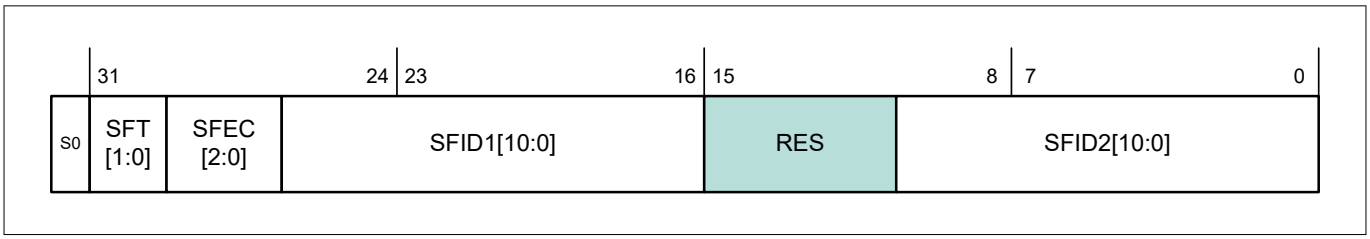


Figure 265 Standard message ID filter

Table 258 S0 [bit31:30] SFT[1:0]: standard filter type

SFT[1:0]	Description
00	Range filter from SFID1[10:0] to SFID2[10:0] (SFID2[10:0] ≥ received ID ≥ SFID1[10:0])
01	Dual ID filter for SFID1[10:0] or SFID2[10:0]
10	Classic filter: SFID1[10:0] = filter, SFID2[10:0] = mask Only those bits of SFID1[10:0] where the corresponding SFID2[10:0] bits are 1 are relevant
11	Filter element disabled

***Note**: With SFT = 11, the filter element is disabled and the acceptance filtering continues (same behavior as with SFEC =000).*

S0 [bit29:27] SFEC[2:0]: Standard filter element configuration.

All enabled filter elements are used for acceptance filtering of standard frames. Acceptance filtering stops at the first matching enabled filter element or when the end of the filter list is reached.

If SFEC[2:0] = 100, 101, or 110 a match sets interrupt flag IR.HPM (High Priority Message) and, if enabled, an interrupt is generated. In this case register HPMS (High Priority Message Status) is updated with the status of the priority match.

Table 259 SFEC[2:0]: set priority and store in RX FIFO

SFC[2:0]	Description
000	Disable filter element
001	Store in RX FIFO 0 if filter matches
100	Store in RX FIFO 1 if filter matches
011	Reject ID if filter matches
100	Set priority if filter matches
101	Set priority and store in RX FIFO 0 if filter matches
110	Set priority and store in RX FIFO 1 if filter matches
111	Store into dedicated RX buffer or as debug message, configuration of SFT[1:0] ignored

S0 [bit26:16] SFID1[10:0]: Standard Filter ID 1.

This bit field has a different meaning depending on the configuration of SFEC[2:0]:

- SFEC[2:0] = 001 to 110

Set SFID1[10:0] according to the SFT[1:0] setting

- SFEC[2:0] = 111

SFID1[10:0] defines the ID of a standard dedicated RX buffer or debug message to be stored. The received identifiers must match, no masking mechanism is used.

S0 [bit15:11] Reserved: Reserved bits.

When writing, always write '0'. The read value is undefined.

S0 [bit10:0] SFID2[10:0]: Standard Filter ID 2.

This bit field has a different meaning depending on the configuration of SFEC[2:0]:

- SFEC[2:0] = 001 to 11

Set SFID2[10:0] according to the SFT[1:0] setting

- SFEC[2:0] = 111

Filter for dedicated RX buffers or for debug messages SFID2[10:9] decides whether the received message is stored into a dedicated RX buffer or treated as message A,B, or C of the debug message sequence.

Table 260 SFID[10:9]] store message into a dedicated RX buffer

SFID[10:9]	Description
00	Store message into a dedicated RX buffer
01	Debug message A
10	Debug message B
11	Debug message C

SFID2[8:6] are reserved bits. When writing, always write '0'.The read value is undefined. SFID2[5:0] defines the offset to the RX buffer Start Address RXBC.RBSA[15:2] to store a matching message.

***Note**: The debug message is used to debug on CAN feature.*

26.4.6 Extended message ID filter element

bit words, and can be configured as a range filter, dual filter,classic bit mask filter, or filter for a single dedicated ID, for messages with 29-bit extended IDs.

Up to 64 filter elements can be configured for 29-bitextended IDs. When accessing an extended message ID filter element, its address is Filter list extended start address (XIDFC.FLESA[15:2]) + 2× index of the filter element (0 to 63).

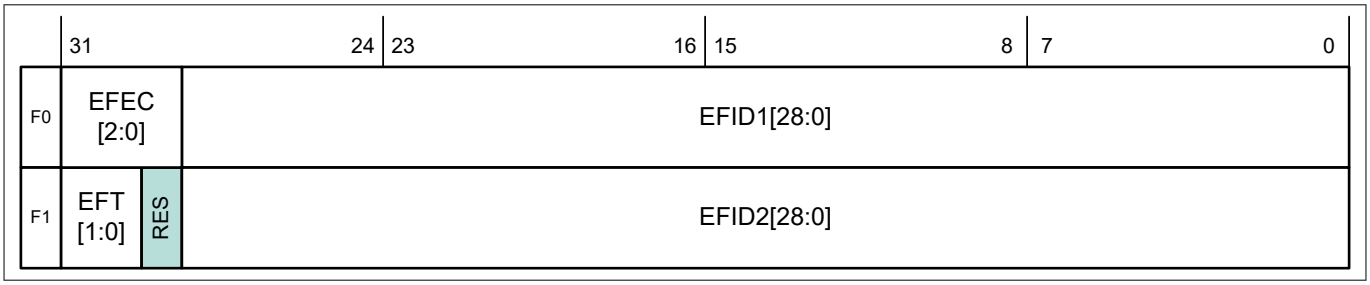


Figure 266 Extended message ID filter

F0 [bit31:29] EFEC[2:0]: extended filter element configuration All enabled filter elements are used for acceptance filtering of extended frames. Acceptance filtering stops at the first matching enabled filter element or when the end of the filter list is reached.

If EFEC[2:0] = 100, 101, or 110 a match sets interrupt flag IR.HPM (High Priority Message) and, if enabled, an interrupt is generated. In this case High Priority Message Status register (HPMS) is updated with the status of the priority match.

Table 261 Extended filter element

EFEC[2:0]	Description
000	Disable filter element
001	Store in RX FIFO 0 if filter matches
010	Store in RX FIFO 1 if filter matches
011	Reject ID if filter matches
100	Set priority if filter matches
101	Set priority and store in RX FIFO 0 if filter matches
110	Set priority and store in RX FIFO 1 if filter matches
111	Store into dedicated RX buffer or as debug message, configuration of EFT[1:0] ignored

F0 [bit28:0] EFID1[28:0]: Extended Filter ID 1

This bit field has a different meaning depending on the configuration of EFEC[2:0].

- EFEC[2:0] = 001 to 110

Set EFID1[28:0] according to the EFT[1:0] setting.

- EFEC[2:0] = 11

EFID1[28:0] defines the ID of an extended dedicated RX buffer or debug message to be stored. The received identifiers must match, only XIDAM masking mechanism is used.

F1 [bit31:30] EFT[1:0]: Extended Filter Type

Table 262 Extended filter type

EFT[2:0]	Description
00	Range filter from EFID1[28:0] to EFID2[28:0] (EFID2[28:0] ≥ received ID ANDed with XIDAM ≥ EFID1[28:0])
01	Dual ID filter Matches when EFID1[28:0] or EFID2[28:0] is equal to received ID ANDed with XIDAM
10	Classic filter: EFID1[28:0] = filter, EFID2[28:0] = mask. Only those bits of EFID1[28:0] where the corresponding EFID2[28:0] bits are 1 are relevant. Matches when the received ID ANDed with XIDAM is equal to EFID1[28:0] masked by EFID2[28:0]
11	Range filter from EFID1[28:0] to EFID2[28:0] (EFID2[28:0] ≥ EFID1[28:0]), XIDAM mask not applied

F1 [bit29] Reserved: Reserved bit

When writing, always write '0'. The read value is undefined.

F1 [bit28:0] EFID2[28:0]: Extended Filter ID 2

This bit field has a different meaning depending on the configuration of EFEC[2:0]:

- EFEC[2:0] = 001 to 110

Set EFID2[28:0] according to the EFT[1:0] setting

- EFEC[2:0] = 111

EFID2[28:0] is used to configure this filter for dedicated RX buffers or for debug messages

EFID2[28:11] are reserved bits. When writing, always write'0'. The read value is undefined.

EFID2[10:9] decides whether the received message is stored into a dedicated RX buffer or treated as message A,B, or C of the debug message sequence.

Table 263 Extended filter type

EFID2[10:9]	Description
00	Store message into a dedicated RX buffer
01	Debug message A
10	Debug message B
11	Debug message C

EFID2[8:6] are reserved bits. When writing, always write '0'.The read value is undefined.

EFID2[5:0] defines the offset to the RX Buffer Start Address.

RXBC.RBSA[15:2] to store a matching message.

***Note**: The debug message is used to debug on CAN feature.*

26.4.7 Trigger memory element

Up to 64 trigger memory elements can be configured. When accessing a trigger memory element, its address is the Trigger Memory Start Address (TTTMC). TMSA plus the index of the trigger memory element (0…63).

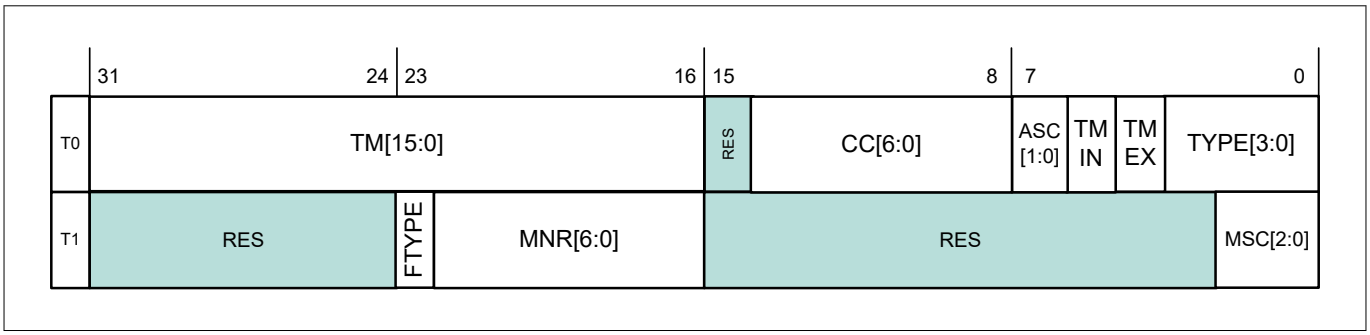


Figure 267 Trigger memory element

T0 Bit 31:16 TM[15:0]: Time Mark

Cycle time for which the trigger becomes active.

Table 264 T0 Bit 14:8 CC[6:0]: cycle code

CC[6:0]	Description
000000x	Valid for all cycles
000001c	Valid every second cycle at cycle count mod2= c
00001cc	Valid every fourth cycle at cycle count mod4 =cc
0001ccc	Valid every eighth cycle at cycle count mod8 =ccc
001cccc	Valid every sixteenth cycle at cycle countmod16 = cccc
01ccccc	Valid every thirty-second cycle at cycle countmod32 = ccccc
1cccccc	Valid every sixty-fourth cycle at cycle countmod64 = cccccc

Cycle count for which the trigger is valid. Ignored for trigger types Tx_Ref_Trigger, Tx_Ref_Trigger_Gap, Watch_Trigger, Watch_Trigger_Gap, and End_of_List.

Table 265 T0 Bit 7:6 ASC[1:0]: asynchronous serial communication

ASC[1:0]	Description
00	No ASC operation
01	Reserved, do not use
10	Node is ASC receiver
11	Node is ASC transmitter

***Note**: ASC functionality is not supported.*

Table 266 T0 Bit 5 TMIN: Time Mark Event Internal

TMIN	Description
0	No action
1	TTIR.TTMI is set when trigger memory element becomes active

Table 267 T0 Bit 3:0 TYPE[3:0]: Trigger type

TYPE [3:0]	Description
0000	Tx_Ref_Trigger - valid when not in gap
0001	Tx_Ref_Trigger_Gap - valid when in gap
0010	Tx_Trigger_Single - starts a single transmission in an exclusive time window
0011	Tx_Trigger_Continous - starts continuous transmission in an exclusive time window
0100	Tx_Trigger_Arbitration - starts a transmission in an arbitrating time window

(table continues...)

Table 267 (continued) T0 Bit 3:0 TYPE[3:0]: Trigger type

TYPE [3:0]	Description
0101	Tx_Trigger_Merged - starts a merged arbitration window
0110	Watch_Trigger - valid when not in gap
0111	Watch_Trigger_Gap - valid when in gap
1000	Rx_Trigger - check for reception
1001	Time_Base_Trigger - only control TMIN,TMEX, and ASC
1010 … 1111	End_of_List - illegal type, causes config error

***Note**:*
- *For ASC operation (ASC = 10, 11) only trigger types Rx_Trigger and Time_Base_Trigger should be used*
- *ASC operation is not supported in this device*

Table 268 T1 Bit 23 FTYPE: filter type

FTYPE	Description
0	11-bit standard message ID
1	29-bit extended message ID

T1 Bit 22:16 MNR[6:0]: message number.

Transmission: Trigger is valid for configured TX buffer number. Valid values are 0 to 31.

Reception: Trigger is valid for standard/extended message ID filter element number. Valid values are 0 to 63 and 0 to127.

T1 Bits 2:0 MSC[2:0]: message status count.

Counts scheduling errors for periodic messages in exclusive time windows. It has no function for arbitrating messages and in event-driven CAN communication (ISO 11898-1:2015).

***Note**:*
- *The trigger memory elements should be written when the M_TTCAN is in INIT state. Write access to the trigger memory elements outside INIT state is not allowed*
- *There is an exception for TMIN and TMEX when they are defined as part of a trigger memory element of TYPETx_Ref_Trigger. In this case they become active at the time mark modified by the actual Reference Trigger Offset (TTOST.RTO)*

26.4.8 RAM watchdog (RWD)

The RAM watchdog monitors the READY output of the MRAM. A MRAM access starts the Message RAM watchdog counter with the value configured by RWD.WDC. The counter is reloaded with RWD.WDC when the MRAM signals successful completion by activating its READY output. In case there is no response from the MRAM until the counter has counted down to zero, the counter stops and interrupt flag IR.WDI is set. The RWD counter is clocked by the host clock(clk_sys). See the registers reference manual for more information on the RWD register.

26.5 Time trigger CAN

26.5.1 Reference message

A reference message is a data frame characterized by a specific CAN identifier. It is received and accepted by all nodes except the time master (sender of the reference message).

For Level 1, the data length must be at least one. For Level 0 and Level 2, the data length must be at least four; otherwise, the message is not accepted as a reference message. The reference message is extended by other data up to the sum of eight CAN data bytes. All the bits of the identifier except the three LSBs characterize the message as a reference message. The last three bits specify the priorities of up to eight potential time masters. The reserved bits are transmitted as logical 0 and are ignored by the receivers. The reference message is configured using the TTRMC register.

The time master transmits the reference message. If the reference message is disturbed by an error, it is retransmitted immediately. In a retransmission, the transmitted Master_Ref_Mark is updated. The reference message is sent periodically, but it is allowed to stop the periodic transmission (Next_is_Gap bit). It can initiate event synchronized transmission at the start of the next basic cycle by the current time master or by one of the other potential time masters.

The node transmitting the reference message is the current time master. The time master is allowed to transmit other messages. If the current time master fails, its function is replicated by the potential time master with the highest priority. Nodes that are neither time master nor potential time master are time-receiving nodes.

Level 1

Level 1 operation is configured via TTOCF.OM = 01 and TTOCF.GEN. External clock synchronization is not available in Level 1.

The information related to the reference message is stored in the first data byte as shown in Table 270. Cycle_Count is optional.

Table 269 First Byte of Level 1 reference message

Bits	7	6	5	4	3	2	1	0
First Byte	Next_is_Gap	Reserved	Cycle_Count[5:0]					

Level 2

Level 2 operation is configured via TTOCF.OM = 10 and TTOCF.GEN.

The information related to the reference message is stored in the first four data bytes as shown in Table 270. Cycle_Count and the lower four bits of NTU_Res are optional. The M_TTCAN does not evaluate NTU_Res[3:0] from received reference messages, it always transmits these bits as zero.

Table 270 First four Bytes of Level 1 reference message

Bits	7	6	5	4	3	2	1	0
First Byte	Next_is_Gap	Reserved	Cycle_Count[5:0]					
Second Byte	NTU_Res[6:4]			NTU_Res[3:0]				Disc_Bit
Third Byte	Master_Ref_Mark[7:0]							
Fourth Byte	Master_Ref_Mark[15:8]							

Level 0

Level 0 operation is configured via TTOCF.OM = 11. External event-synchronized time-triggered operation is not available in Level 0.

The information related to the reference message is stored in the first four data bytes as shown in Table 271. In Level 0, Next_is_Gap is always zero. Cycle_Count and the lower four bits of NTU_Res are optional. The M_TTCAN does not evaluate NTU_Res[3:0] from received reference messages; it always transmits these bits as '0'.

Table 271 First four bytes of Level 0 reference message

Bits	7	6	5	4	3	2	1	0
First byte	Next_is_Gap	Reserved	Cycle_Count[5:0]					
Second byte	NTU_Res[6:4]			NTU_Res[3:0]				Disc_Bit
Third byte	Master_Ref_Mark[7:0]							
Fourth byte	Master_Ref_Mark[15:8]							

26.5.2 TTCAN configuration

26.5.2.1 TTCAN timing

The network time unit (NTU) is the unit for the measurement of time. The NTU is a constant of the whole network and is defined by the network system designer. In TTCAN Level 1, the NTU is the nominal CAN bit time. In the TTCAN Level 0 and Level 2, the NTU is a fraction of the physical second.

The NTU is the base for the local time. The integer part of the local time (16-bit value) is incremented once for each NTU. The cycle time and the global time are both derived from the local time. The fractional part (3-bit value) of the local time, cycle time, and the global time are not readable.

In the TTCAN Level 0 and Level 2, the length of the NTU is defined by the time unit ratio (TUR). The TUR is a non\u0002integer number is given by TUR = TURNA.NAV/TURCF.DC. The length of the NTU is determined by NTU = CAN Clock Period × TUR.

The TUR numerator configuration (NC) is an 18-bit number, TURCF.NCL[15:0] is programmed in the range 0x0000-0xFFFF. TURCF.NCH[17:16] is hard-wired to '0b01'. When the number '0xnnn'n is written to TURCF.NCL[15:0], TURNA.NAV starts with the value 0x10000 + 0x0nnnn = 0x1nnnn. The TUR denominator configuration (TURCF.DC) is a 14-bit number. The TURCF.DC is programmed in the range 0x0001 -0x3FFF; 0x0000 is an illegal value.

In Level 1, NC must be ≥ 4 × TURCF.DC. In Level 0 and Level 2 NC must be ≥ 8 × TURCF.DC to allow the 3-bit resolution for the internal fractional part of the NTU.

A hardware reset presets TURCF.DC to 0x1000 and TURCF.NCL to 0x10000, resulting in an NTU consisting of 16 CAN clock periods. The local time and application watchdog are not started before either the CCCR.INIT is reset or TURCF.ELT is set. The TURCF.ELT is not set before the NTU is configured. Setting TURCF.ELT to '1' also locks the write access to register TURCF.

At start-up TURNA.NAV is updated from NC (= TURCF.NCL +0x10000) when TURCF.ELT is set. In the TTCAN Level 1 there is no drift compensation. TURNA.NAV does not change during operation, it always equals to NC.

In TTCAN Level 0 and Level 2, there are two possibilities for TURNA.NAV to change. When operating as time slave or backup time master, and when TTOCF.ECC is set, TURNA.NAV is updated automatically to the value calculated from the monitored global time speed, as long as the M_TTCAN is in synchronization states In_Schedule or In_Gap. When it loses synchronization, it returns to NC. When operating as the actual time

master, and when TTOCF.EECS is set, the host updates the TURCF.NCL. When the host sets TTOCN.ECS, TURNA.NAV will be updated from the new value of NC at the next reference message. The status flag TTOST.WECS is set when TTOCN.ECS is set and is cleared when TURNA.NAV is updated. The TURCF.NCL is write-locked while TTOST.WECS is set.

In TTCAN Level 0 and Level 2, the clock calibration process adapts TURNA.NAV in the range of the synchronization deviation limit (SDL) of NC ± 2 (TTOCF.LDSDL+5). TURCF.NCL should be programmed to the largest applicable numerical value to achieve the best accuracy in the calculation of TURNA.NAV.

The synchronization deviation (SD) is the difference between NC and TURNA.NAV (SD = |NC - TURNA.NAV|). It is limited by the SDL, which is configured by its dual logarithm

TTOCF.LDSDL (SDL = 2 (TTOCF.LDSDL+5))

and should not exceed the clock tolerance given by the CAN bit timing configuration (SD is calculated at each new basic cycle). When the calculated TURNA.NAV deviates by more than SDL from NC, or if the Disc_Bit in the reference message is set, the drift compensation is suspended, TTIR.GTE is set, and TTOSC.QCS is reset; if Disc_Bit = 1, TTIR.GTD is set. The TUR configuration examples are shown in Table 272

Table 272 TUR configuration examples

TUR	8	10	24	50	510	12500	32.5	100/12	529/17
NC	0x1FFF8	0x1FFFE	0x1FFF8	0x1FFEA	0x1FFFE	0x1FFE0	0x1FFE0	0x19000	0x10880
TURCF.DC	0x3FFF	0x3333	0x1555	0x0A3D	0x0101	0x0001	0xFC0	0x3000	0x0880

TTOCN.ECS schedules NC for activation by the next reference message. TTOCN.SGT schedules TTGTP.TP for activation by the next reference message. Setting of TTOCN.ECS and TTOCN.SGT requires TTOCF.EECS to be set (external clock synchronization enabled) while the M_TTCAN is the actual time master.

The M_TTCAN module provides an application watchdog to verity the function of the application program. The host has to serve this watchdog regularly, else all CAN bus activity is stopped. The Application Watchdog Limit TTOCF.AWL specifies the number of NTUs the watchdog has to be served. The maximum number of NTUs is 256. The Application Watchdog is served by reading register TTOST. TTOST.AWE indicates whether the watchdog is served in time. In case the application failed to serve the application watchdog, the interrupt flag TTIR.AW is set. For software development, the application watchdog may be disabled by programming TTOCF.AWL to '0x00' (see Application watchdog).

26.5.2.2 Message scheduling

TTOCF.TM controls whether the M_TTCAN operates as a potential time master or as a time slave. If it is a potential time master, the three LSBs of the reference message identifier TTRMC.RID define the master priority, 0 being the highest and 7 the lowest priority. Two nodes in the network may not use the same master priority. TTRMC.RID is used to recognize the reference messages. TTRMC.RMPS is not relevant for time slaves.

The initial reference trigger offset TTOCF.IRTO is a 7-bit value that defines (in NTUs) the duration, a backup time master waits before it starts the transmission of a reference message, when a reference message is expected but the bus remains idle. The recommended value for TTOCF.IRTO is the master priority multiplied with a factor depending on the expected clock drift between the potential time masters in the network. The sequential order of the backup time masters, when one of them starts the reference message if the current time master fails, should correspond to their master priority, even with the maximum clock drift.

TTOCF.OM decides whether the node operates in TTCAN Level 0, Level 1, or Level 2. In one network, all potential time masters operates on the same level. Time slaves may operate on Level 1 in a Level 2 network, but not vice versa. The configuration of the TTCAN operation mode via TTOCF.OM is the last step in the setup. When TTOCF.OM = 00 (event-driven CAN communication), the M_TTCAN operates according to ISO 11898-1:2015, without time triggers. When TTOCF.OM = 01 (Level 1), the M_TTCAN operates according to ISO 11898-4, but without the possibility to synchronize the basic cycles to external events, the Next_is_Gap bit in the reference message is ignored. When TTOCF.OM = 10 (Level 2), the M_TTCAN operates according to ISO 11898-4, including

the event\u0002synchronized start of a basic cycle. When TTOCF.OM = 11 (Level 0), the M_TTCAN operates as event-driven CAN but maintains a calibrated global time base similar to Level 2.

TTOCF.EECS enables the external clock synchronization, allowing the application program of the current time master to update the TUR configuration during time-triggered operation, to adapt the clock speed and (in Level 0,2 only) the global clock phase to an external reference.

TTMLM.ENTT in the TT Matrix Limits register specifies the number of expected Tx_Triggers in the system matrix. This is the sum of Tx_Triggers for exclusive single arbitrating and merged arbitrating windows, excluding the Tx_Ref_Triggers.

***Note**: This is usually not the number of Tx_Trigger memory elements; the number of basic cycles in the system matrix and the trigger's repeat factors must be taken into account. An inaccurate configuration of TTMLM.ENTT will result in either a TX count underflow (TTIR.TXU = 1 and TTOST.EL = 01, severity 1) or in a TX count overflow (TTIR.TXO = 1 and TTOST.EL = 10, severity 2).*

***Note**: In case the first reference message seen by a node does not have Cycle_Count zero, this node finishes its first matrix cycle with TX count resulting in a TX count underflow condition. As long as a node is in state, synchronizing its Tx_Triggers will not lead to transmissions.*

TTMLM.CCM specifies the number of the last basic cycle in the system matrix. The counting of basic cycles starts at '0'. In a system matrix consisting of eight basic cycles TTMLM.CCM would be '7'. TTMLM.CCM is ignored by time slaves, a receiver of a reference message considers the received cycle count as the valid cycle count for the actual basic cycle.

TTMLM.TXEW specifies the length of the TX enable window in the NTUs. The TX enable window is the period at the beginning of a time window where a transmission is started. If the sample point of the first bit of a transmit message is not inside the TX enable window, the transmission cannot be started in that time window at all. An example is because of an overlap from the previous time window's message. TTMLM.TXEW is chosen based on the network's synchronization quality and the relation between the length of the time windows and the length of messages.

26.5.2.3 Trigger memory

The trigger memory is part of the MRAM. It stores up to 64 trigger elements. A trigger memory element consists of time mark (TM), cycle code (CC), trigger type (TYPE), filter type (FTYPE), message number (MNR), message status count (MSC), time mark event internal (TMIN), time mark event external (TMEX), and asynchronous serial communication ASC (see Trigger memory element for more information).

The time mark defines the cycle time in which a trigger becomes active. The trigger elements in the trigger memory are sorted by their time marks. The trigger element with the lowest time mark is written to the first trigger memory word. Message number and cycle code are ignored for triggers of type Tx_Ref_Trigger, Tx_Ref_Trigger_Gap, Watch_Trigger, Watch_Trigger_Gap, and End_of_List.

When the cycle time reaches the time mark of the actual trigger, the FSE switches to the next trigger and starts to read it from the trigger memory. For a transmit trigger, the TX handler starts to read the message from the message RAM as soon as the FSE switches to its trigger. The RAM access speed defines the minimum time step between a transmit trigger and its preceding trigger. The TX handler should be able to prepare the transmission before the transmit trigger's time mark is reached. The RAM access speed also limits the number of non-matching (with regard to their cycle code) triggers between two matching triggers, the next matching trigger must be read before its time mark is reached. If the reference message is n NTU long, a trigger with a time mark less than n will never become active and will be treated as a configuration error.

The starting point of cycle time is the sample point of the reference message's start-of-frame bit. The next reference message is requested when cycle time reaches the Tx_Ref_Trigger's time mark. The M_TTCAN reacts to the transmission request at the next sample point. A new Sync_Mark is captured at the start-of-frame bit, but the cycle time is incremented until the reference message is successfully transmitted (or received) and the

Sync_Mark is taken as the new Ref_Mark. At this point, cycle time restarts. As a consequence, cycle time can never (with the exception of initialization) be seen at a value less than n, with n being the length of the reference message measured in NTU.

Length of a basic cycle: Tx_Ref_Trigger time mark + 1 NTU + 1 CAN bit time

The trigger list will be different for all nodes in the TTCAN network. Each node knows only the Tx_Triggers for its own transmit messages, the Rx_Triggers for the receive messages that are processed by this node, and the triggers concerning the reference messages.

26.5.2.3.1 Trigger types

Tx_Ref_Trigger (TYPE = 0000) and Tx_Ref_Trigger_Gap (TYPE = 0001) cause the transmission of a reference message by a time master. A configuration error (TTOST.EL = 11, severity 3) is detected when a time slave encounters a Tx_Ref_Trigger(_Gap) in its trigger memory. Tx_Ref_Trigger_Gap is only used in external event\u0002synchronized time-triggered operation mode. In that mode, Tx_Ref_Trigger is ignored when the M_TTCAN synchronization state is In_Gap (TTOST.SYS = 10).

Tx_Trigger_Single (TYPE = 0010), Tx_Continous (TYPE = 0011), Tx_Trigger_Arbitration (TYPE = 0100), and Tx_Trigger_Merged (TYPE =0101) cause the start of a transmission. They define the start of a time window.

Tx_Trigger_Single starts a single transmission in an exclusive time window when the message buffer's Transmission Request Pending bit is set. After successful transmission, the Transmission Request Pending bit is reset.

Tx_Trigger_Continuous starts a transmission in an exclusive time window when the message buffer's Transmission Request Pending bit is set. After successful transmission, the Transmission Request Pending bit remains set, and the message buffer is transmitted again in the next matching time window.

Tx_Trigger_Arbitration starts an arbitrating time window, Tx_Trigger_Merged a merged arbitrating time window. The last Tx_Trigger of a merged arbitrating time window must be of type Tx_Trigger_Arbitration. A configuration error (TTOST.EL = 11, severity 3) is detected when a trigger of type Tx_Trigger_Merged is followed by any other Tx_Trigger than one of type Tx_Trigger_Merged or Tx_Trigger_Arbitration. Several Tx_Triggers may be defined for the same TX message buffer. Depending on their cycle code, they may be ignored in some basic cycles. The cycle code should be considered when the expected number of Tx_Triggers (TTMLM.ENTT) is calculated.

Watch_Trigger (TYPE = 0110) and Watch_Trigger_Gap (TYPE = 0111) check for missing reference messages. They are used by both time masters and slaves. Watch_Trigger_Gap is only used in external event-synchronized time-triggered operation mode. In that mode a Watch_Trigger is ignored when the M_TTCAN synchronization state is In_Gap (TTOST.SYS = 10).

Rx_Trigger (TYPE = 1000) is used to check for the reception of periodic messages in exclusive time windows. Rx_Triggers are not active until the state In_Schedule or In_Gap is reached. The time mark of an Rx_Trigger should be placed after the end of that message transmission, independent of time window boundaries. Depending on their cycle code, Rx_Triggers may be ignored in some basic cycles. At the Rx_Trigger time mark, it is checked whether the last received message before this time mark and after the start of cycle or previous Rx_Trigger matches the acceptance filter element referenced by MNR. Accepted messages are stored in one of two receive FIFOs, according to the acceptance filtering, independent of the Rx_Trigger. Acceptance filter elements that are referenced by Rx_Triggers should be placed at the beginning of the filter list to ensure that the filtering is finished before the Rx_Trigger time mark is reached.

Time_Base_Trigger (TYPE = 1001) is used to generate internal/external events depending on the configuration of ASC, TMIN, and TMEX.

End_of_List (TYPE = 1010…1111) is an illegal trigger type, a configuration error (TTOST.EL = 11, severity 3) is detected when an End_of_List trigger is encountered in the trigger memory before the Watch_Trigger or Watch_Trigger_Gap.

26.5.2.3.2 Restrictions for the node's trigger list

Two triggers are not active at the same cycle time and cycle count, but triggers that are active in different basic cycles (different cycle code) shares the same time mark.

Rx_Triggers and Time_Base_Triggers may not be placed inside the TX enable windows of Tx_Trigger_Single/ Continuous/Arbitration, but they are placed after Tx_Trigger_Merged.

The triggers placed after the Watch_Trigger (or the Watch_Trigger_Gap when TTOST.SYS = 10) will never become active. The watch triggers themselves will not become active when the reference messages are transmitted on time.

All the unused trigger memory words (after the Watch_Trigger or after the Watch_Trigger_Gap when TTOST.SYS = 10) must be set to trigger type End_of_List.

A typical trigger list for a potential time master will begin with a number of Tx_Triggers and Rx_Triggers followed by the Tx_Ref_Trigger and Watch_Trigger. For networks with external event-synchronized time-triggered communication, this is followed by the Tx_Ref_Trigger_Gap and the Watch_Trigger_Gap. The trigger list for a time slave will be the same but without the Tx_Ref_Trigger and the Tx_Ref_Trigger_Gap.

At the beginning of each basic cycle, that is at each reception or transmission of a reference message, the trigger list is processed starting with the first trigger memory element. The FSE looks for the first trigger with a cycle code that matches the current cycle count. The FSE waits until cycle time reaches the trigger's time mark and activates the trigger. Later, the FSE looks for the next trigger in the list with a cycle code that matches the current cycle count.

Special consideration is needed for the time around Tx_Ref_Trigger and Tx_Ref_Trigger_Gap. In a time master competing for master ship, the effective time mark of a Tx_Ref_Trigger is decremented to be the first node to start a reference message. In backup time masters the effective time mark of a Tx_Ref_Trigger or Tx_Ref_Trigger_Gap is the sum of its configured time mark and the Reference Trigger Offset TTOCF.IRTO. If error level 2 is reached (TTOST.EL = 10), the effective time mark is the sum of its time mark and 0x127. No other trigger elements should be placed in this range; otherwise, the time marks may appear out of order and are flagged as a configuration error. Trigger elements that are coming after Tx_Ref_Trigger are not active until the reference messages comes on time.

There are inter-dependencies between the following parameters:

- Host clock frequency
- Speed and waiting time for Trigger RAM accesses
- Length of the acceptance filter list
- Number of trigger elements
- Complexity of cycle code filtering in the trigger elements
- Offset between time marks of the trigger elements

26.5.2.3.3 Examples of trigger handling

The following example shows how the trigger list is derived from a node's system matrix. Assume that node A is a first time master; a section of the system matrix shown in Table 273.

Table 273 System matrix node A

Cycle count	Time mark1	Time mark2	Time mark3	Time mark4	Time mark5	Time mark6	Time Mark7
0	Tx7					TxRef	Error
1	Rx3		Tx2, Tx4			TxRef	Error
2						TxRef	Error

(table continues...)

Table 273 (continued) System matrix node A

Cycle count	Time mark1	Time mark2	Time mark3	Time mark4	Time mark5	Time mark6	Time Mark7
3	Tx7		Rx5			TxRef	Error
4	Tx7			Rx6		TxRef	Error

The cycle count starts with 0 – 0, 1, 3, 7, 15, 31, 63 (the number of basic cycles in the system matrix is 1, 2, 4, 8, 16, 32, 64). The maximum cycle count is configured by TTMLM.CCM. The cycle code (CC) is composed of repeat factor (value of most significant '1') and the number of the first basic cycle in the system matrix (bit field after most significant '1').

For example, when CC is 0b0010011 (repeat factor: 16, first basic cycle: 3) and maximum cycle count of TTMLM.CCM = 0x3F, matches occur at cycle counts 3, 19, 35, 51.

A trigger element consists of time mark (TM), cycle code (CC), trigger type (TYPE), and message number (MNR). For transmission, MNR references the TX buffer number (0..31). For reception, MNR references the number of the filter element (0..127) that matched during the acceptance filtering. Depending on the configuration of the Filter Type FTYPE, the 11-bit or 29-bit message ID filter list is referenced.

In addition, a trigger element can be configured for Asynchronous Serial Communication (ASC), generation of Time Mark Event Internal (TMIN), and Time Mark Event External (TMEX). The Message Status Count (MSC) holds the counter value (0..7) for scheduling errors for periodic messages in exclusive time windows when the time mark of the trigger element becomes active.

Table 274 Trigger list node A

Trigger	Time Mark TM[15:0]	Cycle Code CC[6:0]	Trigger Type TYPE[3:0]	Mess. No. MNR[6:0]
0	Mark1	0000100	Tx_Trigger_Single	7
1	Mark1	1000000	Rx_Trigger	3
2	Mark1	1000011	Tx_Trigger_Single	7
3	Mark3	1000001	Tx_Trigger_Merged	2
4	Mark3	1000011	Rx_Trigger	5
5	Mark4	1000001	Tx_Trigger_Arbitration	4
6	Mark5	1000100	Rx_Trigger	6
7	Mark6	N/A	Tx_Ref_Trigger	0(Ref)
8	Mark7	N/A	Watch_Trigger	N/A
9	N/A	N/A	End_of_List	N/A

Tx_Trigger_Single, Tx_Trigger_Continous, Tx_Trigger_Merged, Tx_Trigger_Arbitration, Rx_Trigger, and Time_Base_Trigger are only valid for the specified cycle code. For all other trigger types the cycle code is ignored.

The FSE starts the basic cycle by scanning the trigger list starting from zero until a trigger with time mark that is greater than the cycle time is reached, CC matches the actual cycle count, or a trigger of type Tx_Ref_Trigger, Tx_Ref_Trigger_Gap, Watch_Trigger, or Watch_Trigger_Gap is encountered.

When the cycle time reaches TM, the action defined by TYPE and MNR is started. There is an error in the configuration when it reaches End_of_List. At Mark6, the reference message (always TxRef) is transmitted. After transmission, the FSE returns to the beginning of the trigger list. When it reaches Watch Trigger at Mark7, the node is unable to transmit the reference message; error treatment is then started.

26.5.2.3.4 Detection of configuration errors

A configuration error is signaled via TTOST.EL = 11 (severity 3) when:

- The FSE comes to a trigger in the list with a CC that matches the current cycle count but with a time mark less than the cycle time
- The previous active trigger was a Tx_Trigger_Merged and the FSE comes to a trigger in the list with a CC that matches the current cycle count but is it not a Tx_Trigger_Merged , Tx_Trigger_Arbitration nor a Time_Base_Trigger nor an Rx_Trigger
- The FSE of a node with TTOCF.TM = 0 (time slave) encounters a Tx_Ref_Trigger or a Tx_Ref_Trigger_Gap
- Any time mark placed inside the TX enable window (defined by TTMLM.TXEW) of a Tx_Trigger with a matching cycle code
- A time mark is placed near the time mark of a Tx_Ref_Trigger and the reference trigger offset TTOST.RTO causes a reversal of their sequential order measured in the cycle time

26.5.2.4 TTCAN schedule initialization

The synchronization to the M_TTCAN message schedule starts when CCCR.INIT is reset. The M_TTCAN operates time-triggered (TTOCF.GEN = 0) or external event\u0002synchronized time-triggered (TTOCF.GEN = 1). All the nodes start with cycle time zero at the beginning of the trigger list with TTOST.SYS = 00 (out-of-synchronization); no transmission is enabled with a reference message as an exception. The nodes in external event-synchronized time\u0002triggered operation mode ignores Tx_Ref_Trigger and Watch_Trigger and uses Tx_Ref_Trigger_Gap and Watch_Trigger_Gap, until the first reference message decides whether a gap is active.

26.5.2.4.1 Time slaves

After the configuration, a time slave will ignore its Watch_Trigger and Watch_Trigger_Gap when it does not receive any message before reaching the Watch_Triggers. When it reaches Init_Watch_Trigger, interrupt flag TTIR.IWT is set, the FSE is frozen, and the cycle time becomes invalid. However, the node still takes part in CAN bus communication (to give acknowledge or to send error flags). The first received reference message will restart the FSE and the cycle time.

***Note**: Init_Watch_Trigger is not part of the trigger list. It is implemented as an internal counter that counts up to 0xFFFF = maximum cycle time.*

When a time slave receives any message but the reference message before reaching the Watch_Triggers, it will assume a fatal error (TTOST.EL = 11, severity 3), set interrupt flag TTIR.WT, switches off the CAN bus output, and enters the bus monitoring mode (CCCR.MON set to '1'). In the Bus Monitoring mode, it receive messages, but cannot send any dominant bits and therefore, it is not acknowledged.

***Note**: To leave the fatal error state, the host must set CCCR.INIT to '1'. After reset of CCCR.INIT, the node restarts TTCAN communication.*

When no error is encountered during synchronization, the first reference message sets TTOST.SYS = 01 (synchronizing), the second sets the TTCAN synchronization state (depending on its Next_is_Gap bit) to TTOST.SYS = 11 (In_Schedule) or TTOST.SYS = 10 (In_Gap), enabling all Tx_Triggers and Rx_Triggers.

26.5.2.4.2 Potential time master

After the configuration, a potential time master will start the transmission of a reference message when it reaches its Tx_Ref_Trigger (or its Tx_Ref_Trigger_Gap when in external event-synchronized time-triggered operation). It will ignore the Watch_Trigger and Watch_Trigger_Gap when it does not receive any message or transmit the reference message before reaching the Watch_Triggers (the reason assumed is that all other nodes

are in reset or configuration mode and does not acknowledge). When it reaches Init_Watch_Trigger, the attempted transmission is aborted, the interrupt flag TTIR.IWT is set, the FSE is frozen, and the cycle time will become invalid, but the node will still take part in the CAN bus communication (to acknowledge or send error flags). Resetting TTIR.IWT will re-enable the transmission of reference messages until the next time the Init_Watch_Trigger condition is met, or another CAN message is received. The FSE will be restart upon reception of a reference message.

When a potential time master reaches the Watch_Triggers after it has received any message other than the reference message, it will assume a fatal error (TTOST.EL = 11, severity 3), set interrupt flag TTIR.WT, switch off its CAN bus output, and enter the bus monitoring mode (CCCR.MON set to '1'). In bus monitoring mode, it is still able to receive messages, but it cannot send any dominant bits and therefore cannot acknowledge.

When no error is detected during initialization, the first reference message sets TTOST.SYS = 01 (synchronizing), the second sets the TTCAN synchronization state (depending on its Next_is_Gap bit) to TTOST.SYS = 11 (In_Schedule) or TTOST.SYS = 10 (In_Gap), enabling all Tx_Triggers and Rx_Triggers.

A potential time master becomes the current time master (TTOST.MS = 11) when it is the transmitter of the last reference message; otherwise, it is the backup time master (TTOST.MS = 10).

When all potential time masters finish the configuration, the node with the highest time master priority in the network becomes the current time master.

26.5.3 TTCAN gap control

All functions related to gap control apply only when the M_TTCAN is operated in external event-synchronized time-triggered mode (TTOCF.GEN = 1). In this operation mode, the TTCAN message schedule may be interrupted by inserting gaps between the basic cycles of the system matrix. All nodes connected to the CAN network should be configured for external event-synchronized time-triggered operation.

During a gap, all transmissions are stopped and the CAN bus remains idle. A gap is finished when the next reference message starts a new basic cycle. The gap starts at the end of a basic cycle that was started by a reference message with a bit Next_is_Gap = 1; for example, gaps are initiated by the current time master.

The current time master has two options to initiate a gap. A gap can be initiated under software control when the application program writes TTOCN.NIG = 1. The Next_is_Gap bit is transmitted as '1' with the next reference message. A gap can also be initiated under hardware control when the application program writes TTOCN.GCS = 1. When a reference message is started and TTOCN.GCS is set, Next_is_Gap = 1 will be set.

As soon as that reference message is completed, the TTOST.WFE bit will announce the gap to the time between the master and replicas. The current basic cycle will continue until its last time window. The time after the last time window is the gap time.

For the actual time master and the potential time masters, TTOST.GSI will be set when the last basic cycle has finished and the gap time starts. In nodes that are time slaves, the TTOST.GSI bit will remain at '0'.

When a potential time master is in synchronization state In_Gap (TTOST.SYS = 10), it has four options to as follows:

- Under software control by writing TTOCN.FGP = 1
- Under hardware control (TTOCN.GCS = 1), TTOCN.FGP is automatically set when an edge from HIGH to LOW at the internal event trigger input pin is detected and restarts the schedule
- The third option is a time-triggered restart. When TTOCN.TMG = 1, the next register time mark interrupt (TTIR.RTMI = 1) will set TTOCN.FGP and start the reference message
- Any potential time master will finish a gap when it reaches its Tx_Ref_Trigger_Gap, assuming that the event to synchronize to did not occur on time

None of these options can cause a basic cycle to be interrupted with a reference message.

Setting TTOCN.FGP after the gap time has started will start the transmission of a reference message immediately and synchronizes the message schedule. When TTOCN.FGP is set before the gap time has started

(while the basic cycle is still in progress), the next reference message is started at the end of the basic cycle, at the Tx_Ref_Trigger – there will be no gap time in the message schedule.

In time-triggered operation, the bit Next_is_Gap = 1 in the reference message will be ignored, and the bits TTOCN.NIG, TTOCN.FGP, and TTOCN.TMG will be considered.

26.5.4 Stop watch

The stop watch function enables capturing of M_TTCAN internal time values (local time, cycle time, or global time) triggered by an external event.

To enable the stop watch function, the application program must first define local time, cycle time, or global time as the stop watch source via TTOCN.SWS. When TTOCN.SWS is not equal to ‘00’ and TT Interrupt Register flag TTIR.SWE is ‘0’, the actual value of the time selected by TTOCN.SWS will be copied into TTCPT.SWV on the next rising/falling edge (as configured via TTOCN.SWP) on stop watch trigger. This will set the interrupt flag TTIR.SWE. After the application program has read TTCPT.SWV, it may enable the next stop watch event by resetting TTIR.SWE to '0'.

26.5.5 Local time, cycle time, global time, and external clock synchronization

There are two possible levels in time-triggered CAN: Level 1 and Level 2. Level 1 provides only time-triggered operation using cycle time. Level 2 additionally provides increased synchronization quality, global time, and external clock synchronization. In both levels, all timing features are based on a local time base – the local time.

The local time is a 16-bit cyclic counter, it is incremented once in each NTU. Internally the NTU is represented by a 3-bit counter, which can be regarded as a fractional part (three binary digits) of the local time. Generally, the 3-bit NTU counter is incremented eight times each NTU. If the length of the NTU is shorter than eight CAN clock periods (as may be configured in Level 1, or as a result of clock calibration in Level 2), the length of the NTU fraction is adapted, and the NTU counter is incremented only four times each NTU.

Figure 268 describes the synchronization of the cycle time and global time, performed in the same manner by all TTCAN nodes, including the time master. Any message received or transmitted invokes a capture of the local time taken at the message's frame synchronization event. This frame synchronization event occurs at the sample point of each Start-of-Frame (SoF) bit and causes the local time to be stored as Sync_Mark. Sync_Marks and Ref_Marks are captured including the 3-bit fractional part.

Whenever a valid reference message is transmitted or received, the internal Ref_Mark is updated from the Sync_Mark. The difference between Ref_Mark and Sync_Mark is the Cycle Sync Mark (Cycle Sync Mark = Sync_Mark – Ref_Mark) stored in the TTCSM register. The most significant 16 bits of the difference between Ref_Mark and the actual value of the local time is the cycle time (Cycle Time = Local Time – Ref_Mark).

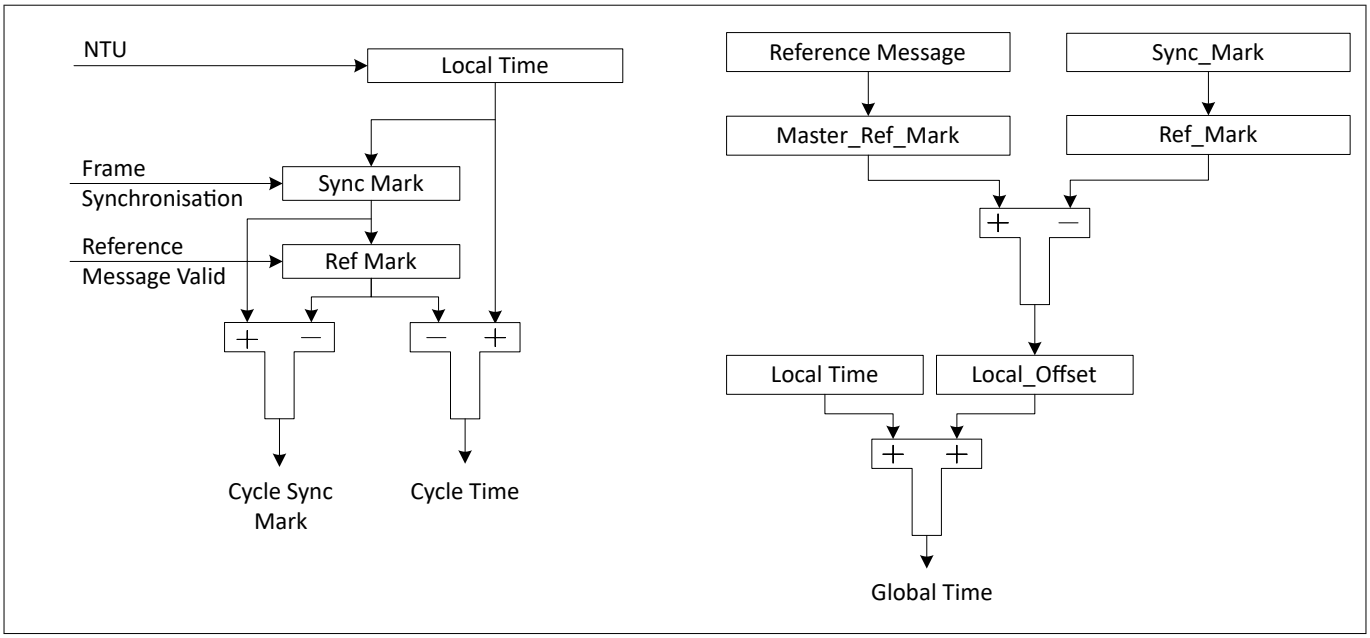


Figure 268 Cycle time and global time synchronization

The cycle time that can be read from TTCTC.CT is the difference of the node's local time and Ref_Mark, both synchronized into the host clock domain and truncated to 16 bits.

The global time exists for TTCAN Level 0 and Level 2 only, in Level 1 it is invalid. The node's view of the global time is the local image of the global time in (local) NTUs. After configuration, a potential time master will use its own local time as global time. This is done by transmitting its own Ref_Marks as Master_Ref_Marks in the reference message (bytes 3 and 4). The global time is read from TTLGT.GT is the sum of the node's local time and its local offset, both synchronized into the host clock domain and truncated to 16 bit. The fractional part is used for clock synchronization only.

A node that receives a reference message calculates its local offset to the global time by comparing its local Ref_Mark with the received Master_Ref_Mark (see Figure 268). The node's view of the global time is local time + local offset. In a potential time master that has never received another time master's reference message, Local_Offset will be zero. When a node becomes the current time master after having received other reference messages first, Local_Offset will be frozen at its last value. In the time receiving nodes, Local_Offset may be subject to small adjustments, due to clock drift, when another node becomes time master, or when there is a global time discontinuity, signaled by Disc_Bit in the reference message. An exception of global time discontinuity, it is provided to the application program by the TTLGT register and smoothed by a low-pass filtering to have a continuous monotonic value.

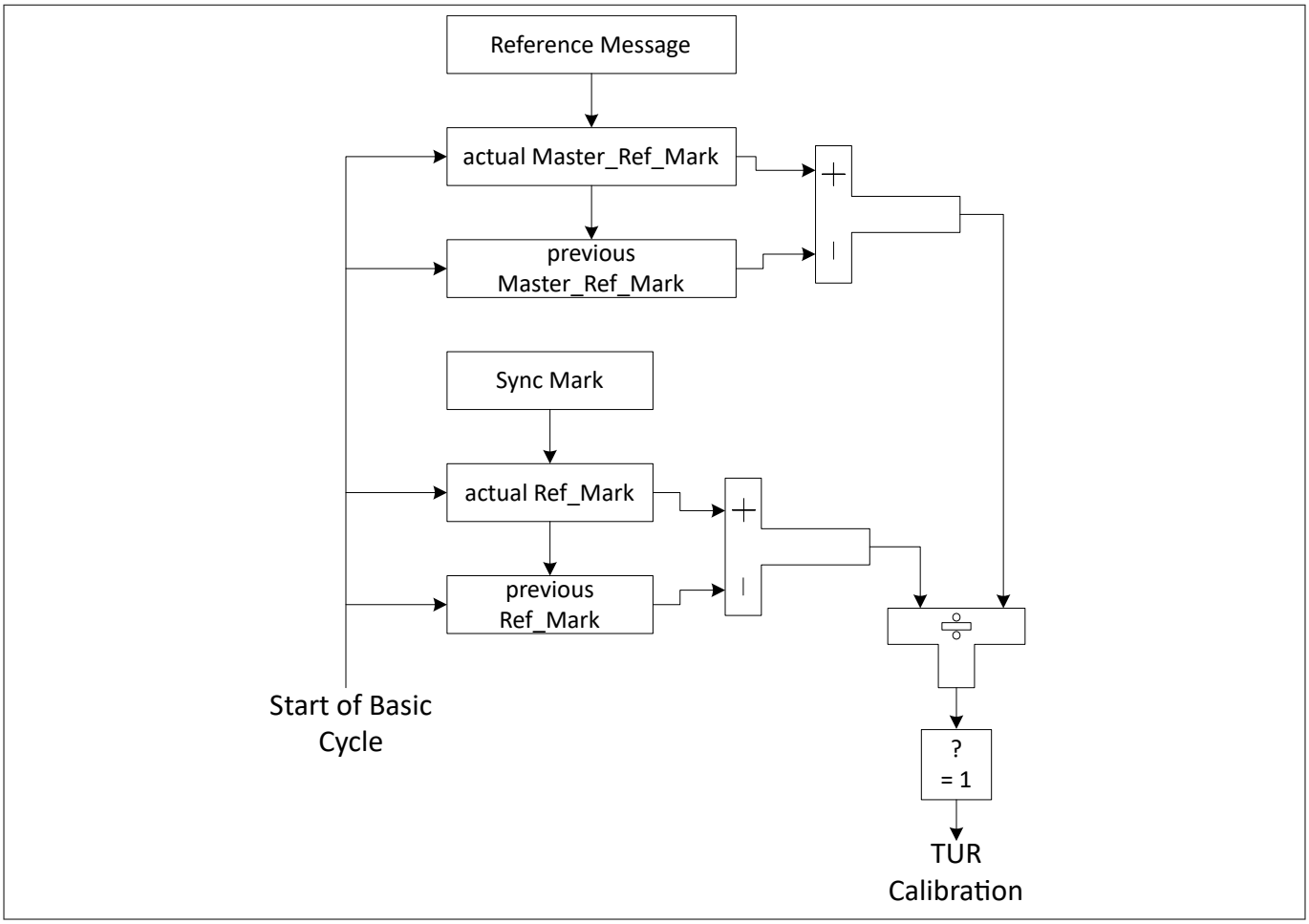


Figure 269 TTCAN Level 0 and Level 2 drift compensation

Figure 269 illustrates how in TTCAN Level 0 and Level 2 the receiving node compensates the drift between its own local clock and the time master's clock by comparing the length of a basic cycle in local time and in global time. If there is a difference between the two values, and the Disc_Bit in the reference message is not set, a new value for TURNA.NAV is calculated. If the synchronization deviation (SD) = |NC – TURNA.NAV| ≤ SDL, the new value for TURNA.NAV takes effect. Otherwise, the automatic drift compensation is suspended.

In TTCAN Level 0 and Level 2, TTOST.QCS indicates whether the automatic drift compensation is active or suspended. In TTCAN Level 1, TTOST.QCS is always '1'.

The current time master may synchronize its local clock speed and the global time phase to an external clock source. This is enabled by the bit TTOCF.EECS. The stop watch function (see Stop watch) may be used to measure the difference in clock speed between the local clock and the external clock. The local clock speed is adjusted by first writing the newly calculated Numerator Configuration Low to TURCF.NCL (TURCF.DC cannot be updated during operation). The new value takes effect by writing TTOCN.ECS to '1'.

The global time phase is adjusted by first writing the phase offset into the TT Global Time Preset register (TTGTP). The new value takes effect by writing TTOCN.SGT to '1'. The first reference message transmitted after the global time phase adjustment will have the Disc_Bit set to '1'.

TTOST.QGTP shows whether the node's global time is in phase with the time master's global time. TTOST.QGTP is permanently '0' in TTCAN Level 1 and when the SDL is exceeded in TTCAN Level 0,2 (TTOST.QCS = 0). It is temporarily '0' while the global time is low-pass filtered to supply the application with a continuous monotonic value. There is no low-pass filtering when the last reference message contains a Disc_Bit = 1 or when TTOST.QCS = 0.

26.5.6 TTCAN error level

The ISO 11898-4 specifies four levels of error severity:

Table 275 Levels of error severity

Level	Severity	Description
S0	No error	
S1	Warning	Only notification of application, reaction application-specific.
S2	Error	Notification of application. All transmissions in exclusive or arbitrating time windows are disabled (that is, no data or remote frames may be started). Potential time mas\u0002ters still transmit reference messages with the Reference Trigger Offset TTOST.RTO set to the maximum value of 127.
S3	Severe error	Notification of application. All CAN bus operations are stopped; transmission of dominant bits is not allowed, and CCCR.MON is set. The S3 error condition remains active until the application updates the configuration (set CCCR.CCE).

If several errors are detected at the same time, the highest severity prevails. When an error is detected, the application is notified by TTIR.ELC. The error level is monitored by TTOST.EL.

The M_TTCAN signals the following error conditions as required by ISO 11898-4:

Table 276 Error condition

Error condition	Description
Config_Error (S3)	Sets error level TTOST.EL to ‘11’ when a merged arbitrating time window is not properly closed or when there is a Tx_Trigger with a time mark beyond the Tx_Ref_Trigger.
Watch_Trigger_Reached (S3)	Sets error level TTOST.EL to ‘11’ when a watch trigger is reached because the reference message is missing.
Application_Watchdog (S3)	Sets error level TTOST.EL to ‘11’ when the application fails to serve the application watchdog. The application watchdog is configured via TTOCF.AWL. It is served by reading the TTOST register. When the watchdog is not served in time, bit TTOST.AWE and interrupt flag TTIR.AW are set, all TTCAN communication is stopped, and the M_TTCAN is set into bus monitoring mode (CCCR.MON set to '1').

(table continues...)

Table 276 (continued) Error condition

Error condition	Description
CAN_Bus_Off (S3)	Entering CAN_Bus_Off state sets the error level TTOST.EL to‘11’. CAN_Bus_Off state is signaled by PSR.BO = 1 and CCCR.INIT = 1.
Scheduling_Error_2 (S2)	Sets error level TTOST.EL to ‘10’ if the MSC of one Tx_Trigger has reached 7. In addition, the interrupt flag TTIR.SE2 is set. TTOST.EL is reset to 00 at the beginning of a matrix cycle when no Tx_Trigger has an MSC of 7 in the preceding matrix cycle.
Tx_Overflow (S2)	Sets error level TTOST.EL to ‘10’ when the TX count is equal or higher than the expected number of Tx_Triggers TTMLM.ENTT and a Tx_Trigger event occurs. In addition, the interrupt flag TTIR.TXO is set. TTOST.EL is reset to 00 when the TX count is no more than TTMLM.ENTT at the start of a new matrix cycle.
Scheduling_Error_1 (S1)	Sets the error level TTOST.EL to ‘01’ if within one matrix cycle. The difference between the maximum MSC and the minimum MSC for all trigger memory elements (of exclusive time windows) is larger than 2, or if one of the MSCs of an exclusive Rx_Trigger has reached 7. In addition, interrupt flag TTIR.SE1 is set. If within one matrix cycle none of these conditions is valid, TTOST.EL is reset to 00.
Tx_Underflow (S1)	Sets error level TTOST.EL to ‘01’ when the TX count is less than the expected number of Tx_Triggers TTMLM.ENTT at the start of a new matrix cycle. In addition, interrupt flag TTIR.TXU is set. TTOST.EL is reset to 00 when the TX count is at least TTMLM.ENTT at the start of a new matrix.

26.5.7 TTCAN message handling

26.5.7.1 Reference message

For potential time masters, the identifier of the reference message is configured via TTRMC.RID. No dedicated TX buffer is required for transmission of the reference message. When a reference message is transmitted, the first data byte (TTCAN Level 1) and the first four data bytes (TTCAN Level 0 and Level 2) will be provided by the FSE.

If the Payload Select reference message TTRMC.RMPS is set, the rest of the reference message's payload (Level 1: bytes 2-8, Level 0 and Level 2: bytes 5-6) is taken from TX Buffer 0. In this case, the data length DLC code from message buffer 0 is used.

Table 277 Number of data bytes transmitted with a reference messages

TTRMC.RMPS	TXBRP.TRP0	Level 0	Level 1	Level 2
0	0	4	1	4
0	1	4	1	4
1	0	4	1	4
1	1	4 + MB0	1 + MB0	4 + MB0

To send additional payload with the reference message in Level 1, a DLC > 1 should be configured. For Level 0 and Level 2 a DLC > 4 is required. In addition, the transmission request pending bit TXBRP.TRP0 of message buffer 0 must be set (see Table 277). If TXBRP.TRP0 is not set when a reference message is started, the reference message is transmitted with the data bytes supplied by the FSE only.

For acceptance filtering of reference messages, the Reference Identifier TTRMC.RID is used.

26.5.7.2 Message reception

Message reception is done via the two RX FIFOs in the same way as for event-driven CAN communication.

The MSC is part of the corresponding trigger memory element and must be initialized to zero during configuration. It is updated while the M_TTCAN is in synchronization states In_Gap or In_Schedule. The update happens at the message's Rx_Trigger. At this point, it is checked at which acceptance filter element the latest message received in this basic cycle is matched. The matching filter number is stored as the acceptance filter result. If this is the same as the filter number defined in this trigger memory element, the MSC is decremented by one. If the acceptance filter result is not the same filter number as defined for this filter element, or if the acceptance filter result is cleared, the MSC is incremented by one. At each Rx_Trigger and at each start of the cycle, the last acceptance filter result is cleared.

The time mark of an Rx_Trigger should be set to a value that ensures reception and acceptance filtering for the targeted message is completed. This should consider the RAM access time and the order of the filter list. It is recommended that the filters used for Rx_Triggers are placed at the beginning of the list. It is not recommended to use an Rx_Trigger for the reference message.

26.5.7.3 Message transmission

For time-triggered message transmission, the M_TTCAN supplies 32 dedicated TX buffers (see TTCAN Configuration on page 379). A TX FIFO or TX queue is not available when the M_TTCAN is configured for time-triggered operation (TTOCF.OM = 01 or 10).

Each Tx_Trigger in the trigger memory points to a particular TX buffer containing a specific message. There may be more than one Tx_Trigger for a given TX buffer if that TX buffer contains a message that is to be transmitted more than once in a basic cycle or matrix cycle.

The application program must update the data regularly and on time, synchronized to the cycle time. The host CPU should ensure that no partially updated messages are transmitted, the host proceeds as follows:

Tx_Trigger_Single/Tx_Trigger_Merged/Tx_Trigger_Arbitration:

- Check whether the previous transmission is completed by reading TXBTO
- Update the TX buffer's configuration and/or payload
- Issue an Add Request to set the TX Buffer Request Pending bit

Tx_Trigger_Continous:

- Issue a cancellation request to reset the TX Buffer Request Pending bit
- Check whether the cancellation has finished by reading TXBCF

- Update TX buffer configuration and/or payload
- Issue an Add Request to set the TX Buffer Request Pending bit

The message MSC stored with the corresponding Tx_Trigger provides information on the success of the transmission.

The MSC is incremented by one when the transmission cannot be started because the CAN bus was not idle within the corresponding transmit enable window or when the message was started but could not be completed successfully. The MSC is decremented by one when the message is transmitted successfully or when the message could have been started within its transmit enable window but was not started because transmission is disabled (M_TTCAN in Error Level S2 or the host is disabled this particular message).

The TX buffers may be managed dynamically – several messages with different identifiers may share the same TX buffer element. In this case the host must ensure that no transmission request is pending for the TX buffer element to be reconfigured by checking TXBRP.

If a TX buffer with pending transmission request should be updated, the host must first issue a cancellation request and check whether the cancellation has completed by reading TXBCF before it starts updating.

The TX handler will transfer a message from the message RAM to the intermediate output buffer at the trigger element, which becomes active immediately before the Tx_Trigger element that defines the beginning of the transmit window. During and after transfer time, the transmitting message may not be updated and its TXBRP bit may not be changed. To control this transfer time, an additional trigger element may be placed before the Tx_Trigger. An example is a Time_Base_Trigger, which does not cause any other action. The difference in time marks between the Tx_Trigger and the preceding trigger should be large enough to guarantee that the TX handler can read four words from the message RAM even at high RAM access load from other modules.

26.5.7.3.1 Transmission in exclusive time windows

A transmission starts time-triggered when the cycle time reaches the time mark of a Tx_Trigger_Single or Tx_Trigger_Continous. There is no arbitration on the bus with messages from other nodes. The MSC is updated according to the result of the transmission attempt. After successful transmission started by a Tx_Trigger_Single, the respective TX Buffer Request Pending bit is reset. After successful transmission, by a Tx_Trigger_Continous the respective TX Buffer Request Pending bit remains set. When the transmission is not successful due to disturbances, it will be repeated the next time one of its Tx_Triggers becomes active.

26.5.7.3.2 Transmission in arbitrating time windows

A transmission is started time-triggered when the cycle time reaches the time mark of a Tx_Trigger_Arbitration. Several nodes may start to transmit at the same time. In this case the message has to arbitrate with the messages from other nodes. The MSC is not updated. When the transmission is not successful (lost arbitration or disturbance), it will be repeated the next time one of its Tx_Triggers becomes active.

26.5.7.3.3 Transmission in merged arbitrating time windows

The purpose of a merged arbitrating time window is to enable multiple nodes to send a limited number of frames, which are transmitted in a immediate sequence, and the order given by CAN arbitration. It is not intended for burst transmission by a single node. Because the node does not have exclusive access within this time window, all requested transmissions may not be successful.

Messages that have lost arbitration or were disturbed by an error, may be retransmitted inside the same merged arbitrating time window. The retransmission will not be started if the corresponding Transmission Request Pending flag was reset by a successful TX cancellation.

In single transmit windows, the TX handler transmits the message indicated by the message number of the trigger element. In merged arbitrating time windows, it can handle up to three message numbers from the trigger list. Their transmissions will be attempted in the sequence defined by the trigger list. If the time mark of

a fourth message is read before the first is transmitted (or canceled by the host), the fourth request will be ignored.

The transmission inside a merged arbitrating time window is not time-triggered. The transmission of a message may start before its time mark, or after the time mark if the bus was not idle.

The messages transmitted by a specific node inside a merged arbitrating time window will be started in the order of their Tx_Triggers. Therefore, a message with low CAN priority may prevent the successful transmission of a following message with higher priority, if there is competing bus traffic. This should be considered for the configuration of the trigger list. Time_Base_Triggers may be placed between consecutive Tx_Triggers to define the time until the data of the corresponding TX buffer needs to be updated.

26.5.8 TTCAN interrupt and error handling

The TT Interrupt Register TTIR consists of four segments. Each interrupt can be enabled separately by the corresponding bit in the TT Interrupt Enable register TTIE. The flags remain set until the host clears them. A flag is cleared by writing a '1' to the corresponding bit position.

The first segment consists of flags CER, AW, WT, and IWT. Each flag indicates a fatal error condition where the CAN communication is stopped. The error conditions require a reconfiguration of the M_TTCAN module before the communication is restarted which is an exception for IWT.

The second segment consists of the flags ELC, SE1, SE2, TXO, TXU, and GTE. Each flag indicates an error condition where the CAN communication is disturbed. If they are caused by a transient failure, such as by disturbances on the CAN bus, they will be handled by the TTCAN protocol's failure handling and do not require intervention by the application program.

The third segment consists of flags GTD, GTW, SWE, TTMI, and RTMI. The first two flags are controlled by global time events (Level 0 and Level 2 only) that require a reaction by the application program. With a Stop Watch Event, internal time values are captured. The Trigger Time Mark Interrupt notifies the application that a specific Time_Base_Trigger is reached. The Register Time Mark Interrupt signals that the time referenced by TTOCN.TMC (cycle, local, or global) equals the time mark TTTMK.TM. It can also be used to finish a gap.

The fourth segment consists of the flags SOG, CSM, SMC, and SBC. These flags provide a means to synchronize the application program to the communication schedule.

26.5.9 Level 0

TTCAN Level 0 is not part of ISO11898-4. This operation mode makes the hardware, that in TTCAN Level 2 maintains the calibrated global time base, also available for event\u0002driven CAN according to ISO 11898-1:2015.

Level 0 operation is configured via TTOCF.OM = 11. In this mode, M_TTCAN operates in event-driven CAN communication; there is no fixed schedule. The configuration of TTOCF.GEN is ignored. External event-synchronized operation is not available in Level 0. A synchronized time base is maintained by the transmission of reference messages.

In Level 0 the trigger memory is not active and need not be configured. The time mark interrupt flag (TTIR.TTMI) is set when the cycle time has reached TTOCF.IRTO × 0x200. It reminds the host to set a transmission request for message buffer 0. The Watch_Trigger interrupt flag (TTIR.WT) is set when the cycle time has reached 0xFF00. These values were chosen to have enough margin for a stable clock calibration. There are no further TT-error-checks. Register time mark interrupts (TTIR.RTMI) are also possible.

The reference message is configured as for Level 2 operation. Received reference messages are recognized by the identifier configured in register TTRMC. For the transmission of reference messages only message buffer 0 may be used. The node transmits reference messages any time that the host sets a transmission request for message buffer 0; there is no reference trigger offset.

Level 0 operation is configured via:

- TTRMC

- TTOCF except EVTP, AWL, GEN
- TTMLM except ENTT, TXEW
- TURCF

Level 0 operation is controlled via:

- TTOCN except NIG, TMG, FGP, GCS, TTMIE
- TTGTP
- TTTMK
- TTIR excluding bits CER, AW, IWT, SE2, SE1, TXO, TXU, SOG (no function)
- TTIR – the following bits have changed function:
  - TTMI not defined by trigger memory - activated at cycle time TTOCF.IRTO × 0x200
  - WT not defined by trigger memory - activated at cycle time 0xFF00

Level 0 operation is signaled via:

- TTOST excluding bits AWE, WFE, GSI, GFI, RTO (no function)

26.5.9.1 Synchronizing

Figure 270 describes the states and state transitions in TTCAN Level 0 operation. Level 0 has no In_Gap state.

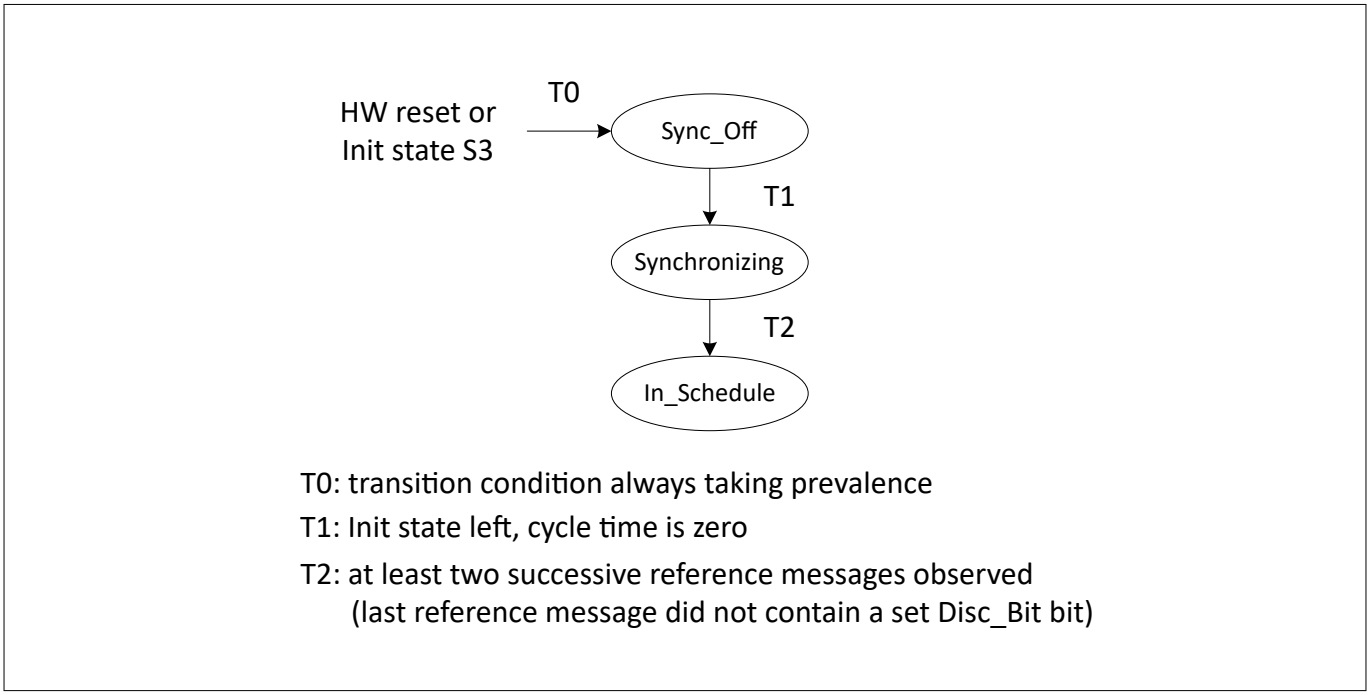


Figure 270 Level 0 Schedule Synchronization State Machine

26.5.9.2 Handling error levels

During the Level 0 operation the following error conditions occurs:

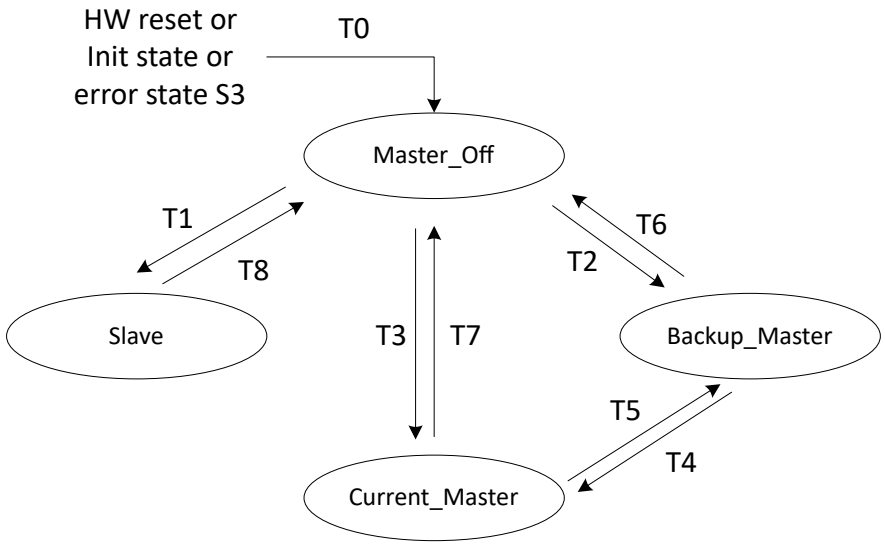
- Watch_Trigger_Reached (S3), reached cycle time 0xFF00
- CAN_Bus_Off (S3)

Because S1 and S2 errors are not possible, the error level can only switch between S0 (No Error) and S3 (Severe Error). In TTCAN Level 0, an S3 error is handled differently. When S3 error is reached, both TTOST.SYS and TTOST.MS are reset, and interrupt flags TTIR.GTE and TTIR.GTD are set.

When S3 (TTOST.EL = 11) is entered, bus monitoring mode is, contrary to TTCAN Level 1 and Level 2, not entered. S3 error level is left automatically after transmission (time master) or reception (time slave) of the next reference message.

26.5.9.3 Master slave relation

Figure 271 describes the master slave relation in TTCAN Level 0. In case of an S3 error, the M_TTCAN returns to state aster_Off



T0: transition condition always taking prevalence
T1: reference message observed when not potential master
T2: reference message observed with master priority != own master priority, error state != S3
T3: reference message observed with master priority = own master priority, error state != S3
T4: reference message observed with own master priority
T5: reference message observed with master priority higher than own master priority
T6: error state S3
T7: error state S3
T8: error state S3

Figure 271 Level 0 Master to Slave Relation

26.5.10 Synchronization to external time schedule

This feature is used to synchronize the phase of the M_TTCAN's schedule to an external schedule (for example, that of a second TTCAN network). It is applicable only when the M_TTCAN is current time master (TTOST.MS = 11).

External synchronization is controlled by the TTOCN.ESCN bit. If TTOCN.ESCN is set, at the rising edge of the internal event trigger pin, the M_TTCAN compares its actual cycle time with the target phase value configured by TTGTP.CTP.

Before setting TTOCN.ESCN, the host should adapt the phases of the two time schedules, for example, by using the TTCAN gap control (see TTCAN Gap Control on page 384). When the host sets TTOCN.ESCN, TTOST.SPL is set.

If the difference between the cycle time and the target phase value TTGTP.CTP at the trigger is greater than nine NTU, the phase lock bit TTOST.SPL is reset, and the interrupt flag TTIR.CSM is set. TTOST.SPL is also reset (and TTIR.CSM is set), when another node becomes time master.

If both TTOST.SPL and TTOCN.ESCN are set, and if the difference between the cycle time and the target phase value TTGTP.CTP is less or equal nine NTU, the phase lock bit TTOST.SPL remains set, and the measured difference is used as the reference trigger-offset value to adjust the phase at the next transmitted reference message.

***Note**: The rising edge detection at the internal pin is enabled at the start of each basic cycle. The first rising edge triggers the comparison of the actual cycle time with TTGTP.CTP. All further edges until the beginning of the next basic cycle are ignored.*

26.6 Setup procedures

This section provides example procedures for configurations of the M_TTCAN channel.

26.6.1 General program flow

This is a general flow to configure the M_TTCAN module.

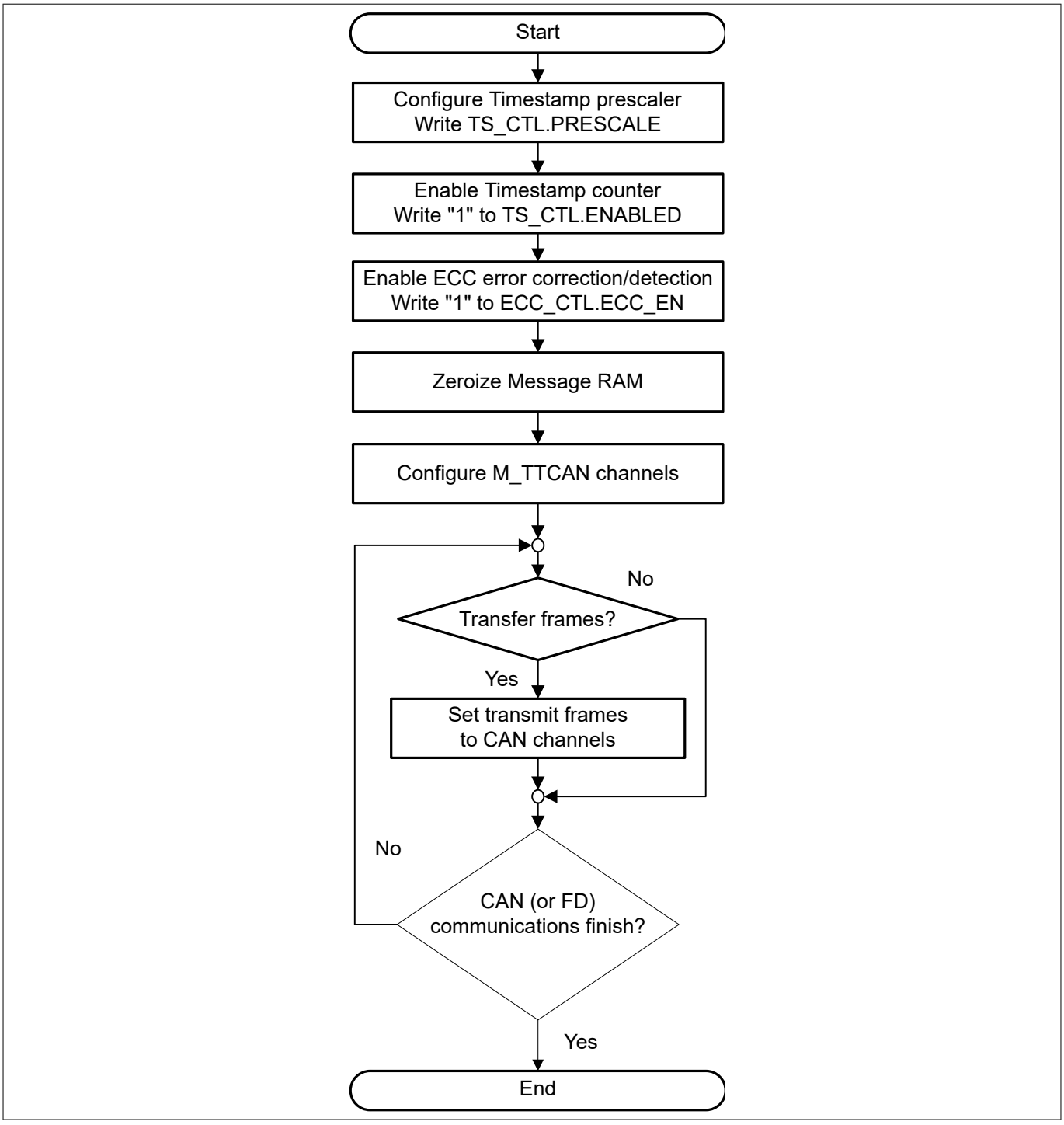


Figure 272 General program flow

26.6.2 Clock stop request

To save power, the application stops to provide the clock to the unused M_TTCAN channel by the steps shown in Figure #.

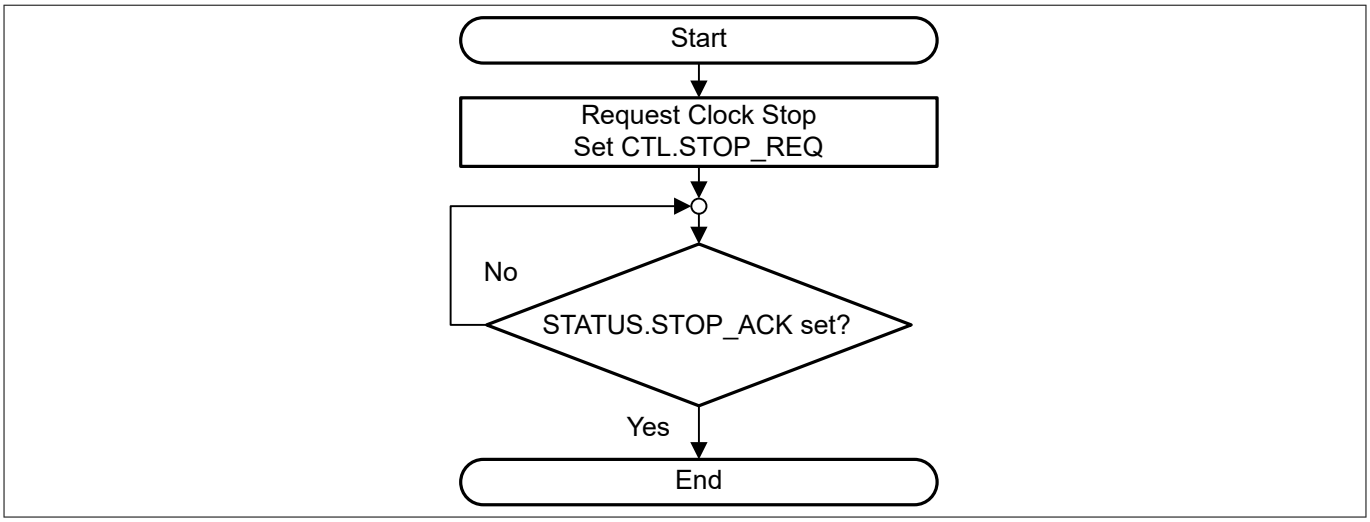


Figure 273 Clock stop request procedure

To resume the providing clock, the CTL.STOP_REQ bit is reset.

26.6.3 MRAM OFF operation

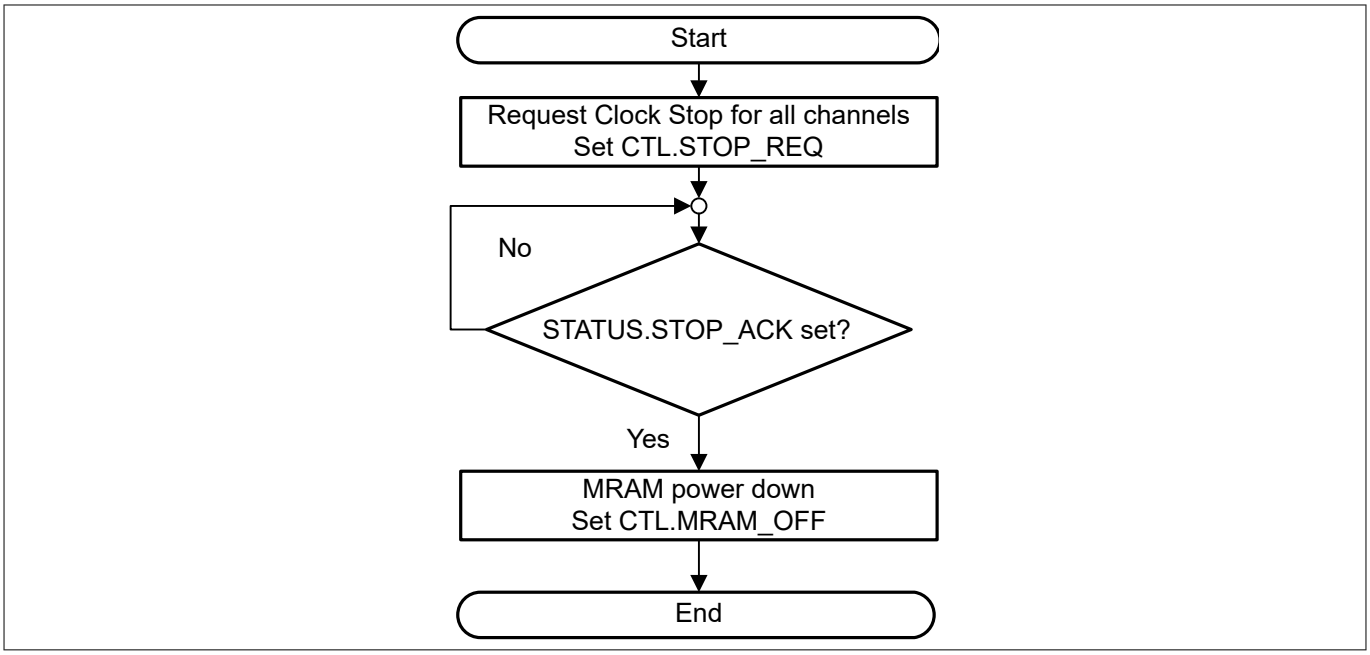


Figure 274 Message RAM OFF operation

26.6.4 MRAM ON operation

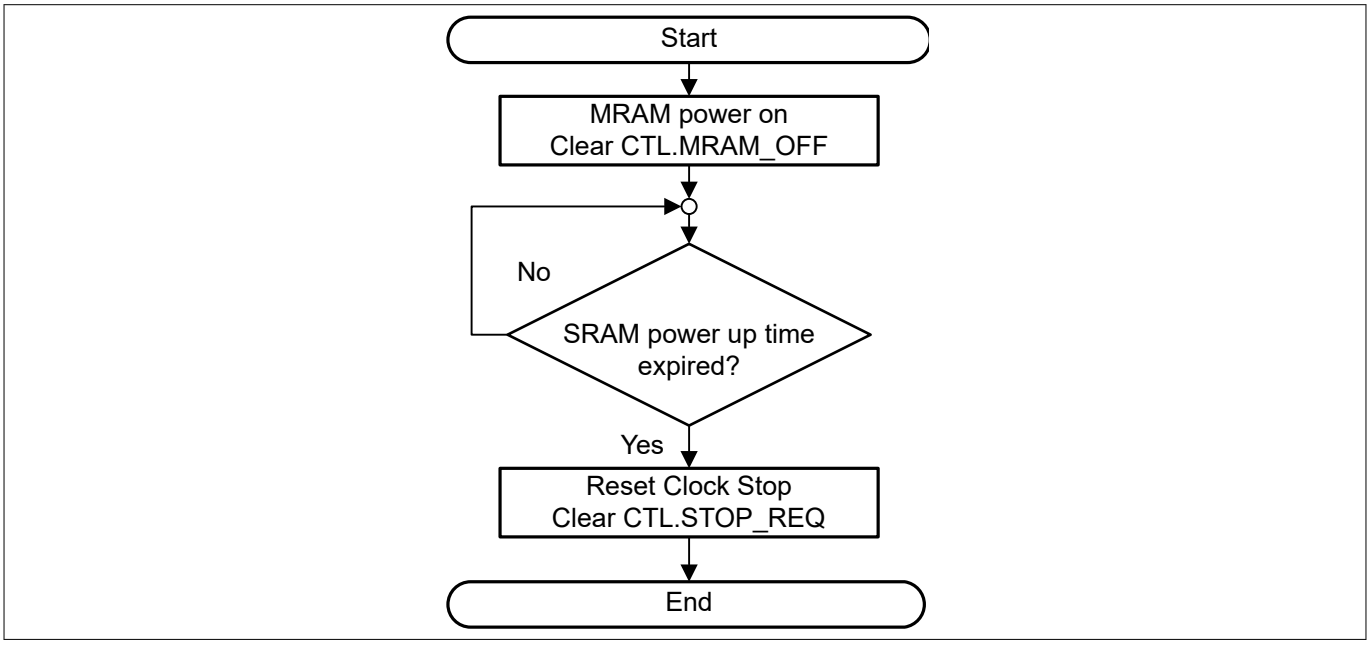


Figure 275 Message RAM On procedure

After switching MRAM ON again, software needs to allow a certain power-up time before MRAM can be used; that is, before STOP_REQ can be de-asserted. The power-up time is equivalent to the system SRAM power-up time specified in the CPUSS.RAM_PWR_DELAY_CTL register.

26.6.5 Procedures specific to M_TTCAN channel

This section describes sample procedure for the M_TTCAN channel. Figure 276 shows the general program flow.

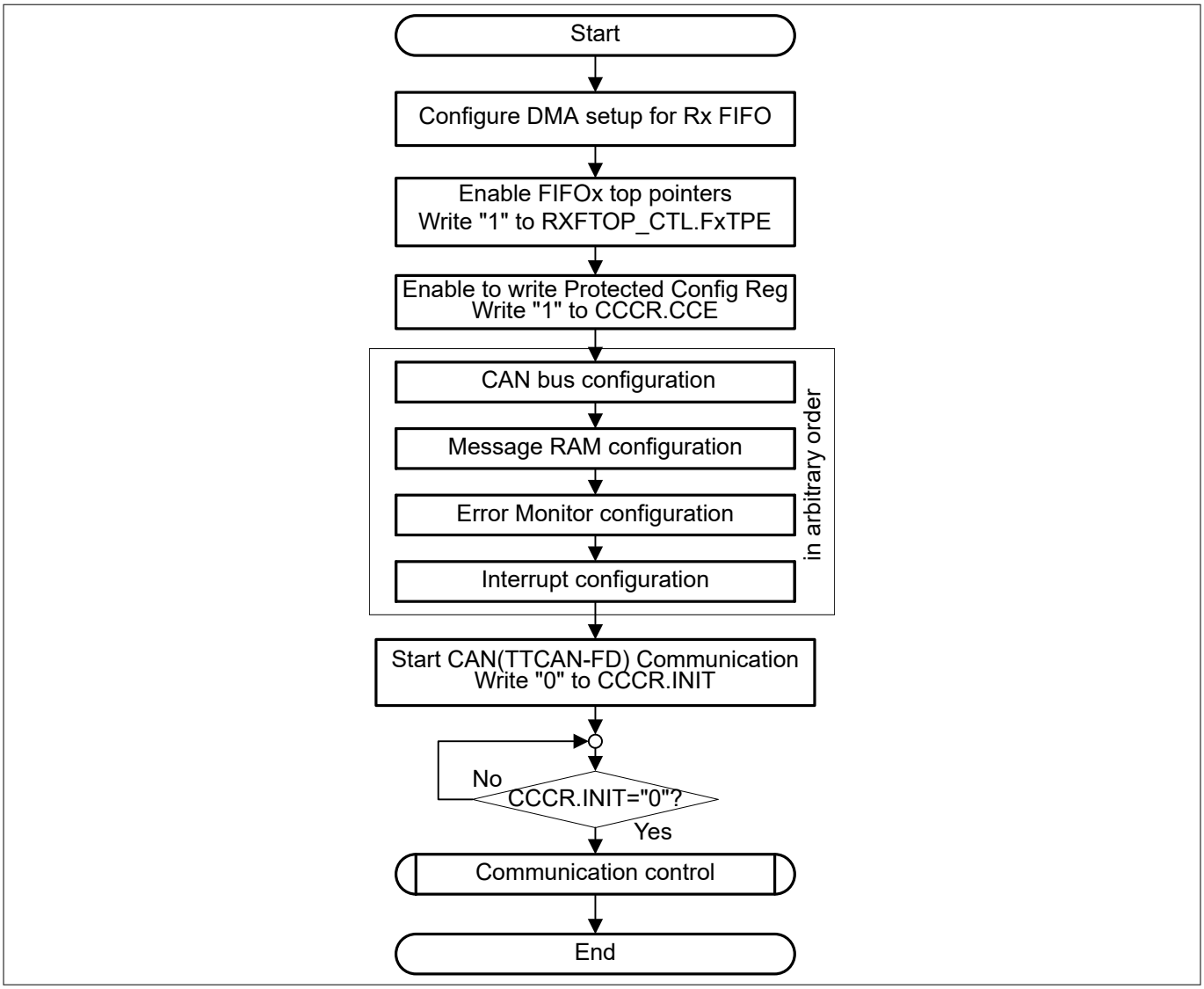


Figure 276 Configuration sequence specific to channel

26.6.6 CAN Bus configuration

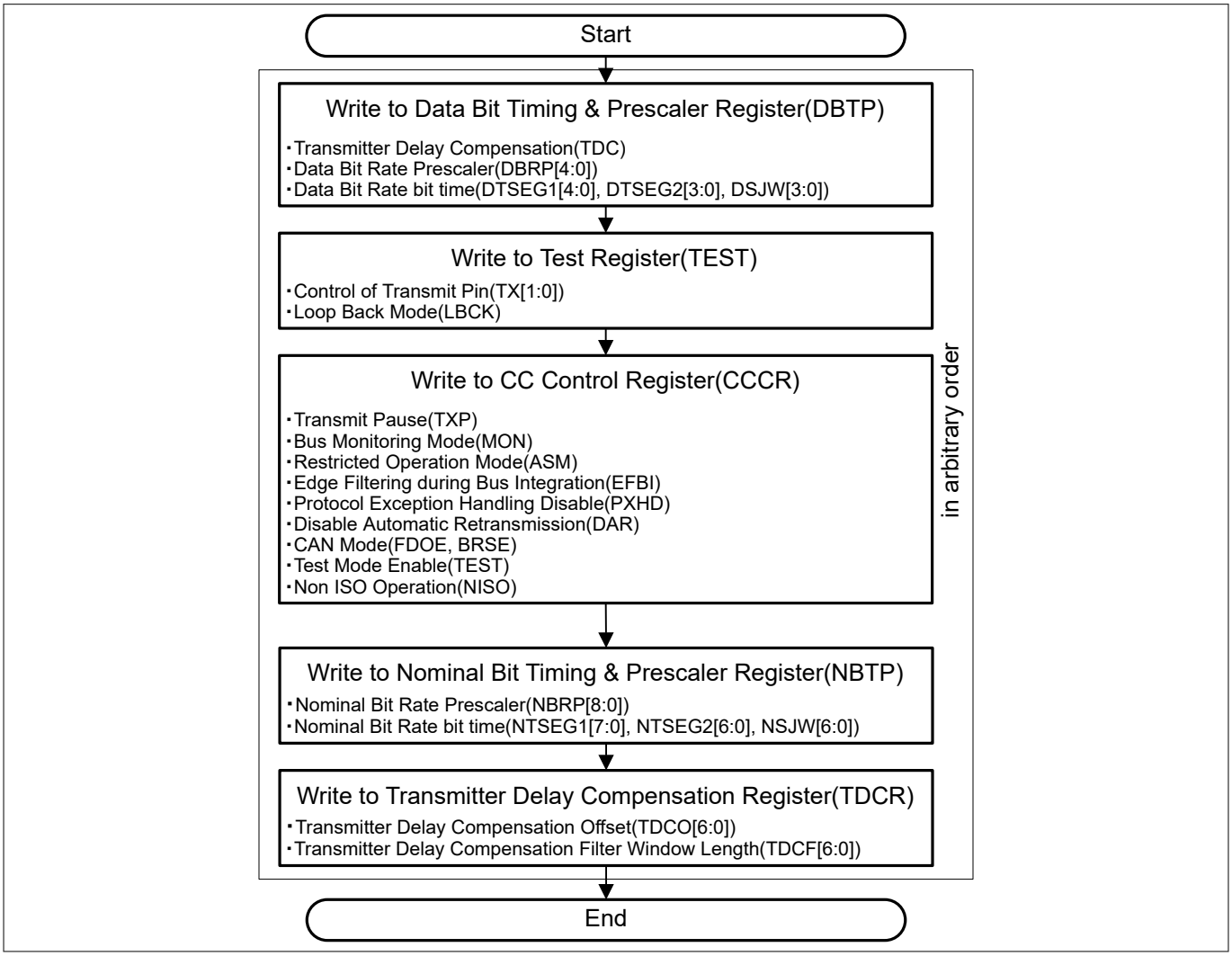


Figure 277 Configuration required for CAN bus

26.6.7 Message RAM configuration

Figure 278 shows an overview of the message RAM configuration.

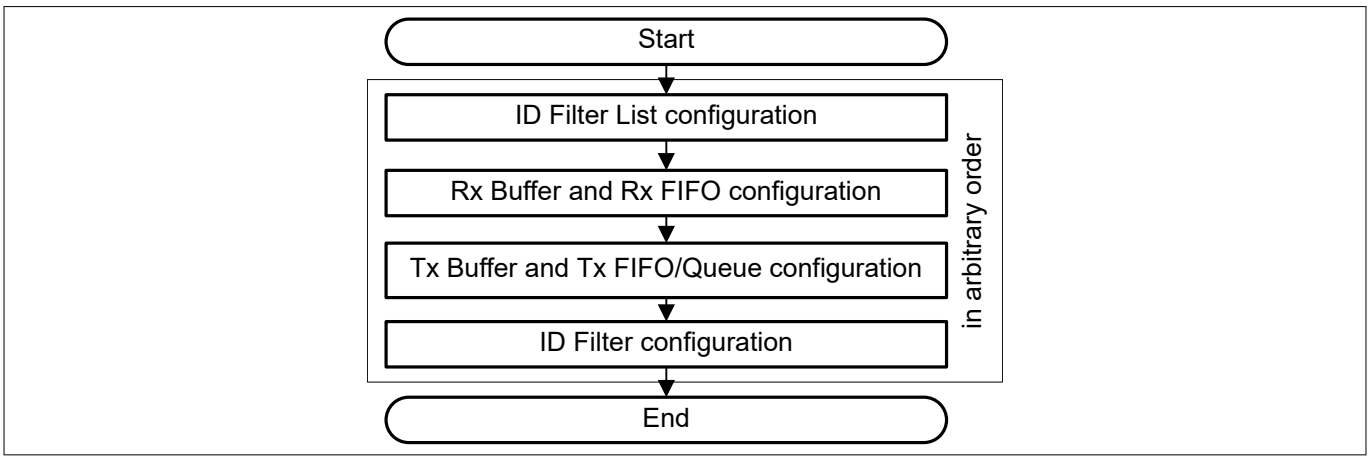


Figure 278 Message RAM configuration overview

Each configuration mentioned in the overview is detailed in the following figures.

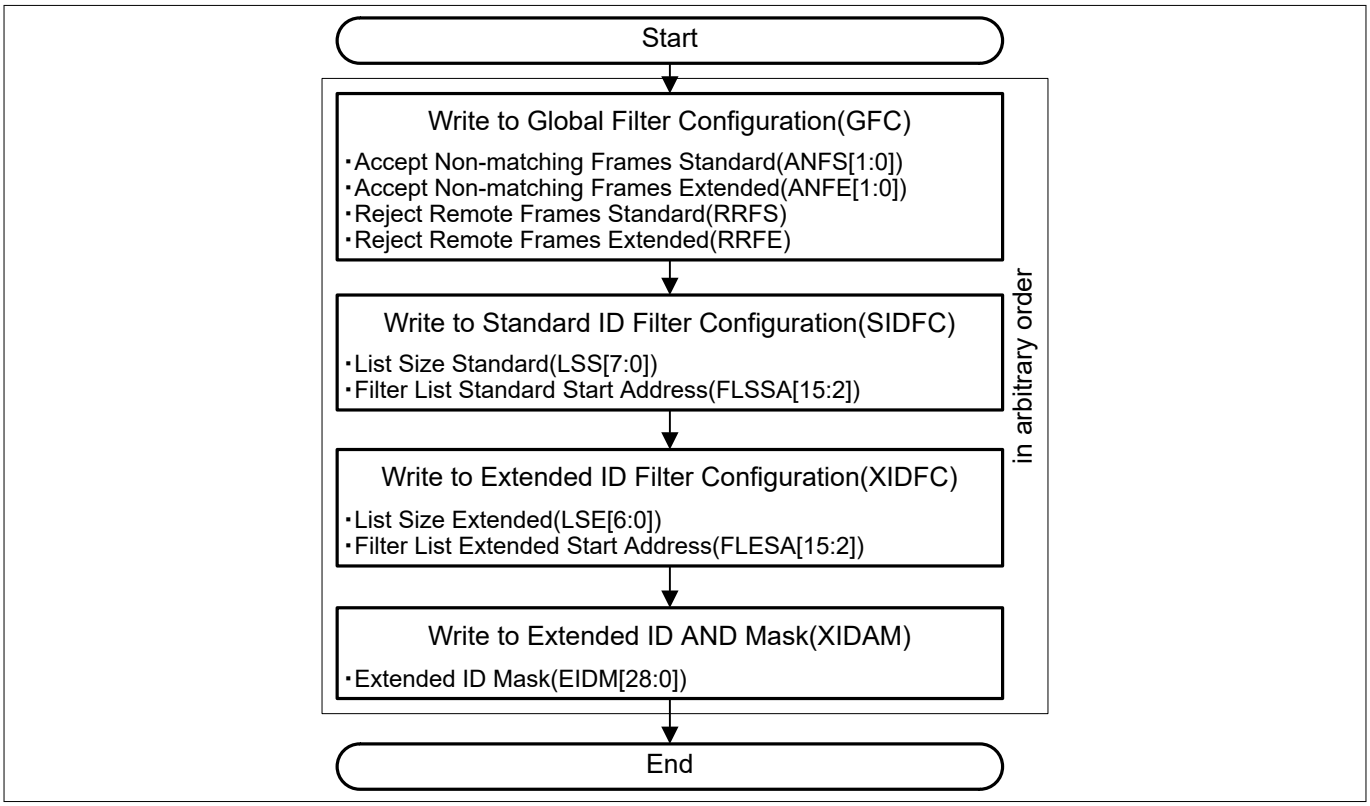


Figure 279 ID filter list configuration

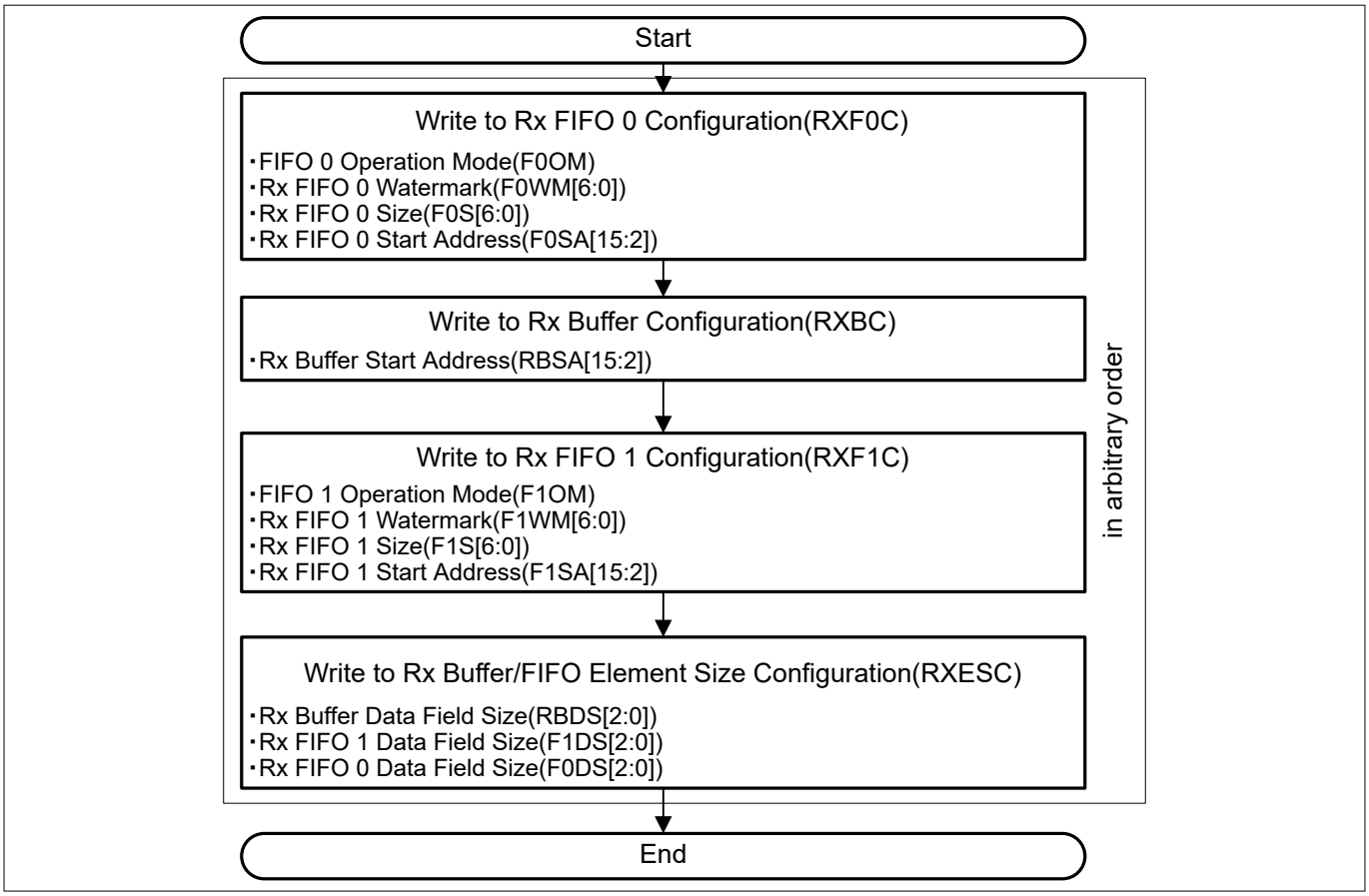


Figure 280 RX FIFO and RX buffer configuration

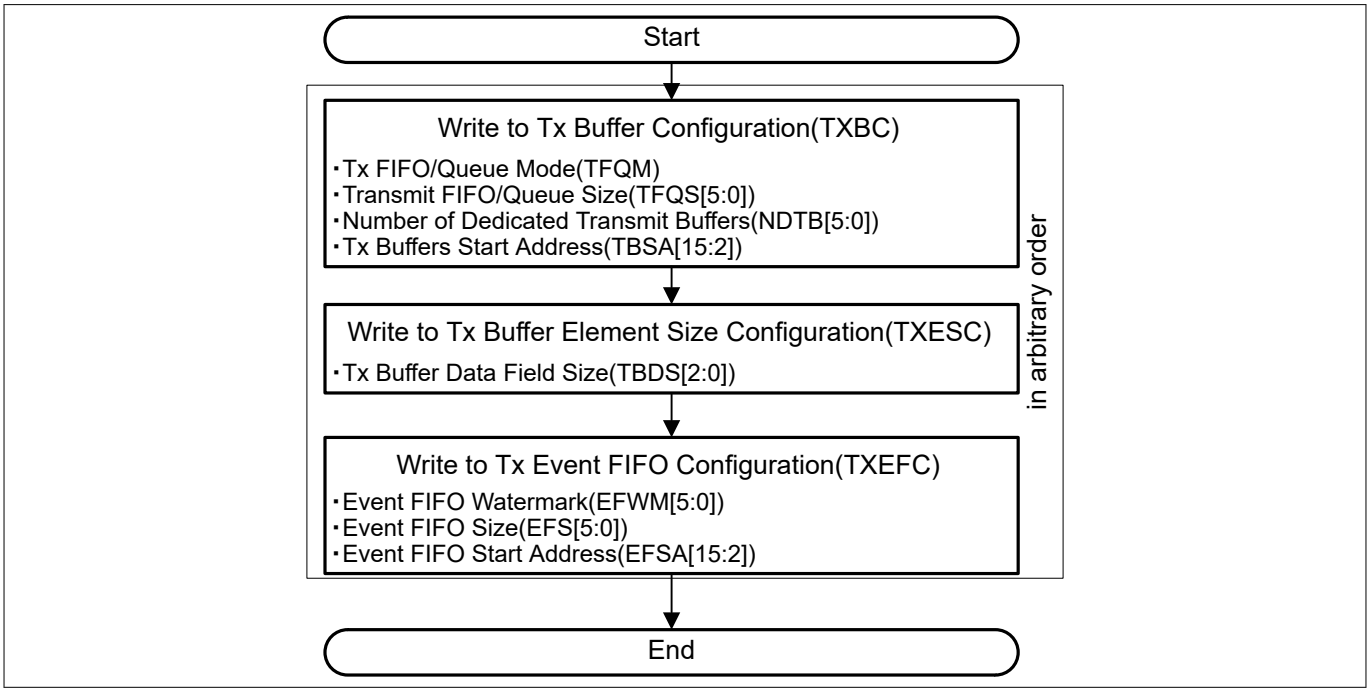


Figure 281 TX Buffer and TX FIFO/Queue configuration

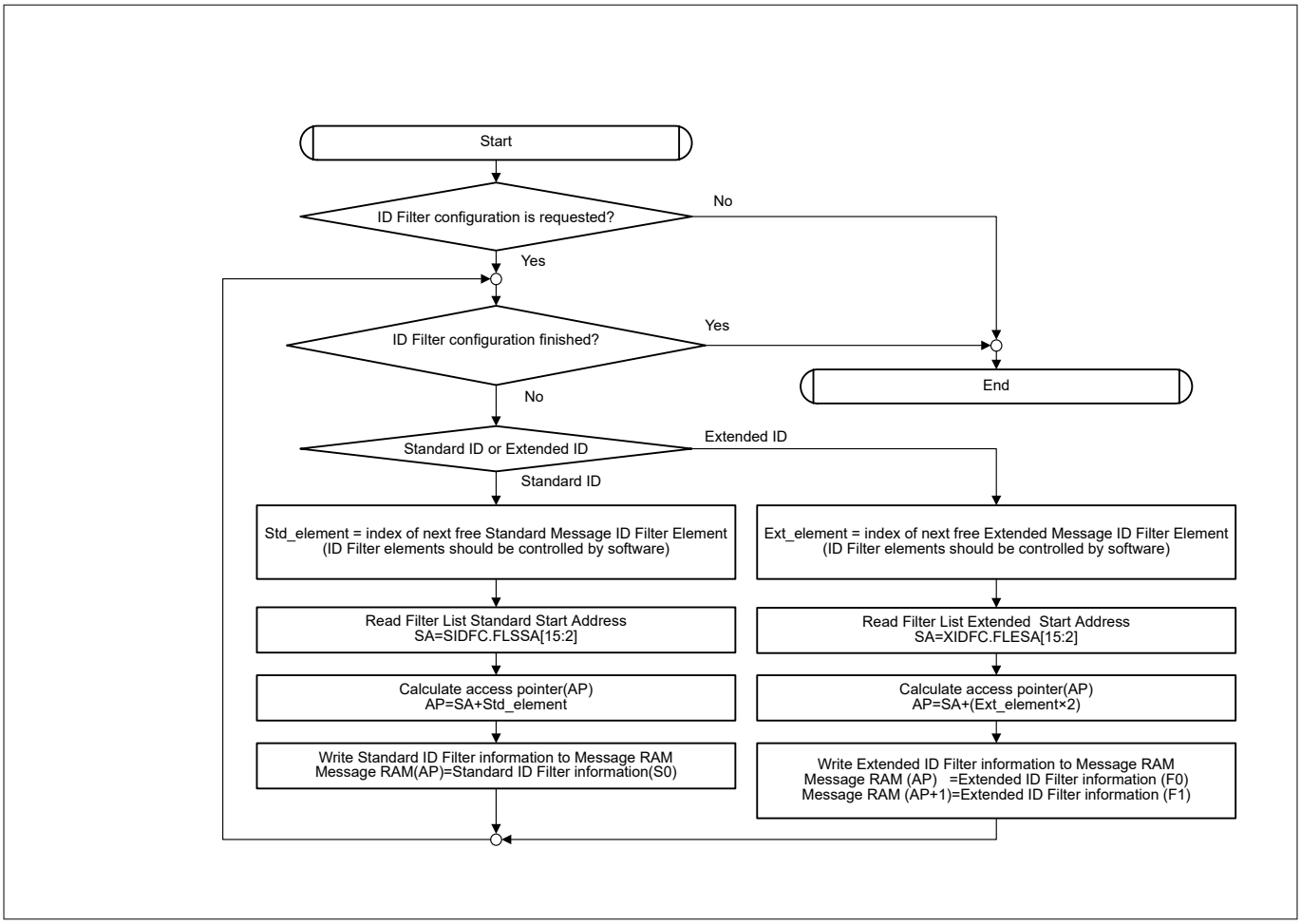


Figure 282 ID filter configuration

26.6.8 Interrupt configuration

Figure 283 shows the general interrupt configuration flow.

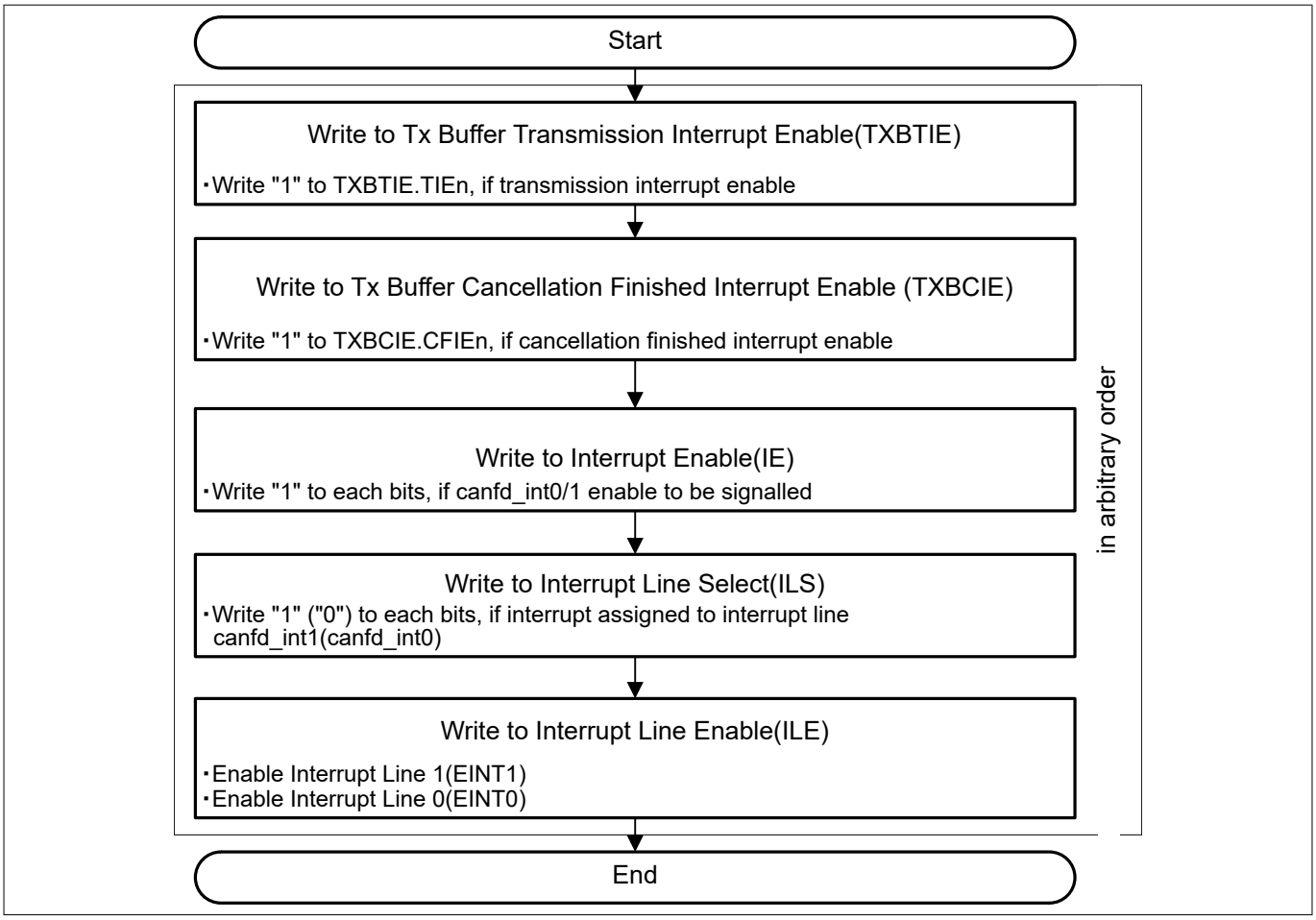


Figure 283 Interrupt configuration

26.6.9 Transmit frame configuration

Figure 284 shows the transmit frame configuration.

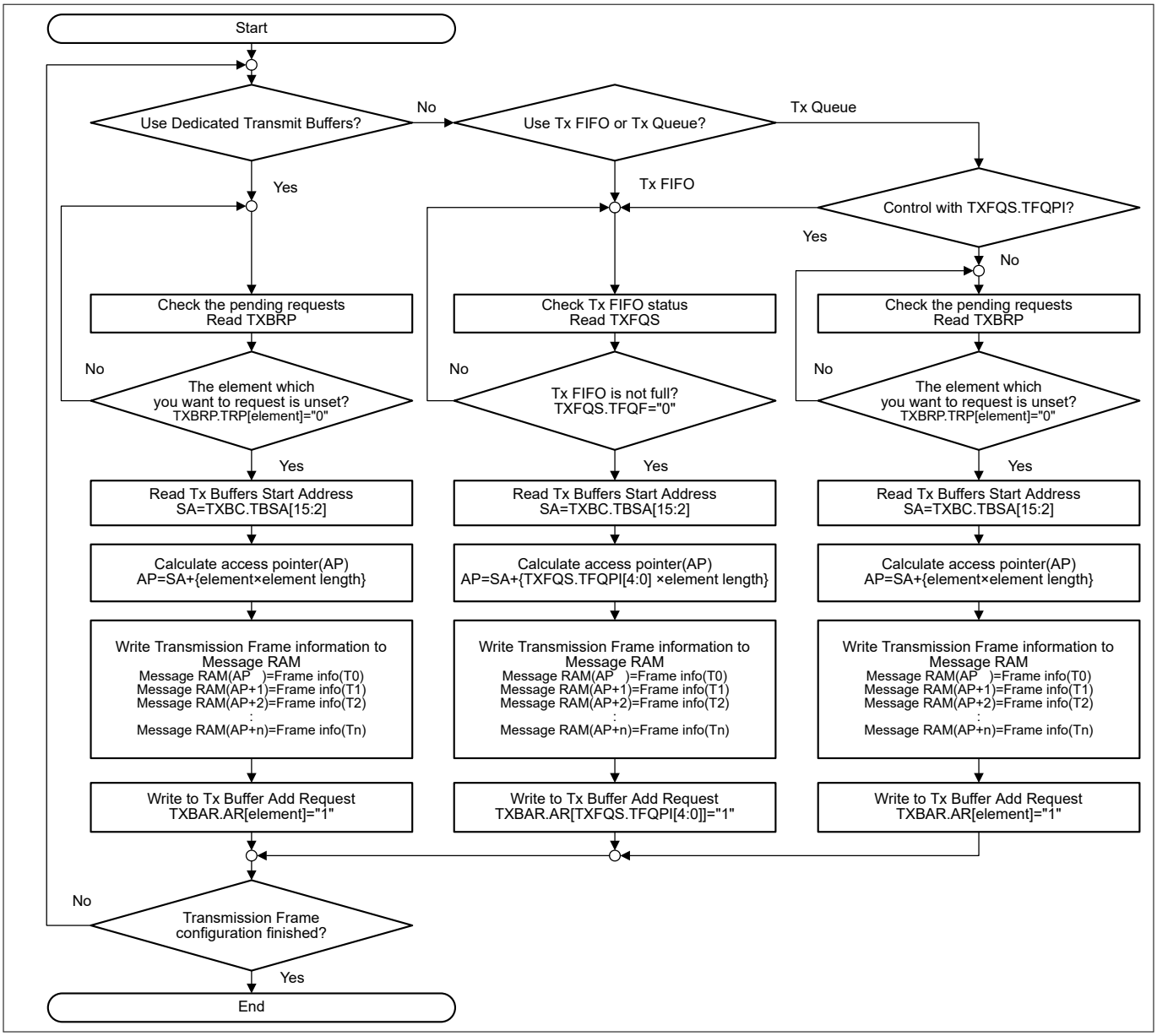


Figure 284 Transmit frame configuration

26.6.10 Interrupt handling

Figure 285 shows a general interrupt handling flow chart.

