CYW20829 architecture reference manual

About this document

Scope and purpose

This CYW20829 MCU technical reference manual provides comprehensive and detailed information about the CYW20829 MCU device hardware functions. This reference manual is not recommended for users new to the CYW20829 MCU or as a guide for developing CYW20829 MCU applications. Use these documents instead:

- Device datasheet
- Peripheral Driver Library (PDL) documentation
- Application notes
- Code examples

Intended audience

This document is intended for anyone who uses the CYW20829 MCU device.

Table of contents

About this document . . . 1
Table of contents . . . 2
1 Introduction . . . 19
1.1 Features . . . 19
1.2 Architecture . . . 21
1.3 Document organization and conventions . . . 21
1.3.1 Major sections . . . 21
1.3.2 Documentation conventions . . . 21
1.3.2.1 Numeric naming . . . 21
1.3.2.2 Units of measure . . . 22
2 CPU . . . 24
2.1 Features . . . 24
2.2 Architecture . . . 24
2.2.1 CPUSS . . . 24
2.2.2 Address and memory maps . . . 25
2.2.2.1 Wait-state lookup tables . . . 26
2.3 Operating modes and privilege levels . . . 26
2.4 Instruction set . . . 27
2.5 Registers summary . . . 27
3 Bus infrastructure . . . 31
3.1 SYSCPUSS AHB5 interconnect . . . 31
3.1.1 AHB5 bus matrix interconnect . . . 32
3.1.2 SYSCPUSS buses . . . 33
3.1.2.1 Cortex® -M33 C-AHB bus . . . 33
3.1.2.2 Cortex® -M33 S-AHB bus . . . 33
3.1.3 DMA bus . . . 33
3.2 BTSS AHB5 interconnect . . . 33
3.3 IPC . . . 34
3.4 AHB5 MMIO . . . 34
3.5 Peripheral interconnect . . . 34
3.5.1 AHB bus error handling . . . 35
3.6 Bus master peripherals . . . 36
3.6.1 Direct Memory Access(DMA-DW) . . . 36
3.6.2 CryptoLite . . . 36
4 Inter-processor communication (IPC) . . . 37
4.1 Features . . . 37
4.2 Architecture . . . 37
4.2.1 IPC channel . . . 37

Table of contents

4.2.2 IPC interrupt . . . 38
4.2.3 IPC channels and interrupts . . . 39
4.3 Implementing locks . . . 40
4.4 Message passing . . . 41
4.5 Typical usage models . . . 43
4.5.1 Full duplex communication . . . 43
4.5.2 Half-duplex with independent event handling . . . 44
4.5.3 Half-duplex with shared event handling . . . 45

5 SRAM . . . 47
5.1 Features . . . 47
5.2 Architecture . . . 47
5.2.1 AHB5 read transfers . . . 47
5.2.2 32-bit AHB5 write transfers . . . 47
5.2.3 Partial AHB5 write transfers . . . 47
5.2.4 Power partition control . . . 47
5.3 Wait states . . . 48

6 eFuse . . . 49
6.1 Features . . . 49
6.2 Architecture . . . 49

7 Interrupts . . . 51
7.1 Features . . . 51
7.2 Architecture . . . 51
7.3 Interrupts and exceptions - operation . . . 52
7.3.1 Interrupt/exception handling . . . 52
7.3.2 Level interrupts . . . 52
7.3.3 Exception vector table . . . 53
7.4 Exception sources . . . 54
7.4.1 Reset exception . . . 54
7.4.2 Non-maskable interrupt exception . . . 54
7.4.3 HardFault exception . . . 55
7.4.4 Memory management fault exception . . . 55
7.4.5 Bus fault exception . . . 55
7.4.6 Usage fault exception . . . 56
7.4.7 Supervisor Call (SVCall) exception . . . 56
7.4.8 DebugMonitor exception . . . 56
7.4.9 PendSupervisory (PendSV) exception . . . 56
7.4.10 System Tick (SysTick) exception . . . 56
7.5 Interrupt sources . . . 57
7.6 Interrupt/exception priority . . . 60
7.7 Enabling and disabling interrupts/exceptions . . . 61
7.8 Interrupt/exception states . . . 62

Table of contents

7.8.1 Pending interrupts/exception 62
7.9 Stack usage for interrupts/exceptions 64
7.10 Interrupts and low-power modes 64
7.11 Interrupt/exception – Initialization/configuration 65

8 Fault monitoring 66
8.1 Features 66
8.2 Architecture 66
8.2.1 Fault report 67
8.2.2 Signaling interface 67
8.2.3 Monitoring 68
8.2.4 Low-power mode operation 68
8.2.5 Using a fault structure 69
8.2.6 CPU exceptions versus fault monitoring 69
8.3 Fault sources 69
8.4 Registers summary 71

9 Direct memory access (DMA DW) 73
9.1 Features 73
9.2 Architecture 74
9.2.1 Standard flow 74
9.3 Channels 75
9.3.1 Channel priority 75
9.3.2 Channel security attributes 75
9.3.3 Channel state 75
9.3.4 Channel interrupt generation 76
9.4 Descriptors 77
9.4.1 Descriptor memory transfer types 78
9.4.1.1 Single transfer 78
9.4.1.2 1D memory transfer 79
9.4.1.3 2D memory transfer 79
9.4.1.4 Cyclic redundancy checker (CRC) transfer 80
9.4.2 Descriptor size 81
9.4.3 Descriptor configuration 81
9.4.4 Descriptor transfer size 83
9.4.5 Descriptors chaining 84
9.5 Triggers 85
9.5.1 Input triggers 85
9.5.2 Pending triggers 87
9.5.3 Output triggers 87
9.6 Performance 88
9.6.1 Single transfer 88
9.6.2 1D transfer 88

Table of contents

9.6.3 2D transfer . . . 89
9.6.4 CRC transfer . . . 89

10 Trigger MUX . . . 90
10.1 Features . . . 90
10.2 Architecture . . . 90
10.3 Functional description . . . 92
10.4 Software triggers . . . 103
10.5 Trigger functionality . . . 103
10.6 Registers summary . . . 106

11 Debug and trace system . . . 109
11.1 Features . . . 109
11.2 Architecture . . . 109
11.2.1 Debug access port (DAP) . . . 111
11.2.1.1 DAP power domain . . . 111
11.2.1.2 DAP security . . . 111
11.2.2 ROM tables . . . 112
11.2.3 Trace . . . 113
11.2.4 Embedded cross triggering . . . 113
11.3 Serial wire debug (SWD) interface . . . 113
11.4 JTAG interface . . . 113
11.5 Programming the external flash . . . 114
11.5.1 SWD port acquisition . . . 114
11.5.2 SWD programming mode entry . . . 115
11.5.3 SWD programming routine executions . . . 115
11.6 CYW20829 SWD debug interface . . . 115
11.6.1 Debug control and configuration registers . . . 115
11.6.2 Breakpoint unit (BPU) . . . 116
11.6.3 Data watchpoint (DWT) . . . 116
11.6.4 Debugging the CYW20829 device . . . 116

12 Cryptographic accelerator . . . 118
12.1 Features . . . 118
12.2 Architecture . . . 118
12.2.1 Randomized clock . . . 119
12.2.2 Hardware/software interface . . . 119
12.3 Block functions . . . 121
12.3.1 AES-128 . . . 121
12.3.2 SHA-256 . . . 122
12.3.3 Vector unit (VU) . . . 123
12.3.4 True Random Number Generator (TRNG) . . . 126
12.4 Interrupts . . . 128
12.4.1 TRNG interrupt . . . 128

Table of contents

12.4.2 Error interrupt . . . 128
12.5 Registers summary . . . 129
13 Protection units . . . 130
13.1 Architecture . . . 131
13.2 Protection context . . . 134
13.2.1 Special protection contexts . . . 135
13.3 Master protection attributes . . . 136
13.3.1 Master security controller (MSC) . . . 136
13.4 Slave protection attributes . . . 137
13.4.1 Memory protection unit (MPU) . . . 137
13.4.2 Memory protection controller (MPC) . . . 137
13.4.2.1 Software view . . . 138
13.4.2.2 MPC MMIO register protection . . . 139
13.4.3 Peripheral protection controller (PPC) . . . 140
13.4.4 PPC access control rules . . . 140
13.4.4.1 Security aware peripherals . . . 141
13.4.4.2 PPC regions . . . 142
14 Secure boot . . . 152
14.1 Features . . . 153
14.2 Architecture . . . 153
14.2.1 Policy and key . . . 153
14.2.1.1 Policies . . . 153
14.2.1.2 Keys . . . 154
14.2.2 Lifecycle stages (LCS) . . . 154
14.2.2.1 IFX_MFG . . . 155
14.2.2.2 Normal provisioned . . . 155
14.2.2.3 Secured . . . 155
14.2.2.4 RMA . . . 155
14.2.3 Boot sequence . . . 155
14.2.3.1 Protection states . . . 156
14.2.4 CM33_L1_BOOT . . . 156
14.2.4.1 Secured boot . . . 156
14.2.4.2 Alternate serial interface . . . 156
14.2.5 Secured provisioning and re-provisioning . . . 157
14.2.6 Protection context used in secure boot . . . 158
15 Power supply and monitoring . . . 159
15.1 Features . . . 159
15.2 Architecture . . . 160
15.3 Power supply . . . 161
15.3.1 Core regulators . . . 161
15.3.2 SRSS reference block . . . 162

Table of contents

15.4 Voltage monitoring . . . 162
15.4.1 Power-on reset (POR) . . . 162
15.4.2 Brownout detection (BOD) . . . 162
15.4.3 Low-voltage detection (LVD) . . . 163
15.5 Sleep control . . . 163
15.6 Registers summary . . . 163

16 Device power modes . . . 165
16.1 Features . . . 165
16.2 Architecture . . . 165
16.2.1 System power modes . . . 168
16.2.1.1 System Low-Power mode . . . 168
16.2.1.2 System Ultra Low-Power mode . . . 168
16.2.1.3 System Deep Sleep mode . . . 169
16.2.1.4 System Hibernate mode . . . 169
16.2.2 CPU power modes . . . 170
16.2.2.1 CPU Active mode . . . 170
16.2.2.2 CPU Sleep mode . . . 170
16.2.2.3 CPU Deep Sleep mode . . . 170
16.2.3 Other operation states . . . 171
16.2.3.1 Reset state . . . 171
16.2.3.2 Off state . . . 171
16.3 Power mode transitions . . . 172
16.4 Summary . . . 174
16.5 Registers summary . . . 175

17 Clocking system . . . 177
17.1 Features . . . 177
17.2 Architecture . . . 178
17.3 Clock sources . . . 178
17.3.1 Internal main oscillator (IMO) . . . 178
17.3.2 Internal low-speed oscillator (ILO) . . . 179
17.3.3 Internal high-frequency oscillator (IHO) . . . 179
17.3.4 Precision internal low-speed oscillator (PILO) . . . 179
17.3.5 Watch crystal oscillator (WCO) . . . 179
17.3.6 External crystal oscillator (ECO) . . . 179
17.3.7 Medium frequency oscillator (MFO) . . . 180
17.3.8 External clock (EXTCLK) . . . 180
17.3.9 Alternate high-frequency clock (ALTHF) . . . 180
17.4 Clock generation . . . 181
17.4.1 Frequency-locked loop (FLL) . . . 181
17.5 Clock trees . . . 181
17.5.1 Path clocks . . . 182

Table of contents

17.5.2 High-frequency root clocks (CLK_HF) . . . 182
17.5.3 Medium-frequency clock (CLK_MF) . . . 183
17.5.4 Low-frequency clock (CLK_LF) . . . 183
17.5.5 Backup clock (CLK_BAK) . . . 184
17.5.6 Timer clock (CLK_TIMER) . . . 184
17.5.7 Group clocks (CLK_SYS) . . . 184
17.6 CLK_HF clock distribution network . . . 185
17.7 Peripheral clock dividers . . . 185
17.7.1 Fractional clock dividers . . . 186
17.7.2 Peripheral clock divider configuration . . . 186
17.7.2.1 Phase aligning dividers . . . 186
17.7.2.2 Connecting dividers to peripherals . . . 188

18 Watchdog timer (WDT) . . . 189
18.1 Features . . . 189
18.2 Architecture . . . 189
18.3 Free-running WDT . . . 190
18.3.1 Overview . . . 190
18.3.2 Watchdog reset . . . 193
18.3.3 Watchdog interrupt . . . 194
18.4 Multi-counter WDT . . . 194
18.4.1 Overview . . . 194
18.4.1.1 MCWDT0_WDT0 and MCWDT0_WDT1 counter operations . . . 195
18.4.1.2 MCWDT0_WDT2 counter operation . . . 197
18.4.2 Enabling and disabling WDT . . . 199
18.4.3 Watchdog cascade options . . . 200
18.4.4 MCWDT reset . . . 201
18.4.5 MCWDT interrupt . . . 202
18.5 Reset cause detection . . . 203
18.6 Registers summary . . . 203

19 Real-time clock (RTC) . . . 204
19.1 Features . . . 204
19.2 Architecture . . . 205
19.3 Description . . . 205
19.3.1 Reading RTC user registers . . . 207
19.3.2 Writing to RTC user registers . . . 207
19.4 Alarm fields . . . 207
19.5 Registers summary . . . 209

20 Reset . . . 210
20.1 Features . . . 210
20.2 Architecture . . . 211
20.2.1 Power-on reset . . . 212

Table of contents

20.2.2 Brownout reset . . . 212
20.2.3 WDT reset . . . 212
20.2.4 Software reset . . . 212
20.2.5 External reset . . . 213
20.2.6 Logic protection fault reset . . . 213
20.2.7 Hibernate wakeup reset . . . 213
20.3 Identifying reset sources . . . 213
20.4 Registers summary . . . 214

21 GPIO . . . 216
21.1 Features . . . 216
21.2 Architecture . . . 216
21.2.1 I/O cell architecture description . . . 218
21.2.2 High-speed I/O matrix (HSIOM) . . . 218
21.2.3 I/O state on power-up . . . 222
21.2.4 Behavior in low-power modes . . . 223
21.3 Functional description . . . 223
21.3.1 Digital input buffer . . . 223
21.3.2 Digital output buffer . . . 224
21.3.2.1 GPIO-OVT pins . . . 224
21.3.2.2 Slew rate control . . . 225
21.3.2.3 Drive strength . . . 225
21.3.2.4 Drive mode . . . 226
21.3.3 Firmware-controlled GPIO . . . 229
21.4 Interrupt . . . 229
21.4.1 Analog I/O . . . 231
21.5 Programmable digital Smart I/O . . . 231
21.5.1 Overview . . . 232
21.5.2 Block components . . . 232
21.5.2.1 Clock and reset . . . 232
21.5.2.2 Synchronizer . . . 235
21.5.2.3 Lookup table (LUT) . . . 236
21.5.2.4 Data unit (DU) . . . 238
21.6 Registers summary . . . 240

22 TCPWM . . . 243
22.1 Features . . . 243
22.2 Architecture . . . 244
22.2.1 Enabling and disabling counters in a TCPWM block . . . 244
22.2.2 Clocking . . . 245
22.2.2.1 Clock pre-scaling . . . 245
22.2.2.2 Count event . . . 245
22.2.3 Trigger inputs . . . 245

Table of contents

22.2.4 Synchronization of multiple counters . . . 249
22.2.5 Trigger outputs . . . 251
22.2.6 Internal events . . . 251
22.2.6.1 Underflow event . . . 251
22.2.6.2 Overflow event . . . 252
22.2.6.3 TC event . . . 252
22.2.6.4 cc0_match (cc1_match) event . . . 253
22.2.7 Interrupts . . . 254
22.2.8 Debug mode . . . 255
22.2.9 PWM outputs . . . 256
22.2.10 Power modes . . . 257
22.3 Operation modes . . . 257
22.3.1 Timer mode . . . 258
22.3.1.1 Configuring counter for timer mode . . . 268
22.3.2 Capture mode . . . 268
22.3.2.1 Configuring counter for capture mode . . . 272
22.3.3 Pulse-width modulation (PWM) mode . . . 272
22.3.3.1 PWM mode functionalities . . . 280
22.3.3.2 Configuring counter for PWM mode . . . 291
22.3.4 Pulse-width modulation with Dead-Time (PWM_DT) mode . . . 291
22.3.4.1 Configuring counter for PWM with Dead-Time mode . . . 295
22.3.5 Dithering (Pseudo-random mode) . . . 296
22.3.6 Quadrature decoder control . . . 297
22.3.6.1 Quadrature clock and direction decoding . . . 302
22.3.6.2 Index control . . . 304
22.4 Recovery . . . 305
22.5 Initialize . . . 305
22.6 Pin status . . . 305
22.7 Registers summary . . . 306

23 Controller area network flexible data-rate (CAN FD) . . . 308
23.1 Features . . . 308
23.2 Architecture . . . 309
23.2.1 Dual clock sources . . . 309
23.2.2 Interrupt line . . . 309
23.3 Functional description . . . 310
23.3.1 Modes of operation . . . 310
23.3.1.1 Normal operation . . . 310
23.3.1.2 CAN FD operation . . . 310
23.3.1.2.1 Transmitter delay compensation . . . 311
23.3.1.3 Restricted operation mode . . . 313
23.3.1.4 Bus monitoring mode . . . 314
23.3.1.4.1 Disable automatic retransmission . . . 314

Table of contents

23.3.1.4.2 Power down (Sleep mode) . . . 315
23.3.1.5 Test mode . . . 315
23.3.1.5.1 External loopback mode . . . 316
23.3.1.5.2 Internal loopback mode . . . 316
23.3.2 Timestamp generation . . . 317
23.3.3 Timeout counter . . . 317
23.3.4 Receive handling . . . 318
23.3.4.1 Acceptance filtering . . . 318
23.3.4.2 Receive FIFOs . . . 321
23.3.4.2.1 Rx FIFO blocking mode . . . 322
23.3.4.2.2 Rx FIFO overwrite mode . . . 323
23.3.4.2.3 Rx FIFO top pointer . . . 323
23.3.4.3 Dedicated Rx buffers . . . 324
23.3.4.4 Debug on CAN support . . . 325
23.3.4.4.1 Filtering debug messages . . . 325
23.3.4.4.2 Debug message handling . . . 326
23.3.5 Transmit handling . . . 326
23.3.5.1 Transmit pause . . . 327
23.3.5.2 Dedicated Tx buffers . . . 327
23.3.5.2.1 Addressing dedicated Tx buffers . . . 327
23.3.5.3 Transmit FIFO . . . 328
23.3.5.4 TX queue . . . 329
23.3.5.5 Mixed dedicated Tx buffers/Tx FIFO . . . 329
23.3.5.6 Mixed dedicated Tx buffer/Tx queue . . . 330
23.3.5.7 Transmit cancellation . . . 330
23.3.5.8 Transmit event handling . . . 330
23.3.6 FIFO acknowledge handling . . . 331
23.3.7 Configuring CAN bit timing . . . 331
23.3.7.1 CAN bit timing . . . 331
23.3.7.2 CAN bit rates . . . 333
23.4 Message RAM . . . 335
23.4.1 Message RAM configuration . . . 335
23.4.2 Rx buffer and FIFO element . . . 336
23.4.3 Tx buffer element . . . 339
23.4.4 Tx event FIFO element . . . 342
23.4.5 Standard message ID filter element . . . 343
23.4.6 Extended message ID filter element . . . 345
23.4.7 Trigger memory element . . . 347
23.4.8 RAM watchdog (RWD) . . . 353
23.5 Time-triggered CAN (TTCAN) . . . 353
23.5.1 Reference message . . . 353
23.5.2 TTCAN configuration . . . 355

Table of contents

23.5.2.1 TTCAN timing . . . 356
23.5.2.1.1 Message scheduling . . . 356
23.5.2.2 Trigger memory . . . 357
23.5.2.2.1 Trigger types . . . 358
23.5.2.2.2 Restrictions for the node's trigger list . . . 359
23.5.2.2.3 Examples of trigger handling . . . 360
23.5.2.2.4 Detection of configuration errors . . . 361
23.5.2.3 M_TTCAN schedule initialization . . . 362
23.5.2.3.1 Time slaves . . . 362
23.5.2.3.2 Potential time master . . . 362
23.5.3 TTCAN gap control . . . 363
23.5.4 Stop watch . . . 364
23.5.5 Local/cycle/global time and external clock synchronization . . . 364
23.5.6 TTCAN error level . . . 367
23.5.7 TTCAN message handling . . . 369
23.5.7.1 Reference message . . . 369
23.5.7.2 Message reception . . . 369
23.5.7.3 Message transmission . . . 370
23.5.7.3.1 Transmission in exclusive time windows . . . 370
23.5.7.3.2 Transmission in arbitrating time windows . . . 371
23.5.7.3.3 Transmission in merged arbitrating time windows . . . 371
23.5.8 TTCAN interrupt and error handling . . . 371
23.5.9 Level 0 . . . 372
23.5.9.1 Synchronizing . . . 372
23.5.9.2 Handling error levels . . . 373
23.5.9.3 Master-slave relation . . . 373
23.5.10 Synchronization to external time schedule . . . 374
23.6 Setup procedures . . . 375
23.6.1 General program flow . . . 375
23.6.2 Clock stop request . . . 377
23.6.3 MRAM OFF operation . . . 377
23.6.4 MRAM ON operation . . . 378
23.6.5 Procedures specific to M_TTCAN channel . . . 378
23.6.6 CAN bus configuration . . . 380
23.6.7 Message RAM configuration . . . 380
23.6.8 Interrupt configuration . . . 385
23.6.9 Transmit frame configuration . . . 386
23.6.10 Interrupt handling . . . 386
23.7 Registers . . . 394
23.7.1 Register map . . . 394
23.7.1.1 Access to reserved register addresses . . . 398
23.7.2 Customer Register . . . 398

Table of contents

23.7.3 Core Release Register (CREL) . . . 399
23.7.4 Endian Register (ENDN) . . . 400
23.7.5 Data Bit Timing and Prescaler Register (DBTP) . . . 400
23.7.6 Test Register (TEST) . . . 401
23.7.7 RAM watchdog (RWD) . . . 403
23.7.8 CC Control Register (CCCR) . . . 404
23.7.9 Nominal Bit Timing and Prescaler Register (NBTP) . . . 406
23.7.10 Timestamp Counter Configuration (TSCC) . . . 407
23.7.11 Timestamp Counter Value (TSCV) . . . 408
23.7.12 Timeout Counter Configuration (TOCC) . . . 408
23.7.13 Timeout Counter Value (TOCV) . . . 409
23.7.14 Error Counter Register (ECR) . . . 410
23.7.15 Protocol Status Register (PSR) . . . 411
23.7.16 Transmitter Delay Compensation Register (TDCR) . . . 413
23.7.17 Interrupt Register (IR) . . . 414
23.7.18 Interrupt Enable (IE) . . . 417
23.7.19 Interrupt Line Select (ILS) . . . 418
23.7.20 Interrupt Line Enable (ILE) . . . 419
23.7.21 Global Filter Configuration (GFC) . . . 419
23.7.22 Standard ID Filter Configuration (SIDFC) . . . 420
23.7.23 Extended ID Filter Configuration (XIDFC) . . . 421
23.7.24 Extended ID AND Mask (XIDAM) . . . 422
23.7.25 High-Priority Message Status (HPMS) . . . 422
23.7.26 New Data 1 (NDAT1) . . . 423
23.7.27 New Data 2 (NDAT2) . . . 423
23.7.28 Rx FIFO 0 Configuration (RXF0C) . . . 424
23.7.29 Rx FIFO 0 Status (RXF0S) . . . 425
23.7.30 Rx FIFO 0 Acknowledge (RXF0A) . . . 426
23.7.31 Rx Buffer Configuration (RXBC) . . . 426
23.7.32 Rx FIFO 1 Configuration (RXF1C) . . . 427
23.7.33 Rx FIFO 1 Status (RXF1S) . . . 428
23.7.34 Rx FIFO 1 Acknowledge (RXF1A) . . . 429
23.7.35 Rx Buffer/FIFO Element Size Configuration (RXESC) . . . 429
23.7.36 Tx Buffer Configuration (TXBC) . . . 430
23.7.37 Tx FIFO/Queue Status (TXFQS) . . . 431
23.7.38 Tx Buffer Element Size Configuration (TXESC) . . . 432
23.7.39 Tx Buffer Request Pending (TXBRP) . . . 433
23.7.40 Tx Buffer Add Request (TXBAR) . . . 434
23.7.41 Tx Buffer Cancellation Request (TXBCR) . . . 434
23.7.42 Tx Buffer Transmission Occurred (TXBTO) . . . 435
23.7.43 Tx Buffer Cancellation Finished (TXBCF) . . . 435
23.7.44 Tx Buffer Transmission Interrupt Enable (TXBTIE) . . . 436

Table of contents

23.7.45 Tx Buffer Cancellation Finished Interrupt Enable (TXBCIE) . . . 436
23.7.46 Tx Event FIFO Configuration (TXEFC) . . . 437
23.7.47 Tx Event FIFO Status (TXEFS) . . . 437
23.7.48 Tx Event FIFO Acknowledge (TXEFA) . . . 438
23.7.49 TT Trigger Memory Configuration (TTTMC) . . . 439
23.7.50 TT Reference Message Configuration (TTRMC) . . . 439
23.7.51 TT Operation Configuration (TTOCF) . . . 440
23.7.52 TT Matrix Limits (TTMLM) . . . 441
23.7.53 TUR Configuration (TURCF) . . . 442
23.7.54 TT Operation Control (TTOCN) . . . 443
23.7.55 TT Global Time Preset (TTGTP) . . . 445
23.7.56 TT Time Mark (TTTMK) . . . 445
23.7.57 TT Interrupt Register (TTIR) . . . 446
23.7.58 TT Interrupt Enable (TTIE) . . . 448
23.7.59 TT Interrupt Line Select (TTILS) . . . 449
23.7.60 TT Operation Status (TTOST) . . . 451
23.7.61 TUR Numerator Actual (TURNA) . . . 452
23.7.62 TT Local and Global Time (TTLGT) . . . 453
23.7.63 TT Cycle Time and Count (TTCTC) . . . 454
23.7.64 TT Capture Time (TTCPT) . . . 454
23.7.65 TT Cycle Sync Mark (TTCSM) . . . 455

24 Serial communications block (SCB) . . . 456
24.1 Features . . . 456
24.2 Architecture . . . 457
24.2.1 Trigger interface . . . 458
24.2.2 Clock and reset interface . . . 458
24.2.2.1 Block enable . . . 459
24.2.2.2 Interrupt interface . . . 459
24.2.3 Operational modes . . . 462
24.2.3.1 Buffer modes . . . 462
24.2.3.2 FIFO mode . . . 462
24.2.3.3 EZ mode . . . 462
24.2.3.4 CMD RESP mode . . . 463
24.2.3.5 Clocking mode . . . 463
24.3 Serial peripheral interface (SPI) . . . 464
24.3.1 Features . . . 464
24.3.2 General description . . . 465
24.3.3 SPI modes of operation . . . 467
24.3.3.1 Motorola SPI . . . 467
24.3.3.2 Texas Instruments SPI . . . 469
24.3.3.3 National Semiconductors SPI . . . 471
24.3.4 SPI buffer modes . . . 472

Table of contents

24.3.4.1 FIFO mode . . . 473
24.3.4.2 EZSPI mode . . . 474
24.3.4.3 Command-response mode . . . 477
24.3.5 Clocking and oversampling . . . 480
24.3.5.1 Clock modes . . . 480
24.3.5.2 Using SPI master to clock slave . . . 482
24.3.5.3 Oversampling and bit-rate . . . 482
24.3.6 SPI master SELECT output timing control . . . 484
24.3.7 SPI parity functionality . . . 486
24.3.8 Loopback . . . 486
24.3.9 Enabling and initializing SPI . . . 486
24.3.10 I/O pad connection . . . 487
24.3.10.1 SPI master . . . 487
24.3.10.2 SPI slave . . . 488
24.3.11 Registers summary . . . 490
24.4 UART . . . 490
24.4.1 Features . . . 491
24.4.2 General description . . . 491
24.4.3 UART modes of operation . . . 492
24.4.3.1 Standard protocol . . . 492
24.4.3.2 SmartCard (ISO7816) . . . 498
24.4.3.3 Infrared data association (IrDA) . . . 499
24.4.3.4 Local interconnect network (LIN) mode . . . 500
24.4.4 Clocking and oversampling . . . 503
24.4.5 Loopback . . . 504
24.4.6 Enabling and initializing UART . . . 504
24.4.7 I/O pad connection . . . 504
24.4.7.1 Standard UART mode . . . 504
24.4.7.2 SmartCard mode . . . 505
24.4.7.3 IrDA mode . . . 506
24.4.7.4 LIN mode . . . 506
24.4.8 Registers summary . . . 507
24.5 Inter-integrated circuit (I^2C) . . . 507
24.5.1 Features . . . 507
24.5.2 General description . . . 508
24.5.3 Terms and definitions . . . 508
24.5.3.1 Clock stretching . . . 509
24.5.3.2 Bus arbitration . . . 509
24.5.4 I^2C modes of operation . . . 509
24.5.4.1 Write transfer . . . 510
24.5.4.2 Read transfer . . . 511
24.5.5 I^2C buffer modes . . . 511

Table of contents

24.5.5.1 FIFO mode . . . 511
24.5.5.1.1 Deep Sleep to Active mode transition in FIFO buffer mode . . . 512
24.5.5.2 EZI²C mode . . . 513
24.5.5.2.1 Deep Sleep to Active mode transition in EZ buffer mode . . . 514
24.5.5.3 Command-response mode . . . 515
24.5.6 Clocking and oversampling . . . 516
24.5.6.1 Glitch filtering . . . 518
24.5.6.2 Oversampling and bit rate . . . 519
24.5.7 Loopback . . . 520
24.5.8 Enabling and initializing I²C . . . 520
24.5.8.1 Configuring for I²C FIFO mode . . . 520
24.5.8.2 Configuring for EZ and CMD_RESP modes . . . 521
24.5.9 I/O pad connection . . . 521
24.5.10 Registers summary . . . 522
24.6 SCB interrupts . . . 522
24.6.1 SPI interrupts . . . 522
24.6.2 UART interrupts . . . 523
24.6.3 I²C interrupts . . . 524

25 PDM-PCM converter . . . 526
25.1 Overview . . . 526
25.2 Features . . . 526
25.3 Architecture . . . 526
25.3.1 Operating modes . . . 528
25.3.1.1 Mono mode . . . 528
25.3.1.2 Stereo mode . . . 529
25.3.2 Clocking . . . 530
25.3.2.1 Full rate sampling . . . 531
25.3.2.2 Halve-rate sampling . . . 531
25.3.3 Sampling delay . . . 531
25.3.4 Filter options . . . 533
25.3.4.1 Cascaded integrator comb (CIC) filter . . . 534
25.3.4.2 Finite impulse response (FIR) filters . . . 535
25.3.4.2.1 FIR0 . . . 535
25.3.4.2.2 FIR1 . . . 535
25.3.4.3 DC-blocking filter . . . 536
25.3.5 Rx FIFO . . . 537
25.3.6 Interrupts . . . 538

26 Serial memory interface (SMIF) . . . 540
26.1 Features . . . 540
26.2 Description . . . 540
26.2.1 Tx and Rx FIFOs . . . 542

Table of contents

26.2.1.1 Tx command FIFO . . . 542
26.2.1.2 Tx data FIFO . . . 543
26.2.1.3 Rx data FIFO . . . 544
26.2.2 MIMO mode . . . 544
26.2.3 XIP mode . . . 544
26.2.4 Cache . . . 545
26.2.5 Arbitration . . . 546
26.2.6 Deselect delay . . . 546
26.2.7 Cryptography . . . 546
26.3 Memory device signal interface . . . 547
26.3.1 Specifying memory devices . . . 548
26.3.2 Connecting SPI memory devices . . . 548
26.3.3 SPI data transfer . . . 556
26.3.4 Example of setting up SMIF . . . 557
26.4 Triggers . . . 559
26.5 Interrupts . . . 560
26.6 Sleep operation . . . 560
26.7 Performance . . . 560

27 Inter-IC sound bus (I^2S) . . . 562
27.1 Features . . . 562
27.2 Architecture . . . 562
27.3 Digital audio interface formats . . . 563
27.3.1 Standard I^2S format . . . 563
27.3.2 Left-justified (LJ) format . . . 565
27.3.3 Time-division-multiplexed (TDM) format . . . 565
27.3.4 Clock polarity and delay options . . . 566
27.4 Interfacing with audio codecs . . . 568
27.5 Clocking features . . . 569
27.6 FIFO buffer and DMA support . . . 570
27.6.1 I^2S Tx FIFO . . . 571
27.6.2 I^2S Rx FIFO . . . 572
27.7 Interrupt support . . . 573
27.8 Watchdog timer . . . 574
27.9 MCLK output function . . . 575
27.10 Registers summary . . . 576

28 Analog-to-digital converter/microphone . . . 578
28.1 Features . . . 578
28.2 System diagram . . . 578
28.3 Block diagram . . . 579
28.3.1 Functional description . . . 579
28.3.1.1 Operating modes . . . 579

Table of contents

28.3.1.2 Interfaces . . . 579
28.3.1.3 Interrupts and triggers . . . 579
28.3.1.4 Recovery . . . 580
28.3.1.5 Initialize . . . 580
28.3.1.6 Pin status . . . 580
29 Bluetooth® Low Energy subsystem (BLESS) . . . 581
29.1 Features . . . 581
29.2 Architecture . . . 582
29.2.1 Link layer controller . . . 583
29.2.1.1 Bluetooth® LE functional modes and configuration . . . 583
29.2.1.2 Firmware reset . . . 584
29.2.2 Clocks . . . 584
29.2.3 Power states . . . 585
29.2.4 Bluetooth® LE radio . . . 588
29.3 Infineon Host Bluetooth® Stack (BTSTACK) . . . 588
Glossary . . . 591
Revision history . . . 602
Trademarks . . . 603
Disclaimer . . . 604

1 Introduction

Infineon's AIROC™ CYW20829 is a high-performance, ultra-low-power, secure MCU + Bluetooth® Low Energy platform purpose-built for IoT applications. It integrates a high-performance microcontroller with Bluetooth® LE 5.4 connectivity, high-performance analog-to-digital conversion for audio input, I²S/PCM, CAN, and LIN for automotive use cases, along with other standard communication and timing peripherals.

AIROC™ CYW20829 provides a high level of integration to minimize external components, reducing device footprint and lowering the cost of implementing Bluetooth® LE solutions. It is an optimal solution for wireless input devices, remotes, keyboards, joysticks, Bluetooth® Mesh, automotive applications, asset tracking, and Bluetooth® LE IoT systems that require 10 dBm RF output power such as lighting and home automation.

1.1 Features

- **32-bit application core subsystem**:
  - 48/96 MHz Arm® Cortex®-M33 CPU with single-cycle multiply and memory protection unit (MPU)
  - Armv8-M architecture
  - CMOS 40 nm process
  - User-selectable core logic operation at either 1.1 V or 1.0 V
  - Active CPU current slope with 1.1 V core operation
  - Cortex®-M33: 40 µA/MHz
  - Active CPU current slope with 1.0 V core operation
  - Cortex®-M33: 22 µA/MHz
  - Datawire (DMA) controller with 16 channels
  - 32 KB cache for greater Execute in Place performance with lower power
- **Memory subsystem**:
  - 256 KB SRAM with power and data retention control
  - OTP eFuse array for security provisioning
- **Bluetooth® LE subsystem**:
  - 48 MHz Arm® Cortex®-M33 CPU with 2.4 GHz RF transceiver with 50 Ω antenna drive
  - Digital PHY
  - Link-layer engine supporting master and slave modes
  - Programmable Tx power up to 10 dBm
  - CYW20829 link-layer engine supports up to 16 simultaneous connections in any combination of central and peripheral devices. For example: 13 central devices and 3 peripheral devices, or 3 central devices and 13 peripheral devices
- **Low-power 1.7 V to 3.6 V operation**:
  - Six power modes for fine-grained power management
  - Deep Sleep mode current of 4.5 µA with 64 KB SRAM retention
  - On-chip DC-DC buck converter
- **Flexible clocking options**:
  - 8 MHz internal main oscillator (IMO) with ±2% accuracy
  - Ultra-low-power 32 kHz internal low-speed oscillator (ILO)
  - Two oscillators: high-frequency 24 MHz for radio PLL and low-frequency 32 kHz watch crystal for LPO
  - 48 MHz low-power internal high-speed oscillator (IHO)

  - Frequency-locked loop (FLL) for multiplying IMO frequency
  - Integer and fractional peripheral clock dividers
- **Quad SPI (QSPI)/serial memory interface (SMIF)**:
  - Execute in Place from external Quad SPI flash
  - On-the-fly encryption and decryption
  - Support for DDR
  - Supports single, dual, and quad interfaces with throughput up to 384 Mbps
- **Serial communication**:
  - Three run-time configurable Serial Communication Blocks (SCBs):
    - First SCB configurable as SPI or I^2C
    - Second SCB configurable as SPI or UART
    - Third SCB configurable as I^2C or UART
    - Only two instances of specific buses (SPI/I^2C/UART) are possible among three SCBs
- **Audio subsystem**:
  - Two pulse-density modulation (PDM) channels
  - One I^2S channel with time division multiplexed (TDM) mode
- **Timing and pulse-width modulation**:
  - Seven 16-bit and two 32-bit timer/counter pulse-width modulator (TCPWM) blocks for MCU functions. Multiple PWMs support color LEDs
  - PWM supports center-aligned, edge, and pseudo-random modes
- **ADC and MIC**: 12-bit sigma-delta switched-capacitor ADC for audio and DC measurements
- **Up to 32 programmable GPIOs**:
  - One I/O port (8 I/Os) enables Boolean operations on GPIO pins; available during system Deep Sleep
  - Programmable drive modes, strengths, and slew rates
  - Two overvoltage tolerant (OVT) pins
  - Up to six GPIOs used for SMIF
- **Security built into platform architecture**:
  - ROM-based Root of Trust via uninterruptible Secure Boot
  - Step-wise authentication of execution images
  - Secure execution of code in execute-only mode for protected routines
  - All debug and test ingress paths can be disabled
  - Up to four protection contexts (one available for customer code)
  - Secure debug support via authenticated debug token
  - Encrypted image support for external SMIF memory
- **Cryptography hardware**:
  - Hardware acceleration for symmetric cryptographic methods and hash functions
  - True random number generation (TRNG) function

1.2 Architecture

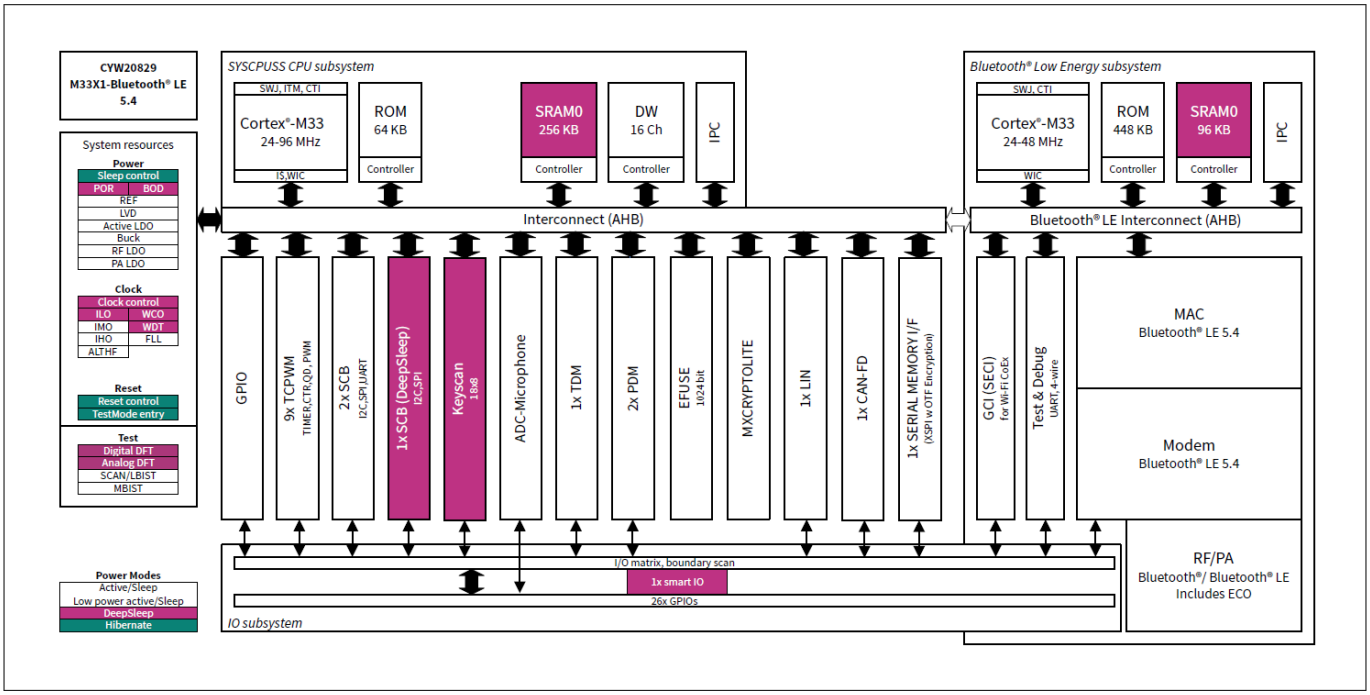


Figure 1 CYW20829 architecture block diagram

Figure 1 shows device subsystems and provides a simplified view of their interconnections. The color code indicates the lowest-power mode in which each block remains functional. For example, the watchdog timer is functional in Deep Sleep mode.

1.3 Document organization and conventions

1.3.1 Major sections

The information in this reference manual is organized as follows:

- **Chapter**: Presents chapters specific to an individual aspect of the device. These chapters provide detailed implementation and use information for each aspect of the integrated circuit
- **Registers summary**: Provides a summary of registers associated with the chapter

1.3.2 Documentation conventions

1.3.2.1 Numeric naming

Hexadecimal numbers appear in uppercase with an appended lowercase 'h' (for example, `14h` or `3Ah`). Hexadecimal numbers may also use the '0x' prefix, which follows the C coding convention. Binary numbers have an appended lowercase 'b' (for example, `01010100b` or `01000011b`). Numbers not marked with 'h' or 'b' are decimal.

1.3.2.2 Units of measure

Table 1 Units of measure used in this document

Symbol	Unit of measure
bps	bits per second
°C	degrees Celsius
dB	decibels
dBm	decibels-milliwatts
fF	femtofarads
G	Giga
Hz	Hertz
k	kilo, 1000
K	kilo, 2^10
KB	1024 bytes, or approximately one thousand bytes
Kbit	1024 bits
kHz	kilohertz (32.000)
kΩ	kilohms
MHz	megahertz
MΩ	megaohms
pA	microamperes
µF	microfarads
µs	microseconds
µV	microvolts
µVrms	microvolts root-mean-square
mA	milliamperes
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

(table continues...)

Table 1 (continued) Units of measure used in this document

Symbol	Unit of measure
V	volts

2 CPU

The CPU subsystem is based on the 32-bit Arm® Cortex®-M33 (CM33) CPU. It is designed to offer security, control, communication, short interrupt response time, high code density, and high 32-bit throughput, while adhering to strict cost and power-consumption budgets. This section provides an overview of the Arm® Cortex® CPU in CYW20829. For detailed information, see the Arm® documentation sets for Cortex®-M33.

2.1 Features

The CYW20829 Arm® Cortex®-M33 (CM33) CPU includes the following features:

- **Clock frequency**: Operates at 96 MHz in low-power mode or 48 MHz in ultra-low-power mode
- **Memory protection unit (MPU)**: Supports up to 8 regions per security state
- **Nested Vectored Interrupt Controller (NVIC)**: Integrated with the CPU core, providing low interrupt latency
- **Wakeup Interrupt Controller (WIC)**: Enables the CPU to wake up from Deep Sleep power mode using Deep Sleep interrupts
- **Instruction cache (I-cache)**: Includes a 32 KB instruction cache on the Cortex®-M33 C-AHB bus with four-way set associativity

2.2 Architecture

CM33 is a 32-bit processor with a 32-bit data path, 32-bit registers, and a 32-bit memory interface. It supports a wide range of instructions in the Thumb-2 instruction set. The CPU operates in two modes: thread mode and handler mode (see Operating modes and privilege levels).

The CM33 instruction set includes the following features:

- **Multiply and multiply-accumulate operations**: Supports signed and unsigned operations with 32×32 -> 32-bit and 32×32 -> 64-bit results, all executed in a single cycle
- **Division operations**: Supports signed and unsigned 32-bit divides, with execution times ranging from 2 to 11 cycles, depending on input values
- **Memory access**: Provides complex memory load and store access capabilities
- **Bit manipulation**: Includes support for complex bit manipulation operations

2.2.1 CPUSS

The CYW20829 CPUSS is a low-power, secure, control, and communication CPU built on the Cortex®-M33 (CM33) core, operating at a clock frequency of up to 96 MHz.

The CM33 CPU features a 3-stage pipeline (3PIP) from Arm®, based on the Armv8-M architecture, and includes a low-power domain with rapid clock-switching capabilities. The CPUSS provides a secure and interruptible boot function, ensuring system integrity is verified after boot and that memory and peripheral access privileges are enforced.

Key features of the CPUSS include:

- **Instruction cache**: Includes a 32 KB, 4-way set associative instruction cache on the C-AHB bus
- **Memory support**:
  - Integrates an on-chip SRAM controller and SRAM memory
  - Provides access to the C-AHB bus for code execution directly from SRAM
  - Features a 128-bit AHB5 bus interface for instruction fetch and data access from internal SRAM, accessible via its aliased address

- **DMA controller**:
  - Incorporates a single DMA controller with 16 channels, supporting Arm® Q-channel functionality
  - These channels are primarily used to service the SCB and TCPWM peripherals
- **Inter-processor communication (IPC)**:
  - Includes one IPC hardware block for application-level communication and synchronization
  - A second IPC resides in the Bluetooth® LE subsystem (BLESS) to enable communication between the CPU and BLESS
- **Clock domain (clk_hf0)**:
  - Operates at up to 96 MHz at 1.1 V
  - Used by the majority of the CPUSS hardware, including the instruction cache, AHB bus infrastructure, DMA, and all on-chip memory controllers, running at a single synchronous frequency
- **Centralized fault reporting**:
  - Implements a centralized fault reporting infrastructure, requiring only a single fault interrupt handler
  - Captures faults such as MPC or PPC violations, peripheral-specific errors, memory controller errors (for example, ECC errors), and time-out errors

The active power domain in the CPUSS is initialized first by default following a power-on reset (POR). For more information, see Device power modes.

2.2.2 Address and memory maps

The CPU has a fixed address map. The default memory map of the CPU spans the range 0x00000000 to 0xFFFFFFFF. For details about the default memory map, see the memory model section of the Arm® Cortex® -M33 processor technical reference manual.

Table 2 Address map

Address range	Name	Use
0x0000 0000 – 0x1FFF FFFF	Code	Program code region. Includes the exception vector table, which starts at address 0.
0x2000 0000 – 0x3FFF FFFF	SRAM	Data region
0x4000 0000 – 0x5FFF FFFF	Peripheral	All peripheral registers. Code execution is not allowed in this region. Bit-band functionality is not supported in this region.
0x6000 0000 – 0x8FFF FFFF	External NVM	SMIF/Quad SPI, see serial memory interface (SMIF). Code execution is allowed from this region.
0xA000 0000 – 0xDFFF FFFF	External Device	Not used
0xE000 0000 – 0xE00F FFFF	Private Peripheral Bus	Provides access to peripheral registers within the CPU core
0xE010 0A000 – 0xFFFF FFFF	Device	Device-specific system registers

Table 3 Internal memory address map

Address range	Memory type	Size
0x0000 0000 – 0x0001 0000	ROM	64 KB
0x2000 0000 - 0x 2004 0000	SRAM	Up to 256 KB

2.2.2.1 Wait-state lookup tables

The wait-state lookup tables display the wait states for different memory modules.

Table 4 Memory wait state

Parameter	Symbol	Values			Note or test condition
		Min.	Typ.	Max.	
Memory wait states					
Wait states	M33_cache	-	-	0	-
Wait states from CM33	M33_SRAM	-	0	3	Max is 3 with ECC enabled

***Note**: The wait-state lookup tables apply to all power modes, including low power (LP) and ultra-low power (ULP) modes.*

2.3 Operating modes and privilege levels

The CM33 CPU features thread and handler operating modes and can function in either the Thumb or Debug operating state. Additionally, the processor can restrict or deny access to certain resources by executing code in either privileged or unprivileged mode.

See the Armv8-M Architecture Reference Manual for detailed information on the modes of operation and execution.

Operating modes: The processor operates in either thread mode or handler mode.

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

Operating states: The processor operates in either the Thumb state or Debug state:

- **Thumb state**: This is the state of normal execution, where the processor runs 16-bit and 32-bit halfword-aligned thumb instructions
- **Debug state**: This is the state where the processor operates in halting debug mode

Privilege levels: The processor operates at two privilege levels: Unprivileged and Privileged.

- **Unprivileged**:
  - The software has limited access to certain instructions. It cannot execute the following:
    - Move from special register to general-purpose register (MRS) and move from general-purpose register to special register (MSR) instructions
    - Change processor state/disable interrupts (CPSID) and change processor state/enable interrupts (CPSIE) instructions
  - It cannot access the system timer, NVIC, or the system control block and has restricted access to memory or peripherals
- **Privileged**:
  - The software has full access to all instructions and system resources

Privilege controls:

- In thread mode, the privilege level (privileged or unprivileged) is controlled by the CONTROL register
- In handler mode, software execution always operates at the privileged level

Only privileged software can write to the CONTROL register to modify the privilege level. Unprivileged software can use the SVC instruction to transfer control to privileged software.

Stack pointer usage:

- In handler mode, the Main Stack Pointer (MSP) is always used. The exception entry and return mechanisms automatically update the CONTROL register, which may change depending on whether the MSP or the Process Stack Pointer (PSP) is used
- In thread mode, use the MSR instruction to set the stack pointer bit in the CONTROL register. After modifying the stack pointer, execute an ISB instruction immediately after the MSR instruction. This ensures that subsequent instructions execute using the updated stack pointer

2.4 Instruction set

Instruction support in the CM33 CPU:

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

The CM33 CPU implements the following instruction sets from the Armv8-M architecture:

- All base instructions
- All instructions in the main extension

For more information about these instructions, see section 3 - Cortex®-M33 Instruction Set of the Arm® Cortex®-M33 devices generic user guide.

Custom Datapath Extension (CDE) instructions:

The processor also implements Custom Datapath Extension (CDE) instructions. The CDE introduces additional instructions in the coprocessor instruction space, including:

- Three classes of instructions that operate on the general-purpose register file

For detailed information about the CDE instructions implemented in the processor, see section 9 - Arm® custom instructions of the Arm® Cortex®-M33 processor reference manual.

2.5 Registers summary

The CM33 CPU has the following 32-bit core registers.

Table 5 CM33 core register set summary

Name	Type[1]	Required privilege[2]	Reset value	Description
R0-R12	RW	Either	UNKNOWN	**R0–R12**: These are general-purpose registers used for data operations.
MSP (R13)	RW	Either	[3]	The Stack Pointer (SP) is register R13. In Thread Mode, the CONTROL register specifies which stack pointer to use: the Main Stack Pointer (MSP) or the Process Stack Pointer (PSP).
PSP (R13)	RW	Either	UNKNOWN	
MSPLIM	RW	Privileged	0x00000000	The stack-limit registers define the extent to which the Main Stack Pointer (MSP) and Process Stack Pointer (PSP) can descend.
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 contains the current program address.
xPSR	RW	Either	[4]	The Program Status Register (PSR) combines the following components: • 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 Application Program Status Register (APSR) holds the current state of the condition flags resulting from previous instruction executions.
IPSR	R0	Privileged	0x00000000	The IPSR contains the except number of the current ISR
EPSR	R0	Privileged	[4]	The Execution Program Status Register (EPSR) contains 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.

(table continues...)

Table 5 (continued) CM33 core register set summary

Name	Type[1]	Required privilege[2]	Reset value	Description
PRIMASK	RW	Privileged	0x00000000	The PRIMASK register prevents the activation of exceptions with configurable priority.
BASEPRI	RW	Privileged	0x00000000	The BASEPRI register specifies the minimum priority level for exception processing.
FAULTMASK	RW	Privileged	0x00000000	The FAULTMASK register prevents the activation of all exceptions, except for the Non-Maskable Interrupt (NMI) and, optionally, the secure HardFault.
CONTROL	RW	Privileged	0x00000000	The CONTROL register manages the stack pointer in use and, optionally, the privilege level when the processor operates in Thread mode.

1) Describes the access type during program execution in the thread and 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.

For more information on the usage of these registers, see the Arm® Cortex®-M33 reference manual. Use the MSR and MRS instructions to access the PSR, PRIMASK, CONTROL, FAULTMASK, and BASEPRI registers.

The following tables detail how the PSR bits are assigned. For comprehensive information, see the Combined Program Status Register section of 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
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

(table continues...)

Table 7 (continued) CM33 IPSR bit assignments

Bit	Name	Usage
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, including processors, memory, peripherals, and accelerators. The bus infrastructure provides a standardized interface for data and control signals, facilitating seamless interaction and efficient data exchange among these components. It typically consists of different types of buses, such as system buses, interconnects, and peripheral buses, each serving specific purposes. For example, system buses and interconnects, such as the Advanced High-Performance Bus (AHB), enable high-bandwidth communication between processors, memory, and other critical components.

CYW20829 features two subsystems, SYSCPUSS and BTSS, which utilize an AHB5-based bus infrastructure to interconnect multiple bus masters with slaves. The bus interconnect incorporates bridges, upsizers, and downsizers to interface with other interconnects of varying bus widths. This design allows access from multiple masters to slaves and supports concurrent access, ensuring efficient operation even when several high-speed peripherals are active simultaneously. Figure 2 shows the structure of this bus infrastructure.

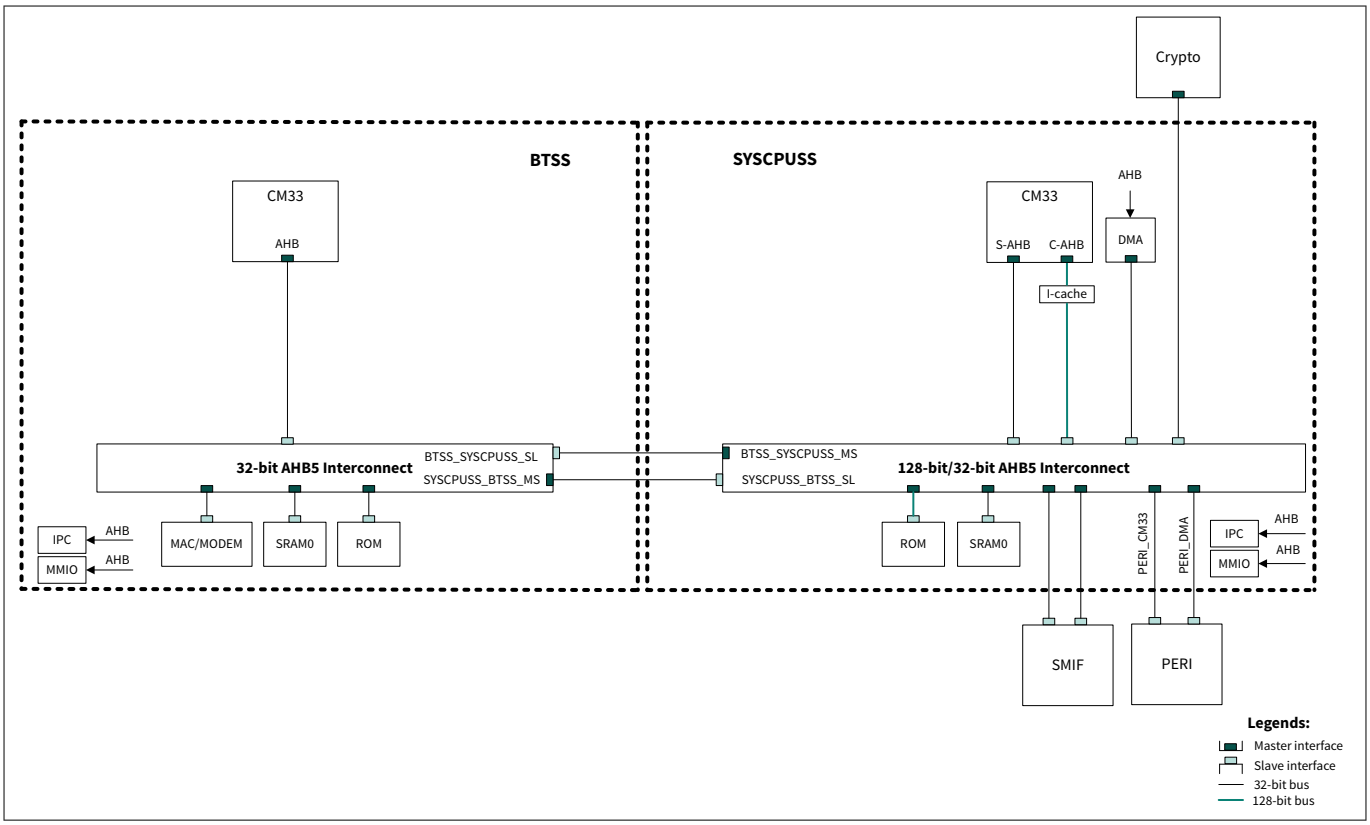


Figure 2 Bus infrastructure for CYW20829

3.1 SYSCPUSS AHB5 interconnect

The SYSCPUSS (CM33 CPU) utilizes a 128-bit and 32-bit multilayer AHB5 bus matrix interconnect. The AHB5 bus masters can reside either within the SYSCPUSS or outside of it (referred to as SYSCPUSS external bus masters). Similarly, the AHB5 bus slaves may be located internally within the SYSCPUSS (such as on-chip memory controllers) or externally (such as slaves connected to the peripheral interconnect (PERI) or to external slave interfaces).

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

- **Masters**:
  - Cortex®-M33 Code AHB (C-AHB) bus via the Instruction cache (I-cache)
  - Cortex®-M33 Slave AHB (S-AHB) bus

- One Direct Memory Access – DataWire
- One cryptographic accelerator (CryptoLite)
- BTSS_SYSCPUSS_MS: BTSS master interface from the BTSS AHB5 interconnect through the AHB5-to-AHB5 bridge

- **Slaves**:
  - One SRAM controller inside SYSCPUSS (SRAM0)
  - One ROM inside SYSCPUSS
  - SYSCPUSS_BTSS_SL: BTSS slave interface to the SYSCPUSS AHB5 interconnect through the AHB5-to-AHB5 bridge
  - CM33 to peripheral interconnect (PERI_CM33)
  - One AHB DMA to peripheral interconnect (PERI_DMA)
  - Serial Memory Interface port

3.1.1 AHB5 bus matrix interconnect

The SYSCPUSS uses a round-robin with latency scheme to implement bus arbitration in the AHB5 interconnect. Table 9 shows the SYSCPUSS AHB5 bus matrix interconnection.

Table 9 AHB5 bus matrix interconnection

Bus masters	Bus slaves					
	ROM	SYSCPUSS_BTSS_SL	SRAM	SMIF	PERI_CM33	PERI_DMA
CM33_C-AHB	**X**[1)]	-[2)]	x[3)]	x	-	
CM33_S-AHB	-	X[4)]	X	X	X	-
DMA	-	X	X	X	-	X
CryptoLite	-	X	X	X	-	X
BTSS_SYSCPUSS_MS	-	-	x	x	-	X

1) "**X**" indicates a connection, and bold font denotes a 128-bit bus.
2) "-" = No connection.
3) “x” indicates a connection and represents an aliased address path.
4) "X" = Connection.

***Notes**:*

1. *Memory aliasing allows the CM33 C-AHB bus (I-Cache) to access either the system SRAM or the external SMIF*
2. *All CM33 C-AHB bus accesses are cacheable*
3. *DMA writes to aliased addresses use the S-AHB bus address space (above 0x20000000)*
4. *Writes to aliased addresses from the DMA or the 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 AHB master (BTSS_SYSCPUSS_MS) and slave (SYSCPUSS_BTSS_SL) ports on the SYSCPUSS side provide a dedicated AHB interface with the BTSS*
6. *The PERI_CM33 and PERI_DMA interfaces provide access to peripherals connected via the peripheral interconnect (PERI)*

3.1.2 SYSCPUSS 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. It is used for instruction fetch and data access to internal and external memories mapped in the code region. The targets of this bus are the internal SRAM, ROM, and external SMIF.

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. It is used by the core to access data located in the peripheral or SRAM areas. The targets of this bus are the internal SRAM, external SMIF, and peripherals connected via PERI_CM33.

3.1.3 DMA bus

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

3.2 BTSS AHB5 interconnect

The BTSS (CM33 CPU) utilizes a 32-bit multilayer AHB5 bus matrix interconnect. The AHB5 bus masters may reside either within the BTSS or outside of it (referred to as BTSS external bus masters). Similarly, the AHB5 bus slaves may be located either inside the BTSS (for example, on-chip memory controllers) or outside of it (for example, slaves connected to the external slave interfaces).

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

- **Masters**:
  - Cortex®-M33 Slave AHB (S-AHB) Bus
  - BTSS_SYSCPUSS_SL: SYSCPUSS master interface from the BTSS AHB5 interconnect through an AHB5-to-AHB5 bridge
- **Slaves**:
  - One SRAM controller inside BTSS (SRAM0)
  - One ROM inside BTSS
  - SYSCPUSS_BTSS_MS: SYSCPUSS slave interface to the BTSS AHB5 interconnect through the AHB5-to-AHB5 bridge
  - MAC/MODEM

3.3 IPC

Inter-processor communication (IPC) via the AHB5 interface enables multi-processor communication and synchronization. This feature is available in both the SYSCPUSS and BTSS domains. For more information, see the Inter-processor communication chapter.

3.4 AHB5 MMIO

The memory-mapped I/O (MMIO) registers for CM33, CryptoLite, DMA, CPUSS, and other peripheral configurations are accessible via the AHB5 MMIO bus interface. This interface is available in both the SYSCPUSS and BTSS domains.

3.5 Peripheral interconnect

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 SYSCPUSS to interface with peripherals operating in both low-power and high-performance domains. These peripheral devices include all memory-mapped registers (MMIO registers) of PERI, PCLK, external peripheral devices, as well as synchronous and asynchronous peripherals mapped in the peripheral address space:

- **0x4200_0000 – 0x43FF_FFFF**: Non-secure address map
- **0x5200_0000 – 0x53FF_FFFF**: Secure address map

In CYW20829, there is one instance of PERI. As shown in Table 10, PERI consists of three peripheral groups that connect slave peripherals in the corresponding group to the SYSCPUSS and other AHB masters.

Table 10 PERI groups

Peripheral group	Slave peripheral	Peripheral type	Description
PERI GROUP 0	Peripheral interconnect MMIO registers	Internal	SYSCPUSS platform component without I/O subsystem
	Peripheral Protection Control (PPC)		
	PERI_PCLK registers		
	APB Slave group-0, SRAM0, ROM, MXCM33-0, DMA, SYSCPUSS, IPC, Fault structures MMIO registers		
	SRSS core	Peripheral	
	SRSS power mode		
	SRSS backup		
	Cryptographic accelerator (CryptoLite)		
	Coordinate rotation digital computer (CORDIC)		
	Debug registers		
PERI GROUP 1	High-Speed I/O Matrix (HSIOM)	Peripheral	Async peripherals
	GPIO		

(table continues...)

Table 10 (continued) PERI groups

Peripheral group	Slave peripheral	Peripheral type	Description
	Smart I/O		
	Control Area Network Flexible Data rate (CAN FD)		
	Local Interconnect Network(LIN)		
	Timer/Counter/PWM (TCPWM)		
	Serial Communications Block (SPI/UART/I2C) (SCB[0])		
	Serial Communications Block (SPI/UART/I2C) (SCB[1:2])		
PERI GROUP 2	DFT	Peripheral	-
	EFUSE		
	SMIF		High-speed interfaces
	Pulse Density Modulation(PDM)		
	Time Division Multiplexed(TDM)		
	Keyscan		-

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, individual slaves can be disabled using `SL_CTL[i].ENABLED[j]`.

Multiplexing of trigger signals is managed by the trigger component. For more information, see Trigger MUX.

3.5.1 AHB bus error handling

The following bus errors are possible in the bus infrastructure:

1. AHB error response detected at master interface
2. AHB error response from peripheral decoder:
   - The peripheral decoder generates an error response in any of the following scenarios:
     - Access to the respective slave is disabled or reset using `SL_*` registers (refer to the CYW20829 MCU registers reference manual for more details on these registers)
     - Access to an unmapped slave address
     - An error response from the respective peripheral itself
3. No response for the AHB transaction: Occurs when the peripheral fails to respond to an AHB transaction within the expected number of cycles

The PERI component uses the fault interface to report these errors. For more details, see Fault monitoring.

3.6 Bus master peripherals

3.6.1 Direct Memory Access(DMA-DW)

The DMA can function as an AHB master within the CPU subsystem. The DMA-DW is optimized for small data size and transactional DMA operations, typically used to transfer bytes between peripherals, such as from the ADC to RAM. The DMA-DW operates using a 32-bit AHB bus. For more details, see the Direct memory access chapter.

3.6.2 CryptoLite

CryptoLite provides an AHB-Lite master interface that supports 8-bit, 16-bit, and 32-bit AHB-Lite transfers. This interface enables the subsystem to access operand data for operations from system memories, such as SRAM. Memory buffer accesses can also be performed as 8-bit, 16-bit, or 32-bit transfers.

4 Inter-processor communication (IPC)

Inter-processor communication (IPC) provides functionality for applications in different protection contexts to communicate and synchronize their activities. IPC hardware is implemented using two register structures:

- **IPC channel**: This structure enables communication and synchronization between applications running in different protection contexts
- **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. This independence allows for the development of various interface models, as shown in Typical usage models.

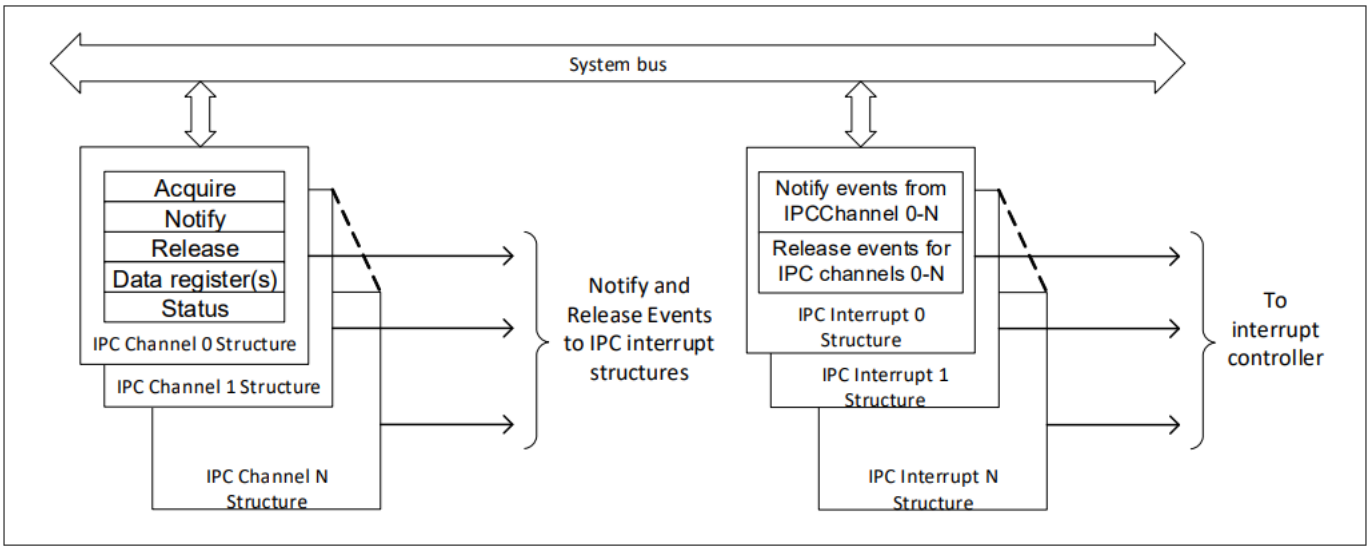


Figure 3 IPC register architecture

4.1 Features

The features of IPC are as follows:

- Implements locks for mutual exclusion between processors
- Allows sending messages between processors
- Supports up to 16 channels for communication
- Supports up to 16 interrupts, which are triggered using "notify" or "release" events from the channels
- CYW20829 supports 4 channel structures and 2 interrupt structures

4.2 Architecture

4.2.1 IPC channel

An IPC channel is implemented using five hardware registers, as shown in Figure 4. These IPC channel registers are accessible to all processors 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 1, the read operation successfully acquires the lock
  - If the SUCCESS field returns 0, the read operation fails to acquire the lock

A single read operation performs two atomic functions:

- Attempts to acquire the lock
- Returns the result of the acquisition attempt (SUCCESS field)

The atomicity of these functions is crucial in a CPU with multiple tasks that can preempt each other. The register also contains bit fields that provide information about the processor that acquired the lock. Once acquired, this register is released by writing any value into the **IPC_STRUCTx_RELEASE** register. If the register is already in the acquired state, subsequent read attempts will not succeed in acquiring it

- **IPC_STRUCTx_NOTIFY**: This register generates 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 one or multiple interrupt structures
- **IPC_STRUCTx_RELEASE**: Writing to this register releases the IPC channel. This register also includes bits corresponding to each IPC interrupt structure. A release event generated from an IPC channel can trigger one or multiple interrupt structures
  - To release the IPC channel without generating an interrupt, write `0` into the IPC release register
- **IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1**: These registers are designed to hold data and are both 32 bits in size. They can be considered as shared data memory for the channel. Typically, these registers hold messages that need to be communicated between processors
  - If the messages are larger than 32 bits, a pointer to the message can be placed in one or both of these registers
- **IPC_STRUCTx_LOCK_STATUS**: This register provides the instantaneous lock status of the IPC channel. It indicates whether the channel is acquired and, if so, provides details such as the processor's ID, protection context, and other information
  - Note that the lock status reflects only the instantaneous state, which could change in the next cycle depending on the activity of other processors on the channel

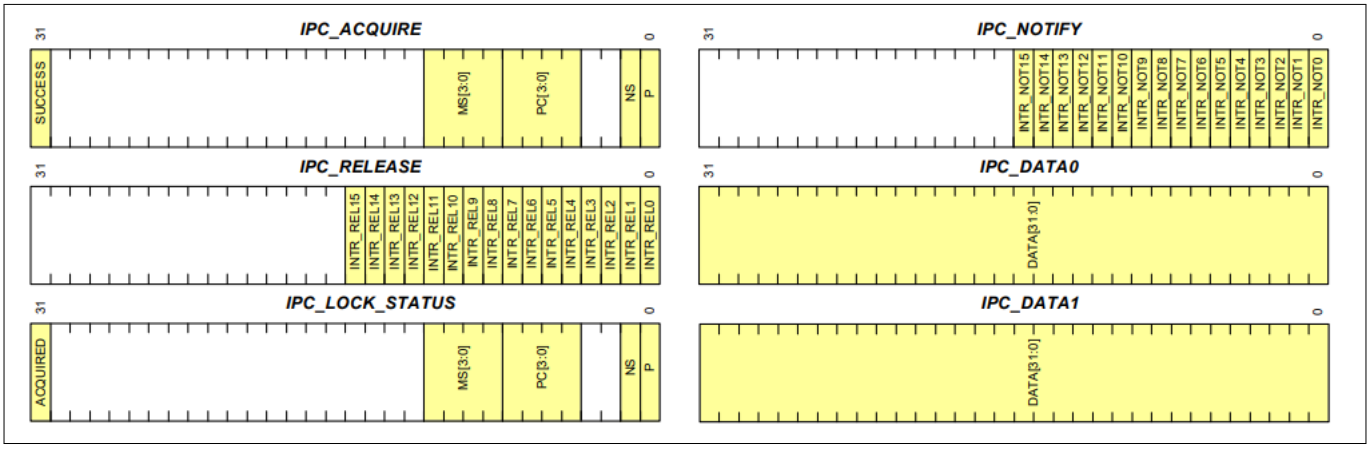


Figure 4 IPC channel structure

4.2.2 IPC interrupt

Each IPC interrupt line in the system is associated with a corresponding IPC interrupt structure. An IPC interrupt is triggered by either a "notify" or "release" event from any IPC channel in the system. It is possible to mask specific sources of these events using the IPC interrupt registers. Figure 5 illustrates the registers within an IPC interrupt structure.

- **IPC_INTR_STRUCTx_INTR**: This register provides the instantaneous status of the interrupt sources. It contains 16 notify bits and 16 release event bits, corresponding to the 16 IPC channels. For example, when a notify event is triggered in IPC channel 0, the corresponding `Notify0` bit is activated in the interrupt register. Writing `1` to a bit in this register clears the corresponding interrupt

- **IPC_INTR_STRUCTx_INTR_MASK**: This register is used to mask interrupt sources. Only the interrupt sources that have their mask bits enabled can trigger an interrupt
- **IPC_INTR_STRUCTx_INTR_SET**: Writing 1 into this register will set the corresponding interrupt
- **IPC_INTR_STRUCTx_INTR_MASKED**: This register provides the instantaneous status of interrupts after applying the mask. The value in this register is calculated as:

```
IPC_INTR_STRUCTx_INTR AND
    IPC_INTR_STRUCTx_INTR_MASK
```

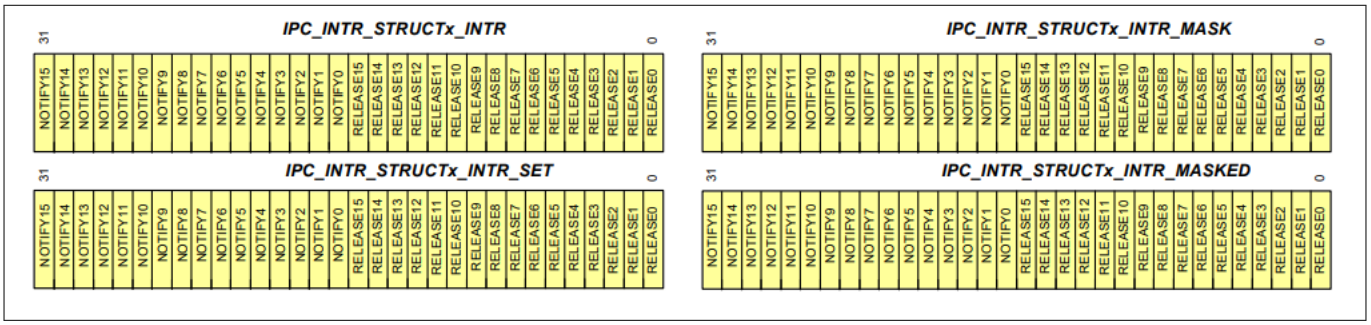


Figure 5 IPC interrupt structure

4.2.3 IPC channels and interrupts

The IPC block includes a set of IPC interrupts associated with it. Each IPC interrupt register structure corresponds to a specific IPC interrupt line. These interrupts can trigger an interrupt on any processor in the system. The routing of interrupts to processors depends on the device architecture.

Each IPC channel has a release register and a notify register, both of which can generate events that drive any of the IPC interrupts.



Figure 6 IPC channels and interrupts

4.3 Implementing locks

IPC channels can be used to implement locks, which are essential in multi-core systems to ensure mutually exclusive access to shared resources. When multiple processors share a resource, they can acquire and release an IPC channel. In this context, an IPC channel can be treated as a lock.

The semantics of this mechanism dictate that access to the shared resource is governed by the processor's ownership of the IPC channel. Processors must first acquire the IPC channel before accessing the shared resource. A failure to acquire the IPC channel indicates that the shared resource is locked because another processor has control of it.

It is important to note that the IPC channel itself does not enforce which processor acquires or releases the channel. Any processor in the system can acquire or release the IPC channel. The code semantics must ensure that the processor acquiring the channel is the one that releases it.

4.4 Message passing

IPC channels can be used to facilitate message communication between processors. In this use case, the channel operates in conjunction with the interrupt structures. The IPC channel is used to lock access to the data registers during the exchange.

The process begins with the sender acquiring the IPC channel and populating the message in the data registers. Once the message is written, the receiver reads the data and then releases the channel. During this period, the channel remains locked, preventing access from other tasks.

To indicate a send operation, the sender generates a notify event on the receiver's IPC interrupt. The receiver responds to this interrupt, reads the data from the data registers, and upon completion, releases the channel. The receiver can optionally trigger a release event on the sender's IPC interrupt to indicate that the operation is complete.

It is important to note that while the channel is "locked," the hardware does not enforce restrictions on access to the data registers. This locking mechanism and its semantics must be implemented and enforced by software.

1. **Sender's acquisition of the IPC channel**: The sender attempts to acquire the IPC channel by reading the `IPC_STRUCTx_ACQUIRE` register. If the channel is successfully acquired, the sender gains ownership of the channel for data transmission. This action also updates the channel's status in the `IPC_STRUCTx_LOCK_STATUS` register. If the channel is not acquired, the processor must wait until the channel becomes available. This can be achieved by polling the channel's `IPC_STRUCTx_LOCK_STATUS` register
2. **Data placement in the IPC channel**: Once the sender acquires the IPC channel, it gains control for communication and populates the data registers (`IPC_STRUCTx_DATA0` and `IPC_STRUCTx_DATA1`) with the message to be sent
3. **Sender notifies the receiver**: After placing the message in the IPC channel, the sender generates a notify event on the receiver's interrupt line. This is done by setting the corresponding bit in the channel's `IPC_STRUCTx_NOTIFY` register
   - This notify event triggers an interrupt at IPC interrupt B (`IPC_INTR_STRUCTx_INTR`)
   - If the notify event is enabled by setting the appropriate mask bit in the `IPC_INTR_STRUCTx_INTR_MASK` register (`[31:16]`), an interrupt is generated for the receiver
4. **Receiver processes the notify event**: Upon receiving IPC interrupt B, the receiver polls the `IPC_INTR_STRUCTx_INTR_MASKED` register to determine which IPC channel triggered the notify event. Based on this information, the receiver identifies the relevant channel and reads the message from the channel's `IPC_STRUCTx_DATA0` and `IPC_STRUCTx_DATA1` registers. At this point, the receiver has successfully received the data sent by the sender
5. **Receiver releases the IPC channel**: After reading the data, the receiver releases the channel, making it available for other processors or processes. Optionally, the receiver can generate a release event on the sender's IPC interrupt A
   - If the corresponding release event is enabled in the `IPC_INTR_STRUCTx_INTR_MASK` register, this will generate a release event interrupt for the sender

Upon receiving the release interrupt, the sender can respond to the event based on the application's requirements. The sender may either attempt to reacquire the channel for further data transmission or proceed to other tasks if the transmission is complete.

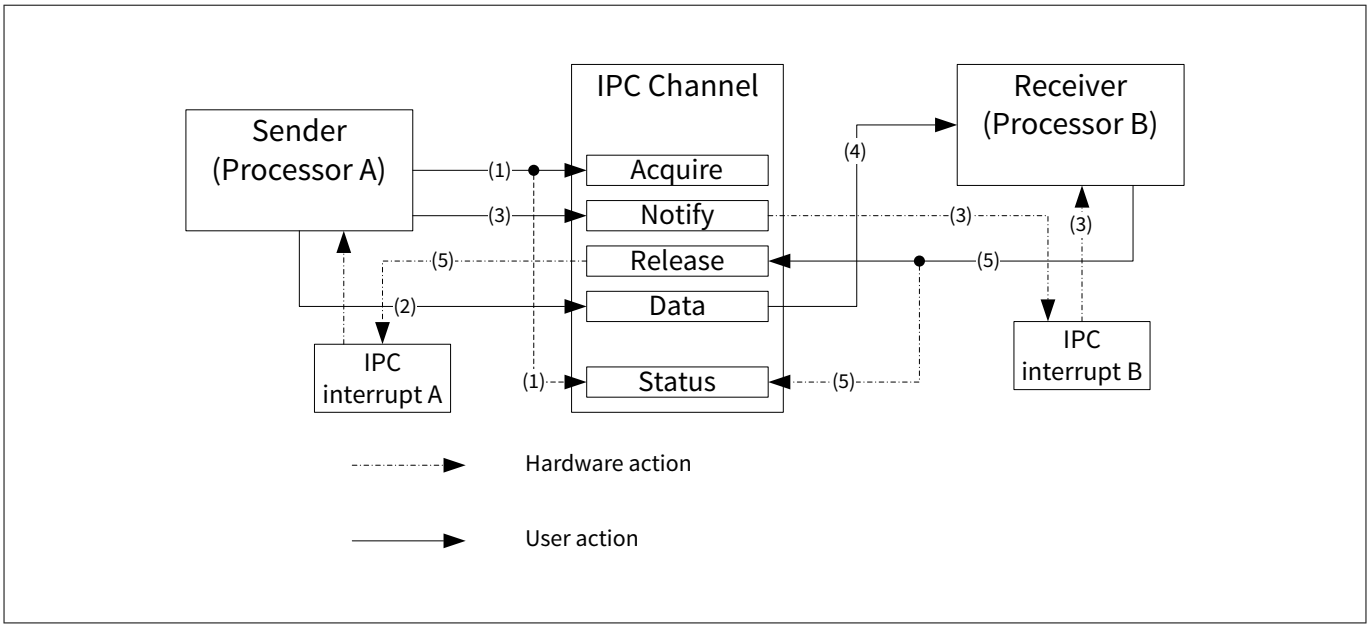


Figure 7 Sending messages using IPC

In the previous example, the size of the transmitted data was limited to 64 bits. For larger messages, the sender can transmit pointers instead of the actual data. The sender allocates a larger message structure in memory and places the pointers to this structure in the data registers (`IPC_STRUCTx_DATA0` and `IPC_STRUCTx_DATA1`). It is important to note that the synchronization of the message read process must be implemented in the user code.

One approach is to stall the IPC channel until the receiver processes all the data in the message packet, allowing the message packet to be rewritten afterward. However, this approach is inefficient, as it stalls the IPC channel and limits other inter-process communications, especially given the finite number of IPC channels available.

A more efficient approach is for the receiver to release the IPC channel immediately after receiving the pointer to the message packet. Synchronization logic is then implemented within the message structure itself. For example, the message can include a flag that the sender sets once the write operation is complete and the receiver clears after the read operation is finished. This ensures proper coordination between the sender and receiver without unnecessarily stalling the IPC channel.

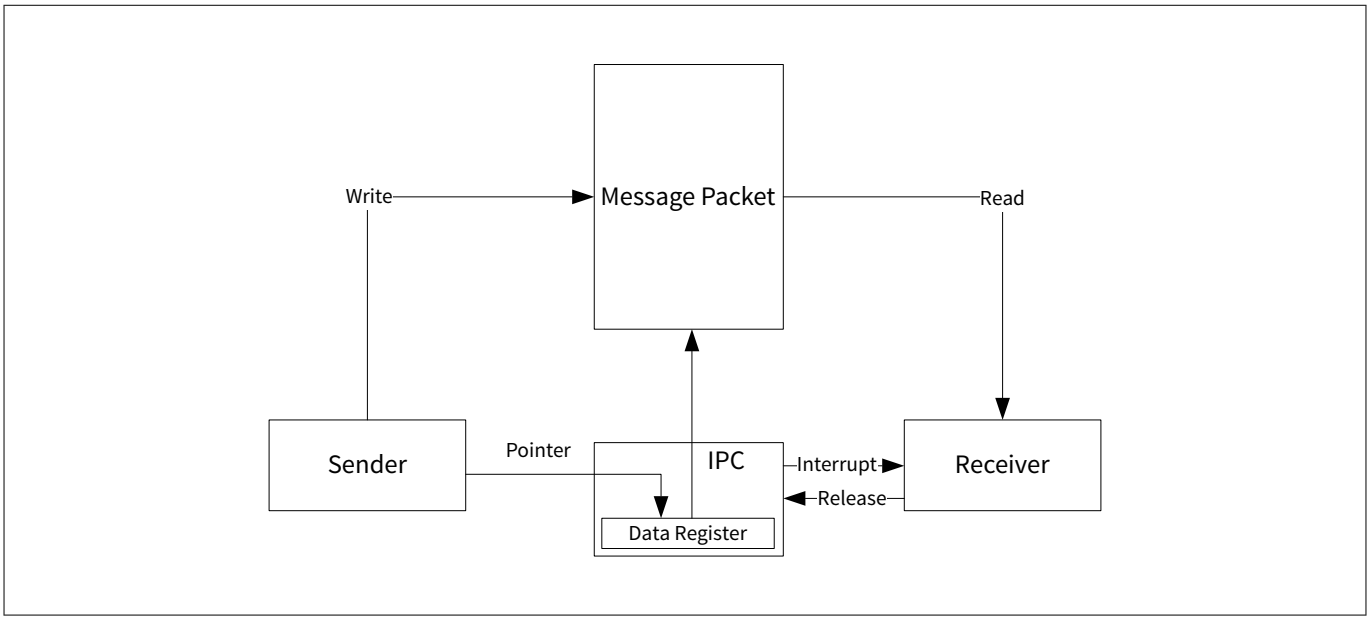


Figure 8 Communicating larger messages

4.5 Typical usage models

The unique channel-and-interrupt architecture of the CYW20829 IPC supports a variety of usage models for multi-core communication. Some examples of these models are listed below. It is important to note that the communication models enabled by the IPC architecture are not limited to those described in this document.

Additionally, this document provides only a high-level overview of usage models and does not delve into the specifics of data management in communication. Data management strategies must be determined based on the specific application use case.

4.5.1 Full duplex communication

In this usage model, an IPC channel is utilized based on the direction of communication between cores. To manage events, one IPC interrupt is assigned per core. In a dual-core system, this setup is shown in Figure 9.

In this example:

- **IPC channel X**: Is dedicated to data communication from Core 0 to Core 1
- **IPC channel Y**: Is dedicated to data communication from Core 1 to Core 0

The IPC interrupt X signals events on Core 1, and its interrupt output is connected to the system interrupts of Core 1. Events are triggered by writing into the IPC interrupt register structure over the system bus. Similarly, IPC interrupt Y is dedicated to Core 0 and signals events on Core 0 system interrupts.

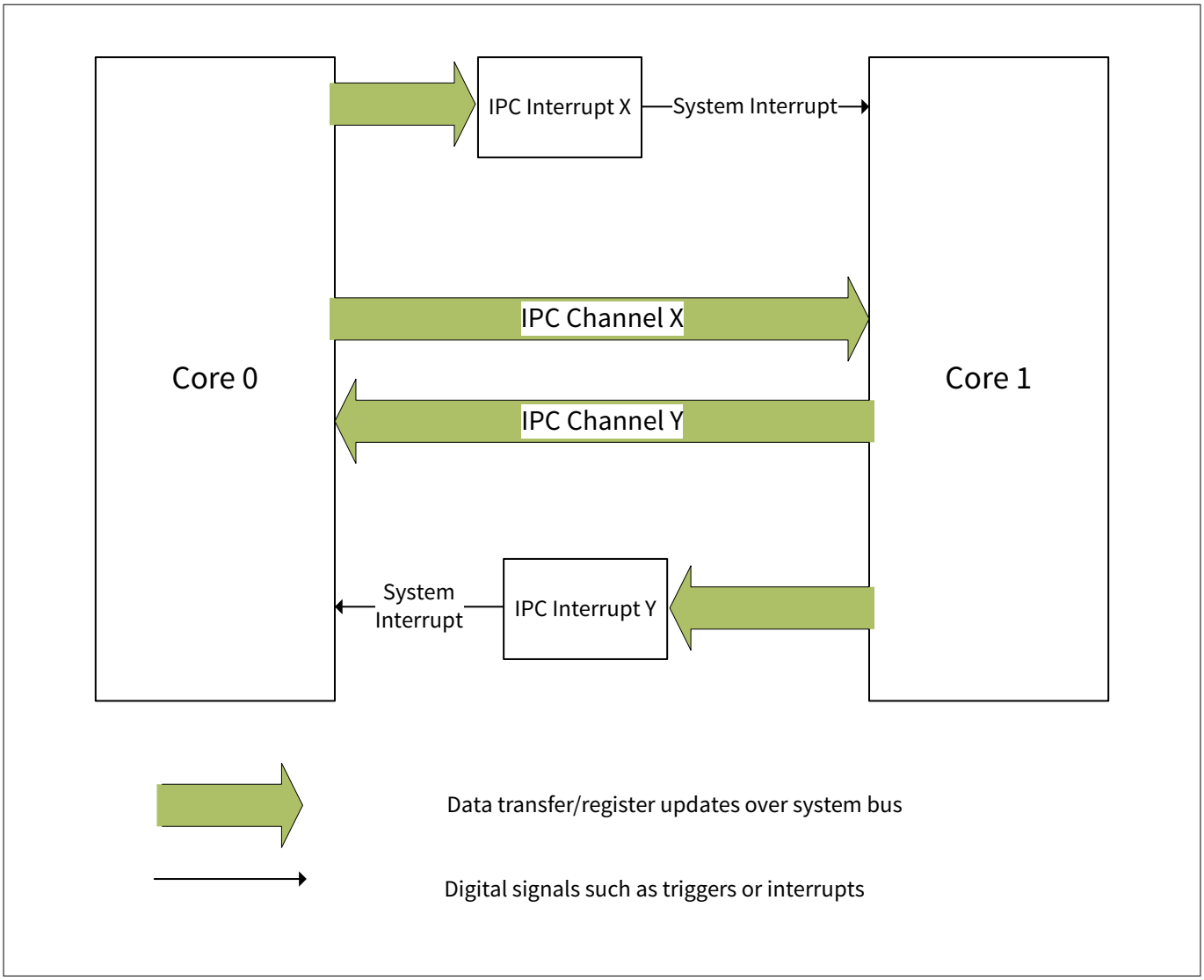


Figure 9 Full-duplex IPC for dual core communication

4.5.2 Half-duplex with independent event handling

In this case, only one IPC channel is used to manage data transfer between the two cores. This means that only one side controls the data transfer at a time. To prevent contention on the shared IPC channel, the channel's lock register must be used.

Despite using a single IPC channel, two independent events are supported through the use of the two IPC interrupt structures. This model is illustrated in Figure 10.

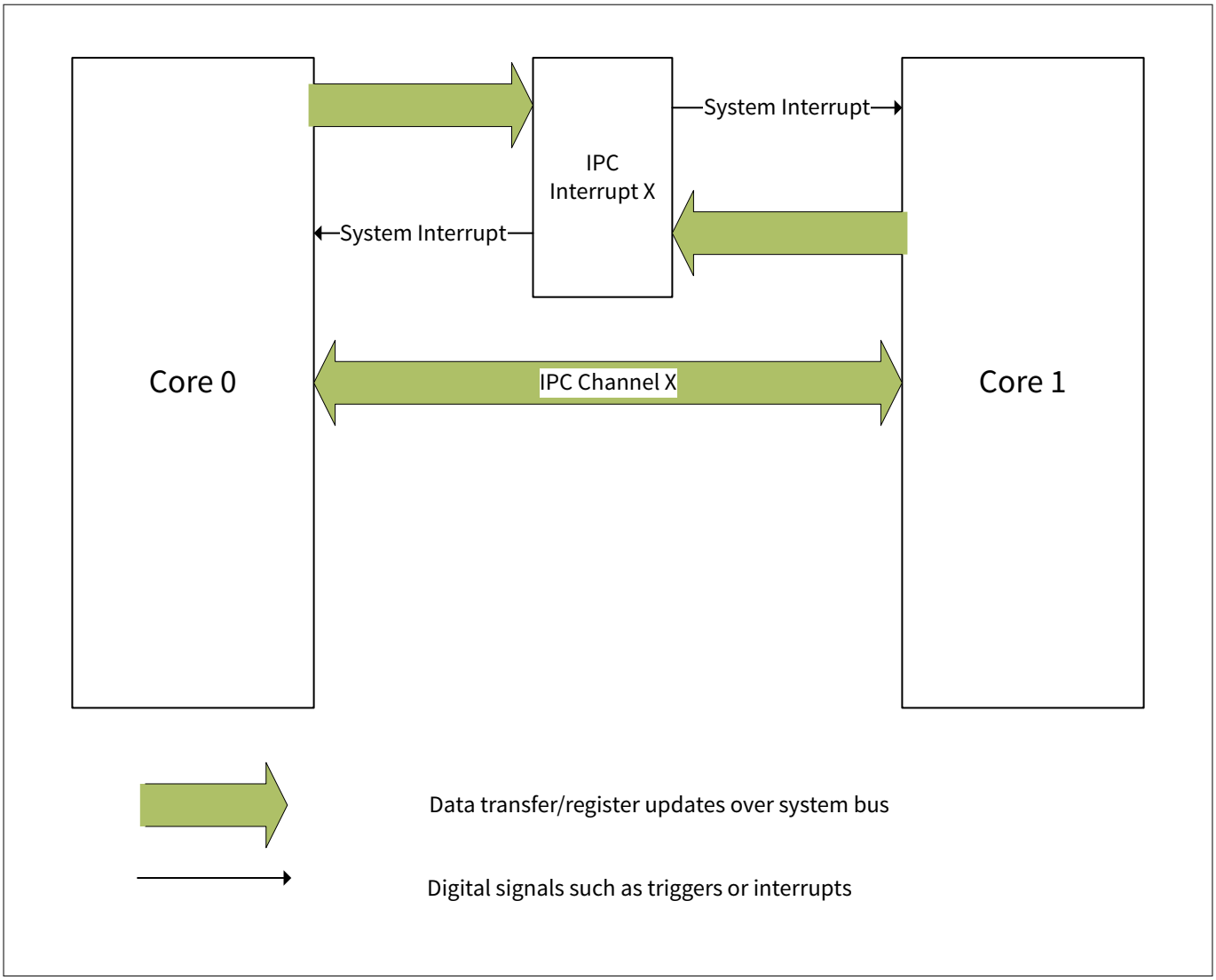


Figure 10 Half-duplex with shared event handling

4.5.3 Half-duplex with shared event handling

In this model, both the IPC channel and interrupt are shared between the two cores. Since the interrupt is also shared, access to the interrupt registers must be managed using the IPC channel's lock register.

As shown in Figure 11, the IPC interrupt is configured to trigger interrupts for both cores. Therefore, each core's interrupt service routine (ISR) must include logic to determine which core is in control of the IPC channel. The ISR should also verify whether the message and event are intended for that core.

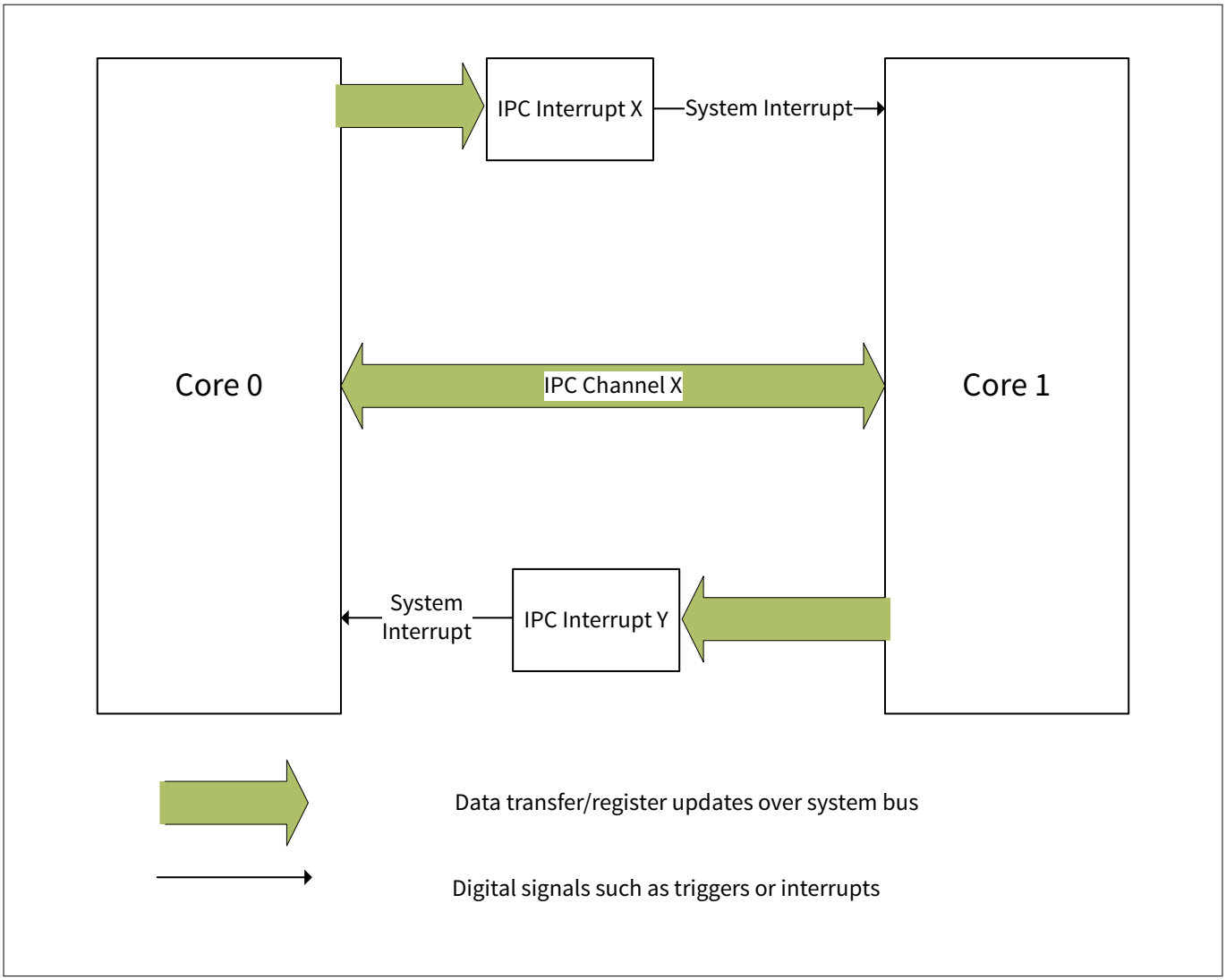


Figure 11 Half-duplex with independent event handling

5 SRAM

This section explains the CYW20829 MCU SRAM controller, including its features, architecture, and wait states. The SRAM controller enables the CPU to read and write portions of the 256 KB SRAM of the CYW20829 MCU.

In the CYW20829 MCU, the total SRAM size is 256 KB. For details about SRAM instances and their start addresses, see the Address and memory maps section of the CPU subsystem.

5.1 Features

The SRAM controller includes the following features:

- Supports 8-, 16-, and 32-bit accesses
- Provides exclusive access support using the Exclusive Access Monitor (EAM) from Arm® SIE-200
- Offers SRAM memory protection through the memory protection controller (MPC)
- Enables SRAM power mode control using the Arm® Power Policy Unit (PPU)

See the device datasheet for electrical specifications.

***Note**: The error correction code (ECC) functionality is disabled in the CYW20829 MCU.*

5.2 Architecture

The SRAM controller includes an AHB5 interface that connects to the AHB5 infrastructure. All SRAM accesses originate from AHB5 transfers. The SRAM controller categorizes AHB5 transfers into the following three types:

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

5.2.1 AHB5 read transfers

An AHB5 read transfer is translated directly into an SRAM read access.

5.2.2 32-bit AHB5 write transfers

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

5.2.3 Partial AHB5 write transfers

Partial writes, such as 8-bit and 16-bit transfers, are supported through a read-modify-write operation. The associated mask is applied during the write process.

5.2.4 Power partition control

The `PWR_MACRO_CTL` register controls the power state of each SRAM power partition independently. Each bit in `PWR_MACRO_CTL.OFF[31:0]` represents the power state of an individual SRAM power partition:

- Setting the bit corresponding to a particular partition turns OFF that partition
- Clearing the bit turns ON the respective partition

It is mandatory to check if `STATUS.PWR_DONE` is set to `1` after modifying the `PWR_MACRO_CTL` register to ensure that the SRAM mode transition completes successfully.

Each power partition contains `(RAM[ ]_MACRO_NR/RAM[ ]_PWR_GROUP_NR)` base SRAM units. In the CYW20829 MCU:

- RAMC0_MACRO_NR = 8
- RAMC0_PWR_GROUP_NR = 8

This configuration results in 8 power partitions, with each partition containing 1 macro.

The PWR_MACRO_CTL_LOCK register prevents accidental writes to the PWR_MACRO_CTL register by restricting read/write access. When the PWR_MACRO_CTL_LOCK[1:0] field is not equal to zero, read/write operations to the PWR_MACRO_CTL register are prohibited.

Note:
- ***Partitions that are OFF***: *A partition that is turned off will lose its data contents. This is also true for partitions in the following system power modes:*
  - *Deep Sleep-Off*
  - *Deep Sleep-RAM*
  - *Deep Sleep*
- ***Partitions that are ON***: *A partition that is turned on:*
  - *Is available for operation in Active power mode*
  - *Retains its data contents in the Deep Sleep-RAM and Deep Sleep system power modes*
  - *Loses its data contents in the Deep Sleep-Off system power mode*

5.3 Wait states

There are no wait states for operations when only AHB5 requests are present. Up to three wait states may occur during read operations if ECC is enabled.

6 eFuse

The eFuse memory includes a set of eFuse bits, which are non-volatile memory macros. Once an eFuse bit is programmed, or “blown”, its value becomes permanent and cannot be changed. Some of these eFuse bits store various unchanging device parameters, such as critical device factory-trim settings, device lifecycle stages, DAP security settings, and encryption keys.

6.1 Features

The CYW20829 eFuses include the following features:

- A total of 1024 eFuse bits
- The eFuse bits are programmed one at a time in a manufacturing environment and cannot be programmed in the field
- Multiple eFuses can be read at the bit or byte level through a PDL API function call or an SROM call. An unblown eFuse reads as logic 0, and a blown eFuse reads as logic 1. There are no hardware connections from eFuse bits to other parts of the device
- SROM system calls are available to read eFuses. The eFuses are only writable by Boot ROM and IFX RAM apps, and write access is limited. For detailed information on programming eFuses, see the CYW20829 programming specifications

6.2 Architecture

The CYW20829 eFuses can only be programmed in a manufacturing environment. VDDIO1 must be set to 2.5 V ±5%, and the device must be in a specific test mode, entered via an XRES key. For more information, see the CYW20829 MCU programming specifications. Table 11 shows the usage of the CYW20829 eFuse bytes.

Table 11 CYW20829 eFuse byte assignment

Offset	No. of bytes	Name	Description
0	4	BOOTROW	BOOTROW contains magic key and LCS
4	4	NUM_ZEROS	Number of zeros for asset hash, factory hash, secure hash, and protected firmware hash
8	4	ACCESS_RESTRICT	Access restrictions for Secure/Normal_No_Secure and Dead
12	4	FULL_TRIM_0	Full Trim 0 (Clock PILO Trim)
16	4	FULL_TRIM_1	Full Trim 1 (Power LVD Trim and BACKUP Trim)
20	4	FULL_TRIM_2	Full Trim 2 (Power Wakeup Trim, Power Reference Trim, and Clock ILO Trim)
24	4	BASIC_TRIM_0	Basic Trim 0 (Power Reference Trim)
28	4	BASIC_TRIM_1	Basic Trim 1 (Clock IMO Trim)
32	4	BASIC_TRIM_2	Basic Trim 2 (Clock IHO Trim)
36	4	BASIC_TRIM_3	Basic Trim 3 (Clock CCO Trim)

(table continues...)

Table 11 (continued) CYW20829 eFuse byte assignment

Offset	No. of bytes	Name	Description
40	4	BASIC_TRIM_4	Basic Trim 4 (Power BODOVP Trim and Power Switching Regulator Trim)
44	4	BASIC_TRIM_5	Basic Trim 5 (Clock CCO Trim, Power PMU Trim, and NUM_ZEROS 0)
48	4	WOUNDING	Wounding (CPUSS_RAM, LIN, CAN_FD), ICV key revocation, and NUM_ZEROS (1,2,3)
52	4	BLE_RADIO_TRIM_0	BLE Radio Trim 0 (32 bits)
56	4	BLE_RADIO_TRIM_1	BLE Radio Trim 1 (32 bits)
60	4	BLE_RADIO_TRIM_2	BLE Radio Trim 2 (8 bits), BLE, BLE access, HCI mode, BLESS key revocation
64	16	ODM_KEY_0_HASH	OEM Key 0 hash (16 bytes hash of OEM public key)
80	16	ODM_KEY_1_HASH	OEM Key 1 hash (16 bytes hash of OEM public key)
96	4	NV_CONTER	Anti-rollback (up to 32 updates)
100	4	OEM_CONFIG	OEM key revocation and SMIF configuration
104	4	NOT_USED_1	NOT_USED_1 (32-bit unused)
108	4	NOT_USED_2	NOT_USED_2 (32-bit unused)
112	4	DEVICE_ID	Device ID (16-bit Silicon ID and 12-bit family ID)
116	4	DIE_0	Die Lot and Die Wafer
120	4	DIE_1	Die X, Die Y, Die Sort, and Die Day
124	4	DIE_2	Die Month, Die Year, and Silicon Revision Id

7 Interrupts

CYW20829 supports interrupts and CPU exceptions on the Cortex®-M33 CPU. Any condition that halts the normal execution of instructions is treated as an exception by the CPU. Consequently, an interrupt request is also treated as an exception. However, in the context of this chapter, interrupts refer to events generated by peripherals external to the CPU, such as timers, serial communication blocks, and port pin signals. Exceptions, on the other hand, refer to events originating within the CPU, such as memory access faults and internal system timer events.

Both interrupts and exceptions cause the current program flow to stop, allowing the CPU to execute the corresponding exception handler or interrupt service routine (ISR).

For registers in this chapter that are referenced as Armv8-M registers, see the Armv8-M architecture reference manual.

7.1 Features

CYW20829 supports the following interrupt features:

- Supports 70 system interrupts
- Four interrupts can be mapped to the non-maskable interrupt (NMI) of the CPU
- All interrupt sources can wake the CPU from CPU Sleep power mode
- 17 interrupt sources are capable of waking the Cortex®-M33 from CPU Deep Sleep power mode
- Integrated nested vectored interrupt controller (NVIC) with the CPU core, providing low interrupt latency
- Wakeup interrupt controller (WIC) enables interrupt detection and CPU wakeup in CPU Deep Sleep power mode
- The vector table can be placed in SRAM memory
- Configurable priority levels for each interrupt, supporting eight levels on the Cortex®-M33
- Supports level-triggered interrupt signals

7.2 Architecture

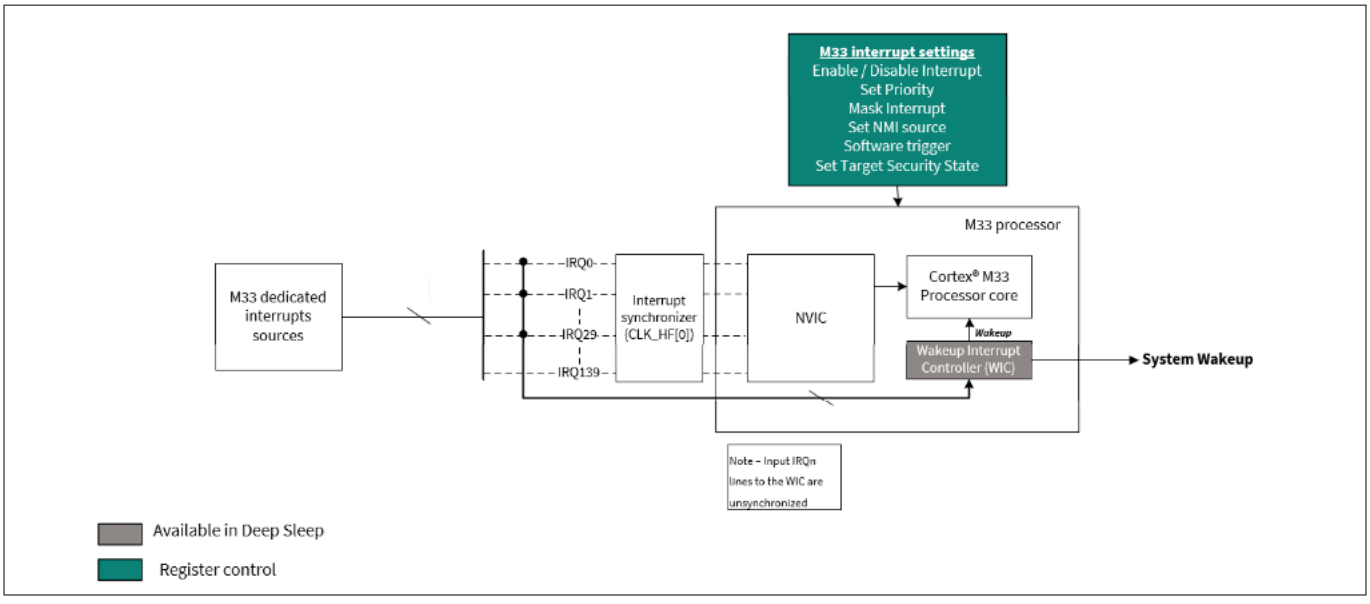


Figure 12 CYW20829 interrupt architecture block diagram

Figure 12 shows the CYW20829 interrupt architecture. The Cortex®-M33 connects system interrupt sources directly to the IRQn inputs of the nested vectored interrupt controller (NVIC). The NVIC handles enabling and

disabling individual interrupt IRQs, resolving interrupt priorities, and managing communication with the CPU core.

Exceptions such as non-maskable interrupts (NMI) and hard faults are not depicted in Figure 12, as they are generated internally by the CPU core and are distinct from interrupts, which are triggered by peripherals external to the CPU.

In addition to the NVIC, CYW20829 includes wakeup interrupt controllers (WIC) and interrupt synchronization blocks. The WIC detects interrupts while the CPU is in the Deep Sleep power mode. The interrupt synchronization blocks ensure that interrupts are synchronized to the CPU's CLK_HF domain, adding a delay of two CLK_HF cycles for synchronization.

7.3 Interrupts and exceptions - operation

7.3.1 Interrupt/exception handling

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

1. Initially, all interrupt and exception signals are assumed to be low (idle or inactive), and the processor is executing the main code. If an interrupt signal is asserted and the interrupt or exception is enabled for servicing by the CPU, the nested vectored interrupt controller (NVIC) registers the signal. The interrupt or exception is then placed in a pending state, waiting to be serviced by the CPU
2. Upon detecting the signal from the NVIC, the CPU saves its current context by pushing the contents of its registers onto the stack
3. The NVIC forwards the exception number of the triggered interrupt to the CPU. Each interrupt and exception has a unique exception number, as detailed in Table 12. Using this exception number, the CPU fetches the address of the corresponding exception handler from the vector table
4. The CPU branches to the fetched address and executes the exception handler associated with the interrupt or exception
5. After completing the exception handler, the CPU restores its registers to their original state by performing stack pop operations. Finally, the CPU resumes execution of the main code

When the NVIC receives an interrupt request while another interrupt is being serviced or when multiple interrupt requests arrive simultaneously, it evaluates the priority of all these interrupts. The NVIC sends the exception number of the highest-priority interrupt to the CPU. As a result, a higher-priority interrupt can preempt the execution of a lower-priority interrupt service routine (ISR) at any time.

Armv8-M CPUs support a feature called tail-chaining. This allows the CPU to optimize the handling of consecutive exceptions. When an exception handler completes, if a pending exception has a higher priority than all stacked exceptions, the CPU skips the stack pop operation and immediately transfers control to the new exception handler.

Exceptions are handled in the same way as interrupts. Each exception event is assigned a unique exception number, which the CPU uses to execute the appropriate exception handler.

7.3.2 Level interrupts

CYW20829 supports only level signals on the interrupt lines (IRQn). Pulse interrupts are not supported by this device.

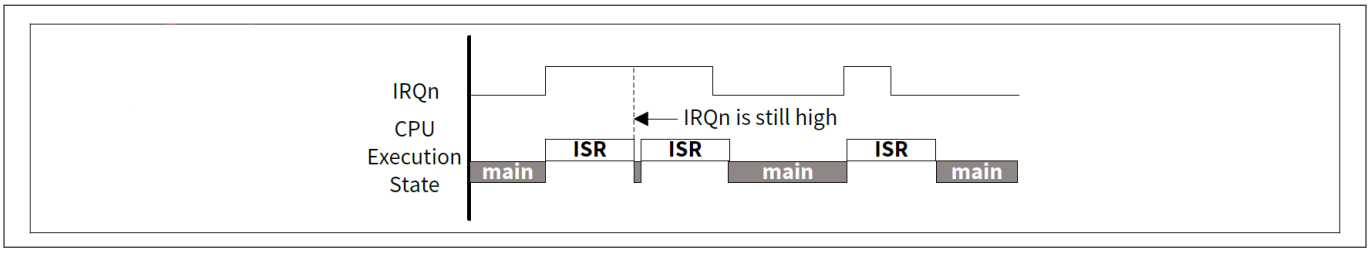


Figure 13 Level interrupts

Figure 13 shows the process of handling a level interrupt. Assuming the interrupt signal is initially inactive (logic low), the following sequence of events occurs:

For level-triggered interrupts, the interrupt signal remains asserted until the NVIC registers the interrupt request. Once registered, the interrupt enters a pending state, which indicates that the CPU has not yet serviced the interrupt request.

The NVIC sends the exception number along with the interrupt request signal to the CPU. As soon as the CPU begins executing the interrupt service routine (ISR), the pending state of the interrupt is cleared.

If the interrupt signal remains high after the ISR completes, the interrupt will again move to the pending state, and the ISR will be executed again. Figure 13 illustrates this behavior for level-triggered interrupts, where the ISR continues to execute as long as the interrupt signal remains high.

7.3.3 Exception vector table

Table 12 provides the exception vector table for the CYW20829, which stores the entry point addresses for all exception handlers in the Cortex®-M33 non-secure state. The CPU uses the exception number to fetch the corresponding address from the vector table.

Table 12 Cortex®-M33 non-secure exception vector

Exception number	Non-secure exceptions	Exception priority	Vector address
-	Initial stack pointer value	-	Start_Address = CM33_NS_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
7-10	Reserved	-	...
11	Supervisory Call (SVCall_NS)	Configurable (0-7)	Start_Address + 0x002C
12	Debug Monitor	Configurable (0-7)	Start_Address + 0x0030
13	Reserved	-	...

(table continues...)

Table 12 (continued) Cortex®-M33 non-secure exception vector

Exception number	Non-secure exceptions	Exception priority	Vector address
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
...	…	...	...
193	IRQ(177)	Configurable (0-7)	Start_Address + 0x0304
194	IRQ(178)	Configurable (0-7)	Start_Address + 0x0308

***Note**: The *start address is initially set to `CM33_NS_VECTOR_TABLE_BASE` on reset and can later be modified by firmware by updating the Cortex®-M33 Armv8-M `VTOR_NS` register.*

The first word (4 bytes) in the vector table is not labeled as exception number zero. This is because it is used to initialize the Main Stack Pointer (MSP) value during a device reset and is not treated as an exception.

The Vector Table Offset Register (VTOR), part of the Armv8-M System Control Block (SCB) registers, is used to configure 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 memory. However, note that the VTOR register can only be updated while the CPU is in privileged mode.

Placing the vector table in volatile memory, such as SRAM, allows for dynamic modification of exception handler addresses by updating the vector table contents. It is important to note that the vector table must be 1024-byte aligned for the Cortex®-M33.

- Exception sources (Exception numbers 1–15): These are explained in detail in the Exception sources section
- Interrupt sources (Exception numbers 16 and Higher): These are detailed in the Interrupt sources section
- Reserved exceptions: These exceptions are not used but have addresses reserved for them in the vector table

7.4 Exception sources

This section explains the different exception sources listed in tables 20 and 21, covering exception numbers 1 to 15.

7.4.1 Reset exception

Device reset is treated as an exception. The reset exception is always enabled and has a fixed priority of –4, which is the highest priority exception.

A device reset can occur for several reasons, including power-on reset (POR), an external reset signal on the XRES pin, or a watchdog reset. For CYW20829, after a device reset, the secure enclave (SE) initiates the secure boot process. Once the SE completes booting, it enables the Cortex®-M33, triggering the Cortex®-M33 secure reset exception. The Cortex®-M33 reset handler then starts the Cortex®-M33 boot code. After the Cortex®-M33 boot code executes successfully, the Cortex®-M33 transitions to the user application code.

7.4.2 Non-maskable interrupt exception

The non-maskable interrupt (NMI) is always enabled and has a fixed priority of –2. Each CPU has its own NMI exception.

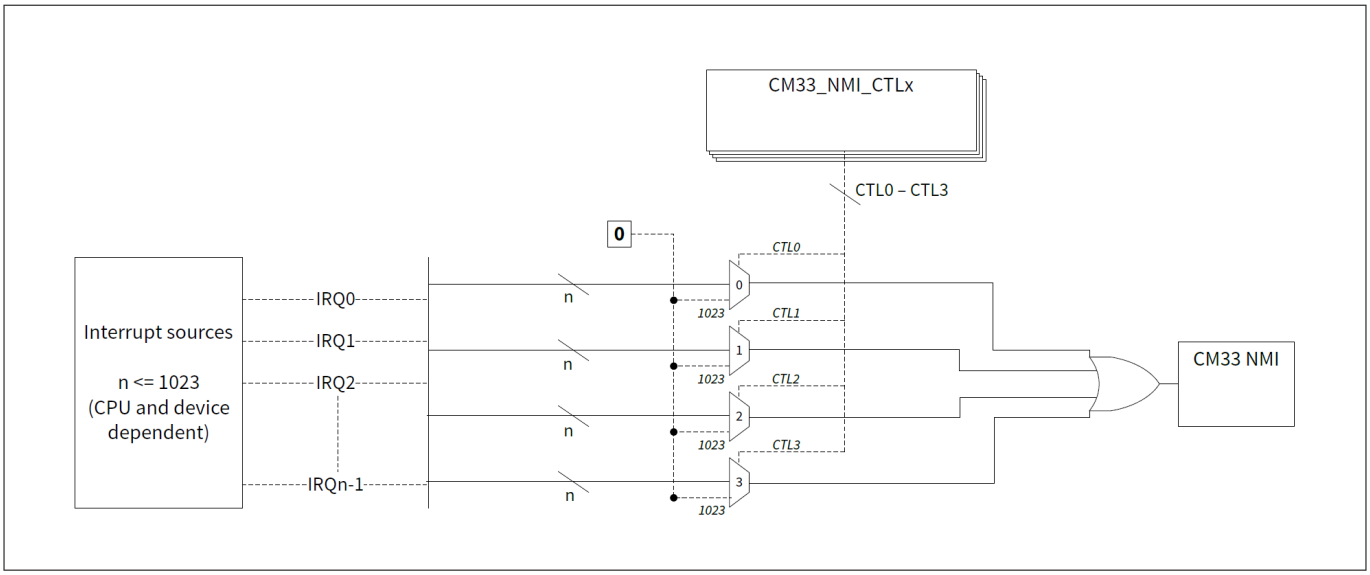


Figure 14 NMI trigger

The Cortex®-M33 provides an option to trigger an NMI exception from up to four interrupts accessible to the CPU (IRQ0 – IRQn). An NMI exception triggered by an interrupt executes the NMI handler, which is pointed to by the active exception vector table.

The `CM33_NMI_CTLx` registers are used to select the interrupt source that triggers the NMI from hardware. By default, these registers are set to a value of `1023`. When a register is set to `1023`, it does not map any interrupt source to the NMI. There are four such registers, and each can map a single interrupt vector to the NMI.

An NMI is triggered when any of the four interrupts are activated. In other words, the interrupts are logically ORed. See Figure 14 for more details.

7.4.3 HardFault exception

CYW20829 Armv8-M CPUs support the HardFault exception. HardFault is an always-enabled exception that is triggered by an error during normal or exception processing. It has a fixed priority of at least –1, meaning it has a higher priority than any exception with configurable priority.

HardFault serves as a catch-all exception for various fault conditions, such as executing an undefined instruction or accessing an invalid memory address. While the CPU does not provide fault status information to the HardFault exception handler, it allows the handler to perform an exception return. This enables the software to recover from the fault and continue execution when a fault condition is manageable.

7.4.4 Memory management fault exception

A MemManage fault is an exception triggered by a memory protection-related fault. This fault occurs due to fixed memory protection constraints for both instruction and data memory transactions. MemManage faults are always used to abort instruction accesses to Execute Never (XN) memory regions. The priority of this exception is configurable and can range from 0 (highest) to 7 (lowest).

7.4.5 Bus fault exception

A Bus fault is an exception triggered by a memory-related fault during an instruction or data memory transaction. This fault may occur due to an error detected on a bus in the memory system. The priority of the Bus fault exception is configurable and can range from 0 (highest) to 7 (lowest).

7.4.6 Usage fault exception

A usage fault is an exception triggered by a fault related to instruction execution. This exception occurs under the following conditions:

- Executing an undefined instruction
- Performing an illegal unaligned access
- Encountering an invalid state during instruction execution
- Experiencing an error on exception return

Additionally, when the core is configured to report them, the following conditions can also cause a usage fault:

- Using an unaligned address for word or half-word memory access
- Performing a division by zero

The priority of the usage fault exception is configurable and can range from 0 (highest) to 7 (lowest).

7.4.7 Supervisor Call (SVCall) exception

The CYW20829 Armv8-M CPUs support the Supervisor Call (SVCall) exception. SVCall is an always-enabled exception triggered when the CPU executes the SVC instruction within the application code. The SVC instruction is used by application software to request services from an underlying operating system. This process, known as a supervisor call, allows the application to issue an SVCall that requires privileged access to the system.

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

7.4.8 DebugMonitor exception

Armv8-M CPUs support the DebugMonitor exception. DebugMonitor is available only when halting debug is disabled and debug monitoring is enabled. To configure this, the C_DEBUGEN bit in the Armv8-M Debug Halting Control and Status Register (DHCSR) must be set to disabled, and the MON_EN bit in the Armv8-M Debug Exception and Monitor Control Register (DEMCR) must be set to enabled.

The DebugMonitor exception is handled by dedicated debug monitoring software running on the same core. A debug event triggers a DebugMonitor exception only if the priority of the DebugMonitor exception is higher than the priority of the currently active exception.

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

7.4.9 PendSupervisory (PendSV) exception

Armv8-M CPUs support the PendSV exception. PendSV is a supervisor call-related exception, similar to SVCall, but it is typically software-generated. PendSV is always enabled, and its priority is configurable, just like SVCall. The priority of the PendSV exception can range from 0 (highest) to 7 (lowest).

7.4.10 System Tick (SysTick) exception

The CYW20829 Armv8-M CPUs include a system timer, known as SysTick, as part of their internal architecture. SysTick is a simple 24-bit decrementing counter used for various timekeeping purposes, such as an RTOS tick timer, a high-speed alarm timer, or a basic counter. The SysTick timer can be configured to generate an interrupt when its count value reaches zero. This interrupt is referred to as a SysTick exception. The priority of the SysTick exception is configurable and can range from 0 (highest) to 7 (lowest).

7.5 Interrupt sources

CYW20829 supports 70 interrupts sourced from various peripherals, with details provided in Table 13. These system interrupts are directly mapped to the Cortex®-M33 NVIC.

The interrupts include standard ones generated by on-chip peripherals such as TCPWM, SCB, the watchdog timer, and others. Each interrupt is typically the logical OR of multiple peripheral states. To determine which specific condition triggered the interrupt, the peripheral interrupt status register must be read within the ISR.

These interrupts are generally level-sensitive interrupts. The appropriate interrupt registers should be cleared in the ISR to de-assert the interrupt. Typically, writing a '1' to the relevant register is required to clear it. If the interrupt register is not cleared in the ISR, the interrupt will remain asserted, causing the ISR to execute continuously. For more details on GPIO interrupts, refer to the I/O System chapter.

As shown in Table 13, 17 interrupts (IRQ0 to IRQ16) can wake the Cortex®-M33 from the CPU Deep Sleep power mode. Additionally, all interrupts are capable of waking the CPU from CPU Sleep power mode.

Table 13 List of CYW20829 Cortex®-M33 interrupt sources

System interrupt	Cortex®-M33 except number	Power mode	Interrupt source
NMI	2	Active	Any of the below 76 IRQ source
IRQ0	16	Deep Sleep	GPIO interrupt - Port 0
IRQ1	17	Deep Sleep	GPIO interrupt - Port 1
IRQ2	18	Deep Sleep	GPIO interrupt - Port 2
IRQ3	19	Deep Sleep	GPIO interrupt - Port 3
IRQ4	20	Deep Sleep	GPIO interrupt - Port 4
IRQ5	21	Deep Sleep	GPIO interrupt - Port 5
IRQ6	22	Deep Sleep	GPIO supply detect
IRQ7	23	Deep Sleep	GPIO - All ports
IRQ8	24	Deep Sleep	SCB#0 interrupt
IRQ9	25	Deep Sleep	MCWDT Interrupt
IRQ10	26	Deep Sleep	Back up domain Interrupt
IRQ11	27	Deep Sleep	SRSS interrupt (LVD)
IRQ12	28	Deep Sleep	CPUSS IPC interrupt #0
IRQ13	29	Deep Sleep	CPUSS IPC interrupt #1
IRQ14	30	Deep Sleep	mxkeyscan interrupt for keyscan edge or FIFO thresh to WIC in M33
IRQ15	31	Deep Sleep	Interrupt from WDT
IRQ16	32	Deep Sleep	Interrupt from BTSS IPC
IRQ17	33	Active	SCB #1 interrupt
IRQ18	34	Active	SCB #2 Interrupt

(table continues...)

Table 13 (continued) List of CYW20829 Cortex®-M33 interrupt sources

System interrupt	Cortex®-M33 except number	Power mode	Interrupt source
IRQ19	35	Active	M33SYSCPUSS DMA#0, Channel#0
IRQ20	36	Active	M33SYSCPUSS DMA#0, Channel#1
IRQ21	37	Active	M33SYSCPUSS DMA#0, Channel#2
IRQ22	38	Active	M33SYSCPUSS DMA#0, Channel#3
IRQ23	39	Active	M33SYSCPUSS DMA#0, Channel#4
IRQ24	40	Active	M33SYSCPUSS DMA#0, Channel#5
IRQ25	41	Active	M33SYSCPUSS DMA#0, Channel#6
IRQ26	42	Active	M33SYSCPUSS DMA#0, Channel#7
IRQ27	43	Active	M33SYSCPUSS DMA#0, Channel#8
IRQ28	44	Active	M33SYSCPUSS DMA#0, Channel#9
IRQ29	45	Active	M33SYSCPUSS DMA#0, Channel#10
IRQ30	46	Active	M33SYSCPUSS DMA#0, Channel#11
IRQ31	47	Active	M33SYSCPUSS DMA#0, Channel#12
IRQ32	48	Active	M33SYSCPUSS DMA#0, Channel#13
IRQ33	49	Active	M33SYSCPUSS DMA#0, Channel#14
IRQ34	50	Active	M33SYSCPUSS DMA#0, Channel#15
IRQ35	51	Active	PROMC Int
IRQ36	52	Active	PPU SRAM0
IRQ37	53	Active	MPC SRAM0

(table continues...)

Table 13 (continued) List of CYW20829 Cortex®-M33 interrupt sources

System interrupt	Cortex®-M33 except number	Power mode	Interrupt source
IRQ38	54	Active	CM33 0 Floating point interrupt
IRQ39	55	Active	CM33-0 CTI interrupt outputs
IRQ40	56	Active	CM33-1 CTI interrupt outputs
IRQ41	57	Active	Expansion bridge AHB error interrupt
IRQ42	58	Active	TCPWM #0, Counter #0
IRQ43	59	Active	TCPWM #0, Counter #1
IRQ44	60	Active	TCPWM #0, Counter #256
IRQ45	61	Active	TCPWM #0, Counter #257
IRQ46	62	Active	TCPWM #0, Counter #258
IRQ47	63	Active	TCPWM #0, Counter #259
IRQ48	64	Active	TCPWM #0, Counter #260
IRQ49	65	Active	TCPWM #0, Counter #261
IRQ50	66	Active	TCPWM #0, Counter #262
IRQ51	67	Active	SMIF interrupt
IRQ52	68	Active	SMIF interrupt
IRQ53	69	Active	TDM0 Audio interrupt RX
IRQ54	70	Active	TDM0 Audio interrupt TX
IRQ55	71	Active	PDM0/PCM0 Audio interrupt
IRQ56	72	Active	PDM0/PCM0 Audio interrupt
IRQ57	73	Active	SRSS Main PPU interrupt
IRQ58	74	Active	PERI PPC interrupt
IRQ59	75	Active	PERI AHB interrupt
IRQ60	76	Active	LIN interrupt, channel #0
IRQ61	77	Active	LIN interrupt, channel #1
IRQ62	78	Active	Crypto interrupt
IRQ63	79	Active	CPUSS PPU interrupt
IRQ64	80	Active	CAN #0, interrupt #0, Channel #0

(table continues...)

Table 13 (continued) List of CYW20829 Cortex®-M33 interrupt sources

System interrupt	Cortex®-M33 except number	Power mode	Interrupt source
IRQ65	81	Active	CAN #0, interrupt #1, Channel #0
IRQ66	82	Active	CAN #0, consolidated interrupt #0
IRQ67	83	Active	ADCMIC interrupt
IRQ68	84	Active	Interrupt indicating BTSS has encountered exception
IRQ69	85	Active	Crypto TRNG interrupt

7.6 Interrupt/exception priority

Exception priority is crucial for exception arbitration when multiple exceptions need to be serviced by the CPU. All exceptions, except Reset, NMI, and HardFault, can be assigned configurable priority levels. Reset and NMI have fixed priorities of –4 and –2, respectively, while HardFault for the Cortex®-M33 has a fixed priority of –1. Lower numerical priority values indicate higher priorities. Consequently, Reset, NMI, and HardFault exceptions have the highest priorities. Other exceptions can be assigned configurable priority levels ranging from 0 to 7.

Armv8-M CPUs support nested exceptions, allowing higher-priority exceptions to interrupt the currently active exception handler. Preemption does not occur if the incoming exception has the same or lower priority as the active exception. Once the higher-priority exception is serviced, the CPU resumes execution of the lower-priority exception handler. With eight priority levels supported, up to eight nested exceptions are possible. If the CPU receives two or more exception requests of the same priority, it services the exception with the lowest exception number first.

The priority of exceptions numbered 4 through 15 can be configured by writing to the three system handler priority registers (SHPR1, SHPR2, SHPR3). Each SHPRn register stores the priority for four exceptions (PRI_N), where each priority corresponds to an exception number. Specifically:

- 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 interrupts is configured by writing to the corresponding interrupt priority registers (NVIC_IPRn). These consist of 124 32-bit registers, with each register storing the priority values for four interrupts. The interrupt number corresponding to each field is represented as 4n + m, where:

- n indicates the register instance
- m specifies the field number within the register

Table 14 Interrupt priority register bit definitions for NVIC_IPRn

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

7.7 Enabling and disabling interrupts/exceptions

The nested vectored interrupt controllers (NVICs) of Armv8-M CPUs include registers that enable or disable interrupts individually through software control. If an interrupt is not enabled, the NVIC does not process interrupt requests for that interrupt line. The Interrupt Set Enable Register (ISERn) and the Interrupt Clear-Enable Register (ICERn) are used to enable and disable interrupts, respectively.

The Armv8-M architecture provides 16 instances of these 32-bit ISER and ICER registers. Each bit within these registers corresponds to an interrupt line, and the interrupt number is represented as 32n + m, where:

- n refers to the register instance
- m represents the bit number within the register

These registers can also be read by software to determine the current enable status of the interrupts. Writing a 1 to a specific bit in the ISER enables the corresponding interrupt, while writing a 1 to a bit in the ICER disables it. It is important to note that writing a 0 to either of these registers does not affect their state.

The enable/disable functionality of interrupts and exceptions demonstrates the register access properties of ISER and ICER. These capabilities are key to controlling interrupt processing at the NVIC level.

Table 15 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
Interrupt Clear Enable Register (ICERn)	Write	1	To disable the Interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled

The ISER and ICER registers are exclusively applicable for managing interrupts. These registers cannot be used to enable or disable exceptions with numbers 1 through 15. These exceptions have dedicated mechanisms for enabling and disabling, as described in Exception sources.

Interrupts and exceptions can also be controlled using the Armv8-M special-purpose mask registers: PRIMASK, FAULTMASK, and BASEPRI. These registers provide additional granularity and global control for masking exceptions and interrupts. Their functionalities are as follows:

1. **PRIMASK register**:
   - Acts as a global exception enable register
   - Masks all configurable-priority exceptions, regardless of their individual enable status
   - Configurable-priority exceptions include all exceptions except Reset, NMI, and HardFault
2. **FAULTMASK register**:
   - Disables all faults and interrupts except the NMI exception
   - Useful for handling critical fault scenarios where all other interruptions must be suppressed
3. **BASEPRI register**:
   - Provides priority-based interrupt masking
   - Disables all interrupts with a priority value equal to or higher (lower priority) than the value in the BASEPRI register. Only interrupts with a lower numerical priority value (higher priority) than BASEPRI can trigger

When the PRIMASK, FAULTMASK, or BASEPRI registers are set, the CPU cannot service any of the masked exceptions or interrupts. These masked exceptions remain in the pending state until the corresponding mask is cleared.

7.8 Interrupt/exception states

Each exception in the Armv8-M architecture can be in one of the following states:

Table 16 Interrupt/exception states

Exception state	Meaning
Inactive	• The exception is neither pending nor active • This occurs when the exception is either disabled or, if enabled, has not been triggered by any event or condition
Pending	The exception request has been received by the CPU/ NVIC and is waiting to be serviced by the CPU.
Active	• The exception is currently being serviced by the CPU • This state describes an exception whose handler is being executed • If a high-priority exception occurs during the execution of a lower-priority exception handler, both exceptions will be in the active state simultaneously (the high-priority exception preempts the lower-priority one)
Active and Pending	The exception is being serviced by the processor, and there is a pending request from the same source during its handler execution.

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

The VECTACTIVE bits ([8:0]) in the ICSR store the exception number of the currently executing exception. This value is zero if the CPU is not executing any exception handler (i.e., the CPU is in thread mode). The value in the VECTACTIVE bitfields is the same as the value in bits [8:0] of the Interrupt Program Status Register (IPSR), which also holds 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 whether an NVIC-generated interrupt is in the pending state.

7.8.1 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. Once the CPU starts executing the exception handler routine for that exception, it transitions from the pending state to the active state.

The NVIC provides the capability to manage the pending states of all interrupt lines through software by offering separate register bits to set and clear the pending status. The Interrupt Set-Pending Register (ISPRn) and the Interrupt Clear-Pending Register (ICPRn) are used to control the pending status of interrupt lines:

- Writing to the ISPRn sets the pending state of the corresponding interrupt
- Writing to the ICPRn clears the pending state of the corresponding interrupt

There are 16 instances of the 32-bit ISPR and ICPR registers. Each bit within these registers corresponds to an interrupt. The interrupt number associated with each bit is represented as 32n + m, where:

- n is the register instance
- m is the bit number within the register

Writing a 1 to a specific bit in the ISPR sets the pending state for the corresponding interrupt, while writing a 1 to the corresponding bit in the ICPR clears the pending state. It is important to note that writing a 0 to these registers has no effect on their state. Table 17 shows the register access properties for these two registers.

Table 17 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

When the pending bit in the ISPR register is set while it is already set, it results in only one execution of the Interrupt Service Routine (ISR). The pending bit can be updated regardless of whether the corresponding interrupt is enabled. However, 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. It is important to note that the ISPR and ICPR registers are used exclusively for peripheral interrupts. These registers cannot be used to manage the pending state of exception numbers 1 to 15.

The pending bits for exceptions 1 to 15 are managed through the Armv8-M System Handler Control and State Register (SHCSR), Interrupt Control and State Register (ICSR), and Debug Exception and Monitor Control Register (DEMCR) as follows:

1. **SHCSR register**: The SHCSR contains the following pending bits for exceptions with corresponding names:
   - HARDFAULTPENDED
   - SECUREFAULTPENDED
   - SVCALLPENDED
   - BUSFAULTPENDED
   - MEMFAULTPENDED
   - USGFAULTPENDED
   - Writing a 1 to these bits sets the pending state of the exception
   - Clearing these bits removes the pending state of the exception
   - There are no separate clear-pending registers for these exceptions
2. **ICSR register**: The ICSR contains the following bits to set the pending state of specific exceptions:
   - **PENDNMISET**: Sets the pending state of the NMI exception when writing a 1

- **PENDSVSET**: Sets the pending state of the PendSV exception when writing a 1
- **PENDSTSET**: Sets the pending state of the SysTick exception when writing a 1
- The ICSR also provides bits to clear the pending state of these exceptions:
  - **PENDNMICLR**: Clears the pending state of the NMI exception when writing a 1
  - **PENDSVCLR**: Clears the pending state of the PendSV exception when writing a 1
  - **PENDSTCLR**: Clears the pending state of the SysTick exception when writing a 1

3. **DEMCR register**: The DEMCR contains the MON_PEND bit, which is used to set or clear the pending state of the DebugMonitor exception

7.9 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 switches to handler mode and begins executing the corresponding exception handler. During this process, the CPU pushes the contents of eight 32-bit internal registers onto the stack. These registers include:

- Program and Status Register (PSR)
- ReturnAddress
- Link Register (LR or R14)
- R12
- R3
- R2
- R1
- R0

The Cortex®-M33 processor has two stack pointers: the Main Stack Pointer (MSP) and the Process Stack Pointer (PSP). However, only one stack pointer can be active at a time.

- In thread mode, the Active Stack Pointer bit in the Control register determines the currently active stack pointer
- In handler mode, the MSP is always used as the stack pointer

The stack pointer always grows downward, pointing to the address of the last-pushed data.

When the CPU is in thread mode and an exception request occurs, it uses the stack pointer defined in the Control register to store the contents of the general-purpose registers. After completing the stack push operations, the CPU switches to handler mode to execute the exception handler.

If a higher-priority exception occurs while the CPU is already executing an exception handler, the MSP is used for stack push and pop operations. This is because the CPU is already in handler mode. For more details, see CPUSS.

7.10 Interrupts and low-power modes

The CYW20829 device allows the CPU to wake up from low-power modes when certain peripheral interrupt requests are generated. The Wakeup Interrupt Controller (WIC) generates a wakeup signal that transitions the CPU into Active mode when one or more wakeup sources trigger an interrupt signal. Once the CPU enters Active mode, the corresponding interrupt service routine (ISR) 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 Deep Sleep modes. Both WFI and WFE instructions can wake the CPU on interrupts, but their behavior differs:

- The WFE instruction requires interrupts to be unmasked in the CPU's Exception Mask Register (PRIMASK). It also puts the CPU to sleep based on the status of an event bit and wakes it up from an event signal, which is typically sent by another CPU
- The WFI instruction does not require PRIMASK unmasking and can wake the CPU from any pending interrupt handled by the NVIC or WIC. However, the WFI instruction cannot wake the CPU from event signals sent by other CPUs

The sequence for entering low-power modes is detailed in the Device power modes section.

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

1. Interrupt sources available in CPU Active, CPU Sleep, and CPU Deep Sleep modes
2. Interrupt sources available only in CPU Active and CPU Sleep modes

See Interrupt sources for details on interrupts that support Deep Sleep. When using the WFE instruction, it is necessary to execute the instruction twice to properly enter and exit Sleep or Deep Sleep modes. This behavior arises 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 directly read or write the value of the event register

WFE instruction behaviour:

1. The first WFE instruction puts the CPU to sleep
2. After a WFE wakeup, the event register is set
3. The second WFE instruction clears the event register
4. Subsequent WFE instructions can then properly put the CPU back to sleep

7.11 Interrupt/exception – Initialization/configuration

This section explains the steps involved in initializing and configuring exceptions in CYW20829.

The first step in using exceptions is to configure the location of the vector table, typically in SRAM. This configuration is performed as described in Exception vector table. The vector table should reside in SRAM if the application needs to change vector addresses dynamically. In the ModusToolbox™ IDE, the vector table for the Cortex®-M33 is configured in SRAM.

The next step is to configure the specific exceptions required for the application. The process includes the following steps:

1. **Configure the exception or interrupt source**:
   - Set up the conditions for interrupt generation
   - The register configuration depends on the specific exception being used
   - Refer to the respective peripheral section for detailed information about the interrupt configuration supported by that peripheral
2. **Define the exception handler function**:
   - Define the handler function for the exception
   - Write the address of this function to the exception vector table
   - See Table 12 for the exception vector table format. The handler address should be written to the appropriate entry corresponding to the exception number
3. **Set up the exception priority**: Configure the priority for the exception, as described in the Interrupt/exception priority section
4. **Enable the exception**: Enable the exception, as explained in Enabling and disabling interrupts/exceptions

8 Fault monitoring

Fault monitoring enables the detection of various faults within the device and allows actions to be taken based on the reported faults. In addition to fault reporting, the fault structures in CYW20829 MCUs provide mechanisms to log data from fault sources and optionally perform a soft reset.

The CYW20829 MCU family supports two centralized fault reporting and monitoring structures that track faults generated within the device. Each fault report structure can monitor and report faults from up to 96 sources, while the CYW20829 specifically supports 30 fault sources.

8.1 Features

The CYW20829 MCU fault report structure supports the following features:

- Monitoring protection unit access violation faults, peripheral-specific errors, memory-controller-specific errors, and similar issues
- Four 32-bit data registers to record fault information
- Soft reset on fault detection while retaining fault information
- Interrupt generation on fault detection
- Trigger output to the DMA for fault data transfer
- Fault-detected output to a pin for external fault handling

8.2 Architecture

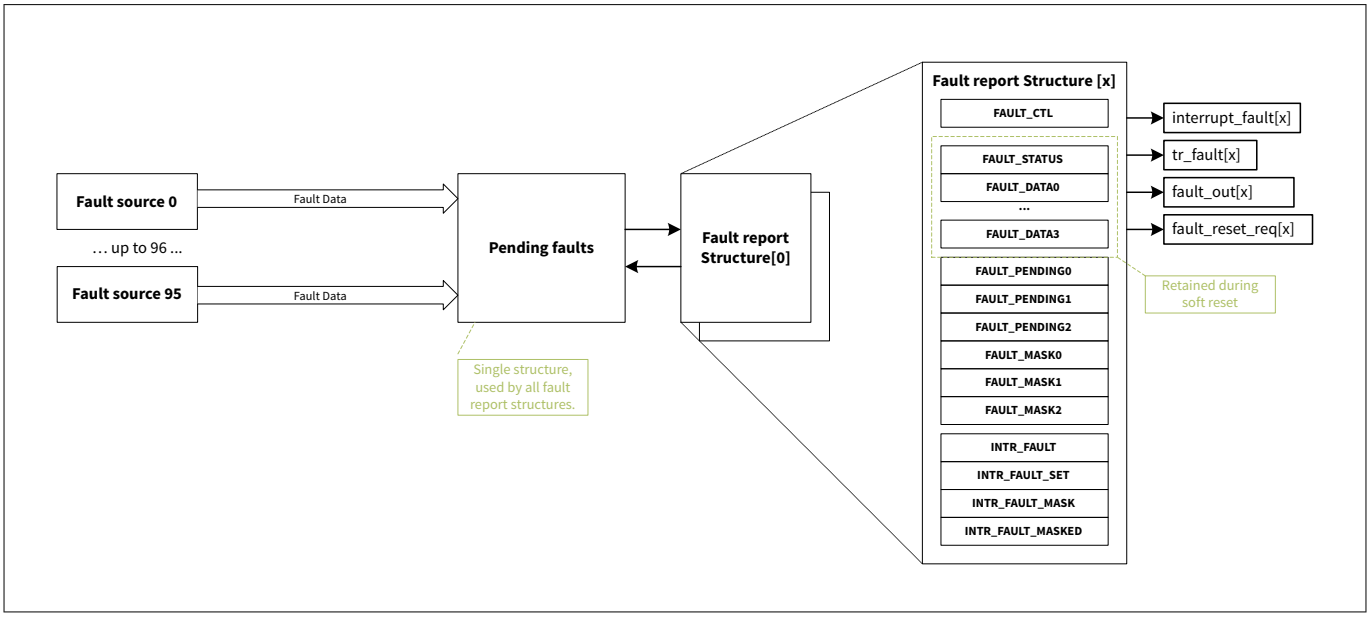


Figure 15 Fault report structure

CYW20829 uses centralized fault report structures. This centralized design enables system-wide, consistent handling of faults, simplifying software development. It requires only a single fault interrupt handler. Each fault report structure provides the fault source and additional fault-specific information through a single set of MMIO registers. This eliminates the need for iterative searches for fault sources and information. All pending faults are available from a single set of MMIO registers.

The fault structures capture faults related to:

- MPC/PPC protection violations
- Peripheral-specific errors
- Memory-controller-specific errors

- ECC errors
- Timeout errors

Fault structures can be configured to capture one or more faults as listed in Figure 15. When a fault structure is configured to capture a specific fault, any occurrence of that fault is recorded as a pending fault. If there are no other pending faults or once all other faults have been processed, the fault data is captured into the fault structure registers. A successful capture can trigger an interrupt, which can then be processed by either Cortex®-M33 depending on the application's requirements.

It is important to note that each fault structure is capable of capturing only one fault at a time. Until the captured fault is serviced, subsequent faults will not be captured by the structure. In addition to capturing faults, the fault structure can optionally perform a soft reset while retaining fault information. This reset results in the RESET_ACT_FAULT reset cause being recorded in the SRSS_RES_CAUSE register.

8.2.1 Fault report

CYW20829 supports a single fault report structure, which includes a dedicated set of control and status registers. This fault report structure captures a single fault at a time. The captured fault information includes:

- **Fault validity bit**:
  - Indicates whether a fault has been captured (FAULT_STRUCT0_STATUS.VALID)
  - This bit is set whenever a fault is captured and should be cleared after processing the fault information
  - New faults are captured only when this bit is set to 0
- **7-bit fault index**: Identifies the fault source (FAULT_STRUCT0_STATUS.IDX)
- **Additional fault information**:
  - Describes fault specifics using registers FAULT_STRUCT0_DATA0 through FAULT_STRUCT0_DATA3
  - The encodings for this fault data can be found in the Fault encodings section

The FAULT_STRUCT0_STATUS register can also be written to by software to trigger a fault manually.

8.2.2 Signaling interface

In addition to capturing fault information, each fault report structure provides a signaling interface to notify the rest of the system of the captured fault. This interface supports the following:

- **Fault interrupt (interrupt_fault[0])**: The fault interrupt is monitored, set, and masked using the FAULT_STRUCT0_INTR, FAULT_STRUCT0_INTR_SET, FAULT_STRUCT0_INTR_MASK, and FAULT_STRUCT0_INTR_MASKED registers. When the interrupt is configured, an interrupt request is sent to the NVIC when FAULT_STRUCT0_STATUS.VALID is set. Only a single interrupt cause is supported, indicating that a fault has been detected. The fault report structure can be read in the interrupt handler to identify and process the fault
- **DMA trigger (tr_fault[0])**: The DMA trigger is enabled by setting FAULT_STRUCT0_CTL.TR_EN. When enabled, the DMA trigger activates upon setting FAULT_STRUCT0_STATUS.VALID. This trigger can be connected to a DMA controller, which transfers the captured fault information from the fault report structure to memory and clears the FAULT_STRUCT0_STATUS.VALID field. For failure analysis, it is recommended to use a memory location that is retained during a warm or soft reset
- **Chip output signal (fault_out[0])**: The output signal is enabled by setting FAULT_STRUCT0_CTL.OUT_EN. When enabled, the fault structure generates an output signal upon setting FAULT_STRUCT0_STATUS.VALID. This signal can be routed out of the device through the HSIOM and used to communicate non-recoverable faults to off-chip components. This may result in a reset of the chip
- **Fault reset request (fault_reset_req[0])**: The reset capability is enabled by setting FAULT_STRUCT0_CTL.RESET_REQ_EN. When enabled, FAULT_STRUCT0_STATUS.VALID triggers a soft reset. The reset 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 device provides four different signaling interfaces, each with its own enabling functionality. Once an interface is enabled, it is activated when FAULT_STRUCT0_STATUS.VALID is set.

Since the device supports a single fault_reset_req signal, the individual fault_reset_req[x] signals from the fault structures are combined into a unified fault_reset_req signal, as shown in Figure 16.

***Note**: The variable x represents different fault report structures that share a single central structure.*

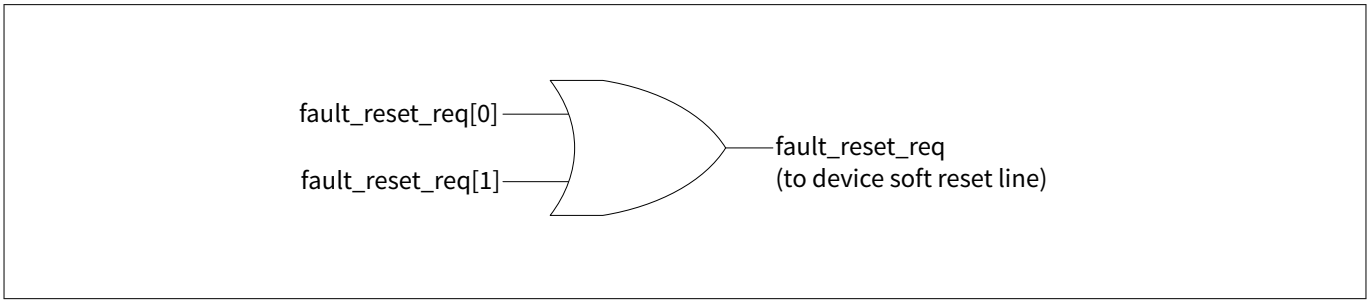


Figure 16 Fault reset

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 reflect the fault sources that are pending. Each register provides a single pending bit for up to 96 fault sources. These FAULT_STRUCT0_PENDINGx registers are mirrored in the fault report structure. The fault source numbering follows the scheme used by FAULT_STRUCT0_STATUS.IDX (see Table 18 for details).

Pending fault sources are faults that have not yet been captured by a fault structure. When a pending fault is captured by a fault structure, the associated pending bit is cleared to 0.

The fault report structure selectively captures faults. The FAULT_STRUCT0_MASK0, FAULT_STRUCT0_MASK1, and FAULT_STRUCT0_MASK2 registers reflect the pending fault sources that are captured by the fault structure. These faults are referred to as "enabled" faults. The fault structure contains its own set of FAULT_STRUCT0_MASK registers.

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

It is important to note that fault capturing does not consider the FAULT_STRUCT0_INTR.FAULT bit. If the 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 fault report structures are reset when the system transitions to 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 during the transition to CPU Deep Sleep power mode.

The fault structure registers can be categorized based on their reset domain and retention capability as follows:

- **Active reset domain**: Registers: FAULT_STRUCT0_PENDING, FAULT_STRUCT0_INTR, FAULT_STRUCT0_INTR_SET, and FAULT_STRUCT0_INTR_MASKED. These registers are not retained in CPU Deep Sleep power mode

- **Deep Sleep reset domain**: Registers: FAULT_STRUCT0_CTL, FAULT_STRUCT0_MASK, and FAULT_STRUCT0_INTR_MASK. These registers are retained in CPU Deep Sleep power mode. However, any system reset will reset them to their default state
- **Soft reset domain**: Registers: FAULT_STRUCT0_STATUS and FAULT_STRUCT0_DATA. These registers are retained through soft resets, which are detectable in the SRSS_RES_CAUSE register. However, hard resets (such as XRES, POR, or BOD) will reset these registers to their default state

8.2.5 Using a fault structure

Follow these steps to configure and use a fault structure:

1. See Table 18 to identify the faults that need to be monitored in the system
2. For firmware fault handling through interrupts:
   a. Set the FAULT_STRUCT0_INTR_MASK.FAULT bit to enable fault interrupts
   b. Clear any pending interrupts by setting the FAULT_STRUCT0_INTR.FAULT bit
   c. Enable the fault interrupt to the CPU by configuring the appropriate ISER register. See Interrupts for details
3. For fault handling through DMA:
   a. Set the FAULT_STRUCT0_CTL.TR_EN bit to enable the DMA trigger
   b. Route the tr_fault[0] signal to the trigger input of the DMA controller
   c. Configure and enable the DMA controller to transfer the FAULT_STRUCT0_STATUS and FAULT_STRUCT0_DATA registers to memory
   d. After the transfer is complete, write 0 to the FAULT_STRUCT0_STATUS register to clear the fault status
4. For fault handling outside the device:
   a. Set the FAULT_STRUCT0_CTL.OUT_EN bit to enable the output signal
   b. Route the fault_out[x] signal to an external pin through HSIOM
   c. Use the signal externally to process the fault, generate an external reset, perform a power cycle, or log fault information
5. If a soft reset is required for any fault, set the FAULT_STRUCT0_CTL.RESET_REQ_EN bit
6. Write 0 to the FAULT_STRUCT0_STATUS.VALID register to clear any captured fault
7. Set the fault index bits in the FAULT_STRUCT0_MASK registers for the faults that need to be captured by the fault structure

8.2.6 CPU exceptions versus fault monitoring

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

1. As a bus error to the master of the faulting bus transfer: This results in Bus, Usage, Memory, or Hard fault exceptions in the CPU
2. As a fault in a fault report structure:
   - The fault can be communicated as a fault interrupt to any processor in the system
   - This allows fault handling to occur on a processor that is not the master of the faulting bus transfer
   - This method is especially useful for faults that make the master of the faulting transfer unresponsive or unreliable

8.3 Fault sources

The fault sources can differ between device families. Table 18 lists the fault sources available in CYW20829.

Table 18 Fault sources

Fault index	Source	Descriptions
0	cpuss.mpu_vio[0]	CM0+ MPU/SMPU violation
1	cpuss.mpu_vio[0]	CRYPTO MPU/SMPU violation
2	cpuss.mpu_vio[0]	DW0 MPU/SMPU violation
3	cpuss.mpu_vio[0]	DW1 MPU/SMPU violation
4	cpuss.mpu_vio[0]	DMAC MPU/SMPU violation
5	cpuss.mpu_vio[0]	SDHC[0] MPU/SMPU violation
6	cpuss.mpu_vio[0]	SDHC[1] MPU/SMPU violation
7	cpuss.mpu_vio[0]	Bus master #7 MPU/SMPU violation (when used)
8	cpuss.mpu_vio[0]	Bus master #8 MPU/SMPU violation (when used)
9	cpuss.mpu_vio[0]	Bus master #9 MPU/SMPU violation (when used)
10	cpuss.mpu_vio[0]	Bus master #10 MPU/SMPU violation (when used)
11	cpuss.mpu_vio[0]	Bus master #11 MPU/SMPU violation (when used)
12	cpuss.mpu_vio[0]	Bus master #12 MPU/SMPU violation (when used)
13	cpuss.mpu_vio[0]	Bus master #13 MPU/SMPU violation (when used)
14	cpuss.mpu_vio[0]	CM4 MPU/SMPU violation (I/D bus)
15	cpuss.mpu_vio[0]	Test Controller MPU/SMPU violation
16	CM4_SYS_MPU	CM4 MPU/SMPU violation (S bus)
17	CM4_CODE_MPU	CM4 code error
18	CM4_CODE_FLASHC_MPU	CM4 code Flash controller errors
19 to 27	Reserved	
28	peri_ms_vio[0]	CM0+ mxsperipheral Master Interface PPU violation
29	peri_ms_vio[1]	CM4 mxsperipheral Master Interface PPU violation
30	peri_ms_vio[2]	DW0 mxsperipheral Master Interface PPU violation
31	peri_ms_vio[3]	DW1 mxsperipheral Master Interface PPU violation

(table continues...)

Table 18 (continued) Fault sources

Fault index	Source	Descriptions
32	peri.group_vio[0]	mxsperipheral group #0 PPU violation
33	peri.group_vio[1]	mxsperipheral group #1 PPU violation
34	peri.group_vio[2]	mxsperipheral group #2 PPU violation
35	peri.group_vio[3]	mxsperipheral group #3 PPU violation
36	peri.group_vio[4]	mxsperipheral group #4 PPU violation
37	peri.group_vio[5]	mxsperipheral group #5 PPU violation
38	peri.group_vio[6]	mxsperipheral group #6 PPU violation
39	peri.group_vio[7]	mxsperipheral group #7 PPU violation
40	peri.group_vio[8]	mxsperipheral group #8 PPU violation
41	peri.group_vio[9]	mxsperipheral group #9 PPU violation
42	peri.group_vio[10]	mxsperipheral group #10 PPU violation
43	peri.group_vio[11]	mxsperipheral group #11 PPU violation
44	peri.group_vio[12]	mxsperipheral group #12 PPU violation
45	peri.group_vio[13]	mxsperipheral group #13 PPU violation
46	peri.group_vio[14]	mxsperipheral group #14 PPU violation
47	peri.group_vio[15]	mxsperipheral group #15 PPU violation
48	cpuss.flashhc_main_bus_err	Flash controller bus error
49 to 95	Reserved	

8.4 Registers summary

Name	Description
FAULT_STRUCT0_CTL	Fault control registers
FAULT_STRUCT0_STATUS	Fault status registers

Name	Description
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_INTR	Fault interrupt cause registers
FAULT_STRUCT0_INTR_SET	Fault interrupt set registers
FAULT_STRUCT0_INTR_MASK	Fault interrupt mask registers
FAULT_STRUCT0_INTR_MASKED	Fault interrupt masked registers

9 Direct memory access (DMA DW)

The 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 of the CPU. The DMA can be configured to perform multiple independent data transfers, all of which are managed by channels. The DMA includes 16 channels.

The CPU incorporates a DMA controller that facilitates data transfers between memory, peripherals, and registers. This enables autonomous and deterministic control of peripherals, such as the PWM, and supports large data transfers from peripherals, such as the ADC, to memory.

The DMA controllers act as bus masters within the M33 domain:

- **M33 domain**: The DMA contains 16 channels with a single transfer engine shared across all channels to arbitrate bus master access. This DMA operates on the 32-bit AHB bus, which shares the clock with the M33 processor. It is optimized for small data transfers and is typically used for transactional DMA operations, such as transferring data between memory and peripherals or vice versa

The DMA is referred to as the data wire (DW) in register names and trigger references in the Trigger MUX chapter.

9.1 Features

The DMA controller offers the following features:

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

9.2 Architecture

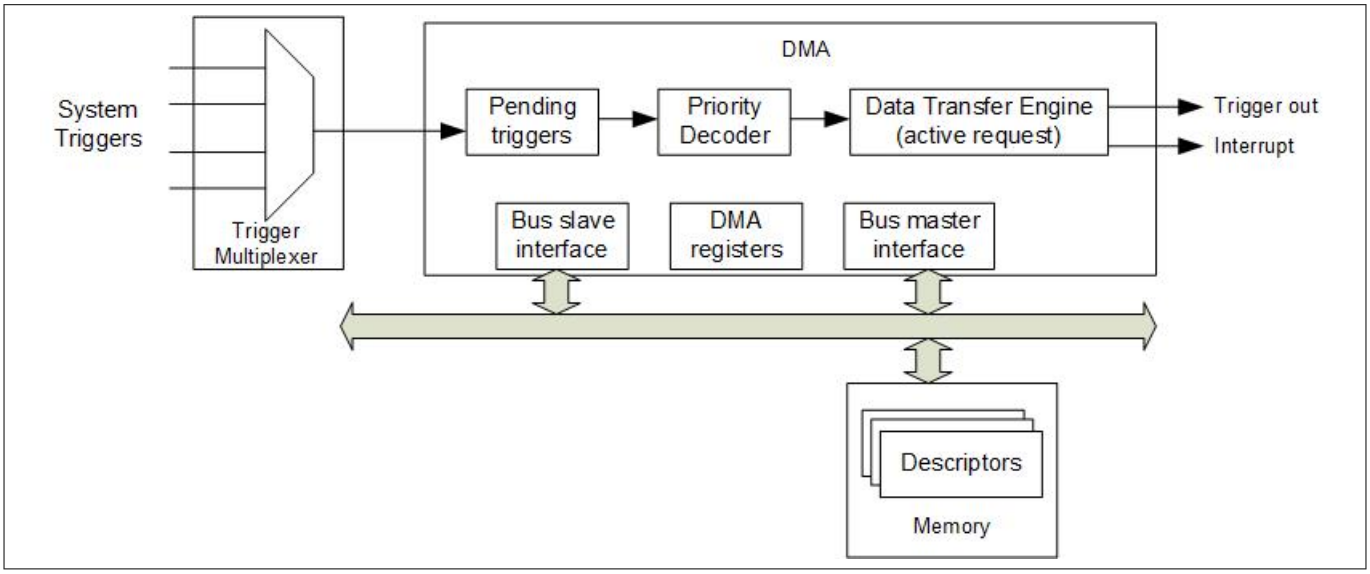


Figure 17 Block diagram

A data transfer is initiated by an input trigger. This trigger may originate from the source of the transfer, the destination of the transfer, CPU software, or another SoC component. Triggers provide Active/Sleep functionality but are not available in CPU/System Deep Sleep 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 1D or 2D memory copies. At the end of a transfer, the descriptor can either initiate a new transfer by pointing to a new descriptor or generate a completion interrupt.

The DMA is divided into several functional blocks, as described below:

- **Pending triggers**: Tracks activated triggers by locally storing them in pending bits. This is essential because multiple channel triggers may be activated simultaneously; however, 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**: Responsible for transferring data from a source location to a destination location. When idle, it is ready to accept the highest-priority activated channel. It also handles reading the channel descriptor from memory
- **AHB slave I/F**: Acts as an AHB bus slave, allowing the main CPU to access controller MMIO control and status registers
- **AHB master I/F**: Acts as an AHB bus master, enabling the controller to initiate AHB data transfers to source and destination locations as well as read descriptors from memory
- **Interrupt logic**: Manages interrupt status for each channel
- **MMIO registers**: Stores descriptors for all channels
- **QCH (Q-channel) controller**: Allows 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 the steps outlined below:

1. The main CPU programs the descriptor chain in memory and associates it with a specific channel. It also programs the channel registers to set the desired attributes for the channel and configures the MMIO register to select 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 via the master interface (I/F). The descriptor structure specifies the channel's data transfers
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 to 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 the channel structure registers to reflect the data transfers. There is no update to the descriptors stored in memory
8. Output trigger generation and interrupt generation are determined by TR_OUT_TYPE and INTR_TYPE, respectively. The Data Wire (DW) can generate an error interrupt if it encounters any errors. The corresponding channel is disabled when an error occurs

9.3 Channels

The DMA controller supports multiple independent data transfers, each managed by a channel. Each channel connects to a specific system trigger through a trigger multiplexer located outside the DMA controller.

9.3.1 Channel priority

Only one DMA channel can move data at a time. Each channel is assigned a priority to determine which channel gets bus access if multiple channels are triggered simultaneously.