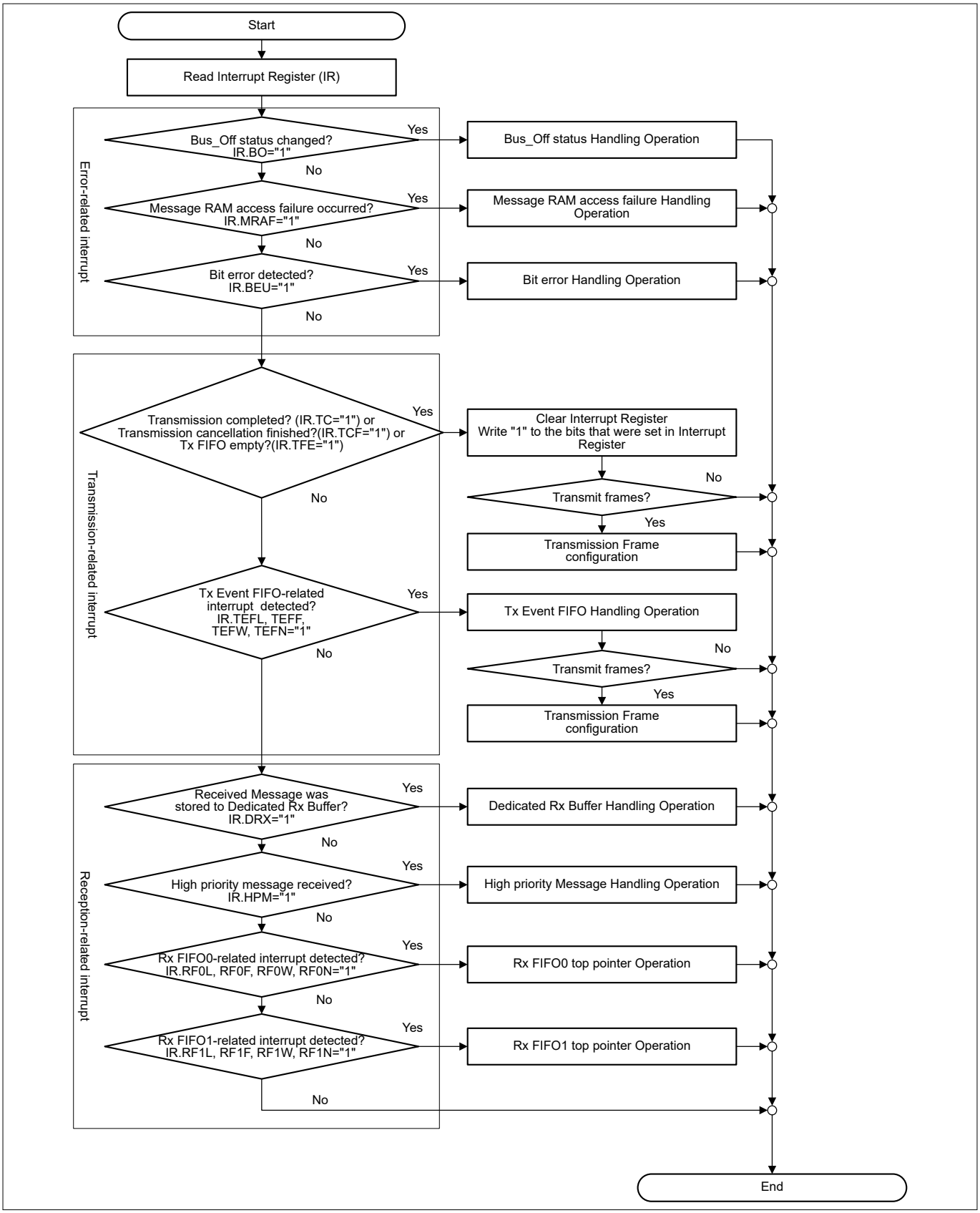


Figure 285 Interrupt handling

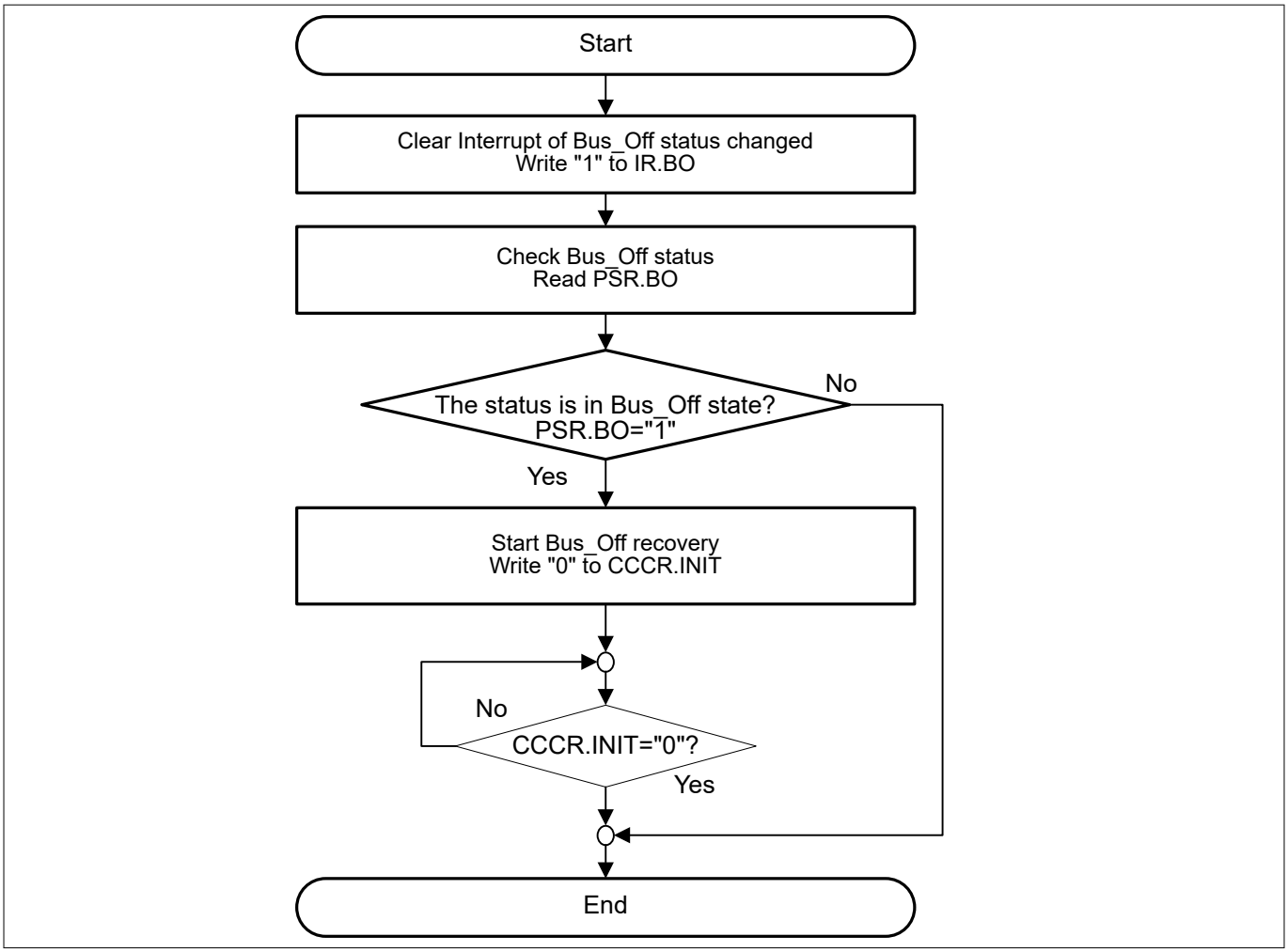


Figure 286 Bus OFF error handling

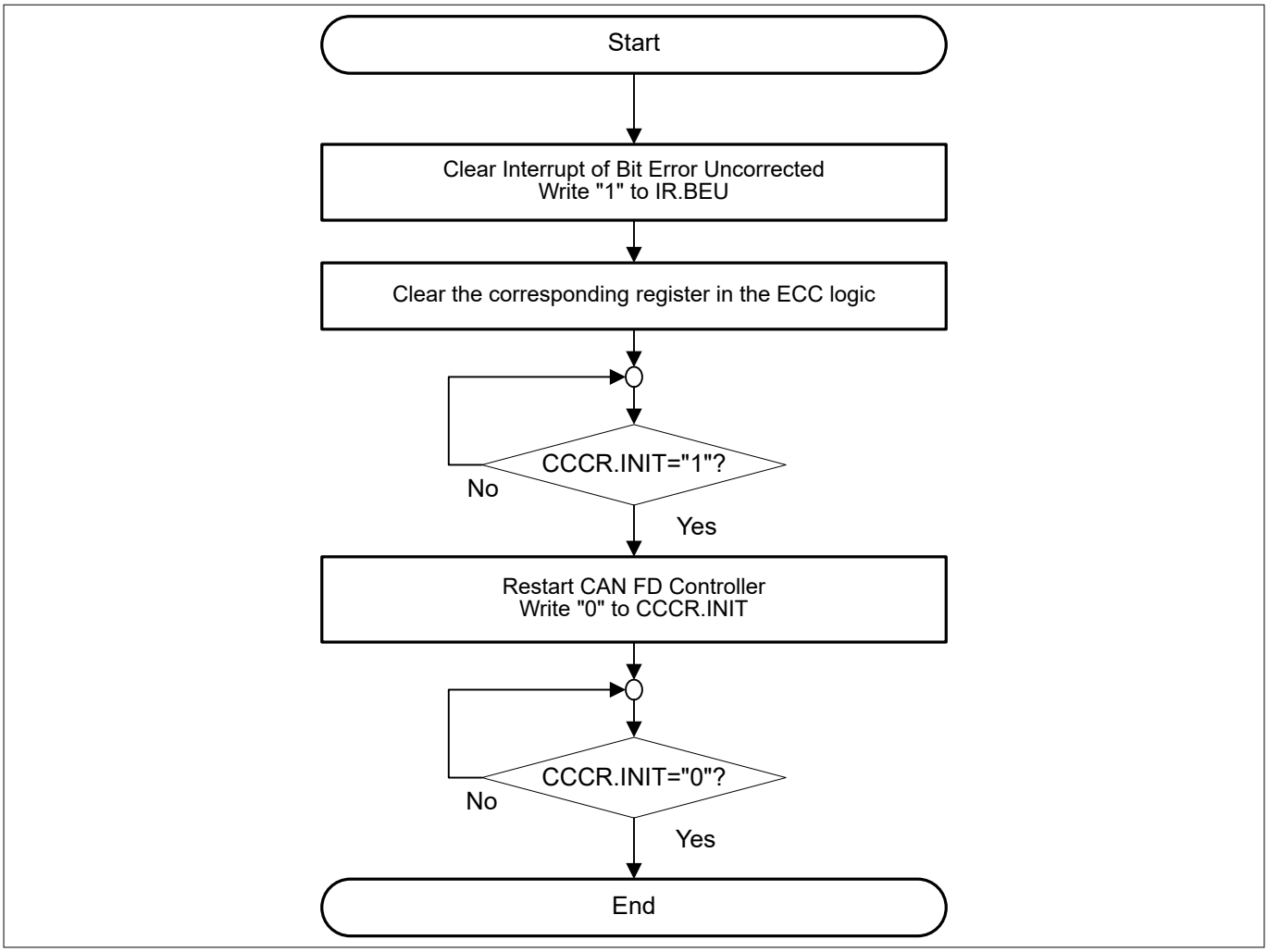


Figure 287 Message RAM access failure handling

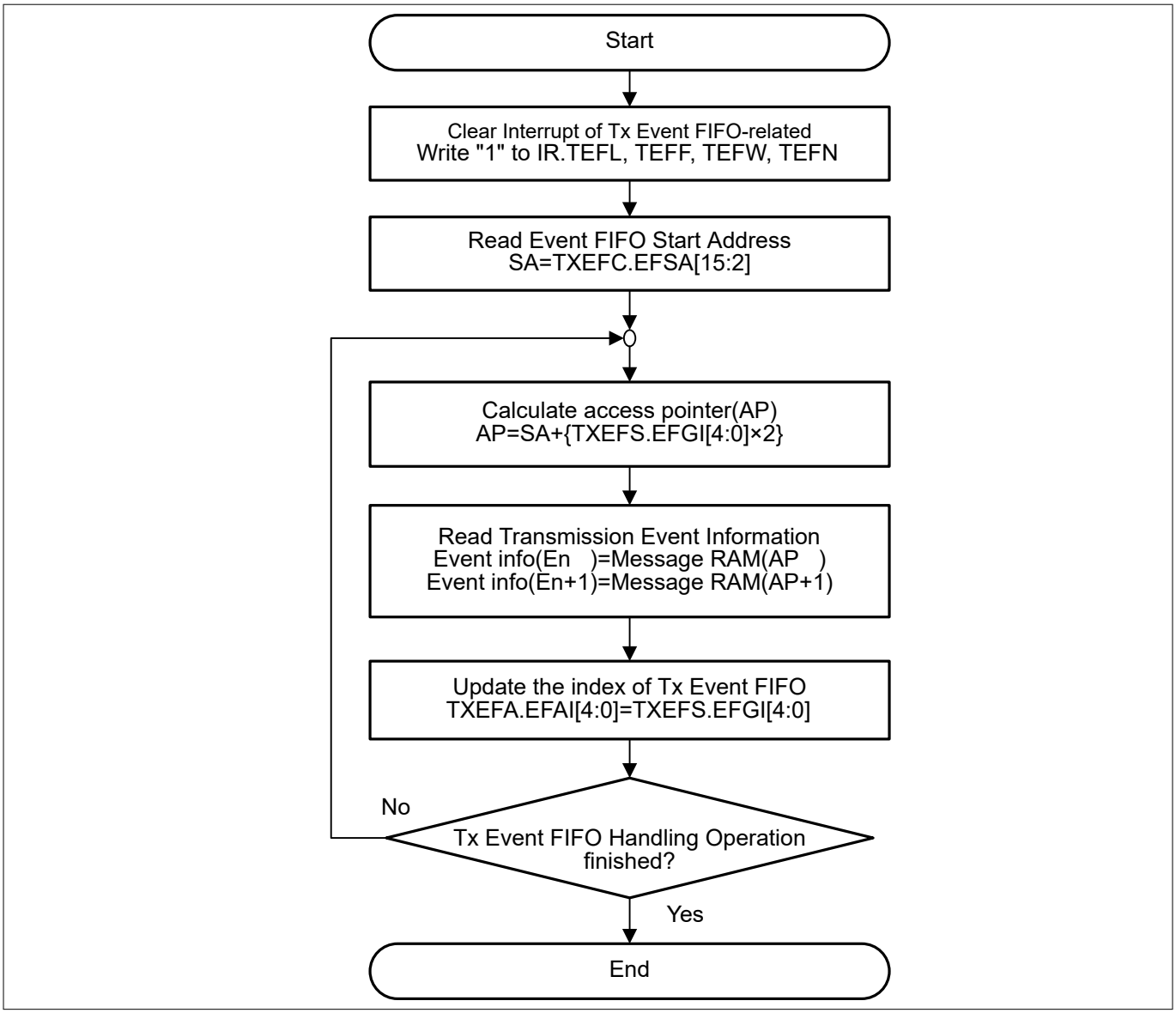


Figure 288 TX Event FIFO handling

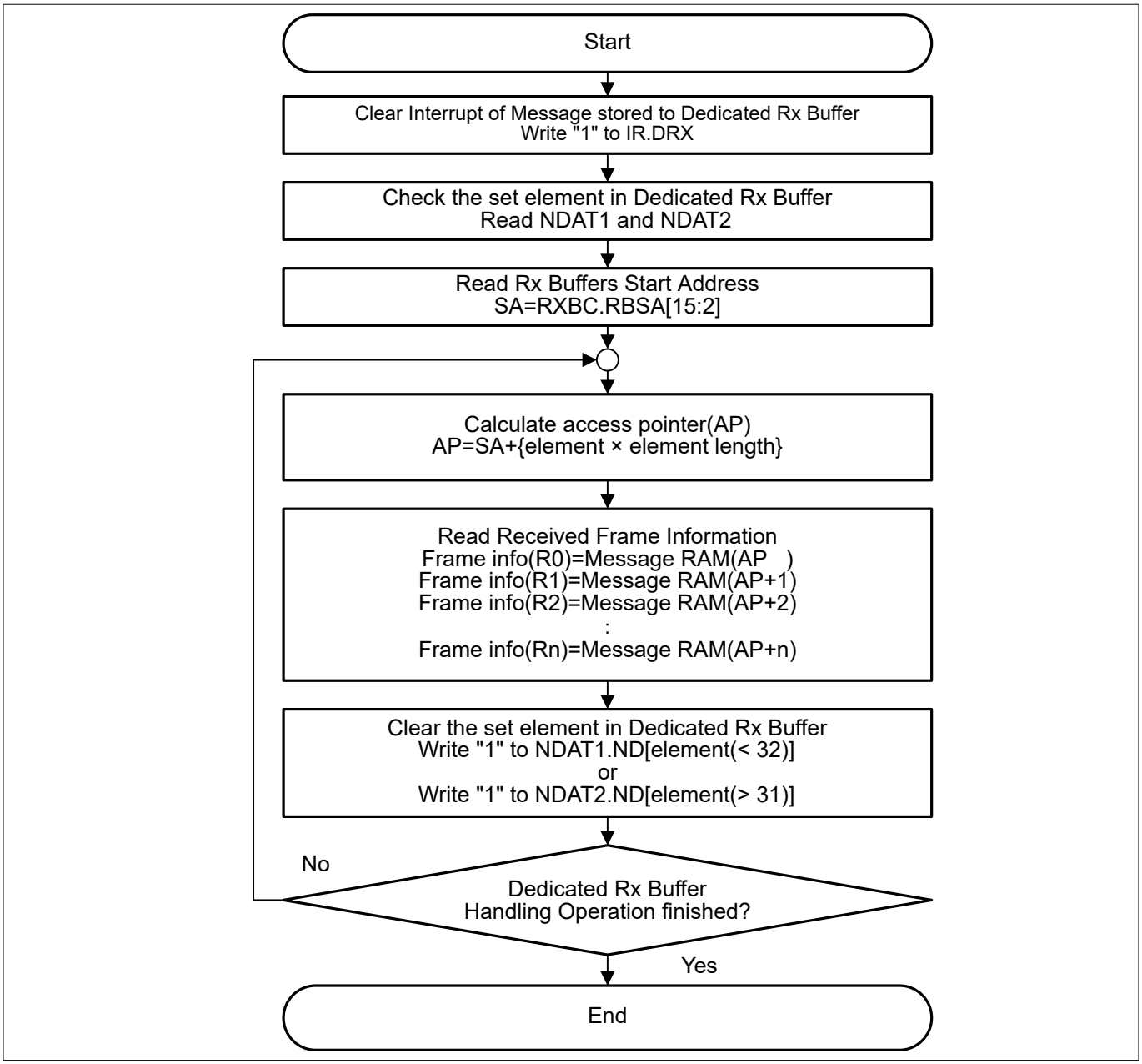


Figure 289 Dedicated RX buffer handling

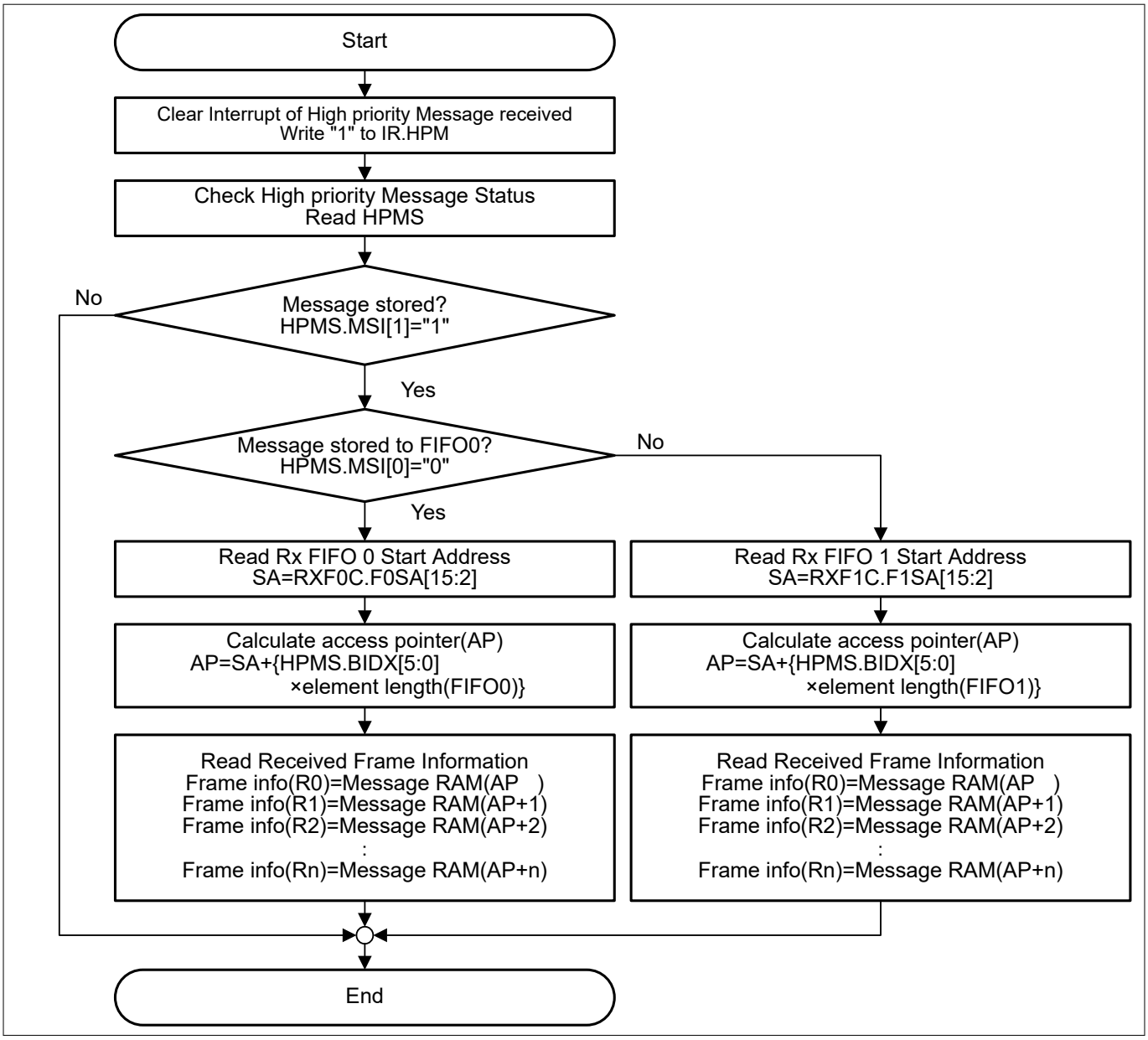


Figure 290 High-priority message handling

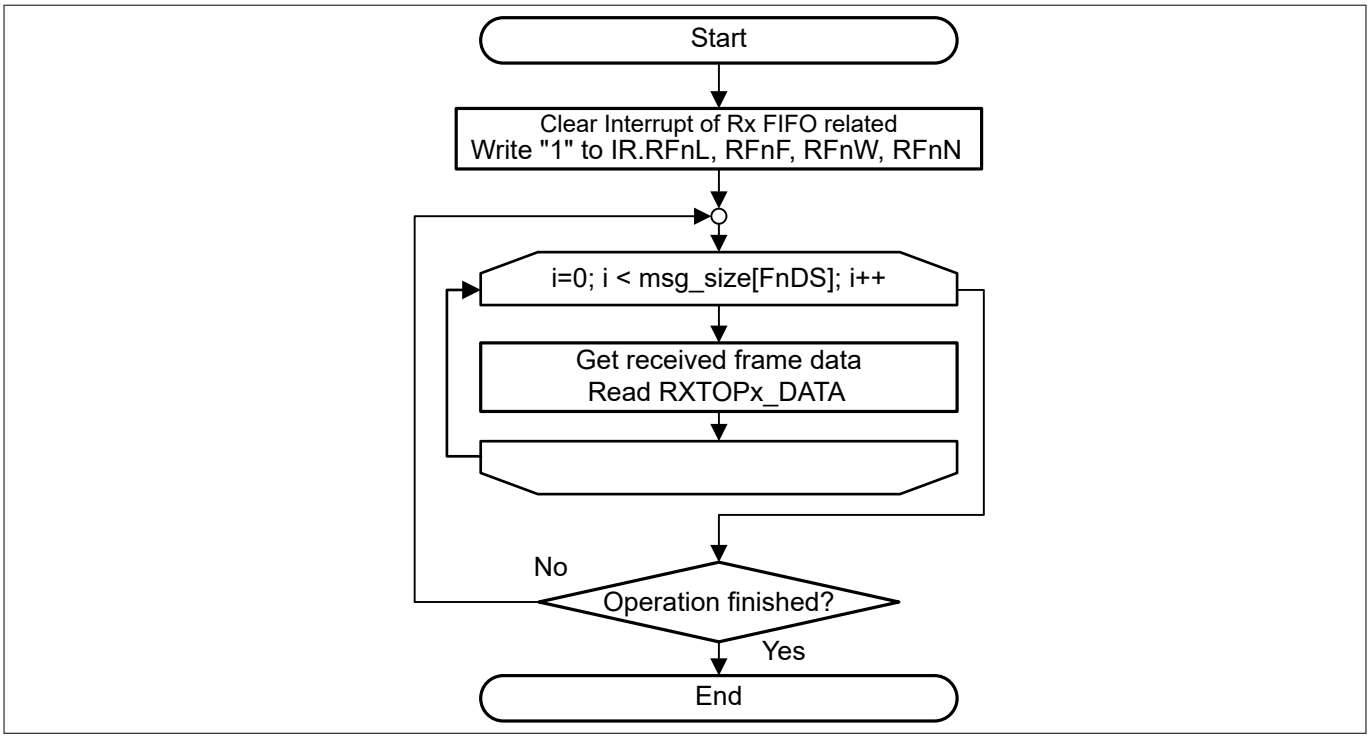


Figure 291 RX FIFO top pointer handling

26.7 Registers summary

For more details on the registers, see the PSC3M/P6 MCU registers RM.

27 Serial Communication Block (SCB)

Each SCB supports the following three serial communication protocols: Serial Peripheral Interface (SPI), universal asynchronous receiver transmitter (UART), and inter-integrated circuit (I2C). An SCB supports only one of the protocols at any given time. PSC3M/P6 MCUs have several SCBs. One of them supports only I2C slave mode and SPI slave mode in the system Deep Sleep power mode.

27.1 Features

The SCB supports the following features:

- Standard SPI master and slave functionality with Motorola, Texas Instruments, and National Semiconductor protocols
- Standard UART transmitter and receiver function with SmartCard interface (ISO7816), IrDA protocol, and local interconnect network (LIN) submodes
- Supports UART half-duplex
- UART flow control (CTS input and RTS output)
- Internal clock input with programmable oversampling
- Standard I2C master and slave functionality
- Each SCB has a 256 byte FIFO for receive (Rx) and transmit (Tx)
- EZ mode for SPI and I2C slaves; allows operation without CPU intervention
- CMD_RESP mode for SPI and I2C slaves; allows operation without CPU intervention and is available only on system Deep Sleep-capable SCB
- System Deep Sleep mode of operation for SPI and I2C slaves (using external clocking), available only on system Deep Sleep-capable SCB
- System Deep Sleep wake-up on I2C slave address match or SPI slave selection, available only on system Deep Sleep-capable SCB
- SPI Master/slave support till 25 MHz, Fast SPI support on one SCB till 50 MHz
- Trigger outputs for connection to DMA
- Multiple interrupt sources to indicate the status of FIFOs and transfers
- Local loop-back control
- Timer added for timeout generation to support SMBUS/PMBUS
- Multiple slave address added to support SMBUS/PMBUS

27.2 Architecture

27.2.1 Trigger interface

The trigger interface provides the status information on the TX FIFO (“tr_tx_req”) and RX FIFO (“tr_rx_req”).

- **DMA trigger signals**: “tr_tx_req” and “tr_rx_req” signals are typically connected to the DW controller
  - “tr_tx_req” indicates that the TX FIFO can accept a data element for transmission
  - “tr_rx_req” indicates that the RX FIFO can provide a (received) data element
- **tr_i2c_scl_filtered signals**: The “tr_i2c_scl_filtered” signal is connected to the SCL input analog filter output. This removes the glitches
- **Serial protocol interfaces**: UART, I2C, and SPI interface signals connect to the High-Speed I/O Matrix (HSIOM) in the IO Sub-System (IOSS)

27.2.2 Clocking modes

An SCB can be clocked either by an internal clock provided by the peripheral clock dividers (clk_scb), or it can be clocked by an external master.

- UART, SPI master, and I2C master modes must use clk_scb
- Only the SPI slave and I2C slave can use the clock from an external master, and only the Deep Sleep capable SCB supports this

Internally- and externally-clocked slave functionality is determined by two register fields of the SCB CTRL register:

- EC_AM_MODE indicates whether SPI slave selection or I2C address matching is internally (‘0’) or externally (‘1’) clocked
- EC_OP_MODE indicates whether the rest of the protocol operation (besides SPI slave selection and I2C address matching) is internally (‘0’) or externally (‘1’) clocked

***Note**: FIFO mode is NOT supported with externally clocked operation (SCBx_CTRL.EC_AM_MODE is ‘0’ or ‘1’); however, data transfer must be done with internal clocking (SCBx_CTRL.EC_OP_MODE is ‘0’).*

- EZ and CMD_RESP modes are supported with externally clocked operation (SCBx_CTRL.EC_OP_MODE is ‘1’)

Before going to Deep Sleep mode, the SCBx_CTRL.EC_ACCESS register should be set to ‘1’. When waking up from Deep Sleep mode and PLL is locked (CLK_SCB is at the expected frequency), SCBx_CTRL.EC_ACCESS should be set to ‘0’.

Table 278 provides an overview of the clocking and buffer modes supported for each communication mode.

Table 278 Clock mode compatibility

	Internally clocked (IC)			Externally clocked (EC) (Deep Sleep SCB only)		
	FIFO	**EZ**	**CMD_RESP**	**FIFO**	**EZ**	**CMD_RESP**
I2C master	Yes	No	No	No	No	No
I2C slave	Yes	Yes	No	No[1)]	Yes	Yes
I2C master-slave	Yes	No	No	No	No	No
SPI master	Yes	No	No	No	No	No
SPI slave	Yes	Yes	No	No[2)]	Yes	Yes
UART transmitter	Yes	No	No	No	No	No
UART receiver	Yes	No	No	No	No	No

1) In Deep Sleep, I2C slave FIFO mode the external-clocked logic can handle slave address matching. It then triggers an interrupt to wake up the CPU.
2) In Deep Sleep, SPI slave FIFO mode the external-clocked logic can handle slave selection detection. It then triggers an interrupt to wake up the CPU. Writes will be ignored and reads will return 0xFF until internal logic takes over. This applies only to the Deep Sleep-capable SCB.

Table 279 I2C slave and SPI clock configuration and mode support

Mode	SCBx_CTRL.EC_AM_MODE is '0'; SCBx_CTRL.EC_OP_MODE is '0'	'SCBx_CTRL.EC_AM_MODE is '1'; SCBx_CTRL.EC_OP_MODE is '0'	'SCBx_CTRL.EC_AM_MODE is '1'; SCBx_CTRL.EC_OP_MODE is '1'
FIFO mode	Yes	Yes	No
EZ mode	Yes	Yes	Yes
CMD_RESP mode	No	No	Yes

27.2.3 Buffer modes

Each SCB has 256 bytes of dedicated RAM for transmit and receive operation. This RAM can be configured in three different modes (FIFO, EZ, or CMD_RESP).

The sections on each protocol provide more details.

- Masters can only use FIFO mode
- I2C and SPI slaves can use all three modes

***Note**: CMD_RESP Mode is available only on the system Deep Sleep-capable SCB.*

- UART only uses FIFO mode

27.2.3.1 FIFO mode

In this mode, the RAM is split into two 128-byte FIFOs, one for transmit (TX) and one for receive (RX). The FIFOs can be configured to be 8 bits × 128 elements or 16 bits × 64 elements; this is done by setting the `SCBx_CTRL.MEM_WIDTH` register.

FIFO operation mode is available only in the CPU Active and system Sleep modes. However, the I2C address or SPI slave select can be used to wake the device from system Deep Sleep on the system Deep Sleep-capable SCB. The status is provided for both the RX and TX FIFOs. There are multiple interrupt sources available, which indicate the status of the FIFOs, such as full or empty; see SCB interrupts.

27.2.3.2 EZ mode

In EZ mode, the RAM is used as a single 256-byte buffer. Both external and internal master sets a base address and reads and writes start from that base address.

EZ Mode is available only for SPI slave and I2C slave.

EZ mode is available in CPU Active and system Deep Sleep power modes.

27.2.3.3 CMD RESP mode

Command Response (CMD_RESP) mode is similar to the EZ mode except that the base address is provided by the CPU not the external master.

CMD_RESP mode is available only for SPI slave and I2C slave. It is available only on the system Deep Sleep-capable SCB.

CMD_RESP mode operation is available in the CPU Active and system Deep Sleep power modes.

27.2.4 Block operation

27.2.4.1 Block enable and recovery

The SCB block (CTRL.ENABLED) can be enabled or disabled by setting the register value to '1' or '0', respectively. To recover the SCB block from unexpected states, it can be reenabled with the `CTRL.ENABLED` register to perform a soft reset of the hardware block and registers. All registers in an ACTIVE power domain are reset when `CTRL.ENABLED` is '0'.

27.2.4.2 Initialization flow after reset

Do the following to initialize the SCB:

1. Program the protocol-specific information using the `SPI_CTRL`, `UART_CTRL` (and `UART_TX_CTRL` and `UART_RX_CTRL`) or `I2C_CTRL` registers. This includes the selection of a submode, master/slave functionality, and transmitter/receiver functionality when applicable
2. Program the generic transmitter (TX_CTRL) and receiver (RX_CTRL) information. This includes enabling the transmitter and receiver functionality
3. Program the transmitter FIFO (TX_FIFO_CTRL) and receiver FIFO (RX_FIFO_CTRL) information
4. Program the CTRL register to enable the SCB. Select the specific operation mode and oversampling factor

27.2.5 Power modes

The SCBs can operate in CPU Active and CPU Sleep power modes. Only SCB[0] can operate in the System Deep Sleep power mode. For more information, see the Clocking system section.

CPU Active to System Deep Sleep power mode transition

Master mode: The master controls the communication, which requests a low-power state or wake-up. In both cases the communication, transition and low-power state are synced.

Slave mode: Transitioning to System Deep Sleep when SCB is configured as Slave and has no control over the external transmitter and in System Deep Sleep, an external Master starts transmitting. It is up to customer software or upper communication protocol to ensure that communication frames are synced with low-power states and the required transition period.

For slave mode, several boundary conditions need to be considered. On an upper level communication protocol, there is usually a command, which requests a low-power state or wake-up. In both cases the communication, transition and low-power state are synced and the limitation is not relevant.

If there is no upper level protocol and power states are asynchronous to the master communication, it is up to the customer code how the expected behavior on both sides can be achieved. For example, for read access from the master, the slave is aware that the data is still in the FIFO. Write access from the master, during transition between power states might be lost.

***Notes**:*

1. *During the CPU Active to System Deep Sleep transition in an externally-clocked mode, any transactions that are sent to the slave during the transition period are lost*
2. *During the CPU Active to System Deep Sleep transition in an internally-clocked mode, there is no automatic clock source switching and no transfers are available in System Deep Sleep mode*

27.3 Serial peripheral interface (SPI)

The SPI protocol is a synchronous serial interface protocol. Devices operate in either master or slave mode. Only a master can initiate the data transfer. The SCB supports the single-master-multiple-slaves topology for SPI. Multiple slaves up to four are supported with individual slave select lines.

27.3.1 Features

- Supports master and slave functionality
- Supports three types of SPI protocols:
  - Motorola SPI – modes 0, 1, 2, and 3
  - Texas Instruments SPI, with coinciding and preceding data frame indicator – mode 1 only
  - National Semiconductor (MicroWire) SPI – mode 0 only
- Master supports up to three slave select lines
  - Each slave select has configurable active polarity (high or low)
  - Slave select can be programmed to stay active for a whole transfer, or just for each byte
- Master supports late sampling for better timing margin
- Master supports continuous SPI clock
- Data frame size programmable from 4 bits to 32 bits
- Variable SELECT output signal timing (SPI master):
  - SELECT setup time (select active to SPI clock), SELECT hold time (SPI clock to select inactive) (for Motorola mode only)
  - Inter-data frame deselect time (select inactive to select active) (for Motorola mode only)
- Parity support (odd and even parity) (for Motorola mode only)
- Interrupts or polling CPU interface
- Programmable oversampling
- MSB or LSB first
- Median filter available for inputs
- Supports FIFO mode, EZ mode (slave only), and CMD_RESP mode (slave only)
- Wake-up interrupt cause activated on slave selection (SCB[0] only)
- Local loop-back control
- Fast SPI operation up to 50 MHz (SCB[5] only)
- Up to 16 supported lines of SPI SSEL

27.3.2 General description

Figure 292 illustrates an example of a SPI master with three slaves.

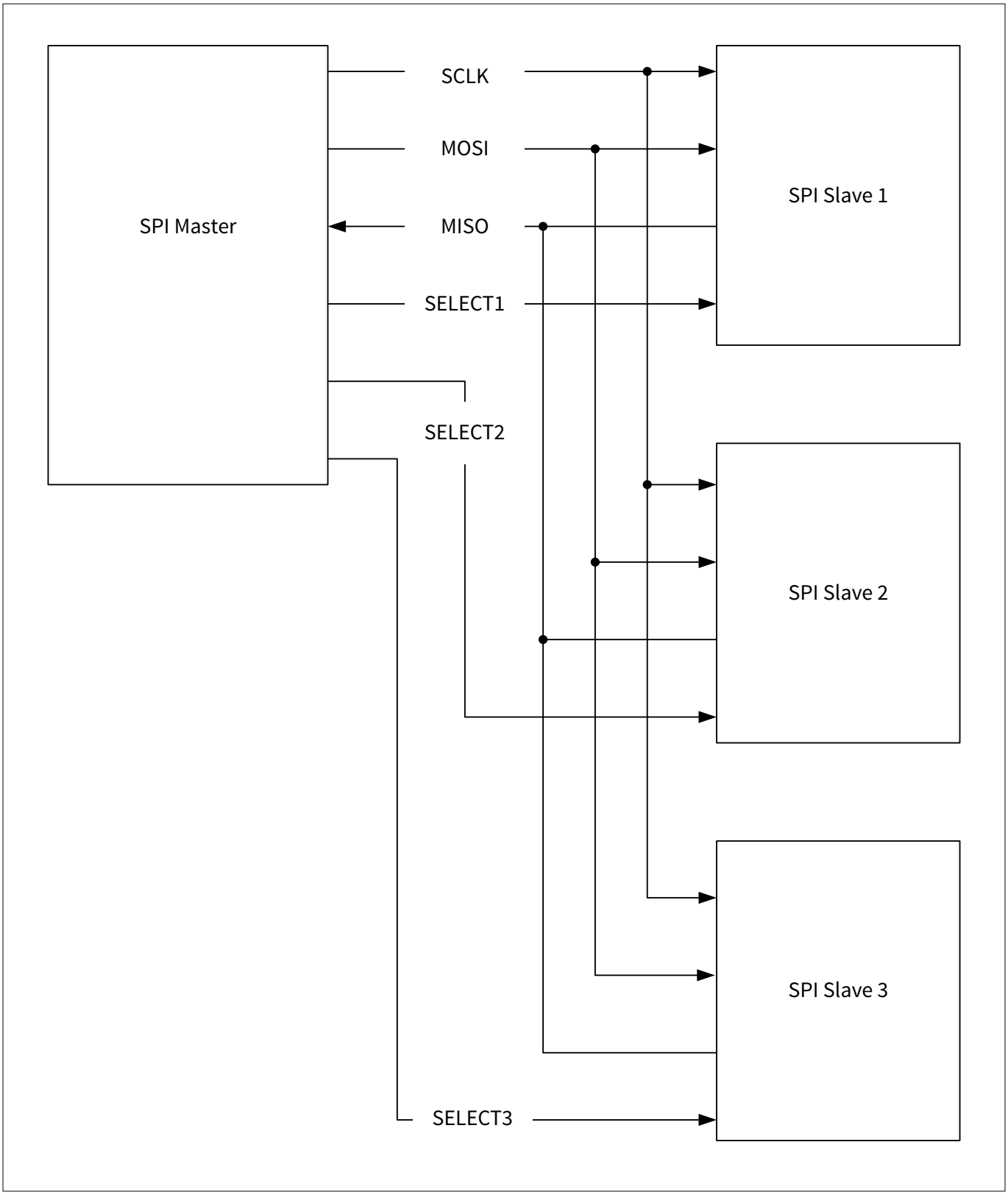


Figure 292 SPI example

A standard SPI interface consists of the following four signals:

- SCLK: Serial clock (clock output from the master, input to the slave)
- MOSI: Master-out-slave-in (data output from the master, input to the slave)

- MISO: Master-in-slave-out (data input to the master, output from the slave)
- SELECT: Typically an active low signal (output from the master, input to the slave)

A simple SPI data transfer involves the following:

- The master selects a slave by driving the SELECT line. Then, it drives the data on the MOSI line and a clock on the SCLK line. The slave uses either of the edges of SCLK depending on the configuration to capture the data on the MOSI line. It also drives the data on the MISO line, which is captured by the master

The data frame size can be configured to any value in the range of 4 bits to 32 bits. The serial data can be transmitted either most significant bit (MSb) first or the least significant bit (LSb) first.

Three different variants of the SPI protocol are supported by the SCB:

- Motorola SPI: This is the original SPI protocol
- Texas Instruments SPI: A variation of the original SPI protocol, in which data frames are identified by a pulse on the SELECT line
- National Semiconductors SPI: A half-duplex variation of the original SPI protocol

Notes about duplex control:

- Motorola and Texas Instruments modes are full-duplex; National Semiconductors mode is half-duplex
- Full-duplex modes also work similar to half-duplex, controlled by SCBx_TX_FIFO_CTRL.FREEZE or SCBx_RX_FIFO_CTRL.FREEZE, to transmit dummy data words or ignore received data words
- The MOSI can be set to Hi-Z state using an IOSS/GPIO configuration

27.3.3 SPI modes of operation

27.3.3.1 Motorola SPI

The original SPI protocol was defined by Motorola. It is a full-duplex protocol. Multiple data transfers may happen with the SELECT line held at '0'. As a result, slave devices must keep track of the progress of data transfers to separate individual data frames. When not transmitting data, the SELECT line is held at '1' and SCLK is typically pulled low.

Clock modes of Motorola SPI

The Motorola SPI protocol has four different clock modes based on how data is driven and captured on the MOSI and MISO lines. These modes are determined by clock polarity (SCBx_SPI_CTRL.CPOL) and clock phase (SCBx_SPI_CTRL.CPHA).

Clock polarity determines the value of the SCLK line when not transmitting data. SCBx_SPI_CTRL.CPOL = 0 indicates that SCLK is '0' when not transmitting data. SCBx_SPI_CTRL.CPOL = 1 indicates that SCLK is '1' when not transmitting data.

The clock phase determines when data is driven and captured. SCBx_SPI_CTRL.CPHA = 0 means sample (capture data) on the leading (first) clock edge, while SCBx_SPI_CTRL.CPHA = 1 means sample on the trailing (second) clock edge, regardless of whether that clock edge is rising or falling. With SCBx_SPI_CTRL.CPHA = 0, the data must be stable for setup time before the first clock cycle.

- Mode 0: SCBx_SPI_CTRL.CPOL is '0', SCBx_SPI_CTRL.CPHA is '0'; Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK. SCLK idle state is '0'
- Mode 1; SCBx_SPI_CTRL.CPOL is '0', SCBx_SPI_CTRL.CPHA is '1': Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK. SCLK idle state is '0'
- Mode 2: SCBx_SPI_CTRL.CPOL is '1', SCBx_SPI_CTRL.CPHA is '0': Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK. SCLK idle state is '1'
- Mode 3: SCBx_SPI_CTRL.CPOL is '1', SCBx_SPI_CTRL.CPHA is '1': Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK. SCLK idle state is '1'

Figure 293 illustrates the driving and capturing of the MOSI/MISO data as a function of SCBx_SPI_CTRL.CPOL and SCBx_SPI_CTRL.CPHA.

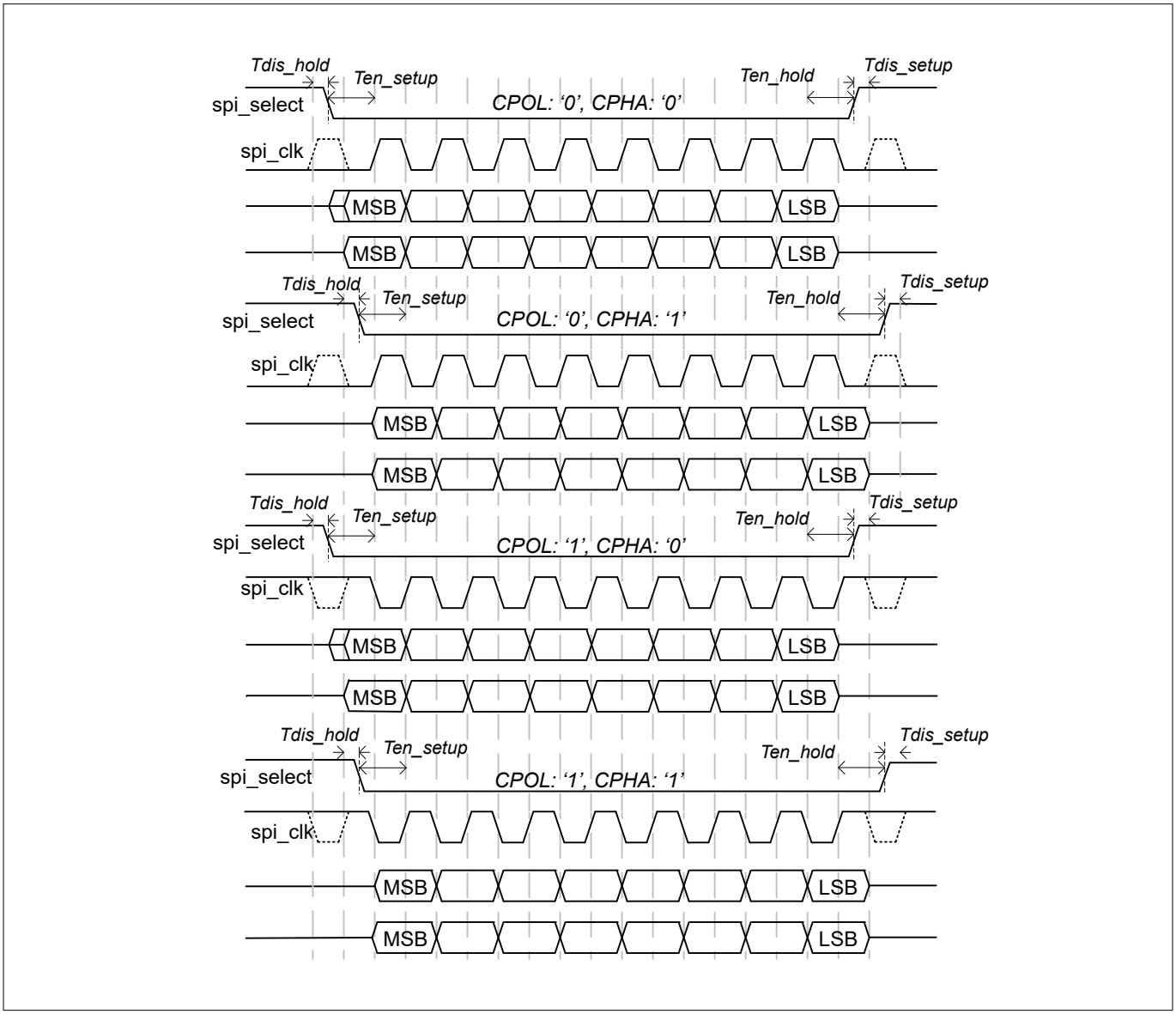


Figure 293 SPI Motorola, four modes

Figure 294 shows a single 8-bit and two successive 8-bit data transfers in mode 0 (SCBx_SPI_CTRL.CPOL is '0', SCBx_SPI_CTRL.CPHA is '0').

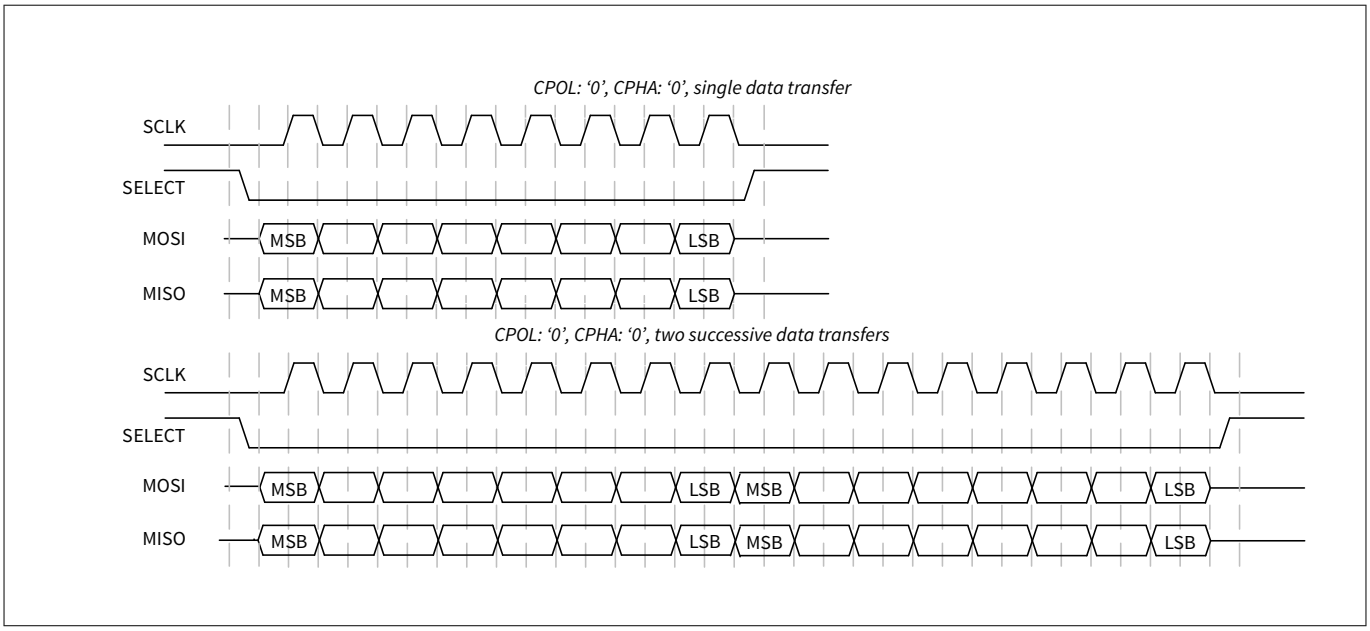


Figure 294 SPI Motorola data transfer example

Configuring SCB for SPI Motorola mode

To configure the SCB for SPI Motorola mode, set various register bits in the following order:

1. Select SPI by writing '01' to the `SCBx_CTRL.MODE` register
2. Select SPI Motorola mode by writing '00' to the `SCBx_SPI_CTRL.MODE` register
3. Select the mode of operation in Motorola by writing to the `SCBx_SPI_CTRL.CPHA` and `SCBx_SPI_CTRL.CPOL` register
4. Follow steps 2 to 4 mentioned in the Enabling and initializing SPI section

For more information on these registers, see the PSC3M/P6 MCU registers reference manual.

27.3.3.2 Texas Instruments SPI

The Texas Instruments' SPI protocol redefines the use of the SELECT signal. It uses the signal to indicate the start of a data transfer, rather than a low active slave select signal, as in the Motorola SPI. As a result, the slave devices do not have to track the progress of data transfers to separate individual data frames. The start of a transfer is indicated by a high active pulse of a single-bit transfer period. This pulse may occur one cycle before the transmission of the first data bit, or may coincide with the transmission of the first data bit. The TI SPI protocol supports only mode 1 (`SCBx_SPI_CTRL.CPOL` is '0' and `SCBx_SPI_CTRL.CPHA` is '1'): data is driven on a rising edge of SCLK and captured on a falling edge of SCLK.

Figure 295 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SELECT pulse precedes the first data bit. Note that how the SELECT pulse of the second data transfer coincides with the last data bit of the first data transfer.

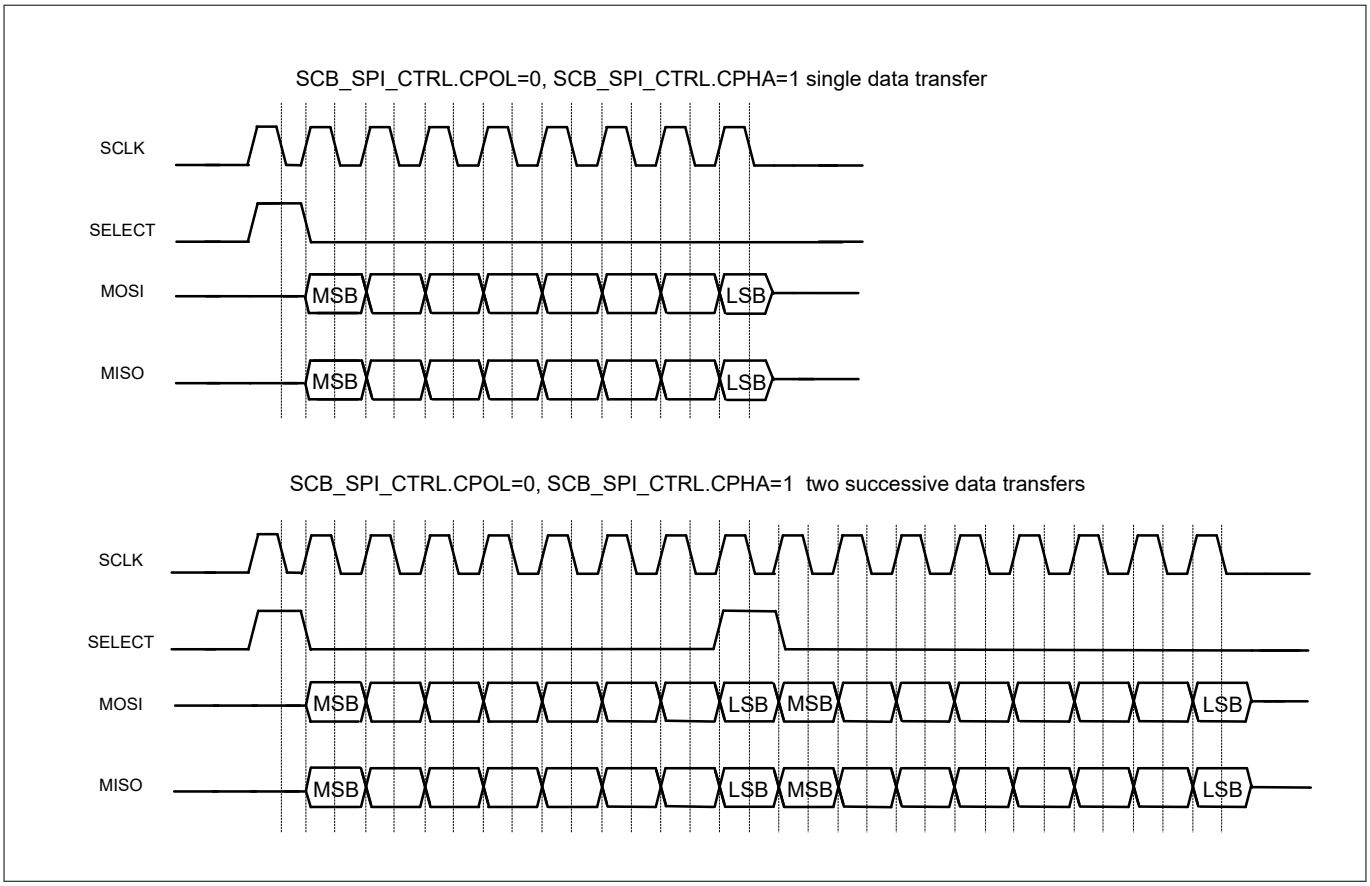


Figure 295 SPI TI data transfer example

Figure 296 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SELECT pulse coincides with the first data bit of a frame.

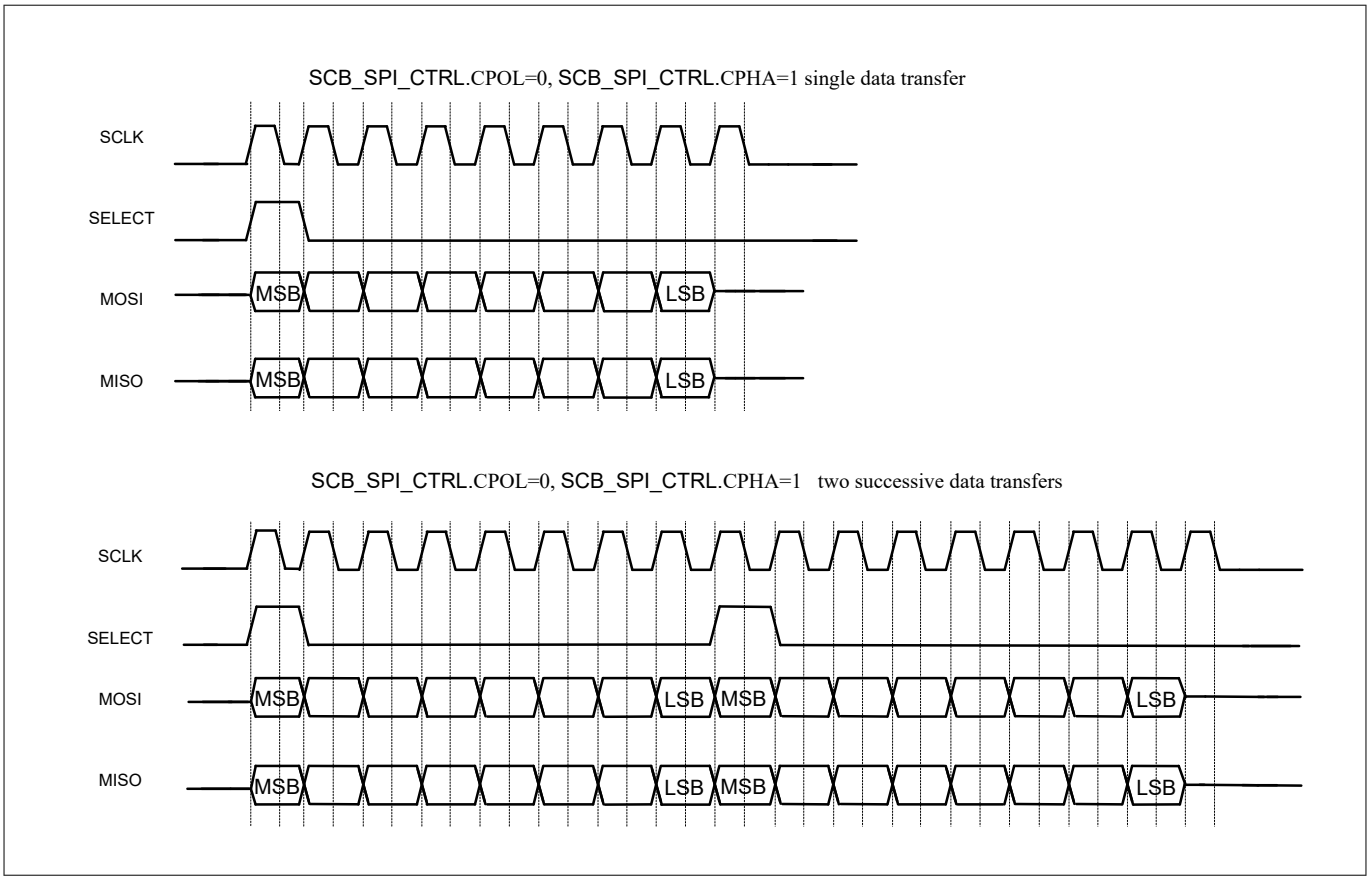


Figure 296 SPI TI data transfer example 2

Configuring SCB for SPI TI mode

To configure SCB for the SPI TI mode, set various register bits in the following order:

1. Select SPI by writing '01' to the `SCBx_CTRL.MODE` register
2. Select SPI TI mode by writing '01' to the `SCBx_SPI_CTRL.MODE` register
3. Select the mode of operation in TI by writing to the `SCBx_SPI_CTRL.SELECT_PRECEDE` register ('1' configures the SELECT pulse to precede the first bit of the next frame and '0' otherwise)
4. Follow steps 2 to 4 mentioned in the Enabling and initializing SPI section

For more information on these registers, see the PSC3M/P6 MCU registers reference manual.

27.3.3.3 National semiconductors SPI

The National Semiconductors' SPI protocol is a half-duplex protocol. In this protocol, transmission and reception do not occur at the same time, but take turns and transmission happens before reception. The transmission and reception data sizes may differ. A single idle (= '0') bit transfer period separates transmission from reception. However, the successive data transfers are not separated by an idle bit transfer period. The National Semiconductors SPI protocol only supports mode 0.

Figure 297 illustrates a single data transfer and two successive data transfers. In both cases, the transfer size of the transmission data is 8 bits and the transfer size of the reception data is 4 bits.

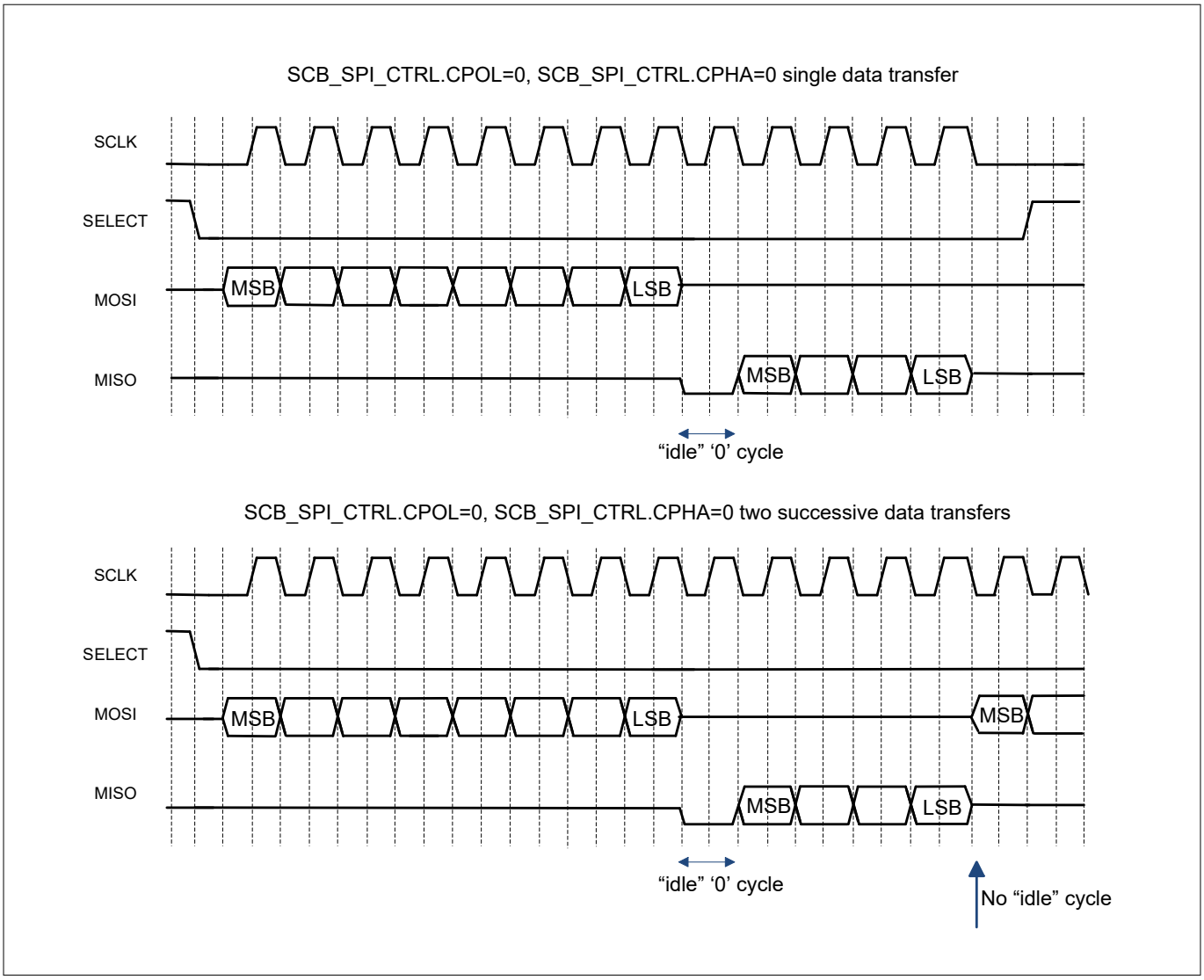


Figure 297 SPI NS data transfer example

Configuring SCB for SPI NS mode

To configure SCB for the SPI NS mode, set various register bits in the following order:

1. Select SPI by writing '01' to the `SCBx_CTRL.MODE` register
2. Select SPI NS mode by writing '10' to the `SCBx_SPI_CTRL.MODE` register
3. Set the `SCBx_SPI_CTRL.CPOL` and `SCBx_SPI_CTRL.CPHA` bit to '0'
4. Follow steps 2 to 4 mentioned in the Enabling and initializing SPI section

For more information on these registers, see the PSC3M/P6 MCU registers reference manual.

27.3.4 SPI buffer modes

SPI can operate in the following three different buffer modes:

- FIFO mode
- EZ mode
- CMD_RESP mode

The buffer is used in different ways in each of these modes. The following subsections explain each of these buffer modes in detail.

27.3.4.1 FIFO mode

The FIFO mode has a TX FIFO for the data being transmitted and an RX FIFO for the data received. Each FIFO is constructed out of the SRAM buffer. The FIFOs are either 32 elements deep with 32-bit data elements or 64 elements deep with 16-bit data elements or 128 elements deep with 8-bit data elements. The width of a FIFO is configured using the SCBx_CTRL.MEM_WIDTH register.

The FIFO mode is available only in CPU Active and Sleep power modes, and not in the system Deep Sleep mode.

Transmit and receive FIFOs allow write and read access. A write access to the transmit FIFO uses the SCBx_TX_FIFO_WR register. A read access from the receive FIFO uses the SCBx_RX_FIFO_RD register.

Transmit and receive FIFO status information is available through the status registers, SCBx_TX_FIFO_STATUS and SCBx_RX_FIFO_STATUS. It is possible to define a programmable threshold that indicates a number of FIFO entries, a trigger/event is generated when the following conditions are met:

- The transmit FIFO has a SCBx_TX_FIFO_CTRL.TRIGGER_LEVEL. A trigger/event is generated when the number of entries in the transmit FIFO is less than SCBx_TX_FIFO_CTRL.TRIGGER_LEVEL
- The receive FIFO has an SCBx_RX_FIFO_CTRL.TRIGGER_LEVEL. A trigger/event is generated when the number of receive FIFO entries is greater than the SCBx_RX_FIFO_CTRL.TRIGGER_LEVEL

These triggers can be connected to a DW channel.

Furthermore, numerous interrupt status bits are provided for both the RX and TX FIFOs. These can be found looking at SCBx_INTR_TX and SCBx_INTR_RX.

System Deep Sleep to CPU Active transition

SCBx_CTRL.EC_AM_MODE = 1, SCBx_CTRL.EC_OP_MODE = 0, FIFO Mode.

MISO transmits 0xFF until the internally clocked logic takes over and the CPU writes to TX FIFO. Data on MOSI is ignored until the internally clocked logic takes over. When the internally clocked logic takes over, there is no guarantee that the internal clock will be at the correct frequency due to PLL/FLL locking times. This may lead to corrupted data in the RX FIFO. Therefore, it is recommended to clear the RX FIFO before writing new data into the TX FIFO after the transition from system Deep Sleep to CPU Active. Another option is to disable CLK_SCB before going to system Deep Sleep, and then wait to enable it until the PLL and FLL have stabilized. The external master needs to be aware that when it reads 0xFF on MISO, the device is not ready yet.

27.3.4.2 EZSPI mode

The EZSPI protocol is based on the Motorola SPI operating in any mode (0, 1, 2, or 3). It allows communication between master and slave without the need for CPU intervention. In PSC3M/P6 MCU, only one SCB block supports EZSPI mode: the Deep Sleep-capable SCB.

The EZSPI protocol defines a single memory buffer with an 8-bit EZ address that indexes the buffer (256-entry array of 8 bits per entry) located on the slave device. The EZ address is used to address these 256 locations. All EZSPI data transfers have 8-bit data frames. The CPU writes and reads to the memory buffer through the SCBx_EZ_DATA registers. These accesses are word accesses, but only the least significant byte of the word is used.

EZSPI has the following three types of transfers:

- A write of the EZ address from the master to the slave
- A write of data from the master to an addressed slave memory location
- A read by the master from an addressed slave memory location

***Note**: When multiple bytes are read or written, the master must keep SELECT low during the entire transfer.*

EZ Address Write

A write of the EZ address starts with a command byte (0x00) on the MOSI line indicating the master's intent to write the EZ address. During the reception of the command byte, the slave transmits (on MISO) either 0xfe (ready) or 0xff (busy). When disabled or reset, the slave transmits (on MISO) 0xff (busy). The byte value can be used by the master to determine whether the slave is ready to accept the SPI request. The slave then drives a reply byte on the MISO line to indicate whether the command is acknowledged (0xFE) or not (0xFF). The second byte on the MOSI line is the EZ address.

Memory Array Write

A write to a memory array index starts with a command byte (0x01) on the MOSI line indicating the master's intent to write to the memory array. During the reception of the command byte, the slave transmits (on MISO) either 0xfe (ready) or 0xff (busy). When disabled or reset, the slave transmits (on MISO) 0xff (busy). The byte value can be used by the master to determine whether the slave is ready to accept the SPI request. The slave then drives a reply byte on the MISO line to indicate that the command was registered (0xFE) or not (0xFF). Any additional write data bytes on the MOSI line are written to the memory array at locations indicated by the communicated EZ address. The EZ address is automatically incremented by the slave as bytes are written into the memory array. When the EZ address exceeds the maximum number of memory entries (256), it remains there and does not wrap around to 0. The EZ base address is reset to the address written in the EZ Address Write phase on each slave selection.

Memory Array Read

A read from a memory array index starts with a command byte (0x02) on the MOSI line indicating the master's intent to read from the memory array. During the reception of the command byte, the slave transmits (on MISO) either 0xfe (ready) or 0xff (busy). When disabled or reset, the slave transmits (on MISO) 0xff (busy). The byte value can be used by the master to determine whether the slave is ready to accept the SPI request. The slave then drives a reply byte on the MISO line to indicate whether the command was registered (0xFE) or not (0xFF). Any additional read data bytes on the MISO line are read from the memory array at locations indicated by the communicated EZ address. The EZ address is automatically incremented by the slave as bytes are read from the memory array. When the EZ address exceeds the maximum number of memory entries (256), it remains there and does not wrap around to 0. The EZ base address is reset to the address written in the EZ Address Write phase on each slave selection.

Figure 298 illustrates the write of the EZ address, write to a memory array, and read from a memory array operation in the EZSPI protocol.

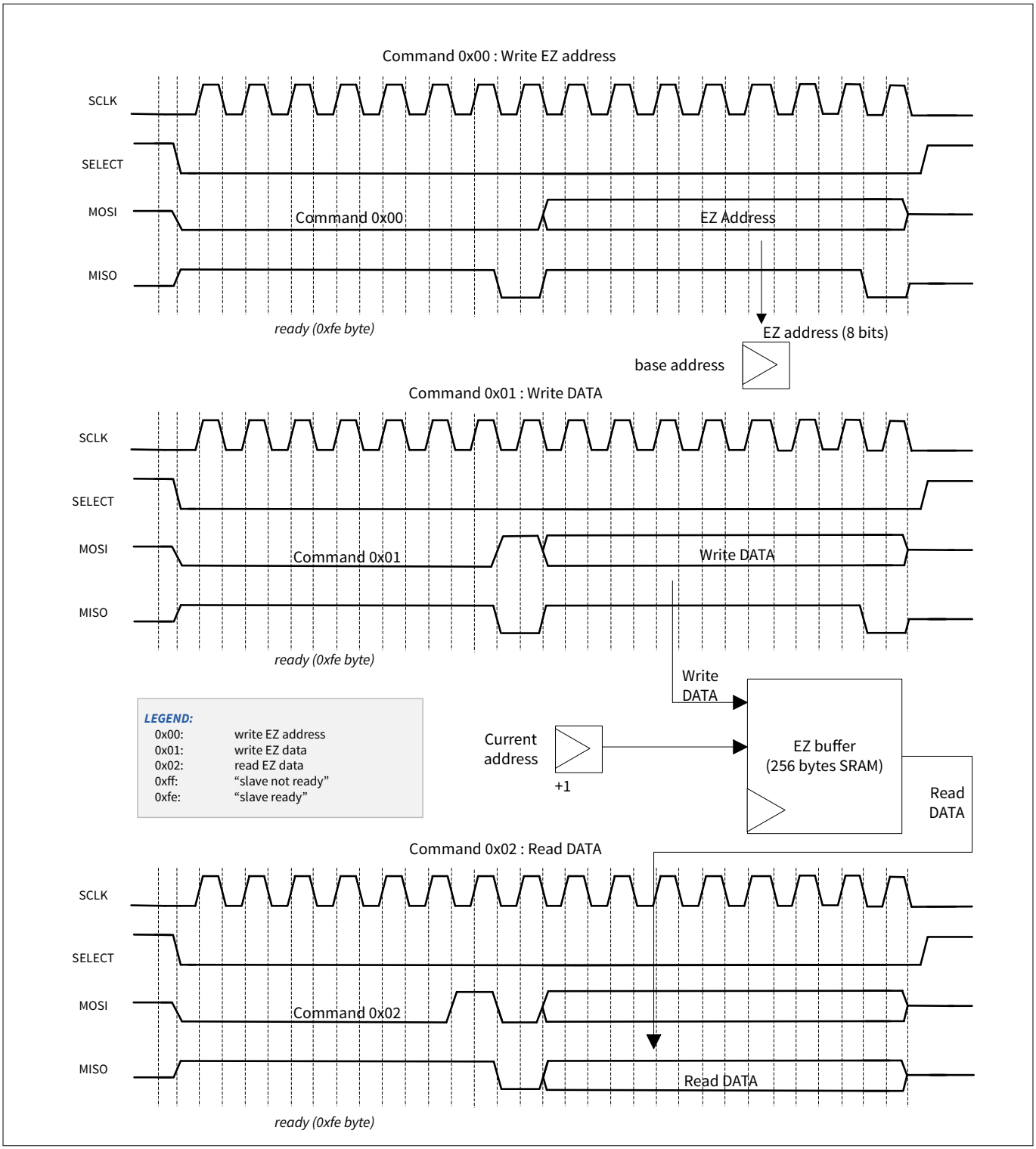


Figure 298 EZSPI example

Configuring SCB for EZSPI mode

By default, the SCB is configured for a non-EZ mode of operation. To configure the SCB for EZSPI mode, set the register bits in the following order:

1. Select EZ mode by writing '1' to the `SCBx_CTRL.EZ_MODE` register
2. Follow the steps in the Motorola SPI section
3. Follow steps 2 to 4 mentioned in the Enabling and initializing SPI section

For more information on these registers, see the PSC3M/P6 MCU registers reference manual.

System Deep Sleep to CPU Active transition

- EC_AM_MODE = 1, SCBx_CTRL.EC_OP_MODE = 0, EZ Mode - MISO transmits 0xFF until the internally clocked logic takes over. Data on MOSI is ignored until the internally clocked logic takes over. When this happens, there is no guarantee that the internal clock will be at the correct frequency due to PLL/FLL locking times. This may lead to corrupted data on MISO and in the EZ memory. Therefore, it is recommended to disable CLK_SCB before going to system Deep Sleep, and then wait to enable it until the PLL/FLL have stabilized. The external master needs to be aware that when it reads 0xFF on MISO, the device is not ready yet
- EC_AM_MODE = 1, SCBx_CTRL.EC_OP_MODE = 1, EZ Mode - When transitioning from system Deep Sleep to CPU Active mode, there is no guarantee that the internal clock will be at the correct frequency due to the PLL/FLL locking times. This situation limits the SPI SCLK frequency to 2 MHz. After the FLL/PLL outputs have stabilized, the clock can run faster

***Note**: Only SCB[0] is Deep Sleep-capable in slave mode.*

27.3.4.3 Command-response mode

The command-response mode is defined only for an SPI slave. In the PSC3M/P6 MCU, only one SCB (SCB[0]) supports the command-response mode. This mode has a single memory buffer, a base read address, a current read address, a base write address, and a current write address that is used to index the memory buffer. The base addresses are provided by the CPU. The current addresses are used by the slave to index the memory buffer for sequential accesses of the memory buffer. The memory buffer holds 256 8-bit data elements. The base and current addresses are in the range [0, 255].

The CPU writes and reads to the memory buffer through the SCBx_EZ_DATA registers. These accesses are word accesses, but only the least significant byte of the word is used.

The slave interface accesses the memory buffer using the current addresses. At the start of a write transfer (SPI slave selection), the base write address is copied to the current write address. A data element write is to the current write address location. After the write access, the current address is incremented by '1'. At the start of a read transfer, the base read address is copied to the current read address. A data element read is to the current read address location. After the read data element is transmitted, the current read address is incremented by '1'.

If the current addresses equal the last memory buffer address (255), the current addresses are not incremented. Subsequent write accesses will overwrite any previously written value at the last buffer address. Subsequent read accesses will continue to provide the (same) read value at the last buffer address. The bus master should be aware of the memory buffer capacity in command-response mode.

The base addresses are provided through SCBx_CMD_RESP_CTRL.BASE_RD_ADDR and SCBx_CMD_RESP_CTRL.BASE_WR_ADDR. The current addresses are provided through SCBx_CMD_RESP_STATUS.CURR_RD_ADDR and SCBx_CMD_RESP_STATUS.CURR_WR_ADDR. At the end of a transfer (SPI slave deselection), the difference between a base and current address indicates how many read/write accesses were performed. The block provides interrupt cause fields to identify the end of a transfer. The command-response mode operation is available in CPU Active and System Deep Sleep power modes.

The command-response mode has two phases of operation:

- **Write phase**: The write phase begins with a selection byte, which has its last bit set to '0' indicating the master writes 8-bit data elements to the slave's memory buffer following the selection byte. The slave's current write address is set to the slave's base write address. Received data elements are written to the current write address memory location. After each memory write, the current write address is incremented
- **Read phase**: The read phase begins with a selection byte, which has its last bit set to '1' indicating a read. The master reads 8-bit data elements from the slave's memory buffer. The slave's current read address is

set to the slave's base read address. Transmitted data elements are read from the current address memory. After each read data element is transferred, the current read address is incremented

During the reception of the first byte, the slave (MISO) transmits either 0x62 (ready) or a different value (busy). When disabled or reset, the slave transmits 0xFF (busy). The byte value can be used by the master to determine whether the slave is ready to accept the SPI request.

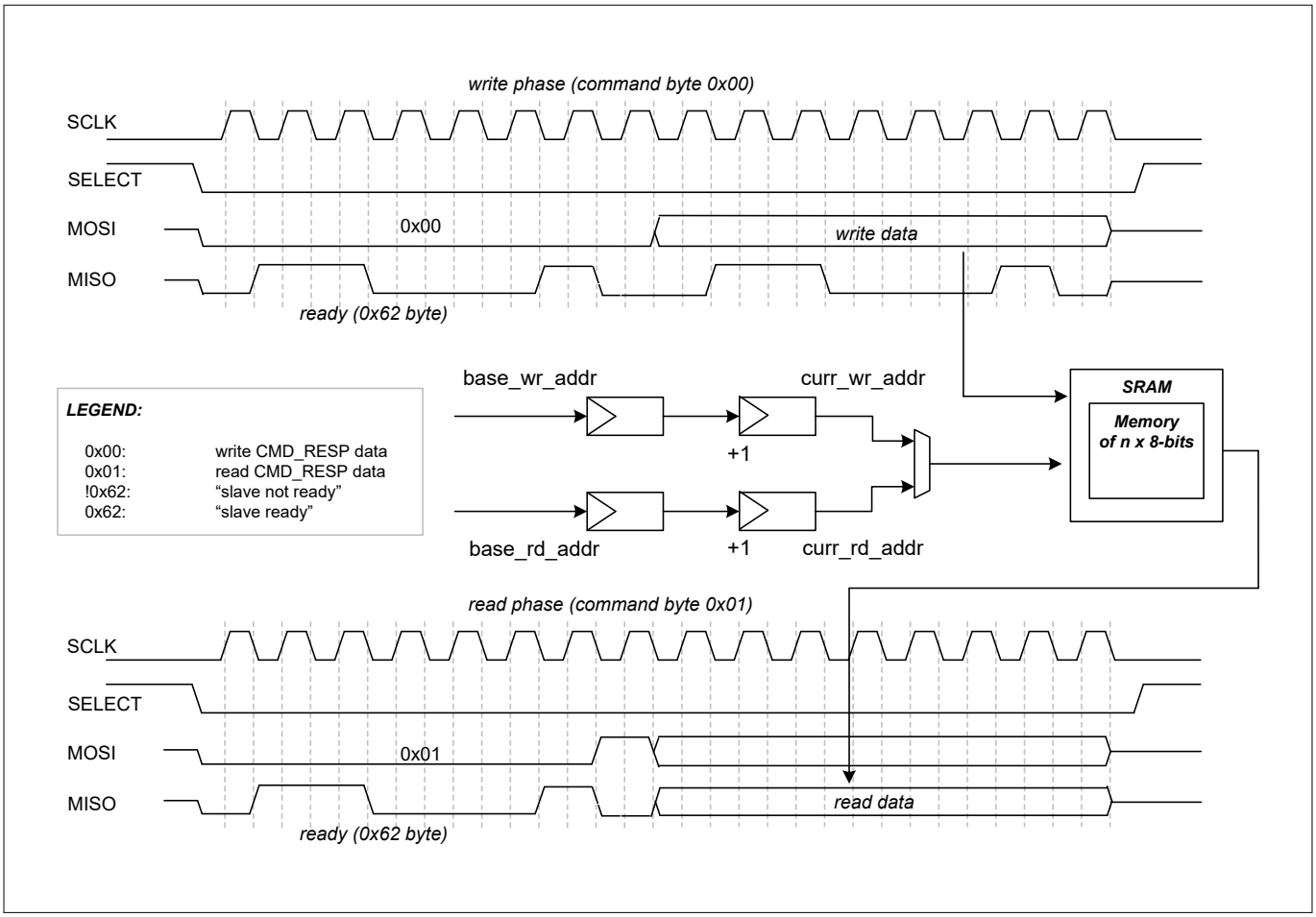


Figure 299 Command-response mode example

***Note**: A slave's base addresses are updated by the CPU and not by the master.*

System Deep Sleep to CPU Active transition

`SCBx_CTRL.EC_AM_MODE` = 1, `SCBx_CTRL.EC_OP_MODE` = 1, CMD_RESP mode.

When transitioning from System Deep Sleep to CPU Active mode, there is no guarantee that the internal clock is at the correct frequency due to PLL/FLL locking times. This situation limits the SPI SCLK frequency to 2 MHz. After the FLL/PLL outputs have stabilized, the clock can run faster.

Configuring SCB for CMD_RESP mode

By default, the SCB is configured for non-CMD_RESP mode of operation. To configure the SCB for CMD_RESP mode, set the register bits in the following order:

1. Select the CMD_RESP mode by writing '1' to the `SCBx_CTRL.CMD_RESP_MODE` register
2. Follow the steps in the Motorola SPI section
3. Follow steps 2 to 4 mentioned in the Enabling and initializing SPI section

For more information on these registers, see the PSC3M/P6 MCU registers reference manual.

27.3.5 Clocking and oversampling

27.3.5.1 Clock modes

The SCB SPI supports both internally- and externally clocked operation modes. The SCBx_CTRL.EC_AM_MODE and SCBx_CTRL.EC_OP_MODE registers determine the SCB clock mode. SCBx_CTRL.EC_AM_MODE indicates whether SPI slave selection is clocked internally (0) or externally (1). SCBx_CTRL.EC_OP_MODE indicates whether the rest of the protocol operation (besides SPI slave selection) is clocked internally (0) or externally (1).

An externally clocked operation uses a clock provided by an external master (SPI SCLK).

***Note**: In the PSC3M/P6 MCU, only the Deep Sleep-capable SCB supports the externally clocked mode of operation and only for SPI slave mode.*

An internally clocked operation uses the programmable clock dividers. For more information on system clocking, see the Clocking system section.

The SCBx_CTRL.EC_AM_MODE and SCBx_CTRL.EC_OP_MODE can be configured as follows:

- EC_AM_MODE is '0' and SCBx_CTRL.EC_OP_MODE is '0': Use this configuration when only the Active mode functionality is required
  - FIFO mode: Supported
  - EZ mode: Supported
  - Command-response mode: Not supported. The slave (MISO) transmits a value different from a ready (0x62) byte during reception of the first byte, if the command-response mode is attempted in this configuration
- EC_AM_MODE is '1' and SCBx_CTRL.EC_OP_MODE is '0': Use this configuration when both CPU Active and System Deep Sleep functionality are required. This configuration relies on the externally clocked functionality to detect the slave selection and relies on the internally clocked functionality to access the memory buffer

The “handover” from external to internal functionality relies on a busy/ready byte scheme. This scheme relies on the master to retry the current transfer when it receives a busy byte and requires the master to support busy/ready byte interpretation. When the slave is selected, SCBx_INTR_SPI_EC.WAKE_UP is set to '1'. The associated System Deep Sleep functionality interrupt brings the system into CPU Active power mode.

- FIFO mode: Supported. The slave (MISO) transmits 0xFF until the CPU is awoken and the TX FIFO is Any data on the MOSI line will be dropped until CLK_SCB is enabled (see System Deep Sleep to CPU Active Transition for more details)
  - EZ mode: Supported. In System Deep Sleep power mode, the slave (MISO) transmits a busy (0xFF) byte during the reception of the command byte. In CPU Active power mode, the slave (MISO) transmits a ready (0xFE) byte during the reception of the command byte
  - CMD_RESP mode: Not supported. The slave transmits (MISO) a value different from a ready (0x62) byte during the reception of the first byte
- EC_AM_MODE is '1' and SCBx_CTRL.EC_OP_MODE is '1'. Use this mode when both CPU Active and System Deep Sleep functionality are required. When the slave is selected, SCBx_INTR_SPI_EC.WAKE_UP is set to '1'. The associated System Deep Sleep functionality interrupt brings the system into CPU Active power mode. When the slave is deselected, SCBx_INTR_SPI_EC.EZ_STOP and/or SCBx_INTR_SPI_EC.EZ_WRITE_STOP are set to '1'
  - FIFO mode: Not supported
  - EZ mode: Supported
  - CMD_RESP mode: Supported

Table 1 SPI mode compatibility

	Internally clocked (IC)			Externally clocked (EC)		
	FIFO	EZ	CMD_RESP	FIFO	EZ	CMD_RESP
SPI master	Yes	No	No	No	No	No
SPI slave	Yes	Yes	No	No[1)]	Yes	Yes

1) In SPI slave FIFO mode, the externally clocked logic does selection detection, then triggers an interrupt to wake up the CPU. Writes will be ignored and reads will return 0xFF until the CPU is ready and the FIFO is populated.

If SCBx_CTRL.EC_OP_MODE is '1', the external interface logic accesses the memory buffer on the external interface clock (SPI SCLK). This allows for EZ and CMD_RESP mode functionality in the CPU Active and System Deep Sleep power modes.

In CPU Active system power mode, the memory buffer requires arbitration between the external interface logic (on SPI SCLK) and the CPU interface logic (on system peripheral clock). This arbitration always gives the highest priority to the external interface logic (host access). The external interface logic takes two serial interfaces clock/bit periods for SPI. During this period, the internal logic is denied service to the memory buffer. The PSC3M/P6 MCU provides two programmable options to address this “denial of service”:

- If the SCBx_CTRL.BLOCK is '1': An internal logic access to the memory buffer is blocked until the memory buffer is granted and the external interface logic has completed access. This option provides normal SCB register functionality, but the blocking time introduces additional internal bus wait states
- If the SCBx_CTRL.BLOCK is '0': An internal logic access to the memory buffer is not blocked, but fails when it conflicts with an external interface logic access. A read access returns the value 0xFFFF:FFFF and a write access is ignored. This option does not introduce additional internal bus wait states, but access to the memory buffer may not take effect. In this case, the following failures are detected:
  - Read failure: A read failure is easily detected because the returned value is 0xFFFF:FFFF. This value is unique as nonfailing memory buffer read accesses return an unsigned byte value in the range 0x0000:0000-0x0000:00ff
  - Write failure: A write failure is detected by reading back the written memory buffer location, and confirming that the read value is the same as the written value

For both options, a conflicting internal logic access to the memory buffer sets SCBx_INTR_TX.BLOCKED field to '1' (for write access) and SCBx_INTR_RX.BLOCKED field to '1' (for read access). These fields can be used as either status fields or as interrupt cause fields (when their associated mask fields are enabled).

If a series of read or write accesses is performed and SCBx_CTRL.BLOCK is '0', a failure is detected by comparing the “logical OR” of all read values to 0xFFFF:FFFF and checking the SCBx_INTR_TX.BLOCKED and SCBx_INTR_RX.BLOCKED fields to determine whether a failure occurred for a series of write or read operations.

27.3.5.2 Using SPI master to clock slave

In a normal SPI master mode transmission, the SCLK is generated only when the SCB is enabled and data is being transmitted. This can be changed to always generate a clock on the SCLK line while the SCB is enabled.

This is used when the slave uses the SCLK for functional operations other than the SPI functionality. To enable this, write '1' to the SCBx_SPI_CTRL.SCLK_CONTINUOUS register.

27.3.5.3 Oversampling and bit-rate

SPI master mode

The SPI master does not support the externally clocked mode. In the internally clocked mode, the logic operates under the internal clock. The internal clock has a higher frequency than the interface clock (SCLK), such that the master can oversample its input signals (MISO).

The SCBx_CTRL.OVS register specifies the oversampling. The oversampling rate is calculated as the value in the SCBx_CTRL.OVS register + 1. In SPI master mode, the valid range is from 4 to 16. For SPI master with MOSI only (transmission only), the valid range is from 2 to 16. The bit rate is calculated as follows:

Bit rate = Input clock/SCBx_CTRL.OVS

Therefore, with an input clock of 100 MHz, the maximum bit rate is 25 Mbps with MISO, or 50 Mbps without MISO.

***Note**: For using a bit rate of 50 Mbps, ensure that the following parameters are set to avoid potential metastable conditions:*

- *CLK_SCB: 200 MHz*
- *OVS: 3*
- *MEDIAN: 1*

The above values indicate how fast the SCB hardware can run SCLK. However, it does not indicate whether the master can correctly receive data from a slave at those speeds. To determine that, calculate the path delay of MISO using the following equation:

$1/2 \times t_{SCLK} \geq t_{SCLK_PCB_D} + t_{DSO} + t_{SCLK_PCB_D} + t_{DSI}$

Where,

- t_{SCLK} is the period of the SPI clock
- $t_{SCLK_PCB_D}$ is the SCLK PCB delay from master to slave
- t_{DSO} is the total internal slave delay, time from SCLK edge at slave pin to MISO edge at slave pin
- $t_{SCLK_PCB_D}$ is the MISO PCB delay from slave to master t_{DSI} is the master setup time

Most slave datasheets list t_{DSO}. It may have a different name, so that look for the MISO output valid after the SCLK edge.

Most master datasheets also list t_{DSI} or master setup time. Calculate $t_{SCLK_PCB_D}$ and $t_{SCLK_PCB_D}$ based on specific PCB geometries.

If the desired speed is not achieved after doing these calculations, then consider using the MISO late sample feature of the SCB. This feature tells the SCB to sample the incoming MISO signal on the next edge of SCLK, and hence enables one-half SCLK cycle more timing margin.

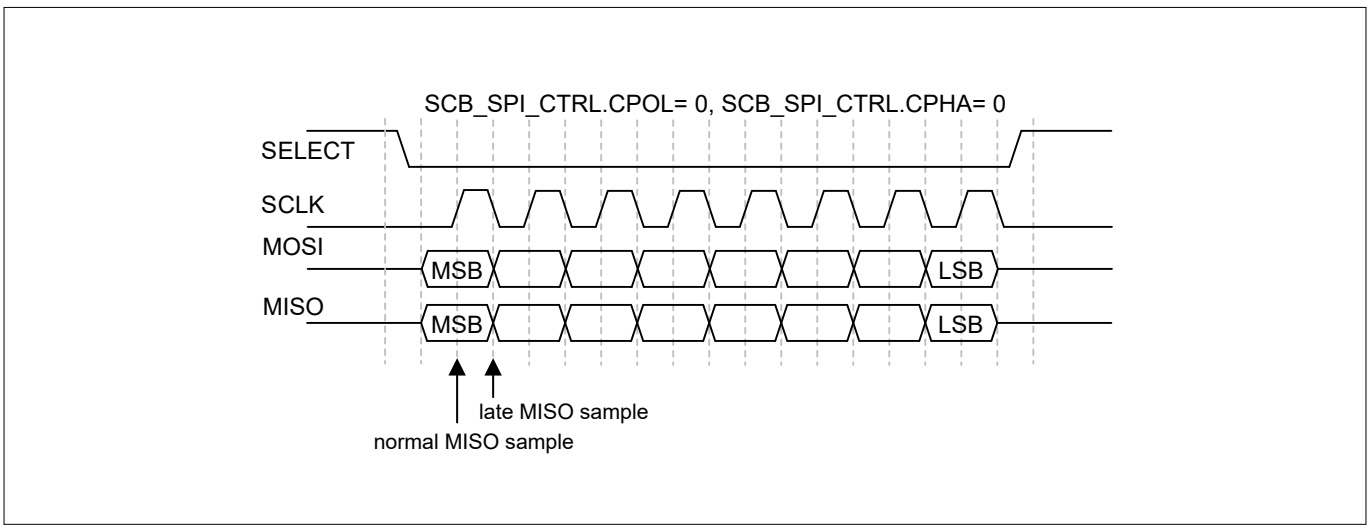


Figure 300 MISO sampling timing

This changes the equation to:

$t_{SCLK} \geq t_{SCLK_PCB_D} + t_{DSO} + t_{SCLK_PCB_D} + t_{DSI}$

Because the late sample helps in achieving better timing, leave it enabled all the time.

The t_{DSI} specification in the device datasheet assumes that the late sample is enabled.

***Note**: The SCBx_SPI_CTRL.LATE_MISO_SAMPLE is set to '1' by default.*

SPI slave mode

In the SPI slave mode, the SCBx_CTRL.OVS register is not used. The data rate is determined by the earlier equations. The late MISO sample is determined by the external master and not by SCBx_SPI_CTRL.LATE_MISO_SAMPLE.

For PSC3M/P6 MCUs, tDSO is provided in the device datasheet. For the internally-clocked mode, it is proportional to the frequency of the internal clock. For example, it may be:

TDSO_EXT = TDSO_INT + 3 × Tclk_scb

TDSO_EXT, output data valid after the SCLK driving edge (external SPI Master clock)

TDSO_INT, when the external SPI Master does not have LATE_SAMPLE

***Note**: For using a bit rate of 50 Mbps, ensure that the following parameters are set to avoid potential metastable conditions:*

- *CLK_SCB: 200 MHz*
- *OVS: 4*
- *MEDIAN: 1*

27.3.6 SPI master SELECT output timing control

The SPI master SELECT output signal's “select” timing is a variable. This applies to the following:

- The SELECT setup time (select active to SPI clock)
- The SELECT hold time (SPI clock to select inactive)
- The inter-data frame deselect time (select inactive to select active)

The following options can be selected for these delays:

- SELECT setup time
  - When SCBx_SPI_CTRL.CPHA = 0: 0.75 or 1.75 SPI clock cycles (for SPI slaves requiring a longer setup time)
  - When SCBx_SPI_CTRL.CPHA = 1: 0.25 or 1.25 SPI clock cycles (for SPI slaves requiring a longer setup time)
- SELECT hold time (SCBx_SPI_CTRL.SSEL_HOLD_DEL register)
  - When SCBx_SPI_CTRL.CPHA = 0: 0.25 or 1.25 SPI clock cycles (for SPI slaves requiring a longer setup time)
  - When SCBx_SPI_CTRL.CPHA = 1: 0.75 or 1.75 SPI clock cycles (for SPI slaves requiring a longer setup time)

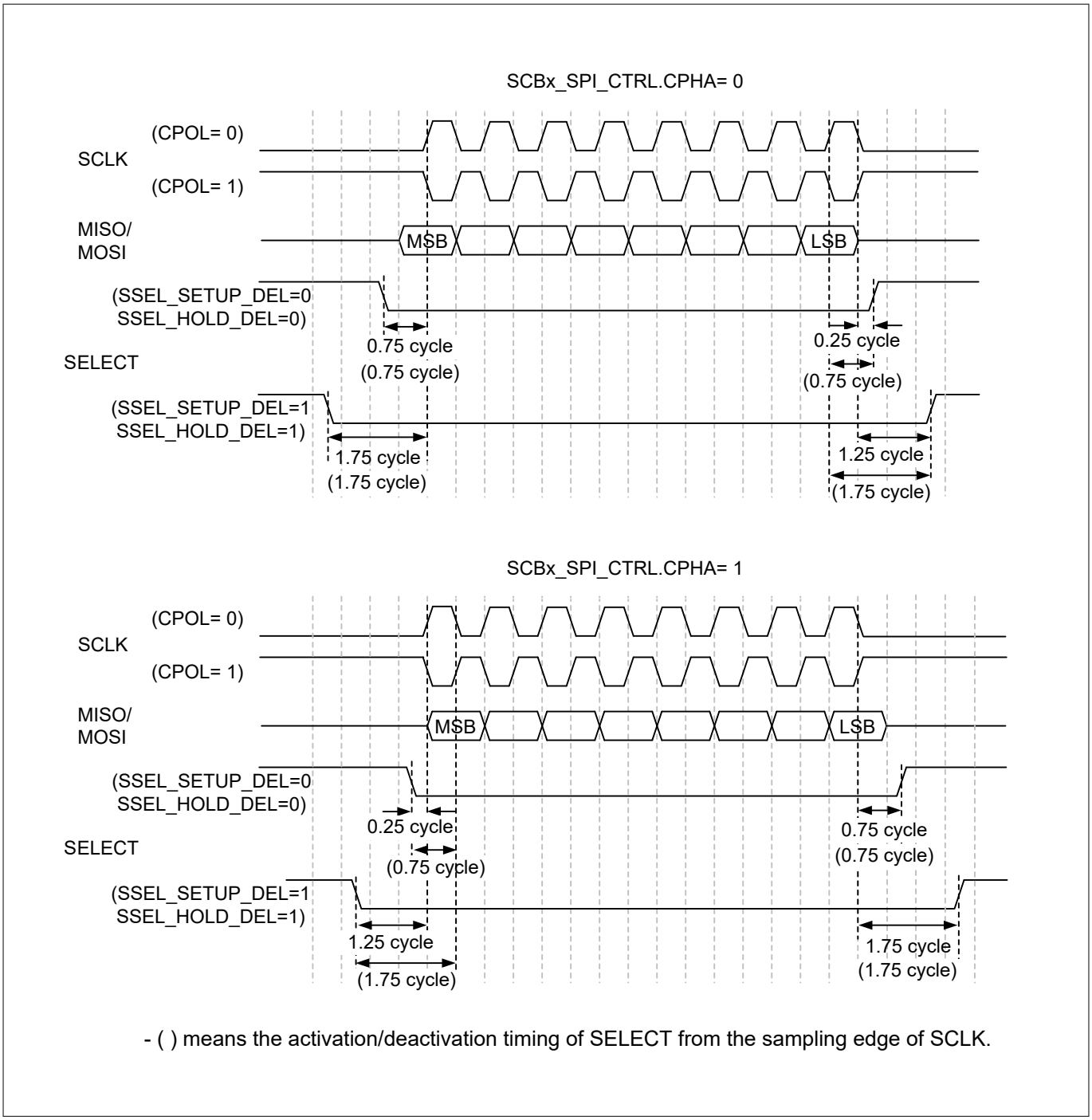


Figure 301 SELECT setup/hold delay

- INTER-FRAME deselect time (`SCBx_SPI_CTRL.SSEL_INTER_FRAME_DEL` register):
  - 1.5 SPI clock cycles or 2.5 SPI clock cycles (for SPI slaves requiring a longer deselect time)

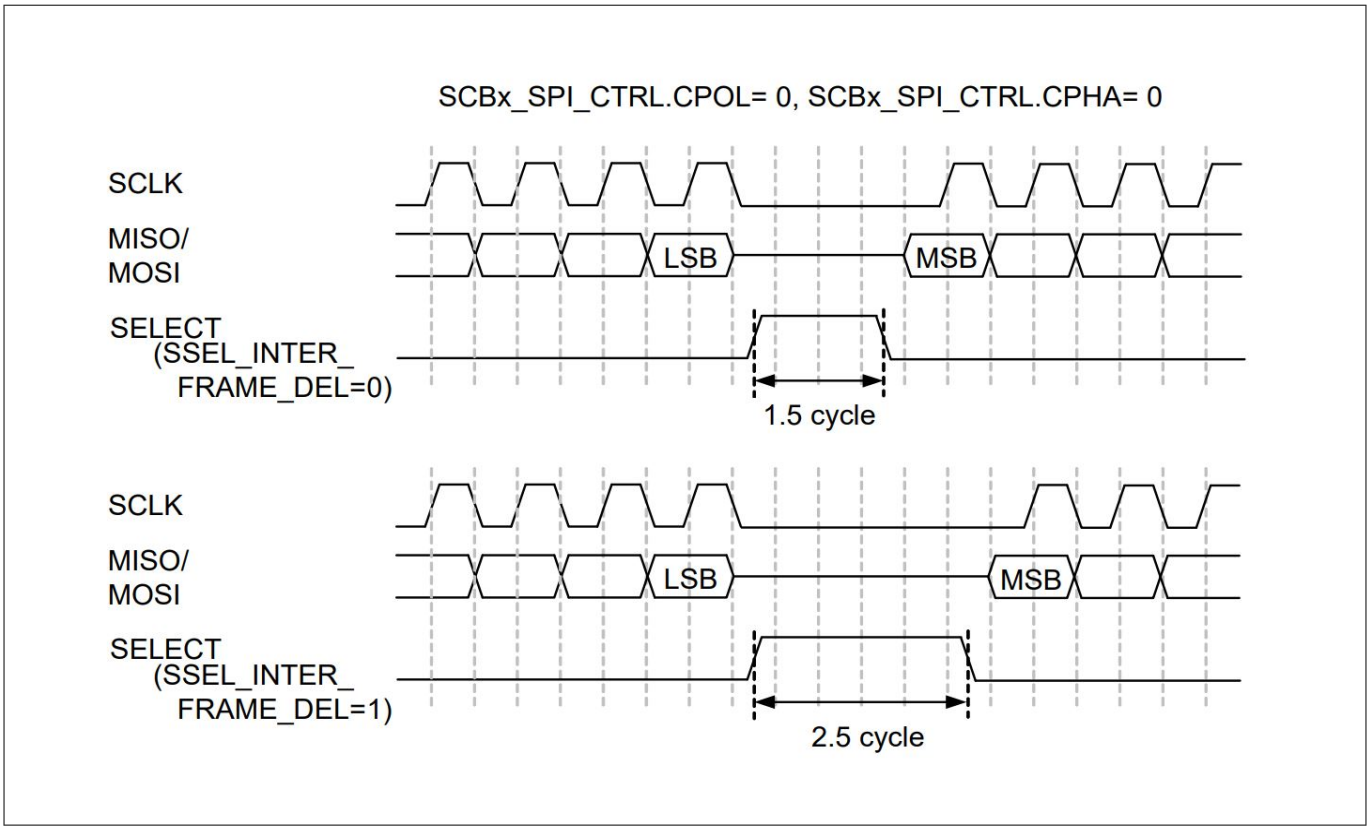


Figure 302 SELECT INTER-FRAME deselect time

27.3.7 SPI parity functionality

The features of the parity functionality feature in SPI mode are as follows:

- Applies to the SPI master and SPI slave with internally-clocked operation
- Adds a parity bit to the data frame and is used to identify single-bit data frame errors. The parity bit directly follows the data frame bits
- Can be configured to be enabled or disabled using `SCBx_SPI_TX_CTRL.PARITY_ENABLED` and `PARITY_ENABLED` individually
- When transmitting, a parity bit can be inserted. When receiving, the parity bit can be checked. If parity fails, it is possible to select whether the received data is sent to the RX FIFO or is dropped and lost, using the `DROP_ON_PARITY_ERROR` register. Additionally, the `PARITY_ERROR` bit in the `SCBx_INTR_RX` is set
- Supports even and odd parity (`SCBx_SPI_TX_CTRL.PARITY`, `SCBx_SPI_RX_CTRL.PARITY`)

27.3.8 SPI SSEL lines

The four SPI Slave Select lines are extended to 16 lines.

A new register “SPI_POLARITY” is implemented to support the extension of SSEL_POLARITY bit field to 16. The new register includes SSEL_POLARITY4 to SSEL_POLARITY16. The already existing SSEL_POLARITY0 to SSEL_POLARITY3 remain in SPI_CTRL register.

27.3.9 Loop-back

In SPI Master mode, the SCB supports internal loop-back from an output signal for MOSI to an input signal for MISO without affecting the information on the pins. It is configured using the `SCBx_SPI_CTRL.LOOPBACK` register.

This loop-back is not supported in National Semiconductors mode.

27.3.10 Enabling and initializing SPI

Do the following to program the SPI:

1. Program the protocol-specific information using the SCBx_SPI_CTRL register. This includes selecting the sub-modes of the protocol (MODE), master-slave functionality (MASTER_MODE), one of four SELECT (SSEL), whether SELECT stays active for a whole transfer, or just for each data frame width (SSEL_CONTINUOUS), and SELECT polarity (SSEL_POLARITY 0-15). EZSPI and CMD_RESP can be used with Slave mode only
2. Program the generic transmitter and receiver information using the SCBx_TX_CTRL and SCBx_RX_CTRL registers:
   a. Specify the data frame width. This should always be 8 for EZSPI
   b. Specify whether MSb or LSb is the first bit to be transmitted/received. This should always be MSb first for EZSPI and CMD_RESP
3. Program the transmitter and receiver FIFOs using the SCBx_TX_FIFO_CTRL and SCBx_RX_FIFO_CTRL registers respectively, as shown in the SCBx_TX_FIFO_CTRL/SCBx_RX_FIFO_CTRL registers. Enable the following only for the FIFO mode:
   a. Set the trigger level (TRIGGER_LEVEL)
   b. Clear the transmitter and receiver FIFO and Shift registers (CLEAR)
4. Enable the block (write a '1' to the SCBx_CTRL.ENABLED register). After the block is enabled, do not change the control bits. Make changes after disabling the block; for example, to modify the operation mode (from Motorola mode to TI mode) or to go from externally-clocked to internally-clocked operation. The change is effective only after the block is re-enabled. Note that re-enabling the block causes reinitialization and the associated state is lost (for example, FIFO content)

27.3.11 Fast SPI Master/Slave support

SCB[5]* is capable of Fast SPI operation of up to 50 MHz. The modes supported are SPI Master, SPI slave with internal clocking, and SPI slave with external clocking.

For Fast SPI operation, the clk_peri needs to operate at a frequency of 200-240 MHz.

27.3.12 I/O pad connection

27.3.12.1 SPI master

In the SPI master mode, the SCB provides data transmit and data receive functionality. Figure 303 and Table 280 list the use of the I/O pads for the SPI master.

Typically, the Strong drive mode (GPIO_PRTx_CFG.DRIVE_MODEy = 6) is used for output signals. When SCB is disabled, the respective out_en signals will be 0, so the output will be High-Z; to avoid the High-Z state, do one of the following:

- Use GPIO to drive the output to idle level
- Use Pull-up or Pull-down drive modes with an internal pull-up/pull-down resistor (fixed resistance number)
- Use the Strong drive mode, using an external pull-up/pull-down resistor (flexible resistance number)

The internal and external pull-up/pull-down resistors have a negative impact on the maximum data rate.

For SPI MISO input in normal full-duplex mode, when the SPI slave device is not selected, its MISO output will be High-Z. If all SPI slave devices connected to the SPI master are not selected, the MISO line will be High-Z. A pull-up resistor is needed on the MISO line to avoid High-Z state.

- High-impedance (High-Z) drive mode, using external pull-up/pull-down resistor
- Pull-Up/Pull-Down drive mode, using internal pull-up/pull-down resistor
- The internal and external pull-up/pull-down resistors have a negative impact on the maximum data rate

Half-duplex mode is not supported, because it drives "strong pull-up, strong pull-down" in normal functional mode (when spi_mosi_out_en = 1).

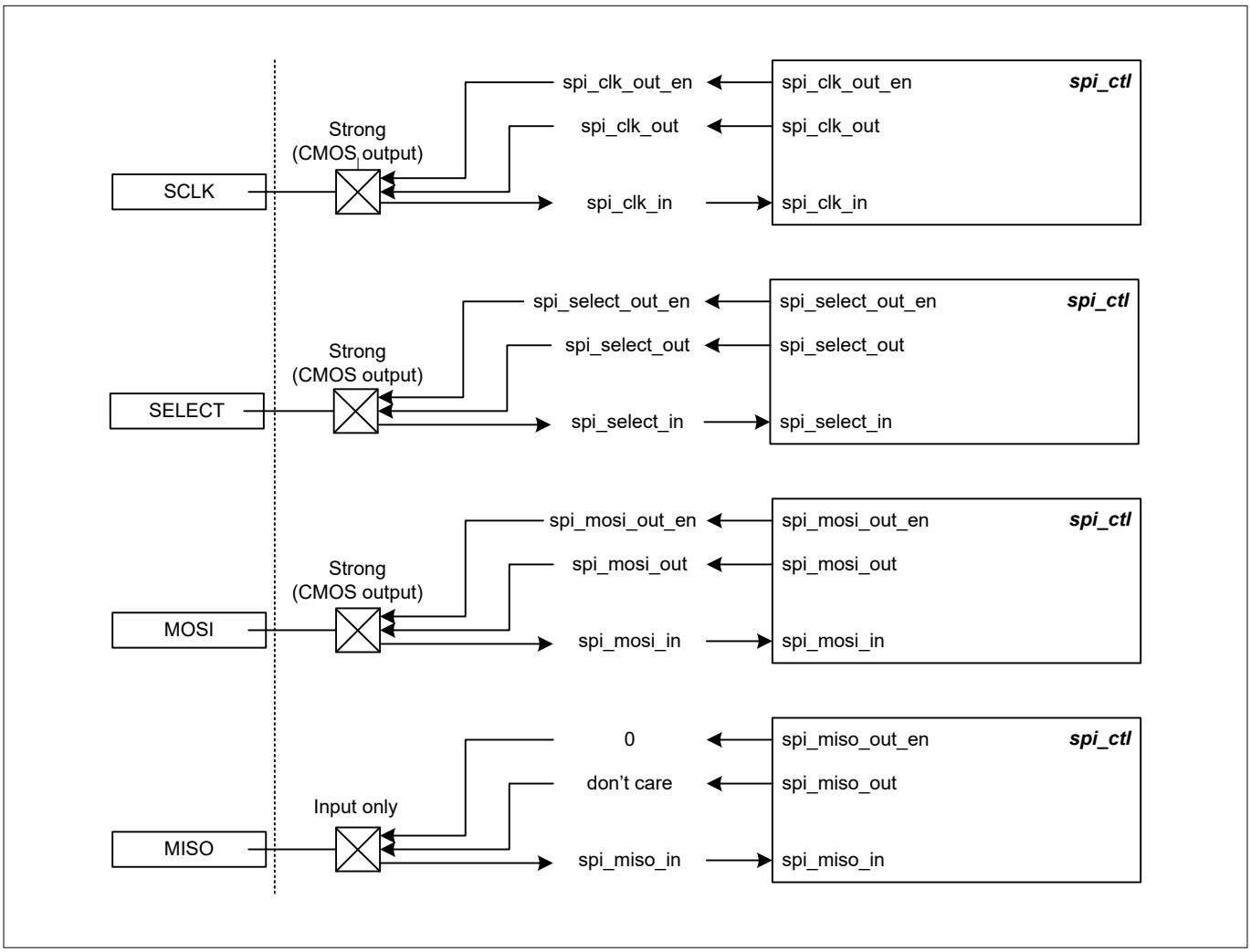


Figure 303 SPI master I/O pad connections

Table 280 SPI master I/O pad connection usage

I/O pads	Drive mode	On-chip I/O signals	Usage
SCLK	Strong (CMOS output)	spi_clk_out_en spi_clk_out	Transmit a clock signal
SELECT	Strong (CMOS output)	spi_select_out_en spi_select_out	Transmit a select signal
MOSI	Strong (CMOS output)	spi_mosi_out_en spi_mosi_out	Transmit a data element
MISO	Input only	spi_miso_in spi_miso_out spi_miso_out_en	Receive a data element

27.3.12.2 SPI slave

In the SPI slave mode, the SCB provides data transmit and data receive functionality. Figure 13 and Table 3 list the use of I/O pads for SPI slave.

Typically, the Strong drive mode (GPIO_PRTx_CFG.DRIVE_MODEy = 6) is used for output signals. When SCB is disabled, the respective out_en signals will be '0'. Therefore, the output will be High-Z. To avoid the High-Z state, do one of the following:

- Use GPIO to drive the output to idle level
- Use Pull-Up or Pull-Down drive modes with an internal pull-up/pull-down resistor (fixed resistance number)
- Use the Strong drive mode, using an external pull-up/pull-down resistor (flexible resistance number)

The internal and external pull-up/pull-down resistors have a negative impact on the maximum data rate.

When SCBx_TX_CTRL.OPEN_DRAIN = 1, MOSI and MISO can be shorted together to work in the half-duplex mode.

- The drive mode of the MISO output can be Open Drain Drives Low only
- Add an external pull-up resistor on the MISO line

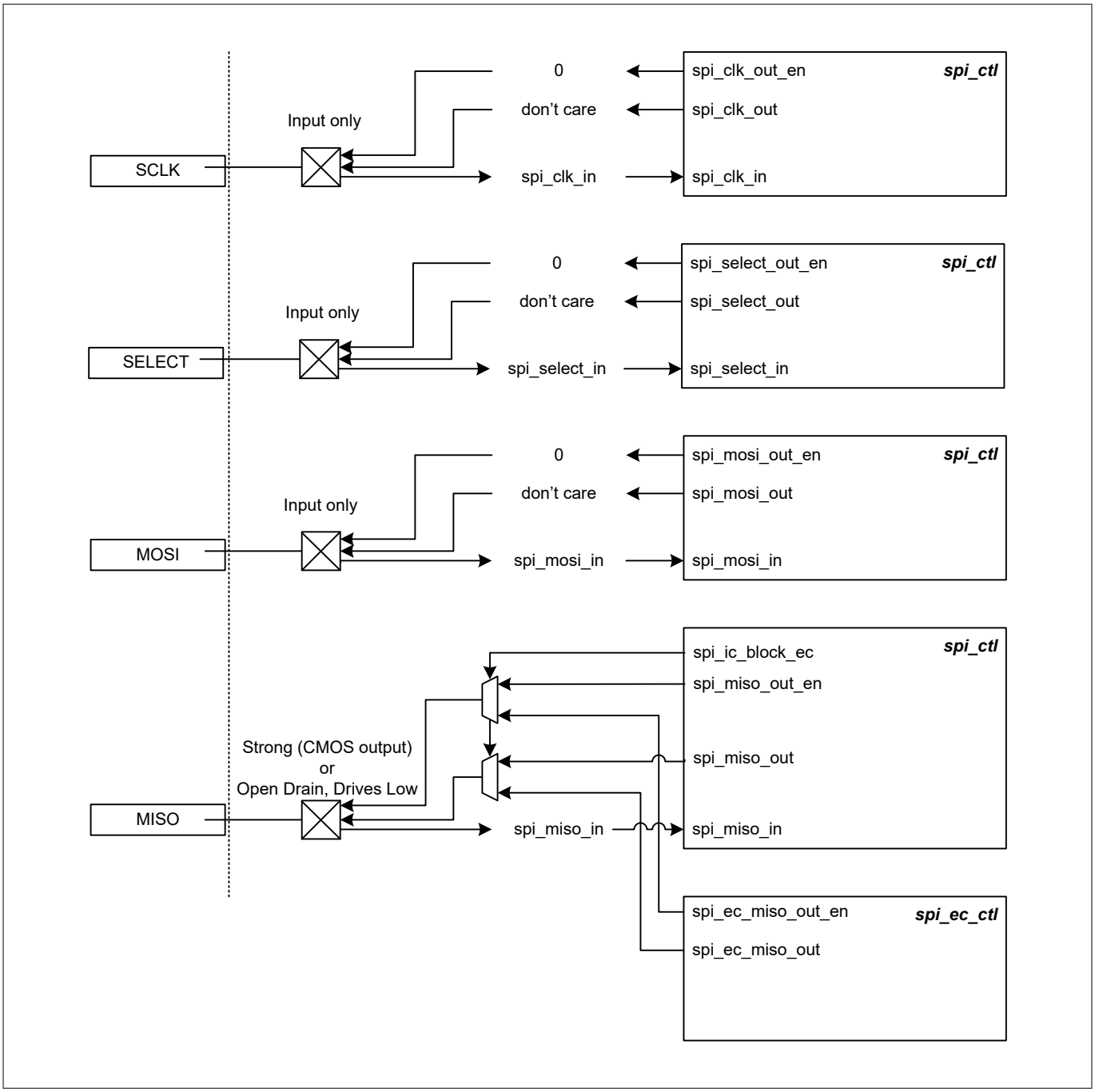


Figure 304 SPI slave I/O pad connections

Table 281 SPI slave I/O signal description

I/O pads	Drive mode	On-chip I/O signals	Usage
SCLK	Input only	spi_clk_out_en spi_clk_out spi_clk_in	Receive a clock signal
SELECT	Input only	spi_select_out_en spi_select_out spi_select_in	Receive a select signal

(table continues...)

Table 281 (continued) SPI slave I/O signal description

I/O pads	Drive mode	On-chip I/O signals	Usage
MOSI	Input only	spi_mosi_out_en spi_mosi_out spi_mosi_in	Receive a data element
MISO	Strong (CMOS output), or open drain drives low	spi_miso_out_en spi_miso_out spi_miso_in	Transmit a data element

27.3.13 Registers summary

The SPI interface is controlled using a set of 32-bit control and status registers. For more information on these registers, see the PSC3M/P6 MCU registers RM.

27.4 UART

The Universal Asynchronous Receiver/Transmitter (UART) protocol is an asynchronous serial interface protocol.

UART communication is typically point-to-point. The UART interface consists of two signals:

- TX: Transmitter output
- RX: Receiver input

Additionally, two side-band signals are used to implement flow control in UART. Note that the flow control applies only to the TX functionality.

- Clear to Send (CTS): This is an input signal to the transmitter. If active, it means that the receiver has signaled to the transmitter that it is ready to receive data and now the transmitter can send the data
- Request to Send (RTS): This is an output signal from the receiver. If active, it indicates that the receiver is ready to receive data
- The RTS of one device is connected to the CTS of the other and vice-versa. When a device is ready to receive data (receiver), it activates it's RTS which in turn activates the CTS of the other device (transmitter) signaling it to send the data

***Note**: The flow control is only available in the UART standard protocol mode and is not available in the following modes - UART Local Interconnect Network (LIN), SmartCard (ISO7816), Infrared Data Association (IrDA), and UART multiprocessor mode.*

27.4.1 Features

The SCB supports the following features:

- Supports UART protocol
  - Standard UART (full duplex, half duplex)
  - Multi-processor mode
- SmartCard (ISO7816) reader
- IrDA
- Supports Local Interconnect Network (LIN)
  - Break detection

  - Baud rate detection
  - Collision detection (ability to detect that a driven bit value is not reflected on the bus, indicating that another component is driving the same bus)
- Data frame size programmable from 4 to 16 bits
- Programmable number of STOP bits, which can be set in terms of half-bit periods between 1 and 7
- Parity support (odd and even parity)
- Programmable oversampling
- Start skipping
- Hardware flow control

27.4.2 General description

Figure 305 illustrates UART TX and RX transfers.

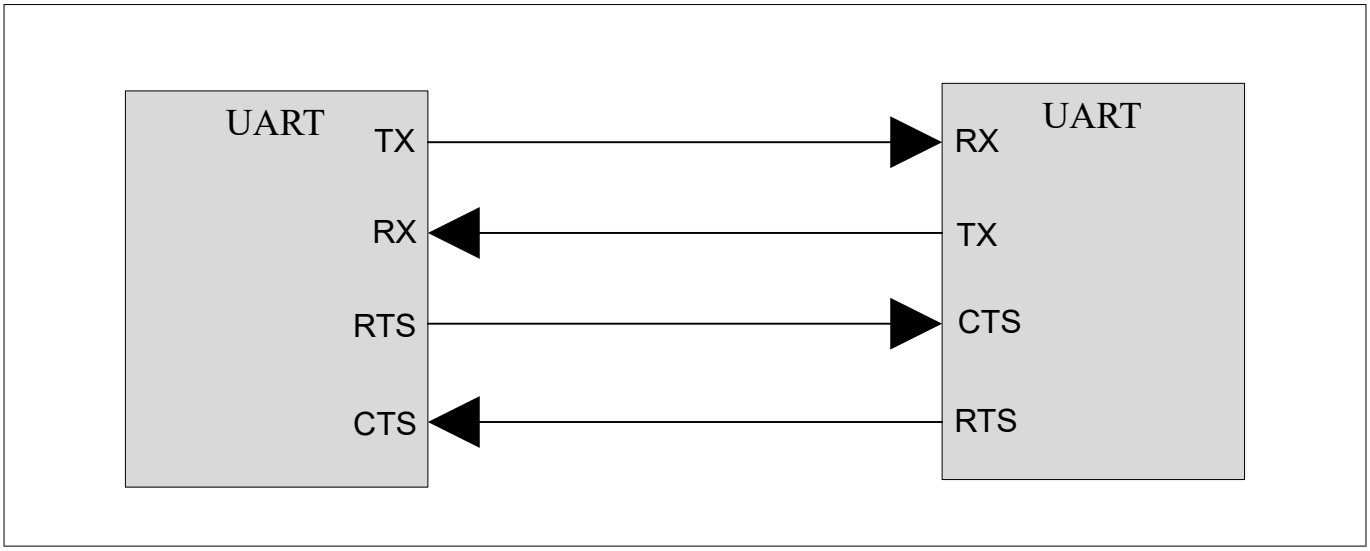


Figure 305 UART Example

A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit and finally completed by one or more stop bits. The start and stop bits indicate the start and end of data transmission. The parity bit is sent by the transmitter and is used by the receiver to detect single-bit errors. Because the interface does not have a clock (asynchronous), the transmitter and receiver use their own clocks; thus, the transmitter and receiver need to agree on the baud rate.

By default, UART supports a data frame width of eight bits. However, the data frame width for both transmit (TX) and receive (RX) can be configured to any value in the range of 4 to 9 bits. This does not include start, stop, and parity bits. The number of stop bits can be in the range of 1 to 7. The parity bit can be either enabled or disabled. If enabled, the type of parity can be set to either even parity or odd parity. The option of using the parity bit is available only in the Standard UART and SmartCard UART modes. For the IrDA UART mode, the parity bit is automatically disabled.

***Note**: The UART interface does not support an external clocking operation. Hence, the UART operates only in the Active and Sleep system power modes. The UART also supports only the FIFO buffer mode.*

27.4.3 UART modes of operation

27.4.3.1 Standard protocol

A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit and finally completed by one or more stop bits. The start bit value is always '0', the data bits values are dependent on the data transferred, the parity bit value is set to a value guaranteeing an even or odd parity over the data bits, and the stop bit value is '1'. The parity bit is generated by the transmitter and can be used by the receiver to detect single-bit transmission errors. When not transmitting data, the TX line is '1' – the same value as the stop bits.

Because the interface does not have a clock, the transmitter and receiver must agree upon the baud rate. The transmitter and receiver have their own internal clocks. The receiver clock runs at a higher frequency than the bit transfer frequency, such that the receiver may oversample the incoming signal.

The transition of a stop bit to a start bit is represented by a change from '1' to '0' on the TX line. This transition can be used by the receiver to synchronize with the transmitter clock. Synchronization at the start of each data transfer allows error-free transmission even in the presence of frequency drift between the transmitter and receiver clocks. The required clock accuracy is dependent on the data transfer size.

The stop period or the amount of stop bits between successive data transfers is typically agreed upon between the transmitter and receiver, and is typically in the range of 1 to 3-bit transfer periods.

Figure 306 illustrates the UART protocol.

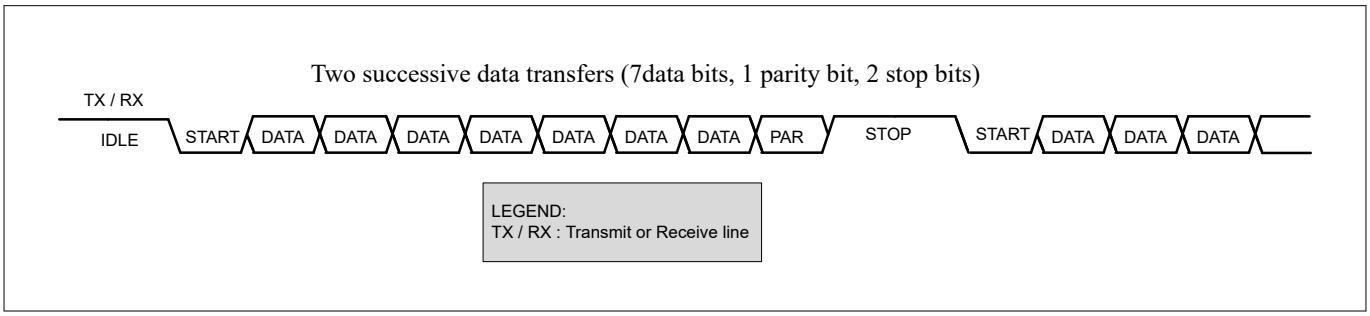


Figure 306 UART standard protocol example

The receiver oversamples the incoming signal, the value of the sample point in the middle of the bit transfer period (on the receiver's clock) is used. This is illustrated by Figure 307. Note the slight drift between transmitter and receiver clock.

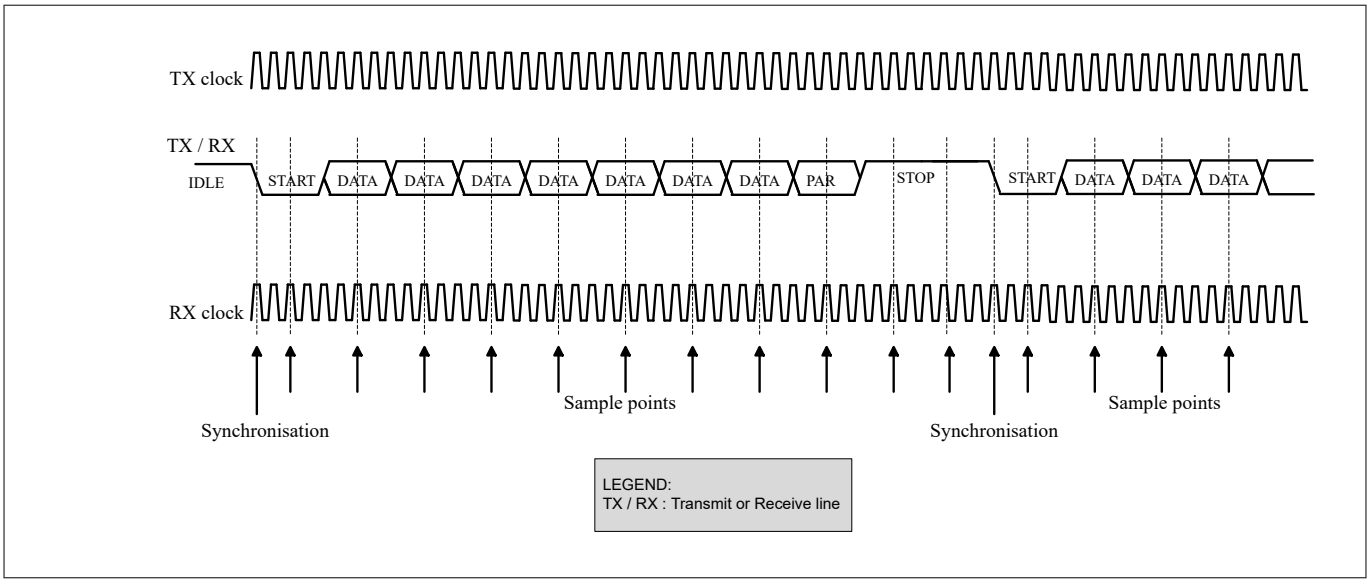


Figure 307 UART standard protocol oversampling (RX clock faster than TX clock)

Alternatively, three samples around the middle of the bit transfer period (on the receiver's clock) are used for a majority vote to reduce the noise sensitivity and to increase accuracy; this is enabled by enabling the MEDIAN filter in the SCBx_RX_CTRL register.

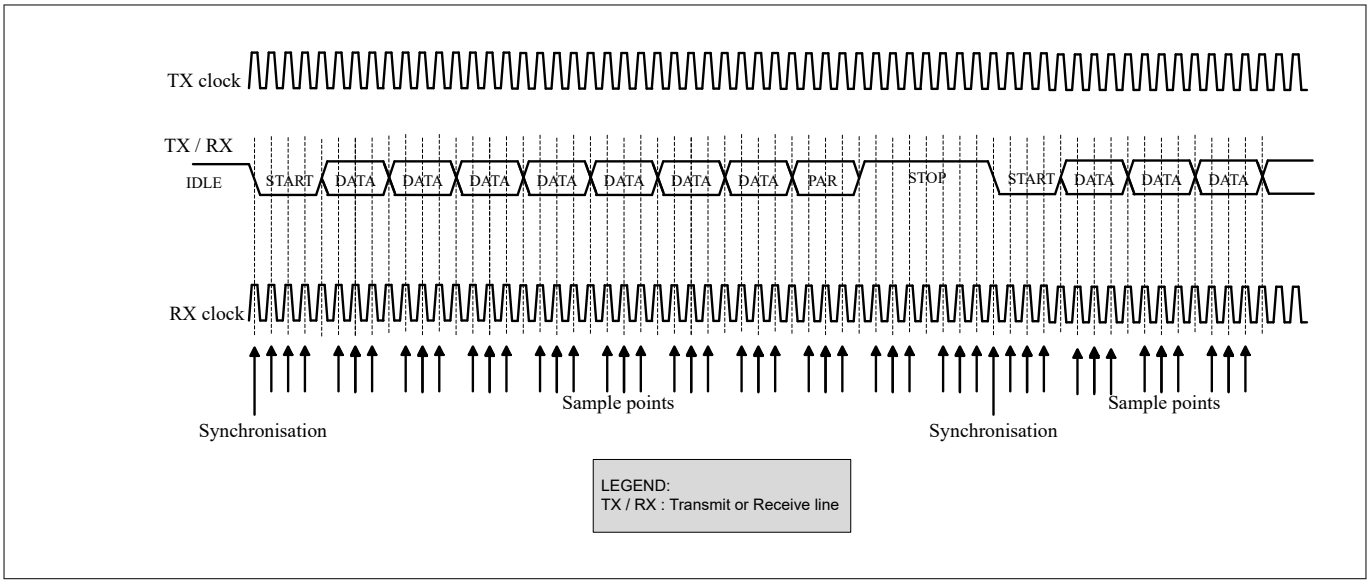


Figure 308 UART standard protocol oversampling (RX clock equal to TX clock)

27.4.3.1.1 Parity

This functionality adds a parity bit to the data frame and is used to identify single-bit data frame errors. The parity bit is always directly after the data frame bits.

The transmitter calculates the parity bit (when SCBx_UART_TX_CTRL.PARITY_ENABLED is '1') from the data frame bits, such that data frame bits and parity bit have an even (SCBx_UART_TX_CTRL.PARITY is '0') or odd (SCBx_UART_TX_CTRL.PARITY is '1') parity. The receiver checks the parity bit (when SCBx_UART_RX_CTRL.PARITY_ENABLED is '1') from the received data frame bits, such that data frame bits and parity bit have an even (SCBx_UART_RX_CTRL.PARITY is '0') or odd (SCBx_UART_RX_CTRL.PARITY is '1') parity.

- Parity applies to both TX and RX functionality and dedicated control fields are available
  - TX functionality: SCBx_UART_TX_CTRL.PARITY and SCBx_UART_TX_CTRL.PARITY_ENABLED
  - RX functionality: SCBx_UART_RX_CTRL.PARITY and SCBx_UART_RX_CTRL.PARITY_ENABLED
- When a receiver detects a parity error, the data frame is either put in the RX FIFO (SCBx_UART_RX_CTRL.DROP_ON_PARITY_ERROR is '0') or dropped (SCBx_UART_RX_CTRL.DROP_ON_PARITY_ERROR is '1')

The Figure 309 illustrates the parity functionality (8-bit data frame).

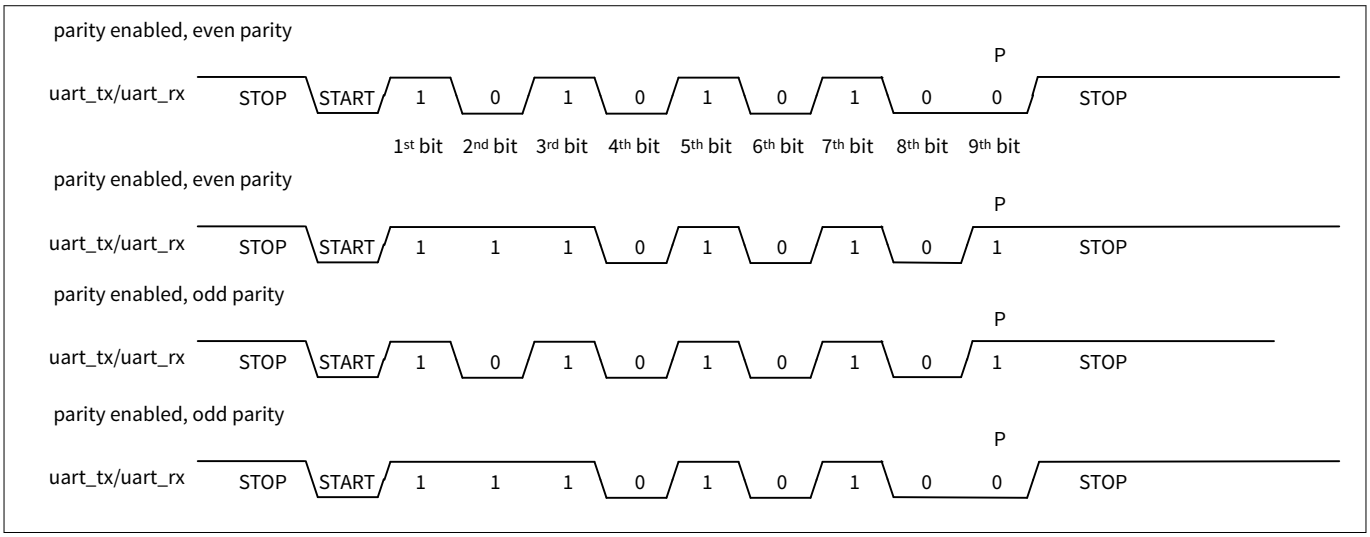


Figure 309 UART parity examples

27.4.3.1.2 Start skipping

Start skipping applies only to receive functionality. The standard UART mode supports "start skipping". Regular receive operation synchronizes on the START bit period (a 1 to 0 transition on the UART RX line), start skipping receive operation synchronizes on the first received data frame bit, which must be a '1' (a 0 to 1 transition on UART RX).

Start skipping is used to allow to wake up from the System Deep Sleep UART RX functionality. The intended process is described as follows:

1. Before entering System Deep Sleep power mode, the UART RX functionality is disabled, and the IOSS/GPIO is programmed to set an interrupt cause to '1' when UART_RX has a '1' to '0' transition (START bit)
2. While in System Deep Sleep, the UART RX functionality is not functional
3. The IOSS/GPIO interrupt is activated on the START bit and the system transitions from System Deep Sleep to CPU Active power mode. This transition takes up to roughly 20 µs
4. The CPU enables the UART RX functionality, with SCBx_UART_RX_CTRL.SKIP_START set to '1'
5. The UART receiver synchronizes the data frame receipt on the next '0' to '1' transition. If the UART RX functionality is enabled in time, this is transition from the START bit to the 1st received data frame bit
6. The UART receiver proceeds with normal operation; that is, synchronization of successive data frames is on the START bit period

Figure 310 illustrates the process.

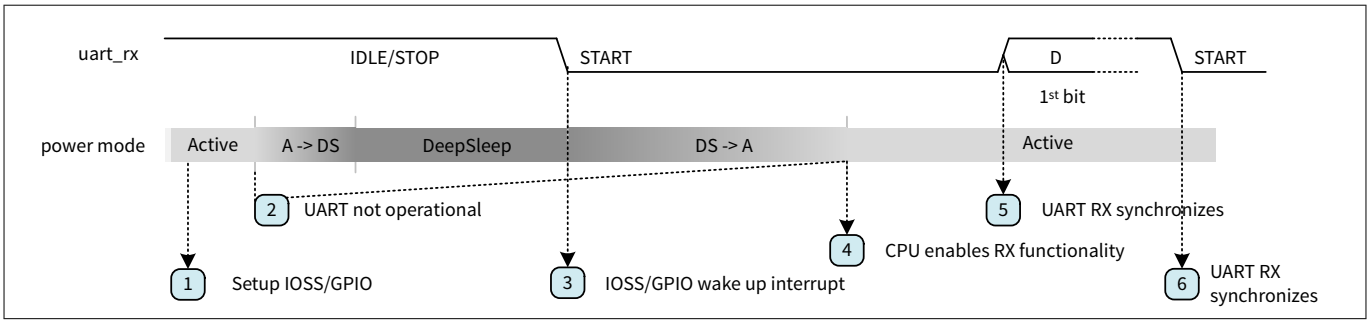


Figure 310 UART Start Skip and Wakeup from System Deep Sleep

Note that the above process works only for lower baud rates. The System Deep Sleep to CPU Active power mode transition and CPU enabling the UART receive functionality should take less than 1-bit period to ensure that the UART receiver is active in time to detect the '0' to '1' transition.

In step 4 of the above process, the firmware takes some time to finish the wake-up interrupt routine and enable the UART receive functionality before the block can detect the input rising edge on the UART RX line.

If the above steps cannot be completed in less than 1-bit time, first send a "dummy" byte to the device to wake it up before sending real UART data. In this case, the SKIP_START bit can be left as 0.

This feature can work, only if the GPIO interrupt is triggered on falling edge of the START bit (not other data bits). Otherwise, the UART will receive corrupted data bytes. It depends on higher level protocols to detect the data corruption and to tell the UART when to sleep safely.

27.4.3.1.3 Half-duplex mode

The UART Half Duplex connection is supported only in Standard UART sub mode (CTRL.MODE is UART and UART_CTRL.MODE is UART_STD). From the protocol point of view, all features of standard UART sub mode are supported except the parallel transmission and reception, which is caused by the single transmission line. To enable the half-duplex pad connection, UART_RX_CTRL.HDRXEN needs to be set to '1'. Set UART_RX_CTRL.HDRXEN='0' for full-duplex communication.

Because the single transmission line is not permanently driven, a weak pull-up resistor to VDDIO needs to be connected either chip internally or externally. This keeps the logic level high during idle time. The transmission pad can be configured to either push-pull or open drain mode, depending on the upper level protocol and transmission speed. The output driver of the pad is only enabled during transmission.

The signal input of the pad needs to be permanently enabled.

While LIN protocol is enabled, the receiving path is also enabled during transmission.

The following figure illustrates the half-duplex mode.

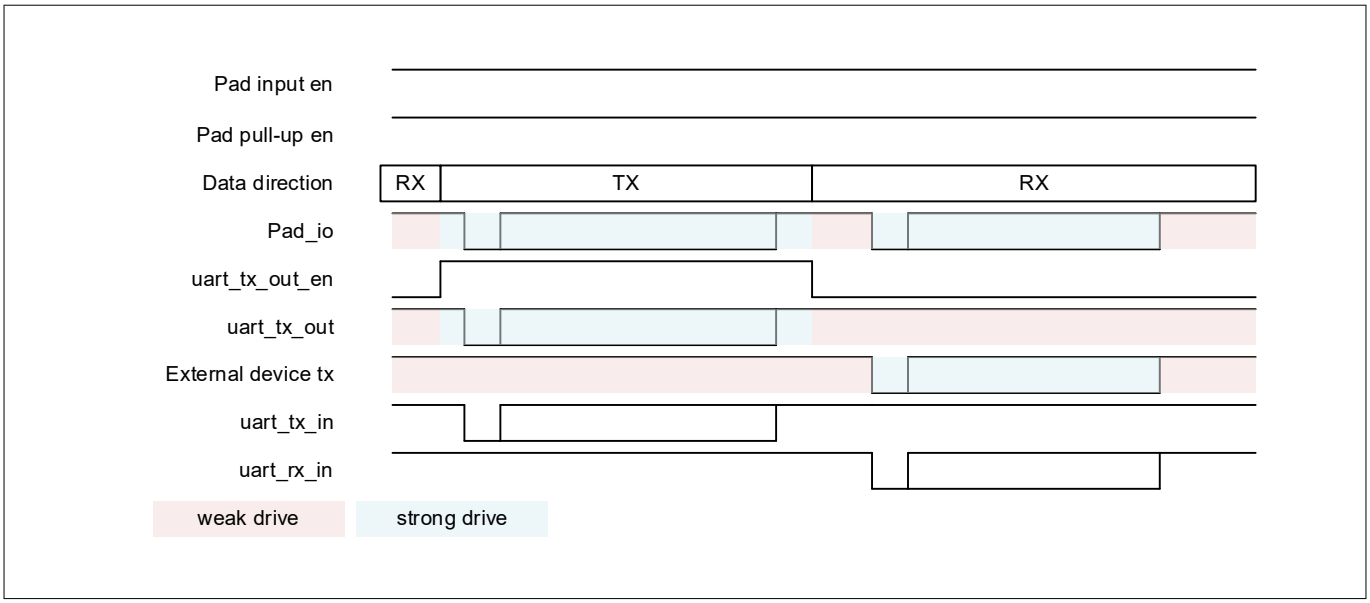


Figure 311 UART Standard Protocol in Half-duplex mode

27.4.3.1.4 Break detection

The UART mode (SCBx_CTRL.MODE is UART) and standard UART sub mode (SCBx_UART_CTRL.MODE is UART_STD) support break detection. Break detection detects when UART_RX is low ’0’ for more than SCBx_UART_CTRL.BREAK_WIDTH bit periods. The break width should be larger than the maximum number of low/’0’ bit periods in a regular data transfer, plus an additional 1-bit period. The additional 1-bit period is a minimum requirement and preferably should be larger. The additional bit period(s) exist for clock inaccuracies between transmitter and receiver.

For example, for an 8-bit data frame size, with parity support, the maximum number of low/’0’ bit periods are 10 (START bit, 8 ‘0’ data frame bits, and one ‘0’ parity bit). Therefore, the break width should be larger than 10 + 1 = 11 (SCBx_UART_CTRL.BREAK_WIDTH can be set to 11).

- Break detection only applies to RX functionality. A UART transmitter can generate a break by temporarily increasing SCBx_TX_CTRL.DATA_WIDTH and transmitting an all “zeroes data” frame. A break is used by the transmitter to signal a special condition to the receiver. This condition may result in a reset, shut down, or initialization sequence at the receiver
- Break detection is part of the LIN protocol
- When a break is detected, the SCBx_INTR_RX.BREAK_DETECT interrupt cause is set to ‘1’

Figure 312 illustrates a regular data frame and break frame (8 bits data frame, parity support, and a break width of 12-bit periods).

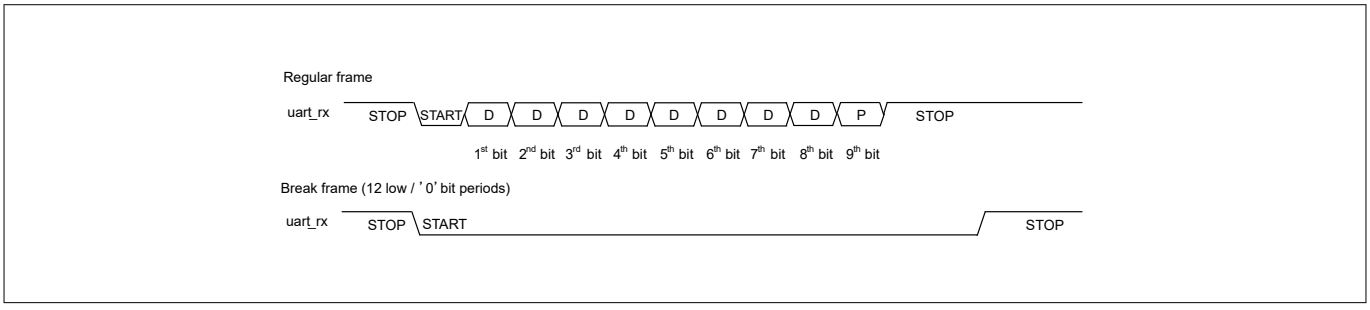


Figure 312 UART – Regular Frame and Break Frame

When SCBx_UART_RX_CTRL.BREAK_LEVEL is set to 1, idle line detection can be performed. For example, after successive transfer of several UART data frames, idle (high) level longer than normal data frame length (start+8 data+1 parity +1 stop) means the end of this successive transfer.

27.4.3.1.5 UART flow control

The standard UART mode supports flow control. Modem flow control controls the pace at which the transmitter transfers data to the receiver. Flow control only applies to TX functionality. Modem flow control is enabled through the SCBx_UART_FLOW_CTRL.CTS_ENABLED register field. When this field is '0', the transmitter transfers data when its TX FIFO is not empty. When '1', the transmitter transfers data when the UART CTS line is active and its TX FIFO is not empty.

- UART RTS (uart_rts_out): This is an output signal from the receiver. When active, it indicates that the receiver is ready to receive data (RTS: Request to Send)
- UART CTS (uart_cts_in): This is an input signal to the transmitter. When active, it indicates that the transmitter can transfer data (CTS: Clear to Send)

The receiver's uart_rts_out signal is connected to the transmitter's uart_cts_in signal. The receiver's uart_rts_out signal is derived by comparing the number of used receive FIFO entries with the SCBx_UART_FLOW_CTRL.TRIGGER_LEVEL field. If the number of used receive FIFO entries are less than SCBx_UART_FLOW_CTRL.TRIGGER_LEVEL, uart_rts_out is activated.

Typically, the UART side-band signals are active low. However, sometimes active high signaling is used.

Therefore, the polarity of the side-band signals can be controlled using bitfields SCBx_UART_FLOW_CTRL.RTS_POLARITY and SCBx_UART_FLOW_CTRL.CTS_POLARITY. Figure 313 gives an overview of the flow control functionality.

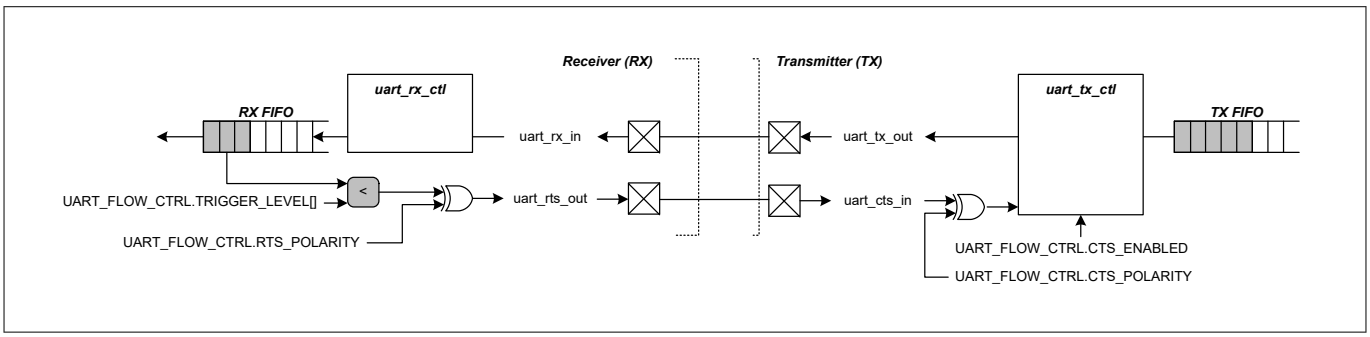


Figure 313 UART flow control connection

27.4.3.1.6 Multi-processor mode

The UART_MP (multi-processor) mode is defined with single-master-multi-slave topology, as Figure 314 shows. This mode is also known as the UART 9-bit protocol because the data field is 9 bits wide. UART_MP is part of Standard UART mode.

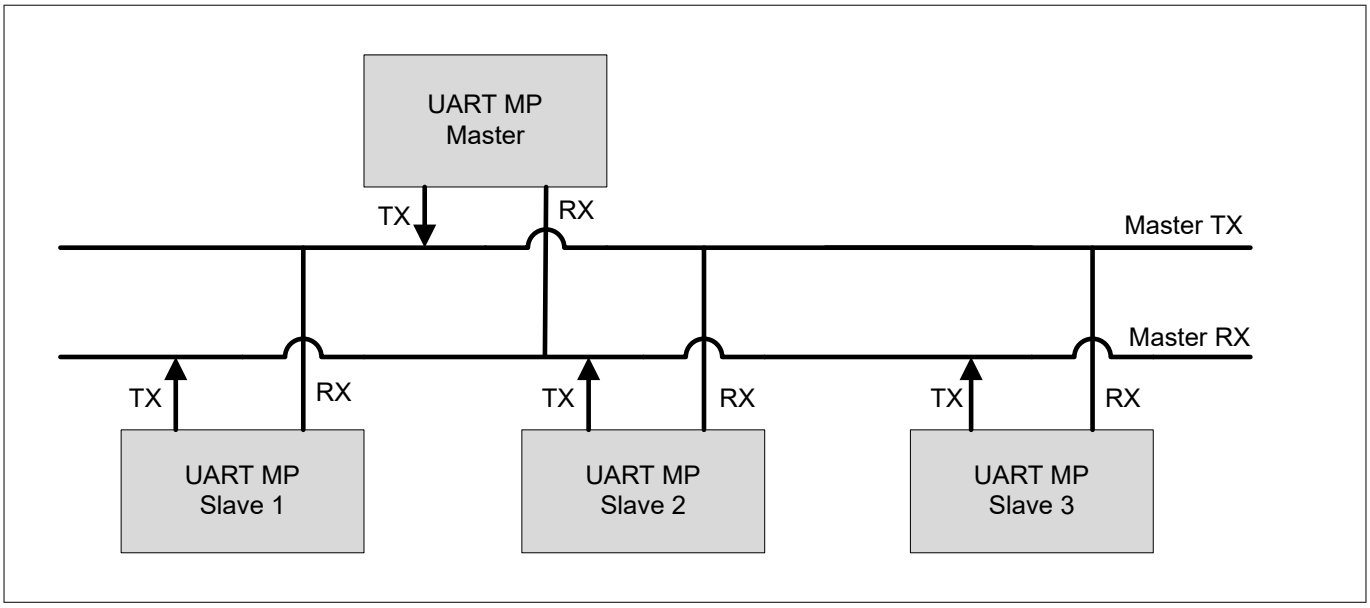


Figure 314 UART MP mode bus connections

The main properties of the UART_MP mode are:

- Single master with multiple slave concept (multi-drop network)
- Each slave is identified by a unique address
- Using 9-bit data field, with the ninth bit as address/data flag (MP bit). When set high, it indicates an address byte; when set low it indicates a data byte. A data frame is illustrated in Figure 315
- Parity bit is disabled

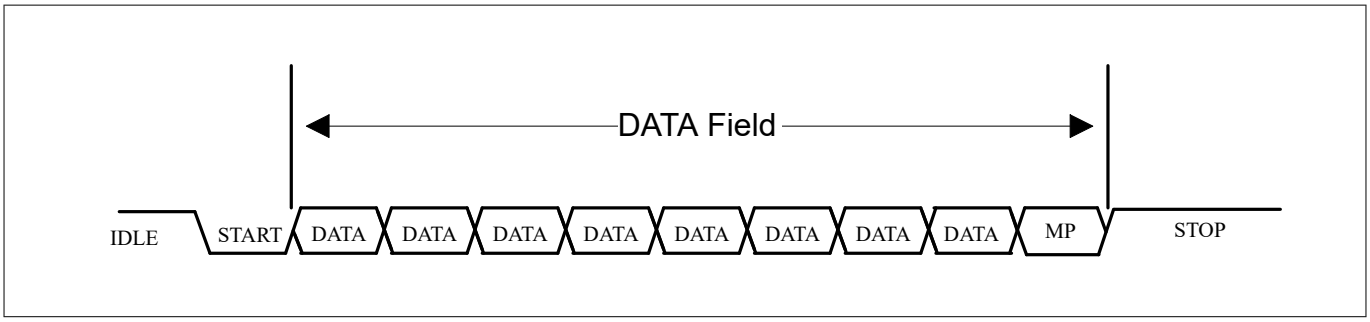


Figure 315 UART MP address and data frame

The SCB can be used as either master or slave device in UART_MP mode. Both SCBx_TX_CTRL and SCBx_RX_CTRL registers should be set to 9-bit data frame size. When the SCB works as a UART_MP master device, the firmware changes the MP flag for every address or data frame. When it works as a UART_MP slave device, the MP_MODE field of the SCBx_UART_RX_CTRL register should be set to '1'. The SCBx_RX_MATCH register should be set for the slave address and address mask. The matched address is written in the RX_FIFO when the ADDR_ACCEPT field of the SCBx_CTRL register is set to '1'. If the received address does not match its own address, then the interface ignores the following data, until the next address is received for compare.

Configuring the SCB as a Standard UART Interface

To configure the SCB as a standard UART interface, set various register bits in the following order:

1. Configure the SCB as UART interface by writing '10b' to the MODE field (bits [25:24]) of the SCBx_CTRL register
2. Configure the UART interface to operate as a Standard protocol by writing '00' to the MODE field (bits [25:24]) of the SCBx_UART_CTRL register

3. To enable the UART MP Mode or UART LIN Mode, write '1' to the MP_MODE (bit 10) or LIN_MODE (bit 12) respectively of the SCBx_UART_RX_CTRL register
4. Follow the steps 2 to 4 described in Enabling and initializing UART

27.4.3.2 SmartCard (ISO 7816)

ISO7816 is an asynchronous serial interface, defined with single-master-single slave topology. ISO7816 defines both Reader (master) and Card (slave) functionality. For more information, refer to the ISO7816 Specification.

Both master (reader) and slave (card) functions are supported by the SCB. This block provides the basic physical layer support with asynchronous character transmission. The UART_TX line is connected to SmartCard I/O line by internally multiplexing between UART_TX and UART_RX control modules.

The SmartCard transfer is similar to a UART transfer, with the addition of a negative acknowledgment (NACK) that may be sent from the receiver to the transmitter. A NACK is always '0'. Both master and slave may drive the same line, although never at the same time.

In a SmartCard transfer, the transmitter drives the start bit and data bits (and optionally a parity bit). After these bits, it enters its stop period by releasing the bus, which results in the line being '1' (the value of a stop bit). After the half-bit transfer period into the stop period, the receiver may drive a NACK on the line (a value of '0') for a one to two bit transfer period. This NACK is observed by the transmitter, which reacts by extending its stop period by a one-bit transfer period. For this protocol to work, the stop period should be at least two bits of transfer period. Note that a data transfer with a NACK takes one-bit transfer period longer than a data transfer without a NACK. Typically, implementations use a tri-state driver with a pull-up resistor, such that when the line is not transmitting data or transmitting the Stop bit, its value is '1'.

Figure 316 illustrates the SmartCard protocol.

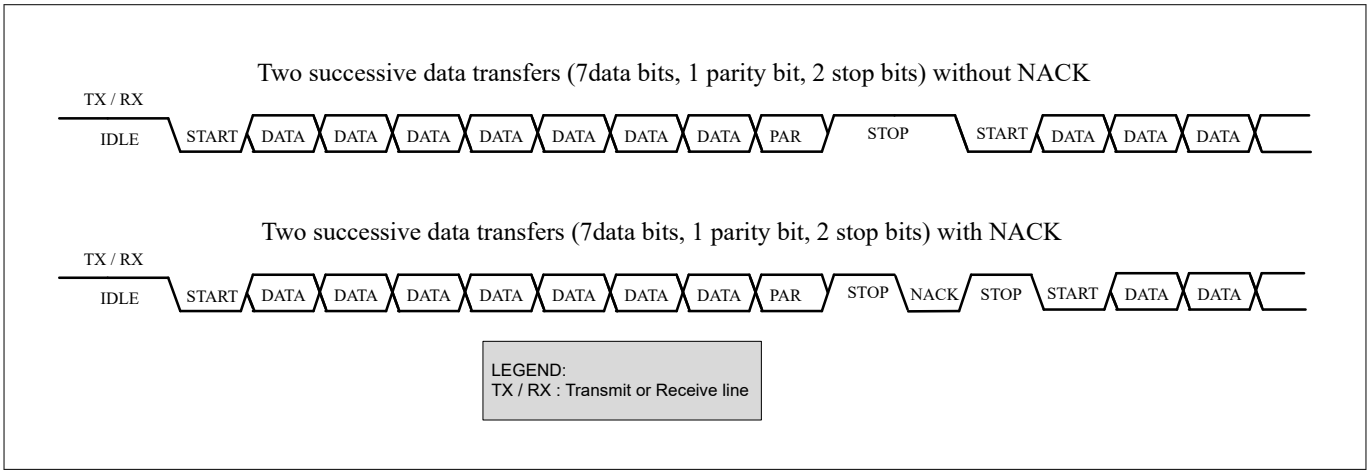


Figure 316 SmartCard example

The communication baud rate while using the SmartCard is given as:

Baud rate = Fscb_clk/Oversample

Where Fscb_clk is the "scb_clk" clock frequency and Oversample is the oversample factor.

Configuring SCB as UART SmartCard interface

To configure the SCB as a UART SmartCard interface, set various register bits in the following order. For more information on these registers, see the UART registers.

1. Configure the SCB as UART interface by writing '10' to the MODE (bits [25:24]) of the SCB_CTRL register
2. Configure the UART interface to operate as a SmartCard protocol by writing '01' to the MODE (bits [25:24]) of the SCB_UART_CTRL register
3. Follow steps 2 to 4 described in Enabling and initializing UART

27.4.3.3 IrDA

The SCB supports the IrDA protocol for data rates of up to 115.2 kbps using the UART interface. It supports only the basic physical layer of the IrDA protocol with rates less than 115.2 kbps. Hence, the system instantiating this block must consider how to implement a complete IrDA communication system with other available system resources.

The IrDA protocol adds a modulation scheme to UART signaling. At the transmitter, bits are modulated, and at the receiver, bits are demodulated. The modulation scheme uses a Return-to-Zero-Inverted (RZI) format. A bit value of '0' is signaled by a short '1' pulse on the line and a bit value of '1' is signaled by holding the line to '0'. For these data rates (<=115.2 kbps), the RZI modulation scheme is used and the pulse duration is 3/16 of the bit period. The sampling clock frequency should be set to 16 times the selected baud rate, by configuring the SCB_OVS field of the SCB_CTRL register. In addition, the PSC3M/P6 MCU supports a low-power IrDA receiver mode, which allows it to detect pulses with a minimum width of 1.41 µs.

Bit rates less than 115.2 kbps can be achieved by configuring the clk_scb frequency. Additional allowable rates are 2.4 kbps, 9.6 kbps, 19.2 kbps, 38.4 kbps, and 57.6 kbps. Figure 317 shows how a UART transfer is IrDA modulated.

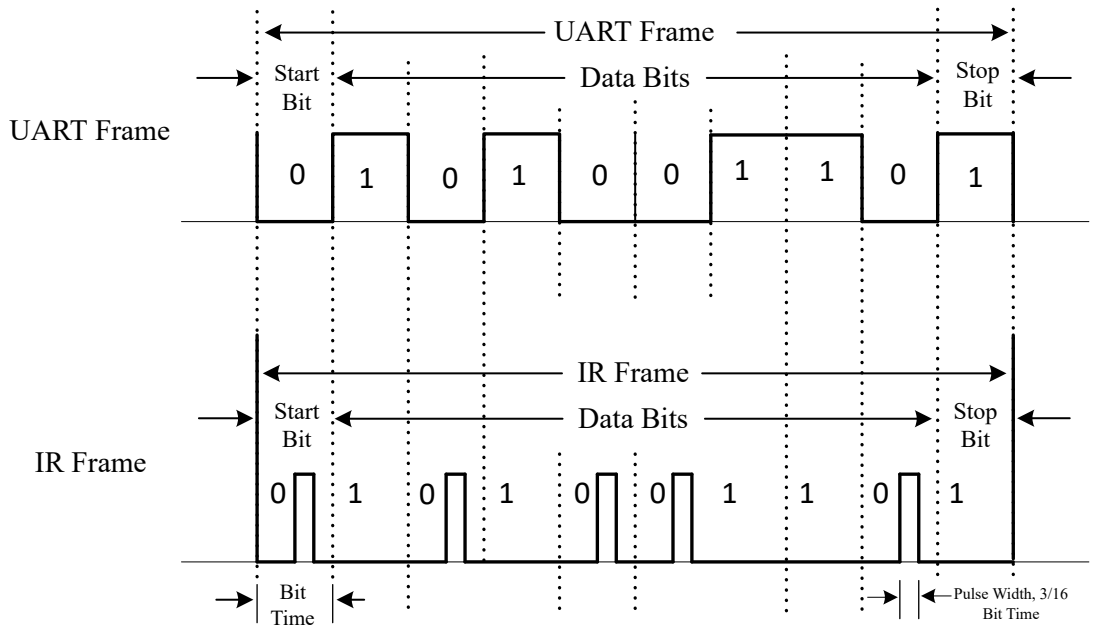


Figure 317 IrDA example

Configuring the SCB as a UART IrDA Interface

To configure the SCB as a UART IrDA interface, set various register bits in the following order. For more information on these registers, see the UART registers.

1. Configure the SCB as a UART interface by writing '10b' to the MODE (bits [25:24]) of the SCB_CTRL register
2. Configure the UART interface to operate as an IrDA protocol by writing '10b' to the MODE (bits [25:24]) of the SCB_UART_CTRL register
3. Enable the Median filter on the input interface line by writing '1' to MEDIAN (bit 9) of the SCB_RX_CTRL register
4. Configure the SCB as described in Enabling and initializing UART

27.4.3.4 UART local interconnect network (LIN) mode

The LIN protocol is supported by the SCB as part of the standard UART. LIN is designed with a single-master-multi-slave topology. There is one master node and multiple slave nodes on the LIN bus. The SCB UART supports only LIN slave functionality. The LIN specification defines both physical layer (layer 1) and data link layer (layer 2). Figure 318 illustrates the UART_LIN and LIN transceiver.

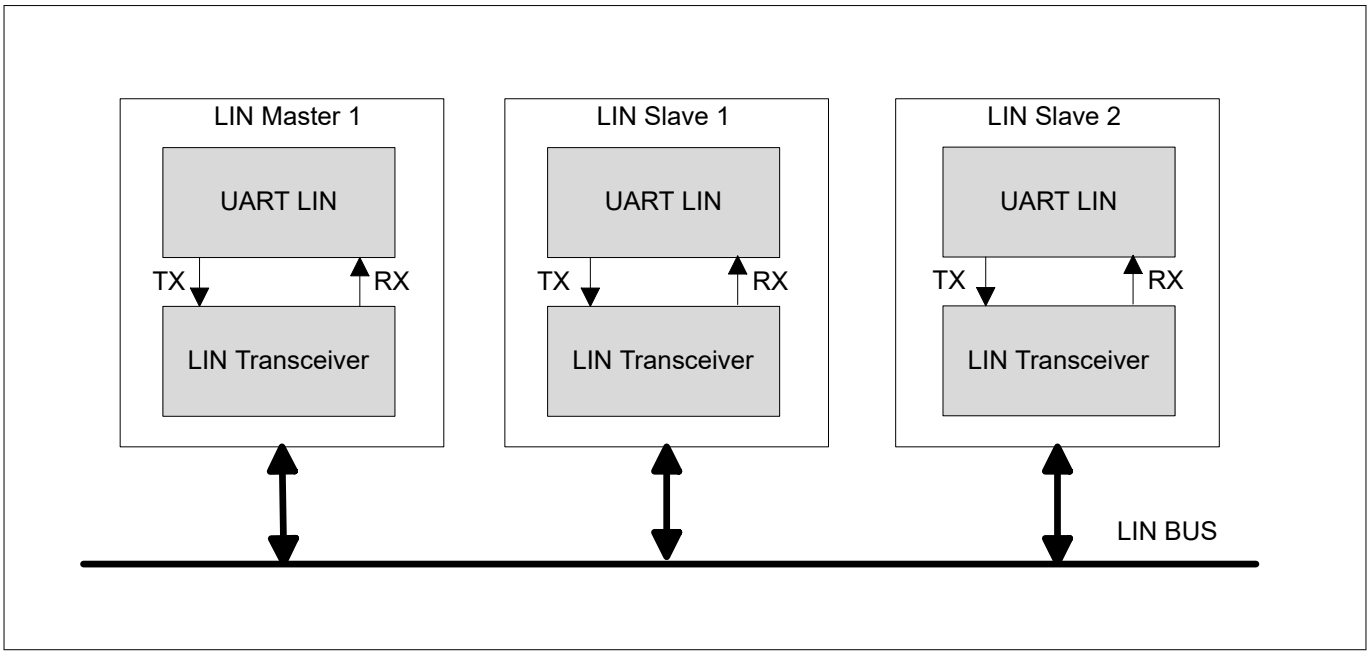


Figure 318 UART_LIN and LIN transceiver

LIN protocol defines two tasks:

- Master task: This task involves sending a header packet to initiate a LIN transfer
- Slave task: This task involves transmitting or receiving a response

The master node supports master task and slave task; the slave node supports only slave task, as shown in Figure 319.

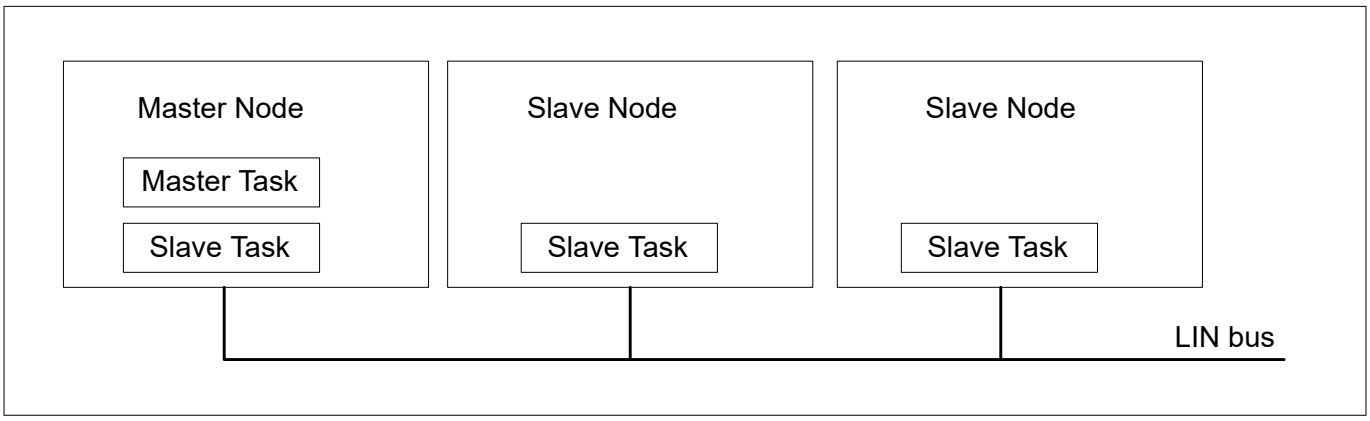


Figure 319 LIN bus nodes and tasks

LIN frame structure

LIN is based on the transmission of frames at predetermined moments of time. A frame is divided into header and response fields, as shown in Figure 320.

- The header field consists of:
  - Break field (at least 13-bit periods with the value ‘0’)

- Sync field (a 0x55 byte frame). A sync field can be used to synchronize the clock of the slave task with that of the master task
  - Identifier field (a frame specifying a specific slave)
- The response field consists of data and checksum

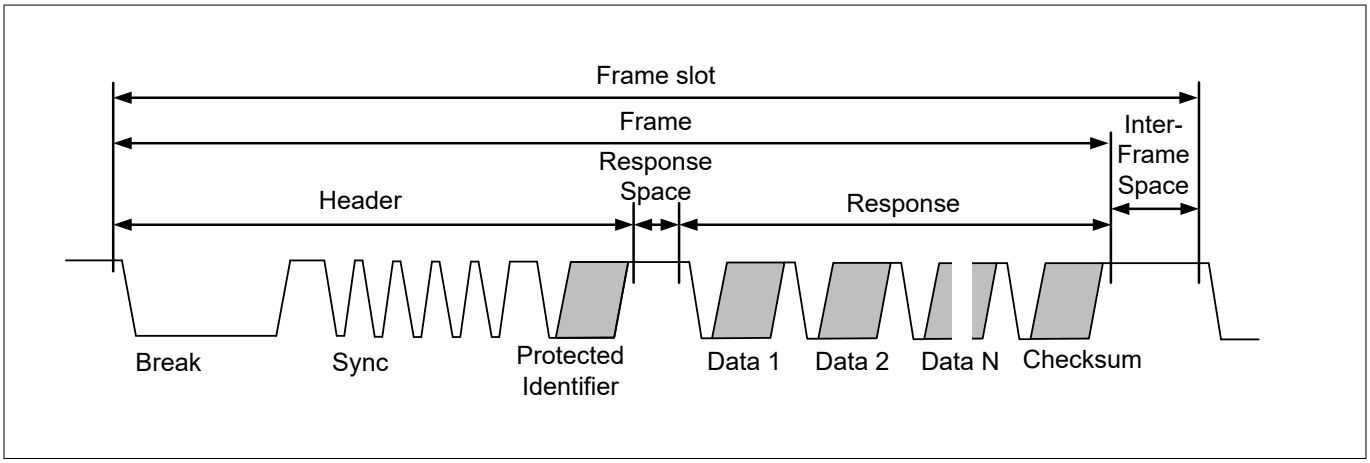


Figure 320 LIN frame structure

In LIN protocol communication, the least significant bit (LSb) of the data is sent first and the most significant bit (MSb) last. The start bit is encoded as zero and the stop bit is encoded as one. The following sections describe all the byte fields in the LIN frame.

Break field

Every new frame starts with a break field, which is always generated by the master. The break field has a logical zero with a minimum of 13-bit times and followed by a break delimiter. The break field structure is as shown in the Figure 321.

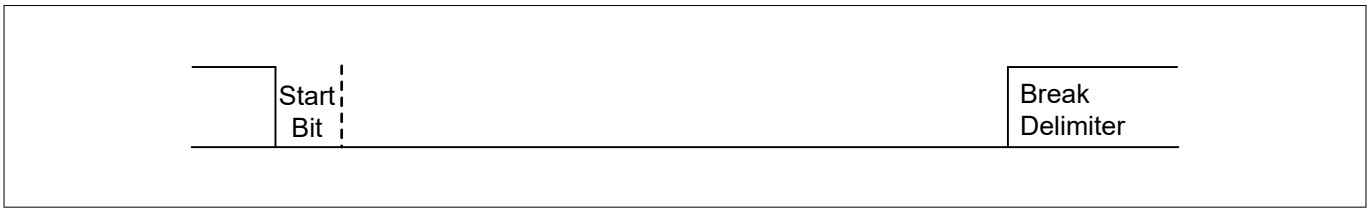


Figure 321 LIN break field

Sync field

This is the second field transmitted by the master in the header field; its value is 0x55. A sync field can be used to synchronize the clock of the slave task with that of the master task for automatic baud rate detection. Figure 322 shows the LIN sync field structure.

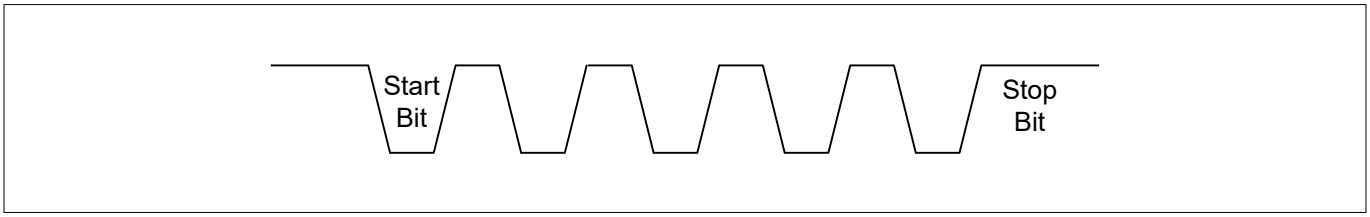


Figure 322 LIN sync field

Protected Identifier (PID) field

A protected identifier field consists of two subfields: the frame identifier (bits 0-5) and the parity (bit 6 and bit 7). The PID field structure is shown in Figure 323.

- Frame identifier: The frame identifiers are divided into three categories -
  - Values 0 to 59 (0x3B) are used for signal carrying frames
  - 60 (0x3C) and 61 (0x3D) are used to carry diagnostic and configuration data
  - 62 (0x3E) and 63 (0x3F) are reserved for future protocol enhancements
- Parity: Frame identifier bits are used to calculate the parity

***Note**: The parity calculation of PID (protected identifier field) will also be handled by a firmware and the SCB block's UART will take the PID as the common data byte.*

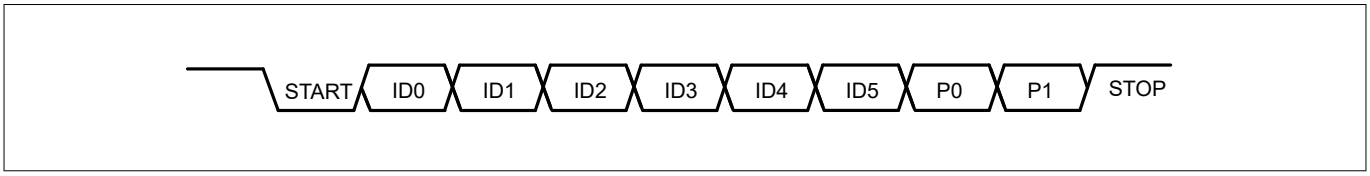


Figure 323 PID field

Data

In LIN, every frame can carry a minimum of 1 byte and a maximum of 8 bytes of data. Here, the LSb of the data byte is sent first and the MSb of the data byte is sent last.

Checksum

The checksum is the last byte field in the LIN frame. It is calculated by inverting the 8-bit sum along with a carryover of all data bytes only or the 8-bit sum with the carryover of all data bytes and the PID field. There are two types of checksums in LIN frames:

- Classic checksum: Calculated over all the data bytes only (used in LIN 1.x slaves)
- Enhanced checksum: Calculated over all the data bytes along with the protected identifier (used in LIN 2.x slaves)

***Note**: The checksum calculation is handled by the firmware, and the SCB block's UART takes it as a common data byte.*

27.4.4 Clocking and oversampling

The UART protocol is implemented using clk_scb as an oversampled multiple of the baud rate. For example, to implement a 100-kHz UART, clk_scb could be set to 1 MHz and the oversample factor set to '10'. The oversampling is set using the SCB_CTRL.OVS register field. The oversampling value is SCB_CTRL.OVS + 1. In the UART standard submode (including LIN) and the SmartCard submode, the valid range for the OVS field is [7,15].

In the UART transmit IrDA submode, this field indirectly specifies the oversampling. Oversampling determines the interface clock per bit cycle and the width of the pulse. This submode has only one valid OVS value–16 (which is a value of '0' in the OVS field of the SCB_CTRL register); the pulse width is roughly 3/16 of the bit period (for all bit rates).

In the UART receive IrDA submode (1.2, 2.4, 9.6, 19.2, 38.4, 57.6, and 115.2 kbps), this field indirectly specifies the oversampling. In normal transmission mode, this pulse is approximately 3/16 of the bit period (for all bit rates). In low-power transmission mode, this pulse is potentially smaller (down to 1.62 µs typical and 1.41 µs minimal) than 3/16 of the bit period (for less than 115.2 kbps bit rates). Pulse widths greater than or equal to two SCB input clock cycles are guaranteed to be detected by the receiver.

Pulse widths less than two clock cycles and greater than or equal to one SCB input clock cycle may be detected by the receiver. Pulse widths less than one SCB input clock cycle will not be detected by the receiver. Note that the SCB_RX_CTRL.MEDIAN should be set to '1' for IrDA receiver functionality.

The SCB input clock and the oversampling together determine the IrDA bit rate. See the registers list for more details on the OVS values for different baud rates.

27.4.5 Enabling and initializing UART

Do the following to program the UART:

1. Program protocol-specific information using the UART_TX_CTRL, UART_RX_CTRL, and UART_FLOW_CTRL registers. This includes selecting the submodes of the protocol, transmitter-receiver functionality, and so on
2. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL
   a. Specify the data frame width
   b. Specify whether MSb or LSb is the first bit to be transmitted or received
3. Program the transmitter and receiver FIFOs using the SCB_TX_FIFO_CTRL and SCB_RX_FIFO_CTRL registers, respectively.
   a. Set the trigger level
   b. Clear the transmitter and receiver FIFO and Shift registers
4. Enable the block (write a '1' to the ENABLE bit of the SCB_CTRL register). After the block is enabled, do not change the control bits. Changes should be made after disabling the block; for example, to modify the operation mode (from SmartCard to IrDA). The change takes effect only after the block is reenabled

Note that reenabling the block causes reinitialization and the associated state is lost (for example, FIFO content).

27.4.6 I/O pad connection

The following sections explains the I/O pad connection of UART

27.4.6.1 UART mode

27.4.6.1.1 Standard UART mode

Figure 324, Figure 325, Figure 326 and Table 282 list the use of the I/O pads for the Standard UART mode.

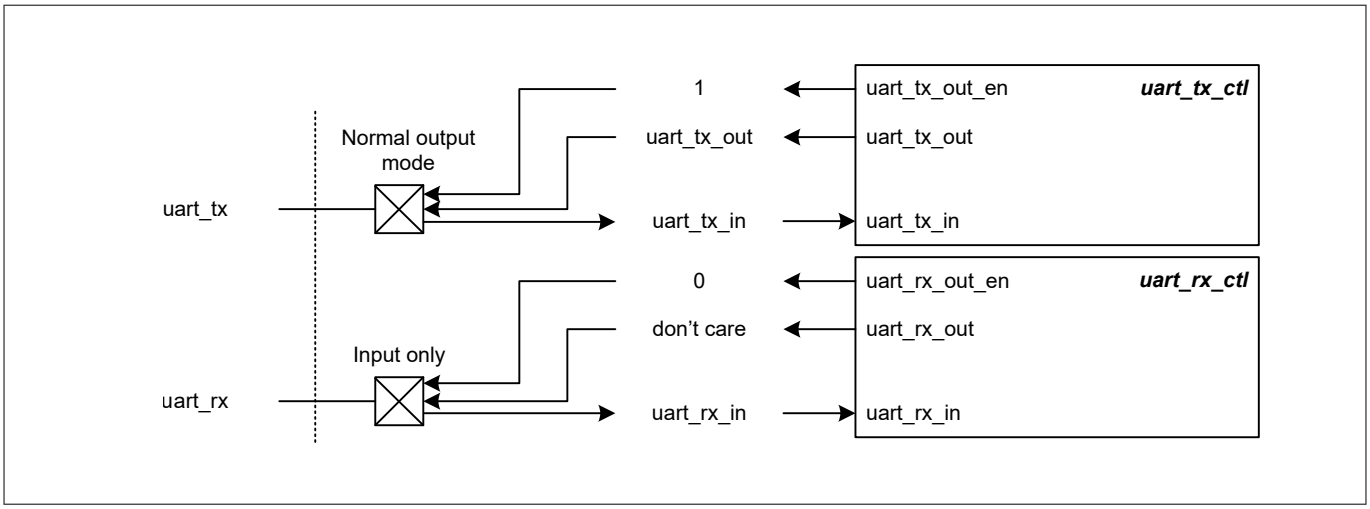


Figure 324 Standard UART mode I/O pad connections

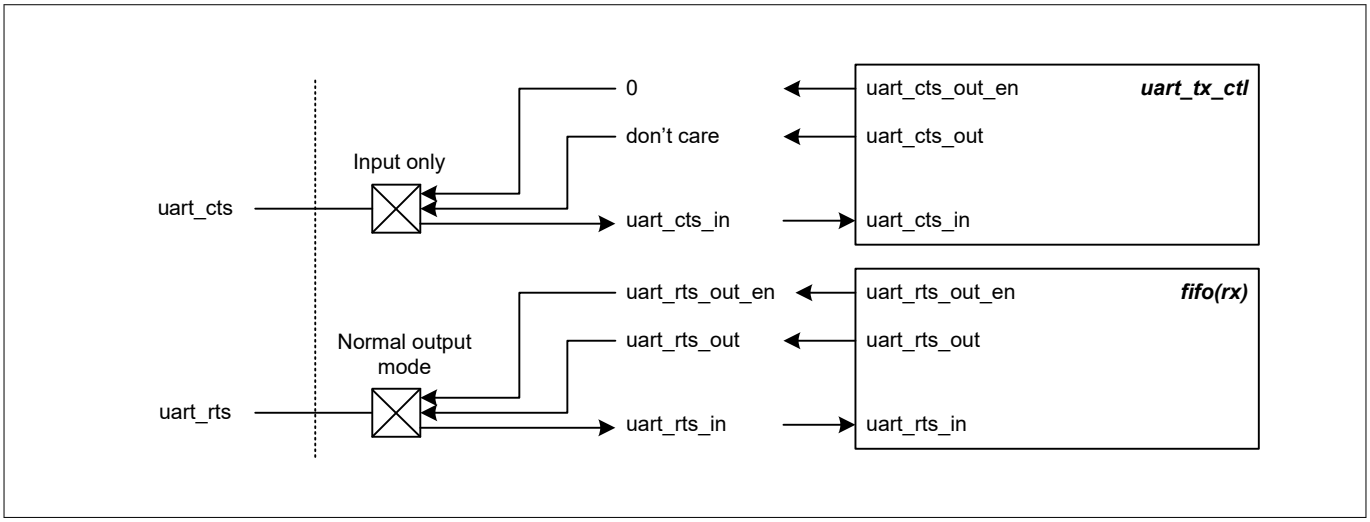


Figure 325 Standard UART mode, flow control I/O pad connections

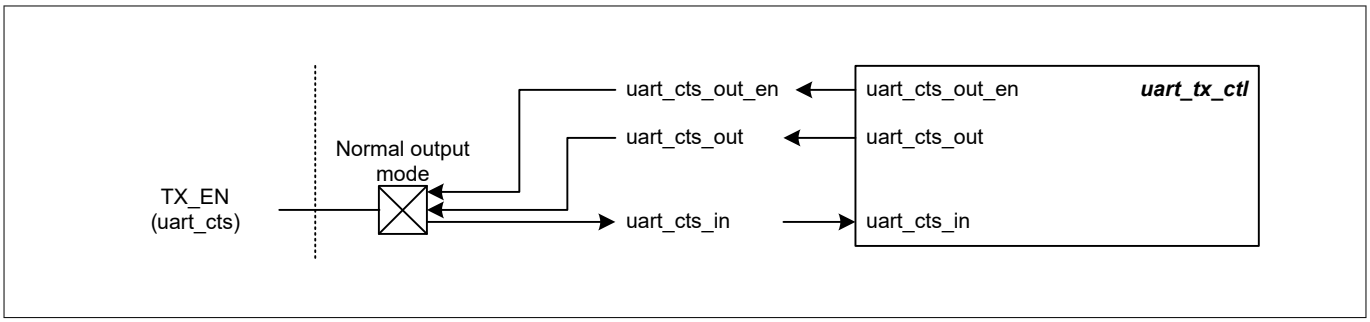


Figure 326 Standard UART mode, CTS reused as TX_EN for RS485

Table 282 UART I/O pad connection usage

I/O pads	Drive mode	On-chip I/O signals	Usage
uart_tx	Normal output mode	uart_tx_out_en uart_tx_out	Transmit a data element

(table continues...)

Table 282 (continued) UART I/O pad connection usage

I/O pads	Drive mode	On-chip I/O signals	Usage
uart_rx	Input only	uart_rx_in	Receive a data element
uart_cts	Input only	uart_cts_in	Indicate peer part readiness to receive data
uart_rts	Normal output mode	uart_rts_out_en uart_rts_out	Indicate SCB readiness to receive data
TX_EN(uart_cts)	Normal output mode	uart_cts_out_en uart_cts_out	Indicate SCB readiness to transmit data

27.4.6.1.2 Half-duplex mode

For half-duplex mode the transmitting and receiving pad is identical. The arbitration who is driving the line needs to be done on protocol level. An internal or external pull-up is required for half-duplex operation. Figure 327 and Table 283 list the use of the I/O pads for the half-duplex mode.

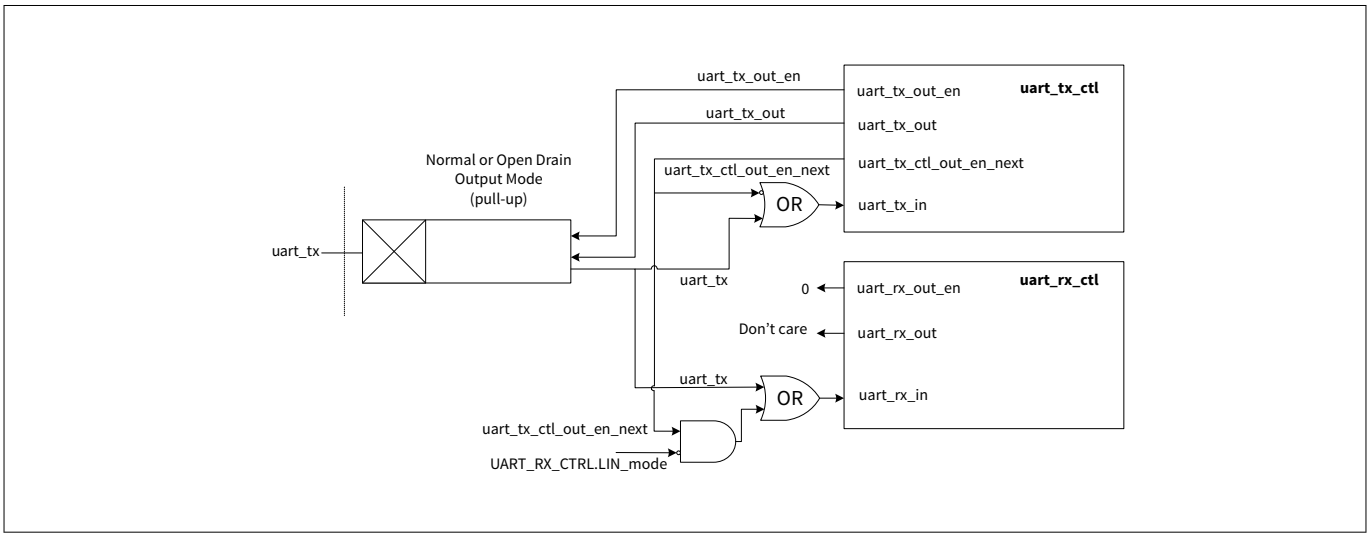


Figure 327 Half-duplex mode I/O pad connections

Table 283 Half-duplex mode I/O pad connections

I/O pads	Drive mode	On-chip I/O signals	Usage
uart_tx	Push-Pull Open-drain Input enabled	uart_tx_out_en uart_tx_out uart_tx_in	Transmit and receive a data element. While uart_tx_out_en is high (transmission), the uart_rx_in path is blocked and during receiving data the uart_tx_in path is blocked. Both are controlled by uart_tx_out_en.

27.4.6.2 SmartCard mode

Figure 328 and Table 284 list the use of the I/O pads for the SmartCard mode.

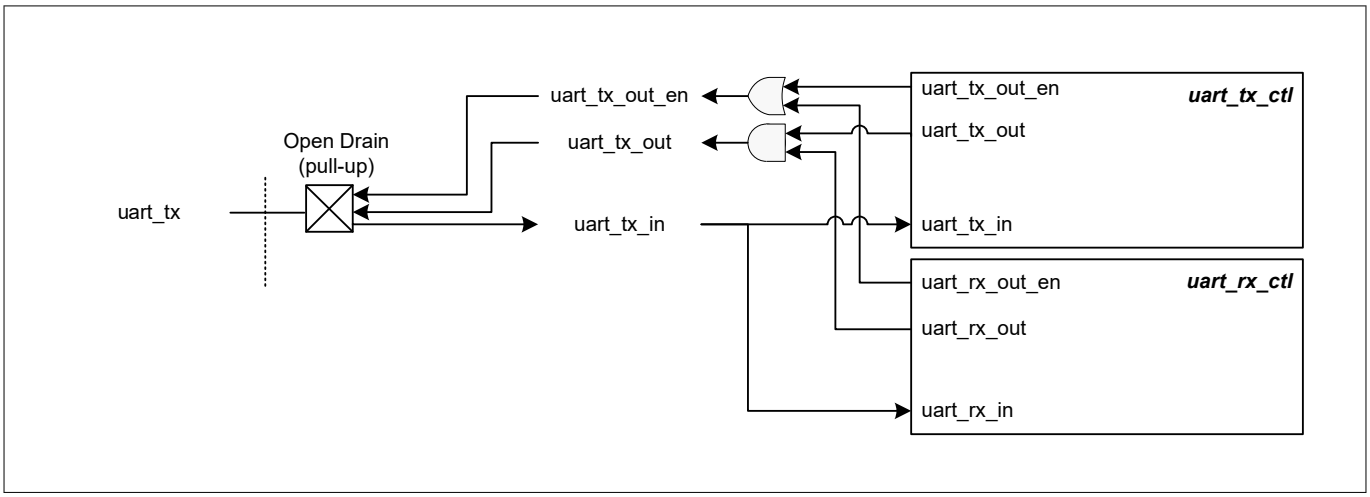


Figure 328 SmartCard mode I/O pad connections

Table 284 SmartCard mode I/O pad connections

I/O pads	Drive mode	On-chip I/O signals	Usage
uart_tx	Open drain with pull-up	uart_tx_in	Used to receive a data element. Receive a negative acknowledgment of a transmitted data element
		uart_tx_out_en uart_tx_out	Transmit a data element. Transmit a negative acknowledgment to a received data element.

27.4.6.3 IrDA mode

Figure 329 and Table 285 list the use of the I/O pads for IrDA mode.

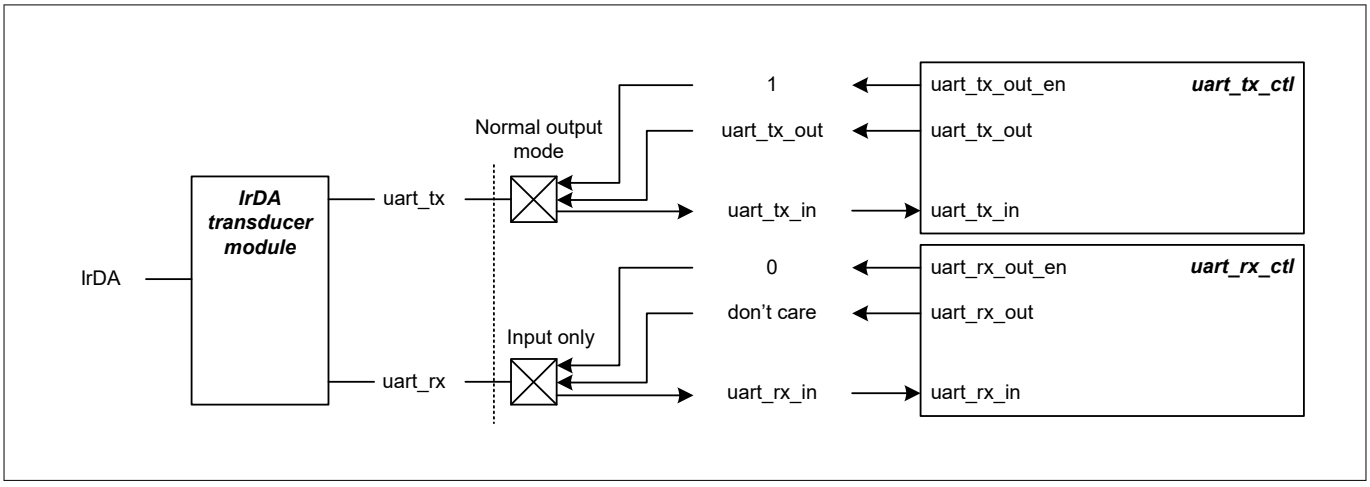


Figure 329 IrDA mode I/O pad connections

Table 285 IrDA mode I/O pad connections

I/O pads	Drive mode	On-chip I/O signals	Usage
uart_tx	Normal output mode	uart_tx_out_en uart_tx_out	Transmit a data element
uart_rx	Input only	uart_rx_in	Receive a data element

***Note**: **IrDA invalid use case scenario:** A glitch is observed on the TX pad when SCB is configured as master using IrDA mode and if the TX pad is '0', SCB is enabled and uart_tx_out_en = 1, and uart_tx_out = 0. The default value of the TX pad is '0', that means, before SCB is enabled, TX pad is '0'. On the pad there is a delay on the output signal which is set when the enable signal is set, thereby triggering a small glitch of around 400 ps. This behavior is caused by delays between pad enable and output signal.*

27.4.6.4 LIN mode

Figure 330 and Table 286 list the use of the I/O pads for LIN mode.

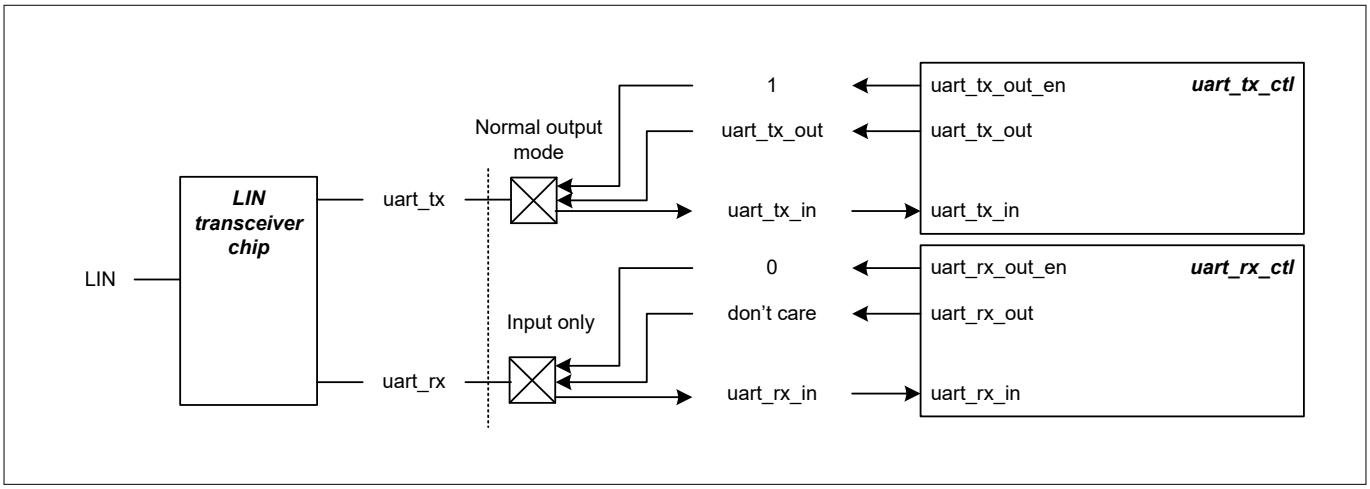


Figure 330 LIN mode I/O pad connections

Table 286 LIN mode I/O pad connections

I/O pads	Drive mode	On-chip I/O signals	Usage
uart_tx	Normal output mode	uart_tx_out_en uart_tx_out	Transmit a data element
uart_rx	Input only	uart_rx_in	Receive a data element

27.4.7 Registers summary

The UART interface is controlled using a set of 32-bit registers. For detailed information on these registers, see the PSC3M/P6 MCU registers reference manual.

27.5 Inter-Integrated Circuit (I2C)

This section explains the I2C implementation in the PSC3M/P6 MCUs. For more information on the I2C protocol specification, see the I2C-bus specification available on NXP's Website. In the PSC3M/P6 MCUs, all SCB blocks

support both I2C master and slave modes; only one SCB (SCB[0]) is available in Deep Sleep power mode and allows externally-clocked operations.

27.5.1 Features

This block supports the following features:

- Master, slave and master-slave functionalities
- Standard mode (100 kHz), fast mode (400 kHz) and fast plus mode (1 MHz)
- Standard 7-bit slave addressing
- 10-bit addressing supported through software
- Multi-master modes and bus arbitration
- Analog glitch filte
- Clock stretching
- Collision detection
- Programmable oversampling of I2C clock signal (SCL)
- FIFO, EZ, and CMD_RESP buffer modes
- Local loopback control
- General call address detection
- Wake up from sleep modes
- Support SMBUS
- Support TGS and MSA
- Support 4-wire I2C model
  - Configurable inversion of 4-wire I2C outputs

27.5.2 General description

Figure 331 illustrates an example of an I2C communication network.

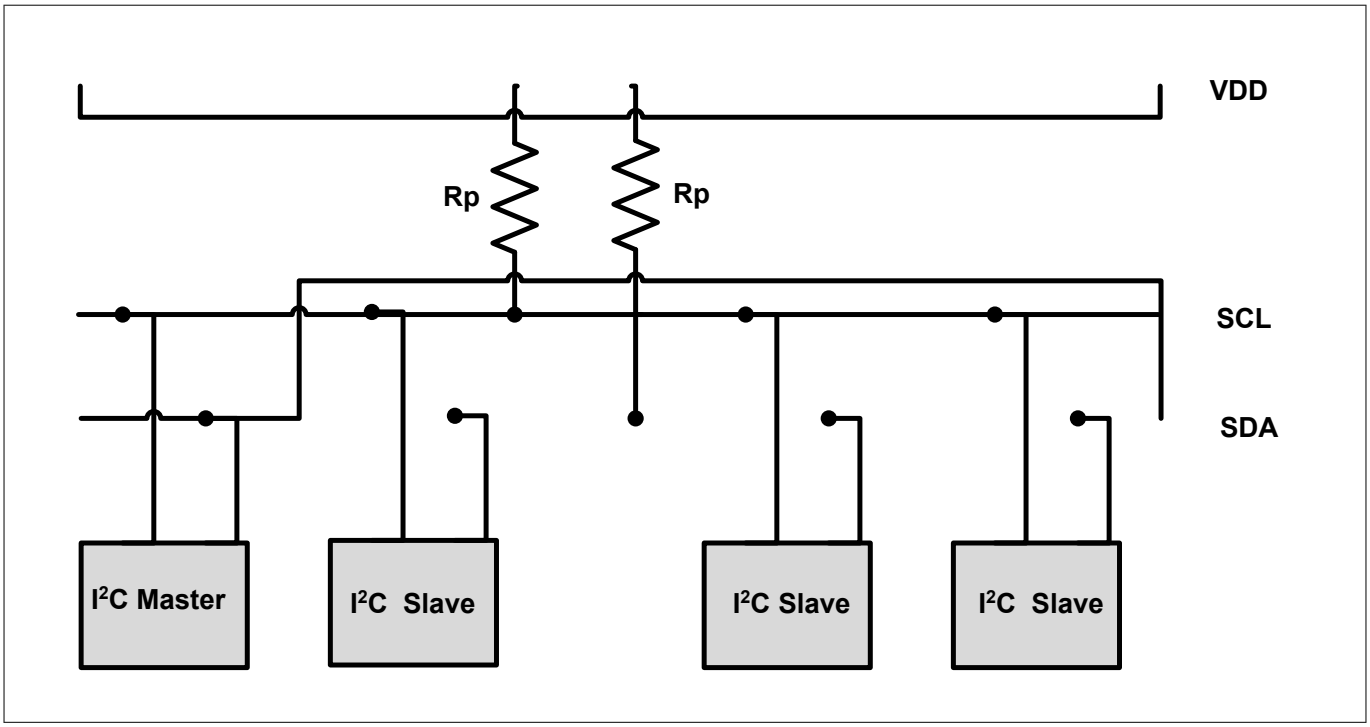


Figure 331 I2C interface block diagram

The standard I2C bus is a two wire interface with the following lines:

- Serial Data (SDA)
- Serial Clock (SCL)

I2C devices are connected to these lines using open collector or open-drain output stages, with pull-up resistors (Rp). A simple master/slave relationship exists between devices. Masters and slaves can operate as either transmitter or receiver. Each slave device connected to the bus is software addressable by a unique 7-bit address.

27.5.3 Terms and definitions

Table explains the commonly used terms in an I2C communication network.

Table 287 Definition of I2C bus terminology

Term	Description
Master	The device that initiates a transfer, generates clock signals, and terminates a transfer
Slave	The device addressed by a master
Multi-master	More than one master can attempt to control the bus at the same time
Arbitration	Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted
Synchronization	Procedure to synchronize the clock signals of two or more devices

27.5.3.1 Clock stretching

When a slave device is not yet ready to process data, it may drive a '0' on the SCL line to hold it down. Due to the implementation of the I/O signal interface, the SCL line value will be '0', independent of the values that any other master or slave may be driving on the SCL line. This is known as clock stretching and is the only situation in which a slave drives the SCL line. The master device monitors the SCL line and detects it when it cannot generate a positive clock pulse ('1') on the SCL line. It then reacts by delaying the generation of a positive edge on the SCL line, effectively synchronizing with the slave device that is stretching the clock. The SCB on the PSC3M/P6 MCU can and will stretch the clock.

27.5.3.2 Bus arbitration

The I2C protocol is a multi-master, multi-slave interface. Bus arbitration is implemented on master devices by monitoring the SDA line. Bus collisions are detected when the master observes an SDA line value that is not the same as the value it is driving on the SDA line. For example, when master 1 is driving the value '1' on the SDA line and master 2 is driving the value '0' on the SDA line, the actual line value will be '0' due to the implementation of the I/O signal interface. Master 1 detects the inconsistency and loses control of the bus. Master 2 does not detect any inconsistency and keeps control of the bus.

27.5.4 I2C modes of operation

I2C is a synchronous single-master, multi-master, multi-slave serial interface. Devices operate in either master mode, slave mode, or master/slave mode. In master/slave mode, the device switches from master to slave mode when it is addressed. Only a single master may be active during a data transfer. The active master is responsible for driving the clock on the SCL line. Table 288 illustrates the I2C modes of operation.

Table 288 I2C modes

Mode	Description
Slave	Slave only operation (default)
Master	Master only operation
Multi-master	Supports more than one master on the bus

Table 289 lists some common bus events that are part of an I2C data transfer. The Write Transfer and Read Transfer sections explain the I2C bus bit format during data transfer.

Table 289 I2C bus events terminology

Bus Event	Description
START	A HIGH to LOW transition on the SDA line while SCL is HIGH
STOP	A LOW to HIGH transition on the SDA line while SCL is HIGH
ACK	The receiver pulls the SDA line LOW and it remains LOW during the HIGH period of the clock pulse, after the transmitter transmits each byte. This indicates to the transmitter that the receiver received the byte properly.

(table continues...)

Table 289 (continued) I2C bus events terminology

Bus Event	Description
NACK	The receiver does not pull the SDA line LOW and it remains HIGH during the HIGH period of clock pulse after the transmitter transmits each byte. This indicates to the transmitter that the receiver received the byte unsuccessfully.
Repeated START	START condition generated by master at the end of a transfer instead of a STOP condition
DATA	SDA status change while SCL is LOW (data changing), and no change while SCL is HIGH (data valid)

With all of these modes, there are two types of transfer - read and write. In write transfer, the master sends data to slave; in read transfer, the master receives data from the slave.

27.5.4.1 Write transfer

- A typical write transfer begins with the master generating a START condition on the I2C. The master then writes a 7-bit I2C slave address and a write indicator (‘0’) after the START condition. The addressed slave transmits an acknowledgment byte by pulling the data line low during the ninth bit time
- If the slave address does not match any of the slave devices or if the addressed device does not want to acknowledge the request, it transmits a no acknowledgment (NACK) by not pulling the SDA line The absence of an acknowledgment, results in an SDA line value of ‘1’ due to the pull-up resistor implementation
- If no acknowledgment is transmitted by the slave, the master may end the write transfer with a STOP The master can also generate a repeated START condition for a retry attempt
- The master may transmit data to the bus if it receives an acknowledgment. The addressed slave transmits an acknowledgment to confirm the receipt of every byte of data written. Upon receipt of this acknowledgment, the master may transmit another data byte
- When the transfer is complete, the master generates a STOP condition

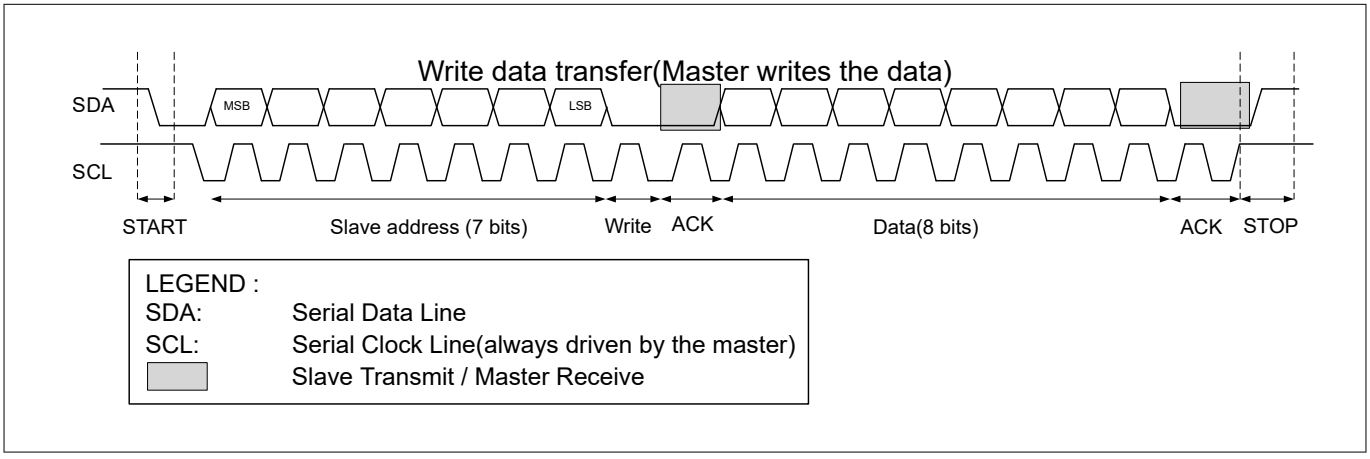


Figure 332 Master write data transfer

27.5.4.2 Read transfer

- A typical read transfer begins with the master generating a START condition on the I2C. The master then writes a 7-bit I2C slave address and a read indicator (‘1’) after the START condition. The addressed slave transmits an acknowledgment by pulling the data line low during the ninth bit time
- If the slave address does not match with that of the connected slave device or if the addressed device does not want to acknowledge the request, a no acknowledgment (NACK) is transmitted by not pulling the SDA line low. The absence of an acknowledgment results in an SDA line value of ‘1’ due to the pull-up resistor implementation
- If no acknowledgment is transmitted by the slave, the master may end the read transfer with a STOP event. The master can also generate a repeated START condition for a retry attempt
- If the slave acknowledges the address, it starts transmitting data after the acknowledgment signal. The master transmits an acknowledgment to confirm the receipt of each data byte sent by the slave. Upon receipt of this acknowledgment, the addressed slave may transmit another data byte
- The master can send a NACK signal to the slave to stop the slave from sending data bytes. This completes the read transfer
- When the transfer is complete, the master generates a STOP condition
- Individual data transfers (of one or more data elements) start with a START event and end with a STOP. Combined data transfers consist of multiple individual transfers that are not separated by STOP events, but by repeated START events only

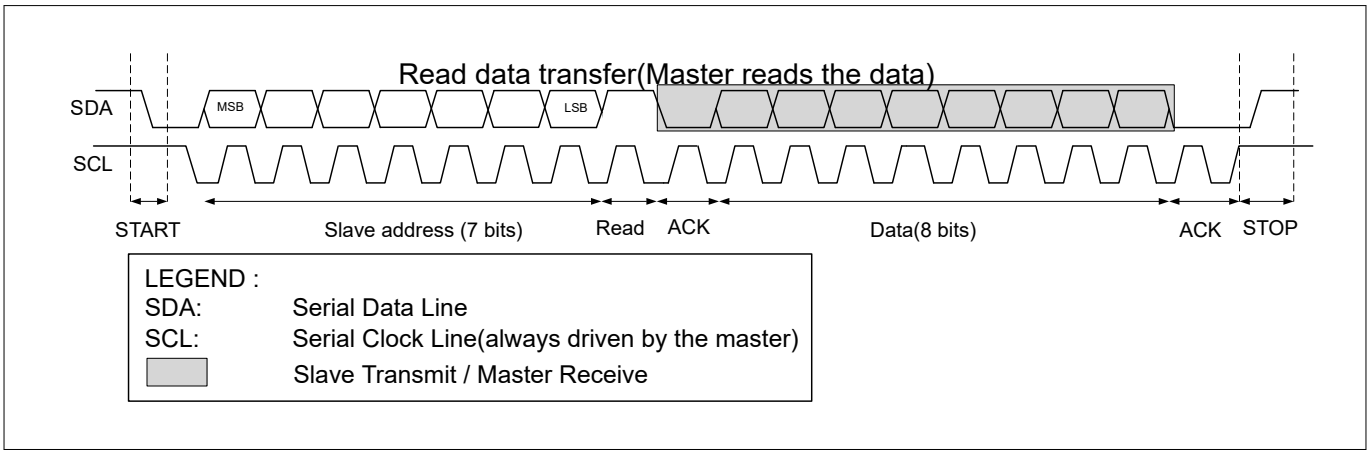


Figure 333 Master read data transfer

27.5.5 I2C buffer modes

I2C can operate in three different buffered modes – FIFO, EZ, and CMD_RESP modes. The buffer is used in different ways in each of the modes. The following subsections explain each of these buffered modes in detail.

27.5.5.1 FIFO mode

The FIFO mode has a TX FIFO for the data being transmitted and an RX FIFO for the data being received. Each FIFO is constructed out of the SRAM buffer. The FIFOs are either 64 elements deep with 16-bit data elements or 128 elements deep with 8-bit data elements. The width of the data elements is configured using the SCBx_CTRL.MEM_WIDTH. For I2C, it is recommended to put the FIFO in BYTE mode because all transactions are a byte wide.

The FIFO mode operation is available only in CPU Active and CPU Sleep power modes. In the system Deep Sleep power mode, only the system Deep Sleep-capable SCB can wake up the device up from sleep through slave

address match. For more information, see the "System Deep Sleep to CPU Active mode transition in FIFO buffer mode" section.

A write access to the transmit FIFO uses the SCBx_TX_FIFO_WR register. A read access from the receive FIFO uses the SCBx_RX_FIFO_RD register.

Transmit and receive FIFO status information is available through status registers SCBx_TX_FIFO_STATUS and SCBx_RX_FIFO_STATUS. When in debug mode, a read from this register behaves as a read from the SCBx_RX_FIFO_RD_SILENT register that is, data will not be removed from the FIFO.

It is possible to define a programmable threshold that indicates a number of FIFO entries, a trigger/event is generated when the following conditions are met:

- For a TX FIFO, a trigger is generated when the number of entries in the transmit FIFO is less than TRIGGER_LEVEL
- For the RX FIFO, a trigger is generated when the number of entries in the FIFO is greater than the TRIGGER_LEVEL

Furthermore, several interrupt status bits are provided as well, which indicate if the FIFOs are full, empty, and so on.

System Deep Sleep to CPU Active mode transition in FIFO buffer mode

SCBx_CTRL.EC_AM_MODE = 1, SCBx_CTRL.EC_OP_MODE = 0, FIFO Mode. The following descriptions only apply to slave mode.

Master Write:

- S_NOT_READY_ADDR_NACK = 0, SCBx_I2C_CTRL.S_READY_ADDR_ACK = 1. The clock is stretched until the internally-clocked logic takes over, at which point the address is ACK'd and the master can start writing data. Before going to Deep Sleep, CLK_SCB needs to be disabled. Upon wake-up from Deep Sleep CLK_SCB must be reenabled; this is when the clock stretch will be released
- S_NOT_READY_ADDR_NACK = 0, SCBx_I2C_CTRL.S_READY_ADDR_ACK = 0. The clock is stretched until the internally-clocked logic takes over and the CPU writes either SCBx_I2C_S_CMD.S_ACK, or SCBx_I2C_S_CMD.S_NACK. Before going to Deep Sleep CLK_SCB needs to be disabled. Upon wake-up from Deep Sleep CLK_SCB must be reenabled, do this before setting SCBx_I2C_S_CMD.S_ACK or SCBx_I2C_S_CMD.S_NACK
- S_NOT_READY_ADDR_NACK = 1, SCBx_I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is NACK'd until the internally-clocked logic takes over. When the internally-clocked logic takes over, there is no guarantee that the internal clock will be at the correct frequency due to PLL/FLL locking times. This may lead to incorrect timing on the I2C bus for the ACK/NACK. To avoid this disable CLK_SCB before going to Deep Sleep, and then re-enable after the PLL/FLL have stabilized

Master Read:

- S_NOT_READY_ADDR_NACK = 0, SCBx_I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is stretched until the internally-clocked logic takes over and the CPU writes data into the TX FIFO. Before going to Deep Sleep CLK_SCB needs to be disabled. Upon wake-up from Deep Sleep CLK_SCB must be reenabled before writing data into the TX FIFO
- S_NOT_READY_ADDR_NACK = 1, SCBx_I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is NACK'd until the internally-clocked logic takes over. When this happens, there is no guarantee that the internal clock will be at the correct frequency due to PLL/FLL locking times. This may lead to incorrect timing on the I^2C bus for the ACK/NACK. To avoid this, disable CLK_SCB before going to Deep Sleep, and then reenable after the PLL/FLL have stabilized

***Note**: When doing a repeated start after a write, wait until the UNDERFLOW interrupt status is asserted before setting the I2C_M_CMD.START bit and writing the new address into the TX_FIFO. Otherwise, the address in the FIFO will be sent as data and not as an address.*

27.5.5.2 EZI2C mode

The EZI2C protocol is a unique communication scheme built on top of the I2C protocol by Infineon Technologies AG. It uses a metaprotocol around the standard I2C protocol to communicate to an I2C slave using indexed memory transfers. This removes the need for CPU intervention.

The EZI2C protocol defines a single memory buffer with an 8-bit address that indexes the buffer (an 256-entry array of 8-bit per entry is supported) located on the slave device. The EZ address is used to address these 256 locations. The CPU writes and reads to the memory buffer through the EZ_DATA registers. These accesses are word accesses, but only the least significant byte of the word is used.

The slave interface accesses the memory buffer using the current address. At the start of a transfer (I2C START/RESTART), the base address is copied to the current address. A data element write or read operation is to the current address location. After the access, the current address is incremented by '1'.

If the current address equals the last memory buffer address (255), the current address is not incremented. Subsequent write accesses will overwrite any previously written value at the last buffer address. Subsequent read accesses will continue to provide the (same) read value at the last buffer address. The bus master should be aware of the memory buffer capacity in EZ mode.

The I2C base and current addresses are provided through I2C_STATUS. At the end of a transfer, the difference between the base and current addresses indicates how many read or write accesses were performed. The block provides interrupt cause fields to identify the end of a transfer. EZI2C can be implemented through firmware or hardware. All SCBs can implement EZI2C through a firmware implementation in both CPU Active and CPU Sleep power modes. The Deep Sleep SCB can implement a hardware-and firmware-based EZI2C with a system Deep Sleep power mode.

EZI2C distinguishes three operation phases:

- Address phase: The master transmits an 8-bit address to the slave. This address is used as the slave base and current address
- Write phase: The master writes 8-bit data element(s) to the slave's memory buffer. The slave's current address is set to the slave's base address. Received data elements are written to the current address memory location. After each memory write, the current address is incremented
- Read phase: The master reads 8-bit data elements from the slave's memory buffer. The slave's current address is set to the slave's base address. Transmitted data elements are read from the current address memory location. After each memory read, the current address is incremented

***Note**: A slave's base address is updated by the master and not by the CPU.*

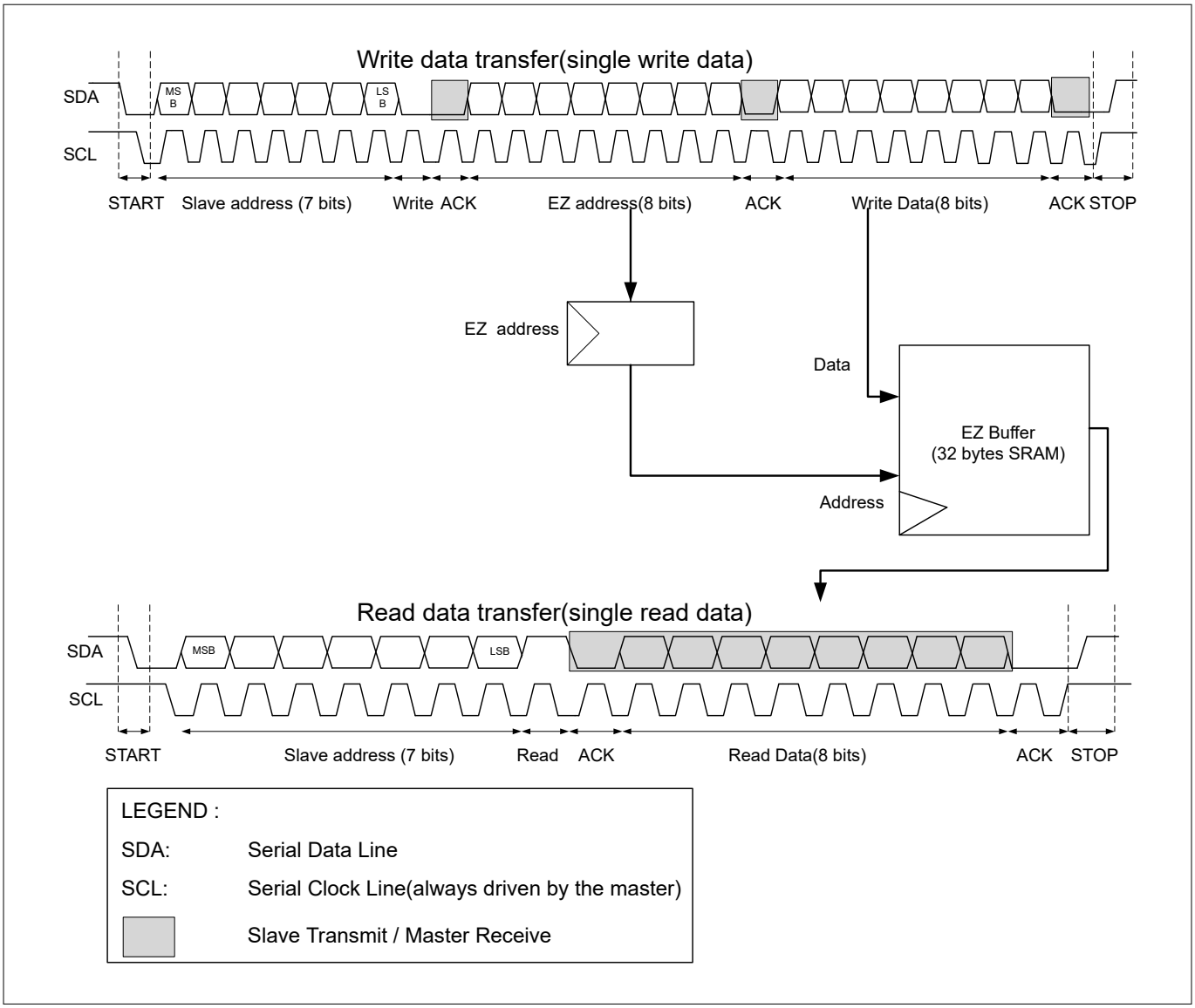


Figure 334 EZI2C Write and Read data transfer

System Deep Sleep to CPU Active mode transition in EZ buffer mode

SCBx_CTRL.EC_AM_MODE = 1, SCBx_CTRL.EC_OP_MODE = 0, EZ Mode.

- S_NOT_READY_ADDR_NACK = 0, SCBx_I2C_CTRL.S_READY_ADDR_ACK = 1. The clock is stretched until the internally-clocked logic takes over, at which point the address is ACK'd and the master can start writing data. Before going to Deep Sleep CLK_SCB needs to be disabled. Upon wake-up from Deep Sleep CLK_SCB must be reenabled this is when the clock stretch will be released
- S_NOT_READY_ADDR_NACK = 1, SCBx_I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is NACK'd until the internally-clocked logic takes over. When this happens, there is no guarantee that the internal clock will be at the correct frequency due to PLL/FLL locking times. To avoid this, disable CLK_SCB before going to Deep Sleep, and then reenable after the PLL/FLL have stabilized

27.5.5.3 Command-response mode

This mode has a single memory buffer, a base read address, a current read address, a base write address, and a current write address that is used to index the memory buffer. The base addresses are provided by the CPU.

The current addresses are used by the slave to index the memory buffer for sequential accesses of the memory buffer. The memory buffer holds 256 8-bit data elements. The base and current addresses are in the range [0 to 255].

The CPU writes and reads to the memory buffer through the SCBx_EZ_DATA registers. These are word accesses, but only the least significant byte of the word is used.

The slave interface accesses the memory buffer using the current addresses. At the start of a write transfer (I^2C START/RESTART), the base write address is copied to the current write address. A data element write is to the current write address location. After the write access, the current address is incremented by '1'. At the start of a read transfer, the base read address is copied to the current read address. A data element read is to the current read address location. After the read data element is transmitted, the current read address is incremented by '1'.

If the current addresses equal the last memory buffer address (255), the current addresses are not incremented. Subsequent write accesses will overwrite any previously written value at the last buffer address. Subsequent read accesses will continue to provide the (same) read value at the last buffer address. The bus master should be aware of the memory buffer capacity in command-response mode.

The base addresses are provided through SCBx_CMD_RESP_CTRL. The current addresses can be viewed in SCBx_CMD_RESP_STATUS. At the end of a transfer (I^2C stop), the difference between a base and current address indicates how many read/write accesses were performed. This block provides interrupts to identify the end of a transfer, which can be found in SCBx_INTR_I2C_EC and SCBx_INTR_SPI_EC register sections.

Command-response mode operation is available in CPU Active and system Deep Sleep power modes. The Command-response mode has two phases of operation:

- Write phase - The write phase begins with a START/RESTART followed by the slave address with read/write bit set to '0' indicating a write. The slave's current write address is set to the slave's base write address. Received data elements are written to the current write address memory location. After each memory write, the current write address is incremented
- Read phase - The read phase begins with a START/RESTART followed by the slave address with read/write bit set to '1' indicating a read. The slave's current read address is set to the slave's base read address. Transmitted data elements are read from the current address memory location. After each read data element is transferred, the current read address is incremented

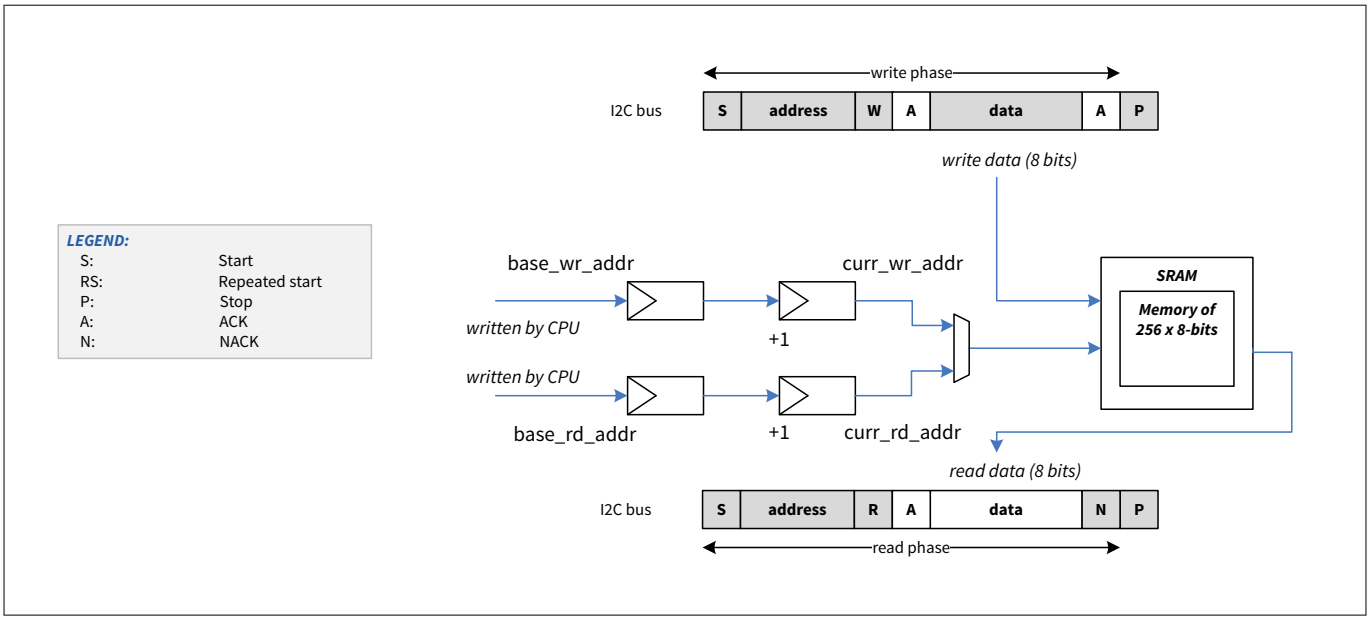


Figure 335 I2C command-response mode

***Note**: A slave's base address is updated by the CPU and not by the master.*

27.5.6 Clocking and oversampling

The SCB I2C supports both internally and externally clocked operation modes. Two bitfields (SCBx_CTRL.EC_AM_MODE and SCBx_CTRL_EC_OP_MODE) in the SCB_CTRL register determine the SCB clock mode. SCBx_CTRL.EC_AM_MODE indicates whether I2C address matching is internally (0) or externally (1) clocked. I2C address matching comprises the first part of the I2C protocol. SCBx_CTRL.EC_OP_MODE indicates whether the rest of the protocol operation (besides I2C address matching) is internally (0) or externally (1) clocked. The externally clocked mode of operation is supported only in the I2C slave mode.

An internally-clocked operation uses the programmable clock dividers. For I2C, an integer clock divider must be used for both master and slave. For more information on system clocking, see the Clocking system section. The internally-clocked mode does not support the command-response mode.

***Note**: In the PSC3M/P6 MCU, only one SCB supports an externally-clocked mode of operation.*

The SCBx_CTRL.EC_AM_MODE and SCBx_CTRL.EC_OP_MODE can be configured in the following ways.

- EC_AM_MODE is '0' and SCBx_CTRL.EC_OP_MODE is '0': Use this configuration when only CPU Active mode functionality is required
  - FIFO mode: Supported
  - EZ mode: Supported
  - Command-response mode: Not supported. The slave NACKs every slave address
- EC_AM_MODE is '1' and SCBx_CTRL.EC_OP_MODE is '0': Use this configuration when both CPU Active and system Deep Sleep functionalities are required. This configuration relies on the externally clocked functionality for the I2C address matching and relies on the internally clocked functionality to access the memory buffer. The "hand over" from external to internal functionality relies either on an ACK/NACK or clock stretching scheme. The former may result in termination of the current transfer and relies on a master retry. The latter stretches the current transfer after a matching address is received. This mode requires the master to support either NACK generation (and retry) or clock stretching. When the I2C address is matched,

SCBx_INTR_I2C_EC.WAKE_UP is set to ‘1’. The associated system Deep Sleep functionality interrupt brings the system into CPU Active power mode
  - FIFO mode: See FIFO mode
  - EZ mode: See EZI2C Mode
  - CMD_RESP mode: Not supported. The slave NACKs every slave address
- EC_AM_MODE is ‘1’ and SCBx_CTRL.EC_OP_MODE is ‘1’. Use this mode when both CPU Active and system Deep Sleep functionalities are required. When the slave is selected, SCBx_INTR_I2C_EC.WAKE_UP is set to ‘1’. The associated system Deep Sleep functionality interrupt brings the system into CPU Active power mode. When the slave is deselected, SCBx_INTR_I2C_EC.EZ_STOP and/or SCBx_INTR_I2C_EC.EZ_WRITE_STOP are set to ‘1’
  - FIFO mode: Not supported
  - EZ mode: Supported
  - CMD_RESP mode: Supported

Table 290 Clock configuration and mode support

Mode	SCBx_CTRL.EC_AM_MODE is '0'; SCBx_CTRL.EC_OP_MODEis '0	SCBx_CTRL.EC_AM_MODE is '1'; SCBx_CTRL.EC_OP_MODE is '0	SCBx_CTRL.EC_AM_MODE is '1'; SCBx_CTRL.EC_OP_MODE is '1'
FIFO mode	Yes	Yes	No
EZ mode	Yes	Yes	Yes
CMD_RESP mode	No	No	Yes

An externally-clocked operation uses a clock provided by the serial interface. The externally clocked mode does not support FIFO mode. If SCBx_CTRL.EC_OP_MODE is ‘1’, the external interface logic accesses the memory buffer on the external interface clock (I2C SCL). This allows for EZ and CMD_RESP mode functionality in CPU Active and system Deep Sleep power modes.

In CPU Active power mode, the memory buffer requires arbitration between an external interface logic(on I2C SCL) and the CPU interface logic (on system peripheral clock). This arbitration always gives the highest priority to the external interface logic (host accesses). The external interface logic takes one or two serial interface clock/bit periods for the I2C. During this period, the internal logic is denied service to the memory buffer. The PSC3M/P6 MCU provides two programmable options to address this “denial of service”:

- If the SCBx_CTRL.BLOCK is '1': An internal logic access to the memory buffer is blocked until the memory buffer is granted and the external interface logic has completed access. For a 100-kHz I2C interface, the maximum blocking period of one serial interface bit period measures 10 µs (approximately 208 clock cycles on a 48 MHz SCB input clock). This option provides normal SCB register functionality, but the blocking time introduces additional internal bus wait states
- If the SCBx_CTRL.BLOCK is '0': An internal logic access to the memory buffer is not blocked, but fails when it conflicts with an external interface logic access. A read access returns the value 0xFFFF:FFFF and a write access is ignored. This option does not introduce additional internal bus wait states, but the access to the memory buffer may not take effect. In this case, the following failures are detected:
- Read Failure: A read failure is easily detected, as the returned value is 0xFFFF:FFFF. This value is unique as nonfailing memory buffer read accesses return an unsigned byte value in the range 0x0000:0000-0x0000:00FF
- Write Failure: A write failure is detected by reading back the written memory buffer location, and confirming that the read value is the same as the written value

For both options, a conflicting internal logic access to the memory buffer sets SCBx_INTR_TX.BLOCKED field to ‘1’ (for write access-es) and SCBx_INTR_RX.BLOCKED field to ‘1’ (for read accesses). These fields can be used as either status fields or as interrupt cause fields (when their associated mask fields are enabled).

If a series of read or write accesses is performed and SCBx_CTRL.BLOCKED is '0', a failure is detected by comparing the logical OR of all read values to 0xFFFF:FFFF and checking the SCBx_INTR_TX.BLOCKED and SCBx_INTR_RX.BLOCKED fields to determine whether a failure occurred for a (series of) write or read operation(s).

27.5.6.1 Glitch filtering

The PSC3M/P6 MCU SCB I2C has analog and digital glitch filters. Analog glitch filters are applied on the i2c_scl_in and i2c_sda_in input signals (AF_in) to filter glitches of up to 50 ns. An analog glitch filter is also applied on the i2c_sda_out output signal (AF_out). Analog glitch filters are enabled and disabled in the SCBx_I2C_CFG register. Do not change the _TRIM bit fields, only change the _SEL bit fields in this register.

Digital glitch filters are applied on the i2c_scl_in and i2c_sda_in input signals (DF_in). The digital glitch filter is enabled in the SCBx_RX_CTRL.MEDIAN.

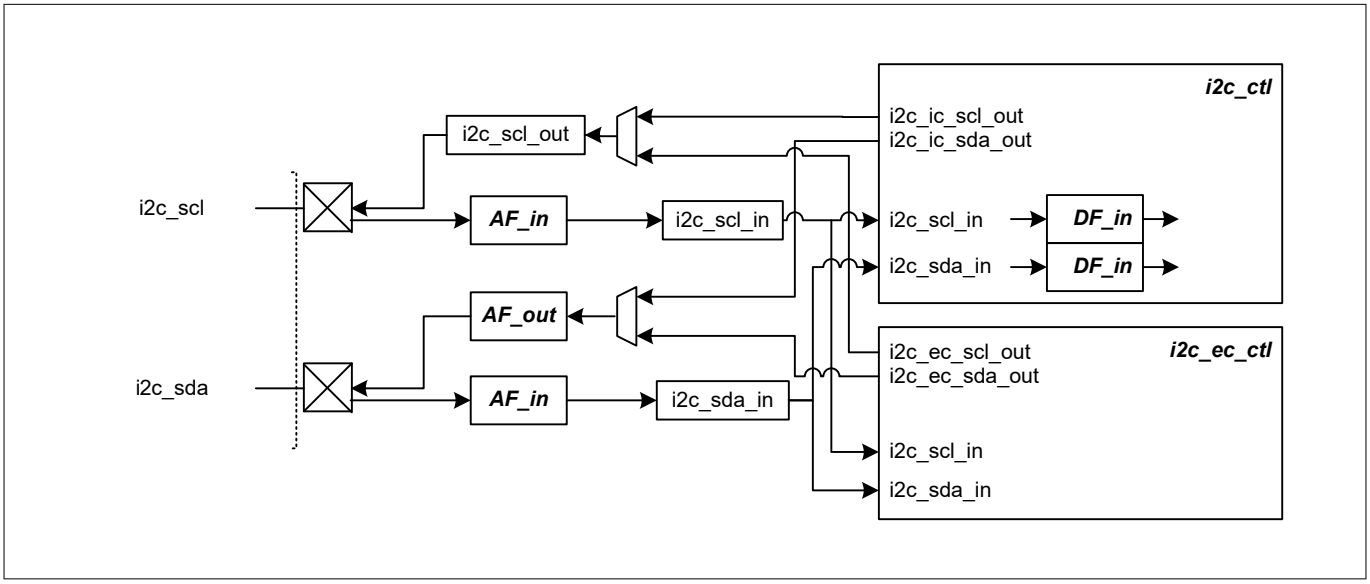


Figure 336 I2C glitch filtering connection

The following table lists the useful combinations of glitch filters.

Table 291 Glitch filter combinations

AF_in	AF_out	DF_in	Comments
0	0	1	Used when operating in internally-clocked mode and in Master in Fast-mode plus (1-MHz speed mode)
1	0	0	Used when operating in internally-clocked mode (SCBx_CTRL.EC_OP_MODE is '0')
1	1	0	Used when operating in externally-clocked mode (SCBx_CTRL.EC_OP_MODE is '1'). Only slave mode.

When operating in EC_OP_MODE = 1, the 100-kHz, 400-kHz, and 1000-kHz modes require the following settings for AF_out:

AF_in	AF_out	DF_in	Comments
1	1	0	100-kHz mode: SCBx_I2C_CFG.SDA_OUT_FILT_SEL = 3 400-kHz mode: SCBx_I2C_CFG.SDA_OUT_FILT_SEL = 3 1000-kHz mode: SCBx_I2C_CFG.SDA_OUT_FILT_SEL = 1

27.5.6.2 Oversampling and bit rate

Internally-clocked Master

The PSC3M/P6 MCU implements the I2C clock as an oversampled multiple of the SCB input clock. In master mode, the block determines the I2C frequency. Routing delays on the PCB, on the device, and the block (including analog and digital glitch filters) all contribute to the signal interface timing. In master mode, the block operates off CLK_SCB and uses programmable oversampling factors for the SCL high SCBx_I2C_CTRL.HIGH_PHASE_OVS and low SCBx_I2C_CTRL.LOW_PHASE_OVS times.

Table 292 I2C frequency and oversampling requirements in I2C master mode

AF_in	AF_out	DF_in	Mode	Supported frequency	SCBx_I2C_CTRL.LOW_PHASE_OVS	SCBx_I2C_CTRL.HIGH_PHASE_OVS	Input Clock Frequency
0	0	1	100 kHz	[62, 100] kHz	[9, 15]	[9, 15]	[1.98-3.2] MHz
			400 kHz	[264, 400] kHz	[13, 15]	[7, 15]	[8.45-10] MHz
			1000 kHz	[447, 1000] kHz	[8, 15]	[5, 15]	[14.32-25.8] MHz
1	0	0	100 kHz	[48, 100] kHz	[7, 15]	[7, 15]	[1.55-3.2] MHz
			400 kHz	[244, 400] kHz	[12, 15]	[7, 15]	[7.82-10] MHz
			1000 kHz	Not Supported			

Table assumes worst-case conditions on the I2C bus. The following equations can be used to determine the settings for your own system. This involves measuring the rise and fall times on SCL and SDA lines in your system.

tCLK_SCB(Min) = (tLOW + tF)/SCBx_I2C_CTRL.LOW_PHASE_OVS

If clk_scb is any faster than this, the tLOW of the I2C specification will be violated. tF needs to be measured in your system.

tCLK_SCB(Max) = (tVD – tRF – 100 nsec)/3 (When analog filter is enabled and digital filter disabled)

tCLK_SCB(Max) = (tVD – tRF)/4 (When analog filter is disabled and digital filter is enabled)

tRF is the maximum of either the rise or fall time. If clk_scb is slower than this frequency, tVD will be violated.

Internally-clocked Slave

In slave mode, the I2C frequency is determined by the incoming I2C SCL signal. To ensure proper operation, CLK_SCB must be significantly higher than the I2C bus frequency. Unlike master mode, this mode does not use programmable oversampling factors. Table assumes worst-case conditions on the I2C bus including the chip internal delay.

Table 293 SCB input clock requirements in SCB input clock requirements in I2C slave mode

AF_in	AF_out	DF_in	Mode	CLK_SCB frequency range
0	0	1	100 kHz	[1.98-12.8] MHz
			400 kHz	[8.45-17.14] MHz
			1000 kHz	[14.32-44.77] MHz
1	0	0	100 kHz	[1.55-12.8] MHz
			400 kHz	[7.82-15.38] MHz
			1000 kHz	[15.84-89.0] MHz

tCLK_SCB(Max) = (tVD – tRF – 100 nsec) / 3 (When analog filter is enabled and digital filter disabled)

tCLK_SCB(Max) = (tVD – tRF) / 4 (When analog filter is disabled and digital filter is enabled)

tRF is the maximum of either the rise or fall time. If CLK_SCB is slower than this frequency, tVD will be violated.

The minimum period of CLK_SCB is determined by one of the following equations:

tCLK_SCB(MIN) = (tSU_DAT(min) + tRF) /16

Master-Slave

In this mode, when the SCB is acting as a master device, the block determines the I2C frequency. When the SCB is acting as a slave device, the block does not determine the I2C frequency. Instead, the incoming I2C SCL signal does.

To guarantee operation in both master and slave modes, choose clock frequencies that work for both master and slave using the tables above.

27.5.7 Loop-back

In master-slave mode, SCB supports internal SCL and SDA lines are routed internally in the peripheral. As a result, it is unaffected by other I2C devices. It is configured using the `SCBx_I2C_CTRL.LOOPBACK` register.

27.5.8 Enabling and initializing I2C

27.5.8.1 Configuring for I2C FIFO mode

Do the following to program the I2C interface:

1. Program protocol-specific information using the `SCBx_I2C_CTRL` register. This includes selecting master-slave functionality (MASTER_MODE, SLAVE_MODE)

2. Program the generic transmitter and receiver information using the SCBx_TX_CTRL and SCBx_RX_CTRL registers.
   a. Specify the data frame width (DATA_WIDTH = 7)
   b. Specify that MSb is the first bit to be transmitted/received (MSB_FIRST = 1)
3. Set the SCBx_CTRL.MEM_WIDTH to '00' to enable the byte mode
4. Program the transmitter and receiver FIFOs using the SCBx_TX_FIFO_CTRL and SCBx_RX_FIFO_CTRL registers respectively.
   a. Set the trigger level (TRIGGER_LEVEL)
   b. Clear the transmitter and receiver FIFO and Shift registers (CLEAR)
5. Program the SCBx_CTRL register to enable the I2C block and select the I^2C mode. For a complete description of the I2C registers, see the PSC3M/P6 MCU registers RM

27.5.8.2 Configuring for EZ and CMD_RESP modes

To configure the I2C block for EZ and CMD_RESP modes, set the following I2C register bits:

1. For EZ mode, select the EZI2C mode by writing '1' to the SCBx_CTRL.EZ_MODE register
2. For CMD_RESP mode, select CMD_RESP mode by writing a '1' to the SCBx_CTRL.CMD_RESP register
3. Set the S_READY_ADDR_ACK (bit 12) and SCBx_I2C_CTRL.S_READY_DATA_ACK (bit 13) register

27.5.9 I/O pad connection

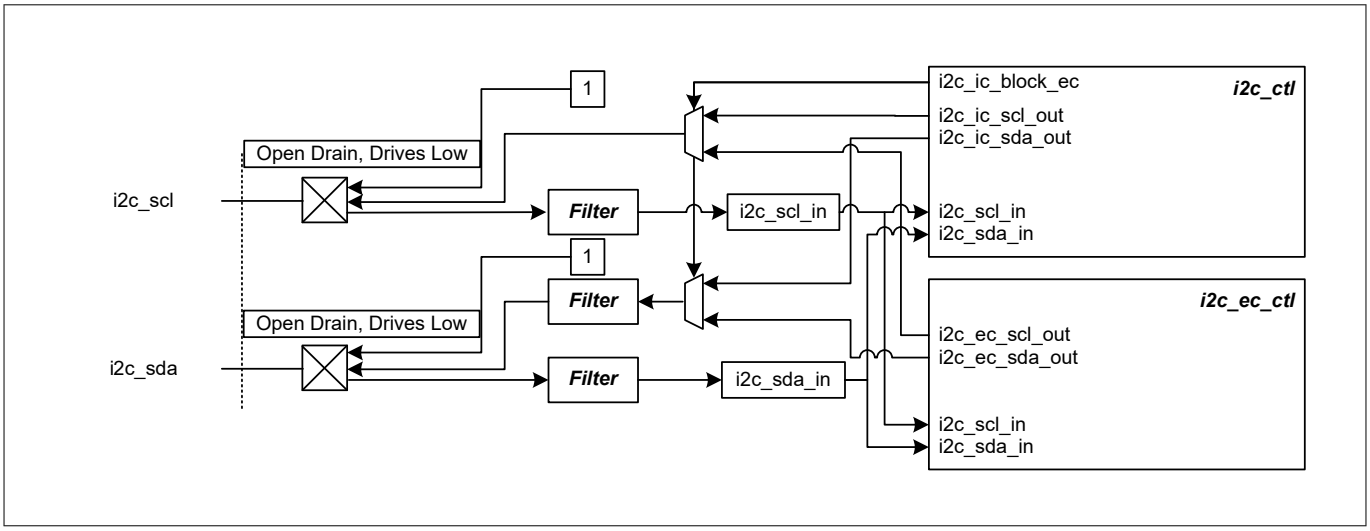


Figure 337 I2C I/O pad connections

Table 294 I2C I/O pad descriptions

I/O pads	Drive mode	On-chip I/O signals	Usage
i2c_scl	Open drain drives low	i2c_scl_in	Receive a clock
		i2c_scl_out	Transmit a clock
		i2c_scl_out_en = 1	i2c_scl_out_en is fixed to high (enabled).
i2c_sda	Open drain drives low	i2c_sda_in	Receive data
		i2c_sda_out	Transmit data

(table continues...)

Table 294 (continued) I2C I/O pad descriptions

I/O pads	Drive mode	On-chip I/O signals	Usage
		i2c_sda_out_en = 1	i2c_sda_out_en is fixed to high (enabled).

27.5.10 4-wire I2C feature

27.5.10.1 Configurable inversion of 4-wire I2C outputs

In order to provide a high flexibility on external circuit design, the output of i2c_sda_do_out and i2c_scl_do_out will provide an configurable inverter.

The following are the new register bits added in the TX_CTRL register:

Bits	Bit Name	Description
18	INV_SDA_DO	'0': no inverter chosen for the I2C SDA for the 4-wire I2C '1': inverted output chosen for I2C SDA for the 4-wire I2C
19	INV_SCL_DO	'0: no inverter chosen for the I2C SCL for the 4-wire I2C 1: inverted output chosen for I2C SCL for the 4-wire I2C

Figure 338 illustrates the implementation circuit:

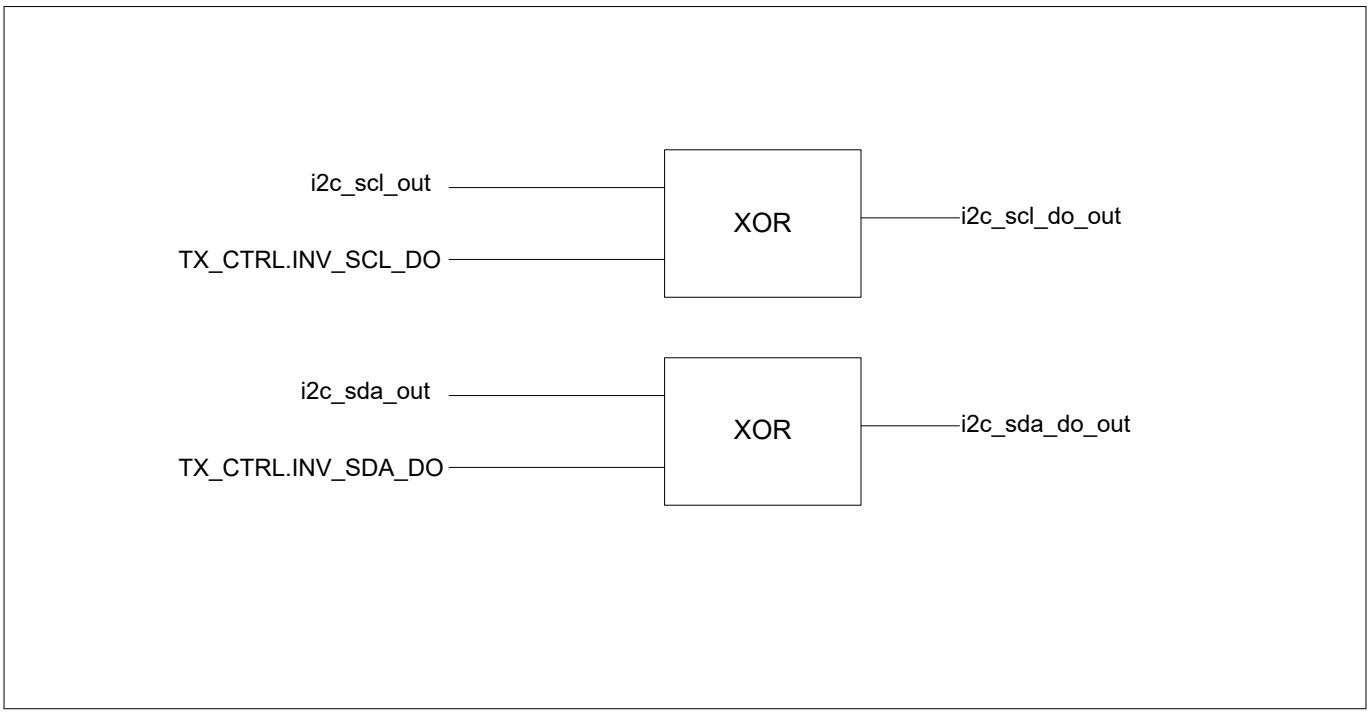


Figure 338 4-wire I2C circuit

27.5.11 Registers summary

The I2C interface is controlled by reading and writing a set of configuration, control, and status registers. For detailed information on these registers, see the PSC3M/P6 MCU registers reference manual.

27.6 Timeout Generation Support (TGS)

The Timeout Generation Support (TGS) functionality in SCB provides a robust mechanism to monitor and handle timeout events in SMBUS/PMBUS protocols. There are three 18-bit down-counters designed to offer flexibility and precision in protocol monitoring, while supporting various communication interfaces (I2C, SPI, UART). The following sections provide an in-depth understanding of the TGS feature, its use cases, behavior, and considerations

27.6.1 Key features of TGS functionality

- **Event-Driven Timeout Monitoring:**
  - The TGS monitors the time between communication events (e.g., clock edges or start/stop conditions)
  - Timeout events (when a counter reaches 0x0000) trigger interrupts SCB (SPI, UART and I2C) and/or resets (only available in I2C Slave mode) to address stuck or incomplete communication states
  - Customers can configure the sensitivity and granularity of timeout behavior
- **Three 18-bit Down-Counters**:
  - Each counter operates independently
  - Gated by the tgs_clk_en input, providing fine-grained control of timeout monitoring
  - Reload, Gating, and Stop events influence the counter behavior
- **Flexibility with Protocol Monitoring**:
  - The TGS supports I2C, SPI (Motorola, TI, NS modes), and UART communication protocols
  - For I2C, the TGS ensures compliance with SMBUS/PMBUS specifications by measuring high/low periods of the SCL line (communication is stuck or incomplete)
  - For SPI, it accommodates protocol-specific behaviors like frame delineation and start/stop conditions
- **Design-Time Configurability**:
  - TGS feature available in both the **Active and Deep Sleep SCB instances**
- **Retention and Non-Retention Registers**:
  - Retention: TGS_CTL (control registers) and TGS_REL (reload values)
  - Non-Retention: TGS_CNT (counter values), reset upon power cycle or deep-sleep entry

27.6.2 Functional description

Counter Operation

- The counters begin decrementing from a reload value upon a reload event
- If the counter value reaches 0x0000:
  - An interrupt is triggered
  - The SCB can optionally reset (via tgs_rst) only in I2C slave mode
- A **Reload event** reloads the counter with its reload value. A **Stop event** stops the counter by writing zero into it and halts decrementing without triggering an interrupt
- Once the counter is enabled, the present counter value will be decremented until 0x0000. In case the initial value is already at 0x0000, nothing will happen until the first trigger event reloads the counter value. Each subsequent reload trigger event also reloads the counter

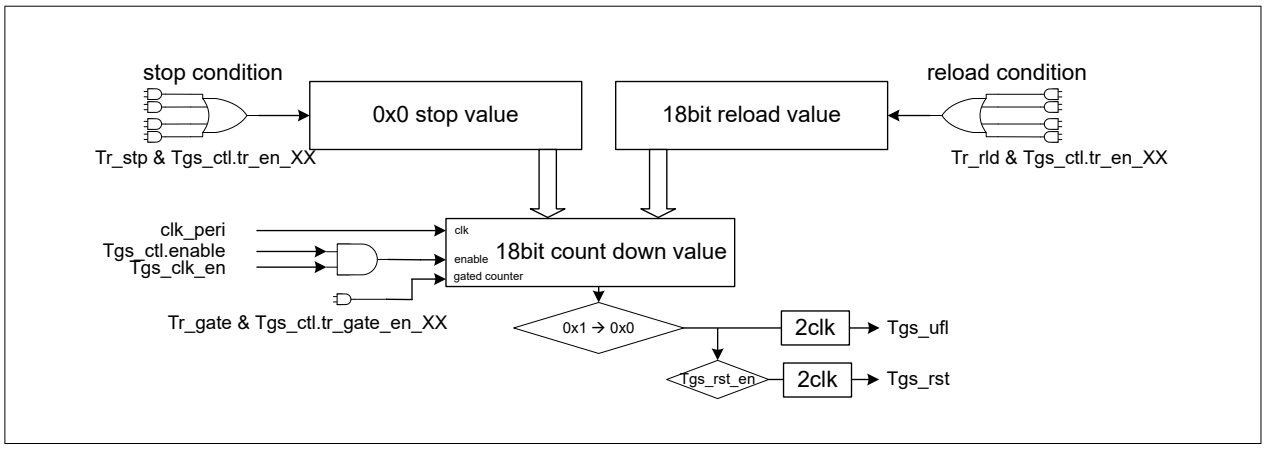


Figure 339 TGS Block diagram

Event Types

1. **Reload Event**:
   - Reloads the counter with the value stored in TGS_RELx.RELOAD
   - Typically triggered at the **start of a frame**
   - Subsequent reload events ensure the counter restarts
2. **Gating Event**:
   - Controls whether the counter decrements or pauses
   - When trigger gate (tr_tgs_gated) is high, the counter pauses. When low, decrementing resumes
3. **Stop Event**:
   - Stop event stops the counter by writing 0x0 into counter
   - Does not trigger an interrupt or reset
   - All events at 0x1 take precedence over the transition to 0x0

Interface Signals

- **Inputs**:
  - clock_tgs_en: Enables the clock for decrementing
- **Outputs**:
  - tr_tgs_reload[2:0]: Indicates that a reload event occurred for a specific counter
  - tr_tgs_stop[2:0]: Indicates that a stop event occurred
  - tr_tgs_gated[2:0]: Indicates when the counter is paused (gated)
  - **Generated Signals**:
    - tgs_underflow: Signals that a counter reached 0x0 (used for interrupts)
    - tgs_rst: Resets the SCB for 2 clock cycles to reinitialize the state machine

Priority of Counter Triggers:

1. Software Write
2. Stop event
3. Reload event
4. Gating Event
5. If the counter is not zero, it will decrement

***Note**: If there is a software write to the TGS counter, the frequency of TGS clock can be changed, since counter write has the priority. Therefore, the customer must expect that the bit-field of the register is updated. An uncertainty of -1 is expected when software is writing to the TGS counter, since frequency of TGS clock is changed*

27.6.3 Intended use case

Setup for a 25ms Timeout:

```
// Configure timeout counter
 Write(TGS_RELx.RELOAD = 0x9C40; TGS_CNTx.COUNT = 0x0000);
 Write(TGS_CTLx.ENABLE = 1b; TGS_CTLx.TR_I2C_P = 1b;
 TGS_CTLx.TR_I2C_S = 1b; TGS_CTLx.TR_FCLK = 1b);
```

- Wait for the first external clock edge.
- Start counting low periods between I2C start and stop conditions.
- On transition to `0x0000`, an interrupt triggers. software can react on the stuck-at clock

Additionally by setting TGS_CTLx.STATE_RST =1b, SCB (Only in I2C slave mode) disable for two clock cycles followed by an enable of SCB can be done, which resets the internal protocol state machine without software interaction. This sequence creates a 2-cycle window, where the module is disabled and will therefore not react on any external event. After reset the FIFO is also initialized

27.6.4 Special cases and protocol-specific behavior

1. **Special scenarios for SPI Triggers (valid for FIFO triggers only):**

   Expected behavior: The priority order of TGS triggers, from high to low: STOP -> RELOAD -> GATE -> decrementing. Due to pulse extenders were introduced, some TGS triggers can now be wider than 1 TGS clock. In case of the SPI, only FIFO (both RX and TX) triggers are affected due to pulse extension and can be extended on scenarios where TGS clock is greater than SCB clock. If a scenario where RELOAD(FIFO) and STOP(spi_clk) conditions are set at the same TGS cycle. The expectation is counter is stopped, because STOP has greater priority then RELOAD. But since RELOAD pulse is wider, counter will reload one clock cycle later. The behavior is expected as per design in Internal Clock mode

2. **Special scenarios for I2C Triggers (valid for FIFO and ACK triggers):**

   Expected behavior: The priority order of TGS triggers, from high to low: STOP -> RELOAD -> GATE -> decrementing. Due to pulse extenders were introduced, some TGS triggers can now be wider than 1 TGS clock. In case of the I2C, FIFO (both TX and RX) triggers and interface triggers (ACK on the rising and falling edge of SCL) are affected due to pulse extension and can be extended on scenarios where TGS clock is greater than SCB clock. If a scenario where RELOAD and STOP conditions are set at the same TGS cycle. The expectation is counter is stopped, because STOP has greater priority then RELOAD. But since RELOAD pulse is wider, counter will reload one clock cycle later. The behavior is expected as per design in IC mode

3. **SPI Triggers behavior for TI mode:**

   In TI mode, the spi_select signal is used to indicate the start of a data transfer. The start of data transfer is indicated by a high active pulse. So, spi_select is the indication of the start of the frame – there is no indication of the end of the frame. In TGS_CTRL register, the counter can be configured to be gated inside or outside of the frame. And the end of frame cannot be declared based on the spi_select signal. Additionally, the polarity of the spi_select is different compared to Motorola and TI modes.

For TI mode,

- polarity=0, high pulse of the spi_select is indication of the start of the frame. Current implementation of the TGS will give TGS_RELOAD on falling edge of the spi_select, TGS_STOP on rising edge of the spi_select
- polarity=1 low (negative)pulse on spi_select is indication of the start of the frame. Current implementation of the TGS will give TGS_RELOAD on rising edge of the spi_select, TGS_STOP on falling edge of the spi_select

4. SPI Master behavior for National Semiconductors (NS) mode:

For the SPI Master, NS mode, with late sample enabled, active instance and valid only for IC mode, the falling edge of the spi_select should be the START condition (start of frame) and rising edge of the spi_select should be the STOP condition (end of frame). The customer must be aware of the fact that in case of SPI master SCB with an always running SPI clock, TGS clock(clk_tgs) should have same or higher frequency than SCB clock (clk_scb). For SCB clock (clk_scb) faster than TGS clock (clk_tgs), select line is de-asserted just before the next SPI clock rising edge (spi_clk is always running). The STOP condition gets cleared on the next rising edge of spi_clk. Since TGS clock (clk_tgs) is slow, so the counter misses the stop condition, so STOP trigger won't be detected

5. Deep-sleep to Active Transition Behavior

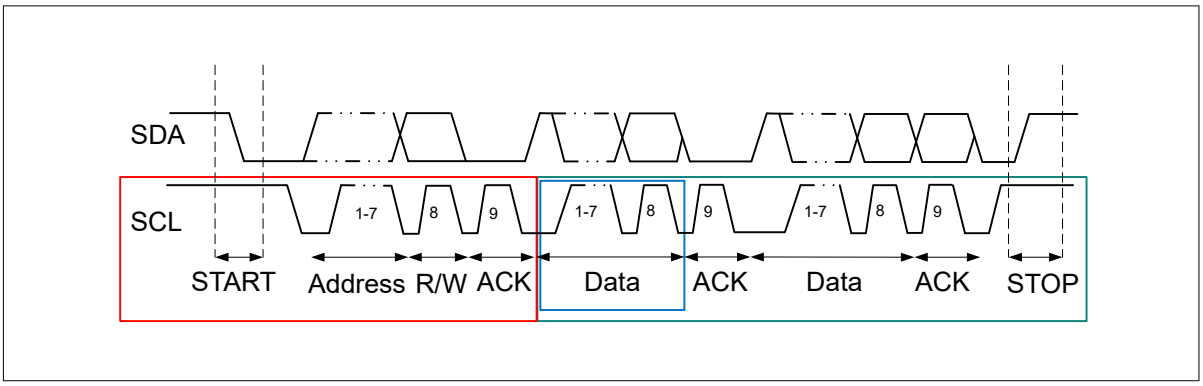


Figure 340 Deep-sleep to Active transition

In the above figure, the red box defines a deep-sleep power mode and the green box defines an active mode. In the case of transition from deep-sleep to active, address matching happens in deep-sleep, address acknowledgment can happen also in deep-sleep. SCB wakes up the system and the remaining of the transaction happens in active state. TGS functionality is only available in active state. There is a risk that TGS does not measure the stuck clock at the edge of the transition because of missing reload conditions which happened in deep-sleep state

A solution for a such scenario is to perform an automatic reload upon transition. For implementation purpose, there is an additional control bit TGS_CTLx.TR_DPSLP2ACT which enables a reload when power transition happens from deep-sleep to active

The counter is reloaded by the value in TGS_RELx.RELOAD like other reload events. Transition between states happen in microseconds range while the timeout is 25ms. So, same reload value can be used without a major effect of transition time

If the TGS_CTLx.TR_DPSLP2ACT is '0', the counter resets upon the power transition without a reload. The counter is idle until the next reload event occurs and proceeds with normal active functionality

6. Constant PMBus input voltage switching level

The I^2C specification defines the switching thresholds for logic levels as a percentage of the supply voltage (VDD). Specifically, the thresholds are:Low Level: Below 30% of VDD (0.3 × VDD).High Level: Above 70% of VDD (0.7 × VDD).This means the switching thresholds dynamically scale with the supply voltage, making the I^2C spec inherently flexible for any VDD level. The PMBus (Power Management Bus) is built on top of the SMBus electrical layer. Therefore, it inherits the SMBus electrical characteristics,

including its fixed switching thresholds. The SMBus specification introduces fixed voltage thresholds for its logic levels, regardless of the supply voltage. These thresholds are:Low Level: ≤ 0.8 V (VIL).High Level: ≥ 2.1 V (VIH).This fixed requirement is stricter and not dependent on the supply voltage, unlike I^2C. GPIO pads GPIO_STD or GPIO_OVT do not meet the fixed switching level requirements of SMBus/PMBus. only TTL or CMOS level can be selected

27.7 Multiple Slave Address(MSA)

Multiple Slave Address is implemented in UART and I2C.

In SCB, instead of a single address with a corresponding mask, 6 slave addresses are been defined with corresponding mask definitions. The SCB slave address is considered a match if any of the six independent addresses correspond to the address transmitted by the master on the bus.

The MMIO register RX_MATCH defines the slave address.

In UART multiprocessor mode, all 8 bits are used.

In I2C slave mode, only bits 7 down to 1 are used. This reflects the organization of the first transmitted byte in an I2C transfer: the first 7 bits represent the address of the addressed slave, and the last 1 bit is a read/write indicator ('0': write, '1': read).

Slave device address mask is a mask that specifies which of the ADDR field bits in the ADDR field take part in the matching of the slave address:

MATCH = ((ADDR & MASK) == ("slave address" & MASK)).

For MSA pairs, if MASK1=0 it means this pair is disabled and will not affect address matching.

The MMIO register RX_MATCH is extended with one additional address with the corresponding MASK definition. In the same way, two additional RX_MATCH registers are added with the same bit definition. This gives the customer the possibility to define up to 6 independent slave addresses with corresponding mask definitions.

For internal clocked FIFO I2C slaves, register CTRL.ADDR_ACCEPT is to control whether the received matched I2C slave address can be pushed into RXFIFO.

27.8 SCB interrupts

SCB supports interrupt generation on various events. The interrupts generated by the SCB block vary depending on the mode of operation.

Table 295 SCB interrupts

Interrupt	Functionality	CPU Active/System Deep Sleep	Registers
interrupt_master	master and SPI master functionality	CPU Active	SCBx_INTR_M, SCBx_INTR_M_SET, SCBx_INTR_M_MASK, SCBx_INTR_M_MASKED
interrupt_slave	I2C slave and SPI slave functionality	CPU Active	SCBx_INTR_S, SCBx_INTR_S_SET, SCBx_INTR_S_MASK, SCBx_INTR_S_MASKED
interrupt_tx	UART transmitter and TX FIFO functionality	CPU Active	SCBx_INTR_TX, SCBx_INTR_TX_SET, SCBx_INTR_TX_MASK, SCBx_INTR_TX_MASKED
interrupt_rx	UART receiver and RX FIFO functionality	CPU Active	SCBx_INTR_RX, SCBx_INTR_RX_SET, SCBx_INTR_RX_MASK, SCBx_INTR_RX_MASKED
interrupt_i2c_ec	Externally-clocked I2C slave functionality	System Deep Sleep	SCBx_INTR_I2C_EC, SCBx_INTR_I2C_EC_MASK, SCBx_INTR_I2C_EC_MASKED
interrupt_spi_ec	Externally-clocked SPI slave functionality	System Deep Sleep	SCBx_INTR_SPI_EC, SCBx_INTR_SPI_EC_MASK, SCBx_INTR_SPI_EC_MASKED
Interrupt_tgs	Timer generation support	CPU Active	SCBx_INTR_TGS, SCBx_INTR_TGS_SET, SCBx_INTR_TGS_MASK, SCBx_INTR_TGS_MASKED

27.8.1 SPI interrupts

SPI interrupts can be classified as Master interrupts, Slave interrupts, TX interrupts, RX interrupts, and externally-clocked (EC) mode interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources

are true. The SCB also provides an interrupt cause register (SCBx_INTR_CAUSE) that can be used to determine the interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bit field. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the PSC3M/P6 MCU registers reference manual. The SPI supports interrupts on the following events:

- SPI master interrupts (SCBx_INTR_M)
  - SPI master transfer done (SPI_DONE)
- SPI Slave interrupts (SCBx_INTR_S)
  - SPI slave deselected after a write EZSPI transfer occurred (SPI_EZ_WRITE_STOP)
  - SPI slave deselected after any EZSPI transfer occurred (SPI_EZ_STOP)
  - SPI Bus Error – Slave deselected unexpectedly in the SPI transfer. The firmware may decide to clear the TX and RX FIFOs for this error. (SPI_BUS_ERROR)
- SPI TX (SCBx_INTR_TX)
  - TX FIFO has fewer entries than the value specified by SCBx_TX_FIFO_CTRL.TRIGGER_LEVEL (TRIGGER)
  - TX FIFO is not full (NOT_FULL)
  - TX FIFO is empty (EMPTY)
- SPI Timer generation support (SCBx_INTR_TGS)
  - Timeout counter 0 underflow (TGS0)
  - Timeout counter 1 underflow (TGS1)
  - Timeout counter 2 underflow (TGS2)

27.8.2 UART interrupts

UART interrupts can be classified as TX interrupts and RX interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources are true. The SCB also provides an interrupt cause register (SCBx_INTR_CAUSE) that can be used to determine the interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bitfield. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the PSC3M/P6 MCU registers. The UART blocks generate interrupts on the following events:

- UART TX (SCBx_INTR_TX)
  - TX FIFO has fewer entries than the value specified by SCBx_TX_FIFO_CTRL.TRIGGER_LEVEL (TRIGGER)
  - TX FIFO is not full (NOT_FULL)
  - TX FIFO is empty (EMPTY)
  - TX FIFO overflow (OVERFLOW)
  - TX FIFO underflow (UNDERFLOW)
  - TX received a NACK in SmartCard mode (UART_NACK)
  - TX done. This happens when the UART completes transferring all data in the TX FIFO and the last stop field is transmitted (both TX FIFO and transmit shifter register are empty). (UART_DONE)
  - Arbitration lost (in LIN or SmartCard modes) (UART_ARB_LOST)
- UART RX (INTR_RX)
  - RX FIFO has more entries than the value specified by SCBx_RX_FIFO_CTRL.TRIGGER_LEVEL (TRIGGER)
  - RX FIFO is not empty (NOT_EMPTY)

  - RX FIFO is full (FULL)
  - RX FIFO overflow (OVERFLOW)
  - RX FIFO underflow (UNDERFLOW)
  - Frame error in received data frame (FRAME_ERROR)
  - Parity error in received data frame (PARITY_ERROR)
  - LIN baud rate detection is completed (BAUD_DETECT)
  - LIN break detection is successful (BREAK_DETECT)
- UART Timer generation support (SCBx_INTR_TGS)
  - Timeout counter 0 underflow (TGS0)
  - Timeout counter 1 underflow (TGS1)
  - Timeout counter 2 underflow (TGS2)

27.8.3 I2C interrupts

I2C interrupts can be classified as Master interrupts, Slave Interrupts, TX interrupts, RX interrupts, and Externally-clocked (EC) mode interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources are true. The SCB also provides an interrupt cause register (SCBx_INTR_CAUSE) that can be used to determine the interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bit field. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the PSC3M/P6 MCU registers reference manual. The I2C block generates interrupts for the following conditions:

- I2C Master (SCBx_INTR_M)
  - I2C master lost arbitration (I2C_ARB_LOST)
  - I2C master received NACK (I2C_NACK)
  - I2C master received ACK (I2C_ACK)
  - I2C master sent STOP (I2C_STOP)
  - I2C bus error (unexpected stop/start condition detected) (I2C_BUS_ERROR)
- I2C Slave (SCBx_INTR_S)
  - I2C slave lost arbitration (I2C_ARB_LOST)
  - I2C slave received NACK (I2C_NACK)
  - I2C slave received ACK (I2C_ACK)
  - I2C slave received Write STOP (I2C_WRITE_STOP)
  - I2C slave received STOP (I2C_STOP)
  - I2C slave received START (I2C_START)
  - I2C slave address matched (I2C_ADDR_MATCH)
  - I2C slave general call address received (I2C_GENERAL)
  - I2C bus error – unexpected stop/start condition detected (I2C_BUS_ERROR)
  - I2C slave received STOP at the end of multiple restarts when address matches at least once (I2C_STOP_ANY)
- I2C TX (SCBx_INTR_TX)
  - TX FIFO has fewer entries than the value specified by SCBx_TX_FIFO_CTRL.TRIGGER_LEVEL (TRIGGER)
  - TX FIFO is not full (NOT_FULL)
  - TX FIFO is empty (EMPTY)
  - TX FIFO overflow (OVERFLOW)
  - TX FIFO underflow (UNDERFLOW)
  - EZ memory (EZ data access) blocked (BLOCKED)
- I2C RX (SCBx_INTR_RX)
  - RX FIFO has more entries than the value specified by SCBx_RX_FIFO_CTRL.TRIGGER_LEVEL (TRIGGER)
  - RX FIFO is not empty (NOT_EMPTY)
  - RX FIFO is full (FULL)
  - RX FIFO overflow (OVERFLOW)
  - RX FIFO underflow (UNDERFLOW)
- I2C Externally-clocked (SCBx_INTR_I2C_EC)
  - Wake up request on address match (WAKE_UP)

  - I2C STOP detection at the end of each transfer (EZ_STOP)
  - I2C STOP detection at the end of a write transfer (EZ_WRITE_STOP)
  - I2C STOP detection at the end of a read transfer (EZ_READ_STOP)
  - I2C STOP detection at the end of multiple restarts when address matches at least once (EZ_STOP_ANY)
- I2C Timer generation support (SCBx_INTR_TGS)
  - Timeout counter 0 underflow (TGS0)
  - Timeout counter 1 underflow (TGS1)
  - Timeout counter 2 underflow (TGS2)

28 Improved inter integrated circuit (I3C)

***Note**: The portions of Improved inter integrated circuit (I3C) section copyrighted ©2023 by Synopsys, Inc. All rights reserved. Used with permission.*

28.1 I3C overview

The Improved inter integrated circuit (I3C) is a standardized communication protocol defined by the MIPI (mobile industry processor interface) alliance. This specification is developed to overcome I2C limitations such as limited speed, external signals needed for interrupts, automatic detection of the devices connected on the bus, and so on, while remaining power-efficient.

I3C is useful for designing mobile wireless products since it provides low cost, low power, fast, and limited number of digital signals needed for interfacing the sensors.

The I3C section follows the terminology of MIPI I3C Basic v1.1.1 (Controller/Target).

28.1.1 Block diagram

The I3C system is designed to ensure reliable transmission and reception of data. The I3C protocol has additional features and deviates from the legacy I2C in many ways, but still maintains backward compatibility with legacy I2C devices with respect to pin connections and frame format. A device on the I3C bus can either be a primary controller, a secondary controller, a I3C target, or a I2C target. Since I3C is a multi-controller protocol, there may be several controllers on a bus, though only one device can act as an active controller at a specific time. The primary controller initializes the bus and starts as the active controller. A secondary controller can then be handed over the control of the bus (or request it) and then become the active controller. Therefore, an I3C bus can have a primary controller, multiple secondary controllers, I3C targets, and also I2C targets.