- Each channel is assigned a priority (CHi_CTL.PRIO) ranging from 0 to 3, where 0 is the highest priority and 3 is the lowest priority. Channels with the same priority form a priority group
- Priority decoding identifies the highest-priority pending channel as follows:
  1. The highest-priority group with pending channels is identified first
  2. Within this priority group, round-robin arbitration is applied. Round-robin arbitration gives the highest priority to lower channel indices within the priority group

Each channel can be marked as preemptable by setting CHi_CTL.PREEMPTABLE to 1. When a higher-priority channel transitions to a pending state, it interrupts a preemptable channel during a 1D transfer. After the higher-priority channel completes its transfer, the preempted channel is reactivated by the DMA controller state machine.

9.3.2 Channel security attributes

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

- **Privileged mode (CHi_CTL.P)**: This attribute can be inherited as either privileged or user
- **Non-secure (CHi_CTL.NS)**: This attribute can be inherited as either secure or non-secure
- **PC (CHi_CTL.PC)**: This field inherits the protection context from the write transaction

9.3.3 Channel state

At any given time, only one channel can actively perform a data transfer. This channel is referred to as the active channel. A channel can be in one of four states.

Table 19 Channel states

Channel state	Register	Description
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 transfers. If multiple channels are pending, the highest-priority pending channel is scheduled.
Blocked	CHi_CTL.ENABLED = 1 CHi_STATUS.PENDING = 0	The channel is enabled and awaiting a trigger to initiate a data transfer.
Disabled	CHi_CTL.ENABLED = 0	The channel is disabled by setting `DW_CHi_CTL.ENABLED` to 0. In this state, the channel trigger is ignored. If an active channel is disabled by software, no assumptions should be made about the channel's state, such as the current "position" of the transfer reflected by the MMIO registers or descriptors. To re-enable the channel, software must prepare new descriptors and reconfigure the channel.
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 to be scheduled.

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

- **Val 0**: Default or inactive state
- **Val 1**: Loading channel descriptor from memory
- **Val 2**: Loading data element from source location
- **Val 3**: Storing data element to destination location
- **Val 4**: CRC functionality (used only when `DESCR_CTL.DESCR_TYPE` is set to CRC)
- **Val 5**: Update of 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 associated interrupt line. The DMA controller generates interrupts upon the completion of a transfer or when an error condition occurs. These interrupts can either wake the CPU at the end of a transfer or signal the CPU to address a DMA error. Each DMA channel has six registers for controlling and interacting with interrupts. For more information on transfer types, see Descriptor memory transfer types.

The DESCR_CTL.INTR_TYPE field determines the type of completion interrupt (when CHi_STATUS.INTR_CAUSE is set to COMPLETION) that 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 completion of a 1D transfer
- **Set 2**: An interrupt is generated after the execution of the current descriptor, irrespective of the value of DESCR_NEXT_PTR.ADDR in the descriptor
- **Set 3**: An interrupt is generated after the execution of the current descriptor, but only if the value of DESCR_NEXT_PTR.ADDR in the descriptor is 0

The CHi_STATUS.INTR_CAUSE field specifies the source of the interrupt when CHi_INTR.CH is set to 1. The following interrupts can be generated by 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)

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

The CHi_INTR_SET register is set to 1 to manually set the corresponding field in CHi_INTR.

The CHi_INTR_MASK register is used to configure the interrupt mask for the specified DMA channel.

The CHi_INTR_MASKED register provides the bitwise AND between CHi_INTR and CHi_INTR_MASK when read.

For error-related interrupt causes (CHi_STATUS.INTR_CAUSE values 2 through 8), the hardware disables the channel by setting CHi_CTL.ENABLED to 0. The interrupt must be cleared before the channel is re-enabled.

The DMA controller operates as a CPU Active power-mode peripheral. The INTR and INTR_SET registers are not retained during CPU/System Deep Sleep power modes, but the INTR_MASK register is retained.

9.4 Descriptors

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

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

Each channel includes the register CHi_IDX, which contains the bit fields X_IDX and Y_IDX. These fields represent the current descriptor's X and Y indices as the transfer progresses. Both fields 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 types of memory transfers: single, 1D, 2D, and CRC (cyclic redundancy check) transfers. The memory transfer type is configured using the DESCR_CTL.DESCR_TYPE field. Depending on the transfer type, up to six configured 32-bit words may be required to support the specified transfer type. The four different memory transfer types are listed in the following sub-sections.

9.4.1.1 Single transfer

The descriptor configuration performs a single data element transfer. The size of the data element can be configured using 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 pseudo-code describes the functionality of a single transfer:

Required configurations:

- DESCR_CTL
- DESCR_SRC
- DESCR_DST
- 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_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_SIZEDST_ADDR[0] = (t_DATA_SIZE)
SRC_ADDR[0]
```

Figure 18 demonstrates how to move 32 bits in memory using a single transfer.

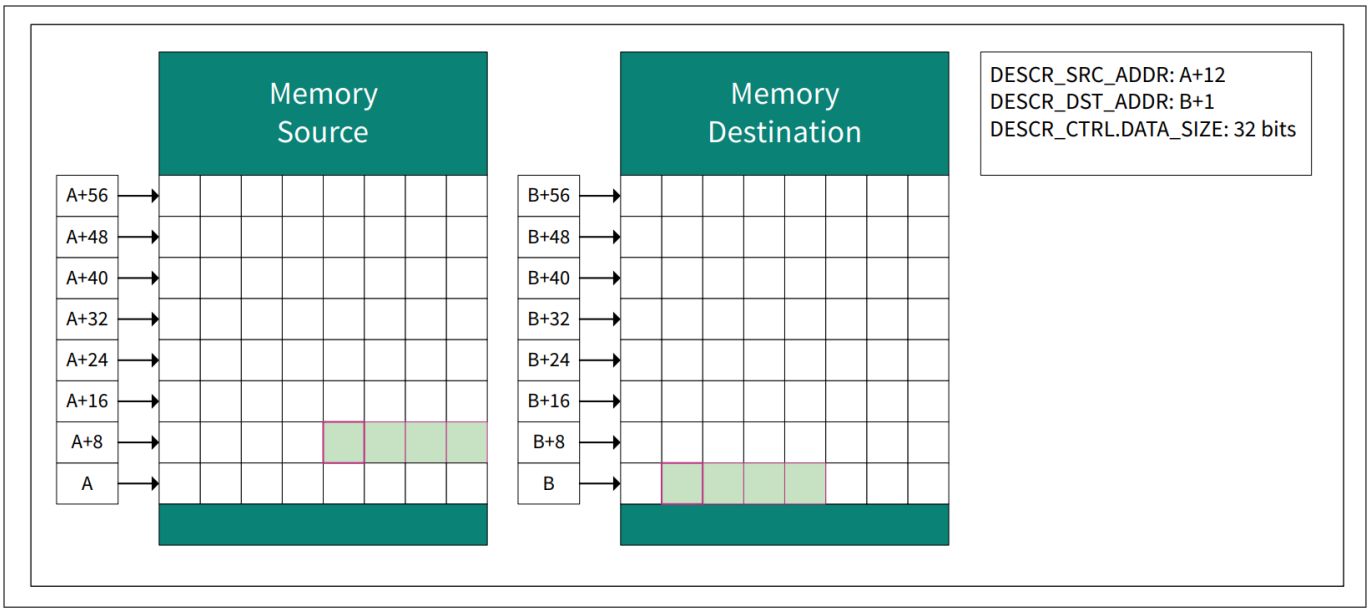


Figure 18 Single transfer

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 pseudo-code outlines the functionality of a 1D memory transfer:

Required configurations:

- `DESCR_CTL`
- `DESCR_SRC`
- `DESCR_DST`
- `DESCR_X_CTL`
- `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
DSCR_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};}
```

Figure 19 demonstrates how to move three 32-bit data elements in memory using a 1D memory transfer.

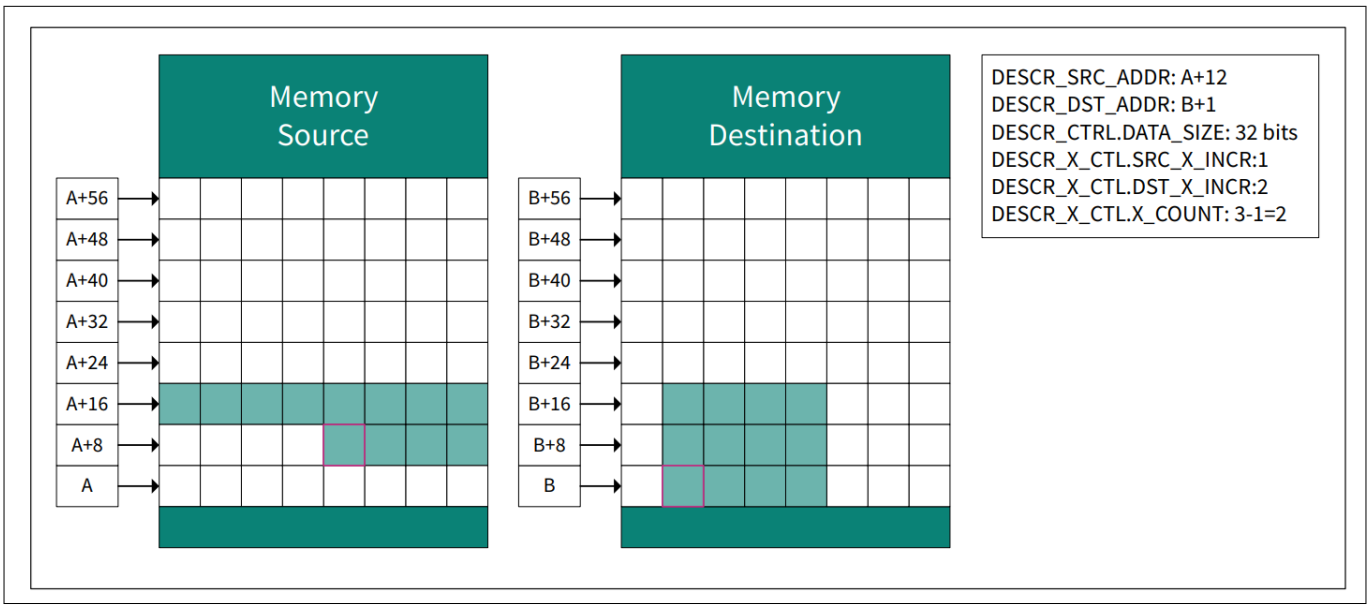


Figure 19 1D memory transfer

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 pseudo-code outlines the functionality of a 2D memory transfer:

Required configurations:

- DESCR_CTL
- DESCR_SRC
- DESCR_DST
- DESCR_X_CTL
- DESCR_Y_CTL
- 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
transfersdefined 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];}}
```

Figure 20 demonstrates how to move six 32-bit data elements in memory using a 2D memory transfer.

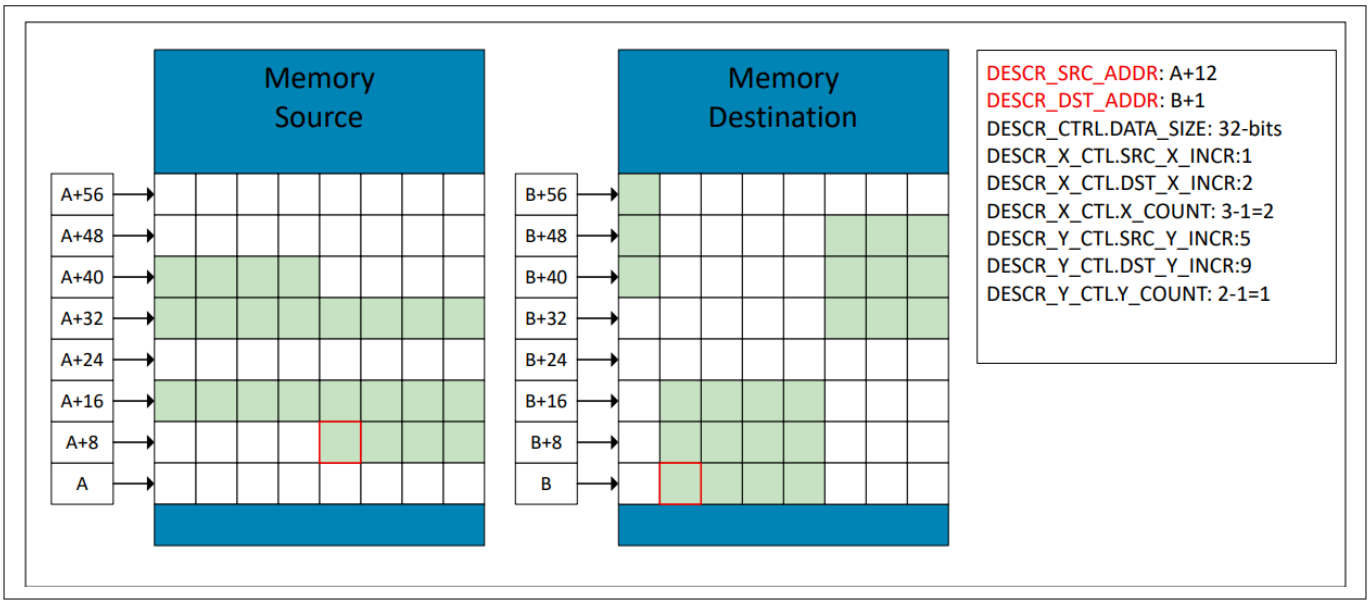


Figure 20 2D memory transfer

9.4.1.4 Cyclic redundancy checker (CRC) transfer

The descriptor configuration performs a one-dimensional "for loop" similar to the 1D memory transfer. However, instead of transferring source data to a destination, 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 shared by all DW channels. It is assumed that the DW channels use the CRC functionality mutually exclusively over time.

The following pseudo-code outlines the functionality of a CRC transfer:

Required configurations:

- DESCR_CTL
- DESCR_SRC
- DESCR_DST
- DESCR_X_CTL
- DESCR_NEXT_PTR

Required registers:

- CRC_CTL
- CRC_DATA_CTL
- CRC_POL_CTL
- CRC_LFSR_CTL
- CRC_REM_CTL
- 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++) {
pdate_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 descriptor type, as only relevant parameters are stored. For example, a single transfer descriptor does not include DESCR_X_CTL and DESCR_Y_CTL parameters.

When a channel is triggered, the descriptor is fetched from memory and loaded 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
- CHi_ACT_DESCR_NEXT_PTR

The DMA controller fetches only the required 32-bit descriptor words based on the specified DESCR_TYPE.

9.4.3 Descriptor configuration

A channel descriptor consists of six 32-bit words. However, not all configurations require all six 32-bit words. When reading the descriptor from memory, only the required descriptor size is fetched. For example, a single transfer requires only four 32-bit words to be read from memory.

The descriptor layout is shown in Figure 21.

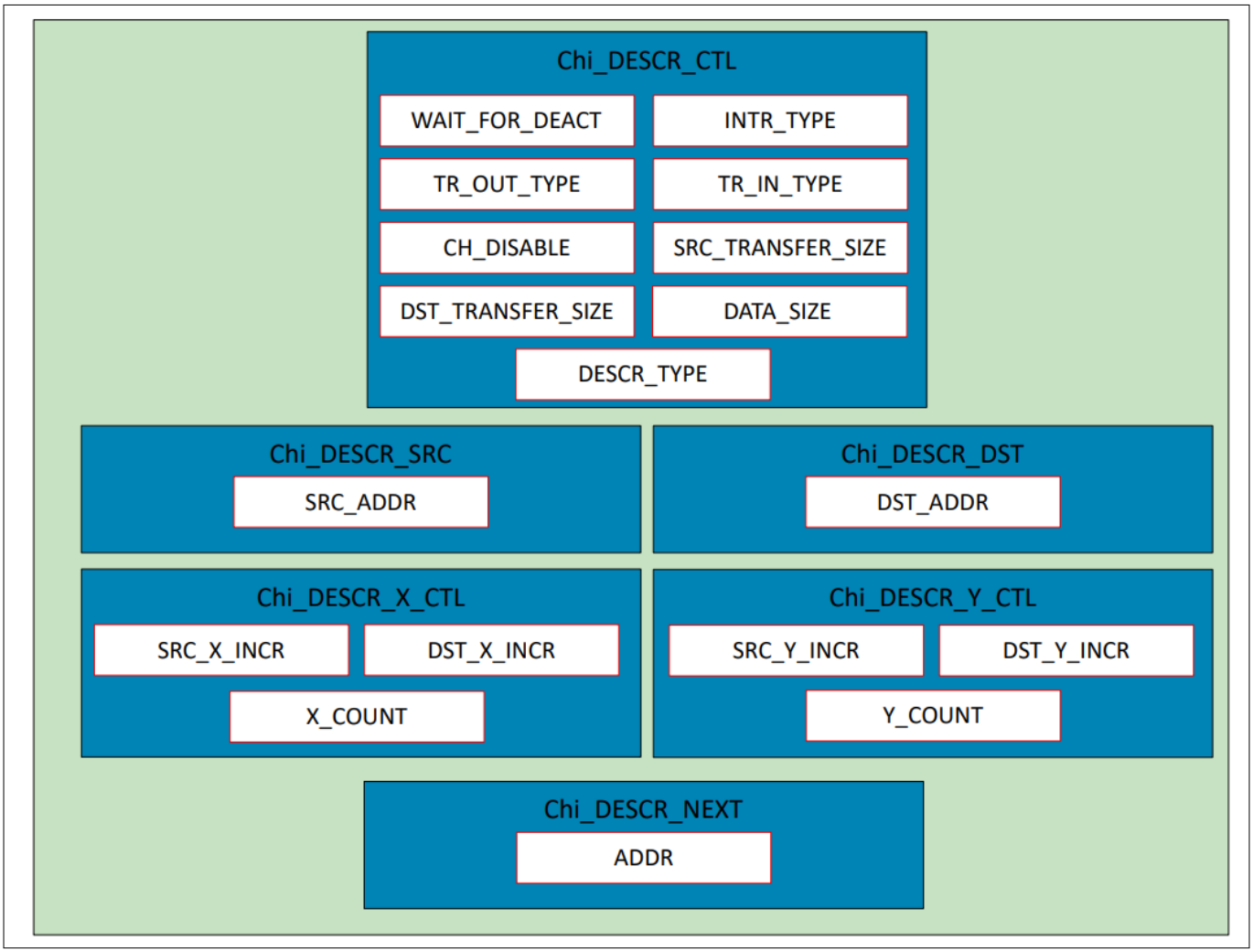


Figure 21 Descriptor layout

Descriptor control (DESCR_CTL)

The descriptor control register is divided into seven controls that are required for all transfer types:

- **WAIT_FOR_DEACT**: Specifies whether the DMA controller waits for the input trigger to deactivate after completing 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 duration can be configured as no wait (pulse-sensitive triggers), up to 4 cycles, up to 16 cycles, or indefinite. For more information, see Trigger MUX
- **INTR_TYPE**: Defines when a completion interrupt is generated. The interrupt is triggered in CHi_INTR. For more information, see Channel interrupt generation
- **TR_OUT_TYPE**: Defines when an output trigger is generated
- **TR_IN_TYPE**: Specifies the action taken on an input trigger that initiates a data transfer
- **CH_DISABLE**: Specifies whether the channel is disabled after completing the current descriptor. This is independent of the value of DESCR_NEXT_PTR
- **SRC_TRANSFER_SIZE**: Specifies the bus transfer size to the source location. It can be set to match the data size specified by DATA_SIZE or fixed at 32 bits
- **DST_TRANSFER_SIZE**: Specifies the bus transfer size to the destination location. It can be set to match the data size specified by DATA_SIZE or fixed at 32 bits

- **DATA_SIZE**: Specifies the data element size when SRC_TRANSFER_SIZE or DST_TRANSFER_SIZE uses DATA_SIZE. The data element size can be set to 8 bits, 16 bits, or 32 bits
- **DESCR_TYPE**: Specifies the transfer type for the descriptor. The DMA supports single, 1D, 2D, and CRC transfers. For more information, see Descriptor memory transfer types

Source and destination address (DESCR_SRC/DESCR_DST)

The source and destination addresses are set in their respective registers within the descriptor. These specify the base addresses for the source and destination locations for the transfer.

- For a single element transfer, the descriptor holds the source/destination address of the data element
- For multiple elements with incremental source or destination addresses (or both), this field holds the address of the first element to be transferred
- For CRC transfers, the field must be programmed with the address of the DW_CRC_LFSR_CTL register. The calculated CRC LFSR state is written to this address using the AHB interface

Descriptor X loop control (DESCR_X_CTL)

The DESCR_X_CTL field is used when 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 value in the range [-2048, 2047]. A value of 0 means no increment, which 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 value in the range [-2048, 2047]. A value of 0 means no increment, which is useful for writing to TX FIFO structures. This field is not used if DESCR_TYPE is set to CRC
- **X_COUNT**: Specifies the number of iterations minus one for the X loop (X_COUNT + 1 equals the number of single transfers in a 1D transfer). This field is an unsigned value in the range [0, 255], representing 1 to 256 iterations

Descriptor Y loop control (DESCR_Y_CTL)

The DESCR_Y_CTL field is used when 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 value 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 value in the range [-2048, 2047]
- **Y_COUNT**: Specifies the number of iterations minus one for the Y loop ((X_COUNT + 1) * (Y_COUNT + 1) equals the total number of single transfers in a 2D transfer). This field is an unsigned value in the range [0, 255], representing 1 to 256 iterations

Descriptor next pointer (DESCR_NEXT_PTR)

The next pointer specifies the address of the next descriptor in the descriptor list:

- If this field is set to 0, the current descriptor is the last descriptor in the 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 divided into source transfer size, destination transfer size, and data size. The data size parameter (DESCR_CTL.DATA_SIZE) defines the width of the bus for the transfer. The source and destination transfer sizes, set by DESCR_CTL.SRC_TRANSFER_SIZE and DESCR_CTL.DST_TRANSFER_SIZE, can be configured as either DATA_SIZE or 32-bit. The DESCR_CTL.DATA_SIZE parameter can have a 32-bit, 16-bit, or 8-bit setting.

The source and destination transfer sizes must match the addressable width of the source and destination, regardless of the width of the actual data being transferred. 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-bit to match the width of the PWM register. This is 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 can still be set to 16-bit because the 16-bit PWM only uses two bytes of data.

Table 20 summarizes the possible combinations of transfer sizes.

Table 20 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-bitzero 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, this is referred to as a descriptor list. The DESCR_NEXT_PTR field contains a pointer to the next descriptor in the chain. This pointer must be aligned to a multiple of 8 bytes.

A channel executes the next descriptor in the chain after completing the current descriptor. For the last descriptor in the chain, the DESCR_NEXT_PTR field is set to 0 (NULL pointer).

It is also possible to create a circular list, where the descriptors form a loop by pointing back to earlier descriptors in the chain. In this case, execution continues indefinitely until an error occurs, or the channel or controller is disabled by software.

9.5 Triggers

Each DMA channel has one input trigger and one output trigger. The input trigger is used to start a data transfer, while the output trigger signals another peripheral at the end of the transfer. Both input and output triggers are configured for each descriptor.

9.5.1 Input triggers

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

Each channel connects to a specific system trigger through a trigger multiplexer located outside the DMA controller. The trigger multiplexer routes trigger signals to their specified destinations, with the DMA being one such destination. For more information, see Trigger MUX.

The DMA controls the behavior of trigger inputs using the `TR_IN_TYPE` and `WAIT_FOR_DEACT` fields, both of which are 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 taken by each channel when triggered. The different trigger types are listed in Table 21

Table 21 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 when the descriptor type is set to "1D" or "2D."
Type 2	A trigger results in the execution of the current descriptor and completes the transfer.
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, based on the configuration of the next descriptor's information.

`WAIT_FOR_DEACT`: The DMA provides controls for managing input trigger deactivation. When a DMA transfer, based on the `TR_IN_TYPE`, is completed, the data transfer engine checks the state of the trigger to ensure it has been deactivated. A second data transfer is initiated only after the first trigger has been deactivated.

The `DESCR_CTL.WAIT_FOR_DEACT` parameter determines when the trigger signal is considered deactivated. The first DMA transfer is initiated when the trigger is activated, but the transfer is not considered complete until the trigger is deactivated. This field is used to synchronize the DMA controller’s data transfers with the agent that generated the trigger.

The different trigger deactivation options are listed in Table 22.

Table 22 Trigger deactivation types

Deactivation types	Description
Type 0	Pulse Trigger (No Wait): • Does not wait for deactivation • When a trigger is detected, the transfer is immediately initiated • After completing the transfer, if the trigger remains active, it is treated as another trigger, and the subsequent transfer is immediately initiated
Type 1	Level-Sensitive (4-cycle Wait): • Waits for four slow clock cycles after the transfer to consider the trigger as deactivated • When a trigger is detected, the transfer is initiated • After completing the transfer, if the trigger is still active, it is treated as another trigger after waiting for four clock cycles • The transfer corresponding to the trigger is considered complete only at the end of the additional four cycles • Trigger output events are affected by this delay, resulting in up to a four-cycle delay for each trigger transaction, which impacts throughput
Type 2	Level-Sensitive (16-cycle Wait): • Waits for 16 slow clock cycles after the transfer to consider the trigger as deactivated • When a trigger is detected, the transfer is initiated • After completing the transfer, if the trigger is still active, it is treated as another trigger after waiting for 16 clock cycles • The transfer corresponding to the trigger is considered complete only at the end of the additional 16 cycles • Trigger output events are affected by this delay, resulting in up to a 16-cycle delay for each trigger transaction, which impacts throughput

(table continues...)

Table 22 (continued) Trigger deactivation types

Deactivation types	Description
Type 3	Pulse Trigger (Indefinite Wait): • Waits indefinitely for the trigger to deactivate • The DMA transfer is initiated only after the trigger signal deactivates • The next transfer is initiated only if the trigger transitions to low and then back to high • If the trigger remains active or does not transition to zero between two transactions, the DMA channel will stall

A DMA transfer can also be triggered by software. Writing a `1` to the `CHi_TR_CMD` register generates a trigger, setting a trigger pending on the channel. The channel must be enabled for the trigger to take effect.

9.5.2 Pending triggers

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

If multiple channels are in a pending state, the DMA uses priority decoding to determine the highest-priority pending channel. The channel selection process is as follows:

1. The highest-priority group with pending channels is identified
2. Within this priority group, round-robin arbitration is applied, giving higher priority to channels with lower channel indices (within the priority group)

The channel priority can be configured using the `CHi_CTL.PRIO` parameter.

9.5.3 Output triggers

Each DMA channel has an output trigger that remains high for at least two slow clock cycles. This trigger is generated upon 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, which is located outside the DMA block (see Trigger MUX), enables a DMA controller's output trigger to be connected to a DMA controller's input trigger. This connection allows the completion of a transfer in one channel to activate another channel or even reactivate the same channel.

`TR_OUT_TYPE`: The output triggers for DMA channels are configured using the `CHi_DESCR_CTL.TR_OUT_TYPE` field, which defines when an output trigger is generated. The different trigger types are listed in Table 23.

Table 23 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 1Dtransfer. 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 1Dtransfer.

(table continues...)

Table 23 (continued) Output trigger types

Trigger type	Description
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 for a DMA transfer can be calculated based on the transfer type specified in DESCR_CTL.DESCR_TYPE.

9.6.1 Single transfer

The DMA controller requires 14 clock cycles (“clk”) to perform a single data transfer, provided there are no wait states on the bus infrastructure for loading descriptors or loading/storing data elements. The breakdown of the clock cycles is as follows:

- **2 cycles**: Trigger synchronization
- **1 cycle**: Detecting the trigger, performing priority decoding, and setting the channel to pending
- **1 cycle**: Controller state machine detects the pending channel and starts the transfer
- **2 cycles**: Loading descriptor channel control information from DW channel configuration memory to DW control flip-flops
- **4 cycles**: Loading the descriptors (DESCR_CTL, DESCR_SRC, and DESCR_DST)
- **1 cycle**: Loading the next pointer (DESCR_NEXT_PTR)
- **3 cycles**: Loading the data from the source and storing the data at the destination

9.6.2 1D transfer

The formula for calculating the number of clock cycles required to complete a 1D transfer of “n” data elements is as follows:

Number of cycles = 12 + n*3 + m

Where:

- **n**: Total number of data elements being transferred
- **m**: Total number of wait states encountered by the DW controller while loading/storing descriptors or data

Breakdown of the fixed 12 cycles in the formula:

- **2 cycles**: Trigger synchronization
- **1 cycle**: Detecting the trigger, performing priority decoding, and setting the channel to pending
- **1 cycle**: Controller state machine detects the pending channel and starts the transfer
- **2 cycles**: Loading descriptor channel control information from DW channel configuration memory to DW control flip-flops
- **4 cycles**: Loading the descriptors (DESCR_CTL, DESCR_SRC, and DESCR_DST)
- **1 cycle**: Loading descriptor X control (DESCR_X_CTL)
- **1 cycle**: Loading the next pointer (DESCR_NEXT_PTR)

Each data element transferred from the source to the destination takes 3 cycles. This is represented by the term n*3 in the formula.

9.6.3 2D transfer

The formula for calculating the number of clock cycles required to complete a 2D transfer of "n" data elements is as follows:

Number of cycles = 13 + n*3 + m

Where:

- **n**: Total number of data elements being transferred
- **m**: Total number of wait states encountered by the DW controller while loading/storing descriptors or data

Breakdown of the fixed 13 cycles in the formula:

- **2 cycles**: Trigger synchronization
- **1 cycle**: Detecting the trigger, performing priority decoding, and setting the channel to pending
- **1 cycle**: Controller state machine detects the pending channel and starts the transfer
- **2 cycles**: Loading descriptor channel control information from DW channel configuration memory to DW control flip-flops
- **4 cycles**: Loading the descriptors (DESCR_CTL, DESCR_SRC, and DESCR_DST)
- **1 cycle**: Loading descriptor X control (DESCR_X_CTL)
- **1 cycle**: Loading descriptor Y control (DESCR_Y_CTL)
- **1 cycle**: Loading the next pointer (DESCR_NEXT_PTR)

Each data element transferred from the source to the destination takes 3 cycles. This is represented by the term n*3 in the formula.

9.6.4 CRC transfer

The formula for calculating the number of clock cycles required to complete a CRC computation for "n" data words is as follows:

Number of cycles = 17 + n*9 + m

Where:

- **n**: Total number of data words for the CRC computation
- **m**: Total number of wait states encountered by the DW controller while loading/storing descriptors or data
- The CRC computation for each data word takes 9 clock cycles, which is represented by the term n*9 in the formula (assuming the CRC descriptor configuration uses word transfers)
- The extra 17 cycles account for:
  - Trigger synchronization
  - Trigger detection
  - Loading of the descriptor
  - Storing the computed CRC value back to the destination

10 Trigger MUX

Every peripheral in CYW20829 is interconnected using trigger signals. Trigger signals are used by peripherals to indicate the occurrence of an event or a transition to a specific state.

These triggers are used to initiate or effect an action in other peripherals. Trigger multiplexers allow users 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 operations between peripherals rather than between a peripheral and the Arm® CPU core.

10.1 Features

- Ability to connect any trigger signal from one peripheral to another
- Two-layer architecture with 9 trigger groups
- Supports a software trigger that can activate any signal in the block
- Supports multiplexing of triggers between peripherals
- One-to-one trigger paths for dedicated triggers that are commonly routed
- Configurable trigger multiplexers with hardware-based trigger manipulation features such as inversion and edge/level detection
- Ability to block triggers during debug mode

10.2 Architecture

Triggers are used to synchronize the functionality of peripherals in hardware, as opposed to software-based synchronization. Peripheral DMA utilizes triggers to initiate the transfer of data elements from one address location to another. For example, an "ADC conversion done" event initiates the transfer of an ADC sample from the ADC module to an SRAM memory location.

Triggers are digital signals generated by peripheral blocks to represent a state, such as a FIFO level, or an event, such as the completion of an action. These trigger signals typically act as initiators for subsequent actions in other peripheral blocks. For instance, consider an ADC peripheral block sampling three channels. After completing the conversion, it generates a trigger signal, which subsequently activates a DMA channel to transfer the ADC data to a memory buffer. This process is shown in Figure 22.

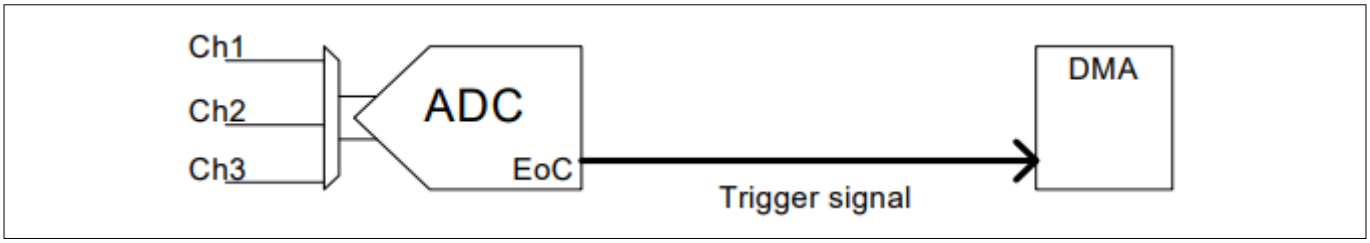


Figure 22 Trigger signal example

The CYW20829 microcontroller includes multiple peripheral blocks, each capable of connecting to other blocks through trigger signals, based on the system's implementation. To enable this interconnection, the hardware features a series of multiplexers designed to route trigger signals from potential sources to designated destinations. This hardware is referred to as the trigger multiplexer block.

The trigger multiplexer block allows any trigger signal generated by a peripheral block within the MCU to be routed to another peripheral block. This routing facilitates the initiation or modification of operations at the destination peripheral block, enabling seamless synchronization and coordination between peripherals.

Triggers are categorized into two types:

1. **High-active, level-sensitive triggers**: This type typically represents a peripheral state. For example, the `tr_fifo_empty` trigger indicates that a FIFO is empty. The trigger remains at logic '1' as long as the FIFO is empty. Deactivating this trigger requires an action by the consumer of the trigger, which, in turn, causes the producer to deactivate it. For instance, writing a data element to the associated FIFO deactivates the `tr_fifo_empty` trigger. These triggers can be generated on any clock
2. **Rising-edge, pulse triggers**: This type usually reflects the occurrence of an event. For example, the `tr_adc_done` trigger signifies that a SAR ADC has completed converting a sample. A rising-edge pulse is generated on the peripheral system interface clock. The trigger remains at logic '1' for a two-cycle pulse on the peripheral system interface clock and then automatically returns to logic '0'

If the consumer of a trigger cannot immediately react to it, it must be able to remember that the trigger occurred. If multiple triggers are generated in quick succession, the consumer may need to handle multiple triggers or choose to miss some of them.

Triggers are asynchronous signals between the producer and consumer peripherals. The consumer peripheral must synchronize input triggers to ensure proper operation. Additionally, for pulse triggers, the consumer peripheral can perform rising-edge detection and store the occurrence of the trigger. Treating triggers as asynchronous signals simplifies timing closure.

At a high level, a trigger is a wire connection between a producer and a consumer peripheral. However, at a more detailed level, triggers involve several processing steps. From a platform perspective, it is important to ensure these steps are implemented consistently. The processing steps for input triggers include trigger multiplexing, synchronization, edge detection, and storage. There is also a distinct processing step for output triggers.

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

The decision to use standard multiplexer components as part of peripheral groups, rather than integrating multiplexer components directly into peripherals, is based on the following rationale:

- A standard multiplexer component ensures a uniform user interface. The process of selecting a trigger input signal for a specific trigger output signal (peripheral input signal) remains the same across all components, promoting ease of use and consistency
- Any additional features offered by the multiplexer components, such as software control over output trigger signal activation, become available to all peripherals. This approach provides shared benefits and avoids the need to replicate such functionality within individual peripherals

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

- **Synchronization**: Synchronize the trigger signal to the peripheral clock domain to ensure proper timing alignment
- **Edge detection**: Detect edges on the trigger signal, such as rising or falling edges, to identify specific events
- **Storage**: Store or remember the trigger signal to handle situations where immediate processing is not possible

A trigger component consists of multiple trigger groups, which can be categorized into two types:

1. **One-to-one connectivity group**: This type of trigger group connects a peripheral input trigger to a specific peripheral output trigger. It establishes a direct one-to-one relationship between the input and output signals
2. **Multiplexer-based connectivity group**: This type of trigger group connects a peripheral input trigger to multiple peripheral output triggers. The selection of the output trigger is controlled through software using the `PERI_TR_GR_TR_CTL.TR_SEL` register

In general, a trigger component can provide up to 16 multiplexer-based trigger groups and up to 16 one-to-one trigger groups. However, the exact number of trigger groups varies depending on the specific device. See the device datasheet for more detailed information.

10.3 Functional description

The trigger component is responsible for multiplexing trigger signals. Trigger input signals typically originate from peripheral output signals, while trigger output signals are usually peripheral input signals.

Examples of trigger input signals include:

- **TCPWM output signals**: For example, a counter reaching a pre-programmed limit
- **ADC output signals**: For example, the completion of an ADC conversion
- **I/O input signals**: Signals generated by external inputs
- **DMA controller output signals**: Indications that a data transfer has completed

Examples of trigger output signals include:

- **TCPWM input signals**: For example, "start a counter"
- **ADC input signals**: For example, "start an ADC conversion"
- **I/O output signals**: Signals sent to external components
- **DMA controller input signals**: Signals to start a data transfer

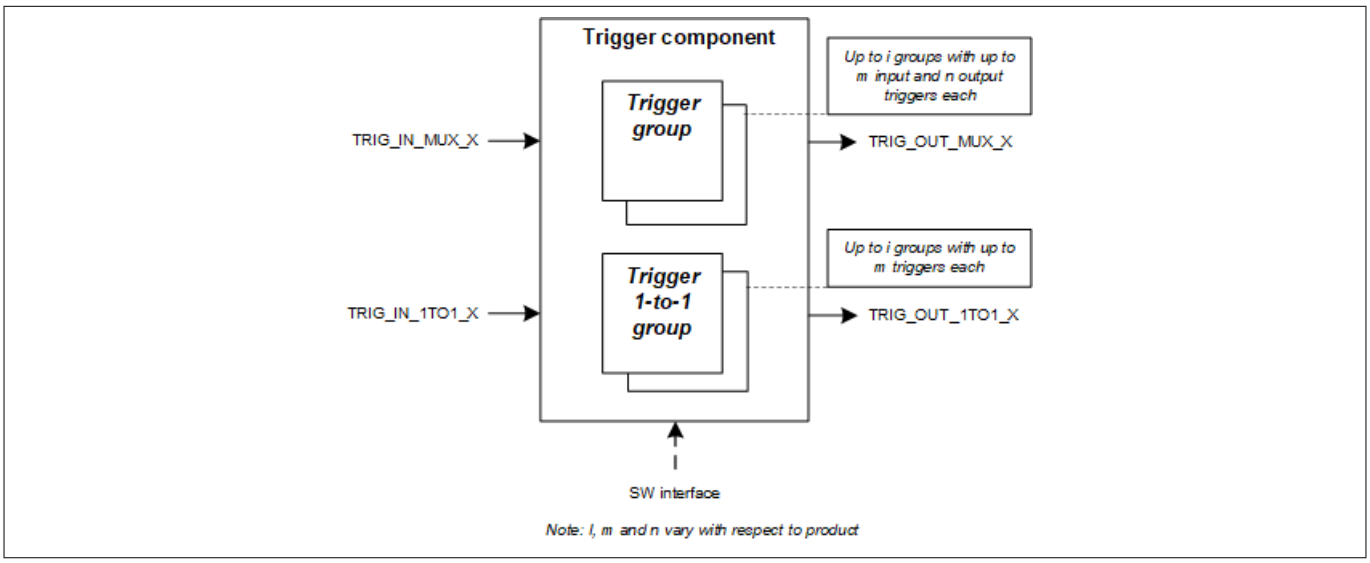


Figure 23 Trigger configuration parameters

Each trigger group is associated with the trigger inputs of a specific peripheral, as shown in Figure 23.

- **Trigger destination peripherals**: Are connected to trigger source peripheral components through `TRIG_IN_MUX_x` and `TRIG_IN_1TO1_x`
- **Trigger source peripherals**: Are connected to trigger destination peripheral components through `TRIG_OUT_MUX_x` and `TRIG_OUT_1TO1_x`

It is crucial to distinguish between the functionality provided by the trigger component in the PERI block and the functionality provided by the peripherals themselves. The trigger component primarily handles the routing and multiplexing of trigger signals, while the peripherals process these signals to perform their specific actions.

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

1. **Multiplexer-based connectivity group**: For a multiplexer-based group, an input trigger (`TRIG_IN_MUX_x`) is selected for each output trigger (`TRIG_OUT_MUX_x`). All output triggers in a group share the same input trigger
2. **One-to-one connectivity group**: For a one-to-one-based group, an input trigger (`TRIG_IN_1TO1_x`) is directly connected to a specific output trigger (`TRIG_OUT_1TO1_x`)

3. **Software-controlled trigger activation**: Software control is provided for trigger activation. For level-sensitive triggers, activation is entirely under software control. For edge-sensitive triggers, activation corresponds to two high (1) cycles on CLK_PERI
4. **Trigger blocking in debug mode**: Trigger propagation can be blocked in debug mode, typically when a CPU is halted. Blocking triggers prevents the consumer peripheral from receiving input triggers during debug operations. Debug mode is indicated by the x_DEBUG_FREEZE_TR_IN1 level-trigger input, which is connected to a CPUSS CTI trigger output

***Note**: The number 1 represents specific peripherals, such as:TCPWM, PASS, PERI, SRSS_WDT, SRSS_MCWDT, etc.*

Hardware edge-detection functionality is provided to support pulse triggers transferring to the output trigger clock domain. This hardware performs asynchronous edge detection, allowing input triggers to operate on a higher clock frequency than the output trigger clock domain. This functionality is specifically intended for pulse triggers, as level-sensitive triggers typically bypass the edge-detection mechanism.

Hardware provides the capability to invert trigger signal levels when needed. Most trigger multiplexers connect all output signals to a common peripheral, with the manipulation logic tied to the clock for that peripheral. However, the debug multiplexer handles outputs across multiple clock domains. This is acceptable because most destinations are level-sensitive signals, I/Os, or other debug-related destinations that may not require clock manipulations.

A peripheral may provide the following functionalities for handling triggers:

- **Synchronization**: Synchronization of output triggers (TRIG_OUT_MUX_x and TRIG_OUT_1TO1_x) to the appropriate clock domain
- **Edge detection**: Detection of edges (rising or falling) on synchronized output triggers
- **Storage/remembering**: Storing or remembering the trigger signal for further processing when immediate action is not possible

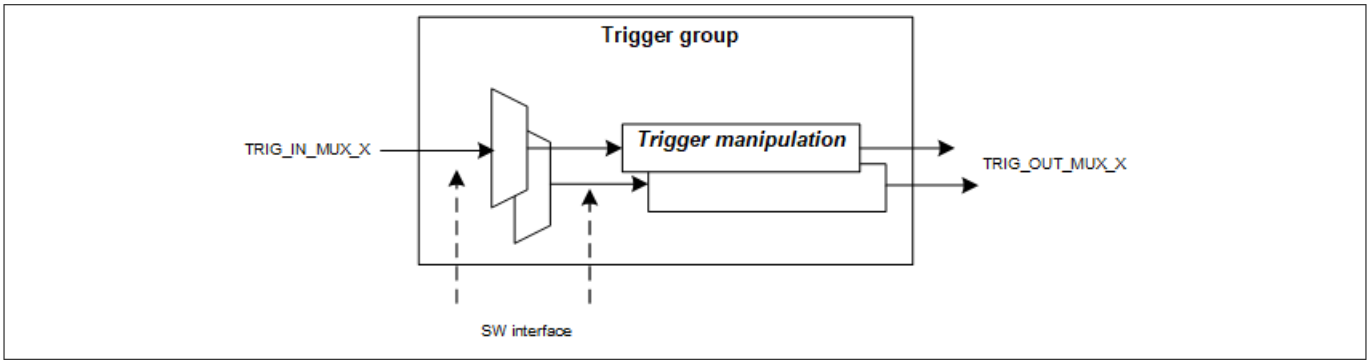


Figure 24 Trigger group

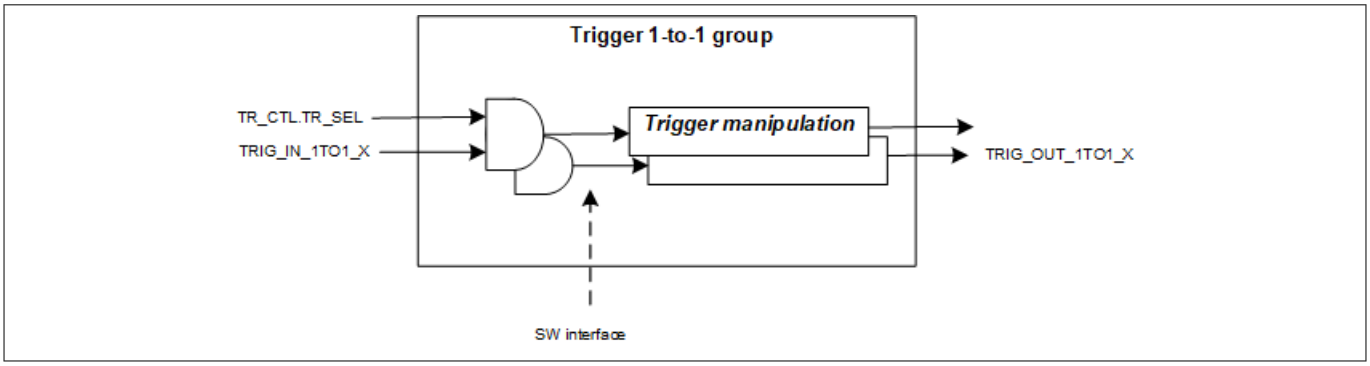


Figure 25 Trigger one-to-one group

***Note**: A one-to-one group includes AND-gate functionality to disable an input trigger.*

An overview of the trigger multiplexer connections for CYW20829 is provided in Table 24.

Table 24 Trigger MUX connections overview

In	Trigger MUX	Out
CPUSS.dw0_tr_out[0:3]	PDMA0	DW0 - 16 channels
TCPWM.tr_out:0		
IOSS[0:7]:0		
CPUSS.cti_tr_out:0		
ADCMIC.tr_adcmic:0		
TDM:0		
PDM:0		
CRYPTO.tr_trng_bitstream:0		
CPUSS.dw0_tr_out[0:7]	TCPWM0 Trigger	TCPWM.tr_all [0:13]
TCPWM.tr_out:1		
SCB:0		
SMIF:0		
TDM:2		
PDM [all]		
IOSS [0:3]		
CPUSS.cti_tr_out:1		
ADCMIC.tr_adcmic:1		
CANFD.tmp_rtp_out:0		
BTSS:0		
CPUSS.dw0_tr_out[8:15]	TCPWM1 Trigger	TCPWM.tr_all [14:27]
TCPWM:2		
SCB:1		
SMIF:1		
TDM:3		
PDM [all]		
IOSS [4:7]		
CPUSS.cti_tr_out:2		
ADCMIC.tr_adcmic:2		
CANFD.tmp_rtp_out:1		
BTSS:1		
CPUSS.dw0_tr_out[0:7]	HSIOM 0 Trigger	HSIOM_TR_IO_OUTPUT[0:1]

(table continues...)

Table 24 (continued) Trigger MUX connections overview

In	Trigger MUX	Out
TCPWM:3		
SCB:2		
TDM:4		
PDM [all]		
CPUSS.cti_tr_out:3		
ADCMIC.tr_adcmic:3		
CANFD[0].tr_tmp_rtp_out:2		
CPUSS.dw0_tr_out[0:15]	CPUSS_CTI Trigger	CPUSS_CTI_TR_IN[0:1] SRSS_MCWDT_DEBUG_FREEZE_TR_IN[0]
TCPWM:4		
SCB:3		
CRYPTO.tr_trng_bitstream:4		
SMIF: 2		
TDM:1		
PDM [all]		
IOSS [0:7]		
CPUSS.cti_tr_out:4		
ADCMIC.tr_adcmic:4		
CANFD[0].tr_tmp_rtp_out:3		
CPUSS.cti_tr_out:5	PERI_FREEZE Trigger	PERI_DEBUG_FREEZE_TR_IN PDM_DEBUG_FREEZE_TR_IN TDM_DEBUG_FREEZE_TR_IN TCPWM_DEBUG_FREEZE_TR_IN
TCPWM:5	TCPWM_PDM Trigger	PDM_TR_ACTIVATE[0:1]
TCPWM:0		
CANFD[0].tr_tmp_rtp_out[0]:4	CAN TT Sync	CAN_TT_TR_IN[0]
SCB[0].tr_tx_req:4	SCB_CAN0	PDMA0_TR_IN[4]
CANFD[0].tr_dbg_dma_req[0]:0		
SCB[0].tr_rx_req:4	SCB_CAN1	PDMA0_TR_IN[5]
CANFD[0].tr_fifo0[0]:0		

***Note**: In this section, the DMA is referred to as the Data Wire (DW) in both register names and trigger references.*

The detailed trigger multiplexer (MUX) connections for CYW20829 are provided in Table 25 to Table 34.

Table 25 PERI trigger source Group 0 (PDMA0 TriggssssserMux)

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	TCPWM0_GRP0_OUT0_[0]	-
6	TCPWM0_GRP0_OUT1_[0]	-
7	TCPWM0_GRP0_OUT0_[1]	-
8	TCPWM0_GRP0_OUT1_[1]	-
9:16	HSIOM_TR_OUT[0:7]	Triggers connected to pins in HSIOM (peri.tr_in)
17:18	CTI_TR_OUT[0:1]	CPUSS debug events
19	ADCMIC_DC_DONE	ADC DC done trigger
20	ADCMIC_DATA_AVAIL	ADC Data available trigger
21	I2S_TDM_TX[0]	I2S triggers
22	I2S_TDM_RX[0]	-
23	PDM_RX[0]	-
24	PDM_RX[1]	-
25	CRYPTO_TR_TRNG	-

Table 26 PERI trigger destination Group 1 (TCPWM0 TriggerMux)

Trigger MUX output	Trigger destination label	Description
0	ZERO	-
1:8	PDMA0_TR_OUT[0:7]	PDMA0 tr_out
9	PDMA0_TR_OUT[0:7]	lower 8 of 16-bit TCPWM counters to TCPWM0
10	TCPWM0_TR_OUT1[0]	-
11	TCPWM0_TR_OUT0[1]	-
12	TCPWM0_TR_OUT1[1]	-
13	TCPWM0_TR_OUT0[256]	lower 8 of 16-bit TCPWM counters to TCPWM0
14	TCPWM0_TR_OUT1[256]	-
15	TCPWM0_TR_OUT0[257]	-
16	TCPWM0_TR_OUT1[257]	-
17	TCPWM0_TR_OUT0[258]	-
18	TCPWM0_TR_OUT1[258]	-
19	TCPWM0_TR_OUT0[259]	-

(table continues...)

Table 26 (continued) PERI trigger destination Group 1 (TCPWM0 TriggerMux)

Trigger MUX output	Trigger destination label	Description
20	TCPWM0_TR_OUT1[259]	-
21	TCPWM0_TR_OUT0[260]	-
22	TCPWM0_TR_OUT1[260]	-
23	TCPWM0_TR_OUT0[261]	-
24	TCPWM0_TR_OUT1[261]	-
25	TCPWM0_TR_OUT0[262]	-
26	TCPWM0_TR_OUT1[262]	-
27	SCB_I2C_SCL[0]	-
28	SCB_TX[0]	SCB TX/RX DMA requests
29	SCB_RX[0]	-
30	CRYPTO_TR_TRNG	-
31	SCB_TX[1]	-
32	SCB_RX[1]	-
33	SCB_I2C_SCL[2]	-
34	SCB_TX[2]	-
35	SCB_RX[2]	-
36	SMIF_TX	SMIF triggers
37	SMIF_RX	-
38	I2S_TDM_TX[0]	I2S triggers
39	I2S_TDM_RX[0]	-
40	PDM_RX[0]	-
41	PDM_RX[1]	-
42	PDM_RX_REQ_ALL	-
43:46	HSIOM_TR_OUT[0:3]	Triggers connected to pins in HSIOM (peri.tr_in)
47:48	CTI_TR_OUT[0:1]	Debug events
49	ADCMIC_DC_DONE	ADC dc done trigger
50	ADCMIC_DATA_AVAIL	ADC Data avaiable trigger
51	CANFD_TT_TR_OUT[0]	CAN TT Sync Outputs
52	BTSS_TR_TX	-
53	BTSS_TR_RX	-

Table 27 PERI trigger source Group 2 (TCPWM1 TriggerMux)

Trigger MUX input	Trigger source label	Description
0	ZERO	-
1:8	PDMA0_TR_OUT[8:15]	PDMA0 tr_out
9	TCPWM0_TR_OUT0[0]	Lower 8 of 16-bit TCPWM counters to TCPWM0
10	TCPWM0_TR_OUT1[0]	-
11	TCPWM0_TR_OUT0[1]	-
12	TCPWM0_TR_OUT1[1]	-
13	TCPWM0_TR_OUT0[256]	Lower 8 of 16-bit TCPWM counters to TCPWM0
14	TCPWM0_TR_OUT1[256]	-
15	TCPWM0_TR_OUT0[257]	-
16	TCPWM0_TR_OUT1[257]	-
17	TCPWM0_TR_OUT0[258]	-
18	TCPWM0_TR_OUT1[258]	-
19	TCPWM0_TR_OUT0[259]	-
20	TCPWM0_TR_OUT1[259]	-
21	TCPWM0_TR_OUT0[260]	-
22	TCPWM0_TR_OUT1[260]	-
23	TCPWM0_TR_OUT0[261]	-
24	TCPWM0_TR_OUT1[261]	-
25	TCPWM0_TR_OUT0[262]	-
26	TCPWM0_TR_OUT1[262]	-
27	SCB_I2C_SCL[0]	SCB TX/RX DMA requests
28	SCB_TX[0]	-
29	SCB_RX[0]	-
30	CRYPTO_TR_TRNG	-
31	SCB_TX[1]	-
32	SCB_RX[1]	-
33	SCB_I2C_SCL[2]	-
34	SCB_TX[2]	-
35	SCB_RX[2]	-
36	SMIF_TX	SMIF triggers
37	SMIF_RX	-
38	I2S_TDM_TX[0]	I2S triggers

(table continues...)

Table 27 (continued) PERI trigger source Group 2 (TCPWM1 TriggerMux)

Trigger MUX input	Trigger source label	Description
39	I2S_TDM_RX[0]	-
40	PDM_RX[0]	-
41	PDM_RX[1]	-
42	PDM_RX_REQ_ALL	-
43:46	HSIOM_TR_OUT[4:7]	Triggers connected to pins in HSIOM (peri.tr_in)
47:48	CTI_TR_OUT[0:1]	Debug events
49	ADCMIC_DC_DONE	ADC dc done trigger
50	ADCMIC_DATA_AVAIL	ADC Data available trigger
51	CANFD_TT_TR_OUT[0]	CAN TT Sync Outputs
52	BTSS_TR_TX	-
53	BTSS_TR_RX	-

Table 28 PERI trigger source Group 3 (HSIOM TriggerMux)

Trigger MUX input	Trigger source label	Description
0	ZERO	-
1:8	PDMA0_TR_OUT[0:7]	PDMA0 tr_out
9	TCPWM0_TR_OUT0[0]	The 32-bit TCPWM counters
10	TCPWM0_TR_OUT1[0]	-
11	TCPWM0_TR_OUT0[1]	-
12	TCPWM0_TR_OUT1[1]	-
13	TCPWM0_TR_OUT0[256]	The 16-bit TCPWM counters
14	TCPWM0_TR_OUT1[256]	-
15	TCPWM0_TR_OUT0[257]	-
16	TCPWM0_TR_OUT1[257]	-
17	TCPWM0_TR_OUT0[258]	-
18	TCPWM0_TR_OUT1[258]	-
19	TCPWM0_TR_OUT0[259]	-
20	TCPWM0_TR_OUT1[259]	-
21	TCPWM0_TR_OUT0[260]	-
22	TCPWM0_TR_OUT1[260]	-
23	TCPWM0_TR_OUT0[261]	-
24	TCPWM0_TR_OUT1[261]	-
25	TCPWM0_TR_OUT0[262]	-

(table continues...)

Table 28 (continued) PERI trigger source Group 3 (HSIOM TriggerMux)

Trigger MUX input	Trigger source label	Description
26	TCPWM0_TR_OUT1[262]	-
27	SCB_I2C_SCL[0]	-
28	SCB_TX[0]	-
29	SCB_RX[0]	-
30	CRYPTO_TR_TRNG	-
31	SCB_TX[1]	-
32	SCB_RX[1]	-
33	SCB_I2C_SCL[2]	-
34	SCB_TX[2]	-
35	SCB_RX[2]	-
36	I2S_TDM_TX[0]	-
37	I2S_TDM_RX[0]	-
38	PDM_RX[0]	-
39	PDM_RX[1]	-
40	PDM_RX_REQ_ALL	-
41:42	CTI_TR_OUT[0:1]	-
43	ADCMIC_DC_DONE	-
44	ADCMIC_DATA_AVAIL	-
45	CANFD_TT_TR_OUT[0]	-

Table 29 PERI trigger destination Group 4 (CPUSS_CTI TriggerMux)

Trigger MUX output	Trigger destination label	Description
0	ZERO	-
1:16	PDMA0_TR_OUT[0:15]	P-DMA0 tr_out
17	TCPWM0_TR_OUT0[0]	The 32-bit TCPWM counters
18	TCPWM0_TR_OUT1[0]	-
19	TCPWM0_TR_OUT0[1]	-
20	TCPWM0_TR_OUT1[1]	-
21	TCPWM0_TR_OUT0[256]	The 16-bit TCPWM counters
22	TCPWM0_TR_OUT1[256]	-
23	TCPWM0_TR_OUT0[257]	-
24	TCPWM0_TR_OUT1[257]	-
25	TCPWM0_TR_OUT0[258]	-
26	TCPWM0_TR_OUT1[258]	-

(table continues...)

Table 29 (continued) PERI trigger destination Group 4 (CPUSS_CTI TriggerMux)

Trigger MUX output	Trigger destination label	Description
27	TCPWM0_TR_OUT0[259]	-
28	TCPWM0_TR_OUT1[259]	-
29	TCPWM0_TR_OUT0[260]	-
30	TCPWM0_TR_OUT1[260]	-
31	TCPWM0_TR_OUT0[261]	-
32	TCPWM0_TR_OUT1[261]	-
33	TCPWM0_TR_OUT0[262]	-
34	TCPWM0_TR_OUT1[262]	-
35	SCB_I2C_SCL[0]	SCB TX/RX DMA requests
36	SCB_TX[0]	-
37	SCB_RX[0]	-
38	CRYPTO_TR_TRNG	-
39	SCB_TX[1]	-
40	SCB_RX[1]	-
41	SCB_I2C_SCL[2]	-
42	SCB_TX[2]	-
43	SCB_RX[2]	-
44	SMIF_TX	SMIF triggers
45	SMIF_RX	-
46	I2S_TDM_TX[0]	I2S triggers
47	I2S_TDM_RX[0]	-
48	PDM_RX[0]	-
49	PDM_RX[1]	-
50	PDM_RX_REQ_ALL	-
51:58	HSIOM_TR_OUT[0:7]	Triggers connected to pins in HSIOM (peri.tr_in)
59:60	CTI_TR_OUT[0:1]	Debug events
61	ADCMIC_DC_DONE	ADC dc done trigger
62	ADCMIC_DATA_AVAIL	ADC Data available trigger
63	CANFD_TT_TR_OUT[0]	CAN TT Sync Outputs

Table 30 PERI trigger source Group 5 (PERI_FREEZE TriggerMux)

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

Table 31 PERI trigger destination Group 6 (TCPWM_PDM TriggerMux)

Trigger MUX output	Trigger destination label	Description
0	ZERO	-
1	TCPWM0_TR_OUT0[0]	The 32-bit TCPWM counters
2	TCPWM0_TR_OUT1[0]	-
3	TCPWM0_TR_OUT0[1]	-
4	TCPWM0_TR_OUT1[1]	-
5	TCPWM0_TR_OUT0[256]	The 16-bit TCPWM counters
6	TCPWM0_TR_OUT1[256]	-
7	TCPWM0_TR_OUT0[257]	-
8	TCPWM0_TR_OUT1[257]	-
9	TCPWM0_TR_OUT0[258]	-
10	TCPWM0_TR_OUT1[258]	-
11	TCPWM0_TR_OUT0[259]	-
12	TCPWM0_TR_OUT1[259]	-
13	TCPWM0_TR_OUT0[260]	-
14	TCPWM0_TR_OUT1[260]	-
15	TCPWM0_TR_OUT0[261]	-
16	TCPWM0_TR_OUT1[261]	-
17	TCPWM0_TR_OUT0[262]	-
18	TCPWM0_TR_OUT1[262]	-

Table 32 PERI trigger source Group 7 (CANTT TriggerMux)

Trigger MUX input	Trigger source label	Description
0	ZERO	-
1	CAN_TT_TR_OUT[0]	CAN TT Sync Outputs

Table 33 PERI trigger destination Group 8 (SCB_CAN0 TriggerMux)

Trigger MUX output	Trigger destination label	Description
0	ZERO	-
1	SCB_TX[0]	-
2	CAN_DBG0	-

Table 34 PERI trigger source Group 9 (SCB_CAN1 TriggerMux)

Trigger MUX input	Trigger source label	Description
0	ZERO	-
1	SCB_RX[0]	-
2	CAN_FIFO0	-

The trigger one-to-one connections for the CYW20829 are provided in Table 35 to Table 39.

Table 35 PERI trigger 1-to-1 Group 0 (SCB DW0 triggers)

Input	Label	Description
0	SCB[1]_TX_TO_PDMA0_TR_IN[6]	-
1	SCB[1]_RX_TO_PDMA0_TR_IN[7]	-
2	SCB[2]_TX_TO_PDMA1_TR_IN[8]	-
3	SCB[2]_RX_TO_PDMA1_TR_IN[9]	-

Table 36 PERI trigger 1-to-1 Group 1 (AUDIOSS PDMA1 triggers)

Input	Label	Description
0	I2S[0]_TX_TO_PDMA0_TR_IN[10]	I2S to PDMA1 triggers
1	I2S[0]_RX_TO_PDMA0_TR_IN[11]	-
2	PDM[0]_RX_TO_PDMA0_TR_IN[12]	-
3	PDM[0]_RX_TO_PDMA0_TR_IN[13]	-
4	PDM[0]_RX_ALL_TO_PDMA0_TR_IN[14]	-

Table 37 PERI trigger 1-to-1 Group 2 (CAN to PDMA0 direct connect)

Input	Label	Description
0	CAN_FIFO1_TO_PDMA0_TR_IN[15]	CAN to PDMA0 direct connect

Table 38 PERI trigger 1-to-1 Group 3 (CAN DW0 triggers (from DW back to CAN))

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

Table 39 PERI trigger 1-to-1 Group 4 (To LIN0)

Input	Label	Description
0	PDMA0_TR_OUT[0]_ACK_TO_LIN	LIN DW trigger
1	PDMA0_TR_OUT[1]_ACK_TO_LIN	LIN DW trigger

10.4 Software triggers

All input and output signals to a trigger multiplexer can be activated through software. This is achieved by writing to the PERI_TR_CMD register. This register enables triggering of the corresponding signal for a specified number of peripheral clock cycles.

10.5 Trigger functionality

Figure 26 provides an overview of a multiplexer trigger group.

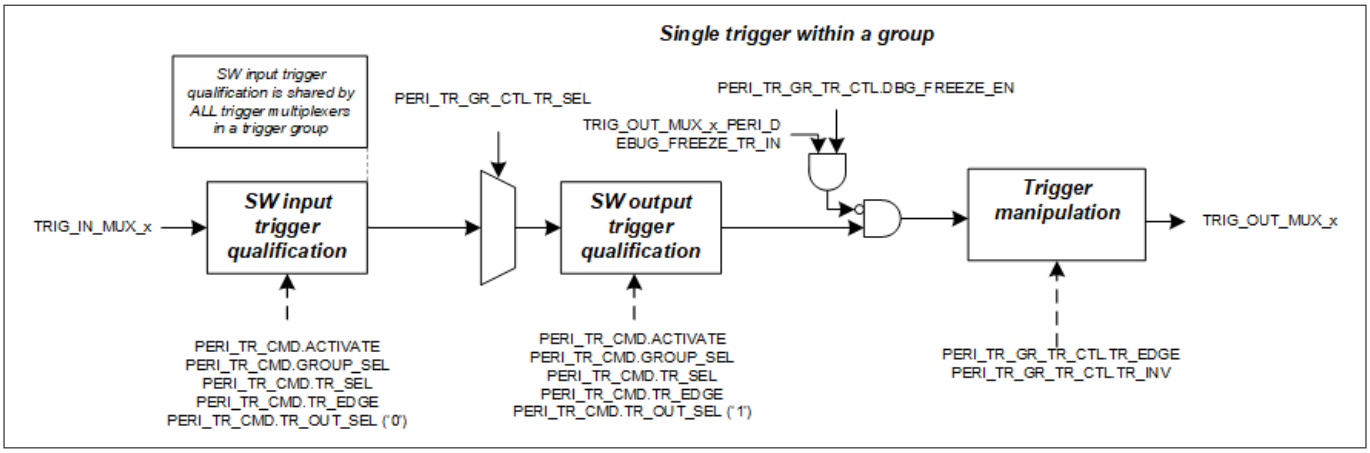


Figure 26 Multiplexer trigger group

Group trigger configuration: The following registers can be configured for trigger groups:

1. `PERI[Peri Group Number]_TR_GR[Group Number].PERI_TR_GR_TR_CTL[Trigger Number].TR_SEL`: Specifies the input trigger to be connected
2. `PERI[Peri Group Number]_TR_GR[Group Number].PERI_TR_GR_TR_CTL[Trigger Number].TR_INV`: Configures whether to invert the trigger signal or not
3. `PERI[Peri Group Number]_TR_GR[Group Number].PERI_TR_GR_TR_CTL[Trigger Number].TR_EDGE`: Configures the trigger as edge-sensitive or level-sensitive

Activating the group trigger via software: The following fields in the `PERI[Peri Group Number]_TR_CMD` register are used to activate a group trigger:

1. `TR_SEL`: Specifies the input for the trigger number
2. `GROUP_SEL`: Specifies the trigger group number
3. `TR_EDGE`: Configures the trigger as edge-sensitive or level-sensitive
4. `OUT_SEL`: Set to 1 for an output trigger and `0` for an input trigger
5. `ACTIVATE`: Set to 1 to activate the configured trigger

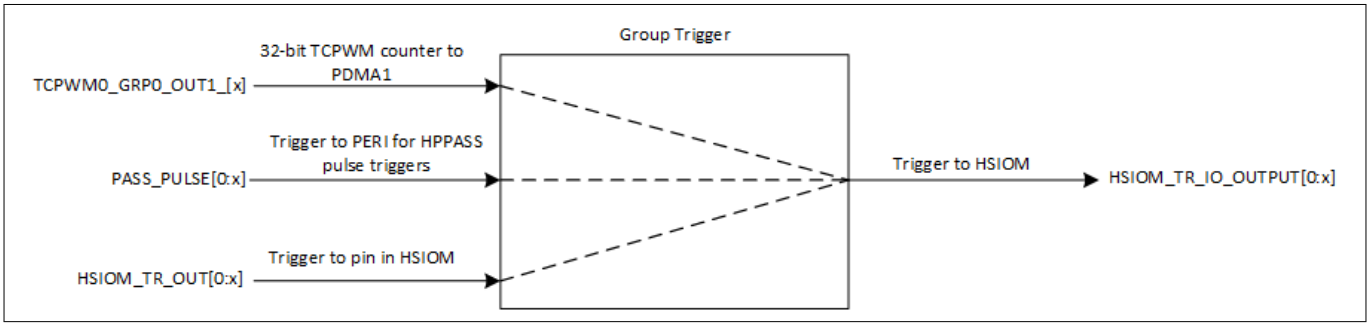


Figure 27 Group trigger

Figure 28 provides an overview of a one-to-one group.

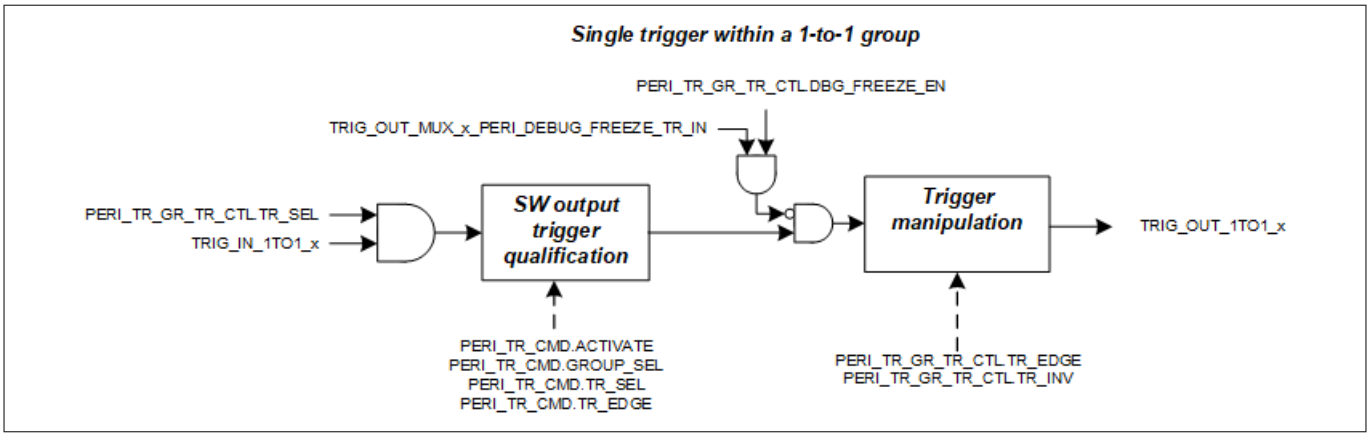


Figure 28 One-to-one trigger group

One-to-one trigger configuration: The following fields are used to configure a one-to-one trigger:

1. `PERI[Peri Group Number]_TR_1TO1_GR[Group Number].PERI_TR_1TO1_GR_TR_CTL[Trigger Number].TR_SEL`: Specifies whether the input trigger is active
   - Set to `True` for an input trigger
   - Set to `False` to use a constant signal level of '0'
2. `PERI[Peri Group Number]_TR_1TO1_GR[Group Number].PERI_TR_1TO1_GR_TR_CTL[Trigger Number].TR_INV`: Configures whether to invert the output trigger signal or not
3. `PERI[Peri Group Number]_TR_1TO1_GR[Group Number].PERI_TR_1TO1_GR_TR_CTL[Trigger Number].TR_EDGE`: Specifies whether the trigger is edge-sensitive or level-sensitive

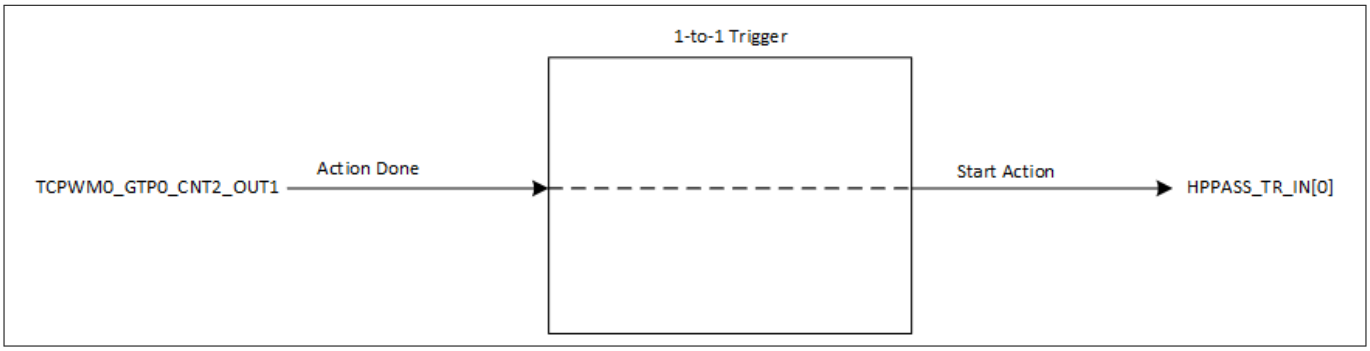


Figure 29 Example of a one-to-one trigger

A trigger group consists of multiple trigger multiplexers, each accompanied by associated trigger manipulation logic. All trigger multiplexers within a trigger group share the same number of input triggers.

As previously mentioned, the trigger manipulation logic includes asynchronous edge-detection functionality, which ensures proper handling of input triggers. The details of the trigger manipulation process are shown in Figure 30.

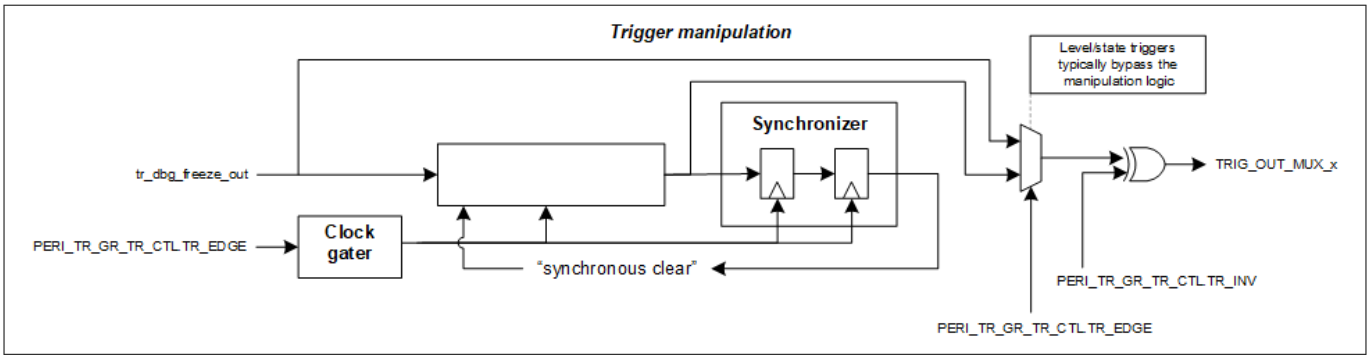


Figure 30 Trigger manipulation

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

***Note**: In some cases, multiple trigger combinations may result in a collision if both triggers are enabled simultaneously. Proper configuration is required to avoid such conflicts.*

10.6 Registers summary

Table 40 Trigger multiplexer registers

Symbol	Name	Description
PERI[i]_TR_CMD	Trigger command Register	Control allows software to activate a specific input or output trigger within the trigger multiplexer structure.

(table continues...)

Table 40 (continued) Trigger multiplexer registers

Symbol	Name	Description
PERI[i]_TR_GR[x]_TR_CTL[y]	Trigger Group Control Register	• Manages group trigger actions by connecting a peripheral input trigger to multiple peripheral output triggers • Allows inversion of the signal • Configures whether the output trigger signal is treated as level-sensitive or edge-sensitive • Specifies whether the output trigger is blocked in debug mode • When the `tr_dbg_freeze` bit (bit 12) is set high, output trigger generation is blocked in debug mode • In the register format: - `i` represents the peripheral group - `x` represents the trigger group - `y` represents the output trigger line number from the multiplexer

(table continues...)

Table 40 (continued) Trigger multiplexer registers

Symbol	Name	Description
PERI[i]_TR_1TO1_GR[x]_TR_CTL[y]	Trigger 1-to-1 Group Control Register	• Controls one-to-one trigger actions by connecting a peripheral input trigger to a specific peripheral output trigger • Allows inversion of the signal • Configures whether the output trigger signal is treated as level-sensitive or edge-sensitive • Specifies whether the output trigger is blocked in debug mode • When the `tr_dbg_freeze` bit (bit 12) is set high, output trigger generation is blocked • In the register format: - `i` represents the peripheral group - `x` represents the trigger group - `y` represents the output trigger line number from the multiplexer

11 Debug and trace system

The CYW20829 program and debug interface provides a communication gateway for an external device to perform programming or debugging. The external device can be either an Infineon-supplied programmer and debugger or a third-party device that supports programming and debugging.

The serial wire debug (SWD) or joint test action group (JTAG) interface can be used as the communication protocol between the external device and CYW20829.

11.1 Features

- Supports programming and debugging through the JTAG or SWD interface
- The CM33 supports an 8-bit instrumentation trace macrocell (ITM)
- Supports the cross triggering interface (CTI) and cross triggering matrix (CTM)
- The CM33 supports four hardware breakpoints and two watchpoints
- Provides read and write access to all memory and registers in the system during debugging, including the CM33 register banks, whether the core is running or halted

11.2 Architecture

Figure 31 shows the block diagram of the program-and-debug interface in CYW20829. The debug access port (DAP) functions as the program and debug interface. The external programmer or debugger, referred to as the "host," communicates with the DAP of CYW20829 "target" using either the serial wire debug (SWD) or joint test action group (JTAG) interfaces.

The debug physical port pins interface with the DAP through the high-speed I/O matrix (HSIOM).

The debug infrastructure is organized into 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 the cross triggering matrix (CTM), the trace port interface unit (TPIU), and the system ROM table

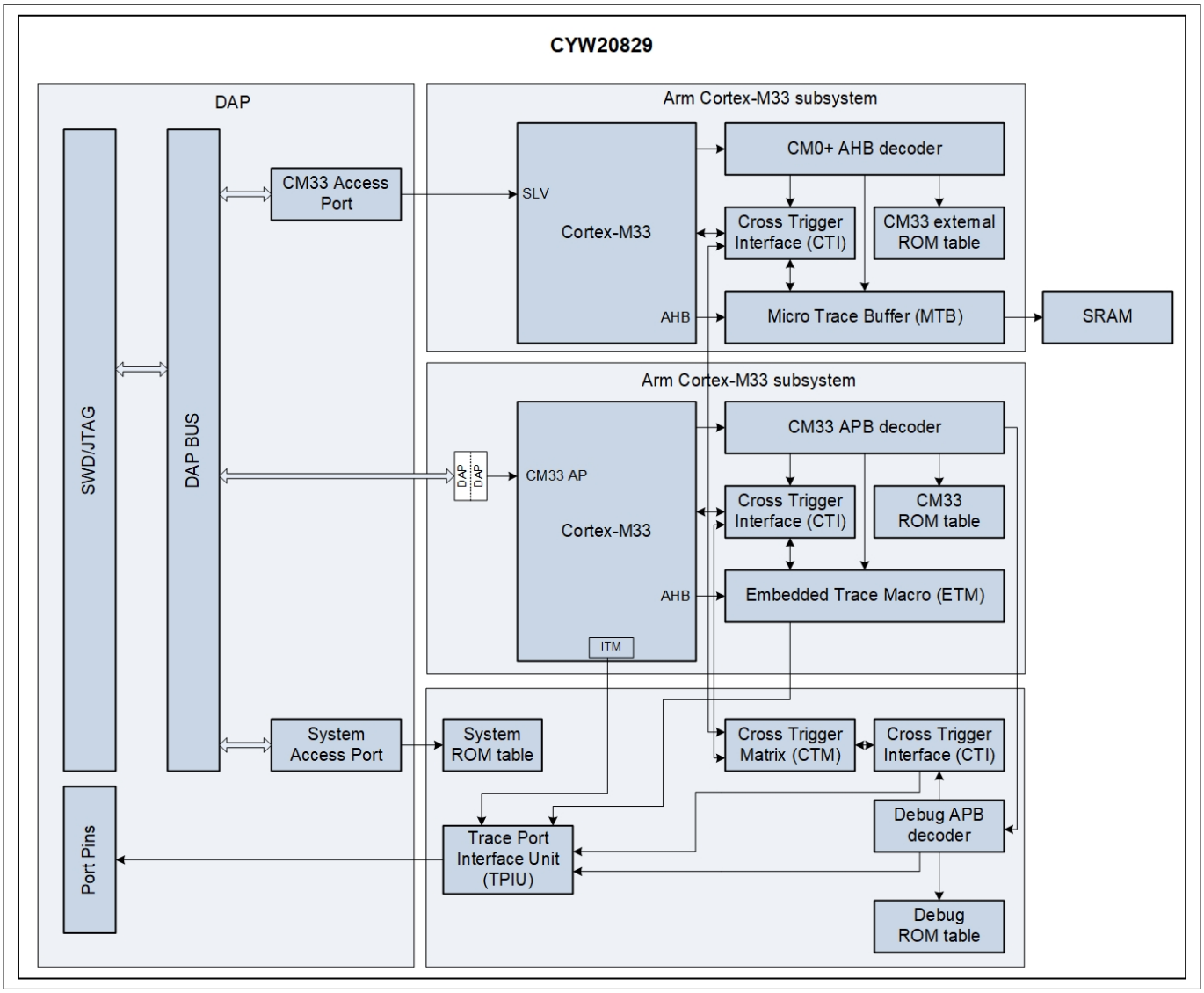


Figure 31 Program and debug interface

The debug access port (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 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)**: Drives trace information out of the chip to an external trace port analyzer
- **Cross-triggering components**:
  - Cross-trigger interface (CTI)
  - Cross-trigger matrix (CTM)
- **ROM tables**

11.2.1 Debug access port (DAP)

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

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

- **Cortex®-M33 AP**:
  - Connects directly to the AHB debug port and provides access to the Cortex®-M33 internal debug components
  - Allows access to the rest of the system through the Cortex®-M33 AHB master interface
  - Provides the debug host the same view as an application running on the Cortex®-M33
  - Enables access to the MMIO of other debug components in the Cortex®-M33 subsystem. These debug components are accessible by the Cortex®-M33 CPU but cannot be reached through the other AHB access ports or by the Cortex®-M33 core itself
- **System-AP**:
  - Provides access to the rest of the system, such as user SRAM and RRAM, through the AHB multiplexer
  - Allows access to the system ROM table, which cannot be reached by any other means
  - The system ROM table provides the chip ID but is otherwise empty

Both the debug port and access port registers are Arm® IPs, defined 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 debugger operates within 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 interface, which uses two pins, SWCLKTCK and SWDIOTMS, that remain operational in Deep Sleep. This allows the SWD debug host to connect during Deep Sleep, while the application is running, or while the chip is powered down.

After connecting the debugger to the chip, the debugger must bring the chip to the Active state before performing any operation. To achieve this, the SWD/JTAG-DP includes a register (`DP_CTL_STAT`) with two power request bits:

- **`CDBGPWRUPREQ`**: Requests debug power
- **`CSYSPWRUPREQ`**: Requests system power

These bits must remain set for the entire duration of the debug session.

The JTAG pins are operational only in Active mode. This means JTAG debugging and JTAG boundary scan are not available when the system is in Deep Sleep. JTAG functionality is accessible only after the chip undergoes a power-on reset.

To perform any operation, move the chip to the Active state. When the debugger attaches, it switches the Cortex®-M33 core to the Active power domain. All other resources must also be moved to the Active state by authenticated software 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 configures the DAP pins so that an SWD or JTAG interface can be connected without additional configuration. When using

a secure policy file, the DAP pins must be explicitly configured and enabled within the secure policy file. For more information on the secure policy file, refer to the Secured Boot section.

The Cortex®-M33 has its own AP_CTL register, which can configure debug features, such as invasive/non-invasive debugging and secure privileged invasive/non-invasive debugging. These debug features are defined in the policy file, which is used by the secure enclave to configure the AP_CTL registers in the protected region of RRAM.

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 ending with _DISABLE are two bits, with the following states:

- **00**: Enabled
- **01**: Disabled
- **1x**: Permanently disabled

For any secure or non-secure debug features to be enabled, AP_CTL.CM33_0_DISABLE must be set to 00.

Table 41 outlines the register configuration in the policy file for non-secure debugging on the Cortex®-M33.

Table 41 Register configuration for non-secure debugging

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, configure the following settings in the policy file for the Cortex®-M33 as shown in Table 42.

Table 42 Register configuration for secure debugging

Debug mode	AP_CTL.CM33_SPID_DISABLE	AP_CTL.CM33_SPNID_DISABLE	AP_CTL.CM33_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 enables a discoverable debug topology. This allows a debug host to locate all debug components on a chip without requiring prior knowledge of the chip's identity. To discover the chip identity and debug topology, the CoreSight ROM tables are used.

The ROM tables are organized in a tree hierarchy. Each AP (Access Port) contains a register with a 32-bit address pointing to the base of the root ROM table for that AP. The root ROM table, known as the 'DAP ROM table,' points to the different APs. The CoreSight DP (Debug Port) component includes a register called BASEPTR, which points to the DAP ROM table. For CYW20829, there are four such root ROM tables: DAP, M33, and System.

Each ROM table consists of 32-bit entries. These entries contain address pointers that either point to the base of the next-level ROM table or to the base of a leaf debug component. Additionally, each ROM table includes a set of ID registers that store JEDEC-compliant identifiers. These identifiers specify the manufacturer, part number, and major and minor revision numbers. For all ROM tables in CYW20829, these IDs are identical.

Each ROM table and CoreSight-compliant debug component also includes component identification registers.

11.2.3 Trace

Cortex®-M33 includes an instrumentation trace macrocell (ITM). This component captures the program execution flow from the CPU and generates trace outputs on the advanced trace bus (ATB). The trace outputs are routed to the coresight trace funnel (CSTF), which combines multiple trace streams onto a single bus.

The debugger includes a 4-bit trace port interface unit (TPIU). The TPIU drives external pins through the IOSS interface, allowing the trace to be captured by an external trace port analyzer (TPA). The TPIU supports simultaneous trace outputs from Cortex®-M33.

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), which provides the ability to trigger (send or receive) events between different debug components. For example, trigger events can be used to:

- Start or stop both CPUs at (almost) the same time
- Start or stop instruction tracing based on the trace buffer being full, not full, or triggered by other events

CoreSight uses two main components to support ECT: the cross-trigger interface (CTI) and the cross-trigger matrix (CTM).

- The CTI interfaces with other debug components, sending and receiving triggers and synchronizing them as needed
- The CTM connects several CTIs, enabling events to be communicated from one CTI to another

The debug infrastructure includes a CTI that connects events primarily to the trace port interface unit (TPIU) and system triggers.

***Note**: The CPU includes an internal CTI that connects events between the CPU core and debug components within the CPU (such as the data watchpoint and trace module, or DWT). These two CTIs are connected via the CTM, allowing events to be sent or received between the two CTIs.*

11.3 Serial wire debug (SWD) interface

CYW20829 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 a unidirectional clock signal (SWDCK). The host programmer always drives the clock line, while either the host or the target drives the data line.

A complete data transfer (one SWD packet) consists of three phases:

1. **Host packet request phase**: The host issues a request to the CYW20829 target
2. **Target acknowledge response phase**: CYW20829 target sends an acknowledgment to the host
3. **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 the Arm® Debug Interface Architecture Specification ADIv6.0.

11.4 JTAG interface

In response to higher pin densities on integrated circuits (ICs), the Joint Test Action Group (JTAG) proposed a method to test circuit boards by controlling the pins on ICs and reading their values through a separate test interface. This method, later formalized as IEEE Standard 1149.1-2001, is based on the concept of a serial shift register routed across all pins of the IC, giving rise to the term "boundary scan." The circuitry at each pin is supplemented with a multipurpose element called a boundary scan cell.

In CYW20829, most GPIO port pins have an associated boundary scan cell (refer to 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), commonly referred to as the JTAG interface.

The JTAG interface consists of the following signals:

- **Test Data In (TDI)**
- **Test Data Out (TDO)**
- **Test Mode Select (TMS)**
- **Test Clock (TCK)**: Clocks the above signals

The TDI, TMS, and TCK signals are inputs to the device, while the TDO signal is the output from the device.

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

11.5 Programming the external flash

The external flash on CYW20829 is programmed using the following sequence:

1. Acquire the SWD port on CYW20829
2. Enter programming mode
3. Execute the device programming routines, including:
   - Silicon ID check
   - Flash programming
   - Flash verification
   - Checksum verification

11.5.1 SWD port acquisition

When the device is reset by asserting the external reset (XRES) pin, the debug port (DP) defaults to JTAG operation. To switch to the serial wire debug (SWD) protocol, reset the device using XRES to place it in its reset state. The host must then send the following SWD connect sequence:

1. Set SWDIO high for at least five SWCLK cycles. This action places the JTAG test access port (TAP) state machine into the Test-Logic-Reset state and selects the IDCODE instruction
2. Transition the device from the JTAG state to the dormant state by writing the 31-bit sequence `0x2EEEEEE6` (most significant bit (MSB) first) on SWDIO. The device enters the 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. Send the 128-bit selection alert sequence `0x49CF9046 A9B4A161 97F5BBC7 45703D98` (MSB first) on SWDIO
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 ensures that the SWD interface is in the line-reset state. The device is now in SWD configuration

CYW20829 uses dormant operation to transition between JTAG and dormant states, as well as between dormant and SWD states.

For additional 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 accomplished by setting the TEST_MODE bit (bit 31) in the test mode control register (MODE register). It is critical to ensure the debug port is configured properly before transitioning to the device programming mode.

11.5.3 SWD programming routine executions

When the device is in programming mode, the external programmer can begin sending the SWD packet sequence to perform programming operations. These operations include flash erase, flash program, checksum verification, and similar tasks. The programming routines are detailed in the Non-volatile memory chapter.

11.6 CYW20829 SWD debug interface

The Cortex®-M33 DAP debugging features are categorized into two types: invasive debugging and noninvasive debugging.

- **Invasive debugging**: This includes program halting, stepping, breakpoints, and data watchpoints
- **Noninvasive debugging**: This includes instruction address profiling and device memory access, such as access to SRAM and other peripheral registers

The DAP consists of three key debug subsystems:

1. **Debug control and configuration registers**: Registers used to manage and configure debugging features
2. **Breakpoint unit (BPU)**: Provides breakpoint support
3. **Debug watchpoint and trace (DWT)**: Provides watchpoint support

For complete details on the debug architecture, refer to the Armv8-M architecture reference manual.

***Note**: The BPU is a part of the Flashpoint and Breakpoint (FPB) unit, as defined in the Arm® Debug Interface Architecture Specification ADIv6.0.*

11.6.1 Debug control and configuration registers

The debug control and configuration registers are used to perform firmware debugging. The registers and their key functions are listed below. For detailed bit-level definitions of these registers, refer to the Armv8-M architecture reference manual.

- **Debug Halting Control and Status Register (CM33_DHCSR)**: Contains the control bits to enable debugging, halt the CPU, and perform single-step operations. It also includes status bits indicating the processor's debug state
- **Debug Fault Status Register (CM33_DFSR)**: Describes the reason for a debug event, such as a CPU halt, breakpoint event, or watchpoint event
- **Debug Core Register Data Register (CM33_DCRDR)**: Stores data to write to or read from the register selected in the CM0P_DCRSR register
- **Debug Exception and Monitor Control Register (CM33_DEMCR)**: Contains enable bits for global debug watchpoint (DWT) block enable, reset vector catch, and hard fault exception catch

11.6.2 Breakpoint unit (BPU)

The Breakpoint unit (BPU) provides breakpoint functionality for instruction fetches. The Cortex®-M33 in the CYW20829 supports four hardware breakpoints. In addition to hardware breakpoints, any number of software breakpoints can be created using the BKPT instruction. The BPU includes the following types of registers:

- **Breakpoint Control Register (CM33_FPB_CTRL)**: This register enables the BPU and stores the number of hardware breakpoints supported by the debug system
- **Breakpoint Compare Register (CM33_FPB_COMPx)**: Each hardware breakpoint is associated with one of these registers. It contains an enable bit for the breakpoint and the compare address value. A typical use case is when an instruction fetch address matches the compare address of a breakpoint, triggering a breakpoint event and halting the processor

11.6.3 Data watchpoint (DWT)

The DWT unit provides watchpoint support for data address accesses or program counter (PC) instruction addresses. The DWT in the Cortex®-M33 supports two watchpoints. Additionally, it facilitates external program counter sampling using a PC sample register, which enables noninvasive coarse profiling of the program counter.

The DWT includes the following key registers:

- **Watchpoint Compare Registers (CM33_DWT_COMPx)**: These registers store the compare values used by the watchpoint comparator to generate watchpoint events. Each watchpoint has an associated `DWT_COMPx` register
- **Watchpoint Function Registers (CM33_DWT_FUNCTIONx)**: These registers store the conditions that trigger watchpoint events. Watchpoint events can be triggered by either program counter events or data address read/write access events. A status bit in these registers is set when the corresponding watchpoint event occurs
- **Watchpoint Comparator PC Sample Register (CM33_DWT_PCSR)**: This register stores the current value of the program counter. It is used for coarse, noninvasive profiling of the program counter

11.6.4 Debugging the CYW20829 device

The host performs debugging on the target device by accessing the debug control and configuration registers, BPU registers, and DWT registers. All of these registers are accessed through the serial wire debug (SWD) interface. In the Cortex®-M33, the debug access port (DAP) translates the SWD packets into appropriate register accesses using the DAP-AHB interface.

Key considerations for debugging:

1. **Acquiring the SWD port**:
   - The first step in debugging is to acquire the SWD port. The acquisition sequence is detailed in the SWD port acquisition section
   - The SWD port is considered acquired when the correct Cortex®-M33 DAP SWDID is successfully read from the target device
2. **SWD pin configuration**:
   - For debug transactions to occur on the SWD interface, ensure the corresponding SWD pins are not used for any other purpose
   - Refer to the I/O subsystem section for details on configuring the SWD port pins to use them exclusively for the SWD interface or for other functions such as LCD and GPIO
   - If debugging is required, the SWD port pins must be dedicated to the SWD interface
3. **Programming without debugging**: If only programming support is needed (without debugging), SWD pins can be used for other purposes when not in use for programming

4. **Enabling debugging**: Once the SWD port is acquired, the external debugger sets the C_DEBUGEN bit in the Debug Halting Control and Status Register (DHCSR) to enable debugging
5. **Executing debugging operations**: After enabling debugging, various operations such as stepping, halting, breakpoint configuration, and watchpoint configuration are performed by writing to the appropriate registers in the debug system

Table 43 Debug features registers

Name	Description
CPUSS_AP_CTL	Configure debug features

12 Cryptographic accelerator

The cryptographic accelerator block, "CryptoLite", offers hardware implementation and acceleration of cryptographic functions. Implementing these functions in hardware consumes less time and energy compared to equivalent firmware implementations. Additionally, the block includes the True Random Number Generation (TRNG) functionality.

12.1 Features

The following are the features of the cryptographic accelerator:

1. Advanced Encryption Standard (AES)-128 to accelerate block cipher functionality. It supports only the encryption of a single 128-bit block with a fixed 128-bit key
2. Secure Hash Algorithm (SHA)-256 to accelerate hash functionality
3. True Random Number Generator (TRNG)
4. Vector unit (VU) to support asymmetric key cryptography, such as RSA and Elliptic-Curve Cryptography (ECC)

12.2 Architecture

Figure 32 gives an overview of the CryptoLite block.

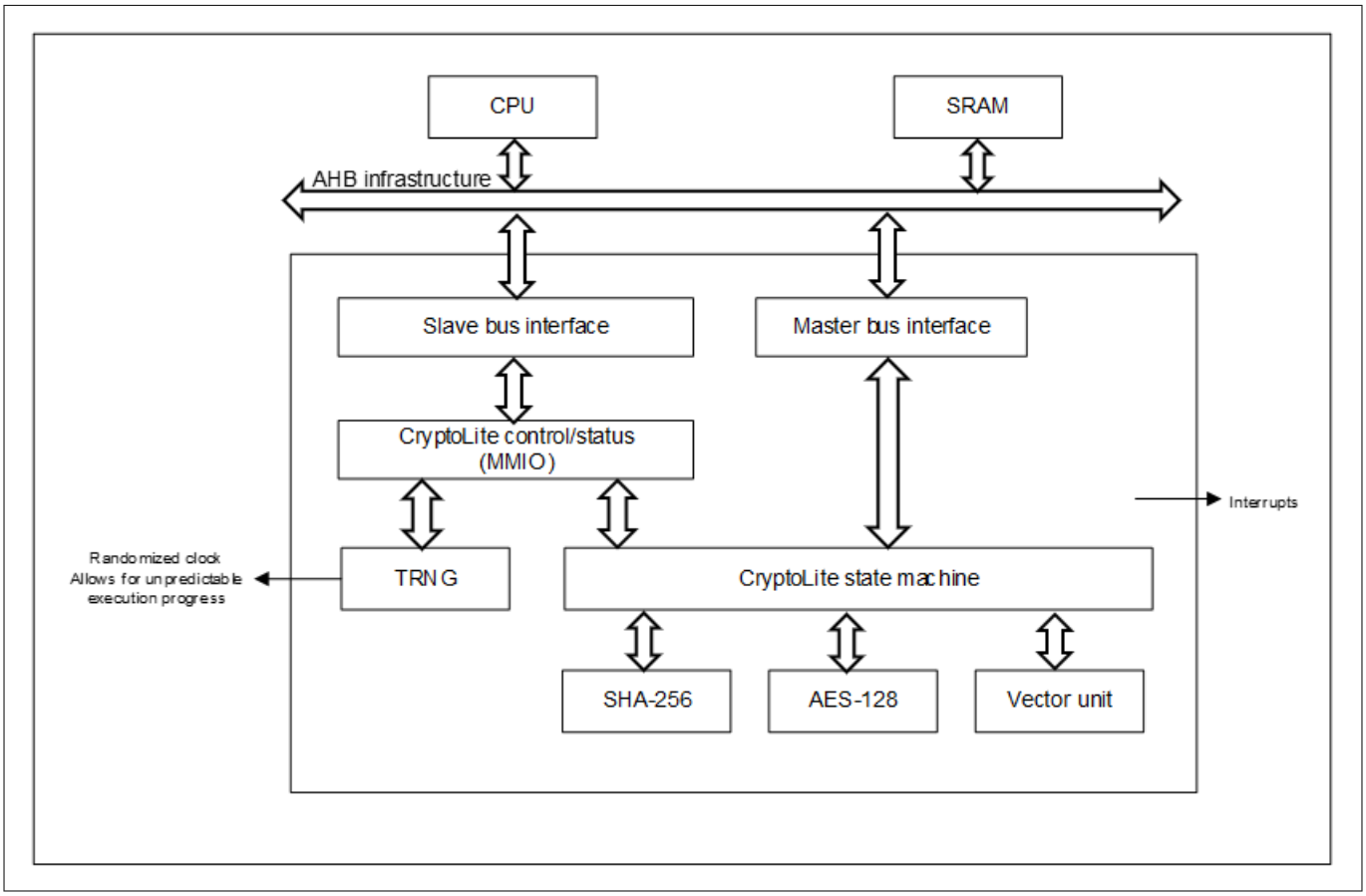


Figure 32 CryptoLite block diagram

The CryptoLite block has the following interfaces:

1. **AHB slave interface**: External bus masters use this interface to access the block's MMIO registers
2. **AHB master interface**: The block uses this interface to access hardware function data structures and operands in memories (SRAM)

3. **Two interrupts**: One interrupt for regular behavior, such as TRNG initialization, and one interrupt for irregular/erroneous behavior, such as when the AHB master interface encounters a bus error
4. **Randomized clock**: This is the TRNG digitized analog signal (DAS) bit-stream sequence
5. **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:

1. **AES-128 component**: Accelerates block cipher functionality. The component supports:
   - Forward encryption of a single 128-bit block with a 128-bit key
2. **SHA-256 component**: Accelerates hash functionality. The component supports:
   - Message schedule calculation for a 512-bit message chunk
   - Processing of a 512-bit message chunk
3. **Vector unit (VU) component**: Accelerates asymmetric key cryptography, such as RSA and ECC. The component supports:
   - Large integer multiplication, addition, and other operations
4. **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 CryptoLite block component provides speed-ups in applications such as secure boot Digital Signature Authentication (DSA or ECDSA) through its SHA-256 and VU hardware acceleration. The digital signature ensures message authentication, integrity, and non-repudiation.

The CryptoLite block operates in Active power mode, and only MMIO configuration registers are retained in Deep Sleep power mode.

***Notes**:*

1. *AHB Lite Slave accesses are 32-bit only; 8/16-bit accesses result in an AHB Lite bus error*
2. *The AHB Lite Master interface inherits the bus transfer attributes of the latest AHB Lite slave interface transfer, typically meaning that the IP has the same privileges as the master that programmed the IP through its MMIO registers*

12.2.1 Randomized clock

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. For more details, see Trigger multiplexer (Trigger MUX).

The default value of the `tr_trng_bitstream` signal is set to a 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 a constant '0'. If the “stop functionality” is not enabled, the DAS bit-stream and the randomized clock will continue to produce random bit sequences. For more details, see True random number generator (TRNG).

12.2.2 Hardware/software interface

The SHA-256, AES-128, and VU components have similar hardware/software interfaces. Each of the following components has a dedicated MMIO register that provides a pointer to a data structure in memory:

- The SHA-256 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, which specifies the desired hardware function and its operands. These data structures are hardware-

function-specific. The block then performs the specified hardware function, and from start to finish, the hardware function is considered busy.

While a hardware function (SHA-256, AES-128, VU) is busy, the CryptoLite block cannot accept a new MMIO register write transfer. Any new MMIO register write transfer will be made pending, which blocks any other (write or read) MMIO register transfers.

However, while a hardware function is busy and the CryptoLite block is not blocked, it 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:

1. Only one hardware function (SHA-256, AES-128, VU) can be busy or active at the same time; they are mutually exclusive. This is desired as the implementation shares hardware resources
2. The data structure pointer MMIO registers cannot be overwritten by software while a hardware function is busy
3. Software may prepare the 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, preventing the data structure pointer MMIO register from being overwritten

***Notes**:*

1. *Software should ensure that the current (busy) and next data structures in memory do not overlap*
2. *The TRNG and a hardware function (SHA-256, 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 set up and 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-256), or a long unsigned integer number (VU). Figure 33 shows a high-level block diagram of the descriptor, data structure, and operands.

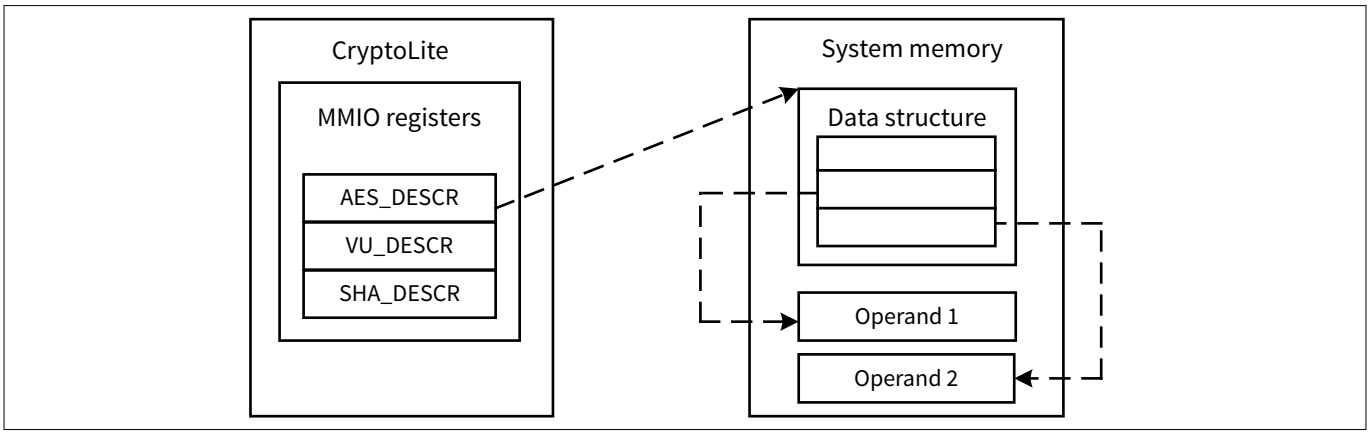


Figure 33 CryptoLite hardware/software interface

Most cryptographic functions are implemented using a combination of CPU software and hardware acceleration.

A hardware function (SHA-256, AES-128, VU) uses an AHB master interface to access memory. If a bus error occurs, it is reported as `CRYPTOLITE_INTR_ERROR.BUS_ERROR`, and the hardware function is terminated.

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 using 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 hardware acceleration for the AES-128 block cipher function. A specific block cipher mode of operation is implemented using both 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 an input for the hardware function
- **Word 1**: Pointer to a 16 * 8-bit block (P). The block is an input for the hardware function
- **Word 2**: Pointer to a 16 * 8-bit block (C). The block is an output for the hardware function

Figure 34 shows the AES-128 data flow.

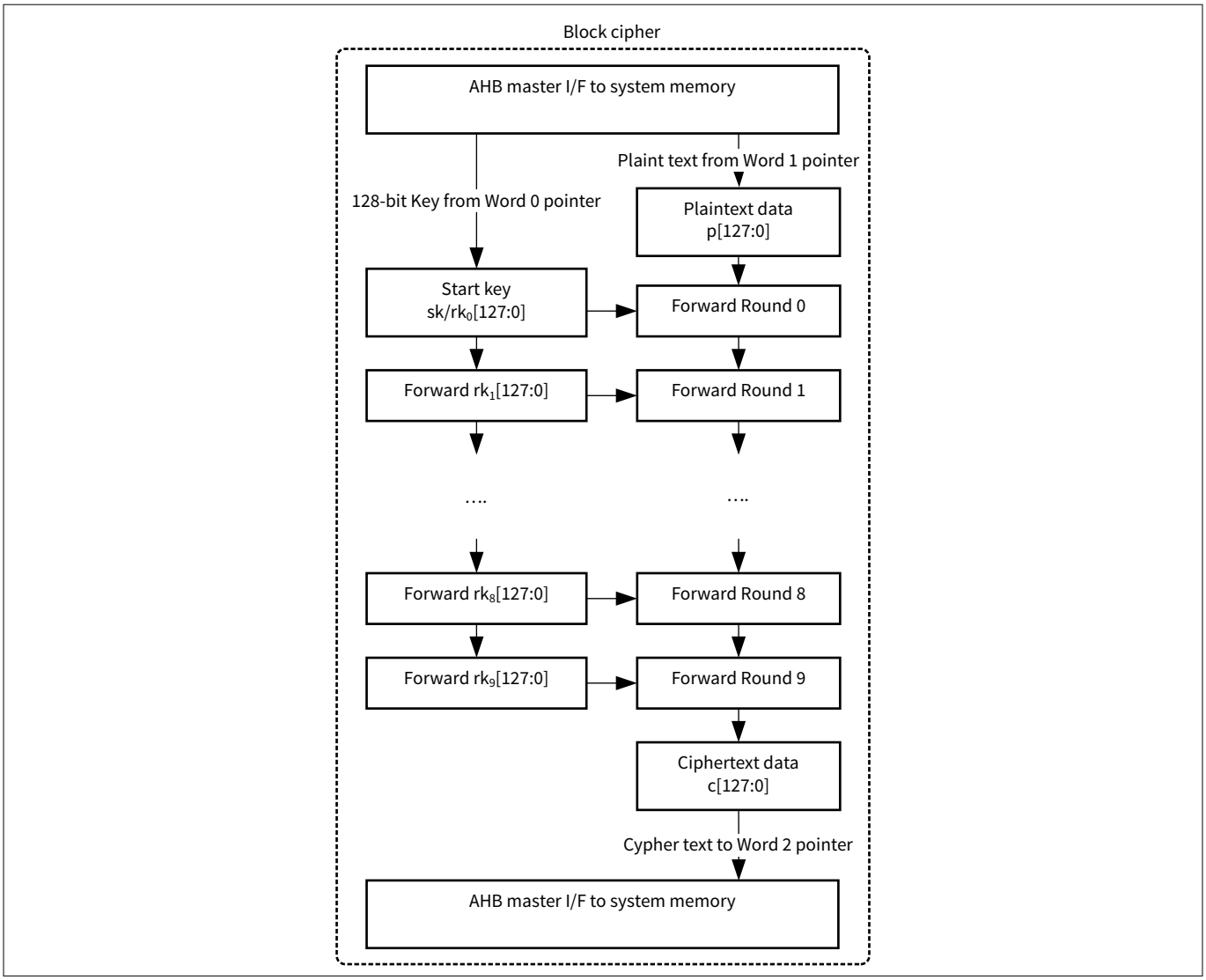


Figure 34 AES-128 data flow

***Note**: The AES-128 block cipher function operates on 8-bit bytes. Therefore, no special endianness considerations are necessary from a hardware point of view.*

12.3.2 SHA-256

The SHA-256 hash function is a member of the SHA-2 hash family. To calculate a message's hash value (also known as a message digest), the message is broken up into multiple 512-bit message chunks. The SHA-256 hash function processes each chunk individually.

The SHA-256 hash function has an internal 256-bit (eight 32-bit words) hash state H[j] (with j = 0, …, 7). For the first message chunk, the initial hash state is set with SHA-256 predefined values. For each successive message chunk, the initial hash state is the previous chunk's final hash state. The final chunk's hash state is the message's hash value.

The SHA-256 hash function performs 64 rounds of a compression function. Each round updates the internal hash state, with each round having a dedicated 32-bit round constant K[i] and a dedicated 32-bit message schedule value W[i].

- The round constants have SHA-256 predefined values and are fixed
- The message schedule values are derived from the 512-bit message chunk and are message-specific

The CryptoLite SHA-256 hash function hardware acceleration consists of two hardware functions:

1. **Message schedule function**: Creates 64 * 32-bit message schedule values W[i] (with i = 0, …, 63) from a 512-bit message chunk M
2. **Process function**: Performs SHA-256 processing of a 256-bit hash state by executing 64 rounds of the 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 input to the process function (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, while 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:

- **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 a process function, Words 1 and 2 are defined as follows:

- **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

***Notes**:*
The SHA-256 hash function operates on 32-bit words and assumes a big-endian memory layout. The CYW20829 MCU uses a little-endian memory layout. This has the following implications:

1. *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*
2. *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*

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, which 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**: This operation is a long integer exclusive-or (XOR) operation
  - `DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0] ^ SRC1[src1_bit_size-1:0]`
- **MOV**: This operation moves or copies a long integer
  - `DST[dst_bit_size-1:0] = SRC0[src0_bit_size-1:0]`
- **LSR1**: This operation shifts a long integer by 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 by 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 by a specified 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
  - `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 operations "greater than or 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 specifies the operation

Table 44 Vector unit opcodes

Opcode	Operation
4’h0	Multiplication (MUL)
4’h1	Addition (ADD)
4’h2	Subtraction (SUB)
4’h3	Exclusive or (XOR)
4’h4	Binary multiplication (XMUL)
4’h5	Logical shift right by 1 (LSR1)
4’h6	Logical shift left by 1 (LSL1)

(table continues...)

Table 44 (continued) Vector unit opcodes

Opcode	Operation
4'h7	Logical shift right (LSR)
4'h8	Conditional subtraction (COND_SUB)
4'h9	Move (MOV)
4'ha to 4'hf	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**: For reserved opcodes (>9), there is no change in hardware state, and the STATUS.BUSY signal will be high for only a few cycles. This is because the CryptoLite hardware does not know if the VU opcode is undefined at the start initiated by the MMIO write. Therefore, CryptoLite will remain busy until it fetches the opcode from the descriptor location and then 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

***Notes**:*
About the operand sizes:

1. *The source and destination operands have dedicated size specifications. The destination operand may be larger than the source operands. This is typically useful for multiplication operations*
2. *When the destination operand size is smaller than the operation result size, the more significant bits of the operation result are ignored. When the destination operand size is larger than the operation result size, the operation result is zero-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, using the same memory locations for the two source operands of an ADD operation will double the source operand
- Use the same memory locations for the source and destination operands. This comes with operation-dependent restrictions. Specifically, the following operations allow the use of the same memory locations for the source and destination operands: ADD, SUB, XOR, MOV, LSR1, and LSL1

The following operations do not allow for the use of the same memory locations for the source and destination operands: MUL, XMUL, LSR, and COND_SUB.

***Note**:* *The VU operates on 32-bit word multiples and assumes a little-endian memory layout. The CYW20829 MCU also uses a little-endian memory layout. As a result, no endian conversion is required.*

12.3.4 True Random Number Generator (TRNG)

The true random number generator (TRNG) component generates true random numbers, each with a fixed size of 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, a jitter is introduced to the ring's oscillating signal. The jittered oscillating signal is sampled to produce a digitized analog signal (DAS). This process 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:

1. An optional reduction step (over up to six ring oscillator DAS bits and over one or multiple DAS bit periods) to increase entropy
2. An optional “von Neumann correction” step to reduce a '0' or '1' bias

The von Neumann correction step processes pairs of reduction bits 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**:* *For a random input bit sequence, the correction step produces an output bit sequence at approximately one-quarter the frequency of the input bit sequence. This is because 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. These true random bit samples are shifted into a register to provide random values of up to 32 bits.

Due to 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 exhibits predictable behavior. However, over time, infinitesimal environmental changes, such as temperature fluctuations, cause increasing deviations from this predictable behavior.

- During the initial delay, the ring oscillator is not a reliable physical noise source
- After the initial delay, the same ring oscillator will exhibit different oscillation behavior and provide a reliable physical noise source

Therefore, the DAS bits can be discarded during the initialization period (`CRYPTOLITE_TRNG_CTL0.INIT_DELAY`).

Figure 35 gives an overview of the TRNG component.

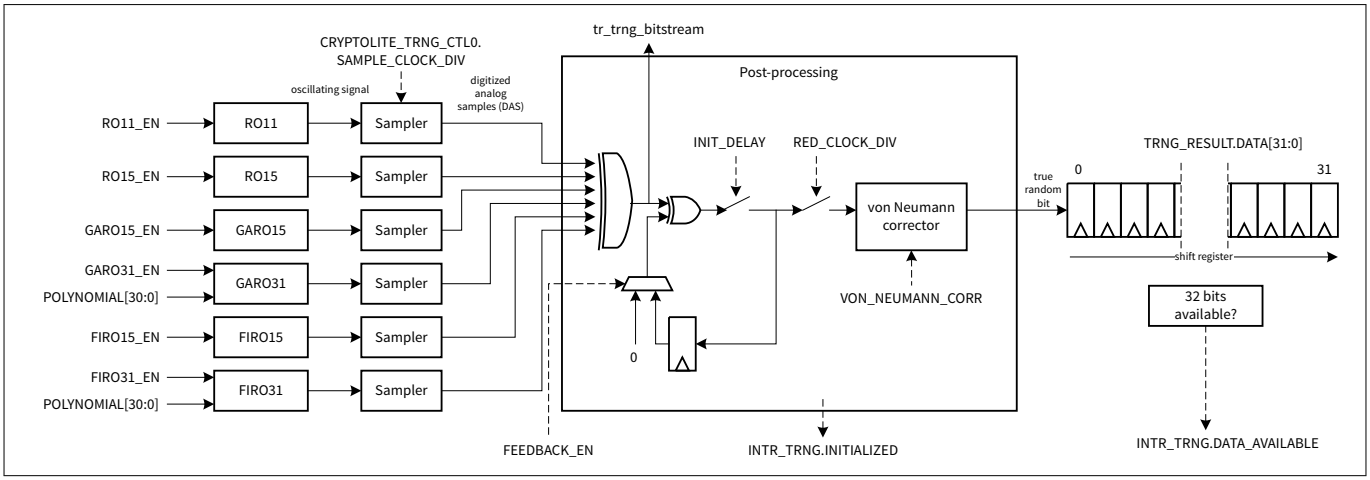


Figure 35 TRNG overview

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.

The `FEEDBACK_EN` bitfield of the `CRYPTOLITE_TRNG_CTL0` register controls the feedback. When set to '0', there is no feedback. This functionality is present because it was found that the “von Neumann” corrector may be less effective with feedback enabled. The "Sampler logic" digitizes the oscillating signal. Figure 36 illustrates this functionality.

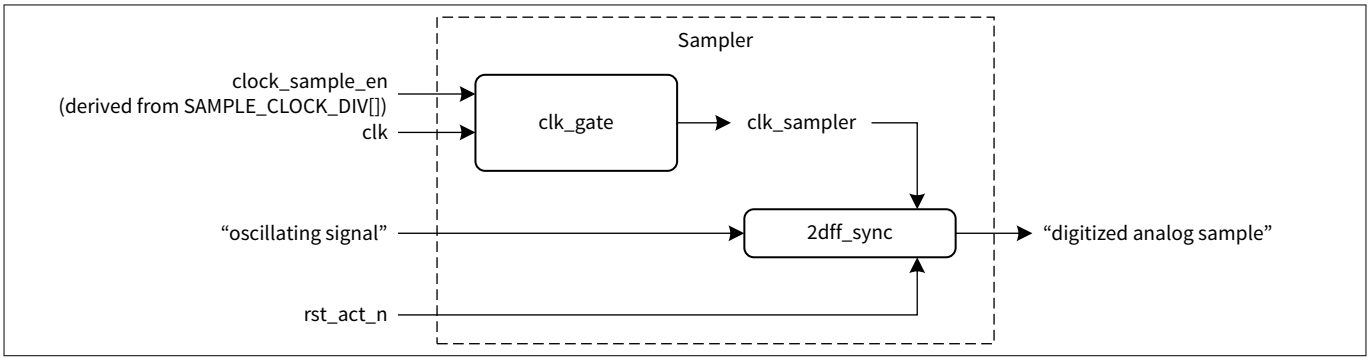


Figure 36 Sampler logic

***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 with 15 inverters
- **GARO31**: A flexible Galois-based ring oscillator with 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 with 15 inverters
- **FIRO31**: A flexible Fibonacci-based ring oscillator with 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 to originate from the CRYPTOLITE_TRNG_CTL1 register.

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 is generated for the following scenarios:

1. TRNG is initialized
2. TRNG data is available in the CRYPTOLITE_TRNG_RESULT register
3. TRNG monitor detects a “repetition count” error
4. TRNG monitor detects an “adaptive proportion” error

The TRNG interrupt has an interrupt mask in the CRYPTOLITE_INTR_TRNG_MASK register. By setting 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) can be 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_INTR_ERROR_MASK register. The status can be checked by reading the CRYPTOLITE_INTR_ERROR_MASKED register to determine 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, which allows software to check for an error after a series of operations rather than after every individual operation.*

12.5 Registers summary

Table 45 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
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 Protection units

The CYW20829 device implements protection units to enforce security by allowing or restricting bus transfers based on specific transfer properties. The rules that determine protection are configured in the MMIO registers of these protection units, which define the protected address space and protection attributes. The hardware that evaluates these attributes to permit or restrict access is the protection unit. The CYW20829 device includes various types of protection units, such as the memory protection unit (MPU), memory protection controller (MPC), peripheral protection controller (PPC), and master security controller (MSC). Each type has a distinct set of MMIO registers that define specific protection regions and their attributes.

The MPC and PPC enable the partitioning of memory and peripheral spaces into multiple regions. These units control the accessibility of each memory or peripheral region based on non-secure attributes and protection context. The MPU, a programmable unit within the CPU, allows privileged software to define memory access permissions.

The protection units allow or restrict bus transfers within the bus infrastructure based on access attributes such as:

- Read or Write attribute
- Execute attribute to distinguish code access from data access
- User or Privileged attribute to differentiate OS or kernel access from task or thread 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 CYW20829 device, examples of bus masters include a Cortex® -M core, DW, or the test controller
- **Protection units**: These hardware blocks implement the protection scheme to safeguard memory, peripherals, and shared resources from bus masters. They enforce the protection rules defined in the protection unit's MMIO registers
- **Protection MMIO registers**: MMIO registers create a register structure in memory that establishes rules for evaluating transfers. Each protection unit is associated with a register structure. An MMIO register structure describes:
  - A memory region on which the rule is applied. A memory region can be defined using either Start/End address or Start/End block index
  - A set of protection attributes, which are properties used to evaluate transfers. These include:
    - R/W/X
    - User or Privileged
    - Protection context

The set of protection attributes that can be configured in MMIO registers depends on the associated protection unit.

Each bus master has its own access attributes that define the bus master's access privileges. A bus master is associated with attributes such as protection context and user/privileged attributes. These are dynamic attributes that change based on the bus master's context and state. However, for some bus masters, certain attributes, such as user/privileged, are fixed and cannot be changed.

Protection attributes appear in two places:

- Protection MMIO registers
- Bus master's protection attributes

The protection units evaluate the access attributes of bus masters against those of the memory or peripheral access attributes. If the access attributes match (between the bus master and memory/peripheral), the protection units allow the transfer; otherwise, the transfer is blocked.

Security and safety considerations include:

- **Security requirements**: Preventing malicious attacks to access secure memory or peripherals
- **Safety requirements**: Detecting accidental (non-malicious) software errors and random hardware errors. It is crucial to enable failure analysis to investigate the root cause of safety violations

13.1 Architecture

When a bus master accesses a memory or peripheral, the protection unit evaluates the access 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 permits the access. In the case of an access violation, a fault condition is triggered, resulting in a bus error. The memory protection controllers (MPC) and peripheral protection controllers (PPC) allow configuring the error response to either a Bus Error or Read-Zero Write-Ignore (RAZ/WI).

The MPU is part of the M33 CPU subsystem and protects the entire 4-GB address space. MPCs are also part of the M33 CPU subsystem and protect memory from unauthorized access. A separate protection unit, PPC, is provided for peripheral protection within the peripheral address space. Additionally, the master security controller (MSC) is implemented to support non-CPU bus masters that lack security extensions.

- **MPU**: The M33 bus master has dedicated MPUs implemented as part of the CPU under the control of privileged software. Typically, MPU access attributes are dynamically updated by privileged software such as the RTOS or TF-M SPM
- **MPC**: MPCs are designed to implement protection in scenarios involving multiple bus masters. They apply protection based on a concept called Protection Context, which allows them to distinguish between different protection contexts and secure memory from non-secure accesses. This makes them effective for multi-master protection
- **PPC**: PPCs are protection units provided in the peripheral register space to safeguard peripherals. Their attributes are similar to those of MPCs, but they are specifically designed to protect peripheral spaces. PPCs distinguish between different protection contexts as well as user-mode and privileged-mode accesses. They implement protection for fixed address regions corresponding to specific peripherals
- **MSC**: MSCs provide security-extension attributes. They configure the security attributes for masters without native security extensions to protect transactions initiated on the bus. Examples of such masters include DW and Crypto

The CYW20829 protection architecture consists of the following components:

- Bus masters with diverse security attributes
- Protection units, including MPU, MPC, PPC, and MSC
- Configurable protection attributes for memory and peripheral spaces
- Mechanisms for error handling, such as Bus Error or RAZ/WI responses

Figure 37 provides an overview of the CYW20829 protection architecture.

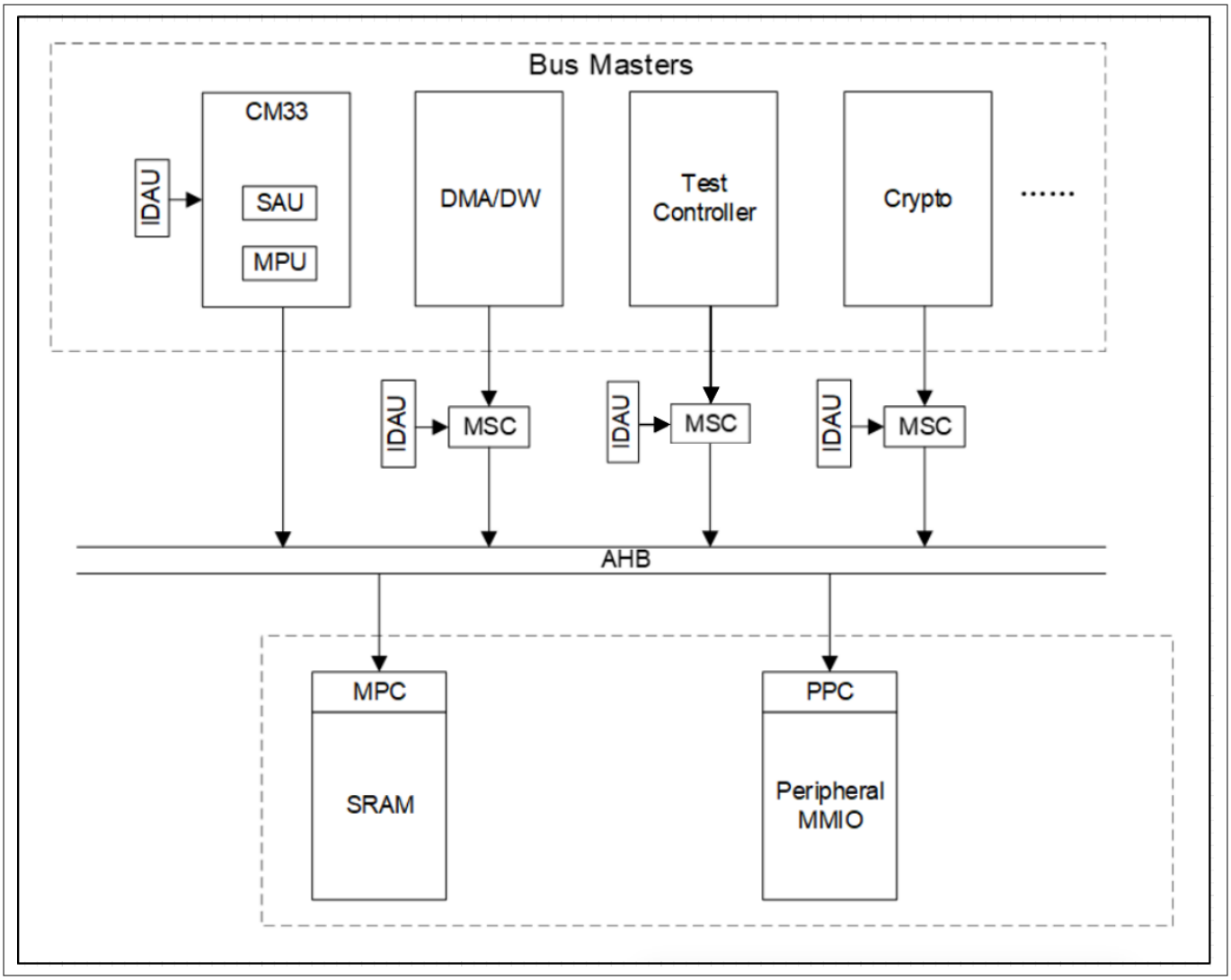


Figure 37 CYW20829 protection architecture

A protection violation occurs when there is a mismatch between a bus master's access attributes and the protection attributes of the memory or peripheral region being accessed. These violations can be logged in the fault-report structure to enable failure analysis. The fault-report structures can generate an interrupt to indicate the occurrence of a fault. This functionality is particularly useful when the violating bus master cannot resolve the bus error on its own and requires another CPU bus master to address the error on its behalf.

A protection violation results in a bus error, which halts the bus transfer before it can reach its intended target. This mechanism ensures that unauthorized or invalid access attempts are effectively blocked.

Figure 38 illustrates the attribute check precedence in the CYW20829 device.

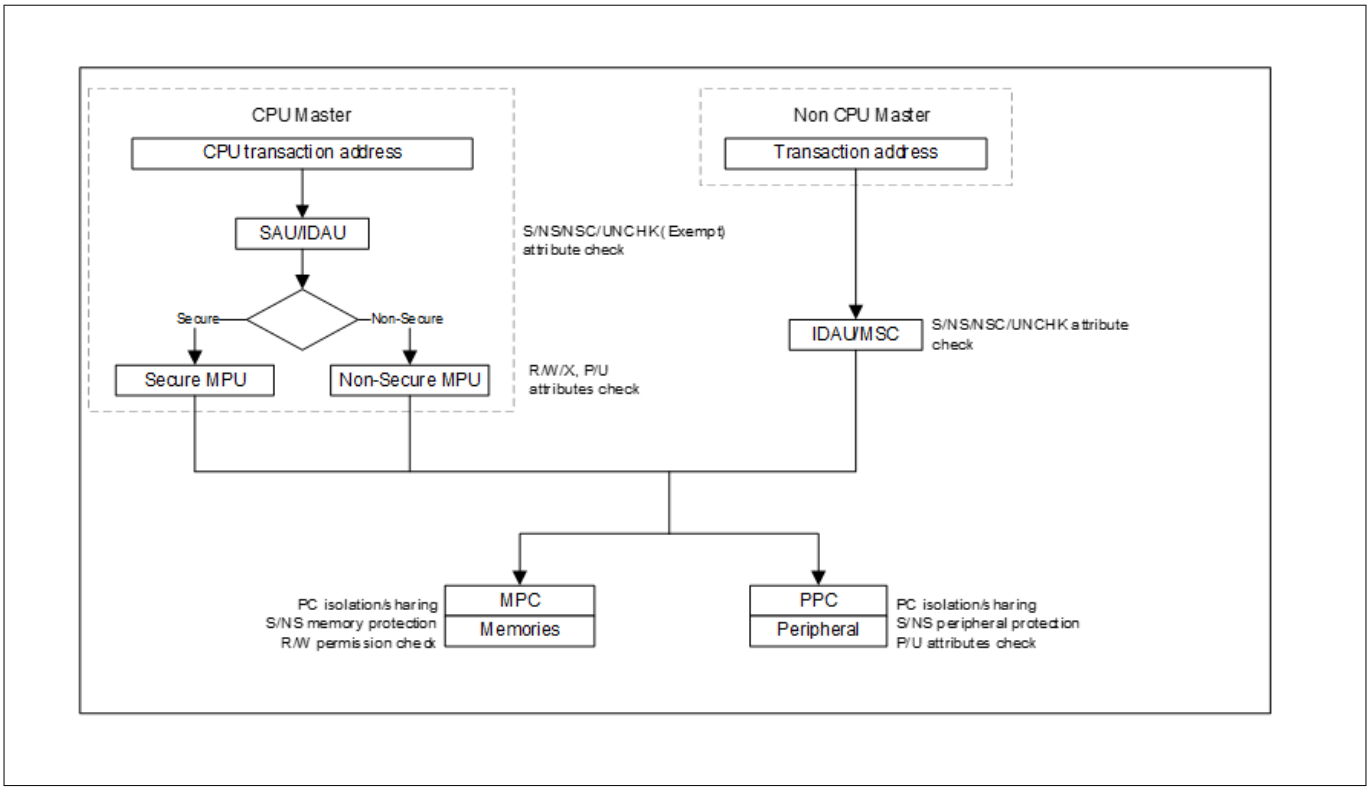


Figure 38 Attribute check precedence

The different types of protection units are designed to address specific use cases within the system. Table 46 summarizes their functionality and primary use cases.

Table 46 CYW20829 protection units

Protection unit	Use	Number of instances
MPU	The memory protection unit (MPU) is used to protect memory between tasks within the same Arm® core. It allows a task in one Arm® core to restrict access to its memory from another task running on the same core.	M33 CPU: 1 Nonsecure MPU with 8 programmable regions for each MPU
MPC	Memory protection controllers (MPC) safeguard memory addresses that are shared between multiple bus masters operating in different protection contexts. This ensures that shared memory is securely accessed based on defined protection rules.	2

(table continues...)

Table 46 (continued) CYW20829 protection units

Protection unit	Use	Number of instances
PPC	Peripheral protection controllers (PPC) are designed to protect the device-defined peripheral memory space. The PPC has fixed regions and is limited to protecting predefined peripheral address regions referred to as PPC regions.	1
MSC adapter	Used for security extension.	2

13.2 Protection context

Protection context is an Infineon proprietary mechanism designed to enforce access restrictions for memory and peripherals shared among multiple bus masters, as well as within the same bus master. Each bus master has a protection context field, denoted as `ms_pcx_pc.pc[3:0]`, which serves as the protection context attribute for all bus transfers initiated by that master. The memory protection controllers (mpcs) and peripheral protection controllers (ppcs) use this protection context attribute to allow or restrict bus transfers between a bus master and memory or peripheral regions.

The protection context for a bus master can be programmed by setting the `ms_pcx_pc.pc[3:0]` (pc[]) field. This allows the configuration of a bus master's protection context. Changes to the protection context are essential for bus masters transitioning between multiple tasks, where each task corresponds to a different protection context. Since the protection context attribute governs access restrictions, any changes to the pc[] field must be carefully managed to maintain system security.

Each bus master is equipped with a `msx_ctl.pc_mask[7:0]` (pc_mask[]) protection context mask field, which specifies the allowed protection contexts for that bus master. This field is a one-hot bit field, meaning each bit represents an individual protection context. A bus master can only operate in protection contexts that are explicitly enabled in its pc_mask[]. Multiple bus masters can share the same protection context.

The active protection context is determined by the `pc[]` field, and any modifications to this field are restricted by the corresponding `pc_mask[]` value. The protection context can only be set to a specific value `i` if the corresponding bit `pc_mask[i]` is set to 1.

For example, when attempting to program `ms_pc_struct[x].pc[3:0]` to 4:

- If `pc_mask[4]` is `1`, the `pc[]` field is successfully set to 4
- If `pc_mask[4]` is `0`, the `pc[]` field remains unchanged

The CYW20829 device supports eight protection contexts, each predefined for a specific task. Table 47 shows the protection context assignments for each task.

Table 47 Protection context assignment to firmware layers

PC	Firmware/Software layer
0	ROM_BOOT
1	Protected Firmware
2	OEM_BL/TF-M
3	OEM NS App
4	
5	

(table continues...)

Table 47 (continued) Protection context assignment to firmware layers

PC	Firmware/Software layer
6	
7	SYS-AP (MXDEBUG600)

Table 48 outlines the bus master's protection attributes as configured by the boot code.

Table 48 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	-	-	PC1, PC2, PC3, PC4, PC5, PC6	PC7
DW	4, 5	Inherited		PC1, PC2, PC3, PC4, PC5, PC6	Inherited
MXCRYPTOLITE	7	Inherited		PC1, PC2, PC3, PC4, PC5, PC6	Inherited
Test Controller	31	-	-	PC7	PC7

The DW controller and CryptoLite accelerator inherit the access control attributes of the bus transfers that were used to program their respective channels or components. This ensures that these masters operate within the same access control parameters defined during their configuration, maintaining consistency in access permissions for subsequent operations.

This design enables masters that generate their own attributes or depend on inherited attributes to function effectively without redundant configuration in the `msx_ctl` register.

13.2.1 Special protection contexts

The active protection context (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 other software. When a switch to PC1, PC2, or PC3 from PC4, PC5, or PC6 is required, it can only be performed using a handler mechanism.

Out of the eight protection contexts, PCs 1, 2, and 3 are considered special. Entry to these special PCs is hardware-controlled. For each special PCx (where x refers to the specific 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 an address matching the value programmed in `MXCM33_CM33_PCx_HANDLER.ADDR[31:0]`, the hardware changes the active protection context to PCx. If the bit `MXCM33_CM33_PC_CTL.VALID[x]` is not set, the PC is treated as an ordinary PC.

Writing to the `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 through the OEM provisioning process. The OEM policy provides a JSON object to program these registers.

For the M33 bus master, the `MS_PC0_PC` register includes the `MS_PC0_PC.PC_SAVED[19:16]` bit field, which holds the saved PC. This saved PC, along with the current PC, implements a two-entry stack. Upon entering a special exception handler, the hardware pushes the current PC onto the stack. It is the responsibility of the exception handler to restore both the active PC and the saved PC before exiting.

When PCx is controlled by hardware (as indicated by the bit MXCM33_CM33_PC_CTL.VALID[x] being set to 1), the protection context cannot be selected by software. In this case, the corresponding bit in the data written to PC_MASK[x] is forced to 0.

***Note**: PC switching with a handler mechanism is implemented only for the M33 CPU and not for other bus masters.*

13.3 Master protection attributes

The protection structures define the rules for different memory regions and their associated access attributes. The protection units utilize a bus master's protection attributes to regulate access based on the rules set by the protection structures. Not all bus masters provide all protection attributes associated with a bus transfer. For example, none of the bus masters has a native protection context attribute. This attribute is dynamically set based on the task executed by the bus master.

To ensure system-wide restricted access, missing attributes are provided by the MMIO register fields. These fields are initialized during the ROM boot process and cannot be modified by user code.

- The PC_MASK[] and PC[] register fields provide protection context functionality
- The MSx_CTL.P register field provides the user/privileged attribute for masters that do not provide their own attribute
- The MSx_CTL.NS register field provides the secure/non-secure attribute for masters that do not provide their own attribute

For masters that do not provide an execute attribute, the execute attribute is set to 0. The platform's DW controller, DMA controller, and Crypto accelerator components inherit the access-control attributes of the bus transfer(s) that programmed the channels or components.

The MSx_CTL registers are only available for masters that are present in the system. For instance, if an external CPUSS bus master is not present, the associated MSx_CTL registers are also not present.

Since the M33 CPU provides its own P and NS attributes, these are used directly. The MSx_CTL.P and MSx_CTL.NS fields are not implemented for the M33 CPU. Additionally, the MS_PCx_PC.PC_SAVED field, which supports special hardware-controlled protection contexts, is only present for the CM33 master.

The MSx_CTL.P, MSx_CTL.NS, and MS_PCx_PC.PC fields are not available for the DW controller and CryptoLite accelerator masters. For these masters, the access-control attributes are inherited from the bus transfer(s) that configured the channels or components.

13.3.1 Master security controller (MSC)

The AHB5 master security controller (MSC) serves as an adapter to connect a legacy AHB-Lite master, or an AHB5 master without the security extension, to an AHB5 system. A master without a security extension can be configured as either secure or non-secure using the MSC by programming the NS bit field, which is part of the MSx_CTL register.

The ROM boot code is responsible for configuring the bus master attributes and locking the configuration to prevent further modifications.

For comprehensive details about the MSC, see the CoreLink SIE-200 Technical Reference Manual by Arm®.

13.4 Slave protection attributes

The access attributes of slaves, including memory and peripherals, are evaluated by the memory protection unit (MPU), memory protection controller (MPC), and peripheral protection controller (PPC) before every transaction with a master.

13.4.1 Memory protection unit (MPU)

The Armv8-M memory protection units (MPUs) are programmable components that are part of the CPU. MPUs allow privileged software to define memory access permissions (read, write, and execute), set memory attributes for normal memory or device memory, and monitor instruction fetch and data access operations initiated by the CPU.

In the CYW20829 platform, the M33 CPU includes eight non-secure MPUs. Any access violation detected by the MPU triggers a MemManage fault.

For more information, see the Armv8-M memory protection unit (MPU) user guide.

13.4.2 Memory protection controller (MPC)

The memory protection controller (MPC) is located on the AHB bus and is equipped in the SRAM. The MPC provides the following functionality:

- Allocates protection-context-based (PC-based) run-time memory separation, isolation, and sharing
- Complies with the Armv8-M memory protection controller specifications
- Makes memory accessible and visible
- Divides the memory space into fixed-size blocks or pages, where the memory block size for SRAM is 2 KB
- Configures the non-secure (NS), read (R), and write (W) attributes for each block across all protection contexts (PCs)
- Offers configurable error response options:
  - `MPCx_CFG.RESPONSE = 1'b0`: Hardware performs Read Zero Write Ignore (RZWI)
  - `MPCx_CFG.RESPONSE = 1'b1`: Hardware generates a bus error

The MPC protection structure includes MMIO registers that program the `LUT_SRAM`. The `LUT_SRAM` stores the security attributes and is connected to the attribution-check logic block, which determines whether to allow or block a transaction.

Figure 39 shows the architecture of the MPC.

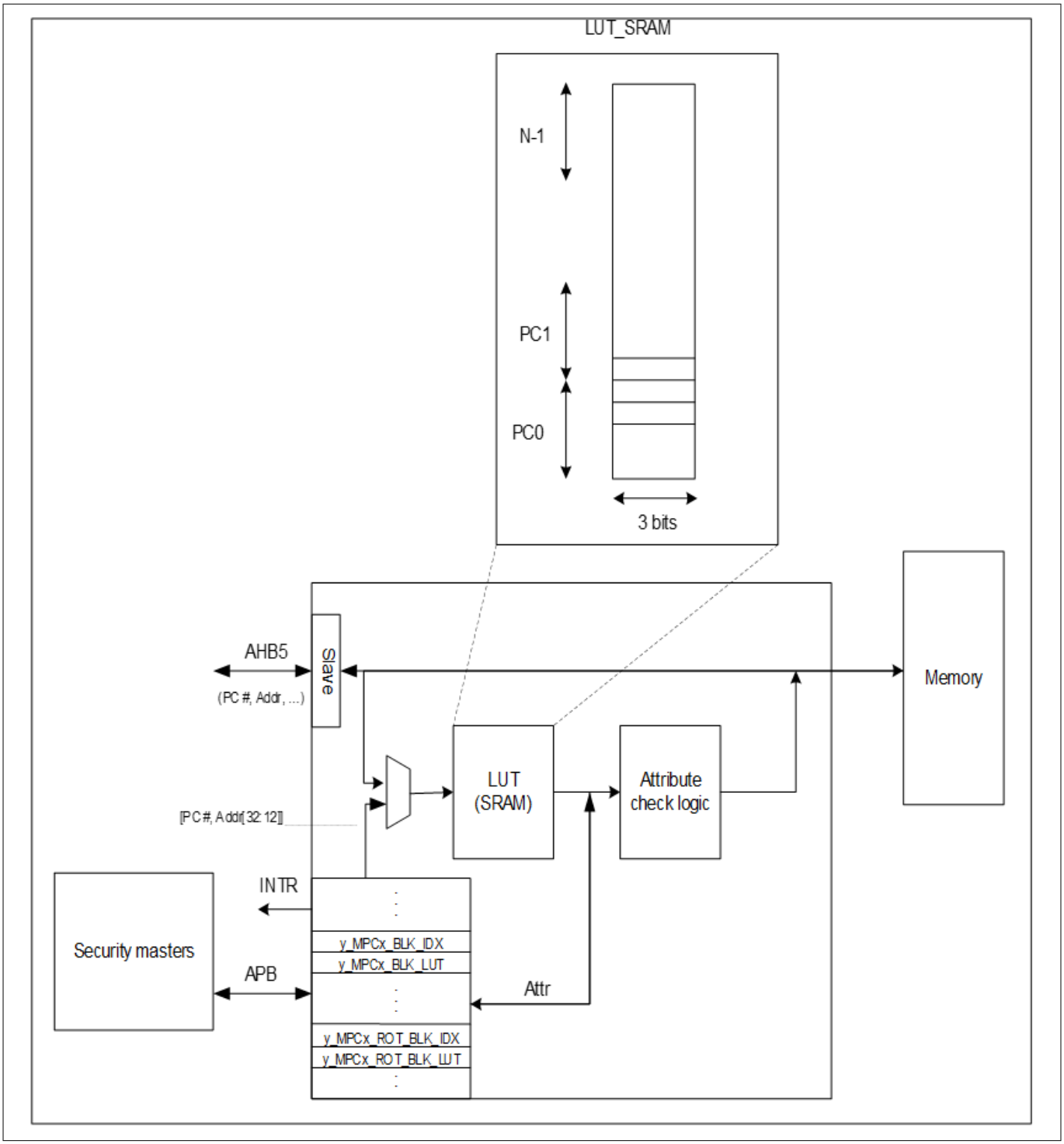


Figure 39 MPC architecture

13.4.2.1 Software view

The memory protection controller (MPC) includes a look-up table composed of SRAM (LUT_SRAM), which contains 3-bit non-secure (NS), read (R), and write (W) attributes for each memory block across all protection contexts (PCs). For each PC, the NSx1, Rx1, and Wx1 attribute bits are reserved and stored in the LUT_SRAM for every memory block. The total number of LUT_SRAM bits is calculated as `3 * PC_NR * the number of blocks in a memory module`, with the maximum supported value of PC_NR being 8.

The MPC configuration is defined as part of the OEM policy and is applied during the Boot ROM process. This configuration cannot be modified at run time by the OEM application. All NS/R/W attributes are set by secure Boot ROM software and are locked to prevent further changes.

Upon reset, the hardware state machine initializes the LUT_SRAM as secure memory with read/write access for all PCs. None of the registers or the LUT_SRAM contents are retained in DEEPSLEEP_RAM or OFF power modes and must be restored following a wakeup event.

Figure 40 provides a software view of the MPC.

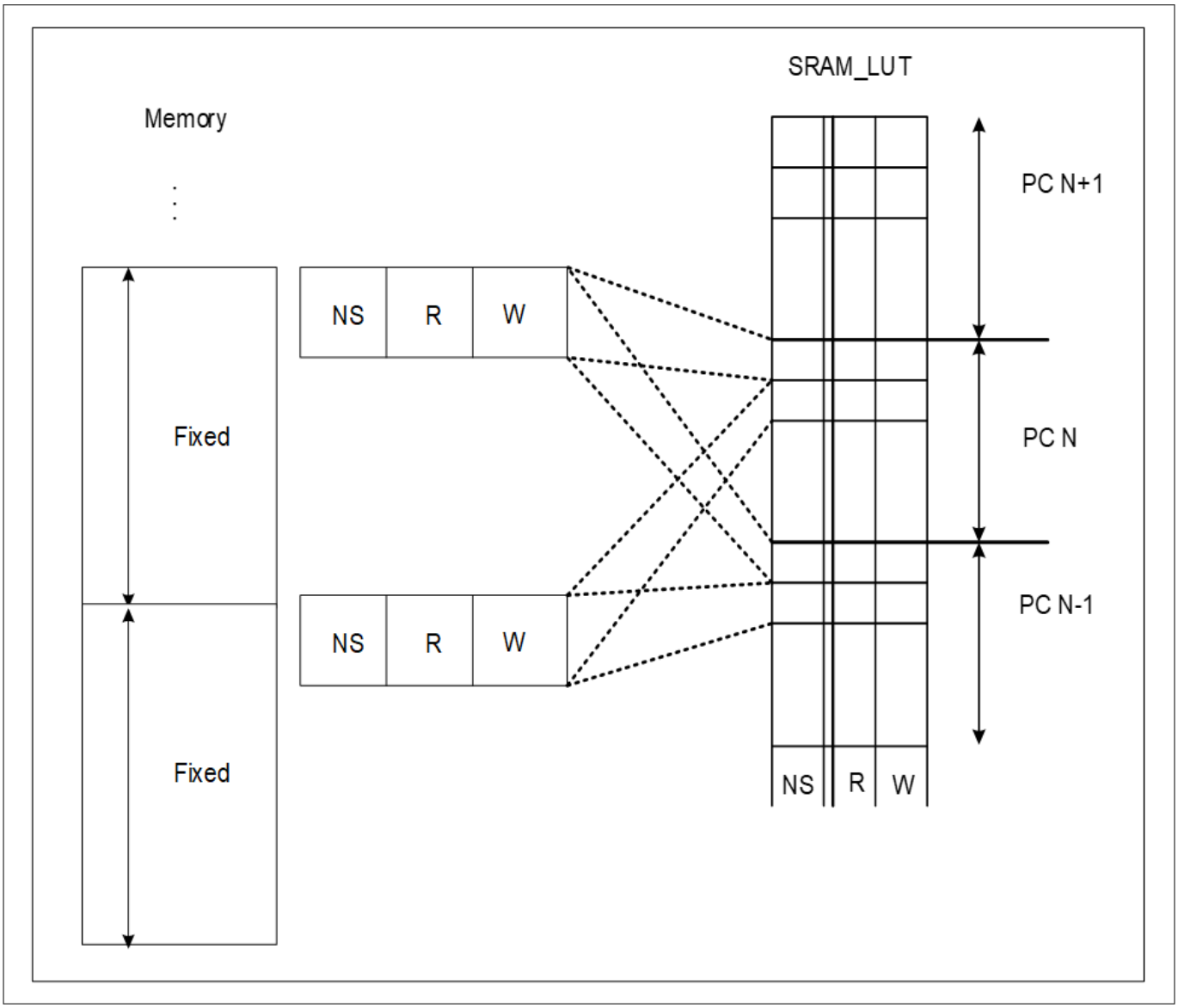


Figure 40 MPC software view

13.4.2.2 MPC MMIO register protection

The MPC MMIO registers are protected by the peripheral protection controller (PPC). Only trusted software is permitted to modify 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.

For example:

- RAMC0_MPC0_PPC_MPC_MAIN controls access to the main SRAM MPC registers

- RAMC0_MPC0_PPC_MPC_PC controls access to the protection-context-related SRAM MPC registers
- RAMC0_MPC0_PPC_MPC_ROT controls access to the root-of-trust SRAM MPC registers

To configure the MPC, the appropriate permissions must first be set by the PPC.

13.4.3 Peripheral protection controller (PPC)

The peripheral protection controller (PPC) provides peripheral access control and protection context (PC). The PPC is positioned between bus masters and peripherals. For each peripheral, the PPC groups the peripheral's MMIO registers, which typically require the same security permissions, into "regions."

For example: Registers exclusively configured by the ROM boot code are grouped together.

Each PPC region can be configured with PC, non-secure (NS), and privilege (P) attributes. A peripheral slave may have multiple PPC regions depending on its access-control requirements. The PPC checks the security attributes of a peripheral region for transactions initiated by the respective masters. If a transaction violates the security attributes, it is blocked.

The PPC provides the following functionality:

- Peripheral protection with an AHB5 bus interface
- Blocking non-secure transfers to secure regions and secure transfers to non-secure regions, treating them as violations
- Blocking non-privileged transfers if the peripheral region is configured as privileged
- Configurable error response options:
  - Read Zero Write Ignore (RZWI)
  - Bus Fault

13.4.4 PPC access control rules

The PPC configuration primarily controls the following registers:

Table 49 PPC registers

Register name	Function
LOCK_MASK	Provides a locking mechanism for each protection context (PC).
PC_MASK	Indicates which protection contexts (PCs) have access to a specific peripheral region. Each peripheral region has its own PC_MASK register.
NS_ATT	Indicates whether a peripheral region is configured as non-secure or secure. Each bit in this register corresponds to a specific peripheral region.
S_P_ATT	Specifies whether access to a secure peripheral region is privileged or unprivileged. Each bit corresponds to a specific peripheral region.
NS_P_ATT	Specifies whether access to a non-secure peripheral region is privileged or unprivileged. Each bit corresponds to a specific peripheral region.

The following conventions are used in the subsequent sections:

- **"i"** represents a protection context (PC) value
- **"j"** represents a peripheral region

Key points about the behavior and configuration of PPC registers:

- After a reset or power cycle, PC_MASK bits are set to high by default, which means access is granted to all PCs
- If PC_MASK[j].PC_MASK[i] = 0x0 and LOCK_MASK[i] = 0, PC"i" has read/write access to the following registers:
  - PC_MASK[j]
  - NS_ATT[j]
  - NS_P_ATT[j]
  - S_P_ATT[j]
- The LOCK_MASK[i] register can only be modified by its corresponding PC"i". Any LOCK_MASK bits accessed by PC"i" and not associated with PC"i" return a Read Zero Write Ignore (RZWI)
- PC"i" has read/write access to peripheral region "j" if PC_MASK[j].PC_MASK[i] = 1'b1

Additional conditions for locked PCs and peripheral regions:

- If (LOCK_MASK[i] & PC_MASK[j].PC_MASK[i]) == 1'b1:
  - The security attributes of peripheral region "j" cannot be changed (except for NS_P_ATT registers). This means PC"i" is "locked" because its corresponding LOCK_MASK bit is set to 1'b1
  - A locked PC"i" cannot write to any PC_MASK register
  - A locked PC"i" cannot gain additional access to any peripheral region or be removed from a peripheral region
  - For any PC"k", if PC_MASK[j].PC_MASK[k] = 1'b1, PC"k" can clear access for other PCs to peripheral region "j" (as long as the region is not locked). However, PC"k" cannot grant additional access to other PCs because PC"i" has locked peripheral region "j"
- If LOCK_MASK[i] == 1'b1 and PC_MASK[j].PC_MASK[i] = 0x0:
  - PC"i" is subject to the locked conditions specified above and does not have read/write access to peripheral region "j"
  - PC"i" cannot modify the security attributes of peripheral region "j" but has read access to the security attributes of that region

The ROM boot code configures and locks certain PPC regions that are shared with PC0. See Table 50 for the list of regions shared with PC0.

For more details, see the CYW20829 Register Reference Manual for the PPC register definitions and programming model.

13.4.4.1 Security aware peripherals

In the CYW20829 platform, peripherals are categorized as either security-aware or security-unaware.

The security-unaware PPC peripheral region (where SECURITY_AWARE = 0) adheres to the following PPC rules:

- Secure transactions can access secure regions only
- Non-secure transactions can access non-secure regions only

The security-aware PPC peripheral region (where 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
  - Non-secure transactions to access non-secure resources only

For security-aware peripherals, the expectation is that the peripheral itself performs the necessary security checks. As a result, the PPC permits both secure and non-secure transactions to pass through, provided all other attributes match.

See Table 50 for a list of security-aware peripherals.

13.4.4.2 PPC regions

The peripheral protection controller (PPC) protects not only peripherals but also the MMIO registers of other components such as the memory protection controller (MPC), peripheral interconnect, system reset and startup subsystem (SRSS), and others.

There are 222 regions in PPC (specifically PPC0 of SYSCPUSS). The ROM boot code configures the following registers for these regions:

- `NS_ATT` (Non-Secure Attribute)
- `S_P_ATT` (Secure Privileged Attribute)
- `NS_P_ATT` (Non-Secure Privileged Attribute)
- `PC_MASK` (Protection Context Mask)

Table 50 outlines the configurations for `PC_MASK`, `NS_ATT`, `S_P_ATT`, and `NS_P_ATT` for all PPC regions as set by the ROM boot code.

Table 50 PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
PERI0_MAIN	42000000	4000	0	All PCs	1	1	1
PERI0_GR0_GROUP	42004010	8	0	All PCs	1	1	1
PERI0_GR1_GROUP	42004040	20	0	All PCs	1	1	1
PERI0_GR2_GROUP	42004080	20	0	All PCs	1	1	1
PERI0_GR3_GROUP	420040c0	20	0	All PCs	1	1	1
PERI0_GR4_GROUP	42004100	20	0	All PCs	1	1	1
PERI0_GR5_GROUP	42004150	8	0	All PCs	1	1	1
PERI0_GR0_BOOT	42004020	4	0	PC0	0	1	1
PERI0_GR1_BOOT	42004060	4	0	PC0	0	1	1
PERI0_GR2_BOOT	420040a0	4	0	PC0	0	1	1
PERI0_GR3_BOOT	420040e0	4	0	PC0	0	1	1
PERI0_GR4_BOOT	42004120	4	0	PC0	0	1	1

(table continues...)

Table 50 (continued) PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
PERI0_GR5_BOOT	42004160	4	0	PC0	0	1	1
PERI0_TR	42008000	8000	0	All PCs	1	1	1
PPC0_PPC_PPC_SECURE	42020000	4000	0	PC0, PC1	1	1	1
PPC0_PPC_PPC_NONSECURE	42024000	4000	0	All PCs	1	1	1
PERI_PCLK0_MAIN	42040000	10000	0	All PCs	1	1	1
CPUSS	42100000	10000	0	All PCs	1	1	1
RAMC0_CM33	42110000	40	0	All PCs	1	1	1
RAMC0_BOOT	42110100	8	0	PC0	0	1	1
RAMC0_RAM_PWR	42110200	100	0	All PCs	1	1	1
RAMC0_MPC0_PPC_MPC_MAIN	42114000	4	0	PC0	0	1	1
RAMC0_MPC0_PPC_MPC_PC	42114100	20	0	PC0	0	1	1
RAMC0_MPC0_PPC_MPC_ROT	42114200	20	0	PC0	0	1	1
PROMC_CM33	42140000	4	0	PC0, PC1	1	1	1
PROMC_MPC0_PPC_MPC_MAIN	42141000	4	0	PC0	0	1	1
PROMC_MPC0_PPC_MPC_PC	42141100	20	0	PC0	0	1	1
PROMC_MPC0_PPC_MPC_ROT	42141200	20	0	PC0	0	1	1

(table continues...)

Table 50 (continued) PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
MXCM33_CM33	42160000	100	0	All PCs	1	1	1
MXCM33_CM33_S	42161000	4	0	PC0	0	1	1
MXCM33_CM33_NS	42161004	4	0	PC0, PC1	1	1	1
MXCM33_BOOT_PC0	42162000	80	0	PC0	0	1	1
MXCM33_BOOT_PC1	42162100	4	0	PC0	0	1	1
MXCM33_BOOT_PC2	42162140	4	0	PC0	0	1	1
MXCM33_BOOT_PC3	42162180	4	0	PC0	0	1	1
MXCM33_BOOT	421621c0	4	0	PC0	0	1	1
MXCM33_CM33_INT	42168000	400	0	All PCs	1	1	1
DW0_DW	42180000	80	0	All PCs	1	1	1
DW0_DW_CRC	42180100	80	0	All PCs	1	1	1
DW0_CH_STRUCT0_CH	42188000	40	0	All PCs	1	1	1
DW0_CH_STRUCT1_CH	42188040	40	0	All PCs	1	1	1
DW0_CH_STRUCT2_CH	42188080	40	0	All PCs	1	1	1
DW0_CH_STRUCT3_CH	421880c0	40	0	All PCs	1	1	1
DW0_CH_STRUCT4_CH	42188100	40	0	All PCs	1	1	1
DW0_CH_STRUCT5_CH	42188140	40	0	All PCs	1	1	1
DW0_CH_STRUCT6_CH	42188180	40	0	All PCs	1	1	1
DW0_CH_STRUCT7_CH	421881c0	40	0	All PCs	1	1	1

(table continues...)

Table 50 (continued) PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
DW0_CH_STRUCT8_CH	42188200	40	0	All PCs	1	1	1
DW0_CH_STRUCT9_CH	42188240	40	0	All PCs	1	1	1
DW0_CH_STRUCT10_CH	42188280	40	0	All PCs	1	1	1
DW0_CH_STRUCT11_CH	421882c0	40	0	All PCs	1	1	1
DW0_CH_STRUCT12_CH	42188300	40	0	All PCs	1	1	1
DW0_CH_STRUCT13_CH	42188340	40	0	All PCs	1	1	1
DW0_CH_STRUCT14_CH	42188380	40	0	All PCs	1	1	1
DW0_CH_STRUCT15_CH	421883c0	40	0	All PCs	1	1	1
CPUSS_ALL_PC	421c0000	80	0	All PCs	1	1	1
CPUSS_DDFT	421c0080	4	0	PC0	0	1	1
CPUSS_CM33_S	421c0100	4	0	All PCs	1	1	1
CPUSS_CM33_NS	421c0120	4	0	All PCs	1	1	1
CPUSS_MSC_INT	421c0200	10	0	All PCs	1	1	1
CPUSS_AP	421c1000	4	0	PC0	0	1	1
CPUSS_BOOT	421c2000	8	0	PC0	0	1	1
MS0_MAIN	421c4000	4	0	PC0	0	1	1
MS4_MAIN	421c4040	4	0	PC0	0	1	1
MS5_MAIN	421c4050	4	0	PC0	0	1	1
MS7_MAIN	421c4070	4	0	PC0	0	1	1
MS31_MAIN	421c41f0	4	0	PC0	0	1	1
MS_PC0_PRIV	421c5000	4	0	PC0	0	1	1

(table continues...)

Table 50 (continued) PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
MS_PC31_PRIV	421c51f0	4	0	PC0	0	1	1
MS_PC0_PRIV_MIR	421c5004	4	0	PC0	0	1	1
MS_PC31_PRIV_MIR	421c51f4	4	0	PC0	0	1	1
MSC_ACG	421c6000	40	0	PC0	0	1	1
CPUSS_SL_CTL_GROUP	421c8000	8	0	All PCs	1	1	1
IPC_STRUCT0_IPC	421d0000	20	1	All PCs	1	1	1
IPC_STRUCT1_IPC	421d0020	20	1	All PCs	1	1	1
IPC_STRUCT2_IPC	421d0040	20	1	All PCs	1	1	1
IPC_STRUCT3_IPC	421d0060	20	1	All PCs	1	1	1
IPC_INTR_STRUCT0_INTR	421d1000	10	1	All PCs	1	1	1
IPC_INTR_STRUCT1_INTR	421d1020	10	1	All PCs	1	1	1
FAULT_STRUCT0_MAIN	421e0000	100	0	All PCs	1	1	1
SRSS_GENERAL	42200000	400	0	All PCs	1	1	1
SRSS_GENERAL2	42200400	40	0	All PCs	1	1	1
SRSS_HIB_DATA	422008a0	10	0	PC0, PC1	1	1	1
SRSS_MAIN	42201000	1000	0	PC0, PC1	1	1	1
SRSS_SECURE	42202000	2000	0	PC0	0	1	1
RAM_TRIM_SRSS_SRAM	42204000	8	0	All PCs	1	1	1
SRSS_DPLL	42204200	40	0	All PCs	1	1	1
SRSS_WDT	4220c000	10	0	All PCs	1	1	1
MAIN	4220d000	40	0	All PCs	1	1	1

(table continues...)

Table 50 (continued) PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
PWRMODE_PWRMODE	42210000	4000	0	PC0, PC1	1	1	1
BACKUP_BACKUP	42220000	100	0	All PCs	1	1	1
BACKUP_B_BREG0	42221000	10	0	All PCs	1	1	1
BACKUP_B_BREG1	42221010	10	0	All PCs	1	1	1
BACKUP_B_BREG2	42221020	20	0	All PCs	1	1	1
BACKUP_B_BREG3	42221080	40	0	All PCs	1	1	1
CRYPTOLITE_MAIN	42230000	100	0	All PCs	1	1	1
CRYPTOLITE_TRNG	42230100	100	0	All PCs	1	1	1
DEBUG600_DEBUG600	42250000	4	0	All PCs	1	1	1
HSIOM_PRT0_PRT	42400000	8	1	All PCs	1	1	1
HSIOM_PRT1_PRT	42400010	8	1	All PCs	1	1	1
HSIOM_PRT2_PRT	42400020	8	1	All PCs	1	1	1
HSIOM_PRT3_PRT	42400030	8	1	All PCs	1	1	1
HSIOM_PRT4_PRT	42400040	8	1	All PCs	1	1	1
HSIOM_PRT5_PRT	42400050	8	1	All PCs	1	1	1
HSIOM_PRT6_PRT	42400060	8	1	All PCs	1	1	1
HSIOM_PRT7_PRT	42400070	8	1	All PCs	1	1	1
HSIOM_PRT8_PRT	42400080	8	1	All PCs	1	1	1
HSIOM_PRT9_PRT	42400090	8	1	All PCs	1	1	1

(table continues...)

Table 50 (continued) PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
HSIOM_AMUX	42402000	10	1	All PCs	1	1	1
HSIOM_MON	42402200	10	1	All PCs	1	1	1
GPIO_PRT0_PRT	42410000	40	1	All PCs	1	1	1
GPIO_PRT1_PRT	42410080	40	1	All PCs	1	1	1
GPIO_PRT2_PRT	42410100	40	1	All PCs	1	1	1
GPIO_PRT3_PRT	42410180	40	1	All PCs	1	1	1
GPIO_PRT4_PRT	42410200	40	1	All PCs	1	1	1
GPIO_PRT5_PRT	42410280	40	1	All PCs	1	1	1
GPIO_PRT6_PRT	42410300	40	1	All PCs	1	1	1
GPIO_PRT7_PRT	42410380	40	1	All PCs	1	1	1
GPIO_PRT8_PRT	42410400	40	1	All PCs	1	1	1
GPIO_PRT9_PRT	42410480	40	1	All PCs	1	1	1
GPIO_PRT0_CFG	42410040	40	1	All PCs	1	1	1
GPIO_PRT1_CFG	424100c0	40	1	All PCs	1	1	1
GPIO_PRT2_CFG	42410140	40	1	All PCs	1	1	1
GPIO_PRT3_CFG	424101c0	40	1	All PCs	1	1	1
GPIO_PRT4_CFG	42410240	40	1	All PCs	1	1	1
GPIO_PRT5_CFG	424102c0	40	1	All PCs	1	1	1
GPIO_PRT6_CFG	42410340	40	1	All PCs	1	1	1

(table continues...)

Table 50 (continued) PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
GPIO_PRT7_CFG	424103c0	40	1	All PCs	1	1	1
GPIO_PRT8_CFG	42410440	40	1	All PCs	1	1	1
GPIO_PRT9_CFG	424104c0	40	1	All PCs	1	1	1
GPIO_GPIO	42418000	40	1	All PCs	1	1	1
GPIO_TEST	42419000	8	1	All PCs	1	1	1
SMARTIO_PRT0_PRT	42420000	100	0	All PCs	1	1	1
SMARTIO_PRT1_PRT	42420100	100	0	All PCs	1	1	1
SMARTIO_PRT2_PRT	42420200	100	0	All PCs	1	1	1
SMARTIO_PRT3_PRT	42420300	100	0	All PCs	1	1	1
SMARTIO_PRT5_PRT	42420500	100	0	All PCs	1	1	1
SMARTIO_PRT6_PRT	42420600	100	0	All PCs	1	1	1
SMARTIO_PRT9_PRT	42420900	100	0	All PCs	1	1	1
DFT	42600000	1000	0	PC0	0	1	1
EFUSE_CTL1	42610000	4	0	All PCs	1	1	1
EFUSE_CTL2	42610100	80	0	PC0	1	1	1
EFUSE_CTL3	42610180	4	0	All PCs	1	1	1
EFUSE_DATA_BOOT1	42610800	80	0	PC0	0	1	1
CANFD0_CH0_CH	42800000	200	0	All PCs	1	1	1
CANFD0_CH1_CH	42800200	200	0	All PCs	1	1	1
CANFD0_MAIN	42801000	40	0	All PCs	1	1	1
CANFD0_BUF	42810000	10000	0	All PCs	1	1	1
SCB0	42820000	10000	0	All PCs	1	1	1

(table continues...)

Table 50 (continued) PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
SCB1	42840000	10000	0	All PCs	1	1	1
SCB2	42850000	10000	0	All PCs	1	1	1
SCB3	42860000	10000	0	All PCs	1	1	1
TCPWM0_GRP0_CNT0_CNT	42a00000	100	0	All PCs	1	1	1
TCPWM0_GRP0_CNT1_CNT	42a00100	100	0	All PCs	1	1	1
TCPWM0_GRP0_CNT2_CNT	42a00200	100	0	All PCs	1	1	1
TCPWM0_GRP0_CNT3_CNT	42a00300	100	0	All PCs	1	1	1
TCPWM0_GRP1_CNT0_CNT	42a10000	100	0	All PCs	1	1	1
TCPWM0_GRP1_CNT1_CNT	42a10100	100	0	All PCs	1	1	1
TCPWM0_GRP1_CNT2_CNT	42a10200	100	0	All PCs	1	1	1
TCPWM0_GRP1_CNT3_CNT	42a10300	100	0	All PCs	1	1	1
TCPWM0_GRP1_CNT4_CNT	42a10400	100	0	All PCs	1	1	1
TCPWM0_GRP1_CNT5_CNT	42a10500	100	0	All PCs	1	1	1
TCPWM0_GRP2_CNT0_CNT	42a20000	100	0	All PCs	1	1	1
TCPWM0_GRP2_CNT1_CNT	42a20100	100	0	All PCs	1	1	1

(table continues...)

Table 50 (continued) PPC regions

PPC region name	Start address(hex)	Size (hex)	SECURITY_AWARE	PC_MASK	NS (NS_ATT)	S_P (S_P_ATT)	NS_P (NS_P_ATT)
TCPWM0_GRP2_CNT2_CNT	42a20200	100	0	All PCs	1	1	1
TCPWM0_GRP2_CNT3_CNT	42a20300	100	0	All PCs	1	1	1
TCPWM0_GRP2_CNT4_CNT	42a20400	100	0	All PCs	1	1	1
TCPWM0_GRP2_CNT5_CNT	42a20500	100	0	All PCs	1	1	1
TCPWM0_TR_ALL_GF_TR_ALL_GF	42a80000	40	0	All PCs	1	1	1
TCPWM0_TR_ALL_SYNC_BYPASS_TR_ALL_SYN_BYPASS	42a90000	4	0	All PCs	1	1	1
TCPWM0_BOOT	42a90800	4	0	PC0	0	1	1

14 Secure boot

Secure boot is the process of obtaining, validating, and starting the product firmware. The BootROM in CYW20829 consists of ROM_BOOT, which is the first code executed after any type of reset. This ROM_BOOT code is stored in the on-chip ROM and is accessible from the CPU subsystem. The following chapter describes each stage of the secure boot process in detail.

Basic definitions

The following terms and definitions will help you understand the terminology used throughout this document. Many of these terms will be explained in more detail within this section:

- **Chain of Trust (CoT):** The root of trust begins with the Infineon code residing in ROM, which cannot be altered. The Chain of Trust is established by validating software blocks, starting from the root-of-trust ROM_BOOT
- **Debug access port (DAP)**: An interface between an external debugger or programmer and the CYW20829 MCU, used for programming and debugging
- **Device identifier composition engine (DICE)**: A security standard created by Trusted Computing Group (TCG) that works by layering the boot process and creating secrets unique to each layer and configuration, based on the Unique Device Secret (UDS)
- **Digest**: A value calculated by applying a hash function to a data set, such as the binary of an application
- **Digital signature**: The encrypted form of a digest (hash of a data set). For example, the encrypted hash of a user application
- **Entrance exam**: A test performed prior to provisioning to verify that the device is genuine and in a known state
- **Hash**: A cryptographic algorithm that generates a unique, repeatable signature for a given block of data. This function is not reversible
- **Hardware security module (HSM)**: A physical computing or programming device that protects and manages digital keys. It is capable of performing digital software signing, encryption, and other cryptographic functions
- **Hardware unique key (HUK)**: A 256-bit device root key used to derive all device-specific keys
- **Inter-processor communication (IPC)**: Hardware used to facilitate communication between secure and non-secure memory
- **Lifecycle stage (LCS)**: The security mode in which the device is operating
- **Memory protection controller (MPC)**: A controller that isolates parts of memory using protection contexts
- **Master security controller (MSC)**: A controller that protects transactions initiated on the bus
- **Non-volatile counter (NV counter)**: A security counter that offers anti-rollback protection
- **One-time programmable (OTP) memory**: Memory that can be written to only once and cannot be modified afterward
- **Protection context (PC)**: Refers to the Protection Context state in this document, although it is commonly used to refer to the Program Counter. This parameter allows each bus master a security state level from 0 to 15. A bus master can be assigned a static or dynamic PC value during application execution. Protection Context provides a precise way of applying memory restrictions. PC=0 is a special state allowing any bus master full access to the entire memory space, including registers. The PC state works together with protection units
- **Public-key cryptography (PKC)**: Also known as asymmetrical cryptography. This encryption technique uses a paired public and private key algorithm for secure data exchange. The private key is used for encryption and must remain secure, while the public key, used for decryption, can be shared widely
- **Public key**: A key that can be shared but must be authenticated and secure to ensure it is not modified
- **Private key**: A secure key that must be protected to prevent unauthorized access. It is used to encrypt data, which can then be decrypted using the associated public key
- **Policy**: A collection of predefined (name, value) pairs that describe what actions and operations are allowed or disallowed on the device to which the policy applies

- **Peripheral protection controller (PPC)**: Configures peripherals to ensure they are accessible only in preconfigured protection contexts assigned by the Root-of-Trust (RoT)