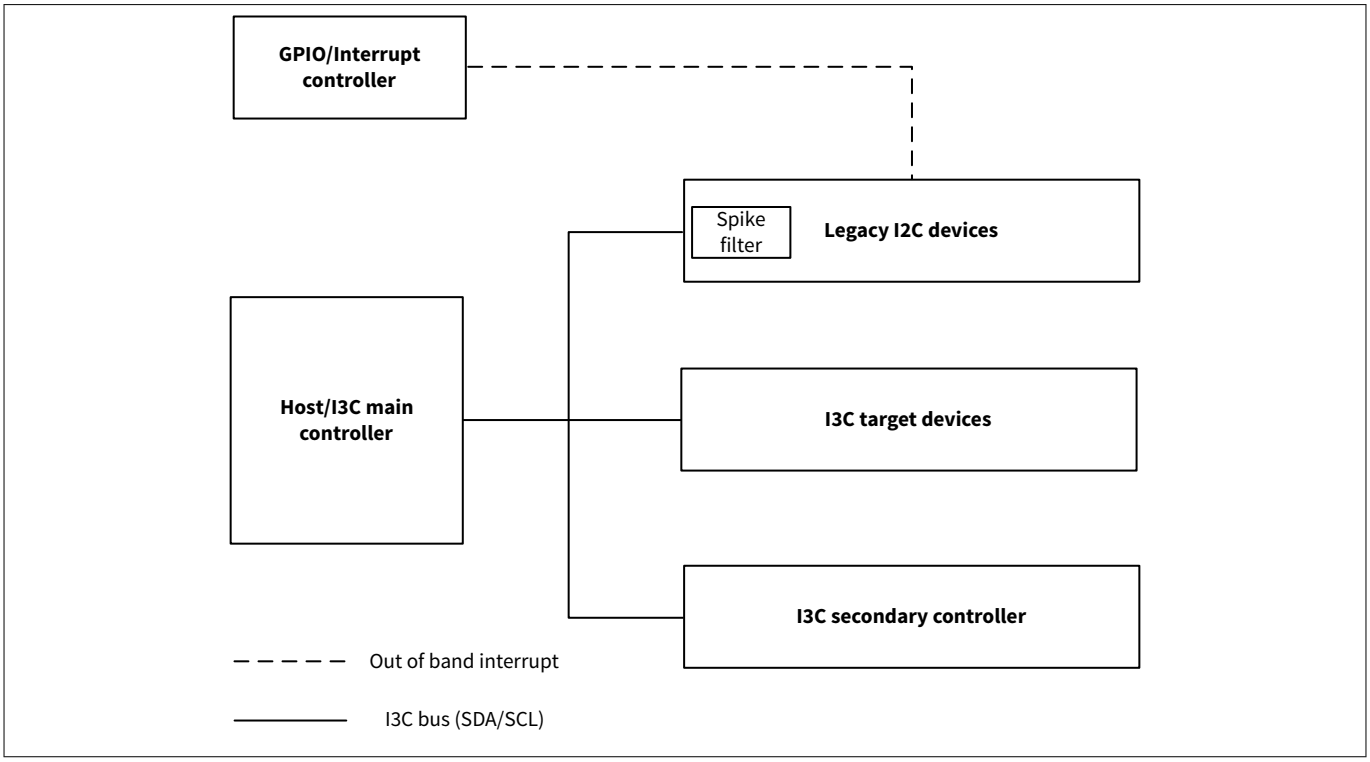


Figure 341 I3C system diagram

The following figure represents the overall architecture of the I3C block:

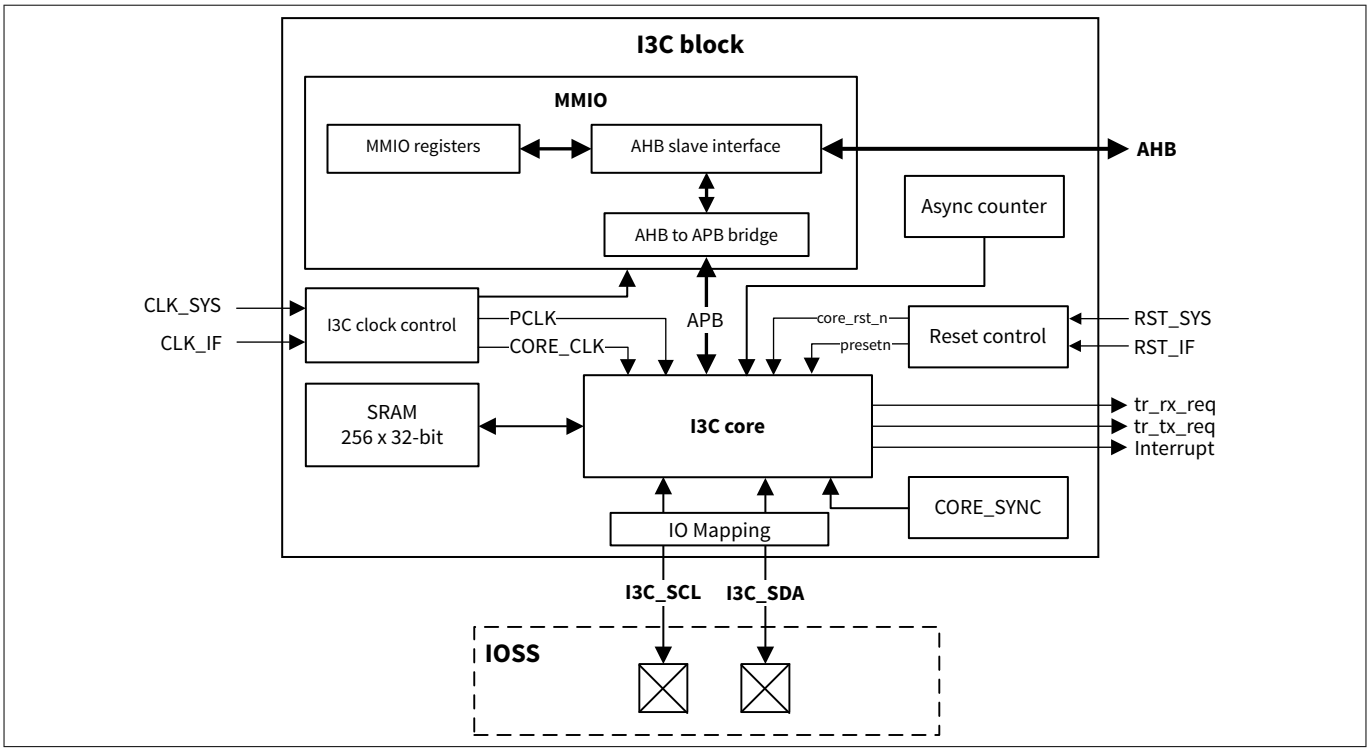


Figure 342 I3C architectural block

The CLK_SYS clock interface is used to generate clocks for controlling the AHB Interface, the AHB/APB Bridge, the MMIO registers, and for providing the input to the I3C block.

The CLK_IF clock interface is used to generate the CORE_CLK interface of the I3C Core. This clock is used to control all I3C-related functionality in the controller mode of operation, such as generating the I3C SCL clock and SDA data. The CORE_CLK should always have at least 10x the desired frequency on the SCL line, since I3C CORE requires at least 5 CORE_CLK cycles to generate a high or low phase of SCL. The exact value can be set using the SCL_I3C_PP_TIMING register.

The async counter implements a counter used by the I3C device when operating in the asynchronous timing control mode 0. In this mode, the device needs to maintain a cycle-accurate tracking of events that happen on the device and on the I3C bus (for example: IBI transmitted/received, IBI acknowledged). The counter can be enabled, disabled, or reset via the I3C_TIMING_CNT_CTRL register.

28.2 Features

Supported features:

- I3C primary controller
- I3C secondary controller and target
- I3C SCL frequency up to 12.5 MHz (SDR data rate 12.5 Mbps, HDR-DDR data rate at 25 Mbps)
- I3C In-band interrupts (IBI)
- I3C dynamic address allocation (DAA)
- I3C hot-join capability
- I3C device characteristics register (DCR)
- I3C common command code (CCC)
- I3C controller mode allows addressing up to 11 target devices
- I3C asynchronous time control support (Timing control Mode 0)
- I3C error detection of CE0, CE2, CE3, and TE0 to TE5

- Target interrupt request (TIR) with payload
- Target RESET CCC
- Vendor-specific CCC
- I3C target reset pattern generation/detection
- Packet-error check (PEC)
- SRAM 1 KB (256x32-bit) for command/response queues and transmit/receive data buffers
- Backward compatible with I2C: works with legacy I2C devices which use Fast-mode (400 KHz) and Fast-mode Plus (1 MHz) with the 50 ns spike filter,However, it does not work with devices that use other I2C modes, devices without the spike filter, or devices that adjust the clock speed

Not supported:

- I3C HDR-TSP/TSR modes
- I3C CDR (clock data recovery)
- I3C address arbitration optimizations
- Bridge device capability
- I2C target role, due to lack of 50 ns spike filter
- I3C synchronous time control support
- Host controller interface (HCI)

28.3 Terms and definitions

Dynamic address assignment (DAA)

The I3C protocol defines a mechanism to automatically discover devices present on the bus, their capabilities, and the dynamic address assignment (DAA) functionality. I3C, configured as an active controller, is responsible for performing a DAA procedure, to provide a unique dynamic address to each I3C device connected to the I3C Bus.

The active controller provides a dynamic address to a device for the following conditions:

1. Upon initialization of the I3C Bus
2. When the device is connected to an already configured I3C Bus (indicated through Hot-join or other methods)

Once a device receives a dynamic address, this assigned dynamic address is used in all the device's subsequent transactions on the I3C Bus.

Hot-join

In the Hot-join mechanism, the target joins or connects with the configured I3C Bus. The I3C active controller assigns a unique dynamic address to each device, either when the bus is initialized or when a new device is connected to a configured I3C Bus.

The Hot-join mechanism is similar to a USB hot plug. This mechanism can be useful in conditions such as when the target wakes up from power-up mode or is physically inserted in the I3C Bus.

In-band interrupts

The I3C protocol allows targets to generate events on their own and allows them to take temporary control of the bus. This mechanism is called In-band interrupt (IBI), it allows devices to generate interrupts without requiring an external signal. The I3C protocol also gives a provision of supporting in-band interrupts within the two-wire interface, which reduces the device pin count and signal paths. The IBI is initiated by the I3C target device to the current controller. Additionally, I3C uses the IBI for generating a Hot-join request (for requesting the host to assign I3C dynamic address) and the controller requests for bus ownership.

I3C common command code (CCC)

The I3C standardized common command code (CCC) is used for anything related to controlling the bus management, for event enable/disable, activity states, payload management, feature management (DAA, controller request, HDR, and timing control), test modes, extensible space (MIPI or vendor-specific), and to broadcast the I3C address (0x7E).

28.4 I/O pad connections

The two-wire I3C serial interface consists of the following:

1. Serial data line (I3C_SDA)
2. Serial clock (I3C_SCL)

The I3C_SDA is used for bidirectional data transfer between the controller and the target and the SCL can be used by the controller for transmitting a clock (to which SDA is synchronous). SCL can have a clock speed of up to 12.5 MHz, enabling peak transfer speeds of 12.5 Mbps in single data rate (SDR) mode or up to 25 Mbps in HDR-DDR mode (which uses rising and falling edges for transferring data).

The I/O control register, CFG_OUT3, extends the drive mode control of the GPIO.

1. When CFG bit is 0, then CFG_OUT3 controls the pull-up/pull-down and drive of the I/O pin

   ***Note**: CFG_OUT3 only controls the decoding for the port when the corresponding bit in CFG = 0 (the default)*

2. When CFG_OUT3[N] is 8 in I3C mode, then CFG_RES[N] controls the pull-down value

I3C_SCL operates in push-pull mode, while I3C_SDA operates in both push-pull and open-drain modes, depending on the following conditions:

- Dynamic address assignment
- Broadcast address transmission phase
- In-band interrupt
- ACK/NACK in 9th bit

28.5 Functional description

The following section describes all the functions of the I3C block.

28.5.1 Operating modes

A peripheral controller provides the following operation mode:

Primary controller

I3C advocates a hierarchical bus architecture, where only one controller (active controller) controls the I3C bus at any given time. It is mandatory to have a primary controller, and secondary controllers on the I3C bus are optional.

The primary controller in the I3C system performs dynamic address assignment (DAA). This controller is responsible for generating the clock, issuing the commands, and controlling the data transfer. Each I3C device has a unique address that is assigned by the controller through DAA during cold power-up or during a Hot-join of the I3C device to the system. The primary controller is the active controller, which has ownership of the bus during cold power-up.

Secondary controller

The secondary controller is an I3C instance, capable of both controller and target functionality. It comes up as a target upon power-on reset. The secondary controller must get the ownership of the I3C Bus to become an

active controller, before initiating any transfer to its associated targets. The secondary controllers obtain the dynamic address from the primary controller, after they are enabled through any of the address assignment.

I3C target

The target is responsible for responding to the commands to the I3C Controller, transmitting or receiving data to/from the controller, and acknowledging to the controller that the transfer is successful.

28.5.2 Bus configurations

The I3C core supports three types of I3C Bus configurations as defined by the MIPI I3C specification, based on the type of devices present on the bus:

Pure bus

This configuration has only I3C devices on the bus.

Mixed fast bus

This configuration has both I3C devices and legacy I2C targets on the bus. In this case, the legacy I2C targets are restricted to the ones that are generally permissible, such as the following:

- Target-only
- No clock stretching
- Have a true I2C 50 ns glitch filter on SCL

Mixed slow/limited bus

This configuration has both I3C devices and legacy I2C targets on the bus. In this case, the legacy I2C targets are restricted to the ones that are selectively backward compatible with the I2C standard, such as the following:

- Target-only
- No clock stretching
- Do not have a true I2C 50 ns glitch filter on SCL

28.5.3 Speed modes

This I3C block supports the following I2C/I3C speed modes:

Legacy I2C speed modes

- Fast-mode
- Fast-mode Plus
- Standard mode, High-speed mode, and Ultra-fast mode are not supported

I3C speed modes

- Single data rate (SDR) mode enabling peak transfer speed of 12.5 Mbps with SCL can have a clock speed of up to 12.5 MHz
- High data rate (HDR) modes enabling peak transfer speed of up to 25 Mbps in high data rate-double data rate (HDR-DDR) mode (which uses rising and falling edges for transferring data)

28.5.4 Controller data structures

28.5.4.1 Command data structure

The command queue of I3C accepts the following four types of data structures to initiate a transfer on the I3C Bus:

- Transfer command data structure
- Transfer argument data structure
- Short data argument data structure
- Address assignment command data structure

If a transfer consists of a payload of at least one byte, then you must write either the transfer argument or the short data argument into the command queue, prior to the transfer command.

CMD_TYPE / BIT OFFSET	31	30	29	28	27	26	25	24	23:21	20:16	15	14:8	7	6	5:3	2:0
Transfer Command	PEC	TOC	RESV	RnW	SDAP	ROC	DBP	RESV	SPEED	DEV_INDX	CP	CMD		TID		CMD_ATTR (0)
Transfer Argument	DATA_LENGTH										DB	RESV				CMD_ATTR (1)
Short Data Argument	DATA_BYTE_2								DATA_BYTE_1		DATA_BYTE_0/DB	RESV		BYTE_STRB		CMD_ATTR (2)
Address Assignment Command	RESV	TOC	RESV			ROC	DEV_COUNT			DEV_INDX	RESV	CMD		TID		CMD_ATTR (3)

Figure 343 I3C command structure

28.5.4.1.1 Transfer command data structure

In the controller mode of operation, the transfer command is used to initiate CCC and private transfers. For transfers with data payload, additional data structure (either transfer argument or short data argument) is used to provide payload details.

Table 296 Data structure for transfer command

Field name	Bit-field	Bit description for transfer command
CMD_ATTR	2:0	**Command attribute** defines the command type and its Bit-Field format. • 0: Transfer command • 1: Transfer argument • 2: Short data argument • 3: Address assignment command • 4-7: Reserved

(table continues...)

Table 296 (continued) Data structure for transfer command

Field name	Bit-field	Bit description for transfer command
TID	6:3	**Transaction ID** This field is used as the identification tag for the commands. The I3C controller returns this ID along with the response upon completion or upon error. • 4'b0000 - 4'b0111 - User-defined TID • 4'b1000 - 4'b1111 - Reserved for I3C controller
CMD	14:7	**Transfer command** This field is used to define the transfer command type. The field can be programmed to: **1.** 8-bit common command code for CCC transfers **2.** 7-bit command code for HDR-TS or HDR-DDR transfers (bit[14] is reserved)
CP	15	**Command present** This bit is used to control whether the transfer should be initiated with the transfer command represented in the 'CMD' filed or not. • 0 - CMD field is not valid • 1 - CMD field is valid This bit is applicable for CCC and HDR transfers.
DEV_INDX	20:16	**Device index** This field is used to refer the device address table for getting the target address. DEV_INDX field points to the offset address of device address table.

(table continues...)

Table 296 (continued) Data structure for transfer command

Field name	Bit-field	Bit description for transfer command
SPEED	23:21	**Speed** This field is used to indicate the speed in which the transfer should be driven. Values (I3C Mode): • 0: SDR0 • 1: SDR1 • 2: SDR2 • 3: SDR3 • 4: SDR4 • 5: HDR-TS • 6: HDR-DDR • 7: I2C FM (JEDEC broadcast CCC only) ***Note**: The HDR-DDR and HDR-TS are supported only if the transfer command structure is set-up to use transfer argument and NOT short data argument.* Values (I2C mode): • 0: I2C FM • 1: I2C FM+ • 2 -7: Reserved
RESV	24	Reserved
DBP	25	**Defining byte present** DBP indicates whether the current CCC command is with defining byte or not. This bit field is valid only when CP (command present) bit is enabled, otherwise this bit is ignored • 0: Defining Byte is not present • 1: Defining Byte is present This bit is applicable only for broadcast and directed SDR CCC transfers

(table continues...)

Table 296 (continued) Data structure for transfer command

Field name	Bit-field	Bit description for transfer command
ROC	26	**Response on completion** This field indicates whether the response status is required or not, after the execution of this command for the successful transfer. • 1 - Response status is required • 0 - Response status is not required ***Notes**:* ***1.*** *The exception to the control is that the response status gets generated when the transfer has encountered an error condition* ***2.*** *It is recommended that the ROC bit is always set to 1 for the Read commands (RnW=1), so that the number of data received is indicated (through DATA_LENGTH field in response port) if the target terminates early than the controller*
SDAP	27	**Short data argument present** This field indicates whether the command written prior to the base command should be treated as short data argument or the transfer argument. • 0 - Prior written command is transfer argument • 1 - Prior written command is short data argument
RnW	28	**Read and write** This bit controls whether a Read or write transfer is performed. • 0 - Write transfer • 1 - Read transfer ***Note**:* *In HDR transfers, this bit is used to set the Read/Write flag of the HDR-TS/HDR-DDR command code.*
TGT_RST	29	**Target reset pattern generation** This bit indicates whether the issued command generates target reset pattern at the end of transfer. This bit is enabled only with • TOC field set to 1 • SPEED field set to 0 to 4 • CP set to 1 • CMD set to RSTACC ***Note**:* *It is recommended to use this bit only for RSTACT CCC's.*

(table continues...)

Table 296 (continued) Data structure for transfer command

Field name	Bit-field	Bit description for transfer command
TOC	30	**Termination on completion** This bit controls whether a STOP need to be issued after the completion of the transfer or not. • 1 - STOP issued after this transfer • 0 - The next transfer starts with RESTART condition
PEC	31	**Parity error check** This bit enables generation and validation of PEC byte for SDR CCC and private transfers. • 0: PEC check is disabled • 1: PEC check is enabled ***Note**: This bit is valid only for SDR Transfers and not for HDR transfers.*

28.5.4.1.2 Transfer argument data structure

Transfer argument data structure is used to provide payload related information when the payload size is greater than 3 bytes.

Table 297 Command data structure for transfer argument

Field name	Bit-field	BIT description for transfer Argument
CMD_ATTR	2:0	**Command attribute** defines the command type and its Bit-field format. • 0: Transfer command • 1: Transfer argument • 2: Short data argument • 3: Address assignment command • 4-7: Reserved
RESV	7:3	Reserved
DB	15:8	**Defining byte value** DB indicates the 8-bit defining byte to be transferred in the CCC transfer. This byte is valid only when both CP and DBP bits are enabled, otherwise the controller ignores this byte.
DL	31:16	**Data length** This field is used to indicate the data length of the transfer. For CCC transfers, it is expected that the correct length be programmed. The controller block deals with all CCCs transparently.

28.5.4.1.3 Short data argument data structure

Short data argument data structure is used to provide payload related information for the TX transfers when the payload size is less than or equal to 3 bytes.

Table 298 Command data structure for Short data argument

Field name	Bit-field	BIT description for short data argument
CMD_ATTR	2:0	**Command attribute** defines the command type and its Bit-field format. • 0: Transfer command • 1: Transfer argument • 2: Short data argument • 3: Address assignment command • 4-7: Reserved
BYTE_STRB	5:3	**Byte strobe** This field is used to select the valid data bytes of the short data argument. • BYTE_STRB[0] - Data Byte -0 Valid qualifier • BYTE_STRB[1] - Data Byte -1 Valid qualifier • BYTE_STRB[2] - Data Byte -2 Valid qualifier Valid combinations = 3'b001, 3'b011 and 3'b111
RESV	7:6	Reserved
DATA_BYTE_0/DB	15:8	**Data Byte -0/Defining Byte** This field is used for storing the data Byte -0. DB indicates the 8-bit defining Byte to be transferred in the CCC transfer. This byte is valid only when both CP and DBP bits are enabled, otherwise the controller ignores this byte. ***Notes**:* *When defining byte (DBP) is enabled in short data argument* ***1.*** *The BYTE_STRB[0] bit is treated to be always enabled whether you set it or not* ***2.*** *The maximum possible payload bytes to be sent is '2' due to the first byte being occupied by the DBP*
DATA_BYTE_1	23:16	**Data byte -1** This field is used for storing the Data byte -1.
DATA_BYTE_2	31:24	**Data byte -2** This field is used for storing the Data byte -2.

***Note**: Short data argument is not supported for HDR transfers be cause in HDR modes, a maximum of two words can be transferred in this format.*

28.5.4.1.4 Address assignment command data structure

In the Controller mode of operation, the address assignment command is used to initiate ENTDAA and SETDASA transfers.

Table 299 Command data structure for address assignment

Field name	Bit-field	BIT description for address assignment
CMD_ATTR	2:0	**Command attribute** defines the command type and its Bit-field format. • 0: Transfer command • 1: Transfer argument • 2: Short data argument • 3: Address assignment command • 4-7: Reserved
TID	6:3	**Transaction ID** This field is used as the identification tag for the commands. The I3C controller returns this ID along with the response upon completion or upon error. • 4'b0000 - 4'b0111 - User-defined TID • 4'b1000 - 4'b1111 - Reserved for I3C controller
CMD	14:7	**Address assignment CCC** This field is used to define the address assignment command type used in the transfer. This field is used for representing the ENTDAA or SETDASA common command codes.
RESV	15	Reserved
DEV_INDX	20:16	**Device index** This field is used to indicate the start pointer of the device table from where the dynamic address is to be picked and assigned to the I3C devices.
DEV_COUNT	25:21	**Device count** This field is used to represent the number of devices to be assigned with the dynamic address.
ROC	26	**Response on completion** This field indicates whether the response status is required or not, after the execution of this command for the successful transfer. • 1 - Response status is required • 0 - Response status is not required ***Note**: The exception to this control is that the response status gets generated when the transfer has encountered an error condition.*
RESV	29:27	Reserved

(table continues...)

Table 299 (continued) Command data structure for address assignment

Field name	Bit-field	BIT description for address assignment
TOC	30	**Termination on completion** This field controls whether a STOP need to be issued after the completion of the transfer or not. • 1 - STOP issued after this transfer • 0 - The next transfer starts with RESTART condition
RESV	31	Reserved

***Note**: This command is not used for the address assignment commands 'SETAASA' Broadcast CCC transfer and 'SETNEWDA' directed CCC transfer. The application has to use regular transfer command for issuing SETAASA or SETNEWDA command.*

***Note**: Dynamic addresses used by the controller during address assignment are picked up from device address table. During address assignment, the first winning device is assigned the dynamic address programmed at Index 0 of device address table, the next device that wins arbitration gets the address programmed at index 1, and so on.*

28.5.4.2 Response data structure

The I3C block returns the status of the transfer command into the response queue with this data structure.

BIT OFFSET / RESP_TYPE	31:28	27:24	23:16	15:0
Response Data	ERR_STS	TID	CCCT	DL

Figure 344 Response data structure

Table 300 Response data structure bit description

Field name	Bit-field	Bit description for response data
DL	15:0	**Data length** For Write transfers, this field represents the remaining data length of the transfer if the transfer is terminated early (remaining data length = requested data length - transferred data length). For Read transfers, this field represents the actual amount of data received in bytes. For Address assignment command, this field represents the remaining device count.
CCCT	23:16	**CCC/HDR header type** This field represents the CCC type of the received vendor extension CCC packet or the HDR header of the received HDR transaction. This field indicates the CCC type when the TID is set to 4'b1111 (reserved for all other transactions). During the controller transactions and target non-HDR transactions, this field returns 8'b0000_0000 and can be considered as do not care.
TID	27:24	**Transaction ID** This field is used as the identification tag for the commands. The I3C controller returns the ID received through commands. • 4'b0000 - 4'b0111 - User-Defined TID (specified in the Transfer/Address assignment command) • 4'b1000 - Controller write data status (as target only) • 4'b1111 - DEFTGTS status (as target only) • 4'b1001 - 4'b1110 - Reserved for I3C controller

(table continues...)

Table 300 (continued) Response data structure bit description

Field name	Bit-field	Bit description for response data
ERR_STS	31:28	**Error type** Defines the error type of the processed command or received vendor extension CCC packet (Target mode). • 0: No error • 1: CRC error • 2: Parity error • 3: Frame error • 4: I3C Broadcast address NACK error • 5: Address NACK'ed. This bit is set in case the target NACKs for dynamic address assignment during ENTDAA process • 6: Receive buffer overflow/Transmit buffer underflow (Only for HDR transfers) • 7: Reserved • 8: Transfer aborted • 9: I2C Target write data NACK error • 10-11: Reserved • 12: PEC error. This bit is set if PEC byte validation error occurs in read transfers when TRANSFER_COMMAND[PEC] bit is set to 1 • 13-15: Reserved

28.5.5 Target data structures

28.5.5.1 Transmit command data structures

The transmit command structure in the I3C target is used to respond with data for a private read command from the active controller.

Table 301 Transmit command data structure

Bits	Name	Memory access	Description
31:16	DATA_LENGTH	W	**Data length** This field is used to indicate the data length of the controller read transfer
15:6	RSVD	W	Reserved for future use

(table continues...)

Table 301 (continued) Transmit command data structure

Bits	Name	Memory access	Description
5:3	TID	W	**Transmit transaction ID** This field is used as the identification tag for the command. ***Note:*** *TID value '7' is reserved for vendor specific CCC's and recommended to avoid setting TID value to 7 for private transfers.*
2:0	CMD_ATTR	W	**Command attribute** Defines the command attribute and its field format. • 0 - Transmit command without IBI • 1-7 - Reserved

28.5.5.2 Response data structure

The response structure I3C APB Target is used to indicate the completion of the transmit command (private controller read operation) or the completion of controller write operation (private controller write operation).

Table 302 Response data structure

Bits	Name	Memory access	Description
31:28	ERR_STATUS	R	• 0: No error • 1: CRC error (controller write in DDR mode) • 2: Parity error (controller write in both DDR and SDR mode) • 3: Frame error (HDR mode controller write) • 6: Underflow/Overflow error • 8: SDA is released from the stuck state after the prescribed amount of time defined by register RELEASE_SDA_TIMING • 10: Controller early termination
27	RX_RSP	R	**Transaction type** This field is used to identify the type of transaction: • 0: Transmit response • 1: Receive response

(table continues...)

Table 302 (continued) Response data structure

Bits	Name	Memory access	Description
26:24	TID	R	Transmit transaction ID This field is used as the identification tag for the transmit command. ***Note**: TID value '7' indicates the response is meant for vendor specific CCC's or DEFTGTS CCC transfers.*
23:16	CCC_HDR_HEADER	R	Command code of CCC or HDR command code • 8'h00 - SDR transfer • CCC Code - Vendor specific CCC's • Others - HDR Command code or in the case of secondary controller, whenever DEFTGTS CCC is received, the DEFTGTS CCC command code is reflected in this field. The application comes to know that this field reflects the CCC command code if TID field is set to 3'b111 along with RX_RESP field set to 1'b1
15:8	Defining Byte/MSB of Data Length	R	Defining Byte or MSB of data length If TID field is set to 3'b111, Indicates the defining Byte value received for the vendor specific directed Write/Read CCC transfers. If TID field is not set to 3'b111, indicates the MSB of the data length in bytes of controller Write transfer or the remaining length of controller Read transfer.
7:0	LSB of Data Length	R	If TID field is not set to 3'b111, Indicates data length in bytes of controller Write transfer or the remaining length of controller Read transfer (Controller termination or target underrun). If TID field is set to 3'b111, Indicates data length received or transmitted in bytes for vendor specific CCC transfers. In case of DEFTGTS which is indicated by TID and RX_RESP field all set to one, this field indicates the device count to the application.

***Note**: **Target data length:** The data length field in response of a private read/write transfer that has experienced any error (has the appropriate error status flag set in response), does not accurately reflect the number of bytes sent or received before the occurrence of the error. Since the error handling flow recommends that the appropriate queues be flushed, the data length reported has no particular significance.*

28.5.6 Bus communication protocols

28.5.6.1 Single data rate (SDR) transfers

In SDR mode, a single bit is sent on each SCL clock. All data is transmitted in byte format, with no limit on the number of bytes transferred per data transfer. After the controller sends the address and R/W bit or the controller transmits a byte of data to the target, the target-receiver must respond with the acknowledge signal (ACK). When a target-receiver does not respond with an ACK pulse, the controller aborts the transfer by issuing a STOP condition. The target must leave the SDA line high so that the controller can abort the transfer.

The single data rate transfers are initiated by the application through the command, as explained in the "Command data structure". Based on the command initiated by the application and device address pointed by the DEV_INDX field initiates command, the controller initiate the transfers on the bus. Table 303 illustrates the decoded transfer type based on transfer command and device address table.

Table 303 Transfer type decoding based on command and device address table

Transfer command				Device address table	Transfer type
CP	CMD[14]	SPEED	RnW	LEGACY_I2C_DEVICE	
0	NA	0 to 4	0	0	I3C private write transfer
0	NA	0 to 4	1	0	I3C private read transfer
1	0	0 or 7	0	0	I3C broadcast CCC write transfer
1	1	0 or 7	0	0	I3C directed CCC write transfer
1	1	0 or 7	1	0	I3C directed CCC read transfer
0	NA	0 or 1	0	1	I2C private write transfer
0	NA	0 or 1	1	1	I2C private read transfer
Others					Illegal combination

The broadcast and directed common command code (CCC) transfers are used for the bus management procedures like:

- Receiving the device characteristics
- Setting the maximum write or read data length
- Enabling or disabling the event generation in the I3C devices
- Informing the bus characteristics to all the existing devices
- Entering the high data rate transfer modes

28.5.6.1.1 Broadcast CCC write transfers

The Broadcast CCC write transfers are initiated on the bus based on the I3C_CORE_COMMAND_QUEUE_PORT register settings as shown in Table 304.

Table 304 Broadcast CCC write transfer programming values

Command attribute	Bit-field	Programmed value	Description
Transfer command	CP	1	Indicates to the controller to consider the CMD field
	CMD[14]	0	Indicates the controller that the transfer is broadcast CCC transfer
	CMD[13:7]	0x00 – 0x7F	Indicates the CCC command to be transferred
	DEV_INDX	NA	This field is not used since the transfer is broadcast CCC
	SPEED	0 (SDR0) or 7 (I2C FM)	Indicates the controller that the transfer should go in SDR0 speed/mode or I2C FM Speed ***Note**: All CCC transfers are initiated with SDR0 Speed in I3C bus topology. CCC transfers can be either initiated with SDR0 or I2C FM speed based on the context of the bus in JEDEC bus topology.*
	DBP	0 or 1	Indicates whether defining byte is present for broadcast CCC transfer or not. • 0: Defining byte is not present • 1: Defining byte is present
	SDAP	0 or 1	• 0: Indicates to consider the transmit data from the Transmit FIFO if Rnw is set to 0 • 1: Indicates the controller to consider the transmit data from the command if RnW is set to 0
	RnW	0	Indicates the transfer is either write or read transfer. • 0 - Indicates that the transfer is a write transfer • 1 - Indicates the transfer is a read transfer
	TGT_RST	0 or 1	Indicates whether the controller should generate target reset pattern at the end of broadcast RSTACT CCC transfer. • 0: Target reset pattern is not generated • 1: Target reset pattern is generated
	PEC	0 or 1	Indicates whether packet error check is enabled for broadcast CCC transfer. • 0: PEC check is disabled • 1: PEC check is enabled
Transfer argument (SDAP=0)	DATA_LENGTH	0 - 65535	Indicates the transfer length of the transfer

(table continues...)

Table 304 (continued) Broadcast CCC write transfer programming values

Command attribute	Bit-field	Programmed value	Description
Short Data Argument (SDAP=1)	BYTE_STRB	0, 1, 3, 7	Indicates the respective data bytes of immediate command are valid

The data in Tx-FIFO (transfer argument) or data Bytes (short data argument) are not required for some broadcast CCC's which does not have payload data Indicated through data length (transfer argument) or BYTE_STRB (short data argument). If the Broadcast CCC does not consist of payload, then you must indicate it with zero in either data length or BYTE_STRB fields based on the command issued.

The controller halts in case of receiving a NACK (It means no I3C device on the bus) for the address header of the broadcast CCC transfer. The controller updates the 'ERR_STS' field with appropriate error information in the response status, halts the controller and gives back the control to the application to resume the operation of the controller through writing '1' to the RESUME bit of the I3C_CORE_DEVICE_CTRL register after taking suitable action.

28.5.6.1.2 Directed CCC write and read transfers

The directed CCC write/read transfers are initiated on the bus based on the I3C_CORE_COMMAND_QUEUE_PORT register settings shown in Table 305.

Table 305 Directed CCC write/read transfer programming values

Command attribute	Bit-field	Programmed value	Description
Transfer command	CP	1	Indicates to the controller to consider the CMD field
	CMD[14]	1	Indicates the controller that the transfer is directed CCC transfer
	CMD[13:7]	0x00 – 0x7F	Indicates the CCC command to be transferred
	DEV_INDX	DEV_INDX	Indicates the index of the device table which consists of the target address to be targeted
	SPEED	0 (SDR0) or 7 (I2C FM)	Indicates the controller that the transfer should go in SDR0 speed/mode or I2C FM speed. ***Note**: All CCC transfers are initiated with SDR0 Speed in I3C bus topology. CCC transfers can be either initiated with SDR0 or I2C FM speed based on the context of the bus in JEDEC bus topology.*
	DBP	0 or 1	Indicates whether defining byte is present for directed CCC transfer or not. • 0: Defining byte is not present • 1: Defining byte is present
	SDAP	0 or 1	• 0: Indicates to consider the transmit data from the transmit FIFO if Rnw is set to 0 • 1: Indicates the controller to consider the transmit data from the command if RnW is set to 0

(table continues...)

Table 305 (continued) Directed CCC write/read transfer programming values

Command attribute	Bit-field	Programmed value	Description
	RnW	0	Indicates the transfer is either write or read transfer. • 0 - Indicates that the transfer is a write transfer • 1 - Indicates the transfer is a read transfer
	TGT_RST	0 or 1	Indicates whether the controller should generate target reset pattern at the end of directed RSTACT CCC transfer. • 0: Target reset pattern is not generated • 1: Target reset pattern is generated
	PEC	0 or 1	Indicates whether packet error check is enabled for directed CCC transfer. • 0: PEC check is disabled • 1: PEC check is enabled
Transfer argument (SDAP=0)	DATA_LENGTH	0 - 65535	Indicates the transfer length of the transfer
Short data argument (SDAP=1)	BYTE_STRB	0, 1, 3, 7	Indicates the respective data bytes of immediate command are valid

The data in Tx-FIFO (transfer argument) or data bytes (immediate data transfer) are not required for some directed write CCC's which does not have payload data Indicated through data length (transfer argument) or BYTE_STRB (short data argument). If the directed CCC does not consist of payload, you must indicate it with zero in either data length or BYTE_STRB fields based on the command issued.

28.5.6.1.3 Directed CCC transfer targeted to multiple targets

Each transfer command Initiates directed CCC transfer to only one target since it consists of only one DEV_INDX field. If the requirement is to transfer the directed CCC command to multiple devices, then you must pipeline the multiple transfer commands in the command queue with TOC bits set to zero and with the different DEV_INDX fields pointing to multiple targets. The controller decodes the pipelined "transfer command" during the transfer of directed CCC transfer and decides the next transfer based on the following:

- If the current command and the pipelined command have same directed CCC, then the controller targets the next target without ending the CCC command
- If the current command and the pipelined command are not the same directed CCC, then the controller ends the CCC command and starts issuing the next transfer as indicated by the pipelined transfer command

The application can set the ROC bit to '0' for the subsequent directed CCC commands and enable the ROC bit in the last CCC command if the directed CCC transfer is targeted to multiple devices to avoid the unnecessary responses.

The controller halts in case of the following conditions:

- Receiving NACK for the address header of the directed CCC transfer (It means no I3C device on the bus)
- Receiving NACK for the target address of the directed CCC transfer

The controller updates the 'ERR_STS' field with appropriate error information in the response status and halts the controller and gives back the control to the application to resume the operation of the controller through writing '1' to the RESUME bit of the I3C_CORE_DEVICE_CTRL register.

28.5.6.1.4 I3C private write or read transfers

The I3C private write and read transfers are initiated on the bus based on the I3C_CORE_COMMAND_QUEUE_PORT register settings shown in Table 306.

Table 306 I3C private write/read transfer programming values

Command attribute	Bit-field	Programmed value	Description
Transfer command	CP	0	Indicates the controller to not consider the CMD field
	CMD[14]	NA	This field is not applicable since CP bit is set to 0
	CMD[13:7]	NA	This field is not applicable since CP bit is set to 0
	DEV_INDX	DEV_INDX	Indicates the index of the device table which consists of the target address to be targeted
	SPEED	0 to 4	Indicates the controller that the transfer should go in SDR mode
	SDAP	0 or 1	• 0: Indicates to consider the transmit data from the transmit FIFO if Rnw is set to 0 • 1: Indicates the controller to consider the transmit data from the command if RnW is set to 0
	RnW	0	Indicates the transfer is either write or read transfer. • 0 - Indicates that the transfer is a write transfer • 1 - Indicates the transfer is a read transfer
	PEC	0 or 1	Indicates whether packet error check is enabled for private SDR transfers. • 0: PEC check is disabled • 1: PEC check is enabled
Transfer argument (SDAP=0)	DATA_LENGTH	0 - 65535	Indicates the transfer length of the transfer
Short Data Argument (SDAP=1)	BYTE_STRB	0, 1, 3, 7	Indicates the respective data bytes of Immediate command are valid

***Note**:*
- *The LEGACY_I2C_DEVICE bit in the device address table pointed by the 'DEV_INDX' field of the transfer command should set to zero for the I3C private transfers*
- *To avoid the initial latencies of the transfer, the controller uses 'TX/RX_START_THLD' before initiating the transfer. 'TX_START_THLD' ensures the threshold level of data is present in the transmit buffer for write transfer and 'RX_START_THLD' level of space is available in the receive buffer for the read transfer before initiating the transfer. This threshold is only applicable for the transfers which are initiated with the START condition and not applicable for the transfers which are initiating with the RESTART condition for SDR transfers*
- *To allow the priority for IBI from the devices, the controller can include address header for the I3C private transfers by enabling the 'IBA_INCLUDE' bit in the I3C_CORE_DEVICE_CTRL register since IBI always wins when arbitrated with address header*

The controller halts in case of the following conditions:

- Receiving NACK for the address header of the private transfers if 'IBA_INCLUDE' bit is enabled in the I3C_CORE_DEVICE_CTRL register
- Receiving NACK for the target address of the private transfers

The controller updates the 'ERR_STS' field with appropriate error information in the response status and halts the controller and gives back the control to the application to resume the operation of the controller through writing '1' to the resume bit of the I3C_CORE_DEVICE_CTRL register.

28.5.6.1.5 I2C private write or read transfers

The I2C private write and read transfers are initiated on the bus based on the I3C_CORE_COMMAND_QUEUE_PORT register settings as shown in Table 307.

Table 307 I2C private write/read transfer programming values

Command attribute	Bit-field	Programmed value	Description
Transfer command	CP	0	Indicates the controller to not consider the CMD field
	CMD[14]	NA	This field is not applicable since CP bit is set to 0
	CMD[13:7]	NA	This field is not applicable since CP bit is set to 0
	DEV_INDX	DEV_INDX	Indicates the index of the device table which consists of the target address to be targeted
	SPEED	0 or 1	Indicates the controller that the transfer should go in I2C SDR mode. Values(I2C mode): • 0x0: I2C Fm • 0x1: I2C Fm+
	SDAP	0 or 1	• 0: Indicates to consider the transmit data from the transmit FIFO if RnW is set to 0 • 1: Indicates the controller to consider the transmit data from the command if RnW is set to 0
	RnW	0	Indicates the transfer is either write or read transfer. • 0 - Indicates that the transfer is a write transfer • 1 - Indicates the transfer is a read transfer
	PEC	0 or 1	Indicates whether packet error check is enabled for private SDR transfers. • 0: PEC check is disabled • 1: PEC check is enabled
Transfer argument (SDAP=0)	DATA_LENGTH	0 - 65535	Indicates the transfer length of the transfer
Short data argument (SDAP=1)	BYTE_STRB	0, 1, 3, 7	Indicates the respective data bytes of immediate command are valid

***Note**:*
- *The 'LEGACY_I2C_DEVICE' bit in the device address table pointed by the 'DEV_INDX' field of the transfer command must be set to '1' for the I2C private transfers*
- *The controller uses the I2C protocol to initiate the I2C transfers for the legacy I2C devices*
- *To allow the priority for the IBI from the I3C devices, the controller can include address header for the I2C private transfers through enabling the 'IBA_INCLUDE' bit in the I3C_CORE_DEVICE_CTRL register*

The controller halts in case of the following conditions:
- Receiving NACK for the address header of the private transfers if 'IBA_INCLUDE' bit is enabled in the I3C_CORE_DEVICE_CTRL register
- Receiving NACK for the target address of the private transfers

The controller updates the 'ERR_STS' field with appropriate error information in the response status and halts the controller and gives back the control to the application to resume the operation of the controller through writing '1' to the RESUME bit of the I3C_CORE_DEVICE_CTRL register.

28.5.6.1.6 Tx-FIFO empty and Rx-FIFO full conditions

At SDR speeds, the controller extends the clock by pulling the SCL low under the following conditions:
- Tx-FIFO empty during the middle of write data transfer at SDR speeds
- Rx-FIFO full during the middle of read data transfer at SDR speeds

28.5.6.1.7 TOC and ROC bit settings for SDR transfers

The termination on completion (TOC) of transfer command is used to decide whether to generate a STOP or continue with the next transfer.
- If TOC bit is set to '1', then controller generates STOP condition after the end of the transfer
- If TOC bit is set to '0', then controller continues the next pipelined transfer based on the current transfer type
  - If the current transfer is either broadcast CCC write or private transfer, then controller generates RESTART after the end of the transfer
  - If the current transfer is the directed CCC transfer, then controller generates RESTART followed by a reserved address (0x7E) after the end of the transfer
- If TOC bit is set to '0' and the next pipelined command is not available, then controller extends the clock by pulling SCL low until the next command is available

The response on completion (ROC) bit of the transfer command is used to generate the response status of the transfer after the end of the transfer. The response status is automatically generated if any error condition is occurred for the transfer irrespective of the ROC bit.

28.5.6.2 High data rate (HDR) transfers

The HDR transfer protocol is different from the SDR protocol and is used to transfer more data at the same SDR frequency. The HDR modes have a bus-wide effect and the controller is required to enter HDR mode through CCC (common command code) before initiating transfer to the target. The targets decode the incoming CCC and enter the respective HDR mode for the reception of the transfer.

***Note**:* *Only High data rate-double data rate (HDR-DDR) mode is supported. In this mode, one bit is sent in each SCL clock transition.*

28.5.6.2.1 High data rate - double data rate (HDR-DDR) transfers

The HDR-DDR write and read transfers are initiated on the bus based on the I3C_CORE_COMMAND_QUEUE_PORT register and settings shown in Table 308.

Table 308 HDR-DDR transfer programming values

Command attribute	Bit-field	Programmed value	Description
Transfer command	CP	1	Indicates to the controller to consider the CMD field
	CMD[14:7]	0x00 – 0xFF	This field indicates either write or read command, is used in the command code of HDR transfer. • 0x00 – 0x1F: I3C reserved write commands • 0x20 – 0x7F: I3C vendor write commands • 0x80 – 0x9F: I3C reserved read Commands • 0xA0 – 0xFF: I3C vendor read Commands
	DEV_INDX	DEV_INDX	Indicates the index of the device table, which consists of the target address to be targeted
	SPEED	6 (HDR-DDR)	Indicates to the controller that the transfer should go in HDR-DDR mode
	SDAP	0	• 0: Indicates to consider the transmit data from the Transmit FIFO if RnW is set to 0
	RnW	0 or 1	• 0 - Indicates that the transfer is a write transfer • 1 - Indicates the transfer is a read transfer ***Note**: This bit is used for the write or read command, used in HDR-CMD (CMD[14]) field.*
Transfer argument (SDAP=0)	DATA_LENGTH	0 - 65535	Indicates the transfer length

***Note**: The HDR transfers are supported only with 'transfer argument command' and not with 'short data argument command', since a maximum of two words can be transferred in short data argument command format and it does not justify the overheads in terms of ENTHDR* CCC, HDR command, and entry and exit patterns.*

The I3C block generates the 'ENTHDR0' CCC to enter the HDR-DDR mode and forms the command code through the CMD field of transfer command and target address. The controller generates the preamble and parity bits according to the protocol, and then sends them along with the write data word. The I3C considers word from the Tx-FIFO and appends the parity bits and preamble bits to form the write data word and transmits the data. The HDR-RESTART and HDR-EXIT pattern is generated in place of the RESTART and STOP conditions.

To avoid the initial latencies of the transfer, the controller uses 'TX/RX_START_THLD' before initiating the transfer. 'TX_START_THLD' indicates whether the threshold level of data is present in the transmit buffer for write transfer or 'RX_START_THLD' level of space is available in the receive buffer for the read transfer before initiating the transfer. This threshold is only applicable for the transfers which are initiated with the START or RESTART condition. If the threshold amount of write data in the Tx-FIFO or empty space in Rx-FIFO is not

available, the controller waits until the threshold amount of data is available to initiate the transfer. If the threshold amount of data is not available for the HDR-DDR transfer, which must be initiated with RESTART, then the controller generates the EXIT pattern followed by STOP for the current transfer and waits for the threshold amount of data and initiates the next transfer with a START condition.

The controller halts in the following conditions:

- Receiving NACK for the address header of the private transfers
- Receiving NACK for the target address (HDR-CMD) of the HDR-DDR private transfers
- During the HDR-DDR transfers, If the controller experiences the transfer underflow or receive overflow
- After executing software initiated abort
- The controller decodes the parity bits, CRC byte, frame mismatch, and validates them for the read transfer and reports in the 'ERR_STS' field of the response status if there is any error

The controller updates the 'ERR_STS' field with appropriate error information in the response status and halts the controller and gives back the control to the application, to resume the operation of the controller through writing '1' to the Resume bit of the I3C_CORE_DEVICE_CTRL register.

28.5.6.2.2 Tx-FIFO empty and Rx-FIFO full conditions

The controller terminates the transfer abruptly during the following conditions and reports it in the 'ERR_STS' field of the response status since the clock extension is not possible in HDR modes:

- Tx-FIFO empty during the middle of write data transfer
- In HDR-DDR Mode, Rx-FIFO full during the middle of read data transfer through 'master abort' feature

28.5.6.2.3 TOC and ROC bit settings for HDR transfers

The termination on completion (TOC) of transfer command is used to decide whether to generate HDR-EXIT or HDR-RESTART to continue with the next transfer.

- If TOC bit is set to '1', then controller generates HDR EXIT after the end of the transfer
- If TOC bit is set to '0', then controller continues the next pipelined transfer based on the current transfer type

Table 309 Implication of TOC and ROC bit settings for HDR mode

Current transfer	Next transfer	Start threshold of next transfer	Controller action
HDR-DDR	HDR-DDR	Met	**1.** Generates HDR-RESTART at the end of transfer **2.** Continue with the HDR command of next transfer
HDR-DDR	HDR-DDR	Not met	**1.** Generates HDR-EXIT at the end of transfer **2.** Wait for the start threshold for the pipelined transfer to be met **3.** Generate the pipelined transfer with START after the start threshold is met
HDR-DDR	SDR	NA	**1.** Generates HDR-EXIT at the end of transfer **2.** Wait for Bus-free time **3.** Initiate the next SDR transfer with START condition

The response on completion (ROC) bit of the transfer command is used to generate the response status of the transfer after the end of the transfer. The response status is automatically generated if any error condition is occurred for the transfer irrespective of the ROC bit.

28.6 Interrupts

The I3C block generates an interrupt to indicate many bus conditions, transfer status, errors, command responses, IBI, FIFO threshold conditions, and so on.

The interrupt is connected to CPUSS, to interrupt CPUSS with events recorded in the interrupt registers. This interrupt is synchronous to PERI_CLK.

The following four interrupt registers are available:

- INTR_STATUS: Interrupt status register
- INTR_STATUS_EN: Interrupt status enable register. The interrupt status is updated in the INTR_STATUS register, only if the corresponding status enable bit is set
- INTR_SIGNAL_EN: Interrupt signal enable register. The interrupt pin is triggered based on INTR_STATUS, only if the corresponding signal enable bit is set
- INTR_FORCE: Interrupt force enable register is used for test purposes. Individual interrupts can be forcefully triggered if the corresponding force enable bit is set, provided the corresponding bit in the INTR_STATUS_EN register is set

28.7 I3C block initialization

This initialization section is common for controller and target roles.

The first step to program the I3C block for any operation mode is to write a ‘1’ to the pre-enable I3C_CTRL[ENABLED] register. While this register is 0, the I3C Core effectively remains in a reset state and issues a bus error for any incoming AHB transactions, preventing further programmability.

***Note**: After programming this register, the I3C block can be further programmed but does not participate in any I3C bus activity.*

Typically, after initiating the hardware/software reset, the application would select the controller or target role by setting the I3C_CORE_DEVICE_CTRL_EXTENDED[DEV_OPERATION_MODE] register. For example, the application selects the primary controller by setting the I3C_CORE_DEVICE_CTRL_EXTENDED[DEV_OPERATION_MODE] register to 0x0.

Set the I3C_CORE_DEVICE_ADDR[DYNAMIC_ADDR] and I3C_CORE_DEVICE_ADDR[DYNAMIC_ADDR_VALID] as the primary controller, self-assigns its address. This setting must be done only when the block is in pre-enabled state (I3C_CTRL[ENABLED] = 1 but I3C_CORE_DEVICE_CTRL[ENABLE] = 0).

***Note**: When dynamic mode switching is SETDASA and application configured the I3C block as target, during SETDASA CCC, if the static address RnW bit is set to '1' instead of `0`, target incorrectly informs the controller of the address assignment acceptance which results in error condition therefore the address is not assigned. In this case application can switch to ENTDAA.*

Optionally, the application chooses the block to operate as a controller or a target without participating in the dynamic mode switching. This can be done by setting the register I3C_CORE_DEVICE_CTRL_EXTENDED[GETACCCR_ACK_CTRL] to 1’b1 to reject (NACK) the incoming GETACCCR CCC in the target mode and setting the register I3C_CORE_IBI_CR_REQ_REJECT[CR_REQ_REJECT) to 32’hFFFF_FFFF to reject (NACK) and disable the incoming controller request (CR) IBIs in the controller mode.

Prior to enabling the I3C block, set a few additional common registers. For both controller and target roles, set the following registers to program the threshold control registers:

- Set the I3C_CORE_QUEUE_THLD_CTRL[CMD_EMPTY_BUF_THLD] register to the required value to control the level of empty spaces in the command queue that are used by the I3C block, to trigger the I3C_CORE_INTR_STATUS[CMD_QUEUE_READY_STAT] interrupt
- Set the I3C_CORE_QUEUE_THLD_CTRL[RESP_BUF_THLD] register to the required value to control the level of status/response entries in the response queue that are used by the I3C block, to trigger the I3C_CORE_INTR_STATUS[RESP_READY_STAT] interrupt
- Set the I3C_CORE_QUEUE_THLD_CTRL[IBI_DATA_THLD] register to the required value which is used to chunk the incoming IBI and report as different statuses in the IBI queue, to make sure that the application can read the incoming IBI data, in parallel with receiving the IBI data from the target
- Set the I3C_CORE_QUEUE_THLD_CTRL[IBI_STATUS_THLD] register to the required value to control the number of IBI status entries (or greater) in the IBI queue that are used by the I3C block, to trigger the I3C_CORE_INTR_STATUS[IBI_THLD_STAT] interrupt
- Set the I3C_CORE_DATA_BUFFER_THLD_CTRL [TX_START_THLD] register to the required value to control the start of the TX transfer on the I3C bus
- Set the I3C_CORE_DATA_BUFFER_THLD_CTRL[RX_START_THLD] register to the required value to control the start of RX transfer on the I3C bus in controller mode, and control the ACK response for a read command on the I3C bus in target mode
- Set the DATA_BUFFER_THLD_CTRL[TX_EMPTY_BUF_THLD] register to the required value to control the level of empty spaces in the TX FIFO that are used by the I3C block either to trigger the I3C_CORE_INTR_STATUS[TX_THLD_STAT] interrupt

28.7.1 Initializing for I3C primary controller operation

This section describes the programming flow for initializing the I3C block registers in addition to the common registers described in I3C block initialization.

In the primary controller role, the block must attribute its own address. For this, set the following register fields in the I3C_CORE_DEVICE_ADDR register:

- Set the DYNAMIC_ADDR field to the dynamic address of the controller. This address must be other than the reserved addresses mentioned in Table 303 of the MIPI I3C specification
- Set the DYNAMIC_ADDR_VALID field to "1" to indicate that the DYNAMIC_ADDR field is valid

The following register fields must be set to program timing registers and all these counts are in terms of the CLK_IF period used:

- I3C_CORE_SCL_I3C_OD_TIMING: Program the I3C_OD_LCNT and I3C_OD_HCNT fields with the required SCL low and high period values for the I3C Open-drain phase
- I3C_CORE_SCL_I3C_PP_TIMING: Program the I3C_PP_LCNT and I3C_PP_HCNT fields with the required SCL low and high period values for the I3C Push-pull phase
- I3C_CORE_BUS_FREE_AVAIL_TIMING: Program the BUS_FREE_TIME field with the required value of bus free period
- I3C_CORE_SCL_I2C_FMP_TIMING: Program the I2C_FMP_LCNT and I2C_FMP_HCNT fields with the required SCL low and high period values for the I2C FM+ transfer
- I3C_CORE_SCL_I2C_FM_TIMING: Program the I2C_FM_LCNT and I2C_FM_HCNT fields with the required SCL low and high period values for the I2C FM transfer
- I3C_CORE_BUS_IDLE_TIMING: Program the BUS_IDLE field with the required value of bus idle period
- I3C_CORE_SCL_EXT_LCNT_TIMING: Program the I3C_EXT_LCNT_1, I3C_EXT_LCNT_2, I3C_EXT_LCNT_3, and I3C_EXT_LCNT_4 fields with the required SCL push-pull low period values for I3C data transfers in SDR1, SDR2, SDR3 and SDR4 speeds respectively

- I3C_CORE_SCL_EXT_TERMN_LCNT_TIMING[I3C_EXT_TERMN_LCNT]: Program the I3C_EXT_TERMN_LCNT field with the required extended SCL push-pull low period value for I3C read termination bit
- I3C_CORE_SDA_HOLD_SWITCH_DLY_TIMING: Program the SDA_TX_HOLD field with the required SDA (Data) Hold delay

28.7.2 Initializing for I3C target operation

This section describes the programming flow for initializing the target registers in addition to the common registers described in I3C block initialization.

- I3C_CORE_BUS_FREE_AVAIL_TIMING: Program BUS_AVAILABLE_TIME field with the required value of bus available period. The count is in terms of the CLK_IF period
- I3C_CORE_BUS_IDLE_TIMING: Program BUS_IDLE_TIME field with the required value of bus idle period. The count is in terms of the CLK_IF period
- I3C_CORE_DEVICE_CTRL: Program IDLE_CNT_MULTIPLIER field with the required value for enabling the target block after the power-on reset
- I3C_CORE_DEVICE_CTRL: Program ADAPTIVE_I2C_I3C field with the required value to select the adaptive mode
- I3C_CORE_DEVICE_ADDR: Program STATIC_ADDR field with the new static address of the target
- I3C_CORE_DEVICE_ADDR: Set the STATIC_ADDR_VALID field to 1 to validate the static address if changed
- I3C_CORE_MAX_DATA_SPEED: Program the MXDS_MAX_RD_SPEED field with the maximum sustainable read speed supported
- I3C_CORE_MAX_DATA_SPEED: Program the MXDS_CLK_DATA_TURN field with the supported clock to data turn around time

Set the respective bits of I3C_CORE_INTR_STATUS_EN and I3C_CORE_INTR_SIGNAL_EN registers to enable or disable the respective interrupts. The following interrupt statuses are available for target mode or secondary controller mode:

- CCC_UPDATED_STS
- DYN_ADDR_ASSGN_STS
- DEFTGTS_STS
- READ_REQ_RECV_STS
- IBI_UPDATED_STS
- BUSOWNER_UPDATED_STS
- TGT_RST_PATTERN_DET_STS

For more information on the description of the interrupts, see I3C_CORE_INTR_STATUS register section.

***Note**: After the hardware reset, the target block needs a few SCL clock edges to come out of the reset state. The application needs to send a "dummy transfer" to get the block out of reset before attempting an address assignment or other operation.*

28.7.3 Enabling the I3C block

This initialization section is common for the controller and target roles. The final initialization step is to set I3C_CORE_DEVICE_CTRL[ENABLE] to ‘1’ to enable the I3C block.

Note:
- *Programming of the ENABLE bit must be the last step before initiating any bus transfers, including handling the IBIs*
- *If the I3C_CORE_DEVICE_CTRL[ENABLE] register is set to 0, the block does not:*
  - *Respond with the ACK to all the incoming transfers in the target mode*
  - *Initiate any transfers, including the generation of SCL, on detecting a START condition of an IBI in the controller mode*
  - *After the hardware reset, the target block needs a few SCL clock cycles to come out of the reset state*

28.8 Using the initialized I3C in primary controller role

Programming flows for I3C when the device role is configured as the I3C controller or when the device mode is programmed as the I3C controller.

28.8.1 Target address assignment

- Set dynamic address from static address (SETDASA) transfer in controller mode
- Enter dynamic address assignment (ENTDAA) transfer in controller mode

SETDASA transfer in controller mode

The main controller assigns dynamic address to any I3C device with known static address using the directed command code – set dynamic address from static address (SETDASA). This is faster than using the ENTDAA procedure. The SETDASA process is initiated by the primary controller after cold power-up of the system.

The programming flow for SETDASA transfer in controller mode consists of two main phases:

1. Issue the SETDASA CCC command: The issue SETDASA CCC command phase generates the command with SETDASA address assignment command. The SETDASA address assignment command is generated by setting the CMD_ATTR field as ‘address assignment command’ and written through COMMAND_QUEUE_PORT Register. The following are the required field settings in the address assignment command:
   - **a.** Set the CMD field to SETDASA CCC code
   - **b.** Set the DEV_INDX field to point the device address table from where the block assigns the dynamic address to the targets
   - **c.** Set the DEV_COUNT field to the number of devices to be addressed

2. Check response status: The check response status phase checks the generated response by reading the RESPONSE_QUEUE_PORT register. The INTR_STATUS[RESP_READY_STAT] interrupt indicates the response available status and you can read the available response from the RESPONSE_QUEUE_PORT register

ENTDAA transfer in controller mode

The active controller assigns dynamic address to any I3C device not assigned earlier with dynamic address, using the broadcast command code – Enter dynamic address assignment (ENTDAA). This process is initiated by the active controller after cold power-up, after completing the dynamic address assignment for static address devices, using SETDASA procedure or if a hot-join request is received from any I3C device. The active controller may choose to use a single or multiple ENTDAA address assignment command to assign a dynamic address for all the connected I3C devices. The number of devices that can be assigned to a single command is limited by the configured value of the device address table and the device characteristics table.

The programming flow for ENTDAA transfer in controller mode consists of three main phases:

1. The application should program the device address table (DEV_ADDR_TABLE_LOCx) with the data structure as follows:
   - 0 -16 : Reserved

- 16 - 22 : Dynamic address (DYN_ADDR)
- 23 : Dynamic address parity (~XOR(DYN_ADDR))
- 24 - 30 : Reserved
- 31 : 0

***Note**: Do not use the reserved fields during address assignment procedure as this may result in unexpected behavior.*

The number of locations to be programmed is decided by the following:

- Number of I3C devices to be assigned with dynamic address with single address assignment command
- Number of locations available in the device address table

2. Issue ENTDAA command: The issue ENTDAA command phase generates the ENTDAA address assignment command. The ENTDAA address assignment command is generated by setting the CMD_ATTR field in the command as 3 and written through the COMMAND_QUEUE_PORT register. The following are the required settings in the COMMAND_QUEUE_PORT register:

a. Set the CMD field to ENTDAA CCC code
b. Set the DEV_INDX field to point the device address table through which the block assigns the addresses to the targets
c. Set the DEV_COUNT field to the maximum number of devices to be addressed with the single command

3. Check response status: The check response status phase detects the generation of response status and reads it to know the status of the issued transfer. The INTR_STATUS[RESP_READY_STAT] interrupt indicates the response available status and you can read the available response through RESPONSE_QUEUE_PORT register

28.8.2 Receiving IBI (all types)

The I3C target devices can initiate communication to the active controller through In-band interrupt (IBI).

The following types of IBIs are possible on an I3C bus:

- Hot-join request (HJ) from a Hot-join capable target
- Target interrupt request (TIR) from a target
- Controller role request (CR) from a controller capable target (secondary controller)

Enable the block to allow the host block to detect (SDA low) and receive the IBI ID. This enables the host block to start providing the SCL clocks (period = I3C_OD_LCNT+I3C_OD_HCNT) to receive the IBI ID from the requesting target device. The host block detects the IBI in the following scenarios:

- Upon detecting low on the SDA input port after a power-on reset (POR)
- Upon detecting an arbitration loss during an address phase of any controller-initiated transfer, following a START condition (not RESTART)
- Upon detecting the SDA input port going (not initiated by the block) low following a STOP condition (target initiated IBI)

28.8.2.1 I3C Hot-join (HJ) request

If the received IBI address matches the Hot-join address (7'b0000010, RnW=0), then the response for the received HJ request is based on the programmed HJ response control indicated in I3C_CORE_DEVICE_CTRL[HJ_REJECT] register.

If the HJ response control is set to 0, then the block responds to the HJ request with ACK and sets the 'IBI_STS' field to ACK to indicate to the application that the block has ACK'd the received HJ request.

If the HJ response control is set to 1, then the block responds to the HJ request with a NACK, followed by issuing a broadcast DISEC CCC command (DISHJ bit set) with the RESTART condition. This disables the HJ request generation from all the unaddressed devices at that instant. The block sets the 'IBI_STS' field to NACK to indicate to the application that the block has rejected the received HJ request. The application can optionally set the HJ reject notify control to get an IBI status for a rejected HJ request. Otherwise, the block moderates the IBI status generation for rejected HJ requests.

Based on the IBI status for the HJ request, the application must issue the ENTDAA to assign the dynamic address for the devices that generate the HJ request.

28.8.2.2 I3C controller role (CR) request

If the received IBI ID matches the dynamic address of any one of the valid entries in the DAT and the received RnW bit is 0, then the response for the received CR is based on the programmed CR response control indicated in I3C_CORE_DEVICE_ADDRESS_TABLEx[CR_REJECT] register.

The secondary controller role allows the application to either set the CR response control to 0 or 1 individually to accept or reject the CR request respectively. The application is expected to set this to 1 for all target entries in the DAT when only the primary controller use case is expected. This requirement allows the controller to reject any CR request, if received unexpectedly from a malfunctioning target device.

28.8.2.2.1 CR reject control

If the CR reject control is set to 1, then the block responds to the CR request with NACK followed by issuing a directed DISEC CCC command (DISMR bit set) with the RESTART condition to the CR initiated target.

The block sets the 'IBI_STS' field as NACK to indicate to the application that the block has rejected the received CR request. The application can optionally set the CR reject notify control indicated in I3C_CORE_IBI_QUEUE_CTRL[NOTIFY_MR_REJECTED] register to get an IBI status for a rejected CR request. Otherwise, the block moderates the IBI status generation for rejected CR requests.

If the CR response control is set to 0, then the block responds to the CR request with ACK and sets the 'IBI_STS' field to ACK, which indicates the application that the block has ACK'd received the CR request. Based on the IBI status for the CR request, the application must follow the controller ownership handover procedure.

28.8.2.3 I3C target interrupt request (TIR)

If the received IBI ID matches with the dynamic address of any one of the valid entries in the DAT and the received RnW bit is 1, then the response for the received TIR is based on the programmed TIR response control indicated in I3C_CORE_DEVICE_ADDR_TABLEx[[TIR_REJECT] register.

28.8.2.3.1 TIR reject control

If TIR reject control is set to 1, then the block NACKs the TIR and then issues a directed DISEC CCC command (DISINT bit set) with the RESTART condition targeting the matching dynamic address.

This disables the TIR generation in the requested target device. The IBI status for the corresponding TIR indicates the application that the incoming IBI is NACK'd through 'IBI_STS' descriptor.

If the TIR response control is set to 0, then the block ACKs the TIR. If the IBI payload control is set to 0, then the block stops generating the SCL clock after acknowledging the TIR.

The block then notifies the application to take necessary action for the accepted TIR. The application must not set the IBI payload control bit to 1, when the target device does not support the mandatory data byte (BCR[2]=0).

28.8.2.4 IBI queue data structure

The IBI queue comprises of the interleaved data (If IBI payload control is supported) received from the target device, and the status generated from the controller for the respective data slice of TIR or HJ/MR. For the configurations that support IBI without payload, only IBI status entry is written in to the IBI queue.

The data portion of the structure comprises of the data received from the target device along with the IBI (MDB + other payload bytes if any) issued for the requesting device. If the data payload size of the TIR goes above the programmed TIR data chunk size in QUEUE_THLD_CTRL[IBI_DATA_THLD], then the controller slices the incoming data bytes into multiple chunks, and generates an IBI status for each chunk. The last data chunk is indicated as LAST_STS=1 in the corresponding IBI status. The below figure shows an example of the IBI queue data structure holding the IBI data (TIR) and the status for an IBI with non-multiple chunk size data.

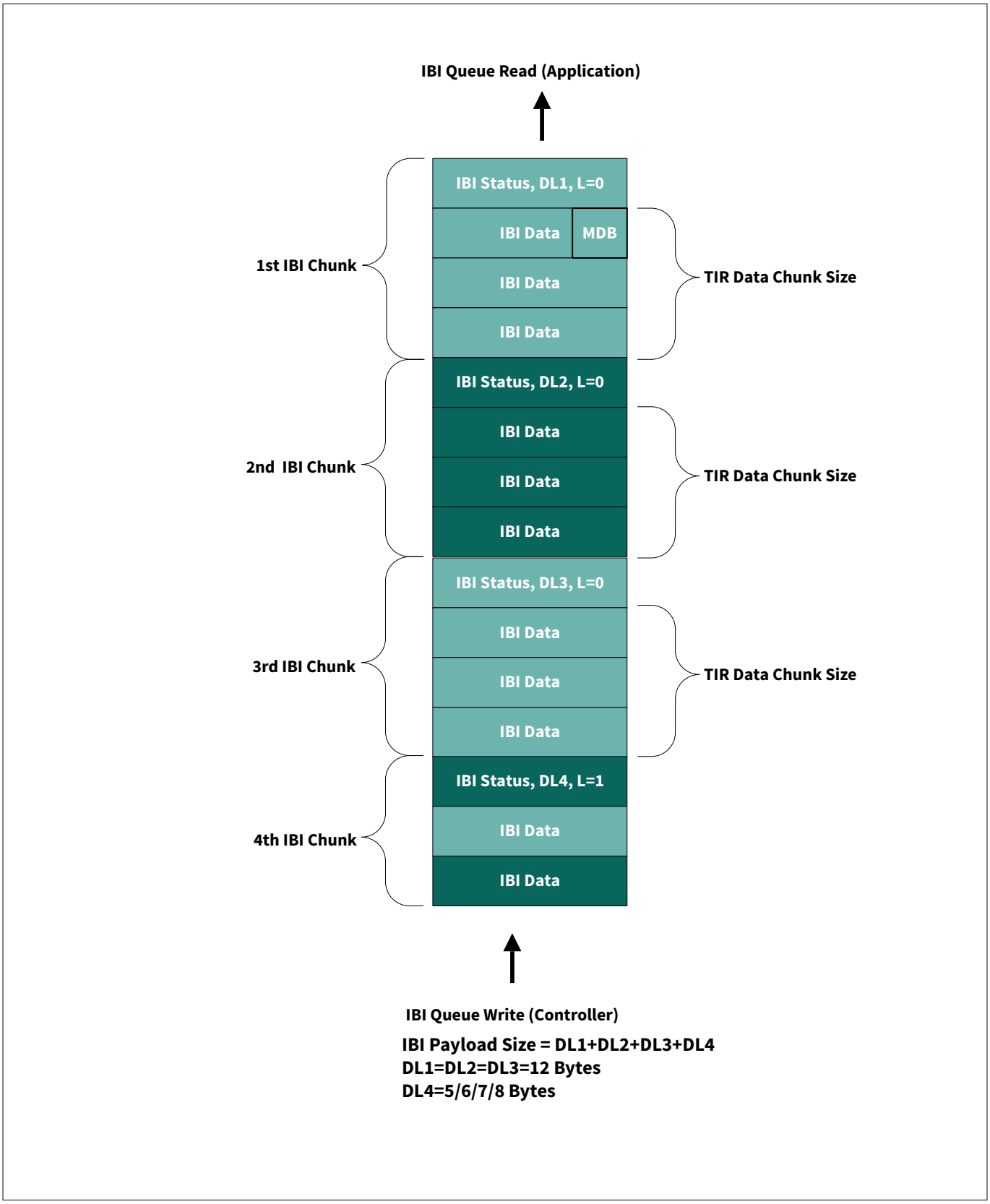


Figure 345 IBI with payload structure (with the last data chunk less than the programmed data chunk size)

The application reads the IBI queue upon detecting the interrupt status I3C_CORE_ INTR_STATUS[IBI_THLD_STS] (Controller only and secondary controller configuration). The interrupt can be

moderated by setting the I3C_CORE_QUEUE_THLD_CTRL[IBI_STATUS_THLD] field. The first location read always provides the IBI status for the "1st IBI chunk", which includes the data length of that chuck in bytes. The application can read the data portion of the chunk without waiting for any further interrupt. The number of data read must be limited to the data length field of the corresponding status.

The below figure shows an example of the IBI queue data structure holding the IBI data and the status for an IBI with exact multiple of programmed TIR data chunk size.

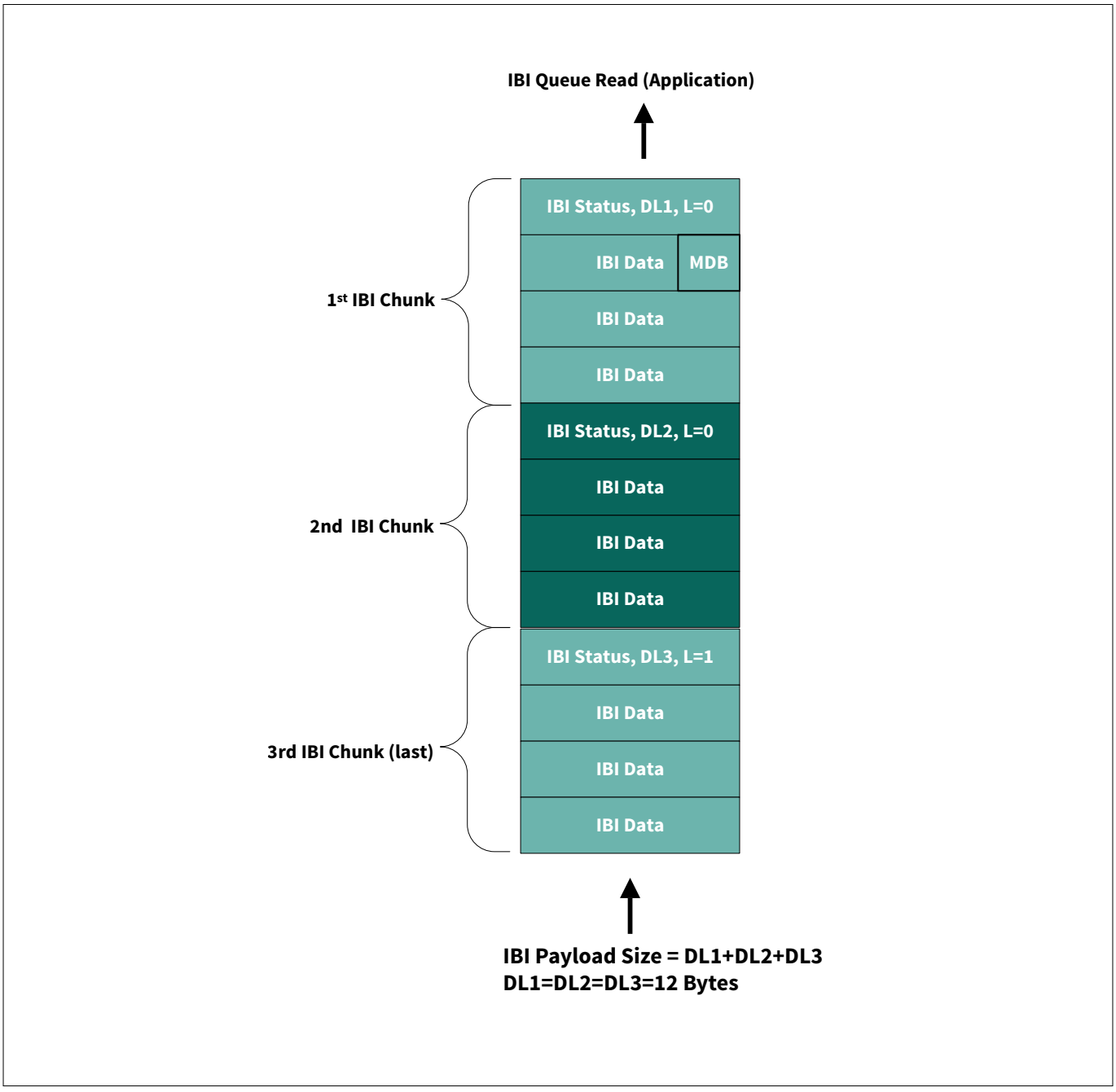


Figure 346 IBI with payload structure (with the last chunk equals to the programmed chunk size)

The below figure shows an example of the IBI queue data structure holding only the IBI status. This structure is applicable for all the IBIs that do not support the payload, which includes the HJ, the TIR without payload, and the CR.

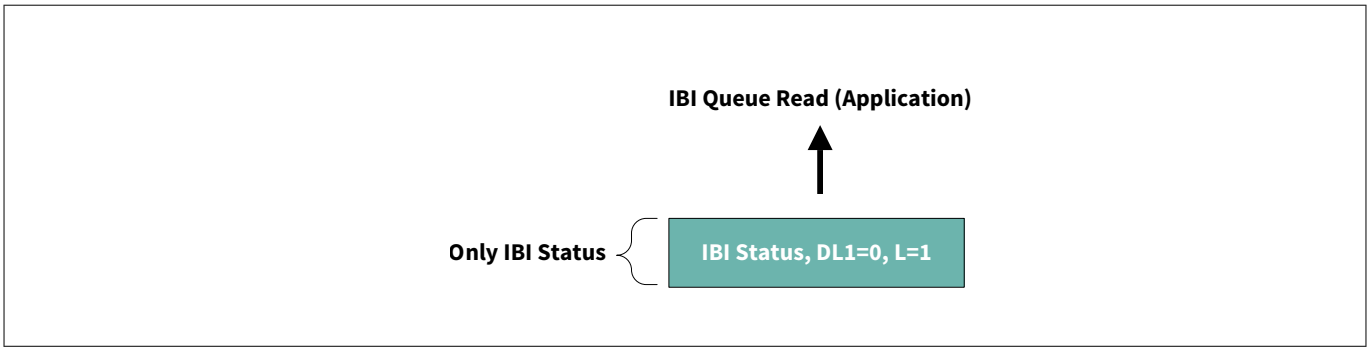


Figure 347 IBI without payload structure

28.8.2.5 IBI notification control

Normally, the block generates an IBI response only for the received IBIs for which the ACK response is sent. Optionally, you can enable the response generation for the NACK'ed IBIs by setting the appropriate reject notify control fields indicated in the I3C_CORE_IBI_QUEUE_CTRL register, configurable as per IBI type (CR, HJ, and TIR).

When the Hot-join reject notify control is set to 1, the block notifies the application by generating an IBI response, when a valid Hot-join request is received, for which a NACK response was sent, followed by a broadcast auto disable CCC command.

When the CR reject notify control is set to 1, the block notifies the application by generating an IBI response, when a valid controller request is received, for which a NACK response is sent followed by a directed Auto disable CCC command to the requesting target device. When the TIR reject notify control is set to 1, the block notifies the application by generating an IBI response when a valid TIR request is received, for which a NACK response is sent followed by a directed auto disable CCC command to the requesting target device.

As a part of the controller initialization procedure, program the following controls after the dynamic address procedure is complete:

- IBI response controls (TIR, CR, and HJ)
  - TIR data chunk size
- IBI reject notify controls (TIR, CR, and HJ)

The INTR_STATUS[IBI_THLD_STS] interrupt indicates the availability of the IBI status, which can be read from the IBI_QUEUE_STATUS register.

28.8.3 CCC read/write transfers

Broadcast CCC transfer in active controller mode

The programming flow for broadcast CCC transfer in controller mode consists of three main phases:

1. Issue broadcast CCC command: The Issue broadcast CCC command phase generates the command with the required broadcast CCC enabled. The following settings are done for the command and then written to the block through I3C_CORE_COMMAND_QUEUE_PORT register.

   The application must write the following combination of the commands based on the required CCC, with or without data:

   - Only transfer command, if the broadcast CCC does not have data
   - Transfer command with either transfer argument or SDAP, if the broadcast CCC consists of data

   The following transfer command settings are required in the I3C_CORE_COMMAND_QUEUE_PORT register:

   - Set CP bit to 1

- Write CMD field with broadcast CCC
- Set RnW bit to 1

2. Push data to transmit FIFO: The push data to transmit FIFO phase continuously fills the data to the transmit FIFO based on the command issued in I3C_CORE_COMMAND_QUEUE_PORT register. The application pushes the data (in Word) to the Tx-FIFO based on the command issued to the I3C_CORE_COMMAND_QUEUE_PORT register. The application can push the data where the CPU can monitor the threshold status signal (I3C_CORE_INTR_STATUS[TX_THLD_STAT]) and then push the data to the Tx-FIFO through I3C_CORE_TX_DATA_PORT
3. Check response status: The check response status phase detects the generation of response status and reads it, to know the status of the issued transfer. The I3C_CORE_INTR_STATUS[RESP_READY_STAT] interrupt indicates the response available status and that you can read the available response from the I3C_CORE_RESPONSE_QUEUE_PORT register

Directed CCC transfer in active controller mode

The programming flow for directed CCC transfer in controller mode consists of four phases:

1. Issue directed CCC command: The Issue directed CCC command phase generates the command with the required directed CCC enabled. The following settings are done for the command and then written to the block through I3C_CORE_COMMAND_QUEUE_PORT register. The application must write the following combination of the commands based on their required CCC, with or without data:
   - Only transfer command, if the directed CCC does not have the data
   - Transfer command with either transfer argument or SDAP, if the directed CCC consists of data

   The following transfer command settings are required in the I3C_CORE_COMMAND_QUEUE_PORT register:
   - Set CP bit to 1
   - Write CMD field with Directed CCC
   - Point the device address table to the respective target address with the DEV_INDX field
   - Set RnW bit to either 1 (read) or 0 (write) based on directed CCC transfer
2. Push data to transmit FIFO: The push data to transmit FIFO phase continuously fills the data to the transmit FIFO based on the command issued in I3C_CORE_COMMAND_QUEUE_PORT register. The application pushes the data (in word) to the Tx-FIFO based on the command issued to the I3C_CORE_COMMAND_QUEUE_PORT register. The application can push the data where the CPU can monitor the threshold status signal (I3C_CORE_INTR_STATUS[TX_THLD_STAT]) and then push the data to the Tx-FIFO through I3C_CORE_TX_DATA_PORT
3. Retrieve data from Rx-FIFO: The Retrieve data from Rx-FIFO phase continuously retrieves the data from the receive FIFO based on the command issued in I3C_CORE_COMMAND_QUEUE_PORT register. The application retrieves the data (in word) from the Rx-FIFO based on the command issued to the I3C_CORE_COMMAND_QUEUE_PORT register. The application can fill the data where the CPU can monitor the threshold status signal (I3C_CORE_INTR_STATUS[RX_THLD_STAT]) and then pop the data from the Rx-FIFO through I3C_CORE_RX_DATA_PORT
4. Check response status: The check response status phase detects the generation of response status and reads it to know the status of the issued transfer. The I3C_CORE_INTR_STATUS[RESP_READY_STAT] interrupt indicates the response available status and the application can read the available response from the I3C_CORE_RESPONSE_QUEUE_PORT register

28.8.4 Private read/write transfers

Private transfers in controller mode

The programming flow for I3C private transfers in controller mode consists of four main phases:

1. Issue private transfers: This phase generates the command with the private write/read transfer to the respective target. The following settings are done for the command and then written to the block

through the I3C_CORE_COMMAND_QUEUE_PORT register. The application must write the following combination of the commands based on their required private transfer, with or without data:

- Transfer command with the short data argument packet, if the data length is less than 3 bytes
- Transfer command with the transfer argument, if the data length is more than 3 bytes
- Read transfer command with transfer argument

The following are the required transfer command settings in the I3C_CORE_COMMAND_QUEUE_PORT register:

- Set CP bit to 0
- Ignore CMD field, as it is invalid for the private transfers
- Point the device address table for the respective Target address with the DEV_INDX field.
  - If the targeted device is I2C, then application has to set the I3C_CORE_DEV_ADDR_TABLE_LOC[31] to 1
  - If the targeted device is I3C, then application has to set the I3C_CORE_DEV_ADDR_TABLE_LOC[31] to 0 of the corresponding device
  - Set RnW bit to either 1 (read) or 0 (write) based on private transfer

2. Push data to transmit FIFO: The push data to transmit FIFO phase continuously fills the data to the transmit FIFO based on the command issued in I3C_CORE_COMMAND_QUEUE_PORT register. The application pushes the data (in Word) to the Tx-FIFO based on the command issued to the I3C_CORE_COMMAND_QUEUE_PORT register. The application can push the data where the CPU can monitor the threshold status signal (I3C_CORE_INTR_STATUS[TX_THLD_STAT]) and then push the data to the Tx-FIFO through I3C_CORE_TX_DATA_PORT

3. Retrieve data from Rx-FIFO: The retrieve data from Rx-FIFO phase continuously retrieves the data from the receive FIFO, based on the command issued in I3C_CORE_COMMAND_QUEUE_PORT register. The application retrieves the data (in word) from the Rx-FIFO based on the command issued to the I3C_CORE_COMMAND_QUEUE_PORT register. The application can fill the data where the CPU can monitor the threshold status signal (I3C_CORE_INTR_STATUS[RX_THLD_STAT]) and then pop the data from the Rx-FIFO through I3C_CORE_RX_DATA_PORT

4. Check response status: The check response status phase detects the generation of response status and reads it to know the status of the issued transfer. The I3C_CORE_INTR_STATUS[RESP_READY_STAT] interrupt indicates the response available status and the application can read the available response through I3C_CORE_RESPONSE_QUEUE_PORT register

28.8.5 Flow to prepare the I3C block to switch to target mode

This section describes the programming flow to prepare the block in the controller mode to relinquish the ownership to a secondary controller. This programming flow is applicable only when the block is configured as a secondary controller. The block can initiate this on its own or as a response to a CR request from a target

- Process all the outstanding commands and responses
- Issue the GETACCCR CCC command as described in “Directed CCC transfer in controller mode”
- Wait for the I3C_CORE_INTR_STATUS[RESP_READY_STAT] interrupt. The interrupt gets generated by the block after the completion of the GETACCCR CCC transfer on the I3C bus
- Wait for either the I3C_CORE_INTR_STATUS[BUSOWNER_UPDATED_STAT] (success) or the I3C_CORE_INTR_STATUS[TRANSFER_ERR_STAT] (Error) interrupt status to know the result
- If the transfer completion status is success, read the response queue and the data buffer to get the received 7-bit dynamic address and parity of the intended target
- Set the I3C_CORE_DEVICE_CTRL[RESUME] bit to 1’b1, to enable the block to start responding for transfers as a target

- If the transfer completion status was an error, read the response queue to find out the type of error. If the data length of the response indicates a non-zero value, then read the data buffer to get the received 7-bit dynamic address and parity (error) of the intended target
- Set the I3C_CORE_DEVICE_CTRL[RESUME] bit to 1'b1, to enable the block to start processing the transfer commands as a controller

28.8.6 Flow for disable and FIFO/queue reset

To disable the controller, write the DEVICE_CTRL[ENABLE] register to 0. The controller might not immediately go into the disabled state. The controller goes into the disabled state in the following scenarios:

- If the controller is busy executing any I3C bus transfers (such as receiving/transmitting an I3C transfer or receiving an IBI), then the controller enters the disabled state only after completing either the transfer command with TOC=1 or the IBI reception
- In HDR mode of operation, the controller continues HDR requests until HDR transfers end with HDR Exit pattern

To indicate to the application that the controller has entered the disabled state, the controller sets the ENABLE bit to 0. The application, after setting the value of 0 to the ENABLE bit, must keep polling the ENABLE bit to check whether the bit value is 0. Once the ENABLE bit is read with a value of 0, the application can be sure that the controller is disabled. Once the controller enters the disabled state, it does not execute any commands from the command queue and does not provide any clock for a new IBI. The application is expected to flush/drain all the queues and the FIFOs before re-enabling the controller. In no other condition should the FIFOs be reset by the application. To enable the controller after it is disabled, the application must set the ENABLE bit to 1.

28.9 Using the initialized I3C in target mode

28.9.1 Private receive (controller write) transfer in target mode

This section describes the flow for the private receive (controller write) transfers. For private receive, the transfer is independent of any command from the application, that is, the private receive transfer does not require any command from the application.

The following flow must be followed by the application software for controller write transfer:

- The application fetches the threshold amount of data from the target block, once the I3C_CORE_INTR_STATUS[RX_THLD_STAT] interrupt is asserted
- The application receives the response interrupt once the number of responses in the response queue reaches the response threshold value. Once the response interrupt is received, the application must read the response status from the response queue
- If the error field bits are set to indicate that the received data from the target block has an error, the application must discard the received data from the memory and follow the error handling flow mentioned in "Error recovery flow"
- If the data length in the response field is greater than the total data received by the application and there is no error bit set, then it should read the residue data from the RX Buffer

28.9.2 Private transmit (controller read) transfer in target mode

This section describes the flow for the private transmit (controller read) transfers. For all private transmit transfers the application must provide a command to the target block indicating the data length to be transmitted and other related parameters whose details are described in the Transmit command data structure section.

The following flow is to be followed by the application software for controller read transfer:

- Issue the TX command to the target block once the I3C_CORE_INTR_STATUS[CMD_READY_STAT] interrupt is received, indicating the number of empty locations in command queue is greater than or equal to threshold value specified by CMD_EMPTY_BUF_THLD register
- The application pushes threshold amount of data from the target block once the I3C_CORE_INTR_STATUS[TX_THLD_STAT] interrupt is asserted
- The application receives the response threshold interrupt once the number of responses in the response queue reaches the response threshold value. Once the response interrupt is received, the application must read the response data from the response queue
- In case of error bits being set or the data length not equal to zero (because of controller early termination), the application must follow the error handling flow mentioned in “Error recovery flow”

28.9.3 Programming flow for generating target interrupt request

This section describes the programming flow for generating the target interrupt request.

- The application must set the I3C_CORE_TGT_INTR_REQ[TIR] bit to 1’b1 and the I3C_CORE_TGT_INTR_REQ [TIR_CTRL] field to 2‘b00 to enable the target block to send the TIR-IBI. The application must ensure that I3C_CORE_TGT_EVENT_STATUS[TIR_EN] bit is set to 1 (enabled by the controller) before programming I3C_CORE_TGT_INTR_REQ [SIR] bit. The target can send up to 4 bytes of payload along with the IBI. For example, to send payload along with IBI, target must additionally program the I3C_CORE_TGT_INTR_REQ[TIR_DATA_LENGTH] field with the number of bytes to be transmitted as IBI payload (valid: 0-4 bytes). In the case of IBI with payload the data must be programmed in the TGT_TIR_DATA register, one step earlier
- If the interrupt is enabled, the application must wait for the IBI completion status interrupt I3C_CORE_INTR_STATUS[IBI_UPDATED_STAT] and then read the TGT_INTR_REQ[IBI_STS] field to check if the IBI got accepted by the controller. If the controller responds with NACK, then the block reattempts the TIR automatically until the controller accepts (ACK) the TIR or until the controller disables the TIR event through DISEC CCC
- Alternatively, the application can poll the I3C_CORE_TGT_INTR_REQ[TIR] bit until it becomes 0 and then read the I3C_CORE_TGT_INTR_REQ[IBI_STS] field, to check if the IBI got accepted by the controller
- If the I3C_CORE_TGT_INTR_REQ[IBI_STS] field indicates that the IBI request is not attempted, the application can request TIR again if the required conditions (For more information, see the description of I3C_CORE_TGT_INTR_REQ[IBI_STS]) are met

For operation in async 0 mode, as target, the application needs to first enable the internal counter used for time stamping events, by writing to the TIMING_CNT_CTRL.CNT_ENABLE bit or optionally resetting the counter. This is a one-time procedure.

***Note**: Only one operation is allowed per access, either enable or reset.*

As a Target, if the operation in async 0 mode has been enabled, the application needs to set the TGT_IBI_REQ.TIR, TGT_IBI_REQ.MDB, and TGT_IBI_REQ.TS bits.

28.9.4 Programming flow for generating controller request

This section describes the programming flow for generating the controller request of the target in the secondary controller configuration.

- The application must set the I3C_CORE_TGT_INTR_REQ[CR] bit to enable the target block to send the IBI. The application must ensure that I3C_CORE_TGT_EVENT_STATUS[CR_EN] bit is set to 1 (enabled by the controller) before programming I3C_CORE_TGT_INTR_REQ [CR] bit

- If the interrupt is enabled, the application must wait for the IBI completion status interrupt I3C_CORE_INTR_STATUS[IBI_UPDATED_STAT] and then read the I3C_CORE_TGT_INTR_REQ[IBI_STS] field, to check if the IBI got accepted by the controller. If the controller responds with NACK, then the block reattempts the CR automatically until the controller accepts (ACK) the CR or disables the CR event through DISEC CCC
- Alternatively, the application can poll the I3C_CORE_TGT_INTR_REQ[CR] bit to 0 and then read the I3C_CORE_TGT_INTR_REQ[IBI_STS] field, to check if the IBI got accepted by the controller
- If the I3C_CORE_TGT_INTR_REQ[IBI_STS] field indicates that the IBI request was not attempted, the application can request CR again if the required conditions (see the description of I3C_CORE_TGT_INTR_REQ[IBI_STS]) are met

For operation in async 0 mode, as controller, the application needs to first enable the internal counter used for time stamping events, by writing to the TIMING_CNT_CTRL.CNT_ENABLE bit or optionally resetting the counter. This is a one-time procedure.

***Note**: Only one operation is allowed per access, either enable or reset.*

As a controller, the time stamping events from the target (for example, IBIs) needs to be enabled per each target individually, by writing to the DEV_ADDR_TABLE1_LOC1.TS field. This is to be done after sending a GETCAPS CCC, to determine if the target is capable of operating in async timing control mode 0, and the SETXTIME CCC to enable operation in that mode.

28.9.5 Programming flow to disable HJ capability

This section describes the programming flow for disabling HJ capability when the HJ feature is already present by configuration.

- The application can program I3C_CORE_TGT_EVENT_STATUS[HJ_EN] to 0 to disable the HJ capability of the block. When this bit is set to 0, the block does not generate HJ request and behaves as a non-HJ device
- When this bit is set to 1, the block generates HJ request
- The programming of I3C_CORE_TGT_EVENT_STATUS[HJ_EN] occurs only once, specifically during the initial configuration and prior to enabling (I3C_CORE_DEVICE_CTRL[ENABLE]) the block

28.9.6 Programming flow to prepare the I3C block to switch to controller mode

This section describes the programming flow to prepare the block in the target mode to take the controllership from the current controller. This programming flow is applicable only when the block is configured as a secondary controller.

- Process all the outstanding responses and data from the response queue and Rx-FIFO respectively, received in the target mode
- Issue the controller request (CR) to signal the current controller for the controllership request
- Prepare the block to accept the GETACCCR CCC by clearing the bit I3C_CORE_DEVICE_CTRL_EXTENDED[GETACCCR_ACK_CTRL]
- Wait for the I3C_CORE_INTR_STATUS[BUSOWNER_UPDATED_STAT] interrupt. The interrupt is generated by the block after the successful completion of the GETACCCR CCC transfer, issued by the current controller
- Process all the outstanding responses and data from the response queue and Rx-FIFO respectively, received after the controller request is accepted (if applicable)
- Flush all the queues and FIFOs to prepare the block in controller mode from a clean state
- Program the DAT with the associated target device information, received through the broadcast DEFTGTS CCC transfer from the main Controller

***Note**: The block will receive information about the other devices in the bus through a DEFTGTS CCC, and this information becomes available in the SEC_DEV_CHAR* family of registers. The application must copy the necessary information from SEC_DEV_CHAR* into the DEVICE_ADDR_TABLE* registers*

- Set the I3C_CORE_DEVICE_CTRL[RESUME] bit to 1'b1, to enable the block to start processing the transfer commands

28.9.7 Command pipeline and aggregation of response queue threshold interrupt

To increase the throughput of the target block, multiple commands can be pipe-lined in the command queue for which the flow for the application is as follows:

- For multiple commands, the command buffer threshold should be programmed to value more than 0 (0 signifies 1 location; 1 location equal to one 32-bit entry)
- The response queue threshold could be programmed to value more than 0 (0 signifies 1 location). This ensures that the application is not interrupted for every interrupt and response queue threshold interrupt aggregation is possible
- The transfer error stat interrupt bit must be set in the target interrupt enable register which when set indicates that the last response in the response queue has the error bit set. This interrupt is to safeguard deadlock condition in target block which can happen when the response having the error field set occupies a position in the response queue which does not meet the threshold requirement and response threshold interrupt is not generated. Once any error is encountered the target block goes into halt state, expecting the application to resume it by setting the resume bit in the device control register. Upon receiving the transfer error stat interrupt, application must read the response queue until it gets to the response that has the error and set the resume bit. This is outlined in the section "Error recovery flow"

28.9.8 Error recovery flow

The I3C controller read or write transactions by I3C target block, can encounter an error condition while transmitting or receiving data from the I3C controller. The transfer error interrupt (TRANSFER_ERR_STAT bit in I3C_CORE_INTR_STATUS register) is set whenever the target block encounters an error condition.

Depending on the response queue status level the RESP_READY_STAT interrupt might be set, if the error response fulfills the response queue threshold value. If the RESP_READY_STAT interrupt is not set, the target application must read the RESP_BUF_BLR value to understand how many responses are present in the response queue. The last response in the response queue is the error response. The target application must read all the response from the response queue and for the last response, it should analyze the ERR_STATUS field of the response. In case if the ERR_STATUS field indicates overflow, Parity, or CRC error, the target application must flush the RX FIFO. The target block NACKs, all transfers once it has encountered an error until the RESUME bit is set in the I3C_CORE_DEVICE_CTRL register from the target application and until the error status is cleared from the I3C_CORE_CCC_DEVICE_STATUS register by GETSTATUS CCC. For any other error status like underflow error or controller early termination, the target application is expected to reset the TX FIFO and CMD FIFO before applying the resume in I3C_CORE_DEVICE_CTRL register.

28.9.8.1 T0-T5 error handling

Detection and recovery from T0-T5 errors summarized MIPI I3C specification is supported by the block. Detection and recovery are transparent to the target application and is internally handled by the block.

T0-T5 error is supported in Target IP T6 is not a mandatory feature, hence, not supported. Handling of T0-T5 is internal to the target and transparent to the application that is, it does not require the intervention of the target application and is not reported in the response.

In case of:

- T0 and T1: All transfers are ignored until HDR exit pattern is detected
- T2: For CCC parity error, the CCC value is not updated and the old values are retained. For private read and write transfer, the appropriate error filed in response is updated
- T3-T4: Updates due to CCCs are halted and the target retains the current values (without updating), until a STOP/START is observed
- T5: Illegally formatted CCC is handled internally. The CCC value is not updated and the old values are retained. No information is provided to the application

28.9.9 CCC updated interrupt flow

The CCC's like maximum read length (MRL), maximum write length (MWL), activity states and IBI request enable are passed on to the application by I3C target block by updating the I3C_CORE_TGT_EVENT_CTRL register and giving an appropriate interrupt (CCC_UPDATED_STAT interrupt). Upon receiving the interrupt, the target application must read the I3C_CORE_TGT_EVENT_CTRL register to understand which CCC has been received. If the received CCC is SETMRL CCC, then the I3C Target waits for the RESUME bit to be set in the I3C_CORE_DEVICE_CTRL register. All transfers are NACK'ed by the target block during the period when it is waiting for the RESUME from the application. On receiving of the SETMRL CCC, the application can flush the commands present in Command Queue and frame new sets of commands with the data length in the command pertaining to the MRL value received from the controller.

The target block does not consider the MRL or the MWL values received from the controller and if the data length in the command is more than the MRL value the controller can do an early termination of the controller read transfer (subject to the controller implementation).

28.9.10 Flow for disable and Tx/Rx/CMD/Response queue request

To disable the I3C target, the I3C_CORE_DEVICE_CTRL[ENABLE] register must be written zero. The block depends on the availability of SCL to complete disable operation. So, the target block may not immediately go into a disable state. The target block goes into the disable state

- Only after completion of the ongoing transfer in SDR mode, any subsequent transfers are NACK'ed
- In HDR mode of operation, the block continues to respond to any HDR requests it receives until an exit pattern is detected
- If the block is disabled when the I3C bus is Idle (that is, no SCL present), then the operation is completed only when SCL is observed on the bus

To indicate to the application that the target block has entered the disabled state, the block sets the ENABLE bit to 0.

The application, after setting the value of 0 to the ENABLE bit, must keep polling the ENABLE bit to check whether the bit value is zero. Once the ENABLE bit is read with a value of zero, the application can be sure that the target block is disabled. The application can reset the command, transmit, and receive FIFO only when the target block is disabled or when the target block is in the Halt state after any error during transmitting or receiving of the data from the I3C controller. In no other conditions, the FIFOs should be reset by the application. After the disable, if the application chooses not to flush the FIFOs, the target block continues with the command and the data present in the FIFO after the disabling condition is removed by the application. To enable the target block after it has been disabled, the application must set the ENABLE bit to 1. Enabling the target block might not be instantaneous, because it is dependent on the availability of SCL clock on the I3C line. If the SCL clock is present, the target block takes two SCL clocks to come out of the disable state.

28.10 List of supported CCCs

28.10.1 Controller mode

The CCCs that are supported in this operating mode are the following:

Table 310 CCCs in controller mode

CCC Code	CCC Type	CCC Name	Supported Level/Notes
0x00	Broadcast	ENEC	Fully supported
0x01	Broadcast	DISEC	Fully supported
0x02	Broadcast	ENTAS0	Fully supported
0x03	Broadcast	ENTAS1	Fully supported
0x04	Broadcast	ENTAS2	Fully supported
0x05	Broadcast	ENTAS3	Fully supported
0x06	Broadcast	RSTDAA	Fully supported
0x07	Broadcast	ENTDAA	Fully supported
0x08	Broadcast	DEFTGTS	Fully supported
0x09	Broadcast	SETMWL	Fully supported
0x0A	Broadcast	SETMRL	Fully supported
0x0B	Broadcast	ENTTM	Fully supported
0x0C	Broadcast	SETBUSCON	Fully supported
0x0D - 0x11	-	-	N/A
0x12	Broadcast	ENDXFER	Supported in the latest DWC_mipi_i3c_v111 release
0x13 - 0x1E	-	<MIPI Reserved>	N/A
0x1F	N/A	-	Not supported
0x20	Broadcast	ENTHDR0	Fully supported
0x21	Broadcast	ENTHDR1	Not supported (see Note 2)
0x22	Broadcast	ENTHDR2	Not supported (see Note 2)
0x23	Broadcast	ENTHDR3	Not supported (see Note 2)
0x24	Broadcast	ENTHDR4	Not supported (see Note 2)
0x25	Broadcast	ENTHDR5	Not supported (see Note 2)

(table continues...)

Table 310 (continued) CCCs in controller mode

CCC Code	CCC Type	CCC Name	Supported Level/Notes
0x26	Broadcast	ENTHDR6	Not supported (see Note 2)
0x27	Broadcast	ENTHDR7	Not supported (see Note 2)
0x28	Broadcast	SETXTIME	Fully supported
0x29	Broadcast	SETAASA	Fully supported
0x2A	Broadcast	RSTACT	Fully supported
0x2B	Broadcast	DEFGRPA	Not supported
0x2C	Broadcast	RSTGRPA	Not supported
0x2D	Broadcast	MLANE	Not supported
0x2E - 0x48	-	<MIPI Reserved>	N/A
0x49 - 0x4C	Broadcast	<MIPI Reserved>	N/A
0x4D - 0x57	Broadcast	<MIPI Reserved>	N/A
0x58 - 0x5B	Broadcast	<MIPI Reserved>	N/A
0x5C - 0x60	Broadcast	<MIPI Reserved>	N/A
0x61 - 0x7F	Broadcast	<Vendor Extensions>	Partially supported (only simple vendor specific CCCs)
0x80	Direct	ENEC	Fully supported
0x81	Direct	DISEC	Fully supported
0x82	Direct	ENTAS0	Fully supported
0x83	Direct	ENTAS1	Fully supported
0x84	Direct	ENTAS2	Fully supported
0x85	Direct	ENTAS3	Fully supported
0x86	Direct	RSTDAA	Fully supported (See Note 1 below)
0x87	Direct Set	SETDASA	Fully supported
0x88	Direct Set	SETNEWDA	Fully supported
0x89	Direct Set	SETMWL	Fully supported
0x8A	Direct Set	SETMRL	Fully supported
0x8B	Direct Get	GETMWL	Fully supported
0x8C	Direct Get	GETMRL	Fully supported
0x8D	Direct Get	GETPID	Fully supported
0x8E	Direct Get	GETBCR	Fully supported
0x8F	Direct Get	GETDCR	Fully supported

(table continues...)

Table 310 (continued) CCCs in controller mode

CCC Code	CCC Type	CCC Name	Supported Level/Notes
0x90	Direct Get	GETSTATUS	Fully supported
0x91	Direct Get	GETACCCR	Fully supported
0x92	Direct	ENDXFER	Is supported in the latest DWC_mipi_i3c_v111 release
0x93	Direct Set	SETBRGTGT	Not supported
0x94	Direct Get	GETMXDS	Fully supported
0x95	Direct Get	GETCAPS	Fully supported
0x96	Direct	SETROUTE	Not supported
0x97	Direct	D2DXFER	Not supported
0x98	Direct	SETXTIME	Fully supported
0x99	Direct	GETXTIME	Fully supported
0x9A	Direct	RSTACT	Fully supported
0x9B	Direct	SETGRPA	Not supported
0x9C	Direct	RSTGRPA	Not supported
0x9D	Direct	MLANE	Not supported
0x9E - 0xBF	Direct	<MIPI Reserved>	N/A
0xC0 - 0xC3	Direct	<MIPI Reserved>	N/A
0xC4 - 0xD6	Direct	<MIPI Reserved>	N/A
0xD7 -0xDA	Direct	<MIPI Reserved>	N/A
0xDB - 0xDF	Direct	<MIPI Reserved>	N/A
0xE0 - 0xFE	Direct	<Vendor Extensions>	Partially supported (only simple vendor specific CCCs)
0xFF	-	<MIPI Reserved>	-

***Notes**:*

1. *RSTDAA in direct mode (CCC 0x86) is deprecated in MIPI I3C v1.1.1. Basic and should not be used*

2. *Only HDR-DDR mode is supported by the block (other HDR modes are not supported)*

28.10.2 Target mode

The CCC transfers shown in below table are supported when I3C is operating in target mode. All unsupported CCC commands are ignored (NACKED) by the target controller.

Table 311 CCCs supported in target mode

CCC type	CCC name
Broadcast	ENTDAA
Direct	SETDASA
Direct	GETSTATUS
Direct	GETMXDS
Broadcast	ENTHDR0
Broadcast	ENTHDR1
Broadcast	ENTHDR2
Direct	GETMRL
Broadcast, Direct	SETMRL
Direct	GETMWL
Broadcast, Direct	SETMWL
Broadcast, Direct	ENEC
Broadcast, Direct	DISEC
Broadcast	RSTDAA
Direct	SETNEWDA
Direct	GETPID
Direct	GETBCR
Direct	GETCAPS
Broadcast, Direct	ENTAS0
Broadcast, Direct	ENTAS1
Broadcast, Direct	ENTAS2
Broadcast, Direct	ENTAS3
Broadcast	DEFTGTS
Direct	GETACCCR
Direct	GETDCR
Broadcast	SETAASA
Direct	SETGRPA
Broadcast, Direct	RSTGRPA
Direct	RSTACT

28.11 Power modes

This I3C block works only in the active power mode.

Some MMIO registers at pre enable level that are retained during the Deep Sleep power mode are as follows:

- CTRL.ENABLED

The SRAM can be retained based on the separate supply set VCCSRAM_VSSD_SET during Deep Sleep power mode.

28.12 Registers summary

Table 312 I3C registers

Symbol	Name	Description
I3C_CTRL	Block level control register	Control register allows enabling the I3C
I3C_TGT_STATUS	Target status register	This register provides additional information when functioning as Target
I3C_CCC_GETSTATUS	Target get status register	This register provides additional information on GETSTATUS CCC's received when functioning as Target
I3C_TIMING_CNT_CTRL	Timing counter control register	Controls the counter used for time stamping I3C relevant events, especially when communicating with asynchronous timing control mode 0 enabled
I3C_TIMING_CNT0	Timing counter 0	Most significant bits of the counter
I3C_TIMING_CNT1	Timing counter 1	Least significant bits of the counter
I3C_CORE_DEVICE_CTRL	Device control register	This register provides general configuration options for the block
I3C_CORE_DEVICE_ADDR	Device address register	When functioning as the active controller, this register is used to program the blocks' self-assigned dynamic address and its respective valid bit. When functioning as a target, this register reflects the static address and assigned dynamic address and their respective valid bits

(table continues...)

Table 312 (continued) I3C registers

Symbol	Name	Description
I3C_CORE_COMMAND_QUEUE_PORT	COMMAND_QUEUE_PORT	When functioning as the active controller, the command descriptor structure is used to schedule the transfers to other devices on I3C bus. There are four types of commands defined. **1.** Transfer command **2.** Transfer argument **3.** Short data argument **4.** Address assignment command When functioning as a Target, the command queue port is used to push commands which enable the block to respond with data for a private read command from the active controller. The reset value of this register is not defined as it is internally mapped to a queue

(table continues...)

Table 312 (continued) I3C registers

Symbol	Name	Description
I3C_CORE_RESPONSE_QUEUE_PORT	RESPONSE_QUEUE_PORT	When functioning as the active controller, the response status for each command is written into the response queue by the controller if the ROC (response on completion) bit is set or if transfer error occurs. Not doing so might result in execution of new commands getting stalled. A new command is executed only if there is space available in response queue to push the corresponding response. When functioning as a target, the response status for each command is written into the response queue. Not doing so might result in execution of new commands getting stalled. A new command is executed only if there is space available in Response queue to push the corresponding response. In both cases, the response queue can be read through this register. It is expected that this register is read whenever RESP_READY_STAT_INTR bit is set in INTR_STATUS register
I3C_CORE_TX_RX_DATA_PORT	Transmit/Receive data port register	Transmit data port for write access and receive data port for read access. This field is mapped to the TX data buffer when writing and to the RX data buffer when reading. The data is always packed in 4-byte aligned data words. If the command length is not aligned to the 4-bytes, then the additional data bytes have to be ignored

(table continues...)

Table 312 (continued) I3C registers

Symbol	Name	Description
I3C_CORE_IBI_QUEUE_DATA	In-Band interrupt queue data register	In-Band interrupt queue data register (shared offset with IBI_QUEUE_STATUS). This register is used when functioning as the active controller after application receives an IBI and the status confirms that it is an IBI with payload. When read from, returns the data from the IBI Queue.
I3C_CORE_IBI_QUEUE_STATUS	In-Band interrupt queue status register	In-Band interrupt status register (shared offset with IBI_QUEUE_DATA). This register is used when functioning as the active controller. It is expected that this register is read whenever INTR_STATUS.IBI_THLD_STS is set. When read from, indicates how the controller responded to incoming IBI (TIR, CR, and HJ) and whether IBI payload is available
I3C_CORE_QUEUE_THLD_CTRL	Queue threshold control register	This register is used to program the threshold settings for the queues in the I3C block
I3C_CORE_DATA_BUFFER_THLD_CTRL	Data buffer threshold control register	This register is used to program the threshold settings for the data buffers in the I3C block
I3C_CORE_IBI_QUEUE_CTRL	IBI queue control register	This register is used to control whether or not to intimate the application if an IBI request is rejected (Nacked). This register is used only when functioning as the active controller
I3C_CORE_RESET_CTRL	Reset control register	This register is used for general software reset and for individual buffer reset

(table continues...)

Table 312 (continued) I3C registers

Symbol	Name	Description
I3C_CORE_TGT_EVENT_STATUS	Target event status register	This register indicates the status/values of some events/ controls that are relevant when functioning as a Target. These values are set by CCCs initiated by the active controller
I3C_CORE_INTR_STATUS	Interrupt status register	This register indicates the status of interrupt
I3C_CORE_INTR_STATUS_EN	Interrupt status enable register.	The interrupt status is updated in INTR_STATUS register only if corresponding status enable bit is set
I3C_CORE_INTR_SIGNAL_EN	Interrupt signal enable register	The interrupt pin is triggered based on INTR_STATUS only if corresponding signal enable bit is set
I3C_CORE_INTR_FORCE	Interrupt force enable register	Individual interrupts can be forcefully triggered if corresponding force enable bit is set, provided the corresponding bit in the INTR_STATUS_EN register is set
I3C_CORE_QUEUE_STATUS_LEVEL	Queue status level register	This register reflects the status level of the queues in the I3C block
I3C_CORE_DATA_BUFFER_STATUS_LEVEL	Data buffer status level register	This register reflects the number of empty/valid entries in the I3C TX/RX buffers
I3C_CORE_PRESENT_STATE	Present state register	The user can get status of the I3C block from this 32-bit read-only register. This register is relevant when functioning as the active controller or as a target. It should be used only for debug information related to the block's internal states
I3C_CORE_CCC_DEVICE_STATUS	Device operating status register	This register valid only when functioning as a target, and reflects the data which the target sends in response to GETSTATUS CC by the active controller

(table continues...)

Table 312 (continued) I3C registers

Symbol	Name	Description
I3C_CORE_DEVICE_ADDR_TABLE_POINTER	Pointer for device address table	This register is only valid when functioning as the active controller
I3C_CORE_DEV_CHAR_TABLE_POINTER	Pointer for device characteristics table	This register is valid only when functioning as the active controller
I3C_CORE_VENDOR_SPECIFIC_REG_POINTER	Pointer for vendor specific registers.	This register is only valid when functioning as the active controller.
I3C_CORE_TGT_MIPI_ID_VALUE	I3C MIPI manufacturer ID register	This register is used When functioning as a target.
I3C_CORE_TGT_PID_VALUE	I3C normal provisional ID register	This register is only valid when functioning as a target, and specifies bits [31:0] of the MIPI provisioned ID for this I3C device
I3C_CORE_TGT_CHAR_CTRL	I3C target characteristic register	This register is used When functioning as a target.
I3C_CORE_TGT_MAX_LEN	I3C max write/read length register	This register is used when functioning as a target.
I3C_CORE_MAX_READ_TURNAROUND	MXDS maximum read turnaround time	This register is only valid when functioning as a target. The values are used in the data response of the target to GETMXDS CCC
I3C_CORE_MAX_DATA_SPEED	Maximum data speed register	This register is only valid when functioning as a target. The values are used in the data response of the target to GETMXDS CCC
I3C_CORE_TGT_INTR_REQ	Target interrupt request register	This register is only valid when functioning as a target, and is used to program the bits that control target interrupt requests and to read the status of the interrupt
I3C_CORE_TGT_TIR_DATA	Target interrupt request data register	This register holds the four byte of TIR DATA that will be passed after the MDB when TIR is initiated by the application on writing into TIR bit in TGT_INTR_REQ register.

(table continues...)

Table 312 (continued) I3C registers

Symbol	Name	Description
I3C_CORE_TGT_IBI_RESP	Target IBI response register	This register stores IBI status and number of data byte remaining due to early
I3C_CORE_DEVICE_CTRL_EXTENDED	Device control extended register	This register defines the initial role of the I3C block on the bus, and whether the block can become the active Controller
I3C_CORE_SCL_I3C_OD_TIMING	SCL I3C Open drain timing register	This register sets the SCL clock high period and low period count for I3C Open Drain transfers. The count value takes the number of cycles of CORE_CLK to maintain the SCL High/Low phases
I3C_CORE_SCL_I3C_PP_TIMING	SCL I3C push pull timing register	This register sets the SCL clock high period and low period count for I3C Push Pull transfers. The count value takes the number of cycles of CORE_CLK to maintain the SCL High/Low phases
I3C_CORE_SCL_I2C_FM_TIMING	SCL I2C fast mode timing register	This register sets the SCL clock high period and low period count for I2C Fast mode transfers. The count value takes the number of cycles of CORE_CLK to maintain the SCL High/Low phases
I3C_CORE_SCL_I2C_FMP_TIMING	SCL I2C fast mode plus timing register	This register sets the SCL clock high period and low period count for I2C Fast mode plus transfers. The count value takes the number of cycles of CORE_CLK to maintain the SCL High/Low phases
I3C_CORE_SCL_EXT_LCNT_TIMING	SCL extended low count timing register	This register sets the extended low period of SCL for SDR1/2/3/4 speed transfers. The extended low period is applied in SDR modes 1/2/3/4 when the external Target device has a low clock-to-data turnaround time

(table continues...)

Table 312 (continued) I3C registers

Symbol	Name	Description
I3C_CORE_SCL_EXT_TERMN_LCNT_TIMING	SCL termination bit low count timing register	This register is used to extend the SCL Low period for Read Termination Bit in SDR0/1/2/3/4 speed transfers
I3C_CORE_SDA_HOLD_SWITCH_DLY_TIMING	SDA hold and mode switch delay timing register	The bits [18:16] of this register are used to control the hold time of SDA during transmit mode in SDR & DDR transfers
I3C_CORE_BUS_FREE_AVAIL_TIMING	Bus free and available timing register	This register sets the bus free time for initiating the transfer when functioning as the active controller or generating IBI when functioning as a target. The numbers are in terms of CORE_CLK cycles
I3C_CORE_BUS_IDLE_TIMING	Bus idle timing register	This register is only valid when functioning as the Target. This register programs the Bus-Idle time period used when a device dynamically joins (hot-join) the I3C bus. This register is used to store the duration, used to detect the Bus-Idle condition if SCL and SDA are held high for the mentioned duration. The numbers are in terms of CORE_CLK cycles
I3C_SCL_LOW_CTR_EXT_TIMEOUT	SCL LOW controller timeout register	The SCL low controller extended timeout register is used to define the duration of the SCL low bus reset patter
I3C_CORE_RELEASE_SDA_TIMING	Release SDA count value	This register is used only in target mode of operation. This register is used to store the duration, used to detect the SCL not toggling and SDA shall be released
I3C_CORE_HDR_FLOW_CONTROL	Target HDR flow control register	This register reflects the flow control mechanism programmed by the controller through ENDXFER CCC
I3C_EXT_TX_DATA_PORT_[X]	Directed read vendor-specific CCC 'X' TX data port register	Transfer data port register for directed read vendor-specific CCC 'X', where 'X' = 0 to 3

(table continues...)

Table 312 (continued) I3C registers

Symbol	Name	Description
I3C_CORE_EXT_0_1_DATA_BUF_STS_LEVEL	Vendor-specific CCC types 0 and 1 data buffer status level register	Data buffer status level register for vendor-specific CCC types 0 and 1
I3C_CORE_EXT_2_3_DATA_BUF_STS_LEVEL	Vendor-specific CCC types 2 and 3 data buffer status level register	Data buffer status level register for vendor-specific CCC types 2 and 3
I3C_CORE_EXT_CMD_REG_[X]	TID, CCC type value and defining byte of vendor-specific CCC 'X' register	CCC type and data length register For vendor-specific CCC type 'X', where 'X' = 0 to 3
I3C_CORE_DEV_CHAR_TABLE[X]_LOC[Y]	Device characteristic table location 'Y' of device 'X'	Device Characteristic Table Location 'Y' of Device 'X'. This register is used in controller mode of operation. 'Y' takes location values from 1 to 4 while 'X' takes device values from 1 to 11. These Registers contains the LSB and MSB 32-bit value of Provisional-ID, Device characteristic value (DCR), Bus characteristic value (BCR) and Device dynamic address assigned in the respective locations of devices. DEV_CHAR_TABLE[X]_LOC1[Y] and SEC_DEV_CHAR_TABLE[X] share the same offset. This register is valid when operating as the active controller
I3C_CORE_DEV_ADDR_TABLE_LOC[X]	Device address table of device 'X'	Device address table of device 'X'. This register is used in controller mode of operation. 'X' takes device values from 1 to 11. This register is valid when operating as the active controller

(table continues...)

Table 312 (continued) I3C registers

Symbol	Name	Description
I3C_CORE_SEC_DEV_CHAR_TABLE[X]	Secondary controller device characteristic table location of device X	Second device characteristic table location 'X'. 'X' takes values from 1 to 11. DEV_CHAR_TABLE[X]_LOC1[Y] and SEC_DEV_CHAR_TABLE[X] share the same offset. Device Characteristic Table Location of Device #X. Used when this secondary controller is preparing to become the Active Controller and collecting the responses from a DEFTGTS CCC

29 Universal serial bus - full speed (USB-FS)

The USB-FS subsystem of PSC3M/P6 MCU implements a full speed USB device compliant to USB 2.0 specification and low/full speed USB host.

29.1 Overview

Figure 348 shows the block diagram of USB subsystem. It comprises of USB device block, USB host block, and USB PHY.

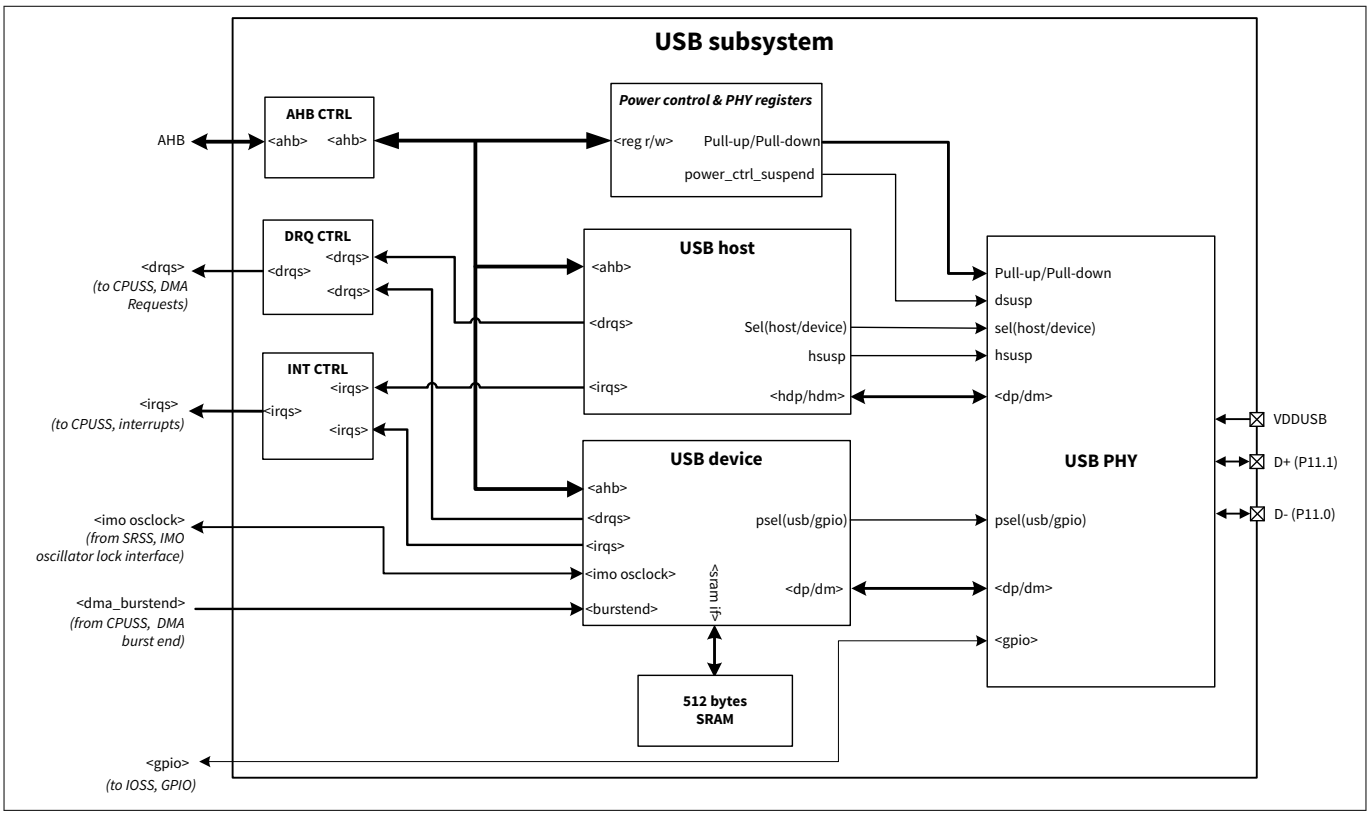


Figure 348 Block diagram of USB subsystem

The USB device block supports a maximum of eight data endpoints in addition to one control endpoint (EP0). The host block supports up to sixteen data endpoints, depending on software configuration. Each data endpoint can be individually programmed as IN or OUT and supports all four transfer types: control, bulk, interrupt and isochronous.

The USB device supports three modes for CPUSS to access USB data: No-DMA mode, Manual DMA mode, and Automatic DMA mode. The maximum USB packet size supported is 512 bytes in the first two modes and 1023 bytes in Automatic DMA mode. The maximum USB packet size supported by the host is 256 bytes.

The following chapters describe the features, architecture, and operations of the USB device and host modes.

29.2 USB device mode

The PSC3M/P6 MCU USB block can act as a USB device that communicates with a USB host. The USB block is available as a fixed-function digital block in the PSC3M/P6 MCU. It supports full-speed communication (12 Mbps) and is designed to be compliant with the USB Specification Revision 2.0. USB devices can be designed for plug-and-play applications with the host and also support hot swapping. This chapter details the PSC3M/P6 MCU USB block and transfer modes. For details about the USB specification, see the USB implementers forum website.

29.2.1 Features

The USB device in the PSC3M/P6 MCU has the following features:

- Complies with USB Specification 2.0
- Supports full-speed peripheral device operation with a signaling bit rate of 12 Mbps
- Supports eight data endpoints and one control endpoint
- Provides a shared 512-byte buffer for data endpoints
- Provides dedicated 8-byte memory for control endpoint (EP0)
- Supports four types of transfers – bulk, interrupt, isochronous, and control
- Supports bus- and self-powered configurations
- Enables USB suspend mode for low power
- Supports three types of logical transfer modes:
  - No DMA mode (Mode 1)
  - Manual DMA mode (Mode 2)
  - Automatic DMA mode (Mode 3)
- Supports maximum packet size of 512 bytes using Mode 1 and Mode 2, and maximum packet size of 1023 bytes for isochronous transfer using Mode 3
- Provides integrated 22 Ω USB termination resistors on the D+ and D–lines, and 1.5 kΩ pull-up resistor on the D+ line
- Supports USB 2.0 Link Power Management (LPM)
- Can be configured using the USB Device Configurator available with the ModusToolbox™ software

29.2.2 Architecture

Figure 349 illustrates the device architecture of the USB block in PSC3M/P6 MCUs. It consists of the USB Physical Layer (USB PHY), Serial Interface Engine (SIE), and the local 512-byte memory buffer.

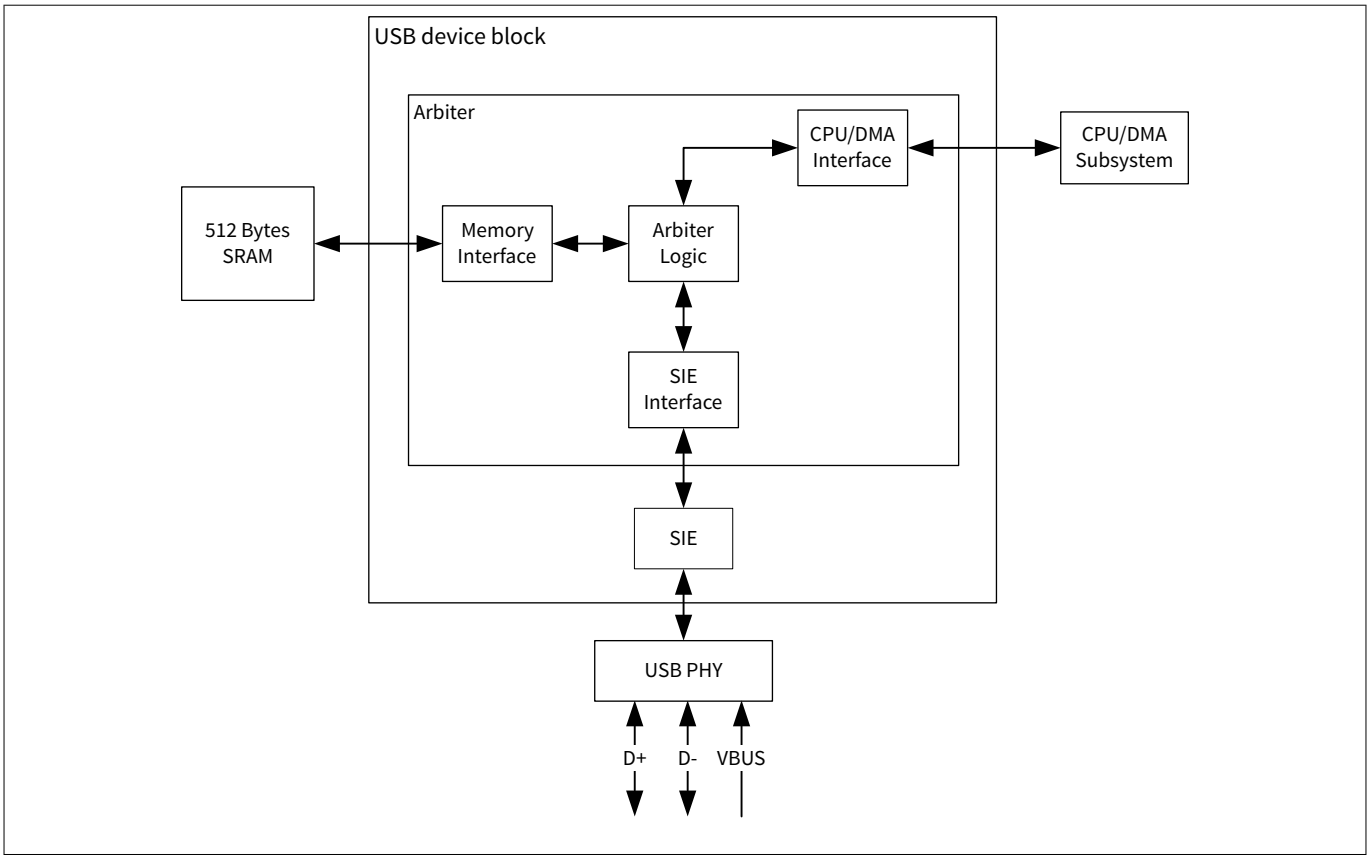


Figure 349 USB device block diagram

29.2.2.1 USB physical layer (USB PHY)

The USB PHY allows physical layer communication with the USB host through the D+, D–, and VBUS pins. It handles the differential mode communication with the host, VBUS detection, and monitoring USB bus events such as Single-ended 0 (SE0), suspend, resume, start of packet (SOP), and so on.

29.2.2.2 Serial interface engine (SIE)

The SIE handles the decoding and creating of data and control packets during transmit and receive. It decodes the USB bit streams into USB packets during receive, and creates USB bit streams during transmit. The following are the features of the SIE:

- Conforms to USB Specification 2.0
- Supports one device address
- Supports eight data endpoints and one control endpoint
- Supports interrupt trigger events for each endpoint
- Integrates an 8-byte buffer in the control endpoint

The registers for the SIE are mainly used to configure the data endpoint operations and the control endpoint data buffers. This block also controls the interrupt events available for each endpoint.

29.2.2.3 Arbiter

The Arbiter handles access of the SRAM memory by the endpoints. The SRAM memory can be accessed by the CPU, DMA, or SIE. The arbiter handles the arbitration between the CPU, DMA, and SIE. The arbiter consists of the following blocks:

- SIE interface module
- CPU/DMA interface
- Memory interface
- Arbiter logic

The arbiter registers are used to handle the endpoint configurations, read address, and write address for the endpoints. It also configures the logical transfer type required for each endpoint.

29.2.2.3.1 SIE interface module

This module handles all the transactions with the SIE. The SIE reads data from the SRAM memory and transmits to the host. Similarly, it writes the data received from the host to the SRAM memory. These requests are registered in the SIE Interface module and are handled by this module.

29.2.2.3.2 CPU/DMA interface block

This module handles all transactions with the CPU and DMA. The CPU requests for reads and writes to the SRAM memory for each endpoint. These requests are registered in this interface and are handled by the block. When the DMA is configured, this interface is responsible for all transactions between the DMA and USB. The block supports the DMA request line for each data endpoint. The behavior of the DMA depends on the type of logical transfer mode configured in the configuration register.

29.2.2.3.3 Memory interface

The memory interface is used to control the interface between the USB and SRAM memory unit. The maximum memory size supported is 512 bytes organized as 256 x 16-bit memory unit. This is a dedicated memory for the USB. The memory access can be requested by the SIE or by the CPU/DMA. The SIE Interface block and the CPU/DMA Interface block handle these requests.

29.2.2.3.4 Arbiter logic

This is the main block of the arbiter. It is responsible for arbitration for all the transactions that happen in the arbiter. It arbitrates the CPU, DMA, and SIE access to the memory unit and the registers. This block also handles memory management, which is either ‘Manual’ or ‘Automatic’. In Manual memory management, the read and write address manipulations are done by the firmware. In Automatic management, all the memory handling is done by the block itself. This block takes care of the buffer size allocation. It also handles common memory area. This block also handles the interrupt requests for each endpoint.

29.2.3 Operation

29.2.3.1 USB clocking scheme

The USB device block should be clocked at 48 MHz with an accuracy of ±0.25%. In the PSC3M/P6 MCU, CLK_HF3 is the clock source. The USB device block also requires a 100-kHz peripheral clock for USB bus reset timing. The required USB clock can be generated using one of the following clocking schemes:

- IMO (trimmed with USB) -> PLL -> CLK_HF3
- ECO (with the required accuracy) -> FLL -> CLK_HF3
- ECO (with the required accuracy) -> PLL -> CLK_HF3
- Use external clock (EXTCLK) with the required accuracy

29.2.3.2 USB PHY

The USB includes the transmitter and receiver (transceiver), which corresponds to the USB PHY. Figure 350 shows the PHY architecture. The USB PHY also includes the pull-up resistor on the D+ line to identify the device as full-speed type to the host. The PHY integrates the 22-Ω series termination resistors on the USB line. The signal between the USB device and the host is a differential signal. The receiver receives the differential signal from the host and converts it to a single-ended signal for processing by the SIE. The transmitter converts the single-ended signal from the SIE to the differential signal, and transmits it to the host. The differential signal is given to the upstream devices at a nominal voltage range of 0 V to 3.3 V.

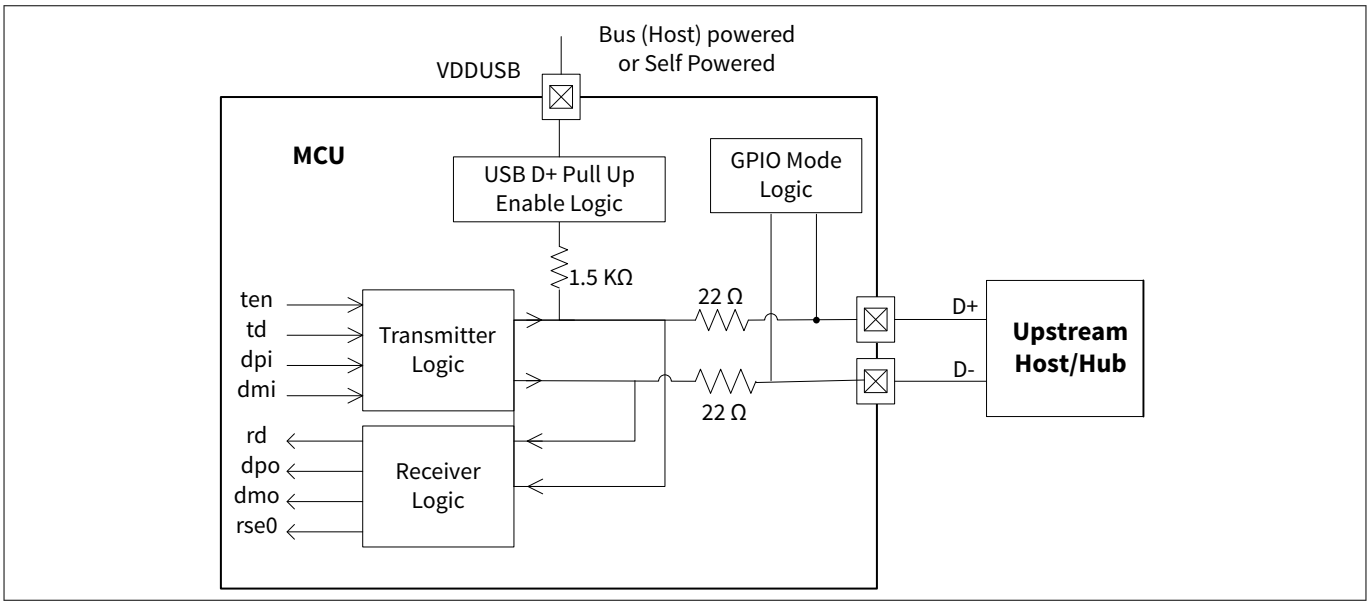


Figure 350 USB PHY architecture

29.2.3.2.1 Power scheme

The USB PHY is powered by the VBUS power pad of the PSC3M/P6 MCU. The VBUS pad can be driven either by the host VBUS (bus-powered) or an external power supply (self-powered).

The USB PHY needs a nominal voltage of 3.3 V for its communication with the host.

29.2.3.2.2 VBUS detection

USB devices can either be bus-powered (power sourced from the host) or self-powered (power sourced from an external power supply). The VDDUSB power pad pin powers the USB PHY and USB I/Os (D+ and D– pins). The presence of VBUS can be detected using the following steps:

1. Enable the interrupt on VDDUSB power pad. For this write ‘1’ to the VDDIO_ACTIVE[3] bit of GPIO_VDD_INTR_MASK register

2. VDDIO_ACTIVE[3] bit of the supply detection interrupt register (GPIO_VDD_INTR) is set to '1' whenever a change to supply is detected. Clear the interrupt cause by writing '1' to the bitfield
3. Check the status of the VDD_ACTIVE[3] bit of the external power supply detection register (GPIO_VDD_ACTIVE). The bit is set to '1' when there is supply and '0' when there is no supply

29.2.3.2.3 USB D+ pin pull-up enable logic

When a USB device is self-powered, the USB specification warrants that the device enable the pull-up resistor on its D+ pin to identify itself as a full-speed device to the host. When the host VBUS is removed, the device should disable the pull-up resistor on the D+ line to not back power the host. The USB PHY includes an internal 1.5-kΩ pull-up resistor on the D+ line to indicate to the host that the PSC3M/P6 MCU is a full-speed device. The pull-up resistor can be enabled or disabled by configuring the DP_UP_EN bit in the USBDEV_LPM_POWER_CTRL register.

29.2.3.2.4 Transmitter and receiver logic

The transceiver block transmits and receives USB differential signals with an upstream device, and includes the USB D+ pull-up resistor used to maintain an idle state on the bus. Output data is differentially transmitted to upstream devices at a nominal voltage of 3.3 V. The differential inputs received from upstream devices are converted into single-ended data and sent to the core logic at a nominal voltage of 1.8 V. The D+ and D– pins are terminated with 22-Ω resistors to meet the USB impedance specification.

29.2.3.2.5 GPIO mode logic

The D+ and D– pins can be used either as GPIO pins or USB I/O pins. This is controlled by the IOMODE bit of the USBDEV_USBIO_CR1 register. This bit should be set HIGH for GPIO functionality and LOW for USB operation.

29.2.3.2.6 Link power management (LPM)

The USB PHY supports link power management (LPM), which is similar to the suspend mode, but has transitional latencies in tens of microseconds between power states, compared to the greater than 20 ms latency associated with suspend/resume modes. For more details on LPM, see the USB 2.0 specification. The following features are supported for LPM:

- The LPM_CTL register should be configured to enable/disable LPM, type of response when LPM is enabled, and the response when a sub PID other than the LPM token is received from the host
- The LPM_STAT register stores the values of the Best Effort Service Latency (BESL) and the remote wakeup feature as sent by the host. The firmware should read this register on the LPM interrupt event and enter the appropriate low-power mode (Deep Sleep or Sleep) based on the BESL value from the host

29.2.3.3 Endpoints

The SIE and arbiter support eight data endpoints (EP1 to EP8) and one control endpoint (EP0). The data endpoints share the SRAM memory area of 512 bytes. The endpoint memory management can be either manual or automatic. The endpoints are configured for direction and other configuration using the SIE and arbiter registers. The endpoint read address and write address registers are accessed through the arbiter.

The endpoints can be individually made active. In the Auto Management mode, the register USBEV_EP_ACTIVE is written to control the active state of the endpoint. The endpoint activation cannot be dynamically changed during runtime. In Manual Memory Management mode, the firmware decides the memory allocation, so it is not required to specify the active endpoints. The USBDEV_EP_ACTIVE register is ignored during the manual memory management mode. The USBDEV_EP_TYPE register is used to control the transfer direction (IN, OUT) for the endpoints. The control endpoint has separate eight bytes for its data (USBDEV_EP0_DR registers).

29.2.3.4 Transfer types

The PSC3M/P6 MCU USB supports full-speed transfers and is compliant with the USB 2.0 specification. It supports four types of transfers:

- Interrupt Transfer
- Bulk Transfer
- Isochronous Transfer
- Control Transfer

For further details about these transfers, see the USB 2.0 specification.

29.2.3.5 Interrupt sources

The USB device block generates 14 interrupts to the CPU. These interrupts are mapped to three general-purpose interrupt lines – INTR_LO, INTR_MED, and INTR_HI. Each of these three interrupt lines has an associated status register, which identifies the cause of the interrupt event. These are the USBLPM_INTR_CAUSE_LO, USBLPM_INTR_CAUSE_MED, and USBLPM_INTR_CAUSE_HI registers. The routing of these interrupts is controlled by the USBLPM_INTR_LVL_SEL register fields. The following events generate an interrupt on one of the three interrupt lines:

- USB start of frame (SOF) event
- USB bus reset event
- Eight data endpoint (EP1 – EP8) interrupt events
- Control endpoint (EP0) interrupt event
- Link power management (LPM) event
- Resume event
- Arbiter Interrupt event

29.2.3.5.1 USB start of frame (SOF) event

The SOF interrupt is generated upon receiving an SOF packet from the USB host. The SOF interrupt is enabled using the SOF_INTR_MASK bitfield in the USBLPM_INTR_SIE_MASK register.

- The SOF interrupt status is reflected in the SOF_INTR status bit in the USBLPM_INTR_SIE status register
- The SOF interrupt status is also available in the SOF_INTR_MASKED bit of the USBLPM_INTR_SIE_MASKED register – this bit is the logical AND of the corresponding SOF bits in the USBLPM_INTR_SIE_MASK register and the USBLPM_INTR_SIE register
- If there is no SOF interrupt for 3 ms, the USB device goes into SUSPEND state

29.2.3.5.2 USB bus reset event

The USB bus reset interrupt is generated when a USB bus reset condition occurs. The bus reset interrupt is enabled by setting the BUS_RESET_INTR_MASK bit in the USBLPM_INTR_SIE_MASK register.

- The bus reset interrupt status is reflected in the BUS_RESET_INTR bit in the USBLPM_INTR_SIE status register
- The bus reset interrupt status is also available in the BUS_RESET_INTR_MASKED bit of the USBLPM_INTR_SIE_MASKED register – this bit is the logical AND of the corresponding bus reset bits in the USBLPM_INTR_SIE_MASK register and the USBLPM_INTR_SIE register
- The SIE logic triggers the counter to start running on the divided version of CLK_PERI when an SE0 condition is detected on the USB bus. When the counter reaches the count value configured in the

USBDEV_BUS_RST_CNT register, the bus reset interrupt is triggered. Typically, divided CLK_PERI is set to 100 kHz and USBDEV_BUS_RST_CNT is set to '10'

29.2.3.5.3 Data endpoint interrupt events

These are eight interrupt events corresponding to each data endpoint (EP1-EP8). Each of the endpoint interrupt events can be enabled/disabled by using the corresponding bit in the USBDEV_SIE_EP_INT_EN register. The interrupt status of each endpoint can be known by reading the USBDEV_SIE_EP_INT_SR status register. An endpoint whose interrupt is enabled can trigger the interrupt on the following events:

- Successful completion of an IN/OUT transfer
- NAK-ed IN/OUT transaction if the corresponding NAK_INT_EN bit in the USBDEV_SIE_EPx_CR0 register is set
- When there is an error in the transaction, the ERR_IN_TXN bit in the USBDEV_SIE_EPx_CR0 register is set and interrupt is generated
- If the STALL bit in USBDEV_SIE_EPx_CR0 is set, then stall events can generate interrupts. This stall event can occur if an OUT packet is received for an endpoint whose mode bits in USBDEV_SIE_EPx_CR0 are set to ACK_OUT or if an IN packet is received with mode bits set to ACK_IN

29.2.3.5.4 Control endpoint interrupt event

The interrupt event corresponding to the control endpoint (EP0) is generated under the following events:

- Successful completion of an IN/OUT transfer
- When a SETUP packet is received on the control endpoint

The EP0 interrupt is setup using the EP0_INTR_SET bit of the USBLPM_INTR_SIE_SET register.

29.2.3.5.5 Link power management (LPM) event

Generated whenever the LPM token packet is received. The LPM interrupt is enabled by setting the LPM_INTR_MASK bit in the USBLPM_INTR_SIE_MASK register. The LPM interrupt status is reflected in the LPM_INTR status bit in the USBLPM_INTR_SIE status register.

The LPM interrupt status is also available in the LPM_INTR_MASKED bit of the USBLPM_INTR_SIE_MASKED register; this bit is the logical AND of the corresponding LPM bits in the USBLPM_INTR_SIE_MASK and USBLPM_INTR_SIE registers.

The firmware needs to read the USBLPM_LPM_STAT register to read the BESL remote wakeup values and appropriately enter the desired low-power mode. Enter the low-power mode in the main code. The exit from LPM is identical to the resume event wakeup in the case of suspend mode.

29.2.3.5.6 RESUME interrupt

The RESUME interrupt is asserted by the USB block when it detects '0' on the DP pad. The RESUME interrupt is enabled by setting the RESUME_INTR_MASK bitfield of the USBLPM_INTR_SIE_MASK register.

- The RESUME interrupt status is reflected in the RESUME_INTR status bit in the USBLPM_INTR_SIE status register
- The RESUME interrupt status is also available in the RESUME_INTR_MASKED bit of the USBLPM_INTR_SIE_MASKED register – this bit is the logical AND of the corresponding RESUME bits in the USBLPM_INTR_SIE_MASK register and the USBLPM_INTR_SIE register

The RESUME interrupt is an Active mode interrupt and not available in Deep Sleep or Hibernate mode.

29.2.3.5.7 Arbiter interrupt event

The arbiter interrupt can arise from five possible sources. Each interrupt source is logically ANDed with its corresponding ENABLE bit and the results are logically ORed to result in a single arbiter interrupt event. The arbiter interrupt event can arise under any of the following five scenarios:

- DMA grant
- IN buffer full
- Buffer overflow
- Buffer underflow
- DMA Termin

DMA grant

This event is applicable in Mode 2 or Mode 3. (See Logical transfer modes section for details on DMA modes). This event is triggered when the DMA controller pulses the Burst-end signal corresponding to that endpoint, for which a DMA request had been raised to the DMA controller earlier. The request may have been either a manual DMA request or an automatic arbiter DMA request. A common grant status exists for both modes of requests. This grant status indicates completion of the DMA transaction. This status indication can be used by firmware to determine when the next manual DMA request can be raised. Multiple requests raised for the same endpoint before the DMA grant status is set will be dropped by the block. Only the first of multiple requests will be transmitted to DMA controller.

IN buffer full

This event status can occur in any of the DMA modes (Mode 1, 2, or 3) and is applicable only for IN endpoints.

- **Store and Forward Mode (Modes 1 and 2):** This status is set when the entire packet data is transferred to the local memory. The check is that data written for the particular endpoint is equal to the programmed byte count for that endpoint in the USBDEV_SIE_EPx_CNT0 and USBDEV_SIE_EPx_CNT1 registers
- **Cut Through Mode (Mode 3):** In this mode, the IN buffer full status is set when the IN endpoint's dedicated buffer is filled with the packet data. The size of this buffer is determined by the value programmed in bits [3:0] of the USBDEV_BUF_SIZE register. This status indication can be used to determine when the mode value in the USBDEV_SIE_EPx_CR0 register can be programmed to acknowledge an IN token for that endpoint

Buffer overflow

This event status is active only in the Cut Through Mode (Mode 3). The following conditions can cause this bit to be set:

- Data overflow on the endpoint dedicated buffer space
  - In an IN endpoint, the dedicated buffer can overflow if the DMA transfer writes a larger number of bytes than the space available in the dedicated buffer. Until an IN token is received for that endpoint, it cannot use the common buffer area, hence resulting in an overflow of data. The possible causes of this buffer overflow can be incorrect programming of either the DMA transfer descriptor transfer size or the USBDEV_BUF_SIZE register
  - In an OUT endpoint, the dedicated buffer can overflow if two OUT transactions occur consecutively. The data from the previous transaction is still present in the common area and the current ongoing transaction fills up the OUT endpoint's dedicated buffer space and overflows. The possible causes of this overflow can be the overall DMA bandwidth constraint due to other DMA transactions or reduced size of the dedicated OUT buffer size
- Common area data overflow
  - In an IN endpoint, the common area overflow occurs when the DMA transfer writes a larger number of bytes than the space available in the common area. This situation may arise due to incorrect

programming of either the DMA transfer descriptor transfer size or the USBDEV_DMA_THRESH and USBDEV_DMA_THRESH_MSB registers
- In an OUT endpoint, the common area overflow occurs when the data written to the common area has not yet been read and new data overwrites the existing data

Buffer underflow

This event is applicable only in the Cut Through mode (Mode 3). This underflow condition can occur only for an IN endpoint. The underflow condition can occur either in the dedicated buffer space or common buffer space. The underflow condition on the dedicated buffer space can either be due to the reduced dedicated IN buffer size or DMA bandwidth constraint. The underflow condition can occur on the common buffer space due to DMA bandwidth constraint and/or lower DMA channel priority.

DMA Termin

This status is set when USBDEV generates a dma_termin signal to indicate the total programmed/received bytes that are written/read by the DMA controller. This status indication can be used by the firmware to reprogram the IN/OUT endpoint for the next transfer. For an OUT endpoint, this indicates that the OUT packet data is available in the system memory for further processing by the application.

29.2.3.6 DMA support

Each of the eight data endpoints has one DMA channel available to transfer data between the endpoint buffer and the SRAM memory. The USB generates the DMA request signals (usb.dma_req[7:0]) to the respective DMA channels to initiate the data transfer for the endpoint. This goes to the Trigger Group 0/1 multiplexer as input triggers that can be routed to one of the trigger inputs of the DMA block. The Burstend signals from the DMA channel to the corresponding endpoint is routed using the Trigger Group 14 multiplexer. For more details, see the Direct memory access (DMA - DW) and Trigger multiplexer (Trigger MUX) chapter.

29.2.4 Logical transfer modes

The USB block in PSC3M/P6 MCUs supports two types of logical transfers. The logical transfers can be configured using the register setting for each endpoint. Any of the logical transfer methods can be adapted to support the three types of data transfers (Interrupt, Bulk, and Isochronous) mentioned in the USB 2.0 specification. The control transfer is mandatory in any USB device.

The logical transfer mode is a combination of memory management and DMA configurations. The logical transfer modes are related to the data transfer within the USB (to and from the SRAM memory unit for each endpoint). It does not represent the transfer methods between the device and the host (the transfer types specified in the USB 2.0 specification).

The USB supports two basic types of transfer modes:

- Store and Forward mode
  - Manual Memory Management with No DMA Access (Mode 1)
  - Manual Memory Management with Manual DMA Access (Mode 2)
- Cut Through mode
  - Automatic Memory Management with Automatic DMA Access (Mode 3)

Table 313 gives a comparison of the two transfer modes.

Table 313 USB transfer modes

Feature	Store and Forward mode	Cut Through mode
SRAM Memory Usage	Requires more memory	Requires less memory

(table continues...)

Table 313 (continued) USB transfer modes

Feature	Store and Forward mode	Cut Through mode
SRAM Memory Management	Manual	Auto
SRAM Memory Sharing	512 bytes of SRAM shared between endpoints. Sharing is done by firmware	Each endpoint is allocated a lesser share of memory automatically by the block. The remaining memory is available as “common area.” This common area is used during the transfer
IN Command	IN package will be NAKed until the entire packet is present in SRAM memory	Memory filled with data only when SRAM IN command is received. Data is given to host when enough data is available (based on DMA configuration). Does not wait for the entire data to be filled
OUT Command	Entire packet is written to SRAM memory on OUT command. After entire data is available, it should be ready by CPU or DMA via the data register from SRAM memory	Waits only for enough bytes (depends on DMA configuration) to be written in SRAM memory. When enough bytes are present, it is immediately copied from SRAM memory to the USB device
Transfer of Data	Data is transferred when all bytes are written to the memory	Data is transferred when enough bytes are available. It does not wait for the entire data to be filled
Types Based on DMA	No DMA mode or manual DMA mode	Only Auto DMA mode
Supported Transfer Types	Ideal for interrupt and bulk transfers	Ideal for Isochronous transfer

Every endpoint has a set of registers that need to be handled during the modes of operation, as detailed in Table 314.

Table 314 Endpoint registers

Register	Comment	Content	Usage
USBDEV_ARB_RWx_WA	Endpoint Write Address Register	Address of the SRAM	This register indicates the SRAM location to which the data in the data register is to be written
USBDEV_ARB_RWx_RA	Endpoint Read Address Register	Address of the SRAM	This register indicates the SRAM location from which the data must be read and stored to the data register

(table continues...)

Table 314 (continued) Endpoint registers

Register	Comment	Content	Usage
USBDEV_ARB_RWx_DR	Endpoint Data Register	8-Bit Data	Data register is read/ written to perform any transaction IN command: Data written to the data register is copied to the SRAM location specified by the WA register. After write, the WA value is automatically incremented to point to the next memory location OUT command: Data available in the SRAM location pointed by the USBDEV_ARB_RWx_RA register is read and stored to the DR. When the DR is read, the value of USBDEV_ARB_RWx_RA is automatically incremented to point to the next SRAM memory location that must be read
USBDEV_SIE_EPx_CNT0 and USBDEV_SIE_EPx_CNT1	Endpoint Byte Count Register	Number of Bytes	Holds the number of bytes that can be transferred IN command: Holds the number of bytes to be transferred to host OUT command: Holds the maximum number of bytes that can be received. The firmware programs the maximum number of bytes that can be received for that endpoint. The SIE updates the register with the number of bytes received for the endpoint
“Mode” bits in USBDEV_SIE_EPx_CR0	Mode Values	Response to the Host	Controls how the USB device responds to the USB traffic and the USB host. Some examples of modes are ACK, NAK, and STALL

In Manual memory management, the endpoint read and endpoint write address registers are updated by the firmware. So the memory allocation can be done by the user. The memory allocation decides which endpoints are active; that is, you can decide to share the 512 bytes for all the eight endpoints or a lesser number of endpoints.

In Automatic memory management, the endpoint read and endpoint write address registers are updated by the USB block. The block assigns memory to the endpoints that are activated using the USBDEV_EP_ACTIVE register. The size of memory allocated depends on the value in the USBDEV_BUF_SIZE register. The remaining memory, after allocation, is called the common area memory and is used for data transfer.

In all of these modes, either the 8-bit endpoint data register or the 16-bit endpoint data register can be used to read/write to the endpoint buffer. While transferring data to the 16-bit data registers, ensure that the corresponding SRAM memory address locations are also 16-bit aligned.

In the following text, the algorithm for the IN and OUT transaction for each mode is discussed. An IN transaction is when the data is read by the USB host (for example, PC). An OUT transaction is when the data is written by the USB host to the USB device. The choice of using the DMA and memory management can be configured using the USBDEV_ARB_EPx_CFG register.

29.2.4.1 Manual memory management with no DMA access

All operations in this mode are controlled by the CPU and works in a store-and-forward operation mode. An entire packet is transferred to the memory and a mode bit (such as ACK IN or ACK OUT) is set by the CPU. The SIE responds appropriately to an IN/OUT token received from the host. All memory space management is handled by the CPU.

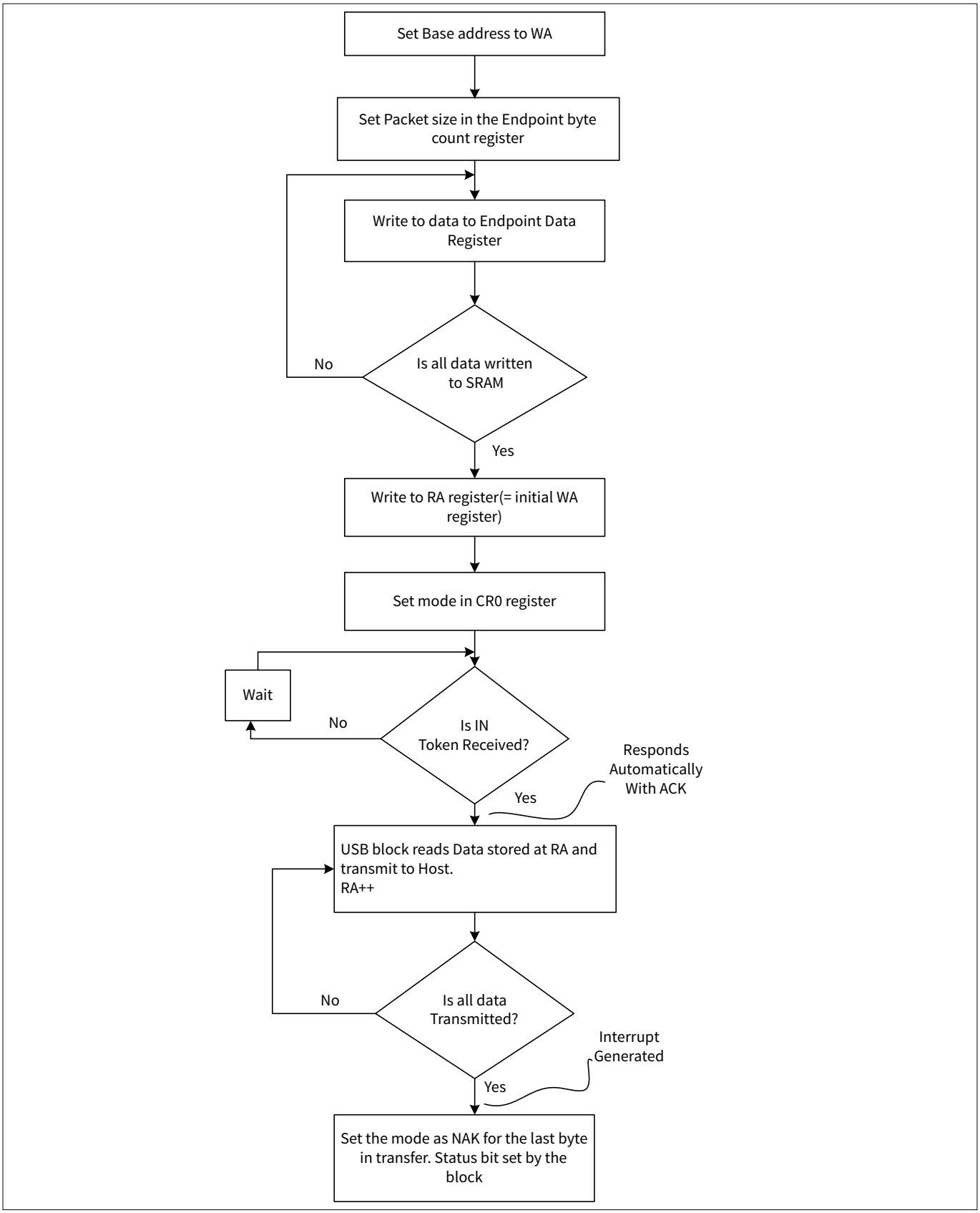


Figure 351 No DMA access IN transaction

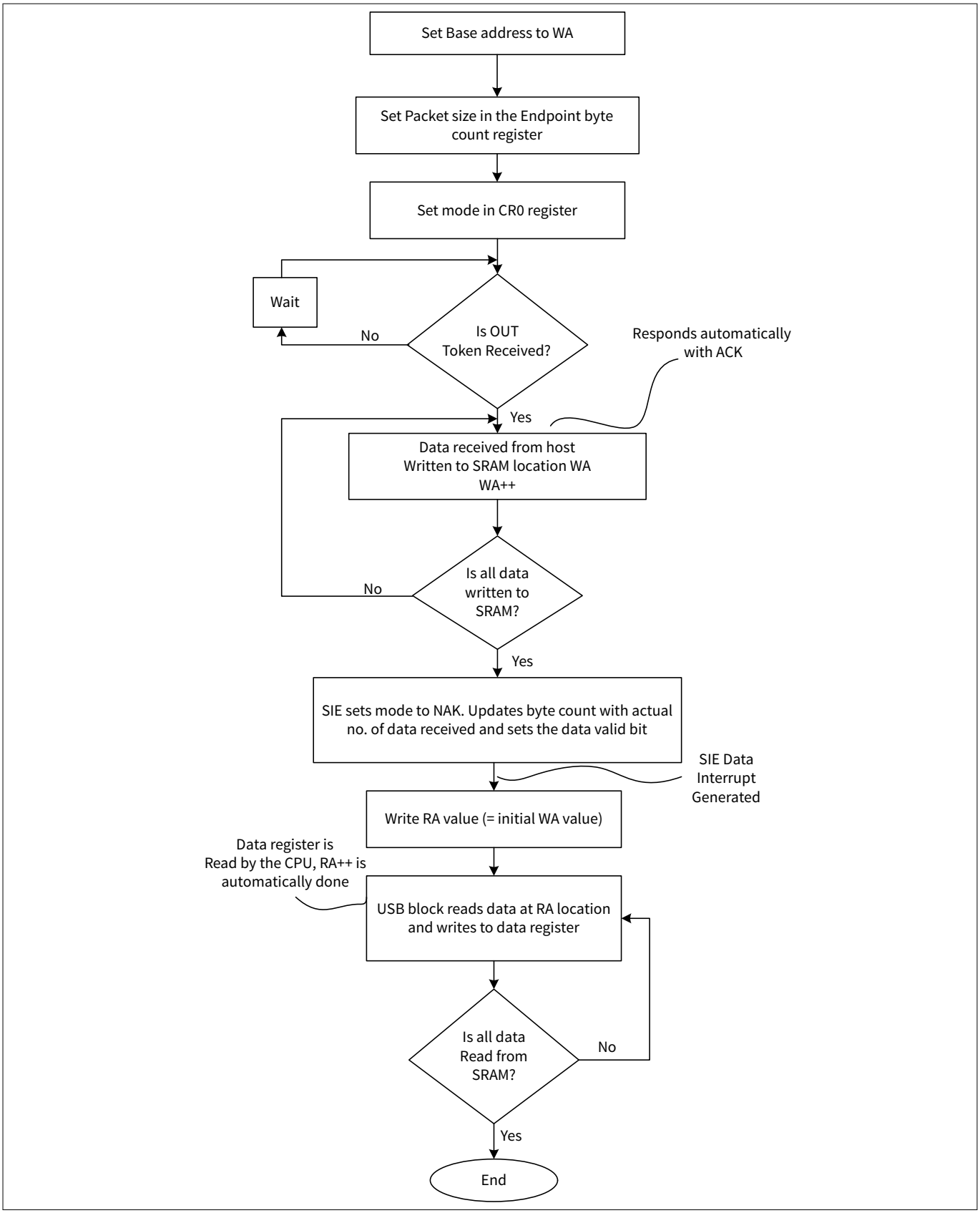


Figure 352 No DMA access OUT transaction

29.2.4.2 Manual memory management with DMA access

This mode is similar to the No DMA Access except that write/read of packets is performed by the DMA. A DMA request for an endpoint is generated by setting the DMA_REQ bit in the USBDEV_ARB_EPx_CFG register. When the DMA service is granted and is done (DMA_GNT), an arbiter interrupt can be programmed to occur. The transfer is done using a single DMA cycle or multiple DMA cycles. After completion of every DMA cycle, the arbiter interrupt (DMA_GNT) is generated. Similarly, when all the data bytes (programmed in byte count) are written to the memory, the arbiter interrupt occurs and the IN_BUF_FULL bit is set.

Figure 353 and Figure 354 show the flow charts for manual DMA IN and OUT transactions respectively.

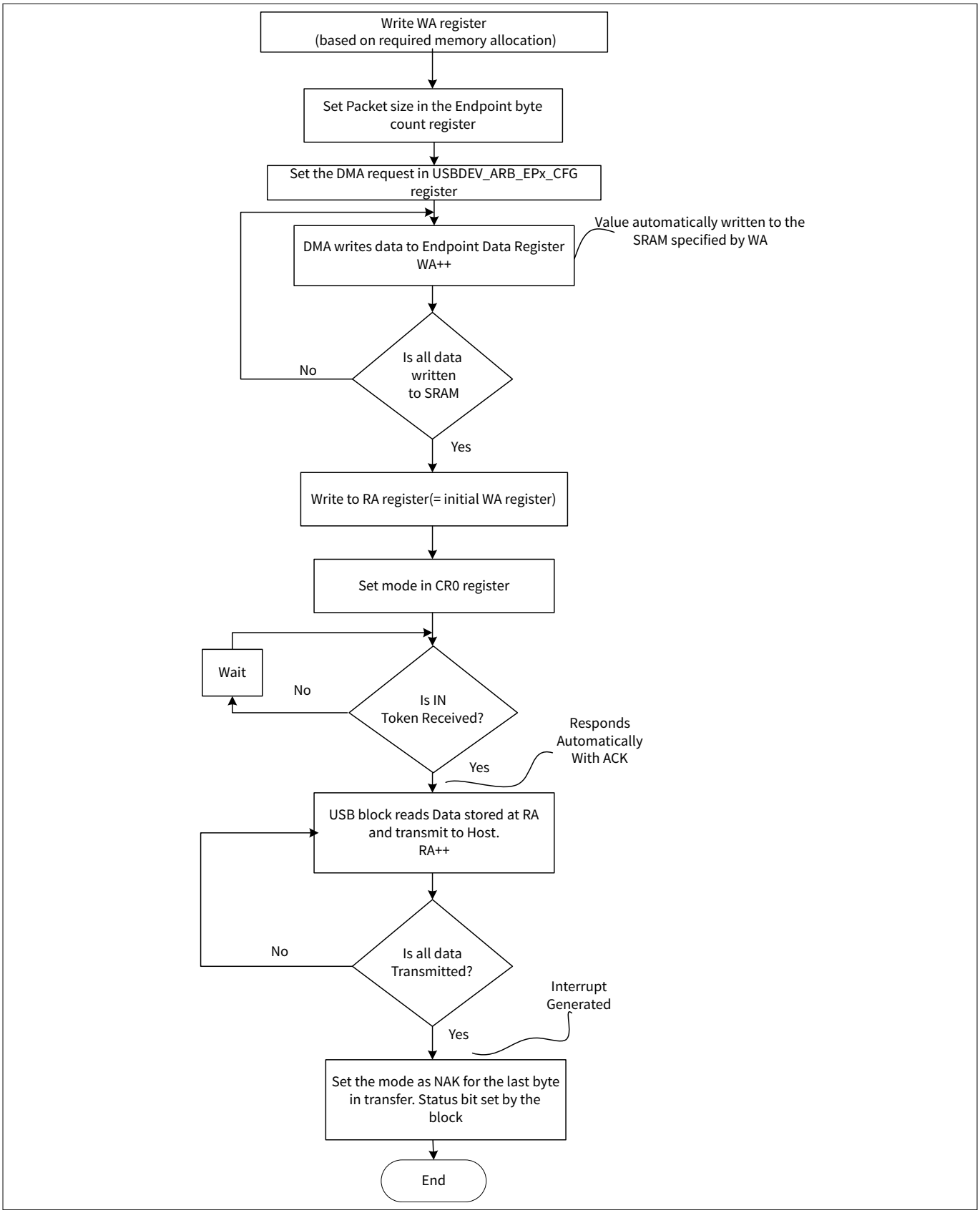


Figure 353 Manual DMA IN transaction

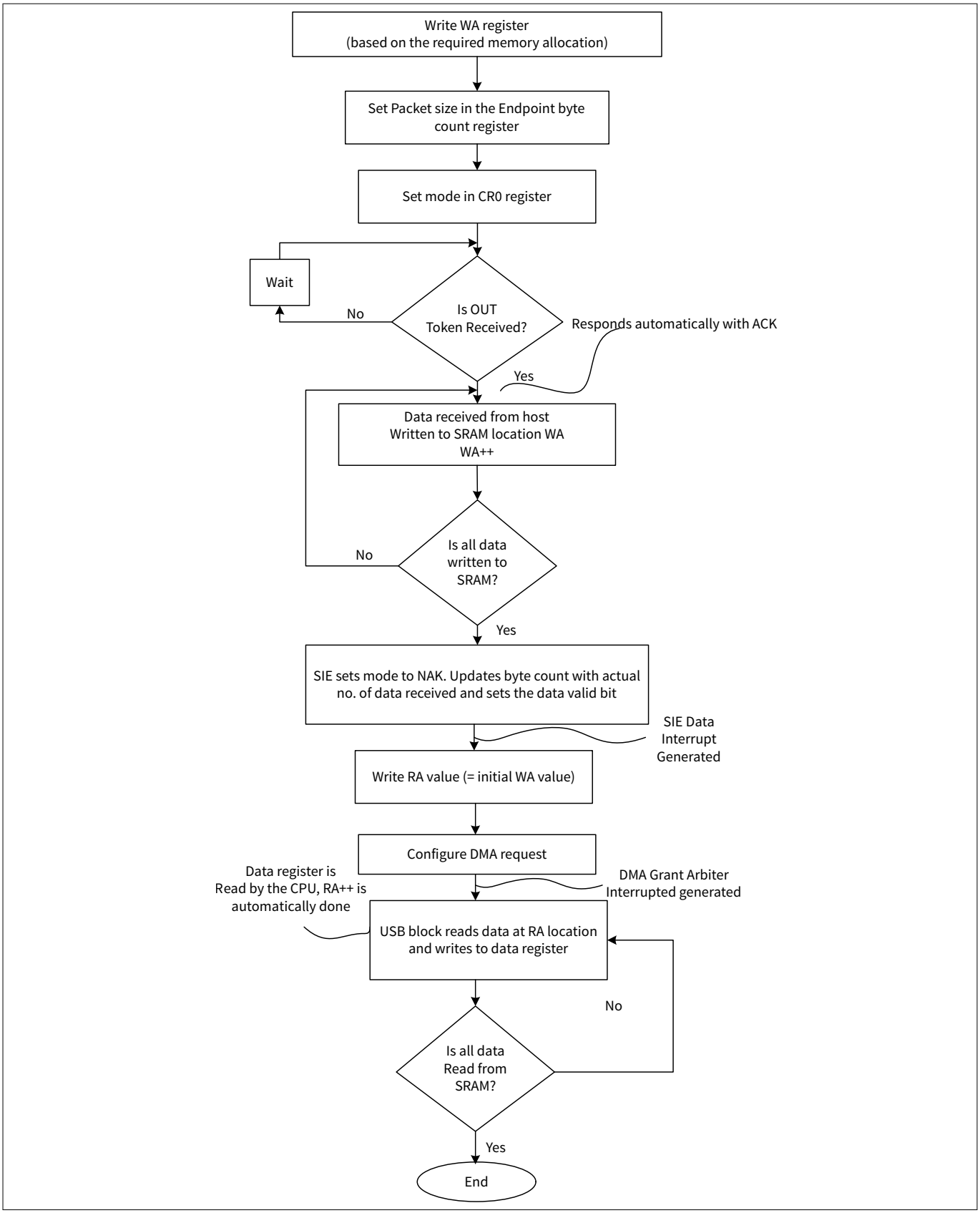


Figure 354 Manual DMA OUT transaction

29.2.4.3 Automatic DMA mode

This is the Automatic memory management mode with auto DMA access. The CPU programs the initial buffer size requirement for IN/OUT packets and informs the arbiter block of the endpoint configuration details for the particular application. The block then controls memory partitioning and handling of all memory pointers. During memory allocation, each active IN endpoint (set by the USBDEV_EP_ACTIVE and USBDEV_EP_TYPE registers) is allocated a small amount of memory configured using the USBDEV_BUF_SIZE register (32 bytes for each of the eight endpoints). The remaining memory (256 bytes) is left as common area and is common for all endpoints.

In this mode, the memory requirement is less and it is suitable for full-speed isochronous transfers up to 1023 bytes.

When an IN command is sent by the host, the device responds with the data present in the dedicated memory area for that endpoint. It simultaneously issues a DMA request for more data for that EP. This data fills up the common area. The device does not wait for the entire packet of data to be available. It waits only for the (USBDEV_DMA_THRES_MSB, USBDEV_DMA_THRES) number of data available in the SRAM memory and begins the transfer from the common area.

Similarly, when an OUT command is received, the data for the OUT endpoint is written to the common area. When some data (greater than USBDEV_DMA_THRES_MSB, USBDEV_DMA_THRES) is available in the common area, the arbiter block initiates a DMA request and the data is immediately written to the device. The device does not wait for the common area to be filled.

This mode requires configuration of the USBDEV_DMA_THRES and USBDEV_DMA_THRES_MSB registers to hold the number of bytes that can be transferred in one DMA transfer (32 bytes). Similarly, the burst count of the DMA should always be equal to the value set in the USBDEV_DMA_THRES registers. Apart from the DMA configuration, this mode also needs the configuration of the USBDEV_BUF_SIZE for the IN and the OUT buffers and the USBDEV_EP_ACTIVE and the USBDEV_EP_TYPE registers.

Each DMA channel has two descriptors and both of them are used in this mode. Each descriptor is considered as a data chunk of 32 bytes and it executes according to the trigger mechanism. The descriptors are chained and hence 64 bytes can be transferred without firmware interaction. When both descriptors complete the endpoint DMA done interrupt and the DMA error interrupt triggers (due to the lack of data to transfer). The descriptors are updated to advance the source SRAM (IN endpoint) or destination SRAM (OUT endpoint) pointer locations and then enabled again. This sequence continues until all data is transferred.

The steps for IN and OUT transactions using automatic DMA mode are shown in Figure 355 and Figure 356.

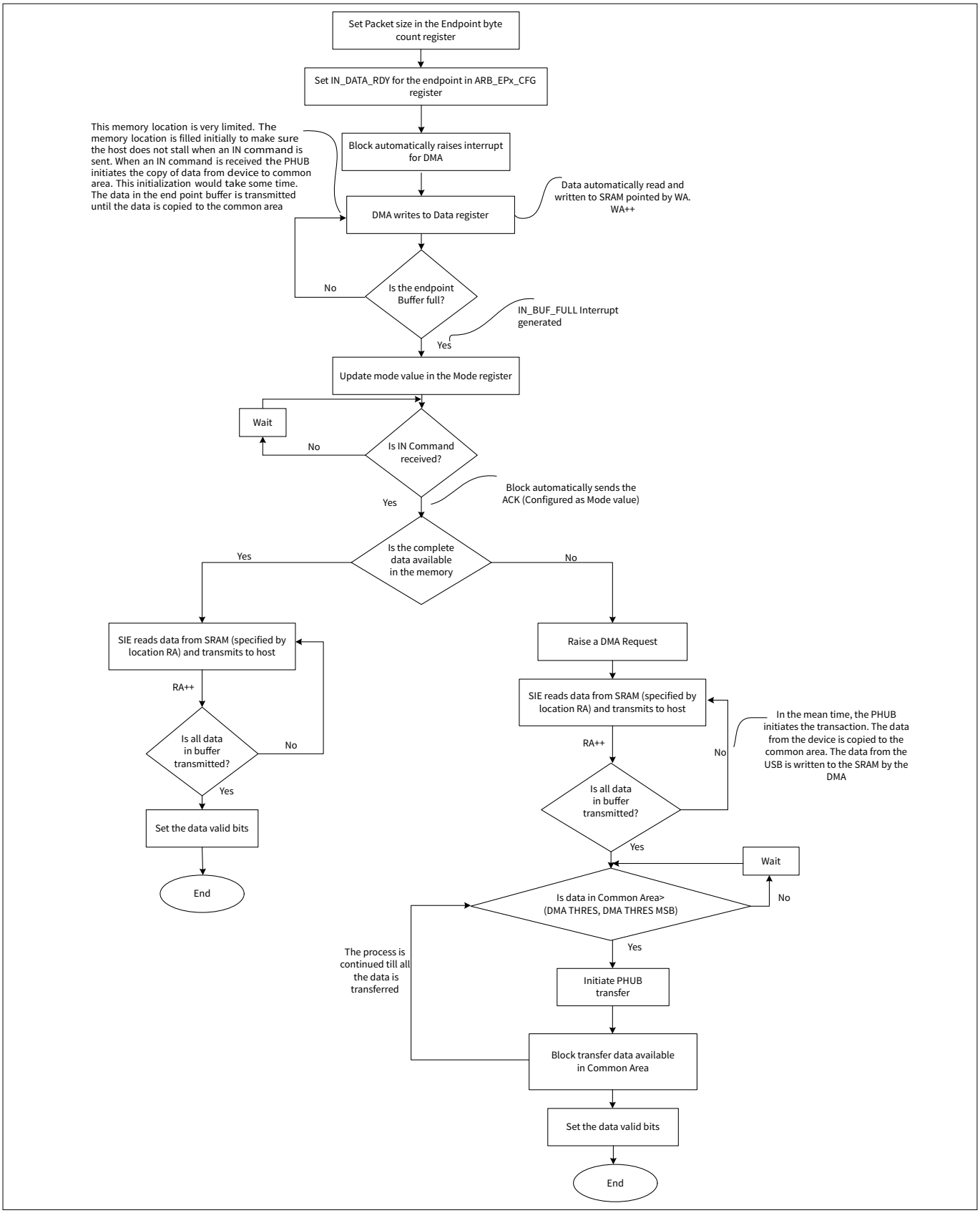


Figure 355 IN Transaction using automatic DMA mode

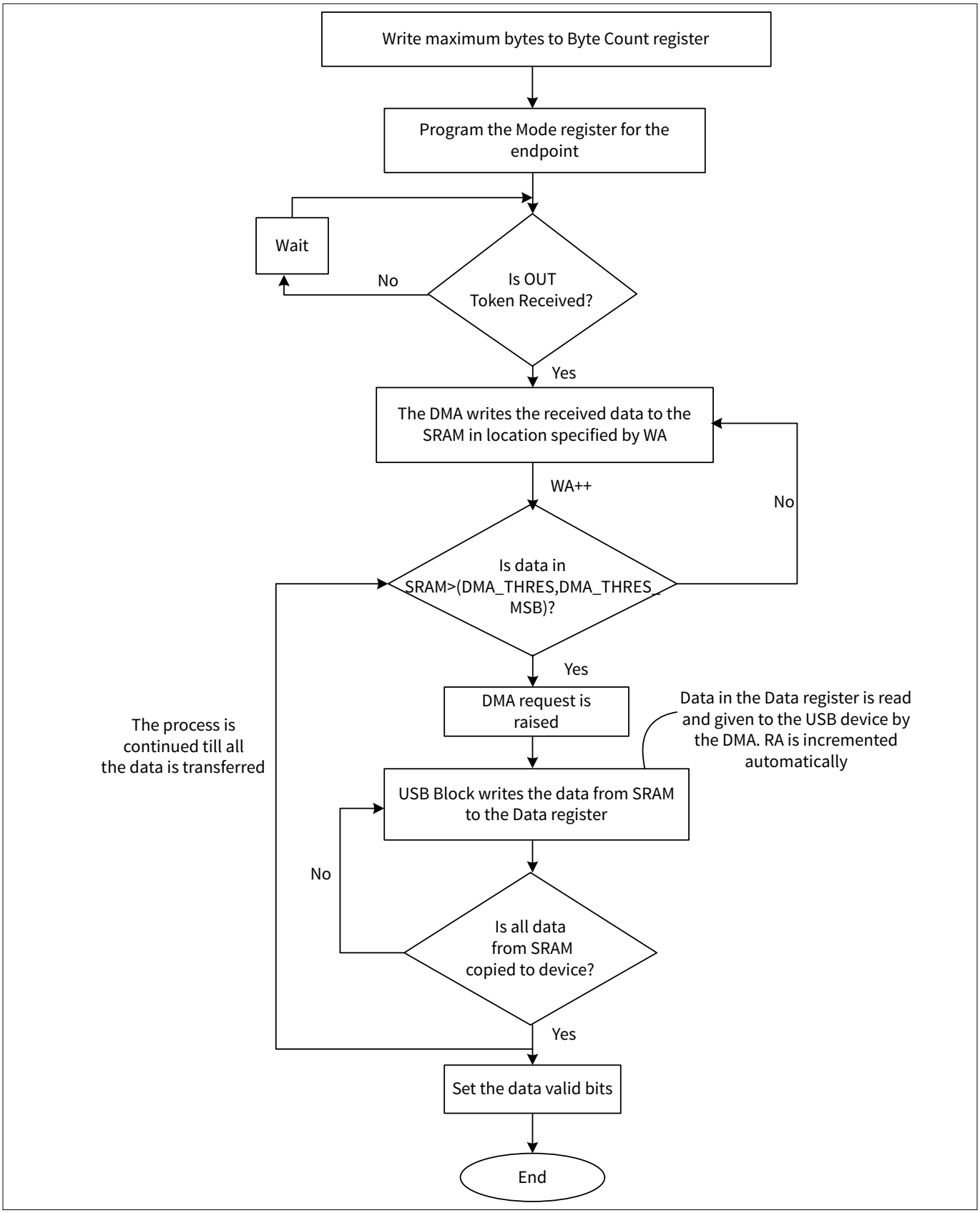


Figure 356 OUT Transaction using automatic DMA mode

29.2.4.4 Control endpoint logical transfer

The control endpoint has a special logical transfer mode. It does not share the 512 bytes of memory. Instead, it has a dedicated 8-byte register buffer (USBDEV_EP0_DRx registers). The IN and OUT transaction for the control endpoint is detailed in the following figures.

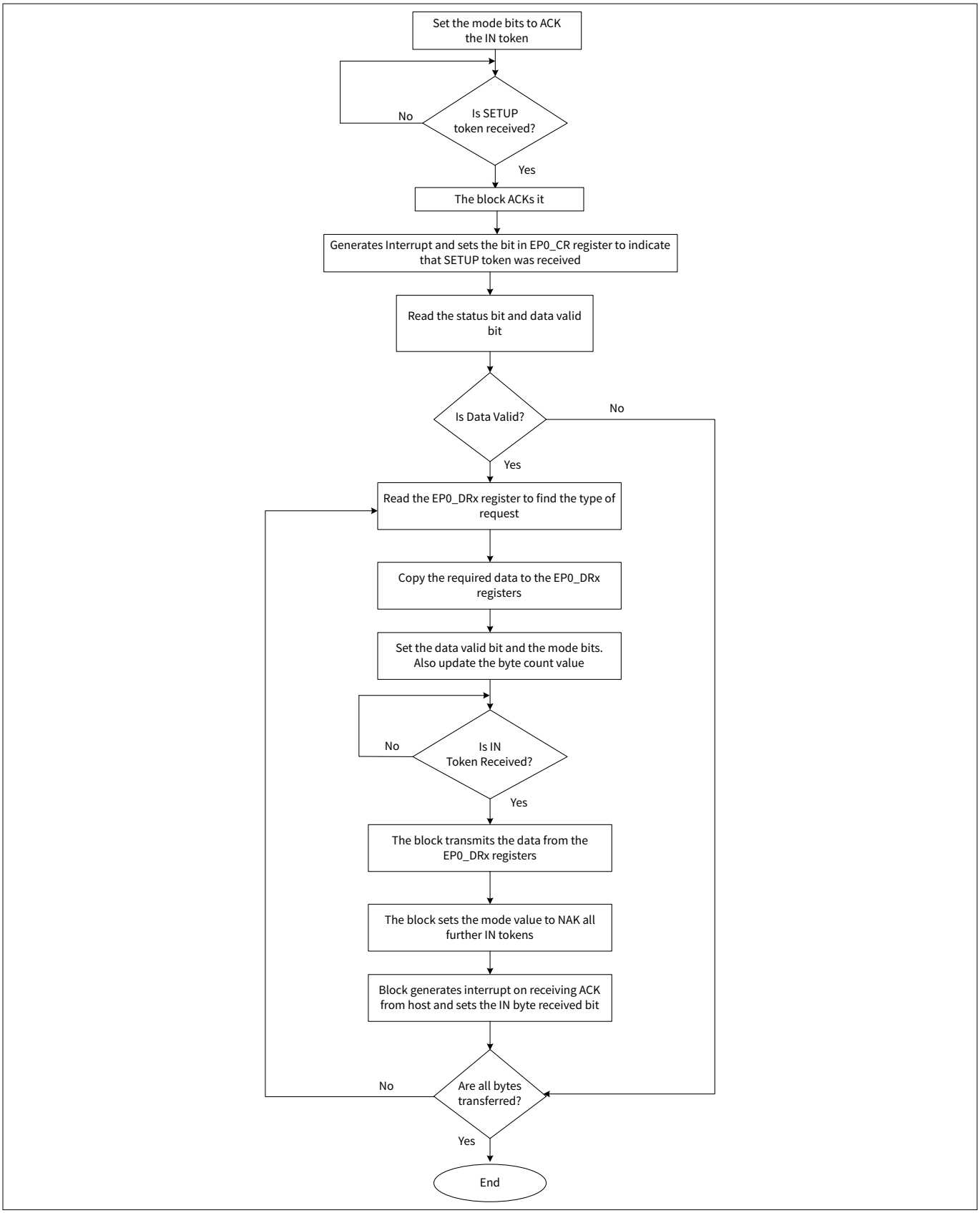


Figure 357 Control endpoint IN transaction

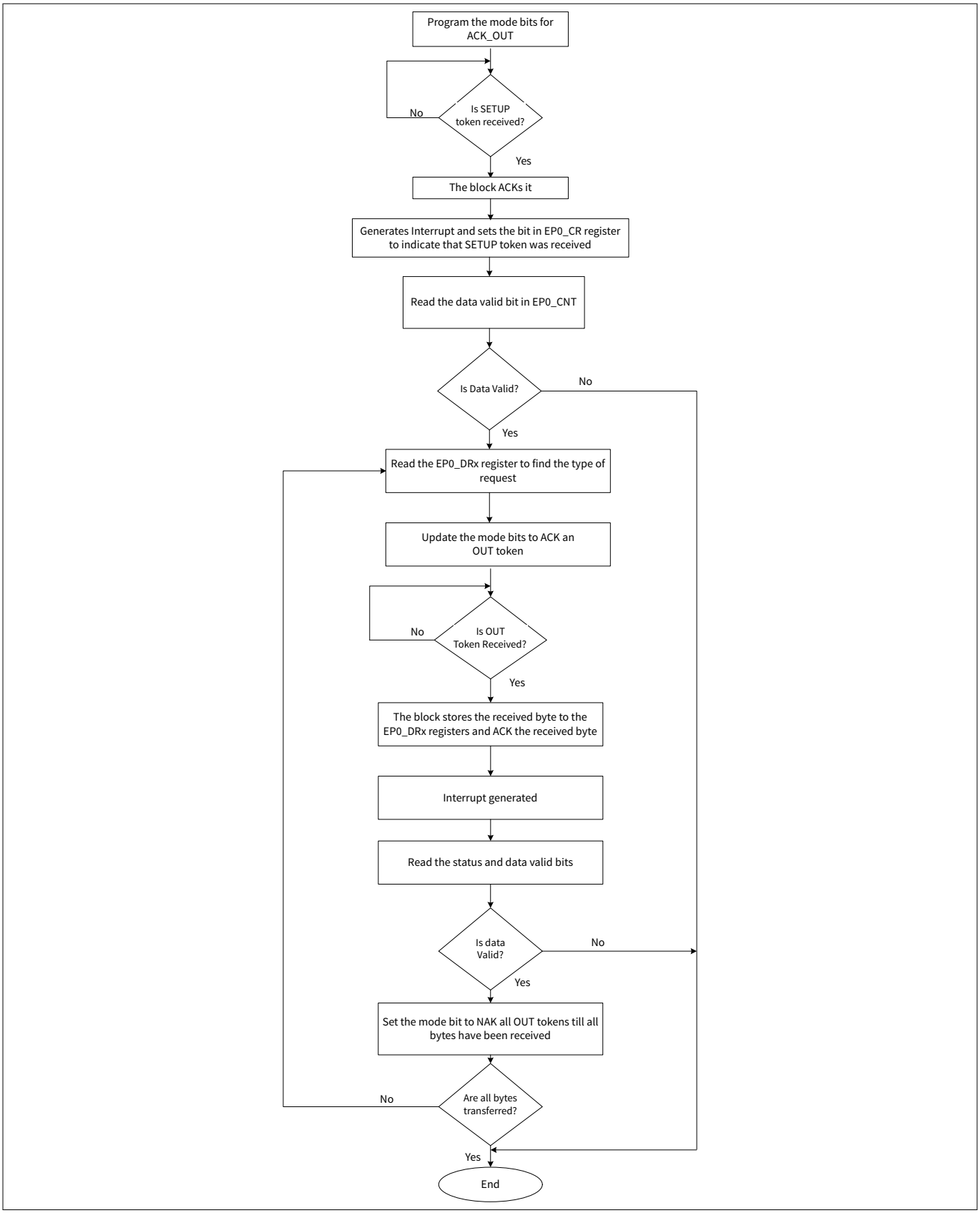


Figure 358 Control endpoint OUT transaction

29.2.5 USB power modes

The USB supports two modes of operation:

- Active mode: In this mode, the USB is powered up and clocks are turned on
- Low-power (Deep Sleep) mode: In this mode, all clocks except the low-frequency clock are turned off

Before entering low-power mode, the firmware should enable the GPIO interrupt (falling edge interrupt) on the D+ pin. The USB suspend mode can be determined by monitoring the SOF interrupt; this means, if there is no SOF interrupt from the USB for more than 3 ms, the USB goes into suspend mode and the block can be put into low-power mode by the firmware.

If there is any activity on the USB bus, D+ will be pulled low, which will cause a CPU interrupt. This interrupt can be used to wake up the USB device.

29.3 USB host mode

The USB subsystem in PSC3M/P6 MCUs can be configured to function as a USB host. The USB host in the PSC3M/P6 MCU supports both full-speed (12 Mbps) and low-speed (1.5 Mbps) devices and is designed to be complaint with the USB Specification Revision 2.0. This chapter details the PSC3M/P6 MCU USB and its operations. For details about the USB specification, see the USB implementers forum website.

29.3.1 Features

The USB host in the PSC3M/P6 MCU has the following features:

- Automatic detection of device connection or disconnection
- Automatic detection of full-speed or low-speed transfer
- Supports USB bus reset function
- Supports IN, OUT, SETUP. and SOF tokens
- Supports Bulk, Control, Interrupt, and Isochronous transfer
- Automatic detection of handshake packet for OUT token and automatic sending of handshake packet for IN token (excluding STALL)
- Supports a maximum packet length of up to 256 bytes
- Supports action against errors (CRC error, toggle error, and timeout)

29.3.2 Architecture

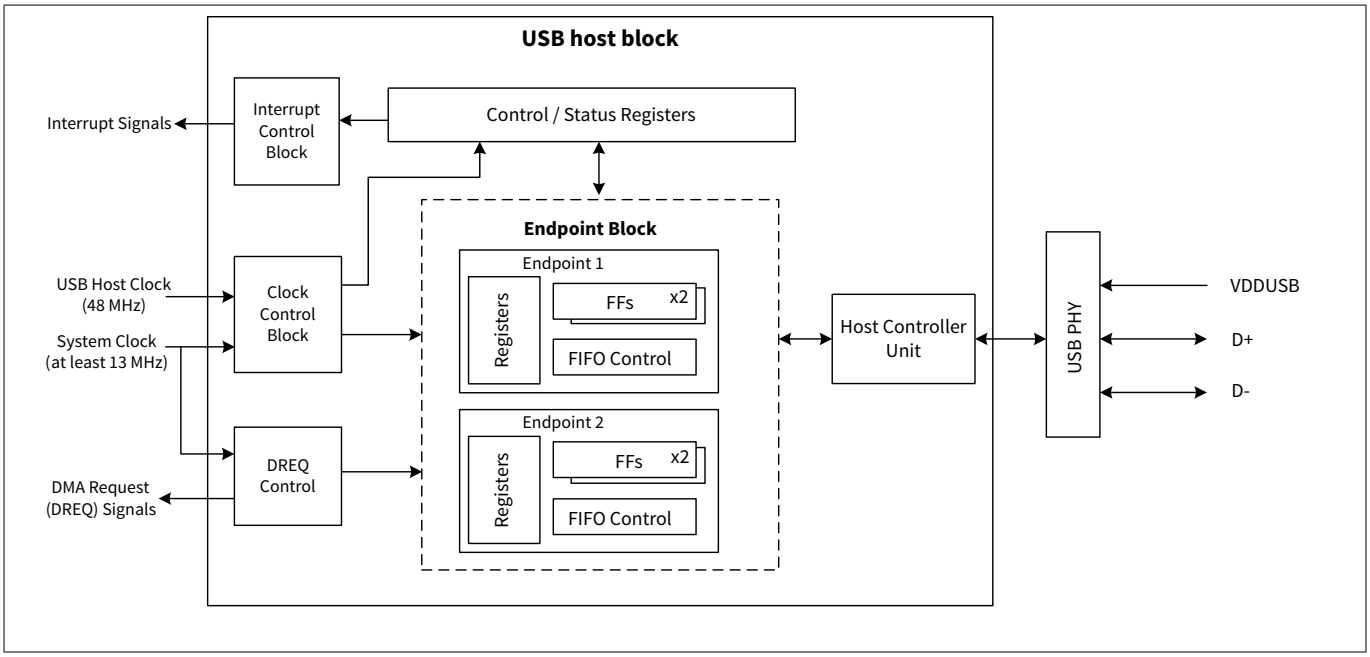


Figure 359 USB host block diagram

29.3.2.1 USB physical layer (USB PHY)

The USB includes the transmitter and receiver (transceiver), which corresponds to the USB PHY. This module allows physical layer communication with the USB device through the D+, D–, and VDDUSB pins. It handles differential mode communication with the device. The USB PHY also includes pull-down resistors on the D+ and D– lines. Differential signaling is used between the USB host and the device. The host controller unit receives the differential signal from the device and converts it to a single-ended signal. While transmitting, the host

controller unit converts the single-ended signal to the differential signal, and transmits it to the device. The differential signal is at a nominal voltage range of 0 V to 3.3 V.

29.3.2.2 Clock control block

This block controls the gated clocks – system clock and USB clock. The USB host requires a clock frequency of 48 MHz. USB host compliance requires a clock source with an accuracy of ±0.25 percent. One way of doing this is to use Clk_HF[3] sourced from an highly accurate ECO. The PLL can be used to generate the required 48-MHz clock from the ECO. See the Clocking system for details on generation of clock required for USB operation.

29.3.2.3 Interrupt control block

This block controls the interrupts associated with USB host operations. The USB host block has three general-purpose interrupt signals: USBHOST_INTR_USBHOST_LO, USBHOST_INTR_USBHOST_MED, and USBHOST_INTR_USBHOST_HI. There are 11 interrupt trigger events associated with USB host operation that can be mapped to any of the three interrupt lines. Each of these three interrupt lines has a status register to identify the interrupt source. These are the USBHOST_INTR_USBHOST_CAUSE_LO, USBHOST_INTR_USBHOST_CAUSE_MED, and USBHOST_INTR_USBHOST_CAUSE_HI, USBHOST_INTR_USBHOST_EP_CAUSE_LO, USBHOST_INTR_USBHOST_EP_CAUSE_MED and USBHOST_INTR_USBHOST_EP_CAUSE_HI registers.

29.3.2.4 Endpoint n (n=1, 2)

The USB host has two endpoints. The maximum buffer size of endpoint 1 is 256 bytes and that of endpoint 2 is 64 bytes. The endpoint buffers are used to send and receive data. If endpoint 1 is used as the transmitter, endpoint 2 must be used as the receiver and vice-versa. The DIR bit of the USBHOST_HOST_EPn_CTL (n=1, 2) register is used to configure the endpoint as an IN buffer (DIR=0) or OUT buffer (DIR=1).

29.3.2.5 DMA request (DREQ) control

The USB host supports data transfer using DMA. DMA operation is enabled by configuring the USBHOST_HOST_DMA_ENBL register. There are two DMA transfer modes: Automatic data transfer mode and packet transfer mode. The DMAE bit of the USBHOST_HOST_EPn_CTL (n=1, 2) register is used to set the mode of DMA transfer. The Host Endpoint Block register (USBHOST_HOST_EPn_BLK; n=1, 2) sets the total number of bytes for DMA transfer.

29.3.3 USB host operations

To operate the USB as a host, the following settings are required:

- Enable the pull-down resistors on both D+ and D– pins. To enable the pull-down resistors, set the DP_DOWN_EN bit and DM_DOWN_EN bit of the power control register (USBLPM_POWER_CTL) to 1
- Set both the HOST and ENABLE bits of the Host Control 0 register (USBHOST_HOST_CTL0) to 1
- Set the USTP bit of the Host Control 1 register (USBHOST_HOST_CTL1) to 0
- Supply the operating clock for USB (48 MHz ± 0.25%). In the PSC3M/P6 MCU, CLK_HF3 is the clock source. The required USB clock can be generated using one of the following clocking schemes:
  - ECO (with the required accuracy) -> PLL -> CLK_HF3
  - Use an external clock (EXTCLK) with the required accuracy

29.3.3.1 Detecting device connection

When an external USB device is not connected, both the D+ and D– pins are set to logic LOW by the pull-down resistors. In the unconnected state, the CSTAT bit of the USBHOST_HOST_STATUS register is '0' and the TMODE bit is undefined. If an external USB device is connected, then the CSTAT bit is set to '1'.

When a device is detected, the CNNIRQ bit of the Interrupt USB host (USBHOST_INTR_USBHOST) register is set to '1'. A device can be detected either by the method of interrupt or by polling. If '1' is set to the CNNIRQM bit of the Interrupt USB host (USBHOST_INTR_USBHOST_MASK) register, a device connection interrupt occurs. To clear this interrupt, write '1' to the CNNIRQ bit of USBHOST_INTR_USBHOST register. When detecting a device connection by polling, set the CNNIRQM bit of the USBHOST_INTR_USBHOST_MASK register to '0' and check when the CNNIRQ bit of USBHOST_INTR_USBHOST changes to '1'.

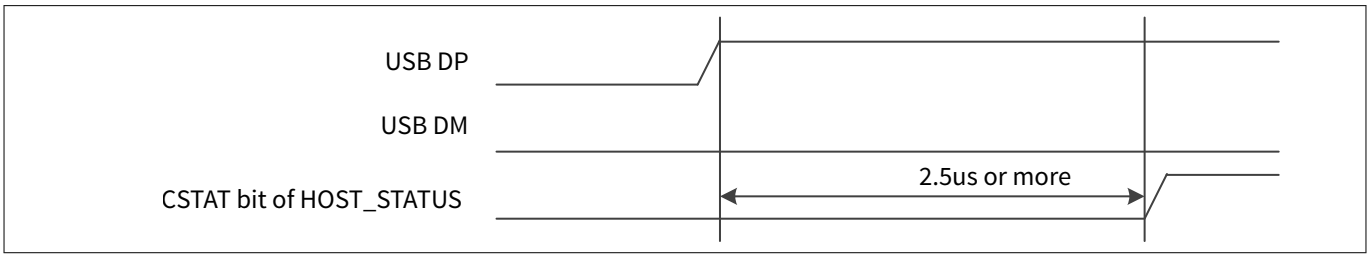


Figure 360 Device connection

***Note**: If the ENABLE bit of the Host Control 0 Register (USBHOST_HOST_CTL0) is '0', or if the RST bit or USTP bit of the Host Control 1 Register (USBHOST_HOST_CTL1) is '1', the TMODE and CSTAT bits of the Host Status Register (USBHOST_HOST_STATUS) cannot be detected.*

29.3.3.2 Obtaining transfer speed of the USB device

To obtain the possible transfer speed of the remote USB device after detecting a connection, check the value of the TMODE bit of the USBHOST_HOST_STATUS register. The following shows the relationships between the transfer speed and the value of the TMODE bit of USBHOST_HOST_STATUS.

- If TMODE=1, then the device is a full-speed device
- If TMODE=0, then the device is a low-speed device

Figure 361 shows the flow chart for device connection detection and obtaining the transfer speed.

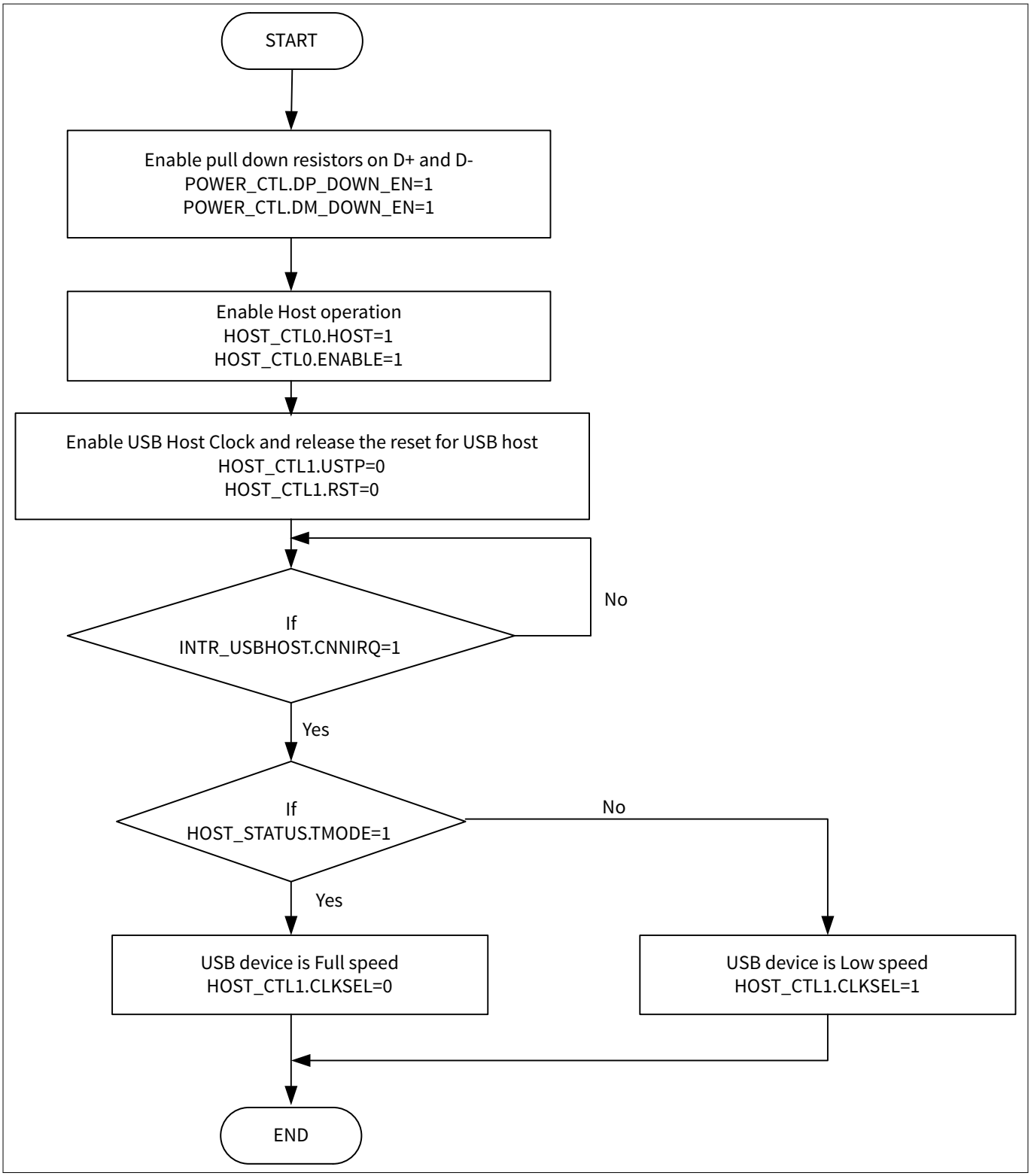


Figure 361 Device connection and transfer speed detection flow chart

29.3.3.3 USB bus reset

The USB bus is reset by sending SE0 for 10 ms or more if the URST bit of the USBHOST_HOST_STATUS register is set to '1'. After the USB bus is reset, the URST bit of HOST_STATUS is set to '0', and the URIRQ bit of USBHOST_INTR_USBHOST is set to '1'. If the URIRQM bit of the USBHOST_INTR_USBHOST_MASK register is

then set to '1', an interrupt occurs. To clear this interrupt, write '1' to the URIRQ bit of USBHOST_INTR_USBHOST.

***Note**: Verify the following before and after resetting the USB bus:*

- *To verify that a device is connected before resetting the USB bus, ensure that the CSTAT bit of the Host Status Register (USBHOST_HOST_STATUS) is set to '1'*
- *When the USB bus is reset, the CSTAT bit of the Host Status Register is changed to '0', and USB Host indicates a disconnection. At this time, the DIRQ bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST) is not set to '1'*
- *After the USB bus resetting has been completed, compare the value of the CLKSEL bit with that of the TMODE bit in the Host Status Register (USBHOST_HOST_STATUS). If the values do not match, update the CLKSEL bit so that they are aligned. Update the CLKSEL bit while the RST bit of the Host Control 1 Register (USBHOST_HOST_CTL1) is '1'*
- *After the USB bus resetting has been completed, verify that the USB device is connected by checking one of the status bits listed below. Once the connection is confirmed, proceed with token processing*
  - *CNNIRQ bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST)*
  - *CSTAT bit of the Host Status Register (USBHOST_HOST_STATUS)*

Figure 362 shows the timing diagram for USB bus reset.

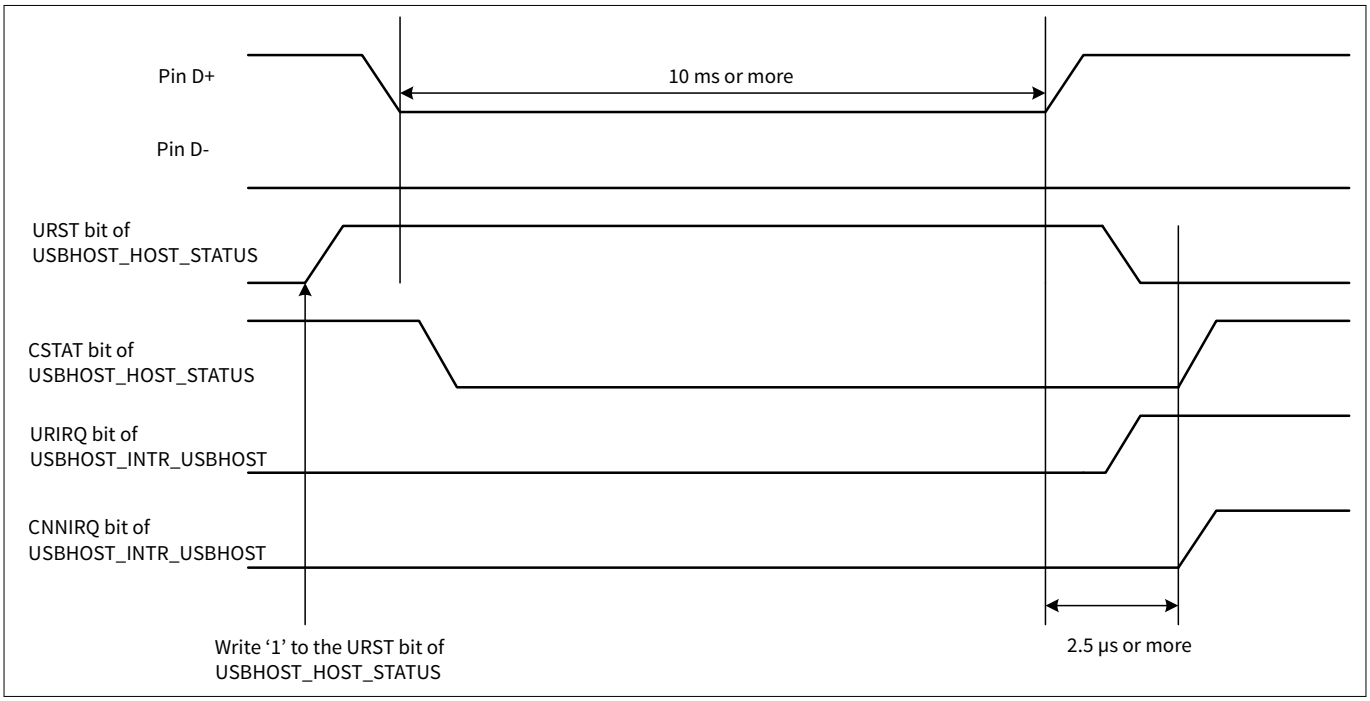


Figure 362 Device reset timing diagram

***Note**: No token is issued if a connection of the USB device is not detected after USB bus resetting has been completed.*

29.3.3.4 USB packets

Exchange of information between the host and the device takes place in the form of packets, which are always initiated by the host. There are three types of USB packets:

- Token

- Data
- Handshake

The following sections explain about the settings required to initiate the packets by the PSC3M/P6 MCU USB host.

29.3.3.4.1 Token packet

When issuing an IN, OUT, ISO or SETUP token, use the following setting steps to send a token packet.

1. Set DIR bits of the Host Endpoint 1 Control Register (USBHOST_HOST_EP1_CTL) and Host Endpoint 2 Control Register (USBHOST_HOST_EP2_CTL) while the BFINI bits of the Host Endpoint 1 Control Register (USBHOST_HOST_EP1_CTL) and Host Endpoint 2 Control Register (USBHOST_HOST_EP2_CTL) are ‘1’
2. Set the BFINI bits of the Host Endpoint 1 Control register (USBHOST_HOST_EP1_CTL) and Host Endpoint 2 Control Register (USBHOST_HOST_EP2_CTL) to ‘0’
3. Set the Host Address Register (USBHOST_HOST_ADDR)
4. Write the required data to the Host Token Endpoint Register (USBHOST_HOST_TOKEN)

When issuing an SOF token, set the Frame Setup Register (USBHOST_HOST_FRAME) and Host EOF Setup Register (USBHOST_HOST_EOF), and write the required data to the Host Token Endpoint Register (USBHOST_HOST_TOKEN). The setting above is not required if no change is made in the USBHOST_HOST_ADDR, USBHOST_HOST_EP1_CTL, USBHOST_HOST_EP2_CTL, USBHOST_HOST_FRAME and USBHOST_HOST_EOF registers.

Endpoint 1 and Endpoint 2 buffers are used to send and receive data. If the DIR bit of the USBHOST_HOST_EP1_CTL register is '1', the Endpoint 1 buffer is an OUT buffer. If the DIR bit of the USBHOST_HOST_EP2_CTL register is '0', then the Endpoint 2 buffer is an IN buffer. Both the endpoints should never be configured as an OUT buffer or IN buffer simultaneously, which implies that if Endpoint 1 is an OUT buffer then Endpoint 2 must be an IN buffer and vice-versa.

Use the following settings to process a token packet:

IN token

1. Set the BFINI bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers to '1'
2. Set the DIR bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers. Note that if Endpoint 1 is an OUT buffer then Endpoint 2 must be an IN buffer and vice-versa
3. Set the BFINI bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers to ‘0’
4. Specify the target address in the USBHOST_HOST_ADDR register
5. Specify the maximum number of bytes for the packet in the PKSn (n = 1, 2) bit field
6. Specify the target endpoint, token (IN token), and toggle data in the USBHOST_HOST_TOKEN register
7. Check if the token transfer is complete; the CMPIRQ bit will be set to ‘1’. To clear this interrupt, write ‘1’ to the CMPIRQ bit of the USBHOST_INTR_USBHOST register. Note that the interrupt is triggered only if the CMPIRQM bit of the USBHOST_INTR_USBHOST_MASK register is ‘1’
8. Check for token packet transfer errors using the USBHOST_HOST_ERR register. Handle the errors, if any, appropriately
9. Read the EPnDRQ (n = 1 or 2) bit of the USBHOST_INTR_HOST_EP register. A value of ‘1’ indicates that the packet transfer ended normally. Clear the EPnDRQ bit by writing ‘1’
10. Read data received from USBHOST_HOST_EPn_RWn_DR register (n =1, 2)

The USB PHY sends a token packet in the order of sync, token, address, endpoint, CRC5, and EOP based on the specified token; however, sync, CRC5, and EOP are sent automatically. After one packet is sent, the CMPIRQ bit of the USBHOST_INTR_USBHOST register is set to ‘1’. The TKNEN bit of the USBHOST_HOST_TOKEN register is set to ‘000’. At this time, if the CMPIRQM bit of USBHOST_INTR_USBHOST_MASK register is '1', an interrupt occurs. To clear this interrupt, write '1' to the CMPIRQ bit of the USBHOST_INTR_USBHOST register.

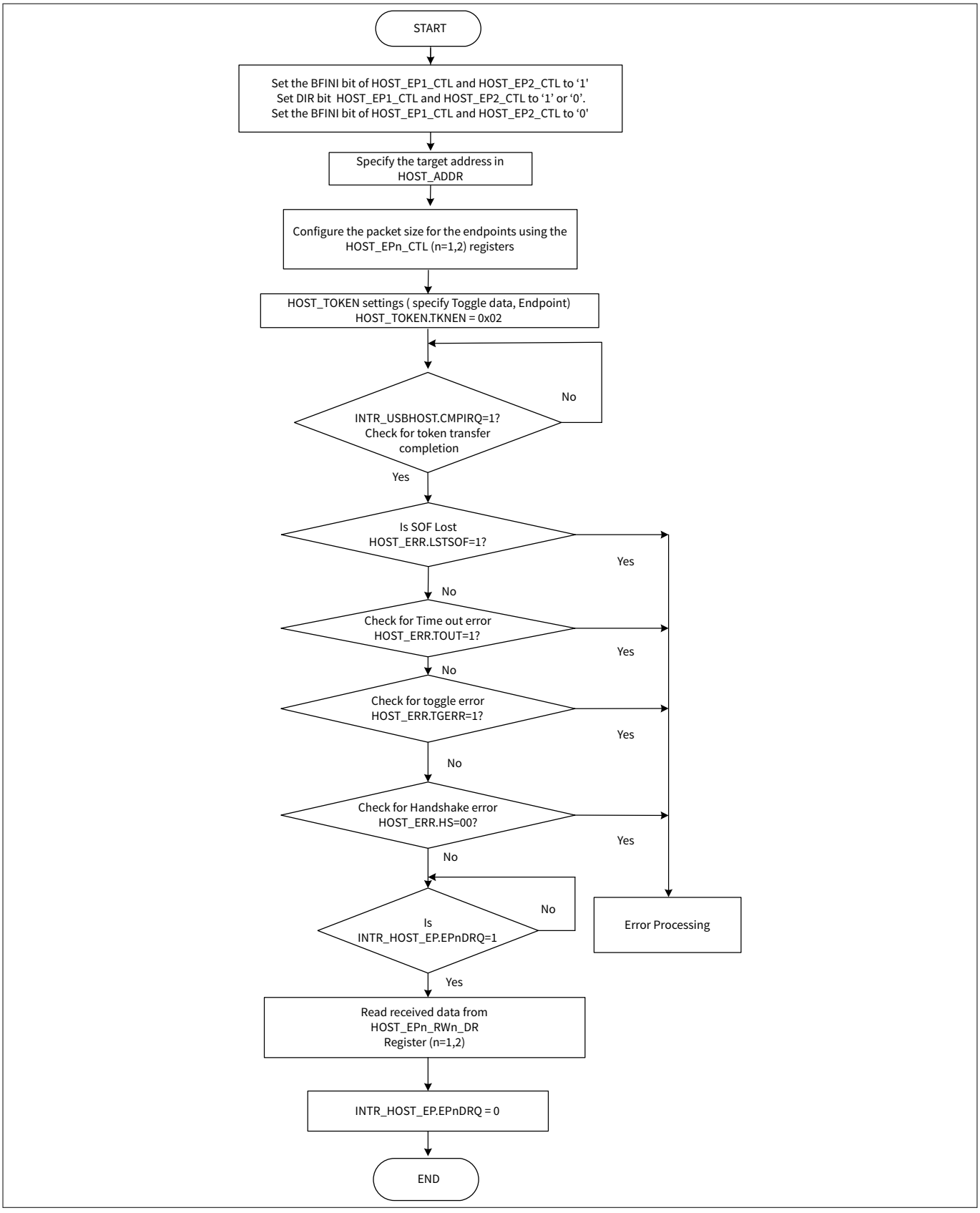


Figure 363 In token flow chart

OUT and SETUP tokens

1. Set the BFINI bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers to '1'
2. Set the DIR bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers. Note that if Endpoint 1 is an OUT buffer then Endpoint 2 must be an IN buffer and vice-versa
3. Set the BFINI bit of the USBHOST_HOST_EP1_CTL and USBHOST_HOST_EP2_CTL registers to ‘0’
4. Specify the target address in the USBHOST_HOST_ADDR register
5. Specify the maximum number of bytes for the packet in the PKSn (n = 1, 2) bit field
6. Write the data to be sent to the Endpoint n (n = 1 or 2) buffer. Use USBHOST_HOST_EPn_WR1_DR for 1-byte data and USBHOST_HOST_EPn_WR2_DR for 2-byte data. Also, set the EPnDRQ (n = 1 or 2) bit of the USBHOST_INTR_HOST_EP register as ‘1’
7. Specify the target endpoint, token (SETUP or OUT token), and toggle data in the USBHOST_HOST_TOKEN register
8. Check if the token transfer is complete; the CMPIRQ bit will be set to ‘1’. To clear this interrupt, write ‘1’ to the CMPIRQ bit of the USBHOST_INTR_USBHOST register. Note that the interrupt is triggered only if the CMPIRQM bit of the USBHOST_INTR_USBHOST_MASK register is ‘1’
9. Check for token packet transfer errors using the USBHOST_HOST_ERR register. Handle the errors, if any, appropriately

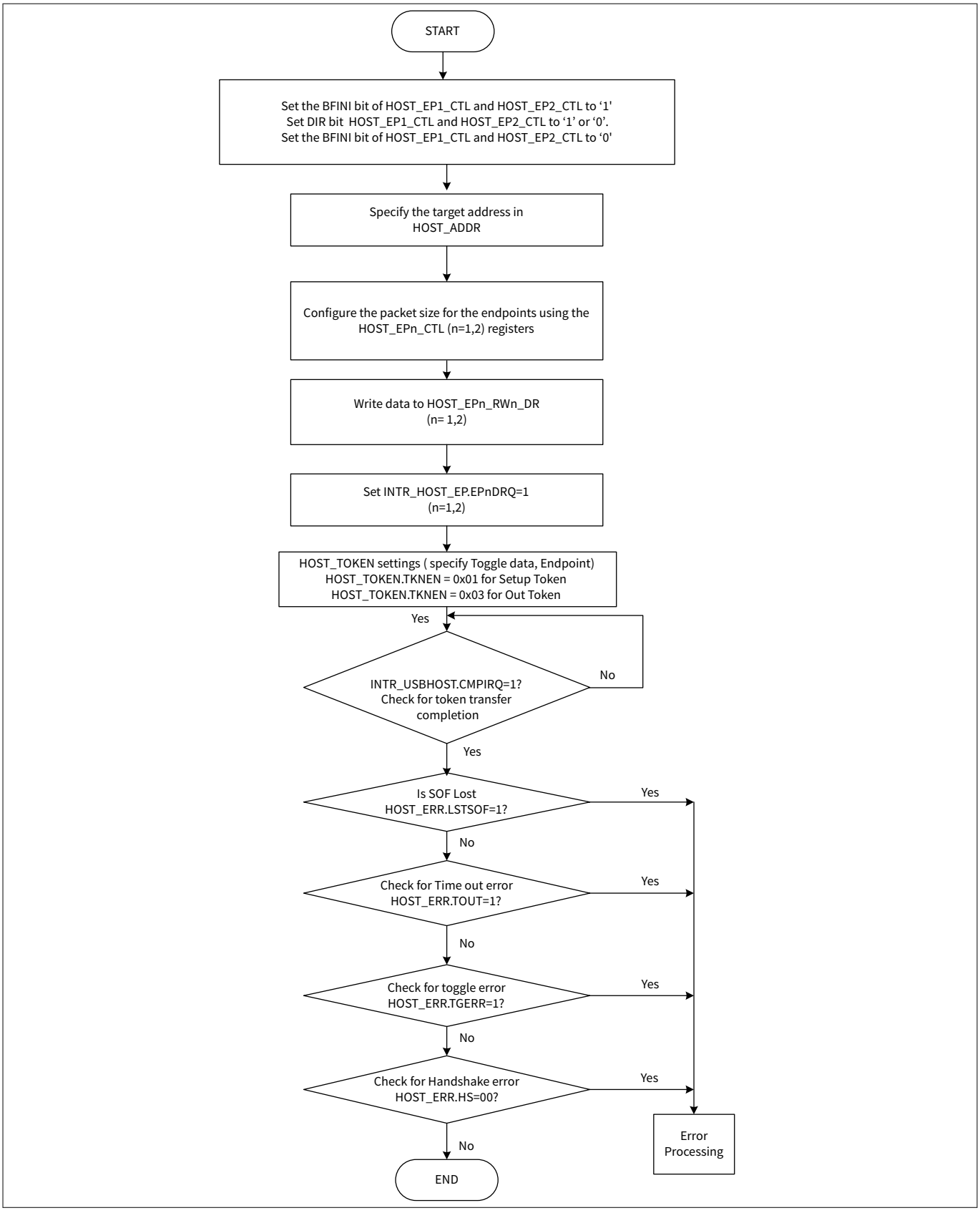


Figure 364 SETUP/OUT token flow chart

SOF token

When issuing an SOF token, specify the EOF time in the Host EOF Setup (USBHOST_HOST_EOF) register and the frame number in the Host Frame Setup (USBHOST_HOST_FRAME) register respectively. Then, specify an SOF token code in the TKNEN bit of the USBHOST_HOST_TOKEN register. After sync, SOF token, frame number, CRC5, and EOP are sent, the SOFBUSY bit of the USBHOST_HOST_STATUS register is set to '1', and the USBHOST_HOST_FRAME register is incremented by one. The CMPIRQ bit of the USBHOST_INTR_USBHOST register is also set to '1', causing the TKNEN bit of the USBHOST_HOST_TOKEN register to be cleared to '000'. To clear a token completion interrupt, write '1' to the CMPIRQ bit of the Host Interrupt (USBHOST_HIRQ) register.

An SOF is automatically sent every 1 ms while the SOFBUSY bit of the USBHOST_HOST_STATUS register is '1'. Figure 365 depicts steps to send an SOF token.

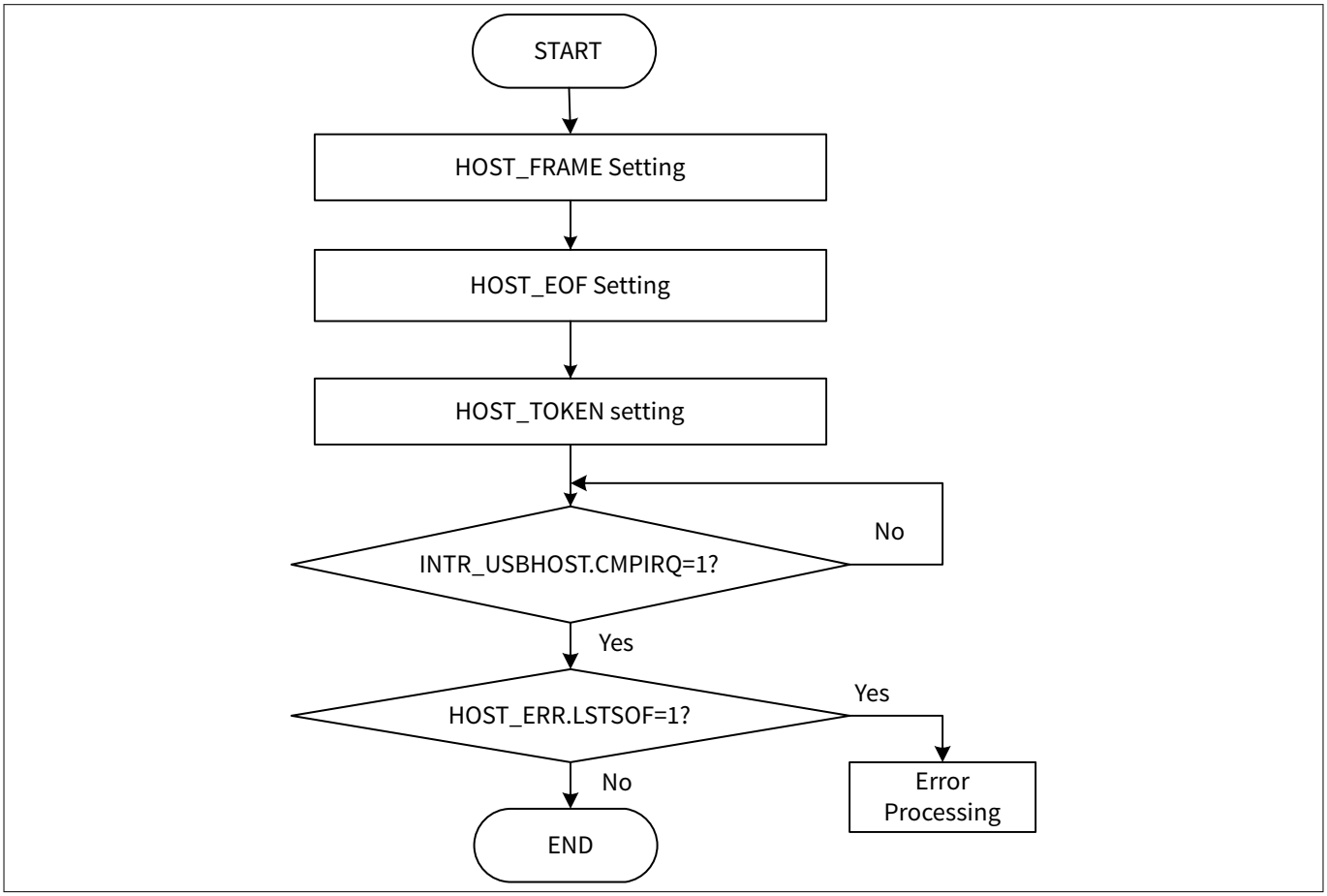


Figure 365 SOF token flow chart

The conditions (SOF stop conditions) to set the SOFBUSY bit of the USBHOST_HOST_STATUS register to '0' are as follows:

- Writing 0 to the SOFBUSY bit of USBHOST_HOST_STATUS
- Resetting the USB bus
- Writing 1 to the SUSP bit of USBHOST_HOST_STATUS
- Disconnecting the USB device (when the CSTAT bit of USBHOST_HOST_STATUS is '0')

The USBHOST_HOST_EOF register is used to prevent the SOF from being sent simultaneously with other tokens. If the TKNEN bit of the USBHOST_HOST_TOKEN register is written within the time period specified by the USBHOST_HOST_EOF register, the specified token is placed into the wait state. After the SOF is sent, the token in the wait state is issued. The USBHOST_HOST_EOF Setup register specifies a 1-bit time as the time unit. When the EOF setting is shorter than the 1-packet time, SOF may be sent doubly during execution of other tokens. In this case, the LSTSOF bit of the Host Error Status (USBHOST_HOST_ERR) register is set to '1' and SOF is not sent. If '1' is set to the LSTSOF bit of the USBHOST_HOST_ERR register, the value of the Host EOF Setup register must be increased.

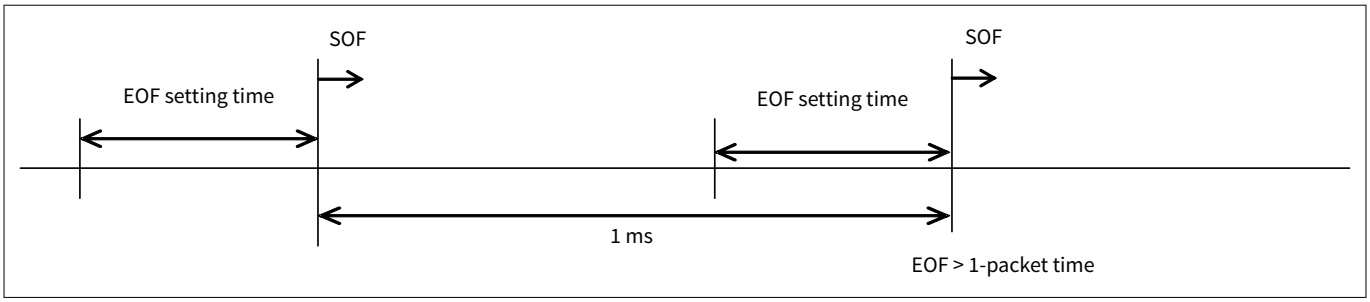


Figure 366 SOF timing

29.3.3.4.2 Data packet

Follow these steps to send or receive a data packet after sending a token packet.

- Transmitting data (host to device)
  - Sync pattern is automatically sent
  - If the TGGL bit of USBHOST_HOST_TOKEN is 0, DATA0 is sent. If the TGGL bit is 1, DATA1 is sent
  - If the DIR bit of the USBHOST_HOST_EP1_CTL register is 1 (that is, if EP1 is an OUT endpoint), select the Endpoint 1 buffer; otherwise, select the Endpoint 2 buffer. Then, send all the target data
  - 16-bit CRC is automatically sent
  - 2-bit EOP is automatically sent
  - 1-bit J State is automatically sent
- Receiving data (device to host)
  - Receive sync
  - Receive toggle data, and compare it with the value of the TGGL bit of HOST_TOKEN
  - If the toggle data matches the value of the TGGL bit, check the DIR bit of HOST_EP1_CTL. If the DIR bit is 1, select the Endpoint 2 buffer; otherwise, select the Endpoint 1 buffer. Then, distribute the received data to the respective buffers
  - Verify the 16-bit CRC when EOF is received

29.3.3.4.3 Handshake packet

A handshake packet is used to notify the remote device of the status of the local device. A handshake packet sends either one of ACK, NAK, and STALL from the receiving side when it is judged that the receiving side is ready to receive data normally. If the USB host receives a handshake packet, the type of the received handshake packet is set to the HS bit of the USBHOST_HOST_ERR register. If the USB Host sends a handshake packet, the type of the sent handshake packet is set to the HS bit of the Host Error Status Register (USBHOST_HOST_ERR).

29.3.3.5 Retry function

When a NAK or CRC error occurs at the end of a packet, if '1' is set to the RETRY bit of the Host Control 2 (USBHOST_HOST_CTL2) register, processing is retried repeatedly for the period specified in the Host Retry Timer Setup (USBHOST_HOST_RTIMER) register.

When an error other than STALL or device disconnection occurs, the target token is retried if the RETRY bit of HOST_CTL2 is 1. The conditions to end retry processing are as follows:

- The RETRY bit of HOST_CTL2 is set to 0
- '0' is detected in the retry timer
- The interrupt flag is generated by SOF (SOFIRQ of USBHOST_INTR_USBHOST = 1)

- ACK is detected
- A device disconnection is detected

The retry timer is activated when a token is sent while the RETRY bit of HOST_CTL2 is ‘1’. The retry time is then decremented by one when a 1-bit transfer clock (12 MHz in full-speed mode) is output. When the retry timer reaches 0, the target token is sent, and processing ends. If a token retry occurs in the EOF area, the retry timer is stopped until SOF is sent. After SOF is sent, the retry timer restarts with the value that is set when the timer stopped.

29.3.3.6 SOF interrupt

The SOFIRQ bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST) is set to ‘1’ at the start of SOF, depending on the configuration of the SOFSTEP bit in the Host Control 2 Register (USBHOST_HOST_CTL2) and the Host SOF Interrupt Frame Compare Register (USBHOST_HOST_FCOMP). If the SOFIRQM bit of the Interrupt USB Host Mask Register (USBHOST_INTR_USBHOST_MASK) is set to ‘1’, an interrupt is generated.

When SOF processing is performed using the Host Token Endpoint Register (USBHOST_HOST_TOKEN), the SOFIRQ bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST) is not set to ‘1’.

When the SOFSTEP bit of the Host Control 2 Register (USBHOST_HOST_CTL2) is ‘0’, the value in the Host SOF Interrupt Frame Compare Register (USB_HOSTHOST_FCOMP) is compared with the lower eight bits of the SOF token frame number. Set the SOFIRQ bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST) to ‘1’ when the SOF is sent and the values match.

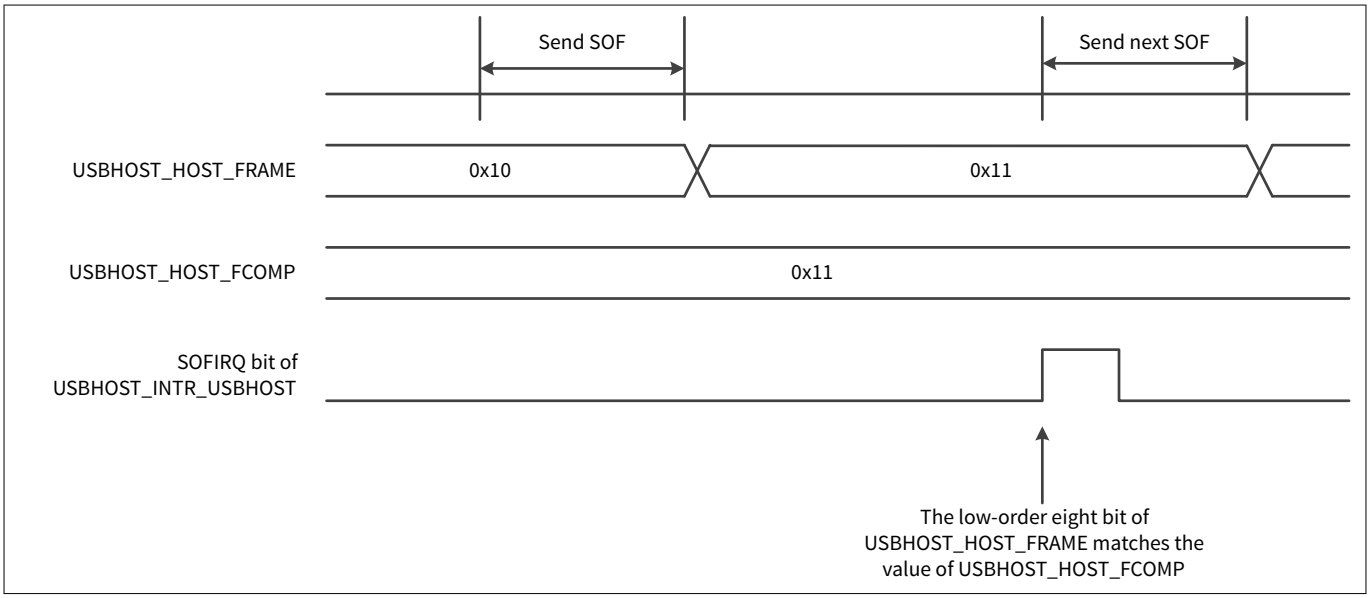


Figure 367 SOF interrupt when the SOFSTEP bit of USBHOST_HOST_CTL2 = 0

Set the SOFIRQ bit in the Interrupt USB Host Register (USBHOST_INTR_USBHOST) to ‘1’ each time an SOF is sent when the SOFSTEP bit of the Host Control 2 Register (USBHOST_HOST_CTL2) is set to ‘1’.

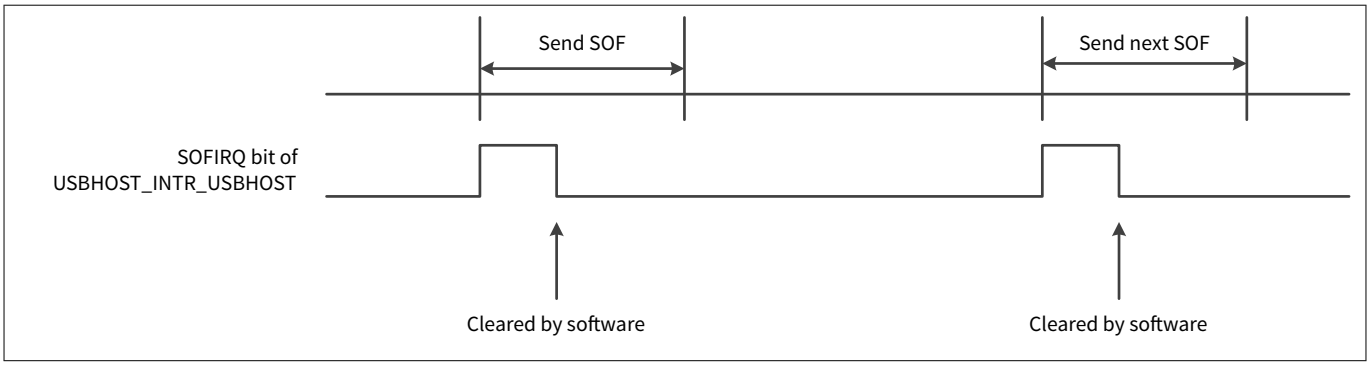


Figure 368 SOF interrupt when the SOFSTEP bit of USBHOST_HOST_CTL2 = 1

If the CANCEL bit of the Host Control 2 Register (USBHOST_HOST_CTL2) is set to '1', the target token is not sent when a token other than SOF is set to the Host Token Endpoint Register (USBHOST_HOST_TOKEN) in the EOF area.

If a token is set in the EOF area, the SOFIRQ bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST) is set to '1', and the TKNEN bits of the Host Token Endpoint Register (USBHOST_HOST_TOKEN) are cleared to '000'. In this case, the token is not sent, and the CMPIRQ bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST) is not set to '1'. At this time, the TCAN bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST) indicates that the token has been canceled.

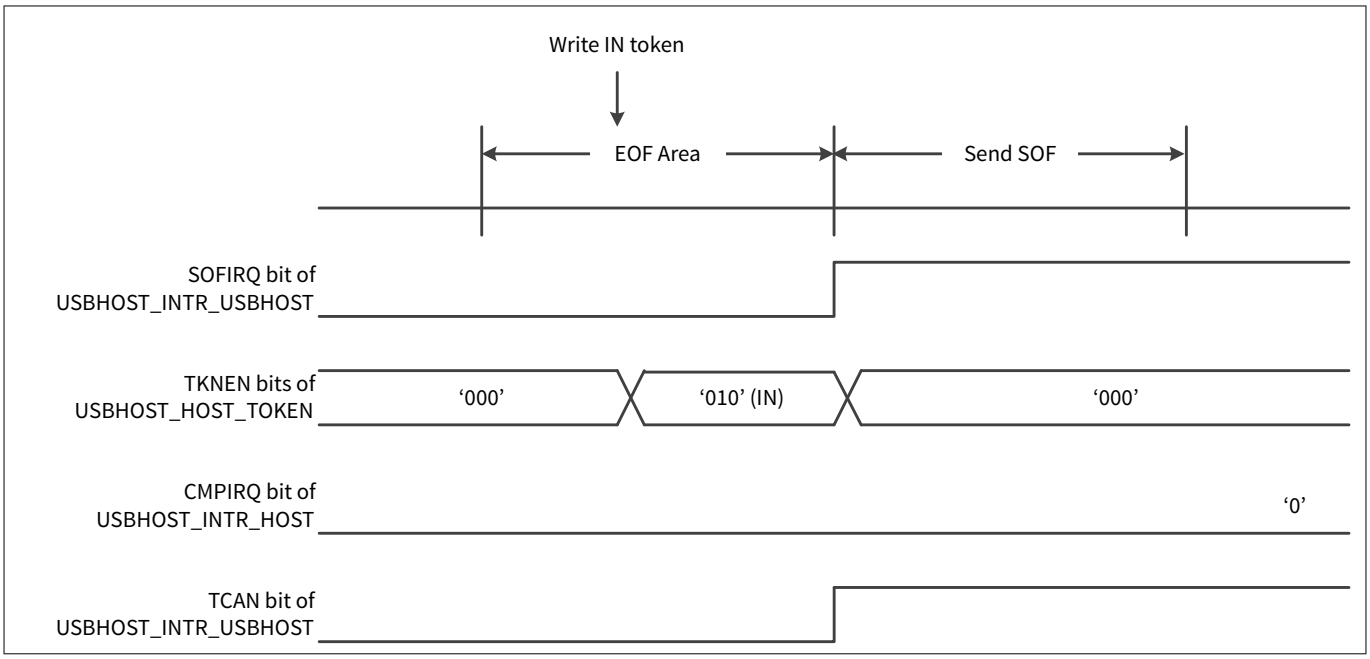


Figure 369 Token cancellation example when CANCEL bit of USBHOST_HOST_CTL2 = '1'

If a token is canceled (INTR_USBHOST.TCAN='1') and needs to be issued again, write a new token to the TKNEN bits of the Host Token Endpoint Register (HOST_TOKEN) after writing '0' to the TCAN bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST).

If the CANCEL bit of the Host Control 2 Register (HOST_CTL2) is '0', the token specified in the Host Token Endpoint Register (HOST_TOKEN) is sent following the next SOF.

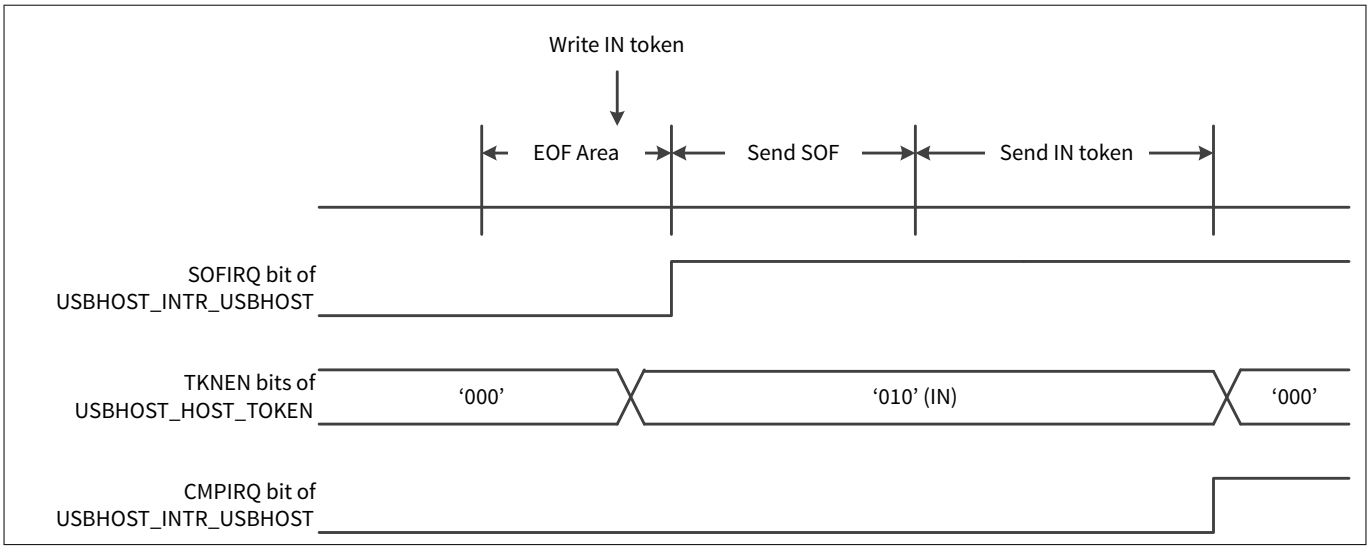


Figure 370 Token operation example when CANCEL bit of USBHOST_HOST_CTL2 = '0'

29.3.3.7 Error status

The USB host supports detection of the following types of errors:

- Stuffing error

  If 1 is writer to six successive bits, 0 is inserted into one bit. If 1 is successively detected in seven bits, it is regarded as a Stuffing error, and the STUFF bit of the USBHOST_HOST_ERR register is set to 1. To clear this status, write '1' to the STUFF bit
- Toggle error

  When sending an IN token, the Toggle Data field of a data packet is compared with the value of the TGGL bit of the USBHOST_HOST_TOKEN register. If they do not match, the TGERR bit of the USBHOST_HOST_ERR register is set to 1. To clear the TGERR bit, write '1' to the TGERR bit of the USBHOST_HOST_ERR register
- CRC error

  When receiving an IN token, the data and CRC of the received data packet are obtained with the CRC polynomial G(X) = X16 + X15 + X2 + 1. If the remainder is not 0x800D, it means that a CRC error has occurred, and the CRC bit of the USBHOST_HOST_ERR register is set to 1. To clear the CRC bit, write '1' to the CRC bit of USBHOST_HOST_ERR. For more details, see the documentation: CRC in USB
- Time Out error

  The TOUT bit of the USBHOST_HOST_ERR register is set and time out error is considered under any of the following conditions:
  - A data packet or handshake packet is not input in the specified time
  - SE0 is detected when receiving data
  - Stuffing error is detected

  To clear the TOUT bit, write 0 to the TOUT bit of the USBHOST_HOST_ERR register
- Receive error

  The PKSn (n = 1, 2) bit of Endpoint Control register (USBHOST_HOST_EPn_CTL; n=1, 2) indicates the receiver packet size. When the received data exceeds the specified receive packet size, the RERR bit of the USBHOST_HOST_ERR register is set to 1. To clear the RERR bit, write '1' to the RERR bit of USBHOST_HOST_ERR
- Lost SOF error

When the LSTSOF bit of the USBHOST_HOST_ERR register is set, it means that the SOF token cannot be sent because another token is in process. When this bit is '0', it means that no lost SOF error is detected

29.3.3.8 End of package

If a packet ends in the USB host, the CMPIRQ bit of the USBHOST_INTR_USBHOST register is set to 1. At this time, if the CMPIRQM bit of the USBHOST_INTR_USBHOST_MASK register is 1, an interrupt occurs.

When a packet ends, the interrupt is generated when the TKNEN bit of the USBHOST_HOST_TOKEN register is set as '001' (SETUP token), '010' (IN token), '011' (OUT token), or '100' (SOF token).

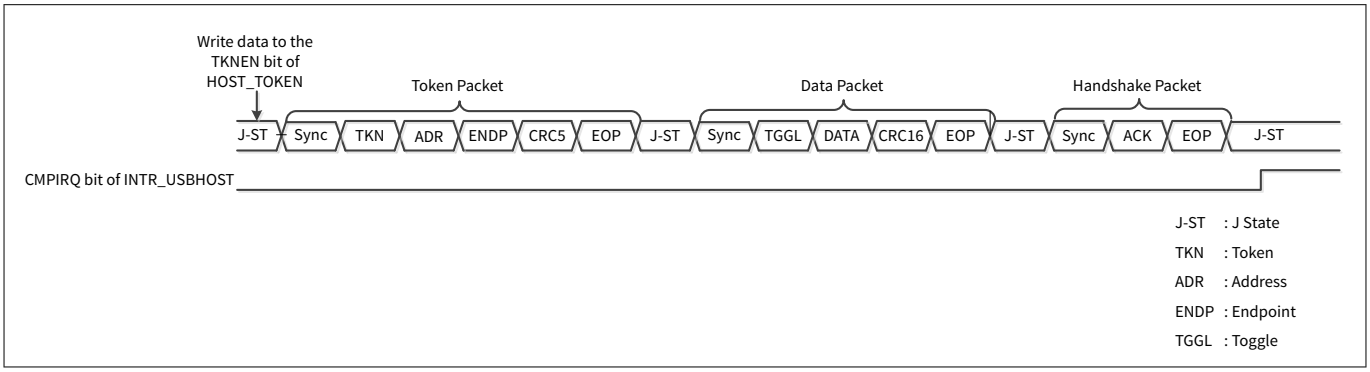


Figure 371 Package end timing diagram (for IN/OUT/SETUP token)

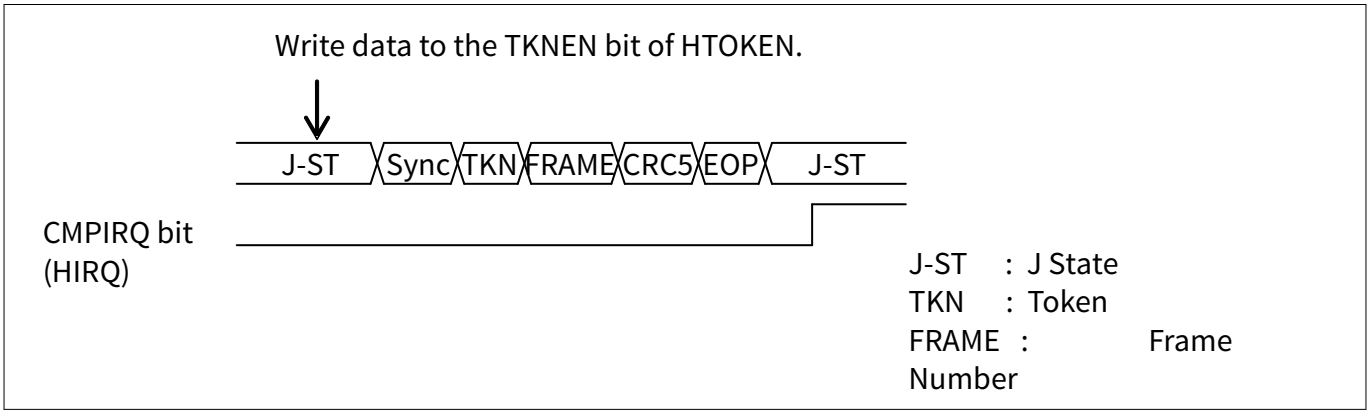


Figure 372 Package end timing diagram (for SOF token)

29.3.3.9 Interrupt source

The USB host generates 11 interrupts to the CPU. These interrupts are mapped to three general-purpose interrupt lines: USBHOST_INTR_LO, USBHOST_INTR_MED, and USBHOST_INTR_HI. Each of these three interrupt lines has an associated status register, which identifies the cause of the interrupt event. These are USBHOST_INTR_USBHOST_CAUSE_LO, USBHOST_INTR_USBHOST_CAUSE_MED, USBHOST_INTR_USBHOST_CAUSE_HI, USBHOST_INTR_HOST_EP_CAUSE_LO, USBHOST_INTR_HOST_EP_CAUSE_MED, and USBHOST_INTR_HOST_EP_CAUSE_HI.

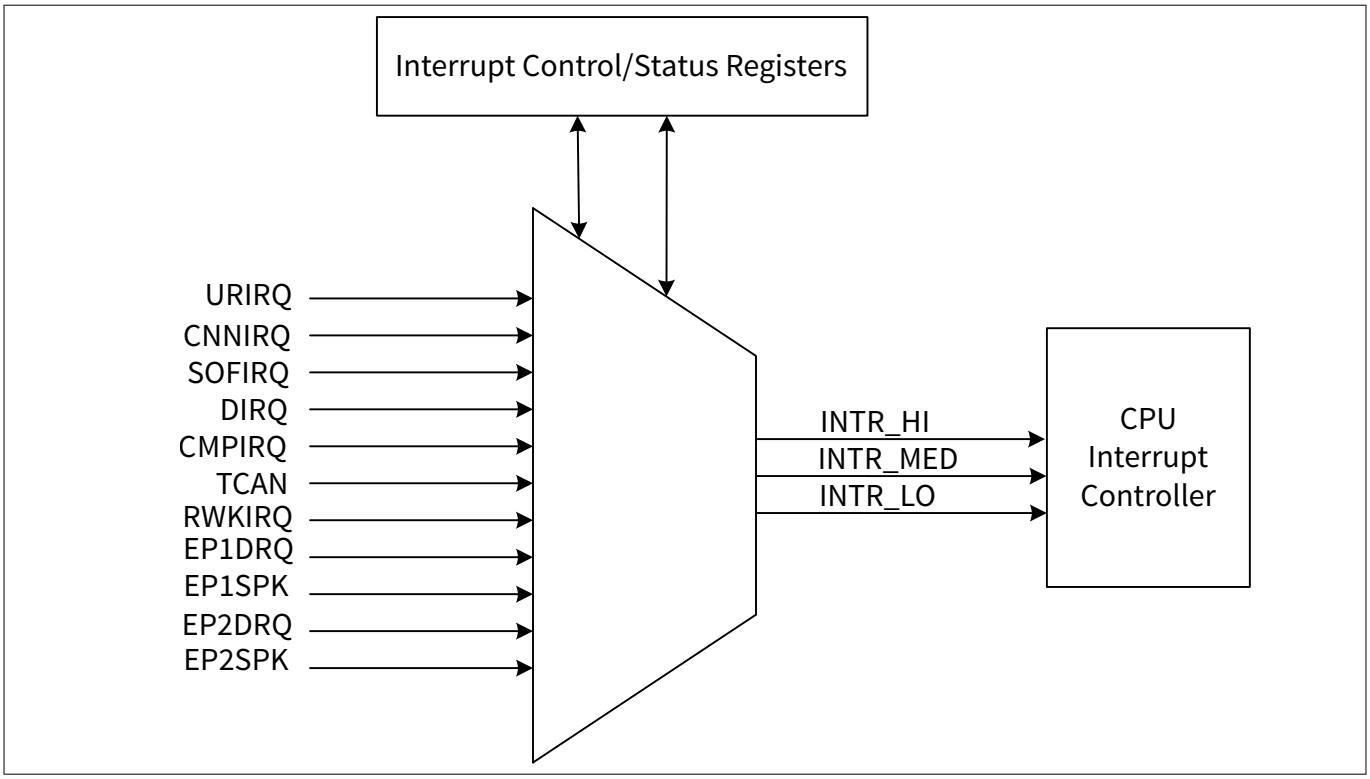


Figure 373 USB host interrupt sources

The following events generate an interrupt on one of the three interrupt lines:

- Start-of-frame (SOF) event

  SOF interrupt is generated when an SOF token is sent. The SOF interrupt is enabled by setting the SOFIRQM bit of the USBHOST_INTR_USBHOST_MASK register. The status of the SOF interrupt is reflected in the SOFIRQ bit of the USBHOST_INTR_USBHOST register
- Device connection and disconnection events

  The CNNIRQ bit of the USBHOST_INTR_USBHOST register is set to ‘1’ when a device connection is detected. The interrupt can be enabled by setting the CNNIRQM bit of the USBHOST_INTR_USBHOST_MASK register. When a device is disconnected, an interrupt is generated if the DIRQM bitfield of USBHOST_INTR_USBHOST_MASK is set to ‘1’. The status of the device disconnection is reflected in the DIRQ bit of the USBHOST_INTR_USBHOST register
- USB bus reset event

  The URIRQ bit of the USBHOST_INTR_USBHOST register is set when the USB bus reset ends. An interrupt is generated if the URIRQM bit of the USBHOST_INTR_USBHOST_MASK register is set to ‘1’
- Token completion event

  When token (IN, OUT, or SETUP token) processing is complete, the CMPIRQ bit of the USBHOST_INTR_USBHOST register is set. An interrupt is generated if the CMPIRQM bit is set to ‘1’
- Interrupt on resume event

  The RWKIRQ bit of the USBHOST_INTR_USBHOST register is set when the host enters the resume state from suspend state. The RWKIRQ bit is set under the following conditions:
  - The SUSP bit of the USBHOST_HOST_STATUS register is set to ‘0’
  - The D+/D– pins are placed in the K-state
  - A device connection/disconnection is detected

  An interrupt is generated on a resume event if the RWKIRQM bit of the USBHOST_INTR_USBHOST_MASK register is set to ‘1’

- Interrupt on token cancellation

  An interrupt is generated when a token send is canceled based on the setting of the CANCEL bit in the Host Control 2 register (USBHOST_HOST_CTL2). If the CANCEL bit is set and if the target token is written to the USBHOST_HOST_TOKEN register in the EOF area (specified in the Host EOF Setup register), the token send is canceled
- Endpoint interrupts

  The USB host has two endpoints: Endpoint1 and Endpoint2. Each endpoint can generate two interrupts. When the packet transfer of an endpoint ends normally, the EPnDRQ (n=1or 2) bit of the USBHOST_INTR_HOST_EP register is set to '1'. An interrupt can be triggered if the EPnDRQM (n=1or 2) bit of the USBHOST_INTR_HOST_EP_MASK register is set to '1'. If the packet size does not match the packet size specified in the PKS bit of the HOST_EPn_CTL (n=1 or 2) register, the EPnSPK (n=1or 2) bit of the USBHOST_INTR_HOST_EP register is set. An interrupt can be configured by setting the EPnSPKM (n=1or 2) bit of the USBHOST_INTR_HOST_EP_MASK register

29.3.3.10 DMA transfer function

Data handled by the USB host can be transferred via DMA between the send/receive buffer and the internal RAM. The DMA Request of Endpoint 1 is generated to usb_dma_req[1] and the DMA Request of Endpoint 2 is generated to usb_dma_req[2].There are two modes of DMA transfer:

- Packet transfer mode
- Automatic buffer transfer mode

29.3.3.10.1 Packet transfer mode

The packet transfer mode transfers each packet according to the configured data size in DMA. This transfer mode can access each buffer of the endpoints.

In the packet transfer mode the OUT direction transfer (host to device) involves the following sequence of steps:

1. After the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register is set and the interrupt handling is entered, configure the DMA register settings relevant to the number of transfers and block size corresponding to the data size to be transferred in the next OUT packet; then, enable DMA to start the transfer
2. After the DMA transfer, clear the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register and the interrupt cause flag of DMAC, and return from the interrupt handling

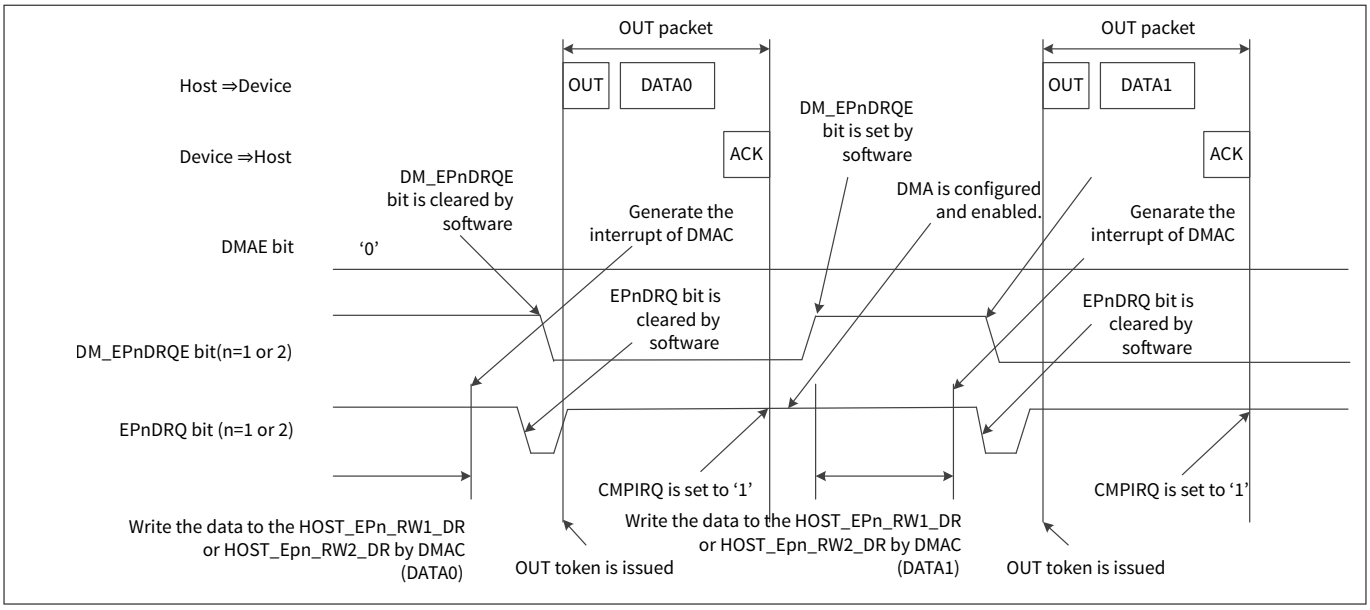


Figure 374 Packet transfer mode: OUT direction

The data transfer in the IN direction (device to host) involves the following sequence of steps:

1. After the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register is set and the interrupt handling is entered, check the transfer data size
2. Configure the DMA register setting relevant to the number of transfers and block size corresponding to the transfer data size, and then enable DMA to start the transfer
3. After the transfer, clear the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register and the interrupt source flag of DMAC, and return from the interrupt handling

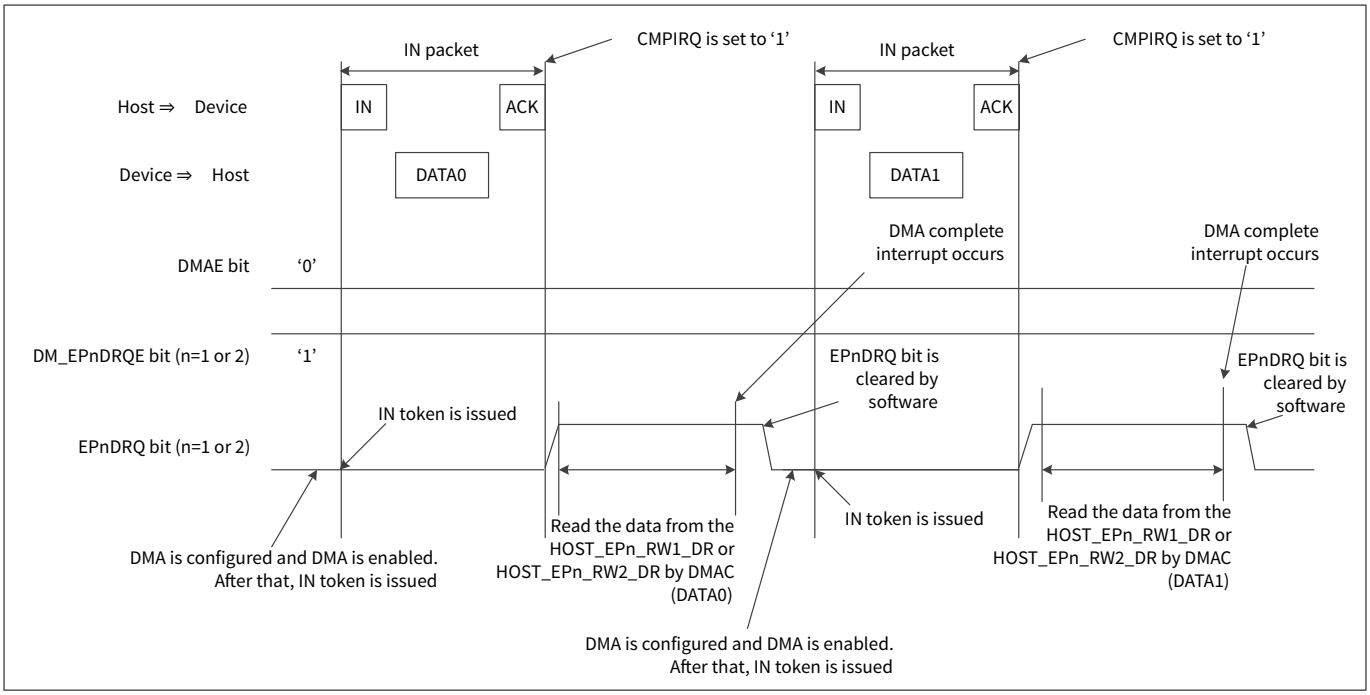


Figure 375 Packet transfer mode: IN direction

29.3.3.10.2 Automatic data transfer mode

Automatic data transfer mode is enabled by setting the DMAE bit of the USBHOST_HOST_EPn_CTL register (n=1 or 2). When the EPnDRQ bit (n=1 or 2) of the USBHOST_INTR_HOST_EP register is set while DMAE is enabled, the DMA request is automatically cleared when the data size corresponding to PKSn (n = 1, 2) bits of the USBHOST_HOST_EPn_CTL register (n=1 or 2) has been transferred. Later, the same sequence is repeated until the data size configured in the DMA is reached. In this mode, configuration by software is not required. Thus, this mode can transfer data automatically by a single setting. It can also transfer even bytes; to transfer odd bytes, a software transfer sequence is required.

The data transfer in the OUT direction (host to device) must be processed in the following sequence:

1. Configure the DMA register setting relevant to the number of transfers and block size corresponding to the total data size, and then enable DMA to start the transfer
2. Set the packet size and total number of bytes for DMA transfer to PKS bits of the USBHOST_HOST_EPn_CTL (n=1 or 2) register and BLK_NUM bits of the USBHOST_HOST_EPn_BLK (n=1 or 2) register, respectively
3. Enable DMAE bit of the USBHOST_HOST_EPn_CTL (n=1 or 2) register and EPnDRQE bit of the Host DMA Enable (USBHOST_HOST_DMA_ENBL) (n=1 or 2) register
4. After the transfer, reconfigure the DMAC using an interrupt generated by the DMA controller, and clear the flag to return from interrupt handling

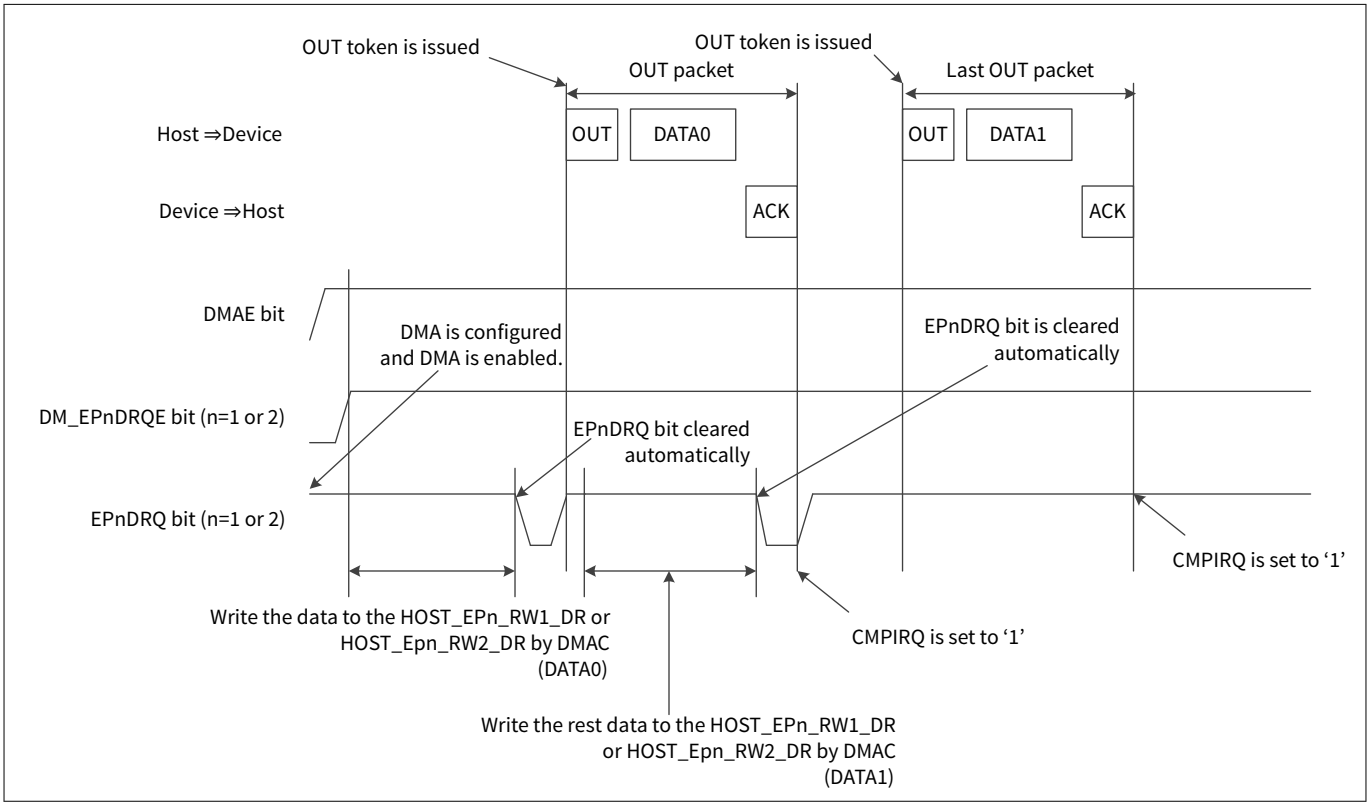


Figure 376 Automatic data transfer mode: OUT direction

To transfer the odd bytes of data via DMA, write the last byte of data by software transfer.