- **Provisioning**: The process of injecting keys, policies, and secrets into the device. Once provisioned, the device can be accessed or modified only with the injected keys, adhering to the relevant policies
- **Platform security architecture (PSA)**: A set of firmware security specifications and open-source reference implementations to help incorporate best-practice security into products
- **Return merchandise authorization (RMA)**: A process for returning devices, typically for repair or replacement
- **Security evaluation standard for IoT platforms (SESIP)**: A security evaluation standard for Internet of Things (IoT) platforms

14.1 Features

The CYW20829 MCU provides the following secured boot features:

- Each boot stage verifies the next-stage image before launching it, thereby establishing a chain of trust up to the first user application
- Implements a hardware root-of-trust with an immutable trust anchor using a pre-programmed multistage BootROM
- Provides a conventional development lifecycle stage (LCS), NORMAL_PROVISIONED, for initial use, and an advanced secured LCS
- Includes a built-in immutable key storage area that can be loaded with required OEM keys and other user keys during provisioning
- Supports the dual-bank feature, which can be enabled or disabled during provisioning
- Provides protected firmware support, which is enabled in the device based on wounding settings
- Supports secure debugging, secure provisioning, and secure firmware updates from Infineon to users
- Allows device provisioning or firmware upgrades through the SWD interface

14.2 Architecture

14.2.1 Policy and key

Policies are predefined name-value pairs that define permitted behavior on the device. The hardware root-of-trust firmware in the device (ROM_BOOT) enforces most of the policies. However, some policies are interpreted and enforced by higher layers of software, such as the OEM bootloader and TrustZone software.

Infineon-owned keys are generated and loaded into the device before it leaves the Infineon factory. Additional keys can be loaded into the device during OEM provisioning.

14.2.1.1 Policies

- **IFX_POLICY**: IFX_POLICY is initially provisioned at the Infineon factory. It describes the debug capabilities, pre-programmed Infineon assets, and securely bootable and upgradable pre-programmed software. These policies are factory programmed and locked by Infineon
- **PROT_FW_POLICY**: This policy consists of PROT_FW_ROT_KEYS, MPN_ROT_KEY_REVOCATION, and protected firmware attributes. PROT_FW_POLICY is provisioned during the NORMAL_PROVISIONED lifecycle stage (LCS)
- **OEM_POLICY**: OEM_POLICY is provisioned by the OEM. This policy includes the OEM_ROT_KEY and other assets. The policy can be changed only if the re-provisioning flag is set to True. If the re-provisioning flag is set to False, the policy can no longer be updated. See the Provisioning user guide for details

14.2.1.2 Keys

All of the keys described in this section are asymmetric ECC 256-bit ECDSA key pairs. Each key pair consists of a private key and a public key.

- **IFX_ROT_KEY**: IFX_ROT_KEY is an asymmetric ECC 256-bit ECDSA key pair used to validate the assets signed and installed by Infineon during the initial manufacturing process. Infineon securely stores the IFX_ROT_PRIVATE_KEY in a hardware security module (HSM) under internal control and stores the IFX_ROT_PUBLIC_KEY in the device's one-time programmable (OTP) memory. All Infineon assets are signed using the IFX_ROT_PRIVATE_KEY and validated using the corresponding IFX_ROT_PUBLIC_KEY stored on the device
- **PROT_FW_ROT_KEY**: PROT_FW_ROT_KEYS are used to authenticate protected firmware. This key can be revoked using PROT_FW_ROT_KEY_REVOCATION
- **OEM_ROT_KEY**: OEM_ROT_KEYS are used to authenticate OEM application images. By default, OEM_ROT_KEY0 is used to authenticate the images. If required, the OEM can perform re-provisioning to switch to OEM_ROT_KEY1. However, re-provisioning can be performed only once and cannot be reversed

14.2.2 Lifecycle stages (LCS)

CYW20829 MCUs have configurable, non-volatile lifecycle stages. A lifecycle stage can only progress to the next stage and cannot be reversed. For example, once the lifecycle moves from NORMAL_PROVISIONED to Secured, it cannot return to NORMAL_PROVISIONED. The device can remain in the Secured state or transition to return merchandise authorization (RMA) as determined by the OEM.

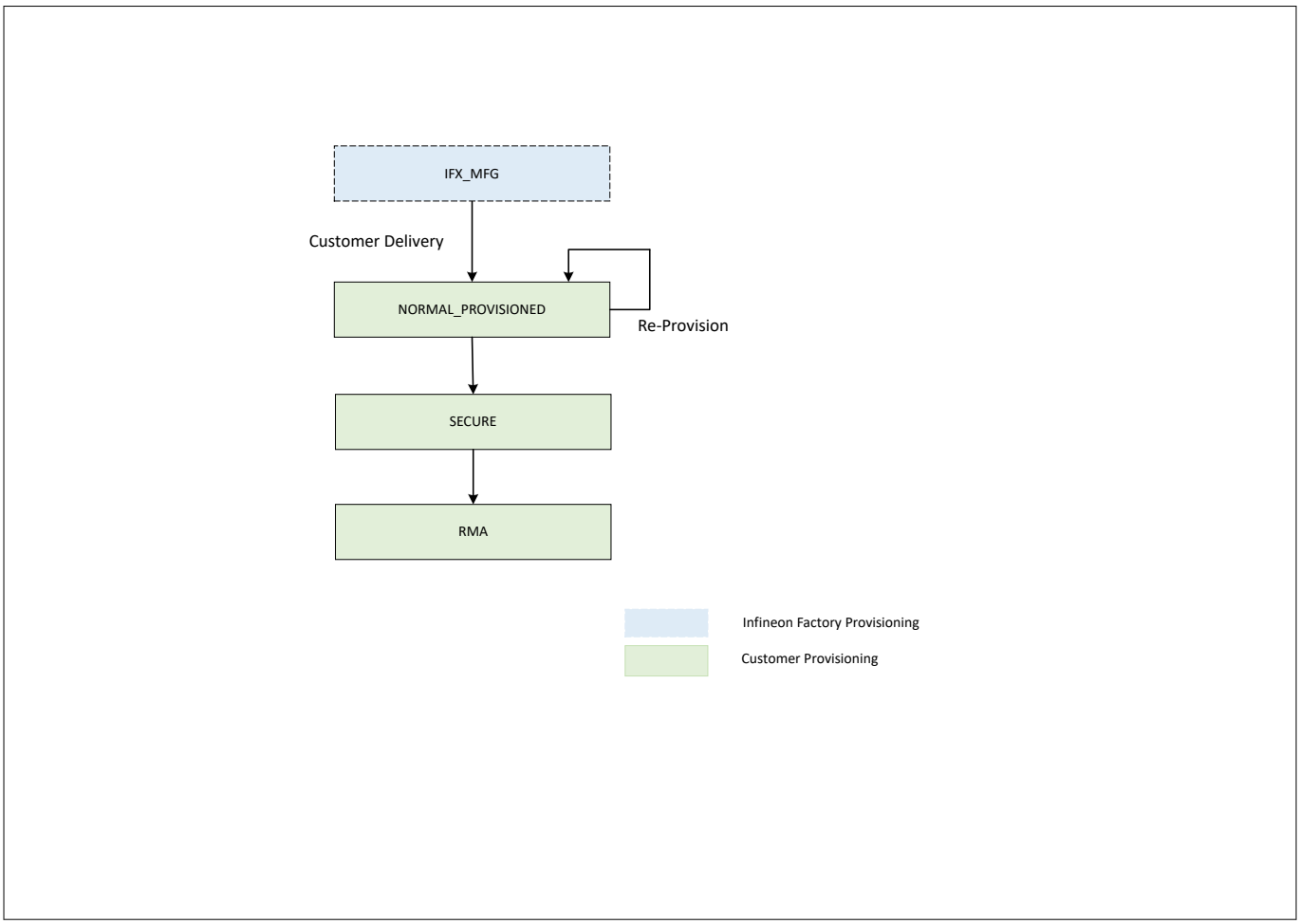


Figure 41 Lifecycle stages

14.2.2.1 IFX_MFG

In the Infineon fabrication facility, parts are produced, tailored, and programmed in the IFX_MFG lifecycle stage (LCS). All necessary testing and trimming are performed in this LCS, and Infineon assets are loaded onto the device. The device is transitioned to the NORMAL_PROVISIONED LCS before being shipped from the Infineon factory.

14.2.2.2 Normal provisioned

By the time the device reaches this state, all trimming and factory testing are completed. Devices are shipped from Infineon in this LCS.

- During NORMAL_PROVISIONED, OEM assets must be provisioned
- The device can be provisioned any number of times in this state
- The OEM_ROT_KEY can be replaced as necessary

14.2.2.3 Secured

This is the in-field operational LCS of a secured device. Before transitioning to this phase, signed, valid application code must be programmed, and the hash of the OEM keys must be provisioned (OEM_ROT_KEYS). In this LCS, the device will boot only if the application is successfully authenticated. If authentication fails at any stage, the next application will not boot, and the device's boot process will terminate.

The device can transition to Secure by blowing OTP bits. The only way to transition to Secure is by invoking the TransitionToSecure RAM application, which performs the following tasks:

- Provisions OEM_ROT_KEYS
- Verifies that all policies are provisioned with authorized production keys
- Writes the SECURE_HASH to the OTP
- Writes the OTP bit to irreversibly move the device to the Secured state

Debug access restrictions must be defined before entering the Secured LCS. For example, the Debug Policy can be set to an immutable No-Debug mode to permanently disable debugging in this LCS. Alternatively, the Debug Policy can be set to 'allow-debug' if debugging capabilities are required in this LCS. Devices with debugging enabled can be screened to ensure they are not used as production parts.

14.2.2.4 RMA

Devices can transition to RMA mode when customers require Infineon to perform failure analysis (FA). The device can be moved to RMA mode by invoking the TransitionToRMA RAM application (RAMApp). After invoking the RAMApp and transitioning to the RMA LCS state, all OEM-owned assets on the device are erased.

In RMA mode:

- The device can no longer be provisioned
- When the device boots, it does not attempt to execute any code in user flash
- The device instead waits for the OpenRMA token, provided through the debug port, signed by Infineon, to enable further internal debugging and analysis of the device

14.2.3 Boot sequence

The secured boot process for CYW20829 begins with ROM_BOOT. As the root-of-trust (RoT), it supports the next-level software authentication using public-key cryptography, ensuring that only authorized software is loaded and executed on the device.

14.2.3.1 Protection states

A lifecycle stage is a strict, irreversible progression determined by writing OTP bits. In addition to this, a boot-time protection setting, known as protection states, is applied by the CYW20829 secure boot.

During every boot, protection settings (MPC and PPC) of the device are applied by the boot process. If a critical error, such as hardware initialization failure or corruption or mismatch of any Infineon-provisioned asset, is detected during the boot, the device enters the CORRUPTED protection state.

Secure boot transitions to the DEAD protection state and halts the execution of additional code if it detects non-critical errors, such as a debug policy verification failure or an OEM Boot verification failure. Access restrictions are enforced in the DEAD protection state for the CM33 core in accordance with the COMMON debug policy.

14.2.4 CM33_L1_BOOT

14.2.4.1 Secured boot

The ROM_BOOT provides the fundamental security features required to establish a secure system. When a device is shipped from Infineon in the NORMAL_PROVISIONED LCS, secured boot is disabled in the policy, and the OEM Bootloader boots to the next application without validating it. Secured boot can be enabled in the NORMAL_PROVISIONED LCS by provisioning. Alternatively, when the device transitions to the Secured LCS, secured boot is enabled automatically.

When secured boot is enabled in the policy:

- The OEM Bootloader validates the first application using OEM_ROT_KEY_PUBLIC
- A chain of trust is established between Infineon-signed firmware and the first user application to provide the highest level of security

Protection states:

A lifecycle stage is a strict, irreversible progression dictated by writing one-time-program (OTP) eFuse bits.

During every boot:

- Protection settings (MPC and PPC) are applied by the boot process
- If a critical error, such as hardware initialization failure, corruption of any Infineon-provisioned asset, or mismatch detection, occurs, the device enters the CORRUPTED state

If secured boot is enabled and authentication of the first user application image fails:

- The device enters the DEAD_BRANCH state
- To recover from this state, the user must re-provision the device with the correct OEM assets in the development lifecycle stage

When the device encounters a debug token verification failure:

- Access to the debug port is denied
- To configure the debug policy, the user must modify the debug section of the OEM Provisioning Policy

14.2.4.2 Alternate serial interface

The BootROM provides simple serial interfaces for performing various operations during manufacturing, including provisioning, downloading protected firmware, and deploying customer applications. These processes use Infineon's proprietary simple device firmware update (DFU) over a serial protocol for communication.

By default, the device firmware update (DFU) feature is not enabled in the original equipment manufacturer's (OEM) policy. To enable this feature, the user must use the serial wire debug (SWD) interface.

Different serial interfaces can be selected through ports P2.2 and P2.3, as shown in Table 51. These ports establish communication between the host (DFU Host Tool) and the CYW20829 MCU target, in accordance with the serial interface pins table.

Table 51 DFU serial interface selection

DFU serial interface selection	P2.3 = Vss	P2.2 = Vdd	P2.3 = High-Z
P2.2 = Vss	Serial Disable	UART	Serial Disable
P2.2 = Vdd	I²C	SPI	Serial Disable
P2.2 = High-Z	Serial Disable	Serial Disable	Serial Disable

***Note**: To set the DFU select pins (P2.2, P2.3 high or low, use external pull-up or pull-down resistors with a resistance of 1 kΩ or less.*

The following default configurations are used for DFU communication protocols:

- **SPI**: Mode = Slave, Motorola 00 (CPHA=0, CPOL=0), Speed ≤ 12 Mbps
- **UART**: Baud = 115200, Bits = 8, Stop Bits = 1, RTS/CTS = No, Parity = None
- **I²C**: Speed = 400 Kbps, Mode = Slave, 7-bit Address, Address = 0x35

Table 52 Serial interface pins

Pin	UART	SPI	I²C
P8.0	-	SS	-
P8.1	Rx	MOSI	SCL
P8.2	-	MISO	-
P8.3	Tx	CLK	SDA

The BootROM DFU buffer size determines the maximum packet size that can be used for communication. The following limits apply to each protocol:

- **SPI, UART**:
  - Input buffer size (host write): 512 bytes payload + 7 bytes protocol overhead
  - Output buffer size (host read): 128 bytes + 7 bytes
- **I²C**:
  - Input buffer size (host write): 256 bytes + 7 bytes
  - Output buffer size (host read): 128 bytes + 7 bytes

14.2.5 Secured provisioning and re-provisioning

Provisioning is the process of configuring a device with an authorized set of keys, certificates, credentials, and firmware images. Provisioning is performed using provisioning applications (RAM Applications) that are designed to run from the device's RAM. These applications program Infineon and OEM assets into the OTP and advance the device LCS to either PRODUCTION_SECURE or NORMAL_PROVISIONED. The device is then ready to launch the user application in these LCS states.

The following rules apply to provisioning based on the LCS:

- **Development LCS**: Allows you to re-provision OEM assets any number of times during the development phase
- **Production LCS**: Does not allow re-provisioning of OEM assets. However, it allows a one-time revocation of the OEM_ROT_KEY

14.2.6 Protection context used in secure boot

Although Protection Context (PC) is not a single hardware block, it is used by all protection units, including MPCs, PPCs, and MSC. Multiple protection contexts are utilized to isolate different security levels within the CYW20829 device. The secure boot process specifically makes use of PC0, PC1, and PC2.

15 Power supply and monitoring

The CYW20829 family supports an operating voltage range of 1.7 V to 3.6 V. It integrates multiple regulators, including an on-chip DC-DC buck converter, which offers a selectable core operating voltage (VCCD) of 1.0 V or 1.10 V to power internal blocks in various power modes. The device supports multiple power-supply rails VDDQ, VCC_BUCK, VDDD, VDDIO, VDDA, VCCPA, VCCRF, VCCI, and VCCD enabling applications to use dedicated supplies for different internal blocks. For example, VDDA powers analog peripherals, such as the ADC and MIC.

The CYW20829 family includes the following power-monitoring and failure-protection features:

- Power-on-reset (POR)
- Brownout detection (BOD)
- Over-voltage protection (OVP)
- Low-voltage detection (LVD)

15.1 Features

The power-supply subsystem of CYW20829 includes the following features:

- Operating voltage range of 1.7 V to 3.6 V
- User-selectable core logic operation at 1.0 V and 1.1 V
- Nine supply rails (VDDQ, VCC_BUCK, VDDD, VDDIO, VDDA, VCCPA, VCCRF, VCCI, and VCCD) for CYW20829 core peripherals
- Multiple on-chip regulators
- One buck-switching regulator
- Two low-dropout regulators (LDOs) supplied by the switching regulator
- One high-voltage regulator supplied directly from VDDD
- Two POR/BOD circuits to monitor VDDD and VCCD
- LVD circuit to monitor VDDD
- OVP circuit to monitor VCCD

15.2 Architecture

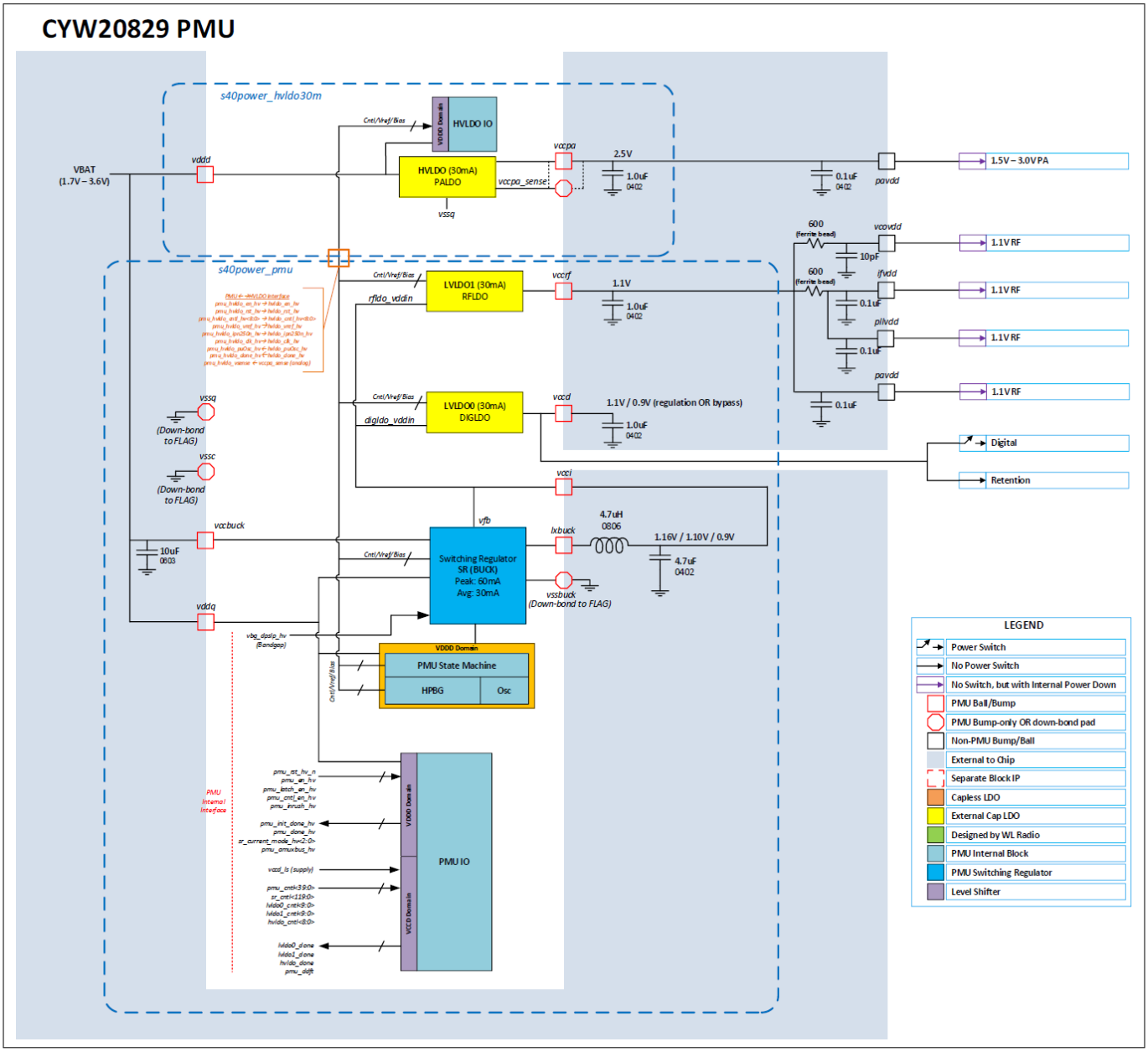


Figure 42 CYW20829 power-supply and monitoring-system block diagram

See the device datasheet for the required values of capacitors and the inductor shown in Figure 42.

The regulators and supply pins/rails in the power-supply and monitoring-system block diagram provide power to various blocks within the device:

- The high-voltage regulator (HVLDO) draws input power directly from the VDDD supply pin
- The switching regulator (SR buck) draws input power from the VCCBUCK supply pin
- Both low-voltage regulators (LVLDOs) draw input power from the output of the switching regulator
- The VCCD supply powers all active-domain and high-frequency peripherals
- The hibernate domain does not include any regulators; peripherals in this domain, such as the PILO, operate directly from the VDDD supply

The PMU includes:

- An internal bandgap reference for use during high-power operation
- An external low-power bandgap reference for low-power modes (sleep and retention)

The PMU does not require external clocking.

The I/O cells operate from various VDDIOx supply pins depending on the port in which they are located:

- The VCCD supply powers the logic inside the I/O cells related to core peripherals
- The VDDA supply powers analog logic, such as the analog mux switches inside the I/O cells

To know which I/Os operate from each supply, see the device datasheet.

In addition to the power rails and regulators, the device monitors supply rails and provides protection against supply failures through the following circuits:

- POR circuit
- BOD circuit
- OVP circuit
- LVD circuit

The Sleep Control block includes the circuits required for entering and exiting low-power modes effectively.

15.3 Power supply

15.3.1 Core regulators

The device includes the following core regulators to power peripherals and blocks in various power modes:

Low-voltage step-down regulators (LVLDO):

- The device includes two LVLDOs to generate core rails for system usage
- LVLDO0 generates the core voltage (VCCD) required for System Low Power (LP) and Ultra-Low Power (ULP) modes, which operate at 1.1 V and 1.0 V, respectively
  - LVLDO0 can operate in either Regulation or Bypass mode
  - Bypass mode allows the switching regulator (SR) input voltage to bypass the LDO directly to the output and can be used in Deep Sleep mode at 0.9 V
- LVLDO1 generates the RF voltage (VCCRF) for the Bluetooth® LE radio
  - The Bluetooth® LE radio requires a 1.1 V VCCRF for operation

Switching (Buck) Core regulator (SR):

- CYW20829 includes a switching (buck) core regulator to regulate the intermediate voltage rail (VCCI) and its derivatives `digldo_vddin` and `rfldo_vddin`
- It operates in two modes depending on system needs:
  - **High-power mode**: Supports active-mode operation over a wide current range (100 µA to 100 mA and above) with low ripple, suitable for high-performance or radio operation
  - **Low-power mode**: Supports retention operation with high efficiency at light load, with higher ripple tolerance, suitable for Deep Sleep and retention modes
- The buck regulator has two regulation loops optimized for different operational areas:
  - High-power: Designed for active operation across a large current range
  - Low-power: Focused on efficiency during light-load retention operation
- The buck regulator's voltage output settings for the LX_BUCK output can be configured to:
  - 1.0 V for ultra-low power mode
  - 1.1 V for low-power mode
  - 1.16 V for high-power mode

- During Deep Sleep mode, the system automatically programs the PMU and SR to output 0.9 V
- The PWR_CBUCK_CTL registers control the buck regulator
- To prevent overload conditions and potential brownout resets (BOR), ensure only necessary peripherals are active in System Low Power and System Ultra-Low Power modes. Overloading the regulator can cause output voltage drops, triggering a BOR

High-voltage step-down regulators (HVLDO):

- This regulator takes input power directly from the external supply (VDDD) and regulates it to a lower target voltage
- The HVLDO output is used exclusively for the internal power amplifier (VCCPA), supporting 10 dBm TX10 radio operation
- The operating voltage range is 2.6 V to 3.6 V
- There is no minimum load requirement for the HVLDO

***Note**: VBAT must be 100 mV higher than the set VCCPA value. For example, if hvldo_out (VCCPA) is set to 2.80 V, the minimum VBAT must be 2.90 V.*

15.3.2 SRSS reference block

High-power bandgap (HPBG)

- During SR high-power operation or profiles where the LVLDOs are operating in regulation, the HPBG is enabled to provide the necessary reference currents and voltages
- All LDOs (LVLDO and HVLDO) use a common reference voltage of 0.5 V, which is provided by the HPBG in most cases
- In cases where the system is transitioning from SR low-power to high-power operation (such as a Deep Sleep to Active transition), the reference voltage is provided by ULBG-based reference blocks instead of the HPBG

15.4 Voltage monitoring

CYW20829 provides multiple voltage-monitoring and supply-failure protection options, including power-on reset (POR), brownout detection (BOD), and low-voltage detection (LVD).

15.4.1 Power-on reset (POR)

Power-on reset (POR) circuits provide a reset pulse during the initial power ramp and monitor the VDDD and VCCD supply voltages. These circuits are active only during the initial chip power-up process and are subsequently disabled. See the CYW20829 datasheet for details on the POR trip-point levels.

15.4.2 Brownout detection (BOD)

A brownout detector (BOD) resets the system before the logic state is lost when VDDD and VCCD power levels drop. The brownout system ensures a reset is triggered before VDDD reaches the minimum operating voltage, protecting all logic, SRAM, and RRAM. The BOD is hardware-controlled and does not have a configurable register. See the device datasheet for details on the BOD trip points and the supported supply ramp rate.

15.4.3 Low-voltage detection (LVD)

An LVD circuit monitors the external supply voltage and generates an interrupt or fault while the device remains within its operating limits. Firmware can utilize the LVD interrupt to implement proactive measures for system protection. The LVD trip point can be customized and adjusted using the PWR_LVD_CTL register.

CYW20829 includes a single LVD on the VDDD rail, which is active and functional during Active and Sleep modes. However, the LVD is not available during Deep Sleep and Hibernate modes. By default, the LVD is disabled to conserve power, as its operation consumes energy.

It is important to note that the LVD does not act as a reset source. System reset functionality is handled by the BOD circuits. However, if required, firmware can initiate a system reset in response to an LVD interrupt.

15.5 Sleep control

Sleep control contains the required logic and circuits to manage power mode transitions, enabling the device to switch between Active/Sleep, Deep Sleep, and Hibernate modes, as well as to wake up from Deep Sleep or Hibernate to Active/Sleep mode. Co-processors and peripherals may continue operating independently of the CPU. To ensure these operations remain unaffected during Deep Sleep entry, the sleep control implements a quiescence request-and-response mechanism.

When a Deep Sleep entry is requested, the CPU sends a quiescence request to the PERI block, which expands this request to the relevant peripherals. These requests are communicated through dedicated interfaces called Q-channels. If any peripheral denies the quiescence request, the Deep Sleep entry is aborted.

The wakeup interrupt controller (WIC) senses inputs capable of waking the system from Deep Sleep or Hibernate modes. Upon detecting a valid wake-up event, the WIC forwards the wake-up request to the main power policy unit (PPU). The PPU executes the wakeup sequence, transitioning the device back to Active mode.

For additional information, see Device power modes.

15.6 Registers summary

Table 53 Power supply and monitoring register list

Name	Description
PWR_CTL	Power mode control register: Controls the device power mode options and allows observation of the current state
PWR_CBUCK_CTL	Core buck control register: Configures the output voltage and mode selection for the core buck regulator
PWR_CBUCK_CTL2	Core buck control register 2: Allows selection between CBUCK mode profiles
PWR_CBUCK_STATUS	Core buck status register: Provides status information for the core buck regulator
PWR_SDR0_CTL	Step down Regulator 0 control register
PWR_SDR1_CTL	Step down Regulator 1 control register
PWR_HVLDO0_CTL	HVLDO0 control register
PWR_LVD_CTL	High-voltage/Low-voltage detector (HVLVD) configuration register

(table continues...)

Table 53 (continued) Power supply and monitoring register list

Name	Description
SRSS_INTR	SRSS interrupt register: Indicates interrupt requests from the SRSS peripheral
SRSS_INTR_MASK	SRSS interrupt mask register: Controls the forwarding of interrupts to the CPU
SRSS_INTR_SET	SRSS interrupt set register: Allows firmware to set interrupts for testing purposes
SRSS_INTR_MASKED	SRSS interrupt masked register: Shows the logical AND of the corresponding SRSS interrupt register and mask register bits

16 Device power modes

The CYW20829 device supports multiple power modes to minimize average power consumption in an application. It operates in four system power modes and three CPU power modes, achieving the best compromise between low power consumption, short startup time, available peripherals, and wakeup sources.

The Peripheral Driver Library (PDL) driver supports all device power mode transitions and is the recommended method for transitioning and configuring CYW20829 power resources.

16.1 Features

CYW20829 power modes have the following features:

- Four system and three CPU power modes aimed at optimizing power consumption in an application
- System ULP mode with reduced operating current and clock frequency while supporting full device functionality
- System Deep Sleep mode with support for multiple wakeup sources and configurable amounts of SRAM retention
- System Hibernate mode with wakeup from I/O, WDT, RTC, and timer alarms

Power consumption in different power modes is controlled by using the following methods:

- Enabling and disabling clocks to peripherals
- Powering on/off clock sources
- Powering on/off peripherals and parts inside the CYW20829 device

16.2 Architecture

CYW20829 can operate in four system and three CPU power modes. These modes are designed to minimize average power consumption in an application. The power modes supported by the CYW20829, in order of decreasing power consumption, are as follows:

- **System Low-Power (LP)**: All peripherals and CPU power modes are available at maximum speed with a regulated voltage of 1.1 V
- **System Ultra-Low-Power (ULP)**: All peripherals and CPU power modes are available but with limited speed and 1.0 V voltage regulation
- **CPU Active**: The CPU is executing code in system LP or ULP mode. The clock source for the CPU core can range from a minimum of 6 MHz to a maximum of 96 MHz
- **CPU Sleep**: CPU code execution is halted in system LP or ULP mode. The clock source to the CPU core is gated in this mode
- **System Deep Sleep**: Only low-frequency peripherals are available after both the MCU subsystem (MCUSS) and Bluetooth® subsystem (BTSS) CPUs enter CPU Deep Sleep mode. This mode has one sub-mode: System Deep Sleep RAM. See Table 55 for details. The device does not enter System Deep Sleep mode until both CPUs are in their CPU Deep Sleep modes and both CPUs activate their System Deep Sleep request signals
- **System Hibernate**: This is the lowest current mode. The device and I/O are frozen, and the device resets on wakeup. Only Bluetooth® LE advertisements are possible upon waking up from Hibernate mode

Table 54 provides information about the CPU frequency and core voltage for different power modes. It helps to understand the relationship between power modes and the corresponding performance characteristics of the CPU.

Table 54 CYW20829 system power mode performance parameters

System power mode	CPU core voltage	SYSCPUSS[1] max frequency	BLE52SS[2] max frequency
System Low-Power (LP)	1.1 V	96 MHz	48 MHz
System Ultra-Low-Power (ULP)	1.0 V	48 MHz	24 MHz

1. SYSCPUSS refers to the Arm® Cortex®-M33 in the CPU subsystem
2. BLE52SS refers to the Arm® Cortex®-M33 in the Bluetooth® LE subsystem

CPU Active, Sleep, and Deep Sleep are standard Arm®-defined power modes supported by the Arm® CPU instruction set architecture (ISA). System LP, ULP, Deep Sleep, Deep Sleep RAM, and Hibernate modes are additional low-power modes supported by CYW20829.

The CPU can be selectively placed in Sleep and Deep Sleep power modes as defined by Arm®. The CPU also implements a Deep Sleep RAM (DS-RAM) mode, in which almost all circuits except RAM are powered off. Data in RAM is retained to maintain the state. Upon exit, the CPU goes through a reset but can use the data in RAM to skip software initialization.

Table 55 provides more details on each power mode.

Table 55 System state in power modes

Power mode	Description	SRAM	Available peripherals	Wakeup sources	Wakeup time	Boot	Reinitialization required
Low-Power Active	All resources are available at maximum speed (up to 96 MHz)	ON	All peripherals	NA	NA	NA	No
Ultra-Low-Power Active	All resources are available but operate at a lower speed (up to 48 MHz)	ON	All peripherals	NA	NA	NA	No
CPU Sleep	The CPU halts execution, but the rest of the system remains available.	ON	All peripherals	All peripherals	NA	Resume	No

(table continues...)

Table 55 (continued) System state in power modes

Power mode	Description	SRAM	Available peripherals	Wakeup sources	Wakeup time	Boot	Reinitialization required
Deep Sleep	The CPU halts execution. Most peripherals are turned OFF. High-frequency clocks are OFF, but low-frequency clocks are available. SRAM memory is retained, and active-mode retention registers are maintained.	Retained	1xSCB key scan RTC WDT/ MCWDT	GPIO SCB 0 WDT/ MCWDT RTC key scan Bluetooth® LE event	60 µs	Resume	No
Deep Sleep RAM	The CPU halts execution. Most peripherals are turned OFF. High-frequency clocks are turned OFF, but low-frequency clocks are available. SRAM memory is retained, but active-mode retention registers are not maintained.	Retained	1xSCB key scan RTC WDT/ MCWDT	GPIO SCB 0 WDT/ MCWDT RTC key scan Bluetooth® LE event	800 µs	Warm boot	Yes

(table continues...)

Table 55 (continued) System state in power modes

Power mode	Description	SRAM	Available peripherals	Wakeup sources	Wakeup time	Boot	Reinitialization required
Hibernate	The CPUs and clocks are turned OFF. GPIO output states are frozen. The RTC alarm and dedicated GPIO wakeup pins are available to wake up the system. The backup domain is available, but SRAM is not retained.	OFF	RTC WDT/ MCWDT GPIO P0.5, P1.4	Hibernate pins '0' and '1' RTC WDT/ MCWDT	2000 µs	Cold boot	No

16.2.1 System power modes

System power modes affect the entire device and can be combined with CPU power modes.

16.2.1.1 System Low-Power mode

System Low-Power (LP) mode is the default operating mode of the device after a reset. In LP mode, all peripherals and CPU power modes are available at their maximum speed. While in system LP mode, the CPUs may operate in any of the standard Arm®-defined CPU modes detailed in the CPU power modes section.

16.2.1.2 System Ultra Low-Power mode

System Ultra Low-Power (ULP) mode is identical to LP mode, with a performance trade-off made to achieve lower system current. This trade-off lowers the core operating voltage, which then requires a reduced operating clock frequency and limited high-frequency clock sources. Table 57 provides a list of resources available in ULP mode along with their limitations.

While in system ULP mode, the CPUs may operate in any of the standard Arm®-defined CPU modes detailed in the CPU power modes section.

Transitioning between LP and ULP modes is performed by reducing the core regulator voltage from the LP mode voltage to the ULP mode voltage. The lower voltage reduces system operating current and slows down signal speeds, requiring a lower maximum operating frequency. Refer to the Core Operating Voltage section in the Power supply and monitoring chapter for details on how to switch between LP and ULP modes.

***Note**: Before lowering the voltage, it is recommended to change (lower) the system clock frequency.*

16.2.1.3 System Deep Sleep mode

In system Deep Sleep mode, all high-speed clock sources are off. This makes high-speed peripherals unusable in this mode. However, low-speed clock sources (PILO and WCO) and peripherals may continue to operate if configured and enabled by the firmware. Additionally, peripherals that do not need a clock or receive a clock from their external interface (for example, I^2C or SPI slave) may also continue to operate if configured for system Deep Sleep operation. CYW20829 offers a configuration option to determine the amount of SRAM that remains active during Deep Sleep mode. Within this mode, there is a sub-mode called System Deep Sleep RAM. For more information, see SRAM.

SYSCPUSS and BLE52SS (Arm® Cortex®-M33) can each enter CPU Deep Sleep mode on their own. SYSCPUSS can use IPC to request that the BLE52SS Arm® Cortex®-M33 enter Sleep mode. However, the entire device enters system Deep Sleep mode only when both CPUs are in CPU Deep Sleep.

The device enters system Deep Sleep mode after the following conditions are met:

1. The LPM_READY bit of the PWR_CTL register should read '1'. This ensures that the system is ready to enter Deep Sleep. If the bit reads '0', the device will wait in system LP or ULP mode instead of system Deep Sleep until the bit is set, at which point the device automatically enters Deep Sleep mode if requested
2. Both Arm® Cortex®-M33s must be in CPU Deep Sleep. This is achieved by setting the SLEEPDEEP bit [2] of the SCR register of both CPUs and then executing the WFI or WFE instruction
3. The HIBERNATE bit [31] of the PWR_HIBERNATE register should be cleared; otherwise, the device enters system Hibernate mode

The switching buck regulator in Deep Sleep mode produces an output voltage of 0.9 V and supplies it to the LVLDO0 regulator, which is expected to have a nominal output voltage of 0.9 V as it operates in Bypass mode. For more details, see Power supply and monitoring. Table 57 provides a list of resources available in system Deep Sleep mode.

Upon waking up, the CPU that initiated the wake-up enters CPU Active mode, while the other CPU continues to remain in CPU Deep Sleep mode. Additionally, the system will transition back to either LP or ULP mode based on the mode that was active before entering system Deep Sleep, and a reset will be performed. Both CPUs may wake up to CPU Active simultaneously from the same wake-up source if so configured.

Interrupts from low-speed, asynchronous, or low-power analog peripherals can cause a CPU wake-up from system Deep Sleep mode. CYW20829 uses buffered writes, so writes to an MMIO register or memory can take a few cycles from the write instruction execution. The only way to ensure that the write operation is complete is by reading the same location after the write. It is required to follow a write by a read to the same location before executing a WFI/WFE instruction to enter CPU Deep Sleep mode.

***Note**: When a debugger is running on either core, the device stays in LP or ULP mode, and the CPUs enter CPU Sleep mode instead of CPU Deep Sleep mode.*

16.2.1.4 System Hibernate mode

System Hibernate mode is the lowest power mode of the device when external supplies are still present and XRES is de-asserted. It is intended for applications in a dormant state. In this mode, all Active LP/ULP mode regulators and the HVLDO regulator are turned off, and GPIO states are automatically frozen. Wake-up is facilitated through dedicated wake-up pins. Internal references are not available in Hibernate mode. Optionally, an RTC alarm from the watchdog timer (16-bit free-running WDT) interrupt can generate a Hibernate wake-up signal. Use the PWR_HIB_WAKE_CTL register to configure the wake-up source from Hibernate mode, for example, set the HIB_WAKE_RTC bit in the PWR_HIB_WAKE_CTL register to enable the RTC alarm wake-up from Hibernate mode.

The brownout detect (BOD) block is not available in Hibernate mode. As a result, the device does not recover from a brownout event in Hibernate mode. Do not enter Hibernate mode in applications that require brownout detection, that is, applications where the supply is not stable. Additionally, make sure that the supply is stable for at least 250 µs before the device enters Hibernate mode. To prevent accidental entry into Hibernate mode in applications that cannot meet these requirements, an option to disable Hibernate mode is provided. Set 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. Debug functionality is lost, and the debugger disconnects when entering Hibernate mode.

16.2.2 CPU power modes

The CPU Active, Sleep, and Deep Sleep modes are the standard Arm®-defined power modes supported by the Arm® Cortex®-M33. All Arm® CPU power modes are available in both system LP and ULP power modes. These CPU power modes affect each CPU independently.

16.2.2.1 CPU Active mode

In CPU Active mode, the CPU executes code, and all logic and memory are powered. The firmware can enable or disable specific peripherals and power domains depending on the application's needs and power requirements. All peripherals are available for use in Active mode. The device enters CPU Active mode following any device reset or wake-up.

16.2.2.2 CPU Sleep mode

In CPU Sleep mode, the CPU clock is turned off, and the CPU halts code execution. Note that in CYW20829, Arm® Cortex®-M33 supports its own CPU Sleep modes, and each CPU can sleep independently of the other CPU's state. All peripherals available in Active mode are also available in Sleep mode. The conditions for the processor to wake up depend on the mechanism that causes it to enter Sleep mode.

- **Wake-up from WFI or sleep-on-exit**: Normally, the processor wakes up only when it detects an exception with sufficient priority to cause exception entry. Some embedded systems might need to execute system restore tasks after the processor wakes up and before it executes an interrupt handler. To achieve this, set the PRIMASK bit to 1 and the FAULTMASK bit to 0. If an enabled interrupt with a higher priority than the current exception priority arrives, the processor wakes up but does not execute the interrupt handler until the PRIMASK is set to zero
- **Wake-up from WFE**: The processor wakes up from Wait For Event (WFE) under the following conditions:
  - It detects an exception with sufficient priority to cause exception entry
  - It detects an external event signal
  - In a multiprocessor system, another processor in the system executes a Send Event (SEV) instruction
  - Additionally, if the SEVONPEND bit in the SCR is set to 1, any new pending interrupt triggers an event and wakes up the processor, even if the interrupt is disabled or has insufficient priority to cause exception entry
- **External event input**: The processor provides an external event input signal. Peripherals can drive this signal either to wake the processor from WFE or to set the internal WFE event register to 1, indicating that the processor must not enter Sleep mode on a later WFE instruction

16.2.2.3 CPU Deep Sleep mode

In CPU Deep Sleep mode, the CPU requests the device to transition into system Deep Sleep mode. During this mode, the main system clock is turned OFF for the CPU, SRAM, and high-speed logic. The low-frequency (32.768

kHz) clock remains operational, allowing low-frequency peripherals to continue clocking. For the system to enter Deep Sleep mode, all CPUs must independently enter CPU Deep Sleep.

When the device is ready, it enters Deep Sleep mode. Upon waking up, the system returns to either Active or Sleep mode, based on the programmed behavior for the particular wake-up interrupt. A debug wake-up from CPU Deep Sleep mode returns to CPU Sleep mode.

16.2.3 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. 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.

16.2.3.1 Reset state

Reset is the device state when an external reset (XRES pin pulled low) is applied or when POR/BOD is asserted. It is important to note that Reset is not a power mode. During the reset state, all components in the device are powered down, and the I/Os are tristated, minimizing power consumption.

16.2.3.2 Off state

The off state represents the device state with no power applied. The reset and off states are discussed for completeness, as they encompass all possible modes and states the device can be in. These states can be utilized in a system to further optimize power consumption.

16.3 Power mode transitions

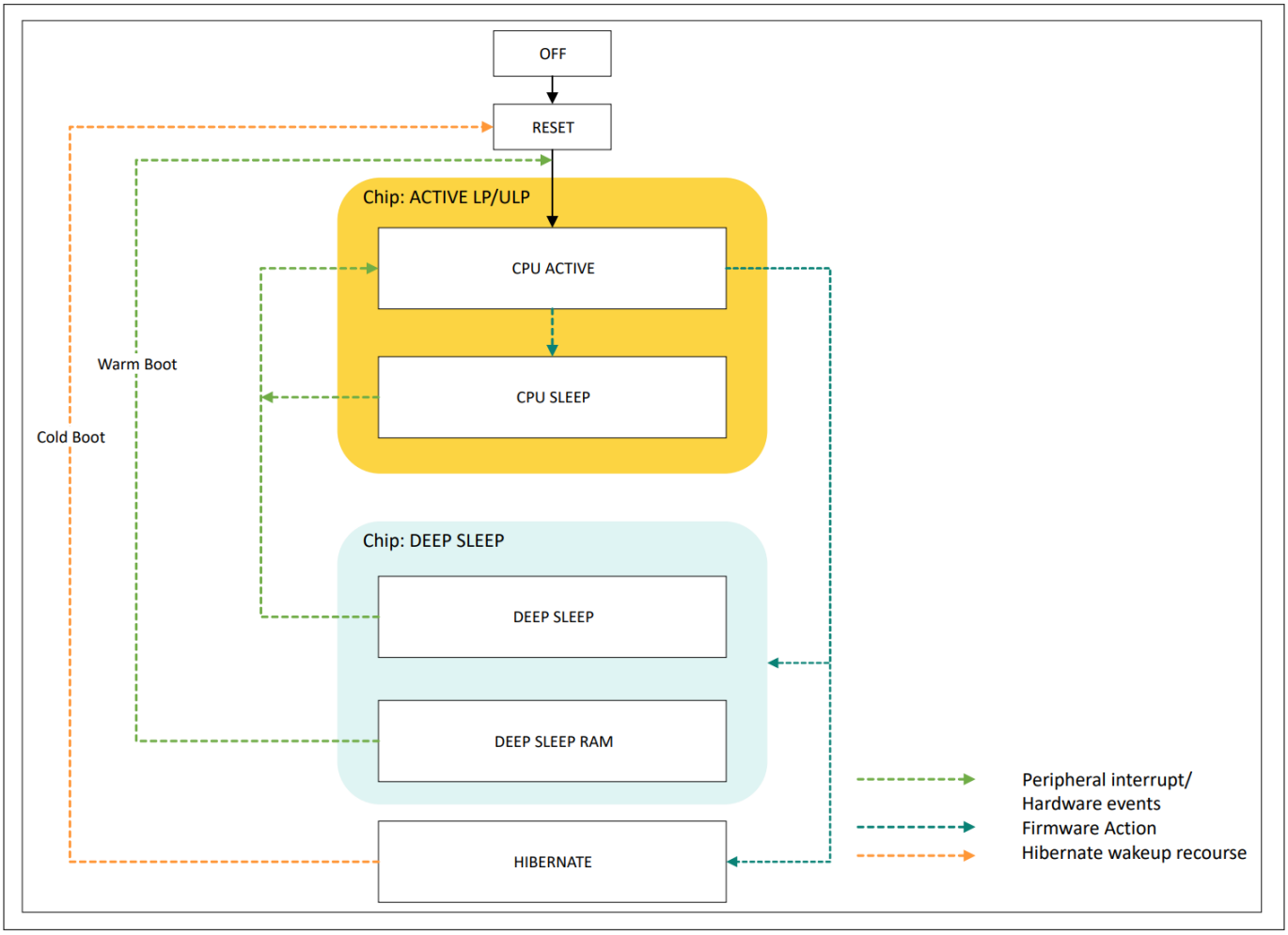


Figure 43 Power mode transitions in CYW20829

You can control power mode switching at the MCU subsystem (MCUSS) level, while the Bluetooth® subsystem (BTSS) manages its own power mode state transitions internally. For the device to enter a low-power mode, such as Deep Sleep, both CPUs must be in Deep Sleep. Figure 44 illustrates the transition between different power modes starting from power-on reset (POR).

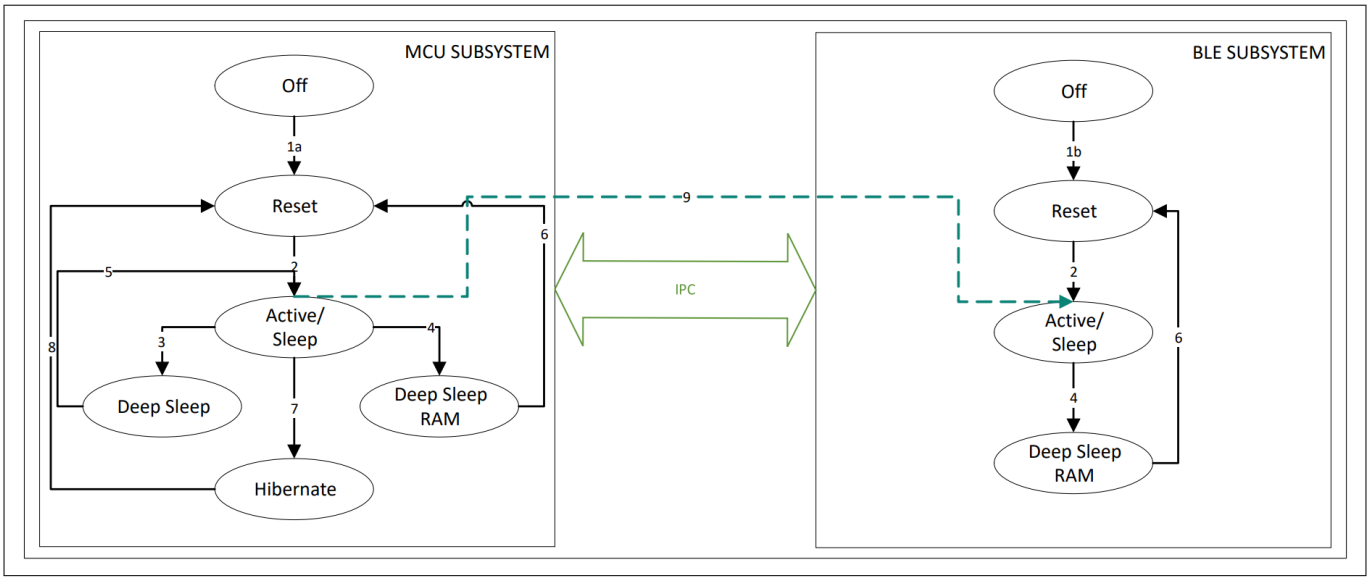


Figure 44 Power mode transitions of MCUSS and Bluetooth® subsystem (BLESS)

Table 56 Power mode transitions and triggers for MCUSS and BLESS

Transition	Description	Transit triggers	Power mode
1a	MCUSS OFF to reset transition	Power supply applied to CYW20829	Active
1b	The Bluetooth® subsystem transitions from the OFF state to the Reset state. It remains in Reset until the MCUSS releases it.	Power supply applied to CYW20829	Active
2	The MCUSS/Bluetooth® subsystem transitions from Reset to Active mode. The Bluetooth® subsystem transition occurs only after the MCUSS releases it from Reset. Each subsystem can move between Active and Sleep modes independently.	During boot/startup, the firmware transitions from Reset to Active mode. The MCUSS downloads configurations to the Bluetooth® subsystem using interprocessor communication (IPC), and both subsystems transition to Active mode.	Active
3	The MCUSS transitions to Deep Sleep, and the Bluetooth® subsystem moves to Deep Sleep RAM state. The Deep Sleep and Deep Sleep RAM power modes do not affect the operation of the Bluetooth® subsystem.	Firmware (OS idle task) transitions from Active to Deep Sleep mode	Deep Sleep

(table continues...)

Table 56 (continued) Power mode transitions and triggers for MCUSS and BLESS

Transition	Description	Transit triggers	Power mode
4	Both the MCUSS and the Bluetooth® subsystem transition to the Deep Sleep RAM state.	Firmware (OS idle task) transitions from Active to Deep Sleep RAM mode	Deep Sleep RAM
5	The MCUSS transitions from Deep Sleep to Active or Sleep mode, resuming execution from the point where it entered Deep Sleep. The Bluetooth® subsystem moves to Active mode via the warm boot handler (Path 6).	Deep Sleep wakeup sources	Active
6	Both the MCUSS and the Bluetooth® subsystem independently transition from Deep Sleep RAM to Active mode and execute their respective warm boot handlers.	Deep Sleep wakeup sources	Active
7	The MCUSS initiates the transition to Hibernate mode, which the Bluetooth® subsystem cannot ignore. The device then enters Hibernate mode.	Firmware transitions from Active to Hibernate mode	Hibernate
8	The MCUSS transitions from the Hibernate to Active mode.	Hibernate wakeup sources	Active
9	The MCUSS releases the Bluetooth® subsystem from Reset.	Bluetooth® LE stack initialization	Active

***Note**: Power mode transitions for the Bluetooth® subsystem, such as from Active to Sleep, are handled internally by the radio controller firmware. The user application does not need to manage these transitions.*

16.4 Summary

Table 57 shows the available operational states for the major blocks. Operational states in low-power modes generally have limited functionality and reduced parametric performance compared to their capabilities in active modes. Additionally, blocks that do not support low-power modes, such as System Deep Sleep and System Hibernate, cannot wake up the CPUs from these modes.

Table 57 Resources available in different power modes

Block		CPU Active	CPU Sleep	System Deep Sleep	System Hibernate
CPU subsystem	CPUs	Y	Y	N	N
	NVIC	Y	Y	N	N
	WIC	Y	Y	Y	N
	SRAM	Y	Y	Programmable	N
	DMA	Y	Y	N	N
Programmable digital	Smart I/O	Y	Y	Y	N
Fixed-function digital	TCPWM	Y	Y	N	N
	SCB	Y	Y	N, one SCB is supported in Deep Sleep mode	N
	QSPI	Y	Y	N	N
	USB	Y	Y	N	N
	CAN FD	Y	Y	N	N
Special function	Audio subsystem	Y	Y	Y(WWD)	N
Analog	ADC/Mic	Y	Y	Y (duty cycled)	N
I/O	GPIO	Y	Y	Y	Y
Bluetooth® LE subsystem	Bluetooth® LE	Y	Y	Retention	N

16.5 Registers summary

Table 58 Register list

Name	Description
PWR_CTL	Controls the device power-mode options and allows observation of the current state.
PWR_CTL2	
PWR_HIBERNATE	Hibernate Mode register
PWPR	The Power Policy Register allows software to configure both power and operating-mode policies. It also includes settings for enabling dynamic transitions and locking. Note that this register does not indicate the current power mode value.
PWCR	The Power Configuration Register, a write-only register, enables software to unlock the PPU from a locked power mode.
PTCR	The Power Mode Transition Configuration Register includes settings that influence the behavior of specific power mode transitions.

(table continues...)

Table 58 (continued) Register list

Name	Description
PWSR	The Power Status Register, a read-only register, contains status information for the power mode, operating mode, dynamic transitions, and lock feature.
PD_SENSE	The Power Domain Dependency Sense Register index <j> contains the sense bits for PD<j>.
PD_SPT	The Power Domain Dependency Support Register index <j> contains the sense bits for PD<j>.

17 Clocking system

The CYW20829 clock system consists of a combination of oscillators, an external clock, and a frequency-locked loop.

17.1 Features

The CYW20829 clock system includes the following clock sources:

- **Internal clock sources**:
  - Internal main oscillator (IMO): Fixed frequency of 8 MHz
  - Internal high-speed oscillator (IHO): Fixed frequency of 48 MHz
  - Internal low-speed oscillator (ILO): Fixed frequency of 32.768 kHz
  - Precision internal low-speed oscillator (PILO): Frequency of 32.768 kHz
  - Medium-frequency oscillator (MFO): Fixed frequency of 2 MHz
- **External clock sources**:
  - External crystal oscillator (ECO): Fixed frequency of 24 MHz
  - Watch crystal oscillator (WCO): Frequency of 32.768 kHz (optional if PILO is used instead)
- **Frequency-locked loop (FLL)**: Used for logic clocking with a frequency of up to 96 MHz

17.2 Architecture

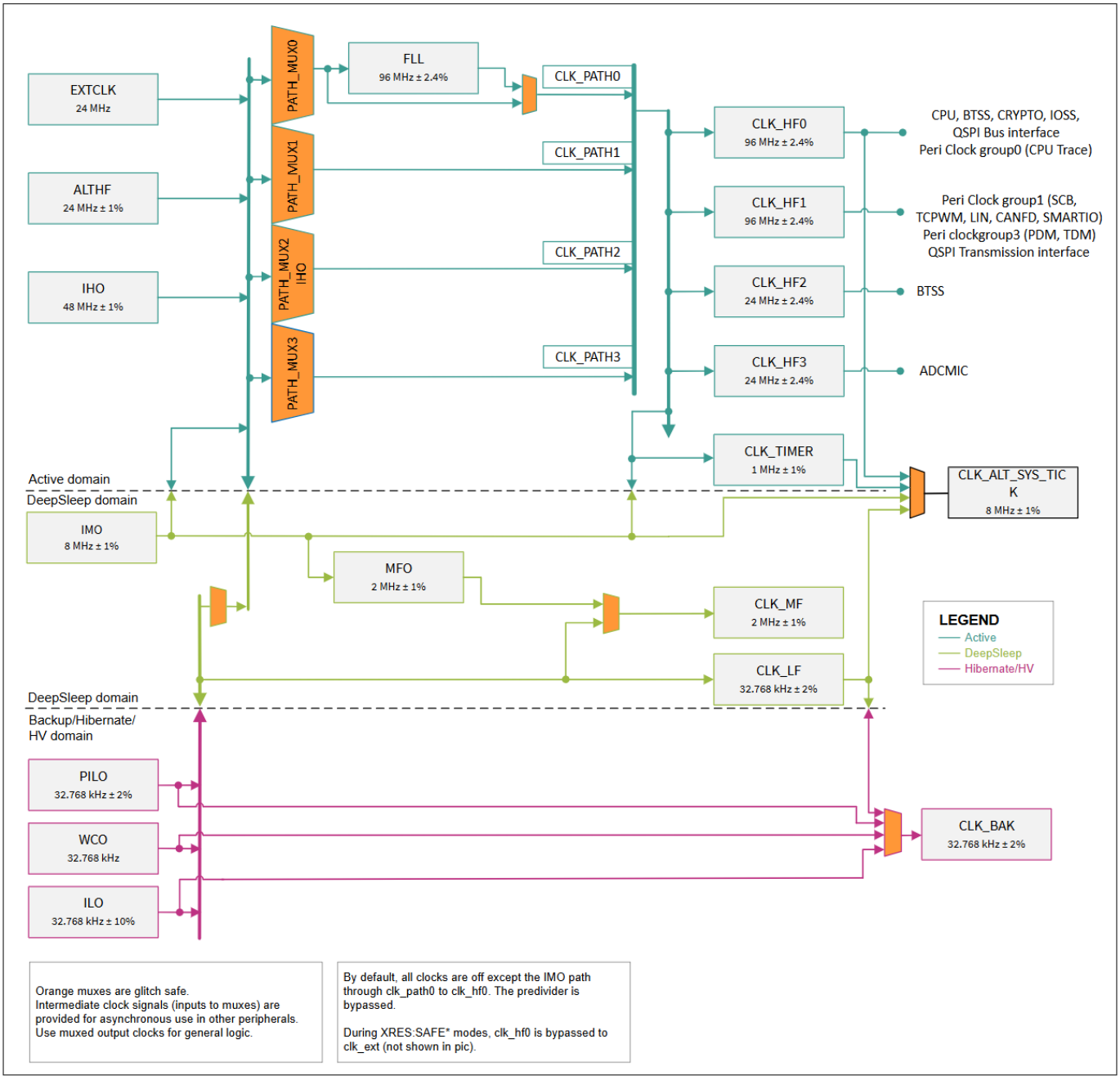


Figure 45 CYW20829 clocking diagram

***Note**: Using the PILO as the ILO clock source increases the boot time.*

17.3 Clock sources

17.3.1 Internal main oscillator (IMO)

The IMO is the primary source of internal clocking. It is trimmed during testing to achieve the specified accuracy. The default frequency of the IMO is 8 MHz, with a tolerance of ±2%. The IMO output can be used by

the FLL to generate a wide range of high-frequency clocks or can be directly utilized by the high-frequency root clocks.

The hardware automatically disables the IMO during Deep Sleep and Hibernate power modes.

17.3.2 Internal low-speed oscillator (ILO)

The ILO operates without requiring any external components and provides a stable clock output at a nominal frequency of 32.768 kHz. It is a very low-power oscillator that functions in all power modes. The ILO can be calibrated against a high-accuracy clock to improve its accuracy. It is always the clock source for the watchdog timer (WDT). Therefore, enabling the WDT will automatically enable the ILO.

17.3.3 Internal high-frequency oscillator (IHO)

The IHO is a high-speed, accurate internal oscillator (crystal-less) that generates a fixed frequency of 48 MHz without requiring any external components. It operates with a tolerance of ±1%.

17.3.4 Precision internal low-speed oscillator (PILO)

The PILO is a precision low-power oscillator that operates at a frequency of 32.768 kHz with an accuracy of 250 ppm. It is designed to replace a watch crystal in Bluetooth® LE products while supporting delay counters with sufficient accuracy to meet Bluetooth® LE requirements. The PILO functions in Hibernate mode and higher power modes. It can also continue operating during XRES to support the RTC if configured.

To enable the PILO, the user must set `CLK_PILO_CONFIG.PILO_EN = 1`. The PILO hardware is optional in CYW20829 and can be enabled during Deep Sleep and Hibernate modes. It can continue operating during XRES if the `CLK_PILO_CONFIG.PILO_BACKUP` option is enabled.

17.3.5 Watch crystal oscillator (WCO)

The WCO is a highly accurate 32.768-kHz clock source and serves as the primary clock source for the RTC. It can also be used as a source for `CLK_LF`, `CLK_MF`, and `CLK_BAK_HV`. The WCO can be enabled or disabled by setting the `WCO_EN` bit in the `CTL` register for the backup domain.

The WCO can operate from external clock or sine wave inputs. In these modes, the WCO must be bypassed by setting the `WCO_BYPASS` bit in the `BACKUP_CTL` register before enabling the WCO. Additionally, GPIOs must be configured for the `WCO_OUT` and `WCO_IN` signals in Analog mode.

***Note**: When the WCO is enabled, chip startup will take longer because clock configuration cannot proceed until the WCO is ready. The WCO may be omitted if the PILO is used as the clock source.*

17.3.6 External crystal oscillator (ECO)

The CYW20829 Bluetooth® LE subsystem uses crystal oscillator circuits to drive an external 24 MHz crystal. These circuits require external crystals to be connected to the external crystal pins of CYW20829.

Figure 46 shows the external crystal oscillator circuit for CYW20829. The component values shown are typical; refer to the ECO specifications for crystal values and the crystal datasheet for load capacitor values. Both the ECO and WCO require balanced external load capacitors. For more details, see the hardware design guidelines. Note that the performance of the oscillator is affected by GPIO switching noise.

***Note**: The 24 MHz ECO within the Bluetooth® LE subsystem supplies the SRSS via the `clk_althf` input.*

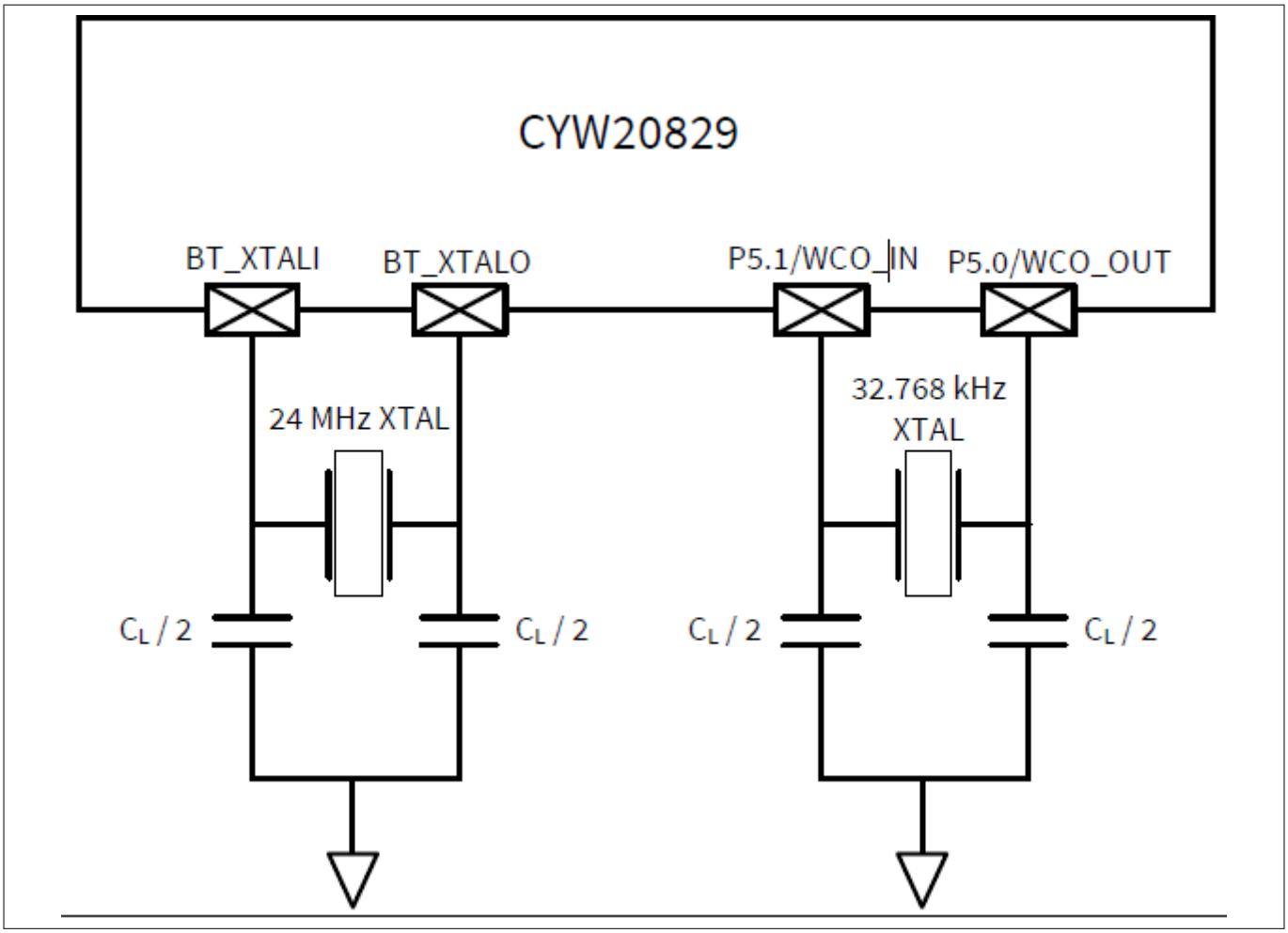


Figure 46 External crystal oscillator circuits for CYW20829 Bluetooth® LE subsystem

17.3.7 Medium frequency oscillator (MFO)

There is no physical circuit for the MFO. It is generated internally by dividing the IMO frequency by 4, which produces a 2 MHz clock.

17.3.8 External clock (EXTCLK)

The external clock is a 24 MHz clock that can be sourced from a signal on a designated I/O pin. This clock can be used as the source clock for the FLL or directly by the high-frequency clocks. When manually configuring a pin as an input to EXTCLK, the drive mode of the pin must be set to high-impedance digital to enable the digital input buffer. To determine the specific pin used for the EXTCLK, see the device datasheet.

17.3.9 Alternate high-frequency clock (ALTHF)

ALTHF is an alternate high-frequency clock. It is a high-speed clock input provided by the Bluetooth® LE subsystem, operating at 24 MHz. This clock is supplied to the SRSS through the `clk_althf` input. The ALTHF path can be used as a source for the FLL or directly as the source for the high-frequency clocks.

17.4 Clock generation

CYW20829 features fully integrated clocking mechanisms. It provides clocks to all blocks that require them, seamlessly switching between different clock sources without introducing glitches. Additionally, it ensures that no metastable conditions occur during clock transitions.

17.4.1 Frequency-locked loop (FLL)

CYW20829 includes one frequency-locked loop (FLL), which resides on clock path 0. The FLL can generate an output clock in the range of 24 MHz to 96 MHz. The input frequency for the FLL must fall between 0.040 Hz and 96 MHz, and it must be at least 2.5 times lower than the CCO frequency.

This configuration allows the IMO to generate much higher clock frequencies for the rest of the system.

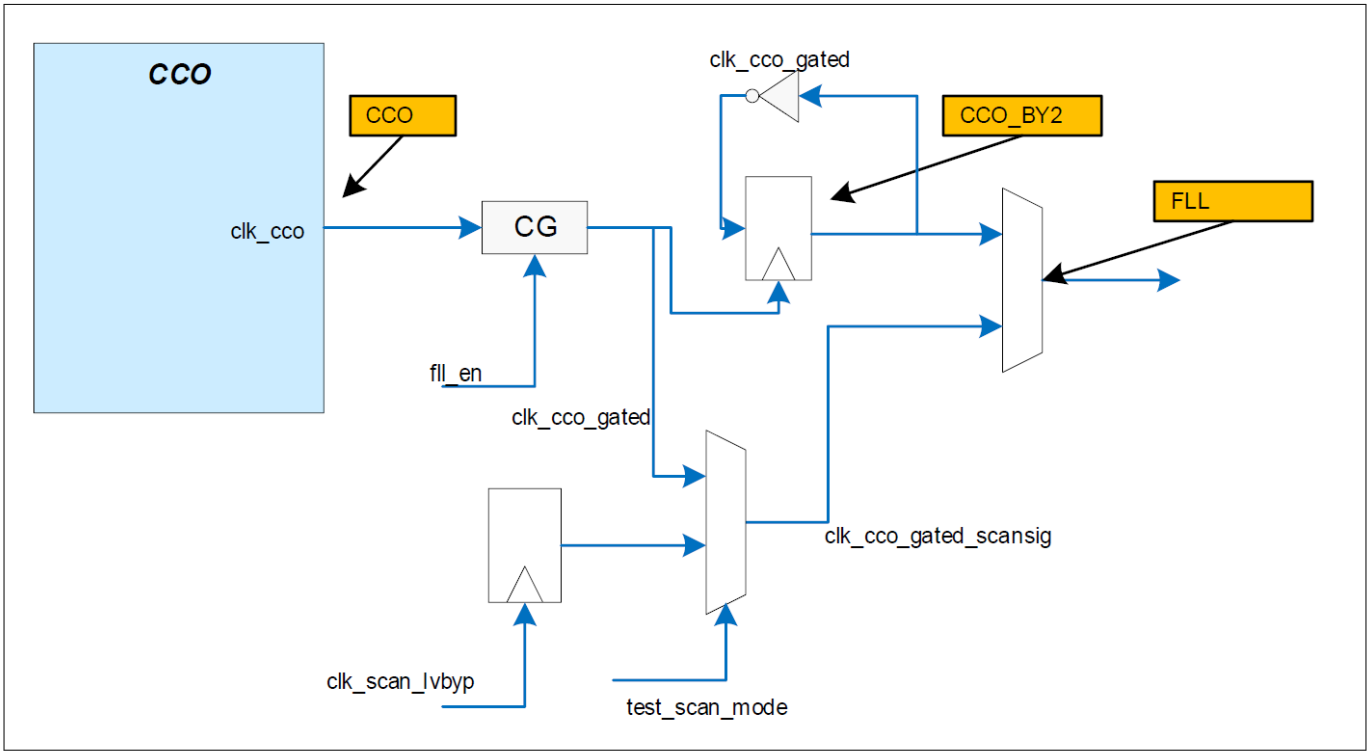


Figure 47 FLL block overview

The current-controlled oscillator (CCO) is a component of the frequency-locked loop (FLL) that generates a clock suitable for most on-chip operations. The FLL multiplies an input clock frequency but does not lock its phase. It achieves lock significantly faster than a PLL and operates with lower current consumption.

The FLL functions as a counter combined with a current-controlled oscillator (CCO). The counter measures the number of output clock edges within a reference clock period and adjusts the CCO until the expected frequency ratio is achieved (locked). After locking, the CCO is dynamically adjusted to maintain the ratio within defined control limits.

The accuracy of the output depends on two factors:

- The number of cycles used for the measurement
- The control step size of the CCO

17.5 Clock trees

The CYW20829 clocks are distributed throughout the device, as shown in Figure 45. The clock trees are described in this section and include the following:

- Path clocks
- High-frequency clock (CLK_HF)
- Medium-frequency clock (CLK_MF)
- Low-frequency clock (CLK_LF)
- Backup clock (CLK_BAK)
- Timer clock (CLK_TIMER)

17.5.1 Path clocks

CYW20829 has four clock paths:

- CLK_PATH0 contains the FLL
- CLK_PATH1, CLK_PATH2, and CLK_PATH3 have direct connections to the high-frequency root clocks

These clock paths serve as input sources for the high-frequency clock roots. Each clock path includes a multiplexer (mux) that selects the source clock for that path. The source clock configuration is controlled through the CLK_PATH_SELECT[i] register.

Table 59 Clock path source selections

Name	Description
PATH_MUX[2 : 0]	0: IMO - Internal Main Oscillator 1: EXTCLK - External R/C Oscillator 2: Reserved 3: ALTHF (24 MHz crystal oscillator from BLESS) [4 : 7]: Reserved

***Note**: Switching away from the originally selected clock requires four cycles of the current clock to complete. During this time, the original clock must remain enabled.*

17.5.2 High-frequency root clocks (CLK_HF)

The CLK_HF are the Active mode clocks and can use any of the following clock sources: IMO, IHO, ALTHF, or EXTCLK. CYW20829 features four high-frequency root clocks (CLK_HF[0 - 3]). Each CLK_HF is assigned to a specific destination on the device, as shown in Table 60.

Table 60 High-frequency root clocks (CLK_HF[i])

Name	Description
CLK_HF0	Peri clock group 0 (CPU) CPU, BTSS, CRYPTO, IOSS, QSPI bus interface
CLK_HF1	Peri clock group 1 (SCB, TCPWM, LIN, CANFD, SMARTIO). Peri clock group 3 (PDM, TDM). QSPI transmission interface
CLK_HF2	BTSS
CLK_HF3	ADCMIC

Each high-frequency root clock has a multiplexer (mux) to select its source and a pre-divider to configure its frequency. This setup is managed through the CLK_ROOT_SELECT[i] register, as shown in Table 60.

Table 61 Clock root select register

Name	Description
ROOT_MUX[3 : 0]	0: CLK_PATH0 1: CLK_PATH1 2: CLK_PATH2 3: CLK_PATH3 [4 : 13]: Reserved
ROOT_DIV[5 : 4]	0: Transparent mode, feed through selected clock source w/o dividing. 1: Divide selected clock source by 2 2: Divide selected clock source by 4 3: Divide selected clock source by 8

CLK_HF[1-3] can be enabled or disabled, while CLK_HF[0] is always enabled because it serves as the clock source for the CPU. To enable or disable CLK_HF[1-3], set or clear the ENABLE bit in the CLK_ROOT_SELECT register.

17.5.3 Medium-frequency clock (CLK_MF)

CYW20829 contains a medium-frequency clock (CLK_MF). The source for this clock, known as the medium-frequency oscillator (MFO), is derived from the IMO and operates at a fixed frequency of 2 MHz (IMO/4).

- CLK_MF is enabled by setting the CLK_MF_SELECT.ENABLE field to '1'
- CLK_MF can be pre-scaled by an integer value between 1 and 256 using the CLK_MF_SELECT.MFCLK_DIV field
- The MFCLK_DIV field should be modified only when the ENABLE field is set to '0'
- CLK_MF (and IMO) can be configured to operate in Deep-Sleep power mode by setting the CLK_MFO_CONFIG.DPSLP_ENABLE field to '1'

17.5.4 Low-frequency clock (CLK_LF)

The low-frequency clock (CLK_LF) in CYW20829 has two input options: PILO and WCO. CLK_LF serves as the source for the multi-counter watchdog timers (MCWDT) and the real-time clock (RTC). The source of CLK_LF is configured in the LFCLK_SEL bits of the CLK_SELECT register.

- Not all products support all clock sources
- Selecting an unsupported clock source results in undefined behavior
- Writes to the LFCLK_SEL field are ignored unless the watchdog timer (WDT) is unlocked using the WDT_LOCK register
- Switching from the originally selected clock takes four cycles of that clock
- The original clock must not be disabled during this switching period

Table 62 LFCLK input selection bits (LFCLK_SEL)

Name	Description
LFCLK_SEL[2 : 0]	0: Reserved 1: WCO - Watch-crystal oscillator. Requires properly configured (including external watch crystal, if used) 2: Reserved 3: PILO - Precision internal low-speed oscillator [4 : 7]: Reserved

17.5.5 Backup clock (CLK_BAK)

The backup clock provides the clock source for the backup domain, specifically for the real-time clock (RTC).

17.5.6 Timer clock (CLK_TIMER)

The timer clock is standardized as a 1-MHz fixed clock derived from the IMO and remains active during Active and Sleep modes. It is turned off in other modes, such as Deep-Sleep.

- The timer clock (`CLK_TIMER`) can be used as a clock source for the profiler
- `CLK_TIMER` can also serve as a clock source for the CPU SYSTICK timer

17.5.7 Group clocks (CLK_SYS)

On the CYW20829 device, peripherals are grouped, and each group is assigned a dedicated group clock, also referred to as `clk_sys`.

- The group clock determines the clock rate for the AHB interface on the peripheral
- It also sets the clock rate for trigger outputs and trigger input synchronization
- Each group clock has an eight-bit divider located in the `CLOCK_CTL` register in the `PERI_GROUP_STRUCT` of the `PERI` register set
- For most applications, these dividers should remain at their default value (divide by 1)

17.6 CLK_HF clock distribution network

PCLK group	Root clock (clk_hf)	Peripherals	Frequency		Description
			LP (1.1 V typ)	ULP (1.0 V typ)	
0	clk_hf0	CPU Trace	48 MHz	24 MHz	–
1	clk_hf1	SCB	96 MHz	48 MHz	Async peripherals: Strobe signals are driven through dividers; Interface clock is generated inside the peripheral with the main group clock.
		TCPWM			
		LIN			
		CANFD			
		SMARTIO			
2	clk_hf0	BTSS	96 MHz	48 MHz	Direct connection pass through from clk_hf. This clock is not used for interface clock, rather it is used for the MMIO clocks of BTSS and CRYPTO. BTSS uses this clock for Master and Slave AHB/MMIO transactions.
		CRYPTO			
3	clk_hf1	PDM	96 MHz	48 MHz	Uses PERI ACLK with default div by 2 option, required interface frequencies are obtained by further division inside the peripheral.
		TDM			
4	clk_hf2	BTSS	48 MHz	48 MHz	RPU clock for BTSS
5	clk_hf3	ADCMIC	24 MHz	24 MHz	Direct connection for ADC, main source of clk_hf3 is clk_althf which is the BTSS ECO clock.

Figure 48 Mapping of clock groups to peripherals

17.7 Peripheral clock dividers

CYW20829 peripherals, such as SCBs and TCPWMs, require a clock. These peripherals can only be clocked using a peripheral clock divider.

- There are one or more of the following types of clock dividers:
  - 8-bit clock dividers
  - 16-bit integer clock dividers
  - 5-bit fractional clock dividers
  - 24.5-bit fractional clock divider
- The output of any of these dividers can be routed to any peripheral
- Integer and fractional clock dividers are available for peripheral use and timing purposes
- Multiple peripheral clock dividers exist, effectively creating multiple separate peripheral clocks
- The 8-bit and 16-bit dividers are integer dividers

For integer dividers:

- A divider value of 1 means the output frequency matches the input frequency (no change)
- Otherwise, the frequency is divided by the value of the divider
- For example, if the input frequency is 50 MHz and the divider is set to 10, the output frequency is 5 MHz

For fractional dividers:

- Five fractional bits allow additional precision in 1/32nd increments

- For example, a divider with an integer value of 3 and a fractional value of 4 (4/32) yields a divider value of 3.125
- Fractional dividers are particularly useful when high-precision clocks are required, such as for UART or SPI serial interfaces

17.7.1 Fractional clock dividers

Fractional clock dividers allow the clock divisor to include a fractional component ranging from 0 to 31/32.

- 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 value of 16 generates a clock frequency calculated as:

  48/(3 + 16/32) = 48/3.5 = 13.7 MHz

Not all clock periods in the resulting 13.7-MHz clock are the same. Some periods will correspond to a 16-MHz clock, while others will correspond to a 12-MHz clock, creating an average frequency of 13.7 MHz.

Fractional dividers are ideal when a high-precision clock is required, such as for UART or SPI serial interfaces. However, fractional dividers are unsuitable for scenarios requiring a low-jitter clock, as the clock periods inherently have a jitter of one CLK_PERI cycle.

17.7.2 Peripheral clock divider configuration

Peripheral clock dividers are configured using registers from the peripheral block, specifically the 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 using the DIV_8_CTL, DIV_16_CTL, DIV_16_5_CTL, or DIV_24_5_CTL registers
  - Each divider has a corresponding register. For example, there are eight DIV_8_CTL registers because there are eight 8-bit dividers
  - In these registers, set the value of the integer divider. If the divider is fractional, configure the fraction portion as well
- After configuring the divider, use the DIV_CMD register to enable it
  - Set the DIV_SEL field to the divider number you want to enable
  - Set the TYPE_SEL field to the corresponding divider type

Examples:

- To enable the 0th 16.5-bit divider, set the DIV_SEL field to 0 and the TYPE_SEL field to 2
- To enable the 10th 16-bit divider, set the DIV_SEL field to 10 and the TYPE_SEL field to 1

17.7.2.1 Phase aligning dividers

For specific use cases, it is necessary to generate clocks that are phase-aligned. For example, consider generating two gated clocks at 24 MHz and 12 MHz, both derived from a 48-MHz CLK_PERI. If phase alignment is not accounted for, the resulting gated clocks appear as shown in Figure 49.

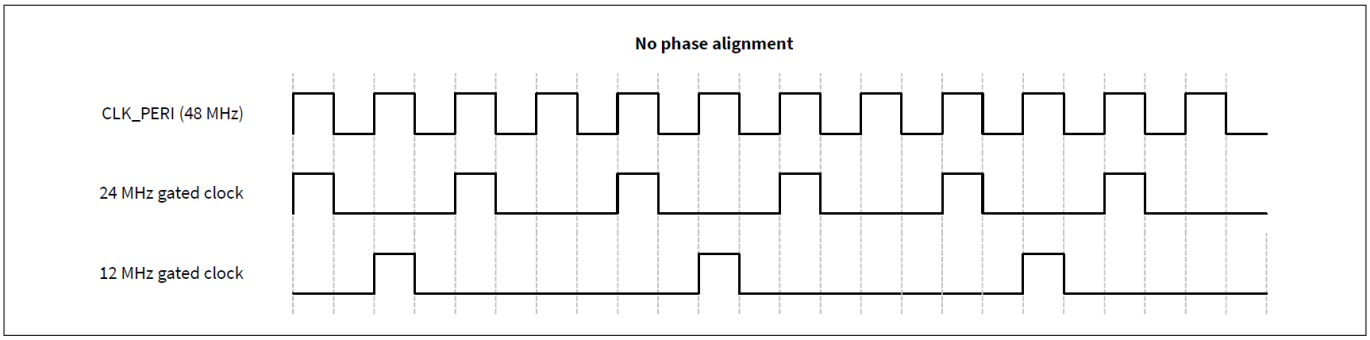


Figure 49 Non phase-aligned clock dividers

These clock signals may or may not be acceptable, depending on the logic functionality implemented between the two clocks.

For instance:

- If the two clock domains communicate with each other, and the slower clock domain (12 MHz) assumes that each high (1) pulse on its clock coincides with a high (1) pulse in the faster clock domain (24 MHz), any phase misalignment becomes unacceptable

To address this issue, dividers can be configured to produce clock signals that are phase-aligned with other enabled clock dividers. For example, if an enabled divider x is used to generate the 24-MHz clock, another divider y can be phase-aligned to divider x and used to generate the 12-MHz clock. The resulting phase-aligned clock signals appear as shown in Figure 50.

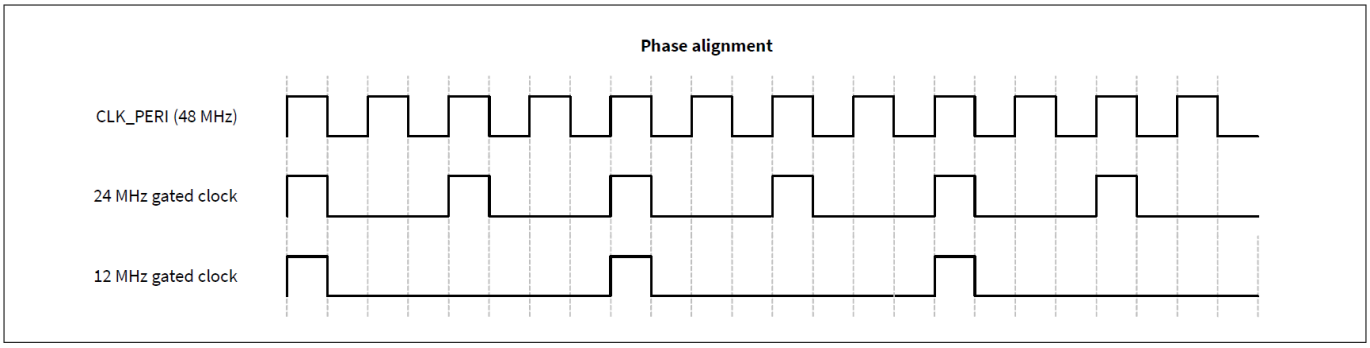


Figure 50 Phase-aligned clock dividers

Phase alignment also applies to fractional divider values. For example, if an enabled divider x is used to generate a 38.4-MHz clock (divide by 1 8/32), another divider y can be phase-aligned to divider x and used to generate a 19.2-MHz clock (divide by 2 16/32). The resulting phase-aligned gated clocks appear as shown in Figure 51.

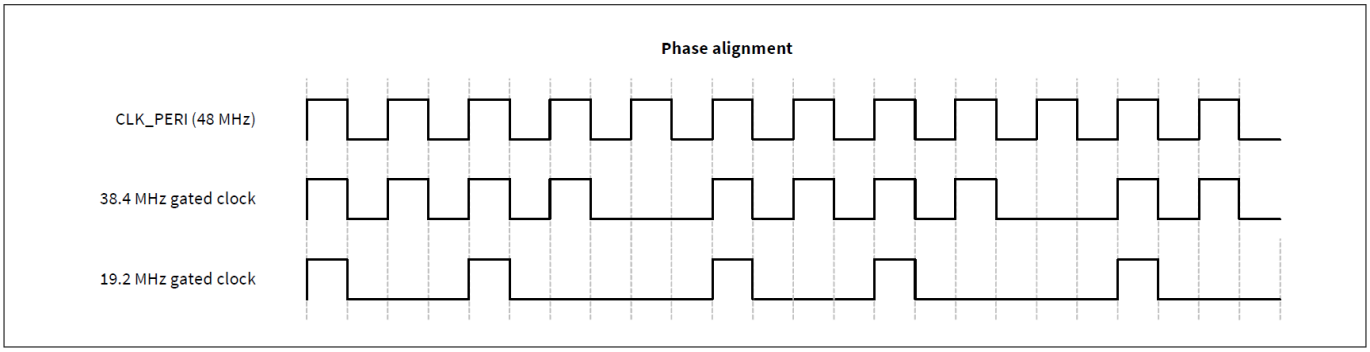


Figure 51 Phase-aligned fractional dividers

Divider phase alignment requires the target divider to be phase-aligned to an already enabled divider. This means dividers must be enabled in a specific order.

- Phase alignment is achieved by controlling the starting moment of the divider counters in hardware
  - When a divider is enabled, its counters are initially set to 0
  - The counters only begin incrementing from 0 to their programmed integer and fractional divider values when the divider to which they are phase-aligned has an integer counter value of 0
- The divider and clock multiplexer control register fields are retained during Deep Sleep power mode
  - However, the divider counters that implement the integer and fractional clock dividers are not retained. These counters are reset to 0 when entering Deep Sleep mode
  - Consequently, when transitioning from Deep Sleep to Active mode, all dividers (and clock signals) are enabled and automatically phase-aligned

Phase alignment is accomplished by configuring the PA_DIV_SEL and PA_TYPE_SEL bits in the DIV_CMD register before enabling the clock.

For example:

- To align the fourth 8-bit divider with the third 16-bit divider:
  - Set DIV_SEL to 4
  - Set TYPE_SEL to 0
  - Set PA_DIV_SEL to 3
  - Set PA_TYPE_SEL to 1

17.7.2.2 Connecting dividers to peripherals

CYW20829 features four high-frequency root multiplexers (CLK_HF[0–3]), which can be connected to programmable peripheral dividers.

For a complete list of peripherals, see Table 63.

Table 63 CYW20829 clock dividers to peripherals

Name	Peri Clock group	Description
CLK_HF[0]	0	Root clock for CPU, BTSS, CRYPTO, IOSS, QSPI Bus Interface
CLK_HF[1]	1	Root clock for SCB, TCPWM, LIN, CANFD, SMARTIO
	3	Root clock for PDM, TDM, QSPI Transmission Interface
CLK_HF[2]	-	Root clock for BTSS
CLK_HF[3]	-	Root clock for ADMIC

18 Watchdog timer (WDT)

The watchdog timer (WDT) is a hardware timer designed to automatically reset the device if the firmware follows an unexpected execution path. When enabled, the WDT must be periodically serviced in the firmware to prevent a reset. If not serviced in time, the timer will expire and trigger a device reset. Additionally, the WDT can function as:

- An interrupt source
- A wakeup source in low-power modes

18.1 Features

The CYW20829 watchdog timer (WDT) component offers the following features:

- **Free-running WDT with**:
  - CLK_LF as the input clock source
  - Periodic interrupt or wakeup generation in Active, Sleep, Deep Sleep, and Hibernate power modes
  - Device reset generation if not serviced within a configurable interval
- **Multi-counter watchdog timer (MCWDT) with**:
  - CLK_LF as the input clock source
  - Periodic interrupt or wakeup generation in Active, Sleep, and Deep Sleep power modes (Hibernate mode is not supported)
  - Device reset generation if not serviced within a configurable interval
  - Two 16-bit counters and one 32-bit independent counter, configurable as:
    - A single 64-bit counter
    - A 48-bit counter with one 16-bit independent counter
    - Two 32-bit cascaded counters

18.2 Architecture

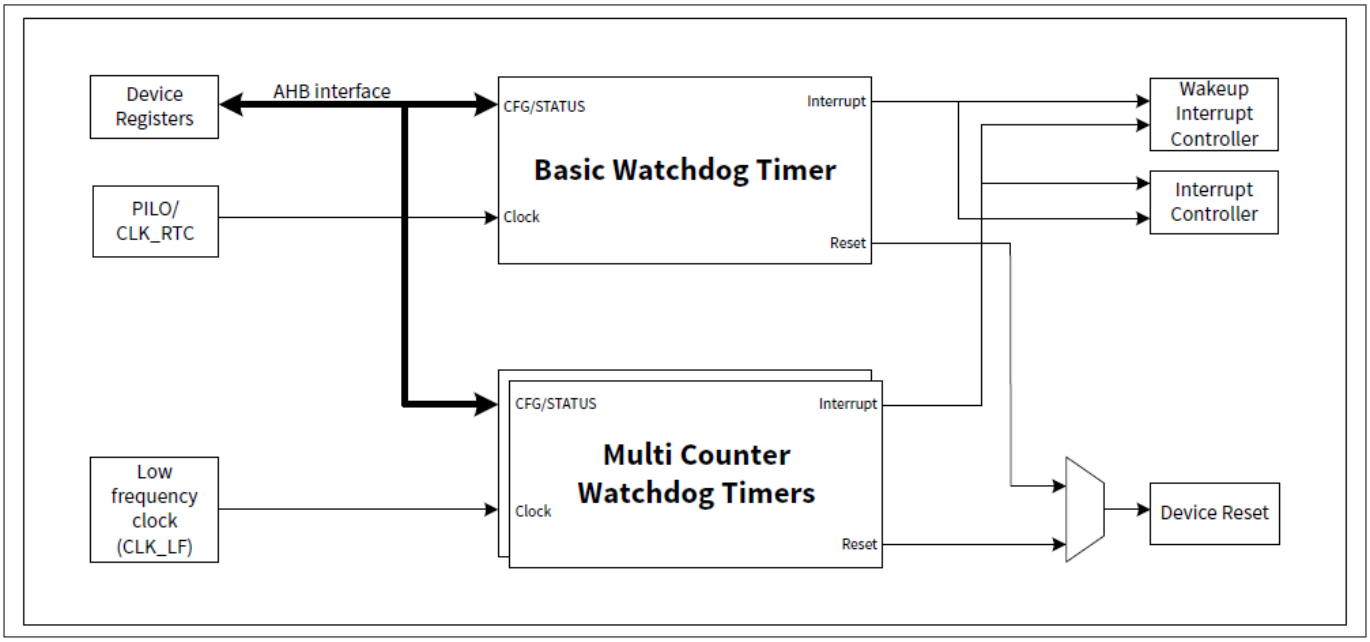


Figure 52 Watchdog timer block diagram

For details on configuring the interrupt for the free-running WDT or MCWDT to the required CPU, see Interrupts. The free-running WDT or MCWDT resource must be managed exclusively by one CPU. It is not designed for simultaneous use by both CPUs due to the complexity of coordination involved.

18.3 Free-running WDT

18.3.1 Overview

Figure 53 describes the functional overview of the free-running WDT:

- The WDT includes a free-running, wraparound up-counter with a maximum resolution of 22 bits
- By default, the counter is clocked by the PILO through the CLK_LF
- The clock source can be modified using the WDT_CTL.WDT_CLK_SEL bits
- The timer can generate:
  - An interrupt on a match event
  - A reset event after the third unhandled interrupt
- The number of bits used for match comparison is configurable, as shown in Figure 53

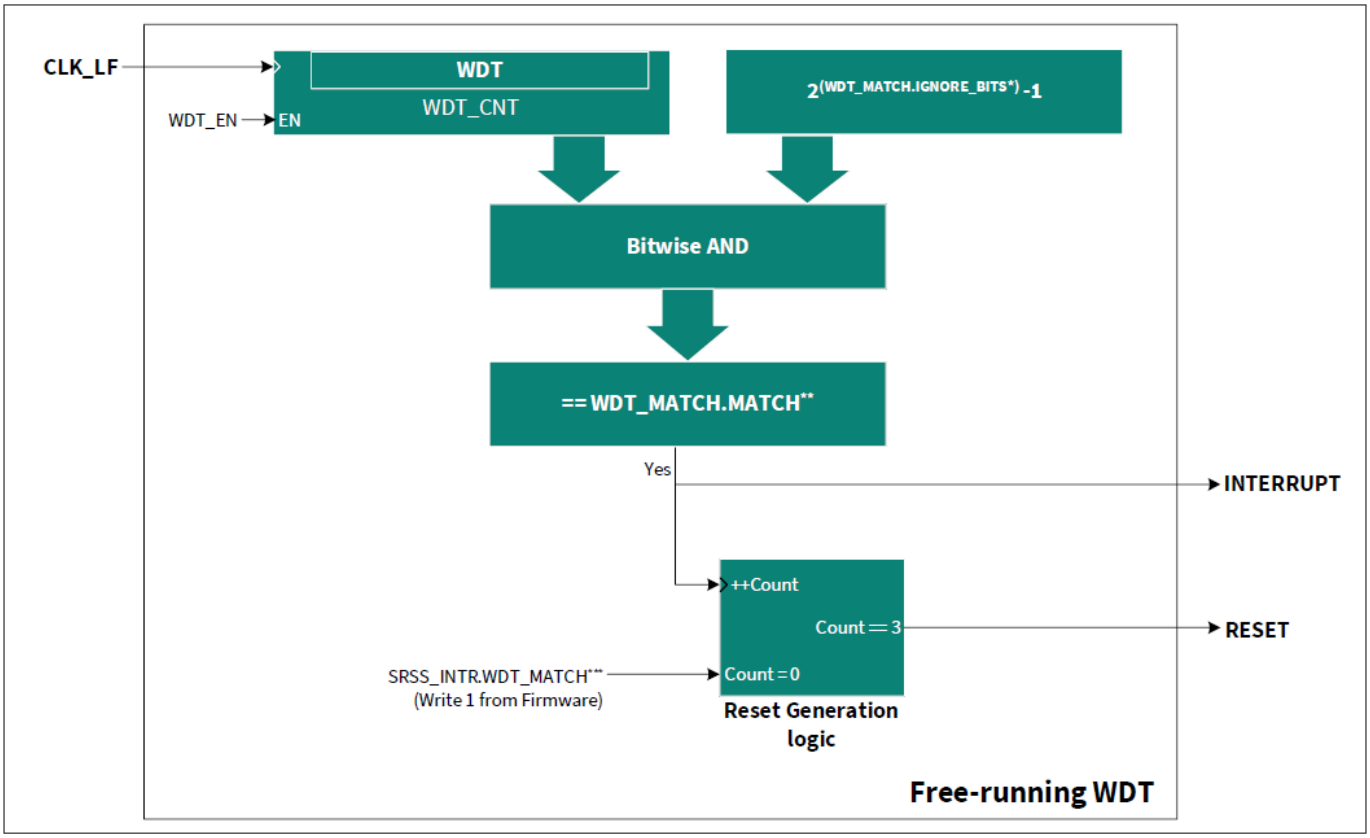


Figure 53 WDT functional diagram

WDT_MATCH2.IGNORE_BITS_ABOVE refers to the value held by bits [4:0] of the WDT_MATCH2 register. WDT_MATCH.MATCH refers to the value held by bits [21:0] of the WDT_MATCH register. SRSS_INTR.WDT_MATCH refers to the WDT_MATCH bit of the SRSS_INTR register.

When enabled, the WDT counts up on each rising edge of the configured clock. When the counter value (stored in the WDT_CNT register) equals the match value stored in the 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 22-bit boundary (0x3FFFFF), at which point it wraps around to 0 and continues counting. 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 (the WDT_MATCH bit of SRSS_INTR). If the firmware fails to clear the interrupt for two consecutive occurrences, the third match generates a device reset.

The WDT also provides a configurable option to set the number of bits 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 least significant bits (LSBs) considered for the match value from the 22-bit count value during the match process. For instance, if these bits are set to 4, bits [5:31] are ignored, and bits [0:4] are considered for the comparison, making the WDT counter behave like a 5-bit counter.

***Note**: Configuring these bits does not reduce the actual counter size . The WDT_CNT register still counts from 0 to 0x3FFFFF (22-bit range).*

The WDT can be enabled or disabled through the WDT_EN bit [0] of the WDT_CTL register. Additionally, the WDT_CTL register provides a mechanism to lock the WDT configuration registers. The WDT_LOCK bits [31:30] in the WDT_CTL register control the lock status of the WDT registers. These lock bits can enable the lock in a single write operation, while unlocking requires two distinct write operations. The WDT_LOCK bits protect multiple registers and fields, including the WDT_EN bit, the WDT_MATCH register, the CLK_ILO_CONFIG register, and the LFCLK_SEL bits [1:0] of the CLK_SELECT register.

It is important to note that the WDT_LOCK bits are not retained in Deep Sleep and Hibernate modes. After a wake-up, these bits reset to their default (LOCK) state. To modify any register protected by the WDT_LOCK bits after a wake-up from Deep Sleep or Hibernate, the WDT must first be unlocked by following the unlock sequence before the desired update is made.

Table 64 explains the various registers and bit-fields used to configure and operate the WDT.

Table 64 Free-running WDT configuration options

Register [Bit_Pos]	Bit_name	Description
WDT_CTL[0]	WDT_EN	Enable or disable the watchdog 0: WDT reset disabled 1: WDT reset enabled
WDT_CTL[5:4]	WDT_CLK_SEL	Select source for WDT 0: ILO 1: PILO 2: BAK

(table continues...)

Table 64 (continued) Free-running WDT configuration options

Register [Bit_Pos]	Bit_name	Description
WDT_CTL[31:30]	WDT_LOCK	Locks or unlocks write access to the watchdog configuration and clock-related registers. When these bits are set, the lock is enabled, restricting modifications to the protected registers. 0: No effect 1: Clear bit 0 2: Clear bit 1 3: Set both bit 0 and 1 (lock enabled) The WDT automatically locks upon a reset. Additionally, this lock is not retained during Deep Sleep or Hibernate modes. As a result, the WDT configuration will be locked again after a wake-up from these modes.
WDT_CNT[31:0]	COUNTER	Current value of the WDT counter
WDT_MATCH[31:0]	MATCH	Match value to generate a watchdog match event
WDT_MATCH2[4:0]	IGNORE_BITS_ABOVE	MSB bit index for matching. Specifies the highest bit index to be used as the most significant bit (MSB) for the match comparison. Bit indices above this setting are not considered during the match process. • This setting determines the time-to-reset for the watchdog, which is triggered after three successive matches • The four least significant bits (LSBs) cannot be ignored for matching. Any setting below 3 behaves as if it is set to 3 • If the configured setting exceeds the total number of bits in the WDT counter, all actual bits in the counter are considered for the match

(table continues...)

Table 64 (continued) Free-running WDT configuration options

Register [Bit_Pos]	Bit_name	Description
SRSS_INTR[0]	WDT_MATCH	WDT interrupt request: This bit is set whenever a watchdog match event occurs. To clear the WDT interrupt, write 1 to this bit.
SRSS_INTR_MASK[0]	WDT_MATCH	Mask for the WDT interrupt 0: WDT interrupt is blocked 1: WDT interrupt is forwarded to the CPU

18.3.2 Watchdog reset

A watchdog timer (WDT) is typically used to protect the device from firmware or system crashes and faults. When the WDT is used to safeguard against system crashes, the WDT interrupt bit should be cleared by a portion of the code that is not directly associated with the WDT interrupt. Otherwise, even if the main firmware function crashes or enters an endless loop, the WDT interrupt vector might remain functional and periodically service (feed) the WDT. This could leave the system in a non-operational state without initiating a reset.

When a debug probe is connected, the device reset is blocked, and instead, an interrupt is generated.

The safest way to use the WDT to protect against system crashes is as follows:

1. Configure the watchdog reset period so that the firmware can reset the watchdog at least once during the watchdog reset period, even under the longest firmware delay path
2. Feed (reset) the watchdog by clearing the interrupt bit regularly in the main body of the firmware code. This is done by writing 1 to the `WDT_MATCH` bit in the `SRSS_INTR` register. Note that this operation does not reset the watchdog counter but merely feeds the watchdog to prevent a reset during the next two match events
3. Avoid resetting (clearing the interrupt for) the watchdog in the WDT interrupt service routine (ISR) if the WDT is being used as a reset source to protect against system crashes. Using the WDT reset feature and its ISR simultaneously is not recommended for this purpose

The recommended steps to use the WDT as a reset source are as follows:

1. Ensure the WDT configuration is unlocked by clearing the `WDT_LOCK` bits [31:30] of the `WDT_CTL` register. Clearing these bits requires two writes to the register, with each write clearing one bit, as explained in Table 64
2. Write the desired `IGNORE_ABOVE_BITS` value in the `WDT_MATCH2` register to set the counter resolution for the match
3. Write a match value to the `WDT_MATCH` register. The match value does not control the watchdog reset period since the counter is not reset on a match event. This value controls only the first interrupt interval, while the successive interrupt intervals are defined by the `IGNORE_BITS_ABOVE` field. The approximate watchdog period (in seconds) is calculated as follows:

$$\text{Watchdog reset period} = \text{ILO}_{\text{period}} \times (2 \times 2^{(\text{IGNORE_BITS_ABOVE})} + \text{WDT_MATCH})$$

Figure 54 Watchdog period equation

4. Clear any pending WDT interrupt by setting the `WDT_MATCH` bit in the `SRSS_INTR` register
5. Enable the PILO by setting the `ENABLE` bit [31] of the `CLK_PILO_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 [31:30]. This action also locks the LFCLK_SEL bits of the CLK_SELECT register
8. In the firmware, feed (clear interrupt) the watchdog by writing 1 to the WDT_MATCH bit in the SRSS_INTR register

18.3.3 Watchdog interrupt

In addition to generating a device reset, the WDT can also generate interrupts. However, note that interrupt servicing and watchdog reset cannot be used simultaneously with the free-running WDT.

The watchdog counter can send interrupt requests to the CPU in CPU Active power modes and to the wakeup interrupt controller (WIC) in CPU Sleep and Deep Sleep power modes. Additionally, the watchdog can wake up the device from Hibernate power mode. The functionality in different power modes is as follows:

- **CPU Active mode**: In Active power mode, the WDT sends an interrupt to the CPU. The CPU acknowledges the interrupt request and executes the interrupt service routine (ISR). The interrupt must be cleared in the ISR
- **CPU Sleep or Deep Sleep mode**: In this mode, the CPU is powered down. The WDT sends its interrupt request to the WIC, which then wakes up the CPU. The CPU acknowledges the interrupt request and executes the ISR. The interrupt must be cleared in the ISR
- **Hibernate mode**: In Hibernate mode, the entire device, except a few peripherals (such as WDT and LPCOMP), is powered down. Any interrupt generated in this mode results in a device reset. Therefore, there is no interrupt service routine or mechanism in this mode

For more details on the power modes of the device, see Device power modes.

Because of its free-running nature, the WDT is not suitable for periodic interrupt generation. The multi-counter watchdog timer (MCWDT) should be used instead, as its counters are specifically designed to generate periodic interrupts. However, if absolutely necessary, the WDT can be used as a periodic interrupt generator by following these steps:

1. Unlock the WDT if this is the first update to the WDT registers after a Deep Sleep or Hibernate wakeup, or a device reset
2. Write the desired IGNORE_ABOVE_BITS value in the WDT_MATCH2 register to set the counter resolution for the match event
3. Write the desired match value to the WDT_MATCH register
4. Clear any pending WDT interrupt by setting the WDT_MATCH bit in the SRSS_INTR register
5. Enable the WDT interrupt to the CPU by setting the WDT_MATCH bit in the SRSS_INTR_MASK register
6. Enable the SRSS interrupt to the CPU by configuring the appropriate ISER register (for details, see Interrupts)
7. In the ISR, perform the following steps:
    a. Unlock the WDT
    b. Clear the WDT interrupt
    c. Add the desired match value to the existing match value. This ensures another interrupt is generated when the counter reaches the updated match value (period)

18.4 Multi-counter WDT

18.4.1 Overview

The multi-counter watchdog timer (MCWDT) component provides an interface to configure one MCWDT hardware block. Each block contains three counters that can be customized for various system utility functions, including free-running counters, periodic interrupts, or watchdog resets. In addition to the MCWDTs, each

device features a standalone watchdog timer (WDT), which can be used independently to generate a watchdog reset or periodic interrupts.

Functional overview of the multi-counter WDT block in the CYW20829 MCU

The CYW20829 MCU includes a single MCWDT block that consists of the following counters:

- **Two 16-bit counters**
  - Named MCWDT0_WDT0 and MCWDT0_WDT1
  - These are 16-bit up counters that can be configured as either free-running counters or counters with a specific 16-bit period
  - These counters can be used to generate an interrupt or reset the device
- **One 32-bit counter**
  - Named MCWDTx_WDT2
  - This is a 32-bit free-running counter and can be configured to generate an interrupt

The counters can be configured to operate independently or in a cascade configuration, allowing for a maximum counter size of 64 bits (using all three counters).

All counters in the MCWDT block are synchronized by the CLK_LF, ensuring consistent operation across all configurations.

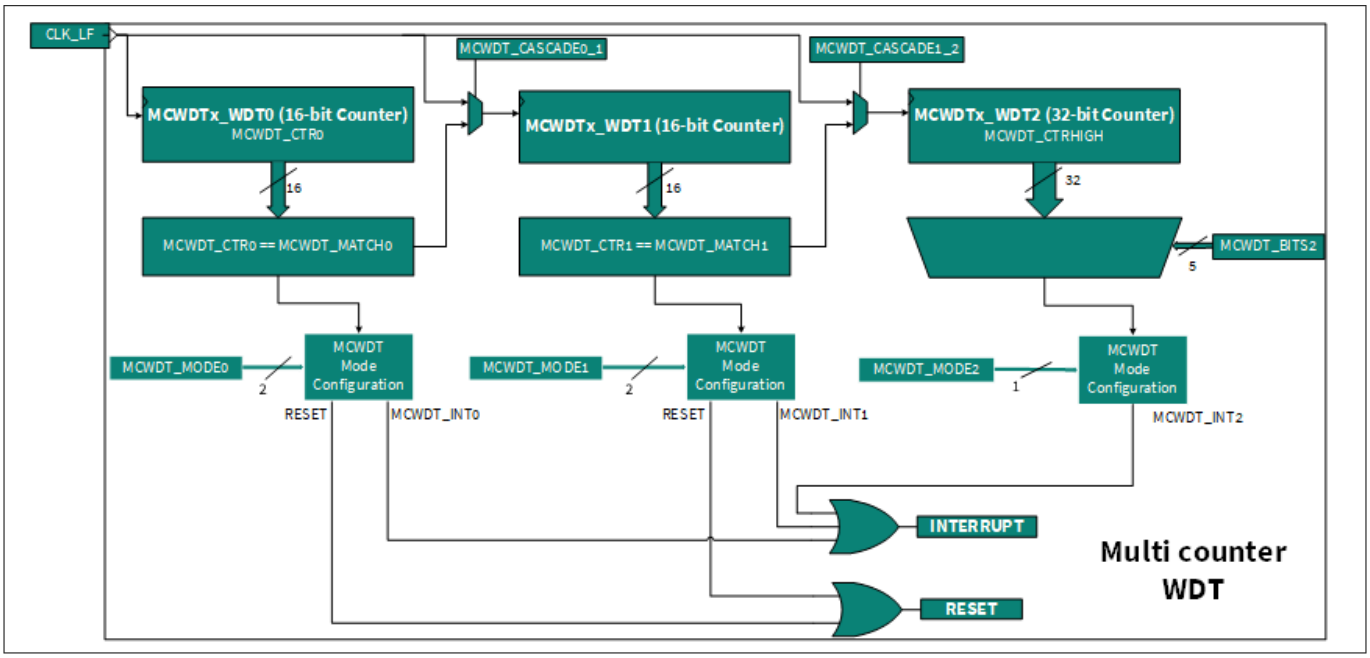


Figure 55 Multi-counter WDT functional diagram

18.4.1.1 MCWDT0_WDT0 and MCWDT0_WDT1 counter operations

MCWDT0_WDT0 and MCWDT0_WDT1 are 16-bit up counters that can be configured as either free-running counters or counters with any desired 16-bit period. These counters can generate an interrupt or trigger a device reset on a match event.

- The WDT_CTR0 bits [15:0] and WDT_CTR1 bits [31:16] of the MCWDT0_CNTLOW register hold the current counter values for 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 values for MCWDT0_WDT0 and MCWDT0_WDT1, respectively
- The WDT_MODEx bits of the MCWDT0_CONFIG register determine the action the watchdog counter performs on a match event (WDT_MATCHx == WDT_CTRx)

The MCWDT0_WDT0 and MCWDT0_WDT1 counters can be configured to execute one of the following actions on a match event:

1. Assert an interrupt (`WDT_INTx`)
2. Trigger a device reset
3. Assert an interrupt on the first two matches and trigger a device reset on the third match (after two unhandled interrupts)

In addition to generating resets and interrupts, the match event can also clear the corresponding counter. This behavior is enabled by setting the `WDT_CLEARx` bit in the `MCWDT0_CONFIG` register.

The operation of the MCWDT0_WDT0 and MCWDT0_WDT1 counters is shown in Figure 56.

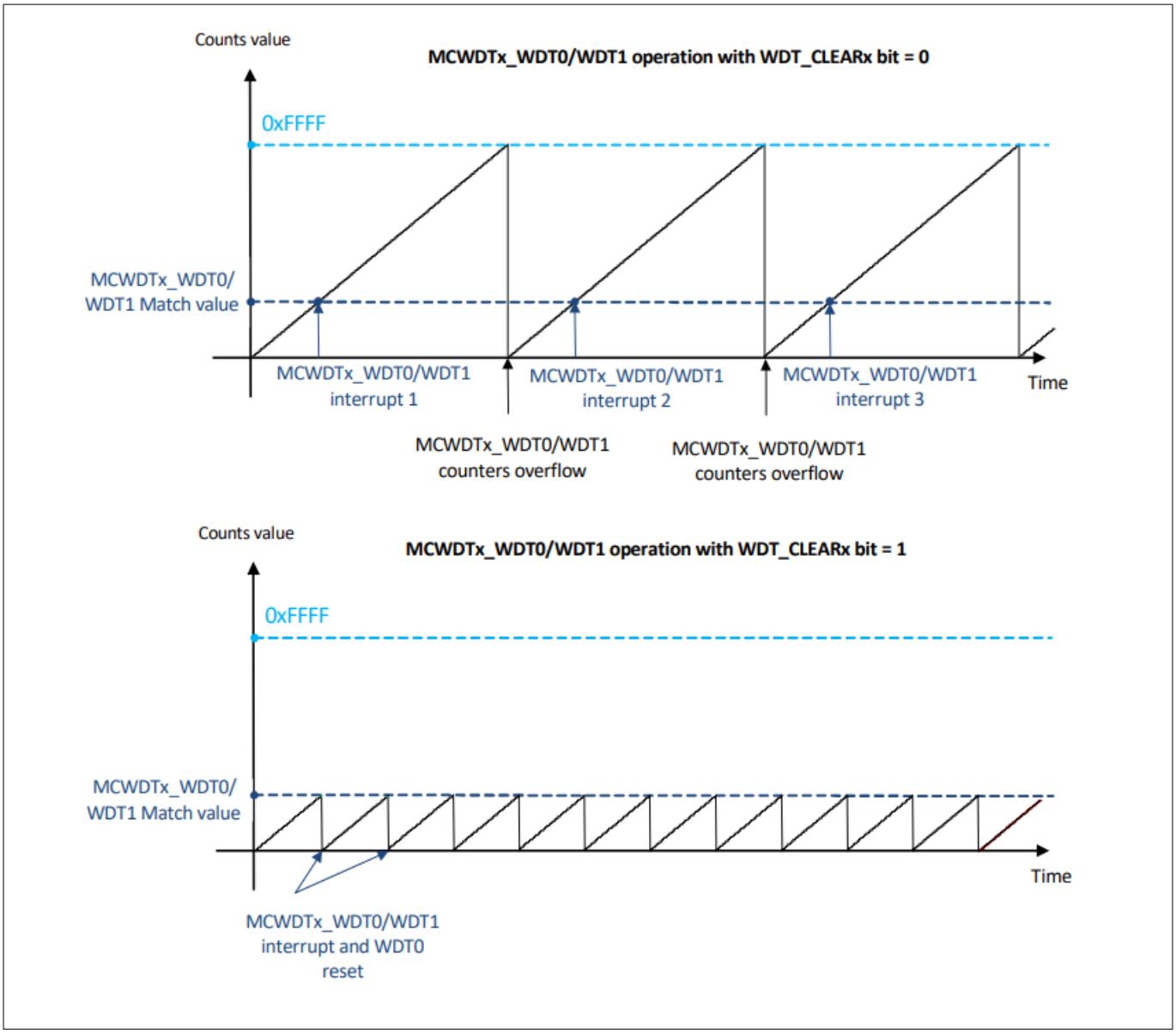


Figure 56 MCWDT0_WDT0/WDT1 operation

Table 65 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	WDT action on a match event (`WDT_CTRx == WDT_MATCHx`) 0: Do nothing 1: Assert interrupt (MCWDT_INTx) 2: Assert device reset 3: Assert interrupt on match and a device reset on the third unhandled interrupt
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
MCWDT0_CNTLOW[15:0] MCWDT0_CNTLOW[16:31]	WDT_CTR0 WDT_CTR1	Current counter values. Bits[15:0] contain the current value of the MCWDT0_WDT0 counter and bits[31:16] contain the current value of the MCWDT0_WDT1 counter.
MCWDT0_MATCH[15:0] MCWDT0_MATCH[16:31]	WDT_MATCH0 WDT_MATCH1	Match values Changing WDT_MATCHx requires 1.5 LFCLK cycles to come into effect. After changing WDT_MATCHx, do not enter the Deep Sleep mode for at least one LFCLK cycle to ensure the WDT updates to the new setting.

18.4.1.2 MCWDT0_WDT2 counter operation

`MCWDT0_WDT2` is a 32-bit, free-running counter that you can configure to generate an interrupt. The `MCWDT0_CNTHIGH` register holds the current WDT2 counter value. WDT2 does not support a match feature. However, you can configure it to generate an interrupt when one of the counter bits toggles. The `WDT_BITS2` field [28:24] in the `MCWDT0_CONFIG` register selects the counter bit that triggers the WDT2 interrupt. The `WDT_MODE2` bit [16] in the `MCWDT0_CONFIG` register selects whether to assert an interrupt on a bit toggle. Figure 2-5 shows the `MCWDTx_WDT2` counter operation.

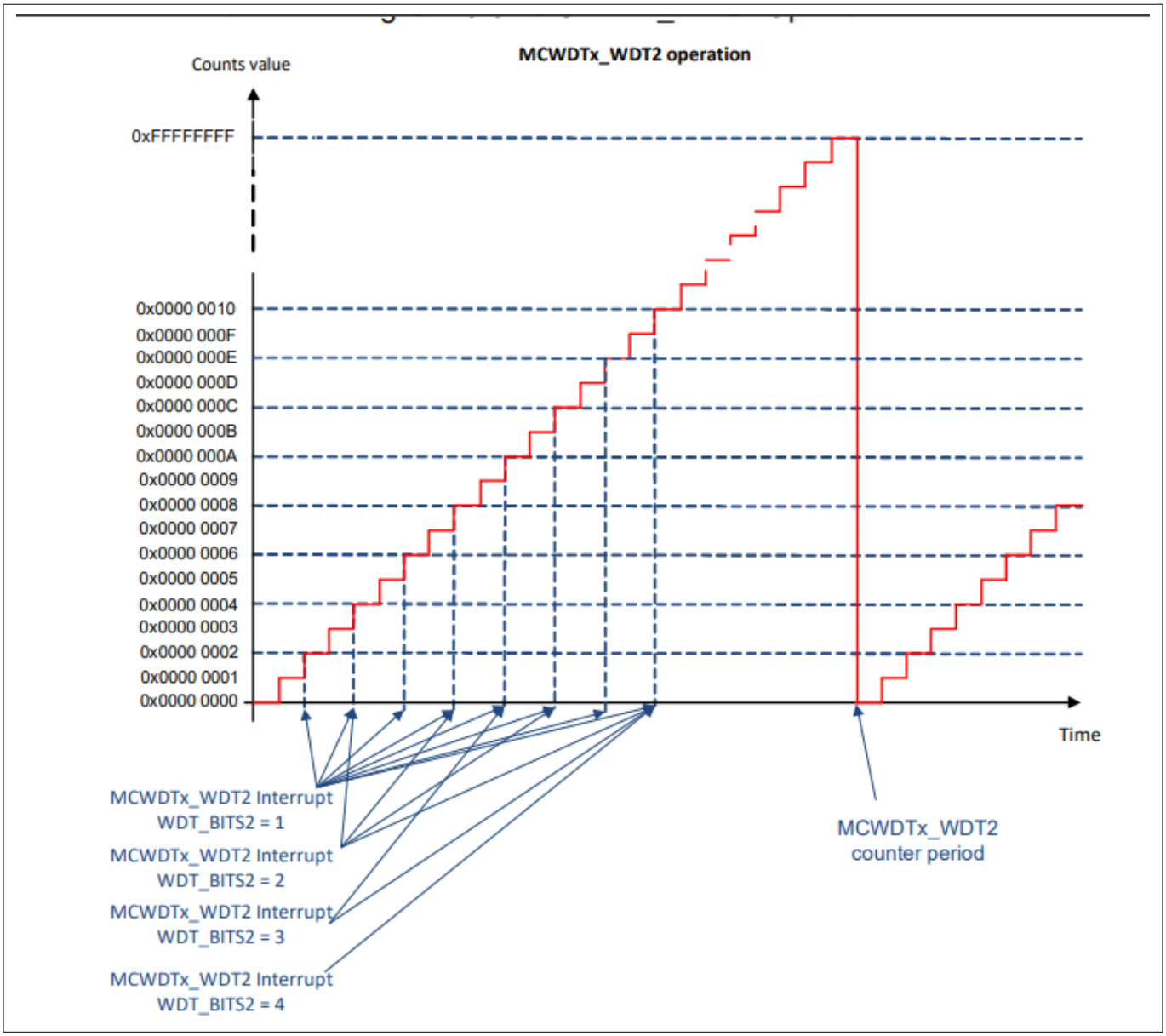


Figure 57 MCWDT0_WDT2 operation

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.

Register [Bit_Pos]	Bit_name	Description
MCWDT0_CONFIG[28:24]	WDT_BITS2	Bit to monitor for MCWDTx_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 231 ticks)
MCWDT0_CNTHIGH[31:0]	WDT_CTR2	Current counter value of MCWDT0_WDT2

18.4.2 Enabling and disabling WDT

Enable an MCWDT counter by setting WDT_ENABLEx in MCWDT0_CTL, and disable it by clearing the bit. Enabling or disabling takes 1.5 LFCLK cycles to take effect. Do not change WDT_ENABLEx more than once during that interval. Use WDT_ENABLEDx in MCWDT0_CTL to monitor the counter's enabled or disabled state.

WDT_RESETx in MCWDT0_CTL clears the corresponding MCWDT0 counter when set in firmware. Hardware clears the bit after the counter resets. This behavior is useful when MCWDT0_WDT0 or MCWDT0_WDT1 is configured to generate a device reset on a match event. In that case, the device resets when the counter reaches the match value. Setting WDT_RESET0 or WDT_RESET1 resets MCWDT0_WDT0 or MCWDT0_WDT1, respectively, and prevents a device reset.

After the MCWDT is enabled, do not write to the MCWDT configuration (MCWDT0_CONFIG) or control (MCWDT0_CTL) registers. Prevent accidental modification by setting MCWDT_LOCK [31:30] in MCWDT0_LOCK. If the application must update the match value (WDT_MATCH) while the MCWDT is running, clear MCWDT_LOCK first. Clearing both bits requires two writes. Write 1 to clear bit 0. Write 2 to clear bit 1. Write 3 to set both bits. Writing 0 has no effect. MCWDT_LOCK protects only MCWDT0_CTL, MCWDT0_CONFIG, and MCWDT0_MATCH. LFCLK select registers are protected by the free-running WDT lock bits.

Table 66 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 x 0: Counter is disabled 1: Counter is enabled
MCWDT0_CTL[1] MCWDT0_CTL[9] MCWDT0_CTL[17]	WDT_ENABLED0 WDT_ENABLED1 WDT_ENABLED2	Indicates the actual enabled/ disabled state of counter x. This bit should be monitored after changing the WDT_ENABLEx bit, to confirm an acknowledgment of the change
MCWDT0_CTL[3] MCWDT0_CTL[11] MCWDT0_CTL[19]	WDT_RESET0 WDT_RESET1 WDT_RESET2	Reset MCWDT counter x to 0. hardware clears the bit when the reset is complete 0: Software - No Action 1: Software - Resets the counter

(table continues...)

Table 66 (continued) Watchdog configuration options

Register [Bit_Pos]	Bit_name	Description
MCWDT0_CTL[31:30]	WDT_LOCK	Locks or unlocks write access to the MCWDT0_CTL (except the WDT_LOCK bits). 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)

18.4.3 Watchdog cascade options

The cascade configuration provides an option to increase the MCWDT counter resolution. The WDT_CASCADE0_1 bit [3] of the MCWDT0_CONFIG register cascades MCWDT0_WDT0 and MCWDT0_WDT1, while the WDT_CASCADE1_2 bit [11] of the MCWDT0_CONFIG register cascades MCWDT0_WDT1 and MCWDT0_WDT2. It is important to note that cascading two 16-bit counters does not create a 32-bit counter. Instead, it results in a 16-bit period counter with a 16-bit prescaler.

For example:

- When cascading MCWDT0_WDT0 and MCWDT0_WDT1, MCWDT0_WDT0 functions as a prescaler for MCWDT0_WDT1
- The prescaler value is defined by the WDT_MATCH0 bits [15:0] in the MCWDT0_MATCH register
- The MCWDT0_WDT1 counter has a period defined by WDT_MATCH1 bits [31:16] in the MCWDT0_MATCH register

The same logic applies to the cascading of MCWDT0_WDT1 and MCWDT0_WDT2.

Table 67 MCWDT cascade configuration

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.

When using cascade mode (WDT_CASCADE0_1 or WDT_CASCADE1_2 set), resetting the counters while the prescaler or lower counter is at its match value and the counter is configured to clear on match causes the upper counter to

increment to 1 instead of remaining at 0. This behavior can be corrected by issuing a second reset to the upper counter approximately 100 µs after the first reset.

The second reset is required only when the first reset is performed while the prescaler counter value is at its match value. Figure 6 illustrates this behavior when MCWDTx_WDT0 and MCWDTx_WDT1 are cascaded, along with the timing for the second reset.

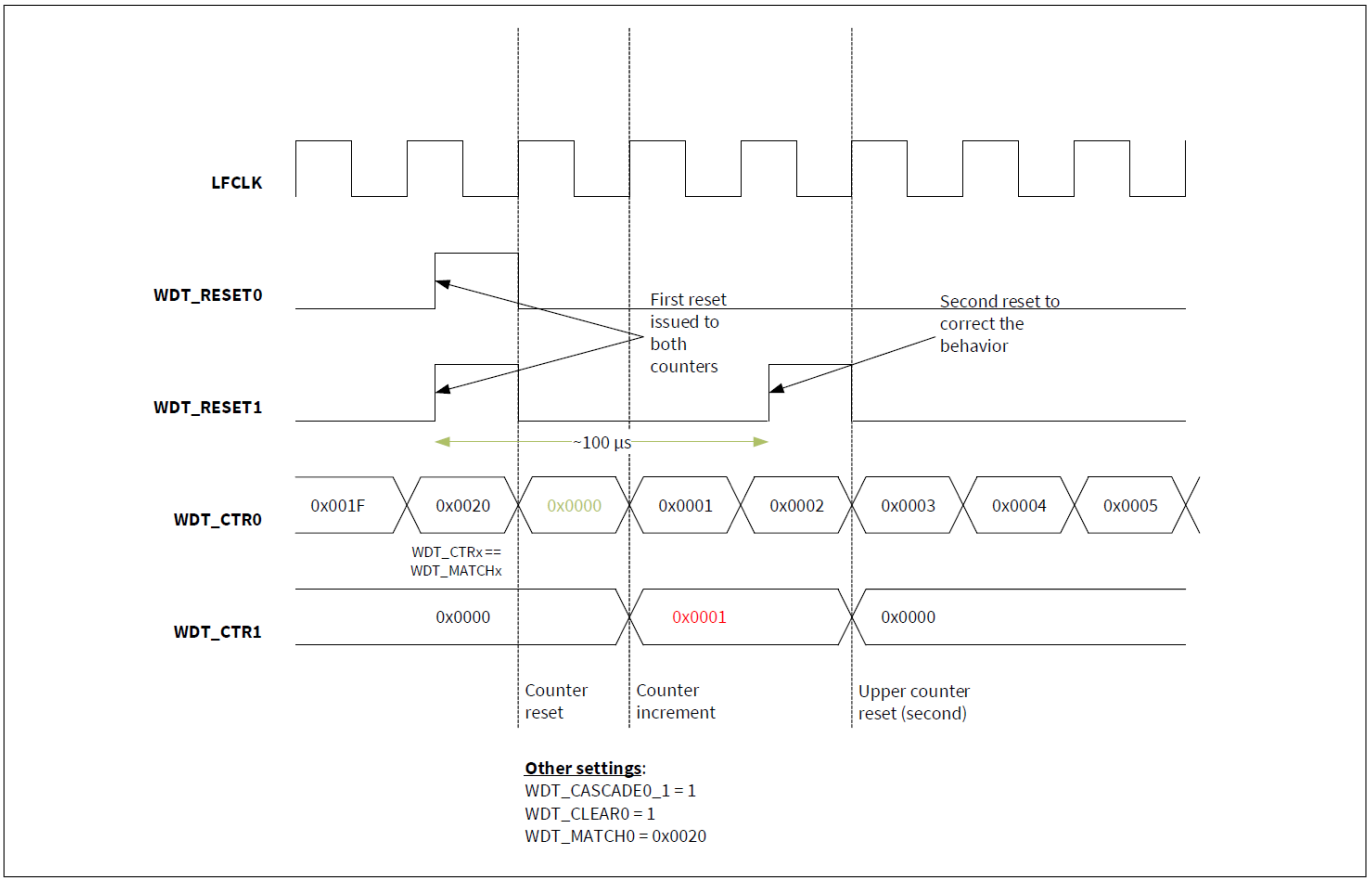


Figure 58 MCWDT reset behavior in cascaded mode

The counters exhibit non-monotonicity under the following cascaded conditions:

- If WDT_CASCADE0_1 is set, WDT_CTR1 does not increment during the cycle immediately after WDT_CTR0 = WDT_MATCH0
- If WDT_CASCADE1_2 is set, WDT_CTR2 does not increment during the cycle immediately after WDT_CTR1 = WDT_MATCH1
- If both WDT_CASCADE0_1 and WDT_CASCADE1_2 are set:
  - WDT_CTR2 does not increment during the cycle immediately after WDT_CTR1 = WDT_MATCH1
  - WDT_CTR1 does not increment during the cycle immediately 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 ensures that the upper counter has incremented after a match event in the periodic counter.

18.4.4 MCWDT reset

MCWDT0_WDT0 and MCWDT0_WDT1 can be configured to generate a device reset similar to the free-running watchdog timer (WDT) reset. When a debug probe is connected, the device reset is blocked, but an interrupt is generated if configured. Follow these steps to configure the MCWDT0_WDT0 or MCWDT0_WDT1 counter of the MCWDT0 block to generate a system reset:

1. Configure the MCWDT to generate a reset by setting the WDT_MODEx bits in the MCWDT0_CONFIG register
2. Set the WDT_MODE0 or WDT_MODE1 bits in the MCWDT0_CONFIG register to one of the following values:
   - 2 (reset on match)
   - 3 (interrupt on match and reset on the third match after two unhandled interrupts)
3. Optionally, set the WDT_CLEAR0 or WDT_CLEAR1 bit in the MCWDT0_CONFIG register for MCWDT0_WDT0 or MCWDT0_WDT1 to reset the corresponding watchdog counter to 0 on a match event. If this option is not configured, the counters will operate in free-running mode
4. Calculate the watchdog reset period to ensure the firmware feeds (resets) the watchdog at least once during the period, even in the longest firmware delay path:
   - For WDT_MODEx == 2, the match value should be equal to the watchdog period
   - For WDT_MODEx == 3, the match value should be one-third of the watchdog period
5. Write the calculated match value to the WDT_MATCH register for MCWDT0_WDT0 or MCWDT0_WDT1
6. Optionally, enable cascading to increase the interval. Note that the legal value for the WDT_MATCH0 or WDT_MATCH1 field is 1 to 65535
7. For WDT_MODEx == 2, set the WDT_RESETx bit in the MCWDT0_CTL register to reset the WDTx counter to 0
8. For WDT_MODEx == 3, set the WDT_INTx bit in the MCWDT0_INTR register to clear any pending interrupts
9. Enable the WDTx by setting the WDT_ENABLEx bit in the MCWDT0_CTL register. Wait until the WDT_ENABLEDx bit is set
10. Lock the MCWDTx configuration by setting the MCWDT_LOCK bits in the MCWDT0_LOCK register
11. In firmware, feed (reset) the watchdog as explained in 4. for calculating the reset period

18.4.5 MCWDT interrupt

When configured to generate an interrupt, the WDT_INTx bits of the MCWDT0_INTR register indicate the status of any pending watchdog interrupts. The firmware must clear the interrupt by setting the WDT_INTx bit. The WDT_INTx bits of the MCWDT0_INTR_MASK register mask the corresponding WDTx interrupt of the MCWDT0 block to the CPU.

To use the WDT as a periodic interrupt generator, perform the following:

1. Write the desired match value to the WDT_MATCH register for MCWDTx_WDT0 or MCWDTx_WDT1, or write the WDT_BITS2 value to the MCWDT0_CONFIG register for MCWDTx_WDT2. Note that the legal value for the WDT_MATCH field is 1 to 65535
2. Configure the WDTx to generate an interrupt by setting the WDT_MODEx bits in the MCWDT0_CONFIG register
   - For MCWDT0_WDT0 or MCWDT0_WDT1, set the WDT_MODE0 or WDT_MODE1 bits to:
     - 1 (interrupt on match)
     - 3 (interrupt on match and reset on the third unhandled interrupt)
   - For MCWDTx_WDT2, set the WDT_MODE2 bit in the MCWDT0_CONFIG register
3. Clear any pending interrupt by setting the WDT_INTx bit in the MCWDT0_INTR register
4. To reset the corresponding watchdog counter to 0 on a match event, set the WDT_CLEAR0 or WDT_CLEAR1 bit in the MCWDT0_CONFIG register for MCWDT0_WDT0 or MCWDT0_WDT1
5. Mask the WDTx interrupt to the CPU by setting the WDT_INTx bit in the MCWDT0_INTR_MASK register
6. Enable the WDTx by setting the WDT_ENABLEx bit in the MCWDT0_CTL register. Wait until the WDT_ENABLEDx bit is set
7. Enable the MCWDT0 interrupt to the CPU by configuring the appropriate ISER register. Refer to the Interrupts chapter in the TRM
8. In the interrupt service routine (ISR), clear the WDTx interrupt by setting the WDT_INTx bit in the MCWDT0_INTR register

Interrupts from all three WDTx counters of the MCWDT block are mapped as a single interrupt to the CPU. Within the ISR, the WDT_INTx bits in the MCWDT0_INTR register can be read to identify the source of the interrupt. Each MCWDT block has its own interrupt to the CPU. For additional interrupt details, see Interrupts.

The MCWDT block can send interrupt requests to the CPU in Active power mode and to the WIC in Sleep and Deep Sleep power modes. It behaves similarly to the free-running watchdog timer.

18.5 Reset cause detection

The RESET_WDT bit [0] in the RES_CAUSE register indicates a reset generated by the free-running watchdog timer (WDT). The RESET_MCWDT0 bit in the RES_CAUSE register indicates a reset generated by the MCWDT0 block. These bits remain set until they are cleared or until a power-on reset (POR), brownout reset (BOD), or external reset (XRES) occurs. All other reset types leave these bits unchanged. For more details, see Reset.

18.6 Registers summary

Table 68 Registers summary

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
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

19 Real-time clock (RTC)

A real-time clock (RTC) is a hardware component used to keep track of the current time and date, with the capability to trigger a callback at a specific time. RTCs are widely used in electronic devices such as computers, smartphones, digital cameras, microwave ovens, washing machines, clocks, and other home appliances, as well as IoT devices and activity trackers, where an accurate and reliable time reference is required.

The RTC operates independently of the main system clock and continues to track time even when the system is powered off or in Sleep mode. Typically, the RTC is equipped with a battery backup to maintain its timekeeping function. This allows it to retain time and date information even when the main power source is disconnected.

An RTC circuit generally includes the following components:

- A crystal oscillator for accurate timekeeping
- A counter to track the passage of time
- Registers for storing time and date information

This functionality ensures that RTCs provide consistent and reliable timekeeping for a wide range of applications.

19.1 Features

The RTC component offers the following features:

- Binary-coded decimal (BCD) time and date format
- Configurable alarm functionality
- Century interrupt feature for century boundary detection
- Configurable daylight savings time (DST) adjustment
- Support for both 12-hour and 24-hour time formats
- Automatic leap-year compensation
- Option to drive the RTC with an external 50 Hz or 60 Hz clock source, with the internal watch crystal oscillator as the default source

19.2 Architecture

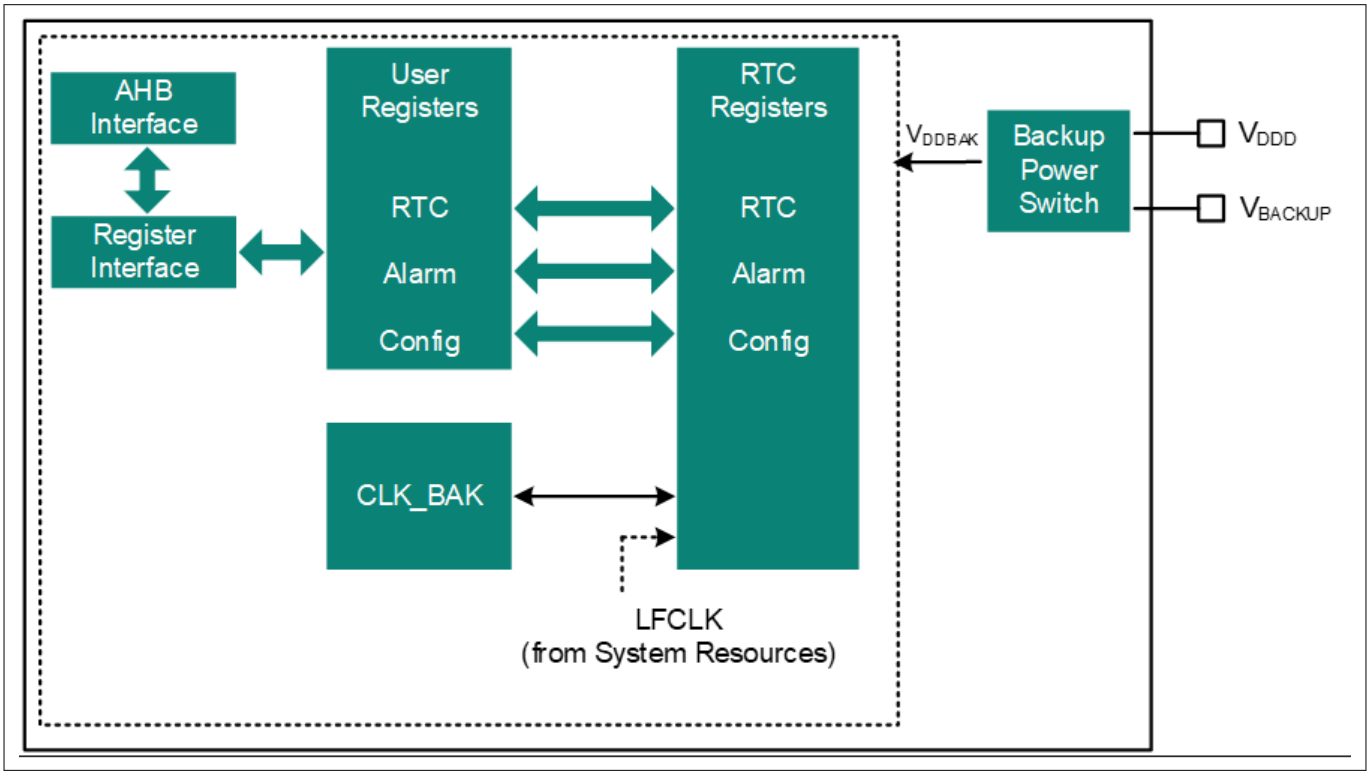


Figure 59 RTC block diagram

The RTC consists of two distinct blocks. The AHB RTC user registers contain all RTC settings and synchronize with the actual RTC registers. However, the AHB RTC user registers do not update automatically. Instead, they are updated on-demand when a read transaction is initiated from the actual RTC registers. To avoid stalling the actual RTC, only the AHB RTC user registers should be accessed.

The components of the RTC include:

- **Crystal oscillator**: Generates a clock signal used by the counter to track time and date
- **Counter**: Increments the time and date register values accordingly
- **Registers**: Store the time and date information
- **Battery backup**: Maintains the timekeeping function even when the main power source is disconnected
- **Control logic**: Manages the operation of the RTC circuit and interfaces with the rest of the system
- **Interface**: Provides a mechanism to set the time and date, read the time and date, and configure other RTC functions

19.3 Description

The RTC consists of eight fields: `RTC_SEC`, `RTC_MIN`, `RTC_HOUR`, `CTRL_12HR`, `RTC_DAY`, `RTC_DATE`, `RTC_MON`, and `RTC_YEAR`. All fields are encoded in binary-coded decimal (BCD) format, where four bits represent a single decimal digit. The RTC supports both 12-hour and 24-hour time formats, with the "hours" field capable of displaying AM/PM flags. The RTC hardware automatically compensates for leap years, eliminating the need for manual adjustments. Leap years are defined as years divisible by 4 and not by 100, or years divisible by 4 and by both 100 and 400.

To perform read or write operations, you must set the `READ` bit and `WRITE` bit in the `BACKUP_RTC_RW` register. Refer to the register details for more information.

The RTC includes an alarm feature that can generate an interrupt, enabling the system to wake up from Sleep, Deep Sleep, and Hibernate power modes. The alarm consists of six fields that correlate with the RTC fields: Month/Date, Day-of-Week, and Hour:Minute:Second. This feature allows the RTC to generate an interrupt when the RTC time matches the time set in the ALARM registers. The RTC provides two physical alarm functions, which can trigger a system reset and startup subsystem (SRSS) backup interrupt, prompting appropriate actions at the scheduled time.

The RTC also supports two modes for daylight savings time (DST): fixed date and relative date. When DST is enabled, the alarm2 block is reserved for executing DST start or stop actions. DST can be optionally enabled and supports any start and end date. These dates can either be a fixed date, such as March 24th, or a relative date.

Table 69 Bitfield details

Bitfield 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 12/24HR 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 12/24HR mode: 1=12HR, 0=24HR
RTC_DAY	3	Calendar Day of the week, 1-7 It is up to the user to define the meaning of the values, but 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 the 12-hour format, which includes an AM/PM flag, and the 24-hour format for the "hours" field. Additionally, the RTC includes a “day of the week” field that counts from 1 to 7. The programmer must define which weekday is represented by the value ‘1’.

The RTC implements automatic leap-year correction for the "date" field (day of the month). If the year is divisible by 4, February (Month = 2) will have 29 days instead of 28. However, when the year reaches 2100, the "year" field rolls over from 99 to 00, and the leap-year correction will be incorrect (2100 is flagged as a leap year, which it is not). To handle this, the RTC raises a century interrupt, allowing firmware to take appropriate actions.

User registers containing these bitfields are BACKUP_RTC_TIME and BACKUP_RTC_DATE. For detailed descriptions of these registers, refer to the corresponding sections in the register technical reference manual (TRM).

Since the user registers are in the high-frequency bus clock domain, while the actual RTC registers operate on the low-frequency 32,768-Hz clock, special care is required when reading from or writing to the RTC registers. The processes for performing these operations are explained in the following section.

19.3.1 Reading RTC user registers

To initiate a read transaction, the firmware must set the READ bit in the BACKUP_RTC_RW register. When this bit is set, the RTC registers are copied to the user registers and frozen, ensuring that a coherent RTC value can be safely read by the firmware. The read transaction is completed by clearing the READ bit.

The READ bit cannot be set under the following conditions:

- The RTC is still busy with a previous operation (indicated by the RTC_BUSY bit in the BACKUP_STATUS register being set)
- The WRITE bit in the BACKUP_RTC_RW register is set

Before setting the READ bit, the firmware must verify that the above conditions are not active.

19.3.2 Writing to RTC user registers

When the WRITE bit in the BACKUP_RTC_RW register is set, data can be written to the RTC user registers. If the WRITE bit is not set, any writes to the RTC user registers are ignored. After completing all RTC write operations, the firmware must clear the WRITE bit for the RTC update to take effect. Once the WRITE bit is cleared, the hardware updates all RTC registers simultaneously on a single WCO clock edge to ensure data coherency.

The WRITE bit cannot be set under the following conditions:

- The RTC is busy with a previous operation (indicated by the RTC_BUSY bit in the BACKUP_STATUS register being set)
- The READ bit in the BACKUP_RTC_RW register is set

Before clearing the WRITE bit, the firmware must ensure that the values written to the RTC fields form a coherent and valid set. The hardware does not validate the written values. Writing invalid values may result in undefined RTC behavior.

If a brownout, reset, or entry into Deep Sleep or Hibernate mode occurs while an RTC update is in progress (with the WRITE bit set), the write operation will not be completed. This is because the WRITE bit is cleared during a reset, and the RTC update is triggered only when the WRITE bit is cleared after a write transaction. If the write operation is still in progress (indicated by RTC_BUSY), data corruption can occur if the system is reset or enters Deep Sleep or Hibernate mode.

19.4 Alarm fields

The Alarm feature enables the RTC to generate an interrupt, which can be used to wake up the system from Sleep, Deep Sleep, and Hibernate power modes.

The Alarm functionality consists of six fields that correspond to the RTC fields: Month/Date, Day-of-Week, and Hour:Minute:Second. Each Alarm field includes an enable bit that must be set to activate matching for that specific field. If the enable bit for a field is cleared, the corresponding field will be ignored during the matching process.

Table 70 Alarm bitfields

Bitfield 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

(table continues...)

Table 70 (continued) Alarm bitfields

Bitfield name	Bit_Pos	Register_name	Description
ALM_HOUR	20:16	BACKUP_ALMx_TIME	Alarm hours, value depending on 12/24HR mode 24HR: [4:0]=0-23 12HR: [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
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. Alarm triggers whenever 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) bit is set but all alarm fields for date and time are disabled, an alarm interrupt will be generated once every second. It is important to note that there is no alarm field for the Year.

Similar to the RTC user registers, the alarm user registers require special procedures for read and write operations. These procedures are explained in the sections Reading RTC user registers and Writing to RTC user registers in the technical reference manual (TRM).

Interrupts must be properly configured for the RTC to generate interrupts or wake-up events. Additionally, 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. For more information, see Device power modes and Interrupts chapters in the TRM.

19.5 Registers summary

Table 71 Registers summary

Bit 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	BACKUP_INTR_MASKED
BACKUP_INTR_MASKED	BACKUP_INTR_MASKED

20 Reset

CYW20829 supports several types of resets to ensure error-free operation during power-up and to allow the device to reset based on user-supplied external hardware or internal software reset signals. Resets are broadly aligned with power domains and global power modes. The resets described in this section cause a reboot that ends in Active mode. Some device blocks may also have local resets, which are described in their respective chapters.

Reset assertion is propagated asynchronously, while reset de-assertion is synchronized to the clock domain where it is applied. CYW20829 includes hardware to record which reset event has occurred, enabling easier identification and debugging of reset conditions.

The fault manager converts a fault into a high-priority interrupt, such as an NMI, to give the processor an opportunity to return to a safe state. Examples of safe states include halting memory writes and releasing peripherals. After responding to the fault, the processor can trigger either a local reset or a system reset. This mechanism allows the system to recover from faults caused by safety circuits, including clock supervision, supply supervision, and the multi-counter watchdog timer.

In cases where the fault manager becomes unresponsive or corrupted, these safety circuits can generate their own direct reset signals, bypassing the fault manager entirely. This ensures that the system can recover even in critical fault scenarios.

20.1 Features

CYW20829 supports the following reset sources, each designed to ensure the device transitions to a known and functional state under various conditions:

1. **Power-on reset (POR)**:
   - The POR holds the device in reset while the power supply ramps up to the required level for proper operation
   - The POR activates automatically during power-up
2. **Brown-out detect (BOD) reset**: The BOD monitors the digital voltage supply (VDDD) and generates a reset if VDDD falls below the minimum required logic operating voltage
3. **External reset (XRES)**:
   - A dedicated pin (XRES) allows the device to be reset using an external source
   - The XRES pin is active LOW and can be connected to either a pull-up resistor to VDDD or an active drive circuit
   - If a pull-up resistor is used, its value must minimize current draw when the pin is pulled LOW; a typical value is 10 kΩ
4. **Watchdog timer (WDT or MCWDT)**: The watchdog timer resets the device if the firmware fails to service it within a specified timeout period
5. **Software-initiated reset**: Firmware can trigger a reset on demand by issuing a software-initiated reset command
6. **Logic-protection fault**:
   - A logic-protection fault can trigger an interrupt or reset the device if unauthorized operating conditions are detected
   - For example, this may occur when a debug breakpoint is reached while executing privileged code
7. **Hibernate wakeup reset**: This reset source is used to bring the device out of the system Hibernate low-power mode

Reset events are asynchronous and guarantee the device will revert to a known state. Some reset sources are recorded in a register that is retained through the reset. This allows software to determine the cause of the reset for diagnostic or recovery purposes.

20.2 Architecture

The following sections describe the reset sources available in the CYW20829 device. Table 72 provides a comprehensive list of all reset sources supported by CYW20829.

Table 72 Reset sources in CYW20829

Reset source	Reset condition	Availability in power modes	Observable	SRAM retention
External reset (XRES)	External reset pin asserted low	All	No	No
Power on reset (POR)	VDDD power on reset detected	All	No	No
Brownout detect (BOD)	Brown-out detected for VDDD, VCCD, and VBAT	All except Off, XRES, Hibernate	No	No
WDT reset	Watchdog timer triggered reset action	Active, sleep, deep sleep	Yes	Yes [1)]
MCWDT reset	Multi-counter WDT triggered reset action	Active, sleep, deep sleep	Yes	Yes [1)]
Hibernate wakeup	Exit from Hibernate mode	Hibernate	Yes (on PWR_HIBERNATE.TOKEN field)	No

1) Assumes an orderly shutdown is handled by the processor before the reset is triggered.

***Notes**:*

1. *Since the System RAM is clocked by `clk_hf0`, a reset triggered by `CSV_HF0` may result in an unstable RAM clock, potentially leading to corrupted RAM contents*
2. *None of the reset sources can reset the Backup system. The Backup domain is only reset under the following conditions:*
   - *All power supplies to the Backup domain are removed, referred to as a "cold start"*
   - *Firmware explicitly triggers a reset using the `BACKUP_RESET` register. For more information, see the Backup System chapter*

Table 73 provides a detailed mapping of reset sources to the destinations impacted by each reset. An 'X' in the table indicates that the corresponding block has at least some registers or bits that are affected by the respective reset source.

Table 73 Mapping of reset sources

Reset type/ Source	Reset cause bits	Data registers in FAULT structures	Hibernate registers	Debug logic	RTC	CM33
XRES	X	X	X	X	-	X
POR	X	X	X	X	X [1)]	X

(table continues...)

Table 73 (continued) Mapping of reset sources

Reset type/ Source	Reset cause bits	Data registers in FAULT structures	Hibernate registers	Debug logic	RTC	CM33
BOD	X	X	X	X	-	X
WDT	-	X	X	X [2)]	-	X
MCWDT	-	X	-	X [2)]	-	X
HIB WAKEUP	X	X	-	X	-	X

1) If the backup power supply is not available.
2) If the reset source is triggered while the device is in Deep Sleep mode.

20.2.1 Power-on reset

Power-on reset (POR) keeps the system in a reset state during power-up. It holds the device in reset until the supply voltage (VDDD) reaches the level specified in the datasheet. POR activates automatically at power-up. For more details, see the Power supply and monitoring section.

POR events do not set a reset cause status bit but can be partially inferred by the absence of any other reset source. If no other reset event is detected, the reset is caused by POR, brown-out detect (BOD), or external reset (XRES).

20.2.2 Brownout reset

Brownout reset (BOD) monitors the chip's digital voltage supply (VDDD) and triggers a reset if V_{DDD} drops below the minimum logic operating voltage specified in the device datasheet. For more details, see Power supply and monitoring.

BOD events do not set a reset cause status bit but may be detectable in some cases. During certain BOD events, V_{DDD} may fall below the minimum logic operating voltage specified in the datasheet but remain above the minimum logic retention voltage. In such scenarios, BOD events can be distinguished from power-on reset (POR) events by checking for logic retention. This is explained further in the Identifying reset sources chapter.

20.2.3 WDT reset

The watchdog timer (WDT) reset triggers a reset if the WDT is not serviced by the firmware within a specified timeout period. When a watchdog reset occurs, the `RESET_WDT` or `RESET_MCWDT0` status bits in the `RES_CAUSE` register are set. These bits remain set until they are cleared by the firmware or until a power-on reset (POR), external reset (XRES), or brownout reset (BOD) takes place. All other reset events leave these bits unchanged.

For further information, refer to the Watchdog timer (WDT) section of this manual.

20.2.4 Software reset

A software-initiated reset enables the CPU to request a reset. The Cortex®-M33 can trigger this reset by writing a `1` to the `TRIGGER_SOFT` bit of the `RES_SOFT_CTL` register, either via a debugger probe or through firmware.

When a software reset occurs, the `RESET_SOFT` status bit in the `RES_CAUSE` register is set. This bit remains set until it is cleared by firmware or until a power-on reset (POR), external reset (XRES), or brownout reset (BOD) occurs. All other resets leave this bit unchanged.

In addition to the software reset via the RES_SOFT_CTL register, the MCU provides an option to trigger an XRES reset from the Cortex®-M33. Writing a 1 to the PXRES_TRIGGER bit in the RES_PXRES_CTL register triggers the XRES interrupt. The RES_PXRES_CTL register is software-accessible and initiates a full-scope reset equivalent to XRES. However, it records the reset cause in a different flag within the RES_CAUSE register.

20.2.5 External reset

External reset (XRES) is triggered by an external signal and causes an immediate system reset when asserted. The XRES pin is active LOW, meaning a logic 1 on the pin has no effect, while a logic 0 causes a reset. Internally, the pin is pulled to logic 1 inside the device. The XRES 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 remains at logic 0. When the pin is released (changed to logic 1), the device performs a normal boot sequence. The logical thresholds for XRES and other electrical characteristics are detailed in the "Electrical Specifications" section of the device datasheet. The XRES functionality is available in all power modes but cannot reset the Backup system.

An XRES event does not set a reset cause status bit. However, it can be partially inferred by the absence of any other reset source. If no other reset event is detected, the reset is caused by POR, BOD, or XRES.

20.2.6 Logic protection fault reset

A logic protection fault reset detects unauthorized protection violations and triggers a device reset when such violations occur. An example of a protection fault is hitting a debug breakpoint while executing privileged code.

20.2.7 Hibernate wakeup reset

A hibernate wakeup reset occurs when one of the Hibernate wakeup sources triggers a device reset to return the system to Active power mode. For more information about Hibernate mode and the available wakeup sources, see Device power modes.

The TOKEN field in the PWR_HIBERNATE register is an 8-bit field retained through the Hibernate wakeup sequence. Firmware can use this field to differentiate a hibernate wakeup reset from a general reset event.

20.3 Identifying reset sources

When the device comes out of reset, it is often useful to determine the cause of the most recent or even older resets. This is accomplished using the RES_CAUSE and RES_CAUSE2 registers, which allocate specific status bits for certain reset sources. These registers record the occurrences of watchdog timer (WDT) resets, software resets, and logic-protection fault resets. However, they do not record occurrences of power-on resets (POR), brown-out detect (BOD) resets, external resets (XRES), or hibernate wakeup resets.

The bits in these registers are set when the corresponding reset occurs and remain set after the reset until cleared by firmware or a loss of retention, such as from a POR, XRES, or BOD event.

Hibernate wakeup resets can be identified by examining the TOKEN field in the PWR_HIBERNATE register, as previously described. However, hibernate wakeup resets caused by an XRES cannot be detected. Other reset sources can be inferred to some extent by analyzing the status of the RES_CAUSE and RES_CAUSE2 registers, as shown in Table 74.

Table 74 Reset cause bits to detect reset source

Register	Bitfield	Number of bits	Description
RES_CAUSE[0]	RESET_WDT	1	A hardware watchdog timer (WDT) reset occurred since the last power cycle.
RES_CAUSE[3]	RESET_TC_DBGRESET	1	The test controller or debugger asserted a reset, which only resets the debug domain.
RES_CAUSE[4]	RESET_SOFT	1	The CPU requested a system reset by writing to TRIGGER_SOFT. This can be done through a debugger probe or via firmware.
RES_CAUSE[5]	RESET_MCWDT0	1	A multi-counter watchdog timer (MCWDT) reset (#0) occurred since the last power cycle.
RES_CAUSE2[15:0]	RESET_CSV_HF	16	The clock supervision logic requested a reset due to a 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	The clock supervision logic requested a reset due to a loss or frequency violation of the reference clock source used to monitor other high-frequency clock sources.

For more details, refer to the RES_CAUSE and RES_CAUSE2 registers in the registers technical reference manual (TRM).

If these methods fail to identify the cause of the reset, it can be one of the non-recorded and non-retention resets: brown-out detect (BOD), power-on reset (POR), or external reset (XRES). These resets cannot be distinguished using on-chip resources.

20.4 Registers summary

Table 75 Reset system register list

Register	Description
RES_CAUSE	Reset cause observation register

(table continues...)

Table 75 (continued) Reset system register list

Register	Description
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
PWR_HIB_DATA	Retains its contents through Hibernate wakeup reset
RES_SOFT_CTL	Application interrupt and reset control register of Cortex®-M33
RES_PXRES_CTL	Control register for programmable external reset (PXRES).

21 GPIO

CYW20829 supports up to 32 general-purpose input/output (GPIO) pins, which offer the following functionalities:

- Eight drive-strength modes
- Analog input mode (input and output buffers disabled) on select I/Os
- Input-only mode
- Weak pull-up with strong pull-down
- Strong pull-up with weak pull-down
- Open-drain with strong pull-down
- Open-drain with strong pull-up
- Strong pull-up with strong pull-down
- Weak pull-up with weak pull-down
- Hold mode for latching the previous state (used to retain the I/O state during Hibernate and Deep Sleep modes)
- Selectable slew rates for dV/dt-related noise control to improve EMI performance