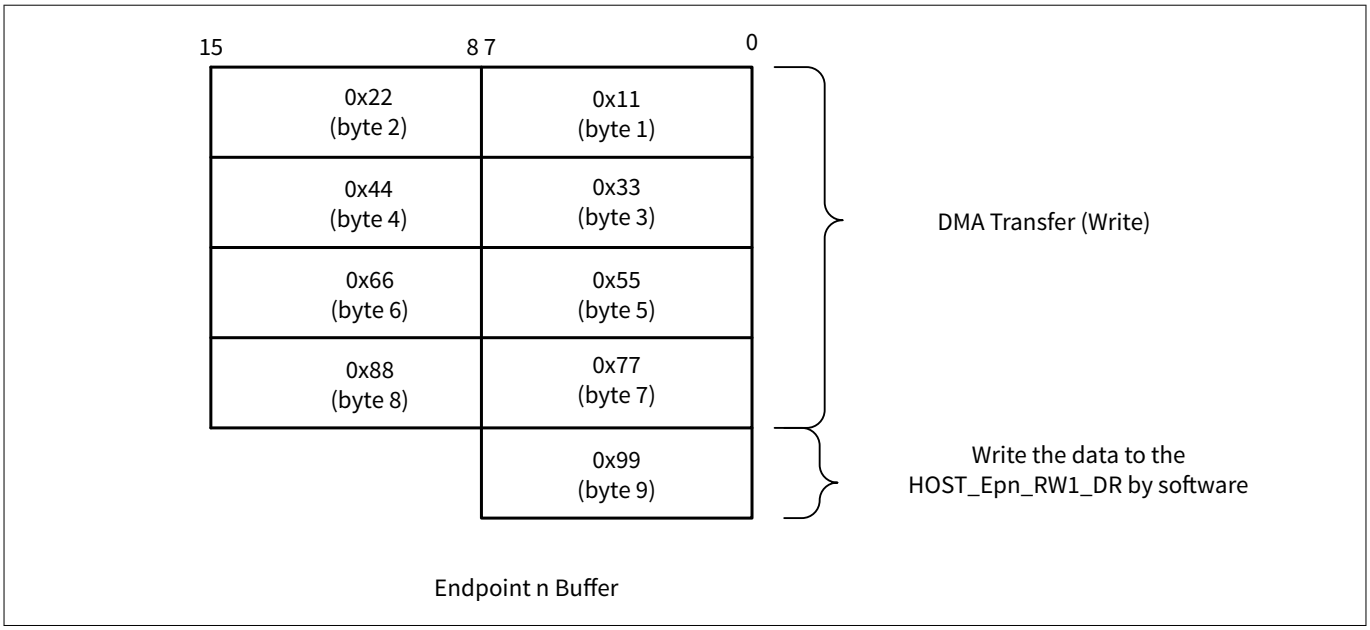


Figure 377 Odd bytes transfer in the OUT direction

The data transfer in the IN direction (device to host) must be processed in the following sequence:

1. Configure the DMA register setting relevant to the number of transfers and block size corresponding to the total data size, and then enable DMA to start the transfer
2. Enable DMAE bit of the USBHOST_HOST_EPn_CTL (n = 1 or 2) register and EPnDRQE bit of the USBHOST_HOST_DMA_ENBL (n = 1 or 2) register
3. After the transfer, reconfigure the DMA controller using an interrupt generated by the DMAC and clear the flag to return from interrupt handling

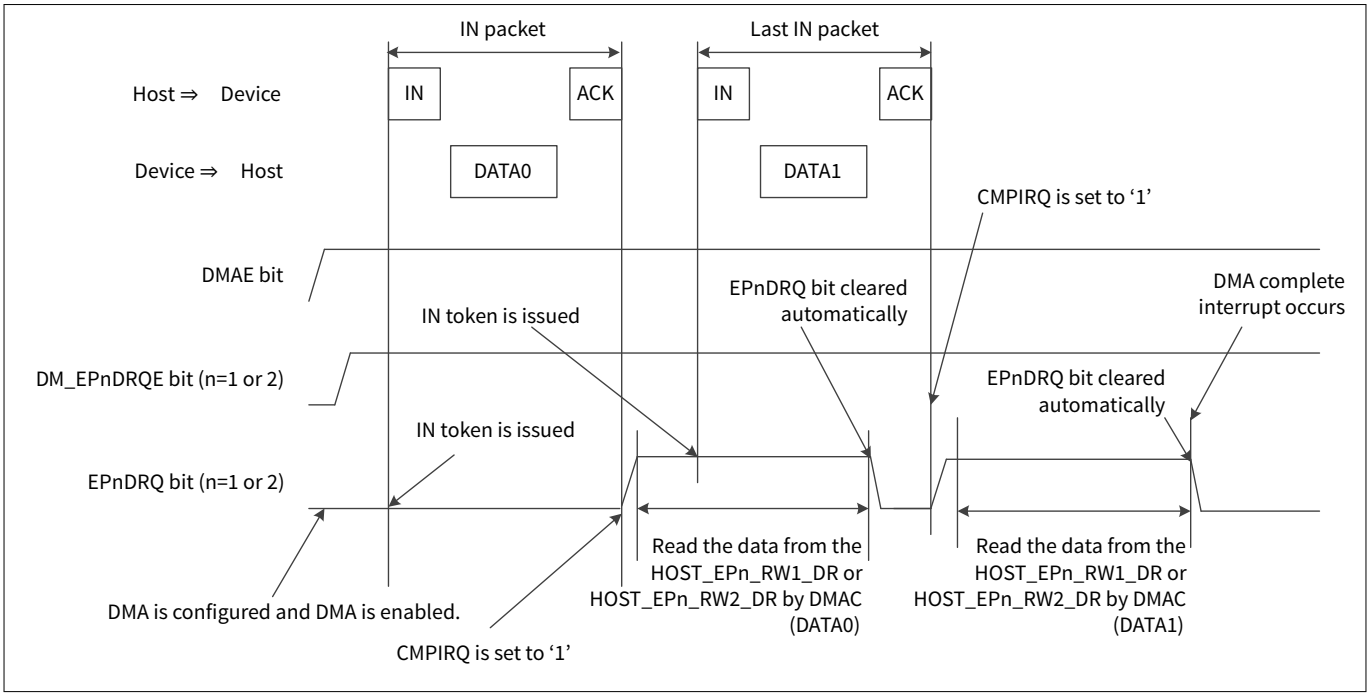


Figure 378 Automatic data transfer mode: IN direction

To transfer the data size corresponding to odd number of bytes via DMA, either of the following methods can be used:

- Use the software transfer only for the last data, and read the low-order byte (HOST_EPn_RW1_DR: n=1 or 2)
- Transfer all the data + 1 byte via DMA, and discard the last data after an endian conversion

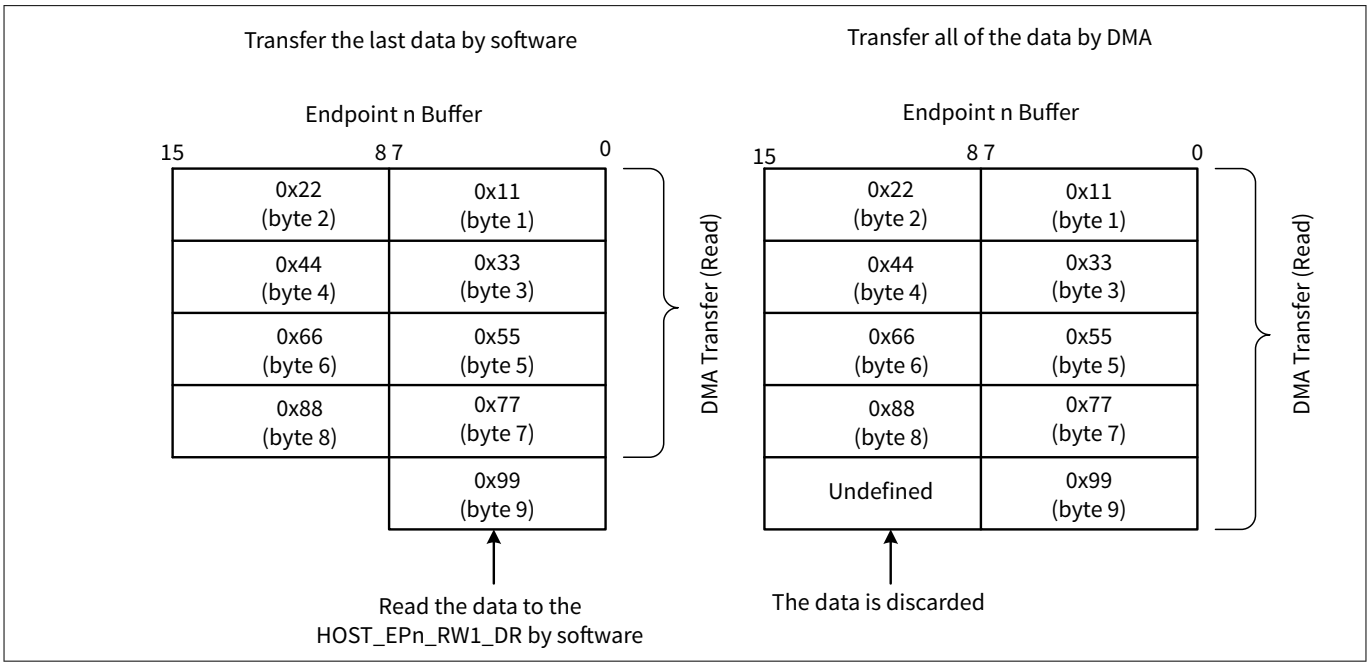


Figure 379 Odd bytes transfer in the IN direction

29.3.3.11 NULL transfer function

If data sent from the USB function is the last packet and satisfies the maximum packet size, the 0-byte can be automatically transferred via the next packet transfer. DMAE bit of the USBHOST_HOST_EPn_CTL (n = 1 or 2) register must be enabled to use this function. This function is valid only in OUT transfer. Set the NULLE bit of the USBHOST_HOST_EPn_CTL (n = 1 or 2) register to enable this.

NULL transfer mode works when the following conditions are met:

- Automatic buffer transfer mode is set (DMAE = '1')
- The number of data bytes to be written is defined by the BLK_NUM field of the Host Endpoint n Block Register (USBHOST_HOST_EPn_BLK, n=1 or 2)

After the last data has been written to the buffer via DMA, the EPnDRQ bit of the Interrupt USB Host Endpoint Register (USBHOST_INTR_HOST_EP) is not set until the 0-byte data is sent. After the final data transfer completes, the DM_EPnDRQE bit of the Host DMA Enable Register (USBHOST_HOST_DMA_ENBL) is set to '0'. Figure 380 shows the timing to access the buffer.

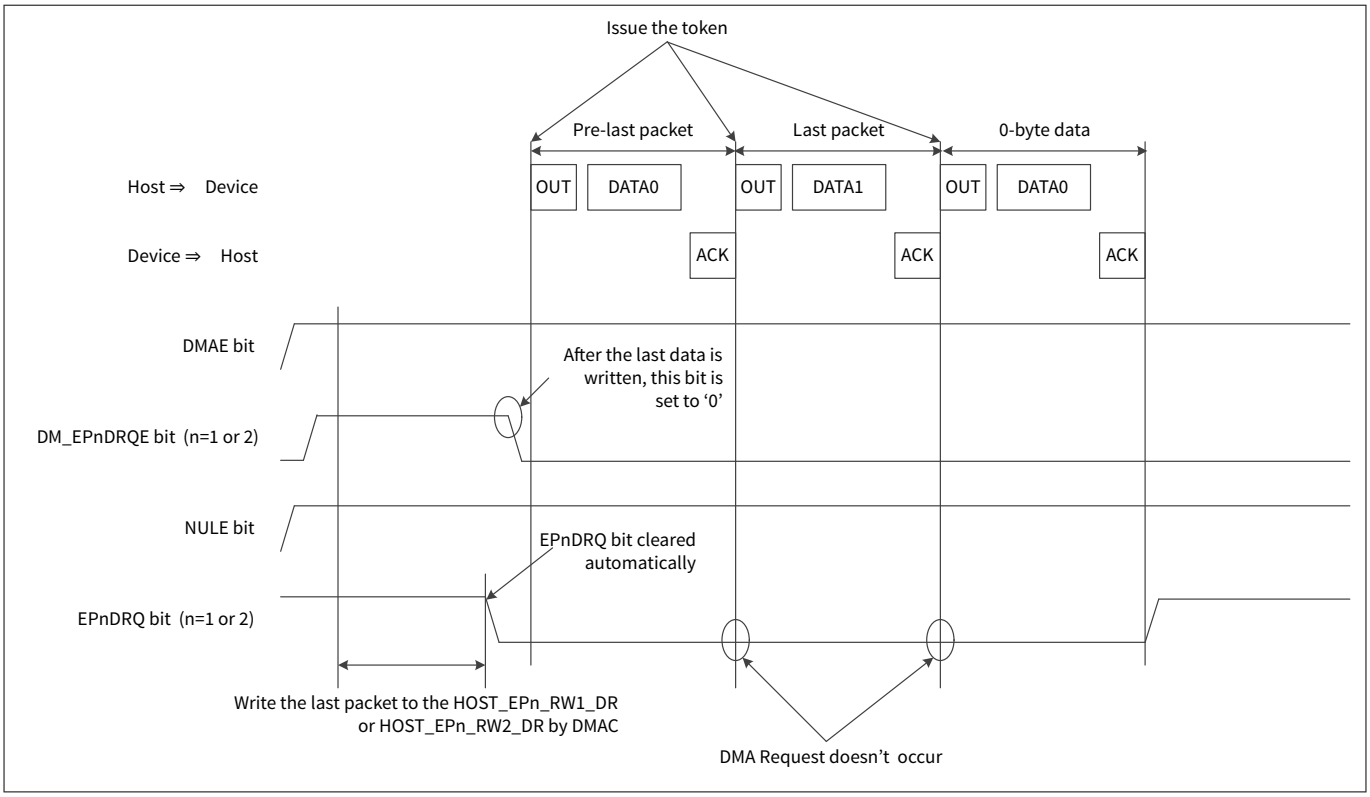


Figure 380 NULL data transfer

29.3.3.12 Suspend and resume operations

The USB host supports suspend and resume operations.

29.3.3.12.1 Suspend operation

If the SUSP bit of the USBHOST_HOST_STATUS register is set to '1', the following procedure is performed, and the USB circuit is placed in the suspend state. At this time, the stopping the operating clock for USB is prohibited.

- The USB bus is placed in the high-impedance state
- All circuit blocks with no clock required are stopped

However, the following operations are prohibited while resetting the USB bus.

- The SOFBUSY bit of the Host Status Register (USBHOST_HOST_STATUS) is set to '1', or the USB Host enters the suspend state during data transfer
- Clocks supplied to the USB are stopped in the suspend state

29.3.3.12.2 Resume operation

The USB bus changes from the suspend state to the resume state to start processing when one of the following conditions is satisfied.

- SUSP bit of the USBHOST_HOST_STATUS register is set to '0'
- The D+ or D– pin is placed in the K-state mode by the device
- A device disconnection is detected
- A device connection is detected

The host can start issuing tokens when RWKIRQ bit of the USBHOST_INTR_USBHOST register is set to ‘1’.

29.3.3.12.3 DeepSleep operation

Ensure that the USB Host is in the suspend state before entering DeepSleep. When entering DeepSleep, the suspend state is reset. As a result, the resume operation cannot be executed. To execute a resume operation after returning from DeepSleep, the firmware must perform the following steps.

1. Se the SUSP bit of the Host Status Register (USBHOST_HOST_STATUS) to ‘1’
2. If the SUSP bit of the Host Status Register (USBHOST_HOST_STATUS) goes into ‘1’, this bit is set to ‘0’
3. When the RWKIRQ bit of the Interrupt USB Host Register (USBHOST_INTR_USBHOST) is set to ‘1’, execute a USB bus reset

After finishing the USB bus reset, issue a token. When returning from the DeepSleep, SOF is not executed automatically. If the SOF wants to be executed, issue an SOF token.

***Note**: In Hibernate mode, the USBPHY is in the Reset state. Therefore, the pad freeze functionality does not work for Hibernate mode.*

29.3.3.13 Device disconnection

The device disconnection timer starts when both the D+ and D– pins are set to LOW. If both D+ and D– remain at LOW for 2.5 µs or more, the device is considered to be disconnected. This then sets the CSTAT bit of the USBHOST_HOST_STATUS register as ‘0’ and the DIRQ bit of the USBHOST_INTR_USBHOST register as ‘1’. At this time, if the DIRQM bit of the USBHOST_INTR_USBHOST_MASK register is 1, an interrupt occurs. To clear this interrupt, write '1' to the DIRQ bit of the USBHOST_INTR_USBHOST register.

If the USB bus is reset, the device is considered to be disconnected. In this case, the CSTAT bit of the USBHOST_HOST_STATUS register is set to ‘0’, but the DIRQ bit of the USBHOST_INTR_USBHOST register is not set to 1.

29.4 USB registers

Table 315 USB device registers

Name	Description
USBDEV_EP0_DR	Control endpoint EP0 data register
USBDEV_CR0	USB control 0 register
USBDEV_CR1	USB control 1 register
USBDEV_SIE_EP_INT_EN	USB SIE data endpoint interrupt enable register
USBDEV_SIE_EP_INT_SR	USB SIE data endpoint interrupt status
USBDEV_SIE_EPx_CNT0	Non-control endpoint count register
USBDEV_SIE_EPx_CNT1	Non-control endpoint count register
USBDEV_SIE_EPx_CR0	Non-control endpoint's control register
USBDEV_USBIO_CR0	USBIO control 0 register
USBDEV_USBIO_CR2	USBIO control 2 register
USBDEV_USBIO_CR1	USBIO control 1 register

(table continues...)

Table 315 (continued) USB device registers

Name	Description
USBDEV_DYN_RECONFIG	USB dynamic reconfiguration register
USBDEV_SOF0	Start of frame register
USBDEV_SOF1	Start of frame register
USBDEV_SOF16	Start of frame register
USBDEV_OSCLK_DR0	Oscillator lock data register 0
USBDEV_OSCLK_DR1	Oscillator lock data register 1
USBDEV_OSCLK_DR16	Oscillator lock data register
USBDEV_EP0_CR	Endpoint 0 control register
USBDEV_EP0_CNT	Endpoint 0 count register
USBDEV_ARB_RWx_WA	Endpoint write address value register. Pointer value increments by 1 when accessed by CPU/debugger
USBDEV_ARB_RWx_WA_MSB	Endpoint write address value register
USBDEV_ARB_RWx_WA16	Endpoint write address value register. Pointer value increments by 2 when accessed by CPU/debugger
USBDEV_ARB_RWx_RA	Endpoint read address value register. Pointer value increments by 1 when accessed by CPU/debugger
USBDEV_ARB_RWx_RA_MSB	Endpoint read address value register
USBDEV_ARB_RWx_RA16	Endpoint read address value register. Pointer value increments by 2 when accessed by CPU/debugger
USBDEV_ARB_RWx_DR	Endpoint data register
USBDEV_ARB_RWx_DR16	Endpoint data register
USBDEV_ARB_EPx_CFG	Endpoint configuration register
USBDEV_ARB_EPx_INT_EN	Endpoint interrupt enable register
USBDEV_ARB_EPx_SR	Endpoint interrupt enable register
USBDEV_ARB_CFG	Arbiter configuration register
USBDEV_ARB_INT_EN	Arbiter interrupt enable register
USBDEV_ARB_INT_SR	Arbiter interrupt status register
USBDEV_BUF_SIZE	Dedicated endpoint buffer size register
USBDEV_EP_ACTIVE	Endpoint active indication register
USBDEV_EP_TYPE	Endpoint type (IN/OUT) indication register
USBDEV_USB_CLK_EN	USB block clock enable register
USBDEV_CWA	Common area write address register
USBDEV_CWA_MSB	Common area write address register
USBDEV_CWA16	Common area write address register
USBDEV_DMA_THRES	DMA burst/threshold configuration register

(table continues...)

Table 315 (continued) USB device registers

Name	Description
USBDEV_DMA_THRES_MSB	DMA burst/threshold configuration register
USBDEV_DMA_THRES16	DMA burst/threshold configuration register
USBDEV_BUS_RST_CNT	Bus reset count register
USBDEV_MEM_DATA	Data register

Table 316 USB device link power management (LPM) registers

Name	Description
USBLPM_LPM_CTL	LPM control register
USBLPM_LPM_STAT	LPM status register
USBLPM_INTR_SIE	USB SOF, BUS RESET, and EP0 interrupt status register
USBLPM_INTR_SIE_SET	USB SOF, BUS RESET, and EP0 interrupt set register
USBLPM_INTR_SIE_MASK	USB SOF, BUS RESET, and EP0 interrupt mask register
USBLPM_INTR_SIE_MASKED	USB SOF, BUS RESET, and EP0 interrupt masked register
USBLPM_INTR_LVL_SEL	Select interrupt level for each interrupt source register
USBLPM_INTR_CAUSE_HI	High-priority interrupt cause register
USBLPM_INTR_CAUSE_MED	Medium-priority interrupt cause register
USBLPM_INTR_CAUSE_LO	Low-priority interrupt cause register

Table 317 USB host registers

Name	Description
USBHOST_HOST_CTL0	Host control 0 register
USBHOST_HOST_CTL1	Host control 1 register
USBHOST_HOST_CTL2	Host control 2 register
USBHOST_HOST_ERR	Host error status register
USBHOST_HOST_STATUS	Host status register
USBHOST_HOST_FCOMP	Host SOF interrupt frame compare register
USBHOST_HOST_RTIMER	Host retry timer setup register
USBHOST_HOST_ADDR	Host address register
USBHOST_HOST_EOF	Host EOF setup register
USBHOST_HOST_FRAME	Host frame setup register
USBHOST_HOST_TOKEN	Host token endpoint register
USBHOST_HOST_EP1_CTL	Host endpoint 1 control register
USBHOST_HOST_EP1_STATUS	Host endpoint 1 status register
USBHOST_HOST_EP1_RW1_DR	Host endpoint 1 data 1-byte register

(table continues...)

Table 317 (continued) USB host registers

Name	Description
USBHOST_HOST_EP1_RW2_DR	Host endpoint 1 data 2-byte register
USBHOST_HOST_EP2_CTL	Host endpoint 2 control register
USBHOST_HOST_EP2_STATUS	Host endpoint 2 status register
USBHOST_HOST_EP2_RW1_DR	Host endpoint 2 data 1-byte register
USBHOST_HOST_EP2_RW2_DR	Host endpoint 2 data 2-byte register
USBHOST_HOST_LVL1_SEL	Host interrupt level 1 selection register
USBHOST_HOST_LVL2_SEL	Host interrupt level 2 selection register
USBHOST_INTR_USBHOST_CAUSE_HI	Interrupt USB host cause high register
USBHOST_INTR_USBHOST_CAUSE_MED	Interrupt USB host cause medium register
USBHOST_INTR_USBHOST_CAUSE_LO	Interrupt USB host cause low register
USBHOST_INTR_HOST_EP_CAUSE_HI	Interrupt USB host endpoint cause high register
USBHOST_INTR_HOST_EP_CAUSE_MED	Interrupt USB host endpoint cause medium register
USBHOST_INTR_HOST_EP_CAUSE_LOW	Interrupt USB host endpoint cause low register
USBHOST_INTR_USBHOST	Interrupt USB host register
USBHOST_INTR_USBHOST_SET	Interrupt USB host set register
USBHOST_INTR_USBHOST_MASK	Interrupt USB host mask register
USBHOST_INTR_USBHOST_MAKSED	Interrupt USB host masked register
USBHOST_INTR_HOST_EP	Interrupt USB host endpoint register
USBHOST_INTR_HOST_EP_SET	Interrupt USB host endpoint set register
USBHOST_INTR_HOST_EP_MASK	Interrupt USB host endpoint mask register
USBHOST_INTR_HOST_EP_MASKED	Interrupt USB host endpoint masked register
USBHOST_HOST_DMA_ENBL	Host DMA enable register
USBHOST_HOST_EP1_BLK	Host endpoint 1 block register
USBHOST_HOST_EP2_BLK	Host endpoint 2 block register

30 Coordinate rotation digital computer (CORDIC)

30.1 Features

CORDIC is a peripheral module to support the CPU with hardware calculations for trigonometric, hyperbolic, complex numbers, and related functions. The CPU can access CORDIC through a peripheral AHB bus. The completion of an operation is signaled via status bits, an interrupt request line, or a trigger done line.

The CORDIC algorithm is a useful convergence method. It allows the performance of vector rotation not only in the Euclidean plane but also in the hyperbolic planes. The CORDIC algorithm is an iterative process where truncation errors are inherent. Higher accuracy is achieved with a greater number of iterations per calculation. The CORDIC iteration count is configurable and is not fixed at 28 iterations. The number of iterations is selected through the **N_ITER** field in the **MXCORDIC_1_1_CON** register, together with a kernel data width of at least 32 bits. The main advantage of using this algorithm is its fast calculation speed compared to software and high accuracy. The generalized CORDIC algorithm has the following CORDIC equations:

Table 318 CORDIC algorithm equations

Equations	Label
$x_{n+1} = x_n - m \cdot d_n y_n 2^{-n}$	Eq.1
$y_{n+1} = y_n + d_n x_n 2^{-n}$	Eq.2
$z_{n+1} = z_n - d_n e_n$	Eq.3

The m factor controls the vector rotation and selects the set of angles for the circular and hyperbolic functions. The n factor is the CORDIC calculation's iterative number.

m = 1: Circular mode with $e_n = \text{atan}(2^{-n})$

m = -1: Hyperbolic mode with $e_n = \text{atanh}(2^{-n})$

CORDIC feature list:

- Modes of operation
  - Supports all CORDIC operating modes for solving circular (trigonometric) and hyperbolic functions
  - Integrated look-up tables (LUTs) for all operating modes
- Circular vectoring mode: Extended support for values of initial X and Y data up to the full range of $[-2^{23}, (2^{23}-1)]$ for solving angle and magnitude
- Circular rotation mode: Extended support for values of initial Z data up to the full range of $[-2^{23}, (2^{23}-1)]$, representing angles in the range $[-\pi, ((2^{23}-1)/2^{23})\pi]$ for solving trigonometry
- 24-bit accessible data width
  - 32-bit kernel data width plus 2 sign extension bits and 1 overflow bit for X and Y each
  - 29-bit kernel data width plus 1 sign extension bit and 1 overflow bit for Z
  - KEEP bit to retain the last value in the MXCORDIC_1_0_KEEP register for a new calculation
- The CORDIC iteration count is configurable and is not fixed at 28 iterations. It can be selected in steps of 4, up to 28 iterations per calculation (up to 27 iterations in hyperbolic mode), via the **N_ITER** field in the **MXCORDIC_1_1_CON** register: 4 iterations per clock cycle, resulting in 7 clock cycles per calculation, plus 3 clock cycles from a set of start (ST) bits to a set of end-of-calculation flags, excluding the time taken for write and read access of data bytes
- Two's complement data processing
  - X and Y data are generally accepted as integer or rational numbers; X and Y must be of the same data form
  - Only exception: X result data with user-selectable option for unsigned result

- Truncation error
  - The result of a CORDIC calculation may return an approximation due to the truncation of LSBs
  - Good accuracy of the CORDIC calculated result data, especially in circular mode
- Interrupt
  - On completion of a calculation
  - Interrupt enabling and corresponding flag

30.2 Architecture

30.2.1 Block diagram

The CORDIC block diagram consists of system interfaces, MMIO, and the CORDIC kernel.

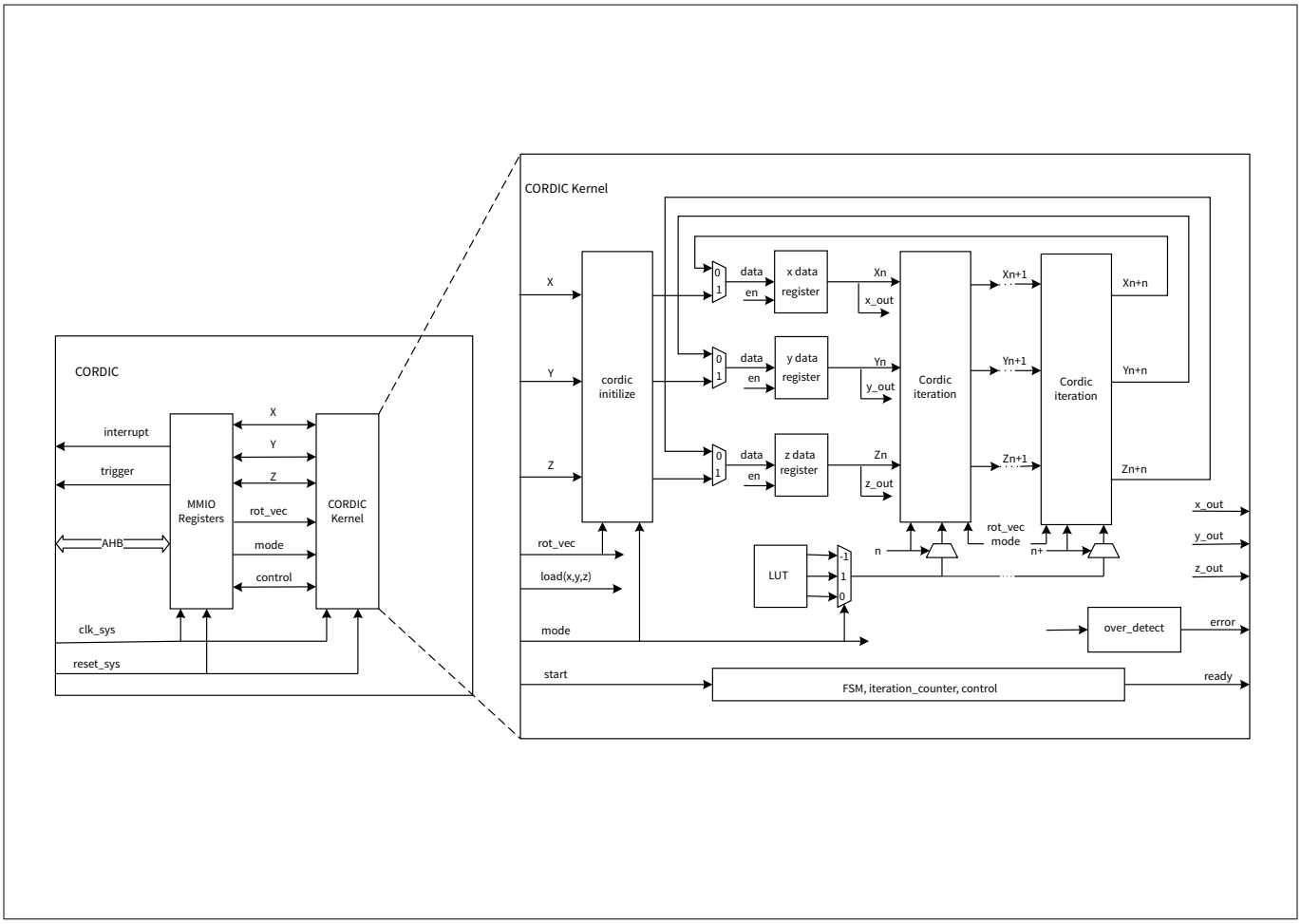


Figure 381 CORDIC block diagram

System interfaces: CORDIC has an interrupt and a trigger interface to communicate with the CPU. It sends interrupt requests to the CPU, usually to signal a completed job or an error during execution. It sends trigger requests to the DW to signal a completed job. The DW can be used to configure and start the CORDIC and to read back the results.

MMIO: The MMIO block contains all control, status, and data registers. The MMIOs are accessed by the AHB slave interface.

CORDIC kernel: It implements the iterative CORDIC algorithm. The X, Y, and Z data registers are configured with initial values via the MMIO interface. When the calculation starts, those registers hold the intermediate results

calculated by the CORDIC_ITERATION blocks. The CORDIC_ITERATION block implements one iteration of the CORDIC algorithm using combination shifters and adders. Four CORDIC_ITERATIONs are instantiated in sequence. After seven clock cycles, the final results (up to 28 iterations) are available in the X, Y, and Z data registers. The sequence is controlled by a finite state machine (FSM) and an iteration counter. The look-up tables (LUT) prove the required constants for the CORDIC algorithm in circular and hyperbolic modes.

30.3 Functional description

This section describes the function of this block.

30.3.1 Operation

The CORDIC coprocessor can be used for circular (trigonometric) and hyperbolic functions in either rotation or vectoring mode. The modes are selectable by software via the MXCORDIC_1_0_CON control register. Initialization of the kernel data register is enabled by clearing the respective KEEPx bits of the register MXCORDIC_1_0_KEEP. If MXCORDIC_1_0_CON.ST_MODE = 1, writing '1' to MXCORDIC_1_0_START_CMD.ST starts a new calculation. Otherwise, by default, if MXCORDIC_1_0_CON.ST_MODE = 0, a new calculation starts after a write access to register MXCORDIC_1_0_CORDZ. The CORDIC iteration count is configurable and is not fixed at 28 iterations. The number of iterations is selected through the **N_ITER** field in the **MXCORDIC_1_1_CON** register. Bit MXCORDIC_1_0_STAT.BSY is set while a calculation is in progress to indicate busy status. At the end of a calculation, hardware clears it. The MXCORDIC_1_0_START_CMD.ST bit is cleared immediately after a new CORDIC calculation is started.

As the first step in starting a CORDIC calculation (the initial data is loaded from the data registers when KEEP is 0 and from the result registers when KEEP is 1), the initial data is loaded from the data registers CORDx to the internal kernel data registers. During the calculation, the kernel data registers always hold the latest intermediate data. On completion of the calculation, the result data is loaded into the result registers CORRx.

The data registers CORDx function as shadow registers, which can be written to without affecting an ongoing calculation. Values are transferred to the kernel data registers only on a valid setting of the MXCORDIC_1_0_START_CMD.ST bit or if MXCORDIC_1_0_CON.ST_MODE = 0 after write access to the X data register CORDX (provided the KEEP bit of the corresponding data is not set). The result data can be read from the result registers CORRx at the end of the calculation (BSY is no longer active). The result data must be read before the start of the next calculation. Any read access on the result registers CORRx while MXCORDIC_1_0_STAT.BSY is set (and the kernel is still running a calculation) causes the kernel to issue a bus wait until MXCORDIC_1_0_STAT.BSY is reset. This ensures that any read access on the result registers CORRx returns a valid result.

At the end of each calculation, MXCORDIC_1_0_STAT.BSY returns to 0. The MXCORDIC_1_0_INTR.CDEOC bit, which denotes the CORDIC end-of-calculation, is set. The interrupt request signal is activated if the interrupt is enabled by MXCORDIC_1_0_INTR_MASK.CDEOC = 1. The MXCORDIC_1_0_INTR.CDEOC flag must be cleared manually via software by writing MXCORDIC_1_0_INTR.CDEOC = 1. The result data in X, Y, and Z are internally checked, and in cases of data overflow, the MXCORDIC_1_0_INTR.CDERR bit is set. An interrupt request signal is activated if the interrupt is enabled by MXCORDIC_1_0_INTR_MASK.CDERR = 1. The MXCORDIC_1_0_INTR.CDERR flag must be cleared manually via software by writing MXCORDIC_1_0_INTR.CDERR = 1.

At the end of each calculation, the hardware trigger is set to ‘1’ for two clock cycles. This signal can be used to trigger the DW to read back the CORDIC results.

Setting the bit MXCORDIC_1_0_START_CMD.ST during an ongoing calculation (if MXCORDIC_1_0_CON.ST_MODE = 1) while MXCORDIC_1_0_STAT.BSY is set has no effect. To start a new calculation, bit MXCORDIC_1_0_START_CMD.ST must be set again at a later time when MXCORDIC_1_0_STAT.BSY is no longer active. In the same manner, changing the operating mode during a running calculation (as indicated by MXCORDIC_1_0_STAT.BSY) has no effect.

In automatic start mode (MXCORDIC_1_0_CON.ST_MODE = 0), if a write access to CORDX occurs while MXCORDIC_1_0_STAT.BSY is set, the new calculation is not started, similar to the case where MXCORDIC_1_0_CON.ST_MODE = 1.

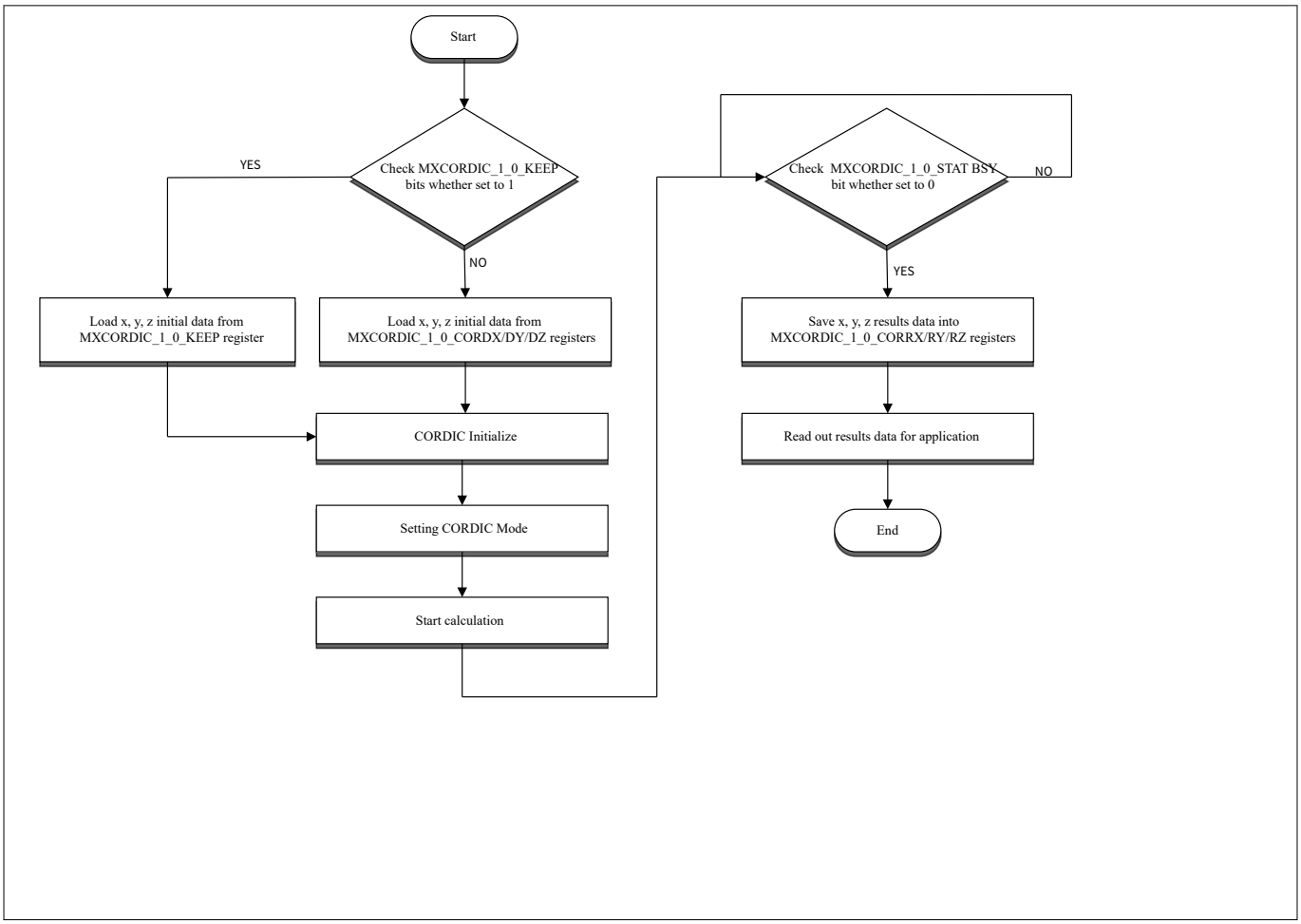


Figure 382 CORDIC operation

30.3.2 Normalized result data

In all operating modes, the CORDIC coprocessor returns normalized result data for X and Y, as shown in the following equation:

X or Y Result Data = (CORDIC Calculated Data)/MPS

***Note**: MPS is the Magnitude PreScaler.*

On the other hand, the interpretation of the Z result data differs, which is also dependent on the CORDIC function used.

For **circular** and **hyperbolic** functions, the accessible Z result data is a normalized integer value; angles in the range $[-\pi, ((2^{23}-1)/2^{23})\pi]$ are represented by $[-2^{23},(2^{23}-1)]$. The CORDIC coprocessor expects the Z data to be interpreted with this scaling:

Input Z Initial Data = Real Z Initial Value (in radians) · 0x800000/π

Real Z Result Value (in radians) = Z Result Data · π/0x800000

The CORDIC calculated data includes an inherent gain factor, K, resulting from the rotation or vectoring. The value K is different for each CORDIC function, as shown in the following table:

Table 319 Approximated CORDIC Gain K

Function	Approximated Gain K
Circular	1.646760258121
Hyperbolic	0.828159360960

30.3.3 Operating modes

The following figure gives an overview of the CORDIC coprocessor operating modes. In this table, X, Y, and Z represent the initial data.

	Rotation Mode	Vectoring Mode
	$d_n = sign(z_n), z_n \rightarrow 0$	$d_n = -sign(y_n), y_n \rightarrow 0$
Circular m = 1 $e_n = \text{atan}(2^{-n})$ n = 0, 1, 2, …, N $K \approx 1.64676$	x → $K[xcos(z) - ysin(z)]$ y → $K[ycos(z) + xsin(z)]$ z → 0 For $\cos(v)$r and $sin(v)$, set $x = 1/K$, $y = 0$ $\tan(v) = \sin(v)/\cos(v)$	x → $K\sqrt{x^2 + y^2}$ y → 0 z → $z + \text{atan}(y/x)$ For $\text{atan}(v)$, set $x = 1, z = 0$ $\text{acos}(w) = \text{atan}\left[\sqrt{1 - w^2}/w\right]$ $\text{asin}(w) = \text{atan}\left[w/\sqrt{1 - w^2}\right]$
Hyperbolic m = -1 $e_n = \text{atanh}(2^{-n})$ n = 1, 2, 3, 4, 4, 5, …13, 13, …, N $K' \approx 0.82815$	x → $K'[xcosh(z) - ysinh(z)]$ y → $K'[ycosh(z) + xsinh(z)]$ z → 0 For $\cosh(v)$ and $\sinh(v)$, set $x = 1/K', y = 0$ $\tanh(v) = \frac{\sinh(v)}{\cosh(v)}$ $e^v = \sinh(v) + \cosh(v)$ $w^t = e^{tln(w)}$	x → $K'\sqrt{x^2 - y^2}$ y → 0 z → $z + \text{atanh}(y/x)$ For $\text{atanh}(w)$, set $x = 1, z = 0$ $\ln(w) = 2\,\text{atanh}\left(\frac{w-1}{w+1}\right)$ $\sqrt{w} = \sqrt{(w + 0.25)^2 + (w - 0.25)^2}$ $\text{acosh}(w) = \ln\left(w + \sqrt{1 - w^2}\right)$ $\text{asinh}(w) = \ln(w + \sqrt{1 + w^2})$

Figure 383 CORDIC operation modes

Usage notes:

- For solving the respective functions, initialize the CORDIC data (X, Y, and Z) with meaningful initial values within the domain of convergence to ensure result convergence. The ‘useful domain’ listed in the table covers the supported domain of convergence for the CORDIC algorithm and excludes the non-meaningful range(s) for the function
- Function limitations must be considered; for example, setting initial X = 0 for atan(Y/X) is not meaningful. Violations of such function limitations may yield incoherent CORDIC result data
- All data inputs are processed and handled as two's complements. The only exception is the user-option for X result data (only) to be read as an unsigned value

- The only case where the result data is always positive and larger than the initial data is the X result data (only) in the circular vectoring mode; therefore, the user may prefer to use the MSB bit as a data bit instead of a sign bit. By setting X_USIGN = 1, the X result data is processed as unsigned data
- For circular and hyperbolic functions, and due to the corresponding fixed LUT, the Z data is always handled as a signed integer Q(1,28), accessible as Q(1,23). The LUTs contain scaled integer values Q(1,28) of atan(2^{-n}) for n = 0, 1, 2, ..., 27 and atanh(2^{-n}) for n = 1, 2, ..., 27, such that angles in the range [-π,((2^{23}-1)/2^{23})π] are represented by integer values ranging [-1,(2^{23}-1)/2^{23}]. Therefore, Z data is limited (not considering the domain of convergence) to represent angles [-π, ((2^{23}-1)/2^{23})π] for these CORDIC functions. Any calculated value of Z outside of this range will result in an overflow error
- The MPS setting has no effect on Z data. Ensure proper initialization of Z initial data to prevent overflow and incorrect result data
- The CORDIC coprocessor is designed such that, with the correct user setting of MPS > 1, there is no internal overflow of the X and Y data, and the read result data is complete. However, note that in these cases, the higher the MPS setting, the lower the resolution of the result data due to the loss of LSB bit(s)
- The hyperbolic rotation mode is limited in terms of result accuracy since the initial Y data must be set to zero. In other words, the CORDIC coprocessor cannot return an accurate result for cosh(Z)+/-sinh(Z) in a single calculation

30.3.4 Domains of convergence

This section does not cover the theory of the inherent limits of CORDIC convergence, which vary with different CORDIC functions in rotation or vectoring mode. For the convergence of result data, there are limitations to the magnitude or value of the initial data and the corresponding useful data form, depending on the operating mode used. The following are generally applicable regarding the convergence of CORDIC result data.

Rotation mode: Z data must converge towards 0. The initial Z data must be equal to or smaller than $\Sigma d_n \cdot e_n$, where e_n is always decreasing for iteration n. In other words, |Z| ≤ Sum of LUT. In a circular function, this means the |Z| ≤ integer value represents 1.74 radians. In a hyperbolic function, the |Z| ≤ integer value represents 1.11 radians.

Vectoring mode: Y data must converge towards 0. The values of initial X and Y are limited by the Z function, which is dependent on the corresponding LUT. In a circular function, |atan(Y/X)| ≤ 1.74 radians. In a hyperbolic function, |atanh(Y/X)| ≤ 1.11 radians. In vectoring mode, the additional requirement is that X > 0.

Circular rotation mode: The full range of Z input [-2^{23}, (2^{23}-1)] representing angles [-π, ((2^{23}-1)/2^{23})π] is supported. There are no limitations on initial X and Y inputs, except for overflow considerations, which can be overcome with MPS settings.

Circular vectoring mode: The full range of X and Y inputs [-2^{23},(2^{23}-1)] are supported, while the Z initial value should satisfy |Z| ≤ π/2 to prevent possible Z result data overflow.

While the operating modes of the CORDIC coprocessor are generally bound by these convergence limits, there are exceptions for the circular rotation and circular vectoring modes, which use additional pre- and post-processing logic to support a wider range of inputs.

***Note**: Considerations should also be given to function limitations, such as the meaning of the result data (for example, dividing by zero is not meaningful). The 'useful domain', included within Figure 383 for each of the main functions, attempts to cover both CORDIC convergence and the useful range of the function.*

***Note**: Input values may be within the domain of convergence; however, this does not guarantee a fixed level of accuracy of the CORDIC result data.*

30.3.5 Overflow considerations

Besides the domain of convergence, consider the limitations on the magnitude of input data to prevent result data overflow.

In all operating modes, the CORDIC coprocessor handles data overflow in the same way. Prevent overflow for X and Y data by setting the MPS bit correctly; the bit's value is partly based on the CORDIC coprocessor operating mode and the application data.

The MPS setting has no effect on the Z data. For circular and hyperbolic functions, any value of Z outside of the range $[-\pi, ((2^{23}-1)/2^{23})\pi]$ cannot be represented and results in a Z data overflow error. Note that kernel data Z has values in the range $[-\pi, ((2^{23}-1)/2^{23})\pi]$ scaled to the range $[-2^{23}, (2^{23}-1)]$, so the written and read values of Z data are always normalized. For a linear function where Z is a real value, the magnitude of Z must not exceed 4 integer bits.

The CORDIC coprocessor accepts (initial) data X, Y, and Z inputs in two's complement format. The result data is also in two's complement format. The only exception is for the X result data in circular vectoring mode. The X result data has a default data format of two's complement, but the user can select via bit CON.X_USIGN = 1 for the X result data to be read as an unsigned value. This option prevents a potential overflow of the X result data (taken together with the MPS setting), as the MSB bit is now a data bit. Note that setting bit CON.X_USIGN = 1 is only effective when operating in the circular vectoring mode, which always yields result data that is positive and larger than the initial data.

30.3.6 Data format

The CORDIC coprocessor accepts initial data (24-bit) X, Y, and Z inputs in two's complement format. The result data (24-bit) is also in two's complement format. The only exception is for the X result data in circular vectoring mode. The X result data has a default data format of two's complement, but the user can select via bit CON.X_USIGN = 1 for the X result data to be read as an unsigned value. This option prevents a potential overflow of the X result data (taken together with the MPS setting), as the MSB bit is now a data bit. Note that setting bit CON.X_USIGN = 1 is only effective when operating in the circular vectoring mode, which always yields result data that is positive and larger than the initial data.

***Note**: The X, Y, and Z initial data registers (MXCORDIC_1_0_CORDX, MXCORDIC_1_0_CORDY, and MXCORDIC_1_0_CORDZ) and result data registers (MXCORDIC_1_0_CORRX, MXCORDIC_1_0_CORRY, and MXCORDIC_1_0_CORRZ) all use the bit[31:8] to save data, and bit[7:0] is reserved.*

Generally, the input data for X and Y can be an integer or a rational number (fraction). However, in any calculation, the data form must be the same for both X and Y. Also, in the case of fractions, X and Y must have the same number of bits for decimal places. The Z data is always handled as an integer based on the normalization factor for a circular or hyperbolic function.

See the Normalized result data section for details on data normalization.

30.3.7 Performance

The CORDIC calculation time increases linearly with increased precision. Increased precision is achieved with a greater number of iterations, which requires an increased width of the data parameters. The CORDIC coprocessor uses barrel shifters for data shifting. The total time from the start of the calculation to the instant the EVFR.CDEOC flag is set is $N/4 + 3$ kernel clock cycles, where N is the number of iterations, the CORDIC iteration count is configurable and is not fixed at 28 iterations. The number of iterations is selected through the **N_ITER** field in the **MXCORDIC_1_1_CON** register (4, 8, 12, 16, 20, 24, or 28) and is not fixed; 28 is the maximum. At the maximum of 28 iterations, this equals 28/4+3 = 10 kernel clock cycles. Note that the EVFR.CDERR flag is valid only for one kernel clock cycle after the onset of EVFR.CDEOC. This timing for one complete calculation is applicable to those modes that involve additional data processing.

***Note**: In hyperbolic modes, the iterations start from n = 1, and the iterations n = 4 and 13 must be repeated to obtain convergence. As a result, a total of n = 27 iterations are executed, except n = 28.*

***Note**: The above timing excludes the time taken for software loading of initial data and reading of the final result data to and from the six data registers.*

30.3.8 Look-up tables

This section describes the look-up tables used in the CORDIC coprocessor kernel. The LUTs are 29 bits wide. Each entry of the atan and atanh LUT is divided into 1 sign bit (MSB), followed by a 28-bit fractional part, Q(1,28).

The Z data is a normalized representation of the actual angle. The internal scaling is such that $[-\pi, ((2^{23}-1)/2^{23})\pi]$ is equivalent to $[-2^{28},(2^{28}-1)]$. The last 5 LSB bits are truncated, as 24-bit data is transferred to the data bus when addressed. From the user's point of view, the angles $[-\pi,((2^{23}-1)/2^{23})\pi]$ are therefore represented by the range $[-2^{23},(2^{23}-1)]$.

Table 320 Circular mode look-up table

Iteration No. n	Scaled atan(2^{-n}) in hex	Iteration No. n	Scaled atan(2^{-n}) in hex
0	4000000	14	145F
1	25C80A4	15	A30
2	13F670B	16	518
3	A2223B	17	28C
4	5161A8	18	146
5	28BAFC	19	A3
6	145EC4	20	51
7	A2F8B	21	29
8	517CA	22	14
9	28BE6	23	A
10	145F3	24	5
11	A2FA	25	3
12	517D	26	1
13	28BE	27	1

In hyperbolic mode, the iterations n = 4 and n = 13 are repeated, while the iterations n = 0 and n = 27 are skipped. In order to converge at the same number of clock cycles as the circular mode, n = 27 is skipped.

Table 321 Hyperbolic mode look-up table

Iteration No. n	Scaled atanh(2^{-n}) in hex	Iteration No. n	Scaled atanh(2^{-n}) in hex
0	-	14	145F
1	2CC2F12	15	A30
2	14D01AC	16	518

(table continues...)

Table 321 (continued) Hyperbolic mode look-up table

Iteration No. n	Scaled atanh(2^{-n}) in hex	Iteration No. n	Scaled atanh(2^{-n}) in hex
3	A3D4E0	17	28C
4	5197FC	18	146
5	28C1C7	19	A3
6	145F9D	20	51
7	A2FA6	21	29
8	517CE	22	14
9	28BE6	23	A
10	145F3	24	5
11	A2FA	25	3
12	517D	26	1
13	28BE	27	-

30.3.9 Interrupt generation

The CORDIC interrupt request signal is generated as illustrated in the following diagram:

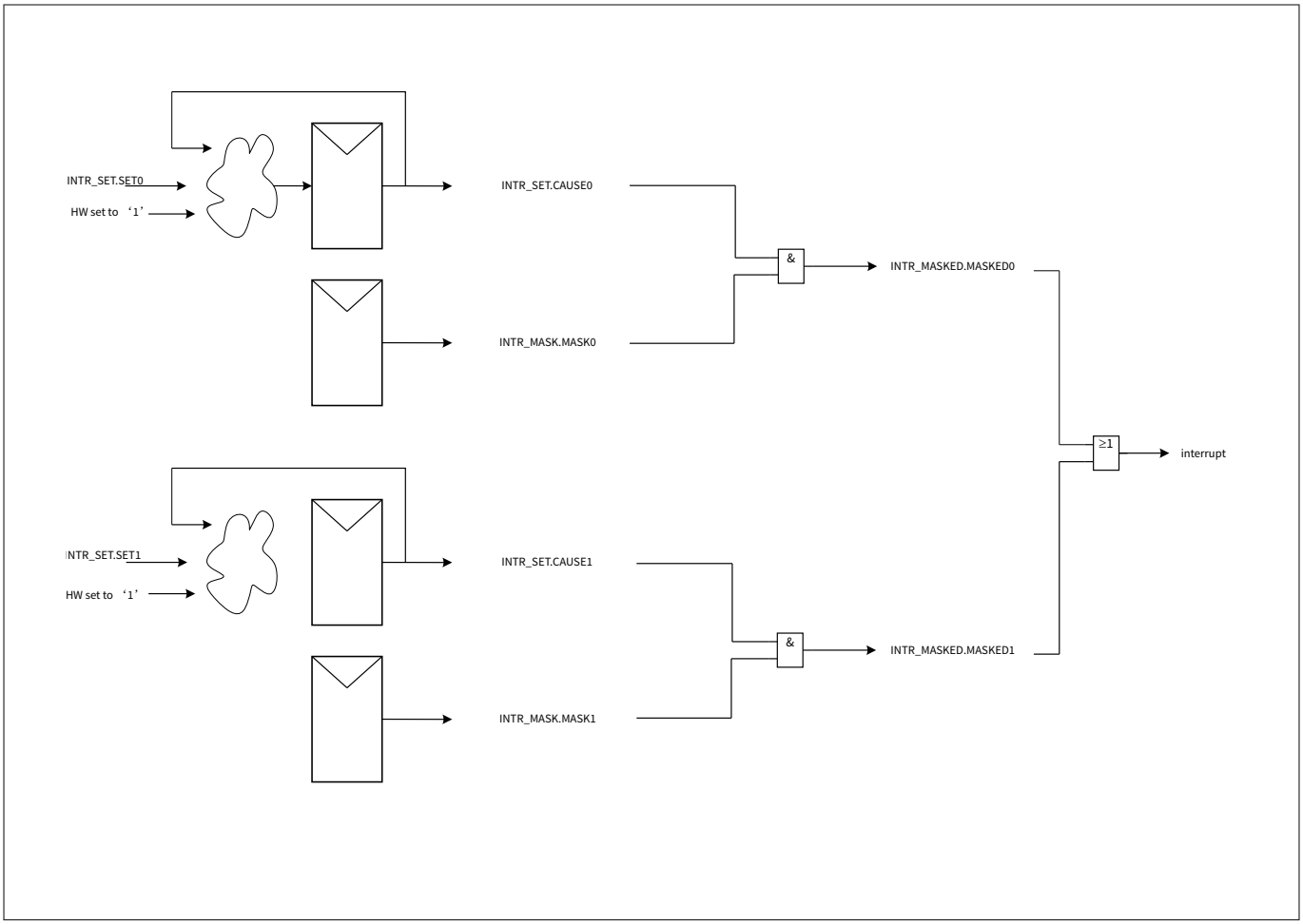


Figure 384 CORDIC interrupts generation

Hardware can set the MXCORDIC_1_0_INTR register interrupt flag bits to 1 when the event signal is detected. The MXCORDIC_1_0_INTR.CDEOC is the end of the calculation event flag. The MXCORDIC_1_0_INTR.CDERR is the calculation error event flag. Software can set the MXCORDIC_1_0_INT_SET register interrupt cause bits to 1. The MXCORDIC_1_0_INTR_MASK register can hold the interrupt mask bits of co-located interrupt cause bits in MXCORDIC_1_0_INT; the co-located interrupt cause bit only causes an interrupt when its mask bit is '1'. The MXCORDIC_1_0_INTR_MASKED register is a logical AND of the MXCORDIC_1_0_INT_INTR and MXCORDIC_1_0_INTR_MASK registers. The user can read out the interrupt request signal directly via the MXCORDIC_1_0_INTR_MASKED register.

***Note**: For the PSC3M/P6 MCU device due to faster calculation by the CORDIC hardware, the use of non-blocking function APIs may not give any speed advantages. It is suggested that the user use the blocking functions directly.*

30.4 Power, clock, and reset

This section gives an overview of the power supply, clocks, and resets used in this block.

30.4.1 Power supply

CORDIC is available only in active power mode. If the system is in DeepSleep mode, the power domain in which CORDIC is placed is powered down. All flops inside CORDIC are reset without retention. There are no power-related interfaces (P- or Q-channels) in the CORDIC. The CORDIC IP does not implement any internal power domains or power switches.

30.4.2 Clock

CORDIC runs on a single clock input (*clk_sys*). This clock is synchronous to the m33syscpuss clock. The clock can be gated centrally (in MXSPERI) by the SL_CTL.ENABLED MMIO.

30.4.3 Reset

CORDIC has one asynchronous reset signal. The reset can be asserted asynchronously, but it has to be de-asserted synchronously to the system clock. There is no reset synchronization inside this module. The reset synchronization is done inside MXSPERI.

30.5 Registers summary

Table 322 CORDIC registers summary

Register	Function
MXCORDIC_1_0_CTL	module enable/disable register
MXCORDIC_1_0_ID	module identification register
MXCORDIC_1_0_INTR	interrupt cause register
MXCORDIC_1_0_INTR_SET	interrupt set register
MXCORDIC_1_0_INTR_MASK	interrupt mask register
MXCORDIC_1_0_INTR_MASKED	Interrupt Masked Register
MXCORDIC_1_0_KEEP	CORDIC keep register

(table continues...)

Table 322 (continued) CORDIC registers summary

Register	Function
MXCORDIC_1_0_CON	CORDIC control register
MXCORDIC_1_0_CORDX	CORDIC X data register
MXCORDIC_1_0_CORDY	CORDIC Y data register
MXCORDIC_1_0_CORDZ	CORDIC Z data register
MXCORDIC_1_0_CORRX	CORDIC X result register
MXCORDIC_1_0_CORRY	CORDIC Y result register
MXCORDIC_1_0_CORRZ	CORDIC Z result register
MXCORDIC_1_0_STAT	CORDIC status register
MXCORDIC_1_0_START_CMD	CORDIC start command register

31 High-performance programmable analog subsystem (HPPASS)

The high-performance programmable analog subsystem (HPPASS) can sense voltage and convert it to a digital code, post process the ADC, and hence make first-level decisions as close to the sensor as possible. It can store the processed decision/data and autonomously stream the data to the CPU. It can also operate autonomously (independent of the CPU).

31.1 Features

The PSC3M/P6 MCU HPPASS has the following features:

- A 12-bit SAR ADC:
  - When operating in a single-channel, the maximum speed of the ADC is 9 Msps
  - 16 samplers, including 12x direct I/O connections and 4x muxed connections, which can produce simultaneous sampling of external analog signals
  - All samplers support gains of 1, 3, 6, 12
  - Up to 16 dedicated analog pads, connected to up to 16 parallel sample stages
  - 8 additional GPIOs can be used as analog inputs by AMUX
  - 2 internal voltage signals (temperature sensor and DAC output voltage) can be connected to a muxed sampler by AMUX
  - AMUX can be controlled by a SAR sequencer, firmware, or autonomous controller (AC)
  - 16 channels for dedicated analog inputs, 8 channels for additional GPIOs, 8 channels for OpAmps output, and 4 channels for internal voltage signals
  - Post processing
    - Pseudo differential conversion
    - Result signed/unsigned format
    - Two 16-tap Finite Impulse Response (FIR) filters
    - Three programmable and configurable 3-pole, 3-zero hardware filters (HWFILT3P3Z)
    - Hardware averaging (Result accumulation, no shifting function) of configurable number of samples (2, 4, 8, 16, 32, 64)
    - A 32-entry FIFO that can subdivided into 1, 2, or 4 FIFOs
    - Eight limit detection resources
  - SAR sequencer
    - Up to eight scan groups
    - Up to 16 samplers can be enabled for simultaneous sampling in a given group
    - Every group can select input for four muxed samplers
    - Every group can be triggered by firmware, trigger from other peripherals (TCPWM and GPIO) or autonomous controller (AC)
    - Every group can select one of three available programmable sample times
    - Two priorities, programmable per group
    - Every group supports continuous mode
  - Self-calibration support for offset, linearity and gain
  - Interrupt generation
    - Scan group done
    - Scan group trigger collision, scan group hold violation

        - FIR done, HWFILT3P3Z done and FIFO level
        - FIFO overflow/underflow
        - Limit detection
        - Result overflow
    - 4 FIFO level triggers, which can trigger other on-chip peripherals (such as DMA, GPIO, TCPWM)
    - On-chip temperature sensor
- Four Operational Amplifiers (OpAmps) for signal amplification and conditioning
    - Each OpAmp can be configured as:
        - Programmable gain amplifier (PGA)
        - Difference amplifier (DA)
        - Unity gain buffer (UGB)
        - Standalone operational amplifier (OpAmp)
    - Inputs to each OpAmp can be driven from analog pins, GPIOs, DAC, reference voltage or ground
    - Each OpAmp can be configured for either single-ended output or fully differential output
    - An OpAmp configured for single-ended output can drive either an internal load or an external load
    - Each OpAmp has four configuration registers that can be dynamically selected based on application requirements
- Nine comparator and slope generators (CSG)
    - Three CSG groups, with each group containing three CSG slices
    - Each CSG group can select a hysteresis magnitude
    - Each CSG group includes a 128-entry waveform look-up table (LUT)
    - Each CSG slice consists of a 10-bit 30 Msps DAC to generate the comparator reference
    - Each comparator positive input can select one of two analog inputs
    - Each comparator negative input can select DAC output or one of two analog inputs
    - DAC supports direct write mode, buffered mode, hysteretic mode, slope generation mode and LUT waveform generation mode
    - DAC start can be triggered by firmware, trigger from other peripherals or autonomous controller
    - DAC update can be triggered by firmware, trigger from other peripherals, autonomous controller or comparator output
    - Comparator post processing
        - Configurable comparator output polarity
        - DAC code change blanking
        - Blanking the comparator output
        - Deglitching the comparator output
        - Comparator output remapping
        - Generate configurable comparator interrupt and or AC trigger
        - Ability to select the comparator output or inverted comparator as an update trigger
    - DAC output can be routed to ADC by AMUX
    - Interrupt generation
        - Comparator output and remap output
        - DAC hardware started
        - DAC slope done, DAC buffer empty
- Two 12-bit R2R DACs with output buffer

- Autonomous controller (AC) is a finite-state machine (FSM) sequencer that works of a 16-entry state transition table, to transition appropriately through various pre-defined states, based on timer, and/or events. Autonomous controller can perform the following functions:
  - Enable and disable internal blocks (SAR ADC, CSG), provide a BLOCK_READY status to firmware
  - Hardware or firmware trigger to execute programmed actions
  - Trigger any of the internal blocks
  - Implement hardware delays
  - Periodic action (for example, triggering) in a timed loop
  - Receive conditions from the internal blocks and acting on those events
  - Generate an AC-based interrupt to the CPU
  - Digital output to as many as five GPIOs
- 16 input triggers, which can configure to a firmware-level trigger, a firmware pulse trigger, hardware trigger A (multiplexer trigger), or hardware trigger B (one-to-one trigger)
- 16 output level triggers, each capable of enabling the following outputs: CSG comparator output, SAR ADC limit detection output, CSG comparator remapped output, and DAC FIFO below-threshold output.
- 16 output pulse triggers, each of which can select one pulse trigger from ADC group done, FIR done, AC trigger and HWFILT3P3Z done

31.2 Architecture

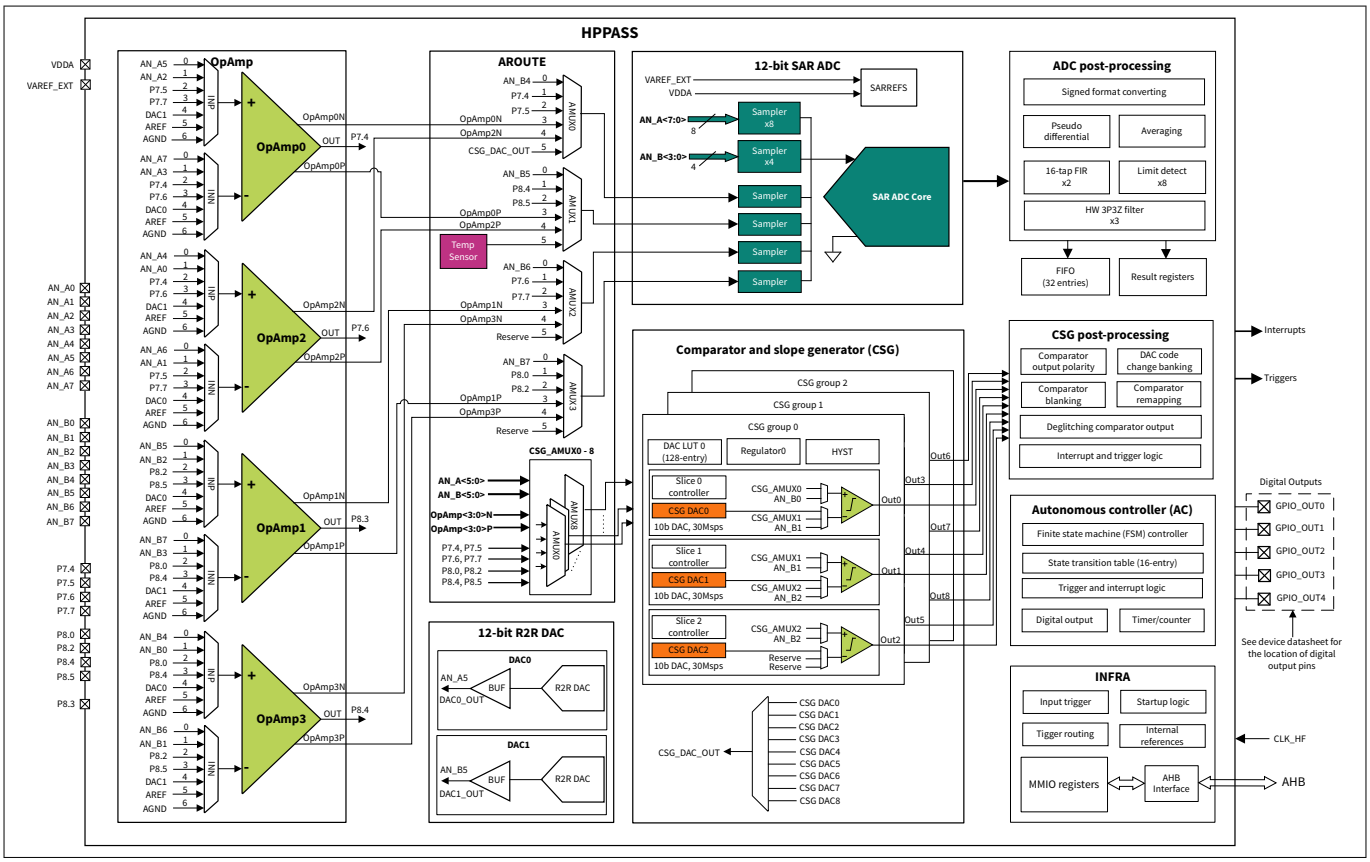


Figure 385 HPPASS block diagram

Figure 385 shows the HPPASS block diagram. The block consists of four fully differential operational amplifiers, with integrated passive components and a programmable switch network, a 12-bit SAR ADC, and a comparator and slope generator (CSG) comprising nine 10-bit 30 Msps DACs and nine high-speed comparators. The HPPASS

also includes two 12-bit R2R DACs with output buffer and an autonomous controller (AC) that provides hardware control and status without the need for CPU or external intervention. The HPPASS has 16 dedicated analog input pins (AN_A0 - AN_A7 and AN_B0 - AN_B7). In addition, the SAR ADC supports eight additional GPIO input pins via AMUXs and provides digital output through five GPIOs (see the PSC3M/P6 MCU data-sheet for details on the SAR ADC GPIO input pins and digital output pins).

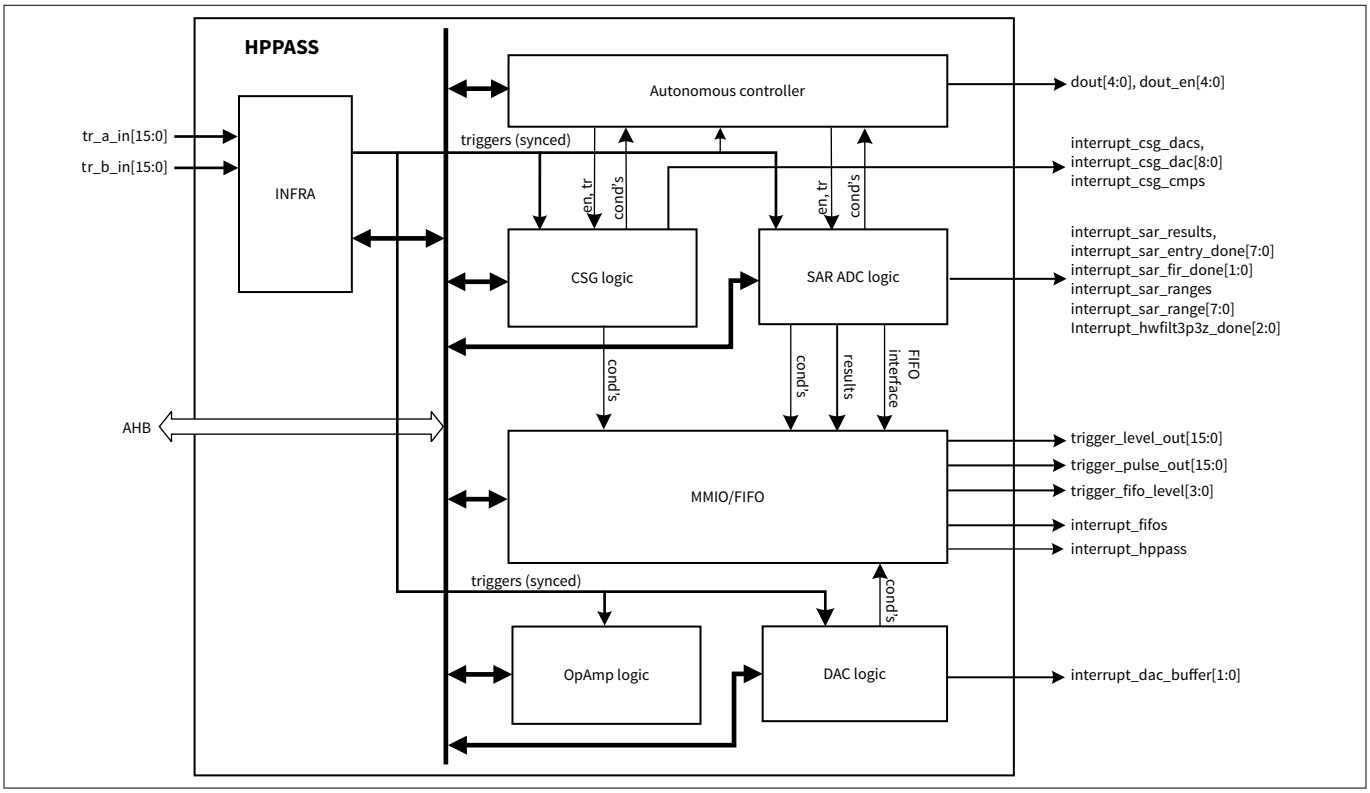


Figure 386 HPPASS digital top-level diagram

The key interfaces for the industrial application are triggers and interrupts; Figure 386 shows the top-level structure of the HPPASS digital system with a simplified view of the connectivity.

Triggers

Input triggers from the Peri can be level (for comparator blanking) or pulse (for SAR ADC sequencer and CSG DAC updates). Output triggers include level-based FIFO triggers for DMA, and also the typical pulse triggers, including comparator state and SAR ADC limit detect. In addition, there are level-based dedicated, low-latency triggers from comparator to Peri for TCPWM KILL.

There are 32 input triggers (“A” and “B”). By default, these inputs are considered asynchronous and are double synced; however, there is an option for single sync. See the Input trigger chapter for the details.

The HPPASS to TCPWM triggers are level-based and include comparator output and SAR ADC limit detect signals. This has full permutability so any of these signals can be routed and ORed into any of these tr_level_out connections. See the Triggers section for the details of output triggers.

The HPPASS to DMA triggers are pulse based which can copy SAR ADC results through DMA to system memory.

The HPPASS FIFO has level-based DMA triggers, which reflect the configured number of results available in the FIFO.

Interrupts

There are both individual and combined interrupts. When using a combined interrupt, read the associated INTR register to know which event caused the interrupt. There are interrupts for DAC buffering and slope generation, as well as the comparator. Interrupts for the SAR ADC include availability of SAR results and limit detection

status. Error conditions are grouped into one general interrupt. See the SAR ADC interrupts chapter and CSG interrupts chapter for the details.

The following sections describe the key-sub blocks details of HPPASS, including infrastructure (INFRA), Operational Amplifiers (OpAmps), SAR ADC, comparator and slope generator (CSG), digital-to-analog converter (DAC), autonomous controller (AC), and temperature sensor.

31.2.1 Infrastructure (INFRA)

The following figure shows the block diagram of the HPPASS infrastructure.

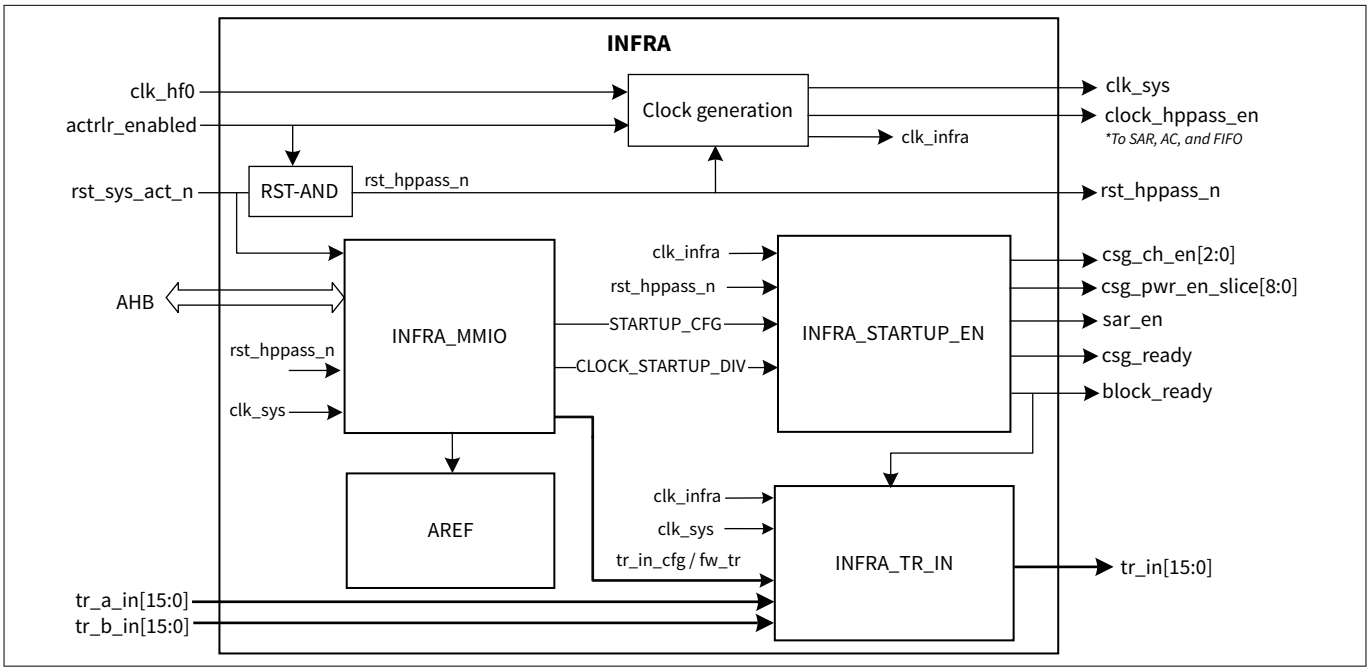


Figure 387 HPPASS INFRA block diagram

The infrastructure contains four main blocks as listed below, and generates clock (clk_sys and clock_hppass_en which is a fixed 1/2 from clk_hf0) and reset (rst_hppass_n) for internal use. For more details on the HPPASS clock, refer to the Clock chapter.

- INFRA_STARTUP_EN: Startup logic to control when to enable SAR and provide the startup sequence for the CSG sub-block
- INFRA_TR_IN: Input triggering structure. Both hardware and firmware trigger are supported
- AREF: Analog reference block that includes a local band-gap to support accurate voltage and current references for HPPASS internal analog circuits
- INFRA_MMIO: AHB interface and registers for AREF control, startup logic, and input triggering control

31.2.1.1 Startup

The INFRA supports SAR and CSG blocks at startup. Both SAR and CSG blocks wait for AREF to be ready before they are enabled and after that, they are independent. The startup sequence is timed by the INFRA startup timer.

AREF is enabled by setting the ENABLED bit of the HPPASS_INFRA_AREFv2_AREF_CTRL register to '1'.

The global enable for the HPPASS is the ENABLED bit of the HPPASS_ACTRLR_CTRL register. When this bit is 0, all analog blocks are powered down, mission logic and status is held in reset, and mission clocks are gated off. The only functionality available is the AHB bus; note that the SAR ADC calibration registers are not writable when the HPPASS is disabled.

Startup is initiated by enabling the AC and running the AC startup instruction(s), provided AREF_EN is enabled prior to AC run. For more details, see the Initialization chapter of AC. At least one AC instruction is required to enable the blocks required for a given application, along with the WAIT_FOR action of the BLOCK_READY condition. See the Autonomous controller (AC) chapter for more details of AC instruction.

The startup block has divided clock, FSM, counters and other logic to time the HPPASS startup, and it generates the BLOCK_READY condition back to the AC. You can either poll for this condition in the HPPASS_ACTRLR_BLOCK_STATUS register, or wait for an AC interrupt before starting the application.

Power on reset (POR)

At POR, AREF needs up to 40 µs startup time. If AREF is already enabled for some time, you can optimize the startup time by changing the COUNT_VAL bit-fields of the HPPASS_INFRA_STARTUP_CFGx register setting.

After AREF is ready, CSG_CH_EN and SAR_EN are enabled. After 10 µs, CSG_PWR_EN_SLICE is enabled. Then, after 32 cycles of clk_csg CSG_READY (CSG_OUT data is valid) is set. The timing delay from CSG_PWR_EN_SLICE to CSG_READY depends on the clk_csg frequency. The following diagram shows the startup flow.

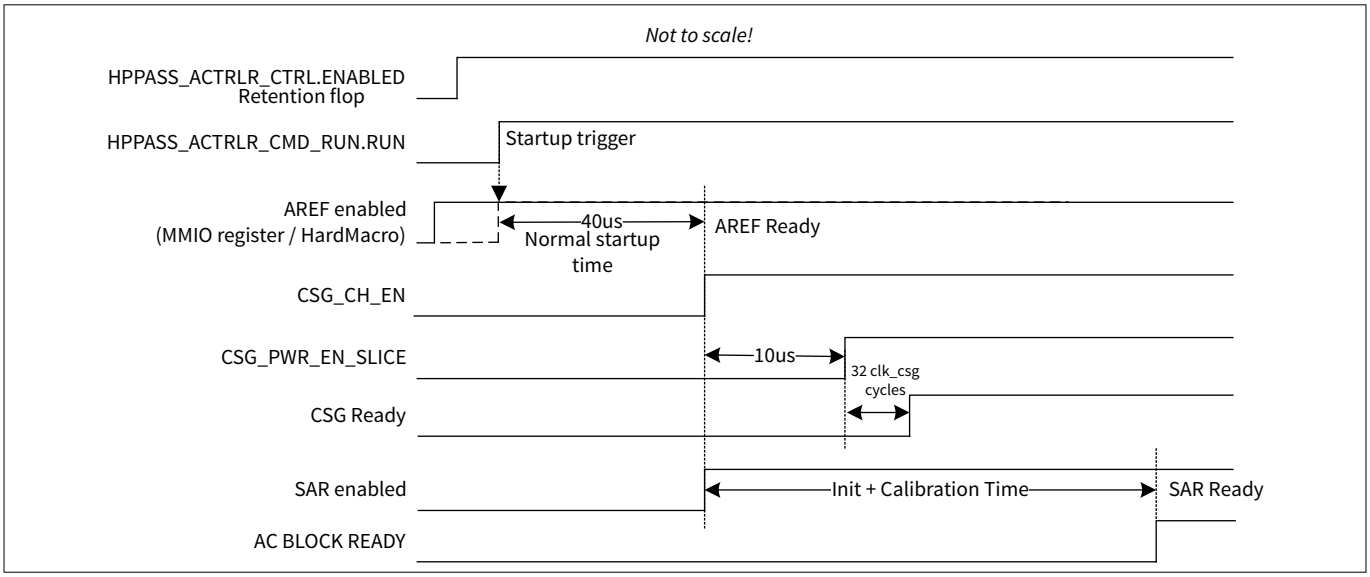


Figure 388 Startup at POR

This typical timing sequence as shown in the above figure can be achieved by programming HPPASS_INFRA_CLOCK_STARTUP_DIV and HPPASS_INFRA_STARTUP_CFG0/1/2/3 registers as shown below. There are a total of 4 sets of HPPASS_INFRA_STARTUP_CFGx (x = 0 to 3) which can support up to four timing delays. In PSC3M/P6 MCU only three registers are used, as shown below.

***Note**: The above timing sequence is critical and should not be modified by the user.*

Startup timer is a divided clock from clk_infra running at 90 MHz when CLK_HF0 is 180 MHz. By default, the DIV_VAL bit-fields of HPPASS_INFRA_CLOCK_STARTUP_DIV register is 7 and it can also be changed by this register.

SAR ADC supports factory-calibration and self-calibration. Infineon recommends using factory calibration because it is faster than self-calibration. For more details, see chapter Calibration. When SAR ADC self-calibration is enabled, the longest delay at POR is for SAR ADC calibration. The length depends on the type of self-calibration, and SAR input clock frequency, please check PSC3M/P6 MCU datasheet to know the maximum time.

Following is the recommended program for the HPPASS startup sequence:

1. Configure the startup clock divider resolution to 200 ns from 90 MHz HPPASS INFRA clock. Calculated DIV_VAL=(200 ns/11.11 ns)-1 = 17 (11_H)

```
HPPASS_INFRA_CLOCK_STARTUP_DIV.DIV_VAL[7:0] = 11H
```

2. Configure HPPASS_INFRA_STARTUP_CFG0 to enable 40 us timer from AREF_EN to SAR_EN(bit 8) and CSG_CH_EN(bit 9). Calculated COUNT_VAL= (40 us/200 ns)-1 =199 ($C7_H$)

```
HPPASS_INFRA_STARTUP_CFG0.COUNT_VAL[7:0] = C7H
HPPASS_INFRA_STARTUP_CFG0.SAR_EN = 1'b1
HPPASS_INFRA_STARTUP_CFG0.CSG_CH_EN = 1'b1
```

3. Configure HPPASS_INFRA_STARTUP_CFG1 to enable 10 us timer from CSG_CH_EN to CSG_PWR_EN_SLICE(bit 10). Calculated COUNT_VAL = (10 us/200 ns)-1 = 49(31_H)

```
HPPASS_INFRA_STARTUP_CFG1.COUNT_VAL[7:0] = 31H
HPPASS_INFRA_STARTUP_CFG1.CSG_PWR_EN_SLICE = 1'b1
```

4. Configure HPPASS_INFRA_STARTUP_CFG2 to enable 400 ns timer from CSG_PWR_EN_SLICE to CSG_READY(bit 11). Assuming clk_csg is running at 90 MHz. 32 cycles of clk_csg is 355.56 ns. Calculated COUNT_VAL = (355.56 ns/200 ns)-1 = 0

```
HPPASS_INFRA_STARTUP_CFG2.COUNT_VAL[7:0] = 0H
HPPASS_INFRA_STARTUP_CFG2.CSG_READY = 1'b1
```

Wake-up from DeepSleep

To reduce SAR ADC calibration wait time when waking up from DeepSleep mode, the users should read back and store the SAR ADC calibration values to Deep Sleep capable RAM before entering Deep Sleep.

As shown in the diagram below, on wake-up the user can write these values back into the SAR ADC calibration registers, thereby bypassing the need for a SAR calibration. The start-up time is significantly reduced because the SAR ADC only has a nominal initialization time, which includes initial sampling.

***Note**: Infineon's Peripheral Driver Library (PDL) function `Cy_HPPASS_DeepSleepCallback()` does all the action required to save or restore the calibration values. For more information, see Peripheral Driver Library API Reference Manual.*

The ENABLED bit of the HPPASS_INFRA_AREFv2_AREF_CTRL register is a retention bit. During DeepSleep, this bit is still enabled, but it will get isolated and forced to "0" to disable the AREF. Upon wake-up, AREF enable will be re-enabled again automatically and AREF will be ready after 40 us. During Deep Sleep-RAM/OFF, that register bit will be disabled. User needs to set ENABLED bit of HPPASS_INFRA_AREFv2_AREF_CTRL register again to enable AREF upon wake-up.

Generally, in any DeepSleep mode operation, the AREF is OFF. The AREF start-up time is the same as POR which is 40 **µs**. If AREF has already been enabled for a while and the DeepSleep duration is known, users can reduce the start-up time by changing COUNT_VAL bit-fields of HPPASS_INFRA_STARTUP_CFG0 register setting.

The ENABLED bit of HPPASS_ACTRLR_CTRL register is a retention flop. Upon wake-up, AC is enabled automatically and you only need to set RUN bit of HPPASS_ACTRLR_CMD_RUN register to instruct AC to run. The rest of the timing sequence will be the same as POR except for SAR ready timing.

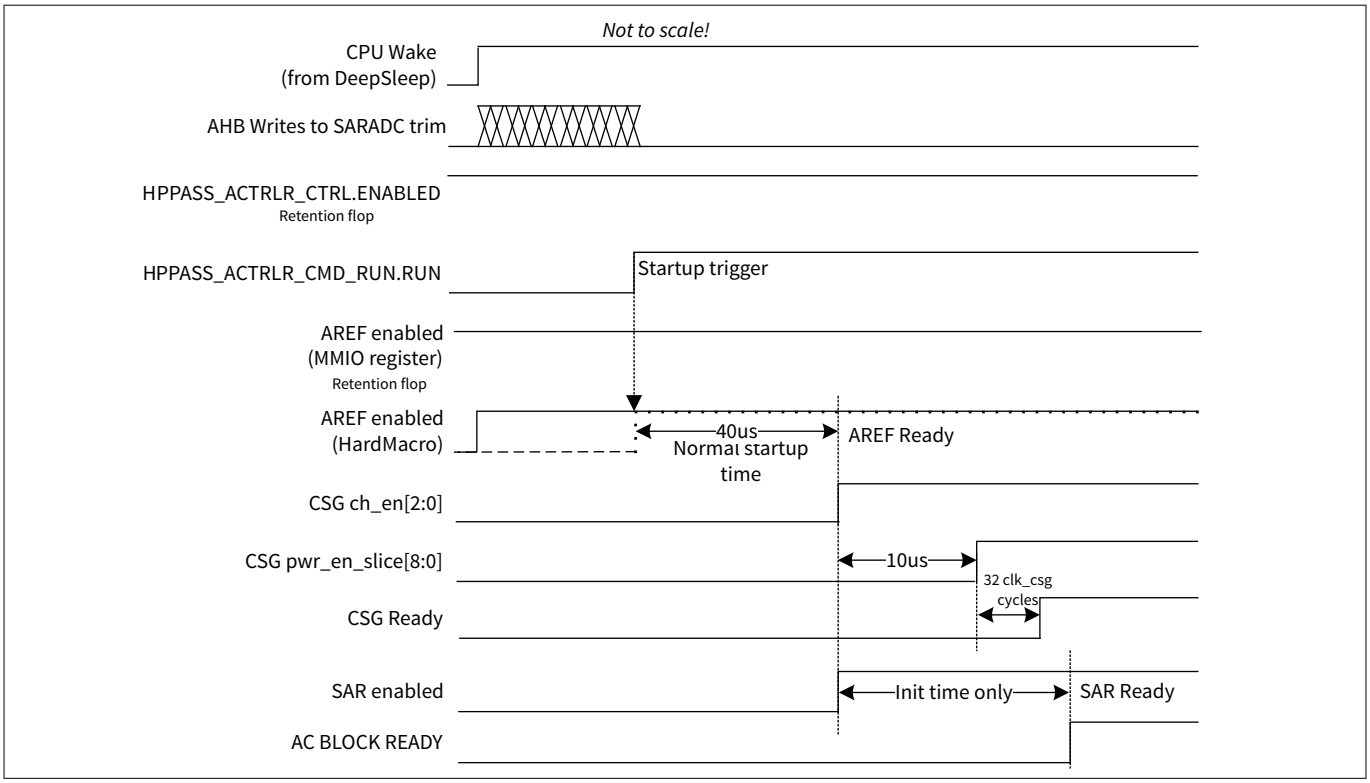


Figure 389 Startup after DeepSleep

31.2.1.2 Input trigger

Two sets of input triggers are connected from the trigger multiplexer to allow for tradeoffs between latency and flexibility: MUXED ("A") and ONE-TO-ONE ("B"). MUXED triggers have many sources, but higher latency, while ONE-TO-ONE is a single source to a single destination and therefore low latency. Set TRx_SEL bit-fields of HPPASS_INFRA_TR_IN_SEL/HPPASS_INFRA_TR_IN_SEL_2 register to select hardware MUXED ("A") or ONE-TO-ONE ("B") trigger, firmware pulse or level trigger.

HPPASS input triggers have the following properties (see diagrams below):

- Internally there are 16 triggers, supporting 8 SAR ADC groups and 9 CSG slices
- Hardware trigger inputs may be pulse or level
- External triggers must be synchronized
- Hardware triggers can be generated as PULSE on positive, negative, or both edges to support flexible operation from TCPWM generated waveforms, set HW_TRx_MODE bit-fields of HPPASS_INFRA_HW_TR_MODE/HPPASS_INFRA_HW_TR_MODE_2 register to select trigger mode
- A self-clearing firmware pulse trigger or firmware level trigger may be selected in place of a hardware trigger for any input trigger, and set the HPPASS_INFRA_FW_TR_PULSE/HPPASS_INFRA_FW_TR_LEVEL registers to generate a firmware pulse or level trigger

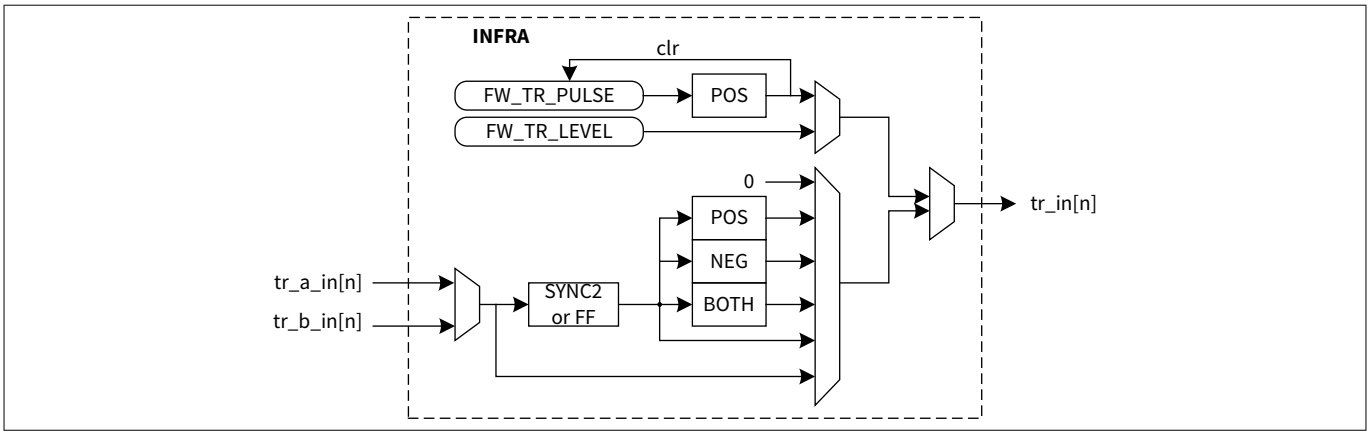


Figure 390 Trigger input processing

Input trigger routing

An important feature of the HPPASS is the ability for multiple processes to be triggered simultaneously. Therefore, the sixteen internal triggers are routed and available for selection in all sub-blocks. In addition, the AC has individual triggers for each of the sub blocks which can be programmed in an autonomous sequence.

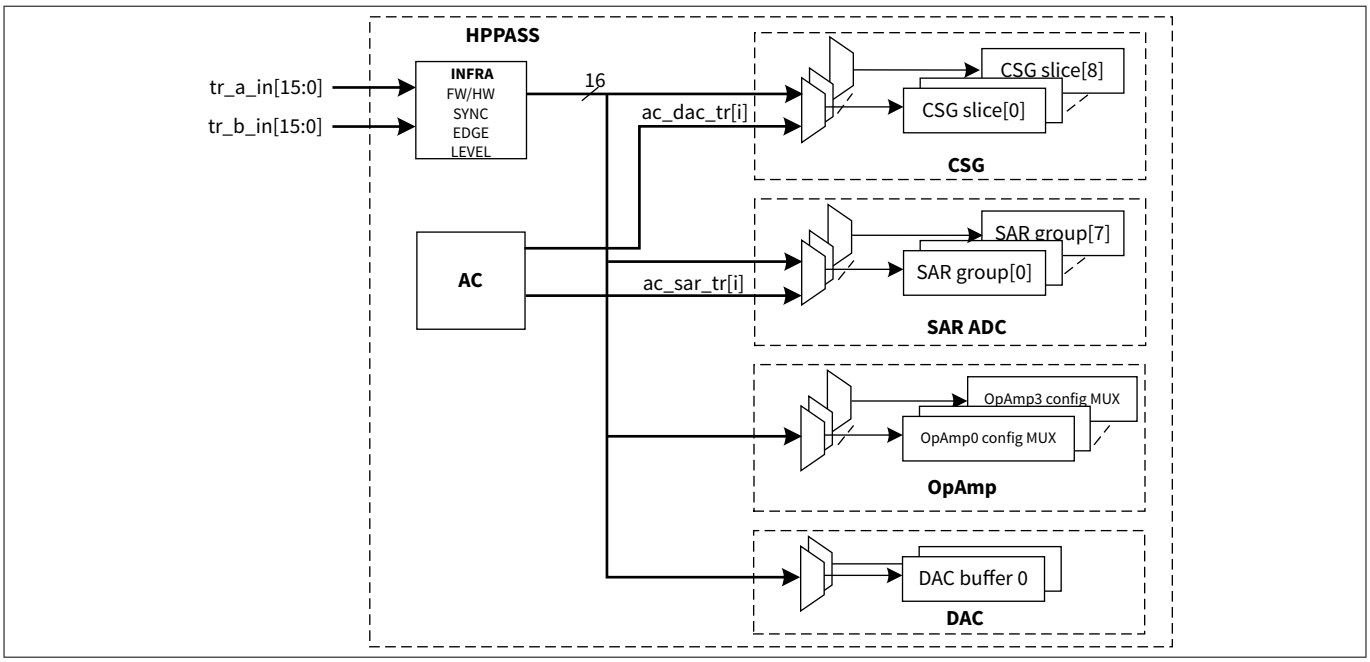


Figure 391 Input trigger routing

31.2.2 SAR ADC

The following figure shows the SAR ADC block diagram.

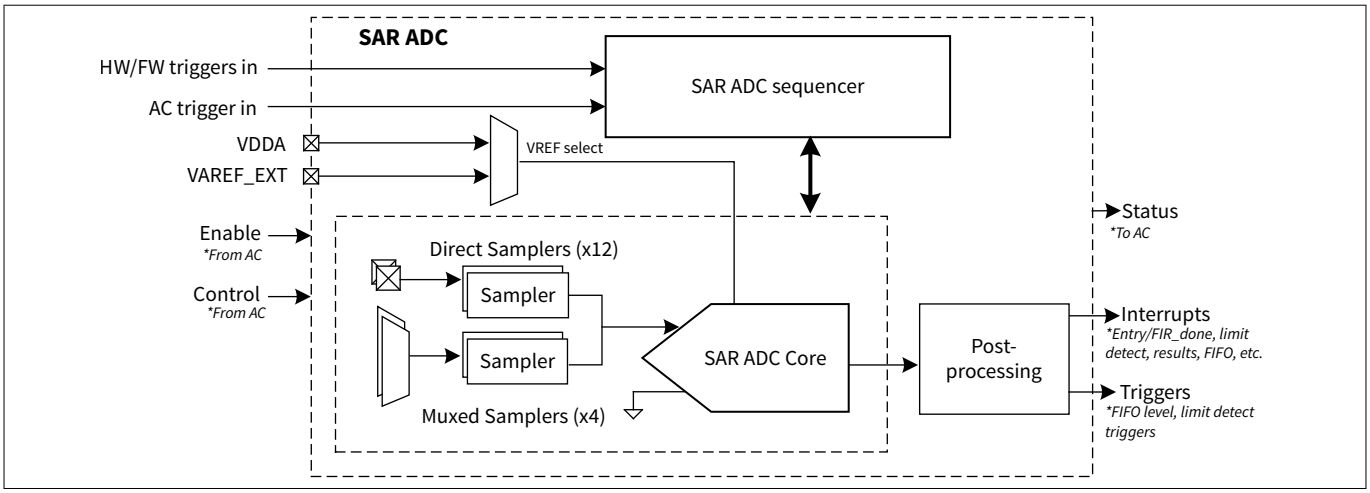


Figure 392 SAR ADC block diagram

The SAR ADC mainly consists of a SAR sequencer, samplers, SAR ADC core, and post-processing blocks. The following sections describe the functions and operations of these blocks.

Note:
- *The user can select VDDA or external reference from the VAREF_EXT pin for SAR ADC reference voltage by setting the VREF_SEL fields of the HPPASS_SAR_CFG_CTRL register. To guarantee SAR ADC performance, select the external reference from the VAREF_EXT pin*
- *When the VDDA supply voltage is less than 2.7 V, setting the LOW_SUPPLY_MODE bit of HPPASS_SAR_CFG_CTRL register to 1 reduces the SAR ADC maximum sample rate (see the datasheet for details)*
- *Toggle SARADC_CLOCK_MODE to control SARADC clock frequency only when VDDA < 2.8 V (ADC requires a frequency below 150 MHz), and change this bit only while the ADC is IDLE (i.e. it is not actively converting) by first setting STARTUP_CFG.SAR_EN = 0 to disable active conversions*

31.2.2.1 Channels

The HPPASS SAR ADC contains 12 direct samplers and four muxed samplers, which correspond to direct inputs and muxed inputs, as shown in the following diagram.

All direct and muxed samplers can be enabled by the HPPASS_SAR_CFG_SAMP_EN register. Ensure the sampler is enabled before the application uses it. All samplers support gains of 1, 3, 6, 12 by the HPPASS_SAR_CFG_SAMP_GAIN register to set the gain value.

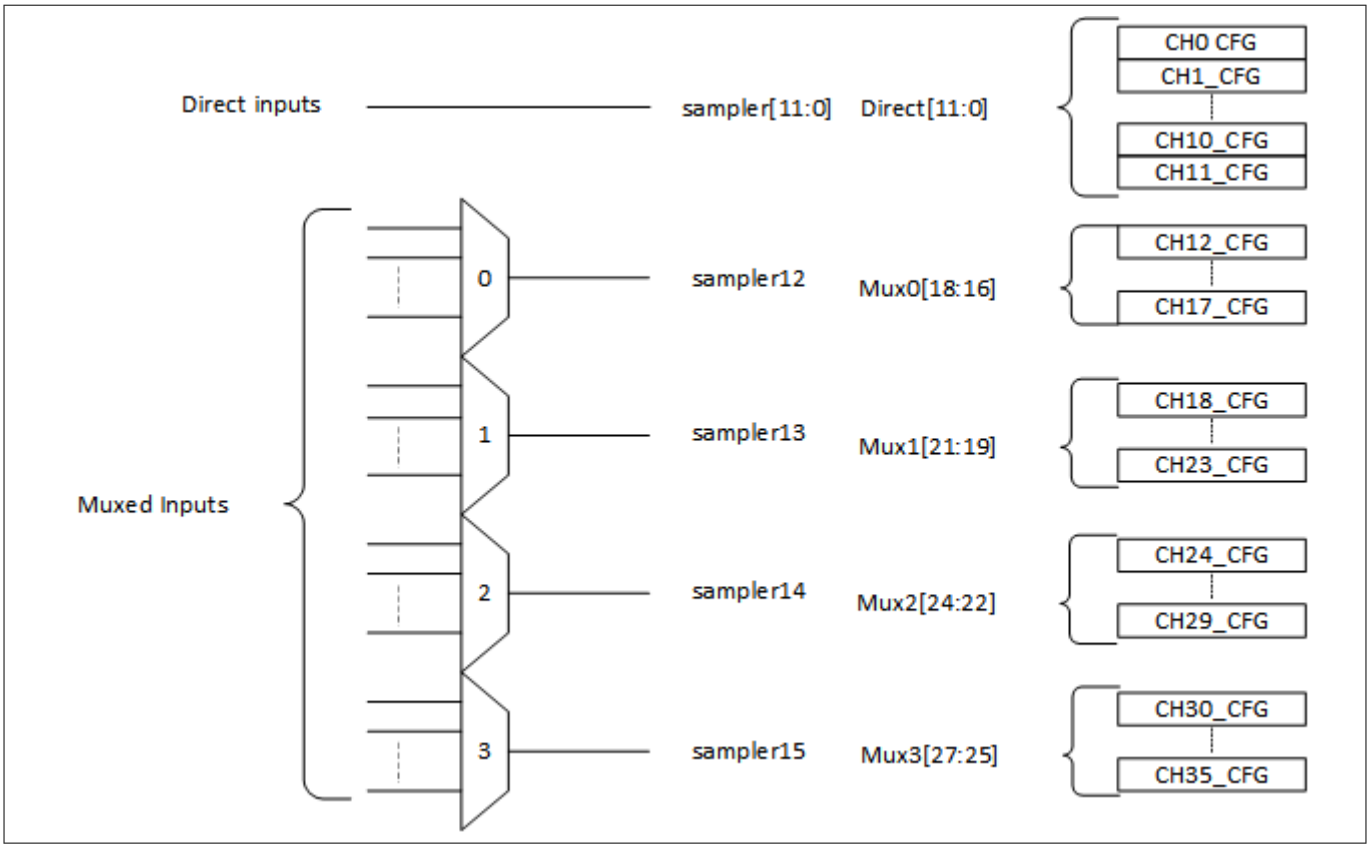


Figure 393 HPPASS SAR ADC channel mapping

SAR ADC has a 36 SAR channel-configuration in the PSC3M/P6 MCU. See Table 323 for actual port to channel mapping. HPPASS has only one temperature sensor and it can be connected to SAR ADC by MUX1 OR MUX2, see Temperature sensor chapter for details.

Table 323 PSC3M/P6 MCU SAR ADC channel mapping

Channel number	Sampler	Channel type	Input
0	Sampler 0	Direct	AN_A0
1	Sampler 1	Direct	AN_A1
2	Sampler 2	Direct	AN_A2
3	Sampler 3	Direct	AN_A3
4	Sampler 4	Direct	AN_A4
5	Sampler 5	Direct	AN_A5
6	Sampler 6	Direct	AN_A6
7	Sampler 7	Direct	AN_A7
8	Sampler 8	Direct	AN_B0
9	Sampler 9	Direct	AN_B1
10	Sampler 10	Direct	AN_B2
11	Sampler 11	Direct	AN_B3

(table continues...)

Table 323 (continued) PSC3M/P6 MCU SAR ADC channel mapping

Channel number	Sampler	Channel type	Input
12	Sampler 12 for MUX 0	MUX0 select 0	AN_B4
13		MUX0 select 1	P7.4
14		MUX0 select 2	P7.5
15		MUX0 select 3	OpAmp0N[O]
16		MUX0 select 4	OpAmp2N[O]
17		MUX0 select 5	CSG_DAC_OUT
18	Sampler 13 for MUX 1	MUX1 select 0	AN_B5
19		MUX1 select 1	P8.4
20		MUX1 select 2	P8.5
21		MUX1 select 3	OpAmp0P[O]
22		MUX1 select 4	OpAmp2P[O]
23		MUX1 select 5	Temperature sensor
24	Sampler 14 for MUX 2	MUX2 select 0	AN_B6
25		MUX2 select 1	P7.6
26		MUX2 select 2	P7.7
27		MUX2 select 3	OpAmp1N[O]
28		MUX2 select 4	OpAmp3N[O]
29		MUX2 select 5	Reserved
30	Sampler 15 for MUX 3	MUX3 select 0	AN_B7
31		MUX3 select 1	P8.0
32		MUX3 select 2	P8.2
33		MUX3 select 3	OpAmp1P[O]
34		MUX3 select 4	OpAmp3P[O]
35		MUX3 select 5	Reserved

The SAR ADC channel supports the following channel post processing features:

- Pseudo differential conversion: PD mode uses two samplers as a differential pair, a positive (POS) input and a negative (NEG) input. The POS and NEG inputs are sampled simultaneously. The hardware computes (POS - NEG) and returns a single conversion result. The result is written to the POS channel's result register
- Signed/Unsigned format: Useful for pseudo differential results and FIR processing
- Limit detection: Selection of 1 of 8 available limit detection resources
- FIFO write option: Selection of 1 of 4 available FIFOs
- Averaging: Accumulation of N results where N is one of {2, 4, 8, 16, 32, 64}

Post processing features that are not per channel, but rather per resource are:

- Two 16-tap FIR blocks: Select 1 of 36 channels as input
- Three 3P3Z IIR blocks: Select 1 of 36 channels as input

See the Post processing chapter for further details on the features.

31.2.2.2 Sequencer

The SAR ADC sequencer is shown in the following diagram:

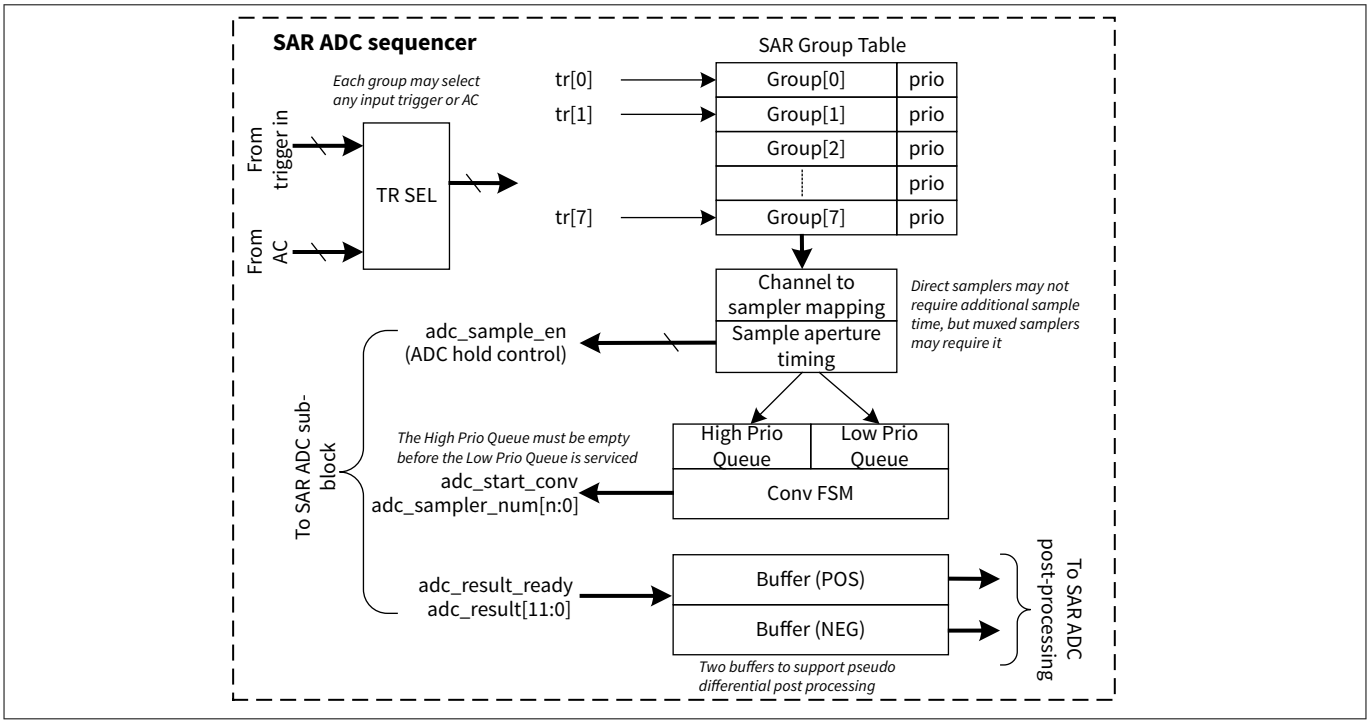


Figure 394 SAR ADC sequencer

The fundamental operation of the SAR ADC sequencer is controlled by a SAR group table, with eight groups programmed according to the user application.

Each entry in the table defines a group of channels that is simultaneously sampled when that row is triggered as each group shares other properties, such as sample time (default is 0) and priority.

Each group supports the control and conversion of all 16 samplers in the SAR ADC.

SAR ADC has three sample times. Each group can select disable (0 sample time) or one of the three available sample times, the sample time value can be set by HPPASS_SAR_CFG_SAMPLE_TIMEx (x = 0 - 2) registers. About the sample time details, check the following section.

The typical flow of the SAR ADC sequencer is as follows:

1. The AC or an external hardware trigger is asserted
2. Sample time cases:
   a. Sample time programming = 0 or disabled: All samplers in that group are immediately transitioned from SAMPLE to HOLD state
   b. Sample time programming > 0: Samplers are put/kept in SAMPLE state for a number of clocks in sample time programming, then transitioned to HOLD state
3. After the programmed sample time (or 0 sample time), the triggered group is allocated to a conversion queue based on the priority programming. The queue method is FIFO. However, if two or more groups are triggered in the same cycle and have the same priority, the lower numbered groups are put ahead of the higher numbered groups in the given queue
4. SAR ADC conversion is ongoing when there are any queued groups. All samplers of groups in the high-priority queue are converted before samplers of groups in the low-priority queue

5. As results are generated from the SAR ADC, any selected post processing for that channel is executed
6. Results are targeted to either result registers or FIFOs. For result registers, when all results for a given group are completed, a group-done interrupt (ENTRY_DONE) is raised to the CPU. For FIFO, the number of results is programmable (FIFO_LEVEL). See the Result handling section for details

The SAR group table defines a set of inputs that are simultaneously sampled. The following figure is the group table structure of the SAR ADC sequencer. See the Registers Technical Reference Manual for the details of HPPASS_SAR_SEQ_ENTRYx (x = 0 - 7) registers.

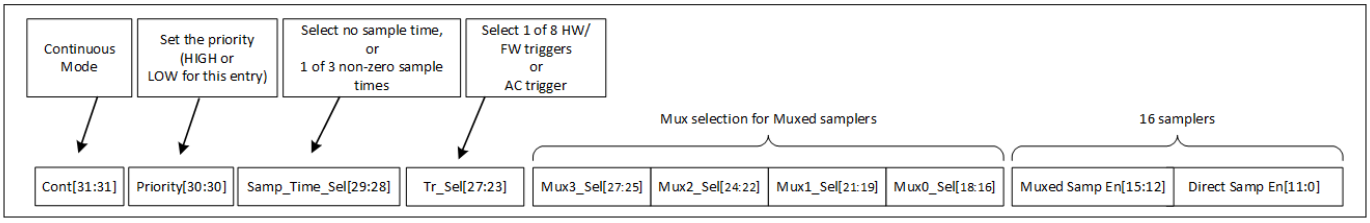


Figure 395 SAR ADC group table structure

- **Cont**: Setting this bit implements a Continuous mode. The group is automatically triggered again when it completes converting all the associated samplers. Note that this may use significant ADC bandwidth. The mitigation for this is to implement the sample time to regulate the trigger frequency
- **Priority**: This field indicates whether the group is put on the high-priority queue or the low-priority queue. High-priority groups are always converted before low-priority groups. High-priority groups are guaranteed to be converted and meet ADC specs. However, low-priority groups may be in the HOLD state for too long and not meet ADC specs. Therefore, check the HPPASS_SAR_CFG_ENTRY_HOLD_VIOLATION status register to ensure that low-priority groups were converted before they expired
- **Sample time select**: This field enables the selection of additional sample time for this group. By default, there is no additional sample time after this group is triggered (samplers are transitioned to the HOLD state immediately on a trigger event) or select 1 of 3 available sample time. HPPASS_SAR_CFG_SAMPLE_TIMEx register configures the sample time
- **TR select**: These bits allow each group to independently select external hardware, firmware, or AC to trigger this group
- **MUX select**: If a muxed sampler is enabled, these fields indicate the setting for the AROUTE input for the indicated muxed sample input. When the group is triggered, the AROUTE mux is changed to the mux selection value. In this case, sampling time is required to allow the input to settle before transitioning to a HOLD state and conversion
- **Sampler enable**: 16 samples - 12 direct and 4 muxed, when set to 1 initiates the active samplers and is converted when this group is triggered

***Note**: The HPPASS_SAR_CFG_SAR_STATUS.BUSY bit indicates the SAR busy status, this bit is not valid when continuous mode enabled in SAR ADC group.*

One or more, or all the available samplers (16) can be enabled for simultaneous sampling in a given group.

Both the direct samplers and MUX samplers are enabled with the indicated bit. However, if a MUX sampler is enabled, the MUX routing is controlled by the MUXx_SEL bit fields of HPPASS_SAR_SEQ_ENTRY register. See the MUX routing control section.

The channel output results are stored in the associated RESULT registers in order, starting with the direct samplers (0-11), then MUX0 (12-17), MUX1 (18-23), MUX2 (24-29), and MUX3 (30-35).

31.2.2.3 MUX routing control

The AROUTE MUX can be controlled by the SAR ADC sequencer, firmware, or autonomous controller (AC). Set the MUXx_CTRL fields of the HPPASS_SAR_CFG_AROUTE_CTRL_MODE register to select the control mode for

every AROUTE MUX, and globally enable AROUTE by the AROUTE_ENABLED bit of this register. By default, it is controlled by the SAR ADC sequencer using values in the SAR group table MUX0/1/2/3_SEL fields of the HPPASS_SAR_SEQ_ENTRYx registers or the HPPASS_SAR_ENTRY_MUX_SEL_ALIAS register.

***Note**:* *Firmware and AC settings are for advanced usage only. These settings override any SAR ADC group setting. Therefore, take care to program groups to not mix the AROUTE_CTRL_MODE settings for the MUXes used in a given group. Also, for a given MUX using any of firmware and AC modes, only 1 SAR ADC group should specify the usage of that MUX.*

When AROUTE MUX control mode is firmware, using HPPASS_SAR_CFG_AROUTE_FW_CTRL and HPPASS_SAR_CFG_AROUTE_FW_CTRL_CLR registers to set and clear MUX setting.

When AROUTE MUX control mode is AC, using SAR_AROUTE_TR and SAR_AROUTE_SEL fields of AC state HPPASS_ACTRLR_TTCFGx_TT_CFG3 (x = 0 - 15) register to control AROUTE MUX setting. See Example 5 - Using AC to control AROUTE.

The following section describes the AROUTE MUX control by SAR ADC sequencer.

Using SAR sequencer to control AROUTE MUX

Each group contains a routing specification for each of the muxed samplers. If any of the muxed samplers is in use in that group, the corresponding SAR_SEQ_ENTRY.MUX(i)_SEL bit fields of should be also programmed.

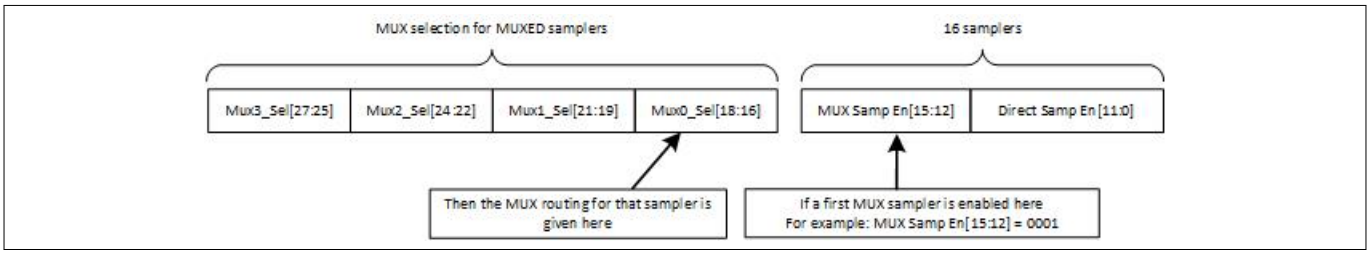


Figure 396 MUX routing selection

Since the MUX settings for a given group are always set at the trigger event, if this setting changes, then include sufficient sample time programming in that group. Select the sample time in the group configuration by the HPPASS_SAR_SEQ_ENTRYx.SAMP_TIME_SEL bits and set the sample time value by HPPASS_SAR_CFG_SAMPLE_TIME register. At the end of the sample time, the muxed sampler(s) are transitioned to hold and queued for conversion.

***Note**:* *Direct samplers can also be enabled with muxed samplers in any given group, but all samplers share the same sample time programming in that group, which may result in a performance overhead.*

***Note**:* *If the initial MUX select never changes, then the muxed sampler is basically equivalent to the direct samplers and the sample time may be optimized/reduced/eliminated.*

Assuming that the user wants to convert different MUX inputs, there are two usage modes for MUX routing control.

1. **Hardware mode**

   In this mode, different groups may be programmed for the same muxed sampler with a different routing selection, and different triggers can be used for these groups:

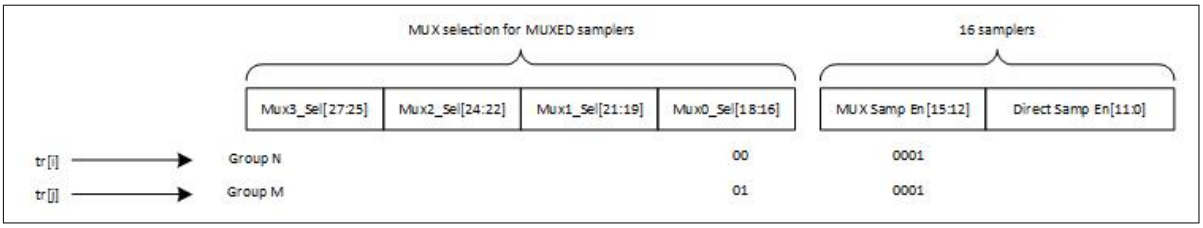


Figure 397 MUX routing selection example for hardware mode

***Note**: The above example uses the same muxed sampler. While group N is sampling or converting, the results are undefined if group M is triggered. Therefore, wait for the group N conversion to complete before trigger group M.*

This method is built into the hardware operation but will require multiple groups, and sample time is always required because the routing is changed at the assertion of the group trigger

2. Firmware mode

In this mode, firmware can update the MUX_SEL field in the same group. This requires some synchronization with regards to triggering of the group. For example, on the group done interrupt (ENTRY_DONE) from the previous trigger, firmware may write a new value to the MUX_SEL to be used for a subsequent trigger.

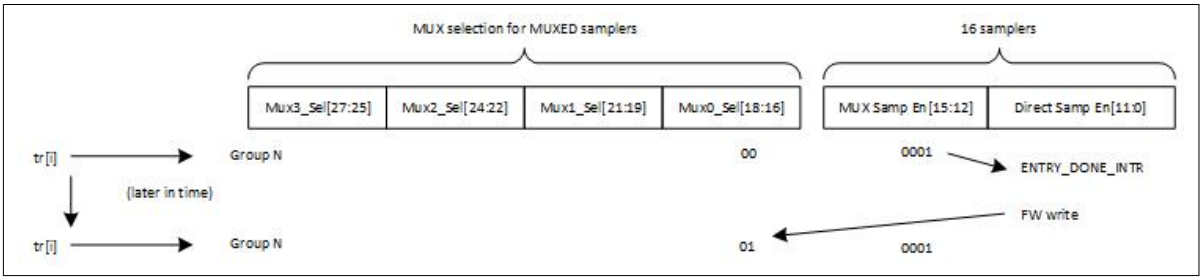


Figure 398 MUX routing selection example for firmware mode

This method uses the same group for different MUX_SEL values, but requires additional firmware coordination (for example, wait until the previous result is available before changing the MUX setting and triggering the next conversion).

There are MUX select alias registers SAR_ENTRY_MUX_SEL_ALIASx that only modify the MUX_SEL fields in the group so no read-modify-write is required. Using this method, depending on the triggering frequency, additional sample time may not be required because the routing may be changed in advance of the trigger

31.2.2.4 Illustration of triggering, sampling and conversion priority

There is one hardware trigger for each SAR ADC group, and each SAR ADC group can select any one of these eight triggers input.

The following diagram shows the simple case of one trigger routed to 2 groups with different priority. As expected all samplers transition to HOLD state simultaneously (assuming 0 sample time), the high-priority samplers are converted first, followed by the low-priority samplers.

After conversion is completed for a given sampler, the sampler reverts to SAMPLE state, and when the result is completed (considering any post processing), the associated ENTRY_DONE interrupt for the given group is raised.

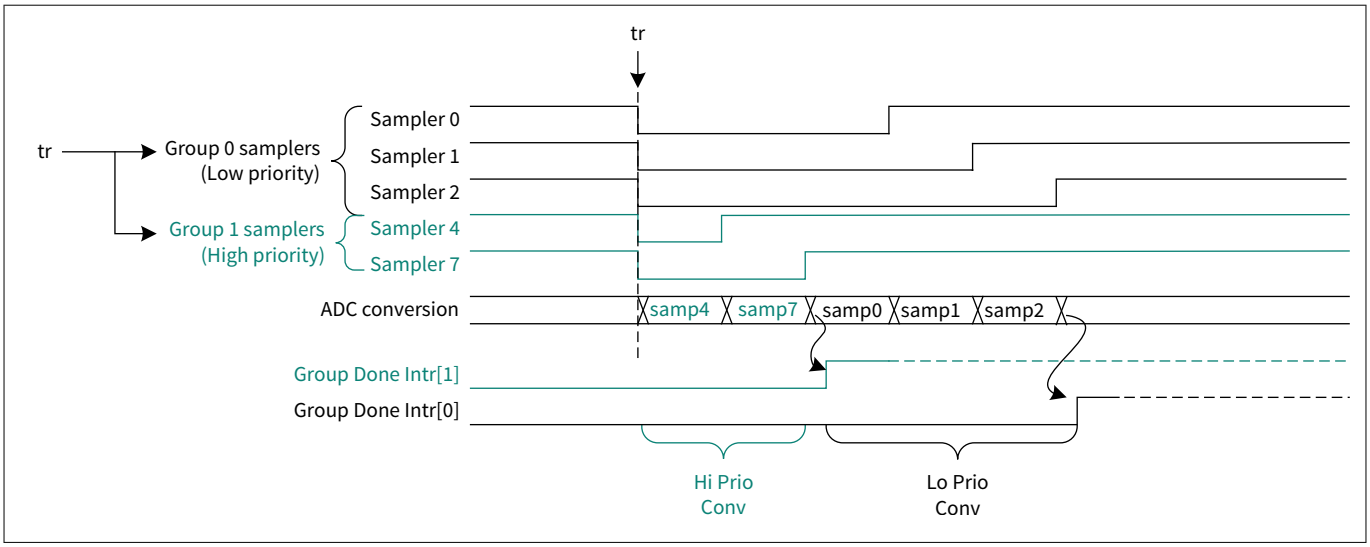


Figure 399 Simple trigger scenario

As the ADC samplers are always sampling or holding, additional sample time programming is not always required.

For example, for direct channels, it is rarely required because that sampler is always sampling the input when not converting.

For muxed channels, it depends on the timing and sequence of triggers. For example, if the user is cycling through muxed channels where the MUX select is changing on the trigger, sampling is required. However, if the user is cycling through muxed channels where the MUX select is not changing, additional sampling may not be required.

The following diagram illustrates the same case but where group 0 has a programmed sampling time greater than 0. On the trigger event, the sampling time, in this case, is extended by the programmed number of cycles.

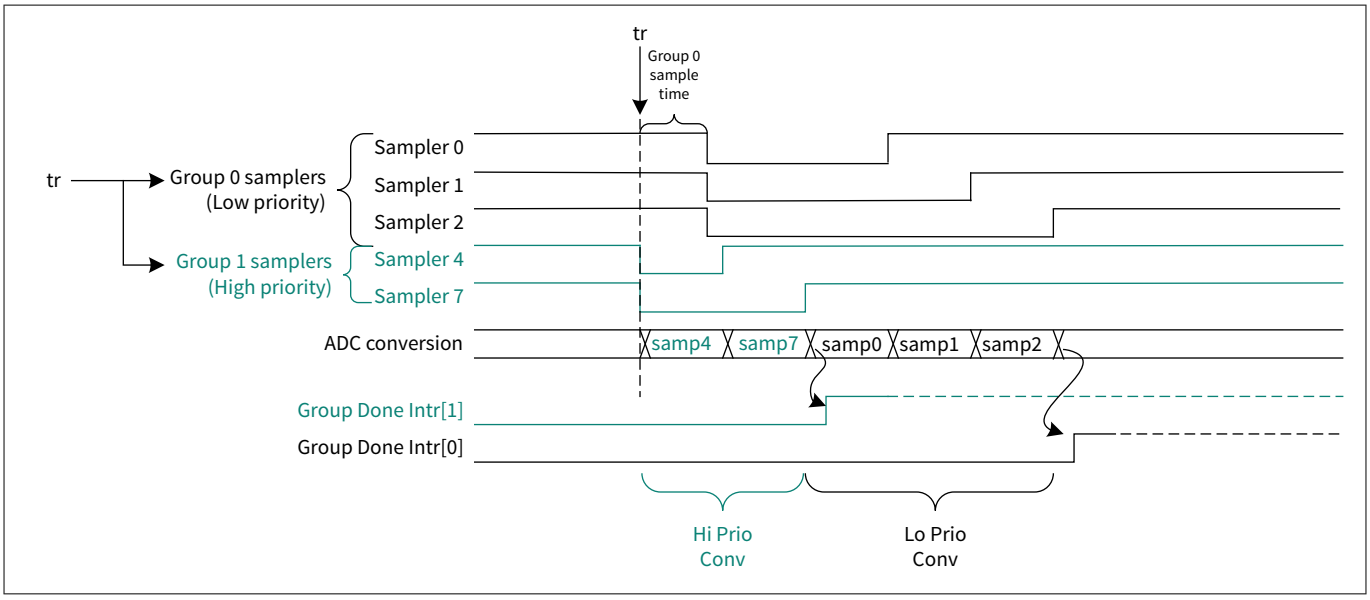


Figure 400 Simple trigger scenario with sampling time

In a more realistic scenario, a new trigger is asserted for different groups, while the channels of a previously triggered group or groups are being held or converted.

This illustrates the high and low priority concept, but also exemplifies that once started, all the samplers in that group are completed, before any other group is started.

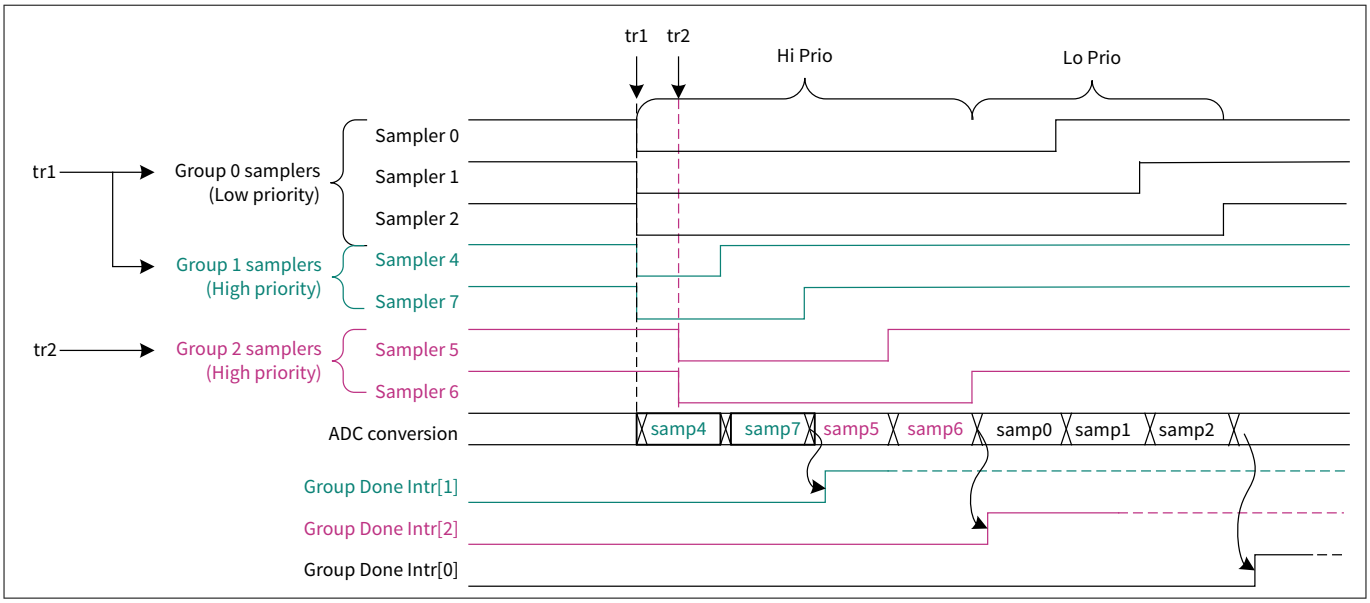


Figure 401 Multiple trigger scenario

31.2.2.5 Error status

SAR ADC has the following error status:

- **Hold violation status**

  The user is responsible to schedule system triggers such that when low-priority channels are triggered and in sampler hold state, the SAR ADC has sufficient bandwidth to convert these groups before leakage renders the sample invalid. However, status will be provided to flag a hold time violation according to the following mechanism:

  - Hold timing is based on a group. Although there may be multiple samplers enabled in that group, if any of samplers in that group violate the hold time, a violation is flagged for the group
  - There is a global HOLD_CNT setting of the HPPASS_SAR_CFG_ENTRY_HOLD_CNT register which is applied to all groups
  - This resolution of each unit of HOLD_CNT is 16 CLK_HF cycles (corresponds to a single ADC conversion)
  - The default MAX hold time is nominally 3 µs. If a group is in the conversion queue and holding for more than this time limit, the HOLD_VIOL status bit of the HPPASS_SAR_CFG_ENTRY_HOLD_VIOLATION register is set. Set the ENTRY_HOLD_VIOLATION bit of the HPPASS_MMIO_HPPASS_INTR_MASK register to enable this error to CPU

- **Trigger collision status**

  If trigger for a given SAR ADC group is asserted while that group is in progress, for example, from the previous trigger until all results are completed, a new trigger is ignored and a trigger collision status is set for this trigger in HPPASS_SAR_CFG_ENTRY_TR_COLLISION register. User can set ENTRY_TR_COLLISION bit of HPPASS_MMIO_HPPASS_INTR_MASK register to enable this error to CPU

- **Result overflow status**

  If the user fails to clear the ENTRY_DONE or FIR_DONE interrupt from a previous trigger, prior to the subsequent ENTRY_DONE or FIR_DONE of the subsequent trigger, the previous results will always be overwritten and the associated overflow status bit will be set in HPPASS_SAR_CFG_RESULT_OVERFLOW register. User also can set RESULT_OVERFLOW bit of HPPASS_MMIO_HPPASS_INTR_MASK register to enable this error to CPU

- **FIFO overflow and underflow status**

  When writing to a full FIFO or reading from an empty FIFO, generate FIFO overflow or underflow interrupt. User can set FIFO_OVERFLOW bits and FIFO_UNDERFLOW bits of HPPASS_MMIO_HPPASS_INTR_MASK register to enable FIFO error to CPU

31.2.2.6 Post processing

The SAR ADC post processing has the following features:

- Pseudo differential hardware subtraction
- Signed format
- Averaging (Result accumulation, no shifting function)
- FIR processing
- 3P3Z processing
- Limit detection

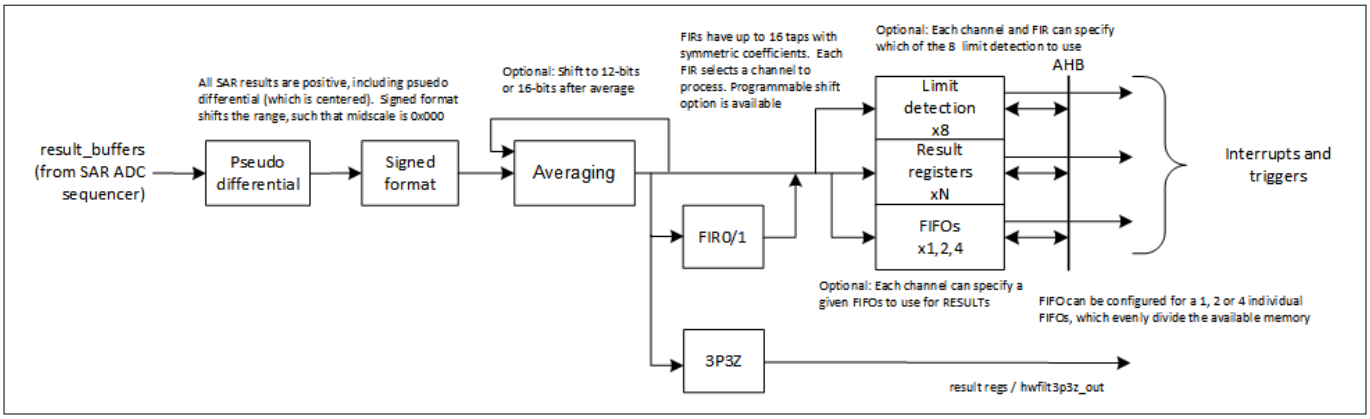


Figure 402 SAR ADC post processing data flow

Pseudo differential

Pseudo differential (PD) mode uses two samplers as a differential pair, one for the positive (POS) input and one for the negative (NEG) input. The device samples POS and NEG simultaneously and computes:**Pseudo differential result = (POS − NEG)**and stores the single conversion result into the POS channel result register. PD mode is enabled per channel using, HPPASS_SAR_CFG_CHAN_CFGx.DIFFERENTIAL_EN bit (See the device Registers Technical Reference Manual for the exact register map.) If pseudo differential mode is enabled for a channel, then the corresponding POS/NEG sampler pair used by PD shall have identical gain. For example channel 0, samplers 0 and channel 1, samplers 1 must be configured to the same gain. Also, the averaging mode of this channel must be disable.

Channel pairing rules

- Direct samplers:PD is supported only on even-numbered direct channels (0, 2, 4, 6, 8, 10). In these pairs, the even channel acts as POS, and the odd channel acts as NEG. If PD is enabled on an odd direct channel (1, 3, 5, 7, 9, 11), it is ignored. Result of the conversion always written to the even/POS channel result register
- Muxed samplers:Only even/odd MUX sampler pairs are supported and therefore pseudo differential configuration may only be specified in the channels associated with the even MUX0 and MUX2. Pseudo differential configuration in channels associated with MUX1 and MUX3 will be ignored

Data format, scaling, and output interpretation: Because the range of the result is doubled for pseudo differential subtraction (max is 0xFFF - 0x000, min is 0x000 - 0xFFF), the subtraction result is 13 bits. 12-bit pseudo-differential is not supported.

Unsigned/signed format

By definition all outputs from the SAR (including the pseudo differential result) is unsigned. Converting to a signed result is a user convenience. The result is simply shifted such that mid scale is represented by 0x000 (subtract 0x800), and sign-extended. This feature is available as channel configuration and most useful for pseudo differential results. For a pseudo differential, result(13-bit), it is centered around 13 bits (0x1000). Therefore

- Result_pd_unsigned (13-bits) = (POS − NEG) + 4096
- Result_pd_signed (14-bits)= (result_pd_unsigned - 4096)

For example if CH0 and CH1 are configured for pseudo differential operation and have results 21 and 42 respectively:

- Result_pd_unsigned = (21 – 42) + 4096 = 4075(0x0FEB)
- Result_pd_signed = (result_pd_unsigned - 4096) = 4075 – 4096 = -21(0xFFEB)

***Note**: The ADC data outputs can be programmed to be MSB aligned or LSB aligned. The post-processed ADC output it written into the 16-bit FIFO and into the 32-bit CHAN_RESULT register after being aligned depending on CHAN_CFG.LEFT_RIGHT_ALIGNMENT.*

Averaging

Averaging (accumulation of N results) may be set for any channel where N is one of { 2, 4, 8, 16, 32, 64}. Since the RESULT registers and FIFO widths are limited to 16-bits, unsigned averaging by 32 will be force right shifted by 1 and signed averaging will be force shifted by 2 to avoid overflow. Similarly for 64 sample accumulation, unsigned result will be >> 2 and signed result will be >> 3.

When averaging is selected for a given channel – a result will only be generated and stored every N triggers. The user has a choice of triggering a given entry which contains averaged channels N times (or periodically), or the AC may be used to periodically re-trigger the given entry. see AC Example 2 - AC trigger SAR averaged group

For different averaging configurations (e.g., single-ended vs pseudo-differential) the result alignment is performed.

Finite impulse response filters (FIR)

There are two FIR resources. Each of the FIR resources can select one of the available channel results to process by setting the CHANNEL_SEL bit fields of the HPPASS_SAR_CFG_FIR_CFGx register. The FIR supported 16-tap with symmetric coefficients. The following figure shows the FIR filter construction, set the FIR coefficients by COEF bits-fields of HPPASS_SAR_FIRx_FIR_COEFSy register.

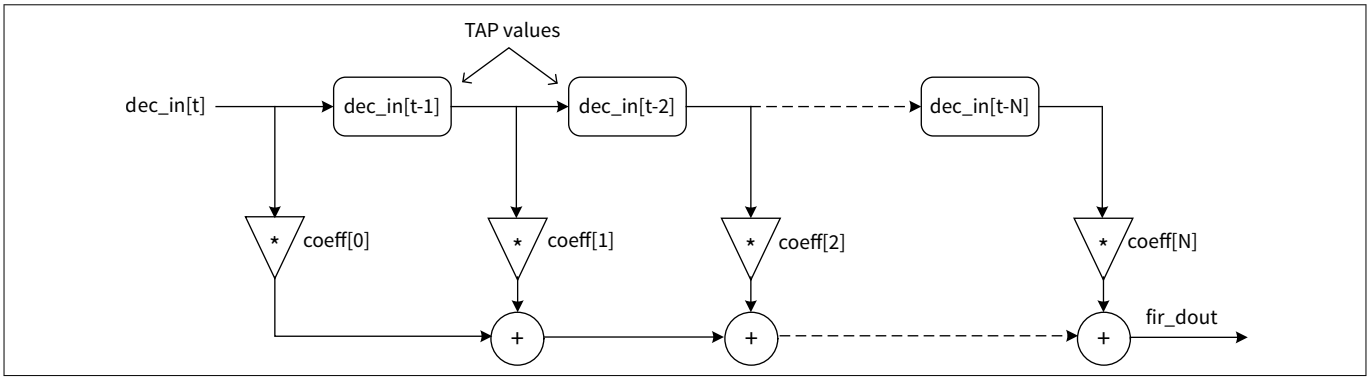


Figure 403 FIR filter construction

Dec_in signifies the input data to the FIR from the decimator, while the [t] signifies the instance in time. As data enters the FIR, it is first multiplied by the initial coefficient value (coeff[0]) then it is added with the product (*) of the previous decimator data [t-1] and the next stored coefficient value (coeff[1]) and so on. This continues until it reaches the number of TAPs enabled (N) which is a programmable value up to 16. The FIR coefficient value(16-bits) can be configured by HPPASS_SAR_FIR0/1_FIR_COEFSx (x = 0 - 15) register, and saved tap values (16-bits) can be read by HPPASS_SAR_FIR0/1_FIR_TAPSx (x = 0 - 15) register.

The coefficient values can range from $-1+1/2^{15}$ to $1-1/2^{15}$ with a resolution of $1/2^{15}$. The coefficient uses twos-complement notation and the decimal point is fixed to the right of the most significant bit.

```
Examples: -0.25 = 0xE000 and 0.25 = 0x2000
```

User has the choice to get a result on every conversion, or set the HPPASS_SAR_CFG_FIR_CFGx.WAIT_TAP_INIT bit to delay the first result until the filter is initialized (16 samples), then continual thereafter.

The post-FIR data is written into the 32-bit FIR_RESULT register depending on FIR_CFG.LEFT_RIGHT_ALIGNMENT.

Infinite Impulse Response Filters (3P3Z)

There are 3 3P3Z filter resources. Each of the 3P3Z filter resources can select one channel to process. The 3 pole 3 zero filter has 3 programmable poles and 3 programmable zeros. It also has the following programmable options:

- maximum/minimum limit control
- coefficient scaling

- input difference gain
- output scaling
- offset control

The output of the 3P3Z filter is 23 bits, it can be fully read back or the results can be scaled / (aligned to MSB/LSB) to fit in the result registers.

Limit detection

There are eight limit detection resources. Any channel and the FIRs can specify any of the eight limit detection resources. When a channel or FIR result is above/below a threshold or inside/outside two thresholds, it generates triggers/interrupts. The HPPASS_SAR_CFG_RANGE_HIGHx and HPPASS_SAR_CFG_RANGE_LOWx registers used to configure threshold values for limit detection resource, the detection options include ABOVE, BELOW, INSIDE, OUTSIDE and configured by the HPPASS_SAR_CFG_RANGE_CFGx.RANGE_COND register.

When multiple channels share a limit detection resource, the resulting status is ORed. Limit conditions may be configured as both an interrupt or level-based trigger. The interrupt status can be read by the RANGE[7:0] bits of HPPASS_SAR_CFG_SAR_RANGE_INTR register. The trigger output can be enabled by the SAR_RANGE_TR bits of HPPASS_MMIO_TR_LEVEL_OUTx register.

For interrupts, in the case where multiple channels or FIRs are ORed using a given limit detection resource, a HPPASS_SAR_CFG_RANGE_STATUS status register may be read to determine the which channel and FIR was the source of the limit detection.

31.2.2.7 Result handling

There are two ways for the system (CPUSS or DMA) to retrieve SAR results:

- FIFO
- RESULT register

FIFO

There are 32 words SRAM available for FIFO storage. A given application may specify 1, 2, or 4 FIFOs which divide this space evenly, which can be done by SEL[1:0] bits of the HPPASS_MMIO_FIFO_CFG register. Each channel and FIR can select any of the available FIFOs. For example, if four FIFOs are selected and there are eight ADC channels, the eight ADC channels can select any of the four FIFOs by configuring the HPPASS_SAR_CFG_CHAN_CFGx.FIFO_SEL register. Conversion results are streamed into a given FIFO with an optional identifying tag that is enabled by the HPPASS_SAR_CFG_CTRL.CHAN_ID_EN bit.

Interrupts and triggers specific to the FIFO status are supported and based on user HPPASS_MMIO_FIFO_LEVELx.LEVEL register programming, level interrupts (LEVEL, OVERFLOW and UNDERFLOW) and level based triggers are available as outputs.

The triggers output of associated FIFO can be enabled individually by setting HPPASS_MMIO_FIFO_CFG.FIFO_TR_EN register.

The FIFO state can be cleared by writing a “1” to HPPASS_MMIO_FIFO_CLR.CLR register, or clearing the HPPASS_ACTRLR_CTRL.ENABLED register to disable AC, or during deep sleep mode. The actual SRAM data does not get reset.

- FIFO write operation

  FIFO can store/write the SAR result till it is full. The FIFO write pointer (WR_PTR) gets incremented on every data write in FIFO memory. If there is an AHB read in progress it is stalled
- FIFO read operation

  FIFO data can be readout by accessing read-only register HPPASS_MMIO_FIFO_RD_DATAx. It is one AHB read address per FIFO. The data can be read from FIFO till it is empty. The FIFO read pointer (RD_PTR) gets incremented on every data read from FIFO memory
- FIFO status

  There are 3 status bits to create the FIFO interrupt and triggers:
  - FIFO_LEVE: set when the number of samples written to the FIFO transitions from the programmed HPPASS_MMIO_FIFO_LEVELx.LEVEL to LEVEL + 1
  - FIFO_OVERFLOW: set at the point when a write request is made to a full FIFO. When FULL, the ADC will not write (over-write) the FIFO
  - FIFO_UNDERFLOW: set at the point when user firmware (via an AHB read) reads the FIFO when it’s EMPTY. Read value is undefined

  In addition to these status bits, there is a read-only HPPASS_MMIO_FIFO_USEDx.USED register that reflect the current number of samples in the FIFO. The RD_PTR and WR_PTR values of HPPASS_MMIO_FIFO_STATUSx are also available, but these are used for debug only.

  Set HPPASS_MMIO_FIFO_CLRx.CLR register to clear RD_PTR and WR_PTR values for associated FIFO. A self-clearing pulse will be generated by hardware to clear the bit when the reset is completed

RESULT register

Each channel and each of the two FIRs have a RESULT register called HPPASS_SAR_CFG_CHAN_RESULTx and HPPASS_SAR_CFG_FIR_RESULTx registers. The results for each group are updated after all the channel conversions for that group are complete. The associated interrupt for that group (ENTRY_DONE) is asserted in conjunction with the result update. The FIRs has a separate interrupt when the FIR result is ready to read (FIR_DONE), one for each FIR.

A HPPASS_SAR_CFG_RESULT_OVERFLOW register is provided to inform the user that a previous value was overwritten before it was read. When receiving an ENTRY_DONE or FIR_DONE interrupt, the user is required to clear the indicated bits of the HPPASS_SAR_CFG_SAR_RESULT_INTR register. If these bits are not cleared in time for the next result update, the RESULT_OVERFLOW bits of SAR_CFG_RESULT_OVERFLOW register is set.

There is a HPPASS_SAR_CFG_RESULT_UPDATED register, which has bits to indicate which channel results have been updated. This status register may be used in conjunction with the combined result interrupt, to determine which SAR ADC result and FIR result registers were updated. The user can write a '1' to any bits that they wish to clear. As an alternative method, the HPPASS_SAR_CFG_SAR_RESULT_INTR register may be read to determine which groups are completed, and thereby indirectly knowing the associated channels.

To optimize data transfer, a SAR result alias register (HPPASS_SAR_CFG_CHAN_RESULT_PACKEDx) is provided to allow for two 16-bit results to be packed into one 32-bit AHB read (see the Registers Technical Reference Manual for more details).

31.2.2.8 SAR ADC interrupts

As shown in the following diagram, result interrupts are based on the SAR ADC groups. An interrupt is generated when the channels in a given SAR ADC group are completed and ready for reading from the HPPASS_SAR_CFG_CHAN_RESULTx registers. Similarly, the FIR or 3P3Z has have associated RESULT registers and associated FIR DONE, 3P3Z DONE status. See RESULT register for more details.

An individual interrupt provided for each group (interrupt_sar_entry_done[7:0]) or FIR done (interrupt_sar_fir_done[1:0]), as well as a combined result interrupt (interrupt_sar_results), see Interrupts chapter for more details. If the combined interrupt is used, either the HPPASS_SAR_CFG_RESULT_UPDATED or HPPASS_SAR_CFG_SAR_RESULT_INTR registers must be read to determine the cause of the interrupt.

Limit detect interrupts generate when a channel or FIR result is above/below a threshold or inside/outside two thresholds. The individual interrupt (interrupt_sar_range[7:0]) provided for each limit detection resources and combined limit detect interrupt (interrupt_sar_ranges). See Limit detection for more details.

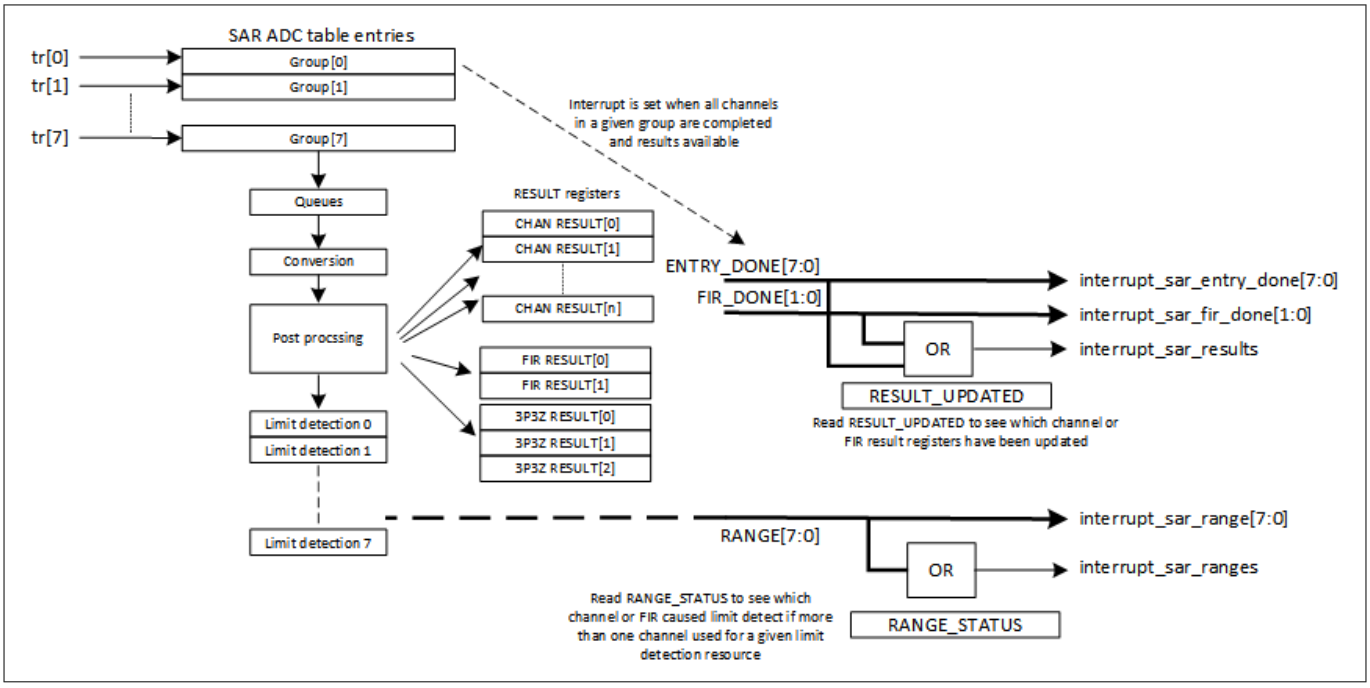


Figure 404 SAR ADC interrupts

31.2.2.9 Calibration

The HPPASS SAR ADC core supports the offset, linearity, and gain calibrations, calibrating offset and linearity for each input sampler and gain for each gain setting. Offset is due to the comparator, sampler, or common

mode voltage mismatches, and linearity mainly originates from process mismatch. Gain errors are due to reference buffer offsets and reference divider mismatches.

When certain events, such as changing reference voltages or gain setting, need to run at least an offset calibration, since the offset does not scale with the reference voltage. Disabling and reenabling the SAR ADC with the same configuration settings does not require another calibration run. However, it is recommended to run offset calibration if temperature changes drastically. Reference and supply voltages must be within the operating ranges and be sufficiently stable during calibration, otherwise, calibration must be repeated.

The SAR ADC has four offset calibration registers to store calibration data (HPPASS_SARADC_CALOFFST0 - 3), 16 linearity calibration registers (HPPASS_SARADC_CALLIN0 - 15), coarse-grained gain calibration register (HPPASS_SARADC_CALGAINC), and fine-grained gain calibration register (HPPASS_SARADC_CALGAINF). See the Registers Technical Reference Manual for the details of these calibration registers.

***Note**: HPPASS must be ENABLED to write and read these calibration registers. When the HPPASS is disabled, these registers are reset.*

HPPASS SAR ADC supports factory-calibration and self-calibration. Infineon recommends using factory-calibration, because it is faster than self-calibration.

The PDL implements the copying of factory calibration values to the SAR ADC calibration registers in the HPPASS initialization API. It is recommended to call the initialization API in the application. For more information, see Peripheral Driver Library API Reference Manual.

Factory-calibration

When MCU is produced, the SAR ADC is calibrated at the specified reference voltage in the factory, and then stores these calibration values to SFLASH. User can copy these factory calibration values to the SAR ADC calibration registers in HPPASS initialization without enabling self-calibration to speed up HPPASS startup. Since the offset calibration is temperature-dependent, SFLASH stores -40, 25, and 125 offset calibration values (SFLASH_SAR_CALOFFST_0/1/2/3_N40C/125C/25C) for every sampler, and the user can call the Infineon Peripheral Driver Library (PDL) API to automatically adjust offset calibration values based on current temperature and reference voltage. The SFLASH_SAR_CAL_LIN_TABLE register stores 16 linearity calibration values for every sampler, and the SFLASH_SAR_CALGAINC and SFLASH_SAR_CALGAINF registers store the gain calibration values. The factory calibration reference voltage is stored in the SFLASH_SAR_CALREFPT register. If the application uses these factory calibration values, copy these values from SFLASH to SAR ADC calibration registers in HPPASS initialization.

Self-calibration

SAR ADC also supports self-calibration, sets the STARTUP_CAL_OFFSET_EN, STARTUP_CAL_LINEARITY_EN, and STARTUP_CAL_GAIN_EN bits of the HPPASS_SAR_CFG_CTRL register to respectively enable offset, linearity, and gain calibration on SAR ADC start-up. When AC enables the SAR, it runs all selected calibration routines in this register. The user must wait for BLOCK_READY, at which time SAR ADC calibration and initialization are complete, before starting the application. When the device enters DeepSleep mode, the value of the SAR ADC calibration registers will be lost. To reduce another SAR ADC calibration wait time when wake up from DeepSleep mode, the user should read back and store the SAR ADC calibration register values in DeepSleep capable RAM. On wake-up, the user can write these values back into the SAR ADC calibration registers. For more details about the startup flow, see the Startup section.

31.2.3 Comparator and slope generator (CSG)

The comparator and slope generator are flexible blocks that compare a 10-bit, 30-Msps DAC value with a selected analog input signal, or compare two analog inputs. Note that the DAC is used only as an internal reference and cannot be measured outside of the chip; however, the ADC can measure it. The output of this block is a 1-bit compare value, which can be used as an internal or external trigger or interrupt source. A common use case is to control the behavior of the TCPWM in industrial applications.

As shown in Figure 405, two logic partitions are wrapped around the CSG analog subblock: one for the DAC code generation and another for comparator output post-processing. Input and output triggers, along with user interaction via the AHB bus, also play a key role in CSG function. This functional unit is referred to as a slice, and the PSC3M/P6 MCU supports nine such slices.

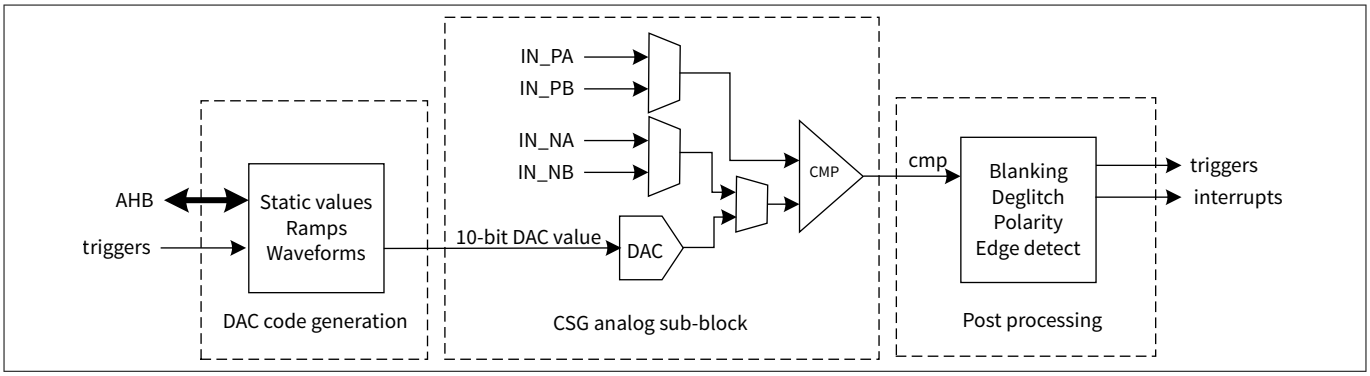


Figure 405 CSG block diagram

Every CSG slice comparator supports a unique set of positive and negative inputs. It can be selected by setting the CMP_POS_SEL and CMP_NEG_SEL fields of the HPPASS_CSG_SLICEx_CMP_CFG (x = 0 - 8) register. Table 324 lists comparator input options for each slice.

Table 324 CSG comparator positive and negative inputs

Slice	Comparator positive inputs	Comparator negative inputs
Slice 0	CSG_AROUTE_MUX0/AN_B0	CSG_DAC0/CSG_AROUTE_MUX1/ AN_B1
Slice 1	CSG_AROUTE_MUX1/AN_B1	CSG_DAC1/CSG_AROUTE_MUX2/ AN_B2
Slice 2	CSG_AROUTE_MUX2/AN_B2	CSG_DAC2
Slice 3	CSG_AROUTE_MUX3/AN_B3	CSG_DAC3/CSG_AROUTE_MUX4/ AN_B4
Slice 4	CSG_AROUTE_MUX4/AN_B4	CSG_DAC4/CSG_AROUTE_MUX5/ AN_B5
Slice 5	CSG_AROUTE_MUX5/AN_B5	CSG_DAC5
Slice 6	CSG_AROUTE_MUX6/ CSG_AROUTE_MUX7	CSG_DAC6/CSG_AROUTE_MUX6/ CSG_AROUTE_MUX7
Slice 7	CSG_AROUTE_MUX7/ CSG_AROUTE_MUX8	CSG_DAC7/CSG_AROUTE_MUX7/ CSG_AROUTE_MUX8
Slice 8	CSG_AROUTE_MUX8/ CSG_AROUTE_MUX6	CSG_DAC8

Here, CSG_AROUTE_MUXx are 4:1 CSG analog input multiplexers. Signal selection through the multiplexer is performed by setting the appropriate value in the AROUTE_SEL field of the CSG_AROUTEMUX_SEL register. The inputs to the multiplexers are shown in Table 325.

Table 325 CSG_AROUTE_MUX inputs

Multiplexer	Inputs
CSG_AROUTE_MUX0	AN_A0/OpAmp0P/AN_A1/AN_B1
CSG_AROUTE_MUX1	AN_A1/OpAmp0N/AN_A2/AN_B2
CSG_AROUTE_MUX2	AN_A2/OpAmp1P/AN_A0/AN_B0
CSG_AROUTE_MUX3	AN_A3/OpAmp1N/AN_A4/AN_B4
CSG_AROUTE_MUX4	AN_A4/OpAmp2P/AN_A5/AN_B5
CSG_AROUTE_MUX5	AN_A5/OpAmp2N/AN_A3/AN_B3
CSG_AROUTE_MUX6	OpAmp3P/P7.5/P7.6/P7.7
CSG_AROUTE_MUX7	P8.0/P8.2/P8.4/P8.5
CSG_AROUTE_MUX8	P7.4/P7.5/P7.6/OpAmp3N

See the Register reference manual for more details.

31.2.3.1 10-bit DAC

There are various modes for generating and controlling the DAC codes, both static and dynamic. The supported modes are listed below:

- Direct firmware control of DAC code
- Buffered firmware control of DAC code
- Two-value (Hysteretic) control of DAC code
- Automatic DAC ramp generation
- Optional use of look-up table (LUT) for DAC arbitrary waveform generation

Figure 406 supports the following features:

- An AHB-accessible DAC_VAL register (HPPASS_CSG_SLICEx_DAC_VAL) that directly drives the DAC code
- Two DAC buffers, DAC_VAL_A (HPPASS_CSG_SLICEx_DAC_VAL_A register) and DAC_VAL_B (HPPASS_CSG_SLICEx_DAC_VAL_B register), whose definitions depend on the selected DAC mode (see the section on DAC modes)
- A programmable slope generator with parameters START, END, DAC_STEP, DAC_PERIOD
- A period counter with fractional capability to precisely modulate the slope of a ramp or waveform
- A START trigger (to start a configured mode) and UPDATE trigger (to advance to the next code), which can be hardware, firmware, or AC (or PERIOD for UPDATE)
- Status for HW_START, SLOPE_DONE and BUFFER_EMPTY for interrupt generation
- If an error is detected in the DAC slope or DAC LUT modes configuration, the DAC FSM will return to the idle state, and no interrupt is generated under the following conditions:
  - (start > end) or ((step_size > (start – end)) in forward modes
  - (start < end) or ((step_size > (end – start)) in reverse modes

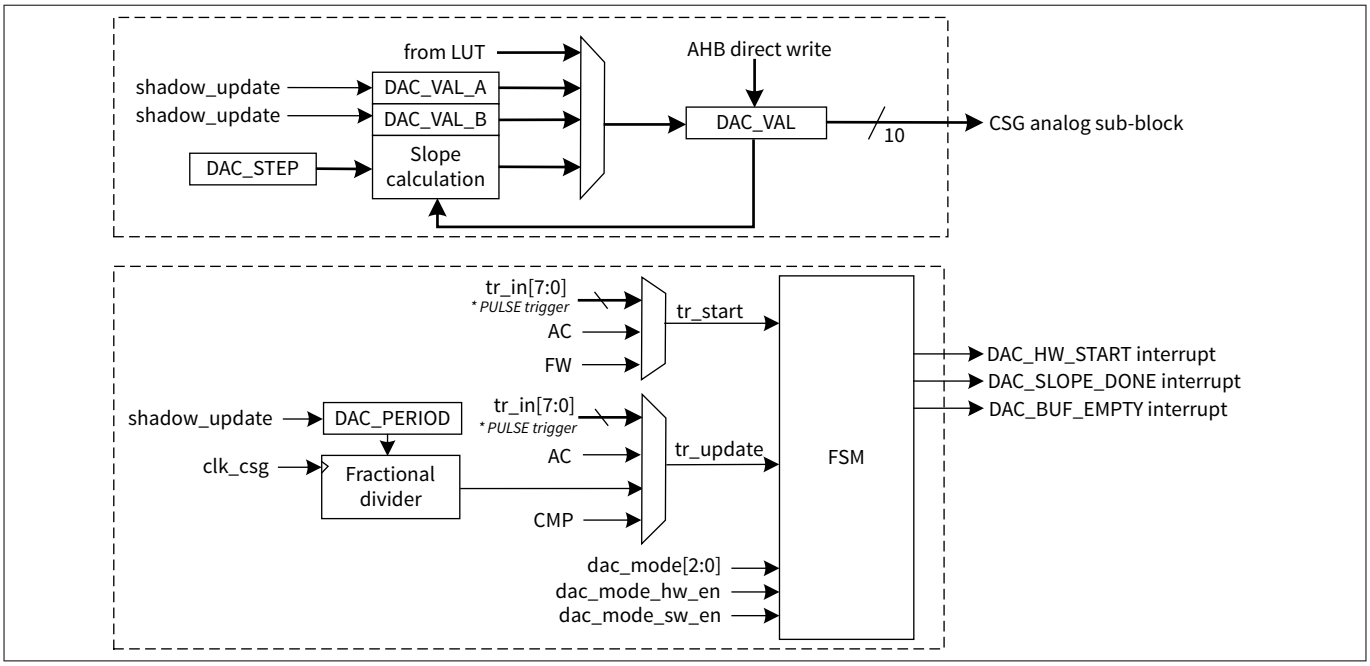


Figure 406 DAC code generation

In addition to supporting CSG DAC output observability, the DAC output can be connected to SAR ADC by setting the VDAC_OUT_SEL bit fields of the HPPASS_CSG_CSG_CTRL register to select the desired CSG slice DAC output for SAR ADC conversion.

The CSG DAC output connects to SAR ADC by AROUTE MUX0 (see SAR ADC Channels chapter for details of AROUTE MUX connectivity). When CSG DAC output is connected to SAR ADC, DAC observability blanking must be enabled (see DAC observability blanking for details).

Performance note: The CSG comparator is a clocked/auto-zeroed comparator, and its response depends on the CLK_CSG frequency and the applied signal conditions. For application timing and electrical limits, refer to the device datasheet and the CSG electrical specifications.

31.2.3.2 DAC modes

The user has multiple options to control the DAC data sequencing. The following modes can be set individually on each slice.

Direct write

At any time, the user can write directly to the HPPASS_CSG_SLICEx_DAC_VAL register via the AHB bus, which directly drives the analog DAC input. This feature can be used to initialize the DAC code before enabling the block.

***Note**: Any write to this register disables any DAC mode currently in progress.*

Buffered mode

The HPPASS_CSG_SLICEx_DAC_VAL_A register acts as a single buffer for the HPPASS_CSG_SLICEx_DAC_VAL register. When this mode is enabled, each asserted UPDATE trigger transfers the value from DAC_VAL_A to DAC_VAL and generates a DAC_BUF_EMPTY interrupt in the HPPASS_CSG_DAC_INTR register. An interrupt is generated only when a new value written to DAC_VAL_A is transferred to DAC_VAL. Figure 407 shows a possible usage scenario illustrating the operation of the buffered mode.

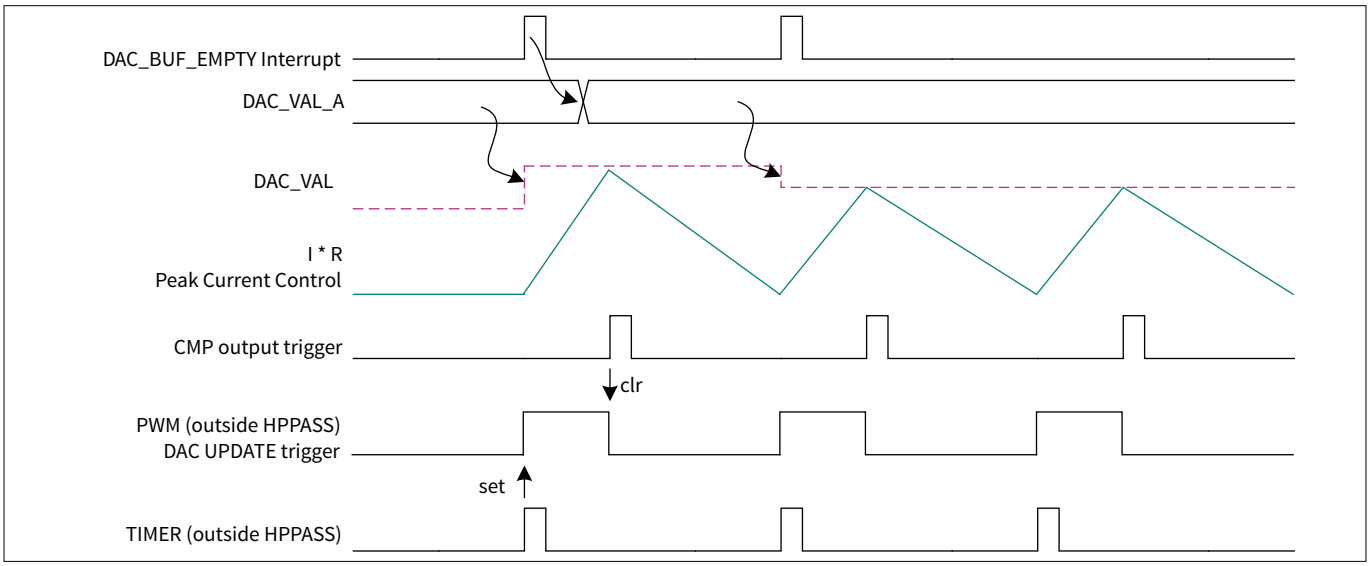


Figure 407 DAC buffered mode

Hysteretic mode

In this mode, two buffered values are provided through the HPPASS_CSG_SLICEx_DAC_VAL_A and HPPASS_CSG_SLICEx_DAC_VAL_B registers. This allows a high sense (A), low sense (B) trip operation using a single comparator.

- When enabled, the high sense value (DAC_VAL_A) is transferred to DAC_VAL for output
- When the UPDATE trigger transitions from 0 to 1, DAC_VAL_B is transferred to DAC_VAL
- When the UPDATE trigger transitions from 1 to 0, DAC_VAL_A is transferred to DAC_VAL

On a DAC_VAL_A update, a DAC_BUF_EMPTY interrupt is generated, and the user may update the buffered values. The hardware uses these values starting with the next DAC_VAL_A update, as shown in the following example usage scenario diagram.

In this mode, the comparator may be used as the update trigger (as shown in Figure 408) and the user has the flexibility to use the CMP output or invert it. By modulating the DAC_VAL_A and DAC_VAL_B values, the CMP period and duty cycle of the comparator output may be modulated.

For level-trigger operation, minimum high/low duration must be greater than or equal to one CLK_CSG cycle.

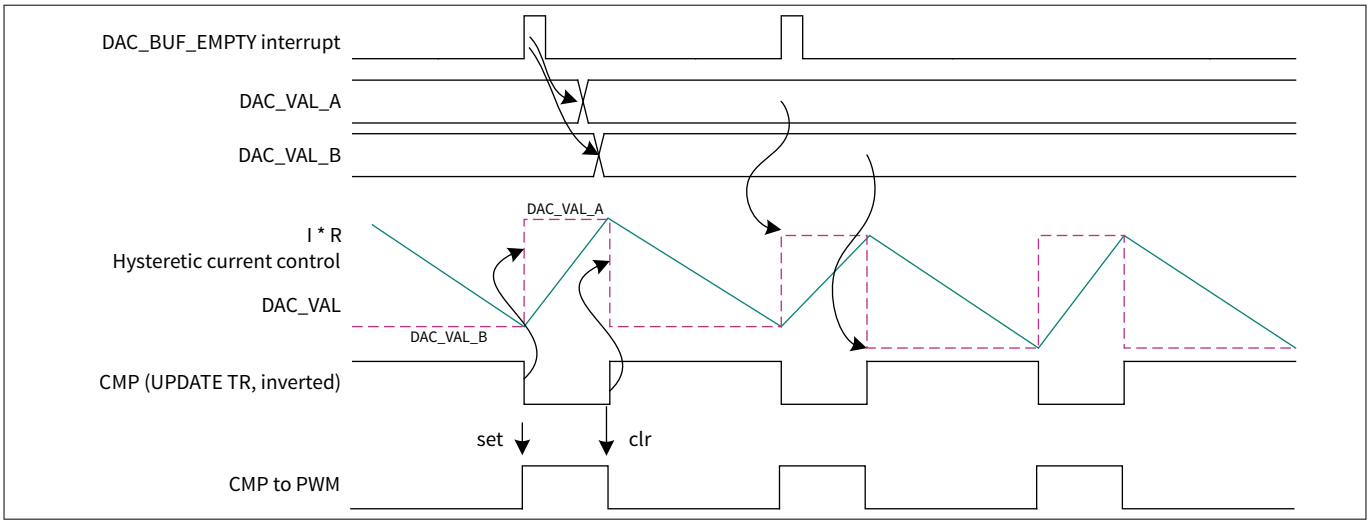


Figure 408 DAC hysteretic mode

Slope generation mode

The slope can be configured using the following slope generation modes. The DAC_STEP value is set by the HPPASS_CSG_SLICEx_DAC_CFG.DAC_STEP bits, while DAC_VAL_A and DAC_VAL_B are set in the HPPASS_CSG_SLICEx_DAC_VAL_A and HPPASS_CSG_SLICEx_DAC_VAL_B registers, respectively.

- **SLOPE_RISING:** Start at DAC_VAL_A, increment by DAC_STEP, and stop when DAC_VAL ≥ DAC_VAL_B
- **SLOPE_FALLING:** Start at DAC_VAL_A, decrement by DAC_STEP, and stop when DAC_VAL ≤ DAC_VAL_B
- **SLOPE_TRIANGULAR:** Start at DAC_VAL_A, increment by DAC_STEP until DAC_VAL ≥ DAC_VAL_B, then decrement by DAC_STEP until DAC_VAL ≤ DAC_VAL_A

As shown in Figure 409 usage, parameter updates only occur at the start of a new slope. Any combination of the START (DAC_VAL_A), END (DAC_VAL_B), DAC_STEP and DAC_PERIOD is buffered for update.

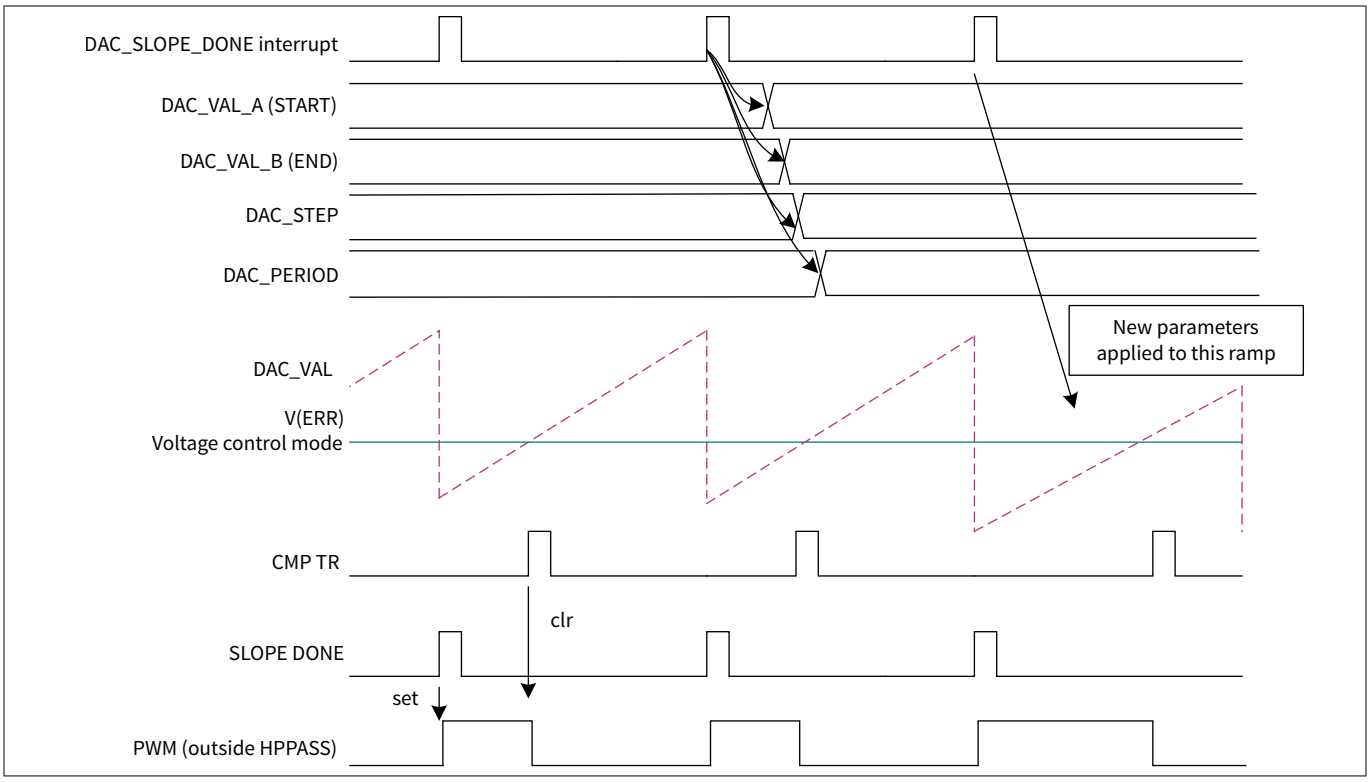


Figure 409 DAC slope generation example

There are three ways to control parameter update:

1. **Continual:** Set the DAC_CONT bit in the HPPASS_CSG_SLICEx_DAC_CFG register to enable continuous mode. In this mode, the hardware updates the buffered parameters at the start of a slope (as shown in the Figure 409) and restarts when the end value is reached. Otherwise, the operation is one-shot mode, and the last output value is held. With this method, it is possible to time the firmware parameter update writes using the Slope Done trigger, which provides the maximum time before the new values are applied to the following slope.

   ***Note**: DAC continuous mode is valid only for slope and LUT generation modes.*

2. **Synchronized:** Set the DAC_PARAM_SYNC_EN bit of HPPASS_CSG_SLICEx_DAC_CFG register. In this mode, the hardware updates parameters (still at the start of the slope) only when the READY bit in the HPPASS_CSG_SLICEx_DAC_PARAM_SYNC register is set to '1'. This bit indicates to the hardware that all buffered parameters have been written. This method prevents partial parameter updates.

   ***Note**: After writing the parameters to the registers, set HPPASS_CSG_SLICEx_DAC_PARAM_SYNC.READY bit to '1'. Read back this bit until the READY bit is '1'.*

   ***Note**: HPPASS_CSG_SLICEx_DAC_PARAM_SYNC.READY bit is cleared by the hardware on the parameter update event.*

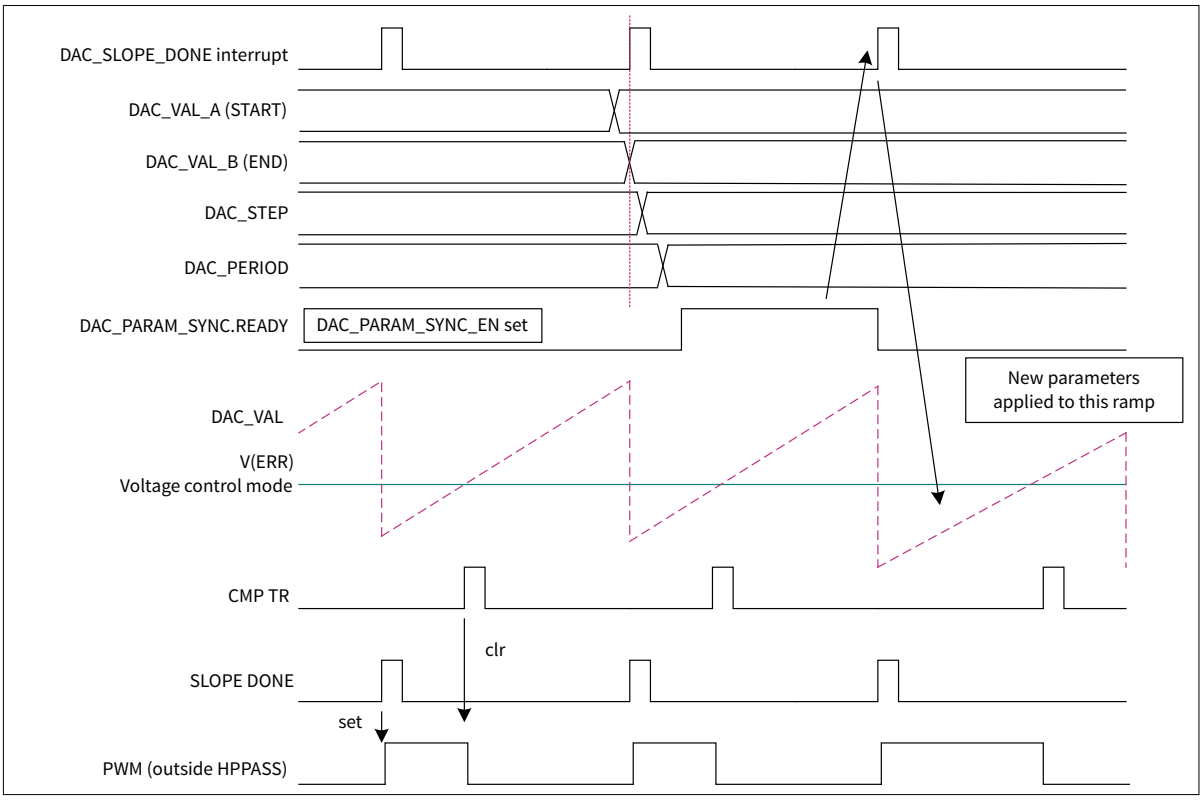


Figure 410 Synchronized parameter update

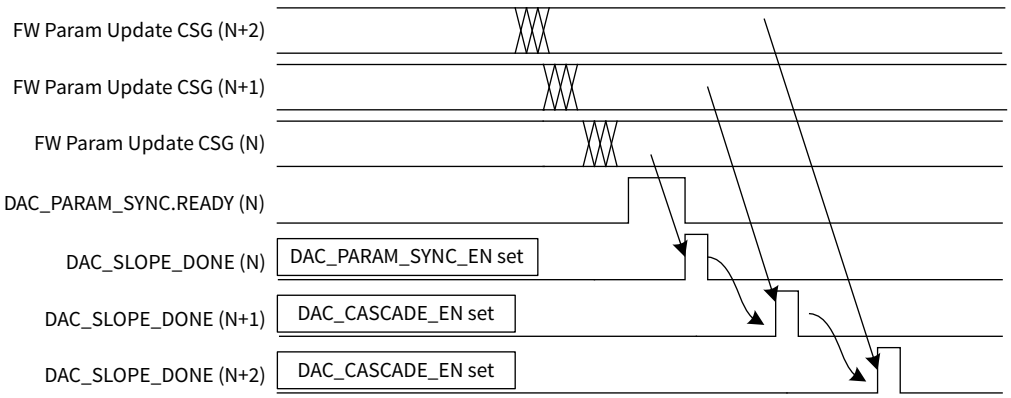
3. **Cascaded:** Set the DAC_CASCADE_EN bit in the HPPASS_CSG_SLICEx_DAC_CFG register to synchronize parameter updates across successively numbered CSG slices. This feature is useful in applications that use multiple CSG slices operating in a fixed phase relationship with each other (for example, three slices).

   Enable the DAC_CASCADE_EN bit in the higher-order CSG slices (N+1 and N +2). In the lower-slice (N), set the DAC_PARAM_SYNC_EN bit in the HPPASS_CSG_SLICEx_DAC_CFG register.

   The firmware then updates the buffered parameters in three slices (in any order). After all updates are complete, the firmware sets the DAC_PARAM_SYNC.READY bit in the lower-order slice. As shown in Figure

411, on the subsequent CSG slope done events, the parameters are updated in a cascaded fashion across these successive slices.

Figure 411 Cascaded parameter update



LUT waveform generation mode

LUT mode operates similarly to slope generation mode, except that the parameters are address-related, rather than data-related. The generated address sequence is routed to a look-up table, where the user may store a custom waveform. Each successive 10-bit data fetch from the table is loaded into DAC_VAL and driven by the DAC for output. See the Slope generation mode for details on updating the control parameters. The maximum update rate is limited to CLK_CSG/2.

- LUT_FORWARD: Start at address DAC_VAL_A, increment the address by DAC_STEP, and stop when the address ≥ DAC_VAL_B
- LUT_REVERSE: Start at address DAC_VAL_A, decrement address by DAC_STEP, and stop when address ≤ DAC_VAL_B
- LUT_FORWARD_REVERSE: Start at address DAC_VAL_A, increment by DAC_STEP until the address ≥ DAC_VAL_B, then decrement the address by DAC_STEP until the address ≤ DAC_VAL_A

***Note**: Note: Step size is DAC_STEP+1.*

***Note**: One LUT is shared among every three CSG slices. In the PSC3M/P6 MCU(P/M) 6 MCU, the CSG includes three LUTs: LUT 0, LUT 1, and LUT 2. Each LUT contains 128 samples. LUT 0 serves DAC0/1/2, LUT 1 serves DAC3/4/5, and LUT 2 serves DAC6/7/8.*

The above modes control the DAC to generate an output voltage for the comparator negative input, and the DAC output value for each CSG slice can be read from the HPPASS_CSG_SLICEx_DAC_STATUS.DAC_VAL register.

31.2.3.3 DAC mode start/stop

A mode can only be changed when no DAC operation is currently in progress.

There are two ways to start a new DAC mode firmware or hardware (see the Registers Technical Reference Manual for details on the HPPASS_CSG_SLICEx_DAC_MODE_START register).

- **HW_START:** When this bit is set, the DAC mode is synchronized to the next DAC_TR_START selected trigger, set DAC_TR_START_SEL fields of HPPASS_CSG_SLICEx_DAC_CFG register to select the DAC start trigger, which can be any of the external hardware trigger inputs or AC trigger.

  ***Note**: if the hardware start trigger is asserted again while the DAC mode is in progress, it will be ignored*

- **FW_START:** The mode is enabled immediately when this bit is set

The HW_START/FW_START bit of HPPASS_CSG_SLICEx_DAC_MODE_START register is automatically reset to '0' on DAC_SLOPE_DONE event, unless the HPPASS_CSG_SLICEx_DAC_CFG.DAC_CONT bit is set for continuous mode.

To stop the current mode's progress, clear the associated start bit. The current DAC_VAL is frozen to the last value, until it is overwritten directly, or a new mode is started.

In addition to disabling the mode by resetting the HW_START and FW_START bit to 0, any direct write to the HPPASS_CSG_SLICEx_DAC_VAL register overrides the current DAC output and disables any DAC mode in progress.

The HPPASS_CSG_SLICEx_DAC_MODE_START register has a read-only BUSY bit to indicate that a DAC mode has been enabled and is currently in progress. If FW_START is used to start the mode, then the FW_START bit is equivalent to this bit and low when the mode is completed or stopped. If HW_START is used to start the mode, then this status bit goes high on the DAC_START_TR event, and low when the mode is completed or stopped.

31.2.3.4 DAC update triggers

Set the DAC_TR_UPDATE_SEL bit-fields of the HPPASS_CSG_SLICEx_DAC_CFG register to select the DAC update trigger. When the selected DAC_TR_UPDATE is asserted, the DAC code is updated, the behavior depends on the selected DAC_MODE. The following triggers are available for DAC update trigger:

- Any of the sixteen HPPASS hardware triggers from other on-chip peripherals or firmware trigger (see Input trigger section for details)
- The CSG slice local DAC_PERIOD counter (the Period divider is programmed in the HPPASS_CSG_SLICEx_DAC_PERIOD register)
- The CSG slice local comparator output or inverted local comparator output (see Post processing section)
- The AC trigger

***Note**: If one of the HPPASS hardware or firmware triggers are selected for updating the trigger, then the trigger must be pulse mode. In addition, the maximum frequency of the active trigger edge must be less than or equal to CSG CLOCK/2.*

DAC period counter

When the DAC_PERIOD is selected, program the period in clock cycles. There is an integer (12-bits) field and an optional fractional (5-bits) field. The fractional portion allows for resolutions up to 1/32 of a clock. See the Registers Reference Manual for the details of the HPPASS_CSG_SLICEx_DAC_PERIOD register.

Fractional operation allows for higher precision slope generation. However, when this feature is used, it results in cycle jitter as shown in Figure 412. In this example, we have an integer that only divides by 4, and below it an integer and fractional divide by $4\frac{1}{2}$. As shown in the following figure, $4\frac{1}{2}$ cycles is generated as the average of alternating between 4 and 5 cycles.

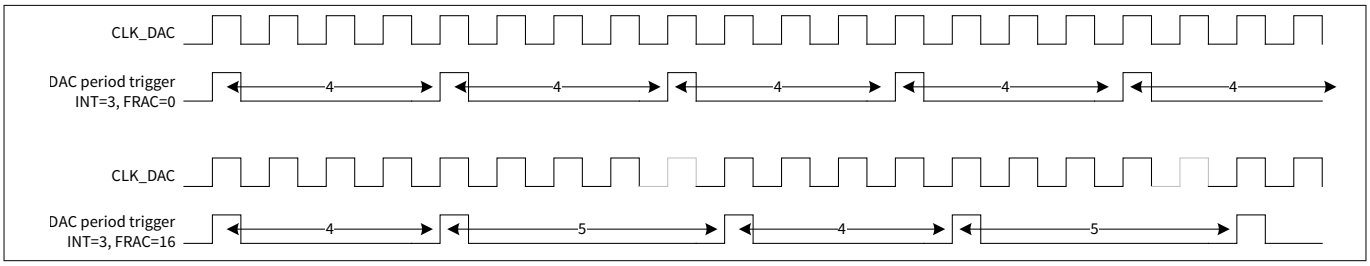


Figure 412 DAC period example

As an example of a more complicated waveform, consider $20\frac{7}{8}$. The implies that in 167 clocks, there are eight triggers, which results in the pattern 20-21-21-21-21-21-21-21.

31.2.3.5 Hysteresis

HPPASS CSG has a configurable hysteresis feature that allows the user to select the hysteresis magnitude for the input using a 3-bit configuration (0 mV to 84 mV). The comparator switching threshold defines the low-to-high transition, while the high-to-low transition is defined by the hysteresis value. All comparators within a group share a common hysteresis setting. Hysteresis can be enabled and configured for a CSG group by using the HPPASS_CSG_CSG_GROUP_CFGx.HYSTERESIS_EN_BANK and HPPASS_CSG_CSG_GROUP_CFGx.HYSTERESIS_SEL register fields, respectively.

31.2.3.6 Post processing

The Figure 413 shows the hardware resources in the CSG comparator post processing. Each function can be enabled independently and hence bypass is not required.

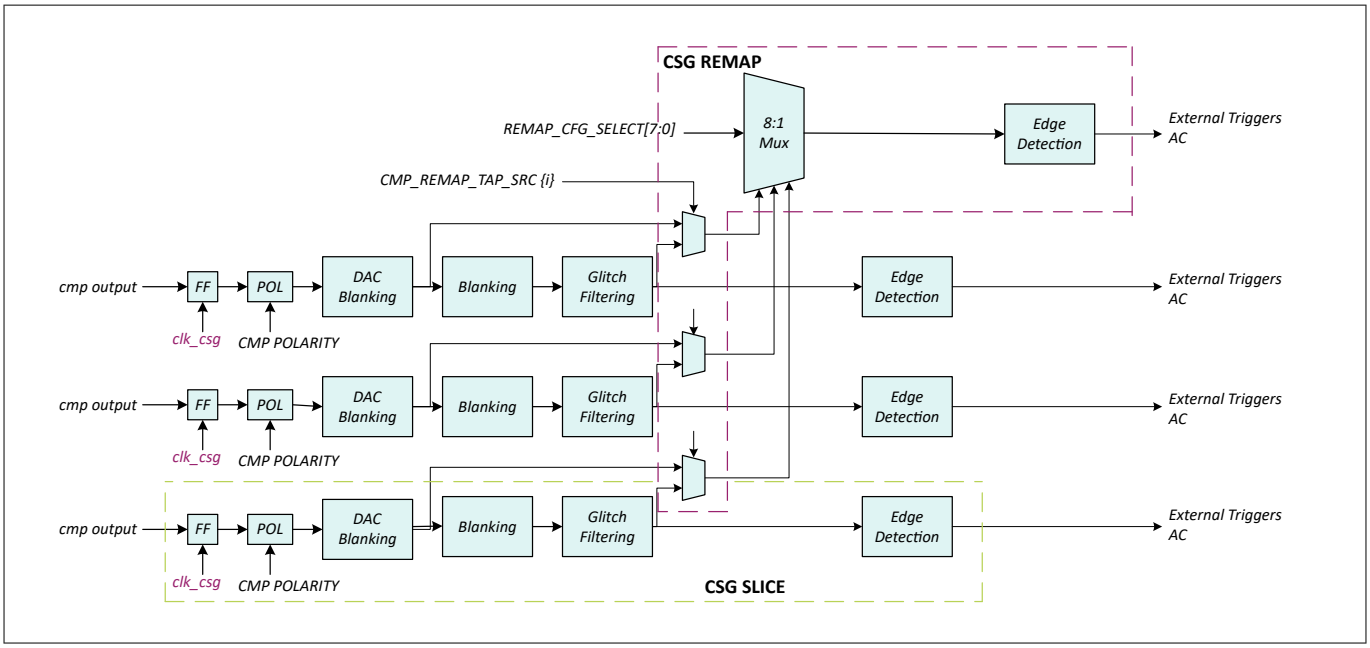


Figure 413 CSG comparator post processing

The main features of the CSG comparator post processing are as follows:

- Configuring of comparator output polarity
- Comparator blanking:
  - Gate the comparator output to ‘0’ using an input trigger
  - Blank the comparator during subsequent output changes following a rising and/or falling edge transition
- DAC code change blanking:
  - Blank the comparator output while the DAC code is changing during normal CSG operation
  - Separate controls to blank the comparator output in the mode where the DAC output is fed back to the ADC
- Deglitching the comparator output that may occur due to a DAC code change; the comparator output is latched for a programmable number of clock cycles
- Generating a configurable (positive edge, negative edge, or both edges) comparator interrupt and/or AC status
- Ability to select the comparator output or inverted comparator output as an update trigger (for example, hysteresis mode usage)
- Generating a configurable comparator interrupt
- Providing an activation trigger feedback for a slice and for the AC CMP condition
- Providing a trigger for the CSG DAC updates

The comparator output polarity and edge mode are separately configured by CMP_POLARITY and CMP_EDGE_MODE bit-fields of the HPPASS_CSG_SLICEx_CMP_CFG register. The comparator output value of every CSG slice can be read by the CMP_VAL bits of the HPPASS_CSG_SLICEx_CMP_STATUS register.

Comparator blanking

When the power switches controlled by the PWM are initially turned on, there can be a large current transient that causes the comparator to trip prematurely. Therefore, a comparator blanking capability is provided via an input trigger, which must be a level trigger.

There are two independent modes of blanking the comparator output:

1. One mode allows a programmable number of CSG clock cycles to be preconfigured to hold and then blank the comparator output. This mode is referred to as edge-based blanking. Blanking is triggered by a preprogrammed trigger (HPPASS_CSG_SLICEx_CMP_CFG.CMP_BLANK_TR_SEL). The blanking windows are programmed separately for the rising edge (HPPASS_CSG_CSG_GROUP_CFGx.RISING_EDGE_DELAY_BLANK) and falling edge (HPPASS_CSG_CSG_GROUP_CFGx.FALLING_EDGE_DELAY_BLANK) of the blanking trigger. To enable this mode, configure the HPPASS_CSG_SLICEx_CMP_CFG.CMP_BLANK_MODE field to DISABLED.

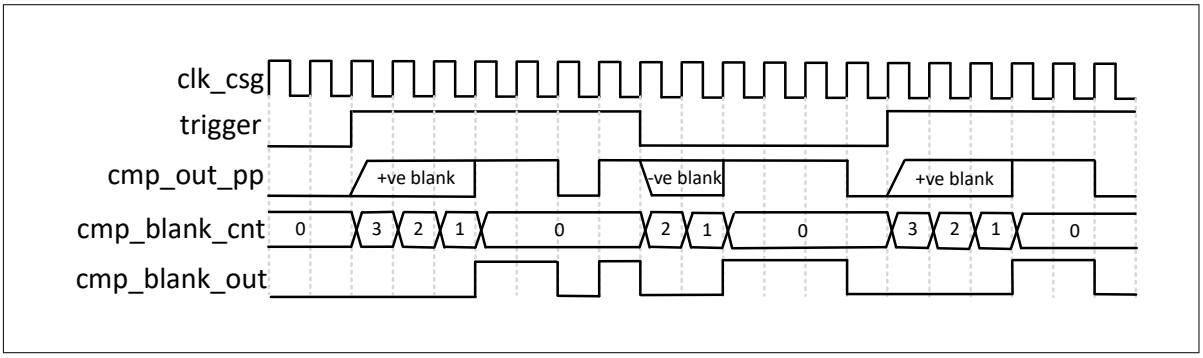


Figure 414 Comparator rising or falling edge blanking operation

A rising or falling edge is detected on the trigger, and a counter is started that holds the comparator output until the counter expires, as is shown in the Figure 415.

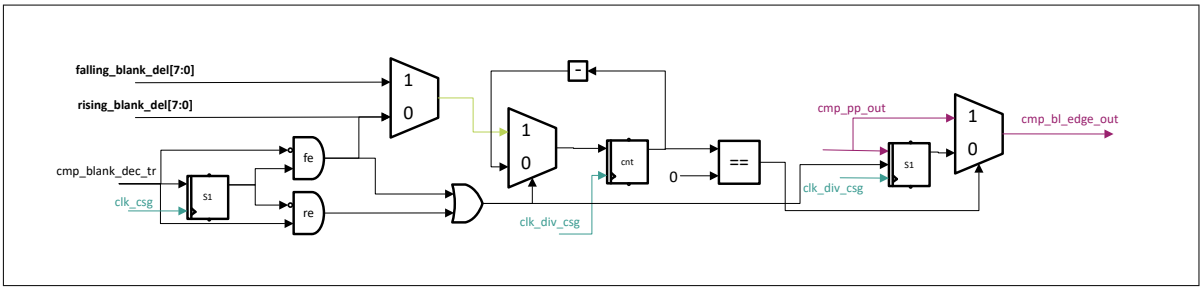


Figure 415 Comparator counterbased blanking logic

2. The other blanking mode gates the comparator output to “0” when a preprogrammed trigger (HPPASS_CSG_SLICEx_CMP_CFG.CMP_BLANK_TR_SEL) goes high or low. The blanking level (high or low) is selected using the HPPASS_CSG_SLICEx_CMP_CFG.CMP_BLANK_MODE register field. Selecting a blanking level enables this mode. If neither GATE_ON_ONE nor GATE_ON_ZERO is selected in the HPPASS_CSG_SLICEx_CMP_CFG.CMP_BLANK_MODE register, operates according to the first mode, edge-based blanking. The blanking operation is shown in the Figure 416.

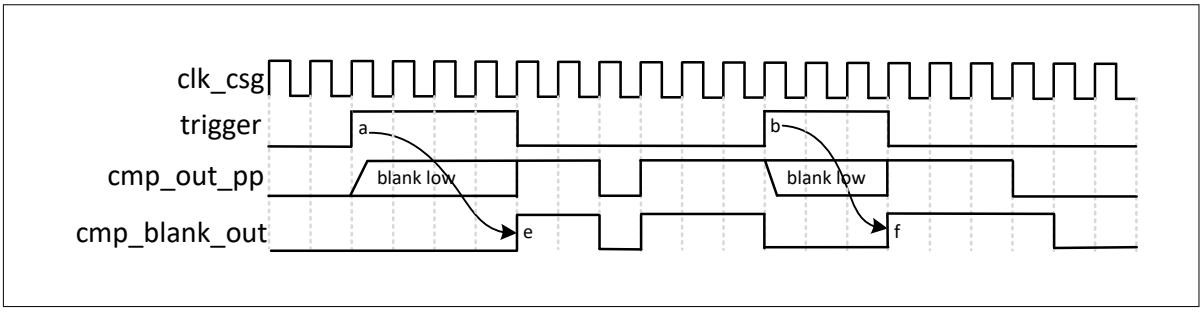


Figure 416 Comparator gate to “0” blanking

The logic for gate to “0” with a trigger is shown in Figure 417. The multiplexor selects between gate to “0” blanking and counter-based blanking.

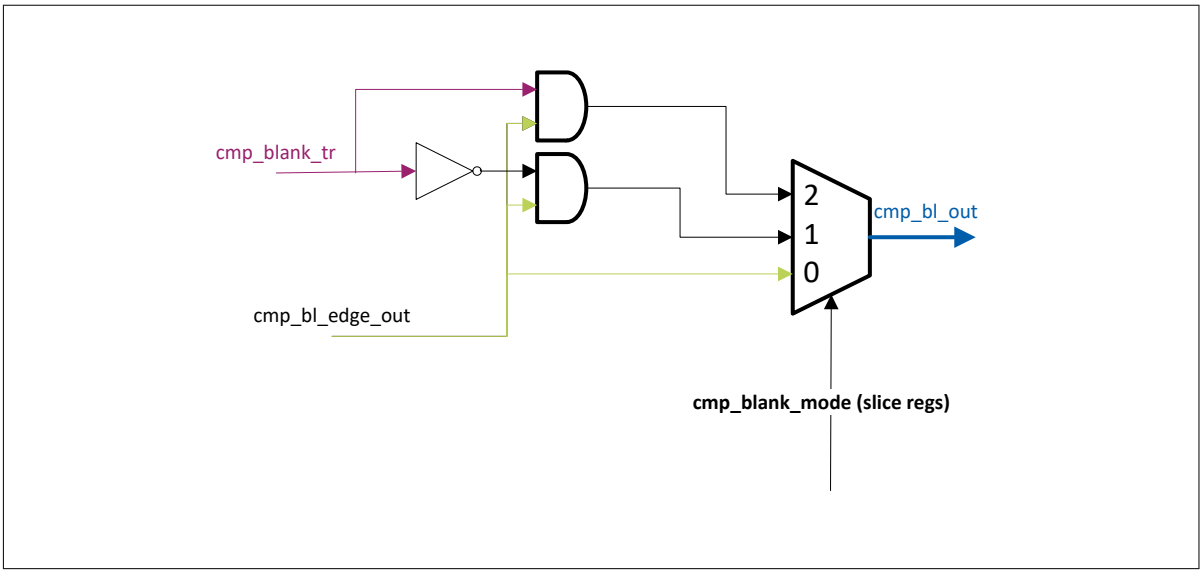


Figure 417 Comparator trigger gate to “0” based blanking logic

***Note**: To add more range to the blanking period, the clock can be divided using the register field HPPASS_CSG_CSG_GROUP_CFGx.CLOCK_DIVIDER.*

Comparator deglitching

The comparator deglitching feature is triggered by a change in the comparator level. Once a change is detected, the comparator level is stored and is not allowed to change again until it remains stable for a preprogrammed number of clock cycles, as defined by HPPASS_CSG_CSG_GROUP_CFGx.CMP_GLITCH_PERIOD. The Figure 418 shows how the comparator deglitching operates.

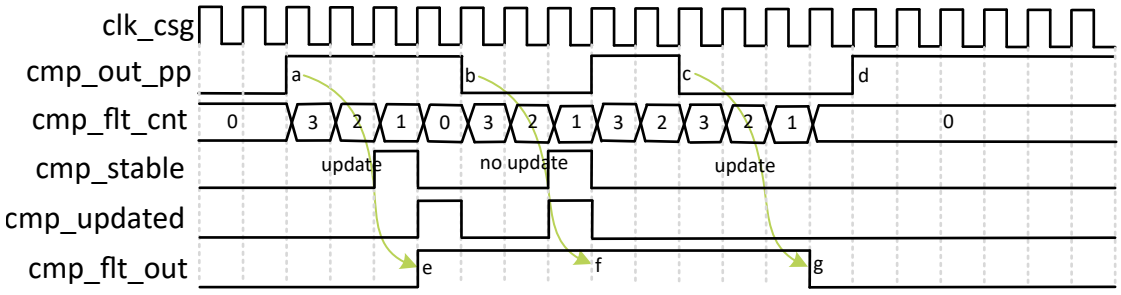


Figure 418 Comparator deglitching operation

Whenever the comparator changes state, the deglitch counter starts and the current comparator output is stored. If the comparator output remains stable for the required number of cycles, the stored value propagates to the output. The Figure 419 shows how the logic is implemented for the deglitch filtering.

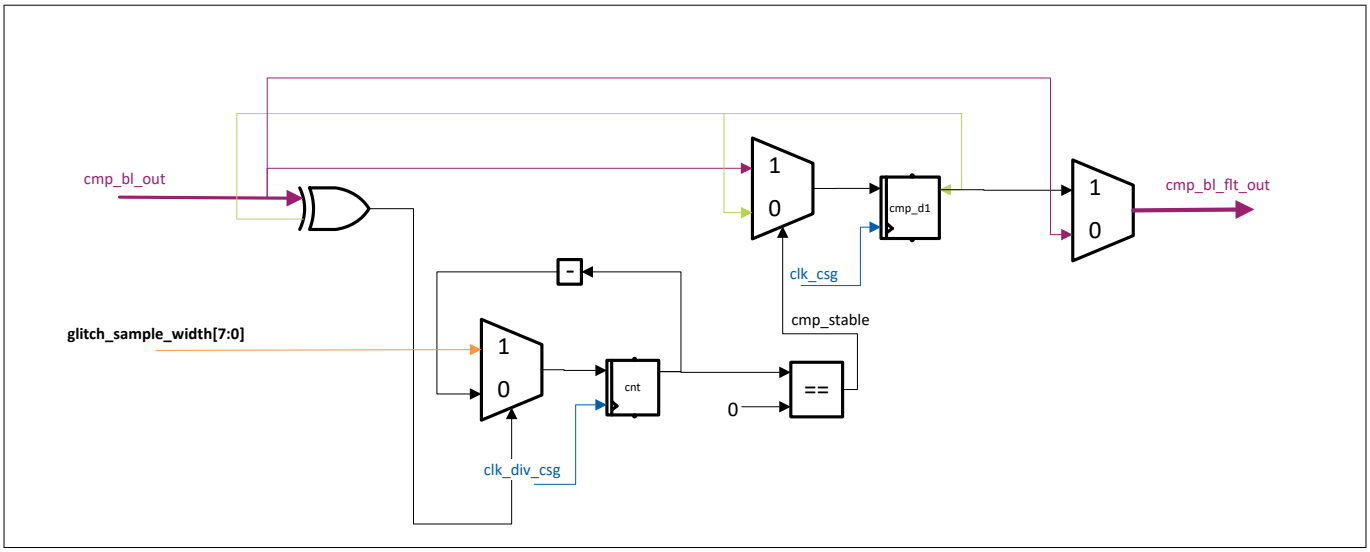


Figure 419 Comparator deglitching logic

DAC update glitch filter

Due to possible transients generated during a DAC update, one or more cycles of the comparator output may need to be suppressed from propagation. Therefore, an optional DAC update glitch filter is provided to latch the current comparator output value for a programmable number of cycles. The number of cycles is configured using the HPPASS_CSG_SLICEx_DAC_CFG.DAC_DEGLITCH bit-fields. Figure 420 shows an example of comparator output latch on DAC update.

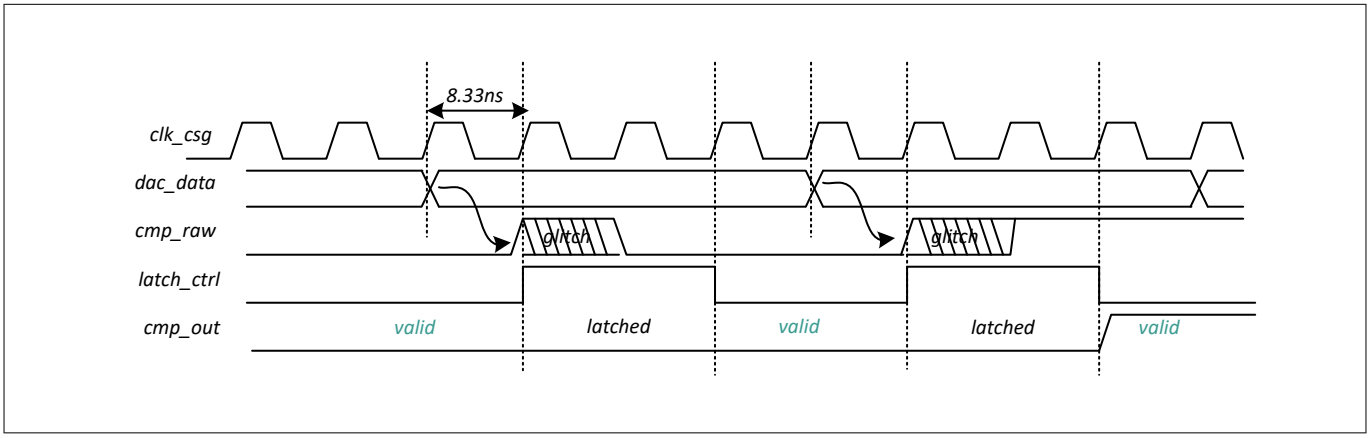


Figure 420 Comparator latch on DAC update

When new data is presented to the DAC, starting with the following cycle (where the new output is propagated), the output of the comparator is latched (the output retains the current value) for the programmed number of cycles. The number of cycles is programmed to two in the above example where the CSG clock is running at 120 MHz.

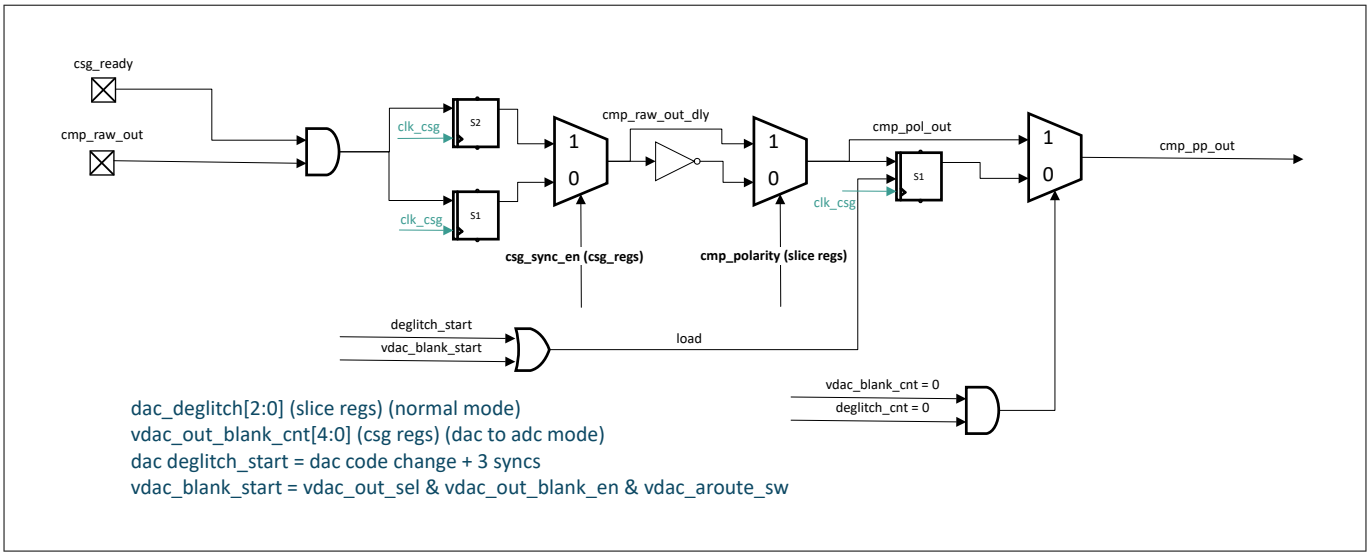


Figure 421 DAC deglitching and observability blanking logic

The raw comparator output from the analog CSG can be captured in the digital domain using either a single register or a double register. The comparator signal can then be inverted, if required, before the DAC deglitch processing.

Note: The HPPASS_CSG_CMP_INTR register reports interrupt status for both the raw slice comparator outputs and the remapped comparator outputs. The raw comparator status is reported in CMP[8:0], while remap interrupt status is reported in CMP_REMAP[11:9].

DAC observability blanking

Figure 422 shows the signal path between CSG DAC output observability to the SAR ADC sampler input. In the event where any two of these switches in series are closed in this path, there may be a short transient on the DAC output, which may cause an incorrect comparator output value.

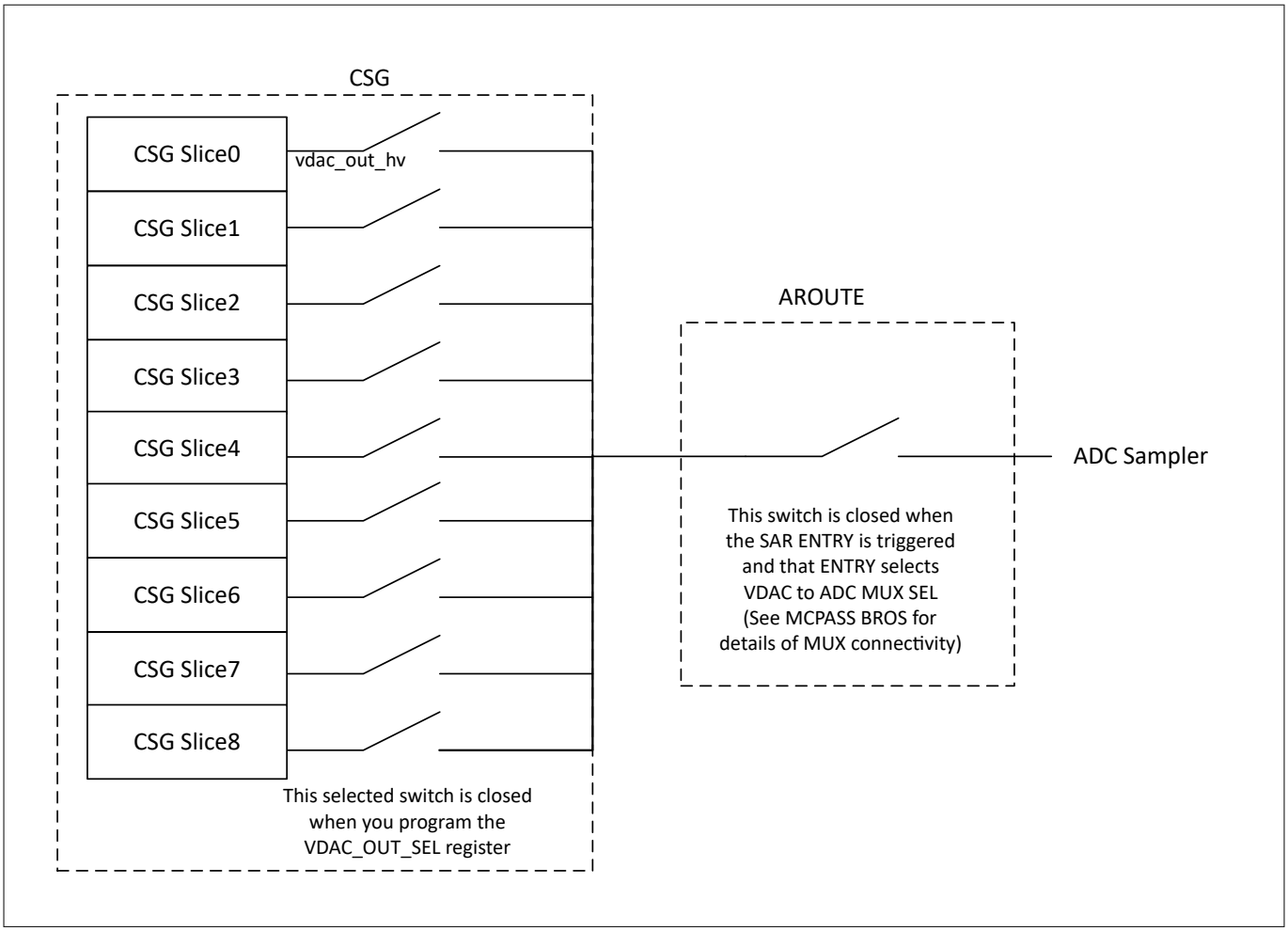


Figure 422 DAC observability routing

To mitigate this issue, use the comparator blanking feature by manually triggering it in software before changing the VDAC_OUT_SEL_MUX setting.

Comparator remapping

The nine comparator outputs from the CSGs can be combined to create three new output triggers. The comparator signals can be taken either from the DAC blanking output or from the comparator deglitching logic to generate the remapped CSG outputs. Comparator outputs 0-2, 3-5 and 6-8 are combined into three independent groups to generate three new triggers. Accordingly, `csg_comp_out[2:0]` maps to the preprogrammed HPPASS_CSG_CSG_OUT_REMAP_CFGx.REMAP_CFG_SELECT[7:0] settings, as described below.

Table 326 Comparator remap input and bit selection

CSG CMP [0]	CSG CMP [1]	CSG CMP [2]	Register Bit Selected
0	0	0	REMAP_CFG_SELECT[0]
0	0	1	REMAP_CFG_SELECT[1]
0	1	0	REMAP_CFG_SELECT[2]
0	1	1	REMAP_CFG_SELECT[3]
1	0	0	REMAP_CFG_SELECT[4]
1	0	1	REMAP_CFG_SELECT[5]
1	1	0	REMAP_CFG_SELECT[6]

(table continues...)

Table 326 (continued) Comparator remap input and bit selection

CSG CMP [0]	CSG CMP [1]	CSG CMP [2]	Register Bit Selected
1	1	1	REMAP_CFG_SELECT[7]

As shown in the Table 326 based on the CSG_CMP inputs diagram, the bit written in the HPPASS_CSG_CSG_OUT_REMAP_CFGx.REMAP_CFG_SELECT register will get selected. For example, if CSG_CMP[0] = 0, CSG_CMP[1] = 1 and CSG_CMP[2] = 1, then the bit 3 of the REMAP_CFG_SELECT is selected. See Figure 423.

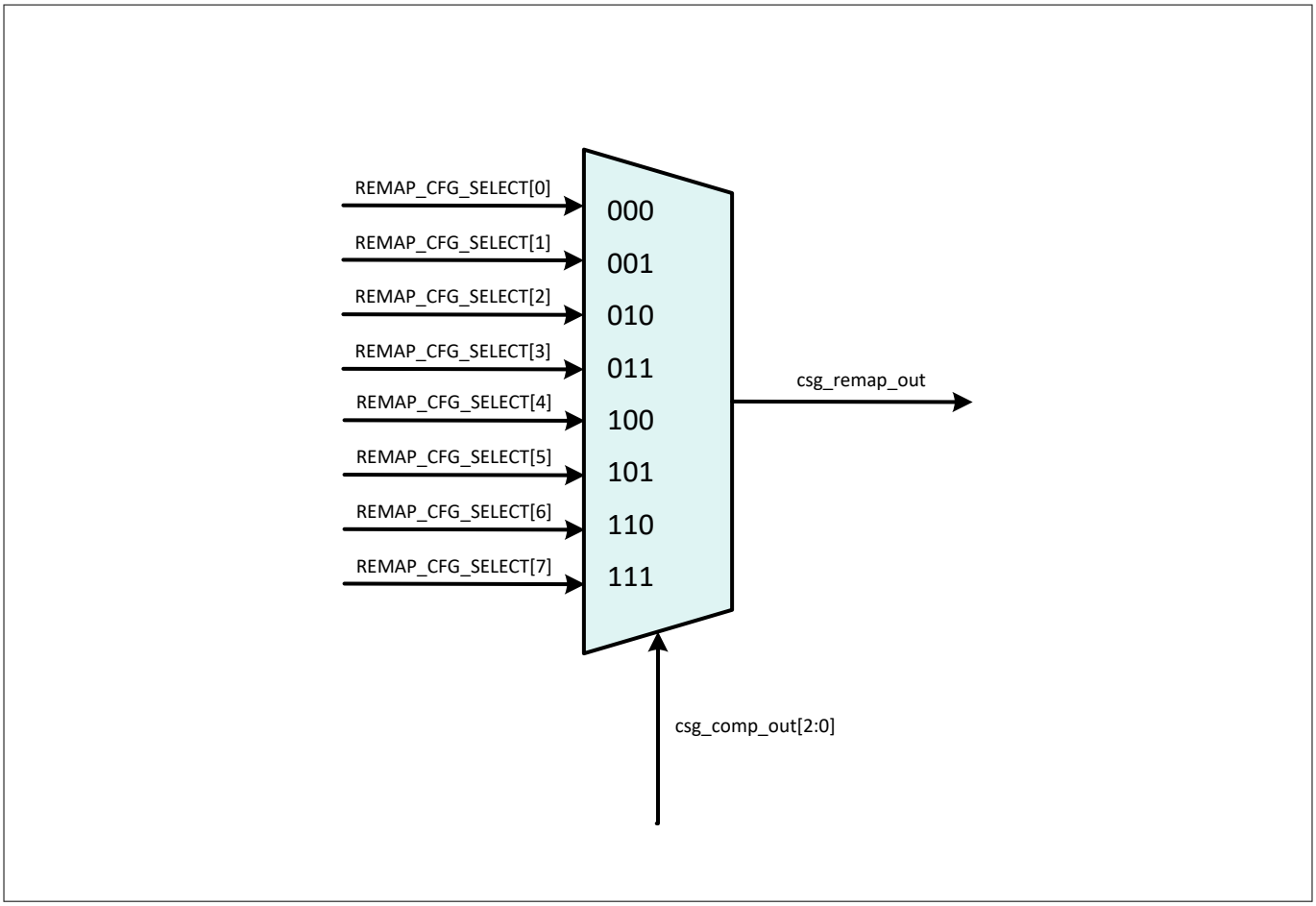


Figure 423 Comparator remap decoder

The inputs to the remapped multiplexor must be level-based and can optionally be edge detected using the HPPASS_CSG_CSG_OUT_REMAP_CFGx.CMP_REMAP_EDGE_MODE register field to create external trigger pulses.

If the output state of the remapped comparator changes while edge detection is enabled, the transition is treated as a new trigger. This transition causes the remapping configuration select to change from 0 to 1 or from 1 to 0, even though no actual input trigger has occurred.

31.2.3.7 CSG interrupts

The HPPASS CSG block has DAC and comparator interrupts to the CPU interrupt controller. A DAC interrupt is generated when the DAC mode starts with a hardware start trigger, DAC slope/LUT done, or DAC buffer empty. The DAC interrupt is enabled by setting the HPPASS_CSG_DAC_INTR_MASK register, and reads the HPPASS_CSG_DAC_INTR_MASKED register to know the interrupt masked status forwarded to the CPU interrupt controller.

An individual interrupt is provided for each CSG DAC (interrupt_csg_dac[8:0]), as well as a combined DAC interrupt (interrupt_csg_dacs) (see Interrupts chapter for more details). If the combined interrupt is used, the HPPASS_CSG_DAC_INTR registers must be read to determine the cause of the interrupt.

The CSG comparators output interrupt only have a combined interrupt, and can read the HPPASS_CSG_CMP_INTR register to know which CSG comparator interrupt is generated. The specified CSG comparator interrupt type can be selected to disabled, rising edge, falling edge, or both edges by setting the CMP_EDGE_MODE bit-fields of HPPASS_CSG_SLICEx_CMP_CFG register, enabled by setting the CMP specified bit of the HPPASS_CSG_CMP_INTR_MASK register, and reads the HPPASS_CSG_CMP_INTR_MASKED register to know the interrupt masked status forwarded to the CPU interrupt controller.

31.2.4 3P3Z Filter (3P3ZFILT)

HPPASS features a programmable 3 poles, 3 zeros filter implemented in hardware. It has 3 structures of compensation filter with support for filter data input manipulation, coefficient scaling, anti-windup and filter data output manipulation like offset and range limit. The filter input data stream can be sourced either from the AHB interface or from a dedicated data path connected to the ADC output.

31.2.4.1 Overview

Each filter structure has the following features:

- 3 poles and 3 zeroes
- Two 16-bit inputs in q0.15 format
- 24-bit output in q0.23 format
- 24-bits coefficients in q0.23 format
- 48-bit multiplier
- 58-bit internal accumulator
- 2-bit gain input and 3-bit attenuator output
- Saturation and rounding operations supported
- Two’s-complement arithmetic
- Anti-windup support in q0.23 format
- Limit output support in q0.23 format
- Offset + output support in q0.23 format
- Maximum execution latency of 9 CPU clock cycles
- Option to start filter processing when data is written to either of the two data registers
- Generates a pulse event upon completion of filter computation
- Register slave interface (connected to ADC outputs)

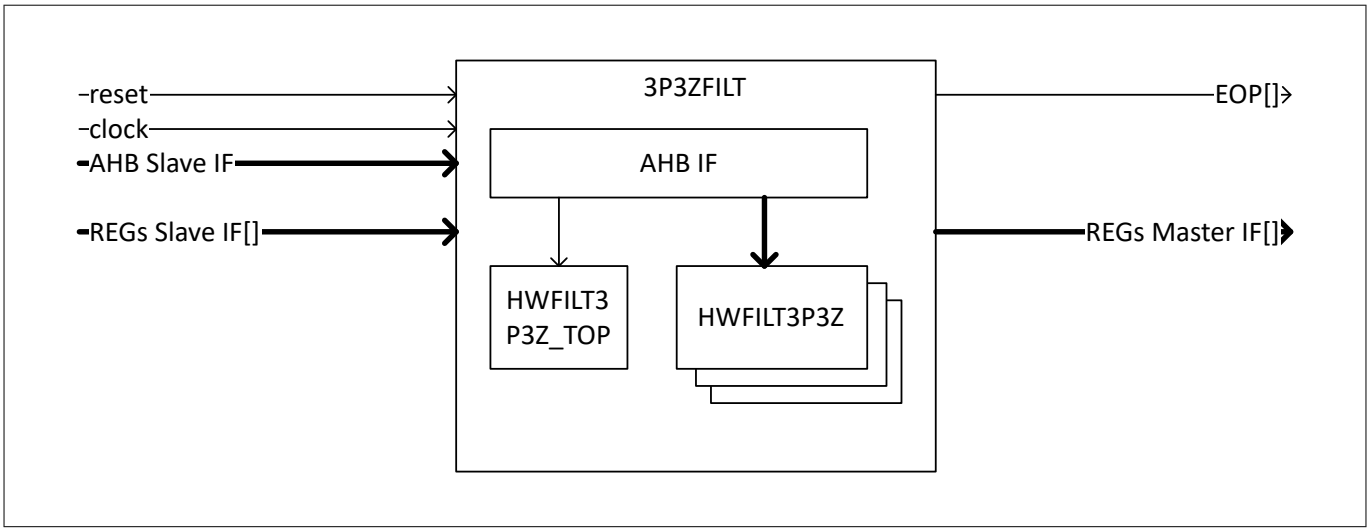


Figure 424 Hardware filter block diagram

All filter data registers are sign-extended to a full 32-bit width when read through the AHB interface.
Hardware filter equations:

$$
\begin{aligned}
&Qxin,\quad Qxref \;=\; 16,\quad Qa,\quad Qb \;=\; 24,\quad Qy \;=\; 24\\
&x(n) = \left(x_{ref} - xin(n)\right) <\!< Gin\\
&acc(n) = \left(\Sigma_{i=0}^{3}\left(b_i \cdot x(n-i)\right) + \Sigma_{j=1}^{3}\left(a_j \cdot y(n-j)\right) + 2^{-Q_y}\right)\\
&y\left(n\right) = \begin{cases} trunc\left(acc(n) <\!< \left(Q_y - 1\right)\right), & -1 \le acc(n) < 1\\ 2^{(Qy-1)} - 1, & acc(n) \ge 1\\ -2^{(Qy-1)}, & acc(n) <\!< -1 \end{cases}\\
&y_AWout\left(n\right) = \begin{cases} y_n(n), & AMin \le y_n(n) \le AMax\\ AMax, & y_n(n) > AMax\\ AMin, & y_n(n) < AMin \end{cases}\\
&y(n) = (y_AWout(n) + (y_AWout(n) - y_n(n)) \;\; * GAIN)\\
&y_tmp(n) = (trunc(y_AWout(n) >\!> Gout) + Offset)\\
&yout\left(n\right) = \begin{cases} y_tmp(n), & LimMin \le y_tmp(n) \le LimMax\\ LimMax, & y_tmp(n) > LimMax\\ LimMin, & y_tmp(n) < LimMin \end{cases}
\end{aligned}
\tag{6}
$$

Figure 425 shows the hardware filter block diagram with registers used:

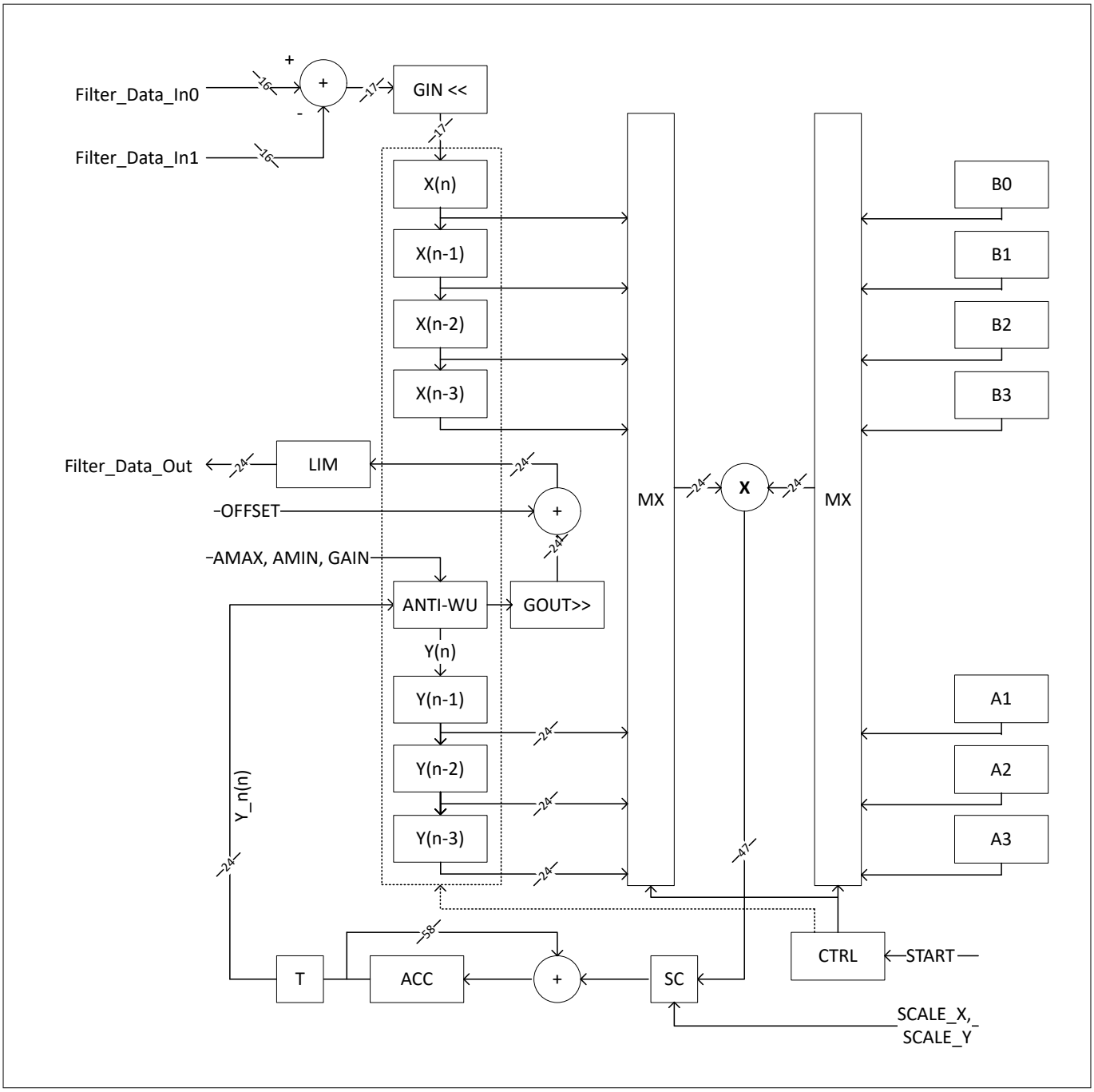


Figure 425 Hardware filter processing block diagram

31.2.4.2 Trigger filter processing

The filter processing can be triggered in the following ways:

1. If `HPPASS_HWFILT3P3Zx_CNFG.EN_TRIG0 = 1`, the processing is started by writing into `HPPASS_HWFILT3P3Zx_DATA_IN0` register
   a. If `HPPASS_HWFILT3P3Zx_CNFG.SRC_SEL0 = 1`, the processing is started by the trigger signal in the register slave

interface. Writes to the HPPASS_HWFILT3P3Zx_DATA_IN0 register update the register contents but do not trigger filter processing

b. If HPPASS_HWFILT3P3Zx_CNFG.SRC_SEL0 = 0, the processing is started by writing to the HPPASS_HWFILT3P3Zx_DATA_IN0 register

2. If HPPASS_HWFILT3P3Zx_CNFG.EN_TRIG1 = 1

a. If HPPASS_HWFILT3P3Zx_CNFG.SRC_SEL = 1, the processing is started by the trigger signal in the register slave interface. Writes to the HPPASS_HWFILT3P3Zx_DATA_IN1 register update the register contents but do not trigger filter processing

b. If HPPASS_HWFILT3P3Zx_CNFG.SRC_SEL = 0, the processing is started by writing to the HPPASS_HWFILT3P3Zx_DATA_IN1 register

***Note**: HPPASS_HWFILT3P3Zx_CNFG.EN_TRIG0 and HPPASS_HWFILT3P3Zx_CNFG.EN_TRIG1 should not be set at the same time.*

31.2.4.3 Anti-Windup

The Anti-Windup (ANTI-WU) feature is supported through both the clipping method and back-calculation method. Options are provided to configure the anti windup max limit, anti-windup min limit, and anti-windup gain. The anti windup feature can be disabled by setting the anti-windup limits registers (AWMAX and AWMIN) to their maximum absolute value, and by setting the GAIN value to “0”. The anti-windup module is shown in Figure 426

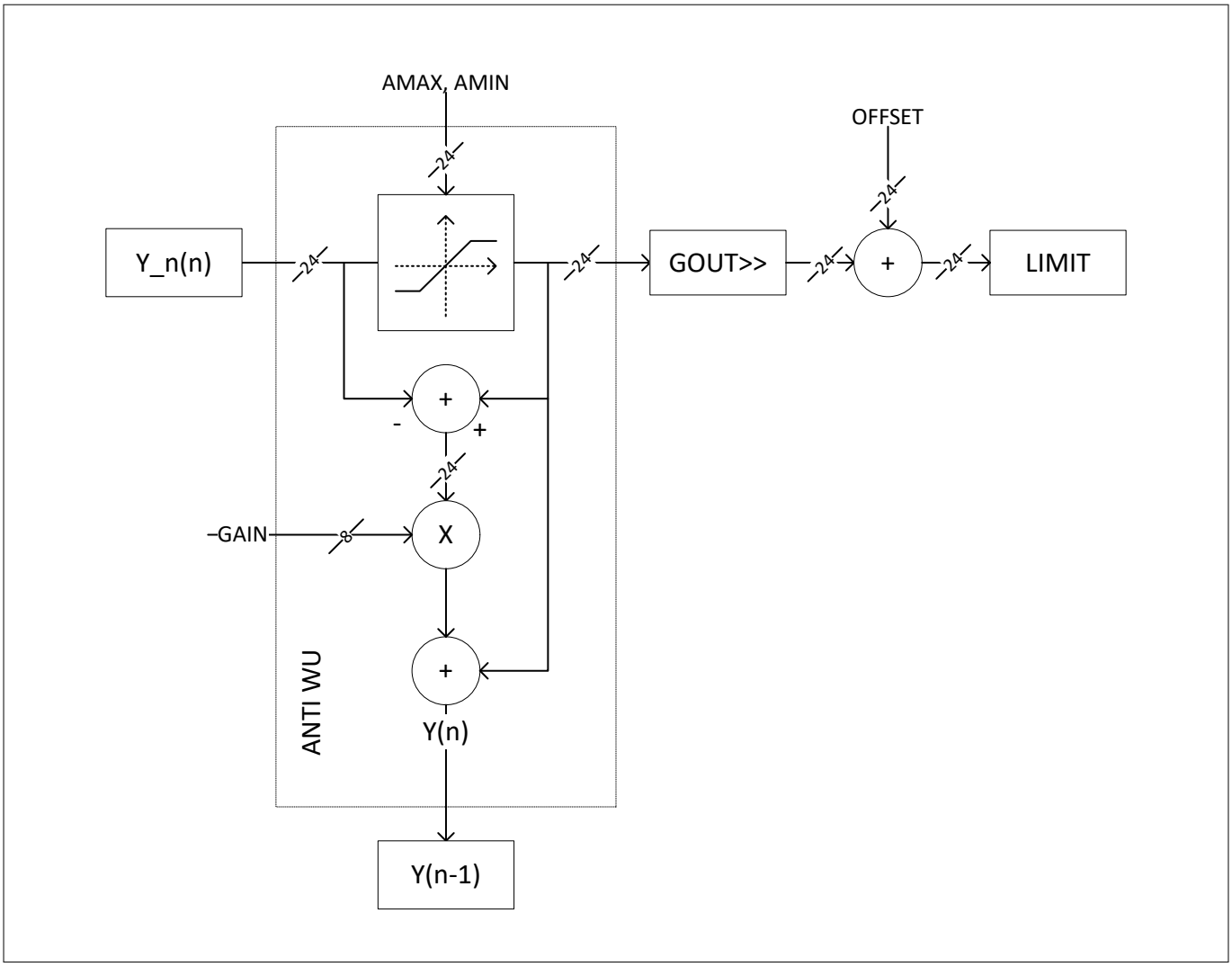


Figure 426 Anti-Windup module

31.2.4.4 Data path connection

The 3P3ZFILT also supports a dedicated data path connection, in which an embedded register is fed directly from the output of the ADC module through the register interface.

The following figure shows the ADC data passing to the 3P3Z filter block and the subsequent data flow from the 3P3Z filter to TCPWM block. It can drive a new duty cycle of PWM without CPU intervention.

The Figure 427 shows the dedicated data path connection in which the ADC output is directly fed to the 3 filter structures. Selection of the ADC channel input for filter instance is configured through the HPPASS_SAR_CFG_HWFILT3P3Z_IND_CTRLx register.

ADC channel selection for filter channels can be done by using the following registers,

3P3Z Filter 0 → HPPASS_SAR_CFG_HWFILT3P3Z_IND_CTRL

3P3Z Filter 1 → HPPASS_SAR_CFG_HWFILT3P3Z_IND_CTRL1

3P3Z Filter 2 → HPPASS_SAR_CFG_HWFILT3P3Z_IND_CTRL2

each allow section of one of SAR channels 0–35 for its corresponding 3P3Z input.

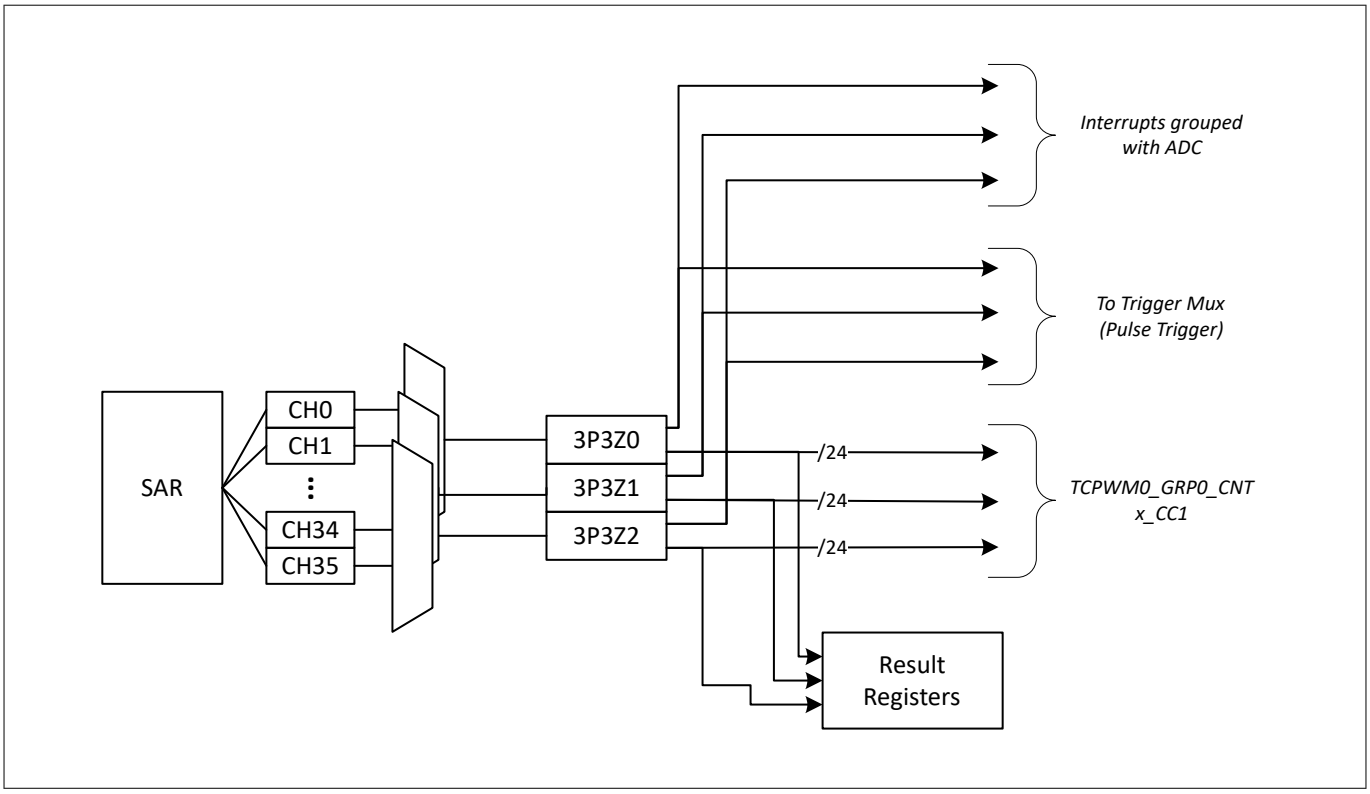


Figure 427 Dedicated data path

31.2.5 Digital to analog converter (DAC)

HPPASS consists of two 12-bit R2R DACs with output buffers that can drive a load capacitance of up to 100 pF with a maximum load current of 10 mA. Multiple methods are supported for supplying data to the DAC input, similar to those used by the CSG logic. The DAC can use either external references or supply as reference based on the input configuration. The MCU output pin is driven by a rail-to-rail 10 mA buffer.

The DAC exhibits very low code transition glitch of < 2mV, ensuring output has almost monotonic code transition voltage shift. In addition to driving an external pin, it can also be internally routed to drive an ADC input.

Please refer Figure 385 for the pin details.

31.2.5.1 Features

- 12-bit R2R monotonic DAC
- 1 MHz maximum DAC update rate
- Settling time of 1 us and accuracy of +/-1LSB
- Less than 2 mV glitch during code transition
- Drive a current load up to 10 mA and capacitive load of up to 100 pF
- Direct and buffered write to the DAC input
- Waveform generation using hysteresis mode and look up table
- FIFO with depth of 8 elements and interrupt generation
- ADC data loop-back mode

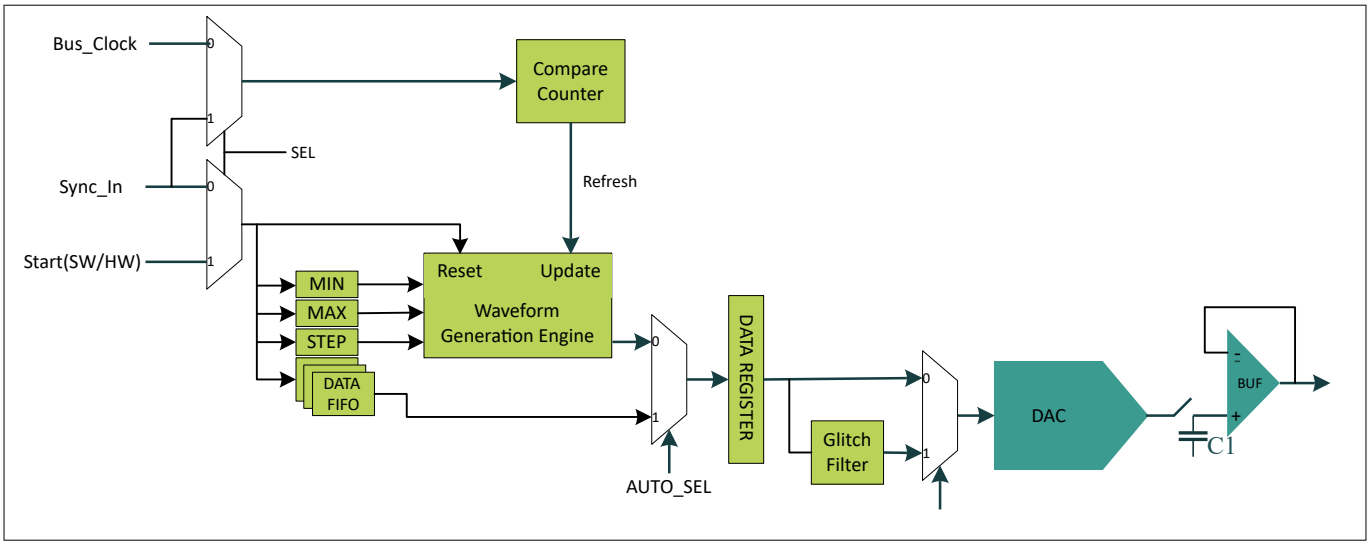


Figure 428 DAC with buffer

This component has two sections, a DAC digital and DAC Analog (DAC and buffer). The DAC digital is shown in the diagram in green colour and the DAC analog is shown in blue colour.

31.2.5.2 DAC Digital

The following features are provided by DAC digital:

- **Direct Write Data**: An AHB accessible DAC_VAL register that directly drives the DAC code
- **Hysteresis/Buffered Mode:** Triggered buffered data with two value (Hysteretic) control of DAC code
- **LUT Mode**: Buffered firmware control of DAC code
- **Slope/Waveform generation Mode:** Automatic DAC ramp generation

In addition to these features, the DAC can also be driven using the following options:

- **FIFO Mode:** Firmware updates the FIFO with DAC code to be registered into DAC_VAL
- **ADC Loopback Source:** For debugging and functional verification, a loopback path is provided from ADC channels to the DACs

Note:

- Maximum DAC update rate is 1 MHz.
- DAC reference source selection is controlled by REF_SEL (see register reference manual).
- DAC output drive capability is limited to 10 mA and a 100 pF load.

Figure 429 shows the DAC digital block diagram,

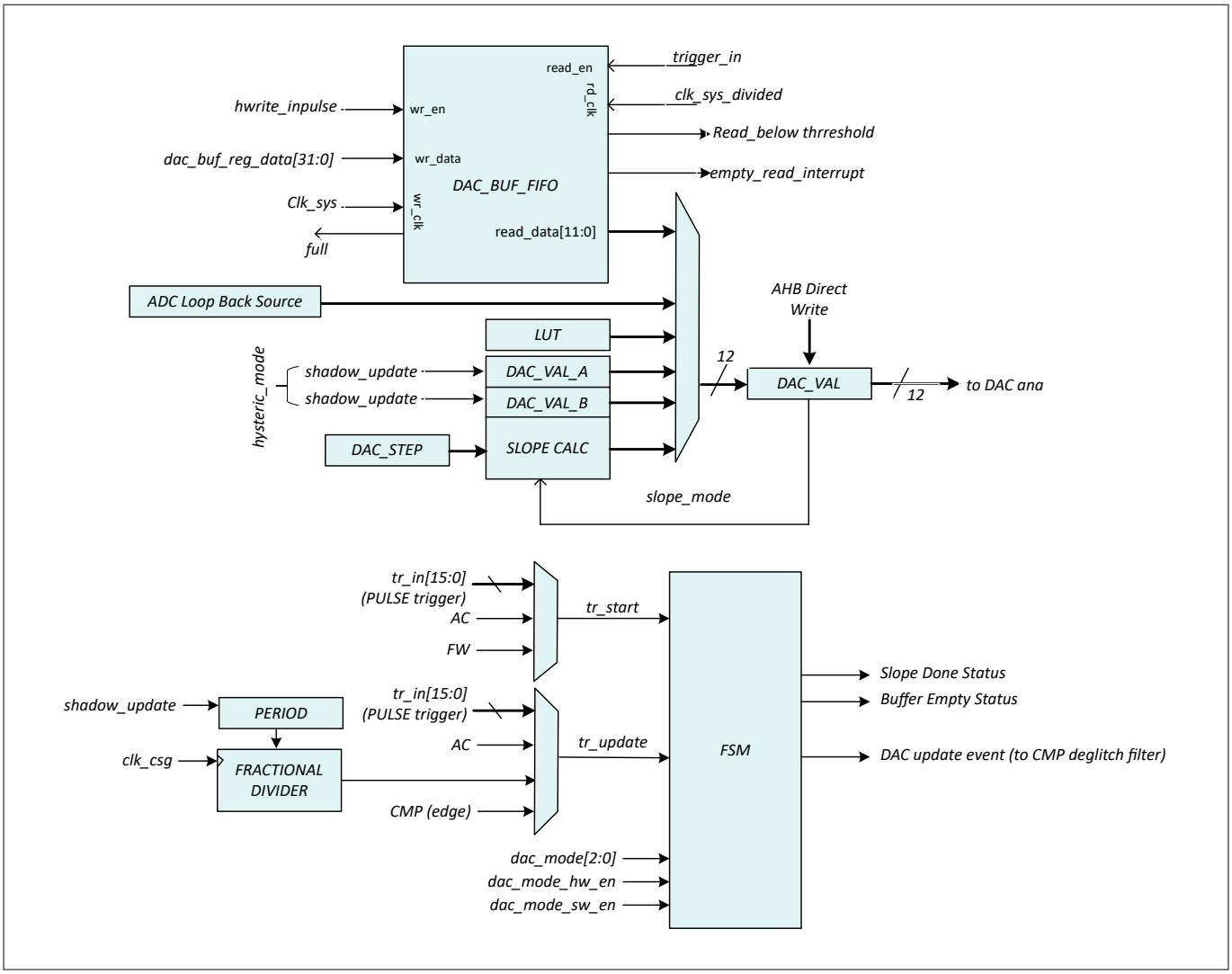


Figure 429 DAC_Buffer block diagram

In the simplified hardware diagram, the resources used to implement the main features of the DAC code-generation logic are shown.

The features supported include:

- A DAC clocked by a divided-down version of `clk_hf`
- An AHB accessible `DAC_VAL` register which directly drives the DAC code
- Two DAC Buffer registers “A” and “B”, whose function depends on the selected mode (see section on DAC modes)
- A programmable slope generator, with parameters: START, END, STEP, PERIOD
- A period counter with fractional capability, used to precisely modulate the slope of a given ramp or waveform
- A START and UPDATE trigger which can be HW, FW, or AC (or PERIOD for UPDATE)
- Status for `RAMP_DONE` and `BUFFER_EMPTY`, which can be used for interrupt generation
- If an error is detected in the configuration of the DAC slope or DAC LUT modes, the DAC finite state machine (FSM) returns to the idle state, and no interrupt is generated under the following conditions:
  - (start > end) or ((step_size > (start – end)) in forward modes
  - (start < end) or ((step_size > (end – start)) in reverse modes

The DAC has control registers for the various modes of operation, including DAC value, buffered DAC value, ramp start/end/step/period, LUT registers and FIFO write-only access through AHB interface. The DAC waveform period can be fractionally adjusted using the period counter. (the period can be averaged between 2 integer values to achieve fractional performance).

A start trigger is used to initiate the DAC and an update trigger is used to advance to the next code which can be driven by the hardware or AC for automatic control or it can be driven directly by the firmware.

The DAC value register is updated on this clock, this value represents the code sent to the DAC. One `clk_hf` cycle later, the `dac_buffer_ctrl_trigger` is asserted to the analog DAC to capture the updated DAC sample. This trigger is a gated pulse derived from `clk_hf`.

Interrupts can be enabled for ramp_done, buffer_empty and hardware start. The status of these events can also be polled through the status registers.

31.2.5.2.1 LUT

A 128-entry LUT is available for the DAC operation. To use the LUT, select `LUT_FORWARD(5)`, `LUT_REVERSE(6)`, or `LUT_FWD_REV(7)` in the register field `HPPASS_DACx_CFG.DAC_MODE`. Then configure the required pattern using the registers `HPPASS_DACx_LUT0` through `HPPASS_DAC0_LUT127`. Refer to the register reference manual for more details.

31.2.5.2.2 FIFO

The FIFO has a depth of 8 and a packet width of 12 bits. Data can be written to the FIFO using 32-bit AHB write accesses. To speed up programming, data samples can be paired in one write. Individual samples may be MSB or LSB-aligned and can be transferred either as single samples or as pairs of samples. These are single word write access and double word write access to the FIFO. In case of single write access, the write pointer increments by 1 for each write clocks and requires 8 write cycles to fill the FIFO. In contrast, double word write access causes the write pointer to increment by 2 for each write cycle, requiring only 4 write cycles to fill the FIFO. The four write options are shown in Figure 430.

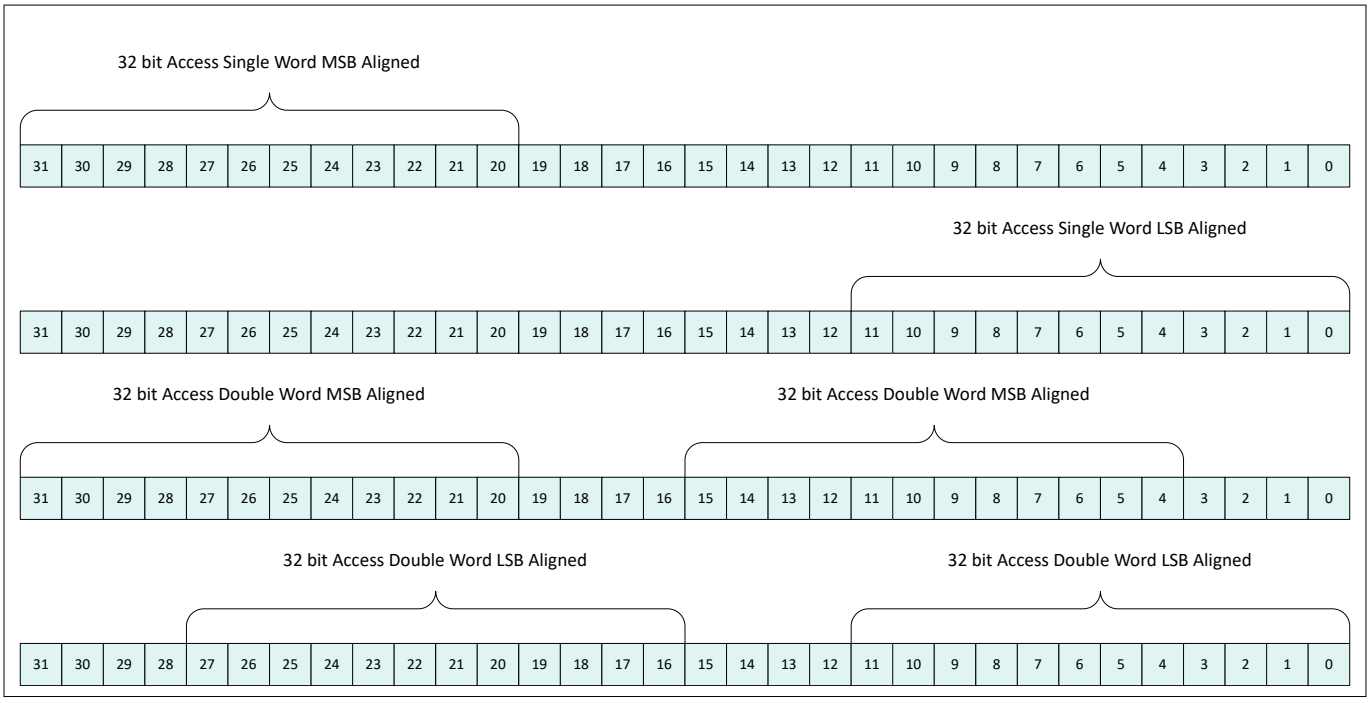


Figure 430 DAC write data alignment

The write-packet data is read from the DAC MMIO register. However, read operations always occur one entry at a time; so read pointer is always incremented by 1 for each valid trigger generated in DAC FSM logic while

operating in FIFO mode. The FIFO empty and full control logic is capable of handling both double-word and single-word writes. In case of double-word writes, the FIFO is set to 'full' if the available location in the FIFO is less than or equals to 1.

The FIFO can generate maskable interrupts when the FIFO depth reaches a low threshold of 3, as well as on overflow and underflow conditions. When the FIFO is empty, the FIFO output holds the last valid value.

The FIFO write operation is controlled based on the configuration of the DAC_BUFFER_FIFO_CTRL.ACCESS_MODE register. When the FIFO reaches the full condition, incoming AHB write transactions are dropped. The FIFO read operation is enabled when the FIFO becomes not empty.

***Notes**:*
When the DAC FIFO becomes empty, the DAC output does not retain the last converted value upon the next update trigger. To retain the last DAC output value when the FIFO is empty, use one of the following methods within the FIFO-empty interrupt service routine (ISR):

1. *Disable DAC trigger generation.*
2. *Refill the FIFO with the last DAC value.*
3. *Write the last DAC value directly to the **DAC_VAL** register.*

31.2.5.2.3 ADC Loop back control

Data from the ADC is looped back to DAC for both debugging and functional feature. At the DAC interface, the loopback data from ADC is updated to the DAC when the ADC ready signal is asserted high. This can be enabled by selecting the option ADC_LOOPBACK(9) in the HPPASS_DACx_CFG.DAC_MODE register filed. The ADC channel that needs to be looped back can be selected in ADC_LOOPBACK_CTRL register.

DAC FSM is added with new state as DAC_STATE_ADCLOOP_BACK to control the ADC loopback data update to DAC.

31.2.5.2.4 DAC input modes

The user has multiple options for controlling DAC data sequencing. The modes described below can be configured independently for each slice.

Direct Write

At any time, the user may write directly to the DAC_VAL register via the AHB bus, which directly drives the analog DAC input. This feature can be used to initialize the DAC code prior to enabling the block.

***Note**:* *Any write to this register disables any DAC mode currently in progress.*

Buffered Mode

The DAC_VAL_A register acts as a single buffer for the DAC_VAL register. When this mode is enabled and subsequent UPDATE triggers are asserted, the value stored in DAC_VAL_A is transferred to DAC_VAL and a DAC_BUF_EMPTY interrupt is generated. An interrupt is generated only when a new value is written to DAC_VAL_A and subsequently transferred to DAC_VAL. A possible usage scenario illustrating the operation of buffered mode is shown Figure 407.

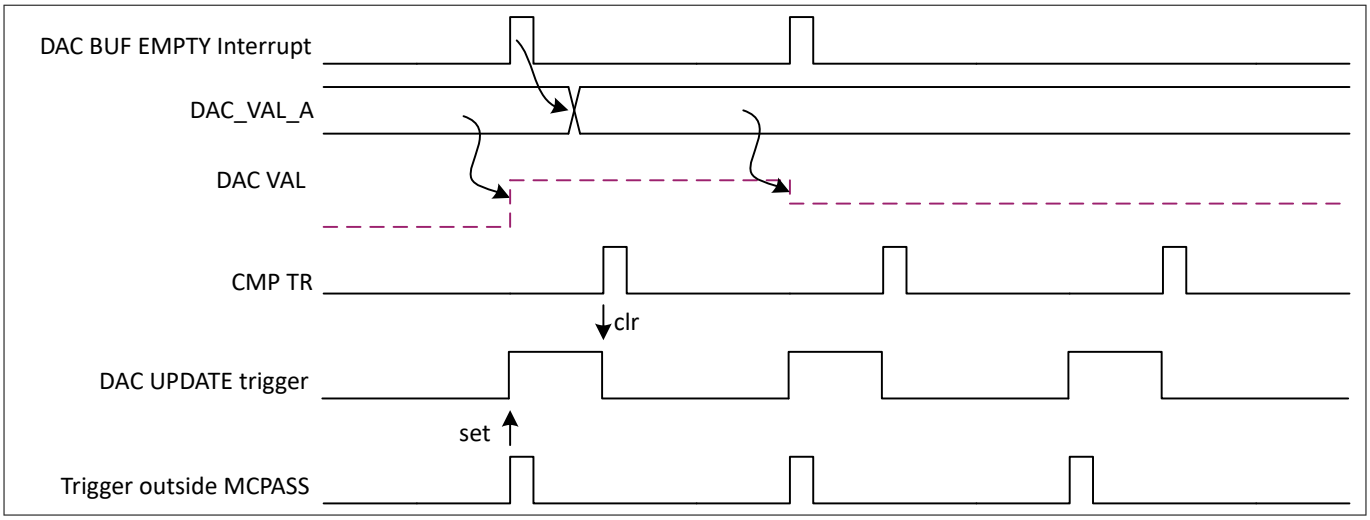


Figure 431 Buffered mode

Hysteretic Mode

In this mode, two buffered values DAC_VAL_A and DAC_VAL_B. This allows a high sense (A) and low sense (B) trip operation using a single comparator.

- When enabled, the high sense value DAC_VAL_A is transferred to DAC_VAL for output
- When the UPDATE trigger transitions from 0 to 1, DAC_VAL_B is transferred to DAC_VAL
- When the UPDATE trigger transitions from 1 to 0, DAC_VAL_A is transferred to DAC_VAL

On the DAC_VAL_A update, a DAC_BUF_EMPTY interrupt is generated, and the user may update the buffered values. The updated values are used by the hardware starting with the next DAC_VAL_A update, as shown in Figure 408.

In this mode, the comparator can be used as the update trigger and the user has the flexibility to use either the CMP output or invert it. By modulating the A and B values, both the **period** and **duty cycle** of the CMP output can be adjusted.