The GPIO pins are organized into logical groups called ports, which can include up to eight pins each. Data output registers store values to be driven on the pins, while pin state registers capture the input states of the pins. Each pin can generate an interrupt if enabled, and every port has an associated interrupt request (IRQ). Port 4 pins support overvoltage-tolerant (OVT) operation, allowing input voltages higher than V_{DDD}. OVT pins are commonly used with I^2C to enable powering off the chip while maintaining physical connectivity to an active I^2C bus without affecting its operation.

GPIO pins can be grouped to source or sink higher currents. However, all GPIO pins, including OVT pins, must not exceed the absolute maximum pull-up voltage specified in the device datasheet's electrical specifications. During power-on and reset states, the pins are set to analog input drive mode, with both input and output buffers disabled, to prevent crowbarring of inputs and excessive turn-on current. A high-speed I/O matrix (HSIOM) handles signal multiplexing, enabling various peripheral and analog signals to connect to an I/O pin.

21.1 Features

- Separate Set, Clear, and Invert registers for output
- Interrupt configurations available on all GPIO pins, with edge-triggering on rising edges, falling edges, or both
- Frozen mode for latching the current state of GPIO pins
- Overvoltage-tolerant pins (`OVT_GPIO`)
- Eight drive modes, including pull-up, pull-down, and open-drain drive High/Low
- Slew-rate control to manage speed for each GPIO pin
- Analog and digital input and output capabilities

21.2 Architecture

The GPIO pins connect to I/O cells, which include an input buffer for digital input with high input impedance and a driver for digital output signals. The I/O cells are connected to digital peripherals through the high-speed I/O matrix (HSIOM). Each pin's HSIOM contains multiplexers that connect the selected peripheral to the pin.

21 GPIO

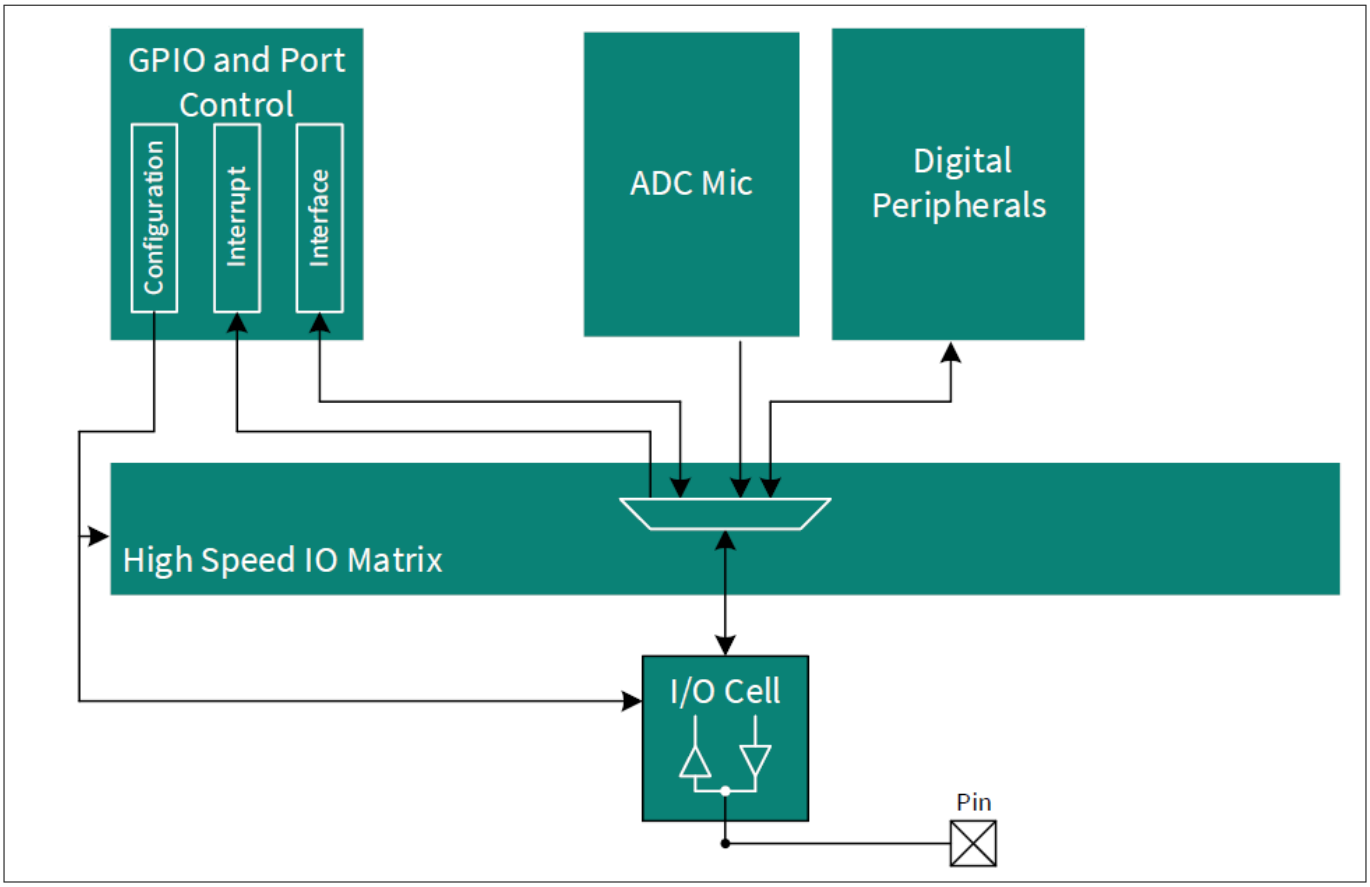


Figure 60 GPIO architecture

21.2.1 I/O cell architecture description

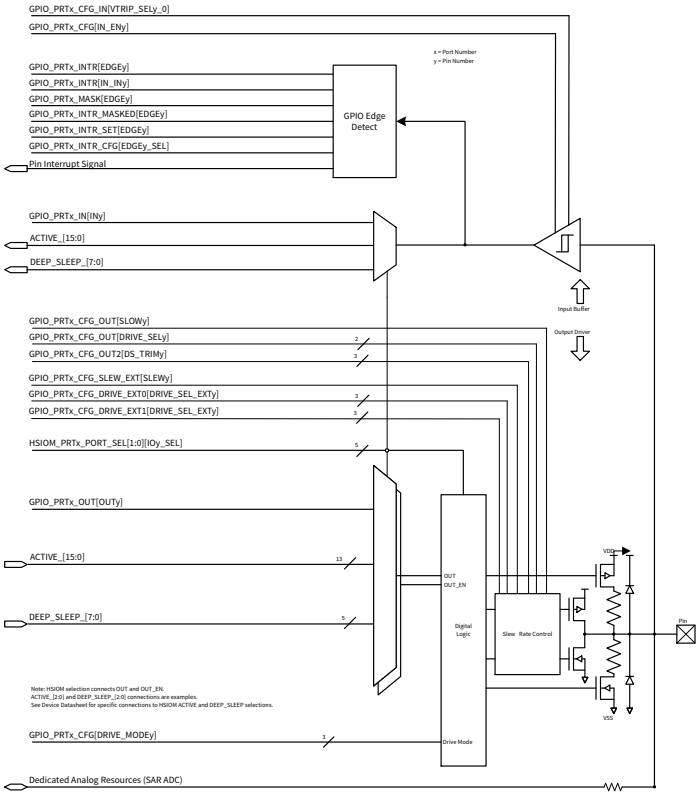


Figure 61 GPIO cell architecture

The configuration information for I/O cells in the GPIO component is communicated through registers. In the System Deep Sleep power mode, these registers are preserved, while in the Hibernate power mode, they are reset to their default values. To enable the Hibernate Interrupt feature, the configuration of the I/O cells is preserved or frozen during transitions into System Deep Sleep or Hibernate power modes. This ensures the configuration signals can be directed within the System High-Power (HP), Low-Power (LP), or Ultra-Low-Power (ULP) modes.

The configuration of I/O cells includes various parameters such as drive mode (pull-up or pull-down) and drive strength. It may also include settings specific to individual I/O pads, such as drive mode, drive strength, slew-rate control (fast or slow), and input buffer mode. Through the I/O port, software can access the I/Os to write output signals or read input signals, enabling bit-banging operations.

The signals for GPIO data input, data output, and data output enable within I/O cells are routed via the high-speed I/O matrix (HSIOM) functional connections. The HSIOM register fields determine the specific connections.

21.2.2 High-speed I/O matrix (HSIOM)

The high-speed I/O matrix (HSIOM) consists of high-speed multiplexers that route internal CPU and peripheral signals to and from GPIOs. The HSIOM enables GPIOs to support multiple functions by multiplexing pin connections to user-selected peripherals. The `HSIOM_PRTx_PORT_SEL[4:0]` registers allow each pin to select a single connection from up to 32 different options.

Table 76 HSIOM values

Alternate function 5-bit [4:0]	Name	Digital driver signal source	Digital input signal destination		Analog switches		Description
			OUT	OUT_EN	AMUXA	AMUXB	
0	GPIO	OUT register	1	IN register	0	0	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	0	0	Active functionality 0 - See the datasheet for specific pin connectivity.
9	ACT_1	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 1 - See the datasheet for specific pin connectivity.
10	ACT_2	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 2 - See the datasheet for specific pin connectivity.
11	ACT_3	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 3 - See the datasheet for specific pin connectivity.
12	DS_0	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	0	0	Deep Sleep functionality 0 - See the datasheet for specific pin connectivity.

(table continues...)

Table 76 (continued) HSIOM values

Alternate function 5-bit [4:0]	Name	Digital driver signal source	Digital input signal destination		Analog switches		Description
			OUT	OUT_EN	AMUXA	AMUXB	
13	DS_1	Deep Sleep Source OUT	Deep Sleep Source OUT_EN	Deep Sleep IN	0	0	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	0	0	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	0	0	Deep Sleep functionality 3 - See the datasheet for specific pin connectivity.
16	ACT_4	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 4 - See the datasheet for specific pin connectivity.
17	ACT_5	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 5 - See the datasheet for specific pin connectivity.
18	ACT_6	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 6- See the datasheet for specific pin connectivity.
19	ACT_7	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 7 - See the datasheet for specific pin connectivity.

(table continues...)

Table 76 (continued) HSIOM values

Alternate function 5-bit [4:0]	Name	Digital driver signal source	Digital input signal destination		Analog switches		Description
			OUT	OUT_EN	AMUXA	AMUXB	
20	ACT_8	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 8 - See the datasheet for specific pin connectivity.
21	ACT_9	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 9 - See the datasheet for specific pin connectivity.
22	ACT_10	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 10 - See the datasheet for specific pin connectivity.
23	ACT_11	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 11 - See the datasheet for specific pin connectivity.
24	ACT_12	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 12 - See the datasheet for specific pin connectivity.
25	ACT_13	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 13 - See the datasheet for specific pin connectivity.
26	ACT_14	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	Active functionality 14 - See the datasheet for specific pin connectivity.

(table continues...)

Table 76 (continued) HSIOM values

Alternate function 5-bit [4:0]	Name	Digital driver signal source	Digital input signal destination		Analog switches		Description
			OUT	OUT_EN	AMUXA	AMUXB	
27	ACT_15	Active Source OUT	Active Source OUT_EN	Active Source IN	0	0	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	0	0	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	0	0	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	0	0	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	0	0	Deep Sleep functionality 7 - See the datasheet for specific pin connectivity.

***Note**: The Active and Deep Sleep sources are pin dependent. Refer to the Pinouts section of the device datasheet for detailed information on the features supported by each pin.*

21.2.3 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, GPIOs can be configured by writing to the associated registers. However, pins that support the debug access port (DAP) connections, such as the SWD lines, are always enabled as SWD lines during power-up. If these pins are left floating, there is a risk of crowbar current because the DAP connection does not include pull-up or pull-down resistors. To avoid this issue, the DAP connection can be disabled or reconfigured for general-purpose use through the HSIOM, but this should be done only after the device has booted and started executing code.

If the JTAG input pin is initially set to SWJ_TRSTN mode (refer to the related device datasheet for the specific pin number), it can be changed to a GPIO pin using the steps below:

1. Set HSIOM_PRTx_PORT_SEL to 0 for that pin
2. Set GPIO_PRTx_CFG to 0 for that pin

21.2.4 Behavior in low-power modes

Table 77 shows the status of GPIOs in low-power modes.

Table 77 GPIO in low-power modes

Low-power mode	Status
CPU Sleep	Standard GPIO, GPIO-OVT, and SIO pins are active and can be driven by most peripherals, such as TCPWMs and SCBs, which operate in CPU Sleep mode. Input buffers are active, allowing an interrupt on any I/O to wake the CPU.
System Deep Sleep	GPIO, GPIO-OVT, and SIO pins connected to System Deep Sleep domain peripherals remain functional. All other pins retain their last output driver state and configuration. Pin interrupts are functional on all I/Os and can be used to wake the device.
System Hibernate	Pin output states and configurations are latched and remain in a frozen state. Pin interrupts are functional only on select I/Os and can be used to wake the device. See the device datasheet for detailed information on specific Hibernate pin connectivity.

21.3 Functional description

21.3.1 Digital input buffer

The digital input buffer provides a high-impedance interface for external digital inputs. The buffer is enabled or disabled by the GPIO_PRTx_CFG.IN_ENy bit, where x represents the port number and y represents 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) determines the pin connection. See the device datasheet for details on the specific connections available for each pin. A port pin can function as both an input and an output simultaneously. If a pin is used solely for analog signals, the input buffer should be disabled to avoid crowbar currents.

Each pin's input buffer trip point and hysteresis can be configured for the following modes:

- CMOS + I^2C
- TTL
- Industrial

CMOS and TTL buffer modes are selected through the GPIO_PRTx_CFG_IN.VTRIP_SELy_0 bit. To enable or disable the industrial mode, use the GPIO_PRTxCFG_IN_AUTOLVL.VTRIP_SELy_1 bit.

***Note**: Set the GPIO_PRTx_CFG_IN mode to CMOS before enabling the industrial mode. The trip levels for CMOS and industrial modes vary.*

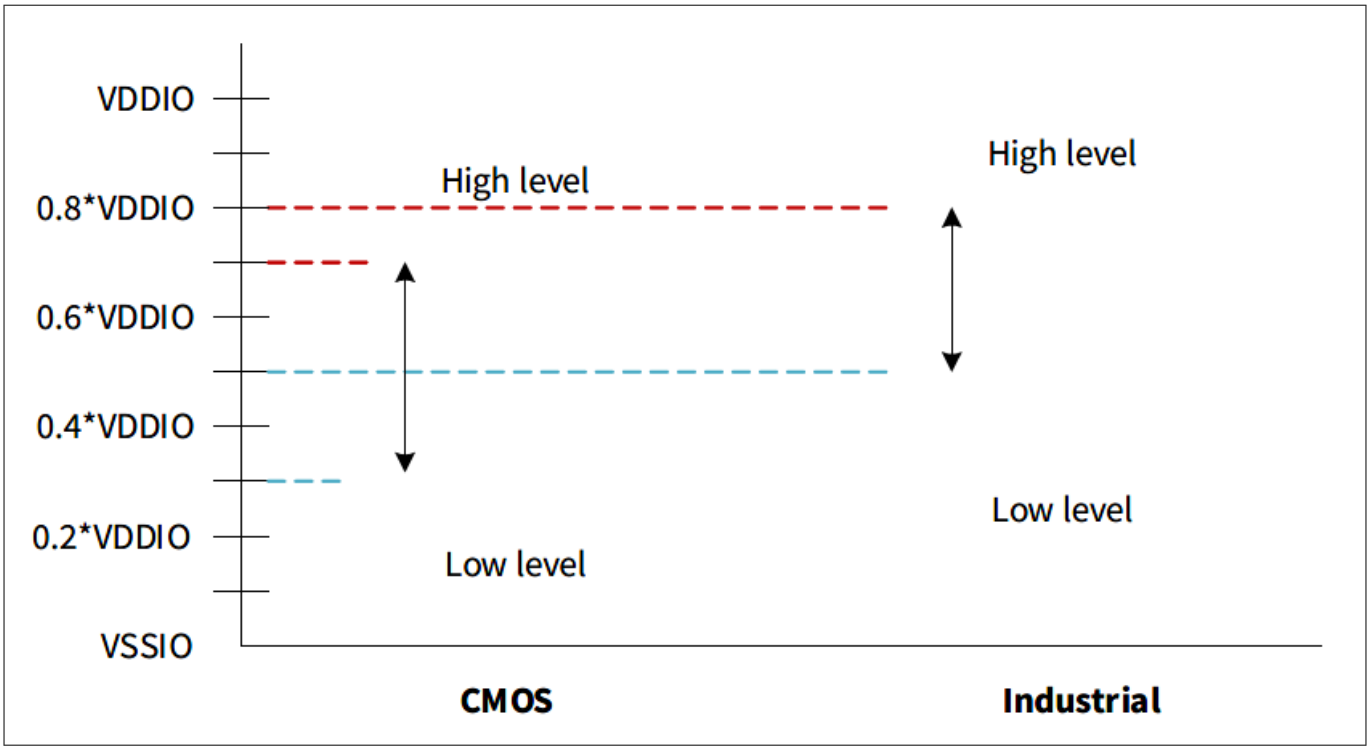


Figure 62 Input buffer mode tripping levels

21.3.2 Digital output buffer

Pins are driven by the digital output driver, which includes circuitry to implement various drive modes and slew-rate control for digital output signals. The HSIOM selects the control source for the output driver. The two primary control sources are:

- Port configuration registers
- Fixed-function digital peripherals

A specific HSIOM connection is selected by writing to the HSIOM Port Select register (`HSIOM_PRTx_PORT_SEL`). Each GPIO pin is equipped with 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 (V_{DDIO}, V_{DDD}, V_{DDA}) or drop below the I/O ground levels (V_{SSIO}, V_{SSD}, V_{SSA}). See the device datasheet for the absolute maximum and minimum allowable GPIO voltage levels.

The digital output driver can be enabled or disabled in hardware by the Output Data register (`GPIO_PRTx_OUT`) associated with the output pin. Peripherals other than the GPIO port can directly control both the output and the output enable of the output buffer.

21.3.2.1 GPIO-OVT pins

Certain device pins are overvoltage-tolerant (OVT) and are designed for interfacing with buses or other signals that may exceed the pin's V_{DDIO} supply. They are also useful in scenarios where the device supply or pin V_{DDIO} is not always present. These pins function identically to regular GPIOs, with the added capability of overvoltage tolerance. GPIO-OVT pins include hardware that compares the `VDDIO` voltage to the pin voltage. If the pin voltage exceeds `VDDIO`, the output driver is disabled, and the pin driver enters a tristated condition. This prevents significant current sinking at the pin and ensures minimal power consumption.

It is important to note that during overvoltage conditions, the input buffer data may become invalid if the external source's specified VOH (output high voltage) and VOL (output low voltage) do not align with the trip points of the input buffer, which are defined by the current V_{DDIO} voltage.

21.3.2.2 Slew rate control

Some GPIO pins support fast and slow output slew-rate options for strong drivers. These options are configured using the SLOW bit in the Port Output Configuration register (GPIO_PRTx_CFG_OUT). By default, the SLOW bit is cleared, and the port operates in fast slew-rate mode. If a slower slew rate is required, this bit can be set. A slow slew rate reduces 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 adjusted to one of four levels using the GPIO_PRTx_CFG_OUT.DRIVE_SELy field. The drive strength field determines the active portion of the output drivers and can influence the slew rate of output signals. The drive strength options are:

- Full drive strength (default)
- Half drive strength
- Quarter drive strength

When the slow slew-rate bit (SLOW) is set, the drive strength must be configured to full drive strength to ensure proper operation.

21.3.2.3 Drive strength

The drive strength of an I/O pin specifies the amount of current the pin can drive or sink while maintaining the minimum VOH and VOL levels. When the output is configured for a fast slew rate, the drive strength can be set to one of four levels using the DRIVE_SEL field in the port output configuration register (GPIO_PRTx_CFG_OUT). The drive strength field determines the active portion of the output drivers and can influence the slew rate of output signals.

The drive strength options are as follows:

- Full drive strength (default)
- Half drive strength
- Quarter drive strength
- One-eighth drive strength

When the slow slew-rate bit (SLOW) is set, the drive strength must be configured to full drive strength for proper operation. The I/O port includes two advanced registers, called the Port Output Drive Select Extension register (GPIO_PRTx_CFG_DRIVE_EXT[0-1]), to configure the drive width of each pin in the respective port. Additionally, the Port Output Configuration Register 2 (GPIO_PRTx_CFG_OUT2) is available to control the drive select trim of each pin in the respective port.

Table 78 Drive select for GPIO_STD

Drive_select	DRIVE_SEL[0:1]	Description
DRIVE_SEL_ZERO	00	Full drive strength: GPIO drives current at its maximum rated specification.
DRIVE_SEL_ONE	01	Full drive strength: GPIO drives current at its maximum rated specification.
DRIVE_SEL_TWO	10	1/2 drive strength: GPIO drives current at one-half of its maximum rated specification.
DRIVE_SEL_TREEE	11	1/4 drive strength: GPIO drives current at one-quarter of its maximum rated specification.

Table 79 Drive select trim of GPIO_PRTx_CFG_OUT2

Drive_select	DS_TRIM[0:2]	Description
DEFAULT	000	Default(50 Ω)
DS_120 Ω	001	150 Ω
DS_90 Ω	010	90 Ω
DS_60 Ω	011	60 Ω
DS_50 Ω	100	50 Ω
DS_30 Ω	101	30 Ω
DS_20 Ω	110	20 Ω
DS_15 Ω	111	15 Ω

21.3.2.4 Drive mode

Each I/O pin is individually configurable to one of eight drive modes using the `DRIVE_MODE[7:0]` field in the Port Configuration register (`GPIO_PRTx_CFG`).

Table 80 lists the available drive modes configured through the `GPIO_PRTx_CFG` register. It provides a detailed breakdown of the drive modes in two distinct columns:

- GPIO functionality
- Fixed peripheral functionality

Table 80 Drive mode settings

Drive mode	Value	GPIO port configuration register				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

(table continues...)

Table 80 (continued) Drive mode settings

Drive mode	Value	GPIO port configuration register				Fixed-function digital peripheral			
		OUT_EN = 1		OUT_EN = 0		OUT_EN = 1		OUT_EN = 0	
Resistive pull up or pull down	7	Weak 1	Weak 0	High-Z	High-Z	Strong 1	Strong 0	Weak 1	Weak 0

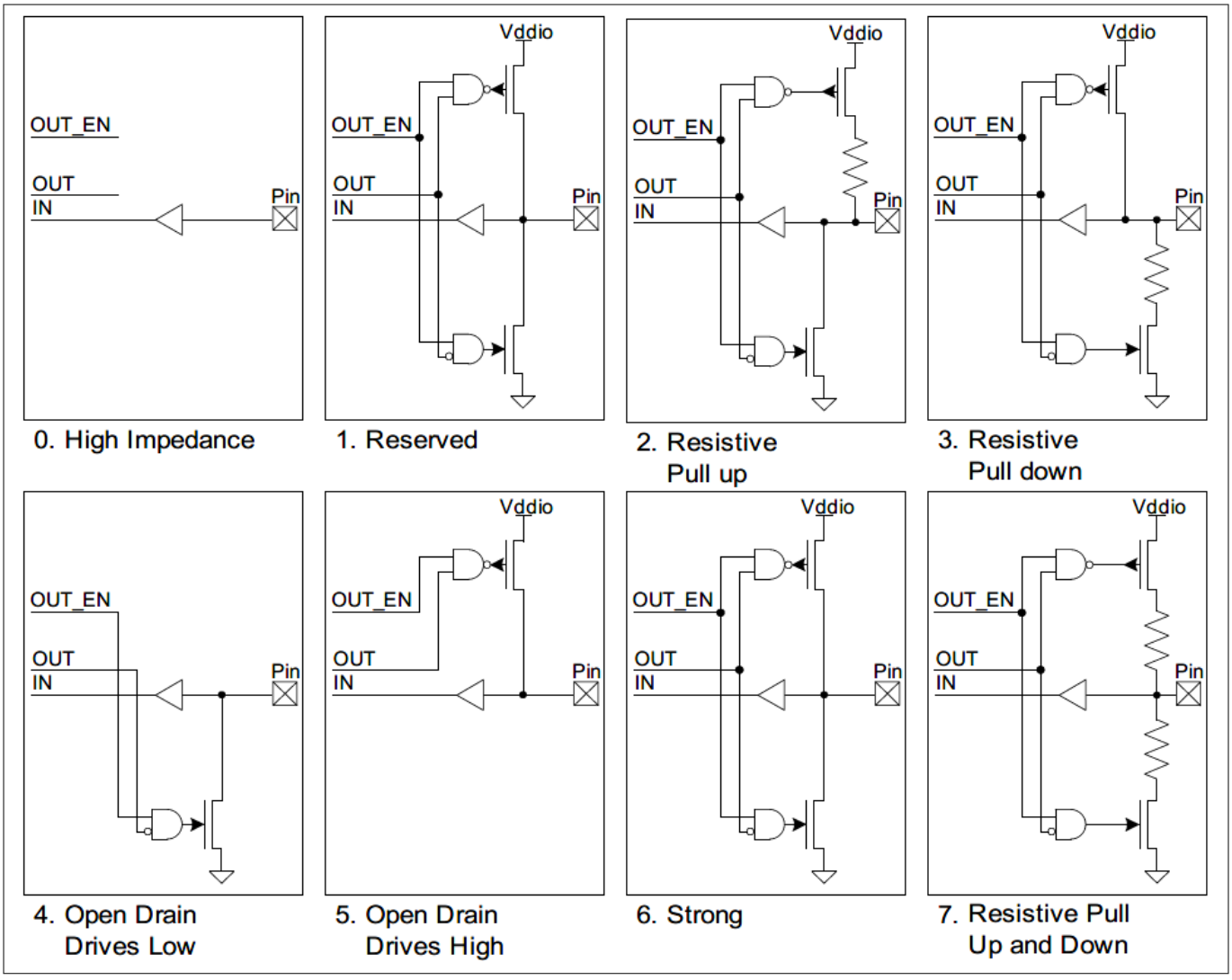


Figure 63 GPIO port drive mode block diagrams

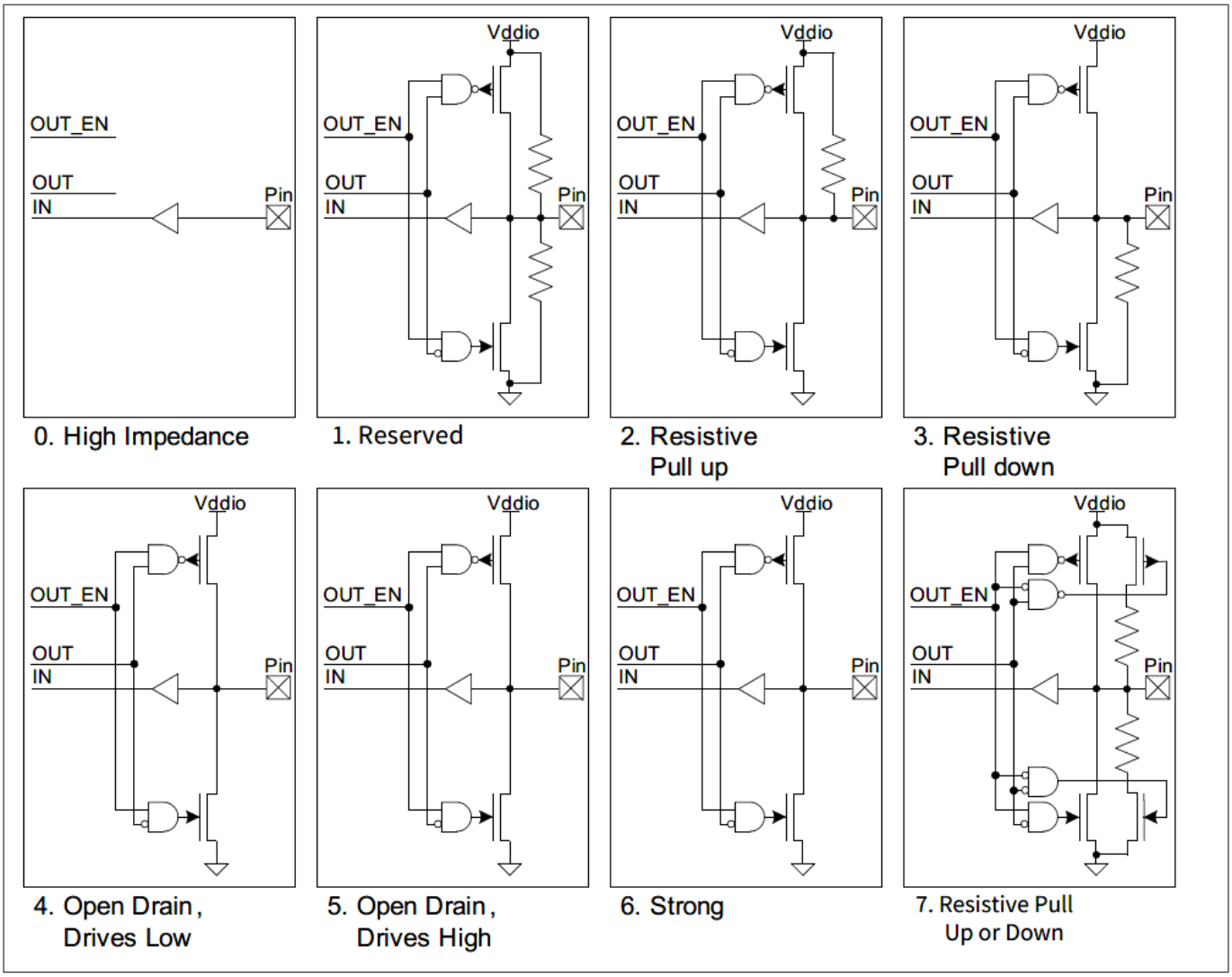


Figure 64 Fixed-function peripheral I/O drive mode block diagrams

The types of GPIO drive modes are as follows:

1. **High-impedance mode (High-Z)**: High-impedance mode is recommended for analog and digital inputs. For digital signals, the input buffer is enabled, while for analog signals, the input buffer is disabled to minimize crowbar current and leakage in low-power designs. This is the default pin reset state, with the input buffer disabled. For the lowest device current, unused GPIOs should be configured in high-impedance mode with the input buffer disabled
2. **Resistive pull-up mode**: In resistive pull-up mode, a series resistance is present in the low state, and strong drive is applied in the high state. Pins in this mode can function as digital inputs or outputs. To enable the pull-up feature, write '1' to the pin's data register bit. This mode is commonly used for interfacing mechanical switches or for open-drain low input scenarios
3. **Resistive pull-down mode**: Resistive pull-down mode includes a series resistance in the high state and strong drive in the low state. Like pull-up mode, it supports digital input or output functionality. Write '0' to the pin's data register bit to enable pull-down functionality. This mode is often used for open-drain high input applications
4. **Open-drain drive high mode**: Open-drain drive high mode provides high impedance in the low state and strong drive in the high state. It is typically used when the signal is externally pulled down. These modes are suitable for bi-directional digital communication

5. **Open-drain drive low mode**: Open-drain drive low mode provides high impedance in the high state and strong drive in the low state. It is used where signals are externally pulled high and is commonly applied to drive I^2C bus signal lines
6. **Strong drive mode**: The strong drive mode offers a strong CMOS output drive in both the high and low states. It is used exclusively for digital outputs and is often employed to generate digital output signals or drive external devices. Strong drive mode is not suitable for inputs under normal conditions
7. **Resistive pull-up or pull-down mode**: In resistive pull-up or pull-down mode, series resistance is present 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 for avoiding short circuits when the pin is driven by external signals

21.3.3 Firmware-controlled GPIO

For standard firmware-controlled GPIO operations using registers, the GPIO mode must be configured 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 reflect the actual logic level present on the GPIO pins, which might differ

Using the GPIO_PRTx_OUT register, read-modify-write sequences can be safely performed on ports with both input and output GPIOs. In addition to the GPIO_PRTx_OUT register, three other registers are available for specific operations:

- GPIO_PRTx_OUT_SET: Sets the output data for specific pins in a port
- GPIO_PRTx_OUT_CLR: Clears the output data for specific pins in a port
- GPIO_PRTx_OUT_INV: Inverts the output data for specific pins in a port

These registers allow users to set, clear, or invert the state of selected pins in a port without impacting other pins. This eliminates the need for read-modify-write operations in most use cases. Writing a '1' to the corresponding bit field in these registers performs the respective operation (set, clear, or invert) on the pin, while writing a '0' has no effect on the pin state. The GPIO_PRTx_IN register is the Port I/O Pad register that provides the actual logic level present on the GPIO pins when read. Write operations to this register have no effect.

21.4 Interrupt

All port pins can generate interrupts. There are two routing methods for pin signals to generate interrupts:

- The pin signal is routed through the GPIO edge detect block with a direct connection to the CPU interrupt controller
- The pin signal is routed through the port adapter and digital system interconnection (DSI) to the CPU interrupt controller

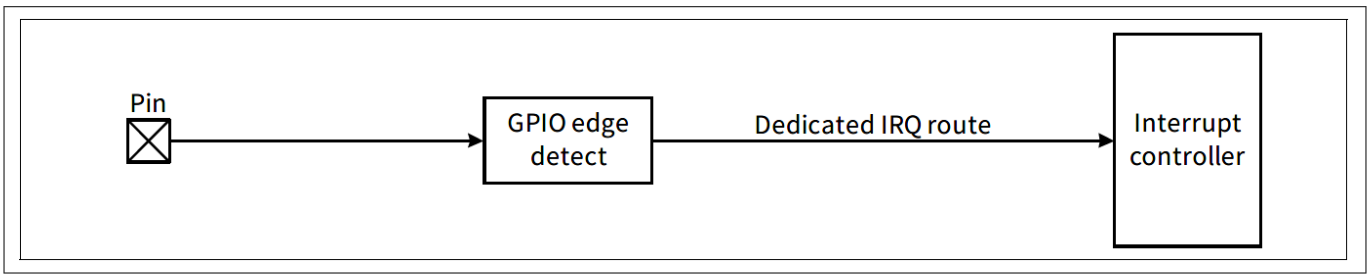


Figure 65 Interrupt signal routing

Interrupt generation is independent of the HSIOM configuration. It is recommended to disable interrupt generation (GPIO_PRTx_INTR_CFG.EDGEy_SEL = 00) to avoid unwanted GPIO interrupts. Figure 66 shows the architecture of the GPIO edge detect block. Each GPIO pin interrupt can be triggered by either a rising or falling edge on the GPIO pin signal, based on the configuration of the GPIO_PRTx_INTR_CFG register. Each of the up to eight GPIO pads in an I/O port has interrupt cause fields that are set to 1 when a GPIO pin input signal exhibits rising or falling edge changes. These edges are captured by the GPIO_PRTx_INTR register.

The GPIO_PRTx_INTR_MASK register for the I/O port specifies which interrupts are propagated as outputs. This propagated GPIO pin interrupt functions as a Deep Sleep interrupt, allowing the GPIO input signal change to wake the CPU from Deep Sleep power mode. This occurs when combined with GPIO port i pin interrupts to form a port interrupt (interrupts_gpio[i]) as shown in Figure 66. Additionally, a register field specifies one specific I/O input signal to route through a 50-ns glitch filter (GPIO_PRTx_INTR_CFG.FLT_SEL register). The glitch filter output has dedicated detection circuitry with dedicated detection control fields, such as GPIO_PRTx_INTR.FLT_EDGE.

Each I/O port is associated with a dedicated interrupt output (interrupts_gpio[i] for port i). Figure 65 shows the interrupt functionality, while Figure 66 shows the GPIO edge detect block architecture.

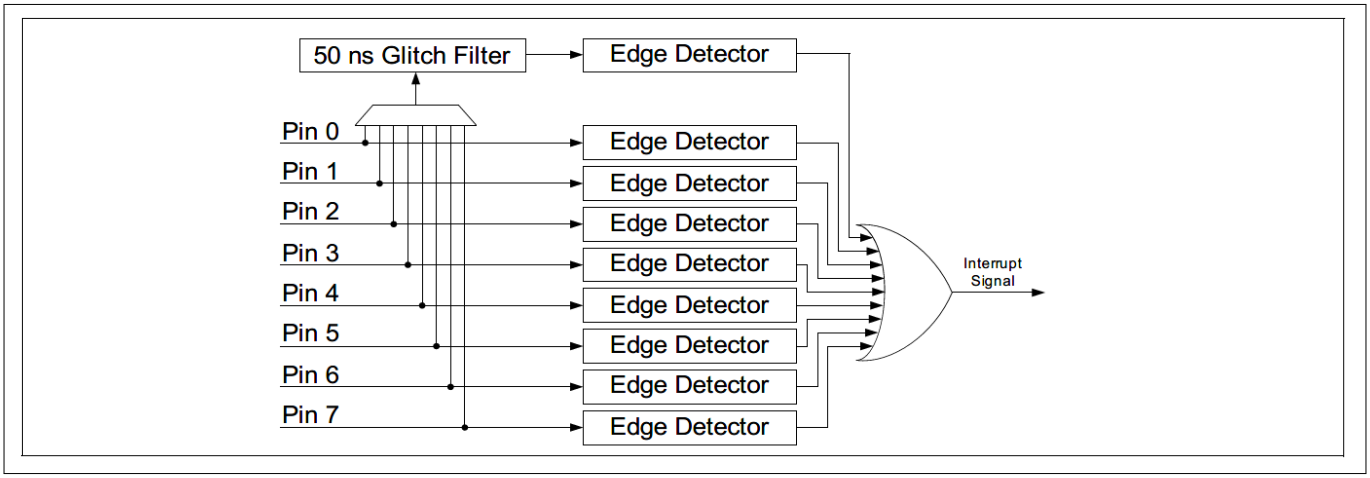


Figure 66 GPIO edge detect block architecture

Table 81 Edge select configurations

Edge select	Configurations
00	Interrupt is disabled
10	Interrupt on rising edge
11	Interrupt on falling edge
11	Interrupt on both edges

In addition to the pins, each port includes a glitch filter connected to its own edge detector. This filter is driven by one of the pins within the same port.

Table 82 FTL select configurations

FTL_SEL	Selected pin
000	Pin 0 is selected
001	Pin 1 is selected
010	Pin 2 is selected

(table continues...)

Table 82 (continued) FTL select configurations

FTL_SEL	Selected pin
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, the GPIO_PRTx_INTR (Port Interrupt Status) register can be read to determine which pin caused the edge. This register provides two pieces of information in a single read operation:

- Latched information indicating which pin detected an edge
- The current status of the pin

Additionally, this register can be used to clear the latched edge state. If the signal is routed through the DSI, bypassing the edge detect block, edge detection should be configured in the interrupt source multiplexer block.

- If the multiplexer is configured in Level mode, the interrupt is triggered repeatedly as long as the pin signal remains high
- Use the rising edge detect option when this route is selected to generate only one interrupt per edge

21.4.1 Analog I/O

Analog resources, such as ADCMIC and external crystal pins for WCO, which require low-impedance routing paths, are assigned to fixed pins. These pins provide direct connections to specific analog blocks through the AMUXBUS. They are optimized for performance and should be prioritized over other pins when using these analog resources. Refer to the device datasheet for details on these dedicated pins.

To configure a GPIO pin as an analog pin connected to the AMUXBUS:

1. Set the GPIO to high-impedance analog mode with the input buffer disabled
2. Route the pin to the appropriate AMUXBUS using the HSIOM_PORT_SELx register

21.5 Programmable digital Smart I/O

Smart I/O adds programmable logic functionality to an I/O port. It integrates Boolean logic operations, such as AND, OR, and XOR, directly into the port. Smart I/O also enables pre- or post-processing of signals between the high-speed I/O matrix (HSIOM) and the I/O port. For example, it can function as digital glue logic by managing input signals with multiple flip-flops without requiring CPU intervention. The HSIOM multiplexes GPIOs, sharing multiple functionalities, into user-selected peripheral devices.

Features:

- Integrates board-level Boolean logic functionality into a port
- Supports pre-processing of HSIOM input signals from GPIO port pins
- Supports post-processing of HSIOM output signals to GPIO port pins
- Compatible with all device power modes except Hibernate mode
- Closely integrates with I/O pads, providing the shortest signal paths with programmability

21.5.1 Overview

This design uses a PWM resource and a Smart I/O resource to generate square waves with slightly different frequencies. These square waves are routed through an exclusive-OR (XOR) gate within the Smart I/O resource, producing a signal with a gradually changing duty cycle. The rate at which the duty cycle changes is proportional to the difference in frequencies between the square waves. This signal is output to IO1 of Smart I/O port 9. When an LED is connected to this output, it exhibits a ramping effect where the LED alternately brightens and dims gradually.

A 10 kHz clock drives the PWM resource, configured with a period of 399 counts and a compare value of 200 counts. This setup generates a 50 percent duty-cycle square wave with a 40 ms period. The Smart I/O is clocked at 99 Hz using a divided clock sourced from `CLK_PERI`. This clock is further divided by four within the lookup tables (LUTs) of the Smart I/O resource, producing a square wave with a 40.4 ms period.

To generate a square wave signal with a time period close to 40 ms, the 99 Hz clock is divided by four using a synchronous sequential circuit implemented with the LUTs of the Smart I/O resource.

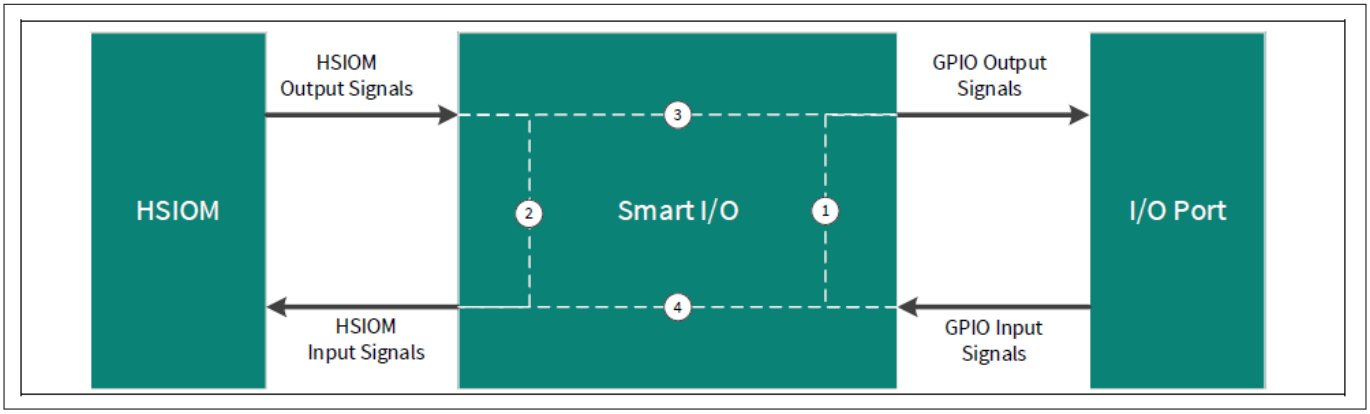


Figure 67 CYW20829 Smart I/O architecture block diagram

The Smart I/O block is positioned in the signal path between the HSIOM and the I/O port. The HSIOM multiplexes output signals from fixed-function peripherals and the CPU to specific port pins and vice versa. The Smart I/O block acts as a bridge in this signal path, allowing it to process signals between the port pins and the HSIOM.

The signal paths supported through the Smart I/O block are as follows:

- Implement self-contained logic functions that directly operate on port I/O signals
- Implement self-contained logic functions that operate on HSIOM signals
- Operate on and modify HSIOM output signals, then route the modified signals to port I/O signals
- Operate on and modify port I/O signals, then route the modified signals to HSIOM input signals

21.5.2 Block components

21.5.2.1 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 shared by all components within the block. The clock and reset sources are configured using 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, including 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.

It is important to note that the selected clock (`clk_block`) for the block's synchronous logic is not phase-aligned with other synchronous logic in the device that uses the same clock. Hence, communication between the 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 clock is derived from the system clock (`clk_sys`) using a peripheral clock divider
- **Low-frequency system clock (`clk_lf`)**:
  - This clock is available in the System Deep Sleep power mode
  - It has an associated reset signal (`rst_lf_dpslp_n`), which is activated if the system enters Hibernate mode or during a power-on reset (POR)

When the block is enabled, the selected clock (`clk_block`) and associated reset signal (`rst_block_n`) are provided to the fabric components. When the block is disabled, no clock is supplied to the fabric, and the reset is asserted, setting the LUT and data unit components to their reset value of '0'.

The I/O input synchronizers introduce a delay of two `clk_block` cycles when enabled. During the first two cycles of operation, the synchronizers may present stale data at their output. To handle this, the reset is activated for the first two clock cycles, putting the block into bypass mode temporarily.

Table 83 Smart I/O clock and reset register description

Register[BIT_POS]	Bit name	Description
SMARTIO_PRTn_CTL[12:8]	CLOCK_SRC[4:0]	The following options are available for configuring the clock and reset source for the Smart I/O block: **1.** **0-7**: `io_data_in[0]` to `io_data_in[7]`. These GPIO input signals are used as clock sources but do not have an associated reset **2.** **8-15**: `chip_data[0]` to `chip_data[7]`. These HSIOM output signals are used as clock sources but do not have an associated reset **3.** **16**: `clk_smartio/ rst_sys_act_n`. • The clock is derived from the peripheral clock divider • The reset is asserted in all power modes except Low Power (LP) and Ultra-Low Power (ULP) • Smart I/O is active only in LP or ULP power modes **4.** **17**: `clk_smartio/ rst_sys_dpslp_n`. • The clock is derived from the peripheral clock divider • Smart I/O is active in all power modes • The clock is not active in the System Deep Sleep power mode **5.** **19**: `clk_lf/rst_lf_dpslp_n`. • The clock is derived from the ILO • Smart I/O is active in all power modes **6.** **20-30**: Constant '0'. Use any of these clock sources when the Smart I/O block is

(table continues...)

Table 83 (continued) Smart I/O clock and reset register description

Register[BIT_POS]	Bit name	Description
		disabled to optimize for low-power consumption **7. 31**: clk_sys/1 • This selection is not intended for active clk_sys operation • When used for asynchronous operation, three clk_sys cycles are required after enabling the block for it to become fully functional (reset deactivated) • This mode is intended only for asynchronous (clockless) block functionality

21.5.2.2 Synchronizer

Each GPIO input signal and device input signal (HSIOM input) can operate either asynchronously or synchronously. To use these signals synchronously, a double flip-flop synchronizer (see Figure 68) is placed in both signal paths. This synchronizer aligns the signal to the Smart I/O clock (clk_block).

The synchronization for each pin or input can be enabled or disabled as follows:

- Set or clear the IO_SYNC_EN[i] bitfield in the SMARTIO_PRTx_SYNC_CTL register to enable or disable synchronization for a GPIO input signal
- Set or clear the CHIP_SYNC_EN[i] bitfield in the SMARTIO_PRTx_SYNC_CTL register to enable or disable synchronization for an HSIOM input signal

Here, 'i' represents the pin number being configured.

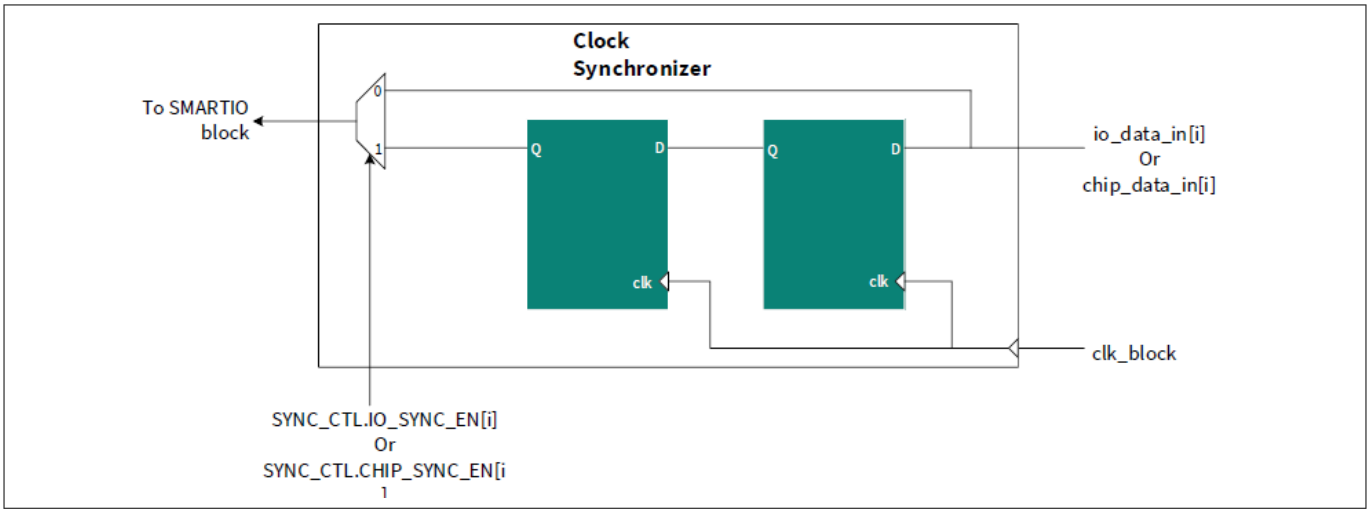


Figure 68 Clock synchronizer

21.5.2.3 Lookup table (LUT)

Each LUT block processes three input signals and generates an output based on the configuration specified in the SMARTIO_PRTx_LUT_CTLy register, where y denotes the LUT number. For each LUT, an 8-bit lookup vector (LUT[7:0]) and a 2-bit opcode (OPC[1:0]) in the SMARTIO_PRTx_LUT_CTLy register determine its operation. The 8-bit vector functions as a lookup table for the three input signals, while the 2-bit opcode specifies the use of the flip-flop. The LUT configuration for different opcodes is shown in Figure 69.

The SMARTIO_PRTx_LUT_SELy registers are used to select the three input signals (tr0_in, tr1_in, and tr2_in) for each LUT. The input signals can originate 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])

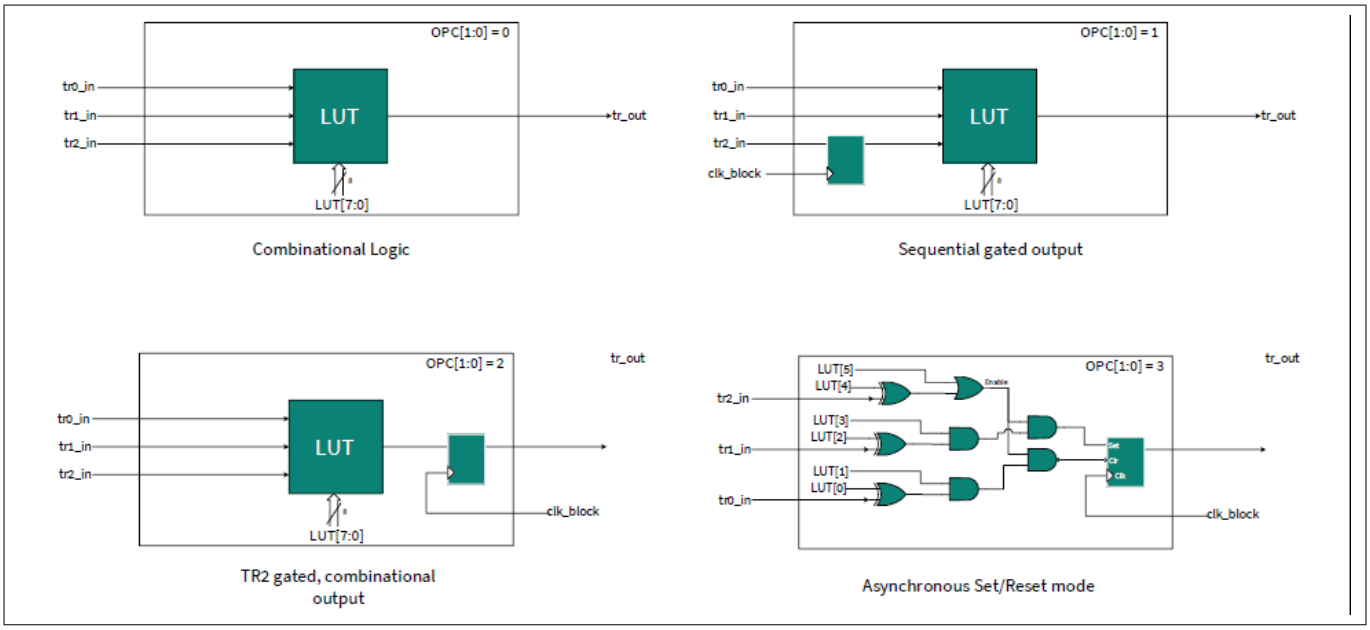


Figure 69 LUT operations

The LUT supports the following four operations, which are selected using a 2-bit Op Code field (see Figure 69):

1. **Combinatorial LUT**:
   - This mode is purely combinatorial
   - Each LUT output is determined by the truth table mapping for the LUT
   - The output is delayed only by the LUT3[x] combinatorial path (basic mode)
2. **Gated input**:
   - The LUT3 input 2 is registered, while the other inputs are directly connected to LUT3[x]
   - The output remains combinatorial
   - This mode enables input synchronization
3. **Gated output**:
   - All inputs are directly connected to LUT3[x], and the output is registered
   - This mode enables output synchronization
4. **Set/Reset flip-flop**: Input signals control a set/reset flip-flop (S/R flip-flop)

Table 84 Register description

Register[BIT_POS]	Bit name	Description
SMARTIO_PRTx_LUT_CTLy[7:0]	LUT[7:0]	The LUT configuration uses the LUT opcode (LUT_OPC), the internal state, and the LUT input signals (tr0_in, tr1_in, and tr2_in) 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 operation of the LUT, as shown in Figure 69.
SMARTIO_PRTx_LUT_SELy[3:0]	LUT_TR0_SEL[3:0]	LUT input signal “tr0_in” source selection:0: Data unit output1: LUT 1 output2: LUT 2 output3: LUT 3 output4: LUT 4 output5: LUT 5 output6: LUT 6 output7: LUT 7 output8: 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 84 (continued) Register description

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 output1: LUT 1 output2: LUT 2 output3: LUT 3 output4: LUT 4 output5: LUT 5 output6: LUT 6 output7: LUT 7 output8: 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]	The encoding for `tr2_in` source selection is the same as for `LUT_TR1_SEL`.

21.5.2.4 Data unit (DU)

Each Smart I/O block includes a Data Unit (DU) component. The DU consists of a simple 8-bit data path capable of performing basic operations, such as increment, decrement, increment/decrement, shift, and AND/OR. The operation performed by the DU is selected using the 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`, and `tr2_in`), similar to the LUT component. These trigger signals initiate the operation defined by the DU opcode.

In addition, the DU 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 as a reference. The 8-bit data input source is configured as one of the following:

- Constant `0x00`
- `io_data_in[7:0]`
- `chip_data_in[7:0]`
- `DATA[7:0]` bitfield of the `SMARTIO_PRTx_DATA` register

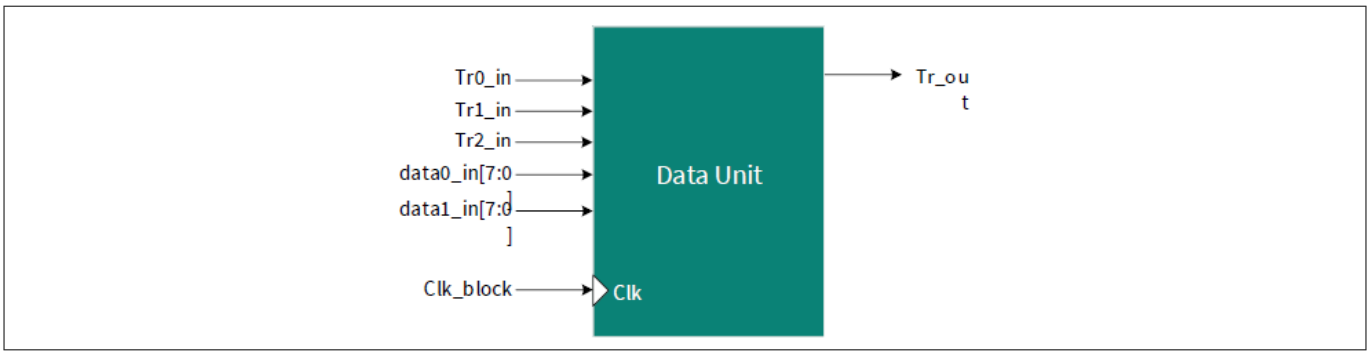


Figure 70 Data unit

- The trigger signals for the Data Unit (DU) are selected using the `DU_TRx_SEL[3:0]` bitfield in the `SMARTIO_PRTx_DU_SEL` register
- The `DU_DATAx_SEL[1:0]` bits in the `SMARTIO_PRTx_DU_SEL` register select the 8-bit input data source
- The size of the DU (the number of bits used by the data path) is defined by the `DU_SIZE[2:0]` bits in the `SMARTIO_PRTx_DU_CTL` register
- The DU generates a single output trigger signal (`tr_out`)
- The internal state (`du_data[7:0]`) is stored in flip-flops and requires the clock signal `clk_block` for operation

Table 85 Register description

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]	Data unit opcode specifies the data unit operation:1: INCR2: DECR3: INCR_WRAP4: DECR_WRAP5: INCR_DECR6: INCR_DECR_WRAP7: ROR8: SHR9: AND_OR10: SHR_MAJ311: SHR_EQL Otherwise: Undefined.
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'.2: 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

(table continues...)

Table 85 (continued) Register description

Register[BIT_POS]	Bit name	Description
SMARTIO_PRTx_DU_SEL[25:24]	DU_DATA0_SEL[1:0]	Data unit input data “data0_in” source selection:0: 0x001: 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.

21.6 Registers summary

Table 86 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.
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.

(table continues...)

Table 86 (continued) Registers summary

Registers	Name	Description
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_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.
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.

(table continues...)

Table 86 (continued) Registers summary

Registers	Name	Description
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

22 TCPWM

The timer/counter/pulse-width modulator (TCPWM) block in CYW20829 uses a 16-bit or 32-bit counter, which can be configured as a timer/counter/pulse-width modulator (PWM), or quadrature decoder.

The block can be used to:

- Measure the period and pulse width of an input signal (timer)
- Find the number of times a particular event occurs (counter)
- Generate PWM signals or decode quadrature signals

This chapter explains the features, implementation, and operational modes of the TCPWM block. The TCPWM supports operation in Active, Low-Power Active, Sleep, and Low-Power Sleep power modes.

22.1 Features

The TCPWM block supports the following operational modes:

- Timer-counter with compare
- Timer-counter with capture
- Quadrature decoding
- Pulse-width modulation (PWM)
- Pseudo-random PWM
- PWM with dead time
- Up, Down, and Up/Down counting modes
- Clock prescaling (division by 1, 2, 4, …, 64, and 128)
- 16-bit or 32-bit counter widths
- Double buffering of compare, capture, and period values
- Underflow, overflow, and capture/compare output signals
- Complementary output for PWMs

It also supports interrupts on:

- **Terminal count**: Depends on the mode and typically occurs on overflow or underflow
- **Capture/compare**: Triggered when the current count is captured in the capture register or when the counter value matches the value in the compare register

22.2 Architecture

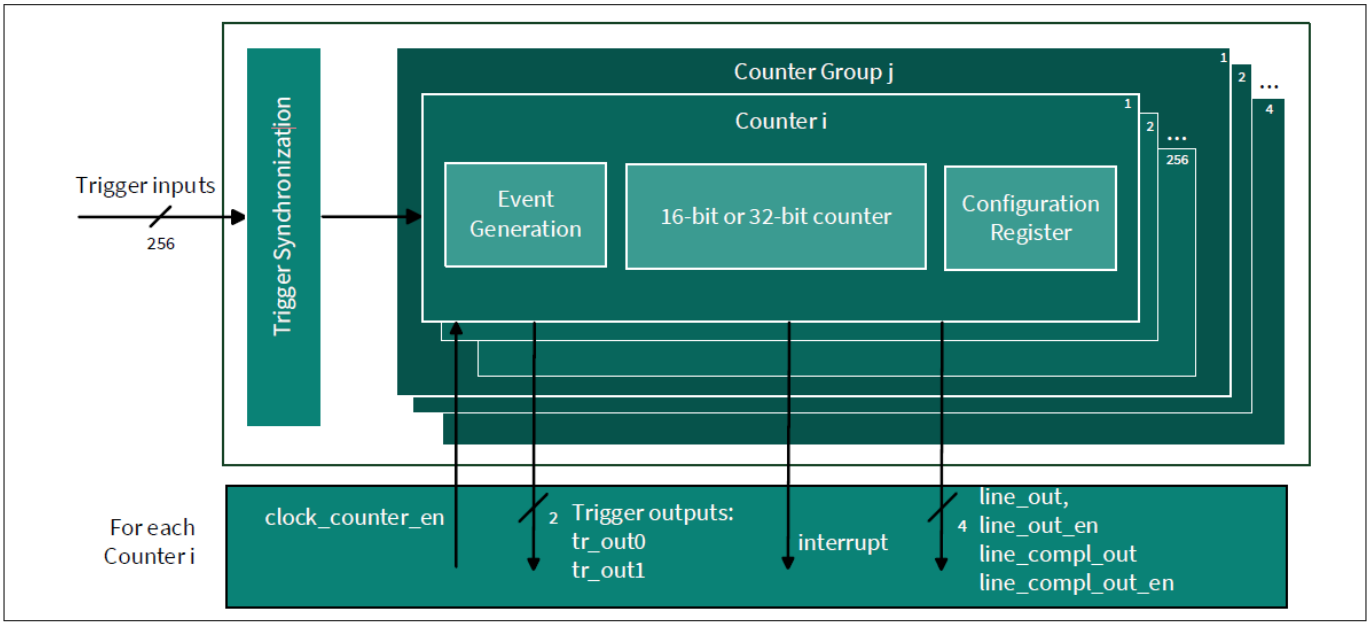


Figure 71 TCPWM architecture block diagram

The TCPWM block can contain up to 32 counters, with each counter being either 16-bit or 32-bit wide. The three primary registers used to control the counters are:

- **TCPWM_CNT_CC**: Captures the counter value in CAPTURE mode. In all other modes, this register stores a value that is compared to the counter value
- **TCPWM_CNT_COUNTER**: Holds the current counter value
- **TCPWM_CNT_PERIOD**: Sets the upper limit of the counter. When the counter counts for 'n' cycles, this field should be set to 'n-1'

The number of 16-bit and 32-bit counters varies depending on the specific device. For more details, see the device datasheet. In this context, a TCPWM refers to the entire block and all the counters within it, whereas a counter refers to an individual counter inside the TCPWM block. Within a TCPWM block, all counters share the same width (16-bit or 32-bit).

The TCPWM block includes the following interfaces:

- **I/O signal interface**:
  - Consists of input triggers such as reload, start, stop, count, and capture
  - Output signals such as PWM, PWM_n, overflow (OV), underflow (UN), and capture/compare (CC)
  - Input triggers are used to initiate specific events within the counter, such as a reload trigger generating a reload event
  - Output signals are generated by internal events (such as underflow, overflow, and capture/compare) and can be connected to other peripherals to trigger events
- **Interrupts**: Provides interrupt request signals for each counter, based on terminal count (TC) or capture/compare (CC) conditions

22.2.1 Enabling and disabling counters in a TCPWM block

To enable or disable the TCPWM block, use the `TCPWM_CTRL_SET` and `TCPWM_CTRL_CLR` register bits, respectively. These registers should be used to prevent race conditions during read-modify-write operations on the `TCPWM_CTRL` register, which manages the enable/disable fields of the counters.

***Note**: The counter must be fully configured before it is enabled. Disabling the counter retains the values stored in its configuration registers.*

22.2.2 Clocking

The TCPWM counter can have its own clock. The only source for this clock comes from the configurable peripheral clock dividers generated by the clocking system. For additional details, see Clocking system.

To select a clock divider for a specific counter within a TCPWM, use the `CLOCK_CTL` register in the peripheral register space. In this section, the clock driving the counter is referred to as `clk_counter`. Event generation is performed based on `clk_counter`.

Another clock, the system clock (`clk_sys`), is used to define the pulse width of the output triggers. The system clock (`clk_sys`) is synchronous with the peripheral clock (`clk_peri`) but can be divided using the `CLOCK_CTL` register.

22.2.2.1 Clock pre-scaling

The `clk_counter` can be further divided within each counter using specific divider values of 1, 2, 4, 8, 16, 32, 64, or 128. This division process is referred to as 'pre-scaling'. The pre-scaling is configured using the `PS_DIV[2:0]` field in the `PS` register.

22.2.2.2 Count event

The counter functionality operates on an active-count pre-scaled clock, which is controlled by a count-event signal. For example, the counter increments or decrements by '1' during each counter-clock cycle when a count event is detected.

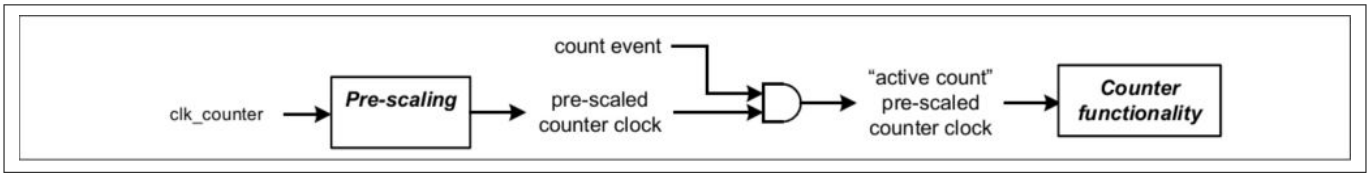


Figure 72 Counter clock generation

All status or output changes occur only on the active-count pre-scaled counter clock. In other words, if the count event is inactive, the counter, status, interrupt, and all outputs will remain unchanged.

For example, in pass-through mode, if the count event goes low while the counter reaches '0' in down-count mode, the terminal count (`tc`) event and the underflow event will be generated on the next pre-scaled counter clock after the count event goes high.

The only exception to this is the immediate kill mode. In this mode, the kill input suppresses the PWM output immediately, regardless of the active-count pre-scaled counter clock.

***Note**: This functionality is not supported in all modes.*

22.2.3 Trigger inputs

Each TCPWM block has 14 `Trigger_In` signals, which are sourced from other on-chip resources, such as other TCPWMs, SCBs, or DMA. The `Trigger_In` signals are shared across all counters within a single TCPWM block. The `Trigger_Mux` registers are used to configure which signals are routed to the `Trigger_In` for each TCPWM block. In addition to the 14 `Trigger_In` signals, two constant trigger inputs, `0` and `1`, are always available.

Two types of trigger signals are synchronized and can be used by the counters to generate events:

- **General-purpose triggers**:
  - These triggers can be used by all counters
  - They are generated by different system blocks and distributed through the trigger infrastructure (for example, peripheral trigger multiplexers). For more details, see Trigger MUX
- **One-to-one triggers**:
  - These are dedicated to individual counters and are only connected to their associated counters
  - They are used for direct trigger connections from trigger sources, such as HPPASS level triggers, to specific TCPWM counters

To configure the routing of signals to the `Trigger_In` for a TCPWM block, use the `TR_IN_SEL0` and `TR_IN_SEL1` trigger mux registers. See Table 87 for a complete list of possible multiplexer settings for selecting input trigger events for a TCPWM block. For each event, two constant trigger inputs are always available: input trigger `0` is constant `0` and input trigger `1` is constant `1`.

For each counter, the trigger input source is selected using the `TCPWM_CNT_TR_CTRL0` register. Each counter can select any of the 16 trigger signals as a source for the following events:

- Reload
- Start
- Stop/Kill
- Count
- Capture/Swap

Table 87 Possible multiplexer settings for selecting input trigger events in a TCPWM block

Input Trigger Selection register	Bit field	Bits	Description
TR_IN_SEL0	CAPTURE0_SEL	7:0	Selects one of the up to 16 input triggers as a Capture0 trigger. In the PWM,PWM_DT, and PWM_PR modes this trigger is used to switch the values if the compare and period registers with their buffer counterparts.
	COUNT_SEL	15:8	Selects one of the 16 input triggers as a count trigger. In QUAD mode, this is the first phase (phi A).
	RELOAD_SEL	23:16	Selects one of the 16 input triggers as a reload trigger. In QUAD mode, this is the index or revolution pulse.

(table continues...)

Table 87 (continued) Possible multiplexer settings for selecting input trigger events in a TCPWM block

Input Trigger Selection register	Bit field	Bits	Description
	STOP_SEL	31:24	Selects one of the 16 input triggers as a stop trigger. In PWM, PWM_DT, and PWM_PR modes, this is the kill trigger.
TR_IN_SEL1	START_SEL	7:0	Selects one of the 16 input triggers as a start trigger. In QUAD mode, this is the second phase (phi B).
	CAPTURE1_SEL	15:8	Selects one of the up to 16 input triggers as a Capture1 trigger.

The following sections describe each TCPWM mode and the function of each input event in detail.

Typical operation uses the reload event 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 resume the counter while keeping its value unchanged from when it was stopped.

If stop, reload, and start events occur simultaneously, the following priority relationship applies:

- A stop event has the highest priority
- A reload event has higher priority than a start event
- As a result, if a reload or start event coincides with a stop event, the reload or start event will have no effect

Before reaching the counter, each `Trigger_IN` can pass through one of the following options:

- Positive-edge detector
- Negative-edge detector
- Both-edge detector
- Direct path (straight through to the counter)

This behavior is controlled using the `TR_IN_EDGE_SEL` register.

When multiple events are detected, they are interpreted as follows:

- In rising-edge and falling-edge modes, multiple events are effectively reduced to a single event, which may result in lost events
- In rising/falling-edge mode:
  - An even number of events are not detected
  - An odd number of events are reduced to a single event
  - This behavior ensures that the alternating pattern of rising and falling edges is preserved, as the mode is typically used for capture events to measure pulse width

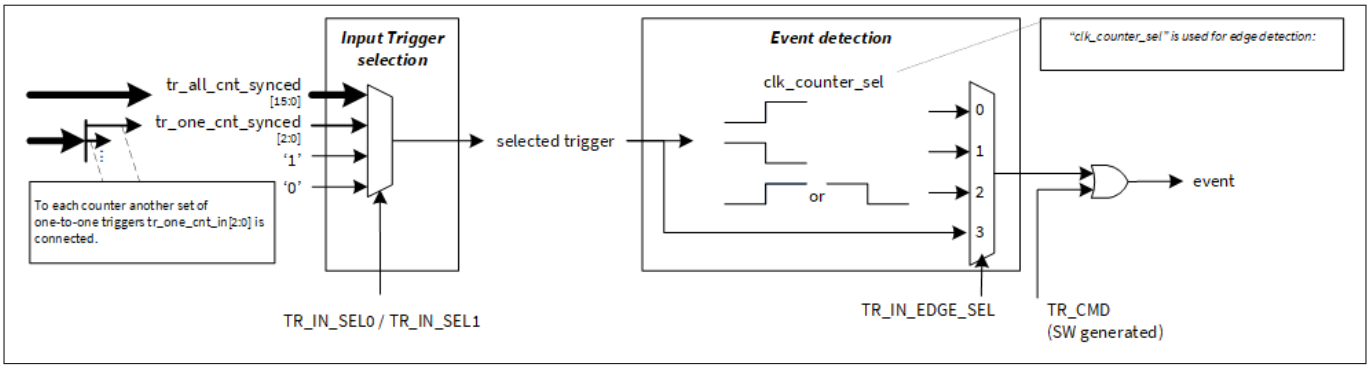


Figure 73 TCPWM input events

According to Table 87, a dedicated input trigger signal for trigger event generation can be defined using the `TR_IN_SEL0` and `TR_IN_SEL1` registers. The selection can be made from the following options:

- Two constant signals (`constant '0'` and `constant '1'`)
- Specific One-to-one trigger input signals
- General-purpose input trigger signals

Figure 73 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 is device-specific. For more details, see the device datasheet.

Table 88 Configuration of the multiplexer for input trigger event generation

Input trigger	Input trigger	Input trigger source
0	Constant ‘0’	Constant '0'
1	Constant ‘1’	Constant '1'
2	PASS (programmable analog subsystem),through 1:1 trigger mux #0	See Trigger MUX.
3	HSIOM	Refer to the 'Alternate pin function' section in the device datasheet.
4	HSIOM	Refer to the 'Alternate pin function' section in the device datasheet.
5	tr_all_cnt_in[0]	See Trigger MUX.
20	tr_all_cnt_in[15]	See Trigger MUX.

***Note**: Input triggers can be generated by various sources. While the general-purpose trigger inputs (`tr_all_cnt_in[0]` to `tr_all_cnt_in[15]`) are sourced solely from the trigger multiplexer block (see Trigger MUX), the one-to-one input triggers can also be generated by external GPIO input pins.*

All trigger inputs are synchronized to `PERI_CLK`. When multiple events occur within the same `counter_clock` period, one or more events may be missed. This is more likely in scenarios involving high-frequency events (with frequencies close to the `counter_clock` frequency) or when the timer configuration uses a pre-scaled (divided) `counter_clock`.

Figure 74 demonstrates the timing and the process by which input triggers are detected by the counter.

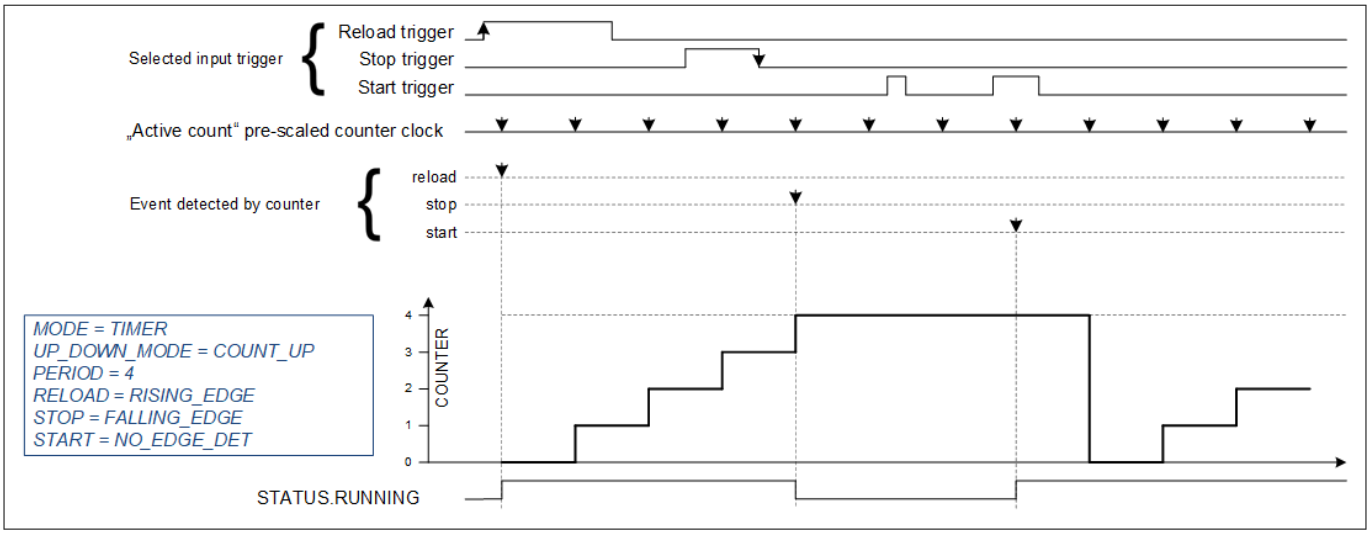


Figure 74 Input trigger detection by active-count pre-scaled counter clock

***Note**: Events marked with arrows in Figure 74 represent those that have been detected by the counter.*

Typically, the count event is a constant 1, and pre-scaling is turned off. In this case, the active-count pre-scaled counter clock is identical to the counter clock. In other cases, edge detection may identify multiple events (on the counter clock) before the next active-count pre-scaled counter clock is reached, where the detected event is then used.

Multiple detected events are handled as follows:

- In rising-edge and falling-edge modes, multiple events are effectively reduced to a single event. As a result, some events may be lost
- **In rising/falling-edge mode**:
  - An even number of events is not detected
  - An odd number of events is reduced to a single event
  - This behavior helps maintain the alternating pattern of rising, falling, rising, and falling edges, as rising/falling-edge mode is typically used for capture events to measure pulse width

For pass-through events:

- These events are not remembered by CLK_PERI. They only take effect if they persist and can be detected by the counter-operation clock
- If the pulse width of a pass-through event is shorter than one counter-operation clock cycle, the event may be missed
- Pass-through detection may cause an event to remain active for multiple counter clocks, which can result in undesirable behavior in the counter and its associated trigger outputs
- Pass-through event detection is typically recommended only for stop and count event types in most functional modes. It can also be used for switch events in PWM/PWM_DT mode if it selects the constant high as the source

22.2.4 Synchronization of multiple counters

The previous sections described hardware-based event generation. In addition to this, software-based event generation is also supported. Events such as reload, start, stop, Capture0, and Capture1 can be generated by writing to the PERI0_TR_CMD registers. These registers are counter-specific and enable software-based event generation for a single counter.

Synchronized software-based event generation, such as starting multiple counters simultaneously, can be achieved by selecting the same trigger signal for all desired counters using the TR_IN_SEL0 and TR_IN_SEL1 registers. A trigger can then be generated through the PERI0_TR_CMD register in the PERI block.

Figure 75 provides an example of how multiple counters can be started synchronously using the TR_CMD functionality of the PERI block.

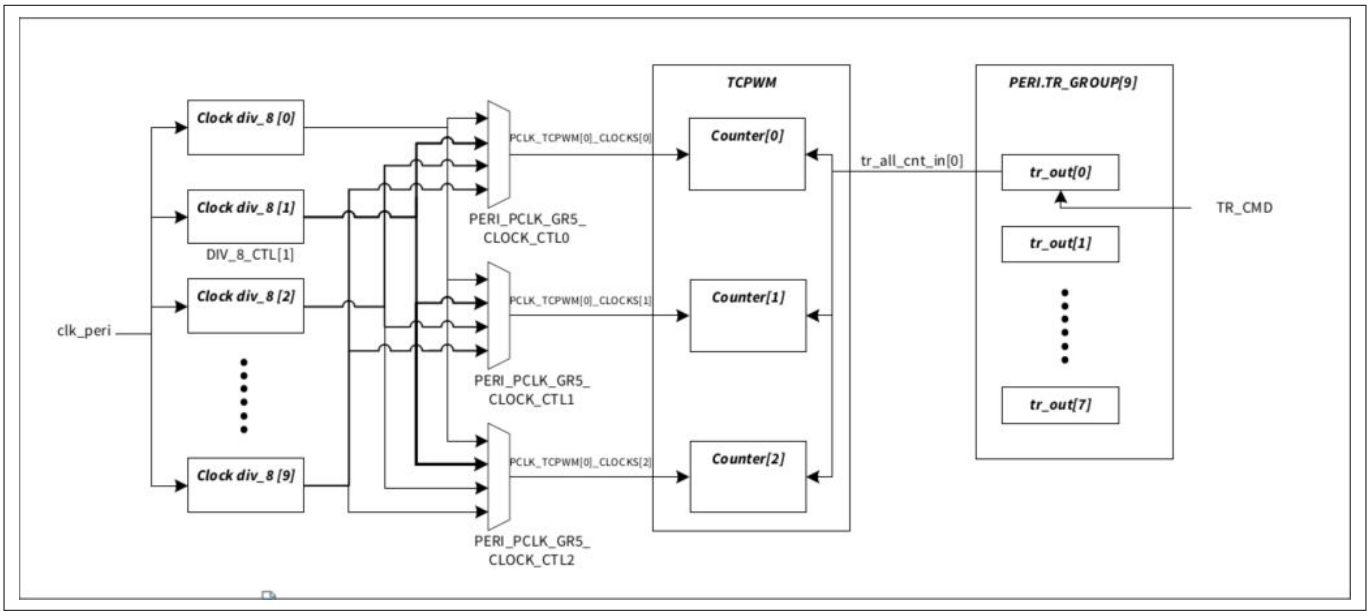


Figure 75 PERI TR_CMD synchronously starts counters

The following example explains the necessary 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 the clock-divider enable signal for the TCPWM. This ensures that all three counters share the same clock-divider enable signal, meaning their clocks are synchronous
- Enable the three counters one by one. The counters will be enabled but will not start running until a start or reload event is detected
- Configure the start event for all three counters to select tr_all_cnt_in[0]. This ensures that all three counters start running synchronously when tr_all_cnt_in[0] is asserted
- Use TR_CMD to generate a trigger pulse on TR_GROUP[9].TR_OUT[0]. This will make all three counters run synchronously

***Note**: All registers mentioned here belong to the PERI block (see Trigger MUX).*

A software-based event is set when the respective bit in TR_CMD is written to 1. The event is then cleared automatically by hardware on the next active-count pre-scaled counter clock. For some events in specific modes, it will be cleared on the next active-count pre-scaled counter clocks that have a terminal count (tc) event.

The event-detection setting (TR_IN_EDGE_SEL) does not affect software-based events.

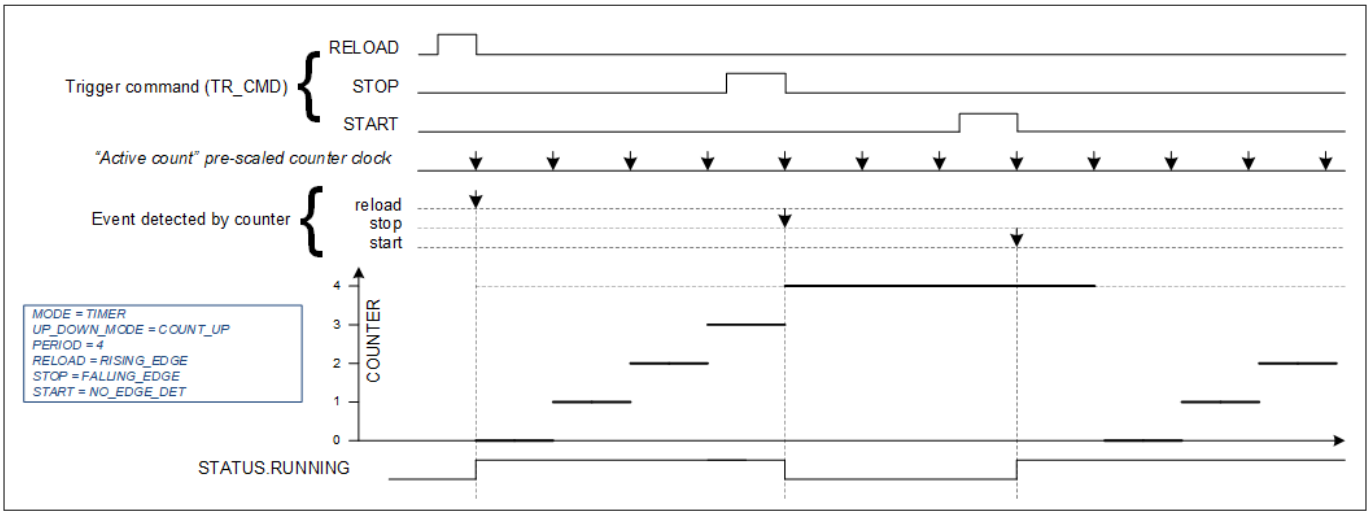


Figure 76 Software trigger command detection by active-count pre-scaled counter clock

22.2.5 Trigger outputs

Each counter can generate three trigger output events. These trigger output events can be routed through the trigger multiplexer (`Trigger Mux`) to other peripherals on the device.

The three trigger outputs are:

- Overflow (OV)
- Underflow (UN)
- Compare/Capture (CC)

The TCPWM block provides a dedicated interrupt output for each counter. This interrupt can be generated for a terminal count (`TC`) or `CC` event. A terminal count (`TC`) is the logical OR of the Overflow (`OV`) and Underflow (`UN`) events.

The generated triggers have two primary uses:

- **Initiating a DW/DMA data transfer**: For example, in PWM mode with an up-counting timer, the overflow event can be used to transfer new period and compare values from memory to the counter's `PERIOD_BUFF` and `CC0_BUFF` registers
- **Reconstructing a PWM signal in a programmable digital component**:
  - As explained in pulse-width modulation (PWM) mode, the PWM line output signal is derived from the `cc_match` event, underflow, and overflow internal events
  - By exposing these internal events as output triggers, other components can reconstruct and potentially modify the PWM signal
  - Note that reconstruction is subject to specific frequency restrictions

22.2.6 Internal events

22.2.6.1 Underflow event

An underflow event occurs during down-counting when the COUNTER reaches zero and is subsequently changed to a different value. In some specific modes, a reload operation may also generate an underflow event. Table 89 summarizes the conditions for underflow event generation in each functional mode.

Table 89 Underflow generation

Mode	Description
Timer	Counter is decrementing and changes from a state in which COUNTER equals zero. Reload event in DOWN, UPDN1, and UPDN2 modes.
Capture	Counter is decrementing and changes from a state in which COUNTER equals zero. Reload event in DOWN, UPDN1, and UPDN2 modes.
PWM	Counter is decrementing and changes from a state in which COUNTER equals zero. Reload event in DOWN, UPDN1, and UPDN2 modes.
PWM_DT	Counter is decrementing and changes from a state in which COUNTER equals zero. Reload event in DOWN, UPDN1, and UPDN2 modes.

22.2.6.2 Overflow event

An overflow event occurs during up-counting when the COUNTER value reaches the PERIOD and is subsequently changed to a different value. In some specific modes, a reload operation may also generate an overflow event. Table 90 summarizes the conditions for overflow event generation in each functional mode.

Table 90 Overflow generation

Mode	Description
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.
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.

22.2.6.3 TC event

A terminal count (`tc`) event occurs as the logical OR of the underflow and overflow events. An exception to this is that a reload event generates either an underflow or overflow event, but not a `tc` event. In quadrature mode, an index generates a `tc` event. Table 91 summarizes the conditions for `tc` event generation in each functional mode.

Table 91 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
PWM	Overflow	Underflow	Underflow	Logic OR of overflow and underflow

(table continues...)

Table 91 (continued) TC generation

Mode	UP	DOWN	UPDN1	UPDN2
PWM_DT	Overflow	Underflow	Underflow	Logic OR of overflow and underflow

22.2.6.4 cc0_match (cc1_match) event

A `cc0_match` event occurs when the `COUNTER` equals `CC0`. This event is generated either when the `COUNTER` is about to change to the value stored in `CC0`, or when the `COUNTER` already equals `CC0` and is about to change to a different value.

A special case exists for 0 or 100 percent duty cycle generation in PWM mode. For additional details, see Pulse-width modulation (PWM) mode. In other specific operational modes, the event is used to indicate that the `CC0` or `CC0_BUFF` registers have been updated.

The `cc1_match` event is determined by the relationship between the `COUNTER` and `CC1`. Apart from this, the behavior of `cc1_match` is the same as that of `cc0_match`.

Table 92 and Table 93 summarize the conditions for `cc0_match` and `cc1_match` event generation in each functional mode.

Table 92 cc0_match generation

Mode	UP	DOWN	UPDN1	UPDN2
Timer	Counter changes from a state in which COUNTER equals CC0.			
Capture	Capture0 event			
PWM	Counter changes to a state in which COUNTER equals CC0.		COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC0. If a second compare function is present in a counter group, 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		COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC0. If a second compare function is present in a counter group,CC0_MATCH_DOWN_EN/ CC0_MATCH_UP_EN will enable/disable cc0_match generation.	

Table 93 cc1_match generation

Mode	UP	DOWN	UPDN1	UPDN2
Timer	Counter changes from a state in which COUNTER equals CC1.			
Capture	Capture1 event			

(table continues...)

Table 93 (continued) cc1_match generation

Mode	UP	DOWN	UPDN1	UPDN2
PWM	Counter changes to a state in which COUNTER equals CC1.		COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC0. If a second compare function is present in a counter group,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 CC1.		COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC0. If a second compare function is present in a counter group,CC0_MATCH_DOWN_EN/ CC0_MATCH_UP_EN will enable/disable cc0_match generation.	

22.2.7 Interrupts

The TCPWM block provides a dedicated interrupt output for each counter. Interrupts are specific to the counter mode and can be generated for a Terminal Count (TC) or Compare/Capture0/1 (CC0/1) event. A TC event is the logical OR of the Overflow (OV) and Underflow (UN) events.

Four registers are used to manage interrupts within this block, as outlined in Table 94.

Table 94 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 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 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 94 (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.

22.2.8 Debug mode

The TCPWM counters support debugging through the `CTRL.DBG_FREEZE_EN` and `CTRL.DBG_SUS_EN` bits. These bits allow per-counter configuration, determining whether the counter operation continues or pauses when the system enters debug mode (for example, after hitting a breakpoint). This feature is particularly useful when using a TCPWM counter as an OS timer.

The pause operation is implemented by gating the `clk_counter` when the system enters debug mode and `DBG_FREEZE_EN` is set to `1`.

Table 95 outlines the behavior of the debug mode based on the states of the CTRL.DBG_FREEZE_EN and CTRL.DBG_SUS_EN bits.

Table 95 Debug mode

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 control through CTRL.KILL_LINE_POLARITY.	0	1
Counter Operation Pauses immediately and enter passive state. Passive state is control through CTRL.KILL_LINE_POLARITY. STATUS.RUNNING bit set to 0.	1	1

22.2.9 PWM outputs

The PWM can output a left-, right-, center-, or asymmetrically-aligned PWM signal. A pulse-width modulated signal is generated by incrementing or decrementing a counter between 0 and PERIOD and comparing the counter value (COUNTER) with the CC value. When COUNTER equals CC, a cc_match event is generated.

To produce a pulse-width modulated signal, combine the cc_match, overflow, and underflow events. The PWM generates two output signals, pwm and pwm_n, which are derived from these events. Each event (underflow, overflow, and cc_match) can be independently set to INVERT, SET, or CLEAR the pwm_dt_input.

The PWM behavior is governed by the values in the PERIOD and CC registers. The software can update the PERIOD_BUFF and CC_BUFF registers without impacting active PWM behavior. This capability highlights the importance of double-buffering these registers.

Table 96 Conditions configuring output line for OV, UN, and CC0/1

Field	Bit	Value	Event	Description
CC0_MATCH_MODEDefault 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_MODEDefault 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	
		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	

(table continues...)

Table 96 (continued) Conditions configuring output line for OV, UN, and CC0/1

Field	Bit	Value	Event	Description
		3	No change	
CC1_MATCH_MODED efault 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	

Each counter has two outputs: `pwm` (`line_out`) and `pwm_n` (`line_compl_out`), which is the complementary output of `pwm`. The `OV`, `UN`, and `CC` conditions are used to control the behavior of `pwm` and `pwm_n`. This configuration is done using the `TCPWM_CNT_TR_CTRL2` register.

Table 97 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

22.2.10 Power modes

22.3 Operation modes

The counter block can operate in four distinct modes, as outlined in Table 98. The `MODE[26:24]` field of the counter control register (`CTRL`) is used to configure the counter to a specific operational mode.

Table 98 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.

(table continues...)

Table 98 (continued) Operational mode configuration

mode	MODE field [26:24]	Description
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.
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 in up, down, or up/down modes by setting the `UP_DOWN_MODE[17:16]` field in the `CTRL` register, as detailed in Table 99.

Table 99 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.

22.3.1 Timer mode

The timer mode is used to measure the duration of an event or the time difference between two events. The timer functionality increments or decrements a counter between `0` and the value stored in the PERIOD register.

While the counter is running, the count value stored in the COUNTER register is compared with the values in the compare/capture registers (CC0 and CC1). When the COUNTER equals CC0, a cc0_match event is generated. Similarly, when the COUNTER equals CC1, a cc1_match event is generated.

Timer functionality is typically used for one of the following purposes:

- **Measuring 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 as the count event

Table 100 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 count direction is set to 'up' • **COUNT_DOWN**: The counter is set to PERIOD and count direction is set to 'down' • **COUNT_UPDN1/2**: The counter is set to '1' and 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 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. Can be used when the counter is running or not running.
Stop	Stops the counter
Count	Count event increments/decrements the counter
Capture0	This event is used as external up/down counting event. External direction control mode is enabled via QUAD_ENCODING_MODE(1:0).
Capture1	Not used

The incrementing and decrementing of the counter are controlled by the count event and the counter clock (clk_counter). Typical operation uses a constant 1 as the count event and a clk_counter without pre-scaling.

More advanced operations are also possible. For example, the count event configuration can be set to count the rising edges of a synchronized input trigger.

Table 101 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, 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_CC0, no input event is required). When parameter GRP_CC1_PRESENT = 1, CC1 and CC1_BUFF are exchanged on a cc1_match event when CTRL.SWAP_ENABLE = “1” (when specified by CTRL.AUTO_RELOAD_CC1). 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). When parameter GRP_CC1_PRESENT = 1, CC1_BUFF are transferred to CC1 on a cc1_match event when CTRL.SWAP_ENABLE = “0” (when specified by CTRL.AUTO_RELOAD_CC1).
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 102 provides a detailed overview of the trigger outputs and the corresponding conditions under which they are activated.

Table 102 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. Only available when parameter GRP_CC1_PRESENT = 1.
Underflow (UN)	Counter is decrementing and changes from a state in which COUNTER equals 0. Reload event generates underflow in COUNT_DOWN, COUNT_UPDN1 or COUNT_UPDN2 mode.

(table continues...)

Table 102 (continued) Timer mode trigger outputs

Trigger outputs	Description
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 103 Timer mode PWM outputs

PWM outputs	Description
LINE_OUT	Not used
LINE_COMPL_OUT	Not used

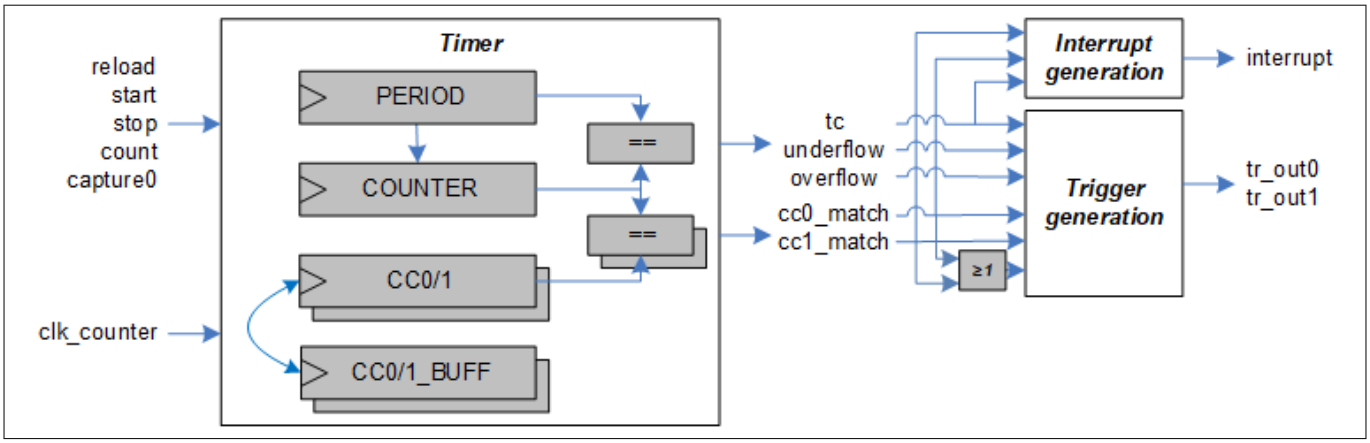


Figure 77 Timer functionality

Note:
- *The triggers `tr_out0` and `tr_out1` are generated based on internal events such as `cc0_match`, `cc1_match`, the logical OR combination of `cc0_match` and `cc1_match`, underflow, overflow, and `tc`. The selection is configured using the `TR_OUT_SEL` register*
- *The timer functionality uses only the PERIOD register and not the `PERIOD_BUFF` register*
- *Writing to the COUNTER register while the counter is running is not recommended*
- *In `COUNT_UPDN2` mode, the software should only modify the PERIOD register during a `tc` condition when the COUNTER is `0` and not when the COUNTER equals PERIOD*

Figure 78 shows an example of a timer in up-counting mode. The counter is initialized to `0` and started using a software-based reload event.

***Note**:*

- *When the counter transitions from a state where COUNTER equals `4`, an overflow and `tc` event are generated*
- *When the counter transitions from a state where COUNTER equals `2`, a `cc0_match` event is generated*
- *The PERIOD register is set to `4`, resulting in an effective repeating counter pattern of `4 + 1 = 5` `clk_counter` periods. The `cc0` register is set to `2`, defining the condition for a `cc0_match` event*

In the following scenarios, a constant count event of `1` and a `clk_counter` without pre-scaling are used:

- If the count event is `0` and a reload event is triggered, the reload will only be registered on the first clock edge when the count event becomes `1`
  - In this case, the first clock edge when the count event is `1` will not be used for counting but will instead be used for the reload operation

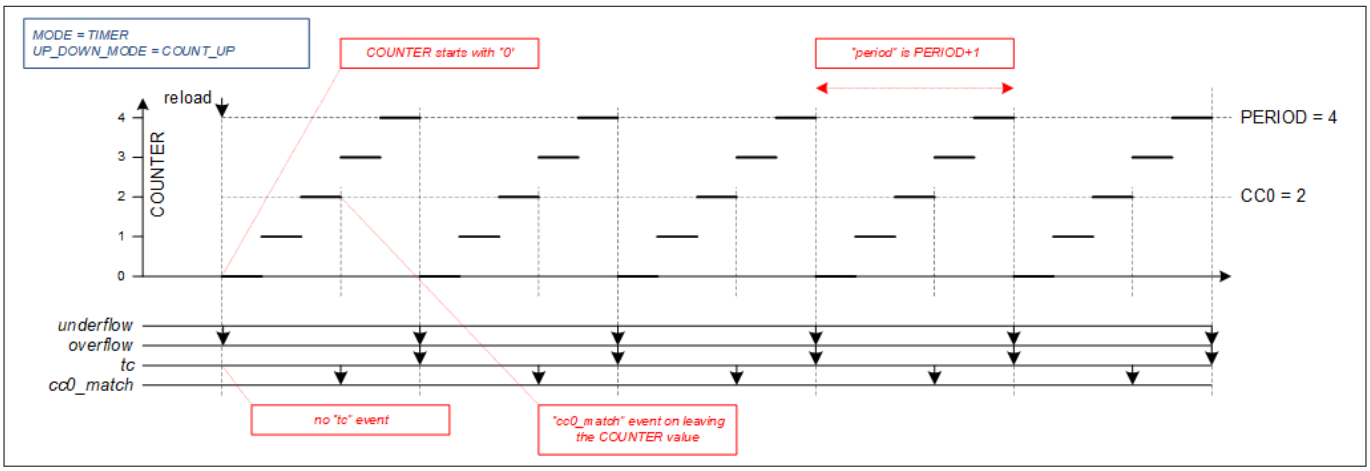


Figure 78 Timer in up-counting mode

Figure 79 a timer in down-counting mode. The counter is initialized to PERIOD and started using a software-based reload event.

***Note**:*

- *When the counter transitions from a state where COUNTER equals `0`, an underflow and `tc` event are generated*
- *When the counter transitions from a state where COUNTER equals `2`, a `cc0_match` event is generated*
- *The PERIOD is set to `4`, resulting in an effective repeating counter pattern of `4 + 1 = 5` counter-clock periods*

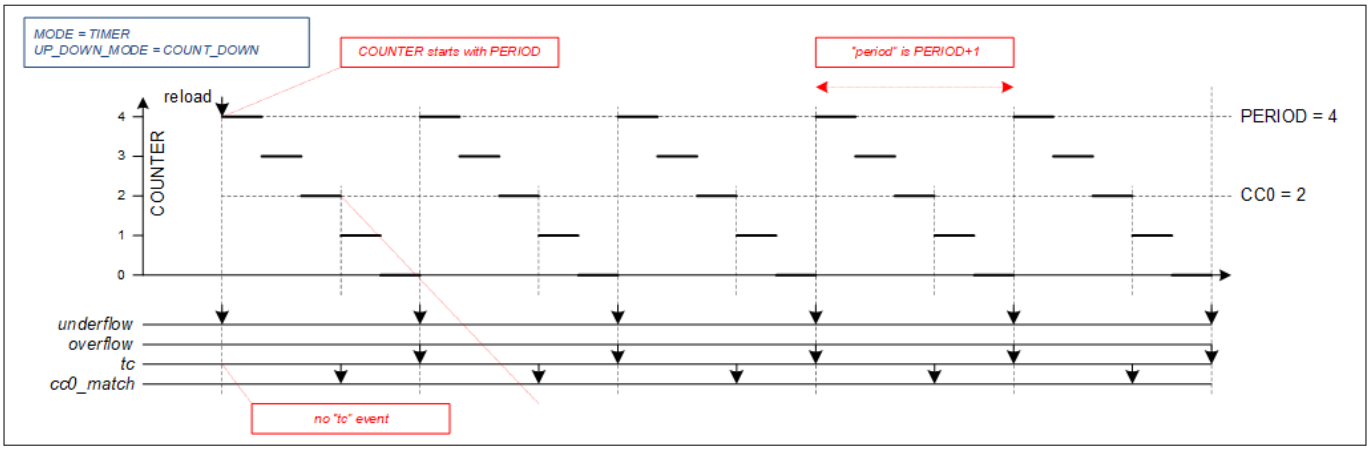


Figure 79 Timer in down-counting mode

Figure 80 shows a timer in up/down-counting mode 1. The counter is initialized to 1 and started using a software-based reload event.

Note:

- *When the counter transitions from a state where COUNTER equals 4, an overflow event is generated*
- *When the counter transitions from a state where COUNTER equals 0, an underflow and tc event are generated*
- *When the counter transitions from a state where COUNTER equals 2, a cc0_match event is generated*
- *The PERIOD is set to 4, resulting in an effective repeating counter pattern of 2 * 4 = 8 counter-clock periods*

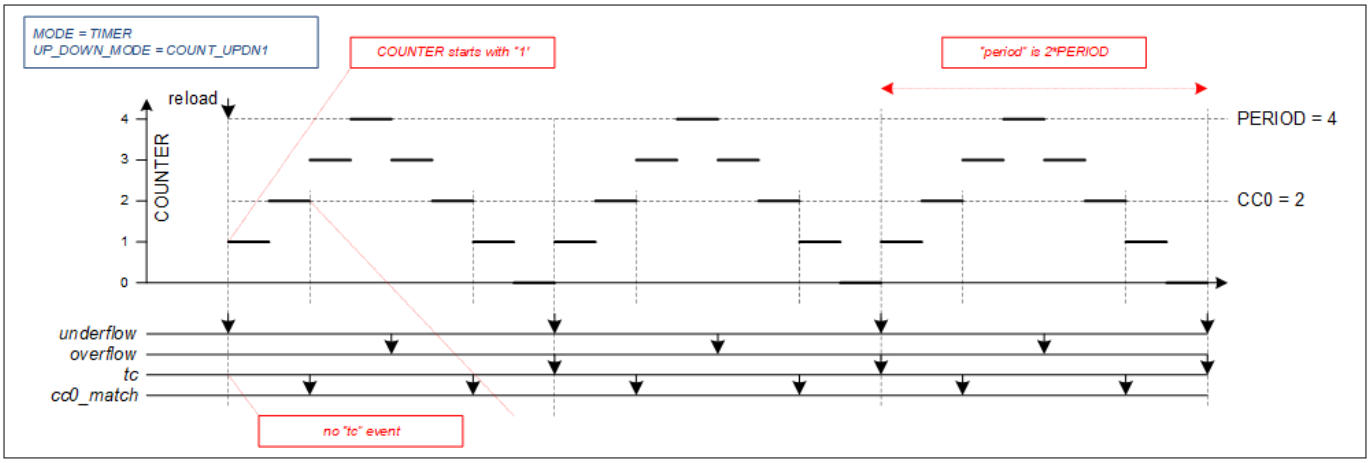


Figure 80 Timer in up/down-counting mode 1

COUNT_UPDN2 has the same behavior as COUNT_UPDN1, except that the tc event in COUNT_UPDN2 generates an overflow event.

Figure 81 shows a timer in one-shot operation mode.

Note:

- *The counter stops on a tc event*
- *In COUNT_UPDN2, the counter stops on a tc event when an underflow occurs*
- *After a tc event, the COUNTER will remain at the next value. In the figures that follow, the counter stays at 0 after it stops running*

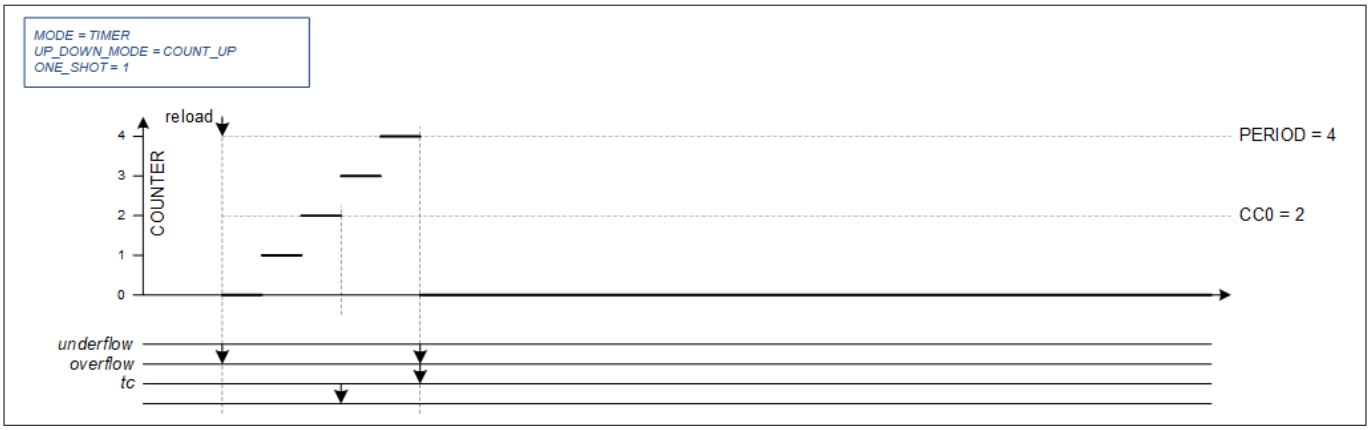


Figure 81 Timer in one-shot mode

Figure 82 shows clock pre-scaling. The supported pre-scaler values are `/1`, `/2`, `/4`, `/8`, `/16`, `/32`, `/64`, and `/128`. The counter is incremented only on every other counter cycle.

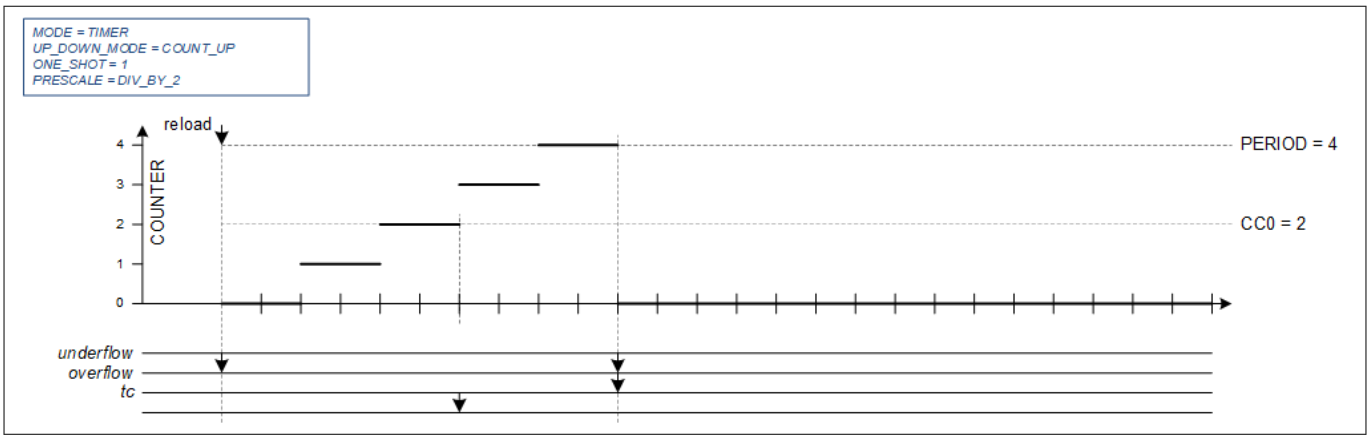


Figure 82 Timer clock pre-scaling

Figure 83 shows a timer that utilizes both the `CC0` and `CC0_BUFF` registers. The values in `CC0` and `CC0_BUFF` are exchanged during a `cc0_match` event.

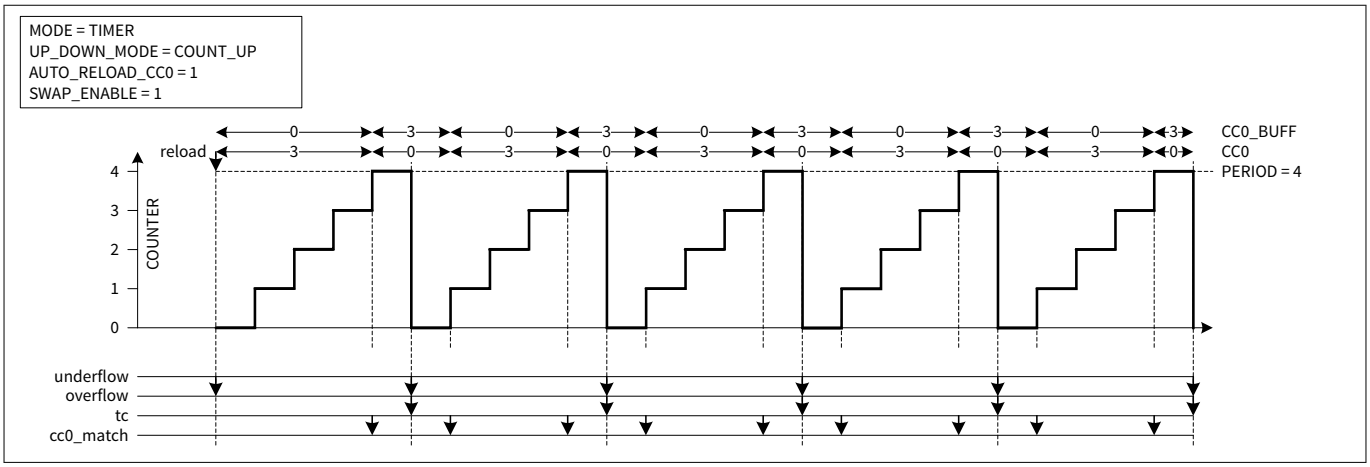


Figure 83 CC0 and CC0_BUFF auto-switch at cc0_match when SWAP_ENABLE = '1'

If `CTRL.SWAP_ENABLE` is set to `0`, the value in the `CC0_BUFF` register is automatically transferred to the `CC0` register by hardware when a `cc0_match` event is asserted. No input event is required to trigger this transfer.

Figure 84 shows a timer that uses both the `CC0` and `CC0_BUFF` registers. The value in `CC0_BUFF` is transferred to `CC0` during a `cc0_match` event.

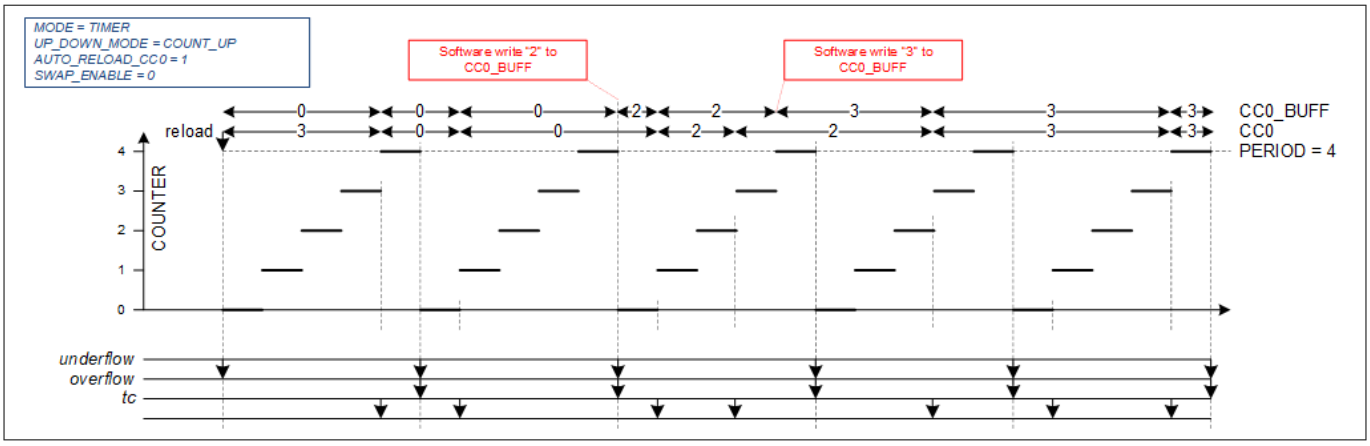


Figure 84 CC0_BUFF auto switch to CC0 at cc0_match with SWAP_ENABLE = '0'

The above description also applies to CC1 and CC1_BUFF when the parameter GRP_CC1_PRESENT is set to 1.

Timer mode supports the following input events: reload, start, stop, and count.

Figure 85 shows a counter being initialized and started (reload event), stopped (stop event), and then restarted or continued (start event).

***Note**: The counter does not change its value when it is not running (STATUS.RUNNING is false.*

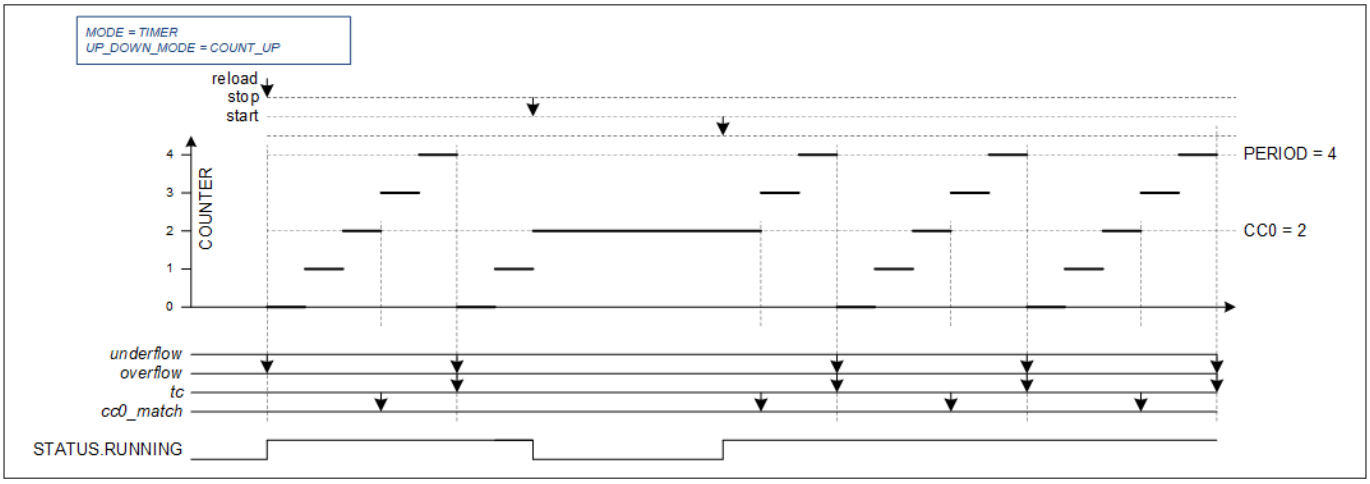


Figure 85 Counter Start/Stopped/Continued

A reload event can occur while the counter is running. The following clarifies the behavior when a reload event coincides with a terminal count (tc) event, a cc0_match event, or a stop event:

- A stop event takes precedence over a reload event
- A reload event takes precedence over the running counter functionality (for underflow, overflow, and tc event generation)

A reload event determines:

- Underflow event generation
- Overflow event generation
- Terminal count (tc) event generation (always 0)

The running counter functionality determines:

- Underflow event generation
- Overflow event generation

- Terminal count (tc) event generation
- cc0_match event generation
- CC0/CC0_BUFF exchange

Figure 86 shows a scenario where the count event becomes low while the counter is running, and a rising-edge-detection stop event asserts when the count event is low (in this example, the count event is used in pass-through mode).

- The COUNTER will stop running and retain its value when the count event is low
- The stop event will not set the running status to low until the count event becomes high (the active-count pre-scaled counter clock)

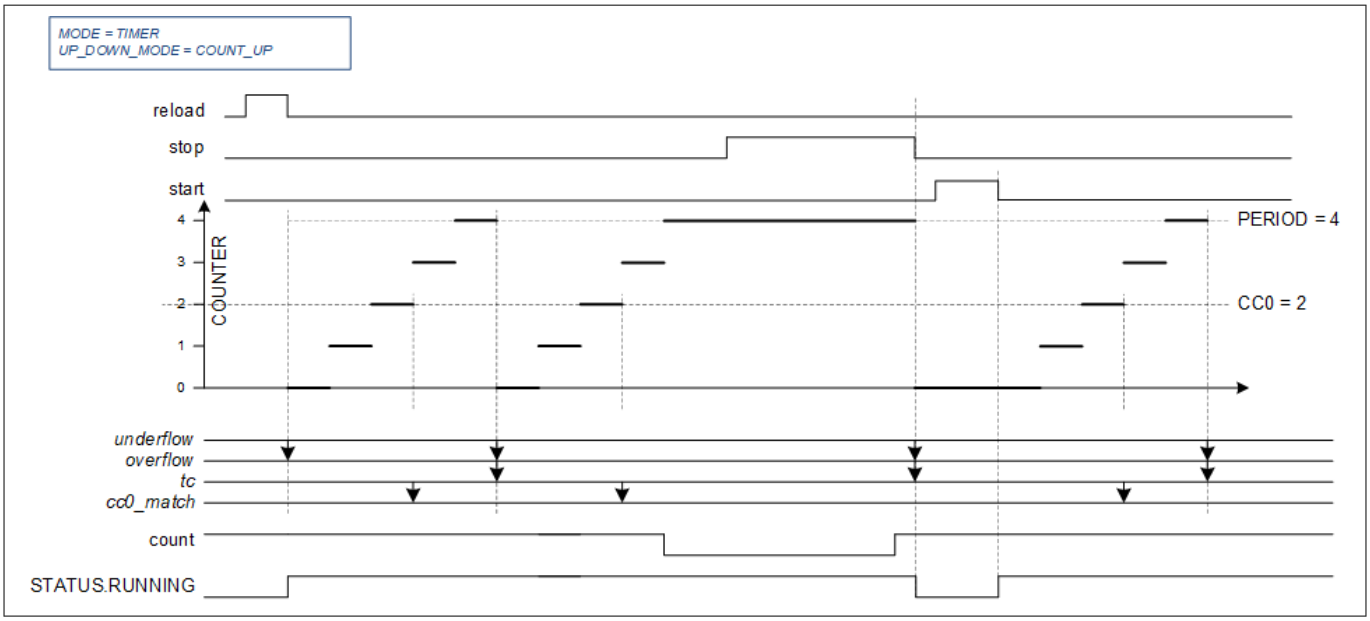


Figure 86 Function of count event

Timer mode supports an external direction control feature through the CTRL.QUAD_ENCODING_MODE bit. In this mode, the direction of counting (up or down) is determined by Capture0 events. When the CTRL.QUAD_ENCODING_MODE bit is set to 0, the feature is disabled.

When enabled, the user can change the current count direction based on the falling edge, rising edge, or level of the Capture0 input signal. The following diagrams demonstrate examples of how the counter changes its counting direction in timer mode based on falling edges, rising edges, or signal levels.

***Note**:*

- *The user must set TR_IN_EDGE_SEL.CAPTURE0_EDGE to 3 (NO_EDGE_DET) to enable this feature*
- *This feature should only be used with COUNT_UP or COUNT_DOWN modes in CTRL.UP_DOWN_MODE. Using COUNT_UPDN1 or COUNT_UPDN2 in CTRL.UP_DOWN_MODE may result in unexpected behavior*

Additionally, a software-based direction change can be triggered through the TR_CMD.CAPTURE0 bit.

- The counter's direction will change upon triggering the event, but the CTRL.QUAD_ENCODING_MODE register must be set to EXT_DIR_RISING in Timer mode
- The hardware automatically clears the TR_CMD.CAPTURE0 bit to 0 once the direction change has taken effect

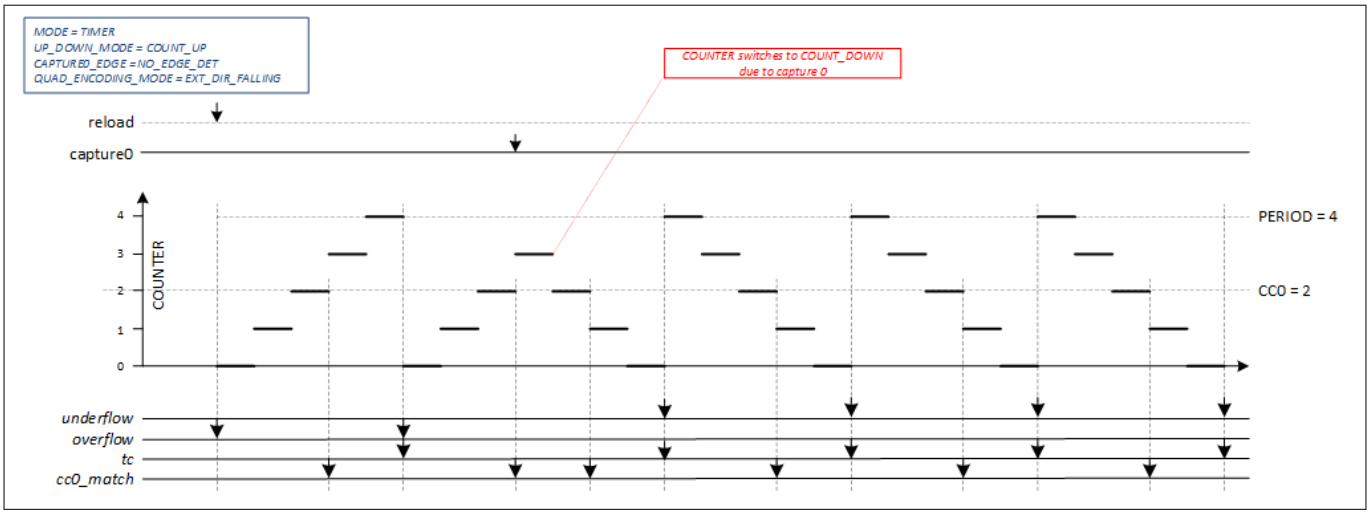


Figure 87 External direction control based on EXT_DIR_FALLING setting

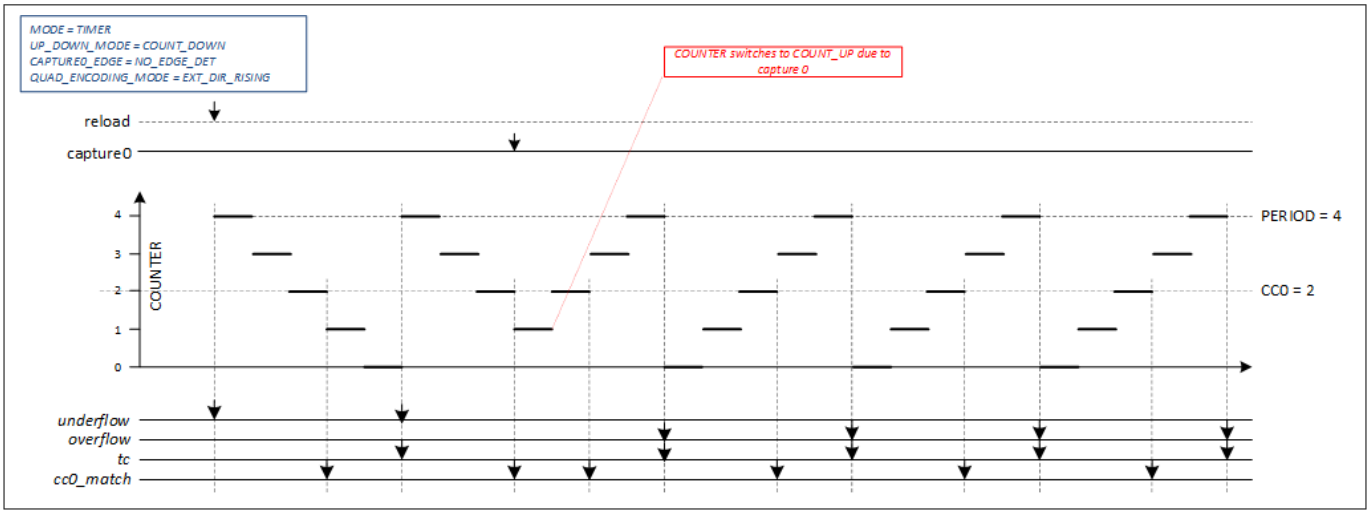


Figure 88 External direction control based on EXT_DIR_RISING setting

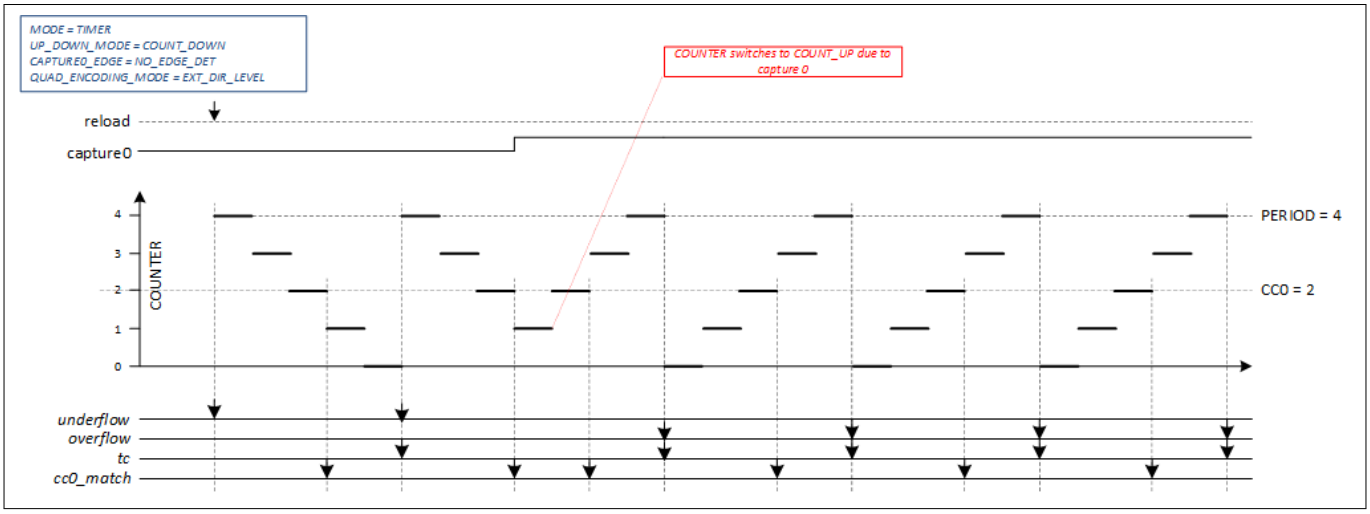


Figure 89 External direction control based on EXT_DIR_LEVEL setting

22.3.1.1 Configuring counter for timer mode

The steps to configure the counter for Timer mode and the corresponding register bits are as follows:

1. Disable the counter by writing `0` to the `ENABLE` bit of the `CTRL` register
2. Select Timer mode by writing `000` to the `MODE[26:24]` field of the `CTRL` register
3. Set the required 16-bit or 32-bit period in the `PERIOD` register
4. Set the 16-bit or 32-bit compare value in the `CC0` register and the buffer compare value in the `CC0_BUFF` register
5. Set the `AUTO_RELOAD_CC0` field of the `CTRL` register, if required, to switch values at every `CC` condition
6. Set clock pre-scaling by writing to the `DT_LINE_OUT_L[7:0]` field of the `DT` register
7. Set the direction of counting by writing to the `UP_DOWN_MODE[17:16]` field of the `CTRL` register
8. Configure the timer to run in either continuous mode or one-shot mode by writing `0` or `1`, respectively, to the `ONE_SHOT[18]` field of the `CTRL` register
9. 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)
10. 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)
11. If required, configure the interrupt for conditions such as `TC`, `CC0_MATCH`, or `CC1_MATCH`
12. Enable the counter by writing `1` to the `ENABLE` bit of the `CTRL` register. If the hardware reload signal is not enabled, provide a reload trigger through firmware by setting the `RELOAD` bit in the `TR_CMD` register to start the counter

22.3.2 Capture mode

The capture functionality increments or decrements a counter between `0` and `PERIOD`. While counting, the `Capture0` event is monitored. When the `Capture0` event is activated, the current counter value (`COUNTER`) is copied to `CC0`, and `CC0` is copied to `CC0_BUFF`.

The `STATUS.CC0_READ_MISS` bit indicates whether the data in `CC0` and `CC0_BUFF` is overwritten before the software has read it. The `STATUS.CC0_READ_MISS` bit is set to `1` if a past capture event is overwritten by new data from the `COUNTER` before the software reads `CC0` or `CC0_BUFF`. The `STATUS.CC0_READ_MISS` bit is cleared to `0` when there is a read operation on either `CC0` or `CC0_BUFF`.

The `Capture1` event and the `STATUS.CC1_READ_MISS` bit are available when the parameter `GRP_CC1_PRESENT` is set to `1`. In this case, when the `Capture1` event is activated, the current counter value (`COUNTER`) is copied to `CC1`, and `CC1` is copied to `CC1_BUFF`.

The capture functionality can be used to measure the width of a pulse by connecting the pulse signal as one of the input triggers and using it as a `Capture0` or `Capture1` event.

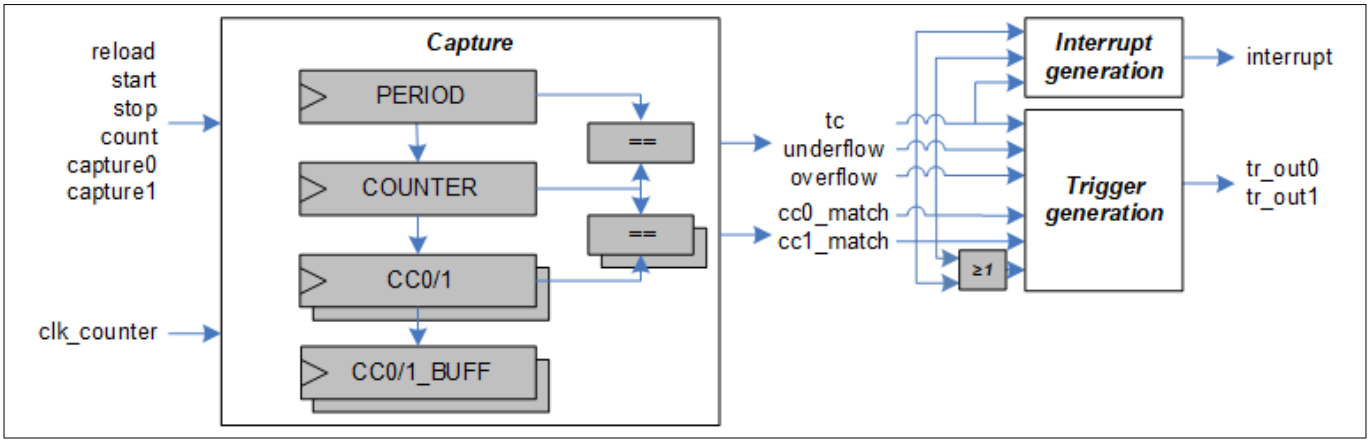


Figure 90 Capture mode function diagram

Table 104 Capture mode trigger input description

Generated events	Usage
Reload	Initializes and starts the counter. Can be used while counter is running or not running. Initialization is dependent on UP_DOWN_MODE: • **COUNT_UP**: Counter is initialized to “0” • **COUNT_DOWN**: Counter is initialized to PERIOD • **COUNT_UPDN1/2**: Counter is initialized to “1”
Start	Starts the counter (counter is NOT initialized by HW).Can be used when the counter is running or not running. When counter is running, 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.Only available when parameter GRP_CC1_PRESENT = 1.

Table 105 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, counter is stopped on tc event when underflow.
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 106 Internal events of capture

Internal events	Description
cc0_match	CC0 is copied to CC0_BUFF and counter value is copied to CC0 (cc0_match equals Capture0 event).
cc1_match	CC1 is copied to CC1_BUFF and counter value is copied to CC1 (cc1_match equals Capture1 event).Only available when parameter GRP_CC1_PRESENT = 1.

(table continues...)

Table 106 (continued) Internal events of capture

Internal events	Description
Underflow	Counter is decrementing and changes from a state in which COUNTER equals “0”.Reload event generate 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 generate 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 107 Capture mode line outputs

PWM outputs	Description
line_out	Not used
line_compl_out	Not used

Capture COUNTER value into CC0/CC0_BUFF:

When a `Capture0` event is detected, the COUNTER value is copied into the `CC0` register, and simultaneously, the value of the `CC0` register is copied into the `CC0_BUFF` register. This feature is useful for measuring the width of a pulse by connecting the pulse signal as one of the input triggers and using it as a `Capture0` event with rising- or falling-edge detection mode.

Figure 91 shows the capture behavior in `COUNT_UP` mode.

Note:
- *`Capture0` event detection uses rising-edge detection. As a result, the `Capture0` event is remembered until the next active count pre-scaled counter clock*
- *When a `Capture0` event occurs:*
  - *The COUNTER value is copied into `CC0`*
  - *The `CC0` value is copied into `CC0_BUFF`*
- *A `cc0_match` event is generated when the counter value is captured*
- *The `CC0_READ_MISS` bit will be set to `1` if the software does not perform a read operation before the third capture event. It is cleared to `0` once the software performs a read operation on either the `CC0` or `CC0_BUFF` register*

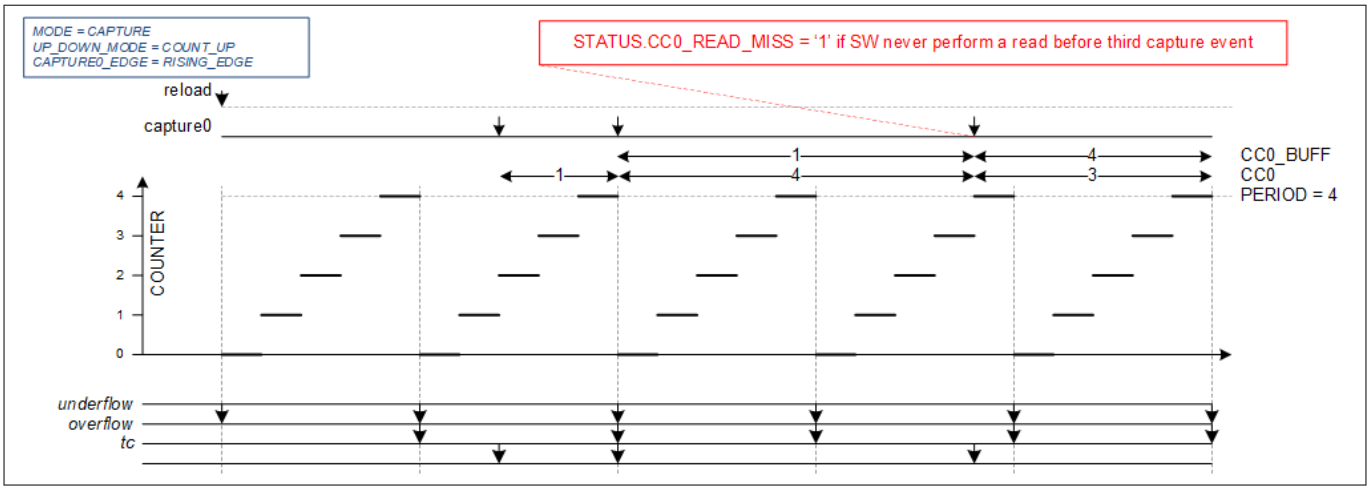


Figure 91 CC0/CC0_BUFF updated by Capture 0 event

When multiple capture events are detected before the next active count pre-scaled counter clock, the capture events are processed as follows:

- In rising-edge or falling-edge detection mode, multiple events are effectively consolidated into a single event
- In rising/falling-edge detection mode, an even number of events is not detected, while an odd number of events is reduced to a single event

The above behavior is illustrated in Figure 92 and Figure 93:

- Figure 92 demonstrates the behavior using rising-edge detection
- Figure 93 demonstrates the behavior using rising/falling-edge detection

In both Figure 92 and Figure 93, a pre-scaler with a factor of 4 is used.

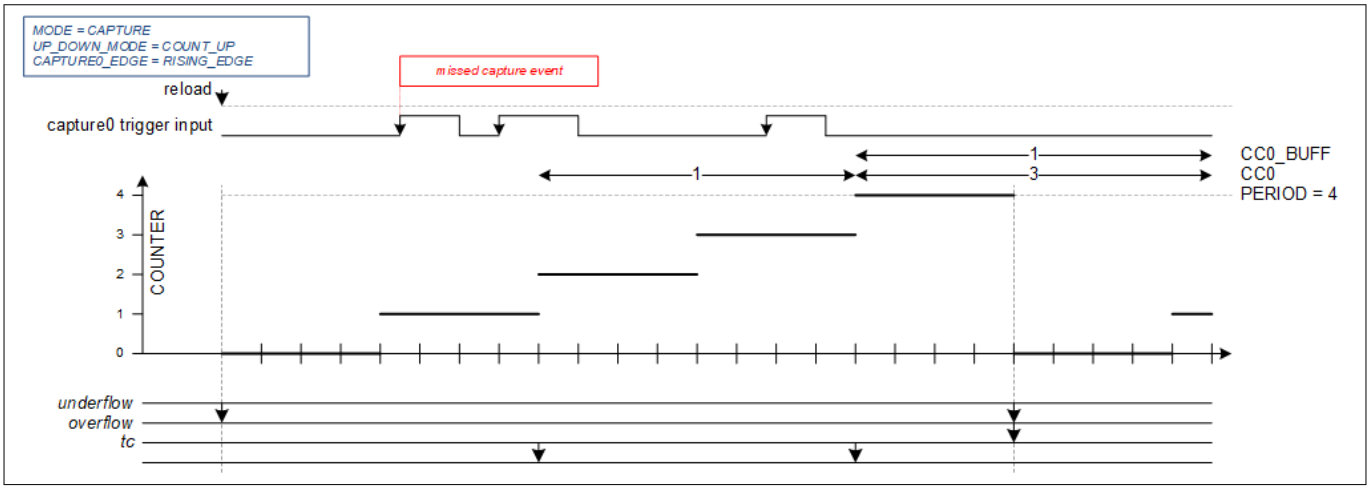


Figure 92 Multiple rising-edge capture

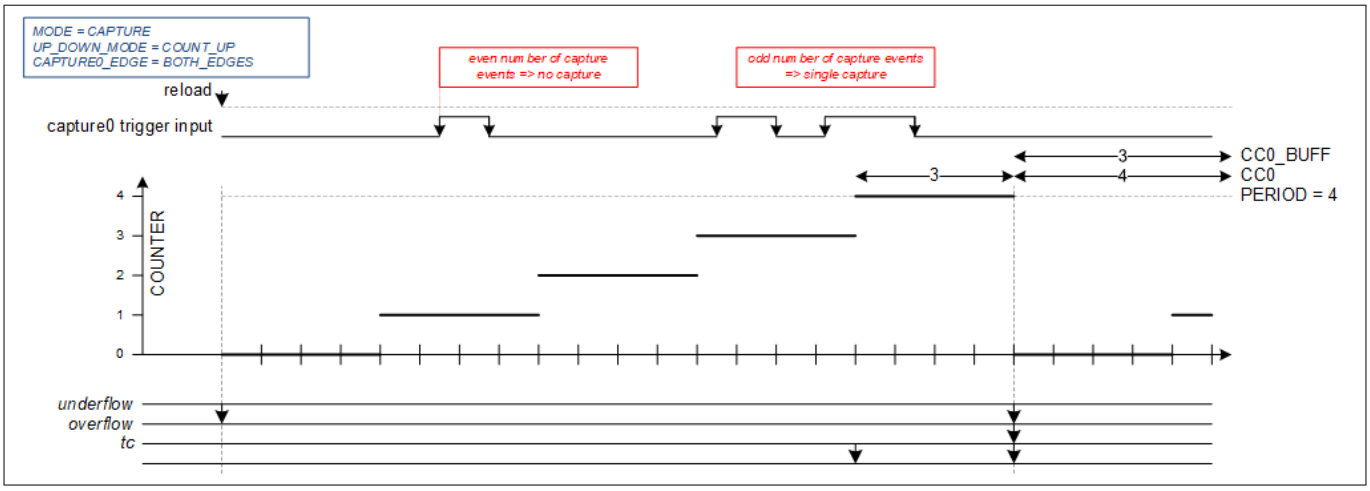


Figure 93 Multiple rising/falling both edge capture

22.3.2.1 Configuring counter for capture mode

The steps to configure the counter for Capture mode operation and the corresponding register bits are as follows:

1. Disable the counter by writing `0` to the `ENABLE` bit of the `CTRL` register
2. Select Capture mode by writing `010` to the `MODE[26:24]` field in the `CTRL` register
3. Set the required 16-bit period in the `PERIOD` register
4. Configure clock pre-scaling by writing to the `DT_LINE_OUT_L[17:16]` field of the `DT` register
5. Set the counting direction by writing to the `UP_DOWN_MODE[17:16]` field in the `CTRL` register
6. Configure the counter to run in either continuous mode or one-shot mode by writing `0` or `1`, respectively, to the `ONE_SHOT[18]` field of the `CTRL` register
7. Select the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count) by writing to the `TR_IN_SEL0` or `TR_IN_SEL1` register
8. Choose the edge of the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count) by writing to the `TR_IN_EDGE_SEL` register
9. If required, configure the interrupt for conditions such as `TC`, `CC0_MATCH`, or `CC1_MATCH`
10. Enable the counter by writing `1` to the `ENABLE` bit in the `CTRL` register. If the hardware reload signal is not enabled, a reload trigger must be provided through firmware using the `RELOAD` bit in the `TR_CMD` register to start the counter

22.3.3 Pulse-width modulation (PWM) mode

The PWM can output left-aligned, right-aligned, center-aligned, or asymmetrically aligned PWM signals. PWM functionality increments or decrements a counter between 0 and the PERIOD. During operation, the counter value is compared with CC0. When the COUNTER equals CC0, a `cc0_match` event is generated.

- When the parameter `GRP_CC1_PRESENT = 1` and the counter is running, the counter value COUNTER is also compared with CC1
- When COUNTER equals CC1, a `cc1_match` event is generated
- On both counter overflow and underflow, `overflow` and `underflow` events are generated

The events `cc0_match`, `cc1_match`, `overflow`, and `underflow` combine 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 also be generated using the COUNT_UPDN2 mode.

- When the parameter GRP_CC1_PRESENT = 1, while the cc0_match event is used for PWM generation, the cc1_match event may trigger an ADC conversion at an arbitrary point within the PWM period for diagnostic purposes
- The cc1_match event can also be utilized for PWM generation
- For example, asymmetric PWM signals can be generated using a combination of cc0_match and cc1_match in the COUNT_UPDN1 mode

When the parameter GRP_AMC_PRESENT = 1, the PWM mode can be used for stepper motor control, including microstepping.

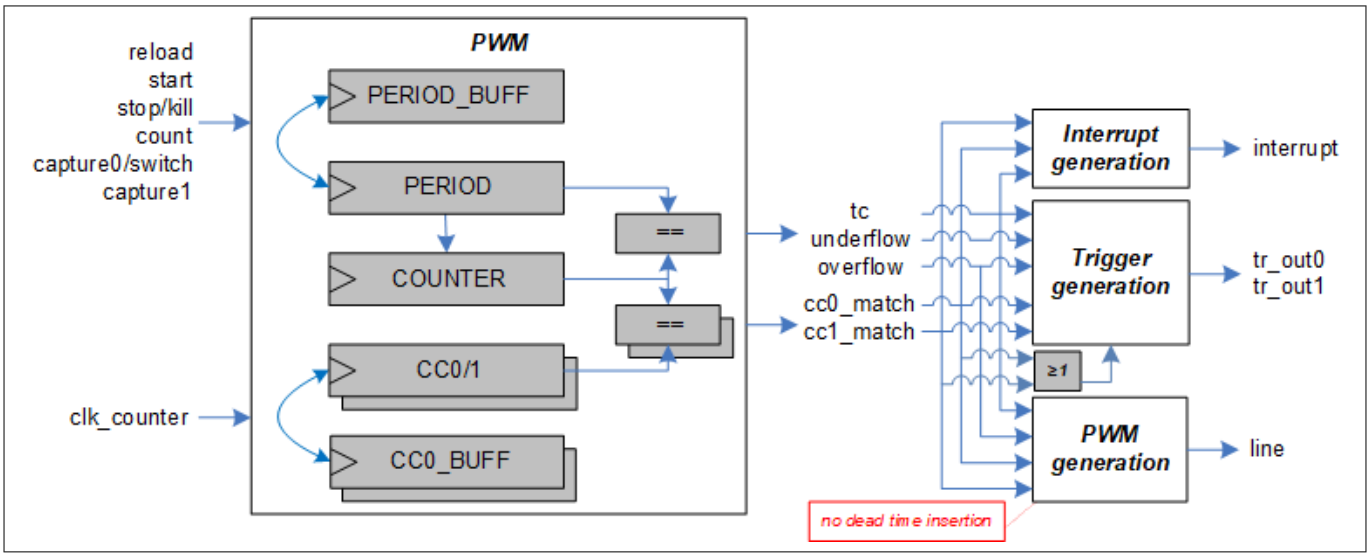


Figure 94 PWM mode functionality

Table 108 PWM mode trigger input description

Trigger Inputs	Usage
Reload	Initializes and starts the counter. This should only be used when the counter is not running. The initialization depends on the UP_DOWN_MODE setting: • **COUNT_UP**: The counter is initialized to 0 • **COUNT_DOWN**: The counter is initialized to PERIOD • **COUNT_UPDN1/2**: The counter is initialized to 1
Start	Starts the counter (counter is NOT initialized by HW).Can be used when the counter is running or not running. When counter is running, 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.

(table continues...)

Table 108 (continued) PWM mode trigger input description

Trigger Inputs	Usage
Capture0	This event functions as a switch event. When active, the following register exchanges occur during a `tc` event, provided they are enabled by the associated control parameters and conditions: • `CC0_BUFF` is transferred to `CC0` • `CC1_BUFF` is transferred to `CC1` when the parameter `GRP_CC1_PRESENT = 1` • `PERIOD_BUFF` is transferred to `PERIOD` • `LINE_SEL_BUFF` is transferred to `LINE_SEL` when the parameter `GRP_SMC_PRESENT = 1` These transfers take place based on the configuration of `CTRL.AUTO_RELOAD_CC0/1`, `CTRL.AUTO_RELOAD_PERIOD`, and `CTRL.AUTO_RELOAD_LINE_SEL`, depending on the value of `CTRL.SWAP_ENABLE`: • When `CTRL.SWAP_ENABLE = "1"`, the specified register values are exchanged • When `CTRL.SWAP_ENABLE = "0"`, the buffered values are directly transferred When a switch event is detected, it remains pending until the next `tc` event occurs. A switch event can be configured to require rising-edge, falling-edge, or both rising and falling-edge modes. Pass-through mode is unsupported unless the selected event is constantly set to `0` or `1`. If a switch event is detected when the counter is not running, the event will be cleared and have no effect.

(table continues...)

Table 108 (continued) PWM mode trigger input description

Trigger Inputs	Usage
Capture1 (Stop1/Kill1)	This event functions as a second stop/kill event and performs the same operation as the `stop0/kill0` event. Both events are OR-combined to ensure either can trigger the desired operation. • This functionality is only available when the parameter `GRP_CC1_PRESENT = 1` • Having two stop/kill events for a PWM enables greater flexibility in selecting triggers **Key benefits of two stop/kill events for PWM operation:** • One stop/kill event can use a common trigger from the PERI trigger multiplexer. This allows synchronous stop/kill operation across multiple PWMs • The other stop/kill event can use a dedicated ADC out-of-range trigger. This allows real-time hardware intervention, for example, by stopping the PWM when the current of the PWM-driven signal exceeds the defined range

Table 109 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, counter is stopped on tc event when underflow.
Auto reload CC	`CC0` and `CC0_BUFF` are exchanged during both a switch event and a `tc` event, as specified by `CTRL.AUTO_RELOAD_CC0`. When the parameter `GRP_CC1_PRESENT = 1`, `CC1` and `CC1_BUFF` are also exchanged during both a switch event and a `tc` event, as specified by `CTRL.AUTO_RELOAD_CC1`.
Auto reload PERIOD	PERIOD and `PERIOD_BUFF` are exchanged during both a switch event and a `tc` event, as specified by `CTRL.AUTO_RELOAD_PERIOD`. ***Note**: In `COUNT_UPDN2` mode, when PERIOD and `PERIOD_BUFF` are exchanged at a `tc` event that coincides with an overflow event, the software must ensure that the PERIOD and `PERIOD_BUFF` values are identical. This prevents unintended behavior caused by mismatched values during the exchange.*

(table continues...)

Table 109 (continued) PWM mode supported features

Supported features	Description
Auto reload LINE_SEL when parameter GRP_SMC_PRESENT = 1	LINE_SEL and LINE_SEL_BUFF are exchanged during both a switch event and a tc event, provided the following conditions are met: • CTRL.AUTO_RELOAD_LINE_SEL is enabled • CTRL.SWAP_ENABLE = "1" When CTRL.SWAP_ENABLE = "0", LINE_SEL_BUFF is directly transferred to LINE_SEL during both a switch event and a tc event, as specified by CTRL.AUTO_RELOAD_LINE_SEL.
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 then back to 0
Kill modes	The behavior is determined by the following parameters: • PWM_IMM_KILL • PWM_SYNC_KILL • PWM_STOP_ON_KILL • PWM_TC_SYN_KILL_DT • PWM_SYN_KILL_DT Refer to the sections below for detailed descriptions and explanations of these parameters.

***Note**: The PWM mode does not support dead-time insertion. This functionality is available in the separate PWM_DT mode.*

Table 110 PWM mode trigger output description

Trigger output	Description
cc0_match	Specified by UP_DOWN_MODE: • **COUNT_UP and COUNT_DOWN**: The counter transitions to a state where COUNTER equals cc0 • **COUNT_UPDN1/2**: The counter transitions from a state where COUNTER equals cc0 - If GRP_AMC_PRESENT = 1, CC0_MATCH_UP_EN and CC0_MATCH_DOWN_EN enable or disable cc0_match generation, depending on whether the counter is in up-counting or down-counting modes - When cc0 = 0 or cc0 = PERIOD, it serves as a special case for generating PWM signals with 0% or 100% duty cycle

(table continues...)

Table 110 (continued) PWM mode trigger output description

Trigger output	Description
cc1_match	Specified by UP_DOWN_MODE: • **COUNT_UP and COUNT_DOWN**: The counter transitions to a state where COUNTER equals CC1 • **COUNT_UPDN1/2**: The counter transitions from a state where COUNTER equals CC1 - If GRP_AMC_PRESENT = 1, CC1_MATCH_UP_EN and CC1_MATCH_DOWN_EN control the enabling or disabling of cc1_match generation, depending on whether the counter is in up-counting or down-counting modes This functionality is only available when the parameter GRP_CC1_PRESENT = 1.
Underflow	The counter decrements and transitions from a state where COUNTER equals 0. A reload event generates an underflow in COUNT_DOWN, COUNT_UPDN1, or COUNT_UPDN2 mode.
Overflow	The counter increments and transitions from a state where COUNTER equals PERIOD. A reload event generates an overflow in COUNT_UP mode.
tc	Specified by UP_DOWN_MODE: • **COUNT_UP**: A tc event is the same as the overflow event • **COUNT_DOWN**: A tc event is the same as the underflow event • **COUNT_UPDN1**: A tc event is the same as the underflow event • **COUNT_UPDN2**: A tc event is the logical OR of the overflow and underflow events Reload events generate underflow or overflow but do not generate a tc event.

Table 111 PWM mode internal events description

PWM outputs	Description
line_out	PWM line output
line_compl_out	Complementary PWM line output

Figure 95 shows the process of line_out and line_compl_out signal generation. Generally, the line_out output reflects the state of the PWM signal line, while the line_compl_out output reflects the inverted version of line. The behavior of the line signal depends on the selected functional modes.

Additionally, the following factors can influence line_out and line_compl_out within different functional modes:

- Line selection
- Dead-time insertion
- Kill function

- Line polarity
- Kill line polarity

PWM output signals are generated using underflow, overflow, and `cc_match` events. Each of these events can be individually configured to `INVERT`, `SET`, or `CLEAR` the `pwm_dt_input`.

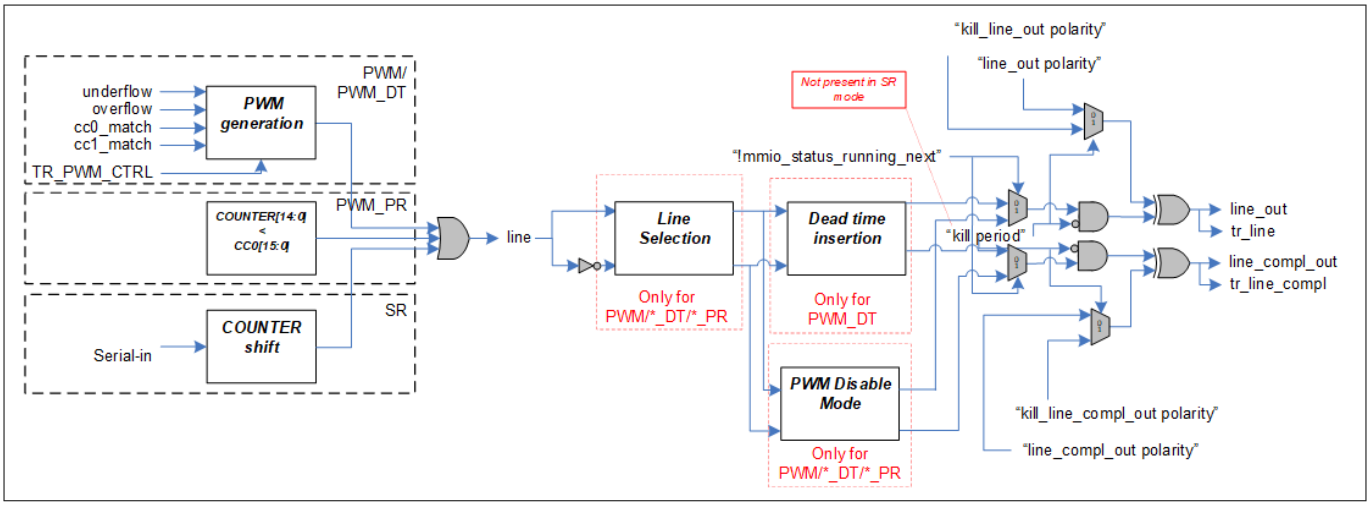


Figure 95 PWM output generation

***Note**: `cc0_match` event in `COUNT_UP` and `COUNT_DOWN` modes is different from the event generation in other TIMER/CAPTURE modes. This distinction ensures the ability to generate PWM signals with 0% and 100% duty cycles.*

PWM behavior depends on the values in the `PERIOD` and `CC0` registers. The software can update the `PERIOD_BUFF` and `CC0_BUFF` registers without affecting the ongoing PWM behavior. A switch/capture event can be used to exchange the values of the compare registers and buffered compare registers. Additionally:

- Period and buffered period register values are switched if `CTRL.SWAP_ENABLE = "1"`
- If `CTRL.SWAP_ENABLE = "0"`, the switch/capture event transfers the value from the buffered compare register to the compare register. Similarly, the buffered period register value is transferred to the period register

Table 112 summarizes the supported kill modes in PWM mode. Any settings not defined in the table are not supported or not recommended.

Table 112 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 next active count clock after kill input is active • PWM output suppress is removed at next active count clock after kill input is inactive

(table continues...)

Table 112 (continued) Kill modes of PWM

Kill mode	Settings	Kill behavior
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 get suppressed immediately after kill input is active • PWM output suppress is removed at 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 next active count clock after kill input is active • PWM output suppress is removed at 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 get suppressed immediately after kill input is active • PWM output suppress is removed at next tc event after kill input is inactive
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 next tc event after kill input is active. If deadtime is enabled, output get suppressed after the tc event + deadtime • PWM output suppress is removed at next tc event after kill input is inactive • This is only support 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 next tc event after kill input is inactive • This is only support in PWM_DT mode

(table continues...)

Table 112 (continued) Kill modes of PWM

Kill mode	Settings	Kill behavior
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 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 get suppressed immediately after kill input is active • PWM output suppress is removed when counter restart after kill input is inactive

22.3.3.1 PWM mode functionalities

***Note**: One-shot mode and clock prescaling are identical to their implementation in timer mode.*

Up/down count modes

Up/down count modes control the counting direction (increment or decrement) while the counter is running.

Figure 96 shows a PWM operating in `COUNT_UP` mode. The counter is initialized to `0` and started with a software-based reload event.

***Note**:*
- *When the counter transitions from a state where `COUNTER` equals `4`, both an overflow and a `tc` event are generated*
- *When the counter transitions to a state where `COUNTER` equals `2`, a `cc0_match` event is generated*
- *The `PERIOD` is set to `4`, resulting in an effective repeating counter pattern of `4 + 1 = 5` counter clock periods*

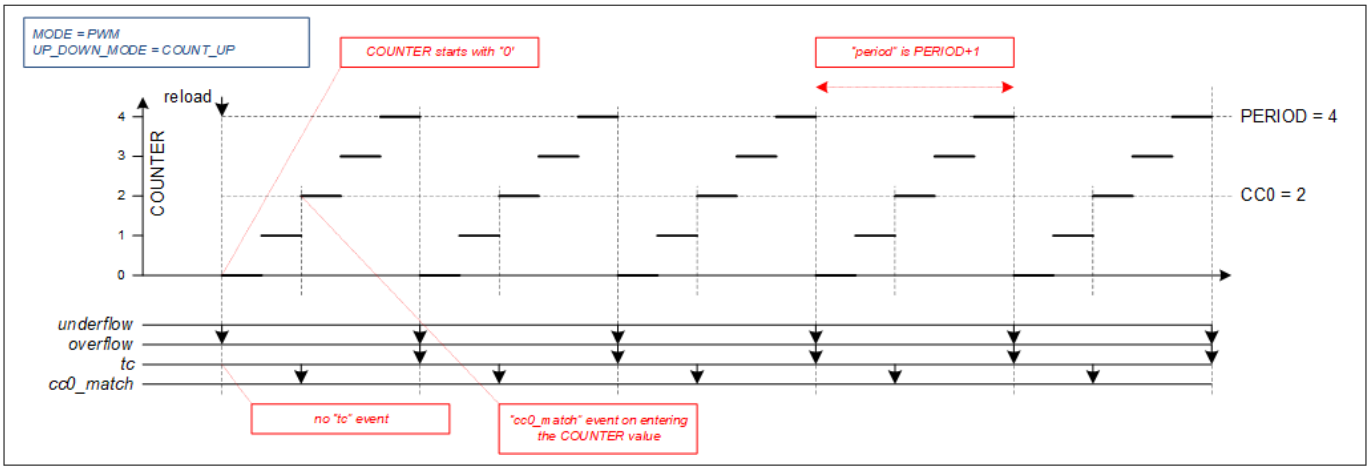


Figure 96 PWM in up-counting mode

Figure 97 shows a PWM operating in down-counting mode. The counter is initialized to PERIOD and started with a software-based reload event.

Note:

- *When the counter transitions from a state where `COUNTER` equals `0`, both an underflow and a `tc` event are generated*
- *When the counter transitions to a state where `COUNTER` equals `2`, a `cc0_match` event is generated*
- *The `PERIOD` is set to `4`, resulting in an effective repeating counter pattern of `4 + 1 = 5` counter clock periods*

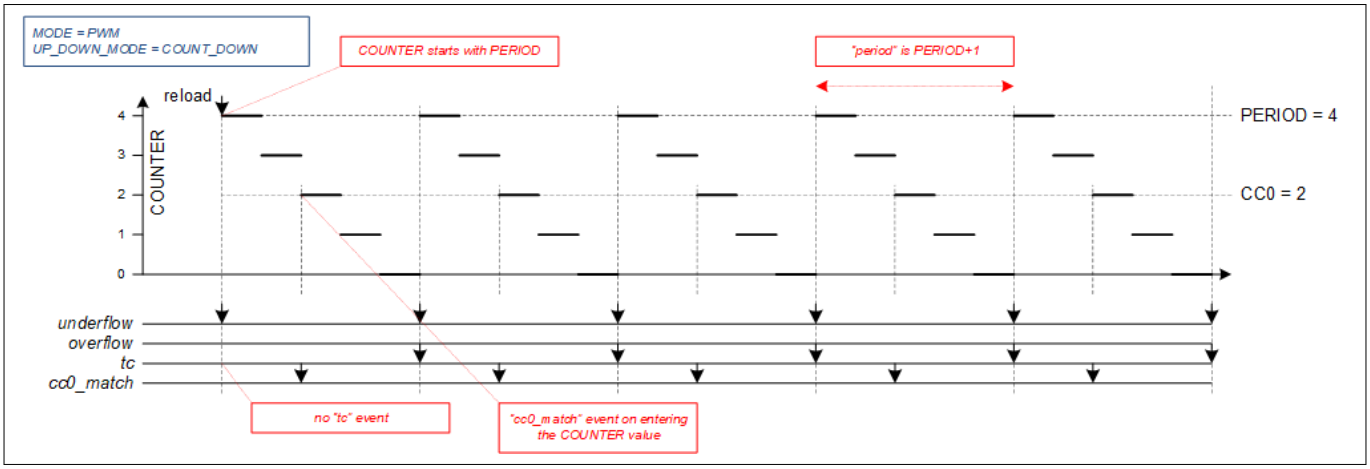


Figure 97 PWM in down-counting mode

Figure 98 shows a PWM operating in `COUNT_UPDN1` mode. The counter is initialized to `1` and started with a software-based reload event.

Note:

- *When the counter transitions from a state where COUNTER equals `4`, an overflow is generated*
- *When the counter transitions from a state where COUNTER equals `0`, both an underflow and a `tc` event are generated*
- *When the counter transitions from a state where `COUNTER` equals `2`, a `cc0_match` event is generated*
  - *Note: The counter value (COUNTER) from before the reload event is not used. Instead, the counter value before the reload event is considered to be `0`*
- *The PERIOD is set to `4`, resulting in an effective repeating counter pattern of `2 * 4 = 8` counter clock periods*

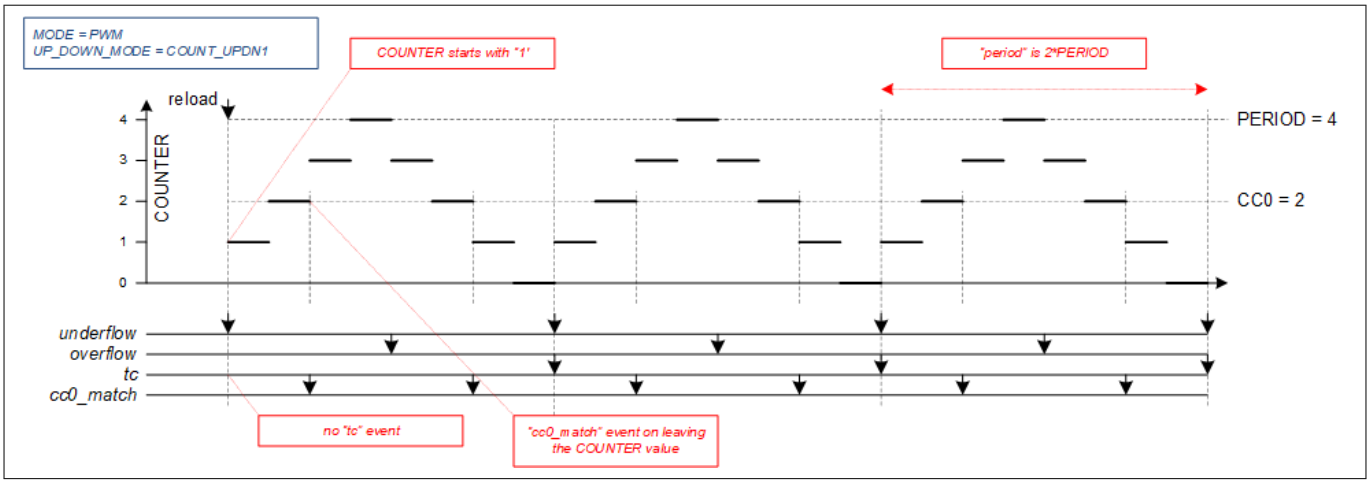


Figure 98 PWM in COUNT_UPDN1 mode

CC0 and PERIOD auto reload with switch event

Auto CC reload and auto PERIOD reload provide a mechanism for dynamically updating the PWM duty cycle and counter period length. An active switch event (`capture0`) is required for such switching activity.

- At a `tc` event, if the switch event is active and `CTRL.AUTO_RELOAD_CC0` is set to `1`, `CC0` and `CC0_BUFF` will exchange their values
- At a `tc` event, if the switch event is active and `CTRL.AUTO_RELOAD_PERIOD` is set to `1`, `PERIOD` and `PERIOD_BUFF` will exchange their values
- If `GRP_CC1_PRESENT = 1`, `CC1` and `CC1_BUFF` will also exchange their values at a `tc` event, provided `capture0` is active

Figure 99, Figure 100, Figure 101, and Figure 102 shows how the period value is updated in `COUNT_UP` mode and `COUNT_DOWN` mode for cases where `SWAP_ENABLE` is set to `1` and `SWAP_ENABLE` is set to `0`.

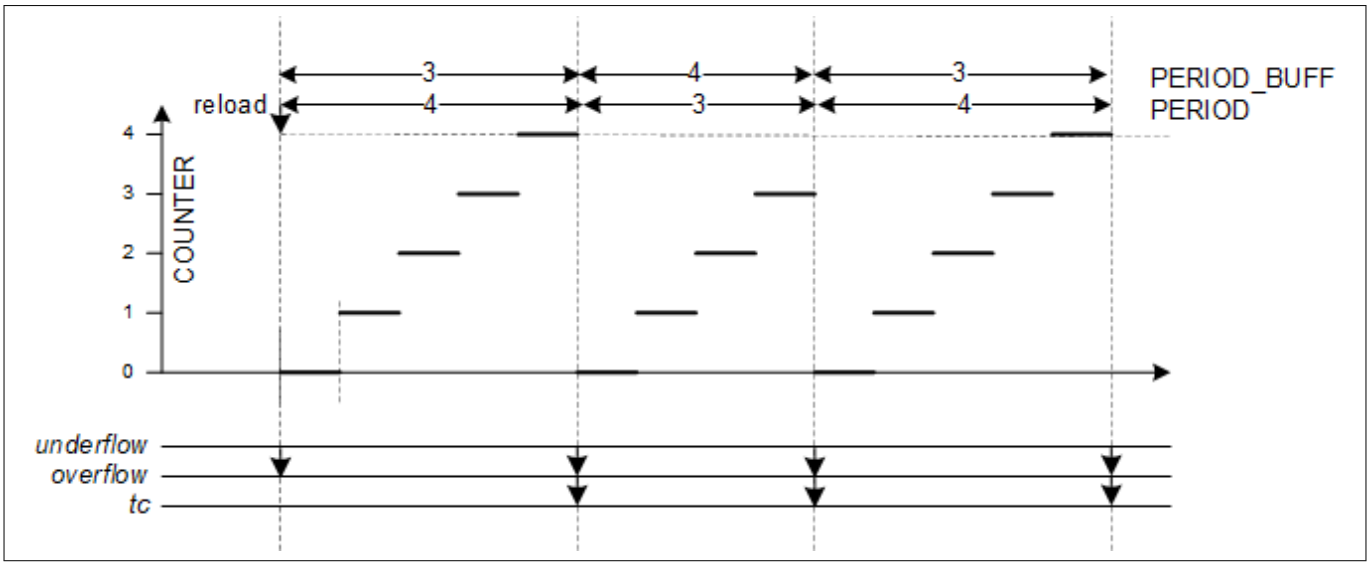


Figure 99 PERIOD/PERIOD_BUFF exchange in COUNT_UP mode (SWAP_ENABLE = "1")

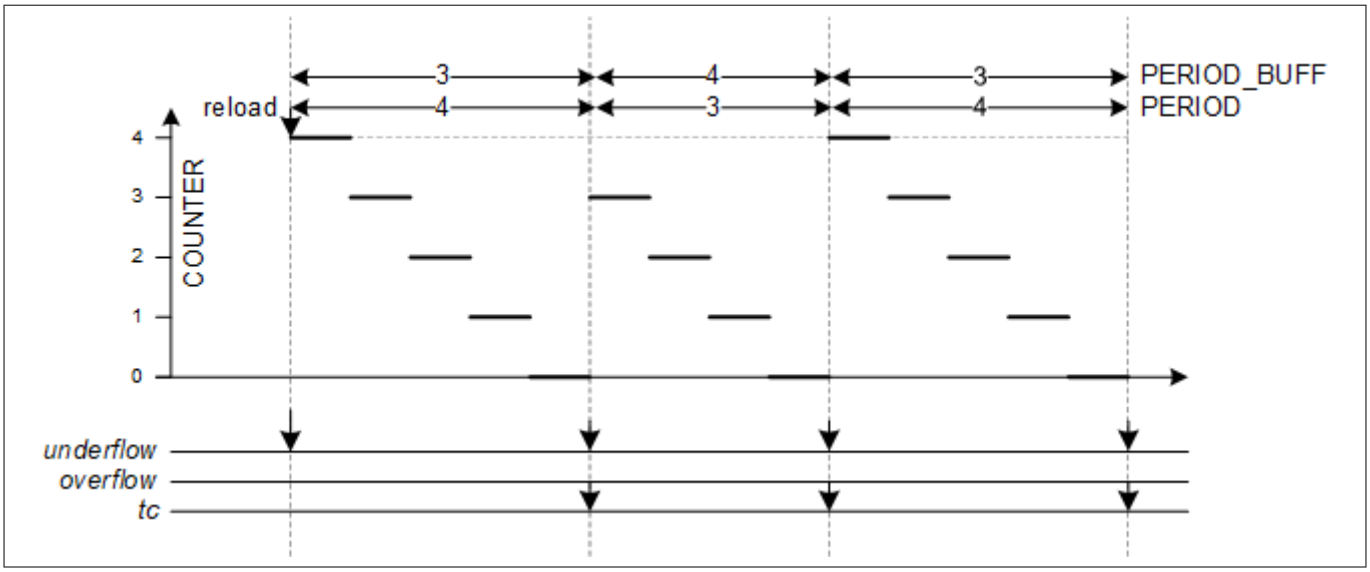


Figure 100 PERIOD/PERIOD_BUFF exchange in COUNT_DOWN mode (SWAP_ENABLE = "1")

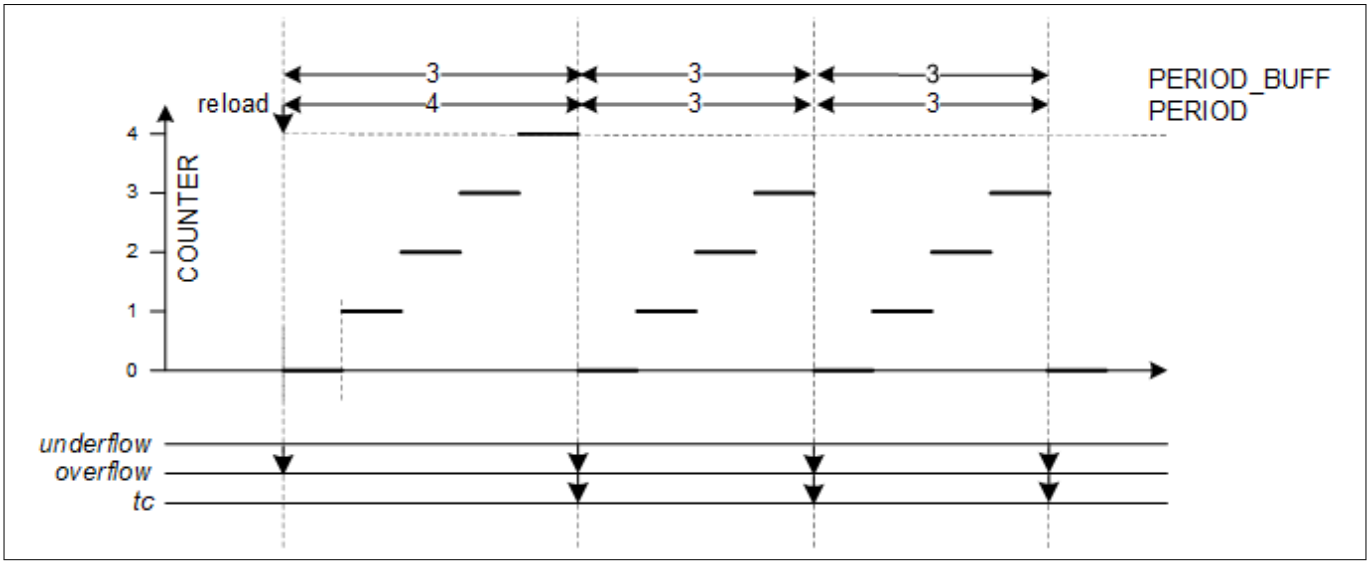


Figure 101 PERIOD/PERIOD_BUFF exchange in COUNT_UP mode (SWAP_ENABLED = "0")

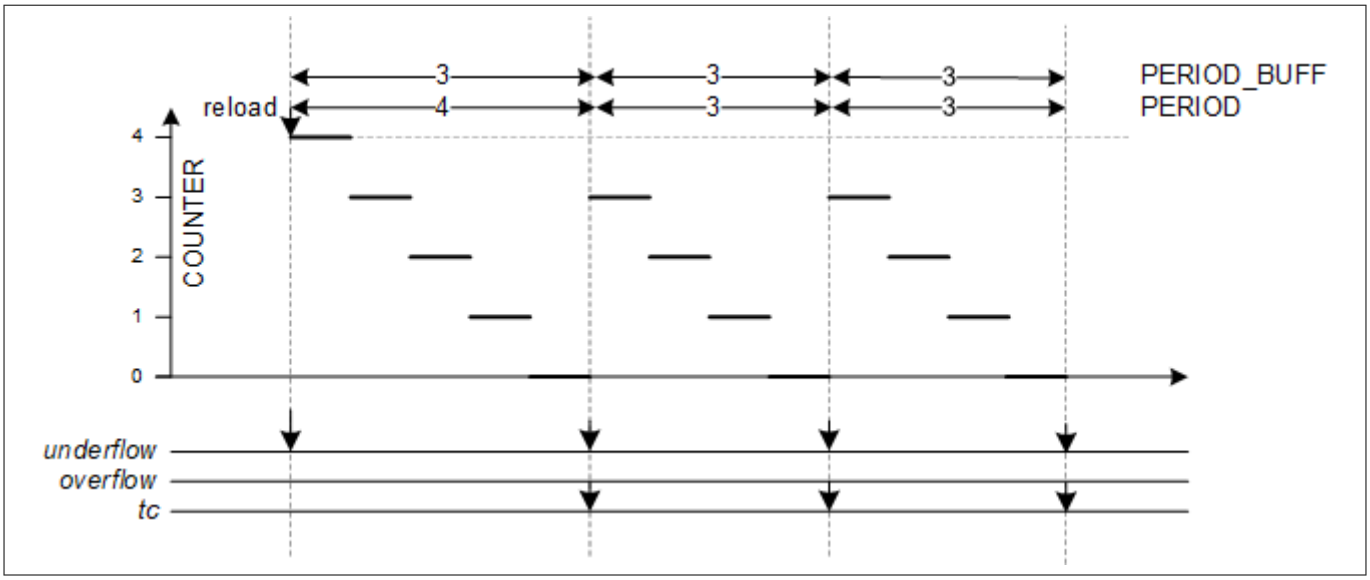


Figure 102 PERIOD/PERIOD_BUFF exchange in COUNT_DOWN mode (SWAP_ENABLE="0")

A potential issue arises when software updates are not completed before the next `tc` event with a pending active switch event. For example, if the software updates `PERIOD_BUFF` before the `tc` event and `CC0_BUFF` after the `tc` event, the switch operation does not reflect the updated value of the `CC0_BUFF` register. To prevent this, the switch event should be generated by the software through a memory-mapped I/O (MMIO) register write after both the `PERIOD_BUFF` and `CC0_BUFF` registers have been updated. The hardware holds the switch event in a pending state until the next `tc` event occurs.

The previous paragraph discussed the synchronized updates of the `CC0/CC0_BUFF` and `PERIOD/PERIOD_BUFF` registers for a single PWM using a software-generated switch event. During motor control, three PWMs are often synchronized, requiring updates to all period and compare register pairs to be coordinated. All three PWMs share the same period and, as a result, have synchronized `tc` events. The switch event for all three PWMs is triggered by the same hardware input, enabling a synchronous update of `CC0` at the same `tc` event after the `PERIOD_BUFF` and `CC0_BUFF` registers for all three PWMs are updated by the software.

Left/Right/Center align PWM with CC0/CC0_BUFF auto reload

PWM can generate left-aligned, right-aligned, and center-aligned outputs with the following features:

- Up/down count mode must be used to achieve different phase-aligned PWM outputs
- The `line` state changes in response to internal events such as `underflow`, `overflow`, and `cc0_match` (TR_PWM_CTRL)

Required settings for each alignment mode:

- **Left-align**: `COUNT_UP` mode with `overflow=SET` and `cc0_match=CLEAR`
- **Right-align**: `COUNT_DOWN` mode with `cc0_match=SET` and `underflow=CLEAR`
- **Center-align**: `COUNT_UPDN1` mode with `overflow=SET`, `underflow=CLEAR`, and `cc0_match=INVERT`

Expected pulse widths for each alignment mode:

- **Left-align (`COUNT_UP`)**: Pulse width = `CC0`
  - To achieve a 100% duty cycle, `CC0` must exceed `PERIOD`. If the `PERIOD` is `0xFFFF` (or `0xFFFFFFFF` for a 32-bit counter), 100% duty cycle cannot be achieved
- **Right-align (`COUNT_DOWN`)**: Pulse width = `CC0 + 1`
  - To achieve a 0% duty cycle, `CC0` must exceed `PERIOD`. If the `PERIOD` is `0xFFFF` (or `0xFFFFFFFF` for a 32-bit counter), 0% duty cycle cannot be achieved
- **Center-align (`COUNT_UPDN1`)**: Pulse width = `2 * (PERIOD - CC0)`

Figure 103 shows a PWM in COUNT_UP mode with varying CC0 values. Figure 103 also demonstrates how a left-aligned PWM can be created using the COUNT_UP mode.

Note:
- *CC0 is updated (replaced by CC0_BUFF, not depicted in the figure) during a tc event*
- *The switch event is held constant at 1*

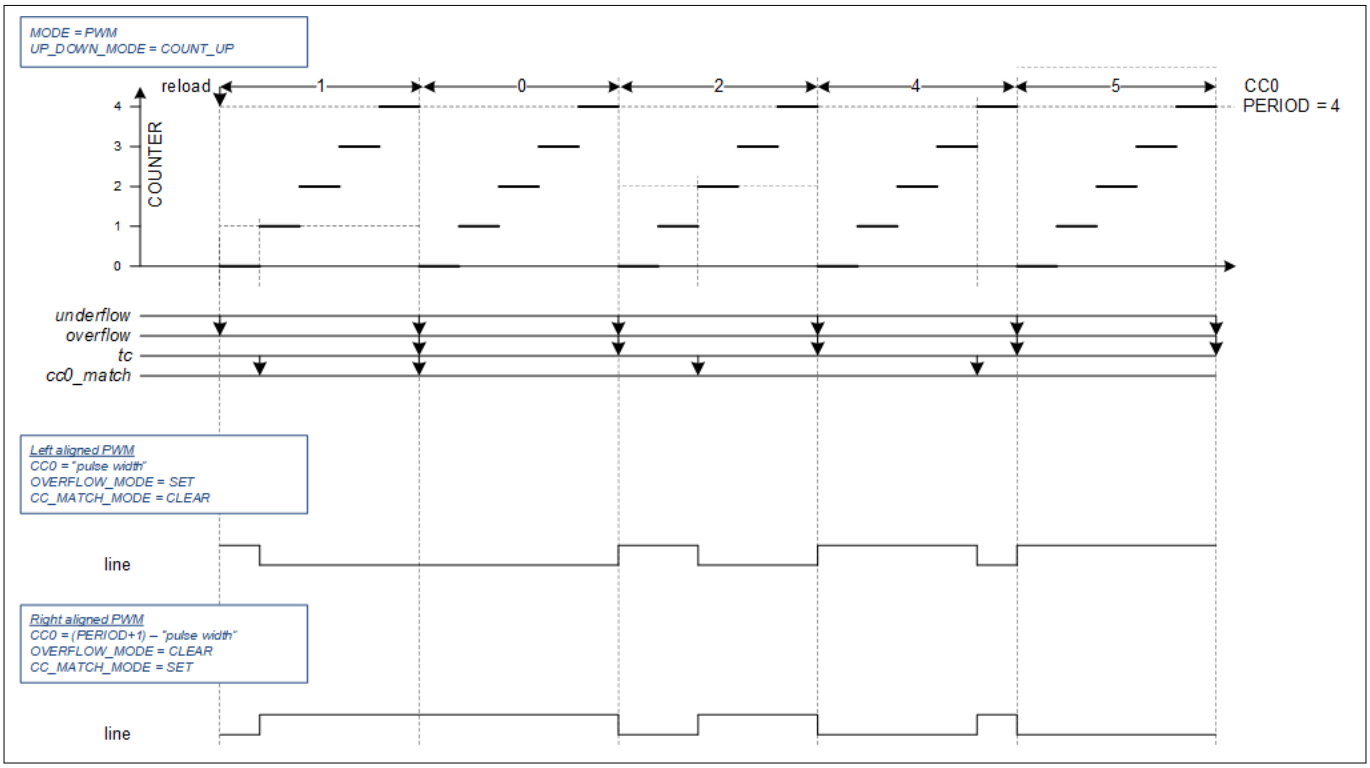


Figure 103 Left align PWM

Figure 104 shows a PWM operating in COUNT_DOWN mode with various CC0 values. Figure 104 also demonstrates how a right-aligned PWM can be created using the COUNT_DOWN mode.

Note:
- *CC0 is updated (replaced by CC0_BUFF, which is not depicted) during a tc event*
- *The switch event remains constant at 1*

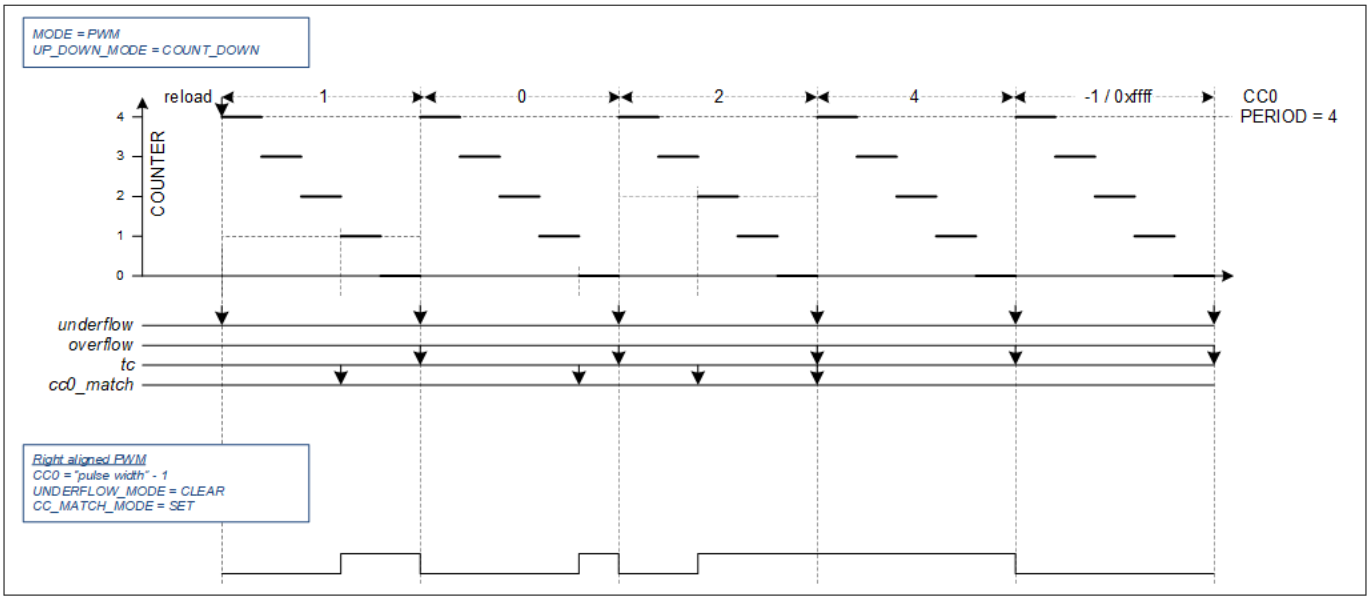


Figure 104 Right align PWM

Figure 105 shows a PWM operating in `COUNT_UPDN1` mode with various `CC0` values. Figure 105 also demonstrates how a center-aligned PWM can be created using the `COUNT_UPDN1` mode.

***Note**: The switch event is generated by hardware trigger 1, which is a constant `1` and, therefore, always active during the `tc` condition.*

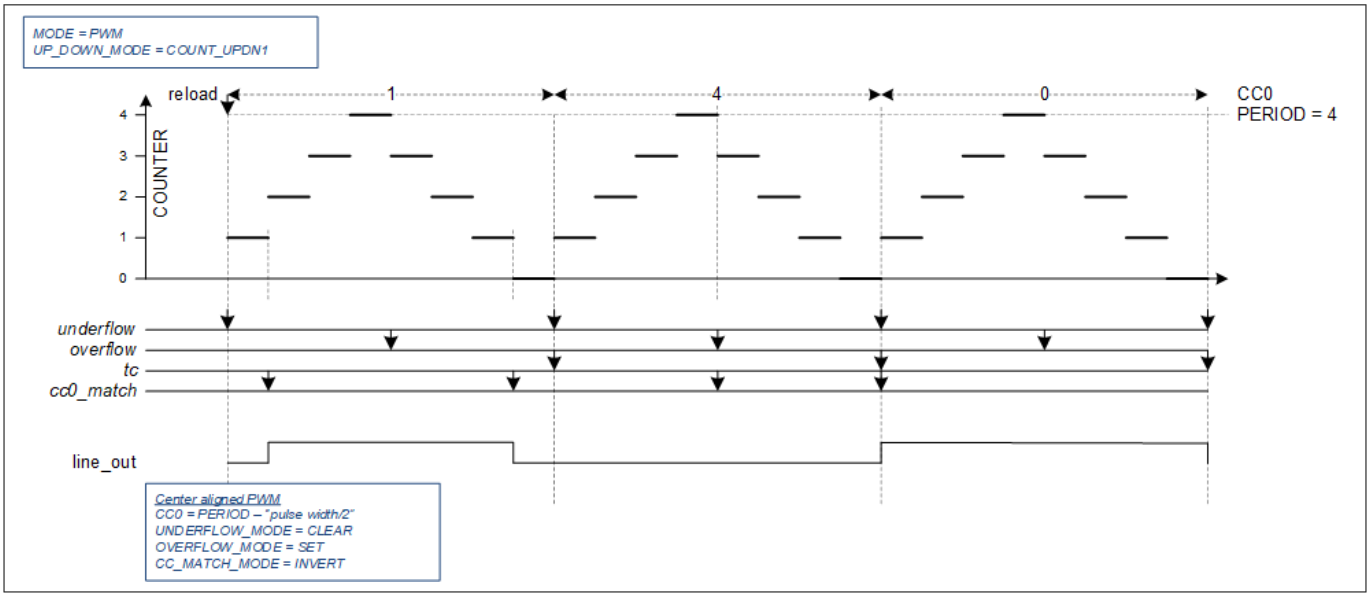


Figure 105 Center align PWM

Figure 106 shows a corner case where `CC0` equals `0` when a reload event occurs. 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.

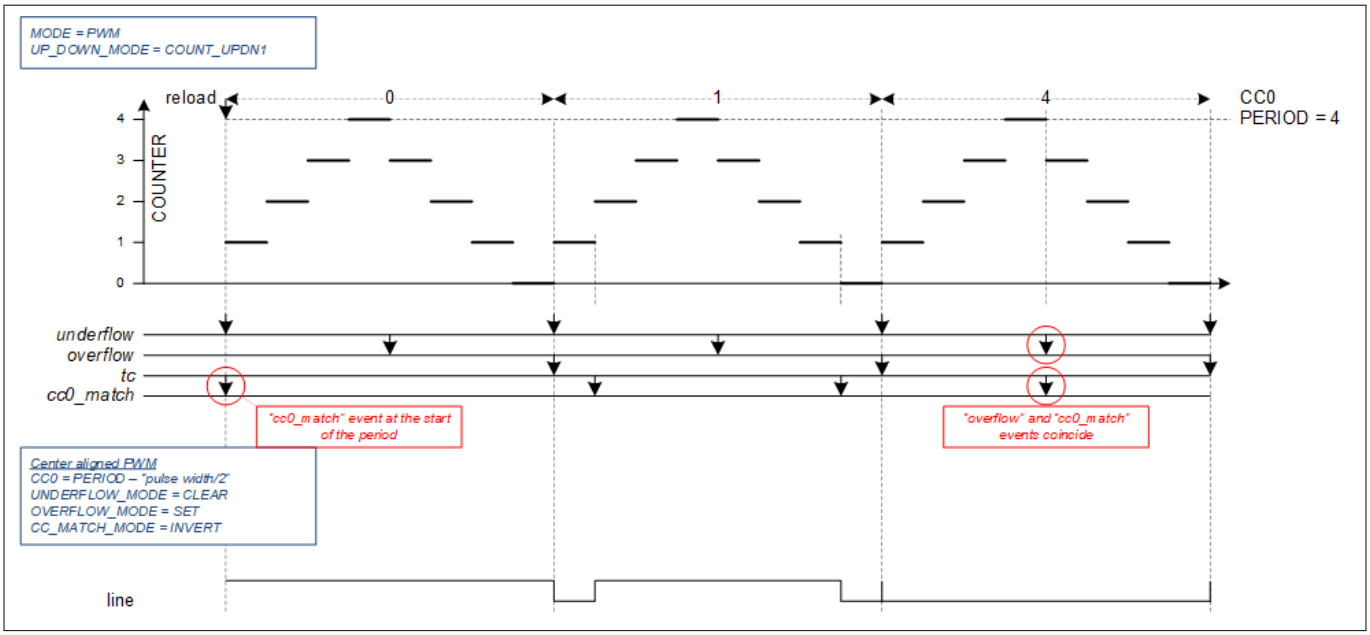


Figure 106 Center align PWM with CC0 = "0" after reload

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 achieved by using different `CC0` values during the up-counting and down-counting phases. The `CC0` and `CC0_BUFF` values are exchanged on an overflow event. However, this restricts the degree of asymmetry of the generated `line` pulse.

Settings required for asymmetric PWM:

- **Asymmetric PWM**: `COUNT_UPDN2` mode with `overflow=SET`, `underflow=CLEAR`, and `cc0_match=INVERT`

An asymmetric PWM typically uses the same period value for up-counting and down-counting. When `PERIOD` and `PERIOD_BUFF` are switched during a `tc` event (overflow or underflow), care must be taken to ensure the following:

- **Within a PWM period**: The period values must remain the same. An overflow switch of `PERIOD` and `PERIOD_BUFF` must not change the period value, meaning that `PERIOD_BUFF` should equal `PERIOD`
- **Between PWM periods**: The period value may change. An underflow switch of `PERIOD` and `PERIOD_BUFF` can change the period value, meaning that `PERIOD_BUFF` may differ from `PERIOD`

Figure 107 shows how the `COUNT_UPDN2` mode is used to generate an asymmetric PWM.

Note: *A `tc` event is generated for both underflow and overflow events. The `tc` event is used to exchange the `CC0` and `CC0_BUFF` values.*

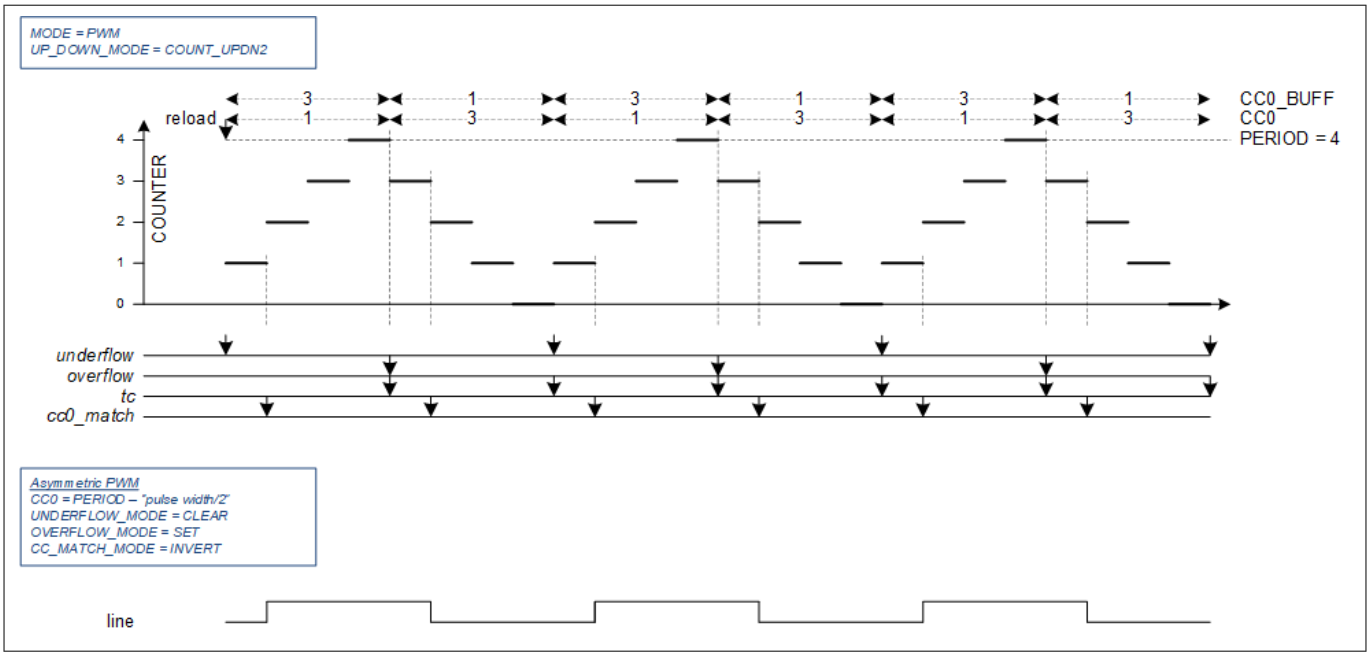


Figure 107 Asymmetric PWM

The previous waveform illustrated the functionality for cases where the CC0 values are neither 0 nor PERIOD. The corner cases, where CC0 values are equal to 0 or PERIOD, are shown below:

Note:

- *When up-counting, if the CC0 value at the underflow event is 0, a cc0_match event is generated*
- *When down-counting, if the CC0 value at the overflow event is PERIOD, a cc0_match event is generated*
- *A tc event is generated during both underflow and overflow events. 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. This overwrites the hardware-updated values resulting from the CC0/CC0_BUFF and PERIOD/ PERIOD_BUFF exchanges*

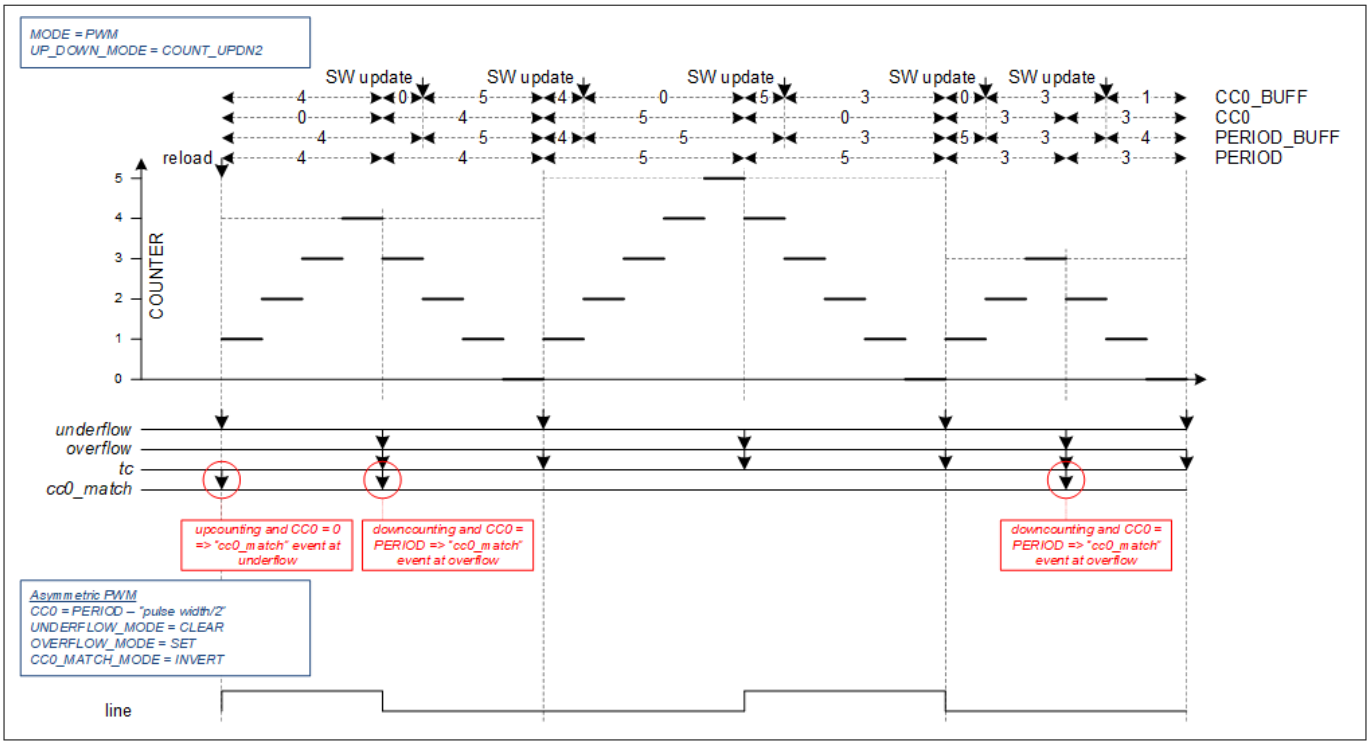


Figure 108 Asymmetric PWM with CC0=0 and CC0=PERIOD duty cycle

When the parameter `GRP_CC1_PRESENT = 1`, the compare 1 function, which uses the `CC1` and `CC1_BUFF` registers to generate the `cc1_match` event, operates similarly to the compare function that uses the `CC0` and `CC0_BUFF` registers to generate the `cc0_match` event.

The `cc1_match` event can also be used to generate PWM output signals. By using both `cc0_match` and `cc1_match` events for PWM output signal generation, an asymmetric PWM can be created. This approach is shown in Figure 109.

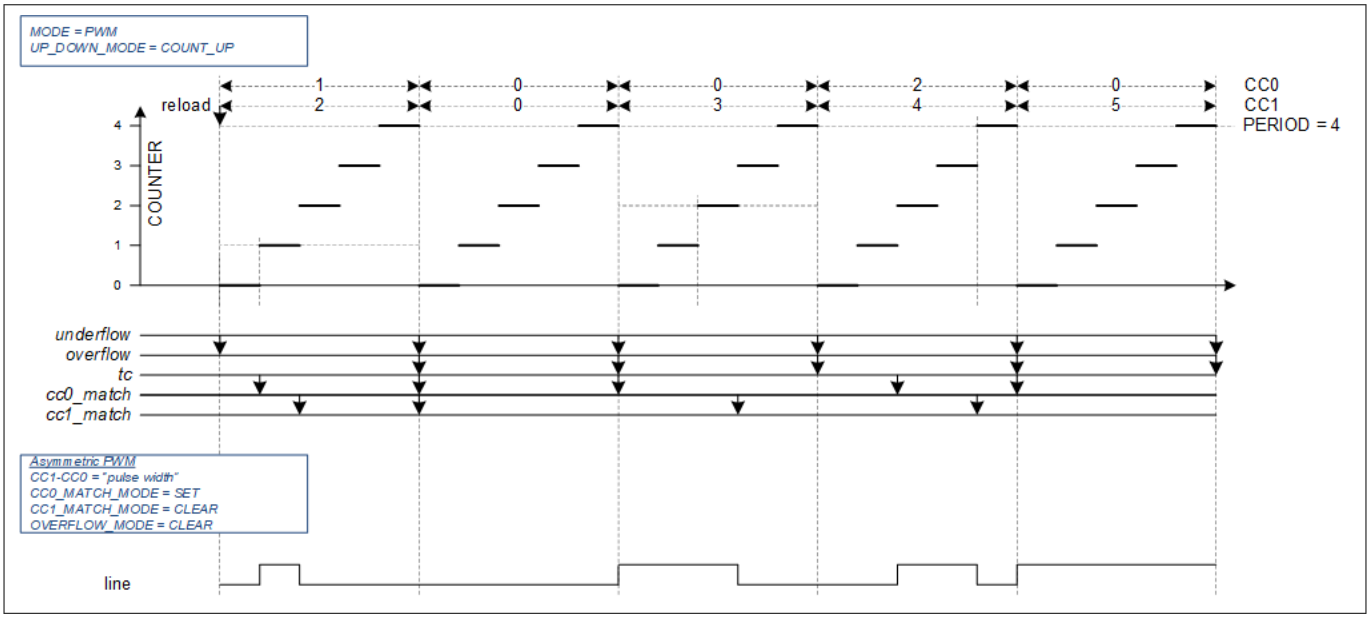


Figure 109 Asymmetric PWM by cc0_match and cc1_match in COUNT_UP mode

Such asymmetric PWM generation offers greater flexibility compared to using a single compare function in the `COUNT_UPDN2` mode. However, if a third compare function is required, for example to trigger an ADC, an additional synchronously running counter must be utilized.

When the parameter GRP_AMC_PRESENT = 1 (for advanced motor control), the generation of compare match 0 (cc0_match) and compare match 1 (cc1_match) events can be enabled or disabled individually for up-counting and down-counting during the COUNT_UPDN1/2 mode. This enables asymmetric PWM generation in the COUNT_UPDN1 mode, where:

- One compare match event modifies the PWM output only during the up-counting phase
- The other compare match event modifies the PWM output only during the down-counting phase

This is illustrated in Figure 110, which shows one of three center-aligned PWM phases used for motor control. The figure depicts a time period where the duty-cycle value is increased from one period to the next, representing the rising part of a sine wave modulated onto the PWM signal.

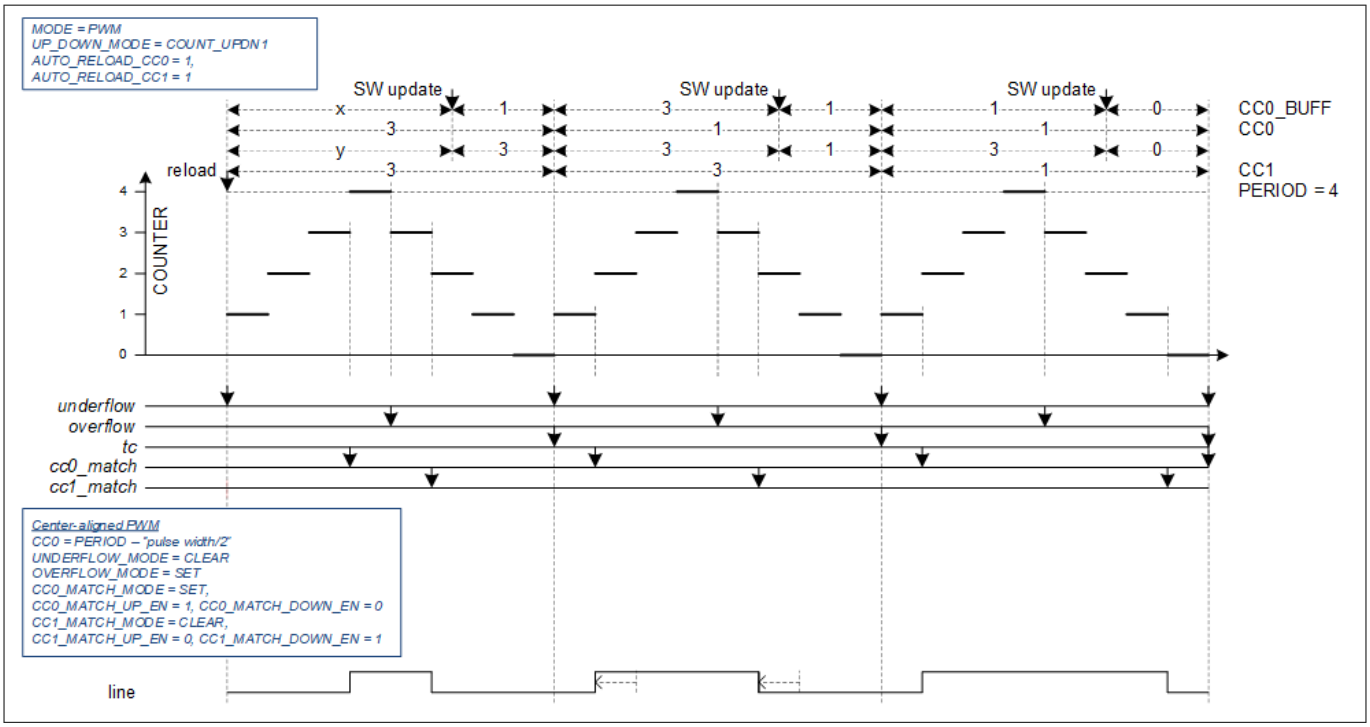


Figure 110 Asymmetric PWM by cc0_match and cc1_match in COUNT_UPDN1 mode

Instead of always using a center-aligned PWM, the phase of the PWM signal can be temporarily shifted to allow single-shunt current measurements. These measurements are performed for motor control of a permanent-magnet synchronous motor (PMSM) during situations where the current duty-cycle values of the three phases do not provide a sufficient measurement window (for example, when the window where one PWM channel is active while the other two are inactive is too small; for details, refer to memo CHRH-36). Compared to creating an asymmetric PWM with a single compare function in the COUNT_UPDN2 mode, this solution uses two independent buffered compare values. It reduces CPU load due to fewer required interrupts, as all updates can be performed within the "ADC done" interrupt service routine. In this routine, the new duty-cycle values are calculated, and a temporary phase shift is introduced for single-shunt current measurement.

Ways to enable asymmetric PWM using the second compare function:

- **COUNT_UP/DOWN mode**:
  - cc0_match = SET, cc1_match = CLEAR
  - Pulse width = CC1 - CC0
- **COUNT_UPDN1/2 mode**:
  - 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 = 2 × PERIOD – CC1 – CC0

22.3.3.2 Configuring counter for PWM mode

The steps to configure the counter for PWM mode operation and the corresponding affected register bits are as follows:

1. Disable the counter by writing `0` to the `ENABLE` bit of the `CTRL` register
2. Select PWM mode by writing `100` to the `MODE[26:24]` field of the `CTRL` register
3. Set the required 16-bit/32-bit period in the `PERIOD` register and the buffered period value in the `PERIOD_BUFF` register for value switching, if needed
4. Set the 16-bit/32-bit compare value in the `CC0`/`CC1` register and the buffered compare value in the `CC0_BUFF`/`CC1_BUFF` registers for value switching, if needed
5. Set the counting direction by writing to the `UP_DOWN_MODE[17:16]` field of the `CTRL` register to configure left-aligned, right-aligned, or center-aligned PWM
6. Configure the `PWM_IMM_KILL`, `PWM_STOP_ON_KILL`, and `PWM_SYNC_KILL` fields of the `CTRL` register as required
7. Configure the `TR_IN_SEL0` or `TR_IN_SEL1` register to select the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
8. Configure the `TR_IN_EDGE_SEL` register to select the edge of the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
9. Control `LINE_OUT` and `LINE_COMPL_OUT` using the `TR_PWM_CTRL` register to set, reset, or invert the signals upon `cc0_match`, `cc1_match`, overflow, or underflow conditions
10. If needed, configure an interrupt on a `TC`, `CC0`, or `CC1` condition
11. Enable the counter by writing `1` to the `ENABLE` bit of the `CTRL` register
12. If the hardware reload signal is not enabled, provide a reload trigger through firmware by setting the `RELOAD` bit in the `TR_CMD` register to start the counter

22.3.4 Pulse-width modulation with Dead-Time (PWM_DT) mode

PWM_DT functionality is similar to PWM functionality, with the following key differences:

- **Dead-time insertion**: PWM_DT supports dead-time insertion, whereas PWM does not support dead-time insertion
- **Clock pre-scaling**: PWM_DT does not support clock pre-scaling, while PWM supports clock pre-scaling
- **Dead-time operation**: Dead-time insertion operates on a preliminary PWM output signal, referred to as `line`

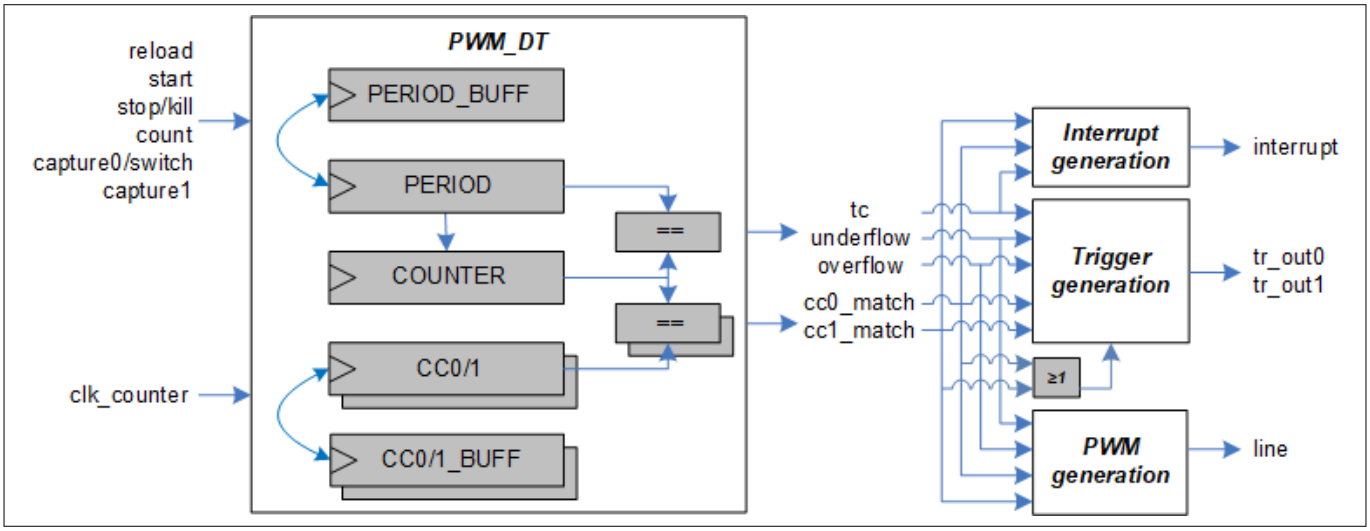


Figure 111 PWM with dead-time functionality

The dead-time generation depends on the design configuration option `GRP_AMC_PRESENT`.

- **When `GRP_AMC_PRESENT = 0`:**
  - The dead time is an 8-bit value (maximum dead time: 255 counter cycles)
  - The same dead time is used before activating both PWM output signals `line_out` and `line_compl_out`. This is specified by the 8-bit register field `DT_LINE_OUT_L`
- **When `GRP_AMC_PRESENT = 1`;**
  - The dead time is a 16-bit value (maximum dead time: 65,535 counter cycles)
  - Different dead times can be applied before activating the 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` (composed 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**: Users must ensure that the dead-time value does not exceed the period value. Failure to do so may lead to undefined behavior.*

Figure 112 shows the dead-time insertion for different dead times and varying output signal polarity settings when `GRP_AMC_PRESENT = 0`.

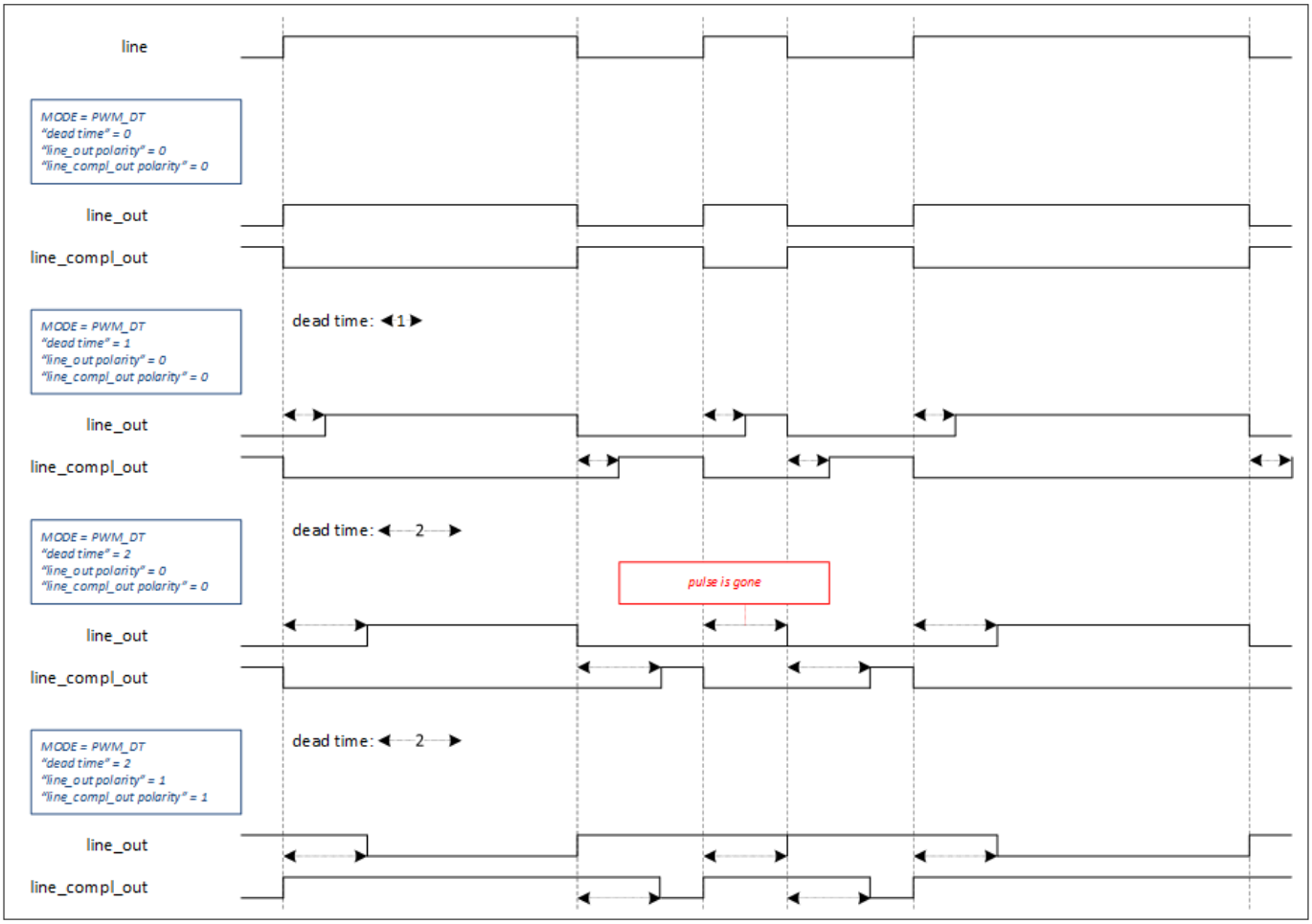


Figure 112 "line_out" and "line_compl_out" with unique dead-time

Figure 113 shows how the polarity settings and stop/kill functionality combine to control the PWM output signals `line_out` and `line_compl_out`.

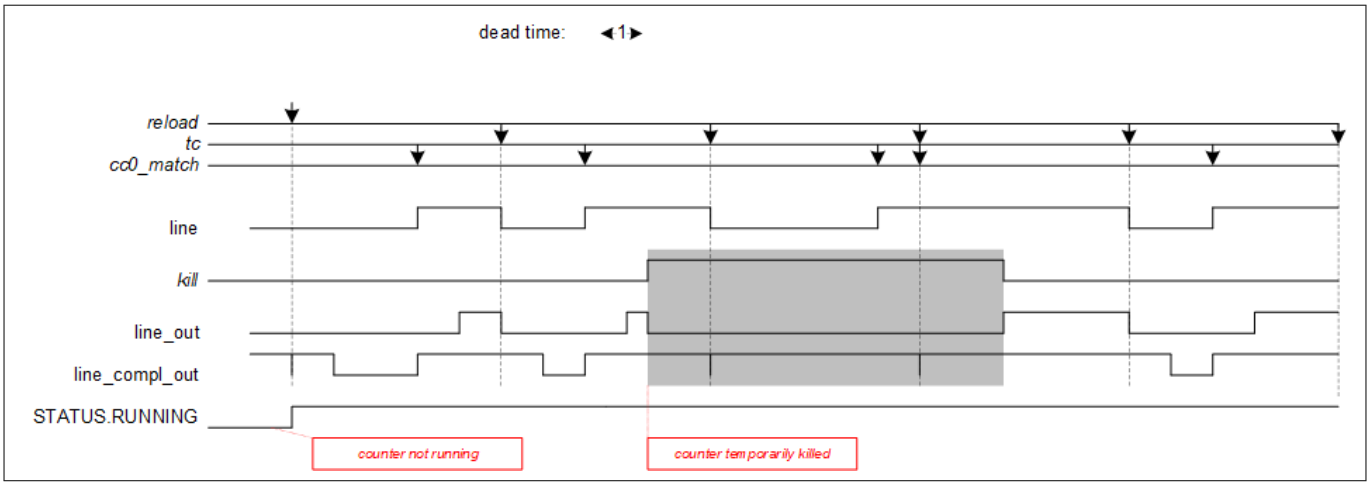


Figure 113 Dead-time, kill input, and polarity function

Figure 114 shows an example of different dead times applied to the PWM output signals `line_out` and `line_compl_out` when `GRP_AMC_PRESENT = 1` and `CTRL.SWAP_ENABLE = 1`.

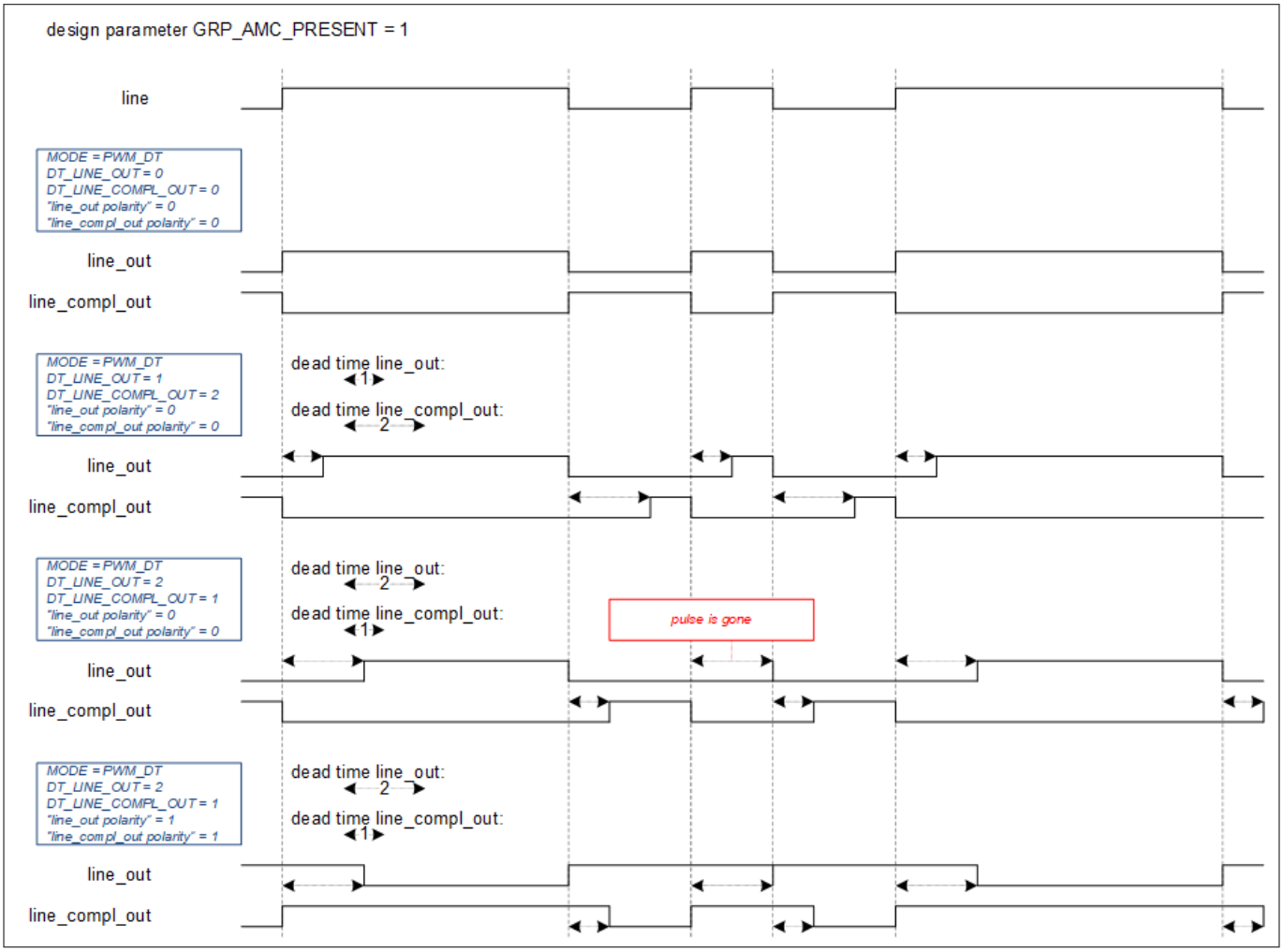


Figure 114 “line_out” and “line_compl_out” with different dead-time

Figure 115 shows an example of driving the two coils of a stepper motor using two TCPWM counters when `CTRL.SWAP_ENABLE = 0`. It illustrates how the software can update the registers for stepper motor control to incorporate dead times for the PWM output signals `line_out` and `line_compl_out`.

- The software can update the dead time in the DT_BUFF register, and this value will transfer to the DT register during the tc event
- In COUNT_UPDN2 mode, the shadow transfer occurs at the period match where the tc and underflow events coincide

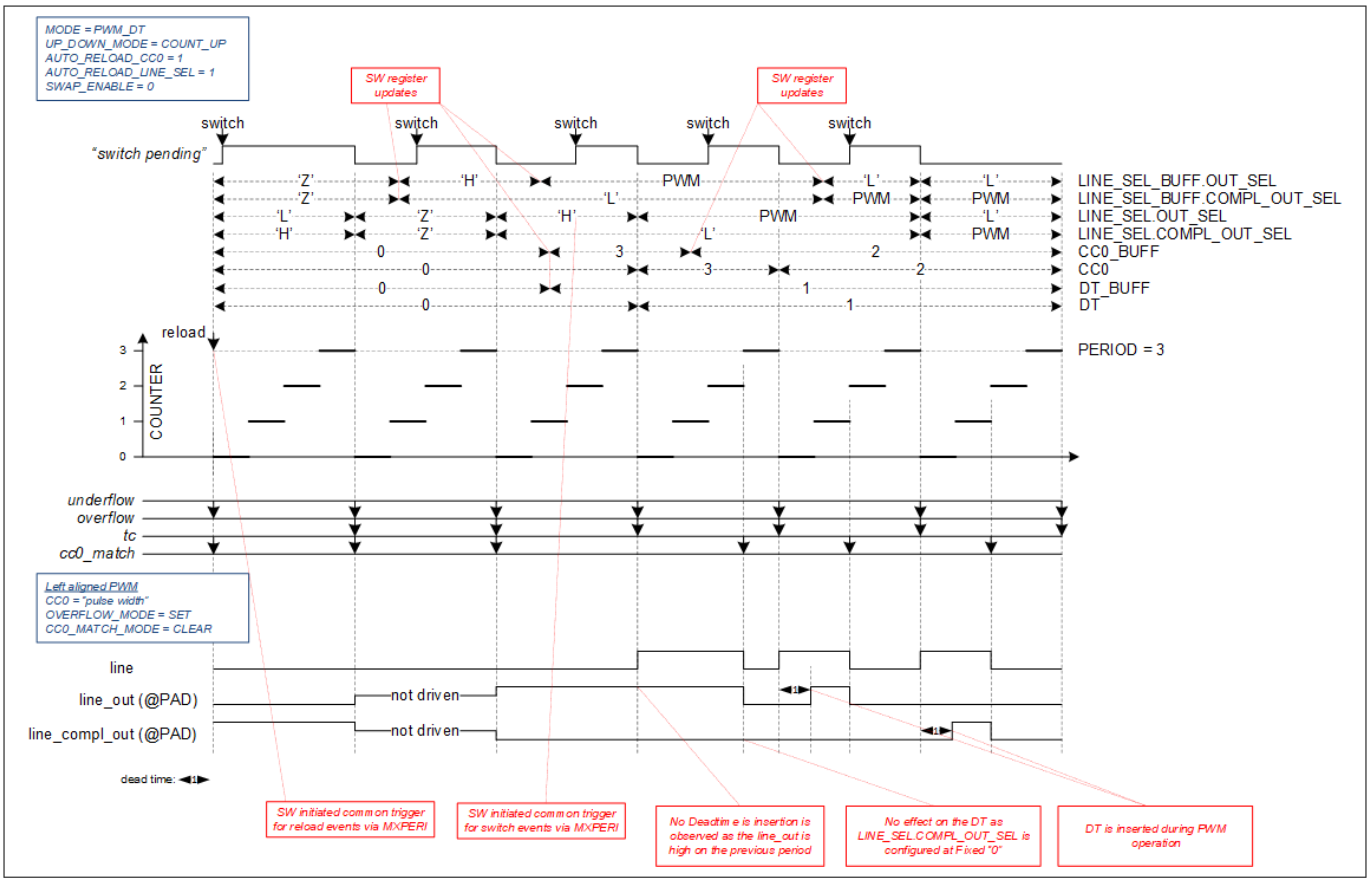


Figure 115 PWM_DT Output Case 1 Scenario (SWAP_ENABLE=0 Case)

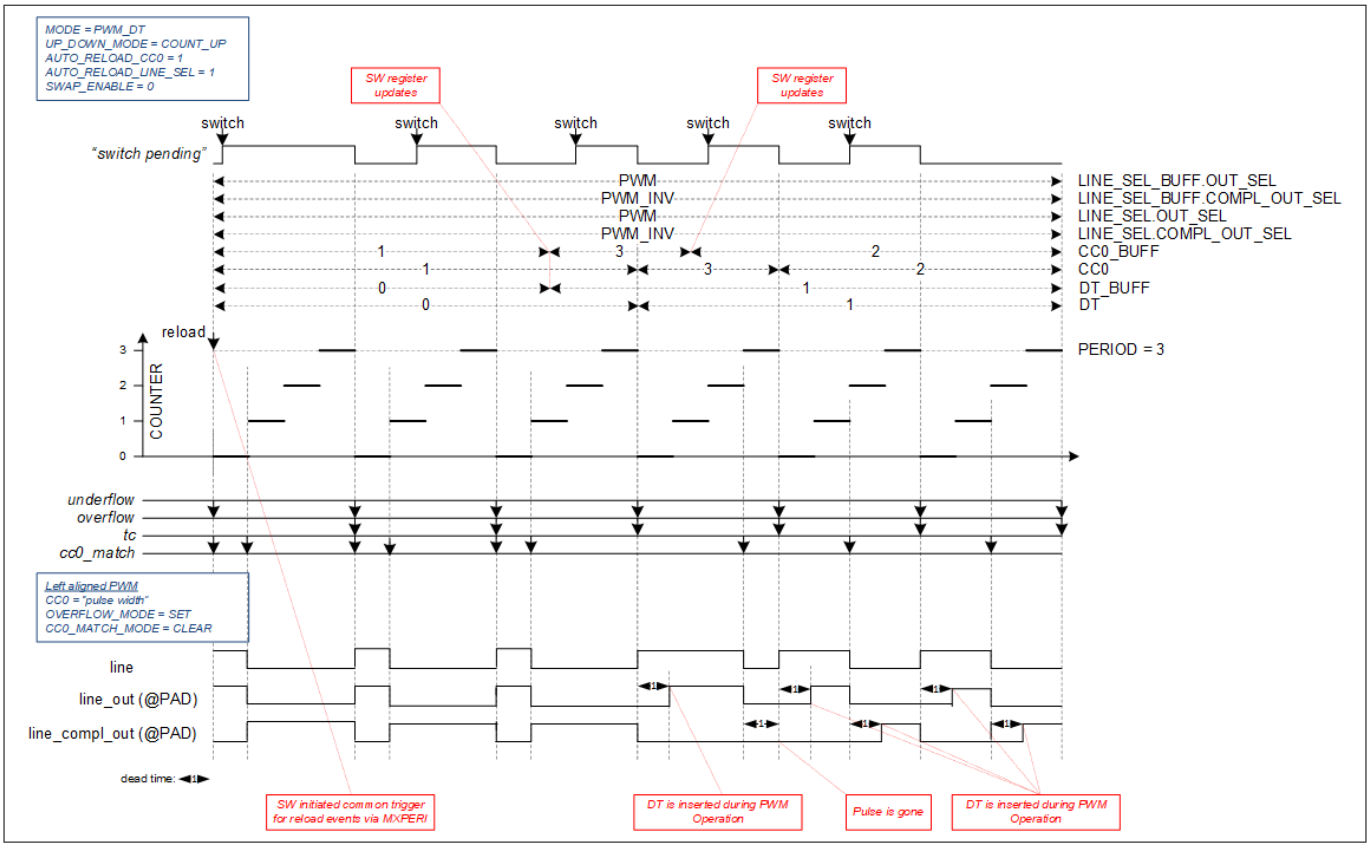


Figure 116 PWM_DT Output Case 2 Scenario (SWAP_ENABLE=0 Case)

22.3.4.1 Configuring counter for PWM with Dead-Time mode

The steps to configure the counter for PWM with Dead-Time mode of operation and the corresponding affected register bits are as follows:

1. Disable the counter by writing `0` to the `ENABLE` bit of the `CTRL` register
2. Select PWM with Dead-Time mode by writing `101` to the `MODE[26:24]` field of the `CTRL` register
3. 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
4. Set the required 16-bit/32-bit period in the `PERIOD` register and the buffered period value in the `PERIOD_BUFF` register for value switching, if needed
5. Set the 16-bit/32-bit compare value in the `CC0/CC1` register and the buffered compare value in the `CC0_BUFF/CC1_BUFF` register for value switching, if needed
6. Set the counting direction by writing to the `UP_DOWN_MODE[17:16]` field of the `CTRL` register to configure left-aligned, right-aligned, or center-aligned PWM
7. Configure the `PWM_IMM_KILL`, `PWM_STOP_ON_KILL`, and `PWM_SYNC_KILL` fields of the `CTRL` register as required
8. 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)
9. Configure the `TR_IN_EDGE_SEL` register to select the edge of the trigger that causes the event (Reload, Start, Stop, Capture0/1, and Count)
10. Control `LINE_OUT` and `LINE_COMPL_OUT` using the `TR_PWM_CTRL` register to set, reset, or invert the signals upon `cc0_match`, `cc1_match`, overflow, or underflow conditions

11. If required, configure an interrupt on a TC, CC0, or CC1 condition
12. Enable the counter by writing 1 to the ENABLE bit of the CTRL register. If the hardware reload signal is not enabled, provide a reload trigger through firmware by setting the RELOAD bit in the TR_CMD register to start the counter

22.3.5 Dithering (Pseudo-random mode)

The dithering feature is available in Group 1 and Group 2 counters and can be used in both PWM and PWM_DT modes. It is enabled through the CTRL.DITHEREN bit and can be applied independently to the period and duty cycle.

Dithering functionality overview

- **Period dithering**:
  - A pseudo-random number generated from the Linear Feedback Shift Register (LFSR) is added to or subtracted from the period value
  - A PWM-like signal is created by comparing the counter value with the period value
  - The generated signal has different frequency/noise characteristics compared to a regular PWM signal
- **Duty cycle dithering**:
  - A pseudo-random number generated from the LFSR is added to or subtracted from the CC0/CC1 values
  - The counter value compared with the CC0/CC1 values determines the duty cycle of the PWM-like signal
  - The signal has a random duty-cycle characteristic compared to a regular PWM signal

Dithering details

1. The value in the PLFSR/DLFSR register must be initialized to a non-zero seed value (set in LFSR.PLFSR/LFSR.DLFSR) by the software before enabling dithering through CTRL.DITHEREN
2. **LFSR polynomial**:
   - The PLFSR/DLFSR registers update their values based on an 8-bit LFSR taps polynomial
   - The LFSR is an 8-bit fixed-length design
3. **Event generation in dithering**:
   - When the counter value equals PERIOD, a tc event is generated. In COUNT_UPDN1/COUNT_UPDN2 modes, this event corresponds to a full PWM period (one overflow and one underflow)
   - When the counter value equals CC0, a cc0_match event is generated
   - When the counter value equals CC1, a cc1_match event is generated
4. The software can control the magnitude of the LFSR value to be added or subtracted by programming the LFSR.Limiter bits.
   - The most significant bit (MSB) of the PLFSR/DLFSR determines whether the generated value is added or subtracted from PERIOD, CC0, or CC1
   - With signed number representation, the randomness introduced to the period/duty cycle is in the range of -2^(N-1) to 2^(N-1)-1, where N is the LFSR length (set to 8 in the design)
5. When dithering is enabled through CTRL.DITHEREN, it is not applied to line_out and line_compl_out immediately. Instead:
   - Dithering is introduced only on a period match event
   - Dithering is disabled through a period match event at the end of the sequence

Note:

- *The LFSR register has a limiter option that allows users to control the magnitude of values added to PERIOD, CC0, and CC1*
- *The values of PERIOD, CC0, and CC1 must be larger than the limiter value*
- *The following constraint must be satisfied:*
  - *CC0 + MAX(Dither Value) < PERIOD + MIN(Dither Value)*
  - *This is controlled using the limiter*

22.3.6 Quadrature decoder control

The Quadrature Decoder mode is selected by setting PCONF.FSEL = 01B or PCONF.FSEL = 11B (in the latter case, the Multi-Channel mode is also enabled).

Within the Quadrature Decoder mode, two distinct subsets are available:

- Standard Quadrature mode
- Direction Count mode

The Standard Quadrature mode is used when the external rotary encoder provides two phase signals, along with an index/marker signal that is generated once per shaft revolution.

The Direction Count mode is used when the external encoder only provides a clock signal and a direction signal.

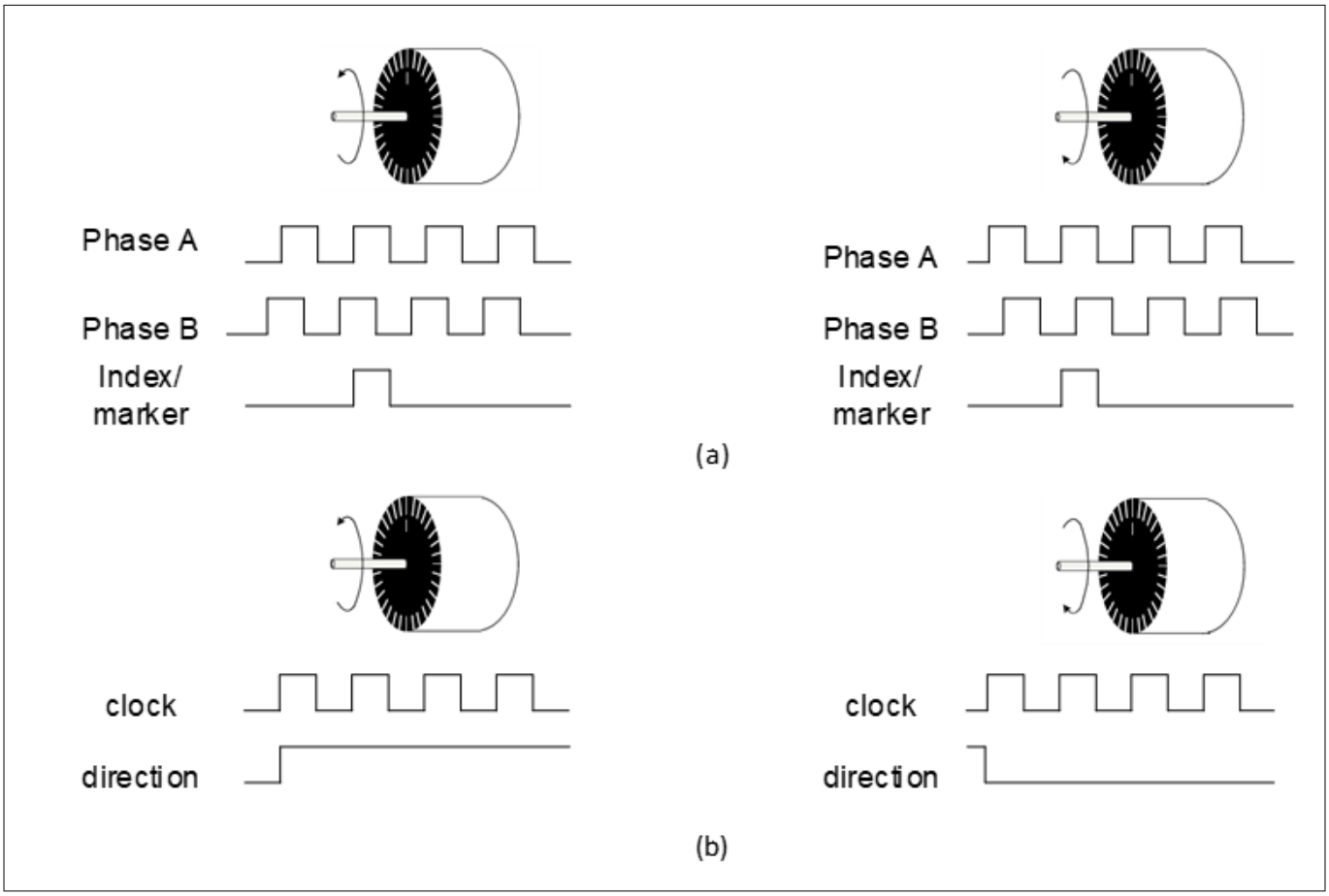


Figure 117 Rotary encoder types - a) Standard two phase plus index signal; b) Clock plus direction

Standard Quadrature mode: The quadrature decoder unit provides a highly flexible configuration stage for Phase A and Phase B signals. Typically, for a clockwise motor shaft rotation, Phase A should lead Phase B.

However, the user can configure which phase leads as well as the specific active state for each signal (see Figure 118).

The selection of Phase A, Phase B, and Index signals is performed using the QDC.PACS, QDC.PBDS, and QDC.IDS control fields.

The quadrature decoder control unit consists of two primary blocks:

- A block that decodes the quadrature clock and determines the motor shaft direction
- A block that manages the index (motor revolution) control

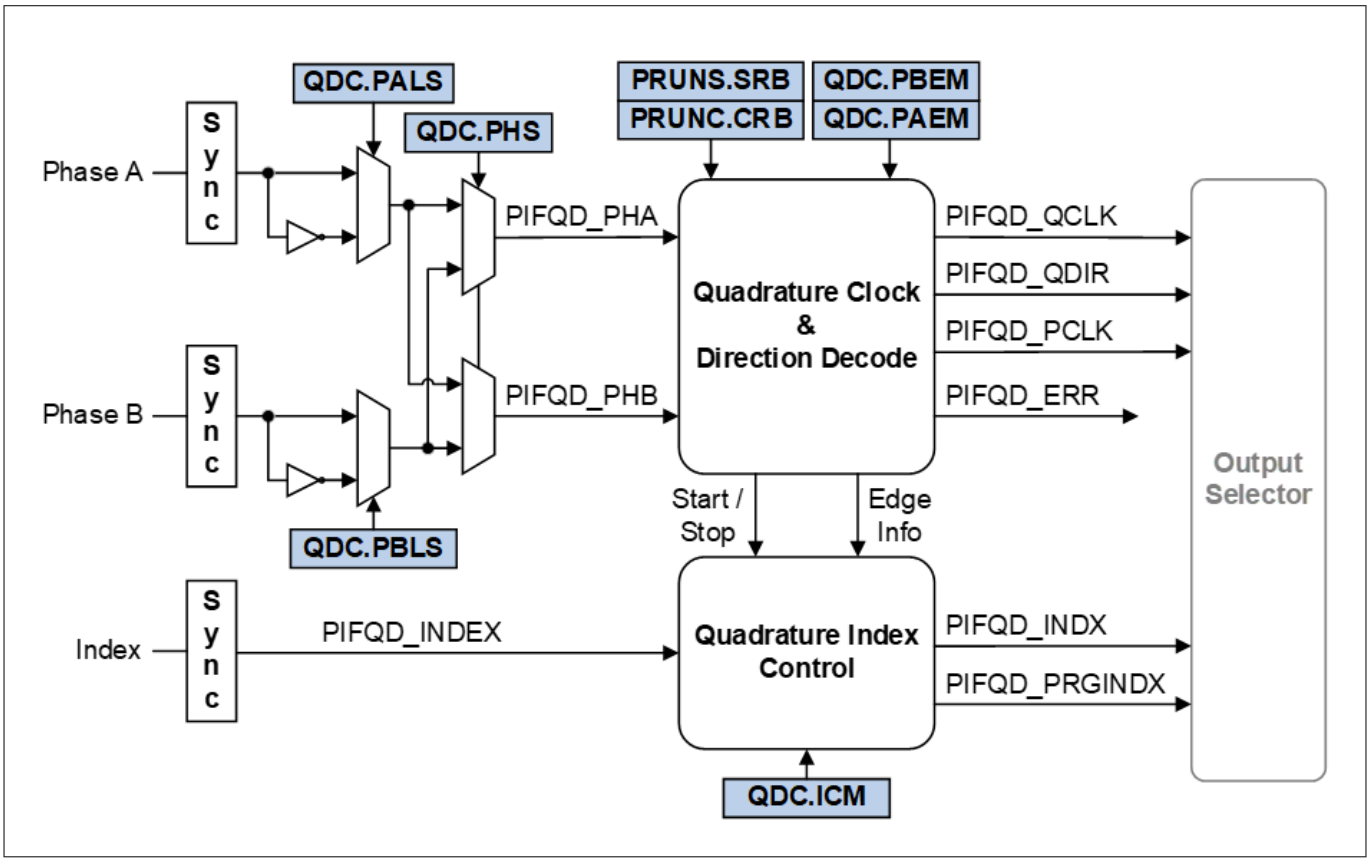


Figure 118 Quadrature decoder control overview

The quadrature clock (QCLK) signal is used for position measurement and can be forwarded to any of the MOTIFx.OUTy outputs. This clock signal is derived by decoding every edge of the phase signals, resulting in a maximum of four clock pulses per phase period.

The application software allows configuration of which specific edges such as rising, falling, both, or none of ${PhaseA} and ${PhaseB} signals are used to decode the quadrature clock (QCLK). See Figure 120 for examples of clock decoding behavior based on the Phase A and Phase B inputs. The edge selection is controlled using the QDC.PAEM and QDC.PBEM registers as follows:

```
QDC.PAEM: Configures edge selection for Phase A
QDC.PBEM: Configures edge selection for Phase B
```

Special case: When QDC.PHS is set to 1, the functions of QDC.PAEM and QDC.PBEM are swapped:

```
QDC.PAEM: Configures edge selection for Phase B
QDC.PBEM: Configures edge selection for Phase A
```

The following points summarize the configuration details:

- Use QDC.PAEM and QDC.PBEM to specify which edges such as rising, falling, both, or none should be used for decoding
- When QDC.PHS = 1, the roles of QDC.PAEM and QDC.PBEM are reversed
- The decoding process enables up to four clock pulses per phase period by detecting all edges of the signals

Figure 119 shows the X1 clock generation behavior.

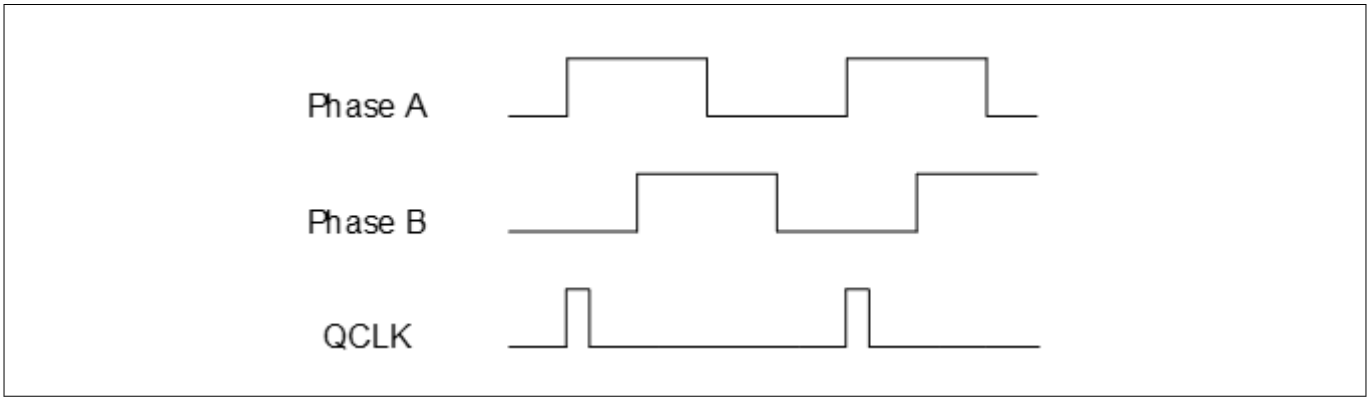


Figure 119 X1 clock generation PAEM=RISE/PBEM=NONE

Figure 120 shows the X2 clock generation behavior.

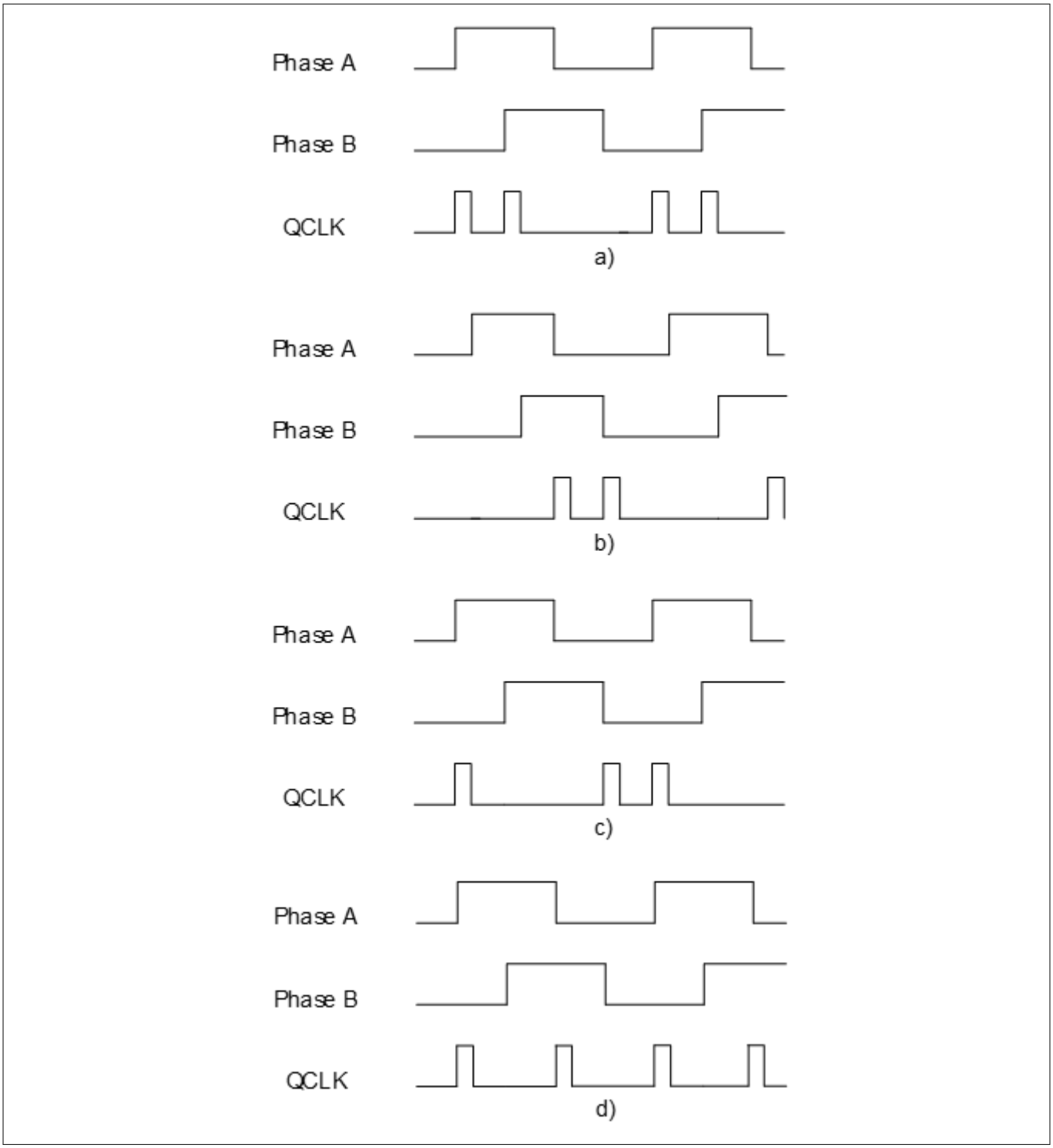


Figure 120 **X2 clock generation: a) PAEM=RISE/PBEM=RISE; b) PAEM=FALL/PBEM=FALL; c) PAEM=RISE/PBEM=FALL; d) PAEM=BOTH/PBEM=NONE**

Figure 121 shows the X4 clock generation behavior.

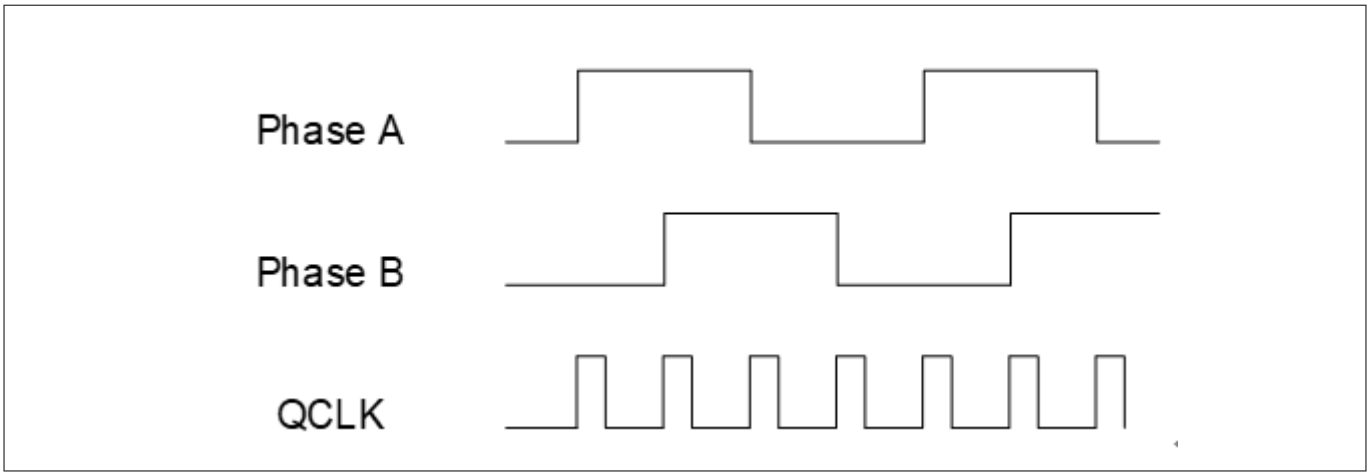


Figure 121 X4 clock generation PAEM=BOTH/PBEM=BOTH

A period clock (PCLK) signal is generated for velocity measurement operations, which can be forwarded to any of the MOTIFx.OUTy outputs. The decoding of this clock is influenced by the QDC.PAEM and QDC.PBEM fields.

The motor rotation direction signal (QDIR) is decoded from the Phase A and Phase B signals. This signal can also be forwarded to any of the MOTIFx.OUTy outputs. The QDIR signal is asserted HIGH when the motor rotates clockwise and LOW when it rotates counterclockwise.

The index control logic memorizes the first edge after the assertion of the index signal. This ensures that the same quadrature transition is used for index event operations. Additionally, it records the direction of the first index, allowing it to control when the revolution increment signal should be asserted. Two types of index events can be forwarded to the MOTIFx.OUTy outputs: INDX event and PRGINDX event.

For a detailed description of these events, see Index control.

An error signal, which is connected to a flag (and optionally capable of generating an interrupt), is triggered when an invalid phase pair is detected. However, this error event is only available as an interrupt and cannot be forwarded to the MOTIFx.OUTy outputs.

The quadrature decoder control uses information from the current and previous phase pairs to decode the direction and clocks. Both phase signals pass through edge-detection logic. The outputs are then evaluated against valid or invalid transition stages (see Figure 122).

The decoder state machine can be reset (excluding flags and retention mmio registers) by writing 1B into the PCONF.ENABLE field. The current state, previous state, and current index value of the quadrature decoder can be read via the PDBG register.

Key configuration and functionality details include:

- PCLK decoding is controlled by QDC.PAEM and QDC.PBEM
- QDIR indicates motor rotation direction and its state (HIGH for clockwise, LOW for counterclockwise)
- The first quadrature transition and direction after the index assertion are stored for index event operations
- The index-related events (INDX and PRGINDX) can be forwarded to MOTIFx.OUTy outputs, as explained in Index control
- Error detection generates an interrupt for invalid phase pairs but is not forwarded to outputs
- The state machine reset is performed via the PCONF.ENABLE register, while states and index values are readable through PDBG

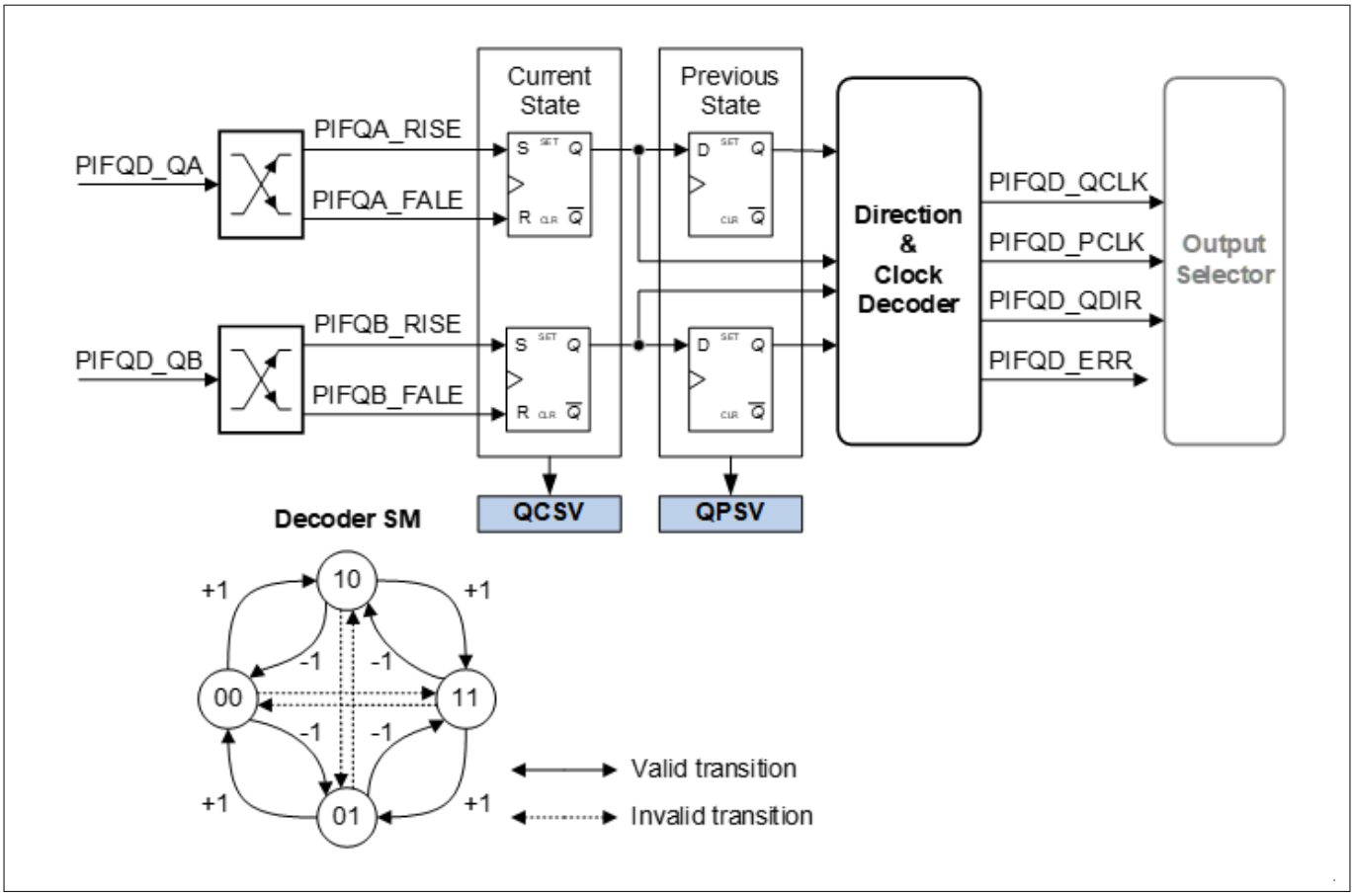


Figure 122 Quadrature decoder states

Direction Count mode

Some position encoders do not output phase signals. Instead, they provide two signals that include clock and direction information. This configuration is referred to as Direction Count mode.

When using this type of position encoder, the user must enable Direction Count mode by setting the PCONF.QDCM bit field to 1B. In this mode, the user selects the input group (MOTIFx.INy) to which the clock and direction signals are connected. This selection is configured using the QDC.PACS and QDC.PBDS control fields.

The input signals, clock and direction, are synchronized with the MOTIF module clock. If enabled, these signals are low pass filtered before being forwarded to any of the MOTIFx.OUTy outputs via the output selector.

Key considerations for Direction Count mode:

- The direction input signal is optional. It can be disabled by setting QDC.PBDS to 00B
- The output signals (PCLK, INDX, PRGINDX, and ERR) are set to a constant zero when operating in this mode

22.3.6.1 Quadrature clock and direction decoding

The quadrature decoder unit generates two clock signals. Each clock is used for specific purposes as described below:

1. **Position control clock (QCLK)**:
   - This clock is used for position control
   - It is generated on every edge transition of the Phase A and Phase B signals
2. **Velocity measurement clock (PCLK)**:
   - This clock is used for velocity measurements

- It is designed to be immune to potential glitches on the phase signals, which may occur at very slow rotation speeds and cause motor vibrations
- These motor vibration glitches are not necessarily caused by electrical noise. Instead, they are slow enough that input filters cannot de-glitch them. See Figure 124
- The PCLK is not generated when the motor changes direction (during a transition in QDIR). The only exception occurs during the motor's initial start-up, in which case PCLK is generated on the first transition of either the Phase A or Phase B signal

The direction signal (QDIR) is decoded according to the rules shown in Figure 123. This figure provides details of all valid transitions for Phase A and Phase B signals.

Figure 124 shows the key differences between QCLK and PCLK when glitches are present on the phase signals.

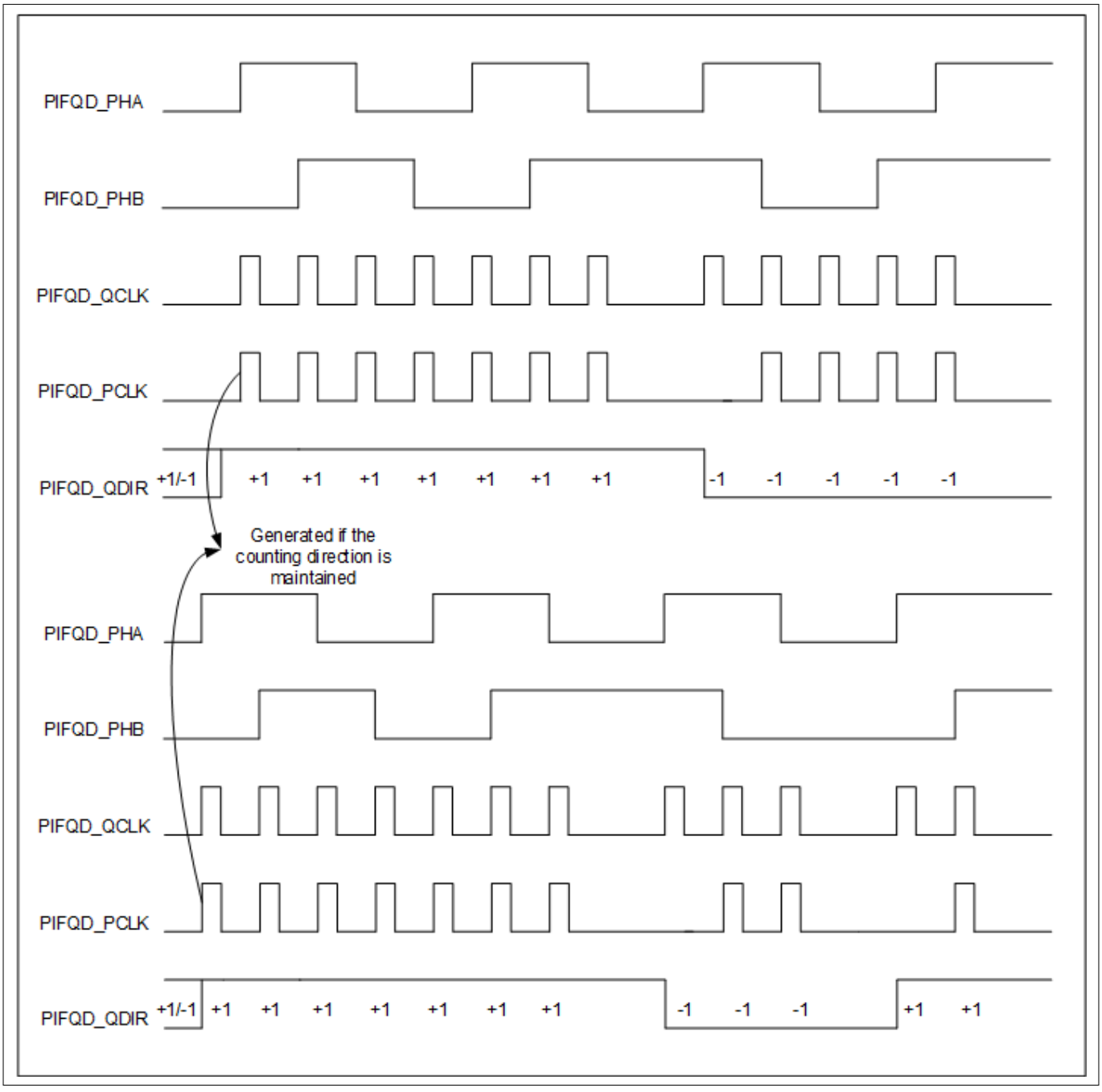


Figure 123 Quadrature clock and direction timing

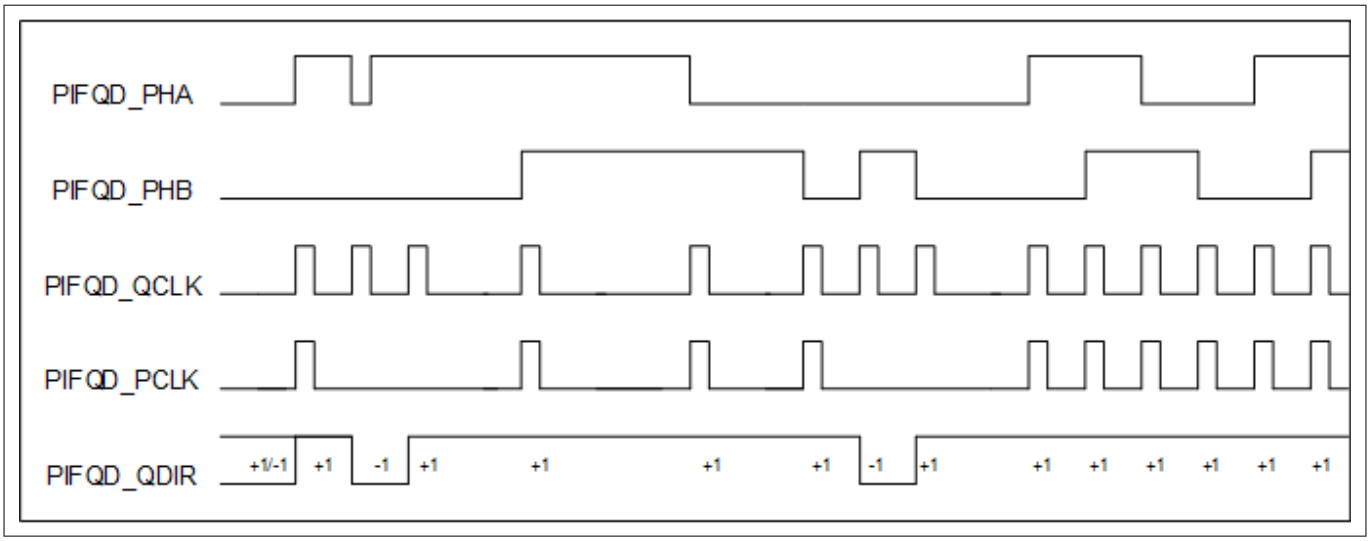


Figure 124 Quadrature clock with jitter

22.3.6.2 Index control

The index control logic provides two distinct output events as follows:

1. **INDX event**:
   - This event is asserted every time an index signal is detected, provided the motor shaft continues to rotate in the same direction
   - It can be used for revolution counting, allowing the software to monitor not only the shaft's position but also the total number of revolutions
2. **PRGINDX event**:
   - This event's activity is programmable via the `QDC.ICM` field
   - Based on the value configured in `QDC.ICM`, the corresponding signal can be generated:
     - On every occurrence of the index signal
     - Only on the first occurrence of the index signal
     - Disabled, where the output is never asserted
   - This functionality is particularly useful for applications that need to reset counters either on every index event or only on the first event

The index control logic records the first phase edge immediately following the index signal to ensure the generated signals are always aligned to the same reference. For example:

- If the rising edge of the Phase B signal is the first phase edge after the index, the subsequent index signals will also be referenced to the rising edge of the Phase B signal if the direction remains unchanged
- If the direction changes, the index signals will be referenced to the falling edge of the Phase B signal

***Note**: If a x2 decoder mode is used (for example, where only the rising edge of either Phase A or Phase B is used to generate the `QCLK`), the index decoding follows this same behavior. No pulses of the index signal are lost.*

See Figure 125 for the timing diagram illustrating the generation of index signals. In this case, `QDC.ICM = 10B`, meaning the `PRGINDX` event is generated on every occurrence of the input index signal.

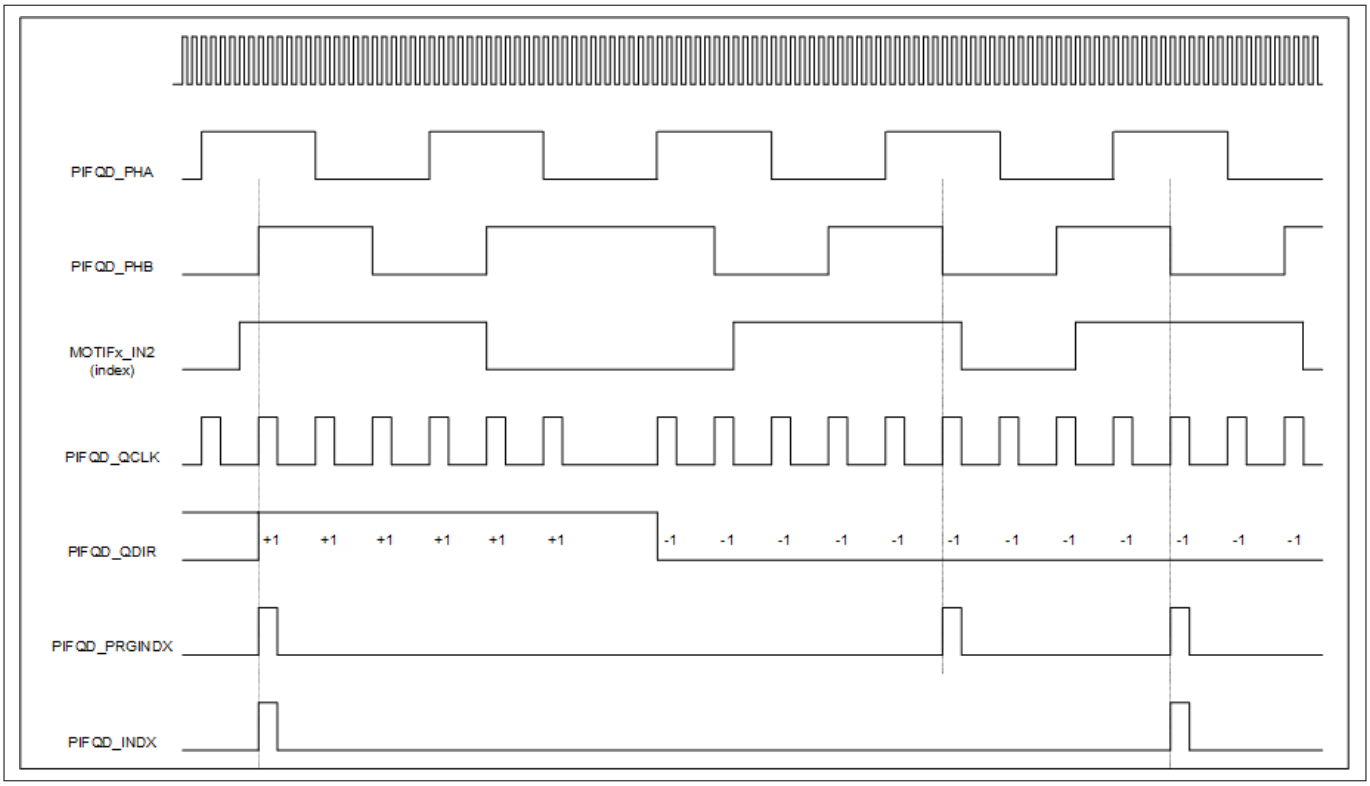


Figure 125 Index signal timing

22.4 Recovery

The TCPWM can be recovered using any active reset event. Examples of such events include:

- Power-on reset (POR)
- External reset (XRES)
- Watchdog timer reset (MCWDT and WDT)
- Brownout detection reset
- Over-voltage and over-current detection reset

The TCPWM is designed to prevent any unexpected or undefined state. It will always recover into a known, predictable state following an active reset event.

22.5 Initialize

The initial state of TCPWM pins is High-Z. The behavior of registers and pins under different conditions is described below:

- Some registers are reset during an active reset
- Certain MMIO registers are retained in Deep Sleep mode
- No registers are retained through Hibernate or other low-power modes

During an active reset, the pin state is restored to High-Z unless the `CTRL.PWM_DISABLE_MODE` field is configured with a non-zero value. The behavior in such a scenario varies based on the specific configuration settings.

22.6 Pin status

When the TCPWM module is unused, the status of the TCPWM pins remains High-Z.

To disable the TCPWM module, ensure the following:

- The ENABLED bit in the CTRL register is set to 0
- The PWM_DISABLE_MODE bit in the CTRL register is set to 0

22.7 Registers summary

Table 113 TCPWM registers summary

Name	Description
TCPWM0_GRPx_CNT0_CTRL	Counter Control register
TCPWM0_GRPx_CNT0_STATUS	Counter Status register
TCPWM0_GRPx_CNT0_COUNTER	Counter Count register
TCPWM0_GRPx_CNT0_CC0	Counter Compare/Capture 0 register
TCPWM0_GRPx_CNT0_CC0_BUFF	Counter Buffered Compare/Capture 0 register
TCPWM0_GRPx_CNT0_CC1	Counter Compare/Capture 1 register
TCPWM0_GRPx_CNT0_CC1_BUFF	Counter Buffered Compare/Capture 1 register
TCPWM0_GRPx_CNT0_PERIOD	Counter Period register
TCPWM0_GRPx_CNT0_PERIOD_BUFF	Counter Buffered Period register
TCPWM0_GRPx_CNT0_LINE_SEL	Counter Line Selection register
TCPWM0_GRPx_CNT0_LINE_SEL_BUFF	Counter Buffered Line Selection register
TCPWM0_GRPx_CNT0_DT	Counter PWM Dead-Time register
TCPWM0_GRPx_CNT0_DT_BUFF	Counter Buffered PWM Dead-Time register
TCPWM0_GRPx_CNT0_PS	Counter Prescalar register
TCPWM0_GRPx_CNT0_TR_CMD	Counter Trigger Command register
TCPWM0_GRPx_CNT0_TR_IN_SEL0	Counter Input Trigger Selection register 0
TCPWM0_GRPx_CNT0_TR_IN_SEL1	Counter Input Trigger Selection register 1
TCPWM0_GRPx_CNT0_TR_IN_EDGE_SEL	Counter Input Trigger Edge Selection register
TCPWM0_GRPx_CNT0_TR_PWM_CTRL	Counter Trigger PWM Control register
TCPWM0_GRPx_CNT0_TR_OUT_SEL	Counter Output Trigger Selection register
TCPWM0_GRPx_CNT0_INTR	Interrupt Request register
TCPWM0_GRPx_CNT0_INTR_SET	Interrupt Set Request register
TCPWM0_GRPx_CNT0_INTR_MASK	Interrupt Mask register
TCPWM0_GRPx_CNT0_INTR_MASKED	Interrupt Masked Request register
TCPWM0_GRPx_CNT0_TR_ONE_SYNC_BYPASS	Sync Bypass register for one to one trigger
TCPWM0_GRP0_CNT0_HRPWM_CTRL	Counter Control register for HRPWM feature
TCPWM0_TR_ALL_SYNC_BYPASS_TR_ALL_SYNC_BYPASS0	Trigger Sync Bypass for group trigger
TCPWM0_MOTIF_GRP1_MOTIF0_PCONF	Global Control register

(table continues...)

Table 113 (continued) TCPWM registers summary

Name	Description
TCPWM0_MOTIF_GRP1_MOTIF0_PSUS	Suspend Configuration
TCPWM0_MOTIF_GRP1_MOTIF0_HIST	Hall Inputs Sample Trigger
TCPWM0_MOTIF_GRP1_MOTIF0_HMEC	Hall Mode Extra Config
TCPWM0_MOTIF_GRP1_MOTIF0_HALP	Hall Current and Expected patterns
TCPWM0_MOTIF_GRP1_MOTIF0_HALPS	Hall Current and Expected shadow patterns
TCPWM0_MOTIF_GRP1_MOTIF0_HOSC	Hall Sensor Output Config
TCPWM0_MOTIF_GRP1_MOTIF0_MCM	Multi-channel Mode Pattern
TCPWM0_MOTIF_GRP1_MOTIF0_MCSM	Multi-channel Mode Shadow Pattern LUT0
TCPWM0_MOTIF_GRP1_MOTIF0_MCMS	Multi-channel Mode Control Set
TCPWM0_MOTIF_GRP1_MOTIF0_MCMC	Multi-channel Mode Control Clear
TCPWM0_MOTIF_GRP1_MOTIF0_MCMF	Multi-channel Mode Flag Status
TCPWM0_MOTIF_GRP1_MOTIF0_MCPF	Multi-channel Pattern Fault
TCPWM0_MOTIF_GRP1_MOTIF0_MOSC	Multi-channel Output Config
TCPWM0_MOTIF_GRP1_MOTIF0_QDC	Quadrature Decoder Configuration
TCPWM0_MOTIF_GRP1_MOTIF0_QOSC	Quadrature Output Config
TCPWM0_MOTIF_GRP1_MOTIF0_MCMEC	Multi-channel Extra Config
TCPWM0_MOTIF_GRP1_MOTIF0_SPFLG	Interrupt Set register
TCPWM0_MOTIF_GRP1_MOTIF0_RPFLG	Interrupt Clear register
TCPWM0_MOTIF_GRP1_MOTIF0_MCSM1	Multi-channel Mode shadow pattern LUT1
TCPWM0_MOTIF_GRP1_MOTIF0_MCSM2	Multi-channel Mode shadow pattern LUT2
TCPWM0_MOTIF_GRP1_MOTIF0_MCSM3	Multi-channel Mode shadow pattern LUT3
TCPWM0_MOTIF_GRP1_MOTIF0_MCSM4	Multi-channel Mode shadow pattern LUT4
TCPWM0_MOTIF_GRP1_MOTIF0_MCSM5	Multi-channel Mode shadow pattern LUT5
TCPWM0_MOTIF_GRP1_MOTIF0_CLUT	Hall Mode LUT Config
TCPWM0_MOTIF_GRP1_MOTIF0_SLUT	Hall Mode LUT Status

***Note**: In TCPWM0_GRPx, the letter x represents the group number.*

Overwriting the same value on registers can lead to different effects, depending on the software access attributes. These effects are detailed in the register map. The access restrictions for TCPWM registers are as follows:

- All status registers are read-only and not writable by software
- The TR_CMD register is written by software and cleared by hardware
- The INTR register is cleared by software and set by hardware by writing 1 to the INTR_SET register
- Reading the INTR_SET register returns the value stored in the INTR register
- Other registers behave as normal and can be overwritten with the same value without special behavior

23 Controller area network flexible data-rate (CAN FD)

The CAN FD controller complies with ISO 11898-1 (CAN specification Rev. 2.0, parts A and B). Additionally, 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:

1. The Rx handler manages message acceptance filtering, transfers received messages from the CAN core to the message RAM, and provides receive message status information
2. The Tx handler transfers transmit messages from the message RAM to the CAN core and provides transmit status information

The CAN FD controller has two separate clock inputs:

1. CAN clock (`PCLK_CANFD[x]_CLOCK_CAN[y]`) is used for CAN operation
2. System clock (`CLK_SYS`) is used for internal block operation

Acceptance filtering is implemented using up to 192 filter elements, each of which can be configured in one of the following ways:

1. As a range
2. As a bit mask
3. As a dedicated ID filter

The CAN FD controller is functional only in Active and Sleep power modes. In Deep Sleep mode, the controller is not functional but remains fully retained except for the shared time stamp (TS) counter. In Hibernate power mode, the controller is neither functional nor retained.

23.1 Features

The CAN FD controller provides the following features:

- Flexible data rate (FD)
- Supports up to 64 data bytes per message
- Maximum data rate of 8 Mbps
- Time-Triggered (TT) communication on CAN (ISO 11898-4: 2004)
- Supports TTCAN protocol levels 1 and 2, entirely implemented in hardware
- AUTOSAR support
- Acceptance filtering
- Two configurable receive FIFOs
- Up to 64 dedicated receive buffers
- Up to 32 dedicated transmit buffers
- Configurable transmit FIFO
- Configurable transmit queue
- Configurable transmit event FIFO
- Programmable loop-back test mode
- Power-down mode support
- A shared message RAM
- ECC protection for the message RAM
- Global fault structure to handle ECC errors
- Receive FIFO top pointer logic
- Enables DMA access for the FIFO operations

- DMA support for debugging messages and received FIFOs
- Shared time-stamp counter

***Note**: See the device datasheet for specific details regarding the number of M_TTCAN groups, M_TTCAN channels within each group, and the total message RAM allocated to each group.*

23.2 Architecture

The M_TTCAN operation is available in Active and Sleep power modes. In Deep Sleep power mode, the IP is fully retained except for the time-stamp counter.

Figure 126 shows the major blocks in CAN FD.

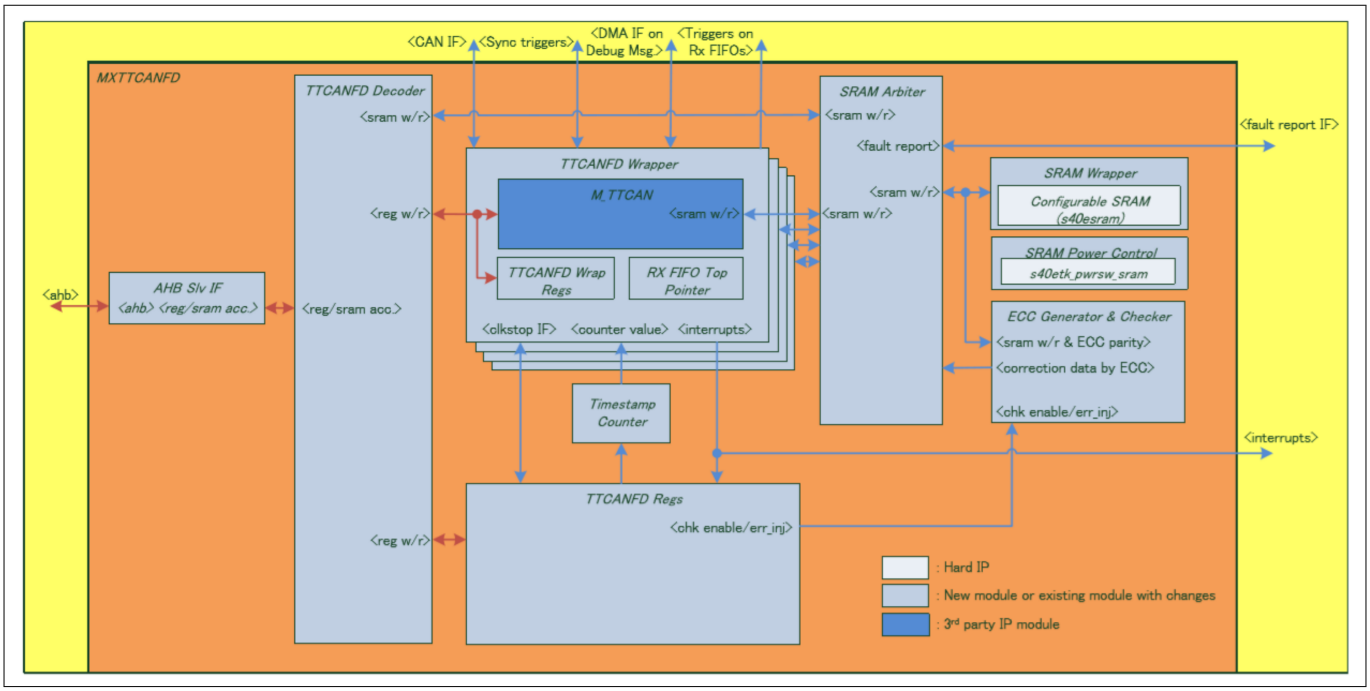


Figure 126 CAN FD block diagram

23.2.1 Dual clock sources

The TTCAN channel has two clock inputs: clk_can and clk_sys.

- The clk_can is derived from the system peripheral clock dividers and is used for CAN (or CAN FD) operation. It is recommended to use frequencies of 20 MHz, 40 MHz, or 80 MHz
- The clk_sys is used for all operations except CAN. For example, it is used for register access, SRAM access, and other internal functions. The clk_sys must run at a frequency equal to or greater than the clk_can

23.2.2 Interrupt line

The M_TTCAN channel provides two interrupt lines: interrupt0 and interrupt1.

- Interrupts from any source within the M_TTCAN channel can be routed to either interrupt0 or interrupt1
- By default, all interrupts are routed to interrupt0
- The interrupt lines can be enabled or disabled independently for each interrupt source by programming the Enable Interrupt Line 0 (ILE.EINT0) and Enable Interrupt Line 1 (ILE.EINT1) registers

23.3 Functional description

23.3.1 Modes of operation

23.3.1.1 Normal operation

The default operating mode of the M_TTCAN after a hardware reset is event-driven CAN communication without time triggers (TTOCF.OM = 00). Before the TT operation mode can be changed, both CCCR.INIT and CCCR.CCE must be set.

When M_TTCAN is initialized and CCCR.INIT is cleared to 0, M_TTCAN synchronizes with the CAN bus and becomes ready for communication. After messages pass the acceptance filtering process, they are stored in one of the following locations:

- A dedicated Rx buffer
- Rx FIFO 0
- Rx FIFO 1

For transmission, dedicated Tx buffers and a Tx FIFO or Tx queue can be initialized or updated.

***Note**: Automated transmission upon receiving remote frames is not implemented.*

23.3.1.2 CAN FD operation

The two variants of CAN FD frame transmission are:

1. CAN FD frame without bit rate switching
2. CAN FD frame where the control, data, and CRC fields are transmitted at a higher bit rate than the beginning and end of the frame

The previously reserved bit in CAN frames with 11-bit and 29-bit identifiers is now decoded as an 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) determine whether the bit rate inside the CAN FD frame is switched:

- A CAN FD bit rate switch is signified when RES is dominant and BRS is recessive
- RES coded as recessive is reserved for future expansion of the protocol

If the M_TTCAN receives a frame with FDF and RES coded as recessive, it signals a protocol exception event by setting the PSR.PXE bit. When protocol exception handling is enabled (CCCR.PXHD = 0), the operation state changes from receiver (PSR.ACT = 10) to integrating (PSR.ACT = 00) at the next sample point. If protocol exception handling is disabled (CCCR.PXHD = 1), the M_TTCAN treats a recessive RES bit as a form error and responds with an error frame.

CAN FD operation is enabled by programming CCCR.FDOE.

- If CCCR.FDOE = 1, transmission and reception of CAN FD frames are enabled
- Transmission and reception of classic CAN frames are always possible
- The frame type (CAN FD or classic CAN) for transmission can be configured via the FDF bit in the respective Tx buffer element
- If CCCR.FDOE = 0, frames are interpreted as classic CAN frames. Receiving a CAN FD frame triggers the transmission of an error frame

When CAN FD operation is disabled:

- No CAN FD frames are transmitted, even if the FDF bit in a Tx buffer element is set
- The settings for FDF and BRS are ignored, and frames are transmitted in classic CAN format

CCCR.FDOE and CCCR.BRSE can only be changed while both CCCR.INIT and CCCR.CCE are set.

- When CCCR.FDOE = 1 and CCCR.BRSE = 0, only the FDF bit in a Tx buffer element is evaluated
- When CCCR.FDOE = 1 and CCCR.BRSE = 1, CAN FD frames with bit rate switching are enabled, and all Tx buffer elements with both FDF and BRS bits set are transmitted in CAN FD format with bit rate switching

A mode change during CAN operation is recommended only under the following conditions:

1. **Higher failure rate in data phase**: If the failure rate in the CAN FD data phase is significantly higher than in the arbitration phase, disable the CAN FD bit rate switching option for transmissions
2. **System startup**: During startup, all nodes transmit classic CAN messages until it is confirmed that all nodes can communicate in CAN FD format. Once verified, all nodes switch to CAN FD operation
3. **Wakeup messages in partial networking**: Wakeup messages in CAN partial networking should always be transmitted in classic CAN format
4. **End-of-line programming**: For end-of-line programming involving non-CAN FD-capable nodes, these nodes are kept in Silent mode until programming is completed. Afterward, all nodes switch back to classic CAN communication

In the CAN FD format, the coding of the data length code (DLC) differs from that in the standard CAN format:

- DLC codes 0 to 8 are coded the same as in standard CAN
- DLC codes 9 to 15, which map to an 8-byte data field in standard CAN, are coded according to values shown in Table 114

Table 114 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 CAN FD frames, the bit timing switches inside the frame after the BRS bit, provided the BRS bit is recessive.

Before the BRS bit, during the CAN FD arbitration phase, nominal CAN bit timing is applied. This timing is defined by the Nominal Bit Timing and Pre-scaler (NBTP) register. Once the BRS bit is reached and identified as recessive, the CAN FD data phase begins. During this phase, data-phase bit timing is used, as defined by the Data Bit Timing and Pre-scaler Register (DBTP).

The bit timing switches back to nominal bit timing under one of the following conditions:

- At the CRC delimiter
- 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 time quanta (tq), the data-phase bit rate is 5 Mbps.

In both CAN FD frame formats, CAN FD and CAN FD with bit rate switching, the value of the error status indicator (ESI) bit depends on the transmitter's error state at the start of the transmission:

- If the transmitter is in error-passive mode, ESI is transmitted as recessive
- If the transmitter is not in error-passive mode, ESI is transmitted as dominant

23.3.1.2.1 Transmitter delay compensation

During the data phase of a CAN FD transmission, only one node functions as the transmitter, while all others act as receivers. The bus line length has no impact on this behavior. When transmitting data via the pin CANx_y_TX, the M_TTCAN receives the transmitted data through its local CAN transceiver via the pin CANx_y_RX. The received

data is delayed by the transmitter delay. If this delay exceeds TSEG1 (the time segment before the sample point), a bit error is detected.

To enable a data-phase bit time shorter than the transmitter delay, a delay compensation mechanism is introduced. Without transmitter delay compensation, the achievable bit rate in the data phase of a CAN FD frame is restricted by the transmitter delay.

The M_TTCAN protocol unit implements a delay compensation mechanism to counteract transmitter delay. This feature enables higher bit rates during the CAN FD data phase regardless of the delay introduced by a specific CAN transceiver.

To check for bit errors in the data phase of the transmitting node, 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 to this error at the next regular sample point. Delay compensation is disabled during the arbitration phase.

The transmitter delay compensation mechanism allows configurations where the data bit time is shorter than the transmitter delay. This functionality is described in ISO 11898-1:2015 and can be enabled by setting the bit CANFDx_CHy_DBTP.TDC.

The position of the SSP is calculated as the sum of:

- The measured delay from the M_TTCAN transmit output CANx_y_TX through the transceiver to the receive input CANx_y_RX
- The transmitter delay compensation offset configured by CANFDx_CHy_TDCR.TDCO

The transmitter delay compensation offset is used to adjust the SSP position within the received bit, such as placing it at half the bit time during the data phase. The position of the SSP is rounded down to the nearest integer multiple of mtq (the PCLK_CANFD[x]_CLOCK_CAN[y] period).

- The actual transmitter delay compensation value is shown in CANFDx_CHy_PSR.TDCV
- This value is cleared when CANFDx_CHy_CCCR.INIT is set and is recalculated during each transmission of a CAN FD frame while CANFDx_CHy_DBTP.TDC is 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 CANx_y_TX to CANx_y_RX and the configured transmitter delay compensation offset (CANFDx_CHy_TDCR.TDCO) must be less than 6 bit times in the data phase
- The sum of the measured delay from CANx_y_TX to CANx_y_RX and the configured transmitter delay compensation offset should be less than or equal to 127 mtq. If this sum exceeds 127 mtq, the maximum value of 127 mtq is used for delay compensation
- The data phase ends at the sample point of the CRC delimiter, which marks the termination of receive-bit checking at the SSP

Transmitter delay compensation measurement

If transmitter delay compensation is enabled by programming CANFDx_CHy_DBTP.TDC = 1, the measurement begins during the transmission of each CAN FD frame at the falling edge from the FDF bit to the res bit. The measurement stops when this edge is detected at the receive input CANx_y_RX of the transmitter. The measurement resolution is one mtq (minimum time quanta).

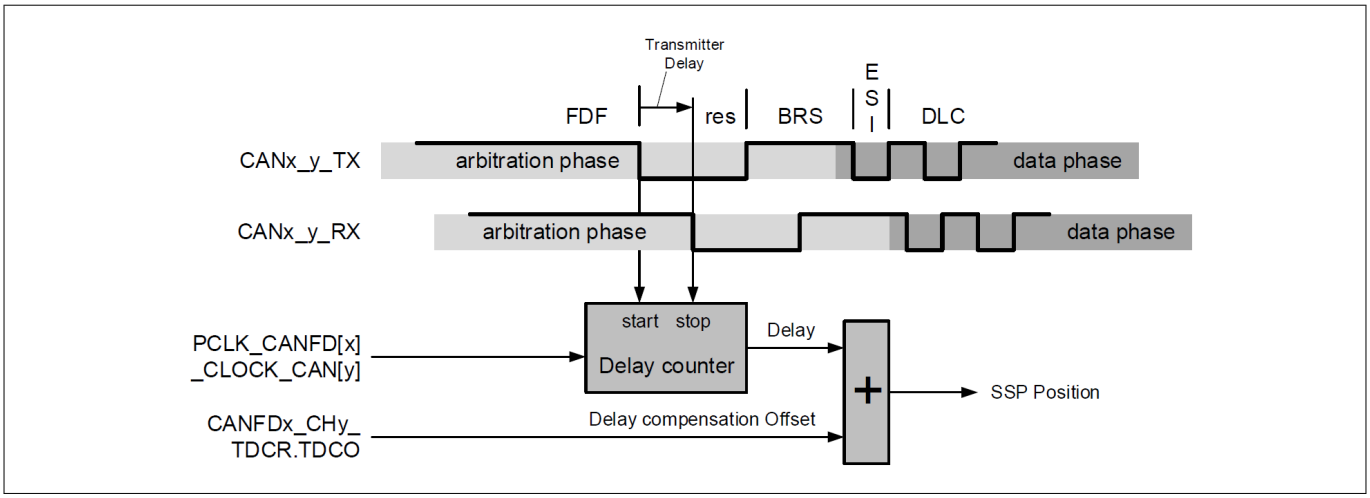


Figure 127 Transmitter delay measurement

A dominant glitch within the received FDF bit can terminate the delay compensation measurement prematurely. This may occur before the falling edge of the received RES bit, resulting in an early SSP position.

To prevent this issue, a transmitter delay compensation filter window can be enabled by programming `CANFDx_CHy_TDCR.TDCF`. This filter defines a minimum value for the SSP position. Dominant edges on `CANx_y_RX`, which would otherwise cause an earlier SSP position, are ignored during transmitter delay measurement.

The measurement stops when the following conditions are met:

- The SSP position is at least the value configured in `CANFDx_CHy_TDCR.TDCF`
- `CANx_y_RX` is in a dominant state (low logic level)

23.3.1.3 Restricted operation mode

In restricted operation mode, the node can perform the following:

- Receive data frames and remote frames
- Acknowledge valid frames

However, in this mode, the node does not:

- Send data frames
- Send remote frames
- Send active error frames
- Send overload frames

In the case of an error or overload condition, the node does not transmit dominant bits. Instead, it waits for the bus to return to an idle condition in order to resynchronize with the CAN communication. The error counters (`CANFDx_CHy_ECR.REC` and `CANFDx_CHy_ECR.TEC`) remain frozen while error logging (`CANFDx_CHy_ECR.CEL`) is active.

The CPU can enable restricted operation mode by setting the restricted operation mode bit (`CANFDx_CHy_CCCR.ASM`). The `CANFDx_CHy_CCCR.ASM` bit can only be set when both `CANFDx_CHy_CCCR.CCE` and `CANFDx_CHy_CCCR.INIT` are set to 1. The CPU can reset `CANFDx_CHy_CCCR.ASM` at any time to disable restricted operation mode.

The CAN FD controller automatically enters restricted operation mode if the Tx handler is unable to read data from the message RAM in time. To exit this mode, the CPU must reset the `CANFDx_CHy_CCCR.ASM` bit.

restricted operation mode can be used in applications that adapt to different CAN bit rates. In such cases:

1. The application tests various bit rates while in restricted operation mode
2. The mode is exited once a valid frame is received

***Note**: Restricted operation mode must not be combined with loopback mode (both internal and external).*

23.3.1.4 Bus monitoring mode

In bus monitoring mode, the M_TTCAN is able to receive valid data frames and valid remote frames but cannot initiate any transmission. The M_TTCAN is set to bus monitoring mode by programming CANFDx_CHy_CCCR.MON to 1 or when error level S3 (CANFDx_CHy_TTOST.EL = 11) is reached.

In this mode, the M_TTCAN transmits only recessive bits on the CAN bus. If the M_TTCAN needs to send a dominant bit, such as an ACK bit, overload flag, or active error flag, the dominant bit is rerouted internally. This allows the M_TTCAN to monitor the internally rerouted dominant bit, even though the CAN bus stays in the recessive state. While in bus monitoring mode, the CANFDx_CHy_TXBRP register is held in a reset state.

The bus monitoring mode is useful for analyzing traffic on the CAN bus without influencing it by sending dominant bits. Figure 128 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.

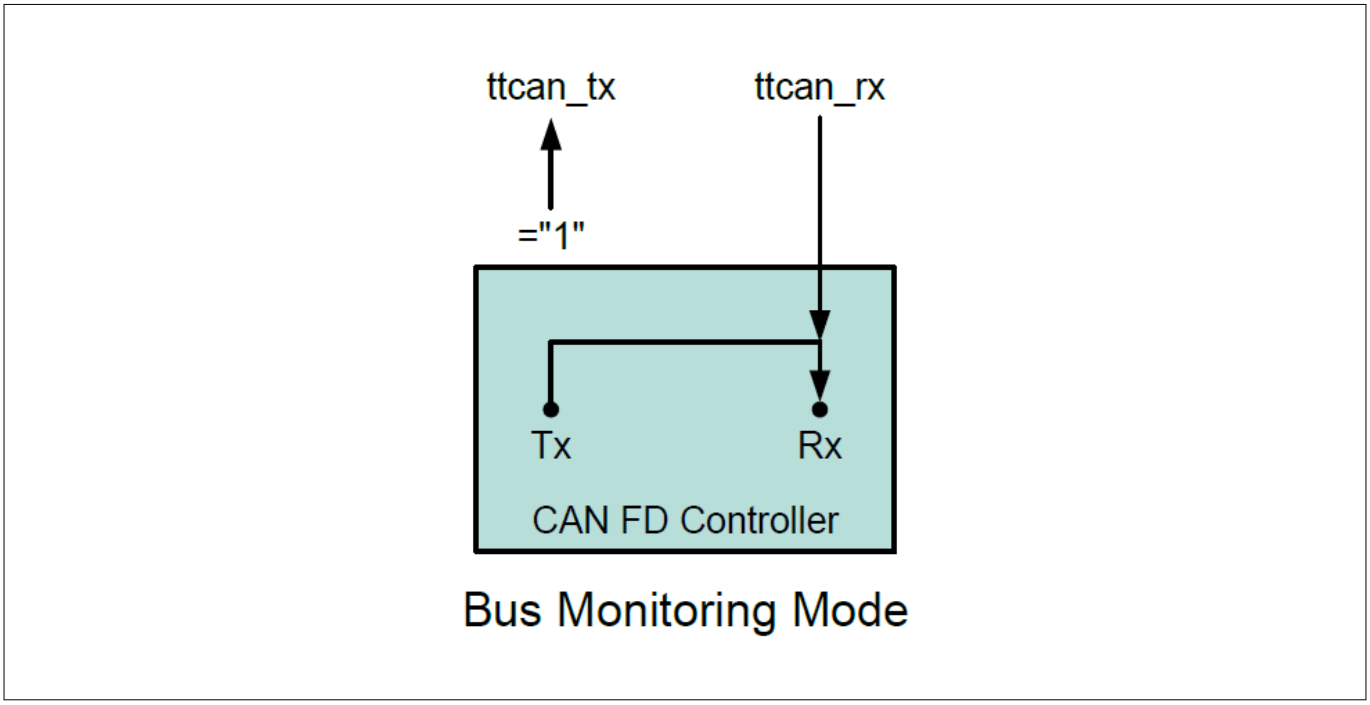


Figure 128 Pin control in bus monitoring mode

23.3.1.4.1 Disable automatic retransmission

The M_TTCAN supports automatic retransmission of frames that have lost arbitration or have been disturbed by errors during transmission. By default, automatic retransmission is enabled. To support time-triggered communication (as described in ISO 11898-1:2015, chapter 9.2), automatic retransmission can be disabled via CANFDx_CHy_CCCR.DAR.

In DAR mode, all transmissions are automatically canceled after they are started on the CAN bus. The Tx request pending bit CANFDx_CHy_TxBRP.TRPx is reset after:

- Successful transmission
- A transmission that has not yet started at the point of cancellation
- A transmission aborted due to lost arbitration
- An error occurring during frame transmission

Successful transmission:

- Corresponding Tx buffer transmission occurred bit CANFDx_CHy_TxBTO.TOx is set
- Corresponding Tx buffer cancellation finished bit CANFDx_CHy_TxBCF.CFx is not set

Successful transmission in spite of cancellation:

- Corresponding Tx buffer transmission occurred bit CANFDx_CHy_TxBTO.TOx is set
- Corresponding Tx buffer cancellation finished bit CANFDx_CHy_TxBCF.CFx is set

Arbitration lost or frame transmission disturbed:

- Corresponding Tx buffer transmission occurred bit CANFDx_CHy_TxBTO.TOx is not set
- Corresponding Tx buffer cancellation finished bit CANFDx_CHy_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).

23.3.1.4.2 Power down (Sleep mode)

The M_TTCAN channel can be set into power-down mode using the clock stop request (CANFDx_CTL.STOP_REQ). While the clock stop request is active, the STOP_REQ bit is read as 1.

When all pending transmission requests have been completed, the M_TTCAN waits until a bus idle state is detected. Once the bus idle state is confirmed, the M_TTCAN sets CANFDx_CHy_CCCR.INIT to 1 to prevent any further CAN transfers. At this point, the M_TTCAN signals readiness for power-down mode by setting the clock stop acknowledge bit (CANFDx_STATUS.STOP_ACK). After the acknowledgment is received from the channel, the hardware automatically disables the clock to the respective channel.

To exit power-down mode, the application must:

1. Reset the CANFDx_CTL.STOP_REQ bit
2. Wait for the M_TTCAN to acknowledge the action by resetting the CANFDx_STATUS.STOP_ACK bit
3. Restart CAN communication by resetting the CANFDx_CHy_CCCR.INIT bit

When the clock stop request is triggered through CANFDx_CTL.STOP_REQ, it must not be cleared until the CANFDx_STATUS.STOP_ACK bit is set.

Note:

- *Do not use the TTCAN register CANFDx_CHy_CCCR.CSR for power-down control. Instead, use CANFDx_CTL.STOP_REQ*
- *Avoid using CANFDx_CHy_CCCR.CSA for acknowledgment. Use CANFDx_STATUS.STOP_ACK instead*

23.3.1.5 Test mode

To enable write access to the TEST register, the test mode enable bit (CCCR.TEST) must be 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. These functions can be used to check the CAN bus physical layer

Due to the synchronization mechanism between the CAN clock and host clock domains, there may be a delay of several host clock periods between writing to TEST.TX and the new configuration becoming visible at the output pin canfd[0].ttcan_tx[0:1]. This also applies when reading the input pin canfd[0].ttcan_rx[0:1] via TEST.RX.

***Note**: Test modes should only be used for production tests or self-tests. Software control of the canfd[0].ttcan_tx[0] pin interferes with all CAN protocol functions. It is not recommended to use test modes for applications.*

23.3.1.5.1 External loopback mode

The M_TTCAN can be set to external loopback mode by programming TEST.LBCK to 1. In this mode, the M_TTCAN treats its own transmitted messages as received messages and stores them, if they pass acceptance filtering, into an Rx buffer or an Rx FIFO.

Figure 129 shows the connection of signals canfd[0].ttcan_tx[0:1] and canfd[0].ttcan_rx[0:1] to the M_TTCAN in external loopback mode. This mode is provided for hardware self-test purposes.

To ensure independence from external stimulation, the M_TTCAN ignores acknowledge errors (recessive bit sampled in the acknowledge slot of a data or remote frame) in loopback mode. In this mode, the M_TTCAN performs an internal feedback from its Tx output to its Rx input.

Key characteristics of the external loopback mode include:

- 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

23.3.1.5.2 Internal loopback mode

Internal loopback mode is entered by programming both the TEST.LBCK and CCCR.MON bits to 1. This mode is used for a 'Hot self-test,' allowing the M_TTCAN to 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 Internal Loopback mode:

- The canfd[0].ttcan_rx[0:1] pin is disconnected from the M_TTCAN
- The canfd[0].ttcan_tx[0:1] pin is held recessive

Figure 129 shows the connection of canfd[0].ttcan_tx[0:1] and canfd[0].ttcan_rx[0:1] to the M_TTCAN in internal loopback mode.

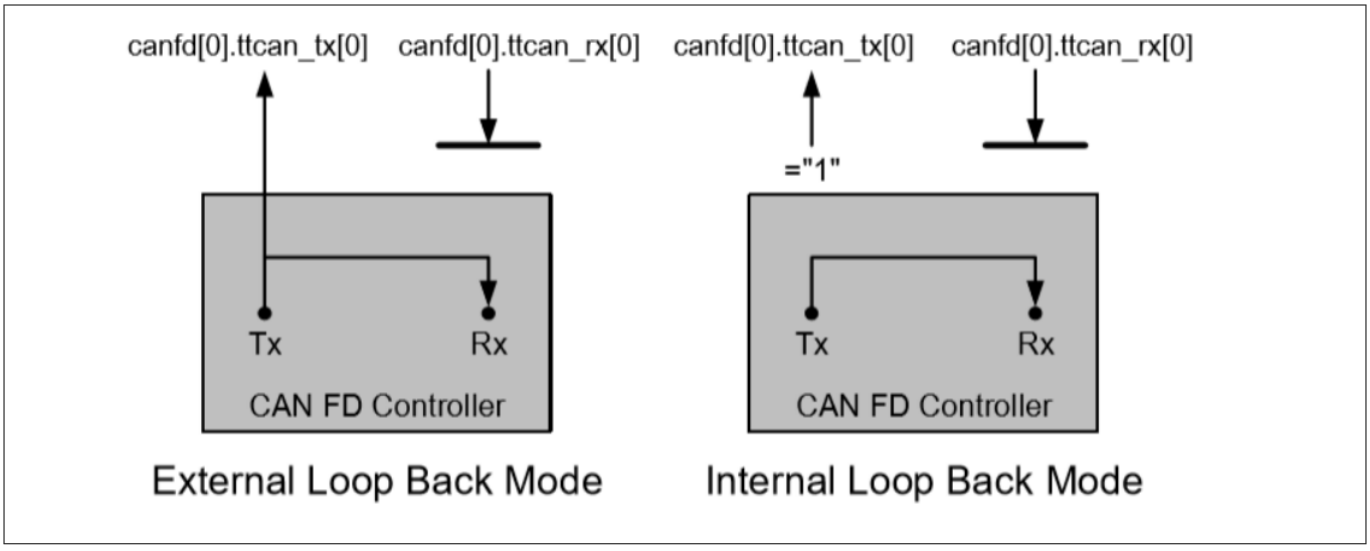


Figure 129 Pin control in loopback modes

Application watchdog

The application watchdog is served by monitoring the CANFDx_CHy_TTOST register. When the application watchdog is not served in time:

1. The CANFDx_CHy_TTOST.AWE bit is set
2. All TTCAN communication is stopped
3. The M_TTCAN automatically enters the bus monitoring mode

The TT application watchdog can be disabled by programming the application watchdog limit (CANFDx_CHy_TTOCF.AWL) to 0x00. However, in a TTCAN application program, the application watchdog should not be disabled.

23.3.2 Timestamp generation

The M_TTCAN channel uses a 16-bit counter to record the timestamps of message transmission or reception. This allows application software to determine the order of events. The timestamp counter is configured via the TS_CTL register.

1. The TS_CTL.ENABLED bit enables the counter
   - When enabled, the counter begins incrementing based on the value of TS_CTL.PRESCALE [15:0]
   - The application can read the current counter value through the TS_CNT register
   - Writing to the TS_CNT register clears its counter value
2. After enabling, the internal prescaler counter increments with every cycle of clk_sys
   - When the prescaler counter value reaches the configured value in TS_CTL.PRESCALE, the timestamp counter increments by 1
   - The internal prescaler counter is reset after each increment of the timestamp counter
3. If the TS_CTL.PRESCALE value is modified, the TS_CNT register should be written to reset the internal prescaler counter. This ensures the internal prescaler immediately follows the updated TS_CTL.PRESCALE value
4. The shared timestamp counter is a wraparound counter. When the counter wraps around, the interrupt flag IR.TSW for the M_TTCAN channel is raised
5. During frame transmission or reception, the timestamp counter value is captured and stored in the following fields:
   - RXTS [15:0] for Rx buffer/Rx FIFO elements
   - TXTS [15:0] for Tx event FIFO elements

***Note**: The TS_CNT value is not retained in Deep Sleep mode, however, the TS_CTL register content is retained.*

23.3.3 Timeout counter

To signal timeout conditions for Rx FIFO 0, Rx FIFO 1, and the Tx event FIFO, the M_TTCAN provides a 16-bit timeout counter. It operates as a down-counter and uses the same prescaler, controlled by TSCC.TCP, as the timestamp counter. A prescaler TSCC.TCP should be configured to clock the timeout counter in multiples of CAN bit times (1 to 16). The timeout counter is configured via the TOCC register. The current counter value can be read from TOCV.TOC.

The timeout counter can only be started when CCCR.INIT = 0. It stops when CCCR.INIT = 1, such as when the M_TTCAN enters the bus_off state.

The operation mode is selected by TOCC.TOS. When operating in continuous mode, the counter starts when CCCR.INIT is reset. Writing 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 starts when the first FIFO element is stored. Writing to TOCV has no effect in this mode. When the counter reaches zero, the interrupt flag IR.TOO is set. In 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 interval at which the timeout counter is decremented may vary due to the synchronization or resynchronization mechanism of the CAN core. If the bit rate switch (BRS) feature in CAN FD is used, the timeout counter is clocked differently in the arbitration and data fields.*

23.3.4 Receive handling

The Rx handler manages acceptance filtering, the transfer of received messages to Rx buffers or one of the two Rx FIFOs, as well as the update of the Rx FIFO put and get indices.

23.3.4.1 Acceptance filtering

The M_TTCAN provides the functionality 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 or FIFO 1. For acceptance filtering, each filter list is processed sequentially, starting from element #0, until the first matching filter element is found. Acceptance filtering stops at the first match, and subsequent filter elements are not evaluated for the respective message.

The main features include:

- Each filter element can be configured as:
  - A range filter (from - to)
  - A filter for one or two specific IDs
  - A classic bit mask filter
- Each filter element supports the following capabilities:
  - Configurable for either acceptance or rejection filtering
  - Can be enabled or disabled individually
- Filters are evaluated sequentially, and the process stops at the first matching filter element
- Related configuration registers include:
  - 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 (via SFEC or EFEC), a match can trigger one of the following actions:

1. Store the received frame in FIFO 0 or FIFO 1
2. Store the received frame in an Rx buffer
3. Store the received frame in an Rx buffer and generate a pulse on the filter event pin
4. Reject the received frame
5. Set the high-priority message interrupt flag IR.HPM
6. Set the high-priority message interrupt flag IR.HPM and store the received frame in FIFO 0 or FIFO 1

Acceptance filtering begins after the complete identifier is received. Once acceptance filtering completes, if a matching Rx buffer or Rx FIFO is found, the message handler transfers the received message data in 32-bit segments to the matching Rx buffer or Rx FIFO.

If the CAN protocol controller detects an error condition (such as a CRC error), the affected message is discarded. The discard operation affects the corresponding Rx buffer or Rx FIFO, which then remains unmodified.

Rx buffer

The new data flag of the matching Rx buffer is not set; however, the Rx buffer may be partially or fully overwritten with the received data. The type of error can be identified through PSR.LEC or PSR.DLEC.

Rx FIFO

The put index of the matching Rx FIFO is not updated; however, the corresponding Rx FIFO element may be partially or fully overwritten with the received data. The type of error can be identified through PSR.LEC or PSR.DLEC. If the matching Rx FIFO operates in overwrite mode, the boundary conditions described in Rx FIFO overwrite mode must be observed.

***Note**: When an accepted message is stored in one of the two Rx FIFOs or in an Rx buffer, the unmodified received identifier is always stored, regardless of the filter used. The result of the acceptance filtering process depends on the sequence of the configured filter elements.*

Range filter

The filter matches all received frames with message IDs that fall within the range defined by SF1ID/SF2ID or EF1ID/EF2ID, respectively.

When range filtering is used with extended frames, there are two possible scenarios:

- **EFT = 00**: The message ID of the received frames is ANDed with the XIDAM before applying the range filter
- **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 single message ID, the filter element should be configured with SF1ID = SF2ID for standard frames or EF1ID = EF2ID for extended frames.

Classic bit mask filter

Classic bit mask filtering is designed to filter groups of message IDs by masking individual bits of a received message ID. In this method, SF1ID/EF1ID is used as the message ID filter, while SF2ID/EF2ID serves as the filter mask.

When using a filter mask:

- A zero bit in the filter mask ignores the corresponding bit position of the configured ID filter. In this case, the value of the received message ID at that bit position is irrelevant for acceptance filtering
- Only the bits of the received message ID where the corresponding mask bits are set to one are considered for acceptance filtering

Different mask configurations result in specific behaviors:

- If all mask bits are one, a match occurs only if the received message ID is identical to the configured message ID filter
- If all mask bits are zero, all message IDs are accepted regardless of their value

Standard message ID filtering

Figure 130 shows the flow for standard message ID (11-bit identifier) filtering. The standard message ID filter element is described in the Standard message ID filter element section.

Controlled by the GFC and SIDFC, the message IDs, the remote transmission request (RTR) bit, and the identifier extension (IDE) bit of received frames are compared against the configured list of filter elements.

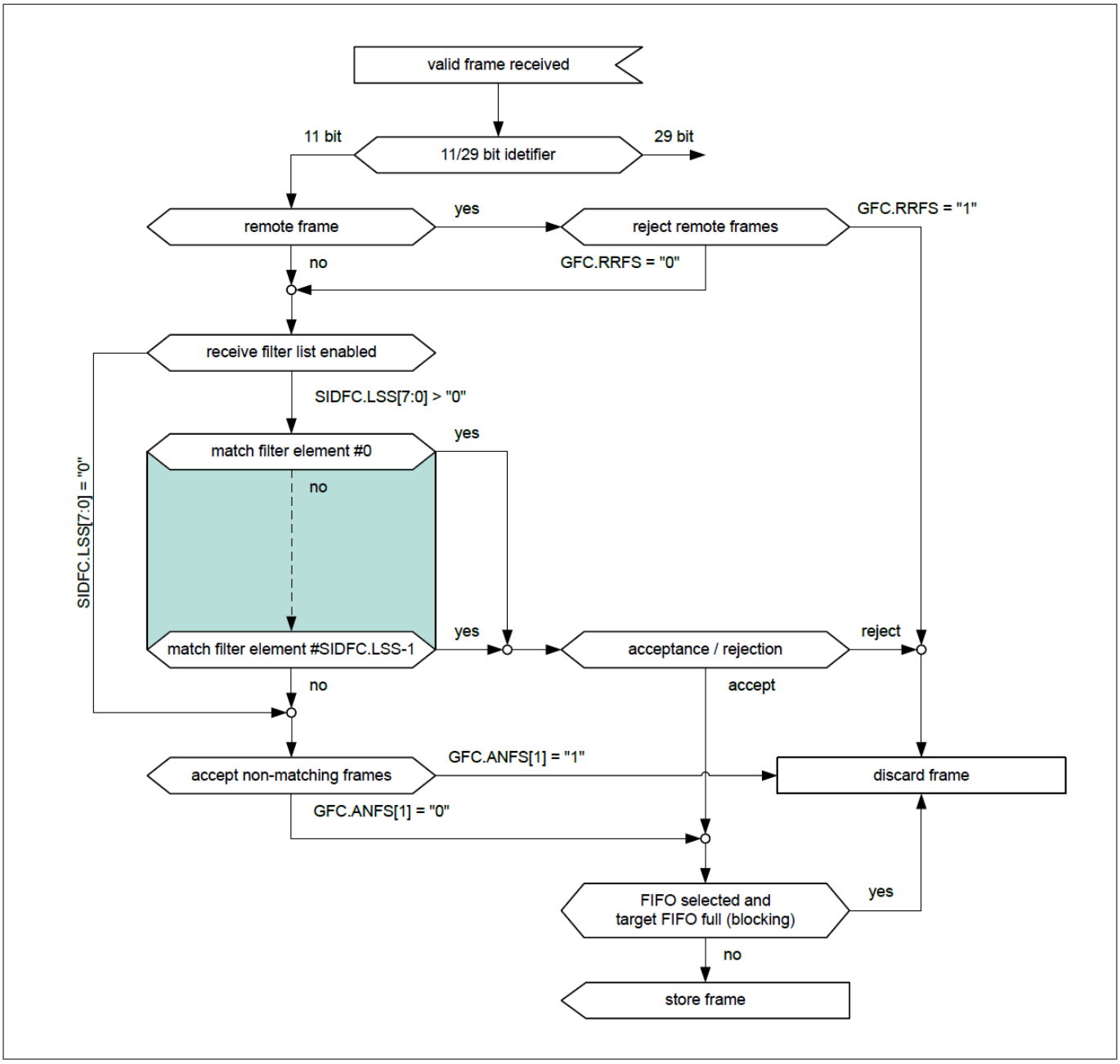


Figure 130 Standard message ID filter flow diagram

Extended message ID filtering

Figure 131 shows the flow for extended message ID (29-bit identifier) filtering. The extended message ID filter element is described in the Extended message ID filter element section.

Controlled by the GFC and XIDFC, the message IDs, the remote transmission request (RTR) bit, and the identifier extension (IDE) bit of received frames are compared against the list of configured filter elements. Before executing the filter list, the extended ID AND mask (XIDAM[28:0]) is applied by performing a bitwise AND operation with the received identifier.

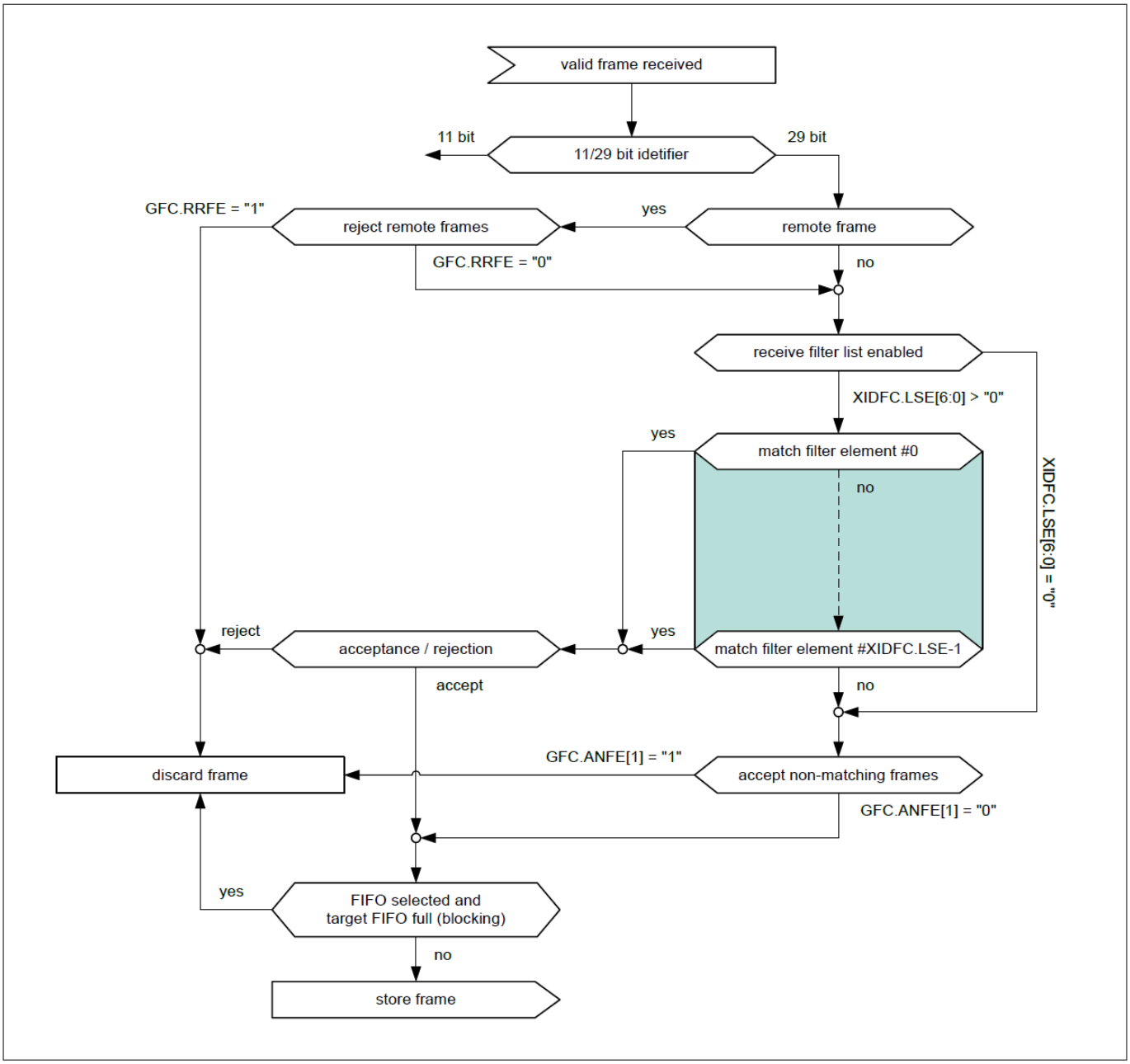


Figure 131 Extended message ID filter flow diagram

23.3.4.2 Receive FIFOs

Rx FIFO 0 and Rx FIFO 1 can each be configured to hold up to 64 elements. The two Rx FIFOs are configured using the `RXF0C` and `RXF1C` registers.

Received messages that pass acceptance filtering are transferred to the Rx FIFO specified by the matching filter element. For detailed information about the available filter mechanisms for Rx FIFO 0 and Rx FIFO 1, see Acceptance filtering. The structure of an Rx FIFO element is described in Rx buffer and FIFO element.

To prevent an Rx FIFO overflow, an Rx FIFO watermark can be configured. When the Rx FIFO fill level reaches the watermark value defined by `RXFnC.FnWM`, the interrupt flag `IR.RFnW` is set. If the Rx FIFO put index reaches the Rx FIFO get index, an Rx FIFO full condition is signaled via `RXFnS.FnF`, and the interrupt flag `IR.RFnF` is also set.

The FIFO watermark interrupt flags can be used to trigger DMA operations. The DMA request for the FIFO remains active until the corresponding trigger condition is cleared by software. Software can clear the trigger by resetting the watermark flag.

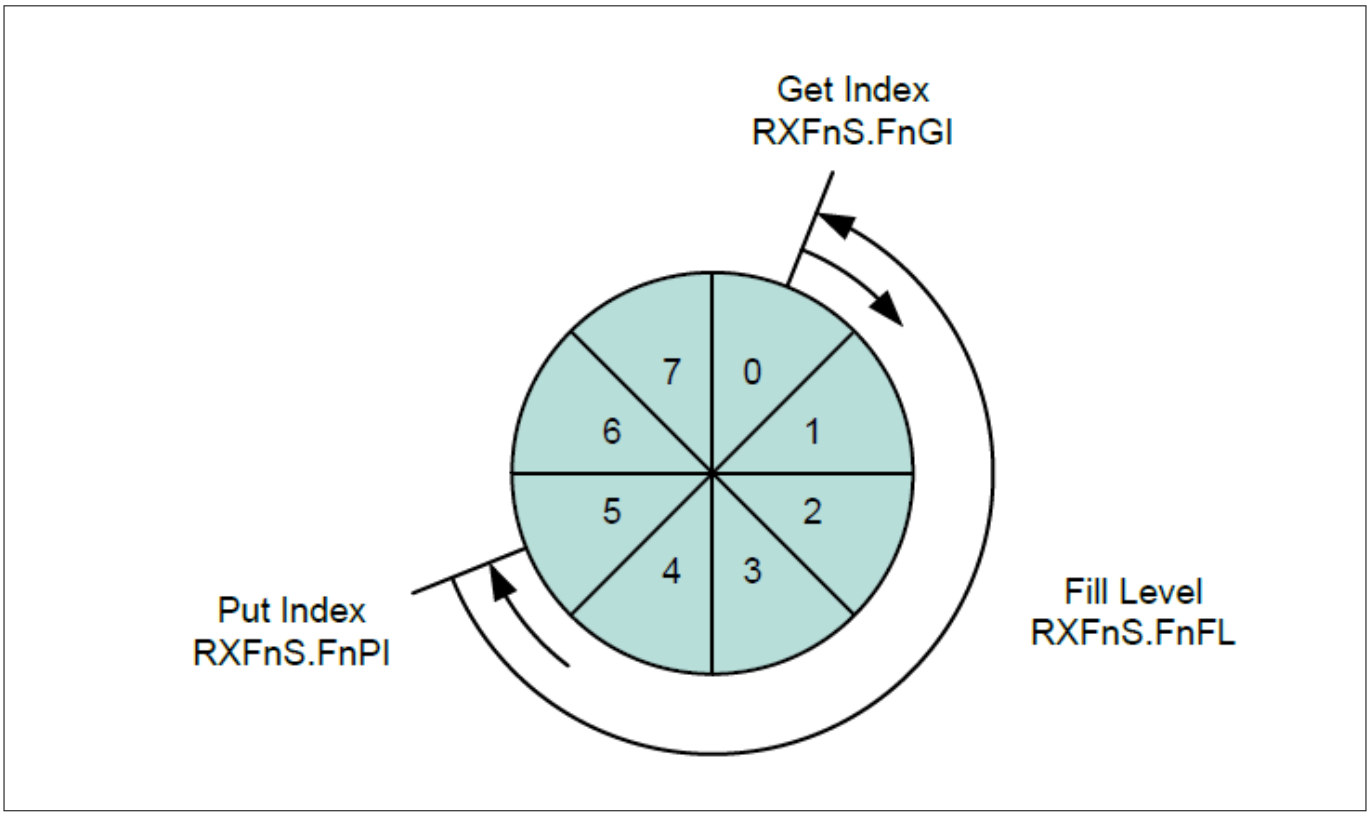


Figure 132 Rx FIFO status

When reading from an Rx FIFO, the Rx FIFO get index (`RXFnS.FnGI`) multiplied by the Rx FIFO element size must be added to the corresponding Rx FIFO start address (`RXFnC.FnSA`). The CAN FD controller includes Rx FIFO top pointer logic to facilitate faster reading from the Rx FIFO. For more details, see Rx FIFO top pointer.

Table 115 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

23.3.4.2.1 Rx FIFO blocking mode

The Rx FIFO blocking mode is configured by setting `RXFnC.FnOM = 0`. This is the default operation mode for Rx FIFOs.

When an Rx FIFO full condition occurs (i.e., `RXFnS.FnPI = RXFnS.FnGI`), no additional 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 indicated by `RXFnS.FnF = 1`, and the interrupt flag `IR.RFnF` is also set.

If a message is received while the corresponding Rx FIFO is full, the message will be discarded, and a message lost condition is indicated by `RXFnS.RFnL = 1`. Additionally, the interrupt flag `IR.RFnL` is set.

23.3.4.2.2 Rx FIFO overwrite mode

The Rx FIFO overwrite mode is configured by setting `RXFnC.FnOM = 1`. When an Rx FIFO full condition occurs (i.e., `RXFnS.FnPI = RXFnS.FnGI`), it is signaled by `RXFnS.FnF = 1`. In this mode, the next accepted message for the FIFO will overwrite the oldest FIFO message. Both the put and get indices are incremented by one.

When an Rx FIFO is operated in overwrite mode, and a full condition is signaled, message reading from the Rx FIFO should begin at least from `Get Index + 1`. This precaution ensures consistent data reading, as a received message might be written by the hardware to memory (MRAM) at the put index while the CPU is simultaneously reading from memory at the get index. In such cases, inconsistent or incomplete data could be read from the Rx FIFO element. Adding an offset to the get index when accessing the Rx FIFO helps avoid this issue. The required offset depends on how quickly the CPU accesses the Rx FIFO.

For example, in Figure 133, an offset of two relative to the get index is shown when reading the Rx FIFO. This results in the loss of two messages stored in elements 1 and 2.

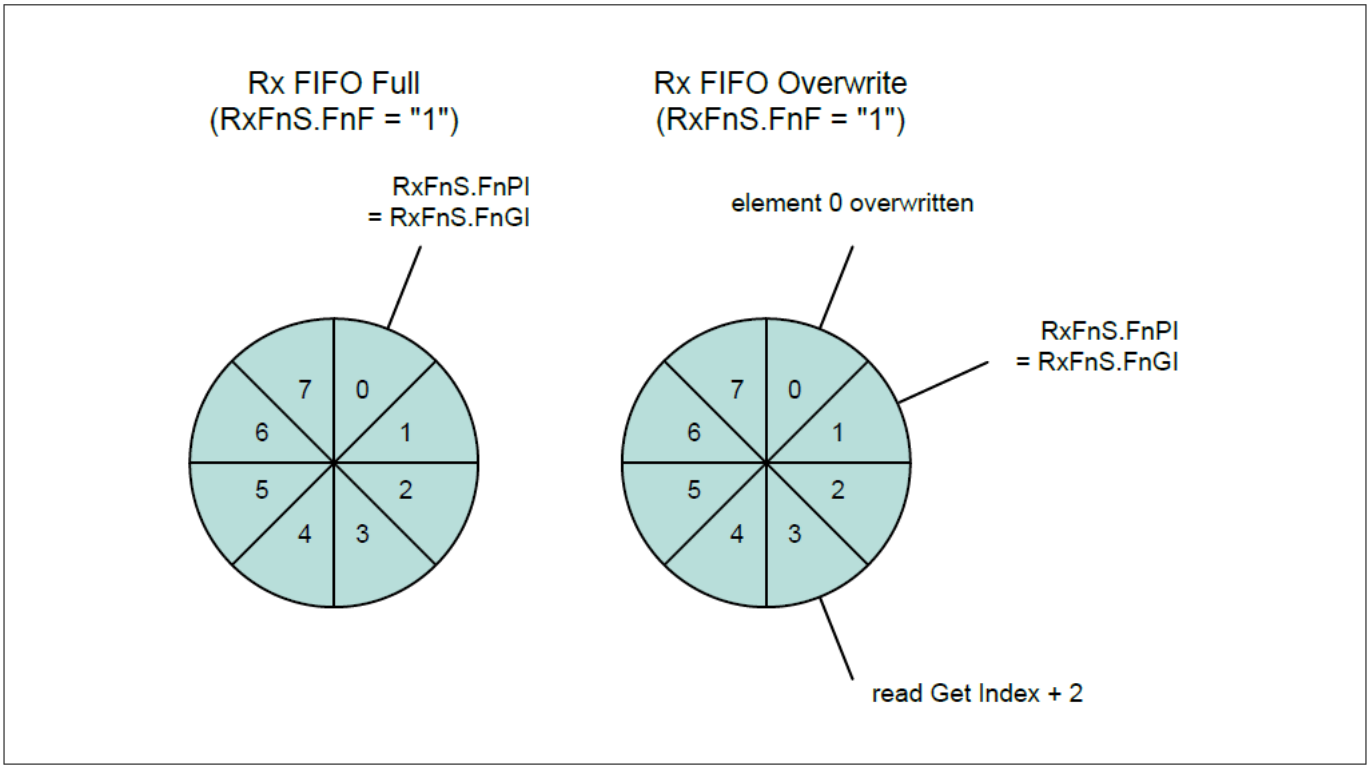


Figure 133 Rx FIFO overflow handling

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 operation increments the get index to the specified element number. If the put index has not advanced to this Rx FIFO element, the Rx FIFO full condition is cleared (i.e., `RXFnS.FnF = 0`).

23.3.4.2.3 Rx FIFO top pointer

The `M_TTCAN` supports two receive FIFOs. Reading from these FIFOs traditionally requires the application to follow these steps:

1. Retrieve the read pointer
2. Calculate the correct MRAM address
3. Read the data from MRAM
4. Update the read pointer

To streamline this process, Rx FIFO top pointer logic has been integrated into the CAN FD controller. This logic provides a single MMIO location (RXFTOPn_DATA; where n = 0, 1) to directly read the data. Using this hardware-based logic offers several benefits:

- Higher performance data access
- Reduced bus traffic
- Lower CPU load
- Reduced power consumption
- Enables DMA access to the FIFO

The top pointer logic is enabled by setting RXFTOP_CTL.FnTPE. Enabling this bit activates the logic, which manages the FIFO's top address (FnTA) and an internal message word counter. The receive FIFO's top status register (RXFTOPn_STAT) shows the corresponding FIFO top address, and the RXFTOPn_DATA register provides the data from that address. Refer to the register definitions for more details on these registers.

When RXFTOPn_DATA is read, the top pointer logic automatically updates the Rx FIFO acknowledge index (RXFnA.FnA) for the M_TTCAN channel.

Note:
- *The top pointer logic is disabled when the channel is in configuration mode (CCCR.CCE = 1). If RXFTOPn_DATA is read while the logic is disabled, it will return invalid data*
- *When the Rx FIFO size is set to 1, the Rx FIFO top pointer function should not be used. Instead, messages should be read directly from MRAM*
- *When the Rx FIFO size is set to 2 or more, there are no such limitations*

23.3.4.3 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.

To use an Rx buffer, a standard or extended message ID filter element must be configured with the following settings:

- SFEC/EFEC = 111
- SFID2/EFID2[10:9] = 00

For more details, see Standard message ID filter element and Extended message ID filter element.

When a received message is accepted by a filter element, the message is stored in the corresponding Rx buffer within the message RAM referenced by that filter element. The format of the stored message is identical to that of an Rx FIFO element. Additionally, the IR.DRX flag (message stored in a dedicated Rx buffer) in the interrupt register is set.

Table 116 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 2	00	00 0001
2	ID message 3	00	00 0010

After the last word of a matching received message is written to the message RAM, the respective new data flag in the NDAT1 and NDAT2 registers is set. While the new data flag is set, the corresponding Rx buffer is locked and cannot be updated by received matching frames. The host should reset the new data flag by writing a '1' to the corresponding bit position.

While the new data flag of an Rx buffer is set, a message ID filter element referencing the specific Rx buffer will not match, causing acceptance filtering to continue. Subsequent message ID filter elements may result in the received message being stored in another Rx buffer, assigned to an Rx FIFO, or rejected, depending on the filter configuration.

Rx buffer handling:

1. Reset the interrupt flag IR.DRX
2. Read the new data registers
3. Read messages from the message RAM
4. Reset the new data flags for the processed messages

23.3.4.4 Debug on CAN support

Debug messages are stored into Rx buffers. Three consecutive Rx buffers (for example, #61, #62, and #63) should be used to store debug messages A, B, and C. The format of debug messages is the same as that used for Rx buffer and Rx FIFO elements. To filter debug messages, standard or extended filter elements should be configured with SFEC/EFEC = 111. Messages that match these filter elements are stored into the Rx buffers addressed by SFID2/EFID2[5:0].

Once message C is stored, the DMA request is activated, allowing the three debug messages to be read from the message RAM under DMA control. While the DMA request is active, the M_TTCAN ensures that the RAM words containing the debug messages are not modified, similar to the behavior of an Rx buffer with its new data flag set.

When the DMA transfer is completed, an acknowledgment from the DMA clears the DMA request. This enables the M_TTCAN to store the next set of debug messages.

23.3.4.4.1 Filtering debug messages

Debug messages are filtered by configuring one standard or 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". In this configuration, the SFID1/SFID2 and EFID1/EFID2 fields have a modified meaning (see Standard message ID filter element and Extended message ID filter element).

- The SFID2/EFID2[10:9] fields control the state machine for debug message handling
- The SFID2/EFID2[5:0] fields determine the storage location for a received debug message

When a debug message is stored, neither the new data flag nor the IR.DRX flag is set. The reception of debug messages can be monitored via RXF1S.DMS.

Table 117 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

23.3.4.4.2 Debug message handling

The debug message handling state machine ensures that debug messages are stored in the correct order across three consecutive Rx buffers. If any message is missing, the state machine restarts the process. The DMA request is activated only after all three debug messages (A, B, and C) are received in the correct order.

The status of the debug message handling state machine is indicated via `RXF1S.DMS`.

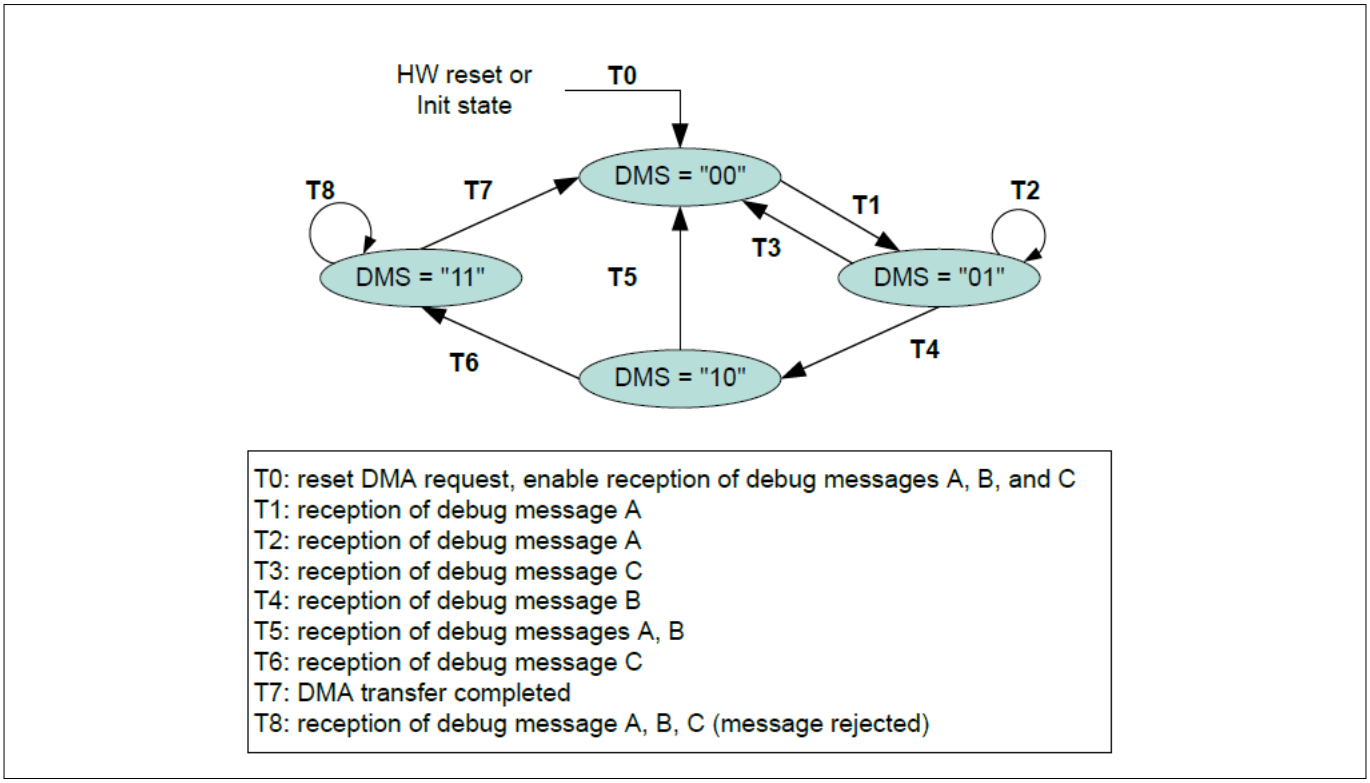


Figure 134 Debug message handling state machine

23.3.5 Transmit handling

The Tx handler manages transmission requests for dedicated Tx buffers, the Tx FIFO, and the Tx queue. It controls the transfer of transmit messages to the CAN core, as well as the put and get indices and the Tx event FIFO. Up to 32 Tx buffers can be configured for message transmission. The transmission mode (Classic CAN or CAN FD) can be configured individually for each Tx buffer element. The structure of the Tx buffer element is described in Tx buffer element, while Table 118 outlines the possible frame transmission configurations.

Table 118 Possible configuration for frame transmission

CCCR		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

(table continues...)

Table 118 (continued) Possible configuration for frame transmission

CCCR		Tx buffer element		Frame transmission
BRSE	FDOE	FDF	BRS	
1	1	1	0	FD without bit rate switching
1	1	1	1	FD with bit rate switching

The Tx handler initiates a Tx scan to identify the highest-priority pending transmission request (the Tx buffer with the lowest message ID). This scan is triggered either when the Tx Buffer Request Pending register (TXBRP) is updated or when a transmission is started.

23.3.5.1 Transmit pause

he transmit pause feature is designed for use in CAN systems where message identifiers are permanently assigned specific values and cannot be easily modified. In such cases, certain message identifiers may inherently hold a higher CAN arbitration priority than others. However, in specific applications, the relative arbitration priority of these messages may need to be inverted. This situation can lead to scenarios where one electronic control unit (ECU) sends a burst of high-priority CAN messages, causing messages from other ECUs with lower arbitration priority to be delayed.

For example, if CAN ECU-1 has the transmit pause feature enabled and is instructed by the application software to transmit four messages, it will pause for two nominal bit times of bus idle after successfully transmitting the first message before attempting to transmit the next message. This pause allows other ECUs with pending messages to initiate their transmission during the idle time, avoiding arbitration contention with ECU-1's next message. After receiving a message, ECU-1 is free 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). When this bit is set, the M_TTCAN controller pauses for two nominal bit times after successfully transmitting a message before starting the next transmission. This behavior ensures that other CAN nodes in the network can transmit their messages, even if their message identifiers have a lower arbitration priority. By default, the transmit pause feature is disabled (CCCR.TXP = 0).

This feature mitigates burst transmissions originating from a single node and protects the CAN network against 'babbling idiot' scenarios, where the application program erroneously sends excessive transmission requests.

23.3.5.2 Dedicated Tx buffers

Dedicated Tx buffers are designed for message transmission with complete control by 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 first.

When the data section of a Tx buffer is updated, a transmission is initiated by adding a request through TXBAR.ARn. The requested messages then arbitrate internally with messages from an optional Tx FIFO or Tx queue and externally with messages on the CAN bus. Transmission is carried out according to the priority of their message IDs.

23.3.5.2.1 Addressing dedicated Tx buffers

A dedicated Tx buffer occupies an element size in 32-bit words within the message RAM, as shown in Table 119. The start address of a dedicated Tx buffer in the message RAM is calculated using the formula: `(Start Address) = (Transmit Buffer Index (0 to 31) × Element Size) + TX Buffers Start Address (`TXBC.TBSA[15:2]`)`

Table 119 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
100	24	8
101	32	10
110	48	14
111	64	18

23.3.5.3 Transmit FIFO

Tx FIFO operation is configured by setting TXBC.TFQM to '0'. Messages stored in the Tx FIFO are transmitted in sequence, 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 Tx FIFO allows the transmission of messages with the same message ID from different Tx buffers in the order they were added to the Tx FIFO. The M_TTCAN calculates the Tx FIFO Free Level (TXFQS.TFFL) as the difference between the Get Index and the Put Index, indicating the number of available (free) Tx FIFO elements.

Writing messages to the Tx FIFO:

1. New transmit messages must be written to the Tx FIFO starting at the Tx buffer referenced by the Put Index (TXFQS.TFQPI)
2. An Add Request increments the Put Index to the next free Tx FIFO element
3. When the Put Index reaches the Get Index, the Tx FIFO full condition is signaled by TXFQS.TFQF = 1. At this point, no additional messages should be written to the Tx FIFO until the next message is transmitted, and the Get Index is incremented

Transmission requests:

- For a single message added to the Tx FIFO, the transmission is requested by writing a '1' to the TXBAR bit corresponding to the Tx buffer referenced by the Tx FIFO Put Index
- For multiple (n) messages added to the Tx FIFO, the messages are written to 'n' consecutive Tx buffers starting from the Put Index, and the transmissions are requested via TXBAR. The Put Index is then cyclically incremented by 'n'
- The number of requested Tx buffers must not exceed the number of free Tx buffers, as indicated by the Tx FIFO Free Level (TXFQS.TFFL)

Behavior of the Get Index and transmission cancellation:

- 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 a pending transmission request, and the Tx FIFO Free Level is recalculated
- When the transmission cancellation is applied to any other Tx buffer, the Get Index and the Tx FIFO Free Level remain unchanged

A Tx FIFO element occupies an element size of 32-bit words in the message RAM, as described in Table 118. The start address of the next available (free) Tx FIFO buffer is calculated as:

Start Address = TX Buffer Start Address (`TXBC.TBSA`) + (Tx FIFO/Queue Put Index (`TXFQS.TFQPI`) × Element Size)

.

23.3.5.4 TX queue

he Tx queue operation is configured by setting TXBC.TFQM to '1'. Messages stored in the Tx queue are transmitted in priority order, starting with the message that has the lowest message ID (highest priority). If multiple Tx queue buffers are configured with the same message ID, the transmission order depends on the buffer numbers where the messages are stored. Since these buffer numbers are influenced by the current state of the Put Index, predicting the precise transmission order is not possible.

Writing messages to the Tx queue:

1. New messages must be written to the Tx buffer referenced by the Put Index (TXFQS.TFQPI)
2. An Add Request increments the Put Index cyclically to the next free Tx buffer
3. If the Tx queue is full (TXFQS.TFQF = 1), the Put Index becomes invalid, and no additional messages should be written to the Tx queue until at least one of the requested messages is transmitted or a pending transmission request is canceled

The application can also use the TXBRP register instead of the Put Index and write messages to any Tx buffer where there is no pending transmission request.

A Tx queue buffer allocates an element size of 32-bit words in the message RAM, as described in Table 118. The start address of the next available (free) Tx queue buffer is calculated as:

Start Address = TX Buffer Start Address (`TXBC.TBSA`) + (TX FIFO/Queue Put Index (`TXFQS.TFQPI`) × Element Size)

23.3.5.5 Mixed dedicated Tx buffers/Tx FIFO

In this configuration, the Tx buffer section in the MRAM is divided into a set of dedicated Tx buffers and a Tx FIFO. The number of dedicated Tx buffers is determined by TXBC.NDTB, while the number of Tx buffers assigned to the Tx FIFO is configured by TXBC.TFQS. If TXBC.TFQS is set to zero, only dedicated Tx buffers are used for message transmission.

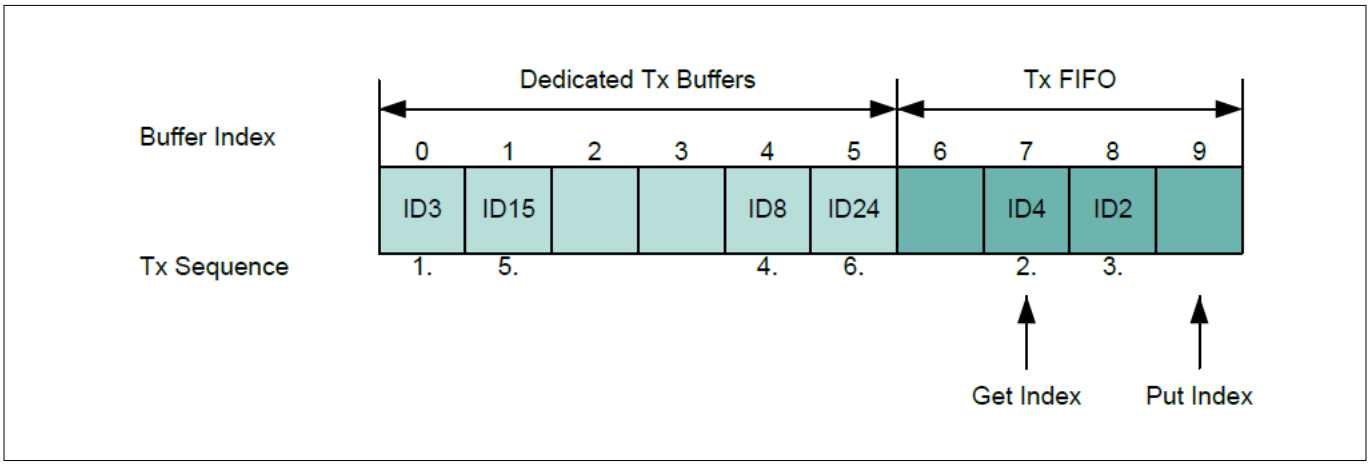


Figure 135 Example of mixed configuration dedicated Tx buffers/Tx FIFO

Tx prioritization:

- The Tx handler scans both the dedicated Tx buffers and the oldest pending Tx FIFO buffer (referenced by TXFS.TFGI)
- The buffer with the lowest message ID is given the highest priority and is transmitted next

23.3.5.6 Mixed dedicated Tx buffer/Tx queue

In this configuration, the Tx buffers section in the MRAM is divided into a set of dedicated Tx buffers and a Tx queue. The number of dedicated Tx buffers is defined by TXBC.NDTB, while the number of Tx queue buffers is configured by TXBC.TFQS. If TXBC.TFQS is set to zero, only dedicated Tx buffers are used for message transmission.

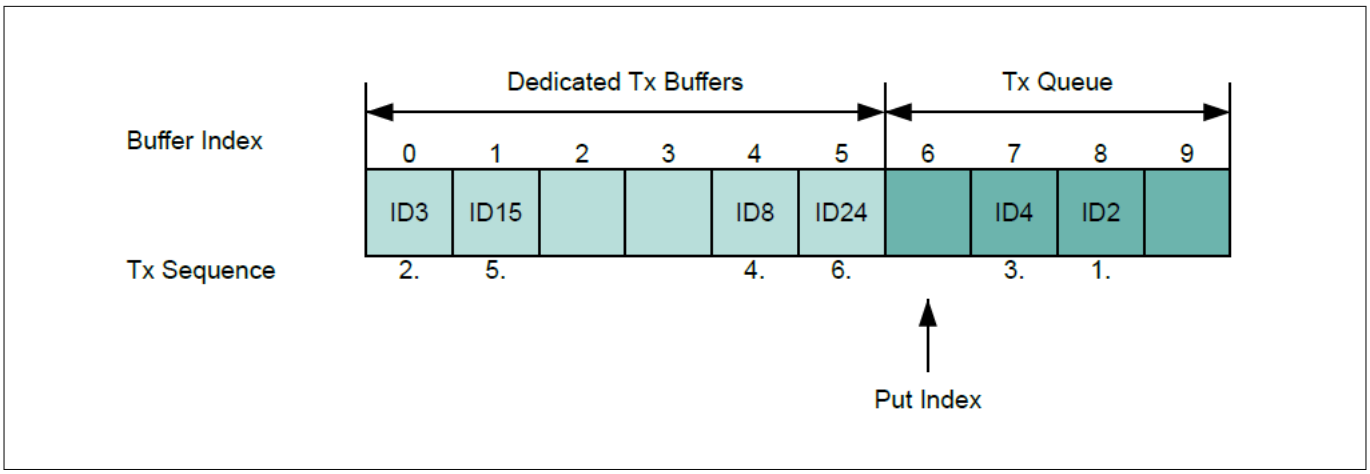


Figure 136 Example of mixed configuration dedicated Tx buffers/Tx queue

Tx prioritization:

- The Tx handler scans all Tx buffers with an active transmission request
- The Tx buffer with the lowest message ID is assigned the highest priority and is transmitted next

23.3.5.7 Transmit cancellation

The 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 a '1' to the corresponding bit position (number of Tx buffers) in the TXBCR register. 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 the case a 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. If the transmission was not successful, 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, a short time window follows where no transmission is initiated, even if another message is 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.*

23.3.5.8 Transmit event handling

To support Tx event handling, the M_TTCAN implements a Tx event FIFO. After the M_TTCAN transmits a message on the CAN bus, the message ID and timestamp are stored in a Tx event FIFO element. To link a Tx event to a 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 hold up to a maximum of 32 elements. The structure of the Tx event FIFO element is described in the Tx event FIFO element section.

The purpose of the Tx event FIFO is to decouple handling of transmit status information from transmit message handling. A Tx buffer contains only the message to be transmitted, while the transmit status is stored separately

in the Tx event FIFO. This separation is especially useful when operating a dynamically managed transmit queue, as it allows a Tx buffer to be reused for a new message as soon as the previous transmission is completed. There is no need to first save the transmission status information from the Tx buffer before overwriting it with a new message.

When a Tx event FIFO full condition is signaled by IR.TEFF, no new elements are written to the Tx event FIFO until at least one element is read, and the Tx event FIFO get index is incremented. If a Tx event occurs while the Tx event FIFO is full, the event is discarded, and the interrupt flag IR.TEFL is set.

To prevent Tx event FIFO overflow, the Tx event FIFO watermark can be configured. When the Tx event FIFO fill level reaches the configured watermark value in TXEFC.EFWM, the interrupt flag IR.TEFW is set. When reading from the Tx event FIFO, the Tx event FIFO get index (TXEFS.EFGI) must be multiplied by two and added to the Tx event FIFO start address (TXEFC.EFSA).

23.3.6 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 updated automatically, which will eventually set the FIFO Get Index to the FIFO Acknowledge Index plus one and thereby update the FIFO fill level.

When the application does not use the Rx FIFO top pointer logic, the Acknowledge Index must be updated. This can be done using one of the following two use cases:

- When only a single element is read from the FIFO (the one being pointed to 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 (the value is the index of the last element read) to update the FIFO Get Index

Because the CPU has free access to the M_TTCAN message RAM, take care when reading FIFO elements in an arbitrary order (without considering the Get Index). This may be useful when reading a high-priority message from one of the two Rx FIFOs. In this case, the FIFO Acknowledge Index should not be written because this will set the Get Index to an incorrect position and alter the FIFO fill level, causing the loss of older FIFO elements.

***Note**: The application must ensure that a valid value is written to the FIFO Acknowledge Index. The M_TTCAN does not check for erroneous values.*

23.3.7 Configuring CAN bit timing

Each node in a CAN network has its own clock generator, typically using a quartz oscillator. The time parameters of the bit time can be configured individually for each CAN node. Even if the oscillators of different CAN nodes operate at slightly different periods, a common bit rate can still be achieved.

The oscillator frequencies may vary slightly due to factors such as temperature changes, voltage fluctuations, or component deterioration. As long as these frequency variations remain within the tolerance range of the oscillators, the CAN nodes can compensate for the differences in bit rates by resynchronizing to the bit stream.

23.3.7.1 CAN bit timing

CAN FD operation defines two types of bit times: the nominal bit time and the data bit time. The nominal bit time is used during the arbitration phase, while the data bit time, which is equal or shorter in length, can be used to accelerate the data phase (see CAN FD operation).

The basic structure of a bit time applies to both the nominal and data bit times. According to the CAN specifications, the bit time is divided into four segments (see Figure 137):

- Synchronization segment (Sync_Seg)
- Propagation time segment (Prop_Seg)
- Phase buffer segment 1 (Phase_Seg1)
- Phase buffer segment 2 (Phase_Seg2)

The sample point, which is the moment when the bus level is read and interpreted as the value of the respective bit, is located at the end of Phase_Seg1.

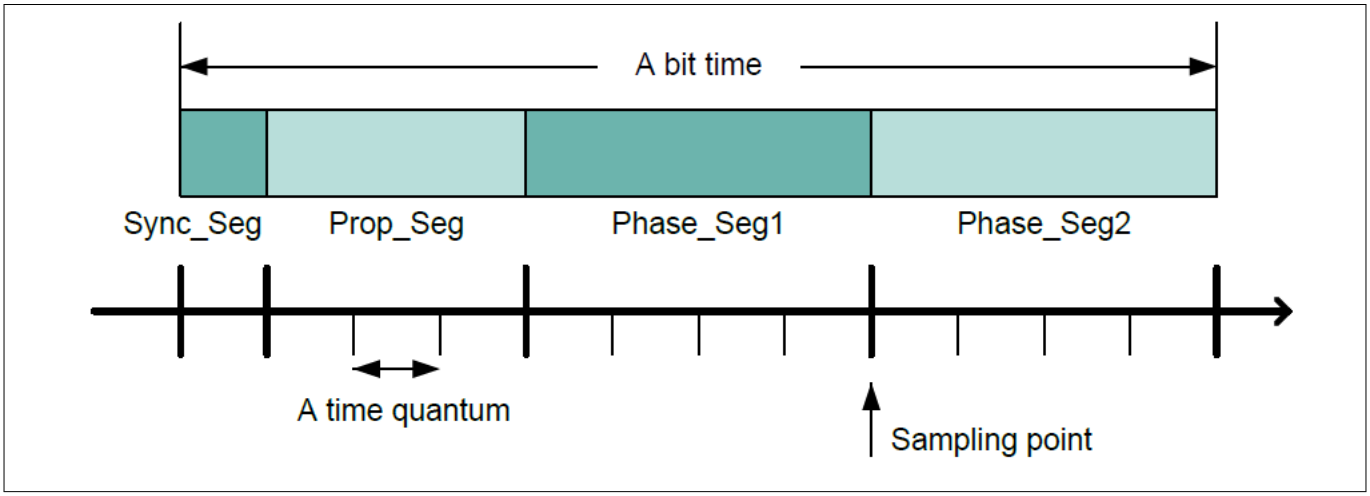


Figure 137 Bit time construction

Each segment is built from a programmable number of time quanta (TQ), with each TQ being a multiple of the time quantum set by `clk_can` and a prescaler. The parameters and prescalers for these segments differ between the nominal and data bit times. These configurations are defined in the Nominal Bit Timing and Prescaler Register (NBTP) and the Data Bit Timing and Prescaler Register (DBTP), as shown in Table 120.

Table 120 Bit time parameters

Parameter	Description
Time quantum tq (nominal) and tqd (data)	Time quantum. 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

(table continues...)

Table 120 (continued) Bit time parameters

Parameter	Description
Phase_Seg1	Phase_Seg1 is used to compensate for edge phase errors before the sampling point. Can be 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. Can be 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

The relationships for the nominal and data bit times result in the following equations:

Nominal bit time:

$$\text{Nominal Bit Time} = [\text{Sync_Seg} + \text{Prop_Seg} + \text{Phase_Seg1} + \text{Phase_Seg2}] \times tq = [1 + (\text{NBTP.NTSEG1}[7:0] + 1) + (\text{NBTP.NTSEG2}[6:0] + 1)] \times [(\text{NBTP.NBRP}[8:0] + 1) \times \text{clk_can period}] \quad (1)$$

Data bit time:

$$\text{Data Bit Time} = [1 + (\text{DBTP.DTSEG1}[4:0] + 1) + (\text{DBTP.DTSEG2}[3:0] + 1)] \times [(\text{DBTP.DBRP}[4:0] + 1) \times \text{clk_can period}] \quad (2)$$

***Note**: The information processing time (IPT) of the CAN FD controller is zero. This means that the data for the next bit becomes available at the first CAN clock edge after the sample point. Consequently, the IPT does not need to be considered when configuring Phase_Seg2, as it is defined as the greater of Phase_Seg1 or the IPT.*

23.3.7.2 CAN bit rates

The bit rate is the inverse of the bit time. As such, the nominal bit rate is expressed as:

$$\text{Nominal Bit Rate} = 1 / [\{1 + (\text{NBTP.NTSEG1}[7:0] + 1) + (\text{NBTP.NTSEG2}[6:0] + 1)\} \times \{(\text{NBTP.NBRP}[8:0] + 1) \times \text{clk_can period}\}] \quad (3)$$

Similarly, the data bit rate is expressed as:

$$\text{Data Bit Rate} = 1 / [\{1 + (\text{DBTP.DTSEG1}[4:0] + 1) + (\text{DBTP.DTSEG2}[3:0] + 1)\} \times \{(\text{DBTP.DBRP}[4:0] + 1) \times \text{clk_can period}\}] \quad (4)$$

From these formulas, it is clear that the bit rates of the CAN FD controller depend on the CAN clock (`clk_can`) period and the configurable ranges of each parameter. Figure 138 and Figure 139 provide examples of the configurable bit rates at varying CAN clock frequencies. Empty cells in the table indicate that the desired bit rate cannot be achieved with 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 138 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 139 Example configuration for data bit rates

***Note**: The user must configure the CAN bit timings in accordance with the relevant CAN standards to ensure proper communication on the CAN bus.*

23.4 Message RAM

The MRAM stores Rx/Tx messages and filter configurations.

***Note**:*

- *The MRAM should be cleared to zero before the 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 cannot be used for general purposes*

23.4.1 Message RAM configuration

The message RAM has a width of 32 bits. The CAN FD controller can be configured to allocate up to 4,480 words in the message RAM (MRAM), though the number of available words may be limited by the size of the actual MRAM. It is not necessary to configure every section listed in Figure 140, and there is no restriction regarding the sequence of the sections.

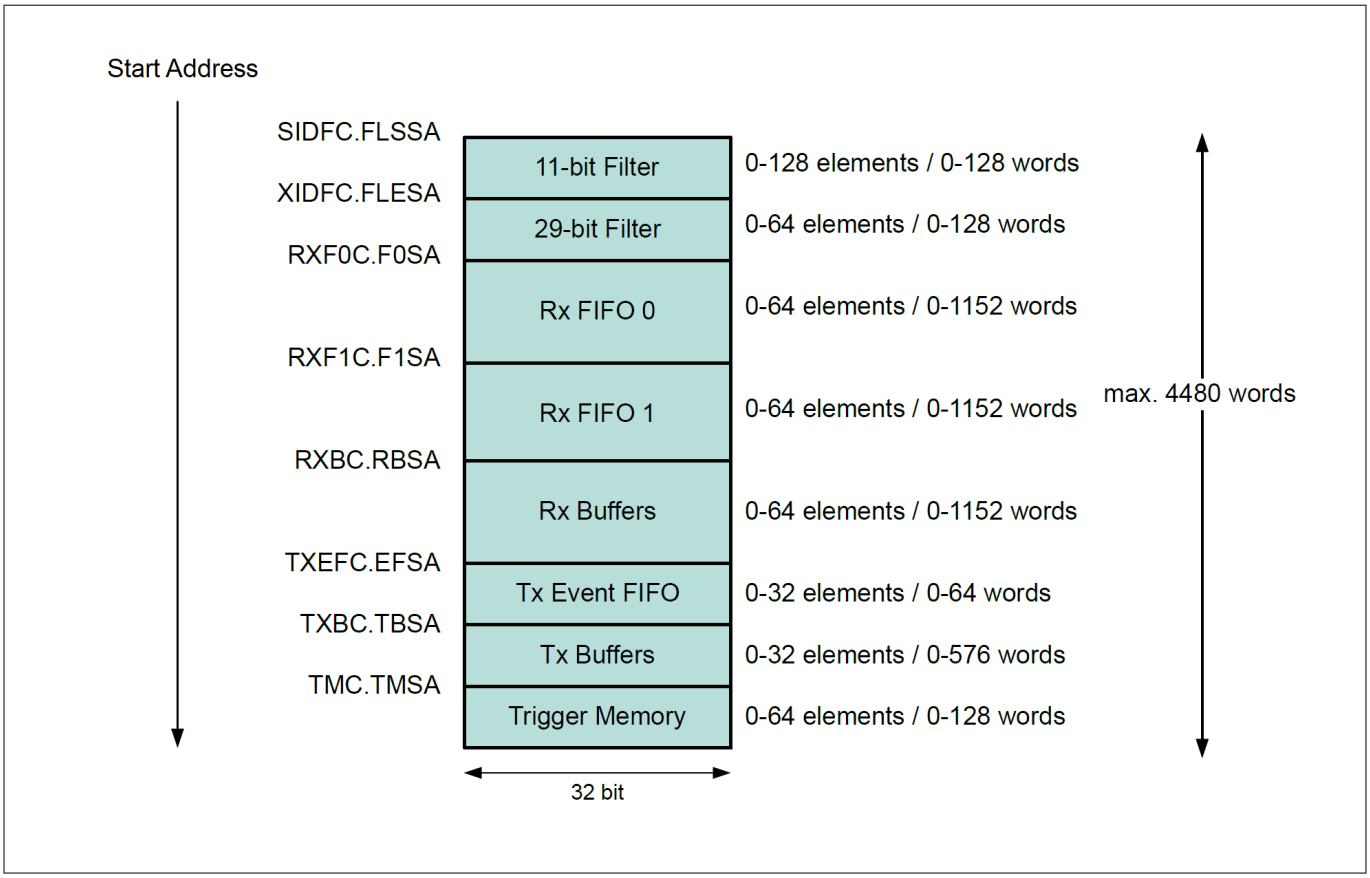


Figure 140 Message RAM configuration

The CAN FD controller addresses the message RAM in 32-bit words rather than single bytes. The configurable start addresses are 32-bit word addresses, where only bits 15 to 2 are evaluated, and the two least significant bits are ignored.

***Note**:*

- *The CAN FD controller does not perform error checking for incorrect configurations of the message RAM. Therefore, the start addresses of different sections and the number of elements in each section must be configured with care to prevent data corruption or loss*
- *The message RAM is accessible by both M_TTCAN and the CPU. A dynamic round-robin scheme is implemented to allocate access between these components*

23.4.2 Rx buffer and FIFO element

An Rx buffer and FIFO element is a block of 32-bit words that holds the data and status of a received frame stored in the message RAM. Up to 64 Rx buffers and two Rx FIFOs can be configured in the message RAM. Each Rx FIFO section can be configured to store up to 64 received messages.

The structure of an Rx buffer and FIFO element is shown in Figure 141. The element size can be configured to store CAN FD messages with up to a 64-byte data field using the RXESC register (Rx buffer/FIFO element size configuration).

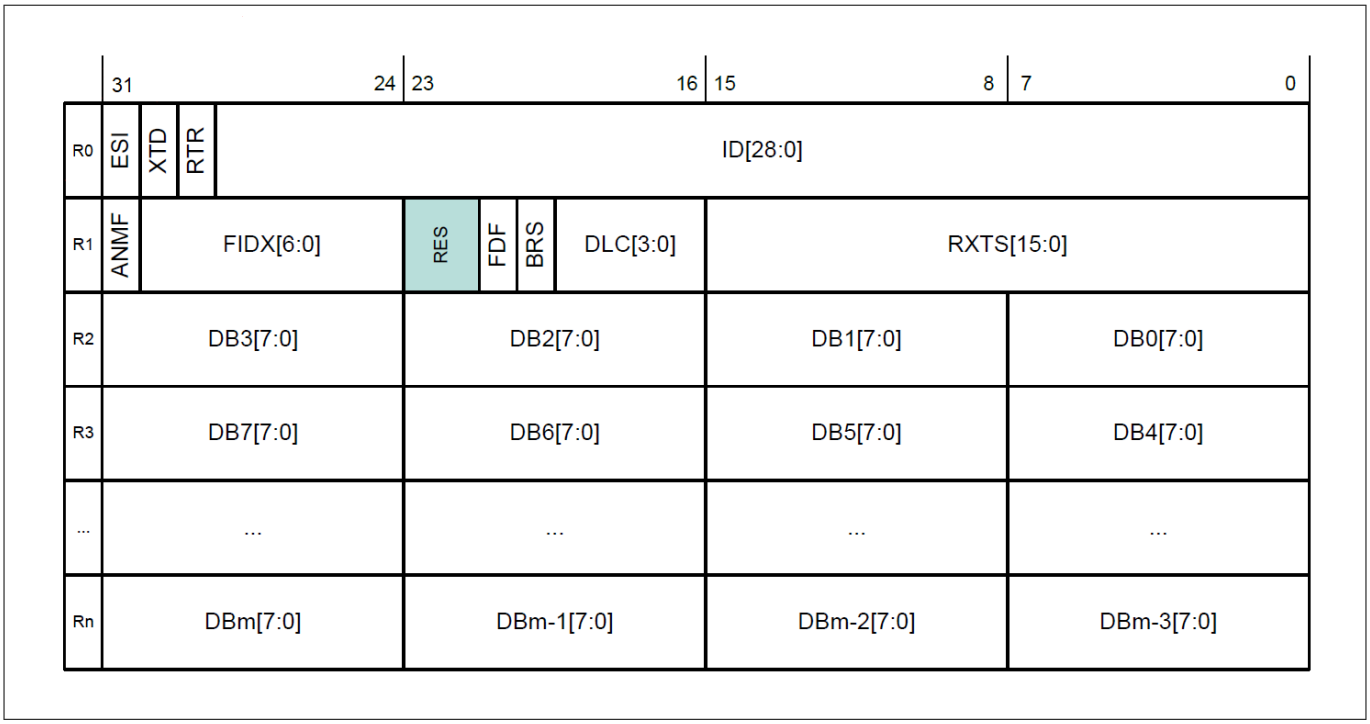


Figure 141 Rx buffer and FIFO

Table 121 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 contains a standard identifier or an extended identifier.

Table 122 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 123 R0[bit 29]: Remote transmission request

Bit	Description
0	Received frame is a data frame
1	Received frame is a remote frame

***Note**: TIn the CAN FD format, remote frames do not exist. For CAN FD frames (FDF = 1), the dominant remote request substitution (RRS) bit replaces the remote transmission request (RTR) bit.*

R0 [bit 28:0] ID[28:0]: Identifier: The identifier is either a standard or an extended identifier, depending on the value of the XTD bit. A standard identifier is stored in ID[28:18].

Table 124 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 can be enabled via GFC.ANFS[1:0] (Accept Non-Matching Frames Standard) and GFC.ANFE[1:0] (Accept Non-Matching Frames Extended).

Table 125 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 [bit 23:22] Reserved: Reserved bits: When writing, always write '0'. The read value is undefined.

Table 126 R1 [bit21] FDF: Extended data length

Bit	Description
0	Classic CAN frame format
1	CAN FD frame format

Table 127 R1 [bit20] BRS: Bit rate switch

Bit	Description
0	Frame received without bit rate switching
1	Frame received with bit rate switching

Table 128 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.

R1 [bit 15:0] RxTS[15:0]: Rx Timestamp: Specifies the timestamp counter value captured at the start of frame reception. The resolution depends on the configuration of the shared timestamp counter pre-scaler, defined by `TS_CTL.PRESCALE[15:0]`.

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
R3 [bit31:24]	DB7[7:0] :	Data Byte 7

R3 [bit23:16]	DB6[7:0] :	Data Byte 6
R3 [bit15:8]	DB5[7:0] :	Data Byte 5
R3 [bit7:0]	DB4[7:0] :	Data Byte 4
-	-	-
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 m-3

Note:
- *Depending on the configuration of the element size (defined by Rx buffer/FIFO element size configuration (RXESC)), the value of Rn will vary, ranging from n = 3 to n = 17*
- *The value of m is a function of n, calculated as m = (n - 1) × 4 - 1*
- *The number of valid data bytes is determined by the data length code*

23.4.3 Tx buffer element

A Tx buffer element is a block of 32-bit words stored in the message RAM that contains the data and control information of a frame to be transmitted by the CAN FD controller.

The Tx buffers section can be configured to hold:

1. Dedicated Tx buffers
2. A Tx FIFO or Tx queue

If the Tx buffers section is shared between dedicated Tx buffers and a Tx FIFO or Tx queue:

1. Dedicated Tx buffers start at the beginning of the Tx buffers section
2. Buffers assigned to the Tx FIFO or Tx queue follow immediately after the dedicated Tx buffers

The Tx handler distinguishes between dedicated Tx buffers and the Tx FIFO or Tx queue by evaluating:

1. `TXBC.TFQS[5:0]` - Transmit FIFO/Queue Size
2. `TXBC.NDTB[5:0]` - Number of dedicated transmit buffers

The element size can be configured to store CAN FD messages with a data field of up to 64 bytes through the `TXESC` register (Tx buffer element size configuration)

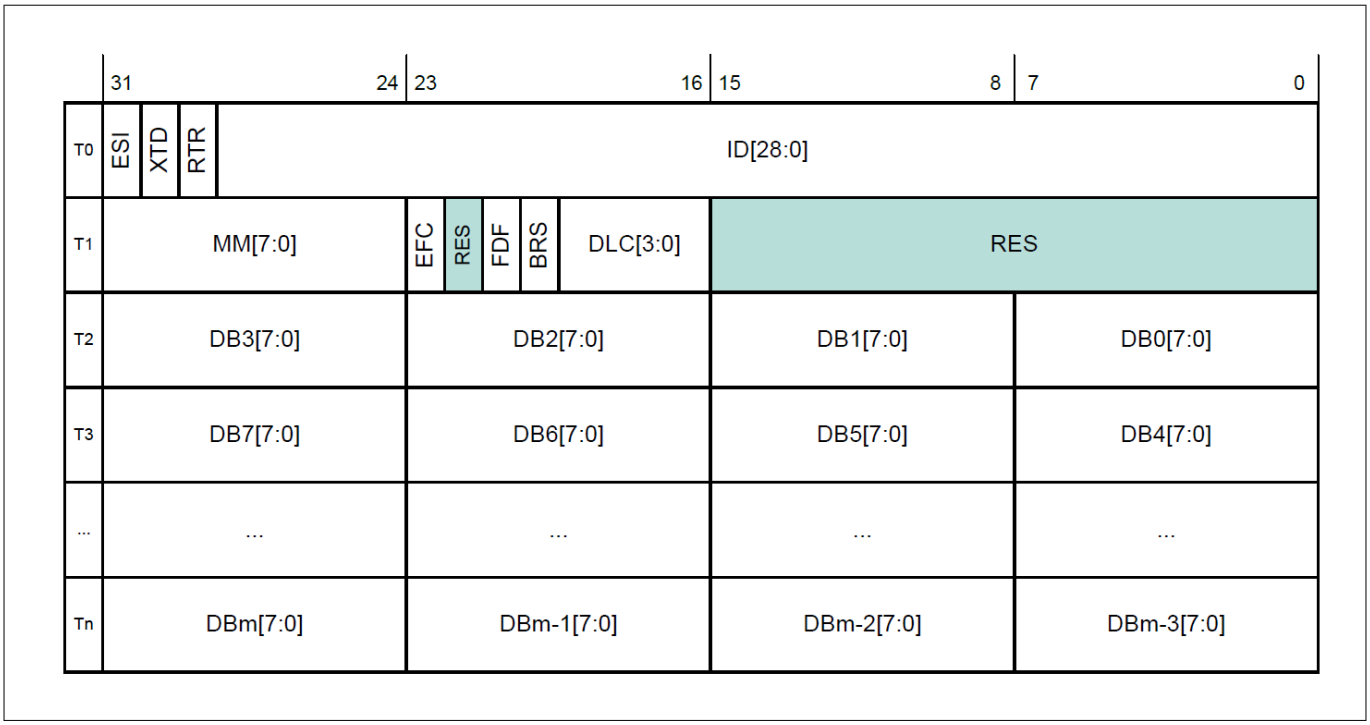


Figure 142 Tx buffer element

Table 129 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

***Notes**:*
The ESI bit of the transmit buffer is ORed with the error-passive flag to determine the value of the ESI bit in the transmitted FD frame.

As specified in the CAN FD protocol:

1. *An error-active node may optionally transmit the ESI bit as recessive*
2. *An error-passive node always transmits the ESI bit as recessive*

Table 130 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 contains a standard or extended identifier.

Table 131 R0 [bit29] RTR: Remote transmission request

Bit	Description
0	Transmit data frame
1	Transmit remote frame

When RTR = 1, the CAN FD controller transmits a remote frame according to ISO 11898-1, even if the FD Operation Enable (CCCR.FDOE) is set to enable transmission in CAN FD format.

T0 [bit 28:0] ID[28:0]: Identifier

- Can be a standard or extended identifier, depending on the XTD bit
- For a standard identifier, the value must be written to ID[28:18]

T1 [bit 31:24] MM[7:0]: Message Marker

- Written by the CPU during Tx buffer configuration
- Copied into the Tx Event FIFO element for identification of Tx message status

Table 132 T1 [bit23] EFC: Event FIFO control

Bit	Description
0	Don't store Tx events
1	Store Tx events

T1 [bit 22] Reserved: Reserved bit

Always write 0 when writing to this bit. The read value is undefined.

Table 133 T1 [bit21] FDF: FD format

Bit	Description
0	Frame transmitted in Classic CAN format
1	Frame transmitted in CAN FD format

Table 134 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 evaluated only when CAN FD operation is enabled (CCCR.FDOE = 1). Bit BRS is evaluated only when, in addition, bit CCCR.BRSE = 1.*

Table 135 T1 [bit19:16] DLC[3:0]: Data length code

DLC[3:0]	Description
0-8	Classic CAN + CAN FD: transmit frame has 0-8 data bytes.
9-15	Classic CAN: transmit frame has 8 data bytes.

CAN FD: Transmit frame supports 12, 16, 20, 24, 32, 48, or 64 data bytes.

T1 [bit 15:0] Reserved: Reserved bits

Always write 0 when writing to these bits. 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 7
T3 [bit23:16]	DB6[7:0] :	Data Byte 6
T3 [bit15:8]	DB5[7:0] :	Data Byte 5
T3 [bit7:0]	DB4[7:0] :	Data Byte 4
-	-	-
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 m-3

Note:
- *Depending on the configuration of the element size (TXESC), Tn will vary from n = 3 to n = 17*
- *The value of m is a function of n, defined as: m = (n – 1) × 4 – 1*

23.4.4 Tx event FIFO element

Each Tx event FIFO element stores information about transmitted messages. By reading the Tx event FIFO, the CPU obtains this information in the order in which the messages were transmitted.

Status information about the Tx event FIFO is available in the TXEFS register (Tx event FIFO status).

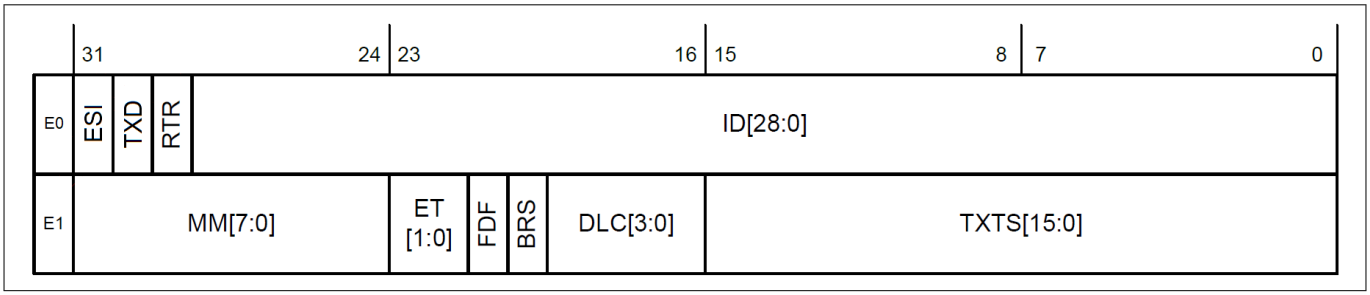


Figure 143 Tx event FIFO element

Table 136 E0 [bit31] ESI: Error state indicator

Bit	Description
0	Transmitting node is error active.
1	Transmitting node is error passive.

Table 137 E0 [bit30] XTD: Extended identifier

Bit	Description
0	11-bit standard identifier
1	29-bit extended identifier

Table 138 E0 [bit29] RTR: Remote transmission request

Bit	Description
0	Data frame transmitted.
1	Remote frame transmitted.

E0 [bit28:0] ID[28:0]: Identifier

E0 [bit 28:0]`ID[28:0]`: Identifier

Standard or extended identifier, depending on the `XTD` bit. A standard identifier is stored in `ID[28:18]`.

E1 [bit 31:24] `MM[7:0]`: Message Marker

Copied from the Tx buffer into the Tx Event FIFO element for identification of Tx message status.

Table 139 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 disable automatic retransmission (DAR) mode.
11	Reserved.

Table 140 E1 [bit21] FDF: FD format

Bit	Description
0	Classic CAN frame format.
1	CAN FD frame format (new DLC-coding and CRC).

Table 141 E1 [bit20] BRS: Bit rate switching

Bit	Description
0	Frame transmitted without bit rate switching.
1	Frame transmitted with bit rate switching.

Table 142 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.

E1 [bit 15:0] `TXTS[15:0]`: Tx timestamp counter value captured at the start-of-frame transmission. The resolution depends on the configuration of the shared timestamp counter pre-scaler `TS_CTL.PRESCALE[15:0]`.

23.4.5 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
- A dual filter
- A classic bit-mask filter
- A filter for a single dedicated ID

These filters are used 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 calculated as: SIDFC.FLSSA[15:2] (Filter List Standard Start Address) + index of the filter element (0 to 127).

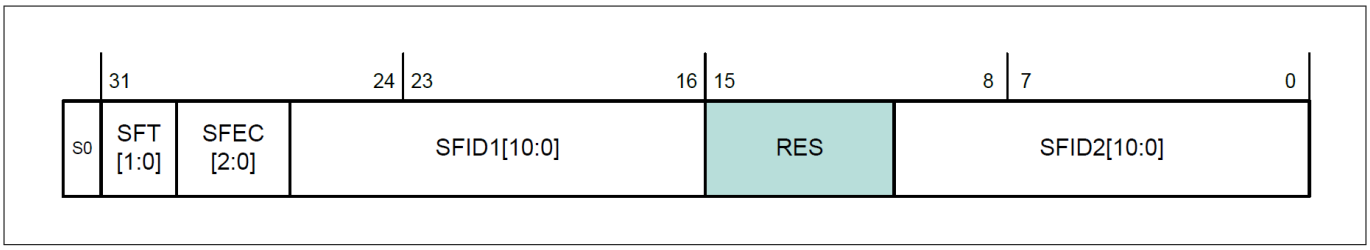


Figure 144 Standard message ID filter

Table 143 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 acceptance filtering continues. This is the same behavior as with SFEC = 000.*

S0 [bit 29:27]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 the interrupt flag IR.HPM (high priority message). If enabled, this generates an interrupt. In this case, the HPMS register (high priority message status) is updated with the status of the priority match.

Table 144 SFEC[2:0]: Set priority and store in RX FIFO

SFEC[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 SFT[1:0] ignored.

S0 [bit 26:16]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 exactly; no masking mechanism is used

S0 [bit 15:11] Reserved: Reserved bits

When writing, always write 0. The read value is undefined.

S0 [bit 10:0] 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 110. Set SFID2[10:0] according to the SFT[1:0] setting
- SFEC[2:0] = 111. Used for filtering dedicated Rx buffers or debug messages. SFID2[10:9] determines whether the received message is stored in a dedicated Rx buffer or treated as message A, B, or C of the debug message sequence

Table 145 SFID[10:9]] Store message into a dedicated Rx buffer

SFID2[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] used to store a matching message.

***Note**: A debug message is used to debug the CAN feature.*

23.4.6 Extended message ID filter element

An extended message ID filter element consists of two 32-bit words and can be configured as:

- A range filter
- A dual filter
- A classic bit-mask filter
- A filter for a single dedicated ID

These filters are used for messages with 29-bit extended IDs. Up to 64 filter elements can be configured for 29-bit extended IDs.

When accessing an extended message ID filter element, its address is calculated as: XIDFC.FLESA[15:2] (filter list extended start address) + 2 × index of the filter element (0 to 63).

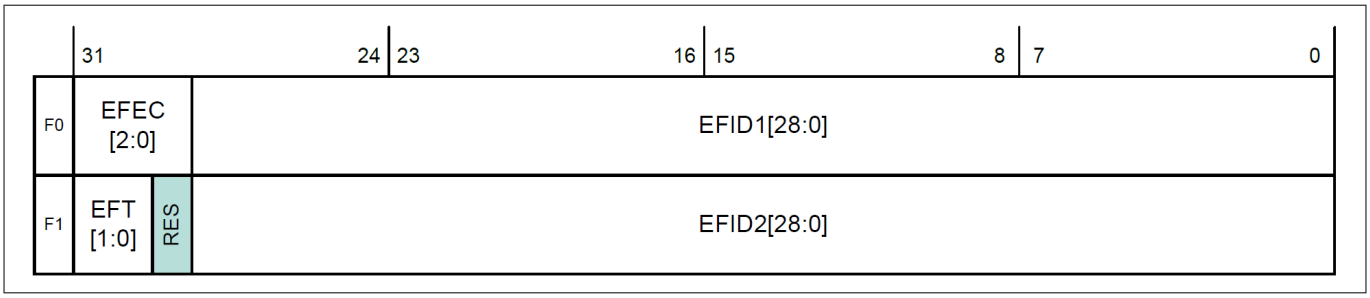


Figure 145 Extended message ID filter

F0 [bit 31:29]`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 the interrupt flag `IR.HPM` (high priority message). If enabled, this generates an interrupt. In this case, the `HPMS` register (high priority message status) is updated with the status of the priority match.

Table 146 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 [bit 28:0]`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] = 111`. `EFID1[28:0]` defines the ID of an extended dedicated Rx buffer or debug message to be stored. The received identifiers must match, and only the `XIDAM` masking mechanism is used

F1 [bit 31:30] `EFT[1:0]`: Extended Filter Type

Table 147 Extended filter type

EFT[1:0]	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.

(table continues...)

Table 147 (continued) Extended filter type

EFT[1:0]	Description
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 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 [bit 29] Reserved: Reserved bit

When writing, always write `0`. The read value is undefined.

F1 [bit 28:0] `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]` determines whether the received message is stored in a dedicated Rx buffer or treated as message A, B, or C of the debug message sequence.

Table 148 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**: A debug message is used to debug the CAN feature.*

23.4.7 Trigger memory element

Up to 64 trigger memory elements can be configured. When accessing a trigger memory element, its address is calculated as the trigger memory start address `TTTMC.TMSA` plus the index of the trigger memory element (`0` to `63`).

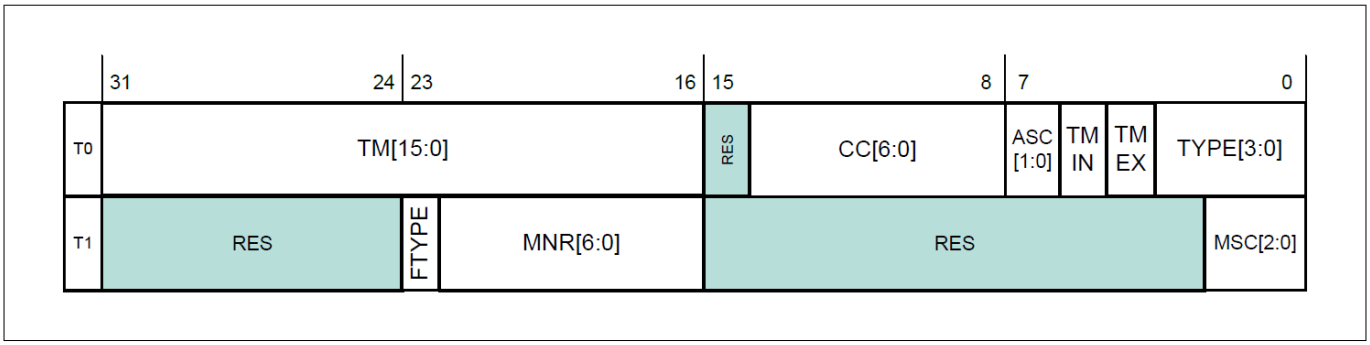


Figure 146 Trigger memory element

T0 [bit 31:16] `TM[15:0]`: Time Mark Cycle time at which the trigger becomes active.

Table 149 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 count mod16 = cccc.
01ccccc	Valid every thirty-second cycle at cycle count mod32 = ccccc.
1cccccc	Valid every sixty-fourth cycle at cycle count mod64 = cccccc.

Cycle count for which the trigger is valid. This value is ignored for trigger types `Tx_Ref_Trigger`, `Tx_Ref_Trigger_Gap`, `Watch_Trigger`, `Watch_Trigger_Gap`, and `End_of_List`.

Table 150 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 151 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 152 T0 Bit 5 TMIN: Time Mark Event External

TMIN	Description
0	No action
1	Pulse at output of Trigger Time Mark with the length of one PCLK_CANFD[x]_CLOCK_CAN[y] period is generated when the time mark of the trigger memory element becomes active and CANFDx_CHy_TTOCN.TTMIE = 1

Table 153 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.
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` or `ASC = 11`), only the trigger types `Rx_Trigger` and `Time_Base_Trigger` should be used*
- *ASC operation is not supported in this device*

Table 154 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 the configured Tx buffer number. Valid values are `0` to `31`
- **Reception**: Trigger is valid for the standard or extended message ID filter element number. Valid values are `0` to `63` and `0` to `127`

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 or in event-driven CAN communication (ISO 11898-1:2015).

Note:
- *The trigger memory elements should be written when the `M_TTCAN` is in the INIT state. Write access to the trigger memory elements outside the 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 `Tx_Ref_Trigger`. In this case, they become active at the time mark modified by the actual reference trigger offset (`TTOST.RTO`)*

ECC for message RAM

The error correcting code (ECC) function of the message RAM enables the detection and correction of data errors in the message RAM. The ECC implementation uses a 7-bit parity for a 32-bit data word.

It has the following features:

- Single error correction and double error detection (SECDED)
- Single-bit error correction in a memory data word
- Detection of single and double-bit errors in a memory data word
- Detection of errors in message RAM address decoding
- Stopping the CAN FD function upon detecting a double-bit (non-correctable) error
- Error injection during data transfer

The following sections describe the types of ECC errors.

Correctable ECC error

When a correctable ECC error is detected, the following will happen:

- The corrected data is returned to the read access master
- No error signal is returned to the master
- The corrected data is written back to the message RAM unless that write is canceled by another write to the same address
- The ECC error is reported in the fault structure as a correctable error with the following information:
  - `DATA0[15:0]`: Violating address
  - `DATA0[22:16]`: ECC syndrome `[6:0]`
  - `DATA0[27:24]`: Master ID (`0–7` = CAN channel ID, `8` = AHB I/F)

For security reasons, the violating data is not reported in the fault structure.

In the unlikely event that two correctable ECC errors occur too close together (before both fault reporting and correction writeback are complete), the second ECC error is neither corrected nor reported in the fault structure. If the same address is read again later, the correction and fault reporting will be performed at that time.

Non-correctable ECC error

When data is read from the message RAM and, during the ECC check, a double-bit error is detected, the following actions are taken:

- An error is reported to the master
- In the case of an AHB master, a bus error occurs
- In the case of an `M_TTCAN` channel, the channel shuts down immediately (`CANFDx_CHy_CCCR.INIT = 1`)
- An interrupt (`BEU`) is raised
- The ECC error is reported to the fault structure as a non-correctable error with the following information:
  - `DATA0[15:0]`: Violating address

  - DATA0[22:16]: ECC syndrome [6:0]
  - DATA0[27:24]: Master ID (0–7 = CAN channel ID, 8 = AHB I/F)
- For security reasons, data is not reported to the fault structure

Unlike a single-bit (correctable) error, a double-bit error is reported to both the master and the fault structures. The read access master needs to know that the read data is not correct, and the fault structure needs to log all ECC errors.

ECC errors on the address bits are always non-correctable.

Address error

An address error is detected when either an M_TTCAN channel or the MCU tries to access a message RAM address that is out of range (address >= MRAM_SIZE). This feature is provided to make software debugging easier.

Address error detection is independent of ECC; it works even if ECC is disabled.

When an address error is detected, the following behavior occurs:

- **For writes**: The error is not reported back to the master
  - Writes are posted, and both the AHB interface and the CAN channels ignore error signaling
- **For reads from an M_TTCAN channel**: The error is reported back to the channel as if it were a non-correctable ECC error. This results in:
  - Immediate shutdown of the channel (CANFDx_CHy_CCCR.INIT = 1) to prevent corrupt data from being sent
  - Raising of an interrupt (BEU)
- **For reads from the AHB interface**: The address error results in a bus error

In all cases read, write, and from any master the address error is reported in the fault structure as a non-correctable error with the following information:

- DATA0[15:0]: Violating address
- DATA0[27:24]: Master ID (0–7 = CAN channel ID, 8 = AHB I/F)
- DATA0[30]: Set for a write access; cleared for a read access
- DATA0[31]: Set to flag an address error

ECC error injection

For functional safety, the ECC error injection feature is provided to allow software to write the ECC bits. This enables software to trigger correctable or non-correctable ECC errors to verify that all related hardware and software components are functioning correctly.

Using this feature, software can inject an ECC error in the background while data is being fetched from the message RAM.

This feature consists of:

- An enable bit (CANFDx_ECC_ERR_INJ.ERR_EN) to enable the ECC error injection logic
- A 7-bit error parity (CANFDx_ECC_ERR_INJ.ERR_PAR) to be written instead of the generated ECC bits
- An address (CANFDx_ECC_ERR_INJ.ERR_ADDR) to specify the message RAM address where ECC error injection is to be performed

When a write is executed to the specified error injection address, the given error parity is used instead of the ECC parity generated by the ECC logic. By restricting this to a single address, the software can perform the functional test without affecting other message RAM accesses.

If a write-back occurs when a correctable error is detected at the specified error injection address, the ECC parity generated by the ECC logic is used.

As described in previous sections, detection of a non-correctable error will result in shutting down either a CAN channel or a CPU (bus error). To allow verification of non-correctable errors without impacting the running application, reporting of a non-correctable error to the master is suppressed for the targeted error injection address.

This error suppression applies to both ECC non-correctable errors and address errors. Software can disable this suppression by disabling the error injection logic (CANFDx_ECC_ERR_INJ.ERR_EN = 0).

***Note**: Depending on the application and use case, there may be a need to protect ECC error injection from unauthorized use. For more details, contact Infineon support.*

ECC parity generation by software

To inject an ECC error for fault generation, ECC parity must be generated by software. Follow this procedure to generate a 7-bit ECC parity.