Info: the DAC_VAL_A and DAC_VAL_B are written on the rising edge on the interrupt

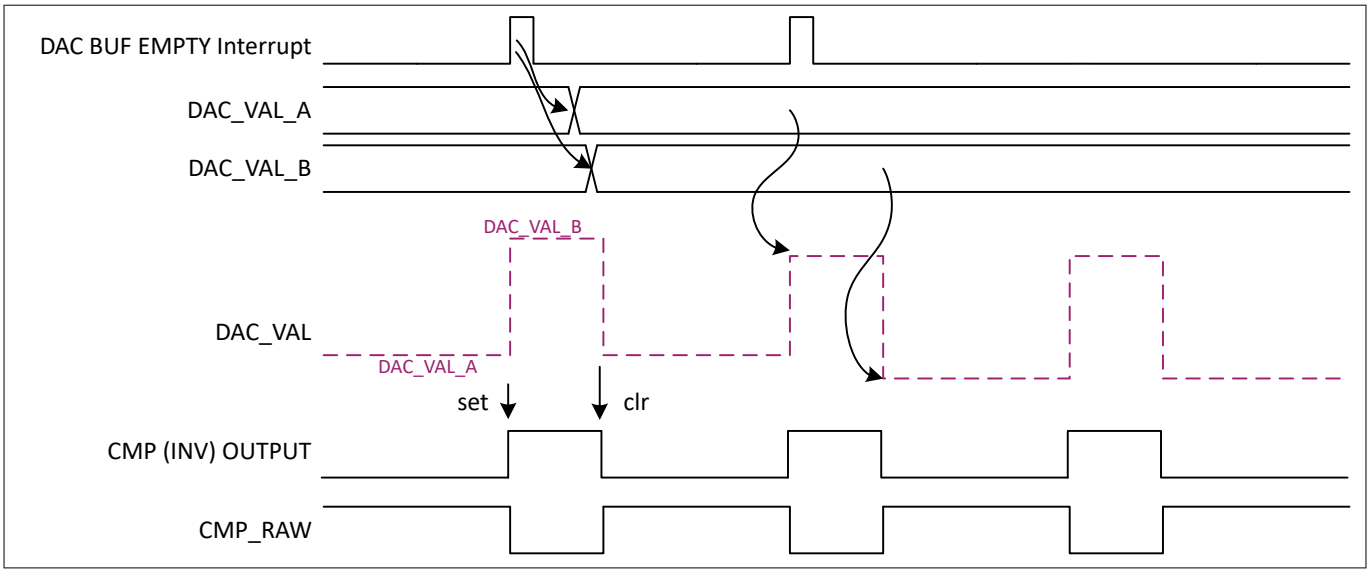


Figure 432 Hysteretic mode

Slope Generation Mode

The slope can be configured as

- RISING: Start at DAC_VAL_A, increment by DAC_STEP, and stop when DAC_VAL >= DAC_VAL_B

- FALLING: Start at DAC_VAL_A, decrement by DAC_STEP, and stop when DAC_VAL <= DAC_VAL_A
- TRIANGULAR: Start at DAC_VAL_A, increment by DAC_STEP until DAC_VAL >= DAC_VAL_B, then decrement by DAC_STEP until DAC_VAL <= DAC_VAL_A

As shown in the Figure 433, parameter updates only occurs at the start of a new slope. Any combination of the START (DAC_VAL_A), END (DAC_VAL_B), DAC_STEP and DAC_PERIOD is buffered for update.

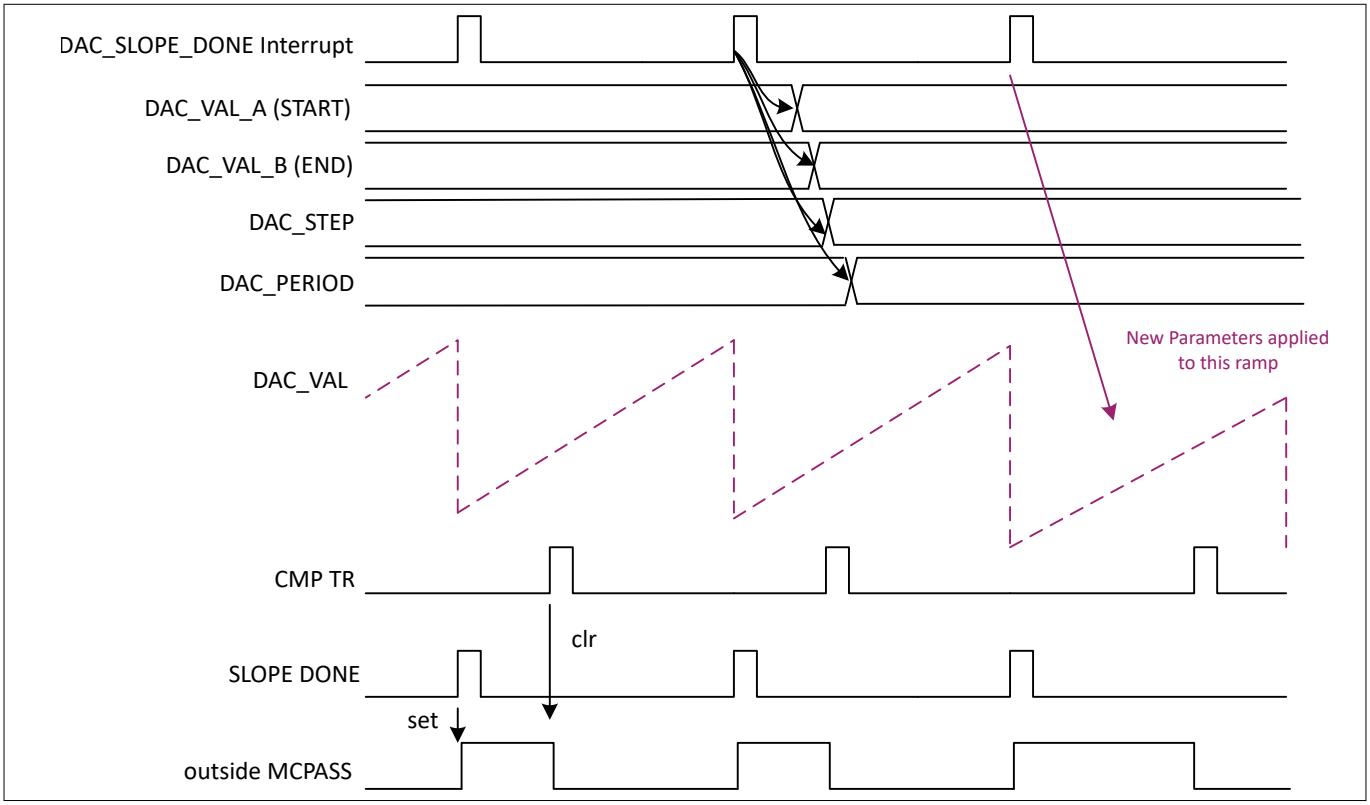


Figure 433 Slope generation example

There are 3 ways to control parameter update.

1. Continual: In this mode, the hardware updates the buffered parameters at the start of each slope (as shown in Figure 433). Using this method, firmware parameter updates can be timed with the slope done trigger, providing the maximum available time before the new values are applied to the subsequent slope.
2. Synchronized: Set the DAC_PARAM_SYNC_EN configuration bit to enable this mode. In this mode, the hardware updates the parameters (still at the start of the slope) only when the DAC_PARAM_SYNC_READY bit is set to '1'. This bit indicates to the hardware that all buffered parameters have been written. This method prevents partial parameter updates.

   ***Note**: Hardware resets the READY bit on the parameter update event*

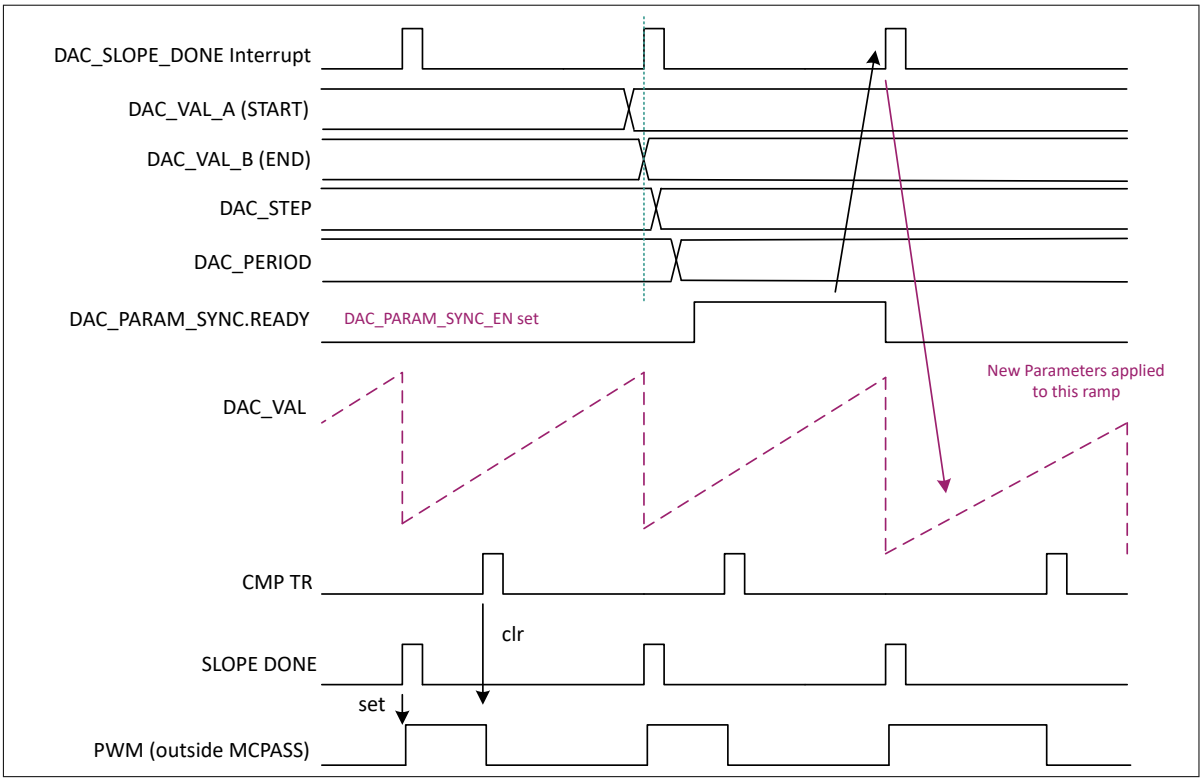


Figure 434 Synchronized parameter update

3. Cascaded: Set the DAC_CASCADE_EN configuration bit to synchronize parameter update across DACs

LUT Waveform Generation

LUT mode operates similarly to slope-generation mode, except that the parameters are address-based rather than data-based. The generated address sequence is routed to a RAM look-up table in which the user can store a custom waveform. Each successive 12-bit data value fetched from the RAM table is loaded into DAC_VAL and driven to the DAC output.

- FORWARD: The start address is DAC_VAL_A. The address is incremented by DAC_STEP and the operation stops when the address is greater than or equal to DAC_VAL_B.
- REVERSE: The start address is DAC_VAL_A. The address is decremented by DAC_STEP and the operation stops when the address is less than or equal to DAC_VAL_A.
- FORWARD_REVERSE: The start address is DAC_VAL_A. The address is incremented by DAC_STEP until it is greater than or equal to DAC_VAL_B, after which the address is decremented by DAC_STEP until it is less than or equal to DAC_VAL_A.

31.2.5.2.5 DAC Mode Start/Stop

A mode may be changed only when none of DAC operation is currently in progress. To change the mode, the user must perform a firmware write to disable the current mode, update the mode and associated parameters as required, and then re-enable the mode.

There are two ways to start a new DAC mode firmware or hardware (See DAC_MODE_START register):

HW_START: When this bit is set, the start of the DAC mode is synchronized to the next selected DAC_TR_START trigger, which may be any of the external hardware trigger inputs.

Note: *If the hardware start trigger is asserted again while the DAC mode is in progress, it will be ignored.*

A firmware-configurable option is provided to reset or restart the DAC mode while it is running.

FW_START: The mode is enabled immediately when this bit is set.

To stop the current mode, clear the associated start bit. The current DAC_VAL is held at the last value until it is either overwritten directly or a new mode is started.

In addition to disabling the mode by resetting HW_START and FW_START bit to 0, any direct write to the DAC_VAL register overrides the current DAC output and disables any DAC mode in progress.

31.2.5.2.6 DAC update triggers

When the selected DAC_TR_UPDATE is asserted, the DAC code is updated (behavior depends on the selected DAC_MODE). The available triggers are as follows:

- Any of the 16 HPPASS hardware triggers (TR0..TR15) sourced from PERI or HPPASS firmware trigger (see INFRA.HW_TR_IN_SEL)
- The DAC local DAC_PERIOD counter. (see HPPASS_DACx_PERIOD)
- The AC trigger

***Note**: when the DAC_PERIOD is selected, the user must program the period in clock cycles. There is an integer (12-bits) field and an optional fractional (5-bits) field. The fractional portion allows for resolutions up to 1/32 of a clock cycle.*

Fractional operation allows for higher-precision slope generation. However, when this feature is enabled, it results in cycle jitter, as illustrated in Figure 435. In this example, an integer only divide by 4, and below it an integer and fractional divide by 4 ½ . As shown, a 4 ½ cycles is generated by alternating between 4 and 5 clock cycles, producing the desired average period.

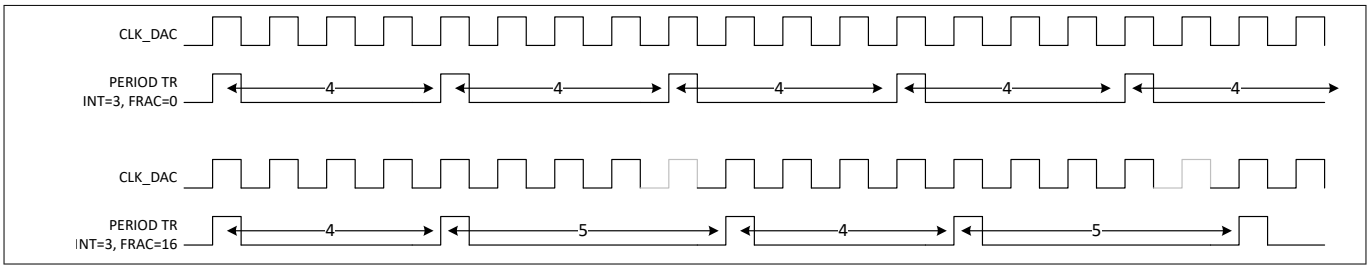


Figure 435 DAC update fractional operation

The jitter is less pronounced with larger integer divider values. For example, in a divide by 100 ¼ , the generated pattern alternates as 100-100-100-101.

As an example of a more complicated waveform, consider a divide by 20⅞ configuration. This implies that over 167 clocks, 8 triggers occur, resulting in the pattern 20-21-21-21-21-21-21-21.

31.2.5.2.7 DAC BUF analog data interface

The DAC data register is updated by the DAC FSM.

A start trigger is issued to output the first DAC sample. Triggers are ignored until the dac_ready signal is asserted by the dac_analog block.

The data transfer between the DAC digital block and the DAC analog block is illustrated in the following figure:

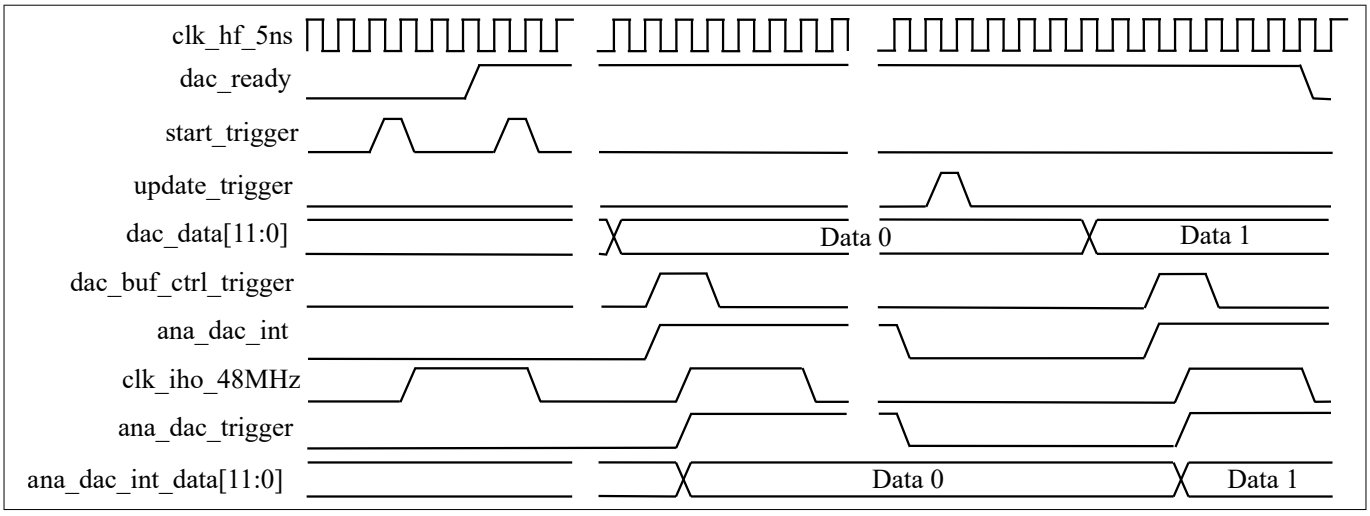


Figure 436 DAC digital to analog data transfer

`dac_data`[11:0] is update on `clk_hf`. One clock cycle later, `dac_buf_ctrl_trigger` is asserted and held high for 2 clock cycles. A minimum time of 10 ns required by the analog to capture the trigger pulse. The `dac_buf_ctrl_trigger` is captured by the analog on `clk_iho` and is used to sample the data.

The HPPASS DAC output supports both external and internal routing modes. The buffered DAC output can be driven to an external analog pad through a rail-to-rail output buffer capable of supporting resistive loads up to 10 mA and capacitive loads up to 100 pF. The exact pad mapping is product-specific and is controlled through DAC analog configuration registers. Internally, the DAC output can be routed within HPPASS, including an ADC loopback path that enables functional verification, calibration, and closed-loop signal processing without external loading. The selection between internal and external routing, along with control of the output buffer and reference source, is configured through DAC control registers. Users must ensure the selected routing mode and load conditions comply with device limitations and product-specific pin requirements.

31.2.6 Operational Amplifiers (OpAmps)

The Operational Amplifier (OpAmp) is a programmable analog signal-conditioning block within the High-Performance Programmable Analog Subsystem (HPPASS). Each OpAmp instance contains a fully differential operational amplifier, a configurable passive network, and a programmable analog switch matrix that enable implementation of multiple analog signal-processing functions. The OpAmp provides amplification, buffering, differential signal conditioning, and signal routing capabilities for motor-control, digital power, sensing, and closed-loop control applications.

HPPASS consists of four OpAmps with integrated passive components and a programmable switch network. Each OpAmp can be configured for either single-ended output or fully differential output. In a single-ended output configuration, an OpAmp can drive either an internal load or an external load, depending on the configuration. OpAmp inputs can be sourced from external inputs, the internal DAC, or a fixed reference.

Features

The OpAmp provides the following features:

- Fully differential operational amplifier architecture
- Single-ended output operation
- Differential output operation
- Integrated programmable switched-resistor network
- Integrated analog input multiplexer
- Internal and external load support
- Rail-to-rail input capability

- Different programmable PGA gain values
- Direct connectivity to ADC, DAC, reference, and analog routing resources

OpAmp can be configured as:

- Programmable gain amplifier (PGA)
- Difference amplifier (DA)
- Unity gain buffer (UGB)
- Standalone OpAmp

Refer to the Figure 385 for additional details.

Preset gain values can be selected using the integrated switched resistive network. The inputs, INN, and INP of OpAmps can be accessed through input multiplexers. Each OpAmp has one output that can directly drive an output pin. If the output pin is not used by an OpAmp, it can be tristated so that it is accessible to other functions. Other OpAmp outputs are used internally. Inputs to the OpAmps can be selected through the input multiplexer from analog GPIOs, the DAC, reference voltages, or ground.

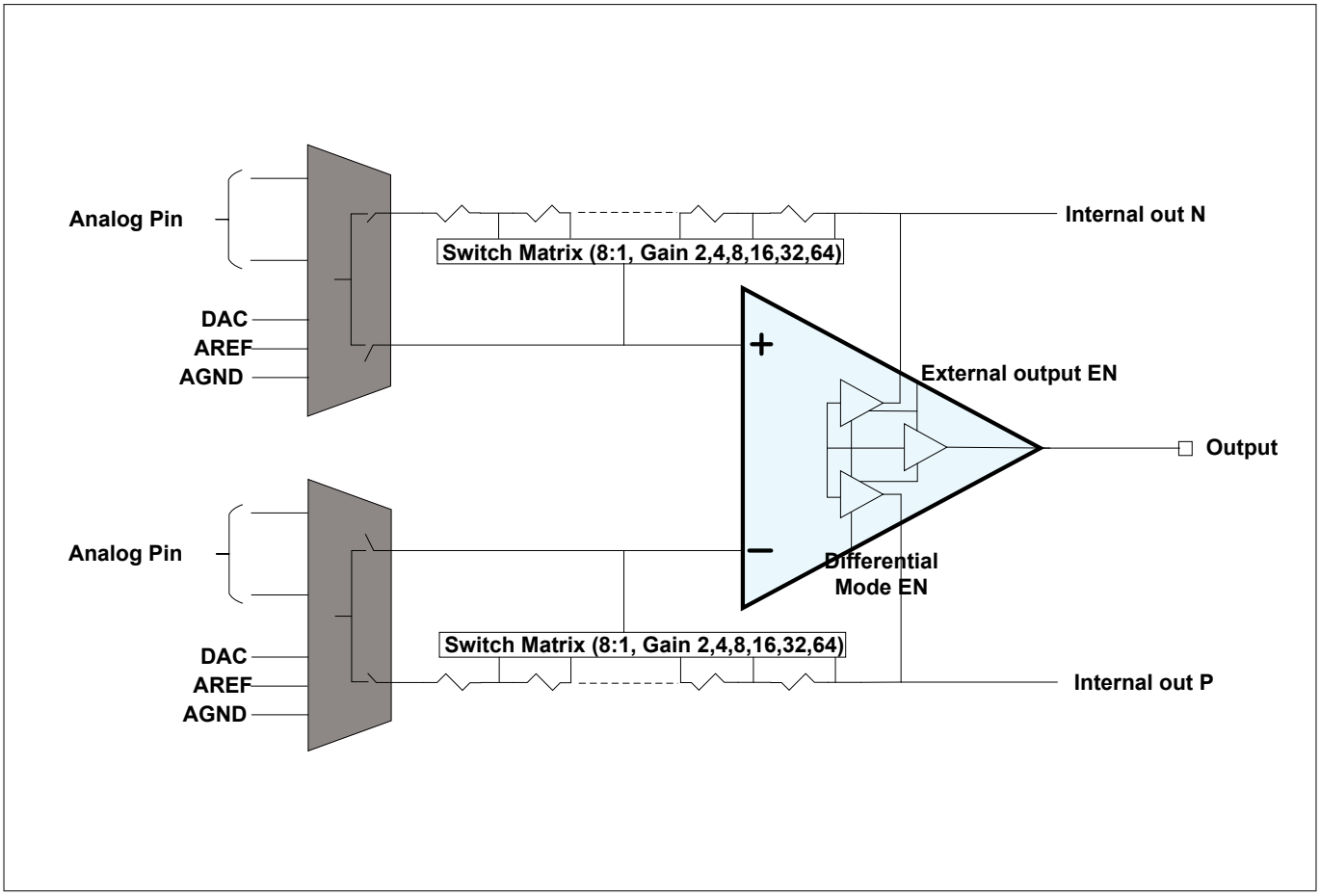


Figure 437 HPPASS OpAmp

***Note**: OpAmp is also called as AFE in some registers and signals*

31.2.6.1 OpAmp digital

The OpAmp supports multiple input sources, selectable gain values, and various output sinks, including the ADC, CSG comparator, and MCU pins. These configurations can be either static or dynamically updated. Each OpAmp has four configuration registers that can be dynamically selected to support different applications. Selection can be performed by firmware alone or by firmware in conjunction with a level-based trigger source.

Figure 438 shows the HPPASS OpAmp and its associated control signals. The OpAmp digital block functions as an MMIO slave, providing registers that control routing of the OpAmp inputs and outputs. Registers required to manage input multiplexing and analog control signals are included within the OpAmp digital block.

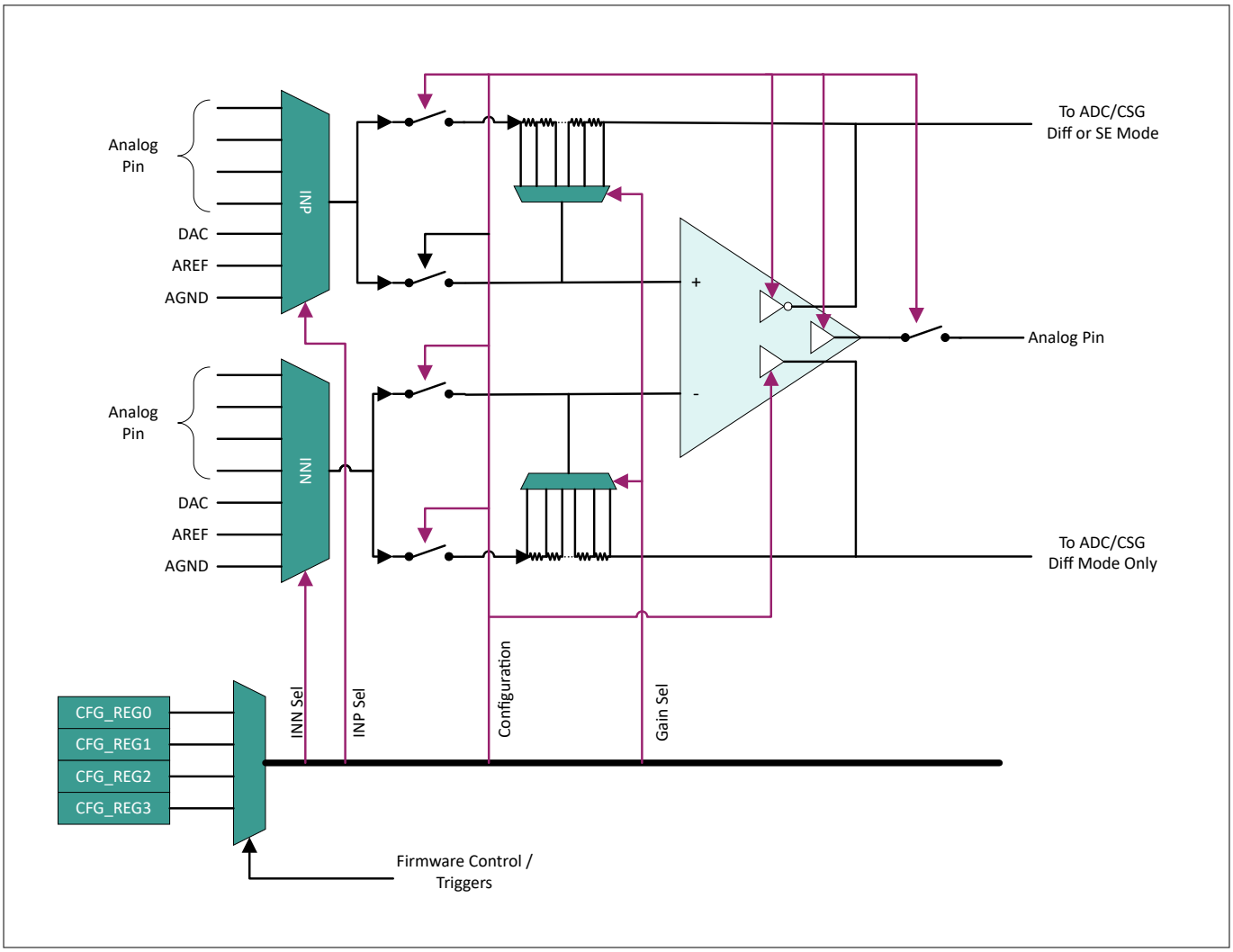


Figure 438 OpAmp control

As shown in the Figure 438, the four configurations of an OpAmp can be programmed using the `CFG_REG0`, `CFG_REG1`, `CFG_REG2`, and `CFG_REG3` registers. Each configuration specifies the positive input, negative input, gain, and mode. The available modes are as follows:

1. Standalone OpAmp
2. Unity gain buffer with single-ended input and internal single-ended output
3. Programmable gain amplifier with differential input and internal differential output
4. Programmable gain amplifier with single ended input and internal single ended output
5. Programmable gain difference amplifier with internal single-ended output

These modes can be selected by programming the `OpAmp_CONFIG.CONFIGURATION` field. The register value for each mode, along with the corresponding input and output connections and supported gain values, is shown in the Table 327.

Table 327 OpAmp mode selection

Value	Input mode	Internal output connections	External output connection	Gain
0	INP and INN	OUTP Hi-Z OUTN is Hi-Z	Enabled	Standalone OpAmp
1	INP single-ended (INN not available)	OUTP single-ended OUTN is Hi-Z	Hi-Z	Unity gain
2	INP/INN differential	OUTP/OUTN differential	Hi-Z	Gain 1, 2, 4, 8, 16, 32, 64
3	INP single-ended (INN not available)	OUTP single-ended OUTN is Hi-Z	Hi-Z	Gain 2, 3, 5, 9, 17, 33, 65
4	INP/INN differential	OUTP single-ended OUTN is Hi-Z	Hi-Z	Gain 1, 2, 4, 8, 16, 32, 64

The positive and negative inputs to the OpAmp, as well as the gain, can be configured using the`AFE_CONFIG.POS_INPUT_SRC`, `AFE_CONFIG.NEG_INPUT_SRC` and `AFE_CONFIG.OPAMP_GAIN` register fields, respectively. Refer Figure 385 for the possible inputs and outputs for OpAmp instances. The input pin selection and OpAmp gain values for different register values are shown in Table 328.

Table 328 OpAmp configuration fields

Value	POS_INPUT_SRC[2:0]	NEG_INPUT_SRC[2:0]	OPAMP_GAIN [2:0]
0	Pin 0	Pin 0	1 (Mode 2 and 4), 2 (Mode 3)
1	Pin 1	Pin 1	2 (Mode 2 and 4), 3 (Mode 3)
2	Pin 2	Pin 2	4 (Mode 2 and 4), 5 (Mode 3)
3	Pin 3	Pin 3	8 (Mode 2 and 4), 9 (Mode 3)
4	DAC	DAC	16 (Mode 2 and 4), 17 (Mode 3)
5	AREF	AREF	32 (Mode 2 and 4), 33 (Mode 3)
6	AGND	AGND	

An OpAmp instance has four configuration registers, it supports four distinct configurations.

After selecting and writing the four configurations to an OpAmp's configuration registers, the system must select one of the four configurations for operation. There are two methods for configuration selection. The first way is via firmware.

The first method uses firmware control. By using the `AFECONFIG_CTRL.MUX_CTRL` field, software selects which configuration to use (0, 1, 2, or 3).

Alternatively, software can enable a level trigger to select between two configurations (0 or 1). When the level trigger is 0x0, the EVEN configuration is selected. When the level trigger is 0x1, the ODD configuration is selected. This implementation is shown in Figure 439.

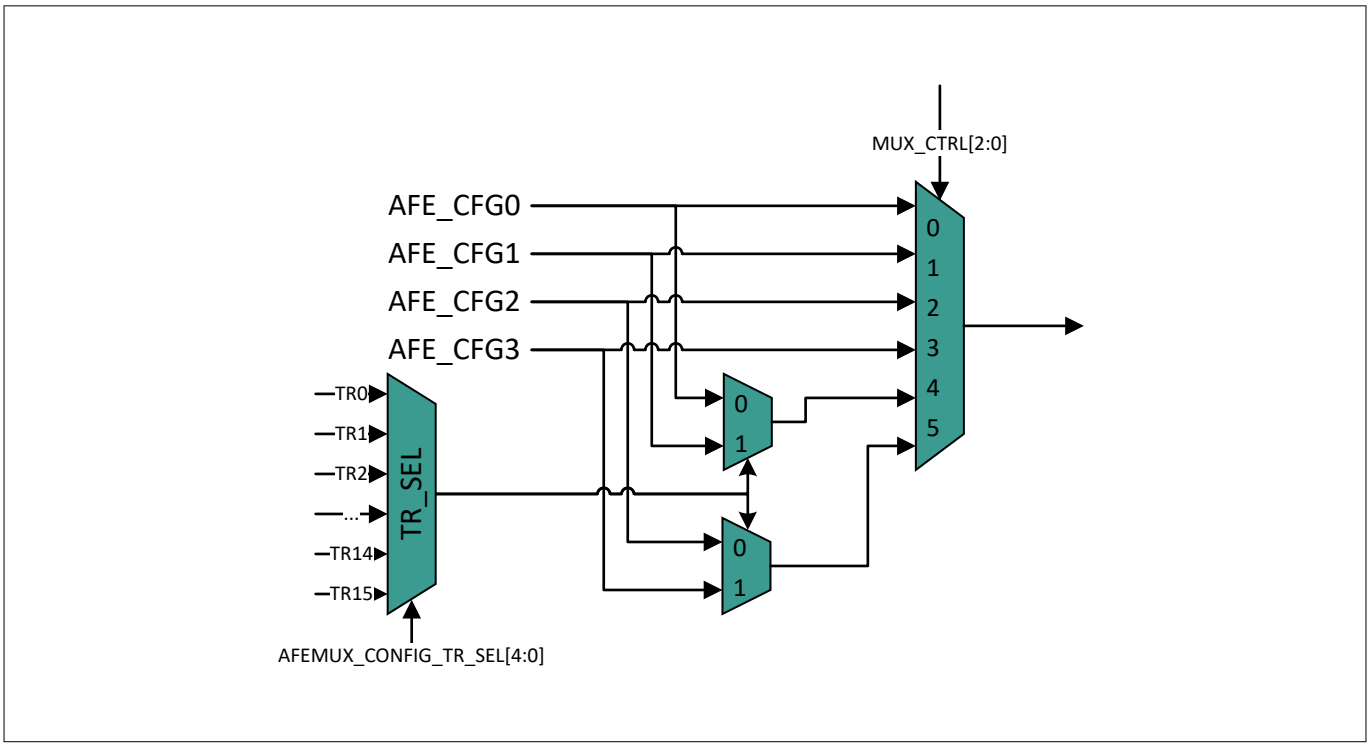


Figure 439 Configuration selection

31.2.6.2 OpAmp configurations

1. Standalone OpAmp

To use the HPPASS OpAmp in this configuration, the OpAmp must be configured in mode 0. It will make the OpAmp work like a standalone OpAmp. Set the `AFE_CONFIG.CONFIGURATION` register field to '0' to select this mode. It will connect the output of the OpAmp to the analog output pin (see datasheet for details) and the positive input pin will be connected to the pin specified in the register field, `AFE_CONFIG.POS_INPUT_SRC`. Similarly, the negative input pin will be connected to the pin specified in the register field, `AFE_CONFIG.NEG_INPUT_SRC`. The following section describes several ways to use this mode.

a. Unity gain amplifier with external output

To make it a unity gain amplifier, it is required to connect the feedback externally. Figure 440 shows the OpAmp configured in this mode.

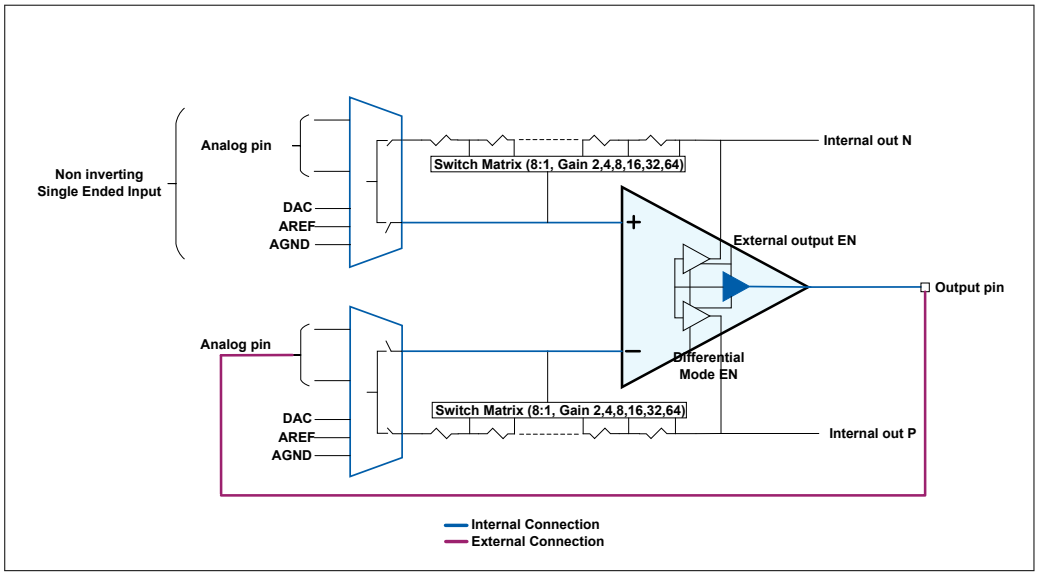


Figure 440 Unity gain amplifier with external load

b. Non-Inverting amplifier with external output

For this configuration, connect the required external connection as shown in Figure 441 to use the OpAmp as a non-inverting amplifier. Similarly, an inverting amplifier configuration can be realized by making the necessary external connections.

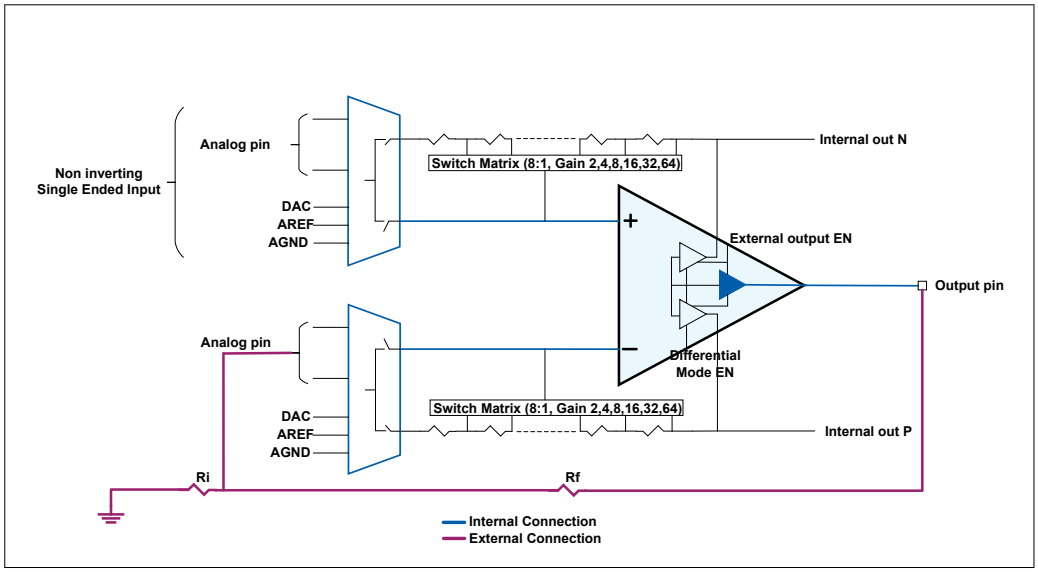


Figure 441 Non-Inverting amplifier with external output

c. Shunt current amplifier with offset for motor control

This is an advanced configuration that is useful for motor control applications. In this configuration, the OpAmp is set to standalone mode and used as a differential amplifier with offset correction. Figure 442 shows the OpAmp configured in this mode.

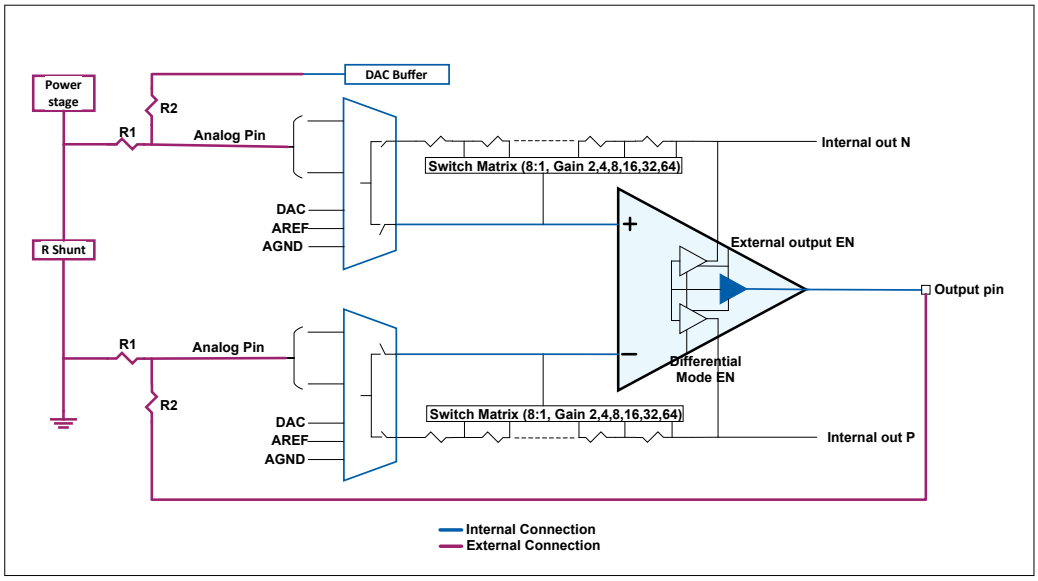


Figure 442 Shunt current amplifier with offset for motor control

2. Unity gain amplifier with single-ended input and internal single-ended output

To set this mode, write '1' to the `AFE_CONFIG.CONFIGURATION` register field. It enables single-ended output connections to the internal targets like ADC. The positive input pin is connected to the pin specified by the `AFE_CONFIG.POS_INPUT_SRC` register field. In this mode, the negative input is internally connected to the positive output, and external connections to the negative inputs are not supported. Figure 443 shows the OpAmp configured in this mode.

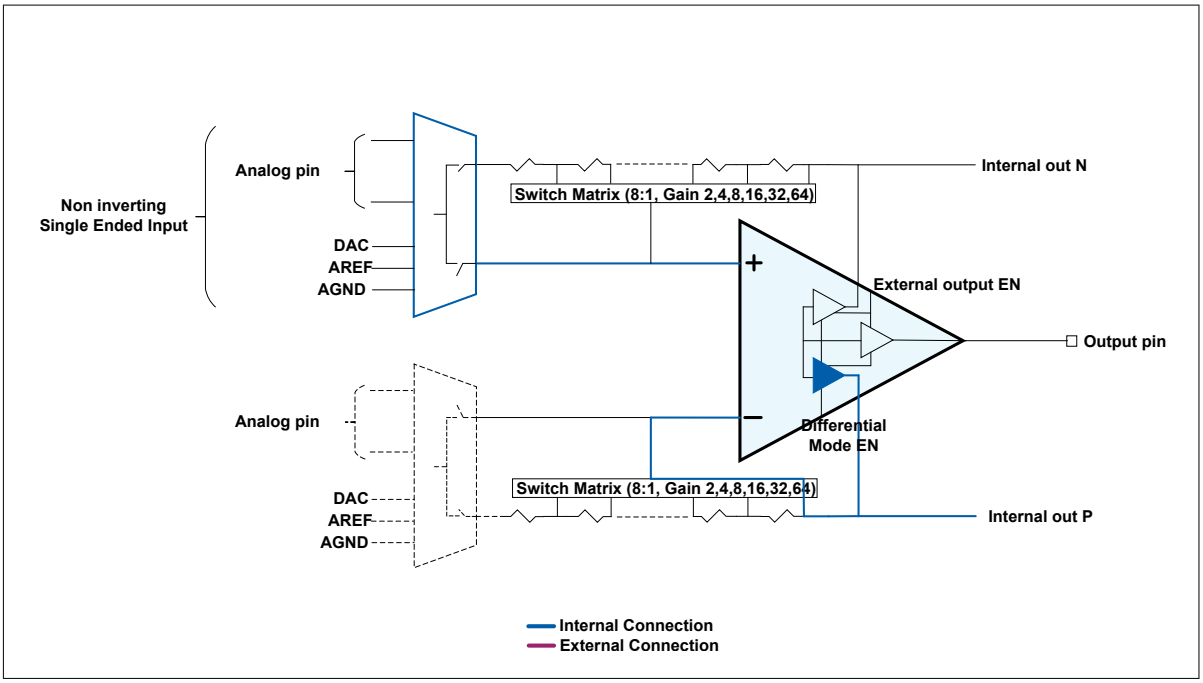


Figure 443 Unity gain amplifier with internal load

3. Programmable gain amplifier with differential input and internal differential output

To set this mode, write '2' to the `AFE_CONFIG.CONFIGURATION` register field. It enables differential output connections to the internal targets like ADC. The positive and negative input pins are connected to the pins specified by the `AFE_CONFIG.POS_INPUT_SRC` and `AFE_CONFIG.NEG_INPUT_SRC`register fields, respectively.

In this mode, the amplifier gain can be configured using the AFE_CONFIG.OPAMP_GAIN register field. It leads to positive and negative inputs of the OpAmp to connect to the inputs and outputs through the register network. Figure 444 shows the OpAmp configured in this mode.

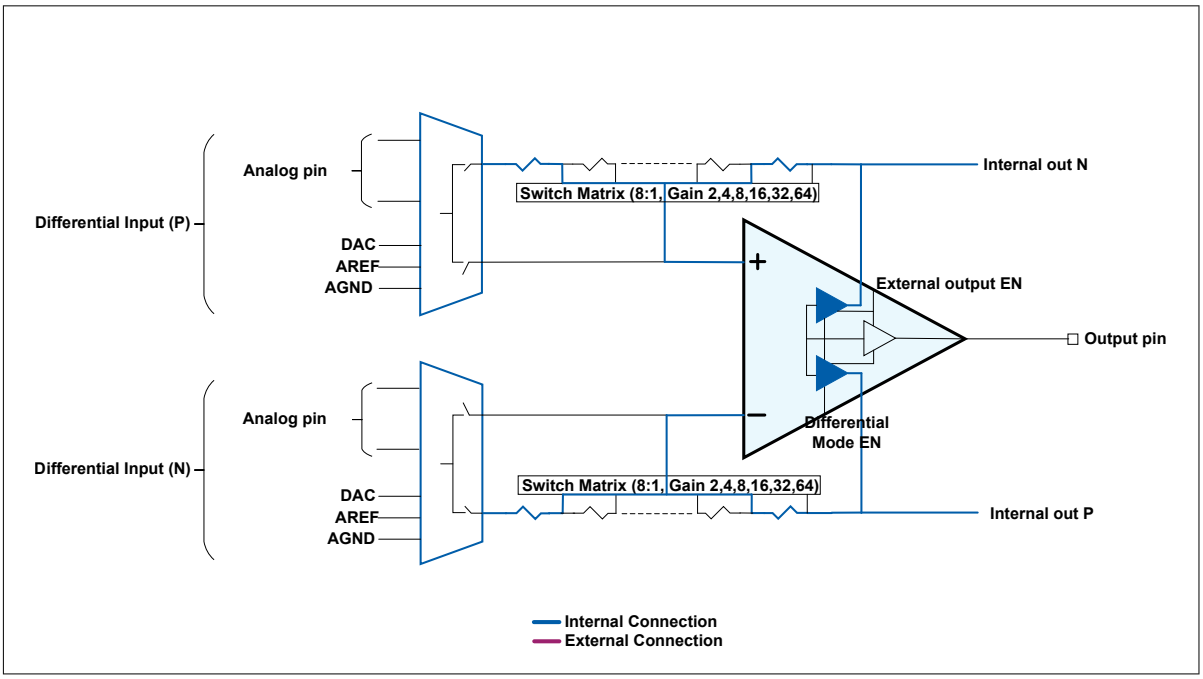


Figure 444 Programmable gain amplifier with differential input and internal differential output

4. Programmable gain amplifier with single-ended input and internal single-ended output

To set this mode, write '3' to the AFE_CONFIG.CONFIGURATION register field. It enables single-ended output connections to the internal targets like ADC. It can be used as either an inverting and a non-inverting amplifier. Figure 445 shows the OpAmp configured in this mode.

If it is used in the non-inverting mode, the input can be selected by specifying it in the register field, AFE_CONFIG.POS_INPUT_SRC. In this case, the negative input is connected to the reference using the AFE_CONFIG.NEG_INPUT_SRC register field. Similarly, to configure it as an inverting amplifier, specify the input using the AFE_CONFIG.NEG_INPUT_SRC register and connect the non-inverting terminal to the reference using the AFE_CONFIG.POS_INPUT_SRC register field.

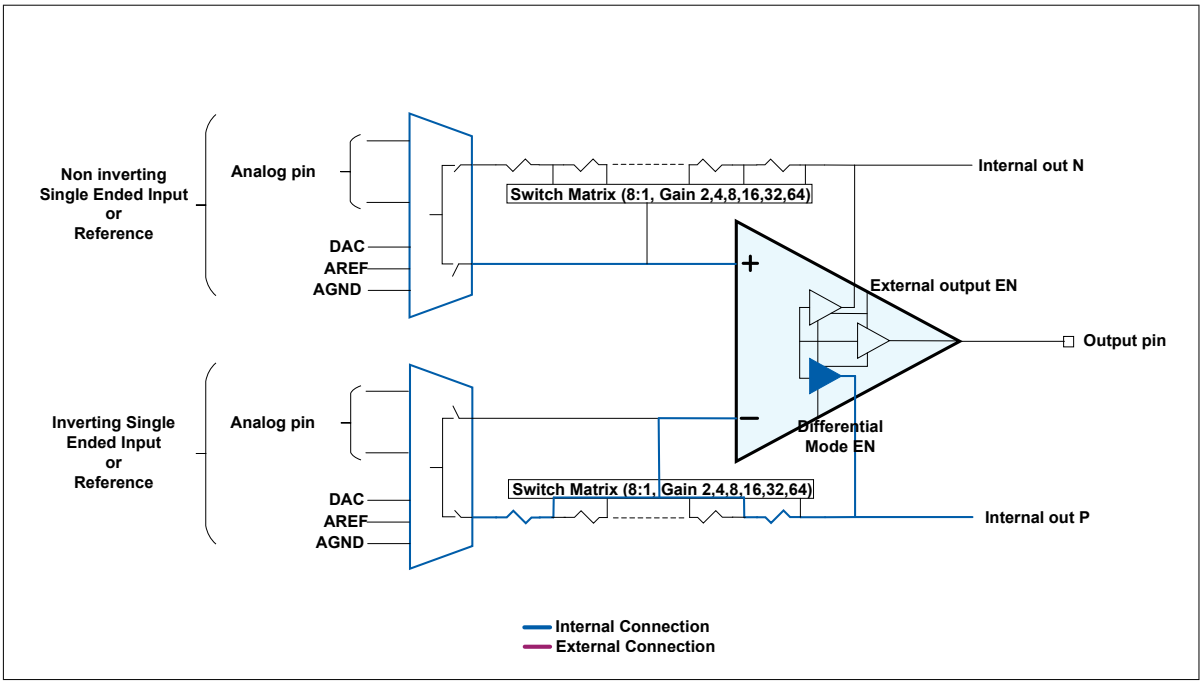


Figure 445 Programmable gain amplifier with single-ended input and internal single-ended output

5. Programmable gain difference amplifier with internal single-ended output

To set this mode, write '4' to the `AFE_CONFIG.CONFIGURATION` register field. It enables single-ended output connections to the internal targets like ADC and the positive and negative input pins will be connected to the pin specified in the `AFE_CONFIG.POS_INPUT_SRC` and `AFE_CONFIG.NEG_INPUT_SRC` register fields, respectively. In this mode, the OpAmp's positive output is connected to the output and inverting input through the resistor divider based on the selection of gain in the `AFE_CONFIG.OPAMP_GAIN` register. Figure 446 shows the OpAmp configured in this mode.

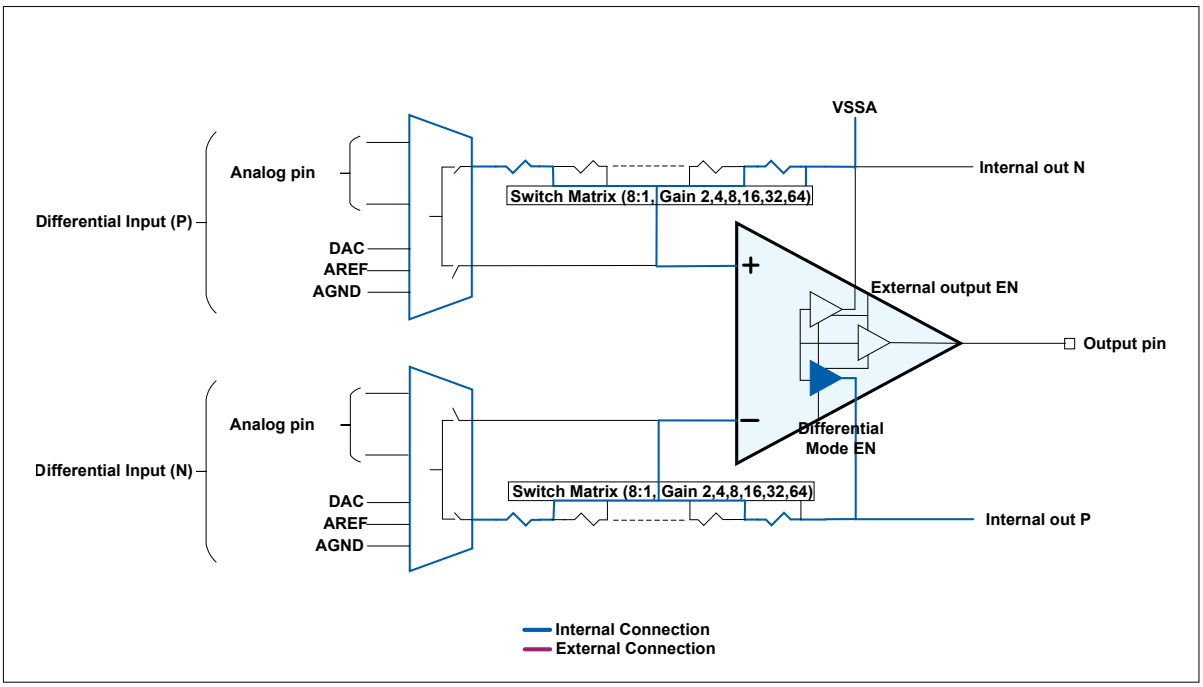


Figure 446 Programmable gain difference amplifier with internal single-ended output

Note: The DAC output can be routed to the OpAmp input. In this configuration, the overall rail-to-rail operating range is limited and will be narrower than the individual operating ranges of the DAC and OpAmp.

31.2.7 Autonomous controller (AC)

31.2.7.1 Overview

The autonomous controller (AC) provides hardware control without the need for CPU or external intervention.

The AC is a programmable state machine. Figure 447 shows the AC simplified block diagram. It consists of a memory-based state transition table ("timer table"), sequencer, and timers.

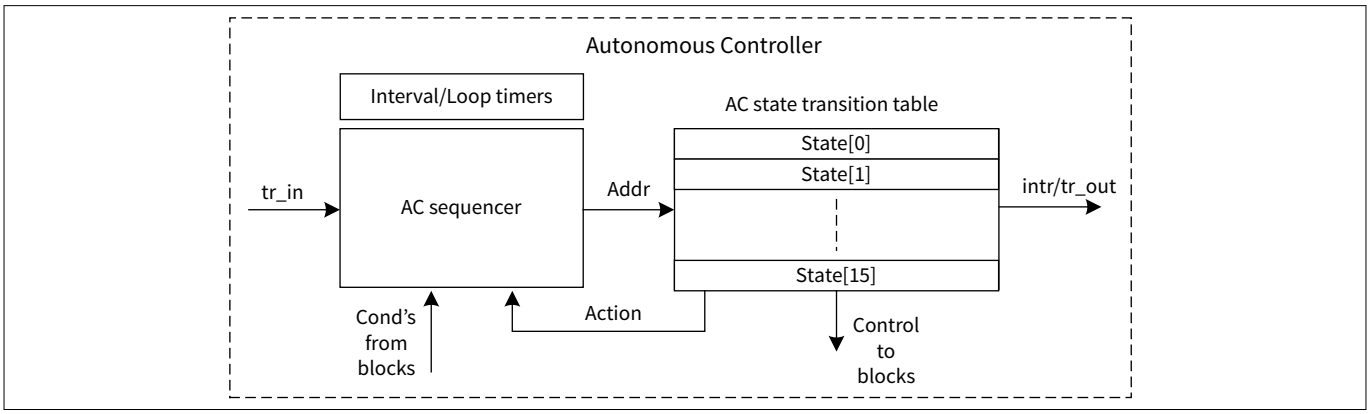


Figure 447 Autonomous controller block diagram

The AC has the following capabilities:

- Enabling and disabling of blocks, providing BLOCK_READY status to firmware
- Ability to be triggered by hardware/firmware and execute programmed actions
- Triggering any of the HPPASS internal blocks (SAR groups, CSG slices)
- Implementing hardware delays
- Taking periodic action (for example, triggering) in a timed loop
- Receiving conditions from the HPPASS internal blocks and acting (for example, triggering) on those events
- Generating an AC-based interrupt to the CPU
- Controlling AROUTE switches (Note: advanced usage only)
- Monitoring SAR ADC limit detection condition and taking an action based on the status

31.2.7.2 Block interface

The AC has a standard interface to each sub-block of HPPASS as shown in Figure 448. The autonomous capability is implemented with a state transition table, and it consists of RAM words that have encoded fields that output controls to hardware resources, and input events generated by those resources.

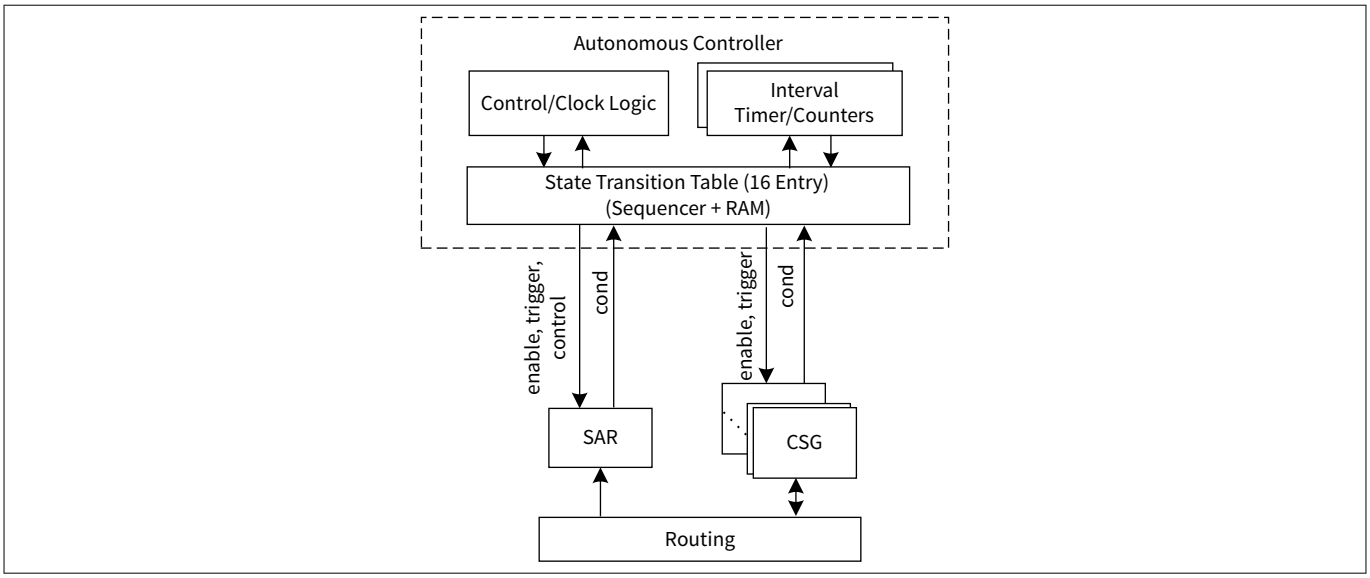


Figure 448 Autonomous controller block interface

The AC block interface consists of:

- **Enable**

  Each functional block has a single-bit enable control. When the bit is disabled (‘0’), the logic is held in reset, clock-gated, and not consuming any dynamic power, and all controls to the analog block are in disabled state. When the bit is enabled (‘1), the logic is active and the clock is ungated

  After enabling a given set of application blocks, wait for the “BLOCK_READY” status before starting the application. Disabling the functional block after the application is completed does not set BLOCK_READY to low. Hence, only poll BLOCK_READY when re-enabling the functional block
- **Trigger**

  Relative to the autonomous controller, the trigger is the method to start the activity in a given functional block (or sequence that block to a new activity, if that is required). A basic example is the SAR ADC trigger. Both SAR and CSG have trigger inputs
- **Conditions**

  The condition inputs are status from the functional block to the controller that can be used to branch the controller sequence. For example: ENTRY_DONE, SAR_RANGE, DAC_SLOPE_DONE
- **Other Configuration/Control**

  The AC is general enough to adapt to other sub-block controls. For example, there is an optional use case using the AC to control the AROUTE MUX selections

31.2.7.3 State transition table

The AC can implement a single threaded programming sequence using a simple format.

The AC program is stored in a RAM table (called a state transition table), which consists of N states (or programming instructions). In PSC3M/P6 MCU, N is 16 so there are 16 programming instructions available.

The state transition table may contain one or more programs that can be called by CPU firmware, like a sub-routine.

In the AC implementation, each instruction state spans 4 registers (TT_CFG0, TT_CFG1, TT_CFG2, TT_CFG3) with the following organization, See the Registers Technical Reference Manual for more details:



Figure 449 State of AC state transition table

A short description of each register function:

- **Unlock**: The purpose of the unlock function is to have a special control of block enables. If you unlock a given block enable in a given state, it's only in that state that the enable is updated. This prevents inadvertent mode transitions in the normal course of AC program execution
- **Sequencer**: This is the "brain" of the AC. In this register the actual program flow is specified with delays, loops, branches, and condition checking
- **CSG**: Used to enable and optionally trigger the CSG slices
- **SAR ADC**: Used to enable the SAR ADC and optionally trigger SAR ADC groups, and control AROUTE

AC sequencer programming flow

The following bit fields of the TT_CFG1 register are associated with the AC sequencer to control program flow.

Table 329 Bit fields of TT_CFG1 register

Bit_Name[Bit_Pos]	Description
BR_ADDR[3:0]	Absolute address of the branch if taken (0 - N-1 possible)
COND[14:8]	Autonomous controller condition
ACTION[18:16]	Autonomous controller action
INTR_SET[19]	AC interrupt set. This will generate a CPU interrupt or hardware trigger output if configured
CNT[31:20]	Used to set the timer/counter values for either interval and loop counts, the actual count is CNT value + 1

The AC sequencer fields are described in more detail in the following section:

- **CONDITIONS**
  This is a device-specific list of block status conditions that are generated and output by the HPPASS sub blocks. The following table lists all AC conditions (see the Registers Reference Manual for more details).

Table 330 Autonomous controller condition

Condition	Description
FALSE	Hard-coded FALSE condition, used for unconditional branch
TRUE	Hard-coded TRUE condition, used for unconditional branch
BLOCK_READY	Set when the block start-up has completed, and indicates that enabled blocks are ready for operation
CNT_DONE	Used for delays and loops. ACTION is WAIT_FOR – this is an interval time; if the ACTION is BRANCH_IF_TRUE/FALSE – this is a loop counter
SAR_ENTRYn_DONE (n = 0 -7)	SAR ADC group done
SAR_RANGEn (n=0 - 7)	SAR ADC limit detection condition is asserted
SAR_BUSY	SAR ADC busy
SAR_FIRn_DONE (n = 0/1)	SAR ADC FIR done
SAR_QUEUE_HI/LO_EMPTY	SAR ADC high/low priority queue is empty

(table continues...)

Table 330 (continued) Autonomous controller condition

Condition	Description
SAR_QUEUES_EMPTY	SAR ADC both queues are empty
HW_TRn_IN (n = 0 - 15)	Hardware/Firmware trigger from INFRA has been asserted
FIFO_LEVELn (n = 0 - 3)	FIFO status level
CSGn_SLOPE_DONE (n = 0 - 8)	CSG slope/LUT waveform done
CSGn_CMP_TR (n = 0 - 8)	CSG comparator trip has occurred
CSG_REMAP_02/35/68_CMP_TR	CSG remap comparator output trigger

Conditions from the block are generated as a pulse to the AC. Because the AC may not always be waiting for a given condition, these conditions are latched and only cleared when checked by the AC. This gives the capability for AC polling of conditions as in the following example.

Example: SAR ADC group done (SAR_ENTRY_DONE) for group 0,1,2 are checked in this polling loop. If the condition is TRUE (set) when checked in this loop, it is cleared and the routine then jumps to a state that re-triggers that SAR ADC group again, then would jump back to this POLL loop.

```
STATE: POLL
    ACTION=BRANCH_IF_TRUE   BR_ADDR=<TR_SAR_ENTRY0>   COND=SAR_ENTRY0_DONE

STATE:
    ACTION=BRANCH_IF_TRUE   BR_ADDR=<TR_SAR_ENTRY1>   COND=SAR_ENTRY1_DONE

STATE:
    ACTION=BRANCH_IF_TRUE   BR_ADDR=<TR_SAR_ENTRY2>   COND=SAR_ENTRY2_DONE

STATE:
    ACTION=BRANCH_IF_TRUE   BR_ADDR=POLL   COND=TRUE
```

- **ACTIONS**

  Each instruction must contain one of these actions:

Table 331 Autonomous controller action

Action	Description
STOP	Stop on this state (default state) This will stop the AC. A CPU firmware write to ACTRLR.CMD_RUN will be required to start execution again at a specified address STOP is always in the last instruction in a given AC routine
NEXT	Go to the next state. Stays in this state for 1 AC clock cycle
WAIT_FOR	Stay in this state and wait for the specified COND, when the condition is True, then go to the next state
BRANCH_IF_TRUE	Check for the specified COND, if the condition is True, go to the specified state in BR_ADDR; if the condition is False, go to the next state
BRANCH_IF_FALSE	Check for the specified COND, if the condition is False, go to the specified state in BR_ADDR; if the condition is True, go to the next state

***Note**: The following code fragments are written in an easy-to-read format (versus register writes) in which a state is delimited by the STATE: with optional label, and the code as <FIELD_NAME> = <FIELD_VALUE> (from HPPASS_ACTRLR_TTCFG registers), see the Registers Technical Reference Manual for more details.*

Example: Trigger SAR ADC group 0 (SAR_ENTRY0) and move to the next state.

```
STATE:
    SAR_TR=0x01    ACTION=NEXT
```

Example: Trigger SAR ADC group 0 (SAR_ENTRY0) and wait in this state until the conversion is complete for that group (SAR_ENTRY0_DONE).

```
STATE:
    SAR_TR=0x01    ACTION=WAIT_FOR    COND=SAR_ENTRY0_DONE
```

Example: Branch to new state if the comparator in CSG slice 0 trips, otherwise move the next state.

```
STATE:
    ACTION=BRANCH_IF_TRUE    BR_ADDR=<STATE#_OR_LABLE>    COND=CSG0_CMP_TR
```

- **INTERRUPT**

  The AC can generate a CPUSS interrupt or pulse trigger in a given state by setting the INTR_SET bit. This may be configured as an interrupt by configuring the HPPASS_MMIO_HPPASS_INTR register set or as a trigger by configuring the HPPASS_MMIO_TR_PULSE_OUT register.

```
STATE:
    INTR_SET=1
```

- **COUNTERS**

  In HPPASS there are two counters, which are implicitly assigned as interval counters or loop counters depending on the context.

  If the condition is WAIT_FOR, then the instruction is an interval timer and the AC stays in that state for CNT clock cycles.

  If the condition is BRANCH_IF_TRUE/FALSE, then the instruction is a loop counter. The initial CNT is loaded and each time this state is executed, the counter is decremented. At ‘0’ a different execution flow will be taken.

  Example: this example is an interval counter to implement timed delays. Wait for 64 HPPASS AC clock cycles in this state, then proceed to the next state.

```
STATE:
    ACTION=WAIT_FOR    COND=CNT_DONE    CNT=64
```

Example: This example is a loop counter. In the case of a loop counter a CNT value is specified with a BRANCH condition. The AC interprets this as a loop. Every time the BRANCH condition is tested, the count is

decremented. During the count down the COND is FALSE, so the code branches to <TOP_OF_LOOP> address. On terminal count, the COND is TRUE, so the code falls through to the next state.

```
STATE:
    ACTION=BRANCH_IF_FALSE    BR_ADDR=<TOP_OF_LOOP>    COND=CNT_DONE    CNT=16
```

31.2.7.4 Initialization

The initialization is the required use case for the AC in HPPASS because AC is designed to coordinate startup timing (refer to Startup for the details).

The following example is the standard two-state program that enables the SAR ADC and three CSG slices and waits for the BLOCK_READY status, then asserts an interrupt to the CPU (optional).

```
STATE:
    SAR_UNLOCK=1    CSG_UNLOCK=0x7    SAR_EN=1    CSG_EN=0x7    ACTION=WAIT_FOR    COND=BLOCK_READY

STATE:
    INTR_SET=1    ACTION=STOP or NEXT
```

Interrupt is optional, firmware may also poll BLOCK_READY status (HPPASS_ACTRLR.BLOCK_STATUS.READY).

To start the HPPASS:

- Enable the AC (HPPASS_ACTRLR_CTRL.ENABLED = 1)
- Set the AC state address (HPPASS_ACTRLR_CMD_STATE) (by default on POR it's 0)
- Run the AC (HPPASS_ACTRLR_CMD_RUN = 1)

The AC (in conjunction with startup timing hardware) enables and sequences the startup of the blocks. Once completed (BLOCK_READY is set), the AC is in a stopped state, with all the blocks ready for the application to start.

31.2.7.5 Examples

The following examples may or may not apply to the practical application. These descriptions demonstrate the types of sequences that can be programmed. Also, it is assumed that the first one or two states of the AC table are always programmed, as previously described in the Initialization chapter and is not included in the examples.

***Note**: Since the AC is composed of actual RAM and not initialized on POR, we must write to all four TT_CFG0-3 registers for any states that are used in a program – even when no programming is needed for that register.*

For more information on SAR ADC and CSG, see the SAR ADC chapter and Comparator and slope generator (CSG) chapter for more details.

Example 1 - Delay the incoming trigger to the SAR ADC or CSG (or both)

In this case, there is an external trigger and the objective is to implement a fixed delay from this trigger, prior to triggering one or more SAR ADC groups or CSG Slices. The following code assumes the external trigger is routed (INFRA configuration) to HPPASS internal TR0, and that SAR ADC group 4 is triggered.

The fixed delay CNT is 2048 (CNT+1). If the SAR ADC is running at 160 MHz, while the AC logic is on a fixed divide by 2, the timing cycle is 80 MHz (12.5 ns); therefore, this delay is 25.6 µs.

```
STATE:TOP
    ACTION=WAIT_FOR    COND=HW_TR0_IN

STATE:
    ACTION=WAIT_FOR    COND=CNT_DONE    CNT=0x7ff

STATE:
    SAR_TR=0x10    ACTION=BRANCH_IF_TRUE    COND=TRUE    BR_ADDR=TOP
```

***Note**: The last instruction action is a jump (unconditional branch).*

***Note**: If the HPPASS AC logic is clocked at 80 MHz, the maximum delay for a single counter is 4096 or 51.2 µs. However, there are two counters in HPPASS, see COUNTERS description of AC sequencer programming flow section for the details. If both are used, one as a loop counter and one as an interval counter inside the loop – the maximum delay is 2^{24} x 12.5 ns – about 200 ms.*

The following is a modified version that first delays the SAR ADC trigger, and then implements a different delay for three CSG Slice DAC triggers.

```
STATE:TOP
    ACTION=WAIT_FOR    COND=HW_TR0_IN

STATE:
    ACTION=WAIT_FOR    COND=CNT_DONE    CNT=0x7ff

STATE:
    SAR_TR=0x10    ACTION=WAIT_FOR    COND=CNT_DONE    CNT=0x3ff

STATE:
    CSG_DAC_TR=0x7    ACTION=BRANCH_IF_TRUE    COND=TRUE    BR_ADDR=TOP
```

***Note**: The trigger action is always in the first cycle of a state, regardless of the WAIT_FOR action.*

Example 2 - AC trigger SAR averaged group

If averaging (accumulation) by N is configured for channels in a given group, only 1 result is generated every N triggers for that ADC group. The user has the following choices to support this:

- External periodic trigger of N times (or continuous)
- Set the CONT (continuous trigger) bit in that ADC group
- Use the AC to trigger the ADC group N times

The benefit of the AC option is for the case when an averaged conversion is triggered by 1 external event. Or when the measurement is at a random rate, for example triggered by firmware.

The following routine waits for an HPPASS internal trigger (can be an external hardware or firmware trigger depending on INFRA trigger programming), then triggers the indicated SAR ADC group N times (16 in this case)

each time waiting for SAR ADC group conversion done (SAR_ENTRY_DONE) condition, and then jumps back to wait for the next HPPASS trigger.

```
STATE:TOP
    ACTION=WAIT_FOR    COND=HW_TR0_IN

STATE:TR_SAR
    SAR_TR=0x01    ACTION=WAIT_FOR    COND=SAR_ENTRY0_DONE

STATE:
    ACTION=BRANCH_IF_FALSE    BR_ADDR=TR_SAR    COND=CNT_DONE    CNT=15

STATE:
    ACTION=BRANCH_IF_TRUE    BR_ADDR=TOP    COND=TRUE
```

Example 3 - Periodic trigger generation

The AC can be used to create a periodic trigger. In this example, the trigger frequency is 10 kHz (If HPPASS digital clock is 80 MHz, 80 MHz/8000 = 10 kHz). Since this is more than the maximum interval delay – it is created using an inner delay and outer loop.

As with any firmware loop – to get exact cycle timing, construct the loop carefully. Note the comments, and the fact that a dummy NEXT cycle was added to make it an even number.

```
STATE:TOP
    ACTION=WAIT_FOR    COND=CNT_DONE    CNT=3997 // 3998x2 cycles

STATE:
    ACTION=BRANCH_IF_FALSE    COND=CNT_DONE    BR_ADDR=TOP    CNT=1 // 1x2 cycles

STATE:
    ACTION=NEXT // 1 cycle

STATE:
    INTR_SET=1    ACTION=BRANCH_IF_TRUE    COND=TRUE    BR_ADDR=TOP // 1 cycle
```

Example 4 - Using AC for firmware trigger command

Although there are 16 triggers which can be allocated to either external hardware or firmware triggers, the following command illustrates how to add firmware triggers.

The following code includes the startup routine and two additional firmware commands.

```
STATE: // state 0
    SAR_UNLOCK=1   CSG_UNLOCK=0x7   SAR_EN=1   CSG_EN=0x7   ACTION=WAIT_FOR   COND=BLOCK_READY

STATE: // state 1, AC is STOPPED after initialization
    INTR_SET=1   ACTION=STOP

STATE: // state 2, setup firmware command for SAR ENTRY trigger
    SAR_TR=0x01   ACTION=STOP

STATE: // state 3, setup firmware command for DAC slices 0,1,2 triggers
    CSG_DAC_TR=0x07   ACTION=STOP
```

To call state 2 which triggers SAR ADC group 0, (assuming the AC is STOPPED):

- Set the AC state address (HPPASS_ACTRLR_CMD_STATE = 2)
- Run the AC (HPPASS_ACTRLR_CMD_RUN = 1)

This will execute the command at state 2 and then STOP.

A routine to trigger CSG DAC slices 0,1,2 is in state 3.

In this way, predefined routines can be stored in the AC table and called by firmware

Example 5 - Using AC to control AROUTE

The following example shows how to cycle through all the MUX options for a muxed sampler in ADC group 0.

At the beginning of each state the routing is changed to the indicated MUX selection and the AC waits in that state until the that ADC group is triggered, sampled, and converted. It then goes to the next state, changes the MUX selection, etc.

In this way, a sequence of routing selections may be configured for a given ADC group and the AC autonomously changes the routing.

```
STATE:TOP
    SAR_AROUTE_TR=1   SAR_AROUTE_SEL=0   ACTION=WAIT_FOR   COND=SAR_ENTRY0_DONE

STATE:
    SAR_AROUTE_TR=1   SAR_AROUTE_SEL=1   ACTION=WAIT_FOR   COND=SAR_ENTRY0_DONE

STATE:
    SAR_AROUTE_TR=1   SAR_AROUTE_SEL=2   ACTION=WAIT_FOR   COND=SAR_ENTRY0_DONE

STATE:
    SAR_AROUTE_TR=1   SAR_AROUTE_SEL=3   ACTION=WAIT_FOR   COND=SAR_ENTRY0_DONE

STATE:
    ACTION=BRANCH_IF_TRUE   COND=TRUE   BR_ADDR=TOP  // Cycle through again
```

31.2.8 Temperature sensor

PSC3M/P6 MCU has an on-chip temperature sensor that is used to measure the internal die temperature.

The temperature sensor consists of a single bipolar junction transistor (BJT) in the form of a diode, the transistor is biased using the reference current from the analog reference block (AREF) with two bias currents 1

uA and 8 uA. Its base-to-emitter voltage (V_{BE}) has a strong dependence on temperature at a constant collector current and zero collector-base voltage. This property is used to calculate the die temperature by measuring the V_{BE} of the transistor using the SAR ADC, as show in Figure 450.

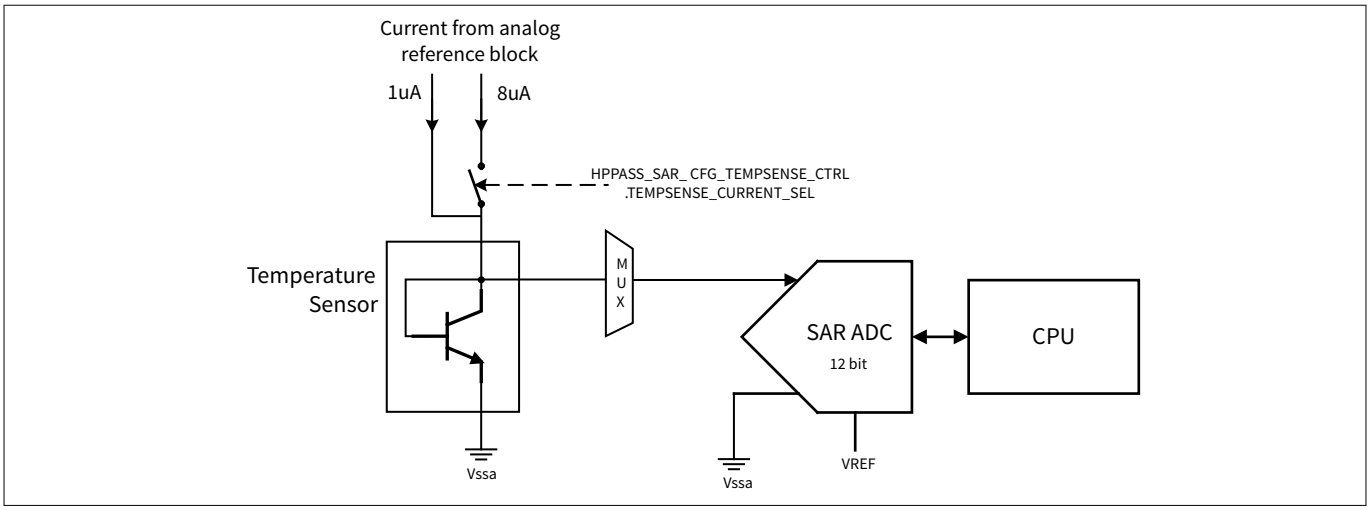


Figure 450 Temperature sensing mechanism

The output of the temperature sensor can be routed to SAR ADC through AMUX1 selection 5, these correspond to ADC channels 23 (see the SAR ADC Channels chapter for more details), the temperature measurement require the following ratio:

$$Ratio = (V_{BE1} - V_{BE2})/V_{BE2} \quad (7)$$

V_{BE1} and V_{BE2} can be measured in the following manner:

- When channel 23 is selected the temperature sensor is biased with a 1 uA current by default, this output corresponds to the measurement V_{BE2}
- By setting TEMPSENSE_CURRENT_SEL bit to “1” in HPPASS_SAR_CFG_TEMPSENSE_CTRL register, an additional 8 uA flows into the BJT, this corresponds to the measurement V_{BE1}

Die temperature can be calculated as given in the following equation:

$$Temperature = A * Ratio^3 + B * Ratio^2 + C * Ratio + D \quad (8)$$

The constant A, B, C, and D are stored in the SFLASH_SAR_TEMP_COEF_A, SFLASH_SAR_TEMP_COEF_B, SFLASH_SAR_TEMP_COEF_C and SFLASH_SAR_TEMP_COEF_D registers during factory calibration.

The flow chart below illustrates the typical sequence of operations as described above:

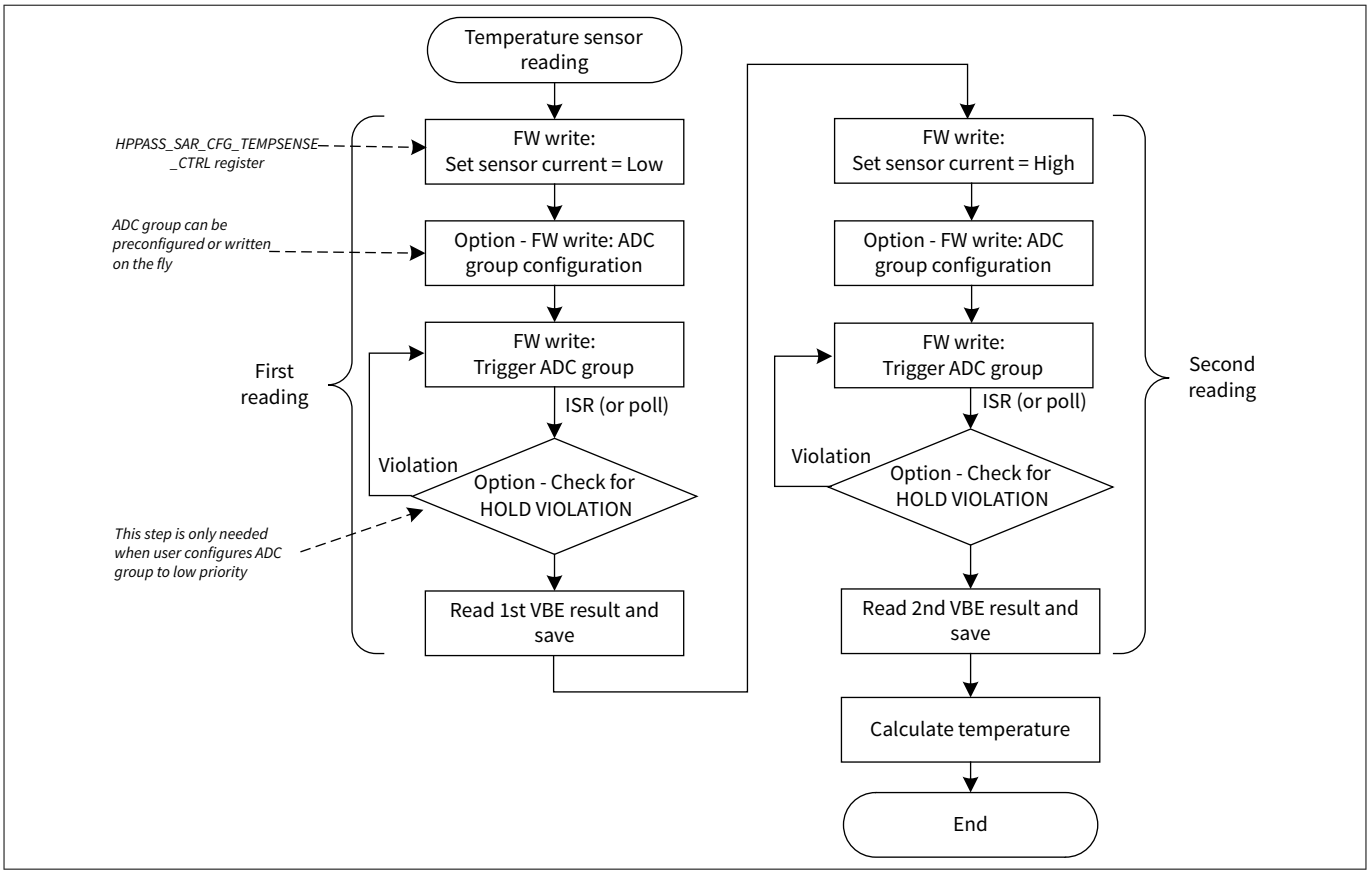


Figure 451 Temperature measurement sequence

In the SAR ADC first reading the V_{BE2} (corresponding to 1uA) is derived, and second reading the V_{BE1} (corresponding to additional 8uA) is derived, then use the equations to calculate temperature.

31.3 Clock

The HPPASS requires only a single clock, clk_hf0 with a maximum frequency of 180 MHz. Figure 452 shows the clock diagram.

The SAR ADC core directly use the clk_hf0 clock. When VDDA is below 2.8 V, the SARADC_CLOCK_MODE bit in the HPPASS_SAR_CFG_CTRL register must be configured to divide clk_hf0 by 2 for the SAR ADC core clock. The clock for the SAR ADC digital Parts is enabled by the AC and also divides clk_hf0 by 2. The CSG clock is configured through peripheral clock divider group 0, with clk_hf0 as its clock source, and is enabled by the AC. The frequency range of CSG clock is 10 MHz to 90 MHz. Refer to the Clocking system chapter for details on the peripheral clock divider. The AC and INFRA clocks also divide the clk_hf0 by 2.

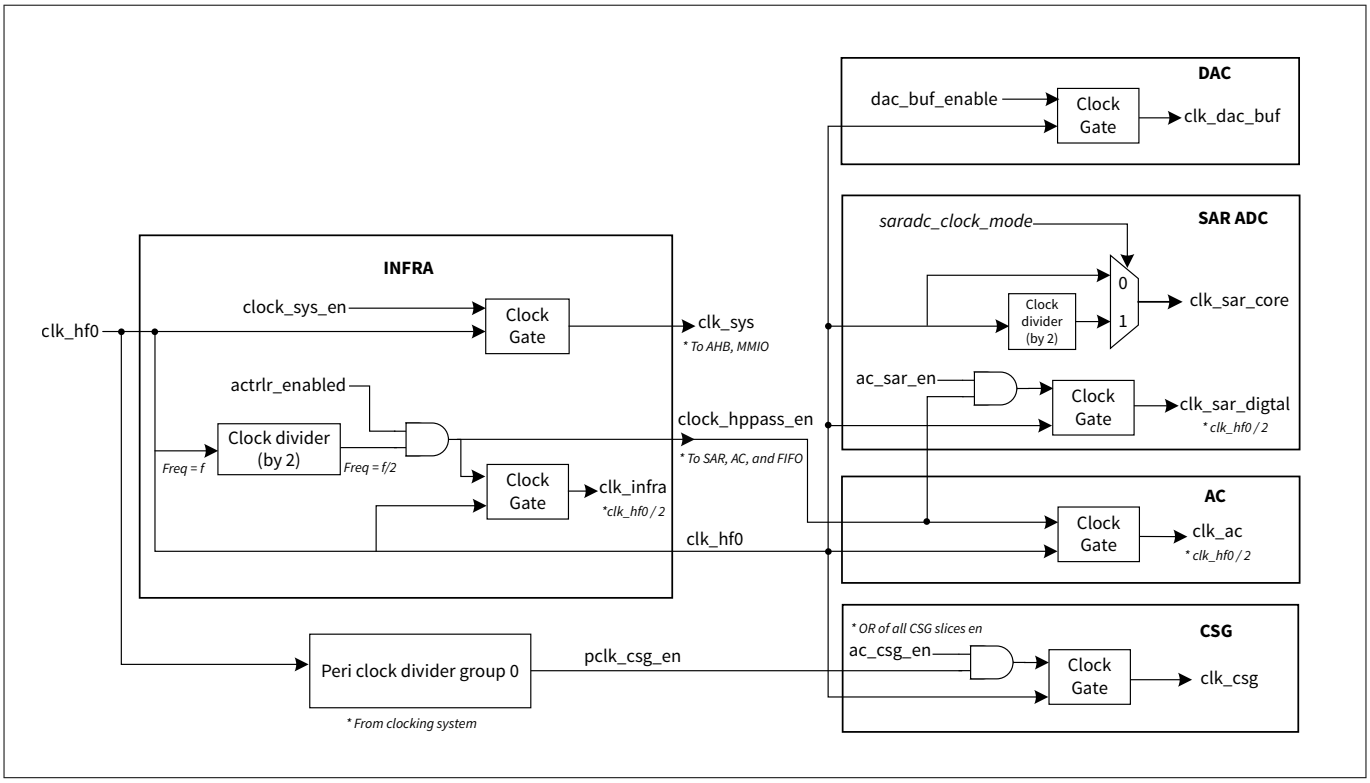


Figure 452 HPPASS clock diagram

31.4 Output interrupts and triggers

Interrupts

There are two main interrupts generated by HPPASS and output as interrupt_hppass and interrupt_fifos.

- **interrupt_hppass** is a combined of interrupt causes that consists of FIFO overflow/underflow, AC programmed interrupt, SAR ADC result overflow and error conditions. About SAR ADC error condition details, see the Error status chapter. Once interrupt is set, user needs to write “1” to HPPASS_MMIO_HPPASS_INTR to reset the interrupt. In order to observe interrupt cause at interrupt_hppass, the respective HPPASS_MMIO_FIFO_INTR_MASK need to be enabled
- **interrupt_fifos** is the combined of FIFO level from each FIFO. FIFO level event occurs when HPPASS_MMIO_FIFO_USEDx.USED >= HPPASS_MMIO_FIFO_LEVELx.LEVEL+1. Unlike interrupt_hppass, user does not require to reset interrupt_fifos. Hardware will reset it when HPPASS_MMIO_FIFO_USEDx.USED < HPPASS_MMIO_FIFO_LEVELx.LEVEL+1. In order to observe FIFO interrupt cause at interrupt_fifos, the respective HPPASS_MMIO_FIFO_INTR_MASK need to be enabled

Triggers

HPPASS have two types of output triggers which are level and pulse, level triggers are intended for use in motor control algorithms primarily to control the TCPWM, whereas pulse triggers are events that are intended primarily for DMA interface and control. There are 3 main triggers generated and output as tr_fifo_level_out[3:0], tr_level_out[15:0] and tr_pulse_out[15:0].

- **tr_fifo_level_out[3:0]** is a level trigger for FIFO level. Trigger causes selection can be configured by FIFO_TR_EN bit-fields of HPPASS_MMIO_FIFO_CFG register. See SAR ADC FIFO chapter for more details
- **tr_level_out[15:0]** is a level trigger for CSG_CMP_TR[8:0], CSG_REMAP_TR[2:0], DAC_FIFO_BELOW_THRESHOLD[1:0] or SAR_RANGE_TR[7:0] outputs. level trigger out causes selection can be configured by TR_LEVEL_OUT register. The selected outputs which are then ORed and routed to the associated level trigger out.

There are 16 sets of HPPASS_MMIO_TR_LEVEL_OUT registers. Means that user have maximum of up to 16 level triggers selection.

There is an option to output synchronous or asynchronous trigger operation by setting HPPASS_MMIO_TR_LEVEL_CFG.BYPASS_SYNC[15:0] register.

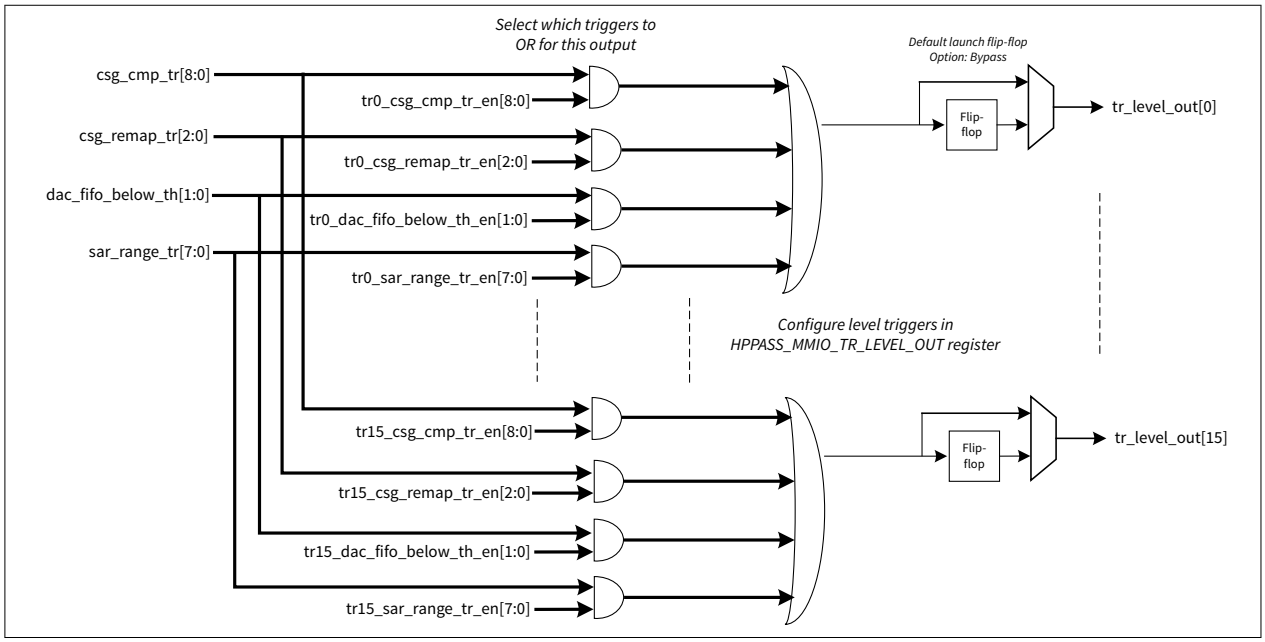


Figure 453 Output level trigger routing

- **tr_pulse_out[15:0]** is a pulse trigger for SAR_ENTRY_DONE[7:0], SAR_FIR_DONE[1:0], AC or HWFILT3P3Z_DONE[2:0] condition completed. Pulse trigger out causes selection can be configured by TR_PULSE_OUT register. The selected outputs which are then muxed and routed to the associated pulse trigger out.

  There are 16 sets of HPPASS_MMIO_TR_PULSE_OUT registers. Means that user have maximum of up to 16 pulse triggers selection.

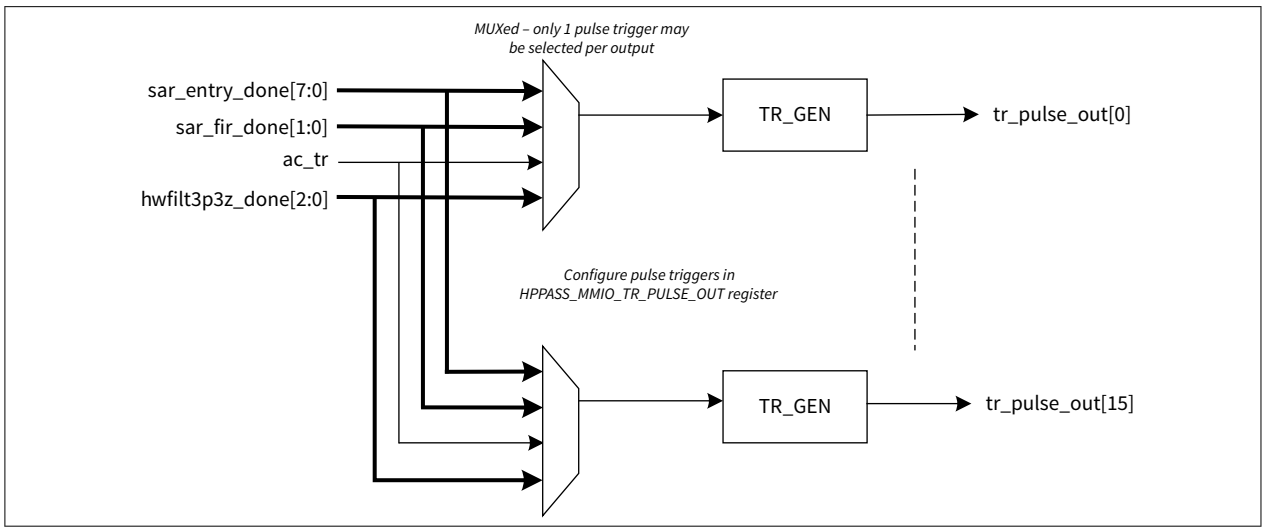


Figure 454 Output pulse trigger routing

31.5 Registers summary

Table 332 HPPASS SAR ADC registers summary

Register	Function
HPPASS_SARADC_CALOFFSTx	SAR ADC offset calibration value registers, 4 offset calibration values with x = 0 to 3
HPPASS_SARADC_CALLINx	SAR ADC linearity calibration value registers, 16 linearity calibration values with x = 0 to 15
HPPASS_SARADC_CALGAINC	SAR ADC coarse-grained gain calibration value register
HPPASS_SARADC_CALGAINF	SAR ADC fine-grained gain calibration value register
HPPASS_SAR_SEQ_ENTRYx	SAR ADC sequencer table group registers, 8 groups with x = 0 to 7
HPPASS_SAR_SEQ_ENTRYx_2	SAR ADC sequencer table group registers 2, 8 groups with x = 0 to 7
HPPASS_SAR_ENTRY_MUX_SEL_ALIASx	SAR ADC sequencer MUX select alias registers, 8 such registers with x = 0 to 7
HPPASS_SAR_CFG_CTRL	SAR ADC control register
HPPASS_SAR_CFG_SAMP_EN	SAR ADC sampler static enable register
HPPASS_SAR_CFG_SAMP_GAIN	SAR ADC sampler gain configuration register
HPPASS_SAR_CFG_SAMPLE_TIMEx	SAR ADC sample time configuration register, 3 sample times with x = 0 to 2
HPPASS_SAR_CFG_FIR_CFGx	FIR configuration registers, 2 FIR filters with x = 0, 1
HPPASS_SAR_CFG_CHAN_CFGx	SAR ADC channel configuration registers, 36 channels with x = 0 to 35
HPPASS_SAR_CFG_RANGE_CFGx	SAR ADC limit detect configuration registers, 8 limit detect resources with x = 0 to 7
HPPASS_SAR_CFG_RANGE_LOWx	SAR ADC limit detect low value registers, 8 limit detect resources with x = 0 to 7
HPPASS_SAR_CFG_RANGE_HIGHx	SAR ADC limit detect high value registers, 8 limit detect resources with x = 0 to 7
HPPASS_SAR_CFG_CHAN_RESULTx	SAR ADC channel result registers, 36 channels with x = 0 to 35
HPPASS_SAR_CFG_FIR_RESULTx	FIR result registers, 2 FIR filters with x = 0, 1
HPPASS_SAR_CFG_RESULT_MASK	SAR ADC channel result mask register
HPPASS_SAR_CFG_RESULT_MASK_2	SAR ADC channel result mask register 2
HPPASS_SAR_CFG_RESULT_UPDATED	SAR ADC channel result status register
HPPASS_SAR_CFG_RESULT_UPDATED_2	SAR ADC channel result status register 2
HPPASS_SAR_CFG_RANGE_STATUS	SAR ADC limit detection status register
HPPASS_SAR_CFG_RANGE_STATUS_2	SAR ADC limit detection status register 2
HPPASS_SAR_CFG_SAR_STATUS	SAR ADC status register

(table continues...)

Table 332 (continued) HPPASS SAR ADC registers summary

Register	Function
HPPASS_SAR_CFG_RESULT_OVERFLOW	SAR ADC overflow status register
HPPASS_SAR_CFG_ENTRY_TR_COLLISION	SAR ADC group trigger collision status register
HPPASS_SAR_CFG_ENTRY_HOLD_VIOLATION	SAR ADC group hold violation status register
HPPASS_SAR_CFG_ENTRY_HOLD_CNT	SAR ADC group hold violation count register
HPPASS_SAR_CFG_ADC_LOOPBACK_CTRL	SAR ADC loopback selection control
HPPASS_SAR_CFG_SAR_RESULT_INTR	SAR ADC interrupt register
HPPASS_SAR_CFG_SAR_RESULT_INTR_SET	SAR ADC interrupt request set register
HPPASS_SAR_CFG_SAR_RESULT_INTR_MASK	SAR ADC interrupt request mask register
HPPASS_SAR_CFG_SAR_RESULT_INTR_MASKED	SAR ADC interrupt request masked register
HPPASS_SAR_CFG_SAR_RANGE_INTR	SAR ADC limit detect interrupt register
HPPASS_SAR_CFG_SAR_RANGE_INTR_SET	SAR ADC limit detect interrupt request set register
HPPASS_SAR_CFG_SAR_RANGE_INTR_MASK	SAR ADC limit detect interrupt request mask register
HPPASS_SAR_CFG_SAR_RANGE_INTR_MASKED	SAR ADC limit detect interrupt request masked register
HPPASS_SAR_CFG_CHAN_RESULT_PACKEDx	SAR ADC result packed register, 18 such registers with x = 0 to 17
HPPASS_SAR_CFG_AROUTE_CTRL_MODE	AROUTE control mode registers
HPPASS_SAR_CFG_AROUTE_FW_CTRL	AROUTE firmware switch control set register
HPPASS_SAR_CFG_AROUTE_FW_CTRL_CLR	AROUTE firmware switch control clear register
HPPASS_SAR_CFG_AROUTE_STATUS	AROUTE switch status register
HPPASS_SAR_CFG_TEMPSENSE_CTRL	Temperature sensor control register
HPPASS_SAR_CFG_HWFILT3P3Z_IND_CTRLx	Hardware 3P3Z filter configuration registers with x = 0 to 2
HPPASS_SAR_FIRx_FIR_COEFSy	FIR coefficient register, 2 16-tap FIRs with x = 0, 1 and y = 0 to 15
HPPASS_SAR_FIRx_FIR_TAPSy	FIR tap vale register, 2 16-tap FIRs with x = 0, 1 and y = 0 to 15

Table 333 HPPASS CSG registers summary

Register	Function
HPPASS_CSG_SLICEx_CMP_CFG	CSG slice comparator control register, 9 slices with x = 0 to 8
HPPASS_CSG_SLICEx_DAC_CFG	CSG slice DAC control register, 9 slices with x = 0 to 8
HPPASS_CSG_SLICEx_DAC_PARAM_SYNC	CSG slice DAC parameter synchronized update register, 9 slices with x = 0 to 8
HPPASS_CSG_SLICEx_DAC_MODE_START	CSG slice DAC mode start register, 9 slices with x = 0 to 8
HPPASS_CSG_SLICEx_DAC_VAL_A	CSG slice DAC value A buffer register, 9 slices with x = 0 to 8
HPPASS_CSG_SLICEx_DAC_VAL_B	CSG slice DAC value B buffer register, 9 slices with x = 0 to 8
HPPASS_CSG_SLICEx_DAC_PERIOD	CSG slice DAC period buffer register, 9 slices with x = 0 to 8

(table continues...)

Table 333 (continued) HPPASS CSG registers summary

Register	Function
HPPASS_CSG_SLICEx_DAC_VAL	CSG slice DAC firmware write register, 9 slices with x = 0 to 8
HPPASS_CSG_SLICEx_DAC_STATUS	CSG slice current DAC value register, 9 slices with x = 0 to 8
HPPASS_CSG_SLICEx_CMP_STATUS	CSG slice comparator status register, 9 slices with x = 0 to 8
HPPASS_CSG_LUT_CFGx_LUT_DATAy	DAC LUT waveform data registers, 3 128-entry LUTs with x = 0 to 2 and y = 0 to 127
HPPASS_CSG_CSG_CTRL	CSG control register
HPPASS_CSG_CSG_GROUP_CFGx	CSG group configuration register with x = 0 to 2
HPPASS_CSG_CSG_OUT_REMAP_CFGx	CSG output remap configuration register with x = 0 to 2
HPPASS_CSG_DAC_INTR	DAC interrupt register
HPPASS_CSG_DAC_INTR_SET	DAC interrupt request set register
HPPASS_CSG_DAC_INTR_MASK	DAC interrupt request mask register
HPPASS_CSG_DAC_INTR_MASKED	DAC interrupt request masked register
HPPASS_CSG_CMP_INTR	CSG comparator interrupt register
HPPASS_CSG_CMP_INTR_SET	CSG comparator interrupt request set register
HPPASS_CSG_CMP_INTR_MASK	CSG comparator interrupt request mask register
HPPASS_CSG_CMP_INTR_MASKED	CSG comparator interrupt request masked register
HPPASS_CSG_VDAC_OUT_BLANK	CSG blanking control for DAC observability register
HPPASS_CSG_CSG_SYNC	CSG sync enable register
HPPASS_CSG_CSG_AROUTEMUX_SELx	CSG AROUTE MUX selection register with x = 0 to 8

Table 334 HPPASS AC registers summary

Register	Function
HPPASS_ACTRLR_CTRL	AC control register
HPPASS_ACTRLR_CMD_RUN	AC run command register
HPPASS_ACTRLR_CMD_STATE	AC state address register
HPPASS_ACTRLR_BLOCK_STATUS	HPPASS block ready status register
HPPASS_ACTRLR_STATUS	AC status register
HPPASS_ACTRLR_CFG	AC configuration register
HPPASS_ACTRLR_CNTR_STATUSx	AC loop/interval counter status registers, 2 counters with x = 0, 1
HPPASS_ACTRLR_TTCFGx_TT_CFG0	AC configuration 0 register, 16 states with x = 0 to 15
HPPASS_ACTRLR_TTCFGx_TT_CFG1	AC configuration 1 register, 16 states with x = 0 to 15
HPPASS_ACTRLR_TTCFGx_TT_CFG2	AC configuration 2 register, 16 states with x = 0 to 15
HPPASS_ACTRLR_TTCFGx_TT_CFG3	AC configuration 3 register, 16 states with x = 0 to 15

Table 335 HPPASS INFRA registers summary

Register	Function
HPPASS_INFRA_TR_IN_SEL	HPPASS trigger input select register 0 - 7
HPPASS_INFRA_TR_IN_SEL_2	HPPASS trigger input select register 8 - 15
HPPASS_INFRA_HW_TR_MODE	HPPASS hardware trigger mode register for trigger 0 - 7
HPPASS_INFRA_HW_TR_MODE_2	HPPASS hardware trigger mode register for trigger 8 - 15
HPPASS_INFRA_FW_TR_PULSE	HPPASS firmware pulse mode trigger register
HPPASS_INFRA_FW_TR_LEVEL	HPPASS firmware level mode trigger register
HPPASS_INFRA_CLOCK_STARTUP_DIV	HPPASS startup clock divider register
HPPASS_INFRA_STARTUP_CFGx	HPPASS startup configuration registers, 4 such registers with x = 0 to 3
HPPASS_INFRA_VDDA_STATUS	Analog voltage status register
HPPASS_INFRA_ANA_CTRL	Analog control register
HPPASS_INFRA_GPIO_ISO_CTRL	GPIO isolation controls register
HPPASS_INFRA_AREFv2_AREF_CTRL	AREF control register

Table 336 HPPASS MMIO registers summary

Register	Function
HPPASS_MMIO_FIFO_INTR	FIFO interrupt request register
HPPASS_MMIO_FIFO_INTR_SET	FIFO interrupt set request register
HPPASS_MMIO_FIFO_INTR_MASK	FIFO interrupt mask register
HPPASS_MMIO_FIFO_INTR_MASKED	FIFO interrupt masked request register
HPPASS_MMIO_HPPASS_INTR	HPPASS interrupt request register
HPPASS_MMIO_HPPASS_INTR_SET	HPPASS interrupt set request register
HPPASS_MMIO_HPPASS_INTR_MASK	HPPASS interrupt mask register
HPPASS_MMIO_HPPASS_INTR_MASKED	HPPASS interrupt masked request register
HPPASS_MMIO_TR_LEVEL_CFG	HPPASS level trigger out configuration register
HPPASS_MMIO_TR_LEVEL_OUTx	HPPASS level trigger out select registers, 16 output level triggers with x = 0 to 15
HPPASS_MMIO_TR_PULSE_OUTx	HPPASS pulse trigger out select registers, 16 output pulse triggers with x = 0 to 15
HPPASS_MMIO_FIFO_CFG	FIFO configuration register
HPPASS_MMIO_FIFO_CLR	FIFO firmware clear register
HPPASS_MMIO_FIFO_LEVELx	FIFO level registers, 4 FIFOs with x = 0 to 3
HPPASS_MMIO_FIFO_RD_DATAx	FIFO read data registers, 4 FIFOs with x = 0 to 3

(table continues...)

Table 336 (continued) HPPASS MMIO registers summary

Register	Function
HPPASS_MMIO_FIFO_USEDx	FIFO used registers, 4 FIFOs with x = 0 to 3
HPPASS_MMIO_FIFO_STATUSx	FIFO status registers, 4 FIFOs with x = 0 to 3

Table 337 HPPASS HW 3P3Z Filter registers summary

Register	Function
HWFILT3P3Z_CTRL	Hardware filter module control register
HWFILT3P3Z_HWFILT3P3Zx_CTRL	Filter control register
HWFILT3P3Z_HWFILT3P3Zx_DATA_IN0	Filter data input 0 register
HWFILT3P3Z_HWFILT3P3Zx_DATA_IN1	Filter data input 1 register
HWFILT3P3Z_HWFILT3P3Zx_CNFG	Filter configuration register
HWFILT3P3Z_HWFILT3P3Zx_CX0	Filter coefficient X0
HWFILT3P3Z_HWFILT3P3Zx_CX1	Filter coefficient X1
HWFILT3P3Z_HWFILT3P3Zx_CX2	Filter coefficient X2
HWFILT3P3Z_HWFILT3P3Zx_CX3	Filter coefficient X3
HWFILT3P3Z_HWFILT3P3Zx_CY1	Filter coefficient Y1
HWFILT3P3Z_HWFILT3P3Zx_CY2	Filter coefficient Y2
HWFILT3P3Z_HWFILT3P3Zx_CY3	Filter coefficient Y3
HWFILT3P3Z_HWFILT3P3Zx_DATA_OUT	Filter data output register
HWFILT3P3Z_HWFILT3P3Zx_OFFSET	Filter offset register
HWFILT3P3Z_HWFILT3P3Zx_LIMMAX	Filter out maximum limit register
HWFILT3P3Z_HWFILT3P3Zx_LIMMIN	Filter out minimum limit register
HWFILT3P3Z_HWFILT3P3Zx_SCALECX	Scale X coefficients factor register
HWFILT3P3Z_HWFILT3P3Zx_SCALECY	Scale Y coefficients factor register
HWFILT3P3Z_HWFILT3P3Zx_GIN	Amplify input difference factor register
HWFILT3P3Z_HWFILT3P3Zx_GOUT	Scale output register
HWFILT3P3Z_HWFILT3P3Zx_AWMAX	Filter Anti Windup Max Limit register
HWFILT3P3Z_HWFILT3P3Zx_AWMIN	Filter Anti Windup Min Limit register
HWFILT3P3Z_HWFILT3P3Zx_AWGAIN	Filter Anti Windup Gain register
HWFILT3P3Z_HWFILT3P3Zx_VERSION	Filter version register

***Note**: In HWFILT3P3Z_HWFILT3P3Zx, 'x' is the filter number.*

Table 338 HPPASS DAC registers summary

Register	Function
DAC_BUFFER_FIFO_DATA_IN	Input data for DAC Buffer Fifo

(table continues...)

Table 338 (continued) HPPASS DAC registers summary

Register	Function
DAC_BUFFER_FIFO_CTRL	DAC Buffer Fifo control and configuration
DAC_BUFFER_CFG	DAC Buffer Control Register
DAC_BUFFER_PARAM_SYNC	DAC Parameter Synchronized Update
DAC_BUFFER_MODE_START	DAC Mode Start
DAC_BUFFER_VAL_A	DAC Vaue A Buffer
DAC_BUFFER_VAL_B	DAC Value B Buffer
DAC_BUFFER_PERIOD	DAC Period Buffer
DAC_BUFFER_VAL	Firmware Write port (excluding Fifo)
DAC_BUFFER_STATUS	Read of Current DAC Value
DAC_BUFFER_ANA_CTRL_1	Control Bits
DAC_BUFFER_ANA_CTRL_2	Control Bits
DAC_BUFFER_INTR	DAC + Buffer Interrupt register
DAC_BUFFER_INTR_SET	DAC + Buffer Interrupt request set register
DAC_BUFFER_INTR_MASK	DAC + Buffer Interrupt request mask
DAC_BUFFER_INTR_MASKED	DAC + Buffer Interrupt request masked
DAC_BUFFER_LUT	DAC LUT waveform data

Table 339 HPPASS OpAmp registers summary

Register	Function
AFE_CONFIG	OpAmp Configuration, includes Input Mux, Op Amp Configuration
AFE_STATUS	OpAmp Status
AFECONFIG_CTRL	OpAmp Configuration Multiplex Control
AFE_ANALOG_CTRL	Analog Control
AFE_ANALOG_TST_CTRL	Analog Test Control

32 Low-power comparator (LPCOMP)

PSC3M/P6 MCU has two low-power comparators, which can perform analog-signal comparison in all system power modes. The low-power comparator output can be:

- Inspected by the CPU
- Used as an interrupt/wake-up source to the CPU when in CPU Sleep mode
- Used as a wake-up source to system resources when in System DeepSleep, DeepSleep-RAM/OFF or Hibernate mode, or fed to a GPIO or trigger multiplexer (see Trigger multiplexer (Trigger MUX) chapter) as an asynchronous or synchronous signal (level or pulse)

32.1 Features

The PSC3M/P6 MCU low-power comparators have the following features:

- Programmable power and speed
- Ultra-low-power mode support
- Each comparator features a one-sided hysteresis option
- Rising edge, falling edge, and combined rising and falling edge detection at the comparator output
- Local reference voltage generation
- Wake-up source from low-power modes

32.2 Architecture

The following figure shows the block diagram for the low-power comparator.

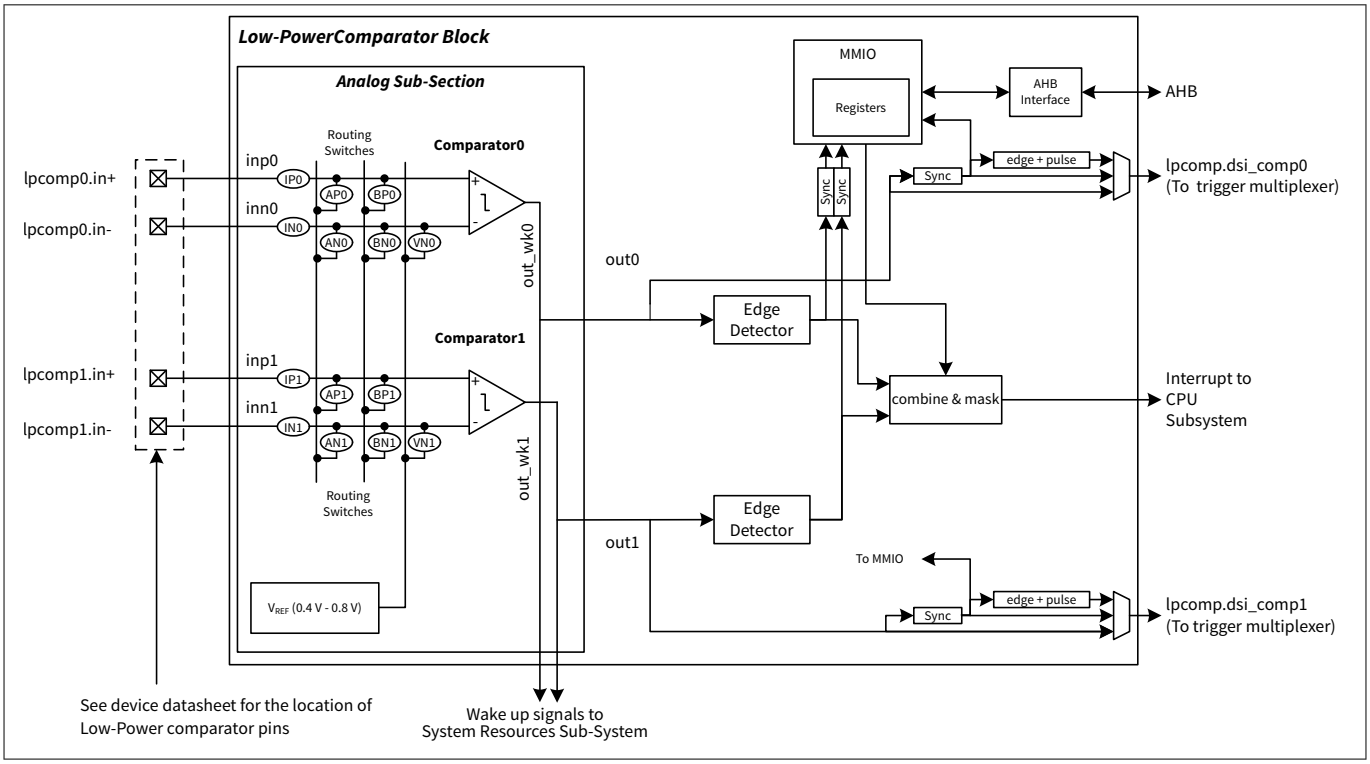


Figure 455 Low-power comparator block diagram

The block, on a higher level, consists of two subsections – analog and digital logic. The analog subsection consists of two comparators with the option of hysteresis, switches to make connection to the inputs, and the reference voltage source. The output from the comparators is driven to the System Resources Subsystem (SRSS) directly for device wake-up from Hibernate mode.

The comparator output is also driven to the digital subsection. This subsection provides different output options, such as synchronous level and pulse output, and options to trigger interrupts on positive, negative, or both edges of the comparator outputs to wake up the device from Sleep, DeepSleep, DeepSleep-RAM, and DeepSleep-OFF modes.

The following sections describe the operation of the PSC3M/P6 MCU low-power comparator, including input configuration, power and speed modes, output and interrupt configuration, hysteresis, and wake-up from low-power modes.

32.2.1 Input configuration

Low-power comparators operate with the following input options:

- Compare two voltages from external pins
- Compare external signals with a locally generated reference voltage. Note that this voltage is not a precision reference and can vary from 0.4 V to 0.8 V

The internal routing switches (IP0, AP0, BP0, IN0, AN0, BN0, VN0, IP1, AP1, BP1, IN1, AN1, BN1, VN1) shown in Figure 455 are controlled using the LPCOMP_CMP0_SW, LPCOMP_CMP1_SW, LPCOMP_CMP0_SW_CLEAR, and LPCOMP_CMP1_SW_CLEAR registers. See the register reference manual to understand how to control these switches.

If the inverting input of a comparator is routed to a local voltage reference, set the LPREF_EN bit in the LPCOMP_CONFIG register to enable the voltage reference. See the register reference manual for the details of these registers.

32.2.2 Output and interrupt configuration

The output can be configured to one of the following options using the control registers (LPCOMP_CMP0_CTRL and LPCOMP_CMP1_CTRL):

- Direct comparator output
- Synchronous comparator output - level
- Synchronous comparator output - pulse

The DSI_BYPASS0 and DSI_BYPASS1 bits of the control register select the direct or synchronous comparator output.

The DSI_LEVEL0 and DSI_LEVEL1 bits of the control register select between pulse or level output. Pulse output is two clk_sys cycles, and it can be generated on a positive edge, a negative edge, or on both edges configured using the INTTYPE0 and INTTYPE1 bits of the control register. Table 340 summarizes the configuration.

Table 340 Output configuration

DSI_BYPASSx	DSI_LEVELx	Output
1	x	Direct comparator output
0	1	Synchronous comparator output - level
0	0	Synchronous comparator output - pulse

In addition, comparator outputs can be routed to GPIOs and other on-chip peripherals, such as DMA and TCPWMs, using the trigger multiplexer. See the Trigger multiplexer (Trigger MUX) chapter for more details.

Firmware readouts of comparator 0 and comparator 1 outputs are available at the OUT0 and OUT1 bits of the LPCOMP_STATUS register (Table 341). The output of each comparator is connected to a corresponding edge detector block. This block determines the edge that triggers the interrupt. The edge selection and interrupt enable are configured using the INTTYPE0 and INTTYPE1 bitfields in the LPCOMP_CMP0_CTRL and LPCOMP_CMP1_CTRL registers for comparator 0 and comparator 1, respectively. Using the INTTYPE0 and

INTTYPE1 bits, the interrupt type can be selected to disabled, rising edge, falling edge, or both edges, as described in Table 341.

During an edge event, the comparator triggers an interrupt. The interrupt request is registered in the COMP0 bit and COMP1 bit of the LPCOMP_INTR register for comparator 0 and comparator 1, respectively. Both comparator 0 and comparator 1 share a common interrupt signal output (see Table 341), which is the logical OR of the two interrupts and mapped as the low-power comparator block's interrupt in the CPU NVIC. Refer to the Interrupts section for details. If both comparators are used in a design, read the COMP0 and COMP1 bits of the LPCOMP_INTR register in the interrupt service routine to know which one triggered the interrupt. Alternatively, the COMP0_MASK bit and the COMP1_MASK bit of the LPCOMP_INTR_MASK register can be used to mask the comparator 0 and comparator 1 interrupts to the CPU. Only the masked interrupts are serviced by the CPU. After the interrupt is processed, clear the interrupt by writing a '1' to the COMP0 and COMP1 bits of the LPCOMP_INTR register in the firmware. If the interrupt to the CPU is not cleared, it says active regardless of the next compare events and interrupts the CPU continuously. Refer to the Interrupts chapter for details.

The LPCOMP_INTR_SET register bits [1:0] can be used to assert an interrupt for firmware debugging.

In the DeepSleep, DeepSleep-RAM or DeepSleep-OFF mode, the wake-up interrupt controller (WIC) can be activated by a comparator edge event, which then wakes up the CPU. Similarly, in the Hibernate mode, the LPCOMP can wake up the system resources subsystem(SRSS), which includes the clocks. Thus, the LPCOMP can monitor a specified signal in low-power modes. See the Power supply and monitoring chapter and the Device power modes chapter for more details.

Table 341 Output and interrupt configuration

Register[Bit_Pos]	Bit_Name	Description
LPCOMP_STATUS[0]	OUT0	Current/Instantaneous output value of Comparator 0
LPCOMP_STATUS[16]	OUT1	Current/Instantaneous output value of Comparator 1
LPCOMP_CMP0_CTRL[7:6]	INTTYPE0	Sets on which edge Comparator 0 will trigger an IRQ 00: Disabled 01: Rising edge 10: Falling edge 11: Both rising and falling edges
LPCOMP_CMP1_CTRL[7:6]	INTTYPE1	Sets on which edge Comparator 1 will trigger an IRQ 00: Disabled 01: Rising edge 10: Falling edge 11: Both rising and falling edges
LPCOMP_INTR[0]	COMP0	Comparator 0 Interrupt: hardware sets this interrupt when Comparator 0 triggers Write a '1' to clear the interrupt
LPCOMP_INTR[1]	COMP1	Comparator 1 Interrupt: hardware sets this interrupt when Comparator 1 triggers Write a '1' to clear the interrupt
LPCOMP_INTR_SET[0]	COMP0	Write a '1' to trigger the software interrupt for Comparator 0
LPCOMP_INTR_SET[1]	COMP1	Write a '1' to trigger the software interrupt for Comparator 1
LPCOMP_INTR_MASK[0]	COMP0_MASK	Write a ‘1’ to enable the Comparator 0 interrupt
LPCOMP_INTR_MASK[1]	COMP1_MASK	Write a ‘1’ to enable the Comparator 1 interrupt

(table continues...)

Table 341 (continued) Output and interrupt configuration

Register[Bit_Pos]	Bit_Name	Description
LPCOMP_INTR_MASKED[0]	COMP0_MASKED	If 1, indicates that Comparator 0 interrupt has triggered
LPCOMP_INTR_MASKED[1]	COMP1_MASKED	If 1, indicates that Comparator 1 interrupt has triggered

32.2.3 Power mode and speed configuration

The low-power comparators can operate in three power modes:

- Normal
- Low-power
- Ultra-low-power

The power or speed setting for comparator 0/1 is configured using the MODE0/1 bitfield of the LPCOMP_CMPx_CTRL register, where x = 0,1, respectively. The power consumption and response time vary depending on the selected power mode; power consumption is highest in normal mode and lowest in ultra-low-power mode, and response time is fastest in normal mode and slowest in ultra-low-power mode. Refer to the device datasheet for specifications for the response time and power consumption for various power settings.

The comparators can also be enabled or disabled using these bitfields, as described in Table 342.

***Note**: The output of the comparator may glitch when the power mode is changed while the comparator is enabled. To avoid this, disable the comparator before changing the power mode.*

Table 342 Comparator power mode selection bits

Register[Bit_Pos]	Bit_Name	Description
LPCOMP_CMP0_CTRL[1:0]	MODE0	Comparator 0 power mode selection 00: Off 01: Ultra-low-power operating mode. Use this mode when the device is in DeepSleep or Hibernate mode 10: Low-power operating mode 11: Normal, full-speed, full-power operating mode See the datasheet for electrical specifications in each power mode.
LPCOMP_CMP1_CTRL[1:0]	MODE1	Comparator 1 power mode selection 00: Off 01: Ultra-low-power operating mode. Use this mode when the device is in DeepSleep or Hibernate mode 10: Low-power operating mode 11: Normal, full-speed, full-power operating mode See the datasheet for electrical specifications in each power mode.

Additionally, the entire low-power comparator system can be enabled or disabled globally using the LPCOMP_CONFIG[31] bit. See the register reference manual for details on these bitfields.

32.2.4 Hysteresis

For applications that compare signals close to each other and slow-changing signals, hysteresis helps to avoid oscillations at the comparator output when the signals are noisy. For such applications, a fixed hysteresis may be enabled in the comparator block. See the device datasheet for the hysteresis voltage range.

The hysteresis level is enabled/disabled by using the HYST0 and HYST1 bitfields in the LPCOMP_CMP0_CTRL and LPCOMP_CMP1_CTRL registers for comparator 0 and comparator 1, as described in Table 343.

Table 343 Hysteresis control bits

Register[Bit_Pos]	Bit_Name	Description
LPCOMP_CMP0_CTRL[5]	HYST0	Enable/Disable hysteresis to Comparator 0 1: Enable Hysteresis 0: Disable Hysteresis See the datasheet for the hysteresis voltage range.
LPCOMP_CMP1_CTRL[5]	HYST1	Enable/Disable hysteresis to Comparator 1 1: Enable Hysteresis 0: Disable Hysteresis See the datasheet for the hysteresis voltage range.

32.2.5 Wake up from low-power modes

The comparator is operational in the device's low-power modes, including Sleep, DeepSleep, DeepSleep-RAM, DeepSleep-OFF and Hibernate modes. The comparator output can wake the device from Sleep, DeepSleep and DeepSleep-RAM/OFF modes. The comparator output can generate interrupts in DeepSleep and DeepSleep-RAM/OFF mode when enabled in the LPCOMP_CONFIG register. Do not set the INTTYPEx bits in the LPCOMP_CMPx_CTRL register to disabled, and set the INTR_MASKx bit in the LPCOMP_INTR_MASK register for the corresponding comparator to wake the device from low-power modes.

The comparators have the following restrictions when operating in DeepSleep, DeepSleep-RAM/OFF and Hibernate modes:

- Comparators should be operated in ultra-low-power mode
- Input to the comparator should come from the dedicated pins. The internal reference, VREF, is available and can be connected to the inverting inputs of the comparator

***Note**: Set the LPREF_EN bit in the LPCOMP_CONFIG register to '1', to use the block in DeepSleep, DeepSleep-RAM/OFF and Hibernate modes. This bit controls the local reference voltage VREF as well as the bias current required for the block's operation in low-power modes.*

32.2.6 Comparator clock

The comparator uses the system main clock clk_sys as the clock which is from CLK_HF[1] for interrupt and output synchronization. See the Clocking system chapter for more details.

32.2.7 Using low-power comparator

Follow these steps to configure the comparator:

1. Configure the internal routing switches to connect the inputs to the comparator. See the LPCOMP_CMP0_SW, LPCOMP _CMP1_SW, LPCOMP_CMP0_SW_CLE AR, and LPCOMP_CMP1_SW_CLEAR registers in the registers reference manual to understand how to control the internal routing switches

2. Enable the LPCOMP block by writing into the ENABLED bit of the LPCOMP_CONFIG register
3. Configure the hysteresis and the output type – direct, synchronous-level, or synchronous-pulse – using the LPCOMP_CMP0_CTRL or LPCOMP_CMP1_CTRL register
4. If the internal reference VREF is being used and connected to an inverting input, enable it by writing into the LPREF_EN bit in the LPCOMP_CONFIG register
5. Set the power mode of the comparator by writing into the mode bits of the LPCOMP_CMP0_CTRL or LPCOMP_CMP1_CTRL register. If operation is required during DeepSleep, DeepSleep-RAM/OFF and Hibernate mode, make sure that the power mode is configured to ultra-low-power and the LPREF_EN bit of the LPCOMP_CONFIG register is set
6. Provide a delay so that the comparator stabilizes before routing the output or enabling the interrupt in the next step. The duration of the delay depends on the power mode configured. See the device datasheet for the delay numbers
7. Route the output to the GPIOs or to other peripherals by configuring the trigger multiplexer
8. If an interrupt is required, configure the interrupt type by writing into the LPCOMP_CMP0_CTRL or LPCOMP_CMP1_CTRL register, and enable the interrupt by writing into the LPCOMP_INTR_MASK register

32.3 Registers summary

Table 344 Low-power comparator register summary

Register	Function
LPCOMP_CONFIG	LPCOMP global configuration register
LPCOMP_INTR	LPCOMP interrupt register
LPCOMP_INTR_SET	LPCOMP interrupt set register
LPCOMP_INTR_MASK	LPCOMP interrupt request mask register
LPCOMP_INTR_MASKED	LPCOMP masked interrupt output register
LPCOMP_STATUS	Output status register
LPCOMP_CMP0_CTRL	Comparator 0 configuration register
LPCOMP_CMP1_CTRL	Comparator 1 configuration register
LPCOMP_CMP0_SW	Comparator 0 switch control register
LPCOMP_CMP1_SW	Comparator 1 switch control register
LPCOMP_CMP0_SW_CLEAR	Comparator 0 switch control clear register
LPCOMP_CMP1_SW_CLEAR	Comparator 1 switch control clear register

Revision history

Document revision	Date	Description of changes
*A	2026-08-31	Release to web

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2026-08-31

Published by

Infineon Technologies AG

81726 Munich, Germany

© 2026 Infineon Technologies AG

All Rights Reserved.

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-rok1761312934928

Important notice

Products which may also include samples and may be comprised of hardware or software or both ("Product(s)") are sold or provided and delivered by Infineon Technologies AG and its affiliates ("Infineon") subject to the terms and conditions of the frame supply contract or other written agreement(s) executed by a customer and Infineon or, in the absence of the foregoing, the applicable Sales Conditions of Infineon. General terms and conditions of a customer or deviations from applicable Sales Conditions of Infineon shall only be binding for Infineon if and to the extent Infineon has given its express written consent.

For the avoidance of doubt, Infineon disclaims all warranties of non-infringement of third-party rights and implied warranties such as warranties of fitness for a specific use/purpose or merchantability. Infineon shall not be responsible for any information with respect to samples, the application or customer's specific use of any Product or for any examples or typical values given in this document.

If this document is marked as 'Preliminary', 'Target', 'Draft' or 'Proposal', Infineon reserves the right to change all information given in this document at any time without notice. Subject to the development and release of a Product for series supply by Infineon, the technical specifications of the Product are set forth in the relevant datasheet provided by Infineon without such marking.

The data contained in this document is exclusively intended for technically qualified and skilled customer representatives. It is the responsibility of the customer to evaluate the suitability of the Product for the intended application and the customer's specific use and to verify all relevant technical data contained in this document in the intended application and the customer's specific use. The customer is responsible for properly designing, programming, and testing the functionality and safety of the intended application, as well as complying with any legal requirements related to its use.

Unless otherwise explicitly approved by Infineon, Products may not be used in any application where a failure of the Products or any consequences of the use thereof can reasonably be expected to result in personal injury.

However, the foregoing shall not prevent the customer from using any Product in such fields of use that Infineon has explicitly designed and sold it for, provided that the overall responsibility for the application lies with the customer. Infineon expressly reserves the right to use its content for commercial text and data mining (TDM) according to applicable laws, e.g. Section 44b of the German Copyright Act (UrhG). If the Product includes security features:

Because no computing device can be absolutely secure, and despite security measures implemented in the Product, Infineon does not guarantee that the Product will be free from intrusion, data theft or loss, or other breaches ("Security Breaches"), and Infineon shall have no liability arising out of any Security Breaches.

If this document includes or references software:

The software is owned by Infineon under the intellectual property laws and treaties of the United States, Germany, and other countries worldwide. All rights reserved. Therefore, you may use the software only as provided in the software license agreement accompanying the software.

If no software license agreement applies, Infineon hereby grants you a personal, non-exclusive, non-transferable license (without the right to sublicense) under its intellectual property rights in the software (a) for software provided in source code form, to modify and reproduce the software solely for use with Infineon hardware products, only internally within your organization, and (b) to distribute the software in binary code form externally to end users, solely for use on Infineon hardware products. Any other use, reproduction, modification, translation, or compilation of the software is prohibited.

For further information on the Product, technology, delivery terms and conditions, and prices, please contact your nearest Infineon office or visit www.infineon.com.