```
CODEWORD_SW[63:0] = 64{1'b0};
CODEWORD_SW[31:0] = ACTUALWORD[31:0];
CODEWORD_SW[((x-1)+32):32] = ADDR[(x-1):0]; // where x = MRAM_ADDR_WIDTH

ECC_P0_SW =
64b00000011_01111111_00110110_11011011_00100010_01010100_00101010_10101011;
ECC_P1_SW =
64b00000101_10111101_11101011_01011010_01000100_10011001_01001101_00110101;
ECC_P2_SW =
64b00001001_11011101_11011100_11101110_00001000_11100010_01110001_11000110;
ECC_P3_SW =
64b00010001_11101110_10111011_10101001_10001111_00000011_10000001_11111000;
ECC_P4_SW =
64b00100001_11110110_11010111_01110101_11110000_00000011_11111110_00000000;
ECC_P5_SW =
64b01000001_11111011_01101101_10110100_11111111_11111100_00000000_00000000;
ECC_P6_SW =
64b10000001_00000011_11111111_11111000_00010001_00101100_10010110_01011111;
```

As shown above, the reduction XOR of the ANDed result of `CODEWORD_SW[63:0]` and the respective ECC constants produces a single parity bit.

```
parity[0] = ^ (CODEWORD_SW[63:0] & ECC_P0_SW)
parity[1] = ^ (CODEWORD_SW[63:0] & ECC_P1_SW)
...
parity[6] = ^ (CODEWORD_SW[63:0] & ECC_P6_SW)
```

`parity[6:0]` gives the seven-bit parity for the 32-bit `ACTUALWORD[31:0]`.

`MRAM_ADDR_WIDTH` in the procedure depends on the total message RAM allocated for one `M_TTCAN` group. Table 155 specifies the `MRAM_ADDR_WIDTH` for each message RAM size.

Table 155 ECC parity generation combinations

Parameter	Combinations											
Message RAM size [KB]	4	8	10	16	20	24	32	36	40	48	56	64
MRAM_ADDR_WIDTH	10	11	12	12	13	13	13	14	14	14	14	14

Message RAM OFF

Message RAM can be turned off to save power by setting the CANFDx_CTL.MRAM_OFF bit. The default value of this bit is 0, and in this configuration, message RAM is retained during Deep Sleep power mode.

All M_TTCAN channels must be powered down before setting the CANFDx_CTL.MRAM_OFF bit. See Power down (Sleep mode) to power down the M_TTCAN channels. When message RAM is off, any access to message RAM may raise an address error (MRAM_SIZE = 0).

After switching the message RAM on again, software must allow a certain power-up time before message RAM can be used, that is, before STOP_REQ can be de-asserted. Check the RAM_PWR_DELAY_CTL register to determine the required time for the message RAM power-up process.

23.4.8 RAM watchdog (RWD)

The RAM watchdog monitors the READY output of the message RAM. A message RAM access starts the RAM watchdog counter with the value configured in RWD.WDC. The counter is reloaded with RWD.WDC when the message RAM signals successful completion by activating its READY output.

If there is no response from the message RAM before the counter counts down to zero, the counter stops and the interrupt flag IR.WDI is set. The RAM watchdog counter is clocked by the host clock (clk_sys).

Refer to the register description section in the TRM for details about the RWD register.

23.5 Time-triggered CAN (TTCAN)

23.5.1 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 (the sender of the reference message).

1. For Level 1, the data length must be at least 1 byte
2. For Level 0 and Level 2, the data length must be at least 4 bytes
   - Otherwise, the message is not accepted as a reference message

The reference message may be extended with additional data, up to a total of 8 CAN data bytes.

All bits of the identifier except the three least significant bits (LSBs) characterize the message as a reference message. The last three bits specify the priorities of up to eight potential time masters. Reserved bits are transmitted as logical 0 and are ignored by receivers.

The reference message is configured using the TTRMC register.

- The time master transmits the reference message
- If the message is disturbed by an error, it is retransmitted immediately
- During retransmission, the transmitted Master_Ref_Mark is updated
- The reference message is sent periodically
- Periodic transmission may be stopped using the Next_is_Gap bit
- Event-synchronized transmission can be initiated at the start of the next basic cycle by the current time master or any potential time master

The node transmitting the reference message is the current time master. The time master is permitted to transmit other messages. If the current time master fails, its function is assumed by the potential time master with the highest priority.

Nodes that are neither the time master nor potential time masters are considered 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
- Information related to the reference message is stored in the first data byte, as shown in Table 156
- Cycle_Count is optional

Table 156 First data 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
- Information related to the reference message is stored in the first 4 data bytes, as shown in Table 157
- Cycle_Count and the lower 4 bits of NTU_Res are optional
- M_TTCAN does not evaluate NTU_Res[3:0] from received reference messages; it always transmits these bits as 0

Table 157 First data byte of Level 2 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
- Information related to the reference message is stored in the first 4 data bytes, as shown in Table 158
- In Level 0, Next_is_Gap is always 0
- Cycle_Count and the lower 4 bits of NTU_Res are optional
- M_TTCAN does not evaluate NTU_Res[3:0] from received reference messages; it always transmits these bits as 0

Table 158 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]							

23.5.2 TTCAN configuration

The network time unit (NTU) is the fundamental unit for measuring time across the network. It is a constant defined by the network system designer. In TTCAN Level 1, the NTU corresponds to the nominal CAN bit time. In contrast, for TTCAN Level 0 and Level 2, the NTU represents a fraction of a physical second. This unit serves as the time base for local time, where the 16-bit integer part of the local time counter increments once per NTU. Both cycle time and global time are derived from this local time. However, the 3-bit fractional parts of local time, cycle time, and global time are not readable.

In TTCAN Level 0 and Level 2, the NTU length is determined by the Time Unit Ratio (TUR), a non-integer value calculated using the formula:

$$TUR = TURNA.NAV/TURCF.DC \quad (5)$$

. The actual NTU duration is then derived as

$$NTU = CAN\ Clock\ Period \times TUR \quad (6)$$

. The TUR numerator configuration (NC) is an 18-bit value, where `TURCF.NCL[15:0]` can be programmed in the range `0x0000` to `0xFFFF`, and `TURCF.NCH17:16[1]()` is fixed at `0b01`. Writing a value `0xnnnn` to `TURCF.NCL[15:0]` results in `TURNA.NAV` being initialized to `0x10000 + 0x0nnnn = 0x1nnnn`. The TUR Denominator Configuration `TURCF.DC` is a 14-bit value, programmable between `0x0001` and `0x3FFF`, with `0x0000` being invalid.

To ensure proper operation, specific constraints must be met. In Level 1, the numerator `NC` must be at least four times the denominator (NC ≥ 4 × TURCF.DC). In Levels 0 and 2, the requirement is stricter: NC ≥ 8 × TURCF.DC, which supports the 3-bit resolution of the NTU's internal fractional part.

Upon a hardware reset, `TURCF.DC` is preset to `0x1000` and `TURCF.NCL` to `0x10000`, resulting in an NTU equivalent to 16 CAN clock periods. Local time and the application watchdog do not start until either `CCCR.INIT` is cleared or `TURCF.ELT` is set. Importantly, `TURCF.ELT` must not be set before the NTU is configured. Once `TURCF.ELT` is set to 1, it locks write access to the `TURCF` register.

At system start-up, `TURNA.NAV` is updated from `NC` when `TURCF.ELT` is set. In Level 1, there is no drift compensation, and `TURNA.NAV` remains constant, always equal to `NC`. However, in Levels 0 and 2, `TURNA.NAV` can change under two conditions. First, when the node operates as a time slave or backup time master and `TTOCF.ECC` is set, `TURNA.NAV` is automatically updated based on the monitored global time speed, provided the `M_TTCAN` is in synchronization states `In_Schedule` or `In_Gap`. If synchronization is lost, `TURNA.NAV` reverts to `NC`.

Second, when the node functions as the actual time master and `TTOCF.EECS` is set, the host can update `TURCF.NCL`. Setting `TTOCN.ECS` causes `TURNA.NAV` to be updated from the new `NC` value at the next reference message. The status flag `TTOST.WECS` is set when `TTOCN.ECS` is asserted and is cleared once the update is complete. During this time, `TURCF.NCL` is write-locked.

Clock calibration in Levels 0 and 2 adjusts `TURNA.NAV` within the synchronization deviation limit (SDL), defined as SDL = NC ± 2^(TTOCF.LDSDL + 5). For optimal accuracy in calculating `TURNA.NAV`, `TURCF.NCL` should be programmed with the highest suitable value. The synchronization deviation (SD), calculated as SD = NC - TURNA.NAV, must remain within the SDL and should not exceed the clock tolerance defined by the CAN bit timing. This deviation is recalculated at the start of each basic cycle.

If `TURNA.NAV` deviates beyond the SDL or if the `Disc_Bit` in the reference message is set, drift compensation is suspended. In such cases, `TTIR.GTE` is set, `TTOSC.QCS` is reset, and if `Disc_Bit = 1`, `TTIR.GTD` is also set. Additionally, `TTOCN.ECS` schedules `NC` for activation at the next reference message, while `TTOCN.SGT` schedules `TTGTP.TP` for activation. Both require `TTOCF.EECS` to be set and the `M_TTCAN` to be the actual time master.

Finally, the `M_TTCAN` module includes an application watchdog to monitor and verify the correct operation of the application program.

Table 159 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

In a time-triggered CAN (TTCAN) system, precise timing and synchronization are critical. To support this, the TTOCN register provides mechanisms to schedule key timing parameters for activation at the next reference message. Specifically, setting TTOCN.ECS schedules the numerator configuration (NC) to be activated with the next reference message. Similarly, setting TTOCN.SGT schedules the trigger time point (TTGTP.TP) for activation at the same moment. These scheduling actions are only valid when external clock synchronization is enabled, which is indicated by setting TTOCF.EECS. Additionally, the node must be operating as the actual time master for these settings to take effect.

To ensure the reliability of the application software running alongside the TTCAN controller, the M_TTCAN module includes an application watchdog. This watchdog monitors whether the host application is functioning correctly. The host must regularly "serve" or reset the watchdog; otherwise, all CAN bus activity is halted to prevent potential system faults or unsafe behavior.

The watchdog's timeout period is defined in terms of NTUs and is configured using the TTOCF.AWL field, which sets the application watchdog limit. The maximum allowable value is 256 NTUs. To serve the watchdog, the host must read the TTOST register. The bit TTOST.AWE indicates whether the watchdog was served in time. If the application fails to serve the watchdog within the configured limit, the interrupt flag TTIR.AW is set, signaling a watchdog timeout.

For development and debugging purposes, the application watchdog can be disabled by setting TTOCF.AWL to 0x00. This allows engineers to test and iterate on application logic without triggering unintended watchdog resets during early development stages.

23.5.2.1 TTCAN timing

23.5.2.1.1 Message scheduling

The TTOCF.TM field determines whether the M_TTCAN operates as a potential time master or as a time slave. When configured as a potential time master, the three least significant bits (LSBs) of the reference message identifier (TTRMC.RID) define the master priority, where 0 is the highest and 7 is the lowest. Each potential time master in the network must have a unique priority value. The TTRMC.RID field is also used to recognize reference messages. The TTRMC.RMPS field is not relevant for time slaves.

The initial reference trigger offset (TTOCF.IRTO) is a 7-bit value that defines, in NTUs, how long a backup time master waits before transmitting a reference message when one is expected but the bus remains idle. The recommended value for TTOCF.IRTO is calculated as the master priority multiplied by a factor that accounts for the expected clock drift between potential time masters. This ensures that backup time masters begin transmission in the correct order, even under maximum clock drift conditions.

The TTOCF.OM field selects the TTCAN operation level. All potential time masters in a network must operate at the same level. Time slaves may operate at Level 1 in a Level 2 network, but not the other way around. Configuring the operation mode via TTOCF.OM is the final step in TTCAN setup. The operation modes are defined as follows:

- TTOCF.OM = 00: Event-driven CAN communication (ISO 11898-1:2015), without time-triggered features
- TTOCF.OM = 01: TTCAN Level 1 (ISO 11898-4), without external event synchronization; the Next_is_Gap bit is ignored

- TTOCF.OM = 10: TTCAN Level 2 (ISO 11898-4), with support for event-synchronized basic cycle start
- TTOCF.OM = 11: TTCAN Level 0, which operates as event-driven CAN but maintains a calibrated global time base similar to Level 2

The TTOCF.EECS bit enables external clock synchronization, allowing the application program of the current time master to update the TUR configuration during time-triggered operation. This enables clock speed adaptation and, in Level 0 and Level 2, also allows adjustment of the global clock phase to an external reference.

The TTMLM.ENTT field in the TT matrix limits register specifies the number of expected Tx_Triggers in the system matrix. This count includes triggers for exclusive single arbitrating and merged arbitrating windows but excludes Tx_Ref_Triggers. It is not necessarily equal to the number of Tx_Trigger memory elements. The actual count depends on the number of basic cycles in the system matrix and the repeat factors of the triggers. Incorrect configuration of TTMLM.ENTT may result in:

- **Tx count underflow**: TTIR.TXU = 1 and TTOST.EL = 01 (severity level 1)
- **Tx count overflow**: TTIR.TXO = 1 and TTOST.EL = 10 (severity level 2)

***Note**: If the first reference message received by a node does not have a Cycle_Count of zero, the node may complete its first matrix cycle with a Tx count that causes a Tx count underflow. While the node is in the synchronization state, triggering transmissions is not permitted.*

The TTMLM.CCM field defines the number of the last basic cycle in the system matrix. Basic cycle counting starts at 0. For example, in a system matrix with eight basic cycles, TTMLM.CCM would be set to 7. This field is ignored by time slaves. A time slave uses the Cycle_Count from the received reference message as the valid count for the current basic cycle.

The TTMLM.TXEW field specifies the Tx enable window length in NTUs. This window defines the period at the beginning of a time window during which a transmission may be initiated. If the sample point of the first bit of a transmit message falls outside this window, the transmission cannot start in that time window. This may occur, for example, due to overlap from a message in the previous time window. The value of TTMLM.TXEW should be selected based on the network's synchronization quality and the relationship between time window lengths and message durations.

23.5.2.2 Trigger memory

The trigger memory is a part of the message RAM and can store up to 64 trigger elements. Each trigger memory element consists of the following fields:

- 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)
- Asynchronous Serial Communication (ASC)

The time mark defines the specific cycle time at which a trigger becomes active. All trigger elements in the trigger memory must be sorted in ascending order based on their time marks. The trigger element with the lowest time mark is stored in the first trigger memory word. For trigger types such as Tx_Ref_Trigger, Tx_Ref_Trigger_Gap, Watch_Trigger, Watch_Trigger_Gap, and End_of_List, the message number and cycle code fields are ignored.

When the cycle time reaches the time mark of the current trigger, the FSE switches to the next trigger and begins reading it from the trigger memory. For a transmit trigger, the Tx handler starts reading the

corresponding message from the message RAM as soon as the FSE activates the trigger. The RAM access speed determines the minimum time interval required between a transmit trigger and its preceding trigger. This ensures that the Tx handler has sufficient time to prepare the message for transmission before the trigger's time mark is reached.

The RAM access speed also limits the number of non-matching triggers (in terms of cycle code) that can exist between two matching triggers. The next matching trigger must be read before its time mark is reached. If the reference message has a length of 'n' NTUs, any trigger with a time mark less than 'n' will never become active and is treated as a configuration error.

The starting point of the cycle time is the sample point of the start-of-frame bit of the reference message. A new reference message is requested when the cycle time reaches the time mark of the `Tx_Ref_Trigger`. The `M_TTCAN` responds to this transmission request at the next sample point. A new `Sync_Mark` is captured at the start-of-frame bit, but the cycle time continues to increment until the reference message is successfully transmitted or received. At that point, the `Sync_Mark` is taken as the new `Ref_Mark`, and the cycle time is restarted.

As a result, the cycle time can never be observed at a value less than 'n', where 'n' is the length of the reference message in NTUs, except during initialization. The total length of a basic cycle is calculated as:

$$\text{Length of a basic cycle} = \text{Tx_Ref_Trigger time mark} + 1\ \text{NTU} + 1\ \text{CAN bit time} \quad (7)$$

The trigger list is unique for each node in the TTCAN network. Each node is aware only of the `Tx_Triggers` for its own transmit messages, the `Rx_Triggers` for the receive messages it processes, and the triggers related to reference messages.

23.5.2.2.1 Trigger types

`Tx_Ref_Trigger` (TYPE = 0000) and `Tx_Ref_Trigger_Gap` (TYPE = 0001) initiate the transmission of a reference message by a time master. A configuration error is detected (`TTOST.EL = 11`, severity 3) if a time slave encounters either of these trigger types in its trigger memory. `Tx_Ref_Trigger_Gap` is used only in external event-synchronized time-triggered operation mode. In this mode, `Tx_Ref_Trigger` is ignored when the `M_TTCAN` synchronization state is `In_Gap` (`TTOST.SYS = 10`).

The following trigger types initiate message transmissions and define the start of a time window:

- `Tx_Trigger_Single` (TYPE = 0010)
- `Tx_Trigger_Continuous` (TYPE = 0011)
- `Tx_Trigger_Arbitration` (TYPE = 0100)
- `Tx_Trigger_Merged` (TYPE = 0101)

`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, this bit is cleared.

`Tx_Trigger_Continuous` also starts a transmission in an exclusive time window when the Transmission Request Pending bit is set. However, after successful transmission, the bit remains set, allowing the message to be transmitted again in the next matching time window.

`Tx_Trigger_Arbitration` initiates an arbitrating time window. `Tx_Trigger_Merged` starts a merged arbitrating time window. The last trigger in a merged arbitrating time window must be of type `Tx_Trigger_Arbitration`. A configuration error (`TTOST.EL = 11`, severity 3) is detected if a `Tx_Trigger_Merged` is followed by any trigger type other than `Tx_Trigger_Merged` or `Tx_Trigger_Arbitration`.

Multiple `Tx_Triggers` may be defined for the same transmit message buffer. Depending on their cycle code, some triggers may be ignored in specific basic cycles. The cycle code must be considered when calculating the expected number of `Tx_Triggers` (`TTMLM.ENTT`).

`Watch_Trigger` (TYPE = 0110) and `Watch_Trigger_Gap` (TYPE = 0111) are used to detect missing reference messages. These triggers are applicable to both time masters and time slaves. `Watch_Trigger_Gap` is used only in

external event-synchronized time-triggered operation mode. In this mode, Watch_Trigger is ignored when the M_TTCAN synchronization state is In_Gap (TTOST.SYS = 10).

Rx_Trigger (TYPE = 1000) is used to monitor the reception of periodic messages in exclusive time windows. Rx_Triggers become active only when the node enters the In_Schedule or In_Gap state. The time mark of an Rx_Trigger should be placed after the expected end of the message transmission, regardless of time window boundaries.

At the Rx_Trigger time mark, the system checks whether the last received message—received after the start of the cycle or the previous Rx_Trigger matches the acceptance filter element referenced by the MNR field. Accepted messages are stored in one of two receive FIFOs based on acceptance filtering, independent of the Rx_Trigger. To ensure timely filtering, acceptance filter elements referenced by Rx_Triggers should be placed at the beginning of the filter list.

Time_Base_Trigger (TYPE = 1001) is used to generate internal or external events, depending on the configuration of the ASC, TMIN, and TMEX fields.

Trigger types from TYPE = 1010 to 1111 are considered End_of_List and are illegal. A configuration error (TTOST.EL = 11, severity 3) is detected if an End_of_List trigger appears in the trigger memory before a Watch_Trigger or Watch_Trigger_Gap.

23.5.2.2.2 Restrictions for the node's trigger list

Two triggers must not be active at the same cycle time and cycle count. However, triggers that are active in different basic cycles (with different cycle codes) may share the same time mark.

Rx_Triggers and Time_Base_Triggers must not be placed within the TX enable windows of Tx_Trigger_Single, Tx_Trigger_Continuous, or Tx_Trigger_Arbitration. They may, however, be placed after a Tx_Trigger_Merged.

Triggers placed after a Watch_Trigger or a Watch_Trigger_Gap (when TTOST.SYS = 10) will never become active. The watch triggers themselves will not become active if reference messages are transmitted on time.

All unused trigger memory words following the Watch_Trigger or 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 begins with a sequence of Tx_Triggers and Rx_Triggers, followed by the Tx_Ref_Trigger and Watch_Trigger. In networks using external event-synchronized time-triggered communication, this list is extended with the Tx_Ref_Trigger_Gap and Watch_Trigger_Gap. The trigger list for a time slave is similar but excludes the Tx_Ref_Trigger and Tx_Ref_Trigger_Gap.

At the beginning of each basic cycle triggered by the reception or transmission of a reference message the trigger list is processed starting from the first trigger memory element. The FSE searches for the first trigger with a cycle code matching the current cycle count. It waits until the cycle time reaches the trigger's time mark and then activates the trigger. After activation, the FSE continues scanning the list for the next matching trigger.

Special attention is required around the time marks of Tx_Ref_Trigger and Tx_Ref_Trigger_Gap. In a time master competing for mastership, the effective time mark of a Tx_Ref_Trigger may be decremented to allow the node to be the first to transmit 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 becomes the sum of the configured time mark and 0x127. No other trigger elements should be placed within this range. Otherwise, time marks may appear out of order and will be flagged as a configuration error. Trigger elements placed after a Tx_Ref_Trigger may never become active if reference messages are received on time.

There are interdependencies 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

23.5.2.2.3 Examples of trigger handling

The following example illustrates how a trigger list is derived from a node's system matrix. Assume that node A is the initial time master. A section of the system matrix is shown in Table 160.

Table 160 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
3	Tx7	-	Rx5	-	-	TxRef	Error
4	Tx7	-	-	Rx6	-	TxRef	Error

The cycle count starts at 0 and follows a pattern based on the number of basic cycles in the system matrix. For example, the cycle counts are 0, 1, 3, 7, 15, 31, and 63, corresponding to system matrices with 1, 2, 4, 8, 16, 32, and 64 basic cycles respectively. The maximum cycle count is configured using the `TTMLM.CCM` register.

The cycle code (CC) is composed of two parts: the repeat factor, which is determined by the position of the most significant '1' bit, and the number of the first basic cycle, which is represented by the bits following the most significant '1'.

Example: If `CC = 0b0010011`, the repeat factor is 16 (based on the most significant '1'), and the first basic cycle is 3. With a maximum cycle count of `TTMLM.CCM = 0x3F`, this configuration results in matches at cycle counts 3, 19, 35, and 51.

A trigger element consists of the following fields:

- Time Mark (TM)
- Cycle Code (CC)
- Trigger Type (TYPE)
- Message Number (MNR)

For transmission, the message number references the transmit buffer number (0 to 31). For reception, it references the acceptance filter element number (0 to 127) that matched during filtering. Depending on the filter type (FTYPE), either the 11-bit or 29-bit message ID filter list is used.

A trigger element can also be configured for:

- Asynchronous Serial Communication (ASC)
- Time Mark Event Internal (TMIN)
- Time Mark Event External (TMEX)

The message status count (MSC) holds a counter value from 0 to 7. It is used to track scheduling errors for periodic messages in exclusive time windows when the trigger element becomes active.

Table 161 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

(table continues...)

Table 161 (continued) 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]
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	Mark 4	1000001	Tx_Trigger_Arbitration	4
6	Mark 4	0b1000100	Rx_Trigger	6
7	Mark 6	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_Continuous`, `Tx_Trigger_Merged`, `Tx_Trigger_Arbitration`, `Rx_Trigger`, and `Time_Base_Trigger` are valid only when their cycle code matches the current cycle count. For all other trigger types, the cycle code is ignored.

The FSE begins each basic cycle by scanning the trigger list from the beginning. It continues until it encounters one of the following conditions:

- A trigger with a time mark greater than the current cycle time
- A trigger whose cycle code matches the current cycle count
- A trigger of type `Tx_Ref_Trigger`, `Tx_Ref_Trigger_Gap`, `Watch_Trigger`, or `Watch_Trigger_Gap`

When the cycle time reaches the time mark of a trigger, the action defined by its type and message number is executed. If the FSE encounters a trigger of type `End_of_List`, it indicates a configuration error.

At Mark6, the reference message (always of type `Tx_Ref_Trigger`) is transmitted. After this transmission, the FSE returns to the beginning of the trigger list. When it reaches the `Watch_Trigger` at Mark7 and the node is unable to transmit the reference message, error handling is initiated.

23.5.2.2.4 Detection of configuration errors

A configuration error is signaled by setting `TTOST.EL = 11` (severity 3) under the following conditions:

- The FSE encounters a trigger in the list with a cycle code that matches the current cycle count, but the trigger's time mark is less than the current cycle time
- The previously active trigger was a `Tx_Trigger_Merged`, and the FSE encounters a trigger with a matching cycle code that is not one of the following:
  - `Tx_Trigger_Merged`
  - `Tx_Trigger_Arbitration`
  - `Time_Base_Trigger`
  - `Rx_Trigger`
- The FSE of a node configured as a time slave (`TTOCF.TM = 0`) encounters a `Tx_Ref_Trigger` or a `Tx_Ref_Trigger_Gap`

- A time mark is placed inside the Tx enable window (defined by TTMLM.TXEW) of a Tx_Trigger with a matching cycle code
- A time mark is placed close to the time mark of a Tx_Ref_Trigger, and the reference trigger offset (TTOST.RTO) causes their order to reverse when measured in cycle time

23.5.2.3 M_TTCAN schedule initialization

Synchronization to the M_TTCAN message schedule begins when CCCR.INIT is reset. The M_TTCAN can operate in either time-triggered mode (TTOCF.GEN = 0) or external event-synchronized time-triggered mode (TTOCF.GEN = 1).

At startup, all nodes begin with a cycle time of zero and start processing their trigger list from the beginning. The synchronization state is initially set to TTOST.SYS = 00 (out of synchronization), during which no transmissions are enabled except for the reference message.

In external event-synchronized time-triggered operation mode, nodes ignore Tx_Ref_Trigger and Watch_Trigger. Instead, they rely on Tx_Ref_Trigger_Gap and Watch_Trigger_Gap until the first reference message determines whether a gap is active.

23.5.2.3.1 Time slaves

After configuration, a time slave will ignore its Watch_Trigger and Watch_Trigger_Gap if no message is received before reaching the watch triggers. When the node reaches the internal Init_Watch_Trigger, the interrupt flag TTIR.IWT is set, the FSE is frozen, and the cycle time becomes invalid. Despite this, the node can still participate in CAN bus communication by acknowledging frames or sending error flags.

***Note**: The Init_Watch_Trigger is not part of the trigger list. It is implemented as an internal counter that counts up to 0xFFFF, which represents the maximum cycle time.*

If a time slave receives any message other than a reference message before reaching the watch triggers, it assumes a fatal error (TTOST.EL = 11, severity 3). In this case, the interrupt flag TTIR.WT is set, the node disables its CAN bus output, and it enters bus monitoring mode by setting CCCR.MON = 1. In this mode, the node can still receive messages but cannot transmit dominant bits and therefore cannot acknowledge frames.

***Note**: To exit the fatal error state, the host must set CCCR.INIT = 1. After resetting CCCR.INIT, the node restarts TTCAN communication.*

If no error occurs during synchronization, the first received reference message sets the synchronization state to TTOST.SYS = 01 (synchronizing). The second reference message then determines the final synchronization state based on the Next_is_Gap bit. If the bit is not set, the state becomes TTOST.SYS = 11 (In_Schedule); if the bit is set, the state becomes TTOST.SYS = 10 (In_Gap). In either case, all Tx_Triggers and Rx_Triggers are enabled.

23.5.2.3.2 Potential time master

After configuration, a potential time master begins transmitting a reference message when it reaches its Tx_Ref_Trigger, or Tx_Ref_Trigger_Gap if operating in external event-synchronized time-triggered mode. If the node does not receive any message or fails to transmit the reference message successfully before reaching the watch triggers, it ignores both Watch_Trigger and Watch_Trigger_Gap. This condition typically occurs when all other nodes are still in reset or configuration and do not acknowledge the transmission.

When the node reaches the internal Init_Watch_Trigger, the attempted transmission is aborted. The interrupt flag TTIR.IWT is set, the FSE is frozen, and the cycle time becomes invalid. Despite this, the node can still participate in CAN bus communication by acknowledging frames or sending error flags. Resetting TTIR.IWT re-enables the transmission of reference messages until the next occurrence of the Init_Watch_Trigger condition or until another CAN message is received. The FSE is restarted upon reception of a reference message.

If a potential time master reaches the watch triggers after receiving any message other than a reference message, it assumes a fatal error (TTOST.EL = 11, severity 3). In this case, the interrupt flag TTIR.WT is set, the node disables its CAN bus output, and it enters bus monitoring mode by setting CCCR.MON = 1. In this mode, the node can still receive messages but cannot transmit dominant bits and therefore cannot acknowledge frames.

If no error is detected during initialization, the first reference message sets the synchronization state to TTOST.SYS = 01 (synchronizing). The second reference message determines the final synchronization state based on the Next_is_Gap bit. If the bit is not set, the state becomes TTOST.SYS = 11 (In_Schedule). If the bit is set, the state becomes TTOST.SYS = 10 (In_Gap). In either case, all Tx_Triggers and Rx_Triggers are enabled.

A potential time master becomes the current time master (TTOST.MS = 11) when it is the transmitter of the most recent reference message. Otherwise, it functions as a backup time master (TTOST.MS = 10).

Once all potential time masters have completed their configuration, the node with the highest time master priority in the network assumes the role of the current time master.

23.5.3 TTCAN gap control

All functions related to gap control apply only when the M_TTCAN operates in external event-synchronized time-triggered mode (TTOCF.GEN = 1). In this mode, the TTCAN message schedule can be interrupted by inserting gaps between the basic cycles of the system matrix. All nodes connected to the CAN network must be configured for external event-synchronized time-triggered operation.

During a gap, all transmissions are halted, and the CAN bus remains idle. A gap ends when the next reference message initiates a new basic cycle. The gap begins at the end of a basic cycle that was started by a reference message with the Next_is_Gap bit set to 1. Gaps are typically initiated by the current time master.

The current time master can initiate a gap in two ways:

- Under software control by writing TTOCN.NIG = 1. The Next_is_Gap bit will be transmitted as 1 in the next reference message
- Under hardware control by writing TTOCN.GCS = 1. If a reference message is started while TTOCN.GCS is set, the Next_is_Gap bit will be set to 1

Once the reference message is transmitted, the TTOST.WFE bit signals the start of the gap to both the time master and time slaves. The current basic cycle continues until its final time window. The time following the last time window is considered the gap time.

For the actual time master and potential time masters, the TTOST.GSI bit is set when the last basic cycle ends and the gap time begins. In time slave nodes, the TTOST.GSI bit remains cleared.

When a potential time master is in the In_Gap synchronization state (TTOST.SYS = 10), it has four options to intentionally end the gap:

- Under software control by writing TTOCN.FGP = 1
- Under hardware control by writing TTOCN.GCS = 1. In this case, TTOCN.FGP is automatically set when a falling edge is detected at the internal event trigger input pin, which restarts the schedule
- Time-triggered restart. When TTOCN.TMG = 1, the next register time mark interrupt (TTIR.RTMI = 1) sets TTOCN.FGP and starts the reference message
- Automatically, when the node reaches its Tx_Ref_Trigger_Gap, assuming the external synchronization event did not occur in time

None of these options can interrupt an ongoing basic cycle with a reference message.

If TTOCN.FGP is set after the gap time has started, the reference message is transmitted immediately, resuming the message schedule. If TTOCN.FGP is set before the gap time begins (while the basic cycle is still in progress), the next reference message is transmitted at the end of the basic cycle at the Tx_Ref_Trigger, and no gap is inserted into the schedule.

In time-triggered operation mode, the Next_is_Gap bit in the reference message is ignored. The following bits are also ignored in this mode:

- CANFDx_CHy_TTOCN.NIG
- CANFDx_CHy_TTOCN.FGP
- CANFDx_CHy_TTOCN.TMG

23.5.4 Stop watch

The stop watch function allows capturing internal time values from the M_TTCAN, such as local time, cycle time, or global time, based on an external event trigger.

To enable the stop watch function, the application program must first select the desired time source by configuring TTOCN.SWS. This field determines whether local time, cycle time, or global time is used as the stop watch source.

When TTOCN.SWS is not equal to 00 and the interrupt flag TTIR.SWE is cleared (0), the current value of the selected time source is copied into the TTCPT.SWV register. This capture occurs on the next rising or falling edge of the stop watch trigger signal, as defined by the TTOCN.SWP configuration. Once the value is captured, the interrupt flag TTIR.SWE is set.

After the application reads the captured value from TTCPT.SWV, it can re-enable the stop watch function by resetting TTIR.SWE to 0, allowing the next stop watch event to be detected and processed.

23.5.5 Local/cycle/global time and external clock synchronization

There are two possible levels in time-triggered CAN:

1. **Level 1**: Provides only time-triggered operation using cycle time
2. **Level 2**: Additionally offers improved synchronization quality, global time, and external clock synchronization

In both levels, all timing features are based on a local time base, referred to as the local time.

The local time is a 16-bit cyclic counter incremented once per Network Time Unit (NTU). Internally, the NTU is represented by a 3-bit counter, which functions as the fractional part (three binary digits) of the local time. Typically, the 3-bit NTU counter is incremented eight times per NTU. If the NTU length is shorter than eight CAN clock periods either due to Level 1 configuration or clock calibration in Level 2, the NTU fraction length is adjusted, and the NTU counter is incremented only four times per NTU.

Figure 147 illustrates the synchronization of cycle time and global time, which is performed identically by all TTCAN nodes, including the time master.

Any message received or transmitted triggers a capture of the local time at the message's frame synchronization event. This event occurs at the sample point of each Start-of-Frame (SoF) bit and causes the local time to be stored as Sync_Mark. Both Sync_Mark and Ref_Mark are captured with the 3-bit fractional part included.

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:

$$\text{Cycle Sync Mark} = \text{Sync_Mark} - \text{Ref_Mark} \quad (8)$$

This value is stored in the register TTCSM.

The most significant 16 bits of the difference between Ref_Mark and the current local time represent the cycle time:

$$\text{Cycle Time} = \text{Local Time} - \text{Ref_Mark} \quad (9)$$

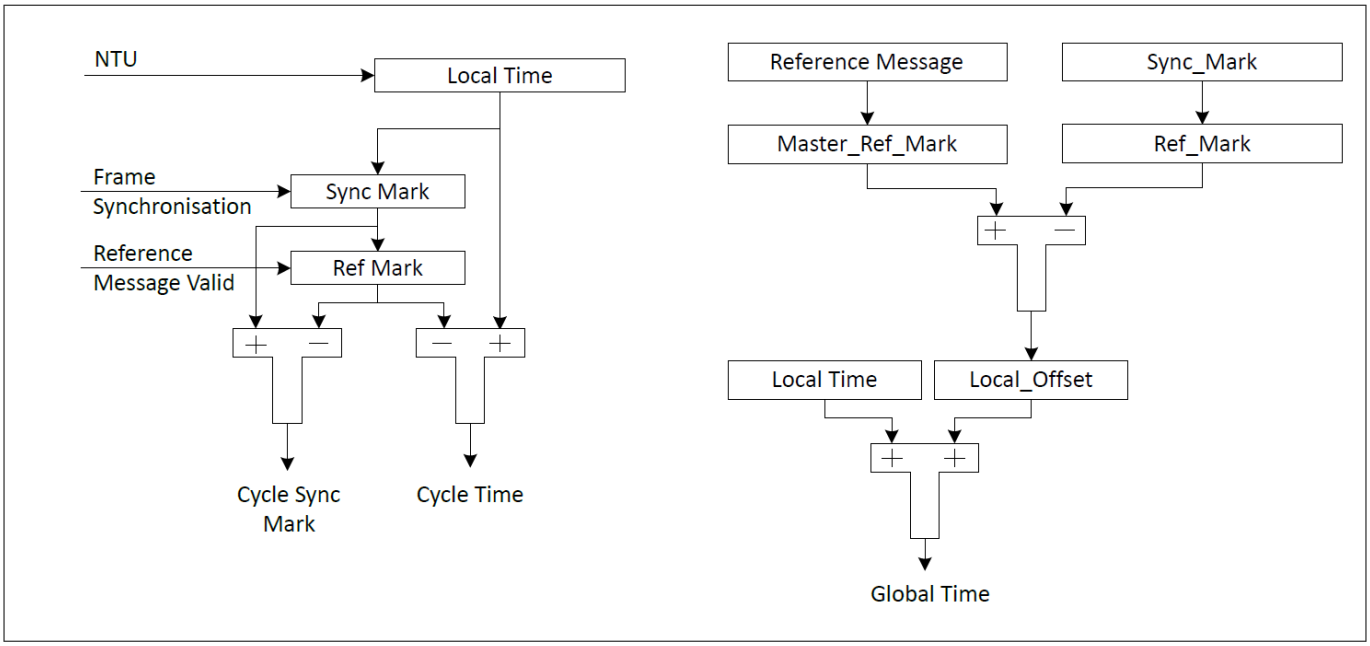


Figure 147 Cycle time and global time synchronization

The cycle time that can be read from `TTCTC.CT` is the difference between the node's local time and `Ref_Mark`, both synchronized into the host clock domain and truncated to 16 bits.

The global time is valid only in TTCAN Level 0 and Level 2. It is invalid in Level 1. A node's view of the global time is its local image of the global time, expressed in local NTUs. After configuration, a potential time master uses its own local time as the global time. This is achieved by transmitting its own `Ref_Marks` as `Master_Ref_Marks` in the reference message (bytes 3 and 4).

The global time that can be read from `TTLGT.GT` is the sum of the node's local time and its local offset. Both values are synchronized into the host clock domain and truncated to 16 bits. The fractional part is used exclusively for clock synchronization.

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 147). The node's view of the global time is:

$$\text{Global Time} = \text{Local Time} + \text{Local Offset} \quad (10)$$

In a potential time master that has not received any reference message from another time master, `Local_Offset` is zero. If a node becomes the current time master after previously receiving reference messages, `Local_Offset` is frozen at its last value.

In time-receiving nodes, `Local_Offset` may undergo small adjustments due to clock drift, a change in the time master, or a global time discontinuity. A global time discontinuity is indicated by the `Disc_Bit` in the reference message.

Except in the case of a global time discontinuity, the global time provided to the application program by the `TTLGT` register is smoothed using low-pass filtering to ensure a continuous and monotonic value.

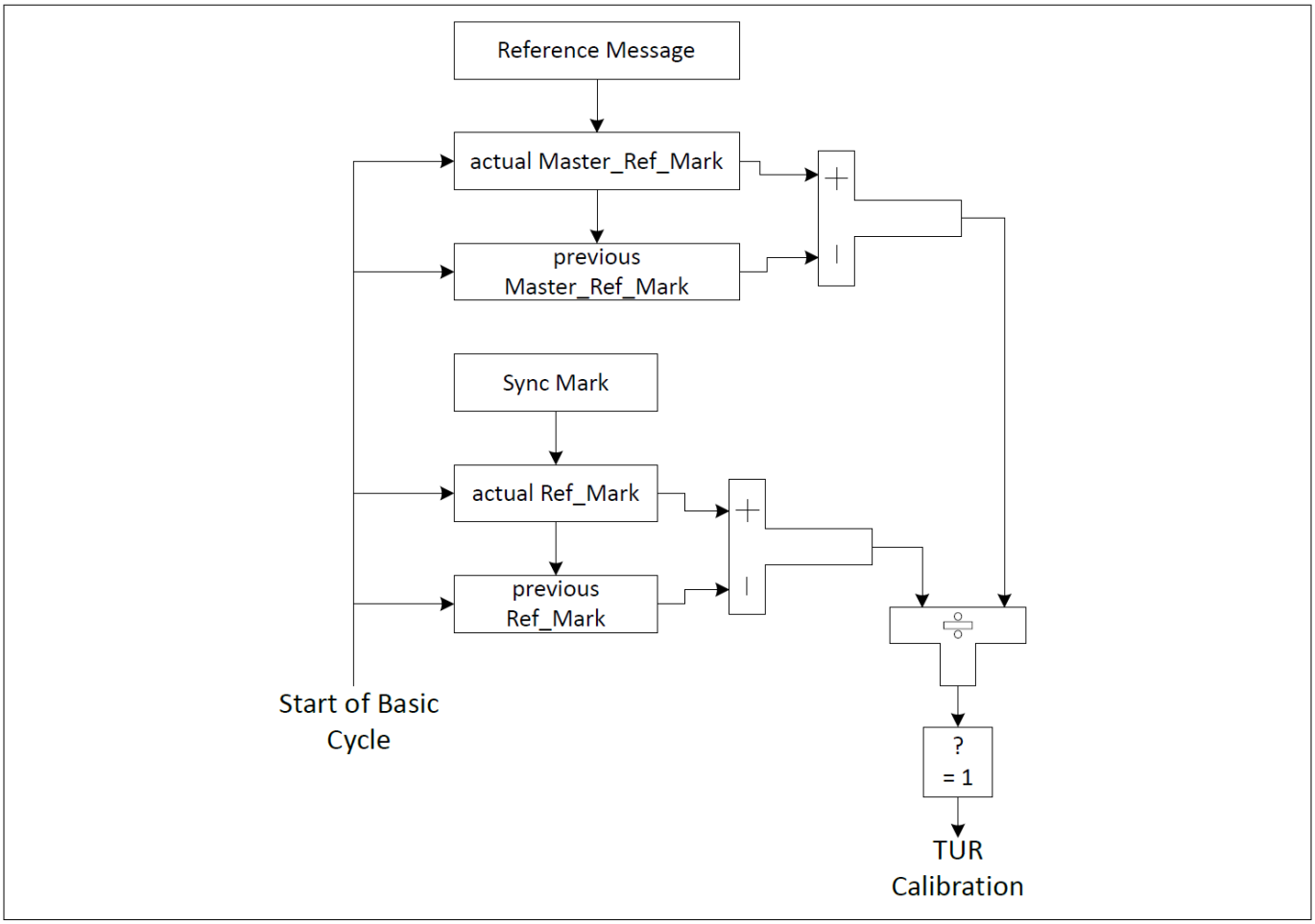


Figure 148 TTCAN Level 0 and Level 2 drift compensation

Figure 148 illustrates how, in TTCAN Level 0 and Level 2, the receiving node compensates for drift between its own local clock and the time master's clock. This is done by comparing the length of a basic cycle measured in local time and in global time. If a difference exists 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), defined as:

$$SD = NC - TURNA.NAV \quad (11)$$

is less than or equal to the synchronization deviation limit (SDL), the new value for `TURNA.NAV` takes effect. Otherwise, automatic drift compensation is suspended.

In TTCAN Level 0 and Level 2, `TTOST.QCS` indicates whether automatic drift compensation is active or suspended. In TTCAN Level 1, `TTOST.QCS` is always set to '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 setting 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.

To adjust the local clock speed:

1. Write the newly calculated Numerator Configuration Low to `TURCF.NCL`
2. Set `TTOCN.ECS` to '1' to activate the new value

***Note**: `TURCF.DC` cannot be updated during operation*

To adjust the global time phase:

1. Write the phase offset into the TT Global Time Preset register (TTGTP)
2. Set TTOCN.SGT to '1' to activate the new value

The first reference message transmitted after a global time phase adjustment will have the Disc_Bit set to '1'.

TTOST.QGTP indicates whether the node's global time is in phase with the time master's global time. In TTCAN Level 1, and when the SDL is exceeded in Level 0 or Level 2 (TTOST.QCS = 0), TTOST.QGTP is permanently set to '0'. It is also temporarily set to '0' while the global time is being low-pass filtered to provide the application with a continuous, monotonic value.

Low-pass filtering is not applied when the last reference message contains Disc_Bit = 1 or when TTOST.QCS = 0.

23.5.6 TTCAN error level

The ISO 11898-4 standard defines four levels of error severity:

Table 162 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 masters 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; that is, 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 multiple errors are detected simultaneously, the error with the highest severity level prevails.

When an error is detected, the application is notified through TTIR.ELC. The current error level is monitored by TTOST.EL.

The M_TTCAN module signals the following error conditions, as specified by ISO 11898-4:

Table 163 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.

(table continues...)

Table 163 (continued) Error condition

Error condition	Description
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').
CAN_Bus_Off (S3)	Entering CAN_Bus_Off state sets 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, 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, 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 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 cycle.

23.5.7 TTCAN message handling

23.5.7.1 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) are provided by the FSE.

If the payload select reference message `TTRMC.RMPS` is set, the remaining payload of the reference message is taken from Tx Buffer 0:

- **For Level 1**: Bytes 2 to 8
- **For Level 0 and Level 2**: Bytes 5 to 6

In this case, the data length code (DLC) from message buffer 0 is used.

Table 164 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 data length code (DLC) greater than 1 must be configured. For Level 0 and Level 2, a DLC greater than 4 is required. Additionally, the transmission request pending bit `TXBRP.TRP0` of message buffer 0 must be set (see Table 164). If `TXBRP.TRP0` is not set when a reference message is initiated, the message is transmitted with only the data bytes supplied by the FSE.

For acceptance filtering of reference messages, the reference identifier `TTRMC.RID` is used.

23.5.7.2 Message reception

Message reception is handled via the two Rx FIFOs, following the same method used 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 `M_TTCAN` is in synchronization states `In_Gap` or `In_Schedule`. The update occurs at the message's `Rx_Trigger`. At this point, the system checks which acceptance filter element matches the latest message received in the current basic cycle. The matching filter number is stored as the acceptance filter result.

If this result matches the filter number defined in the trigger memory element, the MSC is decremented by one. If the result does not match the defined filter number, or if the acceptance filter result is cleared, the MSC is incremented by one. At each `Rx_Trigger` and at each start of cycle, the last acceptance filter result is cleared.

The time mark of an `Rx_Trigger` should be set to ensure that reception and acceptance filtering of the targeted message are completed. This timing must account for RAM access time and the order of the filter list. It is recommended to place filters used for `Rx_Triggers` at the beginning of the filter list.

Using an `Rx_Trigger` for the reference message is not recommended.

23.5.7.3 Message transmission

For time-triggered message transmission, M_TTCAN provides 32 dedicated Tx buffers (see TTCAN configuration). A Tx FIFO or Tx queue is not available when M_TTCAN is configured for time-triggered operation (TTOCF.OM = 01 or 10).

Each Tx_Trigger in the trigger memory points to a specific Tx buffer containing a particular message. Multiple Tx_Triggers may reference the same Tx buffer if the message is to be transmitted more than once within a basic cycle or matrix cycle.

The application must update the data regularly and in sync with the cycle time. The host CPU must ensure that no partially updated messages are transmitted. To achieve this, the host should follow these procedures:

Tx_Trigger_Single/Tx_Trigger_Merged/Tx_Trigger_Arbitration:

- Check whether the previous transmission has completed by reading TXBTO
- Update the Tx buffer's configuration/payload
- Issue an Add Request to set the Tx Buffer Request Pending bit

Tx_Trigger_Continuous:

- Issue a cancellation request to reset the Tx Buffer Request Pending bit
- Check whether the cancellation has completed by reading TXBCF
- Update the Tx buffer's configuration/payload
- Issue an Add Request to set the Tx Buffer Request Pending bit

The message MSC stored with the corresponding Tx_Trigger provides feedback on transmission success. The MSC is incremented by one if transmission cannot start because the CAN bus was not idle within the transmit enable window, or if the message was started but not completed successfully. It is decremented by one if the message is transmitted successfully or could have been started but was not due to transmission being disabled (M_TTCAN in Error Level S2 or host-disabled message).

Tx buffers may be managed dynamically. Multiple messages with different identifiers may share the same Tx buffer element. In such cases, the host must ensure that no transmission request is pending before reconfiguring the buffer by checking TXBRP.

If a Tx buffer with a pending transmission request needs to be updated, the host must first issue a cancellation request and confirm completion by reading TXBCF before proceeding with the update.

The Tx handler transfers a message from message RAM to its intermediate output buffer at the trigger element, which activates immediately before the Tx_Trigger that defines the start of the transmit window. During and after this transfer, the message must not be updated, and its TXBRP bit must not be changed.

To manage this transfer time, an additional trigger element may be placed before the Tx_Trigger. A common example is a Time_Base_Trigger, which performs no other action. The time mark difference between the Tx_Trigger and the preceding trigger should be large enough to ensure the Tx handler can read four words from message RAM, even under high RAM access load from other modules.

23.5.7.3.1 Transmission in exclusive time windows

A transmission is initiated in a time-triggered manner when the cycle time reaches the time mark of a Tx_Trigger_Single or Tx_Trigger_Continuous. There is no bus arbitration with messages from other nodes during this process.

The MSC is updated based on the result of the transmission attempt. After a successful transmission triggered by Tx_Trigger_Single, the corresponding Tx Buffer Request Pending bit is reset. In contrast, after a successful transmission triggered by Tx_Trigger_Continuous, the Tx Buffer Request Pending bit remains set.

If the transmission fails due to disturbances, it will be retried the next time one of its Tx_Triggers becomes active.

23.5.7.3.2 Transmission in arbitrating time windows

A transmission is initiated in a time-triggered manner when the cycle time reaches the time mark of a `Tx_Trigger_Arbitration`. In this mode, multiple nodes may attempt to transmit simultaneously. The message must undergo arbitration with other messages on the CAN bus.

The MSC is not updated during arbitration-based transmission attempts. If the transmission fails due to lost arbitration or disturbances, it will be retried the next time one of its `Tx_Triggers` becomes active.

23.5.7.3.3 Transmission in merged arbitrating time windows

The purpose of a merged arbitrating time window is to allow multiple nodes to transmit a limited number of frames in immediate sequence. The transmission order is determined by CAN arbitration. This mode is not intended for burst transmission by a single node. Since the node does not have exclusive access during this time window, not all requested transmissions are guaranteed to succeed.

Messages that lose arbitration or are affected by transmission errors may be retransmitted within the same merged arbitrating time window. Retransmission will not occur if the corresponding Transmission Request Pending flag was reset due to 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, the Tx handler can process up to three message numbers from the trigger list. These messages are transmitted in the sequence defined by the trigger list. If the time mark of a fourth message is read before the first message is transmitted or canceled by the host, the fourth request is ignored.

Transmission within a merged arbitrating time window is not strictly time-triggered. A message may be transmitted before its time mark or after it, depending on bus availability.

Messages transmitted by a specific node within a merged arbitrating time window are started in the order of their `Tx_Triggers`. A message with lower CAN priority may block the transmission of a subsequent message with higher priority if there is competing bus traffic. This should be considered when configuring the trigger list.

`Time_Base_Triggers` may be placed between consecutive `Tx_Triggers` to define the time by which the corresponding Tx buffer data must be updated.

23.5.8 TTCAN interrupt and error handling

The TT Interrupt Register `TTIR` consists of four segments. Each interrupt can be enabled individually using the corresponding bit in the TT Interrupt Enable register `TTIE`. Interrupt flags remain set until cleared by the host. A flag is cleared by writing a '1' to its corresponding bit position.

Segment 1: Fatal error flags: This segment includes the flags `CER`, `AW`, `WT`, and `IWT`. Each flag indicates a fatal error condition that causes CAN communication to stop. Except for `IWT`, these conditions require reconfiguration of the `M_TTCAN` module before communication can resume.

Segment 2: Disturbance error flags: This segment includes the flags `ELC`, `SE1`, `SE2`, `TXO`, `TXU`, and `GTE`. These flags indicate error conditions that disturb CAN communication. If caused by transient failures, such as bus disturbances, they are handled by the TTCAN protocol's failure management and do not require application-level intervention.

Segment 3: Time-related event flags: This segment includes the flags `GTD`, `GTW`, `SWE`, `TTMI`, and `RTMI`.

- `GTD` and `GTW` are triggered by global time events (Level 0 and Level 2 only) and require application-level response
- `SWE` captures internal time values during a Stop Watch Event
- `TTMI` signals that a specific `Time_Base_Trigger` has been reached
- `RTMI` indicates that the time referenced by `TTOCN.TMC` (cycle, local, or global) matches the time mark `TTTMK.TM`. It can also be used to end a gap

Segment 4: Synchronization flags: This segment includes the flags SOG, CSM, SMC, and SBC. These flags help synchronize the application program with the communication schedule.

23.5.9 Level 0

TTCAN Level 0 is not part of the ISO 11898-4 standard. This mode allows the hardware, which in TTCAN Level 2 maintains a calibrated global time base, to be used for event-driven CAN communication as defined in 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, and the configuration of TTOCF.GEN is ignored. External event-synchronized operation is not supported. A synchronized time base is maintained through transmission of reference messages.

The trigger memory is inactive in Level 0 and does not require configuration. The Time Mark Interrupt flag (TTIR.TTMI) is set when the cycle time reaches TTOCF.IRTO × 0x200, prompting the host to set a transmission request for message buffer 0. The Watch Trigger Interrupt flag (TTIR.WT) is set when the cycle time reaches 0xFF00. These values are selected to provide sufficient margin for stable clock calibration. No additional TT error checks are performed.

Register Time Mark Interrupts (TTIR.RTMI) are also supported. Reference messages are configured as in Level 2. Received reference messages are identified using the identifier configured in register TTRMC. For transmission, only message buffer 0 may be used. The node transmits reference messages whenever the host sets a transmission request for message buffer 0. There is no reference trigger offset.

Configuration registers for Level 0 operation:

- TTRMC
- TTOCF (excluding EVTP, AWL, GEN)
- TTMLM (excluding ENTT, TXEW)
- TURCF

Control registers for Level 0 operation:

- TTOCN (excluding NIG, TMG, FGP, GCS, TTMIE)
- TTGTP
- TTTMK
- TTIR (excluding bits CER, AW, IWT, SE2, SE1, TXO, TXU, SOG, which have no function)

Modified functionality in TTIR for Level 0:

- TTMI: Not defined by trigger memory; activated at cycle time TTOCF.IRTO × 0x200
- WT: Not defined by trigger memory; activated at cycle time 0xFF00

Status signaling for Level 0 operation: TTOST (excluding bits AWE, WFE, GSI, GFI, RTO, which have no function)

23.5.9.1 Synchronizing

Figure 149 illustrates the states and state transitions in TTCAN Level 0 operation. Unlike other levels, Level 0 does not include the In_Gap state.

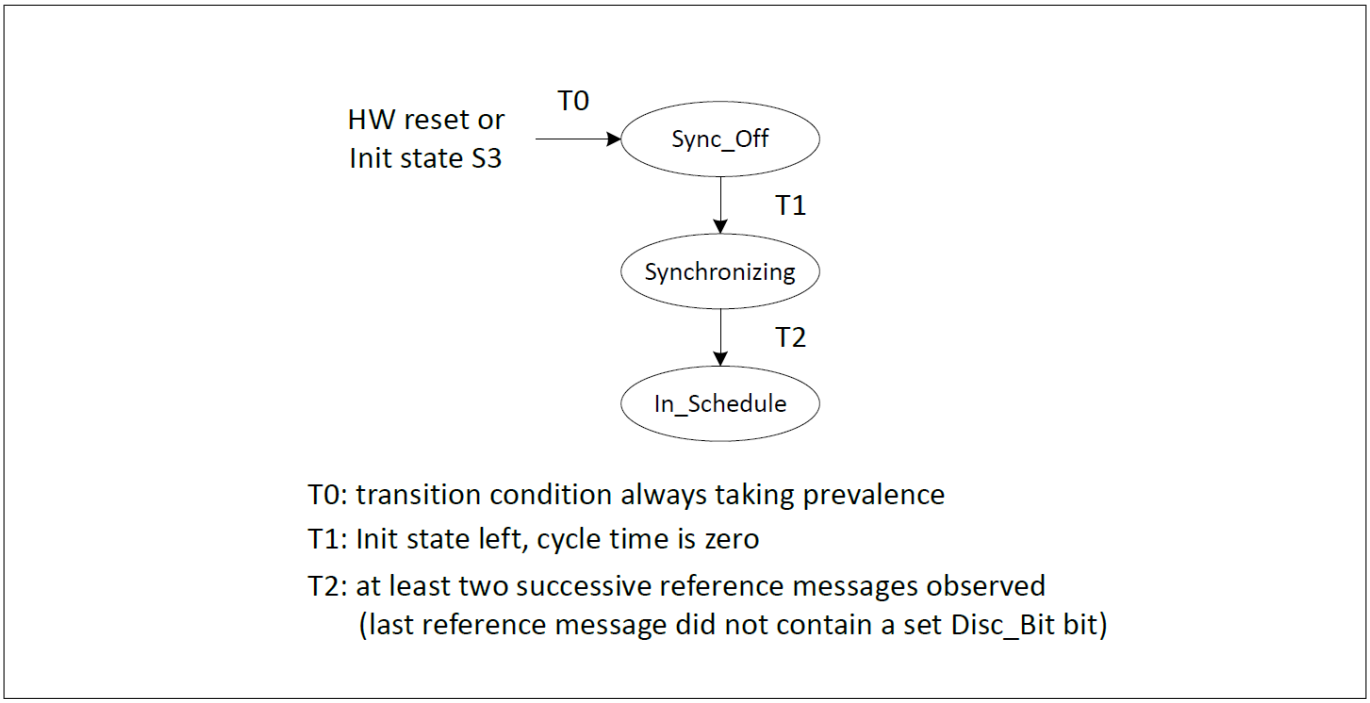


Figure 149 Level 0 schedule synchronization state machine

23.5.9.2 Handling error levels

During Level 0 operation, only the following error conditions may occur:

- `Watch_Trigger_Reached (S3)`: triggered when the cycle time reaches `0xFF00`
- `CAN_Bus_Off (S3)`

Since S1 and S2 error levels are not applicable in Level 0, the error level can only switch between `S0` (No Error) and `S3` (Severe Error).

In TTCAN Level 0, the handling of S3 errors differs from Level 1 and Level 2. When an S3 error is reached:

- `TTOST.SYS` and `TTOST.MS` are reset
- Interrupt flags `TTIR.GTE` and `TTIR.GTD` are set

When the error level `S3` (`TTOST.EL = 11`) is entered, bus monitoring mode is not activated, unlike in Level 1 and Level 2. The S3 error level is exited automatically after transmission (for a time master) or reception (for a time slave) of the next reference message.

23.5.9.3 Master-slave relation

Figure 150 illustrates the master-slave relationship in TTCAN Level 0. In the event of an S3 error, the `M_TTCAN` transitions to the `Master_Off` state.

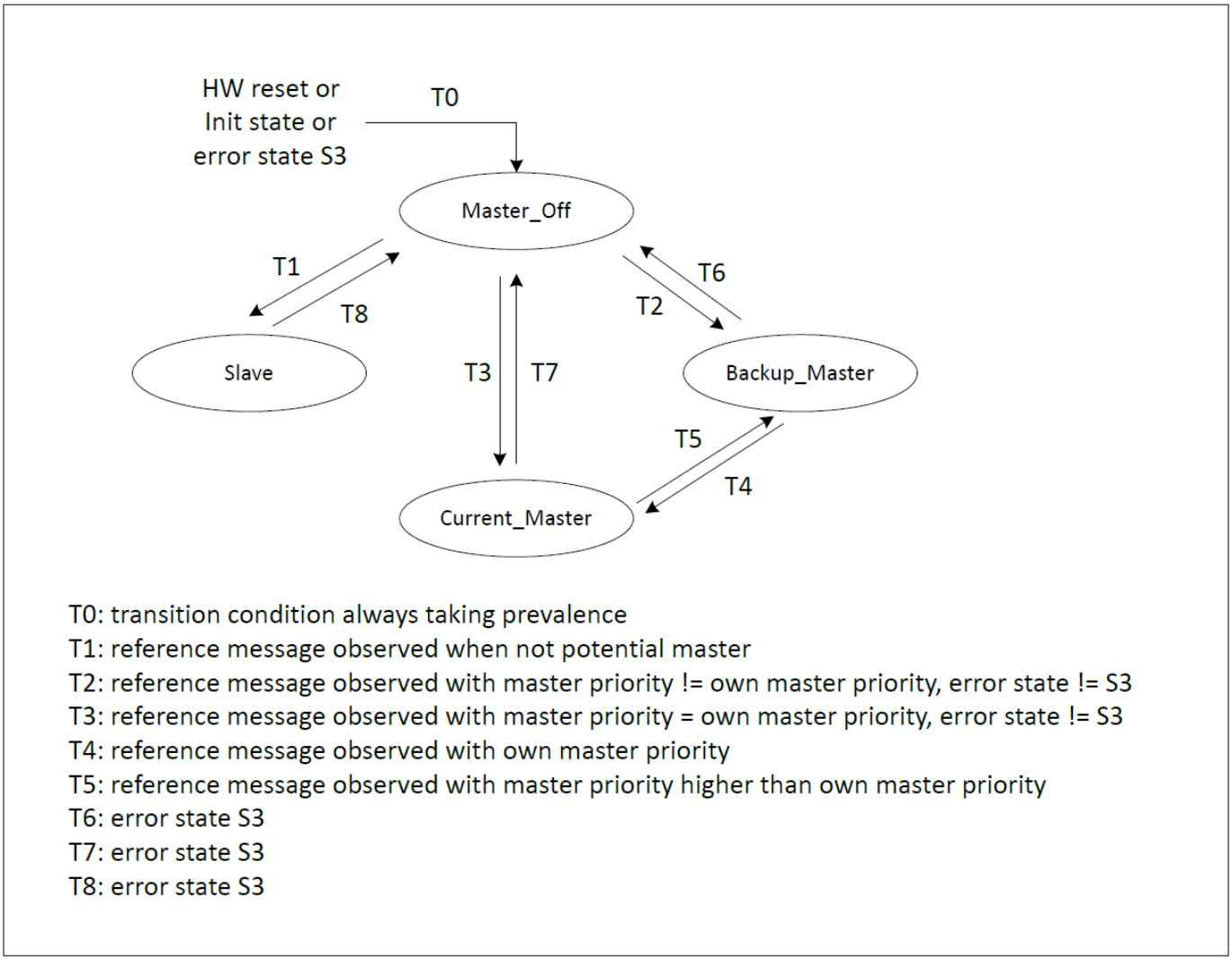


Figure 150 Level 0 master to slave relation

23.5.10 Synchronization to external time schedule

This feature allows synchronization of the M_TTCAN schedule phase with an external schedule, such as that of a second TTCAN network. It is applicable only when M_TTCAN is the current time master (TTOST.MS = 11).

External synchronization is controlled by the TTOCN.ESCN bit. When TTOCN.ESCN is set, the rising edge of the internal event trigger pin causes M_TTCAN to compare its current cycle time with the target phase value configured in TTGTP.CTP.

Before setting TTOCN.ESCN, the host should align the phases of the two time schedules. This can be done using TTCAN gap control (see TTCAN Gap Control section). Once TTOCN.ESCN is set, the phase lock bit TTOST.SPL is also set.

If the difference between the actual cycle time and the target phase value TTGTP.CTP exceeds 9 NTU, the phase lock bit TTOST.SPL is reset, and the interrupt flag TTIR.CSM is set. The phase lock bit is also reset, and TTIR.CSM is set, when another node becomes the time master.

If both TTOST.SPL and TTOCN.ESCN are set, and the difference between the cycle time and TTGTP.CTP is less than or equal to 9 NTU, the phase lock bit remains set. The measured difference is then used as the reference trigger offset value to adjust the phase during the next reference message transmission.

***Note**: 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. Any subsequent edges within the same basic cycle are ignored.*

23.6 Setup procedures

This section provides example procedures for configurations of the M_TTCAN channel.

23.6.1 General program flow

Figure 151 shows a general flow for configuring the M_TTCAN module.

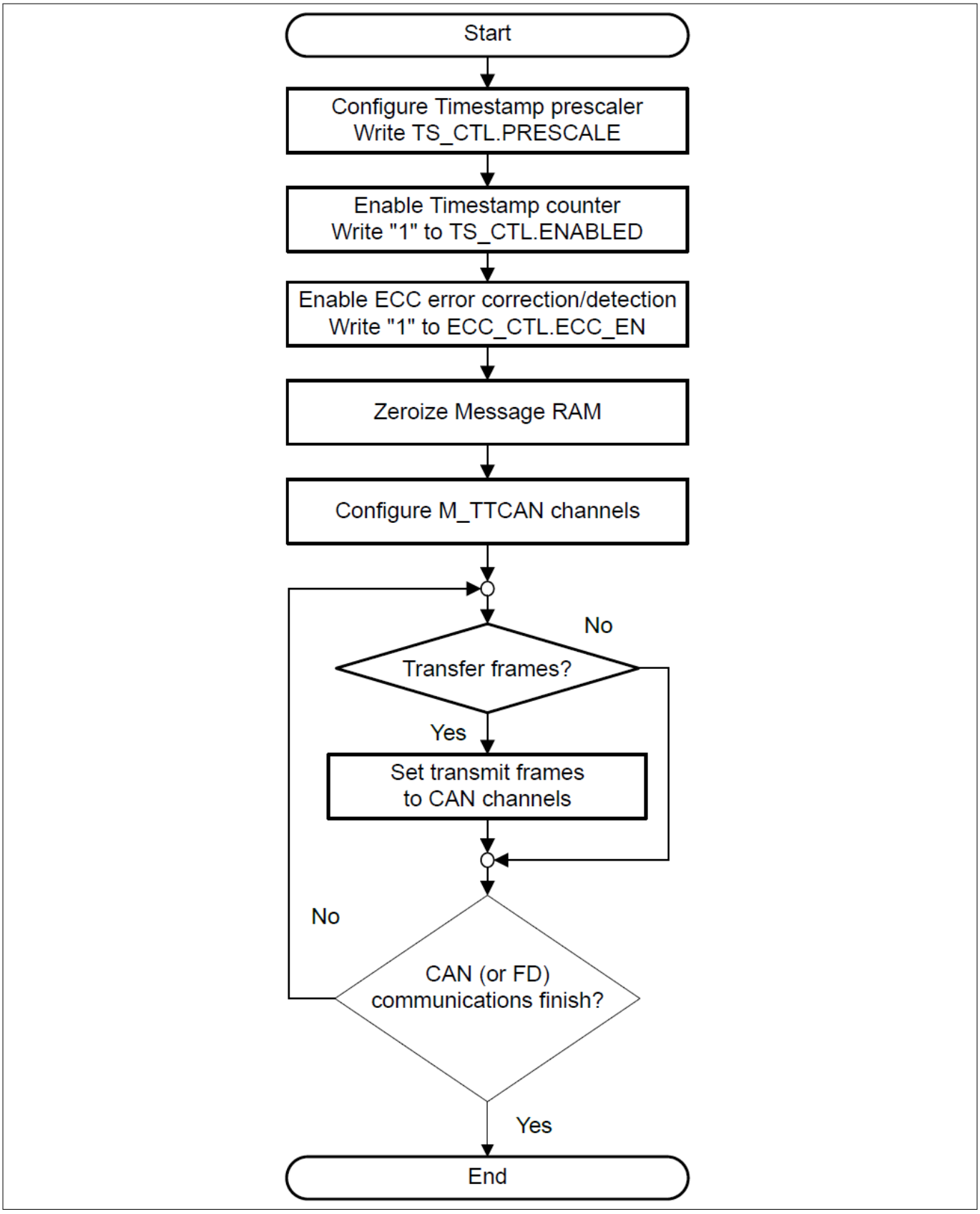


Figure 151 General program flow

23.6.2 Clock stop request

To save power, the application can stop providing clock to an unused M_TTCAN channel by following the steps shown in Figure 152.

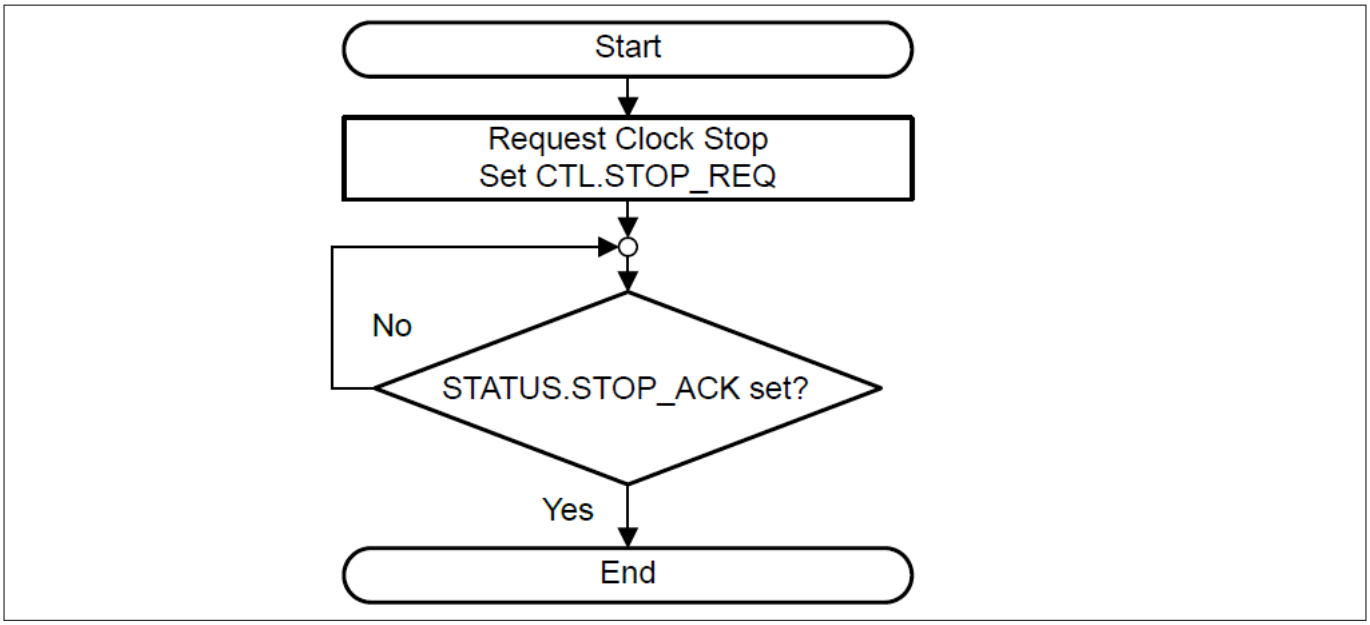


Figure 152 Clock stop request procedure

To resume clock supply, the CTL.STOP_REQ bit must be reset.

23.6.3 MRAM OFF operation

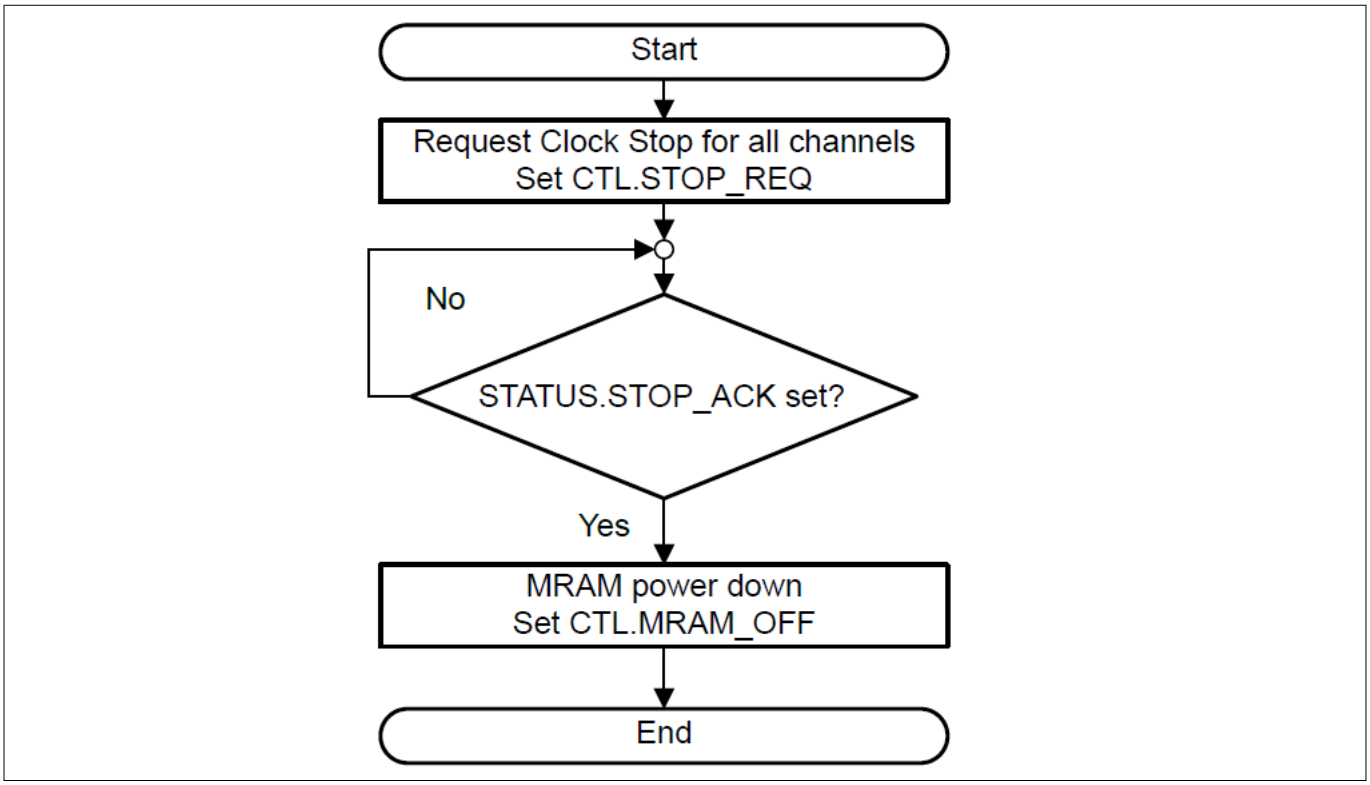


Figure 153 Message RAM OFF operation

23.6.4 MRAM ON operation

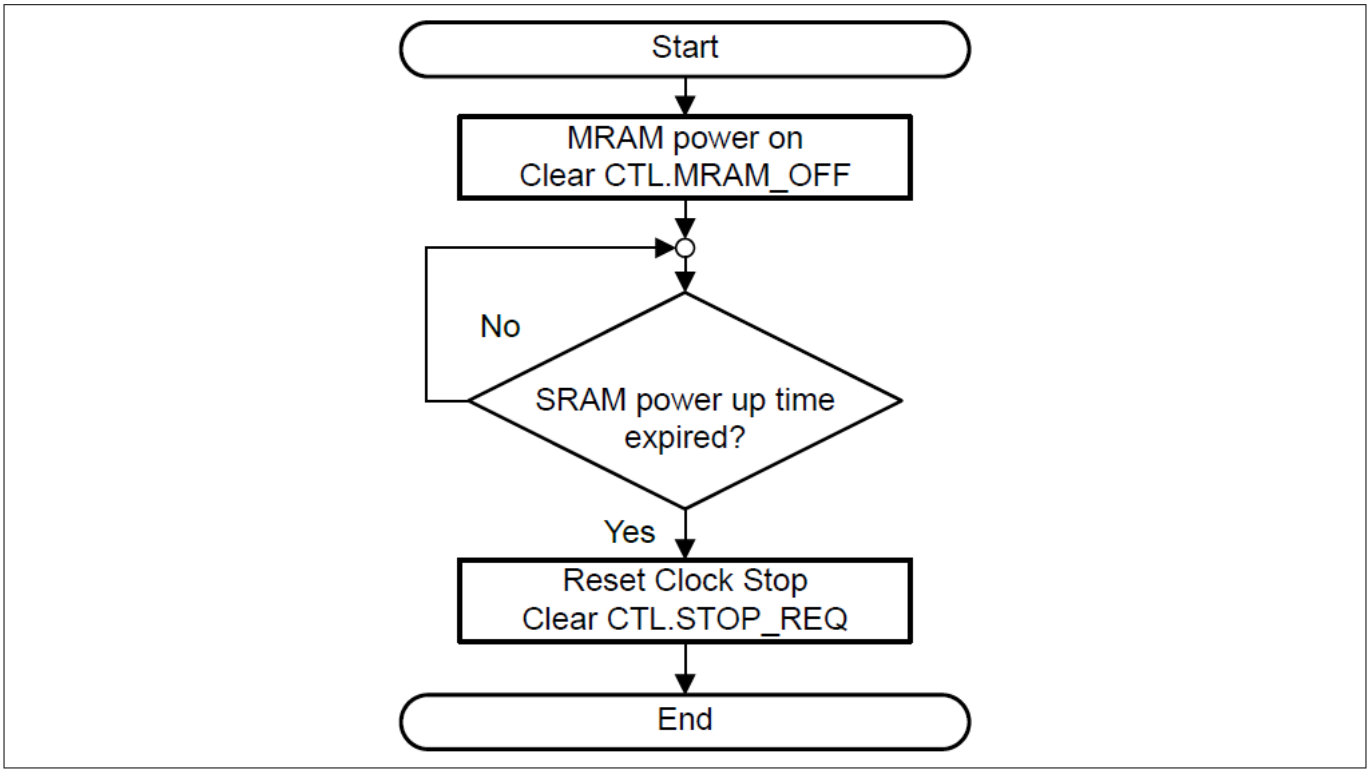


Figure 154 Message RAM ON procedure

After switching MRAM on again, the software must allow a defined power-up time before MRAM can be accessed. This delay must be observed before de-asserting the `STOP_REQ` bit. The required power-up time is equivalent to the system SRAM power-up time specified in the `CPUSS.RAM_PWR_DELAY_CTL` register.

23.6.5 Procedures specific to `M_TTCAN` channel

This section outlines a sample procedure for configuring the `M_TTCAN` channel. Figure 155 illustrates the general program flow used during initialization and setup.

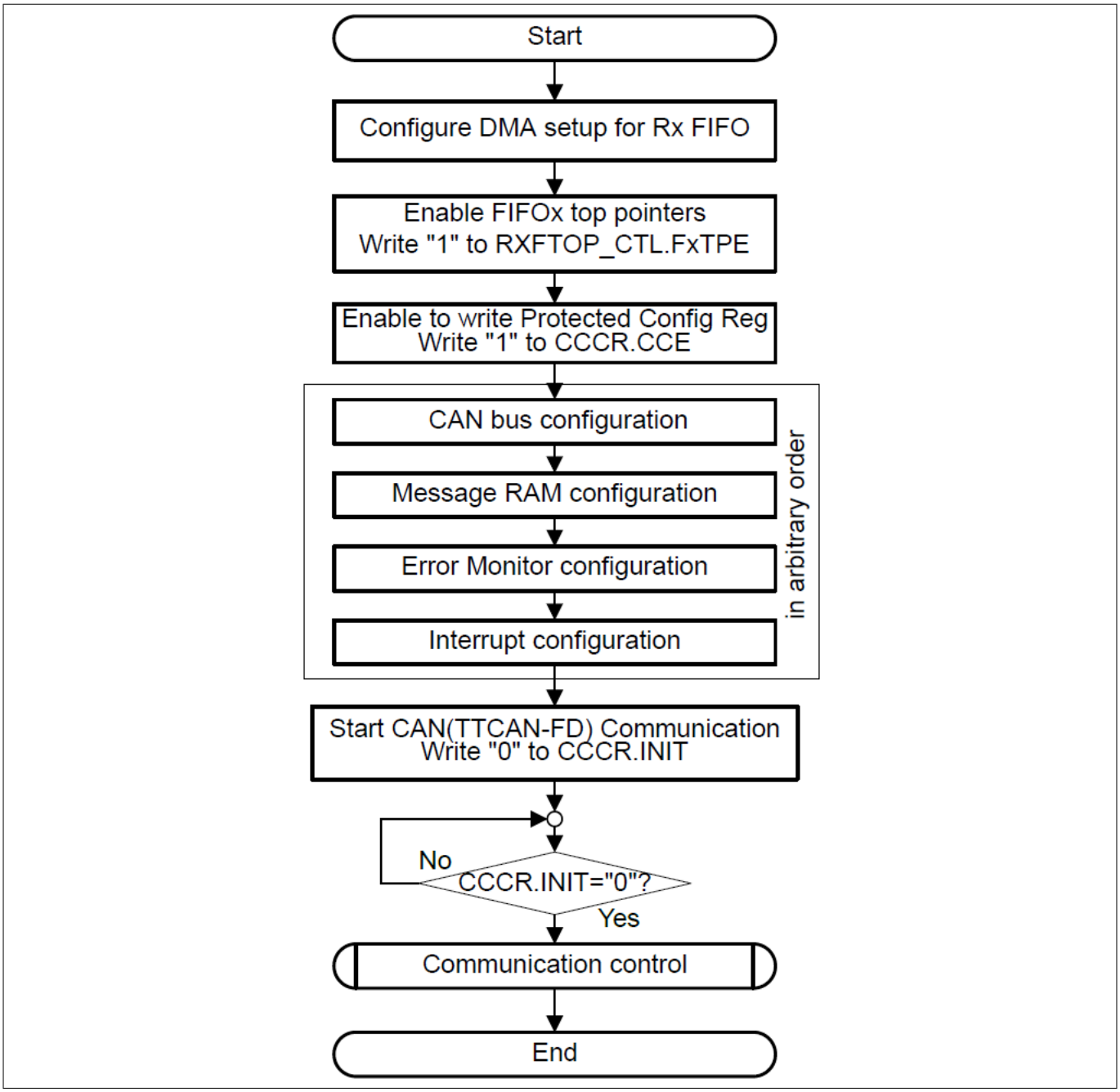


Figure 155 Configuration sequence specific to channel

23.6.6 CAN bus configuration

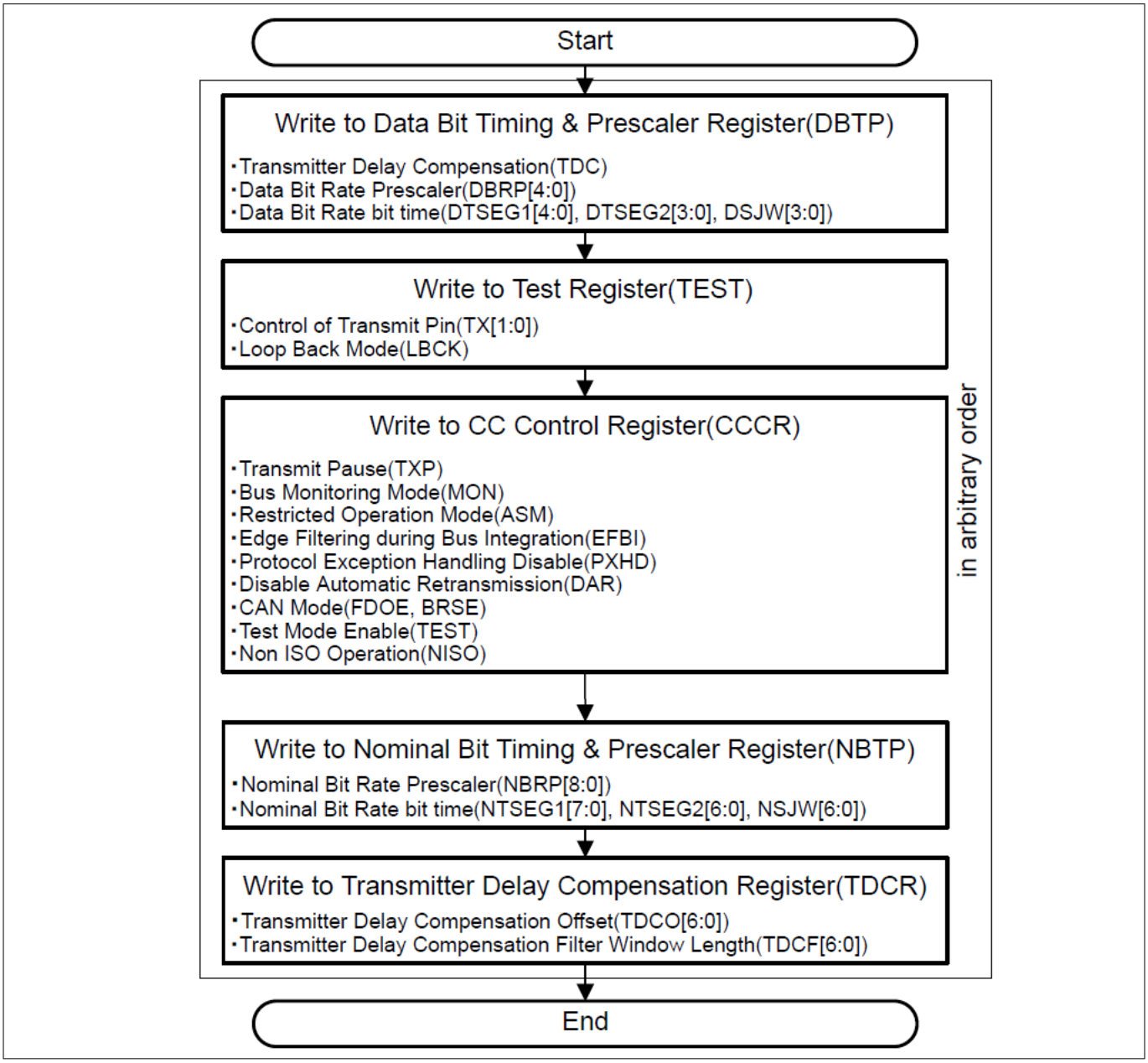


Figure 156 Configuration required for the CAN bus

23.6.7 Message RAM configuration

Figure 157 provides an overview of the message RAM configuration.

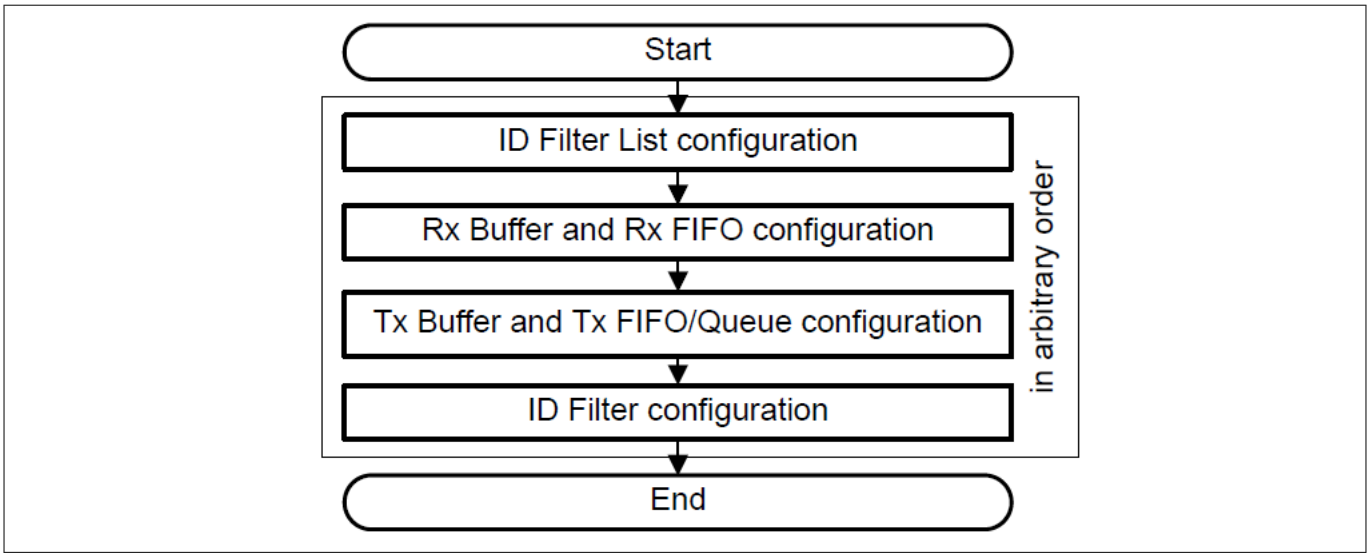


Figure 157 Message RAM configuration overview

Each configuration step shown in the overview is explained in detail in Figure 158, Figure 159, Figure 160, and Figure 161.

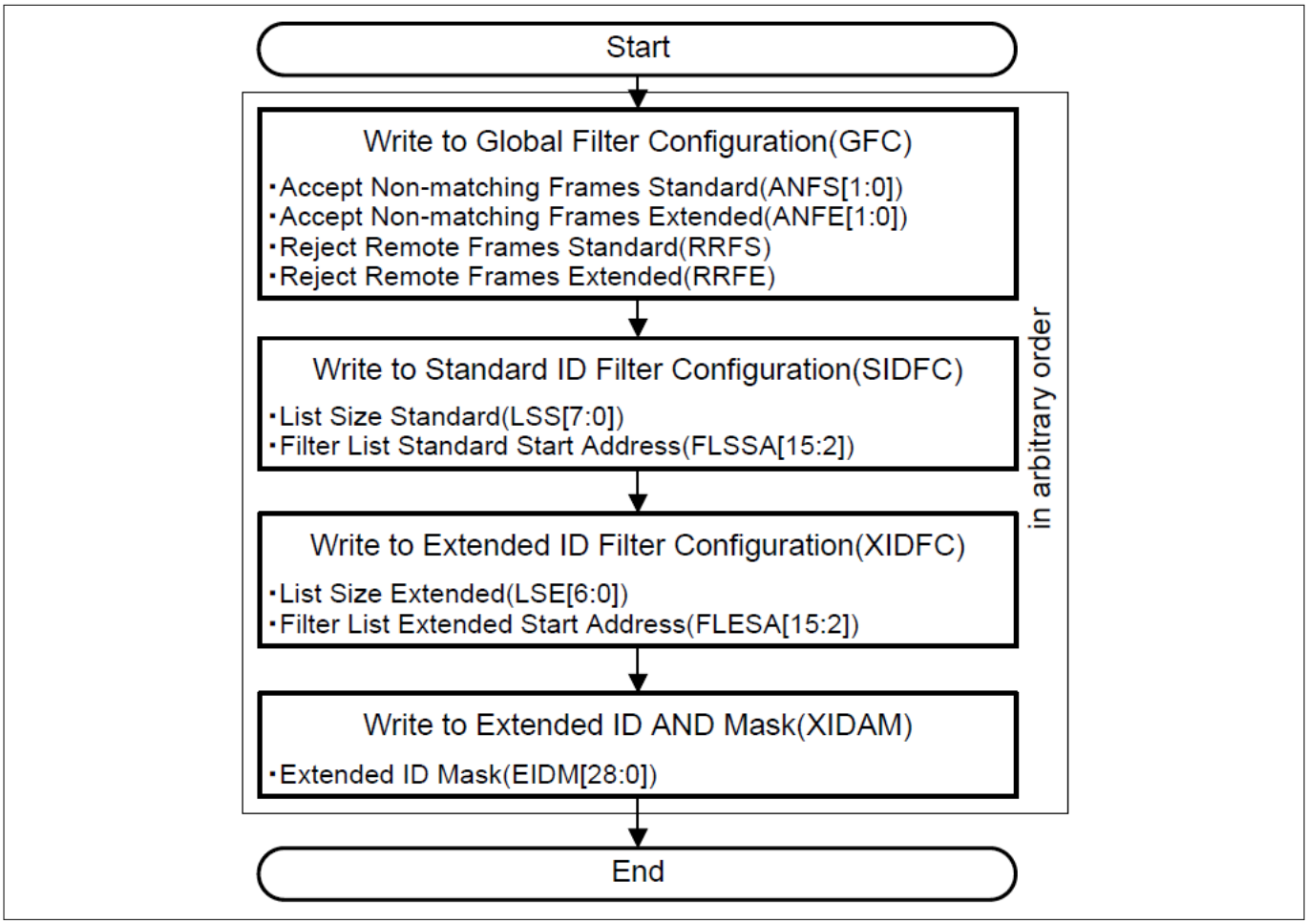


Figure 158 ID filter list configuration

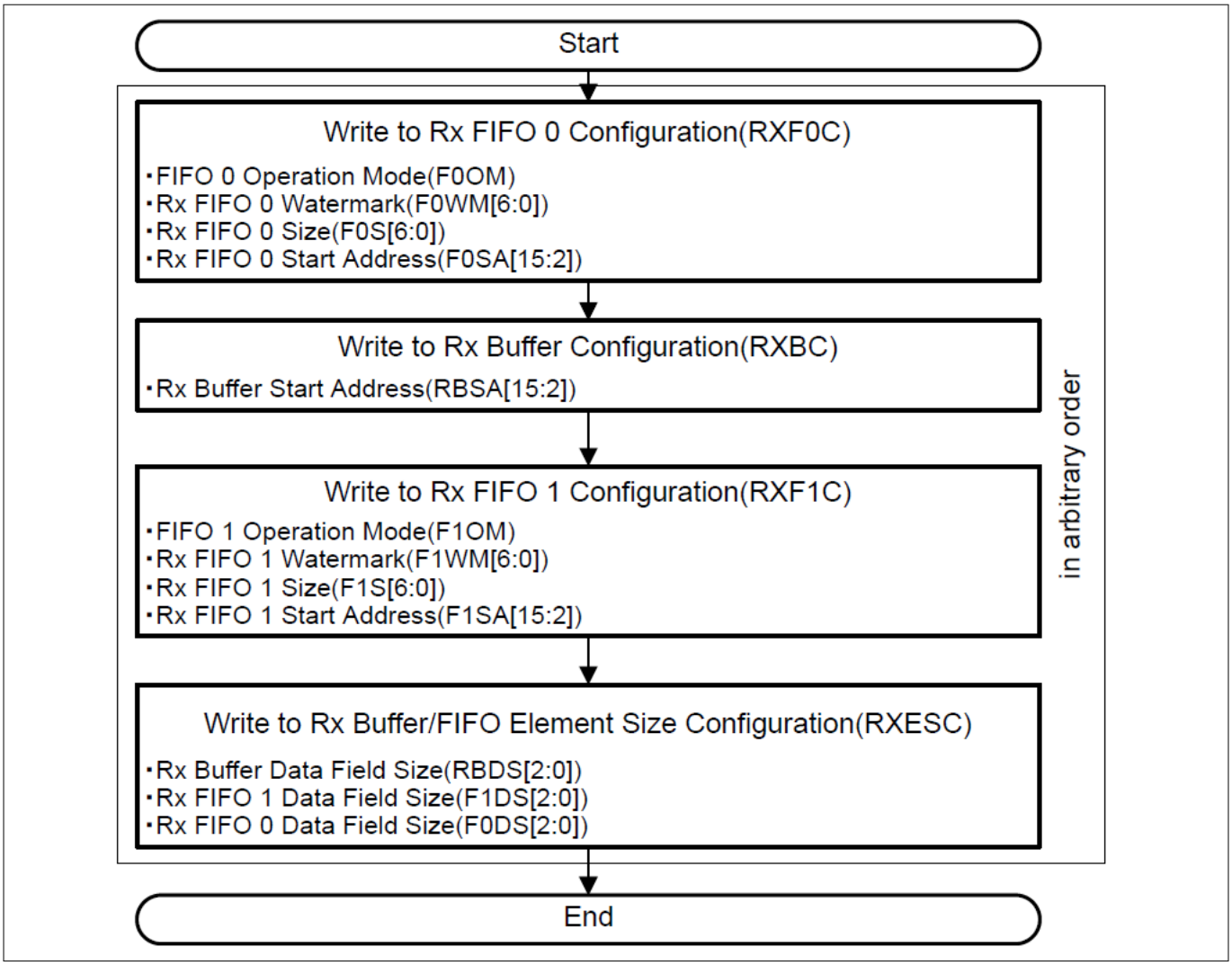


Figure 159 Rx FIFO and Rx buffer configuration

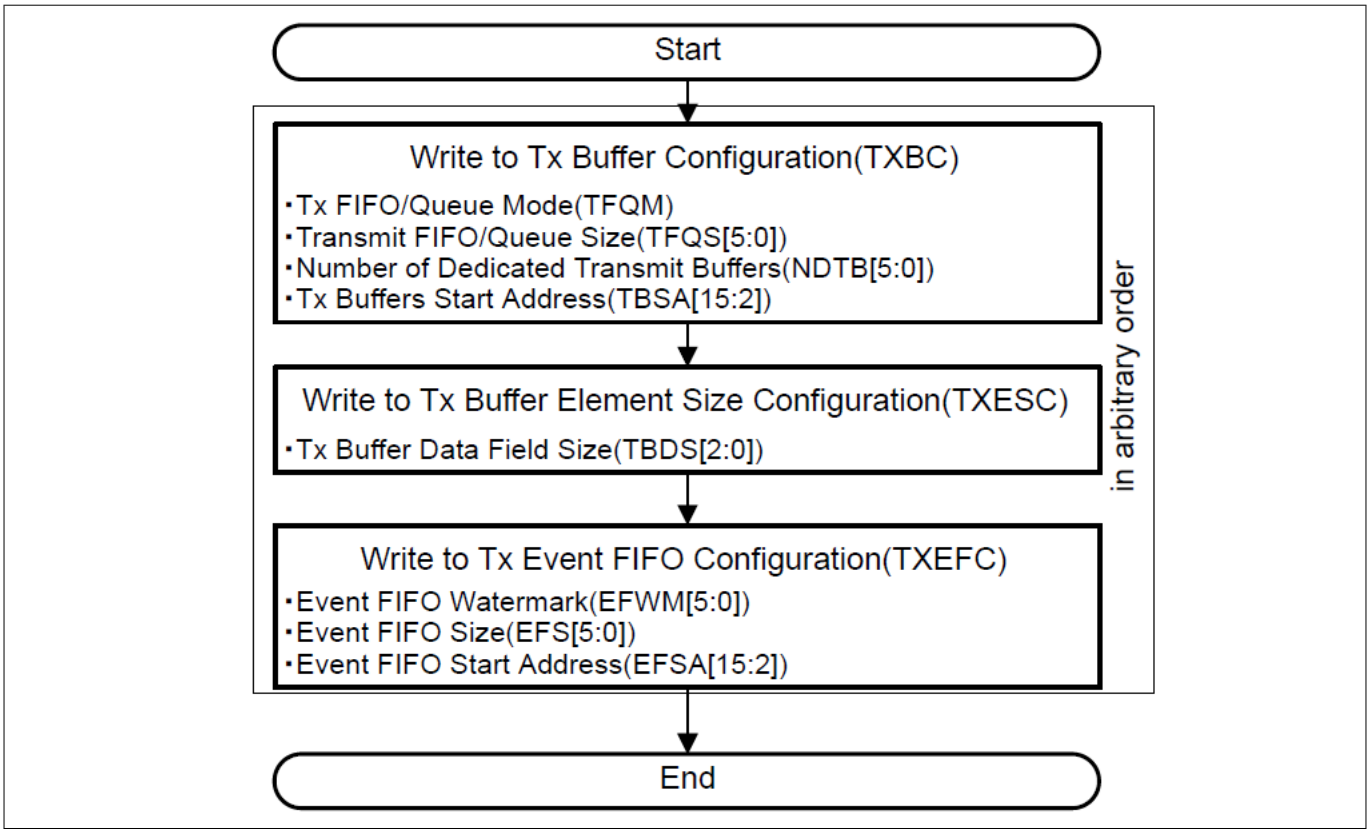


Figure 160 Tx buffer and Tx FIFO/Queue configuration

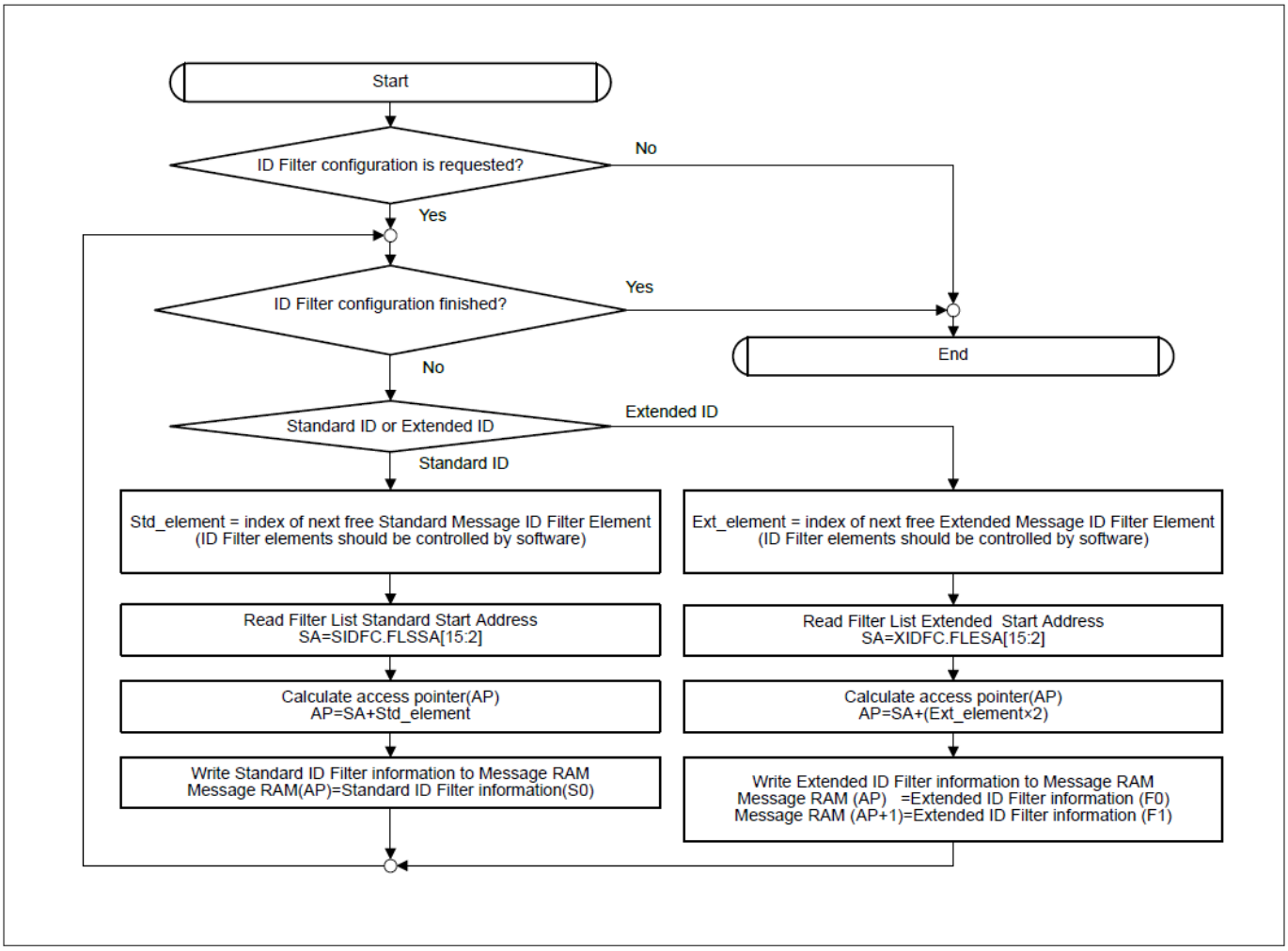


Figure 161 ID filter configuration

23.6.8 Interrupt configuration

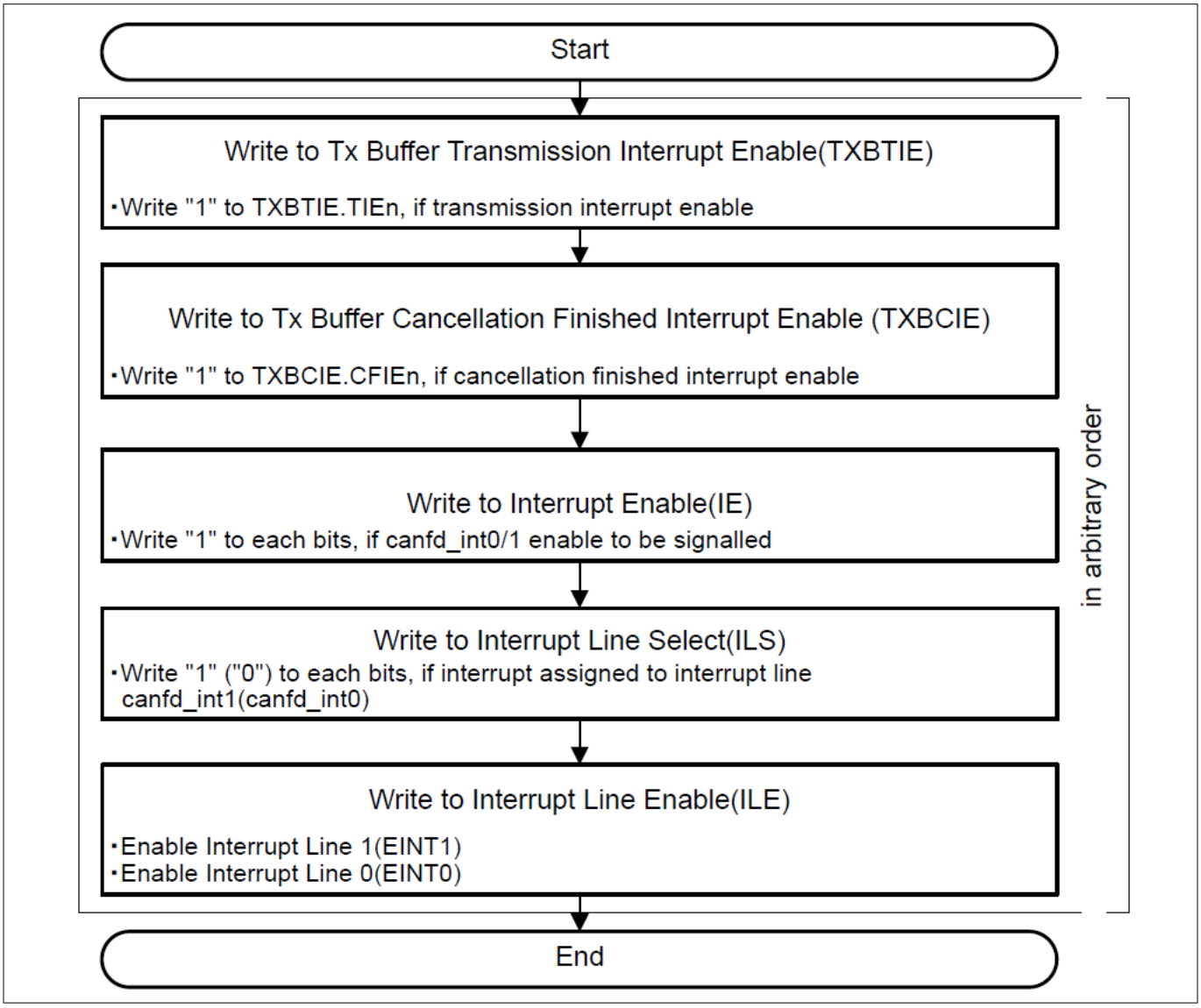


Figure 162 Interrupt configuration

23.6.9 Transmit frame configuration

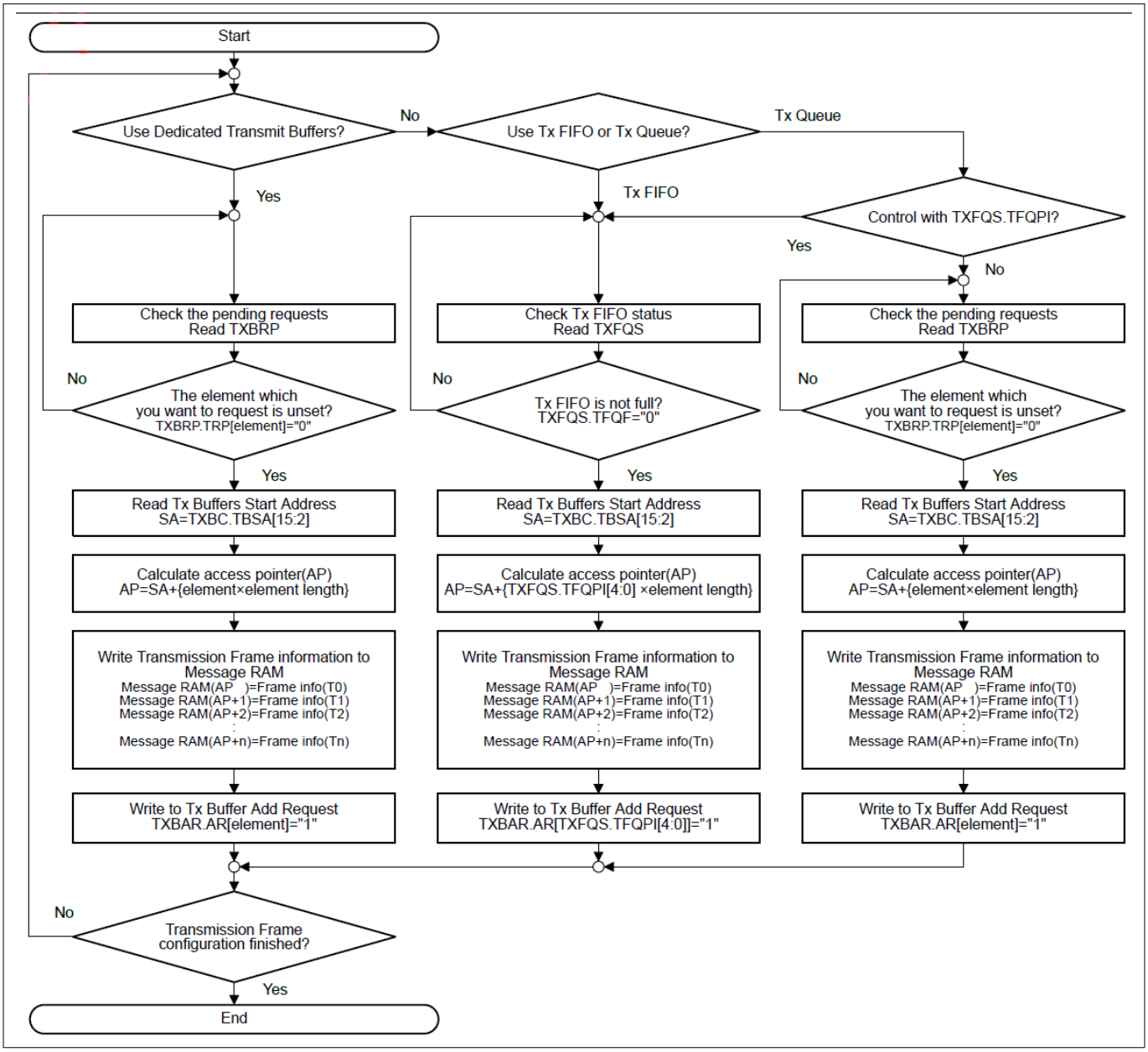


Figure 163 Transmit frame configuration

23.6.10 Interrupt handling

Figure 164 shows a general interrupt handling flow chart.

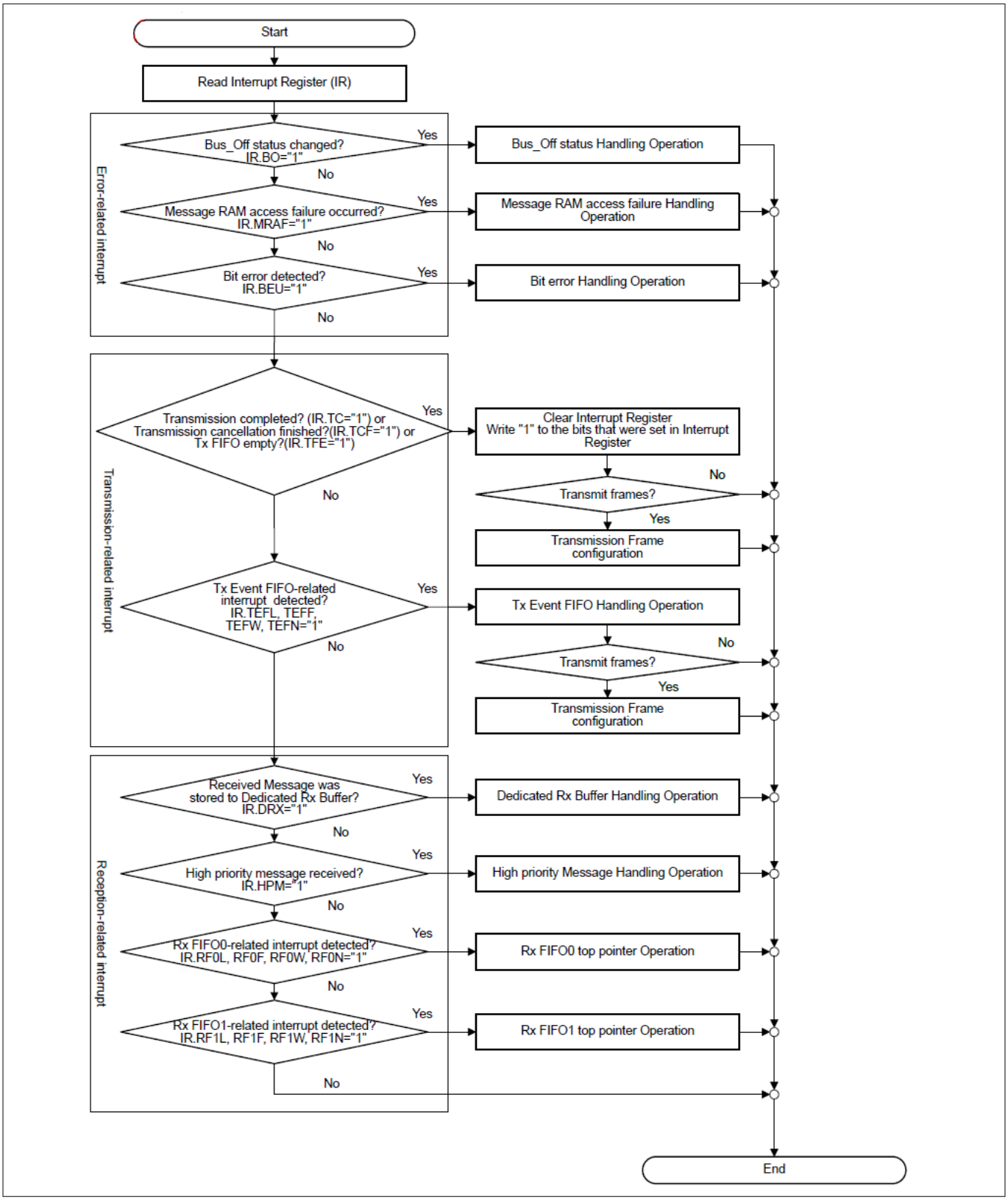


Figure 164 Interrupt handling

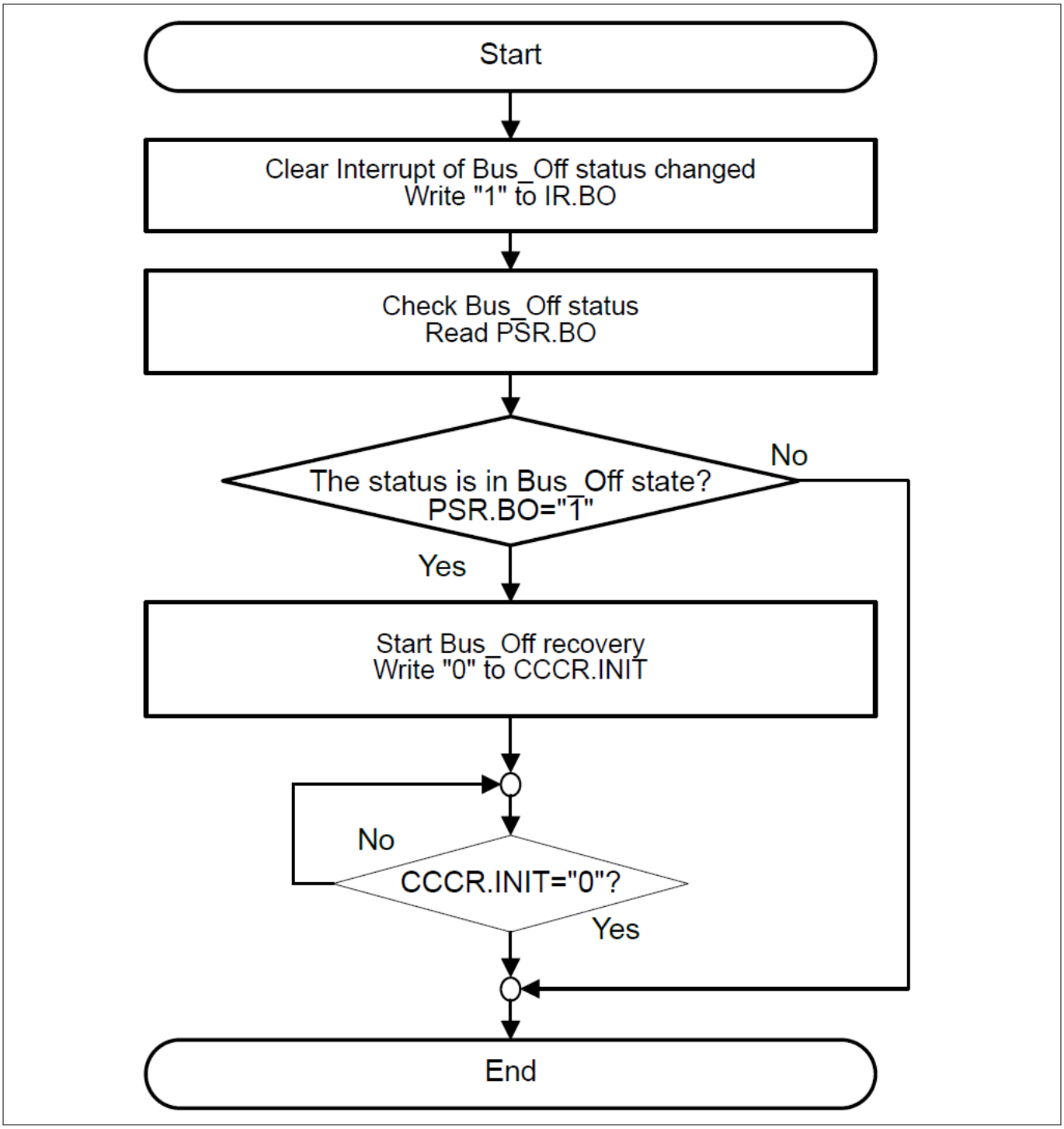


Figure 165 Bus OFF error handling

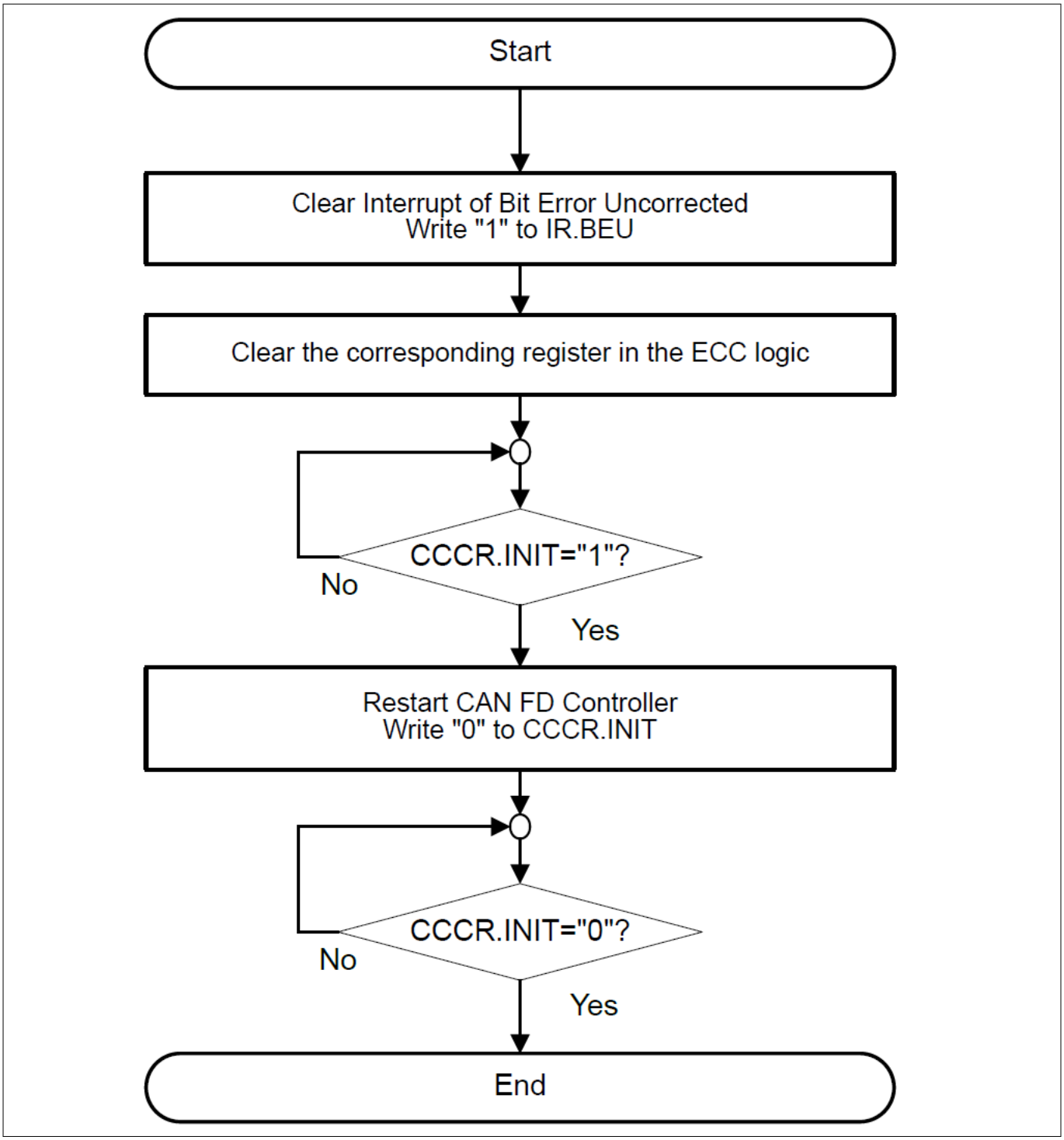


Figure 166 **Figure 16. Bit error handling**

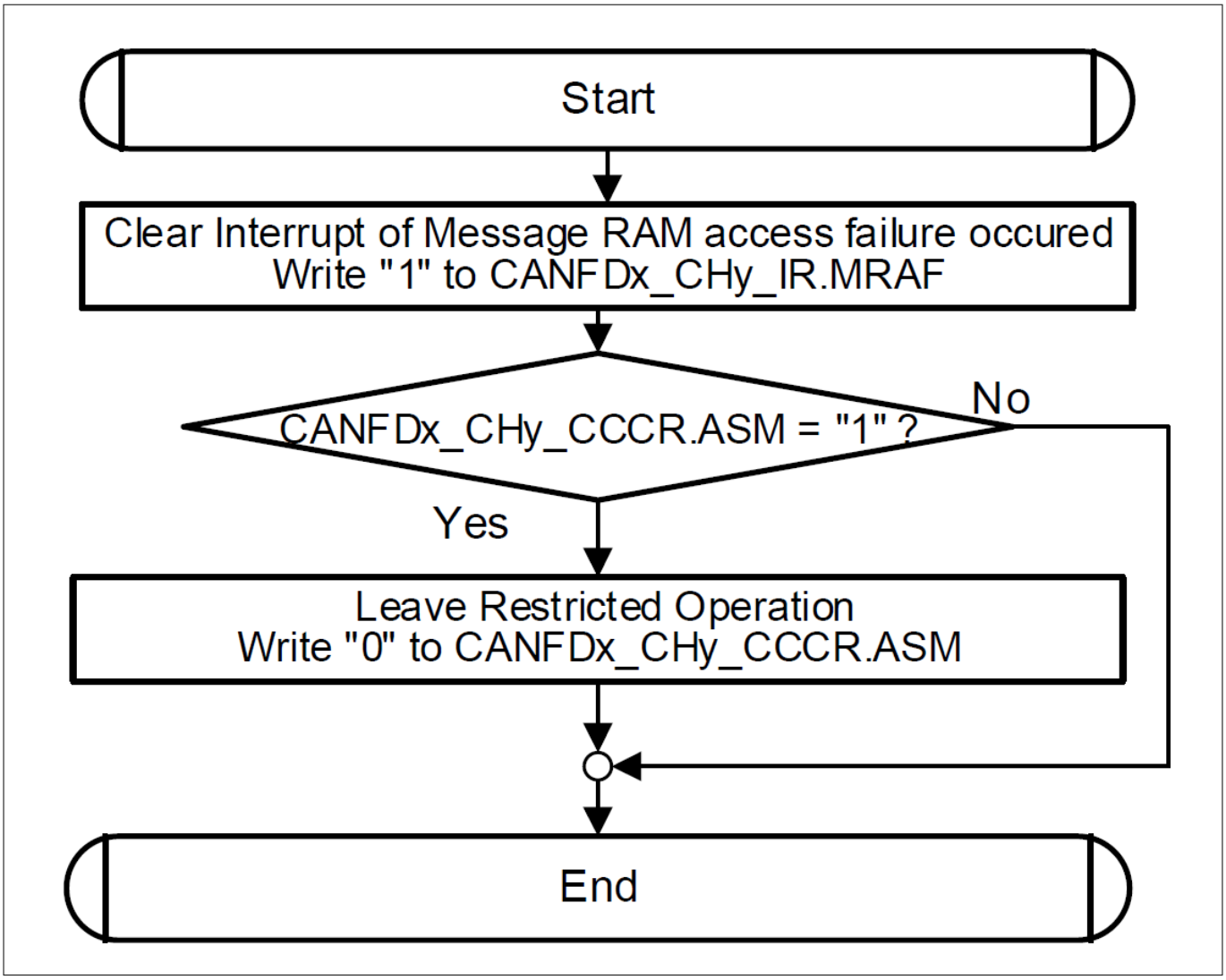


Figure 167 Message RAM access failure handling

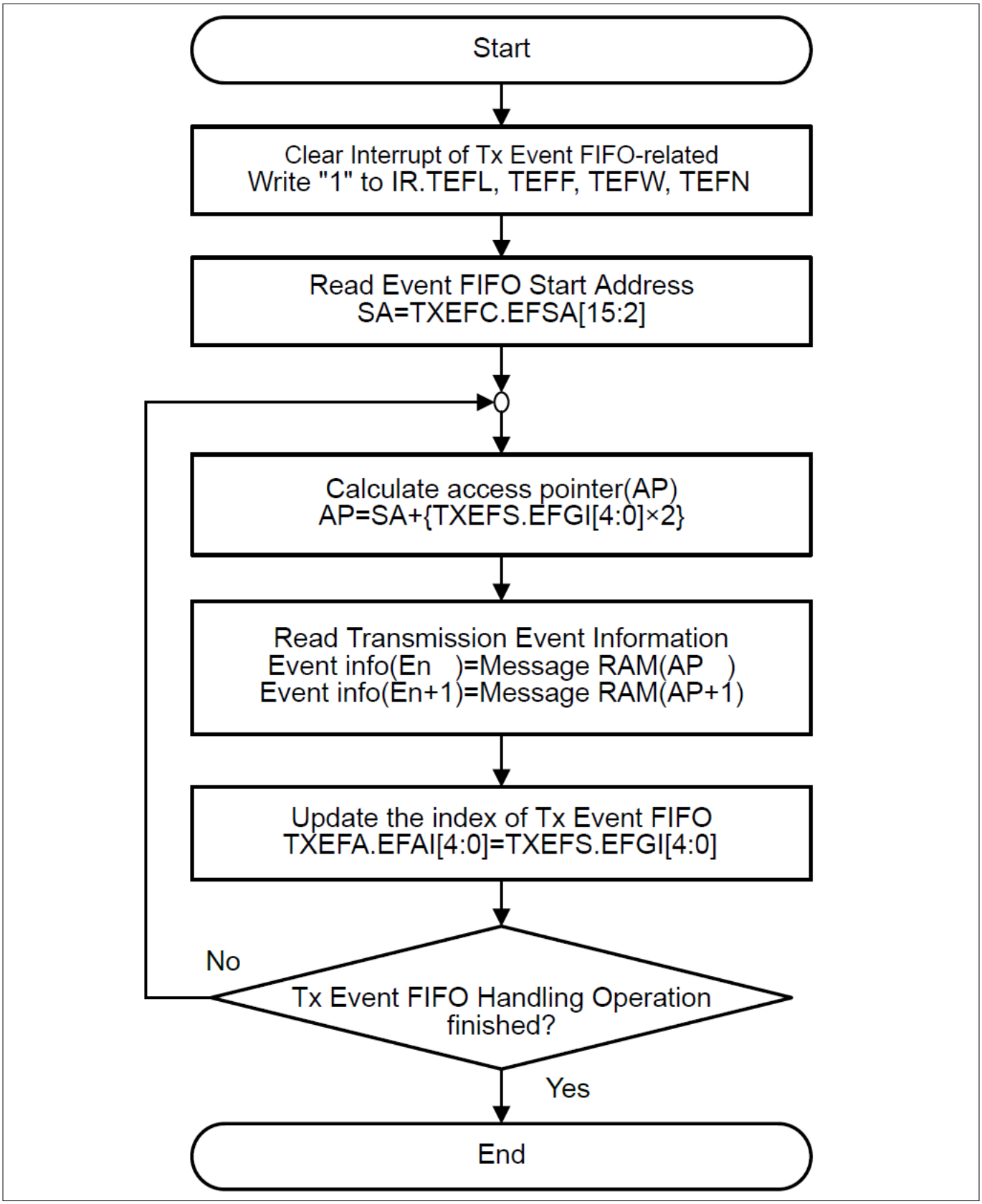


Figure 168 Tx event FIFO handling

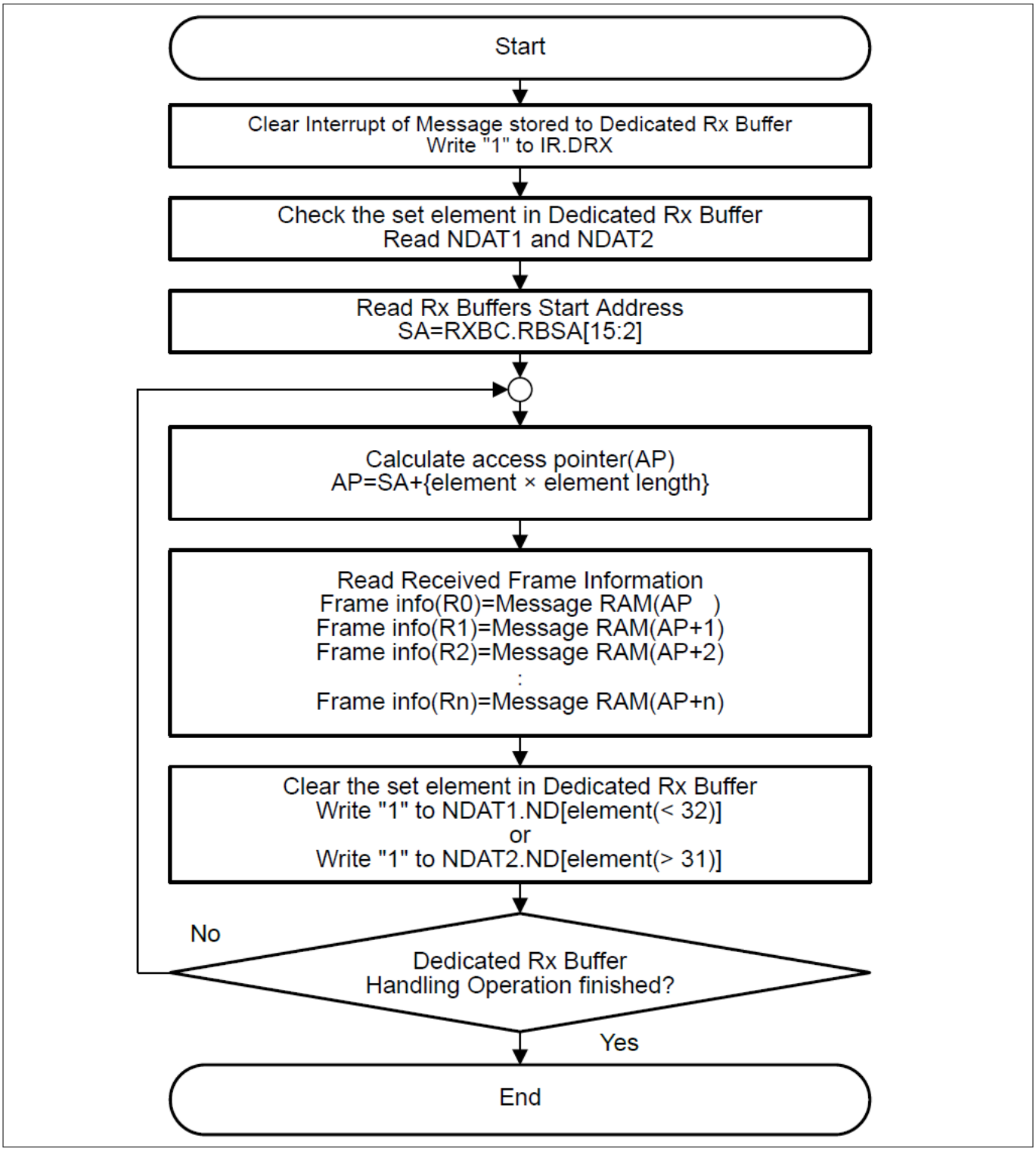


Figure 169 Dedicated Rx buffer handling

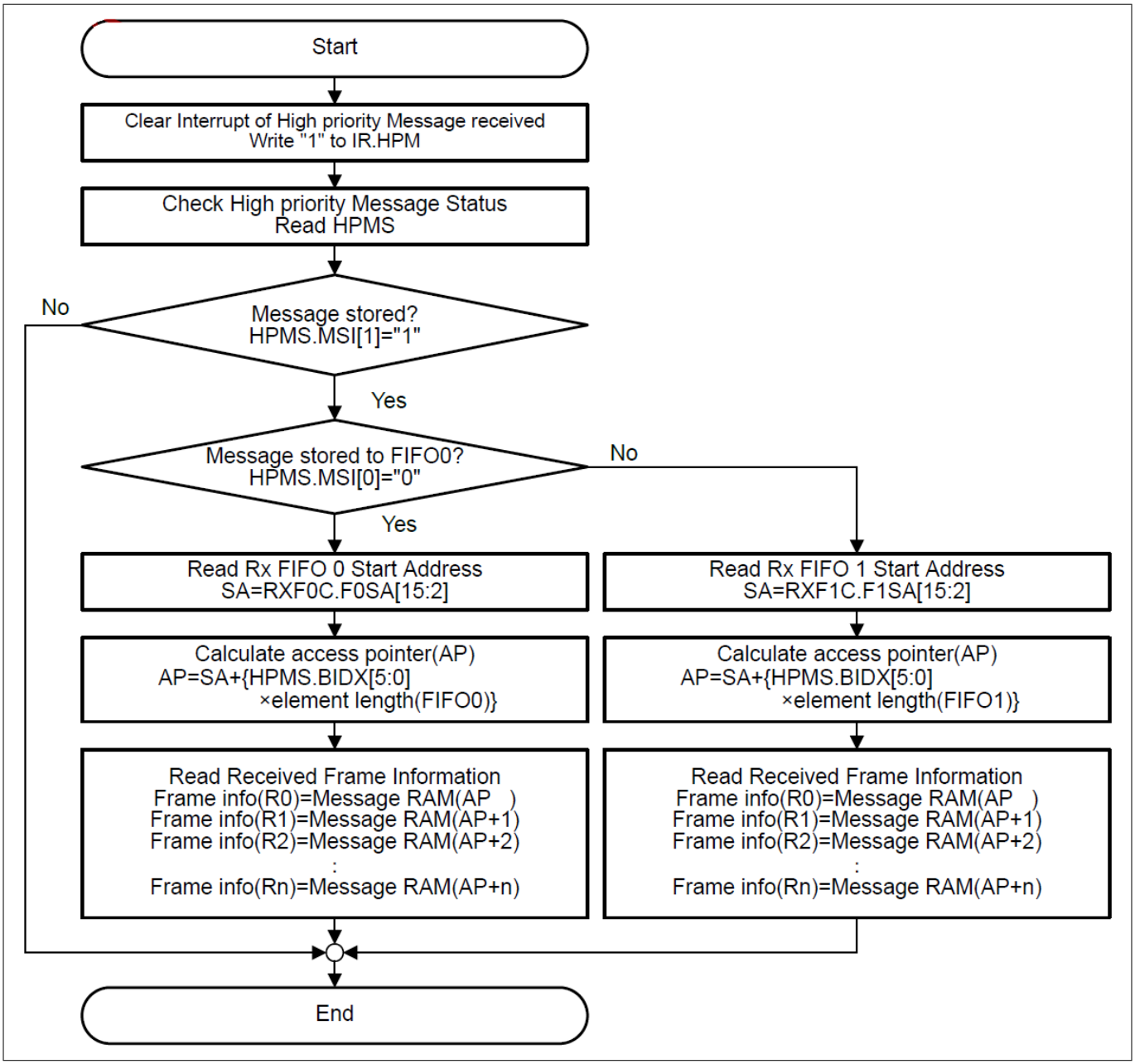


Figure 170 High-priority message handling

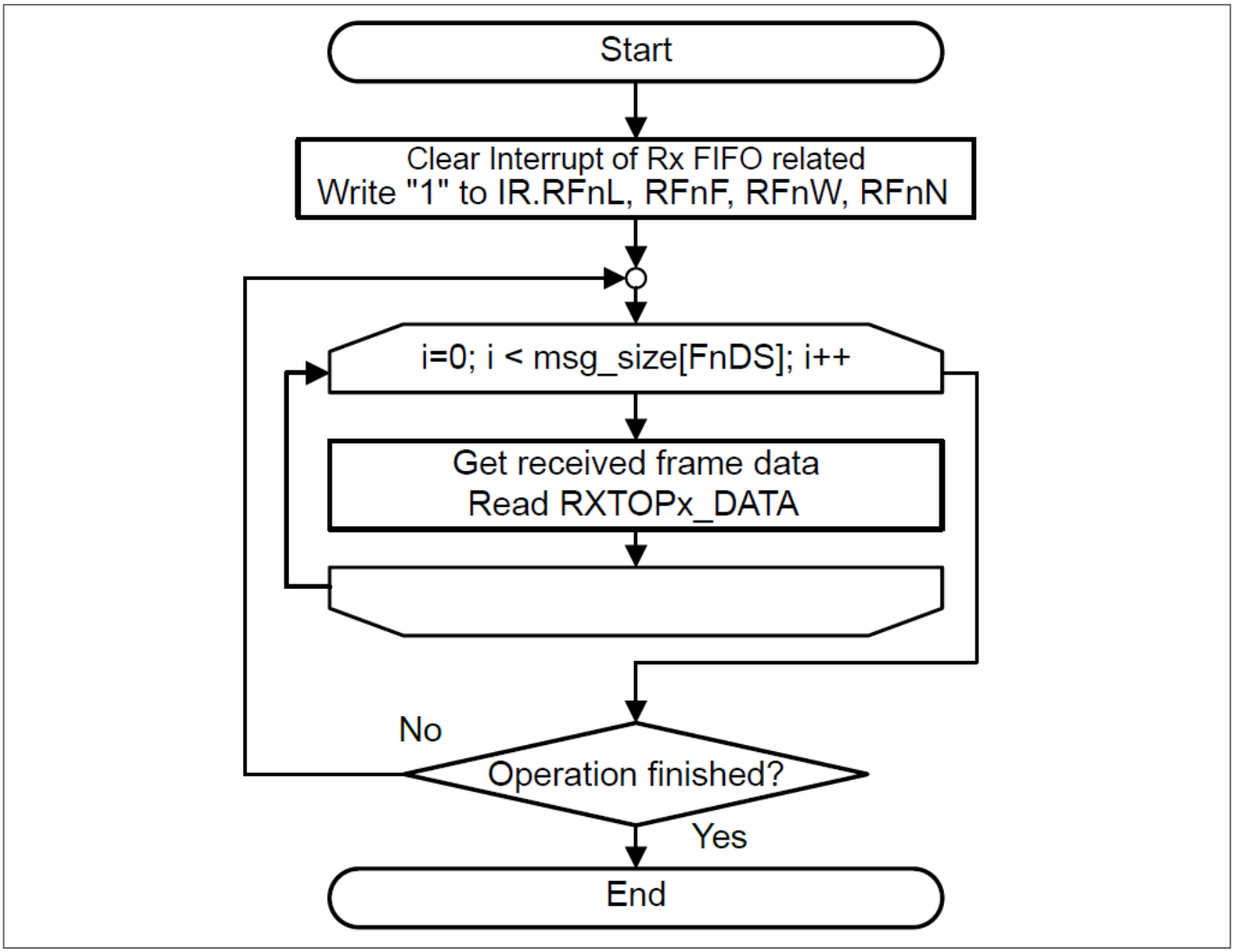


Figure 171 Rx FIFO top pointer handling

23.7 Registers

23.7.1 Register map

The M_TTCAN module allocates an address space of 512 bytes. All registers are organized as 32-bit registers. The module is accessible by the Host CPU through the generic slave interface using a data width of 8-bit (byte access), 16-bit (half-word access), or 32-bit (word access). Write access by the Host CPU to registers or bits marked with `P = Protected Write` is allowed only when `CCCR.CCE = 1` and `CCCR.INIT = 1`. A delay occurs between writing to a command register and updating the related status register bits due to clock-domain crossing.

Table 165 M_TTCAN register map

Address	Symbol	Name	Reset	ACC
0x000	CREL	Core Release Register	rrrd dddd	R
0x004	ENDN	Endian Register	8765 4321	R

(table continues...)

Table 165 (continued) M_TTCAN register map

Address	Symbol	Name	Reset	ACC
0x008	CUST	Customer Register	TBD	TBD
0x00C	DBTP	Data bit Timing and Prescaler Register	0000 0A33	RP
0x010	TEST	Test Register	0000 0000	RP
0x014	RWD	RAM Watchdog	0000 0000	RP
0x018	CCCR	CC Control Register	0000 0001	RWPp
0x01C	NBTP	Nominal Bit Timing and Prescaler Register	0600 0A03	RP
0x020	TSCC	Timestamp Counter Configuration	0000 0000	RP
0x024	TSCV	Timestamp Counter Value	0000 0000	RC
0x028	TOCC	Timeout Counter Configuration	FFFF 0000	RP
0x02C	TOCV	Timeout Counter Value	0000 FFFF	RC
0x030-03C		Reserved (4)	0000 0000	R
0x040	ECR	Error Counter Register	0000 0000	RX
0x044	PSR	Protocol Status Register	0000 0707	RXS
0x048	TDCR	Transmitter Delay Compensation Register	0000 0000	RP
0x04C		Reserved (1)	0000 0000	R
0x050	IR	Interrupt Register	0000 0000	RW
0x054	IE	Interrupt Enable	0000 0000	RW
0x058	ILS	Interrupt Line Select	0000 0000	RW
0x05C	ILE	Interrupt Line Enable	0000 0000	RW
0x060-07C		Reserved (8)	0000 0000	R
0x080	GFC	Global Filter Configuration	0000 0000	RP
0x084	SIDFC	Standard ID Filter Configuration	0000 0000	RP
0x088	XIDFC	Extended ID Filter Configuration	0000 0000	RP

(table continues...)

Table 165 (continued) M_TTCAN register map

Address	Symbol	Name	Reset	ACC
0x08C		Reserved (1)	0000 0000	R
0x090	XIDAM	Extended ID AND Mask	1FFF FFFF	RP
0x094	HPMS	High Priority Message Status	0000 0000	R
0x098	NDAT1	New Data 1	0000 0000	RW
0x09C	NDAT2	New Data 2	0000 0000	RW
0x0A0	RXF0C	Rx FIFO 0 Configuration	0000 0000	RP
0x0A4	RXF0S	Rx FIFO 0 Status	0000 0000	R
0x0A8	RXF0A	Rx FIFO 0 Acknowledge	0000 0000	RW
0x0AC	RXBC	Rx Buffer Configuration	0000 0000	RP
0x0B0	RXF1C	Rx FIFO 1 Configuration	0000 0000	RP
0x0B4	RXF1S	Rx FIFO 1 Status	0000 0000	R
0x0B8	RXESC	Rx FIFO 1 Acknowledge	0000 0000	RW
0x0BC	TXBC	Rx Buffer/FIFO Element Size Configuration	0000 0000	RP
0x0C0	TXFQS	Tx Buffer Configuration	0000 0000	RP
0x0C4	TXFQS	Tx FIFO/Queue Status	0000 0000	R
0x0C8	TXESC	Tx Buffer Element Size Configuration	0000 0000	RP
0x0CC	TXBRP	Tx Buffer Request Pending	0000 0000	R
0x0D0	TXBAR	Tx Buffer Add Request	0000 0000	RW
0x0D4	TXBCR	Tx Buffer Cancellation Request	0000 0000	RW
0x0D8	TXBTO	Tx Buffer Transmission Occurred	0000 0000	R

(table continues...)

Table 165 (continued) M_TTCAN register map

Address	Symbol	Name	Reset	ACC
0x0DC	TXBCF	Tx Buffer Cancellation Finished	0000 0000	R
0x0E0	TXBTIE	Tx Buffer Transmission Interrupt Enable	0000 0000	RW
0x0E4	TXBCIE	Tx Buffer Cancellation Finished Interrupt Enable	0000 0000	RW
0x0E8-0x0EC		Reserved (2)	0000 0000	R
0x0F0	TXEFC	Tx Event FIFO Configuration	0000 0000	RP
0x0F4	TXEFS	Tx Event FIFO Status	0000 0000	R
0x0F8	TXEFA	Tx Event FIFO Acknowledge	0000 0000	RW
0x0FC		Reserved (1)	0000 0000	R
0x100	TTTMC	TT Trigger Memory Configuration	0000 0000	RP
0x104	TTRMC	TT Reference Message Configuration	0000 0000	RP
0x108	TTOCF	TT Operation Configuration	0001 0000	RP
0x10C	TTMLM	TT Matrix Limits	0000 0000	RP
0x110	TURCF	TUR Configuration	1000 0000	RP
0x114	TTOCN	TT Operation Control	0000 0000	RP
0x118	TTGTP	TT Global Time Preset	0000 0000	RW
0x11C	TTTMK	TT Time Mark	0000 0000	RW
0x120	TTIR	TT Interrupt Register	0000 0000	RW
0x124	TTIE	TT Interrupt Enable	0000 0000	RW
0x128	TTILS	TT Interrupt Line Select	0000 0000	RW
0x12C	TTOST	TT Operation Status	0000 0080	R
0x130	TURNA	TUR Numerator Actual	0001 0000	R

(table continues...)

Table 165 (continued) M_TTCAN register map

Address	Symbol	Name	Reset	ACC
0x134	TTLGT	TT Local and Global Time	0000 0000	R
0x138	TTCTC	TT Cycle Time and Count	003F 0000	R
0x13C	TTCPT	TT Capture Time	0000 0000	R
0x140	TTCSM	TT Cycle Sync Mark	0000 0000	R
0x144-1FC	-	Reserved (47)	0000 0000	R

Access types:
R = **Read**
S = **Set on Read**
X = **Reset on Read**
W = **Write**
P = **Protected Write**
p = **Protected set**
C = **Clear/Preset on Write**
r = **Release**
d = **Date**

23.7.1.1 Access to reserved register addresses

If application software accesses a reserved address in the `M_TTCAN` register map (read or write), the interrupt flag `IR.ARA` is set. If enabled, the interrupt is signaled through the assigned interrupt line `m_ttcan_int0` or `m_ttcan_int1`.

When the CPU interface supports byte or half-word accesses, this interrupt flag is set only for accesses to reserved addresses where all four bytes are reserved.

If an optional add-on module such as DMU or TSU is connected to `M_TTCAN`, this flag is also set when a reserved address of that module is accessed. Additionally, accesses to reserved addresses are signaled directly to the host CPU by activating the output signal `m_ttcan_aei_ara` when a reserved address is accessed.

23.7.2 Customer Register

Address `0x08` is reserved for an optional 32-bit customer-specific register. The Customer Register is intended to store customer-specific configuration, control, and status bits. The description of its functionality is not included in this document.

23.7.3 Core Release Register (CREL)

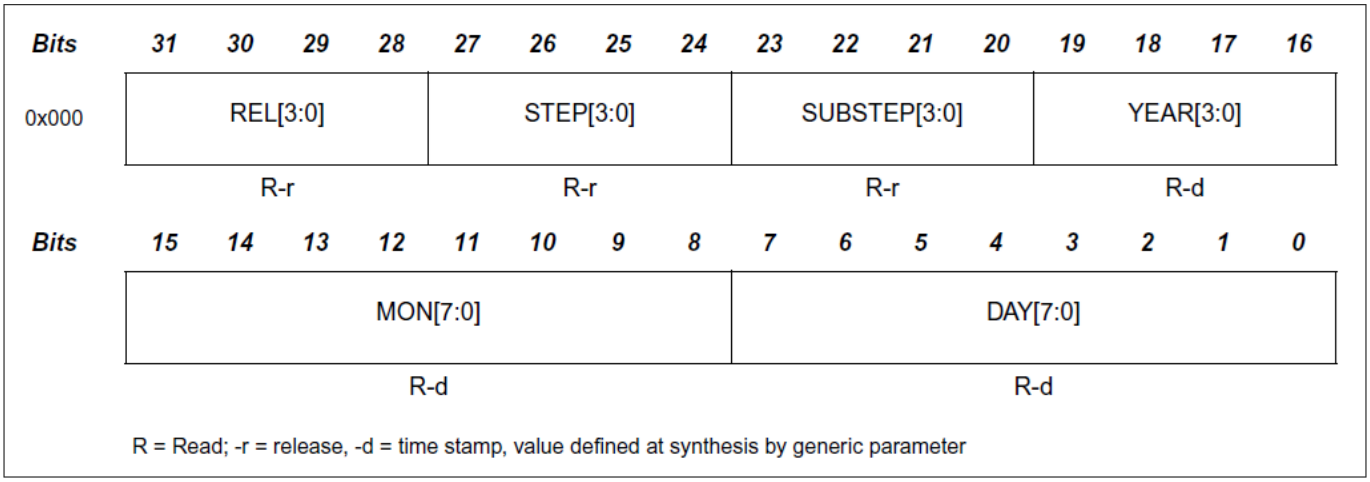


Figure 172 Core Release Register (addresses 0x000)

- **Bits 31:28 REL[3:0]**: Core Release
  One digit, BCD-coded
- **Bits 27:24 STEP[3:0]**: Step of Core Release
  One digit, BCD-coded
- **Bits 23:20 SUBSTEP[3:0]**: Sub-step of Core Release
  One digit, BCD-coded
- **Bits 19:16 YEAR[3:0]**: Time Stamp Year
  One digit, BCD-coded
  This field is set by a generic parameter during M_TTCAN synthesis
- **Bits 15:8 MON[7:0]**: Time Stamp Month
  Two digits, BCD-coded
  This field is set by a generic parameter during M_TTCAN synthesis
- **Bits 7:0 DAY[7:0]**: Time Stamp Day
  Two digits, BCD-coded
  This field is set by a generic parameter during M_TTCAN synthesis

Table 166 Example for coding of revisions

Release	Step	Sub-step	Year	Month	Day	Name
0	1	0	0	03	10	Revision: 0.1.0, Date: 2010/03/10

23.7.4 Endian Register (ENDN)

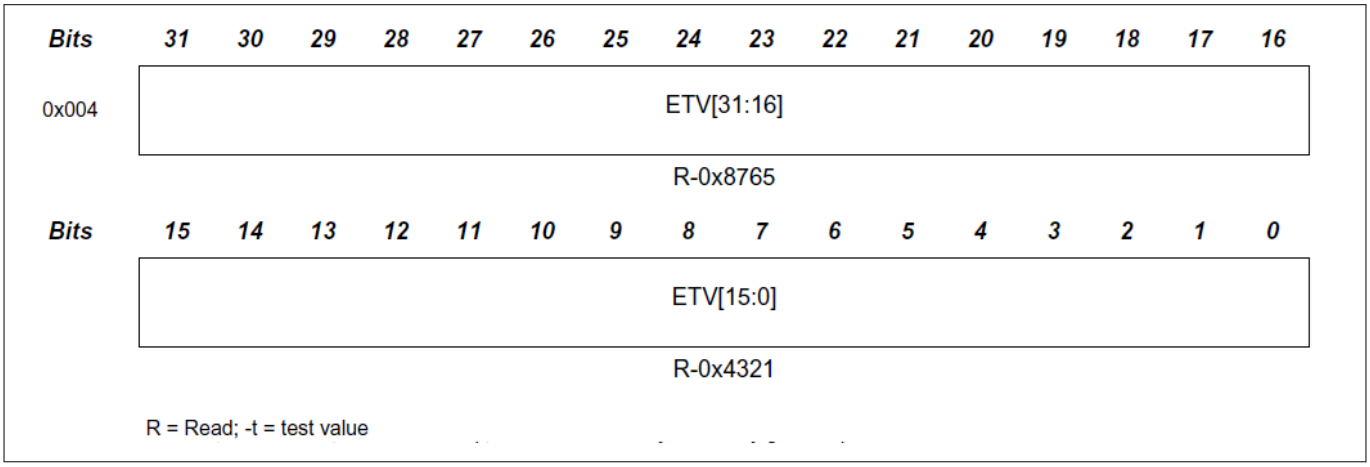


Figure 173 Endian Register (address 0x004)

Bits 31:0 `ETV[31:0]`: Endianness Test Value

The endianness test value is `0x87654321`.

23.7.5 Data Bit Timing and Prescaler Register (DBTP)

This register is writable only when `CCCR.CCE` and `CCCR.INIT` bits are set. The CAN bit time can be programmed in the range of 4 to 49 time quanta (tq). The CAN time quantum can be programmed in the range of 1 to 32 `m_ttcan_cclk` periods.

The time quantum is calculated as:

tq = (DBRP + 1) × mtq

`DTSEG1` is the sum of `Prop_Seg` and `Phase_Seg1`. `DTSEG2` is `Phase_Seg2`.

Therefore, the bit time length is:

- For programmed values: `[DTSEG1 + DTSEG2 + 3] tq`
- For functional values: `[Sync_Seg + Prop_Seg + Phase_Seg1 + Phase_Seg2] tq`

The Information Processing Time (IPT) is zero, meaning data for the next bit is available at the first clock edge after the sample point.

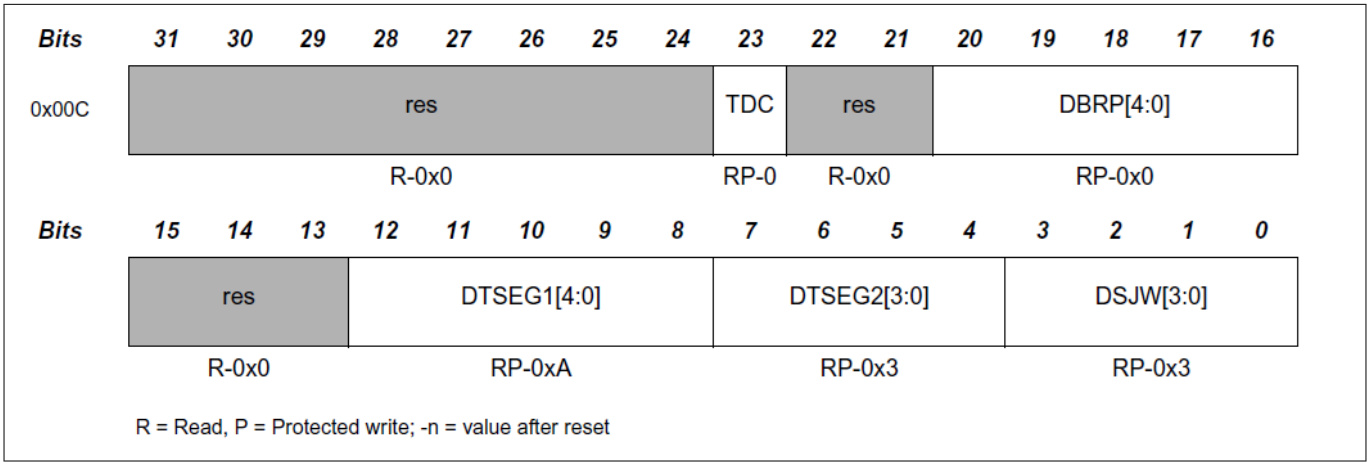


Figure 174 Data Bit Timing and Prescaler Register (address 0x00C)

- **Bit 23 `TDC`**: Transmitter Delay Compensation

0 = Transmitter Delay Compensation disabled

1 = Transmitter Delay Compensation enabled

- **Bits 20:16 DBRP[4:0]**: Data Bit Rate Prescaler

  Range: 0x00–0x1F

  This value divides the oscillator frequency to generate the bit-time quanta. The bit time is built from a multiple of this quanta

  Valid values: 0–31. When TDC = 1, the range is limited to 0 and 1

  Hardware interpretation: One more than the programmed value is used
- **Bits 12:8 DTSEG1[4:0]**: Data Time Segment Before Sample Point

  Range: 0x00–0x1F

  Valid values: 0–31

  Hardware interpretation: One more than the programmed value is used
- **Bits 7:4 DTSEG2[3:0]**: Data Time Segment After Sample Point

  Range: 0x0–0xF

  Valid values: 0–15

  Hardware interpretation: One more than the programmed value is used
- **Bits 3:0 DSJW[3:0]**: Data (Re)Synchronization Jump Width

  Range: 0x0–0xF

  Valid values: 0–15

  Hardware interpretation: One more than the programmed value is used

***Notes**:*

1. *With a CAN clock m_ttcan_cclk of 8 MHz, the reset value 0x00000A33 configures M_TTCAN for a data-phase bit rate of 500 Kbps*

2. *The bit rate configured for the CAN FD data phase via DBTP must be greater than or equal to the bit rate configured for the arbitration phase via NBTP*

23.7.6 Test Register (TEST)

Write access to the Test Register is enabled only when bit CCCR.TEST is set to 1. All Test Register functions return to their reset values when CCCR.TEST is cleared.

Loopback mode and software control of pin m_ttcan_tx are hardware test modes. Programming Tx ≠ "00" may disturb message transfer on the CAN bus.

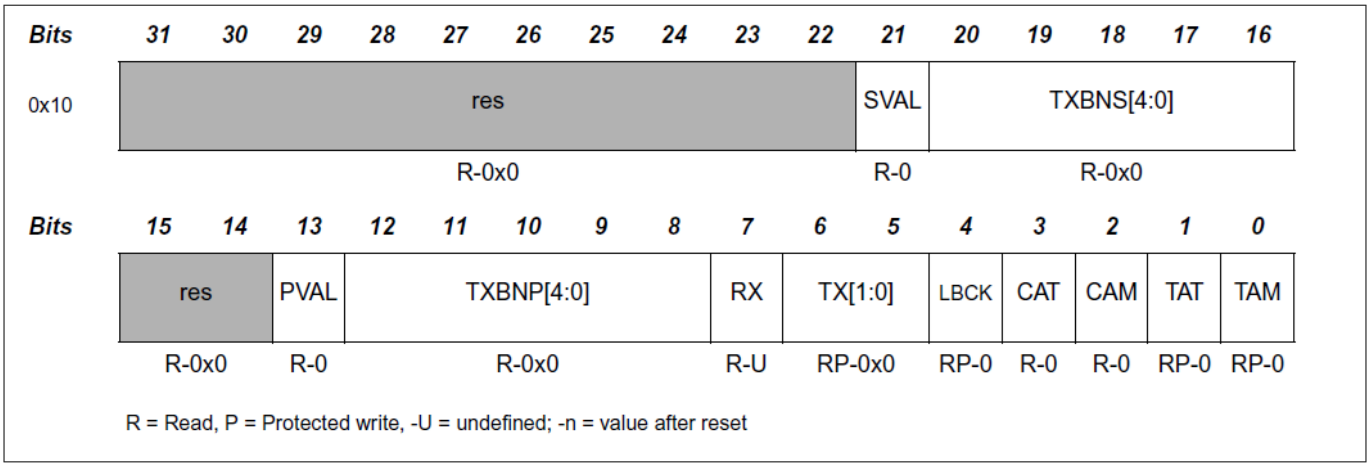


Figure 175 Test Register (address 0x010)

- **Bit 21 SVAL**: Started Valid
  0 = Value of TXBNS not valid
  1 = Value of TXBNS valid
- **Bits 20:16 TXBNS[4:0]**: Tx Buffer Number Started
  Range: 0x00–0x1F
  Indicates the Tx Buffer number of the message whose transmission was started last. Valid when SVAL is set
  Valid values: 0–31
- **Bit 13 PVAL**: Prepared Valid
  0 = Value of TXBNP not valid
  1 = Value of TXBNP valid
- **Bits 12:8 TXBNP[4:0]**: Tx Buffer Number Prepared
  Range: 0x00–0x1F
  Indicates the Tx Buffer number of the message that is ready for transmission. Valid when PVAL is set
  Valid values: 0–31
- **Bit 7 Rx**: Receive Pin
  Monitors the actual value of pin m_ttcan_rx
  0 = CAN bus is dominant (m_ttcan_rx = 0)
  1 = CAN bus is recessive (m_ttcan_rx = 1)
- **Bits 6:5 TX[1:0]**: Control of Transmit Pin
  00 = Reset value; m_ttcan_tx controlled by CAN Core, updated at end of CAN bit time
  01 = Sample Point can be monitored at pin m_ttcan_tx
  10 = Dominant (0) level at pin m_ttcan_tx
  11 = Recessive (1) level at pin m_ttcan_tx
- **Bit 4 LBCK**: Loopback mode
  0 = Reset value; Loopback mode disabled
  1 = Loopback mode enabled
- **Bit 3 CAT**: Check ASC Transmit Control
  Monitors level at output pin m_ttcan_asct
  0 = Output pin m_ttcan_asct = 0
  1 = Output pin m_ttcan_asct = 1

- **Bit 2 CAM**: Check ASC Multiplexer Control
  Monitors level at output pin m_ttcan_ascm
  0 = Output pin m_ttcan_ascm = 0
  1 = Output pin m_ttcan_ascm = 1
- **Bit 1 TAT**: Test ASC Transmit Control
  Controls output pin m_ttcan_asct in test mode, ORed with signal from FSE
  0 = Level at pin m_ttcan_asct controlled by FSE
  1 = Level at pin m_ttcan_asct = 1
- **Bit 0 TAM**: Test ASC Multiplexer Control
  Controls output pin m_ttcan_ascm in test mode, ORed with signal from FSE
  0 = Level at pin m_ttcan_ascm controlled by FSE
  1 = Level at pin m_ttcan_ascm = 1

23.7.7 RAM watchdog (RWD)

The RAM watchdog monitors the READY output of the message RAM (m_ttcan_aeim_ready). A message RAM access via the M_TTCAN generic master interface (m_ttcan_aeim_sel active) starts the message RAM watchdog counter with the value configured in RWD.WDC.

The counter is reloaded with RWD.WDC when the message RAM signals successful completion by activating its READY output. If there is no response from the message RAM before the counter reaches zero, the counter stops and interrupt flag IR.WDI is set.

The RAM watchdog counter is clocked by the host clock (m_ttcan_hclk).

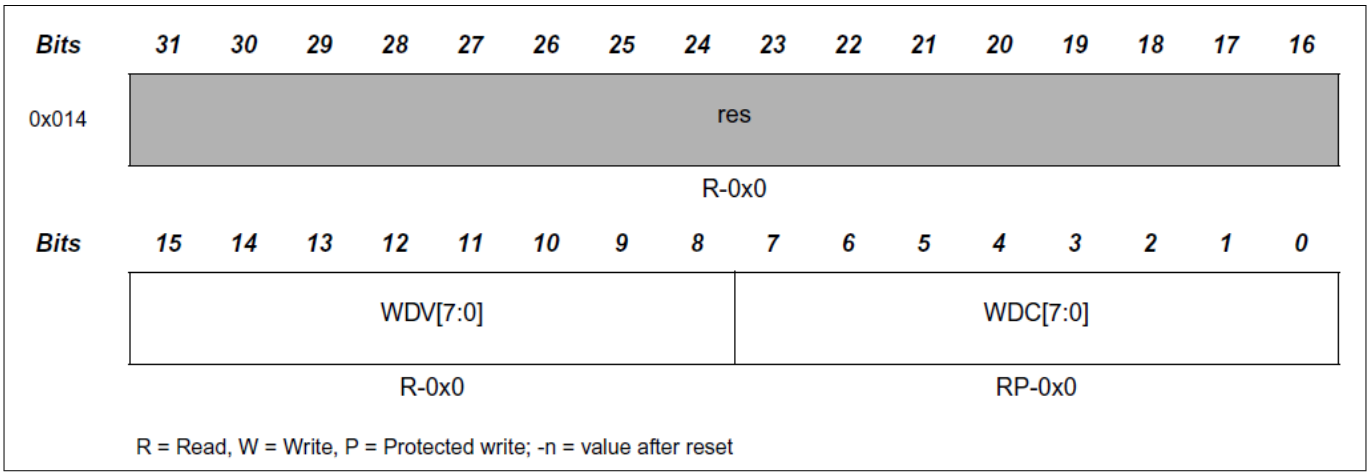


Figure 176 RAM watchdog (address 0x014)

- **Bits 15:8 WDV[7:0]**: Watchdog Value
  Indicates the actual message RAM watchdog counter value
- **Bits 7:0 WDC[7:0]**: Watchdog Configuration
  Defines the start value of the message RAM watchdog counter
  When the reset value is 00, the counter is disabled

23.7.8 CC Control Register (CCCR)

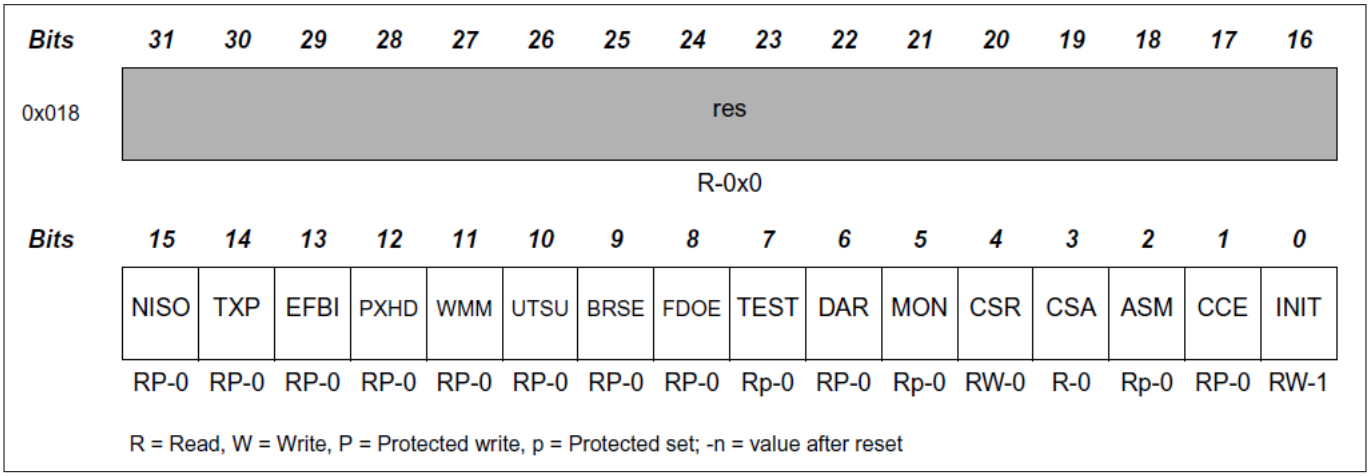


Figure 177 CC Control Register (address 0x018)

- **Bit 15 NISO**: Non-ISO operation
  If this bit is set, M_TTCAN uses the CAN FD frame format as specified by Bosch CAN FD specification v1.0
  0 = CAN FD frame format according to ISO 11898-1:2015
  1 = CAN FD frame format according to Bosch CAN FD specification v1.0

  ***Note**: When the generic parameter iso_only_g is set to 1 during hardware synthesis, this bit becomes reserved and is read as 0. In this case, M_TTCAN always operates with the CAN FD frame format according to ISO 11898-1:2015*

- **Bit 14 TXP**: Transmit Pause
  If this bit is set, M_TTCAN pauses for two CAN bit times before starting the next transmission after successfully transmitting a frame
  0 = Transmit pause disabled
  1 = Transmit pause enabled
- **Bit 13 EFBI**: Edge Filtering during Bus Integration
  0 = Edge filtering disabled
  1 = Two consecutive dominant tq required to detect an edge for hard synchronization
- **Bit 12 PXHD**: Protocol Exception Handling Disable
  0 = Protocol exception handling enabled
  1 = Protocol exception handling disabled

  ***Note**: When protocol exception handling is disabled, M_TTCAN transmits an error frame when it detects a protocol exception condition*

- **Bit 11 WMM**: Wide Message Marker
  Enables the use of 16-bit wide message markers. When WMM = 1, 16-bit internal timestamping is disabled for the Tx Event FIFO
  0 = 8-bit Message Marker used
  1 = 16-bit Message Marker used, replacing 16-bit timestamps in Tx Event FIFO
- **Bit 10 UTSU**: Use Timestamping Unit
  When UTSU is set, 16-bit wide message markers are also enabled regardless of the value of WMM
  0 = Internal timestamping

1 = External timestamping by TSU

***Note**:* *When generic parameter `connected_tsu_g = 0`, there is no TSU connected to M_TTCAN. In this case, bit UTSU is fixed to zero by synthesis*

- **Bit 9 BRSE**: Bit Rate Switch Enable
  0 = Bit rate switching for transmissions disabled
  1 = Bit rate switching for transmissions enabled

  ***Note**:* *When CAN FD operation is disabled (FDOE = 0), BRSE is not evaluated*

- **Bit 8 FDOE**: FD Operation Enable
  0 = FD operation disabled
  1 = FD operation enabled
- **Bit 7 TEST**: Test Mode Enable
  0 = Normal operation; register TEST holds reset values
  1 = Test Mode; write access to register TEST enabled
- **Bit 6 DAR**: Disable Automatic Retransmission
  0 = Automatic retransmission of messages not transmitted successfully enabled
  1 = Automatic retransmission disabled
- **Bit 5 MON**: Bus Monitoring mode
  Bit MON can only be set by the Host when both CCE and INIT are set to 1. The bit can be reset by the host at any time
  0 = Bus Monitoring mode disabled
  1 = Bus Monitoring mode enabled
- **Bit 4 CSR**: Clock Stop Request
  0 = No clock stop requested
  1 = Clock stop requested. When clock stop is requested, first INIT and then CSA will be set after all pending transfer requests have been completed and the CAN bus reaches idle
- **Bit 3 CSA**: Clock Stop Acknowledge
  0 = No clock stop acknowledged
  1 = M_TTCAN may be set in power-down by stopping m_ttcan_hclk and m_ttcan_cclk
- **Bit 2 ASM**: Restricted Operation mode
  Bit ASM can only be set by the host when both CCE and INIT are set to 1. The bit can be reset by the host at any time
  0 = Normal CAN operation
  1 = Restricted Operation mode active
- **Bit 1 CCE**: Configuration Change Enable
  0 = CPU has no write access to protected configuration registers
  1 = CPU has write access to protected configuration registers (while CCCR.INIT = 1)
- **Bit 0 INIT**: Initialization
  0 = Normal operation
  1 = Initialization started

***Note**: Due to synchronization between two clock domains, there may be a delay until the value written to `INIT` can be read back. The programmer must ensure that the previous value written to `INIT` has been accepted by reading `INIT` before setting it to a new value*

23.7.9 Nominal Bit Timing and Prescaler Register (NBTP)

This register is writable only when bits `CCCR.CCE` and `CCCR.INIT` are set. The CAN bit time can be programmed in the range of 4 to 385 time quanta (tq). The CAN time quantum can be programmed in the range of 1 to 512 `m_ttcan_cclk` periods.

The time quantum is calculated as:

`tq = (NBRP + 1) × mtq`

`NTSEG1` is the sum of `Prop_Seg` and `Phase_Seg1`.

`NTSEG2` is `Phase_Seg2`.

Therefore, the bit time length is:

- **For programmed values**: `[NTSEG1 + NTSEG2 + 3] tq`
- **For functional values**: `[Sync_Seg + Prop_Seg + Phase_Seg1 + Phase_Seg2] tq`

The Information Processing Time (IPT) is zero, meaning data for the next bit is available at the first clock edge after the sample point.

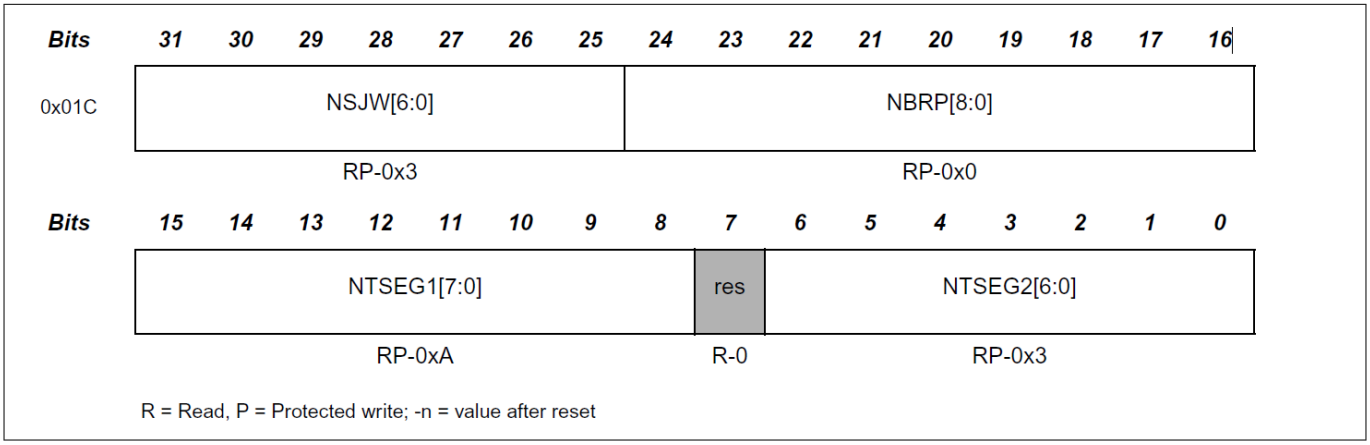


Figure 178 Nominal Bit Timing and Prescaler Register (address 0x01C)

- **Bits 31:25 `NSJW[6:0]`**: Nominal (Re)Synchronization Jump Width
  Range: `0x00–0x7F`
  Valid values: `0–127`
  Hardware interpretation: One more than the programmed value is used
- **Bits 24:16 `NBRP[8:0]`**: Nominal Bit Rate Prescaler
  Range: `0x000–0x1FF`
  Divides the oscillator frequency to generate the bit-time quanta. The bit time is built from a multiple of this quanta
  Valid values: `0–511`
  Hardware interpretation: One more than the programmed value is used
- **Bits 15:8 `NTSEG1[7:0]`**: Nominal Time Segment Before Sample Point
  Range: `0x01–0xFF`
  Valid values: `1–255`

Hardware interpretation: One more than the programmed value is used

- **Bits 6:0 NTSEG2[6:0]**: Nominal Time Segment After Sample Point
  Range: 0x01–0x7F
  Valid values: 1–127
  Hardware interpretation: One more than the programmed value is used

***Note**: With a CAN clock m_ttcan_cclk of 8 MHz, the reset value 0x06000A03 configures M_TTCAN for a bit rate of 500 Kbps.*

23.7.10 Timestamp Counter Configuration (TSCC)

Configuration of the internal 16-bit timestamp counter. The internal timestamp counter is enabled when CCCR.UTSU = 0.

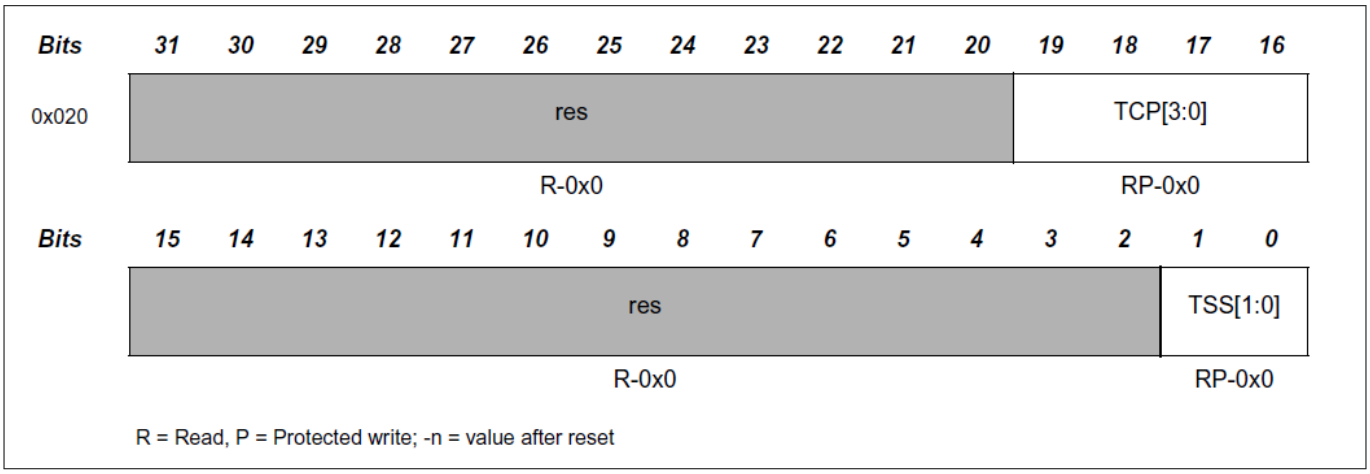


Figure 179 Timestamp Counter Configuration (address 0x020)

- **Bits 19:16 TCP[3:0]**: Timestamp Counter Prescaler
  Range: 0x0–0xF
  Configures the timestamp and timeout counters' time unit in multiples of CAN bit times [1…16]
  Hardware interpretation: One more than the programmed value is used

  ***Note**: With CAN FD, an external counter is required for timestamp generation (TSS = 10)*

- **Bits 1:0 TSS[1:0]**: Timestamp Select
  00 = Timestamp counter value always 0x0000
  01 = Timestamp counter value incremented according to TCP
  10 = External timestamp counter value used
  11 = Same as 00

23.7.11 Timestamp Counter Value (TSCV)

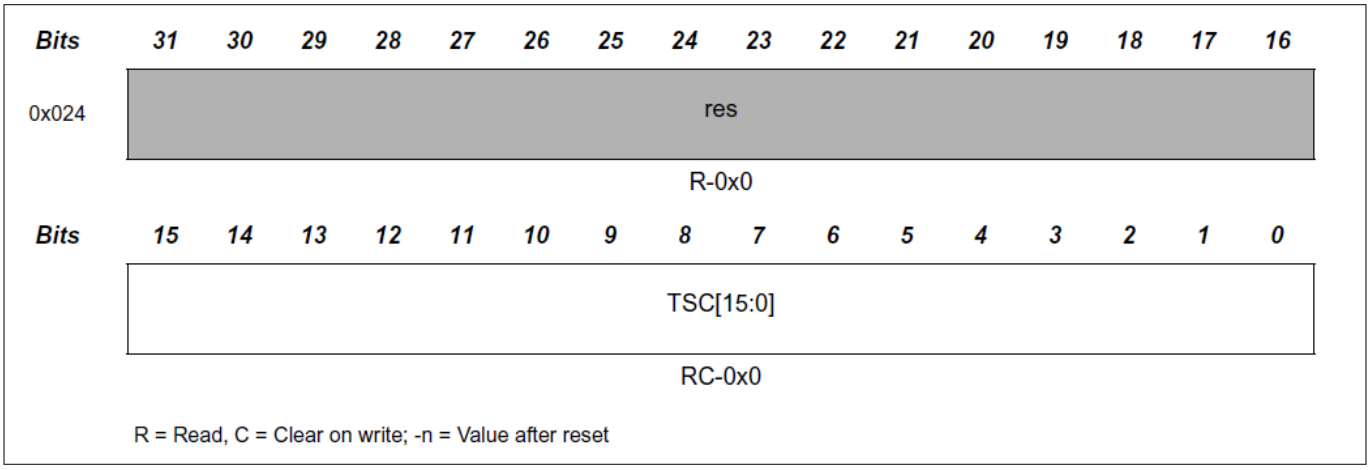


Figure 180 Timestamp Counter Value (address 0x024)

Bits 15:0 `TSC[15:0]`: Timestamp Counter

- Captures the internal or external timestamp counter value at the start of a frame (both Rx and Tx)
- When `TSCC.TSS = 01`, the timestamp counter increments in multiples of CAN bit times `[1…16]` based on the configuration of `TSCC.TCP`
- A wrap-around sets interrupt flag `IR.TSW`
- Write access resets the counter to zero
- When `TSCC.TSS = 10`, `TSC` reflects the external timestamp counter value; write access has no effect

***Notes**:*

1. *A wrap-around occurs when the timestamp counter changes from a non-zero value to zero without a write access to `TSC`*

2. ***Byte access**: Writing any of the register bytes (3, 2, 1, or 0) resets the timestamp counter*

23.7.12 Timeout Counter Configuration (TOCC)

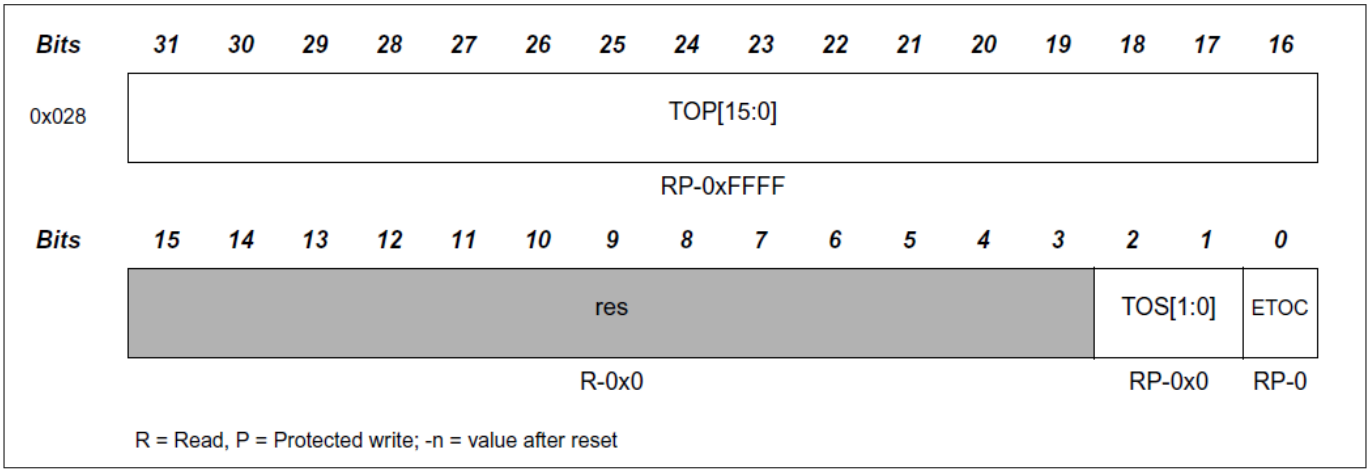


Figure 181 Timeout Counter Configuration (address 0x028)

- **Bits 31:16 `TOP[15:0]`**: Timeout Period
  Start value for the Timeout Counter down-counter

Configures the timeout period

- **Bits 2:1 TOS[1:0]**: Timeout Select
  In Continuous mode, a write to TOCV presets the counter to the value in TOCC.TOP and the counter continues down-counting
  When a FIFO controls the Timeout Counter, an empty FIFO presets the counter to the value in TOCC.TOP
  The counter starts down-counting when the first FIFO element is stored
  - 00 Continuous operation
  - 01 Timeout controlled by Tx Event FIFO
  - 10 Timeout controlled by Rx FIFO 0
  - 11 Timeout controlled by Rx FIFO 1
- **Bit 0 ETOC**: Enable Timeout Counter
  - 0 Timeout Counter disabled
  - 1 Timeout Counter enabled

23.7.13 Timeout Counter Value (TOCV)

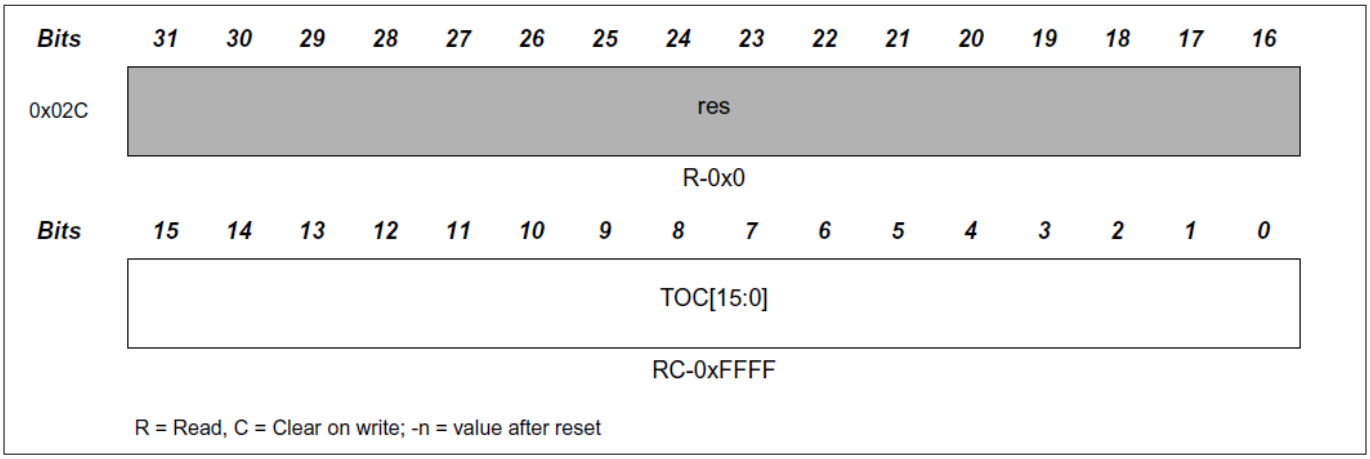


Figure 182 Timeout Counter Value (address 0x02C)

Bits 15:0 TOC[15:0]: Timeout Counter

The Timeout Counter decrements in multiples of CAN bit times from 1 to 16, based on the TSCC.TCP configuration When the counter decrements to zero, it sets interrupt flag IR.TOO and stops the Timeout Counter Configure start and reset or restart conditions through TOCC.TOS

***Note**: Byte access when TOCC.TOS = "00", writing to register byte 3, 2, 1, or 0 presets the Timeout Counter.*

23.7.14 Error Counter Register (ECR)

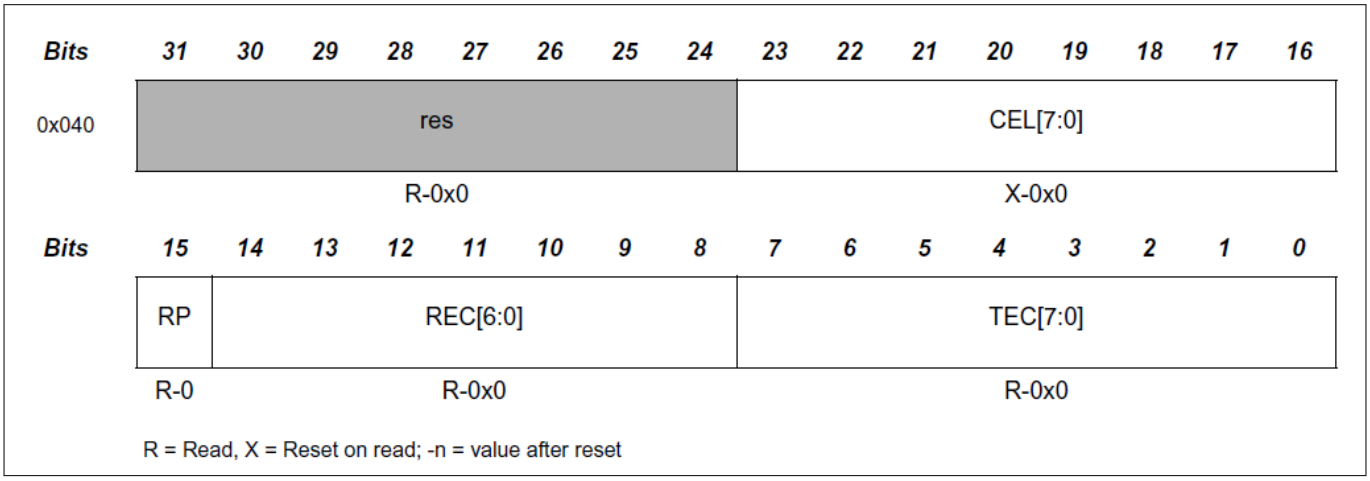


Figure 183 Error Counter Register (address 0x040)

- **Bits 23:16 CEL[7:0]**: CAN Error Logging
  The counter increments each time a CAN protocol error increments the 8-bit Transmit Error Counter TEC or the 7-bit Receive Error Counter REC
  The counter also increments when the Bus_Off limit is reached
  The counter does not increment when only RP is set without changing REC
  CEL increments after REC or TEC increments
  Read access to CEL resets the counter
  The counter saturates at 0xFF
  When the counter is at 0xFF, the next increment of TEC or REC sets interrupt flag IR.ELO
  Byte access: Reading byte 2 resets CEL to zero, reading bytes 3, 1, or 0 has no impact
- **Bit 15 RP**: Receive Error Passive
  - 0 The Receive Error Counter is below the error-passive level of 128
  - 1 The Receive Error Counter has reached the error-passive level of 128
- **Bits 14:8 REC[6:0]**: Receive Error Counter
  Actual state of the Receive Error Counter, values from 0 to 127
- **Bits 7:0 TEC[7:0]**: Transmit Error Counter
  Actual state of the Transmit Error Counter, values from 0 to 255

***Note**: When CCCR.ASM is set, the CAN protocol controller does not increment TEC and REC when it detects a CAN protocol error CEL still increments. This behavior enables monitoring of collisions between CAN frames and ASC frames.*

23.7.15 Protocol Status Register (PSR)

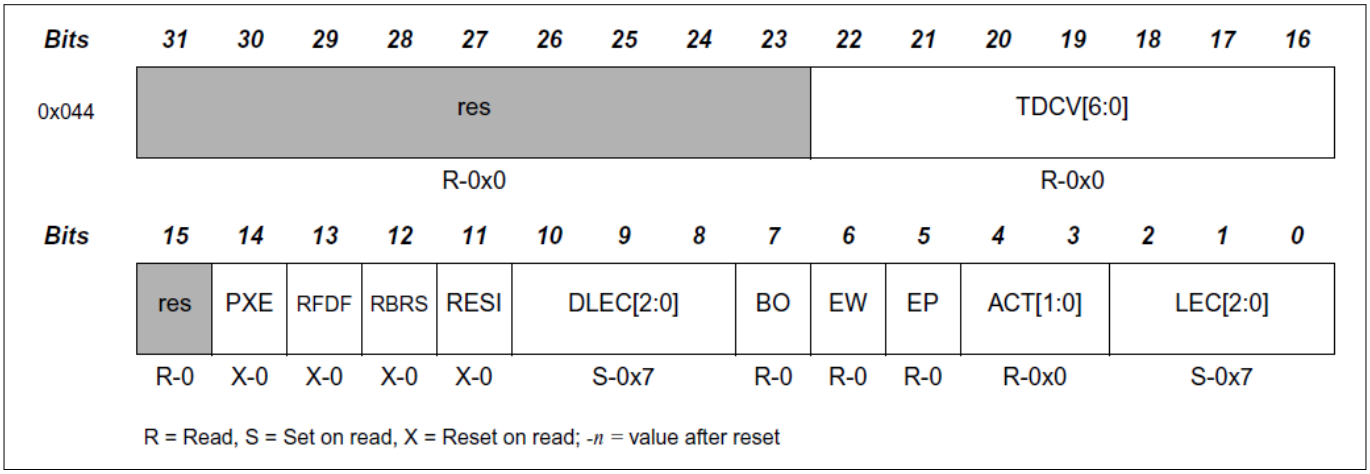


Figure 184 Protocol Status Register (address 0x044)

- **Bits 22:16 TDCV[6:0]**: Transmitter Delay Compensation Value

  Range 0x00 to 0x7F

  Defines secondary-sample-point position as the sum of the measured delay from m_ttcan_tx to m_ttcan_rx and TDCR.TDCO

  In the data phase, SSP position equals the number of mtq between the start of the transmitted bit and the secondary sample point

  Valid values are 0 to 127 mtq

- **Bit 14 PXE**: Protocol Exception Event
  - 0 No protocol-exception event occurred since last read access
  - 1 Protocol-exception event occurred

  Note: ***Byte access**: Reading byte 0 resets PXE, reading bytes 3, 2, or 1 has no impact*

- **Bit 13 RFDF**: Received a CAN FD Message

  This bit is set independent of acceptance filtering
  - 0 Since this bit was reset by the CPU, no CAN FD message has been received
  - 1 A CAN FD-format message with the FDF flag set has been received

  Note: *Byte access: reading byte 0 resets RFDF, reading bytes 3, 2, or 1 has no impact*

- **Bit 12 RBRS**: BRS Flag of Last Received CAN FD Message

  This bit is set together with RFDF, independent of acceptance filtering
  - 0 Last received CAN FD message did not have its BRS flag set
  - 1 Last received CAN FD message had its BRS flag set

  Note: ***Byte access**: Reading byte 0 resets RBRS, reading bytes 3, 2, or 1 has no impact*

- **Bit 11 RESI**: ESI Flag of Last Received CAN FD Message

  This bit is set together with RFDF, independent of acceptance filtering
  - 0 Last received CAN FD message did not have its ESI flag set
  - 1 Last received CAN FD message had its ESI flag set

  Note: ***Byte access**: Reading byte 0 resets RESI, reading bytes 3, 2, or 1 has no impact*

- **Bits 10:8 DLEC[2:0]**: Data-Phase Last Error Code

  Indicates the type of the last error that occurred in the data phase of a CAN FD-format frame with its BRS flag set

  Uses the same coding as LEC

  Clears to 0 when a CAN FD-format frame with its BRS flag set transfers without error, reception or transmission

  Note: ***Byte access:*** *Reading byte 0 sets DLEC to "111", reading bytes 3, 2, or 1 has no impact*

- **Bit 7 BO**: Bus_Off Status
  - 0M_TTCAN is not Bus_Off
  - 1 M_TTCAN is in the Bus_Off state
- **Bit 6 EW**: Warning Status
  - 0 Both error counters are below the error-warning limit of 96
  - 1 At least one error counter has reached the error-warning limit of 96
- **Bit 5 EP**: Error Passive
  - 0M_TTCAN is in the Error-Active state, participates in bus communication, and sends an active error flag when it detects an error
  - 1 M_TTCAN is in the Error-Passive state
- **Bits 4:3 ACT[1:0]**: Activity

  Monitors the module CAN communication state
  - 00 Synchronizing, node is synchronizing on CAN communication
  - 01 Idle, node is neither receiver nor transmitter
  - 10 Receiver, node is operating as receiver
  - 11 Transmitter, node is operating as transmitter
  - A protocol-exception event sets ACT to "00"
- **Bits 2:0 LEC[2:0]**: Last Error Code

  LEC indicates the type of the last error that occurred on the CAN bus

  This field clears to 0 when a message transfers without error, reception or transmission
  - 0 = No Error, no error occurred since LEC was reset by successful reception or transmission
  - 1 = Stuff Error, more than 5 equal bits occurred in sequence in a part of a received message where this is not allowed
  - 2 = Form Error, a fixed-format part of a received frame has the wrong format
  - 3 = AckError, another node did not acknowledge the message transmitted by M_TTCAN
  - 4 = Bit1Error, during message transmission except the arbitration field, the device attempted to send a recessive level, logical value 1, but the monitored bus value was dominant
  - 5 = Bit0Error, during message transmission, or during acknowledge bit, active error flag, or overload flag, the device attempted to send a dominant level, logical value 0, but the monitored bus value was recessive

    During Bus_Off recovery, this status is set each time the controller monitors a sequence of 11 recessive bits. This enables the CPU to monitor Bus_Off recovery progress and determine whether the bus is stuck at dominant or continuously disturbed
  - 6 = CRCError, CRC checksum of a received message was incorrect, received CRC did not match the CRC calculated from received data
  - 7 = NoChange, any read access to the Protocol Status Register reinitializes LEC to 7

When LEC shows 7, no CAN-bus event was detected since the last CPU read access to the Protocol Status Register

***Notes**:*

1. *When a CAN FD-format frame reaches the data phase with BRS set, the next CAN event, error or valid frame, is reported in DLEC instead of LEC An error in a fixed stuff bit of a CAN FD CRC sequence is reported as a Form Error, not a Stuff Error*
2. *Bus_Off recovery sequence per ISO 11898-1:2015 cannot be shortened by setting or resetting CCCR.INIT If the device enters Bus_Off, it sets CCCR.INIT automatically and stops all bus activity After the CPU clears CCCR.INIT, the device waits for 129 occurrences of Bus Idle, equal to 129 × 11 consecutive recessive bits, before resuming normal operation At the end of Bus_Off recovery, Error Management Counters are reset During the waiting time after the CPU clears CCCR.INIT, each monitored sequence of 11 recessive bits writes a Bit0Error code to PSR.LEC This enables the CPU to verify whether the CAN bus is stuck at dominant or continuously disturbed and to monitor Bus_Off recovery progress ECR.REC counts these sequences*
3. ***Byte access**: Reading byte 0 sets LEC to "111", reading bytes 3, 2, or 1 has no impact*

23.7.16 Transmitter Delay Compensation Register (TDCR)

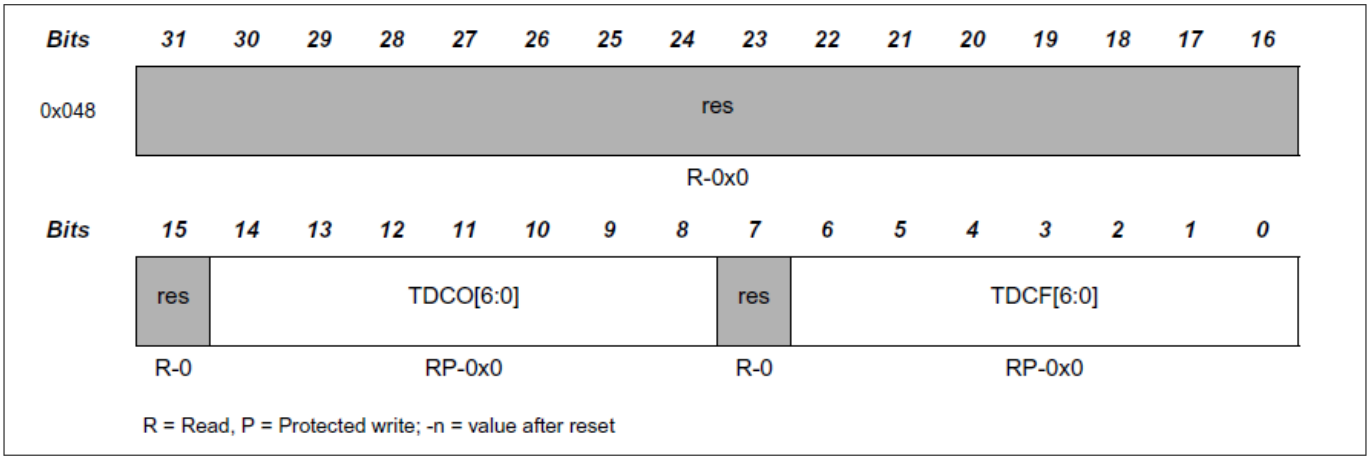


Figure 185 Transmitter Delay Compensation Register (address 0x048)

- **Bits 14:8 TDCO[6:0]**: Transmitter Delay Compensation SSP Offset

  Range 0x00 to 0x7F

  Defines offset distance between measured delay from m_ttcan_tx to m_ttcan_rx and secondary sample point

  Valid values are 0 to 127 mtq
- **Bits 6:0 TDCF[6:0]**: Transmitter Delay Compensation Filter-Window Length

  Range 0x00 to 0x7F

  Defines minimum SSP position

  Ignores dominant edges on m_ttcan_rx that would shift SSP position earlier during transmitter-delay measurement

  Enables this feature when TDCF is greater than TDCO

  Valid values are 0 to 127 mtq

23.7.17 Interrupt Register (IR)

Flags are set when the module detects any listed condition and uses edge-sensitive detection. Flags remain set until the host clears them. Clear a flag by writing `1` to the corresponding bit position. Writing `0` has no effect. A hard reset clears the register. `IE` configuration controls whether the module generates an interrupt, and `ILS` configuration selects which interrupt line signals the interrupt.

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x050	res		ARA	PED	PEA	WDI	BO	EW	EP	ELO	BEU	BEC	DRX	TOO	MRAF	TSW
	R-0		RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
Bits	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	TEFL	TEFF	TEFW	TEFN	TFE	TCF	TC	HPM	RF1L	RF1F	RF1W	RF1N	RF0L	RF0F	RF0W	RF0N
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

R = Read, W = Write; -n = value after reset

Figure 186 Interrupt Register (address 0x050)

- **Bit 29 `ARA`**: Access to Reserved Address
  - `0` = No access to a reserved address occurred
  - `1` = Access to a reserved address occurred
- **Bit 28 `PED`**: Protocol Error in Data Phase Data bit time used
  - `0` = No protocol error in the data phase
  - `1` = Protocol error in the data phase detected `PSR.DLEC != 0,7`
- **Bit 27 `PEA`**: Protocol Error in Arbitration Phase Nominal bit time used
  - `0` = No protocol error in the arbitration phase
  - `1` = Protocol error in the arbitration phase detected `PSR.LEC != 0, 7`
- **Bit 26 `WDI`**: Watchdog Interrupt
  - `0` = No Message RAM Watchdog event occurred
  - `1` = Message RAM Watchdog event occurred due to missing `READY`
- **Bit 25 `BO`**: Bus_Off Status
  - `0` = Bus_Off status unchanged
  - `1` = Bus_Off status changed
- **Bit 24 `EW`**: Warning Status
  - `0` = Error-Warning status unchanged
  - `1` = Error-Warning status changed
- **Bit 23 `EP`**: Error Passive
  - `0` = Error-Passive status unchanged
  - `1` = Error-Passive status changed
- **Bit 22 `ELO`**: Error Logging Overflow
  - `0` = CAN Error Logging Counter did not overflow
  - `1` = CAN Error Logging Counter overflow occurred
- **Bit 21 `BEU`**: Bit Error Uncorrected
  Message RAM bit error detected and not corrected

Controlled by input signal `m_ttcan_aeim_berr[1]` from optional external parity or ECC logic attached to Message RAM

An uncorrected Message RAM bit error sets `CCCR.INIT` to `1` to prevent transmission of corrupted data

- `0` = No bit error detected while reading from Message RAM
- `1` = Bit error detected and uncorrected, for example by parity logic

- **Bit 20 `BEC`**: Bit Error Corrected

  Message RAM bit error detected and corrected

  Controlled by input signal `m_ttcan_aeim_berr[0]` from optional external parity or ECC logic attached to Message RAM

  - `0` = No bit error detected while reading from Message RAM
  - `1` = Bit error detected and corrected, for example by ECC
- **Bit 19 `DRX`**: Message Stored to Dedicated Rx Buffer

  Flag is set when a received message is stored into a dedicated Rx Buffer

  - `0` = No Rx Buffer updated
  - `1` = At least one received message stored into an Rx Buffer
- **Bit 18 `TOO`**: Timeout Occurred
  - `0` = No timeout
  - `1` = Timeout reached
- **Bit 17 `MRAF`**: Message RAM Access Failure

  Flag is set when any of these conditions occur

  Rx Handler does not complete acceptance filtering or storage of an accepted message before the arbitration field of the following message is received

  Acceptance filtering or message storage is aborted, and Rx Handler starts processing the following message

  Rx Handler cannot write a message to Message RAM, and message storage is aborted

  In both cases, FIFO put index is not updated and New Data flag for a dedicated Rx Buffer is not set

  A partially stored message is overwritten when the next message is stored to the same location

  Flag is also set when Tx Handler cannot read a message from Message RAM in time, and message transmission is aborted

  A Tx Handler access failure switches `M_TTCAN` into Restricted Operation mode

  To leave Restricted Operation mode, the host CPU resets `CCCR.ASM`

  - `0` = No Message RAM access failure occurred
  - `1` = Message RAM access failure occurred
- **Bit 16 `TSW`**: Timestamp Wraparound
  - `0` = No timestamp counter wrap-around
  - `1` = Timestamp counter wrapped around
- **Bit 15 `TEFL`**: Tx Event FIFO Element Lost
  - `0` = No Tx Event FIFO element lost
  - `1` = Tx Event FIFO element lost, also set after a write attempt to a Tx Event FIFO of size 0
- **Bit 14 `TEFF`**: Tx Event FIFO Full
  - `0` = Tx Event FIFO not full
  - `1` = Tx Event FIFO full

- **Bit 13 TEFW**: Tx Event FIFO Watermark Reached
  - 0 = Tx Event FIFO fill level below watermark
  - 1 = Tx Event FIFO fill level reached watermark
- Bit 12 TEFN Tx Event FIFO New Entry
  - 0 = Tx Event FIFO unchanged
  - 1 = Tx Handler wrote a Tx Event FIFO element
- **Bit 11 TFE**: Tx FIFO Empty
  - 0 = Tx FIFO non-empty
  - 1 = Tx FIFO empty
- **Bit 10 TCF**: Transmission Cancellation Finished
  - 0 = No transmission cancellation finished
  - 1 = Transmission cancellation finished
- **Bit 9 TC**: Transmission Completed
  - 0 = No transmission completed
  - 1 = Transmission completed
- **Bit 8 HPM**: High Priority Message
  - 0 = No high-priority message received
  - 1 = High-priority message received
- **Bit 7 RF1L**: Rx FIFO 1 Message Lost
  - 0 = No Rx FIFO 1 message lost
  - 1 = Rx FIFO 1 message lost, also set after a write attempt to Rx FIFO 1 of size 0
- **Bit 6 RF1F**: Rx FIFO 1 Full
  - 0 = Rx FIFO 1 not full
  - 1 = Rx FIFO 1 full
- **Bit 5 RF1W**: Rx FIFO 1 Watermark Reached
  - 0 = Rx FIFO 1 fill level below watermark
  - 1 = Rx FIFO 1 fill level reached watermark
- **Bit 4 RF1N**: Rx FIFO 1 New Message
  - 0 = No new message written to Rx FIFO 1
  - 1 = New message written to Rx FIFO 1
- **Bit 3 RF0L**: Rx FIFO 0 Message Lost
  - 0 = No Rx FIFO 0 message lost
  - 1 = Rx FIFO 0 message lost, also set after a write attempt to Rx FIFO 0 of size 0
- **Bit 2 RF0F**: Rx FIFO 0 Full
  - 0 = Rx FIFO 0 not full
  - 1 = Rx FIFO 0 full
- **Bit 1 RF0W**: Rx FIFO 0 Watermark Reached
  - 0 = Rx FIFO 0 fill level below watermark
  - 1 = Rx FIFO 0 fill level reached watermark
- **Bit 0 RF0N**: Rx FIFO 0 New Message
  - 0 = No new message written to Rx FIFO 0
  - 1 = New message written to Rx FIFO 0

23.7.18 Interrupt Enable (IE)

Settings in the Interrupt Enable register specify which status changes in the Interrupt Register are signaled on an interrupt line.

- `0` = Interrupt disabled
- `1` = Interrupt enabled

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x054	res		ARAE	PEDE	PEAE	WDIE	BOE	EWE	EPE	ELOE	BEUE	BECE	DRXE	TOOE	MRAFE	TSWE
	R-0		RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
Bits	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	TEFLE	TEFFE	TEFWE	TEFNE	TFEE	TCFE	TCE	HPME	RF1LE	RF1FE	RF1WE	RF1NE	RF0LE	RF0FE	RF0WE	RF0NE
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

R = Read, W = Write; -n = value after reset

Figure 187 Interrupt Enable (address 0x054)

- **Bit 29 ARAE**: Access to Reserved Address Enable
- **Bit 28 PEDE**: Protocol Error in Data Phase Interrupt Enable
- **Bit 27 PEAE**: Protocol Error in Arbitration Phase Interrupt Enable
- **Bit 26 WDIE**: Watchdog Interrupt Enable
- **Bit 25 BOE**: Bus Off Status Interrupt Enable
- **Bit 24 EWE**: Warning Status Interrupt Enable
- **Bit 23 EPE**: Error Passive Interrupt Enable
- **Bit 22 ELOE**: Error Logging Overflow Interrupt Enable
- **Bit 21 BEUE**: Bit Error Uncorrected Interrupt Enable
- **Bit 20 BECE**: Bit Error Corrected Interrupt Enable
- **Bit 19 DRXE**: Message Stored to Dedicated Rx Buffer Interrupt Enable
- **Bit 18 TOOE**: Timeout Occurred Interrupt Enable
- **Bit 17 MRAFE**: Message RAM Access Failure Interrupt Enable
- **Bit 16 TSWE**: Timestamp Wraparound Interrupt Enable
- **Bit 15 TEFLE**: Tx Event FIFO Event Lost Interrupt Enable
- **Bit 14 TEFFE**: Tx Event FIFO Full Interrupt Enable
- **Bit 13 TEFWE**: Tx Event FIFO Watermark Reached Interrupt Enable
- **Bit 12 TEFNE**: Tx Event FIFO New Entry Interrupt Enable
- **Bit 11 TFEE**: Tx FIFO Empty Interrupt Enable
- **Bit 10 TCFE**: Transmission Cancellation Finished Interrupt Enable
- **Bit 9 TCE**: Transmission Completed Interrupt Enable
- **Bit 8 HPME**: High Priority Message Interrupt Enable
- **Bit 7 RF1LE**: Rx FIFO 1 Message Lost Interrupt Enable
- **Bit 6 RF1FE**: Rx FIFO 1 Full Interrupt Enable
- **Bit 5 RF1WE**: Rx FIFO 1 Watermark Reached Interrupt Enable

- **Bit 4 RF1NE**: Rx FIFO 1 New Message Interrupt Enable
- **Bit 3 RF0LE**: Rx FIFO 0 Message Lost Interrupt Enable
- **Bit 2 RF0FE**: Rx FIFO 0 Full Interrupt Enable
- **Bit 1 RF0WE**: Rx FIFO 0 Watermark Reached Interrupt Enable
- **Bit 0 RF0NE**: Rx FIFO 0 New Message Interrupt Enable

23.7.19 Interrupt Line Select (ILS)

The Interrupt Line Select register assigns an interrupt generated by a specific interrupt flag in the Interrupt Register to one of the two module interrupt lines. To generate an interrupt, enable the selected interrupt line through `ILE.EINT0` or `ILE.EINT1`.

- `0` = Interrupt assigned to interrupt line `m_ttcan_int0`
- `1` = Interrupt assigned to interrupt line `m_ttcan_int1`

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x058	res		ARAL	PEDL	PEAL	WDIL	BOL	EWL	EPL	ELOL	BEUL	BECL	DRXL	TOOL	MRAFL	TSWL
	R-0		RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
Bits	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	TEFLL	TEFFL	TEFWL	TEFNL	TFEL	TCFL	TCL	HPML	RF1LL	RF1FL	RF1WL	RF1NL	RF0LL	RF0FL	RF0WL	RF0NL
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

R = Read, W = Write; -n = value after reset

Figure 188 Interrupt Line Select (address 0x058)

- **Bit 29 ARAL**: Access to Reserved Address Line
- **Bit 28 PEDL**: Protocol Error in Data Phase Line
- **Bit 27 PEAL**: Protocol Error in Arbitration Phase Line
- **Bit 26 WDIL**: Watchdog Interrupt Line
- **Bit 25 BOL**: Bus Off Status Interrupt Line
- **Bit 24 EWL**: Warning Status Interrupt Line
- **Bit 23 EPL**: Error Passive Interrupt Line
- **Bit 22 ELOL**: Error Logging Overflow Interrupt Line
- **Bit 21 BEUL**: Bit Error Uncorrected Interrupt Line
- **Bit 20 BECL**: Bit Error Corrected Interrupt Line
- **Bit 19 DRXL**: Message Stored to Dedicated Rx Buffer Interrupt Line
- **Bit 18 TOOL**: Timeout Occurred Interrupt Line
- **Bit 17 MRAFL**: Message RAM Access Failure Interrupt Line
- **Bit 16 TSWL**: Timestamp Wraparound Interrupt Line
- **Bit 15 TEFLL**: Tx Event FIFO Event Lost Interrupt Line
- **Bit 14 TEFFL**: Tx Event FIFO Full Interrupt Line
- **Bit 13 TEFWL**: Tx Event FIFO Watermark Reached Interrupt Line
- **Bit 12 TEFNL**: Tx Event FIFO New Entry Interrupt Line

- **Bit 11 TFEL**: Tx FIFO Empty Interrupt Line
- **Bit 10 TCFL**: Transmission Cancellation Finished Interrupt Line
- **Bit 9 TCL**: Transmission Completed Interrupt Line
- **Bit 8 HPML**: High Priority Message Interrupt Line
- **Bit 7 RF1LL**: Rx FIFO 1 Message Lost Interrupt Line
- **Bit 6 RF1FL**: Rx FIFO 1 Full Interrupt Line
- **Bit 5 RF1WL**: Rx FIFO 1 Watermark Reached Interrupt Line
- **Bit 4 RF1NL**: Rx FIFO 1 New Message Interrupt Line
- **Bit 3 RF0LL**: Rx FIFO 0 Message Lost Interrupt Line
- **Bit 2 RF0FL**: Rx FIFO 0 Full Interrupt Line
- **Bit 1 RF0WL**: Rx FIFO 0 Watermark Reached Interrupt Line
- **Bit 0 RF0NL**: Rx FIFO 0 New Message Interrupt Line

23.7.20 Interrupt Line Enable (ILE)

Each of the two interrupt lines to the CPU can be enabled or disabled independently by programming bits `EINT0` and `EINT1`.

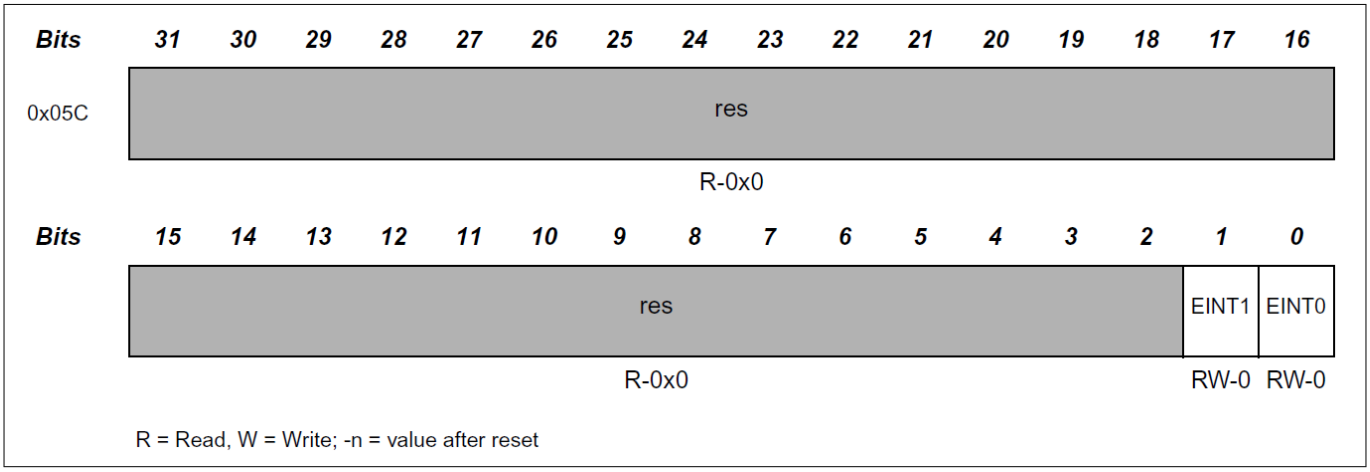


Figure 189 Interrupt Line Enable (address 0x05C)

- **Bit 1 `EINT1`**: Enable Interrupt Line 1
  - `0` Interrupt line `m_ttcan_int1` disabled
  - `1` Interrupt line `m_ttcan_int1` enabled
- **Bit 0 `EINT0`**: Enable Interrupt Line 0
  - `0` Interrupt line `m_ttcan_int0` disabled
  - `1` Interrupt line `m_ttcan_int0` enabled

23.7.21 Global Filter Configuration (GFC)

Global settings for Message ID filtering.

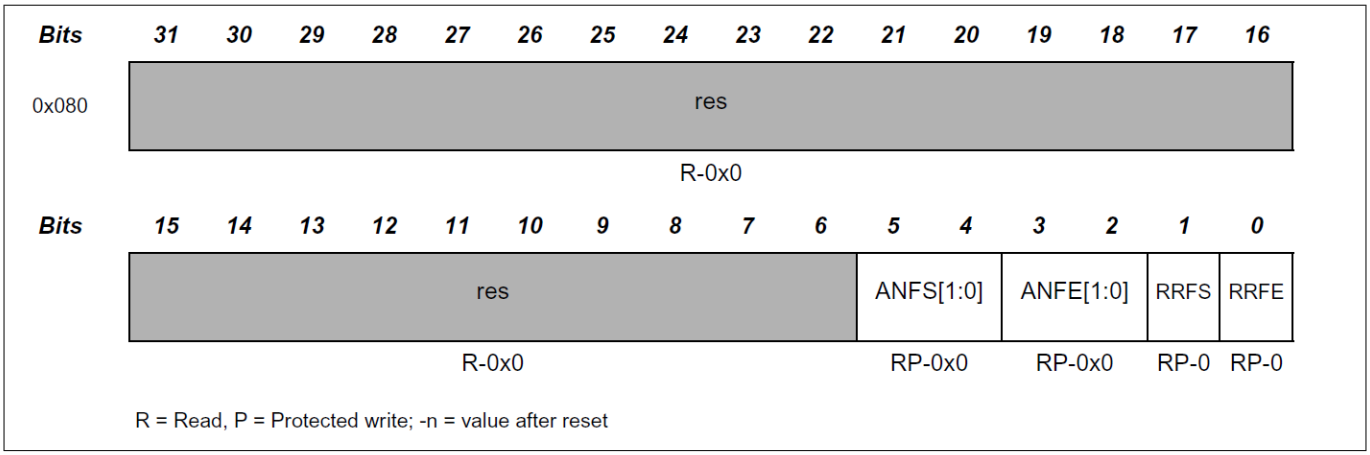


Figure 190 Global Filter Configuration (address 0x080)

- **Bits 5:4 ANFS[1:0]**: Accept Nonmatching Frames Standard
  - Defines how the module handles received messages with 11-bit IDs that do not match any filter-list element
  - 00 = Accept in Rx FIFO 0
  - 01 = Accept in Rx FIFO 1
  - 10 = Reject
  - 11 = Reject
- **Bits 3:2 ANFE[1:0]**: Accept Non-matching Frames Extended
  - Defines how the module handles received messages with 29-bit IDs that do not match any filter-list element
  - 00 = Accept in Rx FIFO 0
  - 01 = Accept in Rx FIFO 1
  - 10 = Reject
  - 11 = Reject
- **Bit 1 RRFS**: Reject Remote Frames Standard
  - 0 = Filter remote frames with 11-bit standard IDs
  - 1 = Reject all remote frames with 11-bit standard IDs
- **Bit 0 RRFE**: Reject Remote Frames Extended
  - 0 = Filter remote frames with 29-bit extended IDs
  - 1 = Reject all remote frames with 29-bit extended IDs

23.7.22 Standard ID Filter Configuration (SIDFC)

Settings for 11-bit standard Message ID filtering.

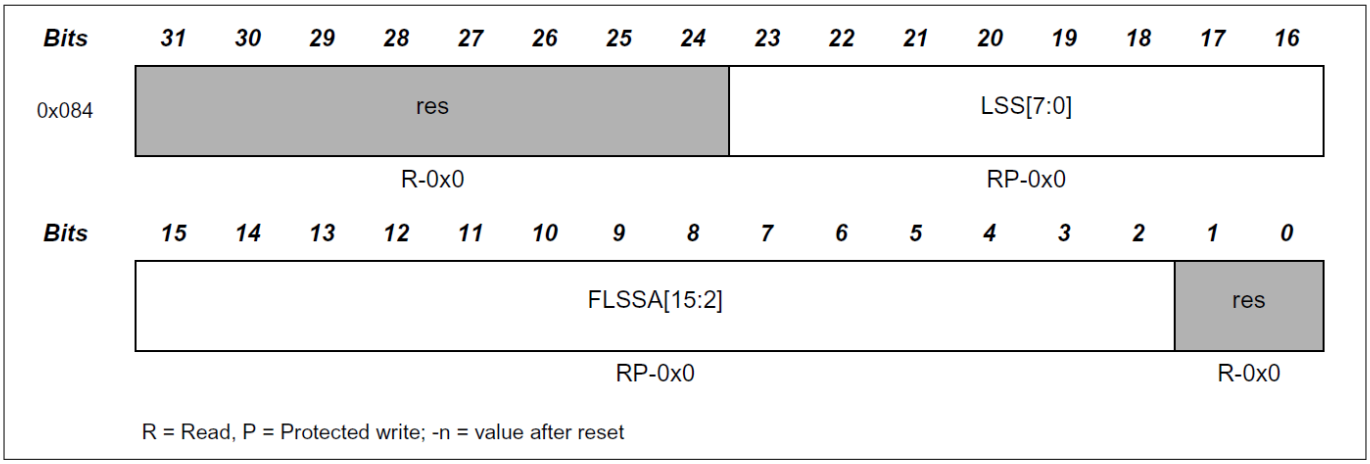


Figure 191 Standard ID Filter Configuration (address 0x084)

- **Bits 23:16 `LSS[7:0]`**: List Size Standard
  - `0` = No standard Message-ID filter
  - `1–128` = Number of standard Message-ID filter elements
  - `> 128` = Interpret values greater than `128` as `128`
- **Bits 15:2 `FLSSA[15:2]`**: Filter List Standard Start Address
  - Start address of standard Message-ID filter list

23.7.23 Extended ID Filter Configuration (XIDFC)

Settings for 29-bit extended Message-ID filtering.

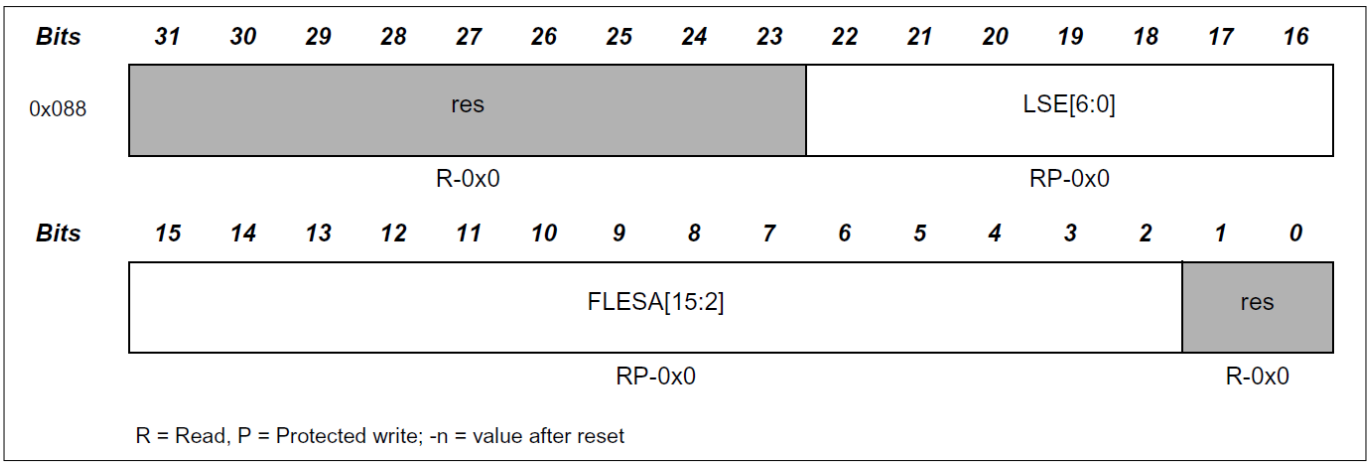


Figure 192 Extended ID Filter Configuration (address 0x088)

- **Bits 22:16 `LSE[6:0]`**: List Size Extended
  - `0` = No extended Message-ID filter
  - `1–64` = Number of extended Message-ID filter elements
  - `> 64` = Interpret values greater than `64` as `64`
- **Bits 15:2 `FLESA[15:2]`**: Filter List Extended Start Address
  - Start address of the extended Message-ID filter list

23.7.24 Extended ID AND Mask (XIDAM)

Bits 28:0 `EIDM[28:0]`: Extended-ID Mask

Sets the mask for acceptance filtering of extended frames by ANDing the Extended-ID mask with the Message ID of a received frame. Use this field to mask 29-bit IDs for SAE J1939. With the reset value set to all `1` bits, the mask is inactive.

23.7.25 High-Priority Message Status (HPMS)

This register updates each time a Message-ID filter element that is configured to generate a priority event matches. Use this register to monitor incoming high-priority message status and to provide fast access to these messages.

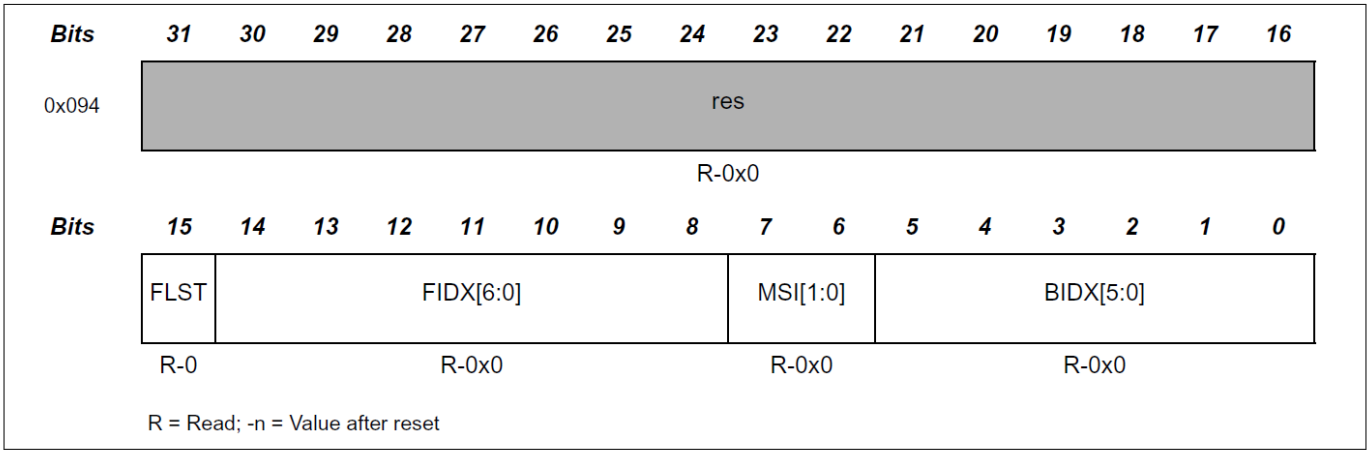


Figure 193 High-Priority Message Status (address 0x094)

- **Bit 15 `FLST`**: Filter List
  - Indicates filter list for the matching filter element
  - `0` = Standard filter list
  - `1` = Extended filter list
- **Bits 14:8 `FIDX[6:0]`**: Filter Index
  - Specifies index of the matching filter element
  - Range: `0` to `SIDFC.LSS - 1` or `XIDFC.LSE - 1`
- **Bits 7:6 `MSI[1:0]`**: Message Storage Indicator
  - `00` = No FIFO selected
  - `01` = FIFO message lost
  - `10` = Message stored in FIFO 0
  - `11` = Message stored in FIFO 1
- **Bits 5:0 `BIDX[5:0]`**: Buffer Index
  - Specifies index of the Rx FIFO element that stored the message
  - Valid only when `MSI[1]` = `1`

23.7.26 New Data 1 (NDAT1)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x098	ND31	ND30	ND29	ND28	ND27	ND26	ND25	ND24	ND23	ND22	ND21	ND20	ND19	ND18	ND17	ND16
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	ND15	ND14	ND13	ND12	ND11	ND10	ND9	ND8	ND7	ND6	ND5	ND4	ND3	ND2	ND1	ND0
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

R = Read; -n = value after reset

Figure 194 New Data 1 (address 0x098)

Bits 31:0 `ND[31:0]`: New Data

Holds the New-Data flags for Rx buffers `0` to `31`. Hardware sets a flag when the corresponding Rx buffer updates from a received frame, and the flag remains set until the host clears it. Clear a flag by writing `1` to the corresponding bit position; writing `0` has no effect. A hard reset clears the register

- `0` = Rx buffer not updated
- `1` = Rx buffer updated from a new message

23.7.27 New Data 2 (NDAT2)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x09C	ND63	ND62	ND61	ND60	ND59	ND58	ND57	ND56	ND55	ND54	ND53	ND52	ND51	ND50	ND49	ND48
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	ND47	ND46	ND45	ND44	ND43	ND42	ND41	ND40	ND39	ND38	ND37	ND36	ND35	ND34	ND33	ND32
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

R = Read; -n = value after reset

Figure 195 New Data 2 (address 0x09C)

Bits 31:0 `ND[63:32]`: New Data

Holds the New-Data flags for Rx buffers `32` to `63`. Hardware sets a flag when the corresponding Rx buffer updates from a received frame, and the flag remains set until the host clears it. Clear a flag by writing `1` to the corresponding bit position; writing `0` has no effect. A hard reset clears the register

- `0` = Rx buffer not updated
- `1` = Rx buffer updated from a new message

23.7.28 Rx FIFO 0 Configuration (RXF0C)

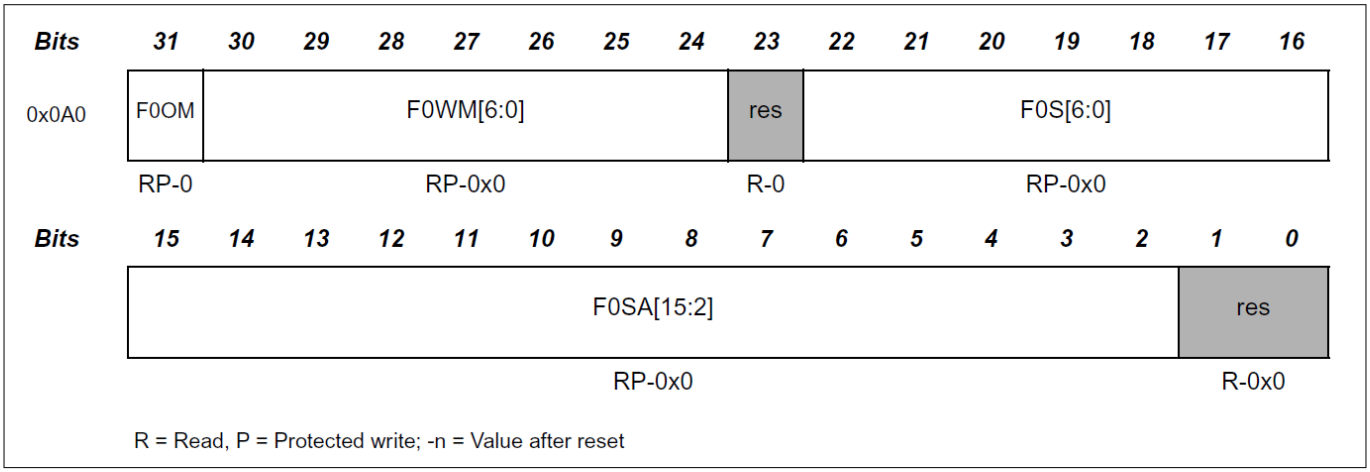


Figure 196 Rx FIFO 0 Configuration (address 0x0A0)

- **Bit 31 `F0OM`**: FIFO 0 Operation mode
  - Operate FIFO 0 in blocking mode or overwrite mode
  - `0` = FIFO 0 blocking mode
  - `1` = FIFO 0 overwrite mode
- **Bits 30:24 `F0WM[6:0]`**: Rx FIFO 0 Watermark
  - Configure level for the Rx FIFO 0 watermark interrupt `IR.RF0W`
  - `0` = Watermark interrupt disabled
  - `1–64` = Level for Rx FIFO 0 watermark interrupt `IR.RF0W`
  - `> 64` = Watermark interrupt disabled
- **Bits 22:16 `F0S[6:0]`**: Rx FIFO 0 Size
  - Set number of Rx FIFO 0 elements
  - `0` = No Rx FIFO 0
  - `1–64` = Number of Rx FIFO 0 elements
  - `> 64` = Interpret values greater than `64` as `64`
  - Index Rx FIFO 0 elements from `0` to `F0S - 1`
- **Bits 15:2 `F0SA[15:2]`**: Rx FIFO 0 Start Address
  - Set start address of Rx FIFO 0 in Message RAM

23.7.29 Rx FIFO 0 Status (RXF0S)

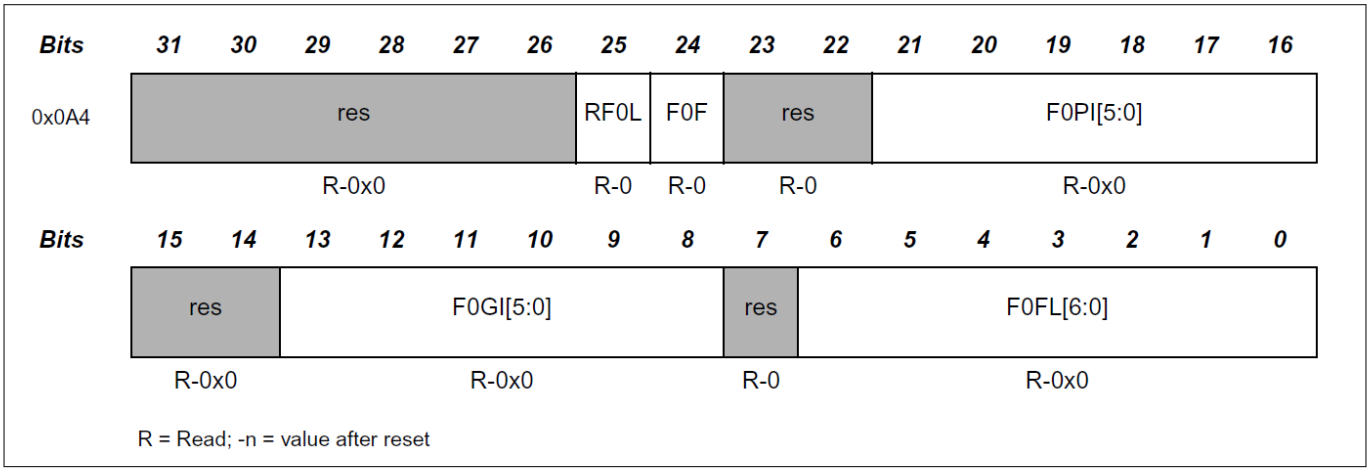


Figure 197 Rx FIFO 0 Status (address 0x0A4)

- **Bit 25 `RF0L`**: Rx FIFO 0 Message Lost
  - This bit copies interrupt flag `IR.RF0L`
  - When `IR.RF0L` resets, this bit also resets
  - `0` = No Rx FIFO 0 message lost
  - `1` = Rx FIFO 0 message lost, also set after a write attempt to Rx FIFO 0 with size `0`
  - Overwriting the oldest message when `RXF0C.F0OM = 1` does not set this flag
- **Bit 24 `F0F`**: Rx FIFO 0 Full
  - `0` = Rx FIFO 0 not full
  - `1` = Rx FIFO 0 full
- **Bits 21:16 `F0PI[5:0]`**: Rx FIFO 0 Put Index
  - Rx FIFO 0 write-index pointer, range `0` to `63`
- **Bits 13:8 `F0GI[5:0]`**: Rx FIFO 0 Get Index
  - Rx FIFO 0 read-index pointer, range `0` to `63`
- **Bits 6:0 `F0FL[6:0]`**: Rx FIFO 0 Fill Level
  - Number of elements stored in Rx FIFO 0, range `0` to `64`

23.7.30 Rx FIFO 0 Acknowledge (RXF0A)

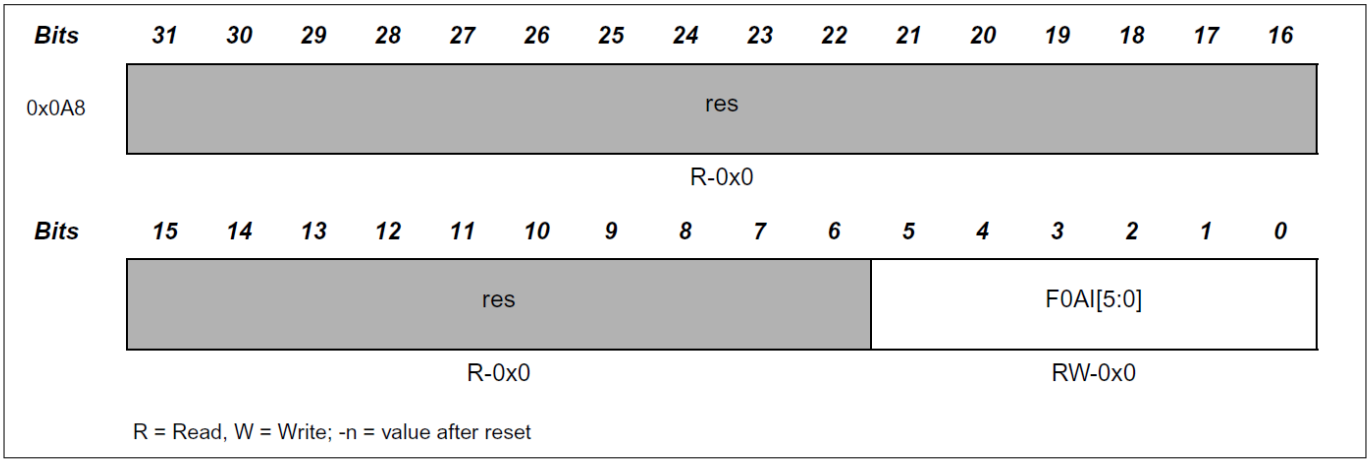


Figure 198 Rx FIFO 0 Acknowledge (address 0x0A8)

Bits 5:0 `F0AI[5:0]`: Rx FIFO 0 Acknowledge Index

Requires the host to write the buffer index of the last-read element to `F0AI` after the host reads a message or a sequence of messages from Rx FIFO 0. This write sets the Rx FIFO 0 Get Index `RXF0S.F0GI` to `F0AI + 1` and updates the FIFO 0 fill level `RXF0S.F0FL`.

23.7.31 Rx Buffer Configuration (RXBC)

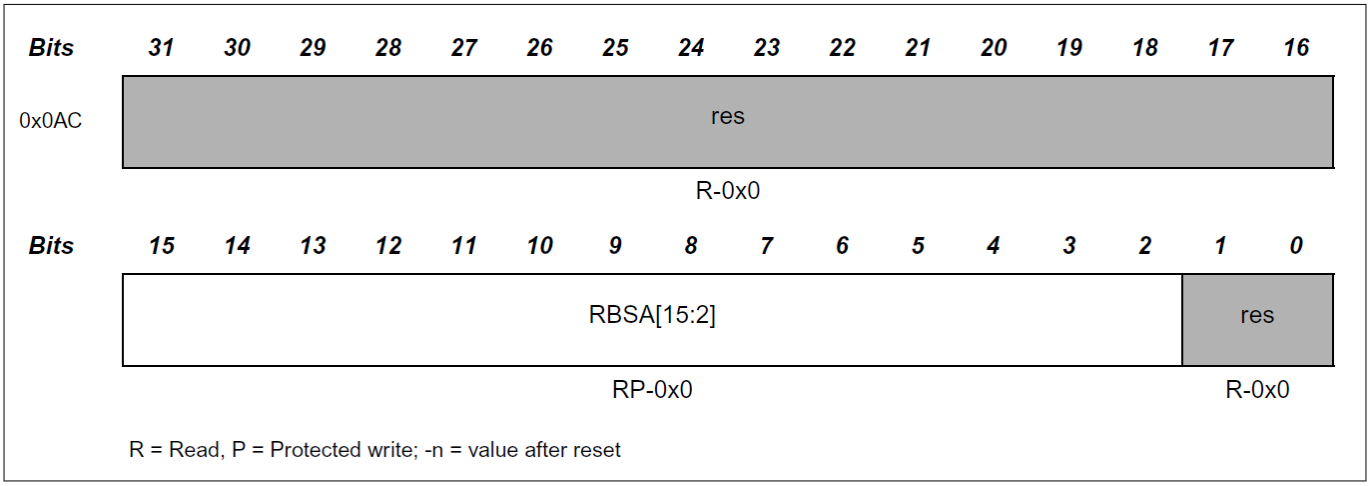


Figure 199 Rx Buffer Configuration (address 0x0AC)

Bits 15:2 `RBSA[15:2]`: Rx Buffer Start Address

Configures the start address of the Rx-buffer section in Message RAM as a 32-bit word address. Use this field to reference debug messages A, B, and C.

23.7.32 Rx FIFO 1 Configuration (RXF1C)

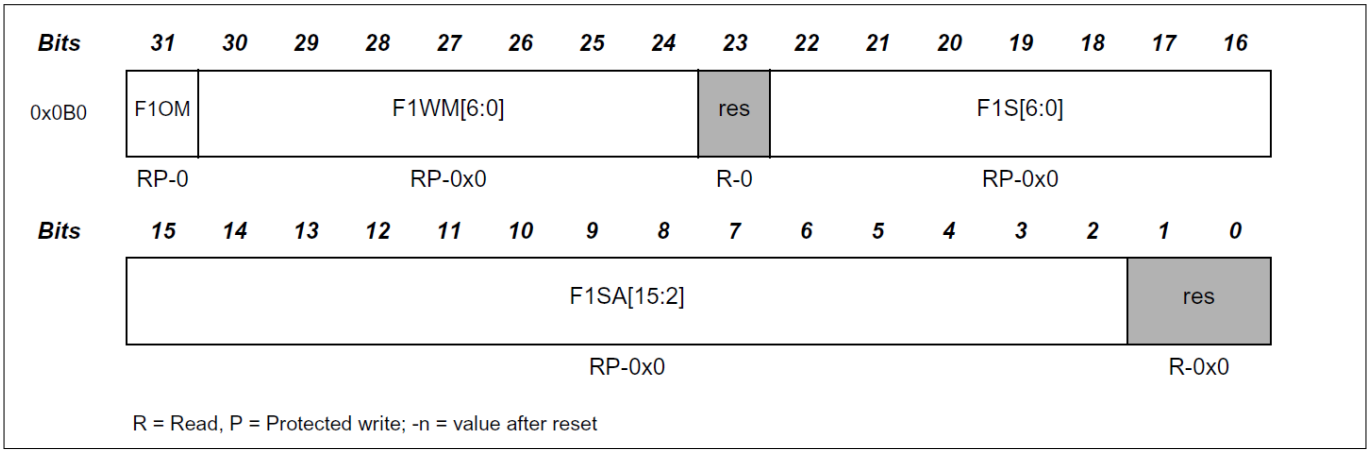


Figure 200 Rx FIFO 1 Configuration (address 0x0B0)

- **Bit 31** `F1OM`: FIFO 1 Operation mode
  - Operate FIFO 1 in blocking mode or overwrite mode
  - `0` = FIFO 1 blocking mode
  - `1` = FIFO 1 overwrite mode
- **Bits 30:24** `F1WM[6:0]`: Rx FIFO 1 Watermark
  - Configure level for the Rx FIFO 1 watermark interrupt `IR.RF1W`
  - `0` = Watermark interrupt disabled
  - `1–64` = Level for Rx FIFO 1 watermark interrupt `IR.RF1W`
  - `> 64` = Watermark interrupt disabled
- **Bits 22:16** `F1S[6:0]`: Rx FIFO 1 Size
  - Set number of Rx FIFO 1 elements
  - `0` = No Rx FIFO 1
  - `1–64` = Number of Rx FIFO 1 elements
  - `> 64` = Interpret values greater than `64` as `64`
  - Index Rx FIFO 1 elements from `0` to `F1S - 1`
- **Bits 15:2** `F1SA[15:2]`: Rx FIFO 1 Start Address
  - Set start address of Rx FIFO 1 in Message RAM

23.7.33 Rx FIFO 1 Status (RXF1S)

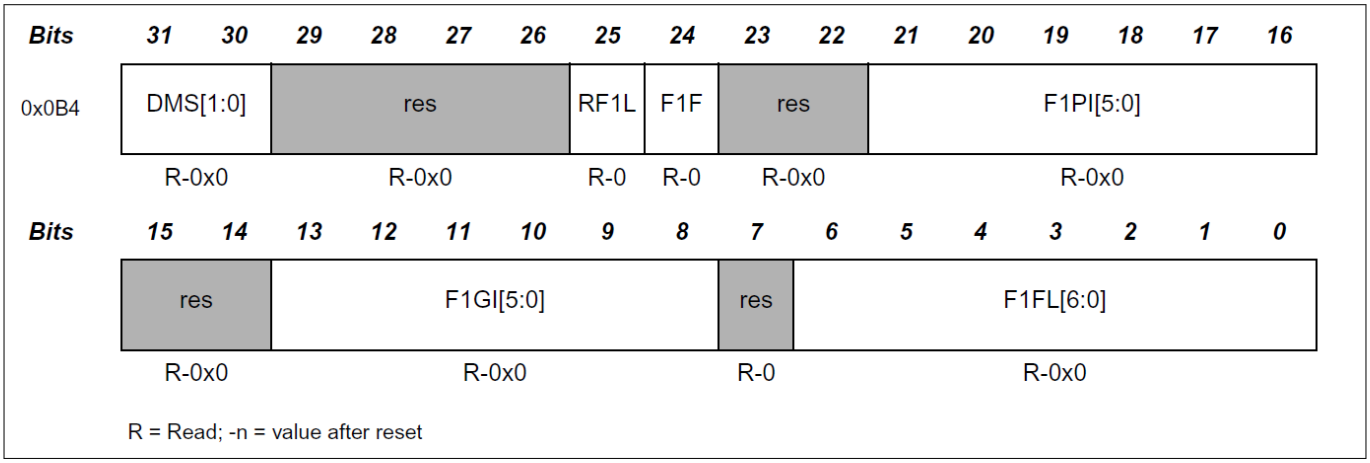


Figure 201 Rx FIFO 1 Status (address 0x0B4)

- **Bits 31:30** `DMS[1:0]`: Debug Message Status
  - `00` = Idle state, waits to receive debug messages, clears DMA request
  - `01` = Debug message A received
  - `10` = Debug messages A and B received
  - `11` = Debug messages A, B, and C received, sets DMA request
- **Bit 25** `RF1L`: Rx FIFO 1 Message Lost
  - Copies interrupt flag `IR.RF1L`
  - Resets when `IR.RF1L` resets
  - `0` = No Rx FIFO 1 message lost
  - `1` = Rx FIFO 1 message lost, also sets after a write attempt to Rx FIFO 1 with size `0`
  - Overwriting the oldest message when `RXF1C.F1OM` = `1` does not set this flag
- **Bit 24** `F1F`: Rx FIFO 1 Full
  - `0` = Rx FIFO 1 not full
  - `1` = Rx FIFO 1 full
- **Bits 21:16** `F1PI[5:0]`: Rx FIFO 1 Put Index
  - Specifies Rx FIFO 1 write-index pointer, range `0` to `63`
- **Bits 13:8** `F1GI[5:0]`: Rx FIFO 1 Get Index
  - Specifies Rx FIFO 1 read-index pointer, range `0` to `63`
- **Bits 6:0** `F1FL[6:0]`: Rx FIFO 1 Fill Level
  - Specifies number of elements stored in Rx FIFO 1, range `0` to `64`

23.7.34 Rx FIFO 1 Acknowledge (RXF1A)

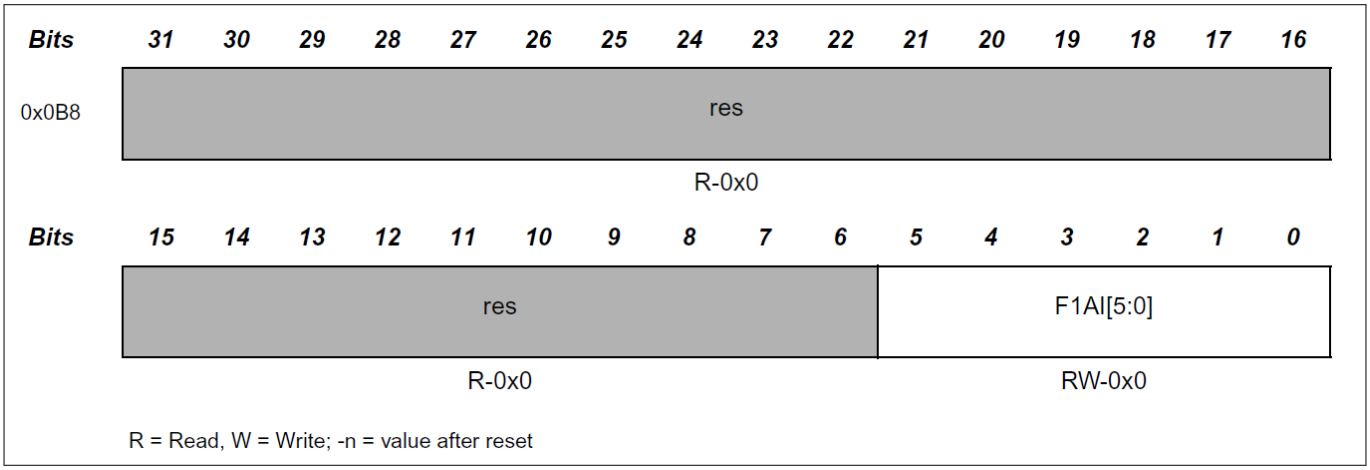


Figure 202 Rx FIFO 1 Acknowledge (address 0x0B8)

Bits 5:0 `F1AI[5:0]`: Rx FIFO 1 Acknowledge Index

Requires the host to write the buffer index of the last-read element to `F1AI` after the host reads a message or a sequence of messages from Rx FIFO 1. This write sets the Rx FIFO 1 Get Index `RXF1S.F1GI` to `F1AI + 1` and updates the FIFO 1 fill level `RXF1S.F1FL`.

23.7.35 Rx Buffer/FIFO Element Size Configuration (RXESC)

Configures number of data bytes for an Rx-buffer or Rx-FIFO element. Use data-field sizes greater than 8 bytes only for CAN FD operation.

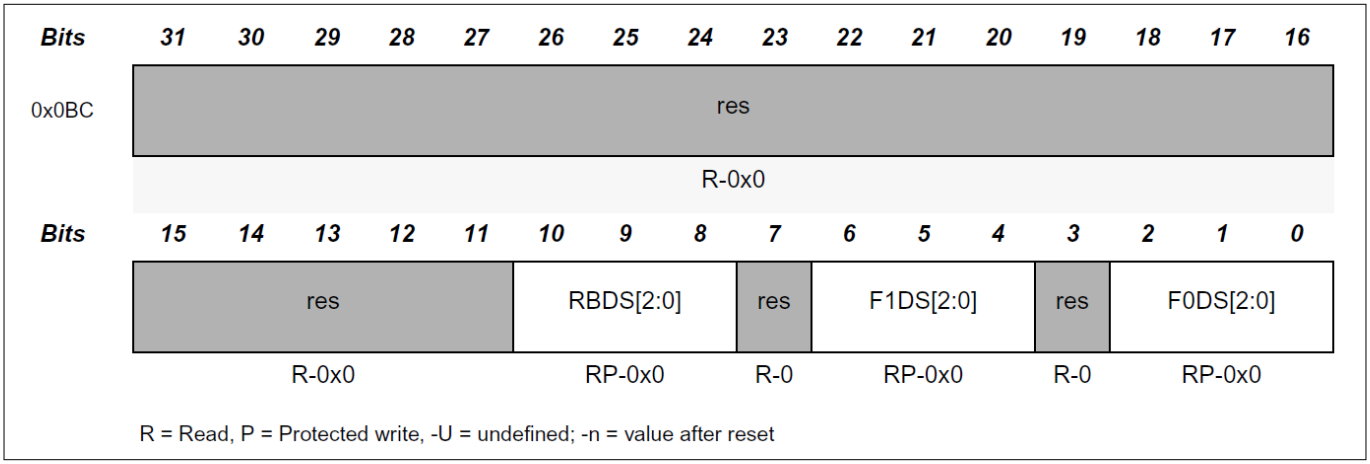


Figure 203 Rx Buffer/FIFO Element Size Configuration (address 0x0BC)

- **Bits 10:8** `RBDS[2:0]`: Rx Buffer Data-Field Size
  - `000` = 8-byte data field
  - `001` = 12-byte data field
  - `010` = 16-byte data field
  - `011` = 20-byte data field
  - `100` = 24-byte data field
  - `101` = 32-byte data field

- 110 = 48-byte data field
- 111 = 64-byte data field

- **Bits 6:4 F1DS[2:0]**: Rx FIFO 1 Data-Field Size
  - 000 = 8-byte data field
  - 001 = 12-byte data field
  - 010 = 16-byte data field
  - 011 = 20-byte data field
  - 100 = 24-byte data field
  - 101 = 32-byte data field
  - 110 = 48-byte data field
  - 111 = 64-byte data field
- **Bits 2:0 F0DS[2:0]**: Rx FIFO 0 Data-Field Size
  - 000 = 8-byte data field
  - 001 = 12-byte data field
  - 010 = 16-byte data field
  - 011 = 20-byte data field
  - 100 = 24-byte data field
  - 101 = 32-byte data field
  - 110 = 48-byte data field
  - 111 = 64-byte data field

***Note**: If an accepted CAN frame has a data-field size that exceeds the data-field size configured for the matching Rx buffer or Rx FIFO, the controller stores only the number of bytes configured by RXESC in the corresponding Rx buffer or Rx-FIFO element. The controller ignores remaining bytes in the frame data field.*

23.7.36 Tx Buffer Configuration (TXBC)

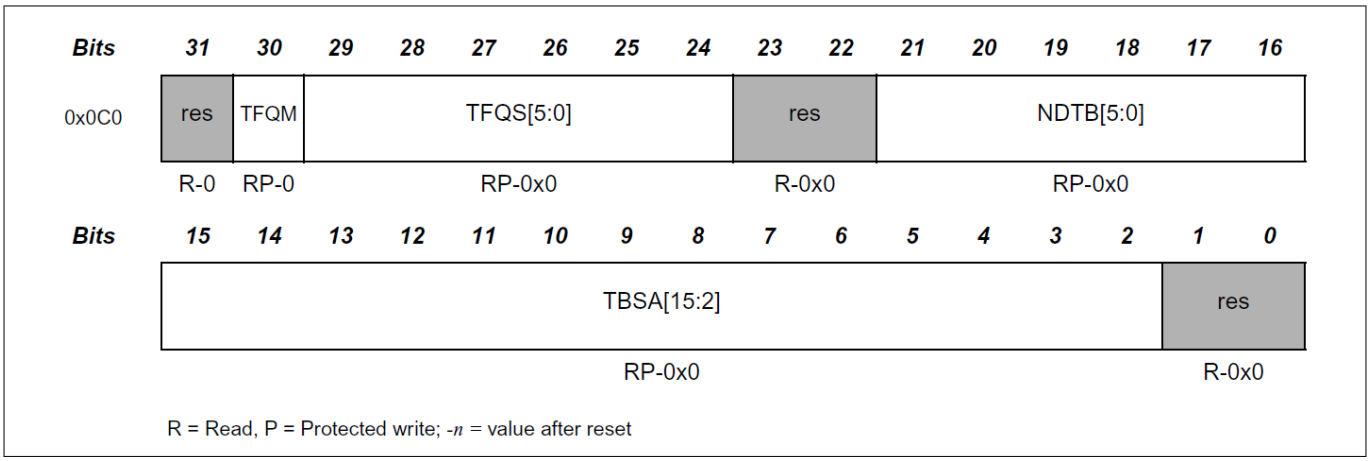


Figure 204 Tx Buffer Configuration (address 0x0C0)

- **Bit 30 TFQM**: Tx FIFO-or-Queue mode
  - Select Tx FIFO operation or Tx queue operation
  - 0 = Tx FIFO operation
  - 1 = Tx queue operation

- **Bits 29:24 TFQS[5:0]**: Transmit FIFO-or-Queue Size
  - Configure number of Tx buffers used for the Tx FIFO or Tx queue
  - 0 = No Tx FIFO or Tx queue
  - 1–32 = Number of Tx buffers used for the Tx FIFO or Tx queue
  - > 32 = Interpret values greater than 32 as 32
- **Bits 21:16 NDTB[5:0]**: Number of Dedicated Transmit Buffers
  - Configure number of dedicated Tx buffers
  - 0 = No dedicated Tx buffers
  - 1–32 = Number of dedicated Tx buffers
  - > 32 = Interpret values greater than 32 as 32
- **Bits 15:2 TBSA[15:2]**: Tx Buffers Start Address
  - Set start address of the Tx-buffers section in Message RAM

***Note**: Ensure TFQS + NDTB does not exceed 32. Hardware does not validate invalid configurations. The Tx-buffer section in Message RAM starts with dedicated Tx buffers.*

23.7.37 Tx FIFO/Queue Status (TXFQS)

The Tx FIFO/Queue Status depends on pending Tx requests listed in TXBRP. Add or cancellation requests can take effect later if a Tx scan is in progress because TXBRP does not update until the scan completes.

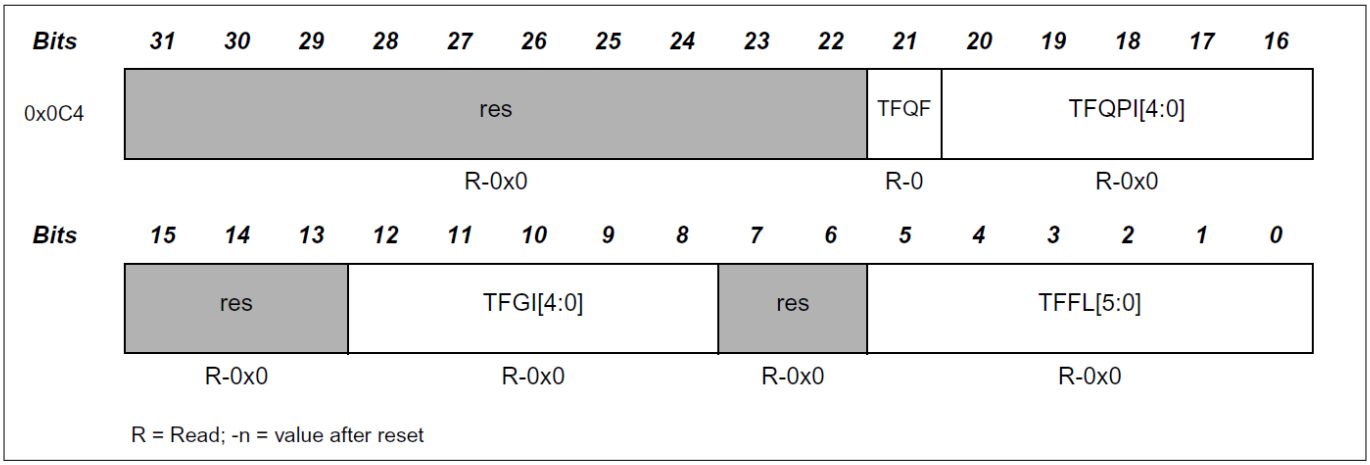


Figure 205 Tx FIFO/Queue Status (address 0x0C4)

- **Bit 21 TFQF**: Tx FIFO/Queue Full
  - 0 = Tx FIFO/queue not full
  - 1 = Tx FIFO/queue full
- **Bits 20:16 TFQPI[4:0]**: Tx FIFO/Queue Put Index
  - Specifies Tx FIFO/queue write-index pointer, range 0 to 31
- **Bits 12:8 TFGI[4:0]**: Tx FIFO Get Index
  - Specifies Tx FIFO read-index pointer, range 0 to 31
  - Reads as 0 when Tx queue operation is configured (TXBC.TFQM = 1)
- **Bits 5:0 TFFL[5:0]**: Tx FIFO Free Level
  - Specifies number of consecutive free Tx FIFO elements starting from TFGI, range 0 to 32
  - Reads as 0 when Tx queue operation is configured (TXBC.TFQM = 1)

***Note**: In mixed-configuration setups that combine dedicated Tx Buffers with a Tx FIFO or a Tx Queue, the `Put Index` and `Get Index` specify the Tx Buffer number, starting from the first dedicated Tx Buffers. For example, with 12 dedicated Tx Buffers and a Tx FIFO of 20 buffers, a `Put Index` of 15 points to the fourth buffer in the Tx FIFO.*

23.7.38 Tx Buffer Element Size Configuration (TXESC)

Configures the number of data bytes in a Tx Buffer element. Use data-field sizes greater than 8 bytes only for CAN FD operation.

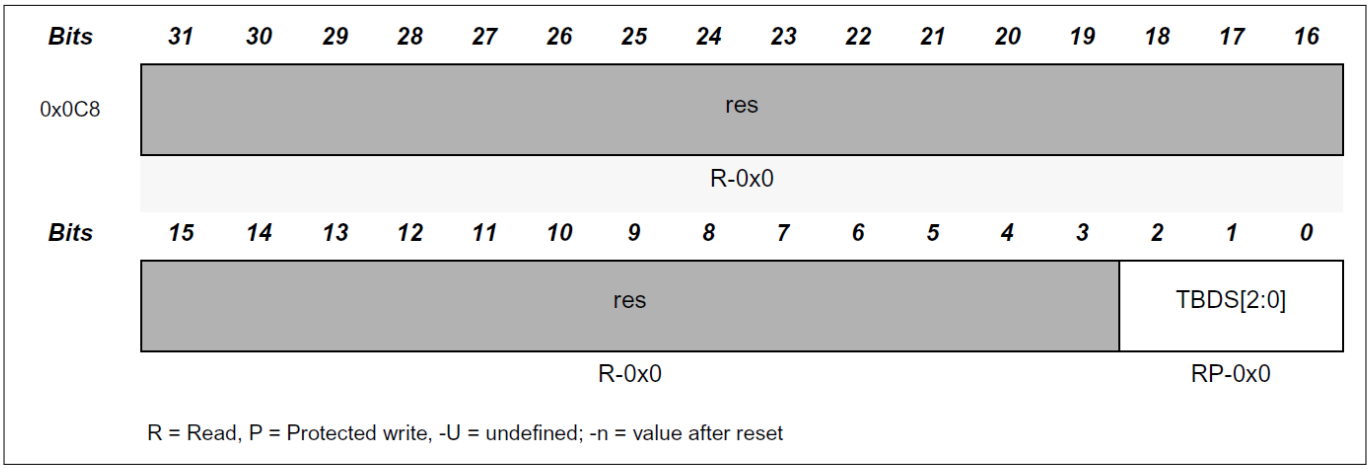


Figure 206 Tx Buffer Element Size Configuration (address 0x0C8)

Bits 2:0 `TBDS[2:0]`: Tx Buffer Data Field Size

- `000` = 8-byte data field
- `001` = 12-byte data field
- `010` = 16-byte data field
- `011` = 20-byte data field
- `100` = 24-byte data field
- `101` = 32-byte data field
- `110` = 48-byte data field
- `111` = 64-byte data field

***Note**: If the data length code (DLC) in a Tx Buffer element is set to a value greater than the Tx Buffer data-field size in `TXESC.TBDS`, the controller transmits any undefined bytes as `0xCC` padding bytes.*

23.7.39 Tx Buffer Request Pending (TXBRP)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x0CC	TRP31	TRP30	TRP29	TRP28	TRP27	TRP26	TRP25	TRP24	TRP23	TRP22	TRP21	TRP20	TRP19	TRP18	TRP17	TRP16
	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
Bits	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	TRP15	TRP14	TRP13	TRP12	TRP11	TRP10	TRP9	TRP8	TRP7	TRP6	TRP5	TRP4	TRP3	TRP2	TRP1	TRP0
	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0

R = Read; -n = value after reset

Figure 207 Tx Buffer Request Pending (address 0x0CC)

Bit 31:0 `TRP[31:0]`: Transmission Request Pending

Each Tx Buffer has its own Transmission Request Pending bit. Set these bits through register TXBAR. Reset these bits after the requested transmission completes or when you cancel the request through register TXBCR.

Set TXBRP bits only for Tx Buffers configured through TXBC. After you set a TXBRP bit, the controller starts a Tx scan to find the pending Tx request with the highest priority, which is the Tx Buffer with the lowest Message ID.

A cancellation request resets the corresponding Transmission Request Pending bit in register TXBRP. If transmission has already started when you request cancellation, the controller resets the bit at the end of the transmission, regardless of whether the transmission succeeds. The controller resets the cancellation request bits immediately after it resets the corresponding TXBRP bit.

After you request cancellation, the controller signals a completed cancellation through TXBCF:

- After successful transmission, together with the corresponding TXBTO bit
- When transmission has not started at the time of cancellation
- When transmission aborts due to lost arbitration
- When an error occurs during frame transmission

In DAR mode, the controller automatically cancels any transmission that does not succeed. The controller sets the corresponding TXBCF bit for each unsuccessful transmission.

- 0 = No transmission request pending
- 1 = Transmission request pending

***Note**: TXBRP bits set while a Tx scan is in progress are not considered during that Tx scan. If you request cancellation for such a Tx Buffer, the controller cancels the `Add Request` immediately and resets the corresponding TXBRP bit.*

23.7.40 Tx Buffer Add Request (TXBAR)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x0D0	AR31	AR30	AR29	AR28	AR27	AR26	AR25	AR24	AR23	AR22	AR21	AR20	AR19	AR18	AR17	AR16
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	AR15	AR14	AR13	AR12	AR11	AR10	AR9	AR8	AR7	AR6	AR5	AR4	AR3	AR2	AR1	AR0
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

R = Read, W = Write; -n = value after reset

Figure 208 Tx Buffer Add Request (address 0x0D0)

Bit 31:0 `AR[31:0]`: Add Request

Each Tx Buffer has its own Add Request bit. Writing `1` sets the corresponding Add Request bit. Writing `0` has no effect. This allows the host to request transmission for multiple Tx Buffers with a single write to TXBAR. TXBAR bits are set only for Tx Buffers configured through TXBC. If no Tx scan is running, the controller resets the bits immediately. If a Tx scan is running, the bits remain set until the Tx scan completes

- 0 = No transmission request added
- 1 = Transmission request added

***Note**: If you apply an Add Request to a Tx Buffer that already has a pending transmission request (the corresponding TXBRP bit is already set), the controller ignores the Add Request.*

23.7.41 Tx Buffer Cancellation Request (TXBCR)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x0D4	CR31	CR30	CR29	CR28	CR27	CR26	CR25	CR24	CR23	CR22	CR21	CR20	CR19	CR18	CR17	CR16
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CR15	CR14	CR13	CR12	CR11	CR10	CR9	CR8	CR7	CR6	CR5	CR4	CR3	CR2	CR1	CR0
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

R = Read, W = Write; -n = value after reset

Figure 209 Tx Buffer Cancellation Request (address 0x0D4)

Bit 31:0 `CR[31:0]`: Cancellation Request

Each Tx Buffer has its own Cancellation Request bit. Writing `1` sets the corresponding Cancellation Request bit. Writing `0` has no effect. This allows the host to request cancellation for multiple Tx Buffers with a single write to TXBCR. TXBCR bits are set only for Tx Buffers configured through TXBC. The bits remain set until the corresponding TXBRP bit resets

- 0 = No cancellation pending
- 1 = Cancellation pending

23.7.42 Tx Buffer Transmission Occurred (TXBTO)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x0D8	TO31	TO30	TO29	TO28	TO27	TO26	TO25	TO24	TO23	TO22	TO21	TO20	TO19	TO18	TO17	TO16
	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TO15	TO14	TO13	TO12	TO11	TO10	TO9	TO8	TO7	TO6	TO5	TO4	TO3	TO2	TO1	TO0
	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0

R = Read; -n = value after reset

Figure 210 Tx Buffer Transmission Occurred (address 0x0D8)

Bit 31:0 `TO[31:0]`: Transmission Occurred

Each Tx Buffer has its own Transmission Occurred bit. The controller sets these bits when it clears the corresponding TXBRP bit after a successful transmission. The controller resets these bits when you request a new transmission by writing 1 to the corresponding bit in register TXBAR

- 0 = No transmission occurred
- 1 = Transmission occurred

23.7.43 Tx Buffer Cancellation Finished (TXBCF)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x0DC	CF31	CF30	CF29	CF28	CF27	CF26	CF25	CF24	CF23	CF22	CF21	CF20	CF19	CF18	CF17	CF16
	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CF15	CF14	CF13	CF12	CF11	CF10	CF9	CF8	CF7	CF6	CF5	CF4	CF3	CF2	CF1	CF0
	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0

R = Read; -n = value after reset

Figure 211 Transmit Buffer Cancellation Finished (address 0x0DC)

Bit 31:0 `CF[31:0]`: Cancellation Finished

Each Tx Buffer has its own Cancellation Finished bit. The controller sets these bits when it clears the corresponding TXBRP bit after you request cancellation through TXBCR. If the corresponding TXBRP bit is not set at the time of cancellation, the controller sets CF immediately. The controller resets these bits when you request a new transmission by writing 1 to the corresponding bit in register TXBAR

- 0 = No Tx Buffer cancellation
- 1 = Tx Buffer cancellation finished

23.7.44 Tx Buffer Transmission Interrupt Enable (TXBTIE)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x0E0	TIE31	TIE30	TIE29	TIE28	TIE27	TIE26	TIE25	TIE24	TIE23	TIE22	TIE21	TIE20	TIE19	TIE18	TIE17	TIE16
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TIE15	TIE14	TIE13	TIE12	TIE11	TIE10	TIE9	TIE8	TIE7	TIE6	TIE5	TIE4	TIE3	TIE2	TIE1	TIE0
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

R = Read, W = Write; -n = value after reset

Figure 212 Tx Buffer Transmission Interrupt Enable (address 0x0E0)

Bit 31:0 `TIE[31:0]`: Transmission Interrupt Enable

Each Tx Buffer has its own Transmission Interrupt Enable bit

- 0 = Transmission interrupt disabled
- 1 = Transmission interrupt enabled

23.7.45 Tx Buffer Cancellation Finished Interrupt Enable (TXBCIE)

Bits	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0x0E4	CFIE31	CFIE30	CFIE29	CFIE28	CFIE27	CFIE26	CFIE25	CFIE24	CFIE23	CFIE22	CFIE21	CFIE20	CFIE19	CFIE18	CFIE17	CFIE16
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0
Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	CFIE15	CFIE14	CFIE13	CFIE12	CFIE11	CFIE10	CFIE9	CFIE8	CFIE7	CFIE6	CFIE5	CFIE4	CFIE3	CFIE2	CFIE1	CFIE0
	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0	RW-0

R = Read, W = Write; -n = value after reset

Figure 213 Tx Buffer Cancellation Finished Interrupt Enable (address 0x0E4)

Bit 31:0 `CFIE[31:0]`: Cancellation Finished Interrupt Enable

Each Tx Buffer has its own Cancellation Finished Interrupt Enable bit

- 0 = Cancellation finished interrupt disabled
- 1 = Cancellation finished interrupt enabled

23.7.46 Tx Event FIFO Configuration (TXEFC)

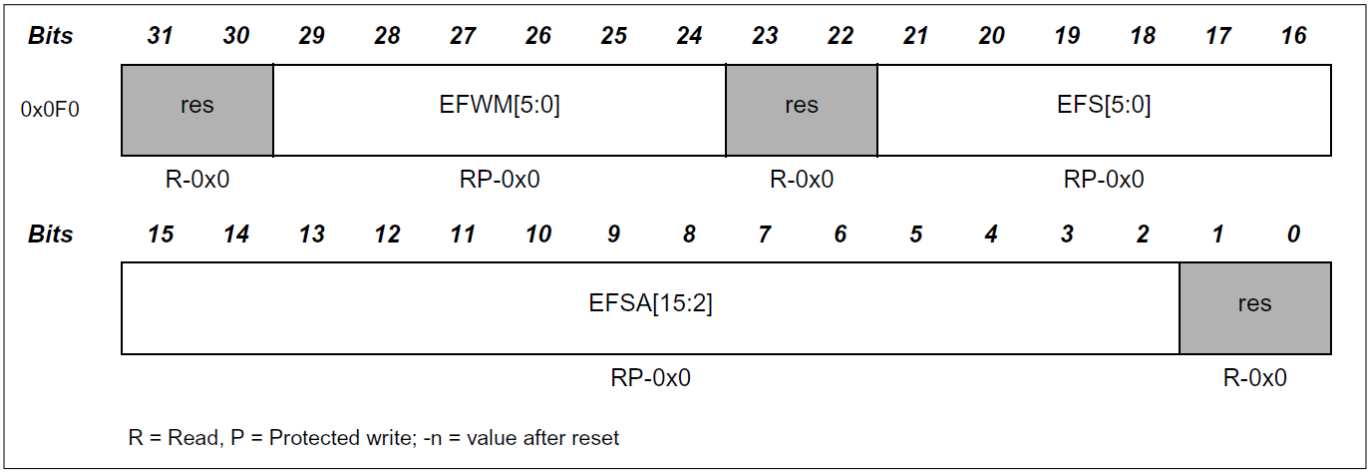


Figure 214 Tx Event FIFO Configuration (address 0x0F0)

- **Bit 29:24 `EFWM[5:0]`**: Event FIFO Watermark
  - Sets Tx Event FIFO watermark level for interrupt `IR.TEFW`
  - 0 = Watermark interrupt disabled
  - 1 to 32 = Level for Tx Event FIFO watermark interrupt `IR.TEFW`
  - Greater than 32 = Watermark interrupt disabled
- **Bit 21:16 `EFS[5:0]`**: Event FIFO Size
  - Configures Tx Event FIFO size
  - 0 = Tx Event FIFO disabled
  - 1 to 32 = Number of Tx Event FIFO elements
  - Greater than 32 = Values greater than 32 are interpreted as 32
  - Tx Event FIFO elements are indexed from 0 to `EFS - 1`
- **Bit 15:2 `EFSA[15:2]`**: Event FIFO Start Address
  - Sets start address of Tx Event FIFO in Message RAM

23.7.47 Tx Event FIFO Status (TXEFS)

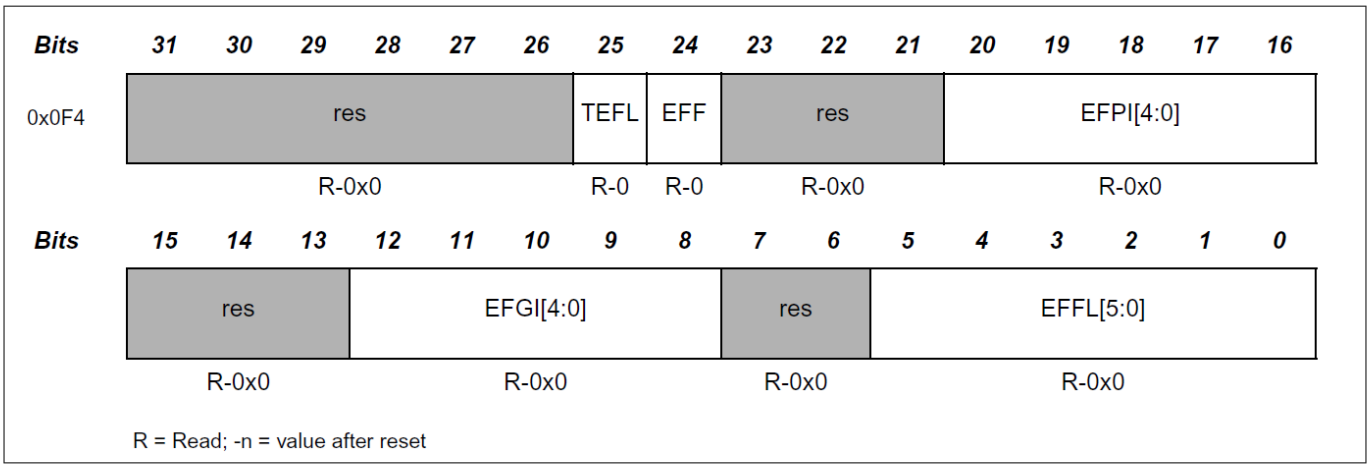


Figure 215 Tx Event FIFO Status (address 0x0F4)

- **Bit 25 TEFL**: Tx Event FIFO Element Lost
  - Mirrors interrupt flag IR.TEFL
  - When IR.TEFL resets, this bit also resets
  - 0 = No Tx Event FIFO element lost
  - 1 = Tx Event FIFO element lost, also set after a write attempt to a Tx Event FIFO with size 0
- **Bit 24 EFF**: Event FIFO Full
  - 0 = Tx Event FIFO not full
  - 1 = Tx Event FIFO full
- **Bit 20:16 EFPI[4:0]**: Event FIFO Put Index
  - Tx Event FIFO write index pointer, range 0 to 31
- **Bit 12:8 EFGI[4:0]**: Event FIFO Get Index
  - Tx Event FIFO read index pointer, range 0 to 31
- **Bit 5:0 EFFL[5:0]**: Event FIFO Fill Level
  - Number of elements stored in Tx Event FIFO, range 0 to 32

23.7.48 Tx Event FIFO Acknowledge (TXEFA)

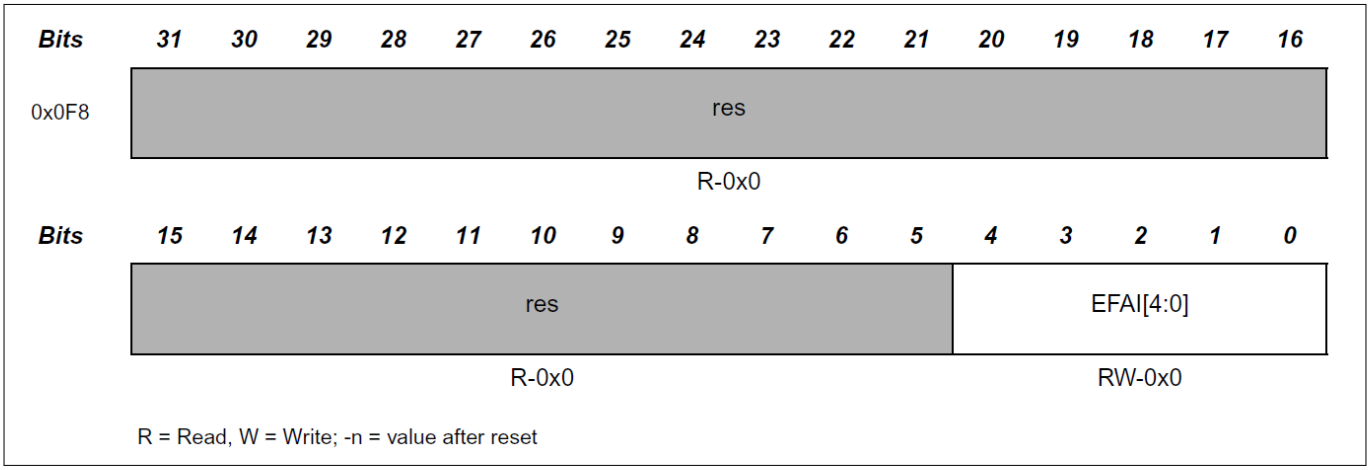


Figure 216 Tx Event FIFO Acknowledge (address 0x0F8)

Bit 4:0 EFAI[4:0]: Event FIFO Acknowledge Index

After the host reads one element or a sequence of elements from the Tx Event FIFO, it must write the index of the last element read to EFAI. This write sets the Tx Event FIFO Get Index TXEFS.EFGI to EFAI + 1 and updates the Event FIFO Fill Level TXEFS.EFFL.

23.7.49 TT Trigger Memory Configuration (TTTMC)

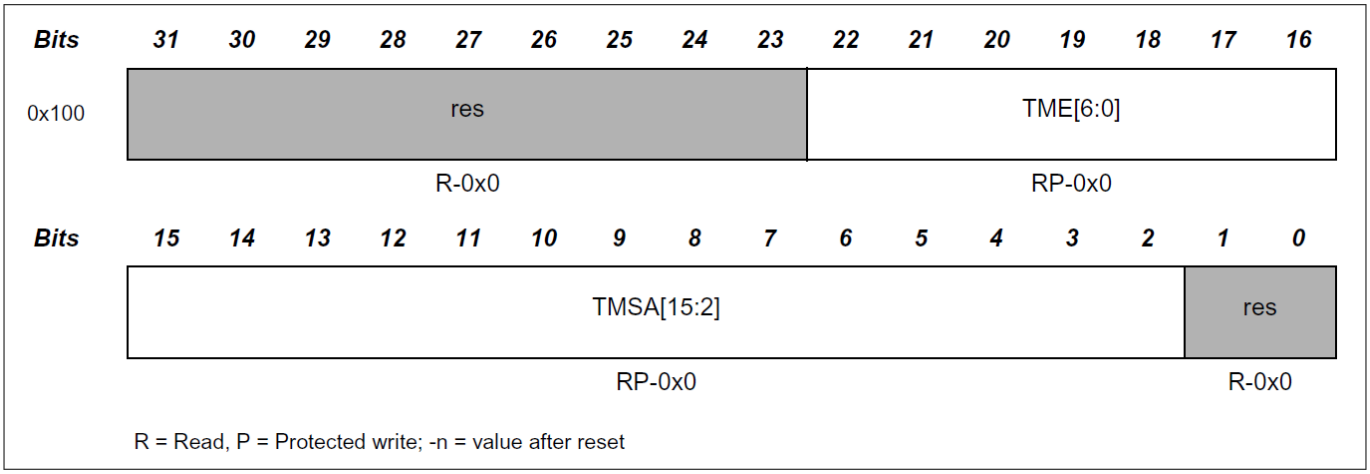


Figure 217 TT Trigger Memory Configuration (address 0x100)

- **Bit 22:16 `TME[6:0]`**: Trigger Memory Elements
  - 0 = No Trigger Memory
  - 1 to 64 = Number of Trigger Memory elements
  - Greater than 64 = Values greater than 64 are interpreted as 64
- **Bit 15:2 `TMSA[15:2]`**: Trigger Memory Start Address
  - Start address of Trigger Memory in Message RAM

23.7.50 TT Reference Message Configuration (TTRMC)

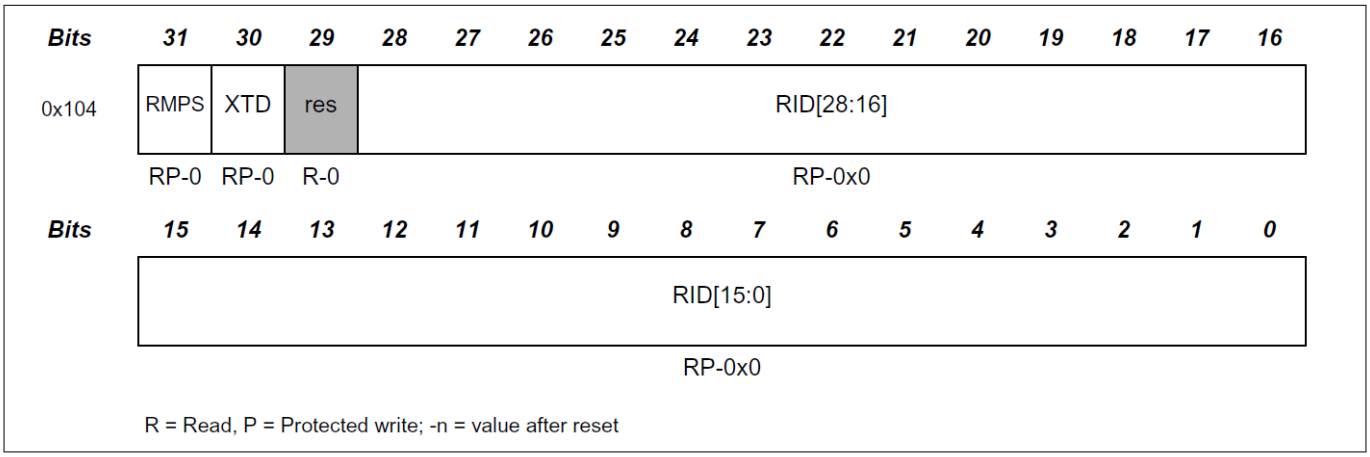


Figure 218 TT Reference Message Configuration (address 0x104)

- **Bit 31 `RMPS`**: Reference Message Payload Select
  - Ignore for time-slaves
  - 0 = Reference message has no additional payload
  - 1 = Take following elements from Tx Buffer 0
    - Message Marker `MM`
    - Event FIFO Control `EFC`

- Data Length Code DLC
- Data Bytes DB
  - **Level 1**: bytes 2 to 8
  - **Level 0, 2**: bytes 5 to 8

- **Bit 30 XTD**: Extended Identifier
  - 0 = 11-bit standard identifier
  - 1 = 29-bit extended identifier
- **Bit 28:0 RID[28:0]**: Reference Identifier
  - Transmits identifier with reference message and uses it for reference message filtering
  - Use standard or extended reference identifier based on bit XTD
  - For a standard identifier, write value to ID[28:18]

23.7.51 TT Operation Configuration (TTOCF)

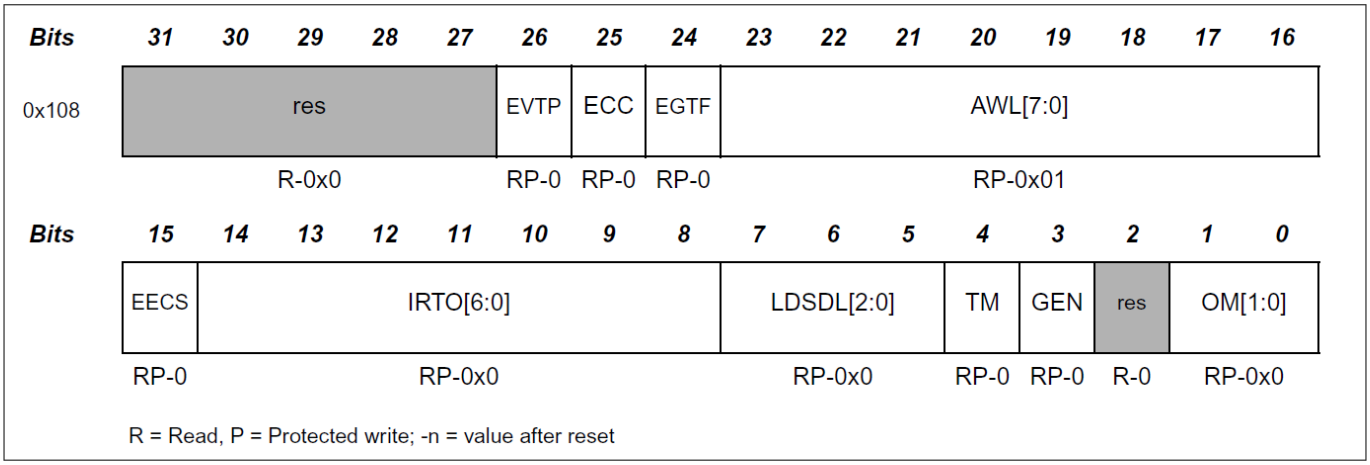


Figure 219 TT Operation Configuration (address 0x108)

- **Bit 26 EVTP**: Event Trigger Polarity
  - 0 = Rising-edge trigger
  - 1 = Falling-edge trigger
- **Bit 25 ECC**: Enable Clock Calibration
  - 0 = Automatic clock calibration in TTCAN Level 0 and 2 disabled
  - 1 = Automatic clock calibration in TTCAN Level 0 and 2 enabled
- **Bit 24 EGTF**: Enable Global Time Filtering
  - 0 = Global time filtering in TTCAN Level 0 and 2 disabled
  - 1 = Global time filtering in TTCAN Level 0 and 2 enabled
- **Bit 23:16 AWL[7:0]**: Application Watchdog Limit
  - Program AWL to 0x00 to disable the application watchdog
  - 0x00 to 0xFF = Maximum time after which the application must serve the application watchdog
  - The application watchdog increments once per 256 NTUs
- **Bit 15 EECS**: Enable External Clock Synchronization
  - If enabled, update TUR configuration (TURCF.NCL only) during TTCAN operation
  - 0 = External clock synchronization in TTCAN Level 0 and 2 disabled
  - 1 = External clock synchronization in TTCAN Level 0 and 2 enabled

- **Bit 14:8 IRTO[6:0]**: Initial Reference Trigger Offset
  - 0x00 to 0x7F = Positive offset range from 0 to 127
- **Bit 7:5 LDSDL[2:0]**: LD of Synchronization Deviation Limit
  - Configure Synchronization Deviation Limit SDL using dual logarithm LDSDL
  - SDL = 2^(LDSDL + 5)
  - Do not exceed clock tolerance defined by CAN bit-timing configuration
  - 0x0 to 0x7 = LD of Synchronization Deviation Limit (SDL ≤ 32 to 4096)
- **Bit 4 TM**: Time Master
  - 0 = Time Master function disabled
  - 1 = Potential Time Master
- **Bit 3 GEN**: Gap Enable
  - 0 = Strictly time-triggered operation
  - 1 = External event-synchronized time-triggered operation
- **Bit 1:0 OM[1:0]**: Operation mode
  - 00 = Event-driven CAN communication, default
  - 01 = TTCAN Level 1
  - 10 = TTCAN Level 2
  - 11 = TTCAN Level 0

23.7.52 TT Matrix Limits (TTMLM)

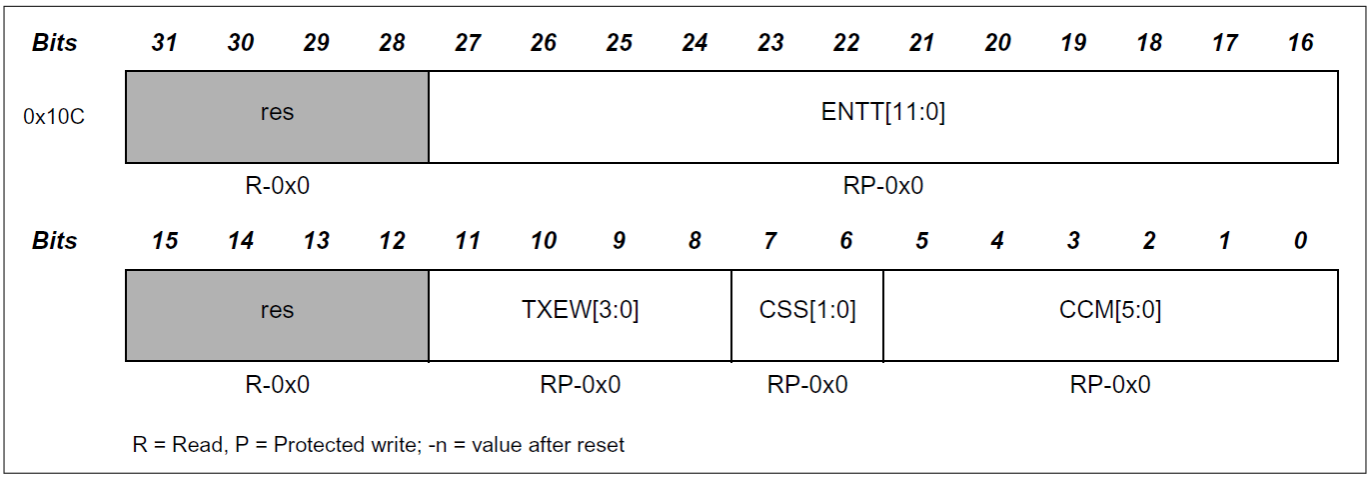


Figure 220 TT Matrix Limits (address 0x10C)

- **Bit 27:16 ENTT[11:0]**: Expected Number of Tx Triggers
  - 0x000 to 0xFFF = Expected number of Tx triggers in one Matrix Cycle
- **Bit 11:8 TXEW[3:0]**: Tx Enable Window
  - 0x0 to 0xF = Length of Tx enable window, 1 to 16 NTU cycles
- **Bit 7:6 CSS[1:0]**: Cycle Start Synchronization
  - Enables sync-pulse output on pin m_ttcan_soc
  - 00 = No sync pulse
  - 01 = Sync pulse at start of basic cycle

- 10 = Sync pulse at start of matrix cycle
- 11 = Reserved

- **Bit 5:0 CCM[5:0]**: Cycle Count Max
  - 0x00 = 1 Basic Cycle per Matrix Cycle
  - 0x01 = 2 Basic Cycles per Matrix Cycle
  - 0x03 = 4 Basic Cycles per Matrix Cycle
  - 0x07 = 8 Basic Cycles per Matrix Cycle
  - 0x0F = 16 Basic Cycles per Matrix Cycle
  - 0x1F = 32 Basic Cycles per Matrix Cycle
  - 0x3F = 64 Basic Cycles per Matrix Cycle
  - Others = Reserved

***Note**: ISO 11898-4, Section 5.2.1 requires configuring only the listed cycle-count values. Other values are possible, but they can lead to inconsistent matrix cycles.*

23.7.53 TUR Configuration (TURCF)

The NTU length is NTU = CAN clock period × NC/DC. NC is an 18-bit value. Upper bits NCH[17:16] are hard-wired to 0b01, so NC ranges from 0x10000 to 0x1FFFF. NCL configures the initial value for TURNA.NAV[15:0]. Hardware reset sets DC to 0x1000, and do not program DC to 0x0000.

- **Level 1**: NC ≥ 4 × DC and NTU = CAN bit time
- **Level 0 and 2**: NC ≥ 8 × DC

In TTCAN Level 0 and Level 2, the clock-drift compensation function can change the effective TUR to align the node-local NTU view with the time-master NTU view. Automatic drift compensation does not change DC. It adjusts TURNA.NAV around NC within the Synchronization Deviation Limit range defined by TTOCF.LDSDL. Program NC and DC to the largest suitable values to improve computational accuracy during drift compensation.

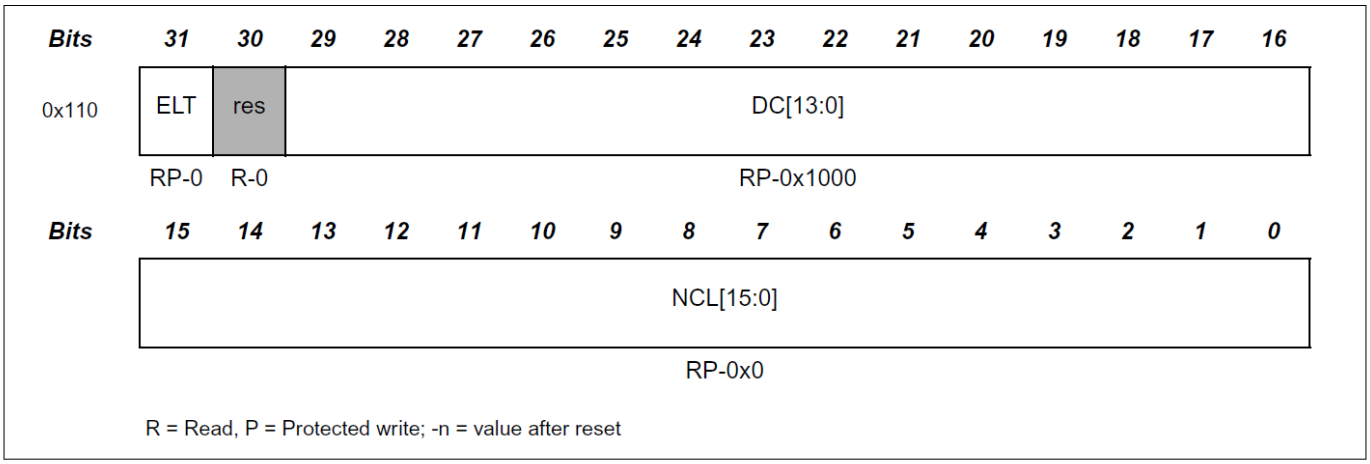


Figure 221 TUR Configuration (address 0x110)

- **Bit 31 ELT**: Enable Local Time
  - 0 = Local time stopped, default
  - 1 = Local time enabled
  - Set ELT to start local time
  - Local time remains active until you reset ELT or until next hardware reset
  - TURCF.DC locks when TURCF.ELT = 1

- If you write ELT = 0, the readable value remains 1 until the new value synchronizes into the CAN clock domain
- During synchronization, write access to other register bits remains locked

- **Bit 29:16 DC[13:0]**: Denominator Configuration
  - 0x0000 = Illegal value
  - 0x0001 to 0x3FFF = Denominator configuration
- **Bit 15:0 NCL[15:0]**: Numerator Configuration Low
  - Write access to NCL is allowed only when TURCF.ELT = 0 or when TTOCF.EECS = 1
  - If you write a new NCL value outside TT Configuration Mode, the new value takes effect when TTOST.WECS clears to 0
  - NCL locks when TTOST.WECS = 1
  - 0x0000 to 0xFFFF = Numerator configuration low

Note: *If NC < 7 × DC in TTCAN Level 1, subsequent time marks in Trigger Memory must differ by at least 2 NTU cycles.*

23.7.54 TT Operation Control (TTOCN)

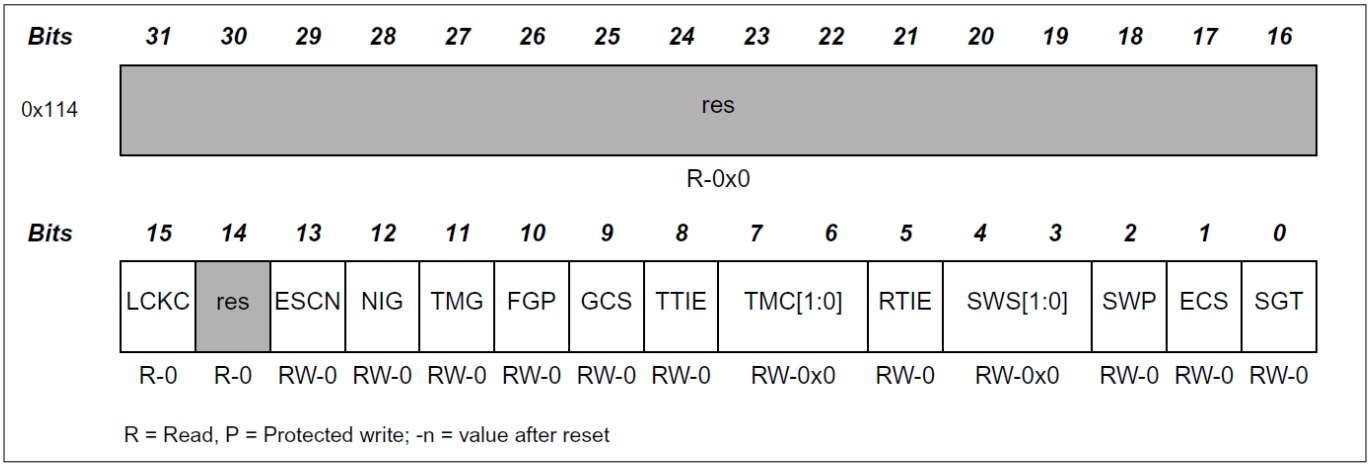


Figure 222 TT Operation Control (address 0x114)

- **Bit 15 LCKC**: TT Operation Control Register Locked
  - Set by a write access to TTOCN
  - Reset when the updated configuration synchronizes into the CAN-clock domain
  - 0 = Write access to TTOCN enabled
  - 1 = Write access to TTOCN locked
- **Bit 13 ESCN**: External Synchronization Control
  - Enables M_TTCAN to synchronize its cycle-time phase to an external event signaled by a rising edge on pin m_ttcan_evt (see Section 4.11)
  - 0 = External synchronization disabled
  - 1 = External synchronization enabled
- **Bit 12 NIG**: Next Is Gap
  - Set only when M_TTCAN is the active Time Master and external event-synchronized time-triggered operation is enabled (TTOCF.GEN = 1)

  - 0 = No action, reset by reception of any reference message
  - 1 = Transmit next reference message with `Next_is_Gap = 1`
- **Bit 11 `TMG`**: Time Mark Gap
  - 0 = Reset by each reference message
  - 1 = Start next reference message when Register Time Mark interrupt `TTIR.RTMI` is activated
- **Bit 10 `FGP`**: Finish Gap
  - CPU sets this bit, each reference message resets this bit
  - 0 = No reference message requested
  - 1 = Application requested start of reference message
- **Bit 9 `GCS`**: Gap Control Select
  - 0 = Gap control independent of `m_ttcan_evt`
  - 1 = Gap control by input pin `m_ttcan_evt`
- **Bit 8 `TTIE`**: Trigger Time Mark Interrupt Pulse Enable
  - Configure external time-mark events through trigger memory element `TMEX` (see Section 2.4.7)
  - Generate a trigger time-mark interrupt pulse when the trigger memory element becomes active and `M_TTCAN` is in synchronization state `In_Schedule` or `In_Gap`
  - 0 = Trigger Time Mark Interrupt output `m_ttcan_tmp` disabled
  - 1 = Trigger Time Mark Interrupt output `m_ttcan_tmp` enabled
- **Bit 7:6 `TMC[1:0]`**: Register Time Mark Compare
  - 00 = No Register Time Mark Interrupt generated
  - 01 = Register Time Mark Interrupt if Time Mark equals cycle time
  - 10 = Register Time Mark Interrupt if Time Mark equals local time
  - 11 = Register Time Mark Interrupt if Time Mark equals global time
  - Recommendation when changing time-mark reference (cycle, local, global)
    - Write `TMC = 00`
    - Reconfigure `TTTMK`
    - Set `TMC` to intended time reference
- **Bit 5 `RTIE`**: Register Time Mark Interrupt Pulse Enable
  - Configure register time-mark interrupts through `TTTMK`
  - Generate a register time-mark interrupt pulse with a length of 1 NTU when the time reference selected by `TTOCN.TMC` equals `TTTMK.TM`, independent of synchronization state
  - 0 = Register Time Mark Interrupt output `m_ttcan_rtp` disabled
  - 1 = Register Time Mark Interrupt output `m_ttcan_rtp` enabled
- **Bit 4:3 `SWS[1:0]`**: Stop Watch Source
  - 00 = Stop Watch disabled
  - 01 = Actual cycle-time value copied to `TTCPT.SWV`
  - 10 = Actual local-time value copied to `TTCPT.SWV`
  - 11 = Actual global-time value copied to `TTCPT.SWV`
- **Bit 2 `SWP`**: Stop Watch Polarity
  - 0 = Rising-edge trigger
  - 1 = Falling-edge trigger
- **Bit 1 `ECS`**: External Clock Synchronization
  - Writing 1 to `ECS` sets `TTOST.WECS` when the node is the active Time Master

- ECS resets after 1 host-clock period
- External clock synchronization takes effect at the start of the next basic cycle

- **Bit 0 SGT**: Set Global Time
  - Writing 1 to SGT sets TTOST.WGDT when the node is the active Time Master
  - SGT resets after 1 host-clock period
  - Global-time preset takes effect when the node transmits the next reference message with Master_Ref_Mark modified by the preset value written to TTGTP

23.7.55 TT Global Time Preset (TTGTP)

If TTOST.WGDT is set, the current Time Master transmits the next reference message with Master_Ref_Mark adjusted by the preset value and with Disc_Bit set to 1, which presets global time in all nodes simultaneously. The node resets TP to 0x0000 each time a reference message with Disc_Bit set to 1 becomes valid, or when the node is not the current Time Master. After setting TTOCN.SGT, the node locks TP while TTOST.WGTD is set to 1 until the reference message with Disc_Bit set to 1 becomes valid, or until the node is no longer the current Time Master.

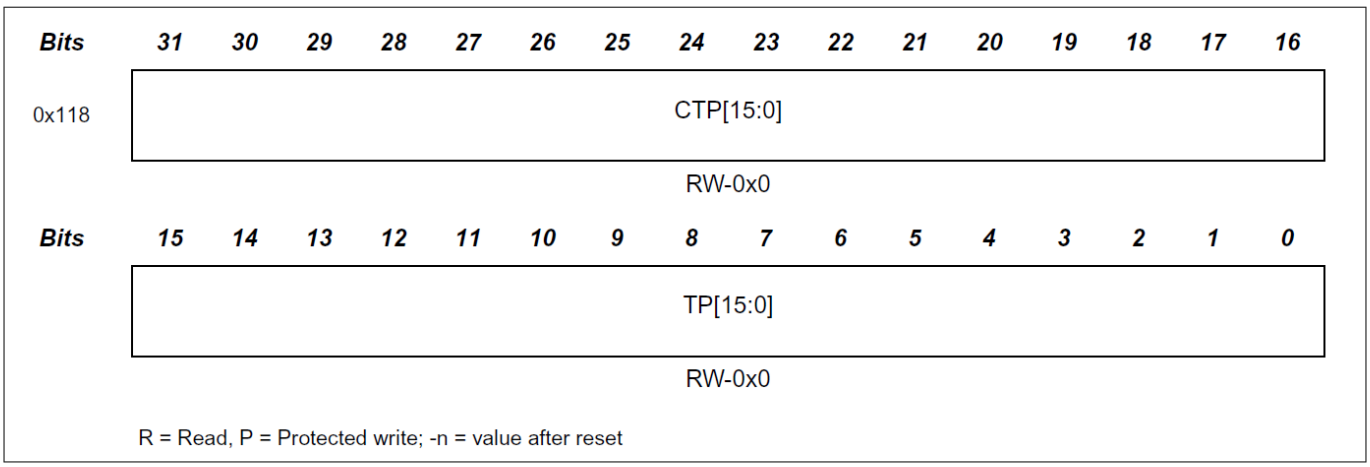


Figure 223 TT Global Time Preset (address 0x118)

- **Bits 31:16CTP[15:0]**: Cycle-Time Target Phase
  - CTP is write-protected while TTOCN.ESCN or TTOST.SPL is set
  - 0x0000–0xFFFF defines the target cycle-time value at which a rising edge of m_ttcan_evt is expected
- **Bits 15:0TP[15:0]**: Time Preset
  - TP is write-protected while TTOST.WGTD is set
  - 0x0000–0x7FFF Next Master Reference Mark = Master Reference Mark + TP
  - 0x8000 Reserved
  - 0x8001–0xFFFF Next Master Reference Mark = Master Reference Mark - (0x10000 - TP)

23.7.56 TT Time Mark (TTTMK)

The node generates a time-mark interrupt (TTIR.RTMI = 1) when the time base selected by TTOCN.TMC (cycle time, local time, or global time) equals TM.

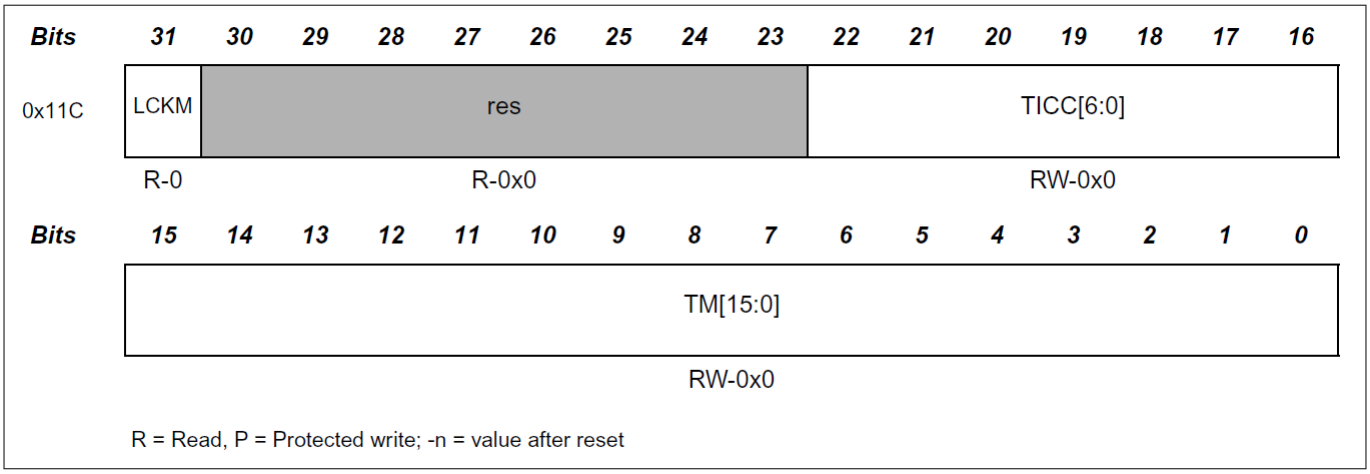


Figure 224 TT Time Mark (address 0x11C)

- **Bit** `31LCKM`: TT Time-Mark Register Locked
  - Write access to `TTOCN` registers always sets `LCKM`
  - Write access to `TTTMK` sets `LCKM` when `TTOCN.TMC` ≠ `"00"`
  - `LCKM` resets when the registers synchronize into the CAN clock domain
  - `0` = Write access to `TTTMK` enabled
  - `1` = Write access to `TTTMK` locked
- **Bits** `22:16TICC[6:0]`: Time-Mark Cycle Code
  - Cycle count for which the time mark is valid
  - `0b000000x` valid for all cycles
  - `0b000001c` valid every second cycle at cycle count `mod 2 = c`
  - `0b00001cc` valid every fourth cycle at cycle count `mod 4 = cc`
  - `0b0001ccc` valid every eighth cycle at cycle count `mod 8 = ccc`
  - `0b001cccc` valid every sixteenth cycle at cycle count `mod 16 = cccc`
  - `0b01ccccc` valid every thirty-second cycle at cycle count `mod 32 = ccccc`
  - `0b1cccccc` valid every sixty-fourth cycle at cycle count `mod 64 = cccccc`
- **Bits** `15:0TM[15:0]`: Time Mark
  - `0x0000–0xFFFF` Time Mark

***Note**: When you use byte access to register `TTTMK`, first disable the time-mark compare function by setting `TTOCN.TMC = "00"` to prevent compares against inconsistent register values.*

23.7.57 TT Interrupt Register (TTIR)

The flags are set when any listed condition occurs (edge-sensitive). Each flag remains set until the Host clears it. Clear a flag by writing a `1` to the corresponding bit position. Writing a `0` has no effect. A hard reset clears the register.

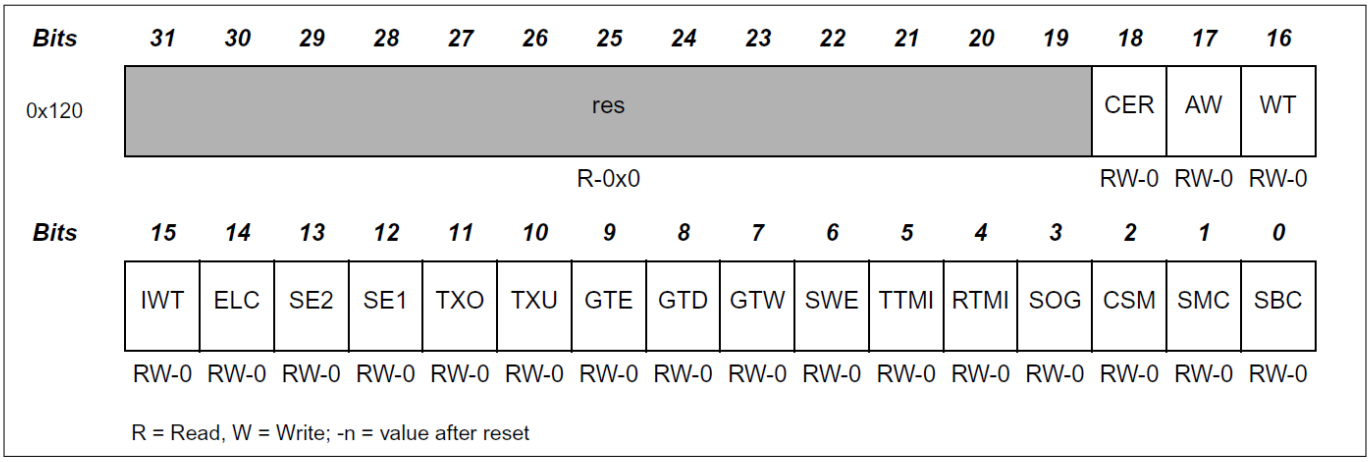


Figure 225 TT Interrupt Register (address 0x120)

- **Bit 18CER**: Configuration Error
  - Trigger occurs out of order
  - 0 = No error detected in trigger list
  - 1 = Error detected in trigger list
- **Bit 17AW**: Application Watchdog
  - 0 = Application watchdog served in time
  - 1 = Application watchdog not served in time
- **Bit 16WT**: Watch Trigger
  - 0 = No missing reference message
  - 1 = Missing reference message (Level 0: cycle time 0xFF00)
- **Bit 15IWT**: Initialization Watch Trigger
  - Node restarts initialization when Host resets IWT
  - 0 = No missing reference message during system startup
  - 1 = System startup blocked due to missing reference message
- **Bit 14ELC**: Error Level Changed
  - Not set when error level changes during initialization
  - 0 = No change in error level
  - 1 = Error level changed
- **Bit 13SE2**: Scheduling Error 2
  - 0 = No scheduling error 2
  - 1 = Scheduling error 2 occurred
- **Bit 12SE1**: Scheduling Error 1
  - 0 = No scheduling error 1
  - 1 = Scheduling error 1 occurred
- **Bit 11TXO**: Tx Count Overflow
  - 0 = Number of Tx triggers as expected
  - 1 = More Tx triggers than expected in one matrix cycle
- **Bit 10TXU**: Tx Count Underflow
  - 0 = Number of Tx triggers as expected
  - 1 = Fewer Tx triggers than expected in one matrix cycle

- **Bit 9GTE**: Global Time Error
  - Synchronization deviation SD exceeds the limit specified by TTOCF.LDSDL, TTCAN Level 0 and Level 2 only
  - 0 = Synchronization deviation within limit
  - 1 = Synchronization deviation exceeded limit
- **Bit 8GTD**: Global Time Discontinuity
  - 0 = No global-time discontinuity
  - 1 = Global-time discontinuity occurred
- **Bit 7GTW**: Global Time Wrap
  - 0 = No global-time wrap occurred
  - 1 = Global-time wrap occurred from 0xFFFF to 0x0000
- **Bit 6SWE**: Stop Watch Event
  - 0 = No rising or falling edge detected at stop-watch trigger pin m_ttcan_swt
  - 1 = Rising or falling edge detected at stop-watch trigger pin m_ttcan_swt
- **Bit 5TTMI**: Trigger Time-Mark Event Internal
  - Configure internal time-mark events by trigger memory element TMIN
  - Set when the trigger memory element becomes active and M_TTCAN is in synchronization state In_Gap or In_Schedule
  - 0 = Time mark not reached
  - 1 = Time mark reached (Level 0: cycle time TTOCF.IRTO × 0x200)
- **Bit 4RTMI**: Register Time-Mark Interrupt
  - Set when time referenced by TTOCN.TMC (cycle time, local time, or global time) equals TTTMK.TM, independent of synchronization state
  - 0 = Time mark not reached
  - 1 = Time mark reached
- **Bit 3SOG**: Start of Gap
  - 0 = No reference message received with Next_is_Gap bit set
  - 1 = Reference message with Next_is_Gap bit set becomes valid
- **Bit 2CSM**: Change of Synchronization Mode
  - 0 = No change in master-to-slave relation or schedule synchronization
  - 1 = Master-to-slave relation or schedule synchronization changed, also set when TTOST.SPL resets
- **Bit 1SMC**: Start of Matrix Cycle
  - 0 = No matrix cycle started since Host reset the bit
  - 1 = Matrix cycle started
- **Bit 0SBC**: Start of Basic Cycle
  - 0 = No basic cycle started since Host reset the bit
  - 1 = Basic cycle started

23.7.58 TT Interrupt Enable (TTIE)

Settings in the TT Interrupt Enable register define which status changes in the TT Interrupt Register generate an interrupt.

- 0 = TT interrupt disabled
- 1 = TT interrupt enabled

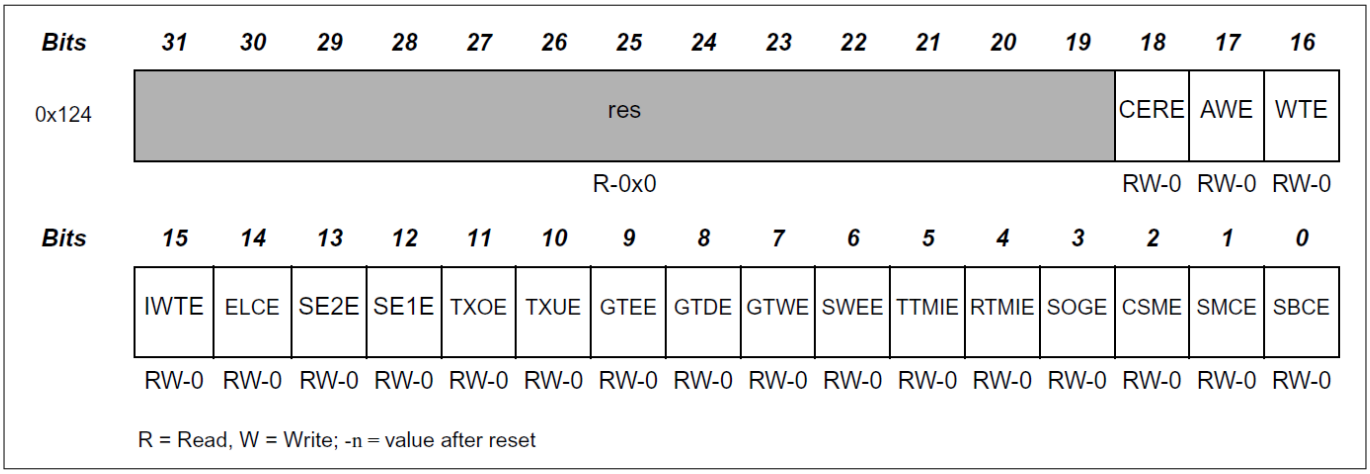


Figure 226 TT Interrupt Enable (address 0x124)

- **Bit 18 CERE**: Configuration Error Interrupt Enable
- **Bit 17 AWE**: Application Watchdog Interrupt Enable
- **Bit 16 WTE**: Watch Trigger Interrupt Enable
- **Bit 15 IWTE**: Initialization Watch Trigger Interrupt Enable
- **Bit 14 ELCE**: Change Error-Level Interrupt Enable
- **Bit 13 SE2E**: Scheduling Error 2 Interrupt Enable
- **Bit 12 SE1E**: Scheduling Error 1 Interrupt Enable
- **Bit 11 TXOE**: Tx Count Overflow Interrupt Enable
- **Bit 10 TXUE**: Tx Count Underflow Interrupt Enable
- **Bit 9 GTEE**: Global Time Error Interrupt Enable
- **Bit 8 GTDE**: Global Time Discontinuity Interrupt Enable
- **Bit 7 GTWE**: Global Time Wrap Interrupt Enable
- **Bit 6 SWEE**: Stop Watch Event Interrupt Enable
- **Bit 5 TTMIE**: Trigger Time-Mark Event Internal Enable
- **Bit 4 RTMIE**: Register Time-Mark Interrupt Enable
- **Bit 3 SOGE**: Start of Gap Interrupt Enable
- **Bit 2 CSME**: Change of Synchronization Mode Interrupt Enable
- **Bit 1 SMCE**: Start of Matrix Cycle Interrupt Enable
- **Bit 0 SBCE**: Start of Basic Cycle Interrupt Enable

23.7.59 TT Interrupt Line Select (TTILS)

The `TT Interrupt Line Select` register routes an interrupt, generated by a specific interrupt flag in the `TT Interrupt Register`, to one of the two module interrupt lines. To generate the interrupt, enable the selected line by setting `ILE.EINT0` or `ILE.EINT1`.

- `0` = Assign TT interrupt to interrupt line `m_ttcan_int0`
- `1` = Assign TT interrupt to interrupt line `m_ttcan_int1`

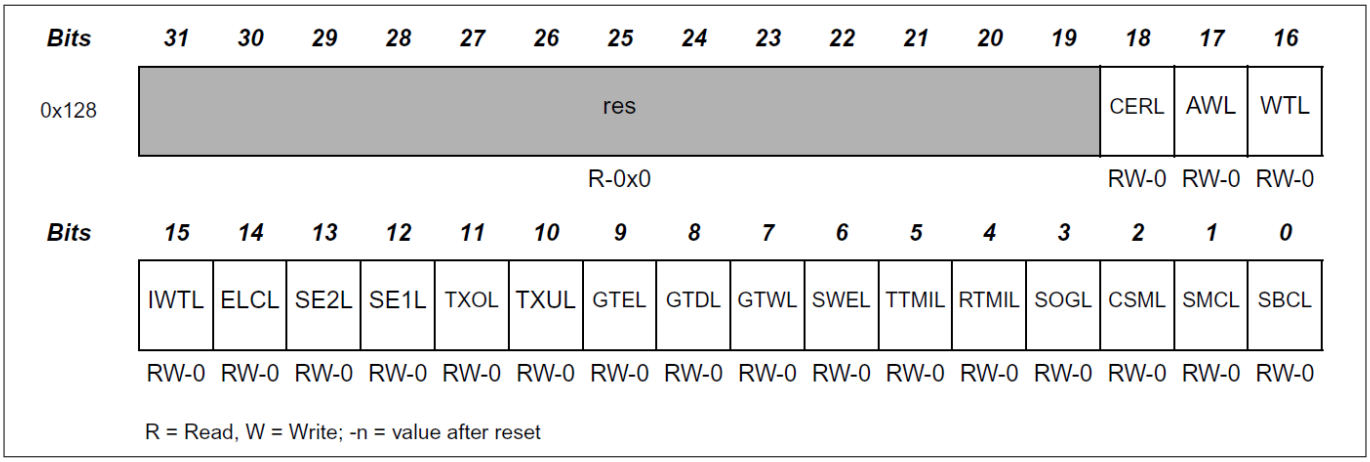


Figure 227 TT Interrupt Line Select (address 0x128)

- **Bit 18 CERL**: Configuration Error Interrupt Line
- **Bit 17 AWL**: Application Watchdog Interrupt Line
- **Bit 16 WTL**: Watch Trigger Interrupt Line
- **Bit 15 IWTL**: Initialization Watch Trigger Interrupt Line
- **Bit 14 ELCL**: Change Error-Level Interrupt Line
- **Bit 13 SE2L**: Scheduling Error 2 Interrupt Line
- **Bit 12 SE1L**: Scheduling Error 1 Interrupt Line
- **Bit 11 TXOL**: Tx-Count Overflow Interrupt Line
- **Bit 10 TXUL**: Tx-Count Underflow Interrupt Line
- **Bit 9 GTEL**: Global Time Error Interrupt Line
- **Bit 8 GTDL**: Global Time Discontinuity Interrupt Line
- **Bit 7 GTWL**: Global Time Wrap Interrupt Line
- **Bit 6 SWEL**: Stop-Watch Event Interrupt Line
- **Bit 5 TTMIL**: Trigger Time-Mark Event Internal Line
- **Bit 4 RTMIL**: Register Time-Mark Interrupt Line
- **Bit 3 SOGL**: Start of Gap Interrupt Line
- **Bit 2 CSML**: Change of Synchronization Mode Interrupt Line
- **Bit 1 SMCL**: Start of Matrix Cycle Interrupt Line
- **Bit 0 SBCL**: Start of Basic Cycle Interrupt Line

23.7.60 TT Operation Status (TTOST)

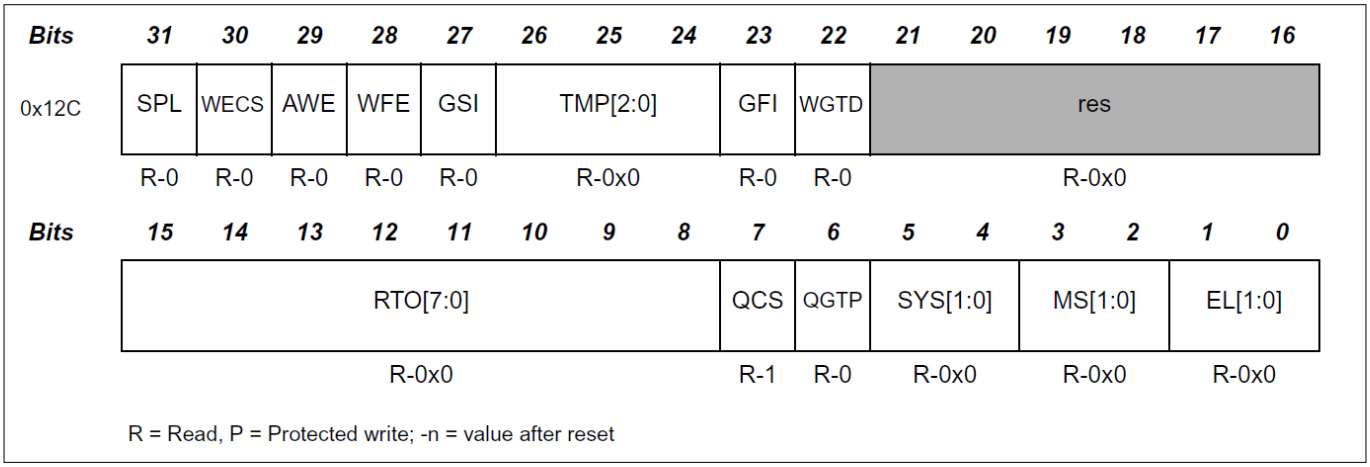


Figure 228 TT Operation Status (address 0x12C)

- **Bit `31SPL`**: Schedule-Phase Lock
  - Bit is valid only when external synchronization is enabled (`TTOCN.ESCN = 1`)
  - When valid, bit indicates that the difference between cycle time configured by `TTGTP.CTP` and the cycle time at the rising edge on pin `m_ttcan_evt` is less than or equal to `9 NTU`
  - `0` = Phase outside range
  - `1` = Phase inside range
- **Bit `30WECS`**: Wait for External-Clock Synchronization
  - `0` = No external-clock synchronization pending
  - `1` = Node waits for external-clock synchronization to take effect, bit resets at the start of the next basic cycle
- **Bit `29AWE`**: Application-Watchdog Event
  - Node serves the application watchdog by reading `TTOST`
  - When the watchdog is not served in time, node sets `AWE`, stops all TTCAN communication, and sets `M_TTCAN` to Bus Monitoring Mode
  - `0` = Application watchdog served in time
  - `1` = Application watchdog not served in time
- **Bit `28WFE`**: Wait for Event
  - `0` = No gap announced, reference message with `Next_is_Gap = 0` resets bit
  - `1` = Reference message with `Next_is_Gap = 1` received
- **Bit `27GSI`**: Gap-Started Indicator
  - `0` = No gap in schedule, each reference message resets bit, bit resets for all time slaves
  - `1` = Gap time after basic cycle has started
- **Bits `26:24TMP[2:0]`**: Time-Master Priority
  - `0x0–0x7` Priority of the current Time Master
- **Bit `23GFI`**: Gap-Finished Indicator
  - CPU write to `TTOCN.FGP` sets bit, time-mark interrupt sets bit when `TMG = 1`, input pin `m_ttcan_evt` sets bit when `TTOCN.GCS = 1`
  - `Ref_Trigger_Gap` does not set bit
  - Another node finishing the gap by sending a reference message does not set bit

  - 0 = Bit resets at the end of each reference message
  - 1 = M_TTCAN finished gap
- **Bit 22WGTD**: Wait for Global-Time Discontinuity
  - 0 = No global-time preset pending
  - 1 = Node waits for the global-time preset to take effect, bit resets when node transmits a reference message with Disc_Bit = 1 or after node receives a reference message
- **Bits 15:8RTO[7:0]**: Reference-Trigger Offset
  - Value is a signed integer in the range -127 (0x81) to 127 (0x7F)
  - No notification occurs when the lower limit of -127 is reached
  - When M_TTCAN becomes Time Master (MS[1:0] = "11"), reset of RTO is delayed due to synchronization between Host and CAN clock domains
  - For time slaves, node reads value configured by TTOCF.IRTO
  - 0x00–0xFF Actual reference-trigger offset value
- **Bit 7QCS**: Quality of Clock Speed
  - Relevant only in TTCAN Level 0 and Level 2, otherwise fixed to 1
  - 0 = Local clock speed not synchronized to Time-Master clock speed
  - 1 = Synchronization deviation ≤ SDL
- **Bit 6QGTP**: Quality of Global-Time Phase
  - Relevant only in TTCAN Level 0 and Level 2, otherwise fixed to 0
  - 0 = Global time not valid
  - 1 = Global time in phase with Time Master
- **Bits 5:4SYS[1:0]**: Synchronization State
  - 00 = Out of synchronization
  - 01 = Synchronizing to TTCAN communication
  - 10 = Schedule suspended by gap (In_Gap)
  - 11 = Synchronized to schedule (In_Schedule)
- **Bits 3:2MS[1:0]**: Master State
  - 00 = Master_Off, no master properties relevant
  - 01 = Operating as time slave
  - 10 = Operating as backup Time Master
  - 11 = Operating as current Time Master
- **Bits 1:0EL[1:0]**: Error Level
  - 00 = Severity 0, no error
  - 01 = Severity 1, warning
  - 10 = Severity 2, error
  - 11 = Severity 3, severe error

23.7.61 TUR Numerator Actual (TURNA)

TTCAN Level 1 provides no drift compensation because NAV = NC. In TTCAN Level 0 and Level 2, the node calculates drift between the node local clock and the Time Master local clock. The node compensates this drift when the Synchronization Deviation, defined as the difference between NC and the calculated NAV, is not greater than 2^(TTOCF.LDSDL + 5). When TTOCF.LDSDL ≤ 7, this condition limits NAV to (NC - 0x1000) ≤ NAV ≤ (NC + 0x1000).

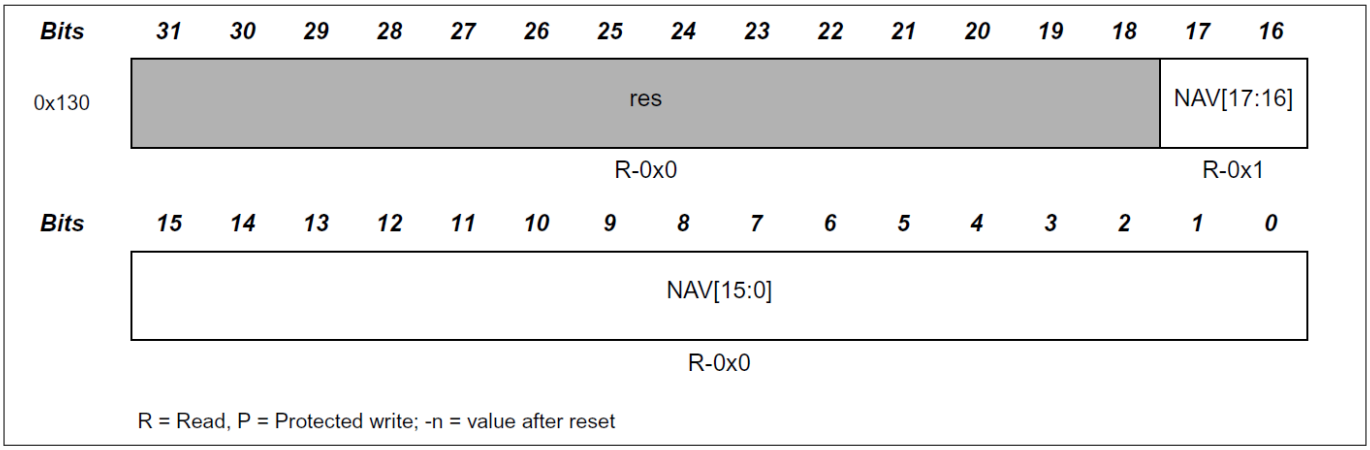


Figure 229 TUR Numerator Actual (address 0x130)

Bits `17:0` `NAV[17:0]`: Numerator Actual Value

- ≤ `0x0EFFF` Reserved
- `0x0F000–0x20FFF` Actual numerator value
- ≥ `0x21000` Reserved

23.7.62 TT Local and Global Time (TTLGT)

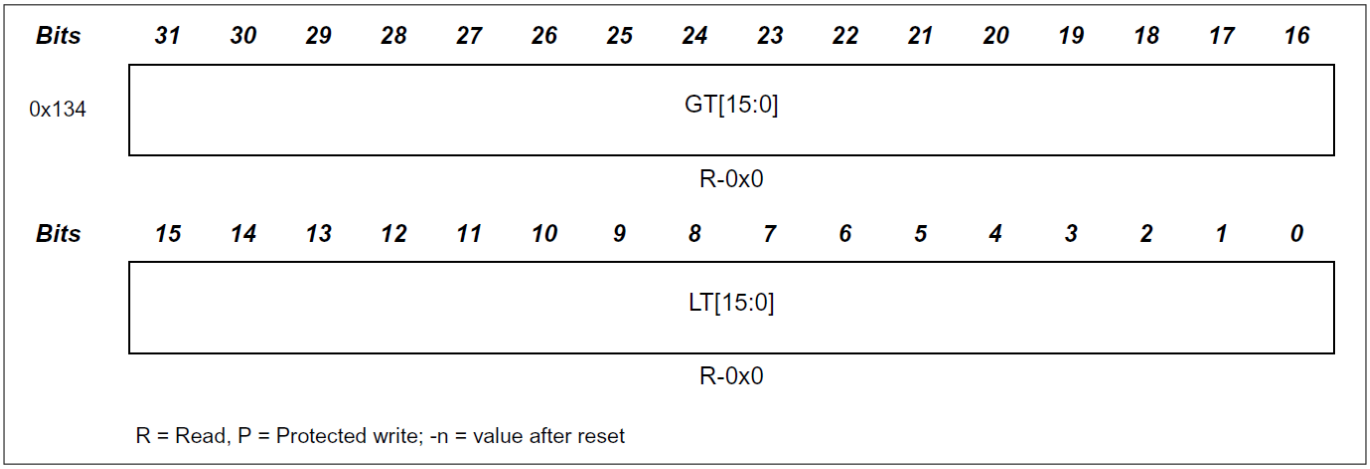


Figure 230 TT Local and Global Time (address 0x134)

- **Bits `31:16GT[15:0]`**: Global Time
  - Non-fractional part of the sum of node local time and node local offset
  - `0x0000–0xFFFF` Global-time value for the TTCAN network
- **Bits `15:0LT[15:0]`**: Local Time
  - Non-fractional part of local time, incremented once per local `NTU`
  - `0x0000–0xFFFF` Local-time value for the TTCAN node

23.7.63 TT Cycle Time and Count (TTCTC)

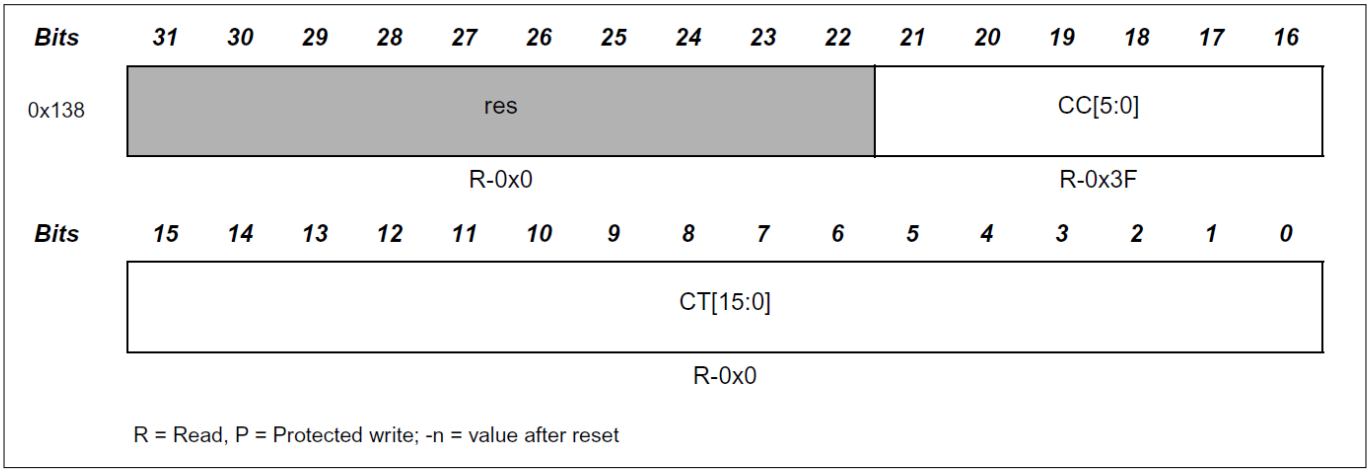


Figure 231 TT Cycle Time and Count (address 0x138)

- **Bits** `21:16CC[5:0]`: Cycle Count
  - `0x00–0x3F` Number of the current Basic Cycle in the System Matrix
- **Bits** `15:0CT[15:0]`: Cycle Time
  - Non-fractional part of the difference between node local time and `Ref_Mark`
  - `0x0000–0xFFFF` Cycle-time value of the TTCAN Basic Cycle

23.7.64 TT Capture Time (TTCPT)

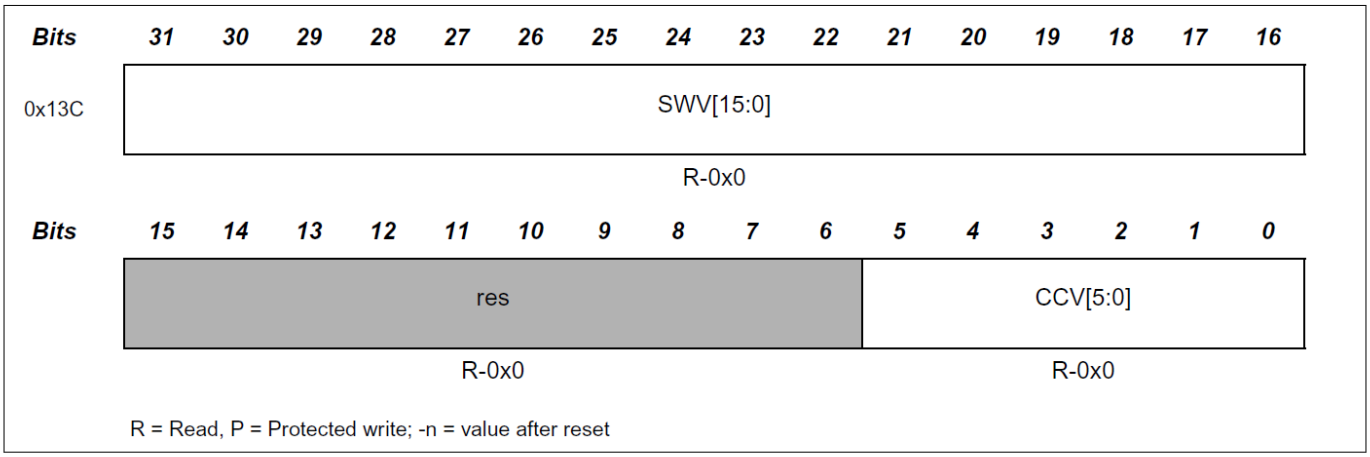


Figure 232 TT Capture Time (address 0x13C)

- **Bits** `31:16SWV[15:0]`: Stop-Watch Value
  - On a rising or falling edge, configured by `TTOCN.SWP`, at stop-watch trigger pin `m_ttcan_swt`, when `TTOCN.SWS ≠ "00"` and `TTIR.SWE = 0`, the node copies the time value selected by `TTOCN.SWS` (cycle time, local time, or global time) to `SWV` and sets `TTIR.SWE` to `1`
  - Reset `TTIR.SWE` to enable capture of the next stop-watch value
  - `0x0000–0xFFFF` Captured stop-watch value
- **Bits** `5:0CCV[5:0]`: Cycle-Count Value
  - Captures cycle-count value together with `SWV`
  - `0x00–0x3F` Captured cycle-count value

23.7.65 TT Cycle Sync Mark (TTCSM)

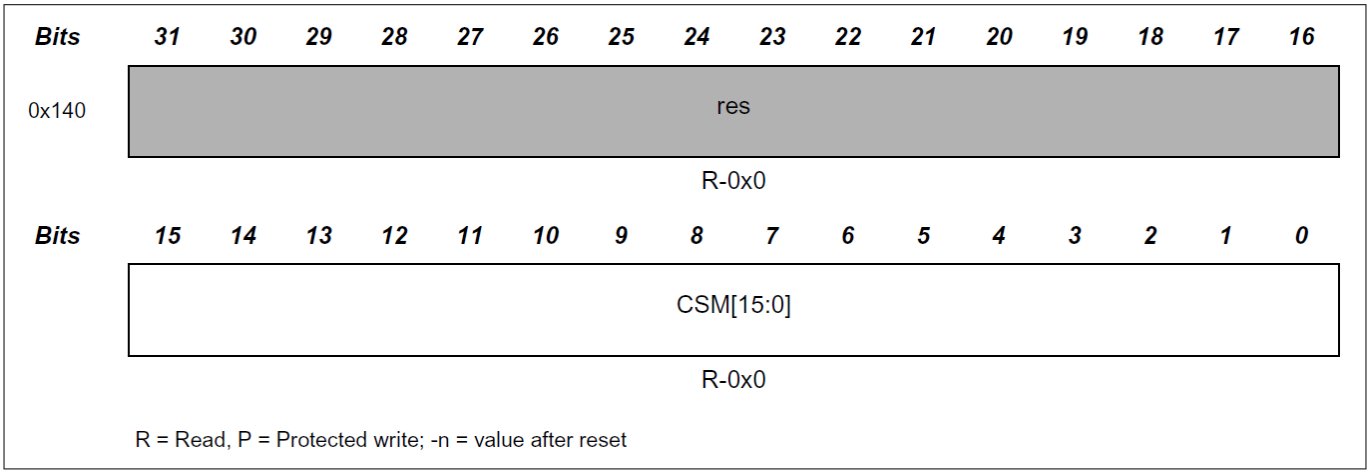


Figure 233 TT Cycle Sync Mark (address 0x140)

Bits `15:0` **`CSM[15:0]`**: Cycle-Sync Mark

- Measure `CSM` in cycle time
- Update `CSM` when the reference message becomes valid
- Retain `CSM` value until the next reference message becomes valid
- `0x0000–0xFFFF` Captured cycle-time value

24 Serial communications block (SCB)

The serial communications block (SCB) supports three serial communication protocols:

- Serial Peripheral Interface (SPI)
- Universal Asynchronous Receiver Transmitter (UART)
- Inter-Integrated Circuit (I^2C)

An SCB can only support one protocol at any given time.

The CYW20829 device includes three run-time configurable SCBs with the following protocol options:

- **First SCB**: Configurable as SPI or I^2C
- **Second SCB**: Configurable as SPI or UART
- **Third SCB**: Configurable as I^2C or UART

One SCB (SCB #0) is capable of operating in System Deep Sleep mode when using an external clock. In this mode, SCB #0 can function as either an SPI slave or an I^2C slave.

24.1 Features

The SCB supports the following features:

- Standard I^2C master and slave functionality
- Trigger outputs for connections to DMA (Direct Memory Access)
- Each SCB includes a 256-byte FIFO for receive (Rx) and transmit (Tx)
- Multiple interrupt sources to monitor the status of FIFOs and data transfers
- Local loopback control for testing and validation
- Deep Sleep-capable SCB features:
  - **EZ mode**: For SPI and I^2C slaves, allowing operation without CPU intervention
  - **CMD_RESP mode**: For SPI and I^2C slaves, also enabling operation without CPU intervention
  - **Low-power (Deep Sleep) operation mode**: For SPI and I^2C slaves using external clocking
  - Wakeup from Deep Sleep on I^2C slave address match or SPI slave selection

24.2 Architecture

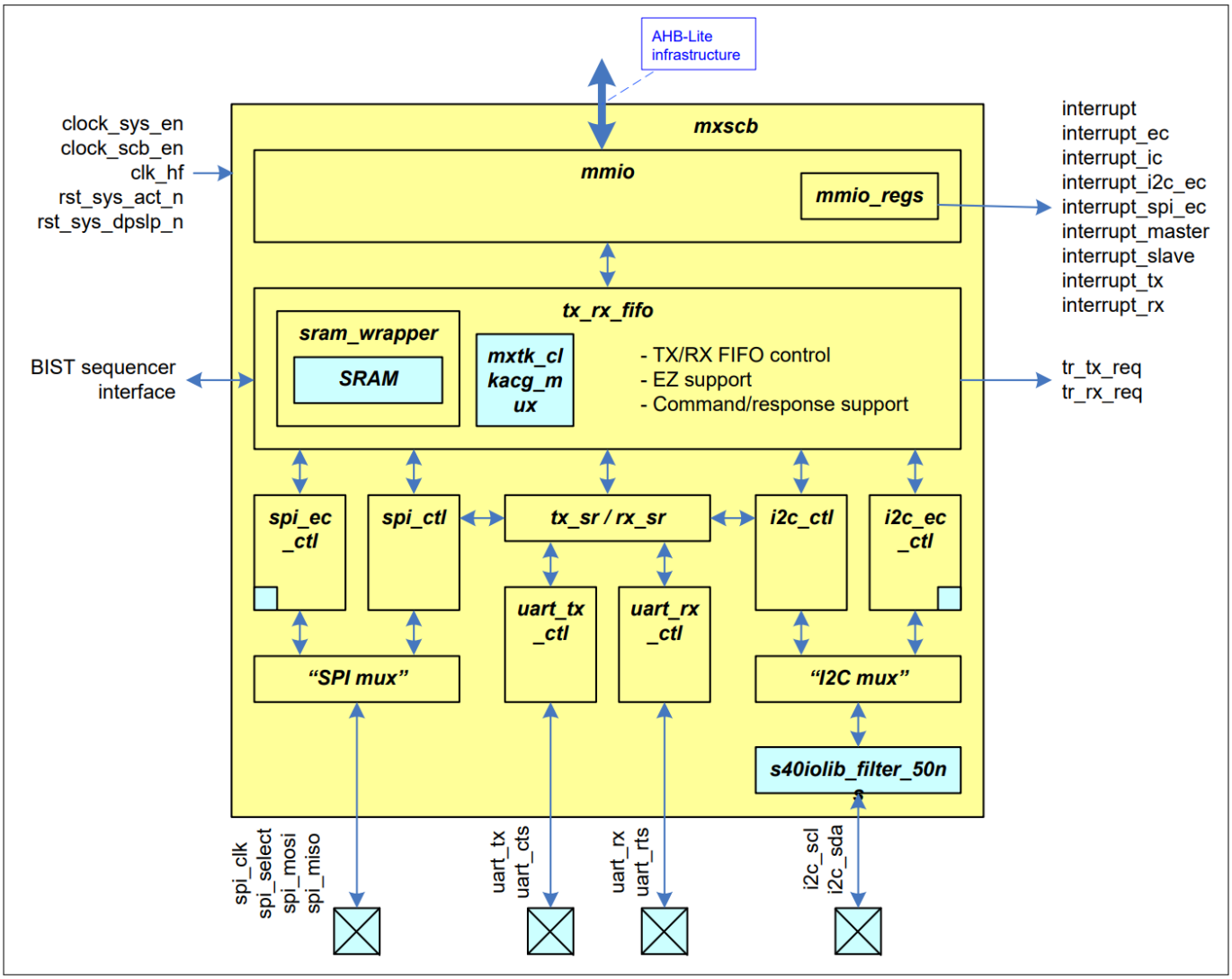


Figure 234 CYW20829 SCB architecture

AHB-Lite bus interface

The SCB block connects to the bus infrastructure through an AHB-Lite interface. This interface allows bus masters, such as the CPU, to access the SCB block's registers. These registers control the SCB's operation and provide status information. Details about the registers and their fields can be found in the register map.

The AHB-Lite interface manages all accesses to the SCB block's 64-Kbyte memory aperture. Its functionality includes the following:

- An AHB-Lite bus error is generated for non-32-bit accesses
- An AHB-Lite bus error is generated for non-aligned 32-bit accesses
- Read accesses to memory aperture locations not populated by a register return `0`
- Write accesses to memory aperture locations not populated by a register are ignored, and no AHB-Lite bus error is generated

24.2.1 Trigger interface

DMA/DW trigger signals

The trigger interface provides status information for the Tx FIFO (SCB_TX[x]) and Rx FIFO (SCB_RX[x]), where x signifies the block number. The behavior of these signals is described below:

- **Tx FIFO trigger (SCB_TX[x])**:
  - Indicates that the Tx FIFO can accept a data element for transmission
  - Controlled by SCBx.TR_TX_REQ
- **Rx FIFO trigger (SCB_RX[x])**:
  - Indicates that the Rx FIFO can provide a received data element
  - Controlled by SCBx.TR_RX_REQ

Both signals are level-sensitive.

The trigger interface is typically connected, either directly or indirectly, to a DW/DMA controller. The timing for Rx and Tx FIFO operations is as follows:

- **Rx FIFO read case**:
  - It requires 2 CLK_AHB cycles and 1 CLK_PERI cycle from the AHB read operation to clearing SCB_RX[x]
  - If CLK_AHB equals CLK_PERI, it takes three CLK_PERI cycles
- **Tx FIFO write case**:
  - It requires 3 CLK_AHB cycles and 1 CLK_PERI cycle from the AHB write operation to clearing SCB_TX[x]
  - If CLK_AHB equals CLK_PERI, it takes four CLK_PERI cycles

SCB_I²C_SCL[x] signal

The SCB_I²C_SCL[x] signal (where x represents the block number) is designed for detecting timeouts on the I²C SCL line. This signal is connected to the SCL input analog filter's output, ensuring that glitches are removed.

Key features and uses include:

- Along with the TCPWM block, it supports SMBus timeout detection, as required by the SMBus specification
- It can also be used to detect SCL stretching

Serial protocol interfaces

The SPI, UART, and I²C signal interfaces connect to the High-Speed I/O Matrix (HSIOM) within the I/O Subsystem (IOSS). These connections provide flexible routing of on-chip signals to external I/O pads.

Key features of the interface include:

- The HSIOM multiplexes between on-chip signals and I/O pads
- The multiplexing flexibility is chip-specific and is controlled by the IOSS/HSIOM registers
- The HSIOM enables a single serial interface to be exposed at multiple pad locations, offering system-level flexibility

If SMARTIO functionality is available, it can enhance these interfaces by providing additional configurability:

- It can short UART_TX and UART_RX to support single-line, half-duplex UART operation
- It can short MOSI and MISO to support single-line, half-duplex SPI operation

24.2.2 Clock and reset interface

The SCB block receives the following clock and reset signals for its operation:

1. **High-frequency clock (CLK_PERI)**:
   - This clock is used to derive:
     - AHB-Lite interface clock (CLK_SYS)
     - SCB functionality clock (CLK_SCB)
   - Clock relationships:

```
CLK_PERI ≥ CLK_SYS (CLK_GR6) = CLK_AHB ≥ CLK_SCB
           (PCLK_SCB_CLOCK)
```

2. **System clock enable**: clock_sys_en is provided by the SRSS and is used to derive the system clock (CLK_SYS) from CLK_PERI
3. **SCB clock enable**: clock_scb_en is provided by the PCLK component in the PERI block and is used to derive the SCB functionality clock (CLK_SCB) from CLK_PERI
4. **SCB clock division**: CLK_SCB can be generated using either an integer or fractional divider from the PERI block:
   - The fractional divider causes varying cycle times in the generated CLK_SCB
   - The integer divider must be used for I^2C and SPI (synchronous interfaces)
   - Both integer and fractional dividers can be used for UART (asynchronous interfaces)
5. **Usage of CLK_SCB**:
   - CLK_SCB is used in internally-clocked mode
   - ***Note***: *CLK_SCB is available only in Active and Sleep power modes. Internally-clocked mode is not available in Deep Sleep mode*
   - Serial interface protocols, such as UART Tx/Rx and I^2C/SPI master functionality, use CLK_SCB as an oversampled multiple of the desired interface clock. For example:
     - To implement a 100-kHz UART, set CLK_SCB to 1 MHz and choose an oversample factor of 10 (SCBx_CTRL.OVS = 10 – 1)
6. **Reset signals**:
   - A low-active system reset for Active mode functionality: rst_sys_act_n
   - A low-active system reset for Deep Sleep mode functionality: rst_sys_dpslp_n
7. **Externally-clocked slave mode**: In this mode, serial interface input signals serve as clocks for the SCB:
   - **For I^2C**: SCL
   - **For SPI**: SCLK
   - These clocks are asynchronous to CLK_SYS and CLK_AHB, which are derived from CLK_PERI
   - Reset signals in this mode are derived from rst_sys_dpslp_n and exhibit synchronous de-assertion with reference to the serial interface clock

24.2.2.1 Block enable

Further details about initialization are provided in the description field of the SCBx_CTRL.ENABLED register.

Key behavior of non-retention registers:

- All registers marked as "NonRetention" will be reset to their default state when the block is disabled
- This includes the SCBx_INTR_XXX and SCBx_INTR_XXX_SET registers

24.2.2.2 Interrupt interface

The SCB block supports six types of interrupts. Each interrupt type is associated with dedicated registers.

Table 167 Interrupt interface signals and registers

Interrupt	Functionality	Active/Deep Sleep	Sync/Async	Registers
interrupt_master	I^2C master and SPI master functionality	Active	Sync	SCBx_INTR_M, SCBx_INTR_M_SET, SCBx_INTR_M_MASK, SCBx_INTR_M_MASKED
interrupt_slave	I^2C slave and SPI slave functionality	Active	Sync	SCBx_INTR_S, SCBx_INTR_S_SET, SCBx_INTR_S_MASK, SCBx_INTR_S_MASKED
interrupt_tx	UART transmitter and Tx FIFO functionality	Active	Sync	SCBx_INTR_TX, SCBx_INTR_TX_SET, SCBx_INTR_TX_MASK, SCBx_INTR_TX_MASKED
interrupt_rx	UART receiver and Rx FIFO functionality	Active	Sync	SCBx_INTR_RX, SCBx_INTR_RX_SET, SCBx_INTR_RX_MASK, SCBx_INTR_RX_MASKED
interrupt_i2c_ec	Externally clocked I^2C slave functionality	Deep Sleep	Async	SCBx_INTR_I2C_EC, SCBx_INTR_I2C_EC_MASK, SCBx_INTR_I2C_EC_MASKED
interrupt_spi_ec	Externally clocked SPI slave functionality	Deep Sleep	Async	SCBx_INTR_SPI_EC, SCBx_INTR_SPI_EC_MASK, SCBx_INTR_SPI_EC_MASKED

The SCB block generates interrupts based on its mode of functionality:

- **Active functionality interrupts**: These are generated synchronously to `CLK_PERI`
- **Deep Sleep functionality interrupts**: These are generated asynchronously to `CLK_PERI` and require synchronization within the CPUSS interrupt multiplexer

For chips with a limited number of interrupt lines, the SCB block also provides combined-functionality interrupts as follows:

- `interrupt_ec`: The logical OR of `interrupt_i2c_ec` and `interrupt_spi_ec`

- `interrupt_ic`: The logical OR of `interrupt_master`, `interrupt_slave`, `interrupt_tx`, and `interrupt_rx`
- `interrupt`: The logical OR of all six individual interrupts

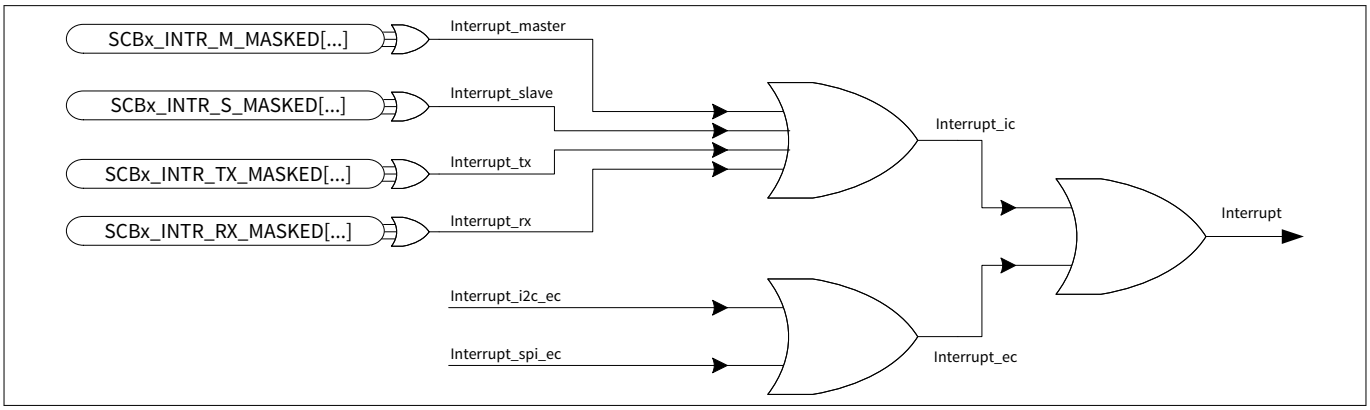


Figure 235 Interrupt lines

The SCB block generates interrupts based on the clock source of the associated logic:

- **Internal-clocked logic interrupts**:
  - `SCBx_INTR_M`
  - `SCBx_INTR_S`
  - `SCBx_INTR_TX`
  - `SCBx_INTR_RX`
- **External-clocked logic interrupts**:
  - `SCBx_INTR_I2C_EC`
  - `SCBx_INTR_SPI_EC`

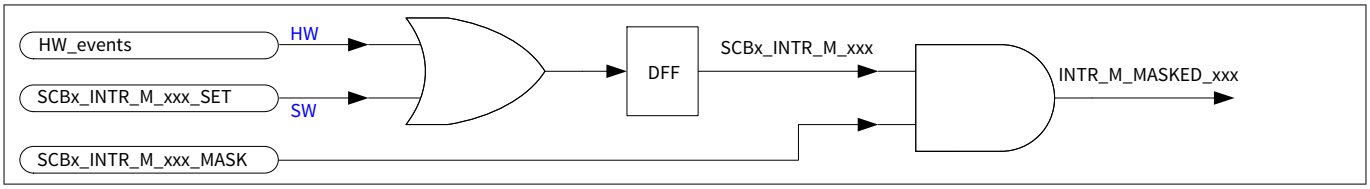


Figure 236 SCBx_INTR_M generation

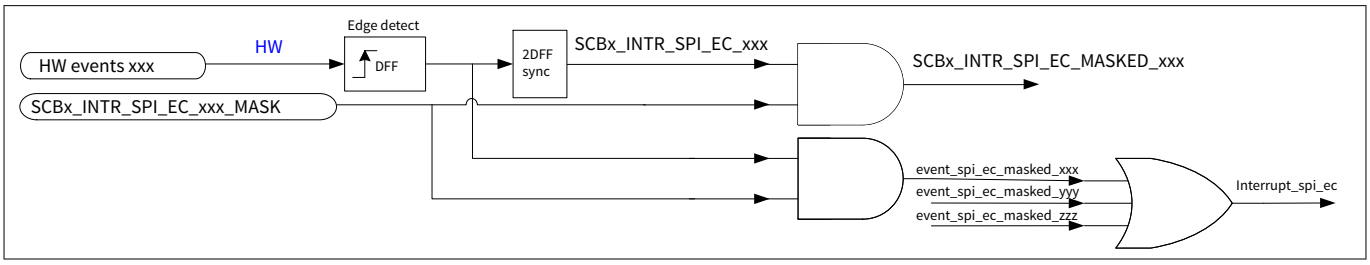


Figure 237 SCBx_INTR_SPI_EC and interrupt_spi_ec generation

Interrupt request registers, such as `SCBx_INTR_M`, have the following behavior:

- These registers can only be set by hardware (`HW RW1S`)
- They can only be cleared by software (`SW RW1C`)

To prevent interrupts from being triggered by events from previous transactions, the following procedure should be followed:

- Before enabling an interrupt mask register bit (for example, `SCBx_INTR_M_MASK.I2C_STOP`), the firmware should clear the corresponding interrupt request register (for example, `SCBx_INTR_M.I2C_STOP`)

24.2.3 Operational modes

24.2.3.1 Buffer modes

Each SCB includes 256 bytes of dedicated RAM for transmit and receive operations. This RAM can be configured in three different modes: FIFO, EZ, or CMD_RESP. Below is a high-level description of these modes, while detailed protocol-specific information is provided in later sections.

- **Masters**: Can only use FIFO mode
- **I2C and SPI slaves**: Can operate in all three modes—FIFO, EZ, and CMD_RESP
- **UART**: Operates using FIFO mode only
- **CMD_RESP mode**: Is available exclusively on Deep Sleep-capable SCBs

This document discusses the hardware implementation of EZ mode. For details on the firmware implementation, see the PDL. Figure 10 provides an overview of the buffer modes configured using a dedicated SRAM. Based on the `SCBx_CTRL.MEM_WIDTH` register, the SCB FIFOs are configured as follows:

- **Two 32-deep FIFOs**: For up to 32-bit data elements (`SCBx_CTRL.MEM_WIDTH = 2`)
- **Two 64-deep FIFOs**: For up to 16-bit data elements (`SCBx_CTRL.MEM_WIDTH = 1`)
- **Two 128-deep FIFOs**: For up to 8-bit data elements (`SCBx_CTRL.MEM_WIDTH = 0`)

In addition to the FIFO architecture, the following memory buffers are available:

- A 256-byte EZ memory buffer
- A 256-byte CMD_RESP memory buffer

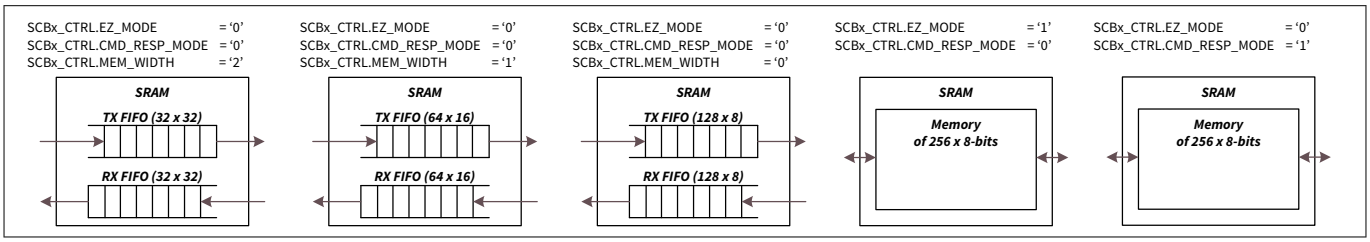


Figure 238 Buffer modes using a dedicated SRAM

24.2.3.2 FIFO mode

In FIFO mode, the RAM is divided into two 128-byte FIFOs: one for transmit (Tx) operations and one for receive (Rx) operations. The FIFOs can be configured for the following sizes:

- 8 bits x 128 elements
- 16 bits x 64 elements

The FIFO mode of operation is available only in Active and Sleep power modes. However, on Deep Sleep-capable SCBs, the I^2C address or SPI slave select signal can be used to wake the device from Deep Sleep.

Write access to the Tx FIFO is performed using the `SCBx_TX_FIFO_WR.DATA` register. Read access from the Rx FIFO is performed using the `SCBx_RX_FIFO_RD.DATA` register. To read a data frame without removing it from the FIFO, the `SCBx_RX_FIFO_RD_SILENT.DATA` register can be used.

Status information is provided for both Tx and Rx FIFOs, such as full and empty states. Multiple interrupt sources are available to indicate the status of FIFOs. For more details, see SCB interrupts.

24.2.3.3 EZ mode

In EZ mode, the RAM is utilized as a single 256-byte buffer. The external master sets a base address, and all subsequent read and write operations proceed starting from that base address.

Key features of EZ mode include:

- EZ mode is only available for SPI slave and I²C slave configurations
- It is supported exclusively on Deep Sleep-capable SCBs
- EZ mode operates in both Active and Deep Sleep power modes

24.2.3.4 CMD RESP mode

Command Response (CMD_RESP) mode operates similarly to EZ mode, with the key difference being that the base address is provided by the CPU instead of an external master.

Key features of CMD_RESP mode include:

- CMD_RESP mode is available only for SPI slave and I²C slave configurations
- It is supported exclusively on Deep Sleep-capable SCBs
- CMD_RESP mode operates in both Active and Deep Sleep power modes

24.2.3.5 Clocking mode

The SCB can operate using either an internal clock provided by the peripheral clock dividers or an external clock provided by an external master.

Clocking requirements:

- UART, SPI Master, and I²C master modes must use the internal 24 MHz clock
- Only SPI Slave and I²C slave modes can use an external clock from an external master, and this is supported exclusively on Deep Sleep-capable SCBs

Internally- and externally-clocked slave functionality is determined by two fields in the SCBx_CTRL register:

- **SCBx_CTRL.EC_AM_MODE**: Determines whether SPI slave selection or I²C address matching is clocked internally (0) or externally (1)
- **SCBx_CTRL.EC_OP_MODE**: Determines whether the main protocol operation (apart from SPI slave selection and I²C address matching) is clocked internally (0) or externally (1)

When operating as an externally-clocked slave, consider the following:

- FIFO mode is not supported with externally-clocked operation (SCBx_CTRL.EC_OP_MODE = 1)
- EZ and CMD_RESP modes are supported with externally-clocked operation (SCBx_CTRL.EC_OP_MODE = 1)

When using Deep Sleep mode with externally-clocked functionality:

- Before entering Deep Sleep mode, set the SCBx_CTRL.EC_ACCESS register to 1
- Upon waking from Deep Sleep mode and after the PLL is locked (ensuring CLK_SCB is at the expected frequency), reset the SCBx_CTRL.EC_ACCESS register to 0

Table 168 Clock mode compatibility

	Internally clocked (“IC")			Externally clocked (“EC”)		
	FIFO	**EZ**	**CMD_RESP**	**FIFO**	**EZ**	**CMD_RESP**
I²C master	Yes	No	No	No	No	No
I²C slave	Yes	Yes	No	Yes [a]	Yes	Yes
I²C master-slave	Yes	No	No	No	No	No
SPI master	Yes	No	No	No	No	No
SPI slave	Yes	Yes	No	Yes [b]	Yes	Yes

(table continues...)

Table 168 (continued) Clock mode compatibility

	Internally clocked (“IC")			Externally clocked (“EC”)		
	FIFO	**EZ**	**CMD_RESP**	**FIFO**	**EZ**	**CMD_RESP**
UART transmitter	Yes	No	No	No	No	No
UART receiver	Yes	No	No	No	No	No

***Notes**:*

1. *In Deep Sleep mode, the external-clocked logic can handle I^2C slave address matching. When an address match is detected, it triggers an interrupt to wake up the CPU*

2. *In Deep Sleep mode, the external-clocked logic can handle SPI slave selection detection. When a slave selection is detected, it triggers an interrupt to wake up the CPU. While the internal logic is inactive:*

- *Write operations are ignored*
- *Read operations return `0xFF`*

This functionality applies only to Deep Sleep-capable SCBs

24.3 Serial peripheral interface (SPI)

The SPI protocol is a synchronous serial interface that supports both master and slave modes of operation. The master device is responsible for initiating data transfers.

The SCB supports a single-master, multiple-slaves topology for SPI communication. Each slave device is connected to an individual slave-select (SS) line.

SPI data transfer process:

- During data transfer, the master selects a slave by asserting its SS line
- The master transmits data on the Master-Out-Slave-In (MOSI) line and provides a clock signal on the SCLK line
- Depending on the configuration, the slave captures data from the MOSI line using either the rising or falling edge of the SCLK signal. Simultaneously, the slave sends data back to the master on the Master-In-Slave-Out (MISO) line, which the master captures

24.3.1 Features

The SPI functionality provides support for both master and slave modes, with a wide range of configurable options for enhanced performance and flexibility.

- **Supported SPI protocols**:
  - **Motorola SPI**: Modes 0, 1, 2, and 3
  - **Texas Instruments SPI**: Mode 1 only, with coinciding and preceding data frame indicators
  - **National Semiconductor (MicroWire) SPI**: Mode 0 only
- **Key features**:
  - The master supports up to four slave-select lines, each with configurable active polarity (high or low)
  - Slave-select behavior can be programmed to remain active for the entire transfer or activate for each byte

  - The master supports features like:
    - Late sampling for improved timing margins
    - Continuous SPI clock for uninterrupted communication
  - Data frame size is programmable from 4 bits to 32 bits
- **Flexible SELECT timing (SPI master)**: Configurable SELECT signal timings, including:
  - SELECT setup time (slave select active to SPI clock)
  - SELECT hold time (SPI clock to slave select inactive)
  - Inter-data-frame deselect time (slave select inactive to active transition)
- **Additional functional features**:
  - Parity support for both odd and even parity checks
  - Data transfer configurable as MSB-first or LSB-first
  - Interrupts or polling available for CPU interface
  - Programmable oversampling for precise clocking
  - Median filter available for input signal noise suppression
  - Wake-up interrupt support upon slave selection
- **Supported modes**:
  - FIFO mode
  - EZ mode (available for slaves only)
  - CMD_RESP mode (available for slaves only)
- **Control and data handling features**:
  - Local loopback control for testing
  - Interrupt and polling options for efficient data management

24.3.2 General description

Figure 239 illustrates an example of SPI master with four slaves.

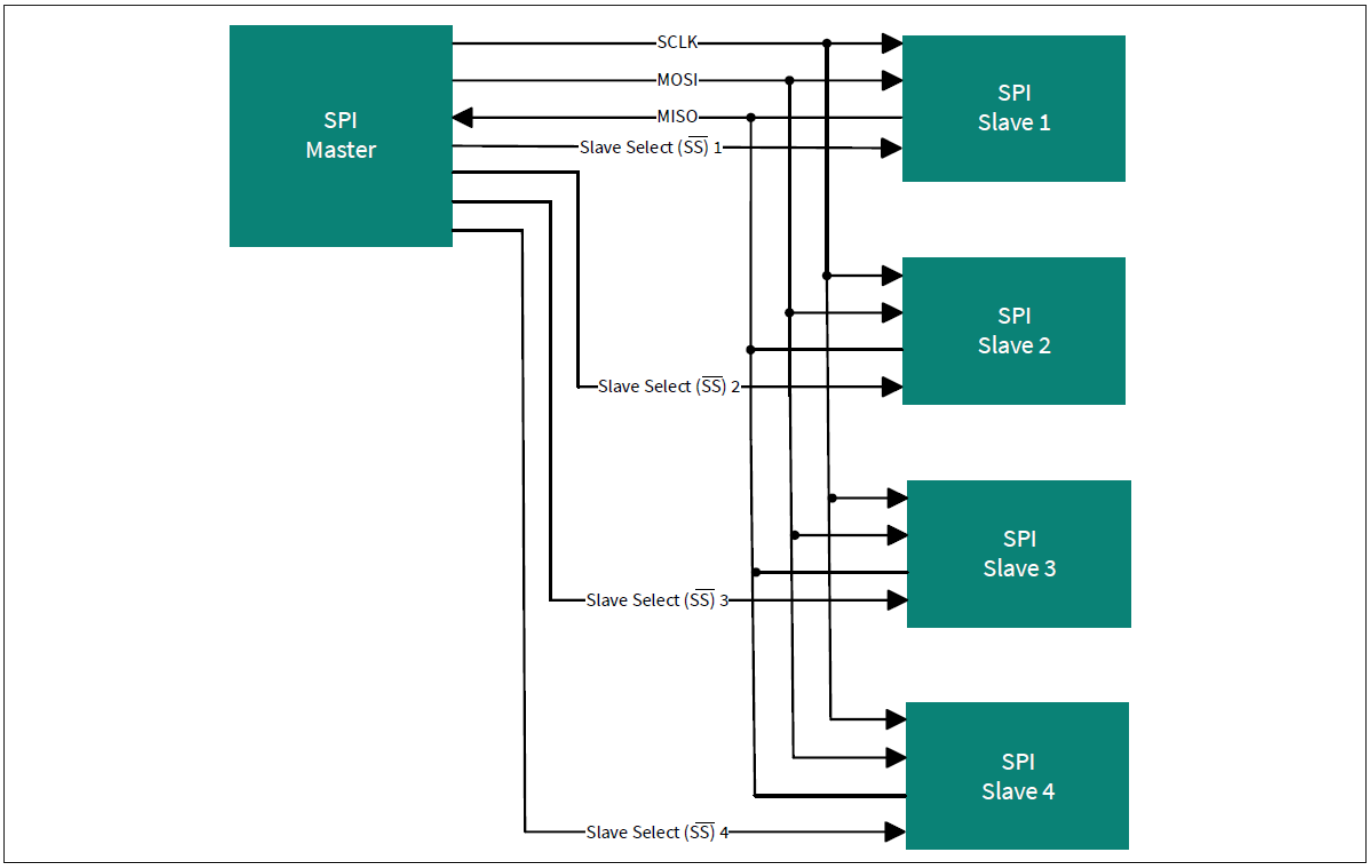


Figure 239 SPI example

A standard SPI interface consists of four primary signals:

- **Serial clock (SCLK)**: Clock signal output from the master and input to the slave
- **Master-Out-Slave-In (MOSI)**: Data signal output from the master and input to the slave
- **Master-In-Slave-Out (MISO)**: Data signal input to the master and output from the slave
- **Slave select (SS)**: Typically an active-low signal output from the master and input to the slave

SPI data transfer process: A simple SPI data transfer involves the following steps:

1. The master selects a slave by asserting its SS line
2. The master drives data on the MOSI line and provides a clock signal on the SCLK line
3. Depending on the configuration, the slave captures data on the MOSI line using either edge of the SCLK signal. At the same time, the slave drives data on the MISO line, which is captured by the master

Data frame size

- By default, the SPI interface supports a data frame size of eight bits (1 byte)
- The data frame size can be configured to any value in the range of 4 to 32 bits
- Serial data can be transmitted with the most significant bit (MSb) first or the least significant bit (LSb) first

Supported SPI protocol variants: The SCB supports three variants of the SPI protocol:

- **Motorola SPI**: The original SPI protocol
- **Texas Instruments SPI**: A variation of the SPI protocol where data frames are identified by a pulse on the SS line
- **National Semiconductors SPI**: A half-duplex variation of the original SPI protocol

***Note**:*
- *Motorola SPI and Texas Instruments SPI are full-duplex protocols*
- *National Semiconductors SPI operates in half-duplex mode*
- *Full-duplex modes can also emulate half-duplex behavior using the following methods:*
  - *SCBx_TX_FIFO_CTRL.FREEZE can be used to transmit dummy data words*
  - *SCBx_RX_FIFO_CTRL.FREEZE can be used to ignore received data words*
- *The MOSI signal can be set to a High-Z state using the IOSS/GPIO configuration*

24.3.3 SPI modes of operation

24.3.3.1 Motorola SPI

The original SPI protocol was defined by Motorola. It is a full-duplex protocol. Multiple data transfers can occur while the SELECT line is held at '0'. As a result, slave devices must track the progress of data transfers to distinguish individual data frames. When no data is being transmitted, the SELECT line remains at '1', and SCLK is typically pulled low.

Clock modes of Motorola SPI

The Motorola SPI protocol supports four different clock modes, which depend 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 data is not being transmitted:
  - 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
- **Clock phase**: Determines when data is driven and captured:
  - SCBx_SPI_CTRL.CPHA = 0 means data is sampled (captured) on the leading (first) clock edge
  - SCBx_SPI_CTRL.CPHA = 1 means data is sampled on the trailing (second) clock edge, regardless of whether the clock edge is rising or falling
  - With SCBx_SPI_CTRL.CPHA = 0, data must be stable for the setup time before the first clock cycle

The four clock modes, including their specific configurations, are as follows:

1. **Mode 0**:
   - SCBx_SPI_CTRL.CPOL = 0, SCBx_SPI_CTRL.CPHA = 0
   - Data is driven on the falling edge of SCLK
   - Data is captured on the rising edge of SCLK
   - SCLK idle state is 0
2. **Mode 1**:
   - SCBx_SPI_CTRL.CPOL = 0, SCBx_SPI_CTRL.CPHA = 1
   - Data is driven on the rising edge of SCLK
   - Data is captured on the falling edge of SCLK
   - SCLK idle state is 0
3. **Mode 2**:
   - SCBx_SPI_CTRL.CPOL = 1, SCBx_SPI_CTRL.CPHA = 0
   - Data is driven on the rising edge of SCLK
   - Data is captured on the falling edge of SCLK
   - SCLK idle state is 1
4. **Mode 3**:
   - SCBx_SPI_CTRL.CPOL = 1, SCBx_SPI_CTRL.CPHA = 1
   - Data is driven on the falling edge of SCLK

- Data is captured on the rising edge of SCLK
- SCLK idle state is 1

Figure 240 illustrates how MOSI/MISO data is driven and captured as a function of SCBx_SPI_CTRL.CPOL and SCBx_SPI_CTRL.CPHA.

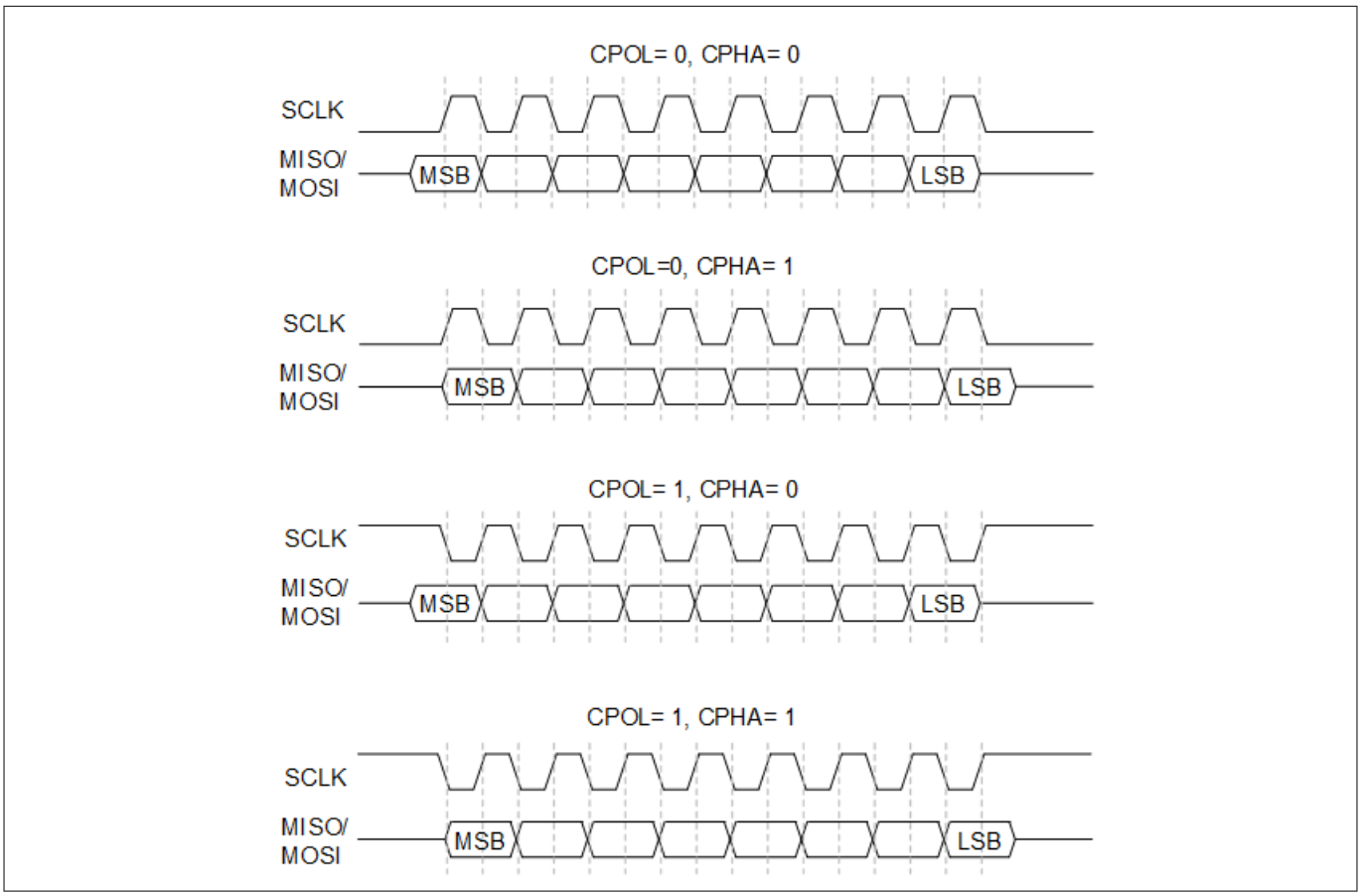


Figure 240 SPI Motorola clock modes

Figure 241 illustrates a single 8-bit data transfer and two successive 8-bit data transfers in Mode 0 (SCBx_SPI_CTRL.CPOL = 0, SCBx_SPI_CTRL.CPHA = 0).

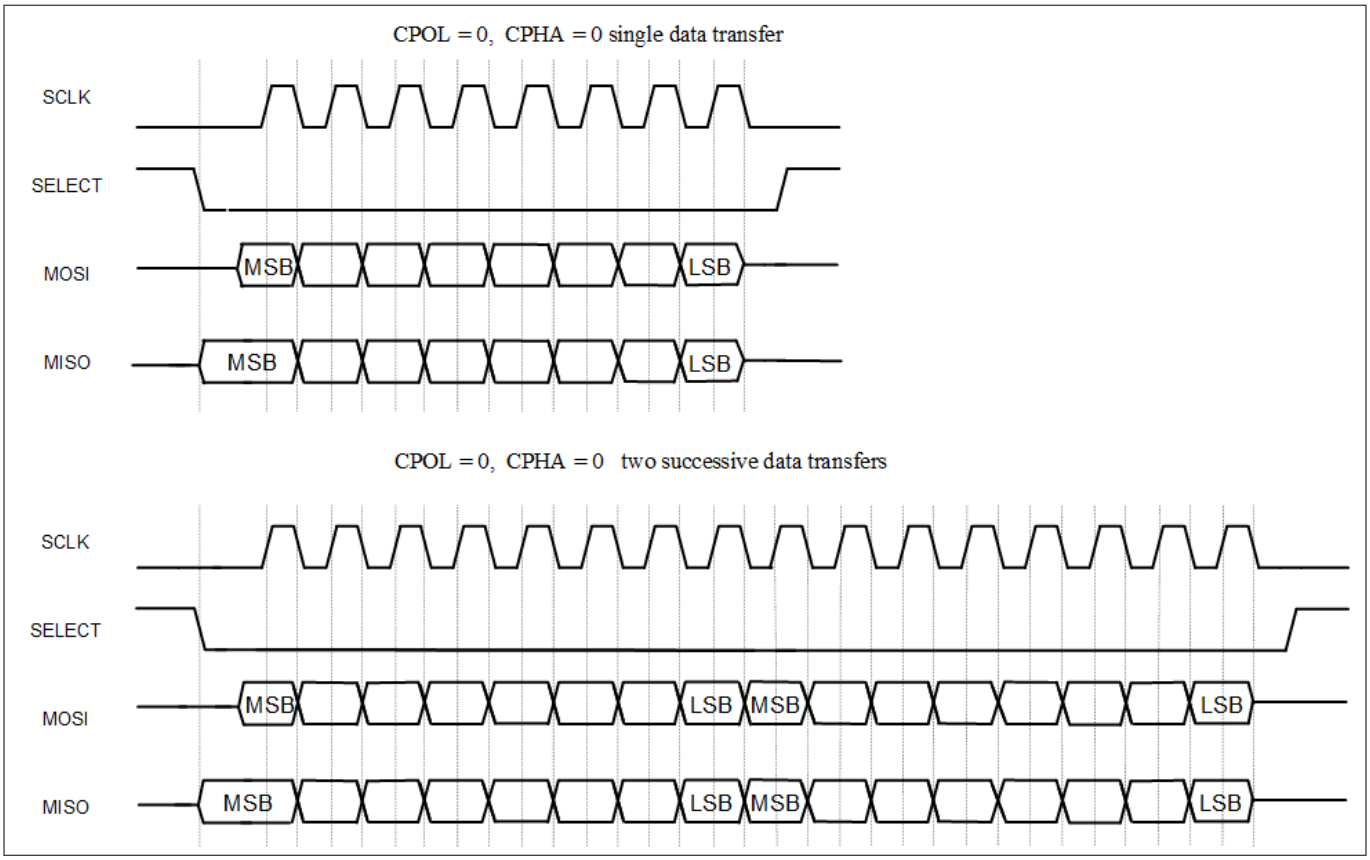


Figure 241 SPI Motorola data transfer example

Configuring SCB for SPI Motorola mode

To configure the SCB for SPI Motorola mode, do the following:

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. Configure the mode of operation in Motorola SPI by writing to the `SCBx_SPI_CTRL.CPHA` and `SCBx_SPI_CTRL.CPOL` registers
4. Follow 2. through 4. in the Enabling and initializing SPI section

For detailed information on these registers, see the CYW20829 registers reference manual.

24.3.3.2 Texas Instruments SPI

The Texas Instruments SPI protocol redefines the use of the SELECT signal. Unlike the Motorola SPI protocol, which uses a low-active SELECT signal to indicate the active state of slave devices, the TI SPI protocol uses the SELECT signal to indicate the start of a data transfer.

As a result, slave devices do not need to track the progress of data transfers to separate individual data frames. The start of a transfer is denoted by a high-active pulse lasting for a single-bit transfer period. This pulse can occur either one clock cycle before the transmission of the first data bit or coincide with the transmission of the first data bit.

The TI SPI protocol operates exclusively in Mode 1 (`SCBx_SPI_CTRL.CPOL = 0` and `SCBx_SPI_CTRL.CPHA = 1`). In this mode:

- Data is driven on the rising edge of `SCLK`
- Data is captured on the falling edge of `SCLK`

Figure 242 illustrates a single 8-bit data transfer and two successive 8-bit data transfers where:

- The SELECT pulse occurs before the transmission of the first data bit of the initial data transfer
- The SELECT pulse of the second data transfer coincides with the last data bit of the first data transfer

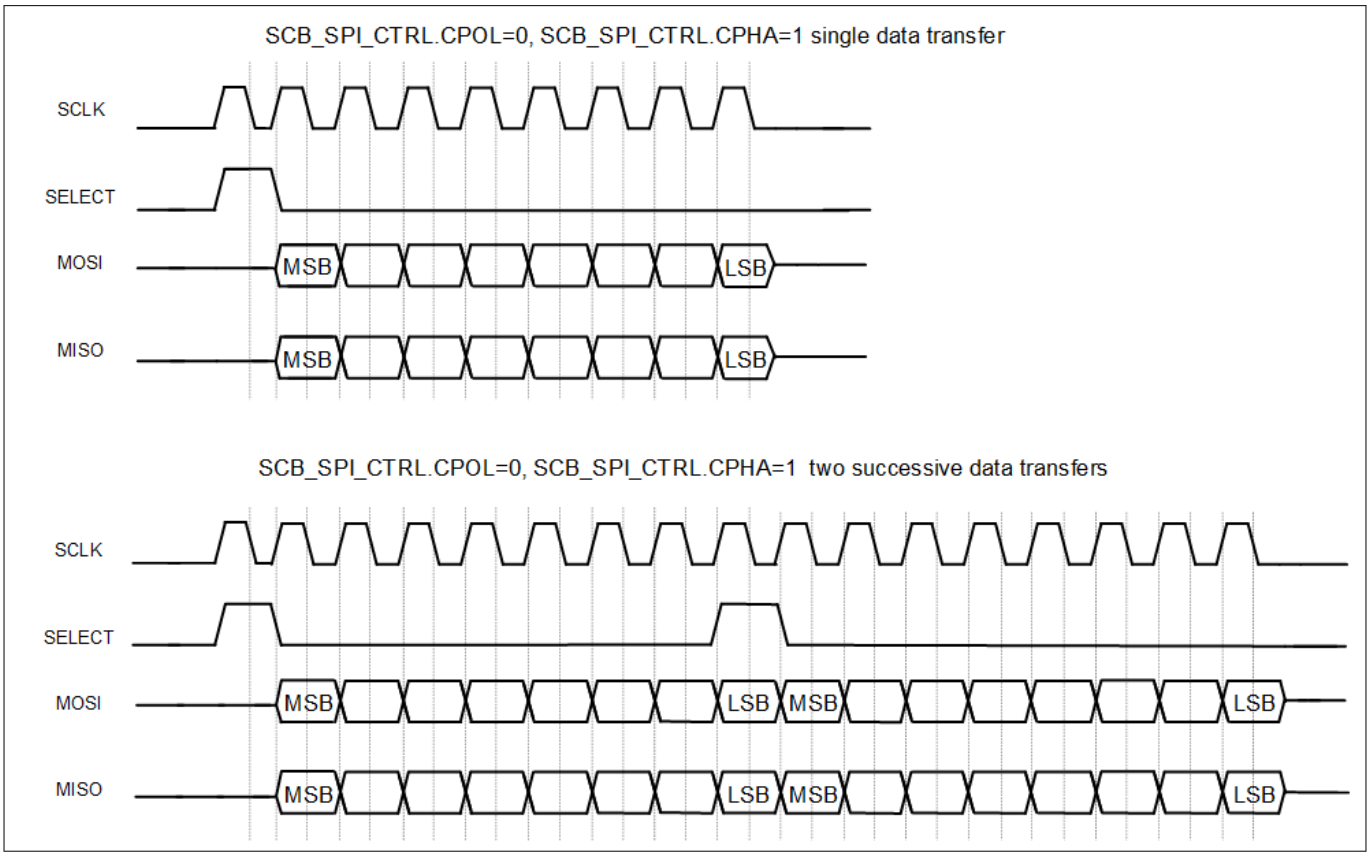


Figure 242 SPI TI data transfer example

Figure 243 illustrates a single 8-bit data transfer as well as two successive 8-bit data transfers. In this example, the SELECT pulse coincides with the first data bit of each frame.

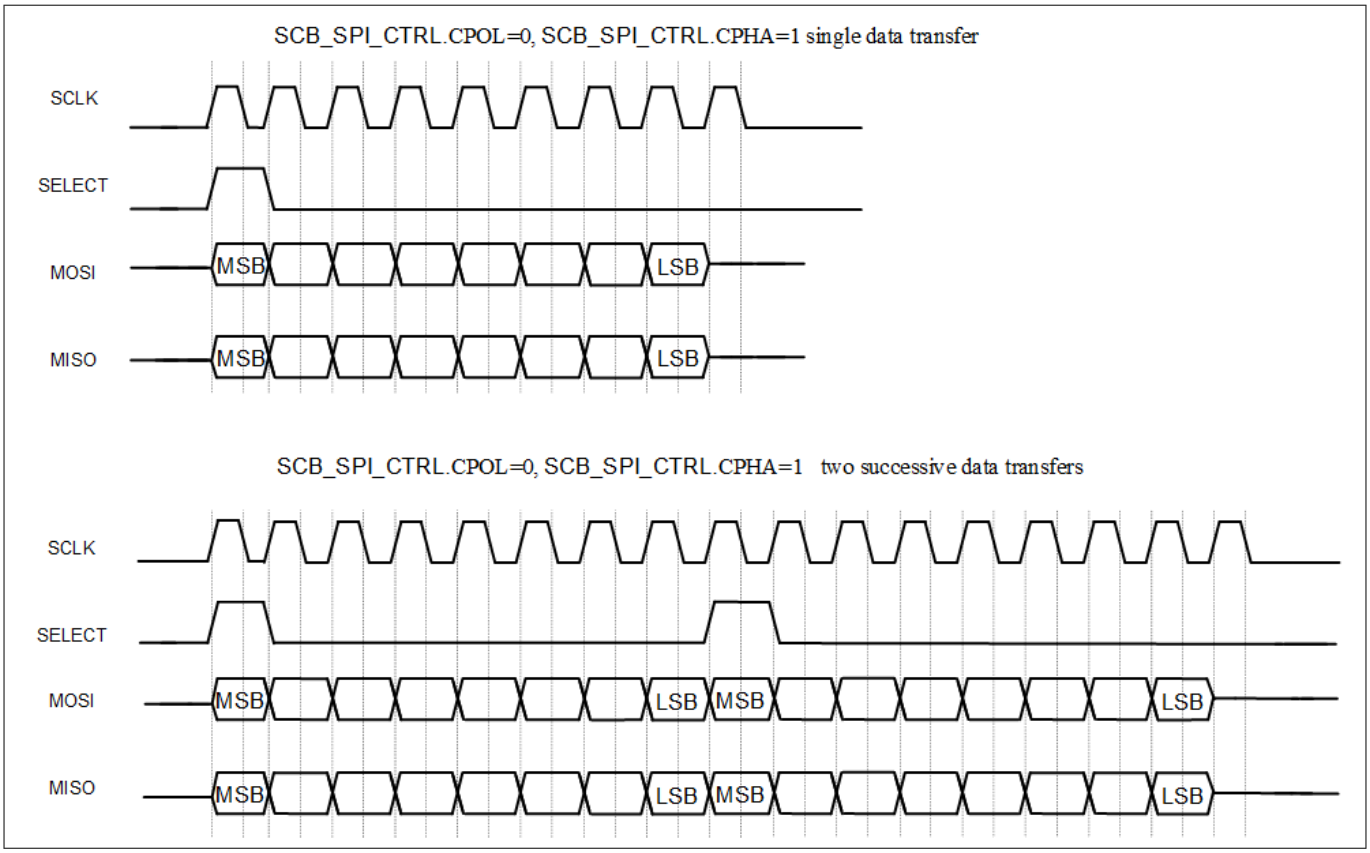


Figure 243 SPI TI data transfer example 2

Configuring SCB for SPI TI mode

To configure the SCB for SPI TI mode, follow these steps:

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. Configure the mode of operation in TI by writing to the `SCBx_SPI_CTRL.SELECT_PRECEDE` register:
   a. Write `1` to configure the SELECT pulse to precede the first bit of the next frame
   b. Write `0` if the SELECT pulse does not precede the first bit of the next frame
4. Follow 2. through 4. in the Enabling and initializing SPI section

For detailed information on these registers, see the CYW20829 registers reference manual.

24.3.3.3 National Semiconductors SPI

The National Semiconductor SPI protocol is a half-duplex protocol. In this protocol, transmission and reception do not occur simultaneously but take turns, with transmission occurring before reception. The sizes of the transmitted and received data may differ. A single idle (`0`) bit transfer period separates transmission from reception. However, successive data transfers are not separated by an idle bit transfer period.

The National Semiconductor SPI protocol supports only Mode 0.

The following figure illustrates a single data transfer and two successive data transfers. In each case, the transmission data transfer size is eight bits, and the reception data transfer size is four bits.

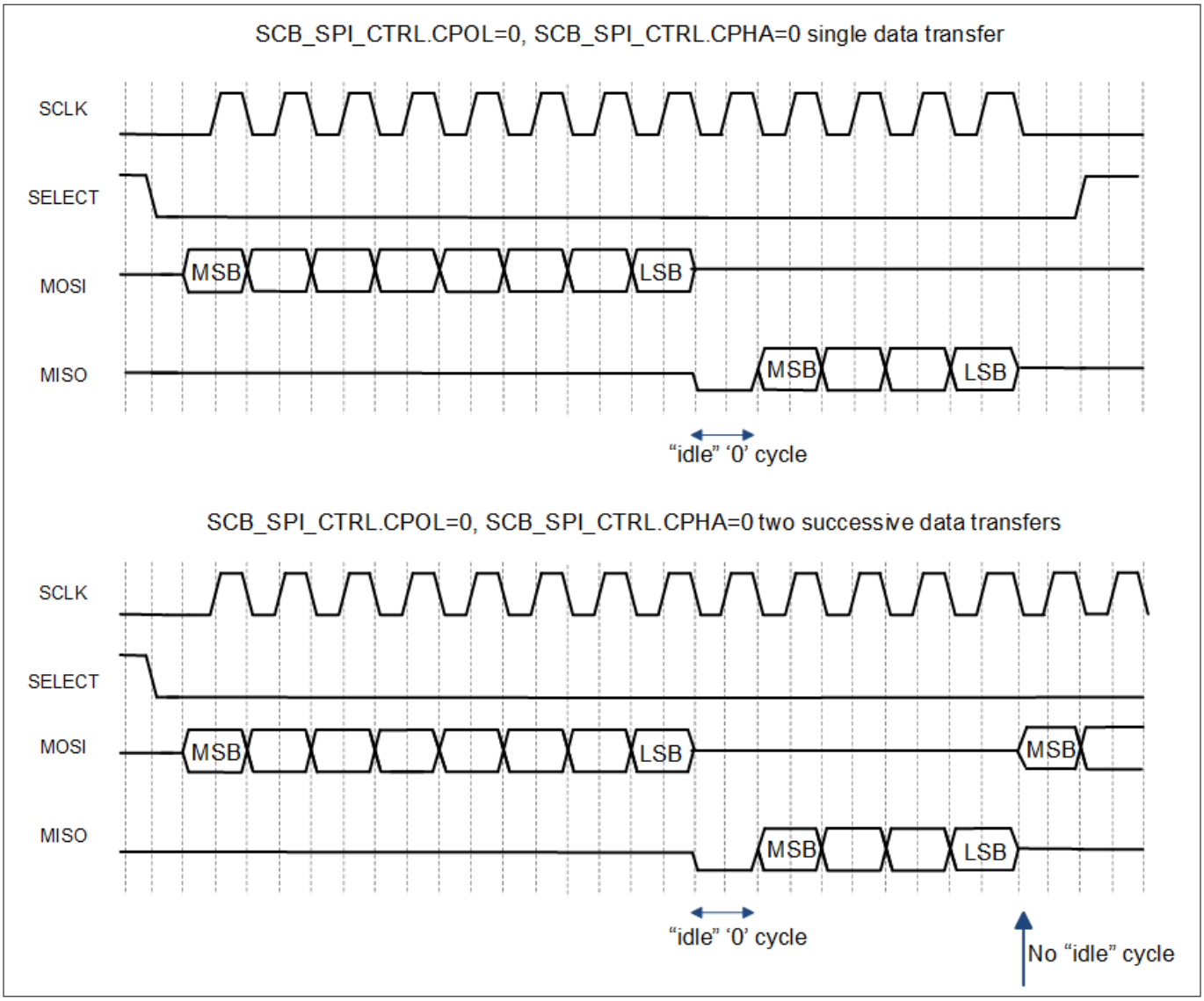


Figure 244 SPI NS data transfer example

Configuring SCB for SPI NS mode

To configure the SCB for SPI NS (National Semiconductor) mode, follow these steps:

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 clock settings:
   a. Write `0` to the `SCBx_SPI_CTRL.CPOL` bit
   b. Write `0` to the `SCBx_SPI_CTRL.CPHA` bit
4. Follow 2. through 4. in the Enabling and initializing SPI section

For detailed information on these registers, see the CYW20829 registers reference manual.

24.3.4 SPI buffer modes

SPI can operate in three different buffer modes:

- FIFO mode
- EZ mode
- CMD_RESP mode

The buffer is used differently in each of these modes. The following subsections provide detailed explanations of each buffer mode.

24.3.4.1 FIFO mode

The FIFO mode provides a Tx FIFO for data being transmitted and an Rx FIFO for data being received. Each FIFO is constructed from the SRAM buffer. Depending on the configuration, the FIFOs can have the following depths:

- 32 elements deep for 32-bit data elements
- 64 elements deep for 16-bit data elements
- 128 elements deep for 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 Active and Sleep power modes, except on Deep Sleep-capable SCBs, where it is also available in Deep Sleep mode.

Tx and Rx FIFO access:

- The Tx FIFO supports write access using the SCBx_TX_FIFO_WR register
- The Rx FIFO supports read access using the SCBx_RX_FIFO_RD register

FIFO status and trigger control:

- The Tx and Rx FIFO status information is accessible through the SCBx_TX_FIFO_STATUS and SCBx_RX_FIFO_STATUS registers
- A programmable threshold can be defined to trigger events based on the number of FIFO entries:
  1. **Tx FIFO trigger**:
     - The SCBx_TX_FIFO_CTRL.TRIGGER_LEVEL defines the threshold
     - An event is generated when the number of entries in the Tx FIFO is less than the SCBx_TX_FIFO_CTRL.TRIGGER_LEVEL
  2. **Rx FIFO trigger**:
     - The SCBx_RX_FIFO_CTRL.TRIGGER_LEVEL defines the threshold
     - An event is generated when the number of entries in the Rx FIFO is greater than the SCBx_RX_FIFO_CTRL.TRIGGER_LEVEL

These triggers can be connected to a DMA channel for streamlined data handling. The SCBx_INTR_TX and SCBx_INTR_RX registers provide numerous interrupt status bits for both the Tx and Rx FIFOs.

Deep Sleep to Active transition

When SCBx_CTRL.EC_AM_MODE = 1 and SCBx_CTRL.EC_OP_MODE = 0, the FIFO mode behaves as follows:

- The MISO line transmits 0xFF until the internally-clocked logic takes control, and the CPU writes to the Tx FIFO
- Data on the MOSI line is ignored until the internally-clocked logic takes over
- Once the internally-clocked logic takes control, the internal clock may not initially operate at the correct frequency due to PLL/FLL locking times. This can result in corrupted data within the Rx FIFO

To address this issue, it is recommended to:

1. Clear the Rx FIFO before writing new data into the Tx FIFO after transitioning from Deep Sleep to Active mode
2. Alternatively, disable CLK_SCB before entering Deep Sleep and enable it only after the PLL and FLL have stabilized

The external master must be aware that receiving 0xFF on the MISO line indicates the device is not yet ready.

24.3.4.2 EZSPI mode

The EZSPI protocol defines a single memory buffer with an 8-bit EZ address that indexes a buffer on the slave device. This buffer consists of a 256-entry array, where each entry is eight bits wide. The EZ address is used to access these 256 memory locations. All EZSPI data transfers use 8-bit data frames.

The CPU interacts with the memory buffer by writing to or reading from the SCBx_EZ_DATA registers. These accesses are word-access operations, but only the least significant byte of the word is used.

EZSPI supports three types of transfers:

1. Write the EZ address from the master to the slave device
2. Write data from the master to the memory location addressed on the slave device
3. Read data by the master from an addressed memory location on the slave device

***Note**: When multiple bytes are read or written, the master must hold SELECT low throughout the entire transfer.*

EZ address write

A write of the EZ address begins with a command byte (0x00) transmitted on the MOSI line, indicating the master's intent to write the EZ address. During the reception of the command byte, the slave responds on the MISO line with either:

- 0xFE to indicate it is ready
- 0xFF to indicate it is busy

When the slave is disabled or reset, it transmits 0xFF (busy) on the MISO line. This byte enables the master to determine whether the slave is ready to accept the SPI request.

After the command byte, the slave drives a reply byte on the MISO line to indicate whether the command is acknowledged (0xFE) or not acknowledged (0xFF). The second byte transmitted on the MOSI line is the EZ address.

Memory array write

A write to a memory array index begins with a command byte (0x01) transmitted on the MOSI line, indicating the master's intent to write to the memory array. During the reception of the command byte, the slave responds on the MISO line with either:

- 0xFE to indicate it is ready
- 0xFF to indicate it is busy

When the slave is disabled or reset, it transmits 0xFF (busy) on the MISO line. This byte allows the master to determine whether the slave is ready to accept the SPI request.

After this, the slave transmits a reply byte on the MISO line to indicate the status of the command:

- 0xFE if the command was successfully registered
- 0xFF if the command was not registered

Any additional data bytes written on the MOSI line are stored in the memory array locations starting at the EZ address communicated earlier. The EZ address is automatically incremented by the slave as data bytes are written into the memory array.

- If the EZ address exceeds the maximum number of memory entries (256), it remains at the maximum address and does not wrap around to 0
- The EZ base address resets to the previously written EZ address during the EZ Address Write phase upon each slave selection

Memory array read

A read from a memory array index begins with a command byte (0x02) transmitted on the MOSI line, indicating the master's intent to read from the memory array. During the reception of the command byte, the slave responds on the MISO line with either:

- `0xFE` to indicate it is ready
- `0xFF` to indicate it is busy

When the slave is disabled or reset, it transmits `0xFF` (busy) on the MISO line. This byte allows the master to determine whether the slave is ready to accept the SPI request.

The slave then transmits a reply byte on the MISO line to indicate the status of the command:

- `0xFE` if the command was successfully registered
- `0xFF` if the command was not registered

Any additional data bytes read on the MISO line are fetched from the memory array locations starting at the EZ address communicated earlier. The EZ address is automatically incremented by the slave as data bytes are read from the memory array.

- If the EZ address exceeds the maximum number of memory entries (256), it remains at the maximum address and does not wrap around to `0`
- The EZ base address resets to the previously written EZ address during the EZ Address Write phase upon each slave selection

Figure 245 illustrates the write of the EZ address, write to a memory array, and read from a memory array operations in the EZSPI protocol.

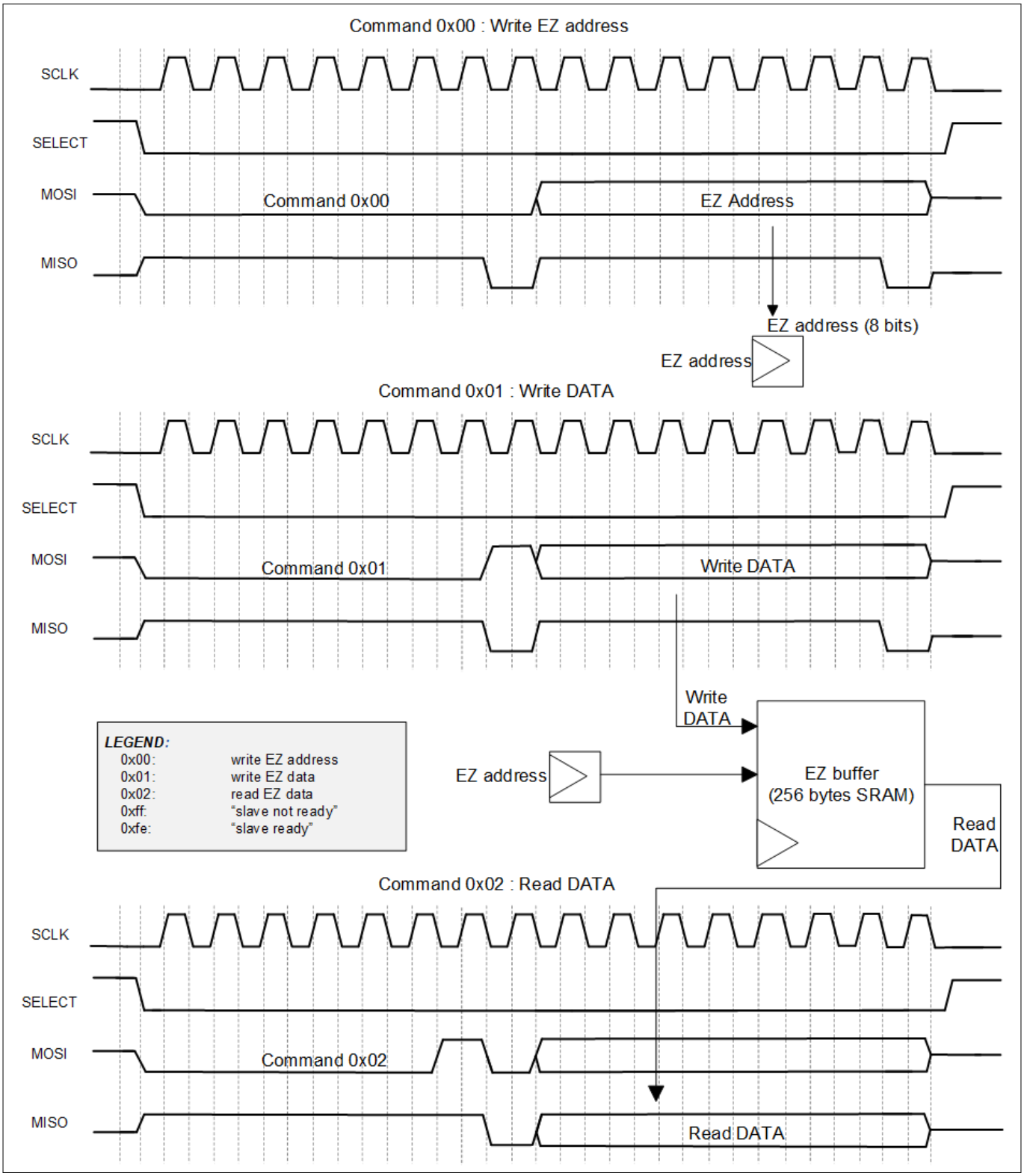


Figure 245 EZSPI example

Configuring SCB for EZSPI mode

By default, the SCB is configured for non-EZ mode operation. To configure the SCB for EZSPI mode, follow these steps:

1. Select EZ mode by writing `1` to the `SCBx_CTRL.EZ_MODE` register

2. Follow the steps outlined in the Motorola SPI section
3. Follow 2. through 4. in the Enabling and initializing SPI section

For detailed information on these registers, see the CYW20829 registers reference manual.

Deep Sleep to Active transition

1. `EC_AM_MODE = 1`, `SCBx_CTRL.EC_OP_MODE = 0`, **EZ mode**:
   - The MISO line transmits `0xFF` until the internally-clocked logic takes control
   - Data on the MOSI line is ignored until the internally-clocked logic takes over
   - When the internally-clocked logic becomes active, there is no guarantee that the internal clock will immediately run at the correct frequency due to PLL/FLL locking times. This may result in corrupted data on the MISO line and in the EZ memory
   - **Recommendation**: Before entering Deep Sleep, disable `CLK_SCB`. Re-enable it only after the PLL/FLL outputs have stabilized
   - The external master must recognize that a `0xFF` reading on MISO indicates that the device is not yet ready
2. `EC_AM_MODE = 1`, `SCBx_CTRL.EC_OP_MODE = 1`, **EZ mode**:
   - During the transition from Deep Sleep to Active mode, there is no guarantee that the internal clock will operate at the correct frequency due to PLL/FLL locking times
   - As a result, the SPI SCLK frequency is limited to 2 MHz during this phase
   - Once the PLL/FLL outputs stabilize, the clock can safely operate at higher frequencies

***Note**: Only SCB[0] supports Deep Sleep in slave mode.*

24.3.4.3 Command-response mode

The CPU writes to and reads from the memory buffer through the `SCBx_EZ_DATA` registers. While these accesses are word-based, only the least significant byte of each word is used.

Memory buffer addressing:

- The slave interface accesses the memory buffer using current addresses:
  1. At the start of a write transfer (SPI slave selection), the base write address is copied to the current write address
     - Data written by the master is stored at the current write address location
     - After each write access, the current write address is incremented by `1`
  2. At the start of a read transfer, the base read address is copied to the current read address
     - Data transmitted to the master is fetched from the current read address location
     - After each read access, the current read address is incremented by `1`
- If the current addresses reach the last memory buffer address (`255`):
  - The current addresses do not increment further
  - Subsequent writes overwrite the value at the last buffer address
  - Subsequent reads repeatedly provide the value at the last buffer address
  - The bus master should be aware of the memory buffer capacity in command-response mode

Address and status information:

- The base addresses are configured through:
  - `SCBx_CMD_RESP_CTRL.BASE_RD_ADDR` for the base read address
  - `SCBx_CMD_RESP_CTRL.BASE_WR_ADDR` for the base write address

- The current addresses are available through:
  - SCBx_CMD_RESP_STATUS.CURR_RD_ADDR for the current read address
  - SCBx_CMD_RESP_STATUS.CURR_WR_ADDR for the current write address
- At the end of a transfer (SPI slave de-selection), the difference between the base address and the current address indicates the number of read/write operations performed

The block includes interrupt cause fields to identify the end of a transfer. Command-response mode is supported in Active and Deep Sleep power modes.

The command-response mode has two phases of operation:

1. **Write phase**:
   - Begins with a selection byte, where the last bit is set to 0 (indicating a write operation)
   - The master writes 8-bit data elements to the slave's memory buffer after the selection byte
   - The slave:
     - Copies its base write address to the current write address
     - Stores the received data elements at the current write address locations
     - Increments the current write address after each memory write
2. **Read phase**:
   - Begins with a selection byte, where the last bit is set to 1 (indicating a read operation)
   - The master reads 8-bit data elements from the slave's memory buffer
   - The slave:
     - Sets its base read address to the current read address
     - Provides the data elements from the current read address locations
     - Increments the current read address after each read operation

Slave status during transfers:

- During the reception of the first byte:
  - The slave transmits 0x62 (ready) or another value (busy) on the MISO line
  - If the slave is disabled or reset, it transmits 0xFF (busy)
- The master uses the transmitted byte to determine if the slave is ready to accept the SPI request

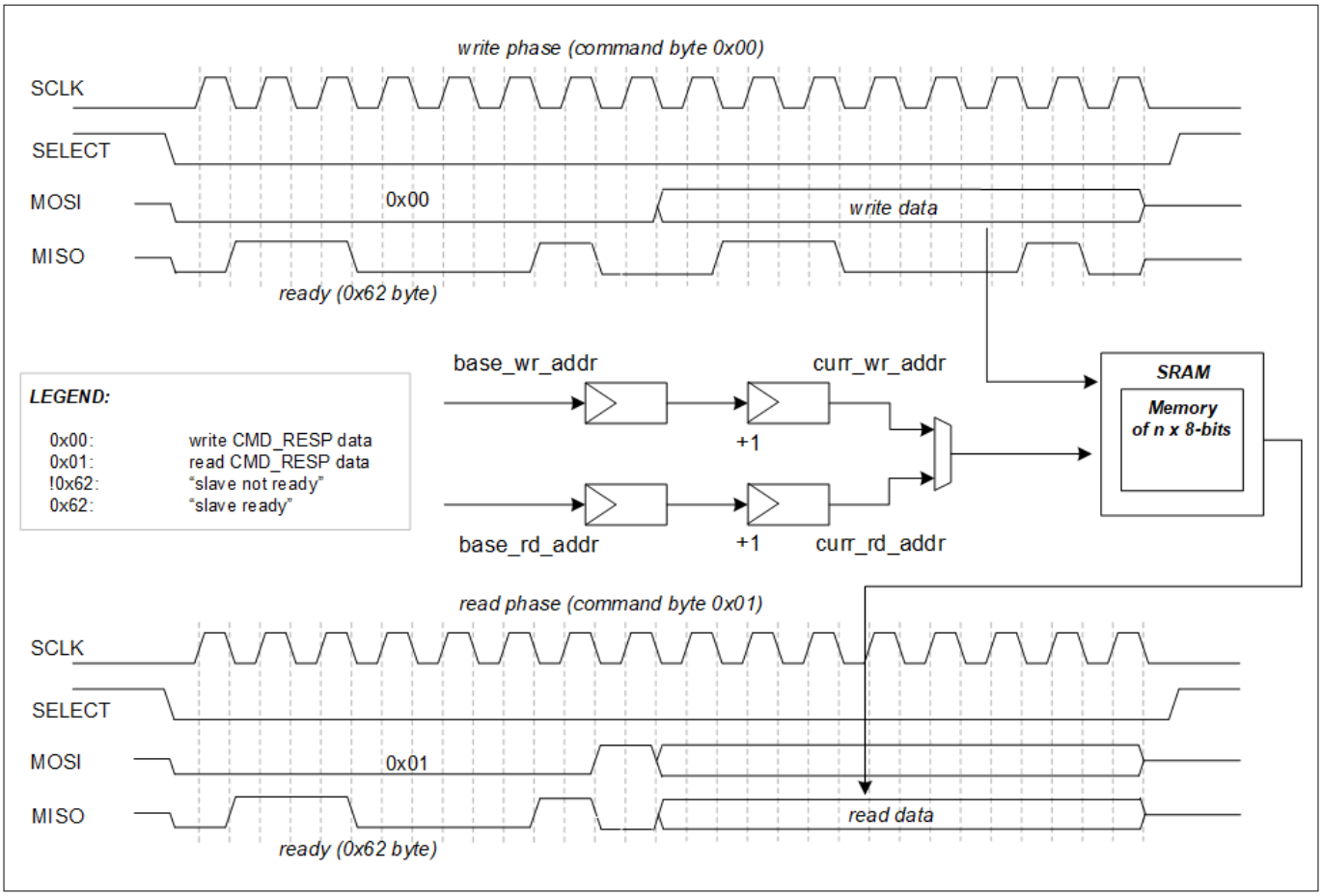


Figure 246 Command-response mode example

***Note**: A slave's base addresses are updated by the CPU, not by the master.*

Deep Sleep to Active transition

`SCBx_CTRL.EC_AM_MODE = 1, SCBx_CTRL.EC_OP_MODE = 1, CMD_RESP Mode`

When transitioning from Deep Sleep to Active mode, the internal clock may not operate at the correct frequency due to PLL/FLL locking times. During this phase, the SPI SCLK frequency is limited to 2 MHz. Once the FLL/PLL outputs have stabilized, the clock can operate at higher frequencies.

Configuring SCB for CMD_RESP mode

By default, the SCB is configured for non-CMD_RESP mode operation. To configure the SCB for CMD_RESP mode, follow these steps:

1. Select CMD_RESP mode by writing `1` to the `SCBx_CTRL.CMD_RESP_MODE` register
2. Follow the steps outlined in the Motorola SPI section
3. Follow 2. through 4. in the Enabling and initializing SPI section

For detailed information on these registers, see the CYW20829 registers reference manual.

24.3.5 Clocking and oversampling

24.3.5.1 Clock modes

The SCB SPI supports both internally-clocked and externally-clocked operation modes. The SCBx_CTRL.EC_AM_MODE and SCBx_CTRL.EC_OP_MODE registers determine the SCB clock configuration:

- **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)

Externally-clocked operation: Uses a clock provided by an external master (SPI SCLK)

Note: *In the CYW20829 MCU, only the Deep Sleep-capable SCB supports externally-clocked mode, and only for SPI slave mode.*

Internally-clocked operation:

- Uses programmable clock dividers. For more information on system clocking, refer to the Clocking system chapter

EC_AM_MODE = 0 and SCBx_CTRL.EC_OP_MODE = 0

Use this configuration for Active mode functionality only

- **FIFO mode**: Supported
- **EZ mode**: Supported
- **Command-response mode**: Not supported
  - The slave (MISO) transmits a value different from a ready (0x62) byte during the reception of the first byte if command-response mode is attempted in this configuration

EC_AM_MODE = 1 and SCBx_CTRL.EC_OP_MODE = 0

Use this configuration when both Active and Deep Sleep functionality are required

- Relies on externally-clocked functionality to detect slave selection
- Relies on internally-clocked functionality to access the memory buffer
- The “handover” from external to internal functionality uses a busy/ready byte scheme:
  - The master retries the current transfer if it receives a busy byte
  - The master must support busy/ready byte interpretation
- When the slave is selected, SCBx_INTR_SPI_EC.WAKE_UP is set to 1. This Deep Sleep functionality interrupt wakes the system into Active mode

Mode support:

- **FIFO mode**: Supported
  - The slave (MISO) transmits 0xFF until the CPU is woken up and the Tx FIFO is written
  - Data on the MOSI line is dropped until CLK_SCB is enabled (refer to Deep Sleep to Active Transition for more information)
- **EZ mode**: Supported
  - In Deep Sleep mode, the slave (MISO) transmits a busy (0xFF) byte during the reception of the command byte
  - In Active mode, the slave (MISO) transmits a ready (0xFE) byte during the reception of the command byte
- **CMD_RESP mode**: Not supported
  - The slave transmits (on MISO) a value different from a ready (0x62) byte during the reception of the first byte

EC_AM_MODE = 1 and SCBx_CTRL.EC_OP_MODE = 1

- Use this configuration when both Active and Deep Sleep functionality are required
  - When the slave is selected, SCBx_INTR_SPI_EC.WAKE_UP is set to 1. This Deep Sleep functionality interrupt wakes the system into Active 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

  Mode support:
  - **FIFO mode**: Not supported
  - **EZ mode**: Supported
  - **CMD_RESP mode**: Supported

Table 169 SPI modes 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 detects the slave selection and triggers an interrupt to wake up the CPU. Write operations are ignored, and read operations return 0xFF until the CPU is ready and the FIFO is populated.

If SCBx_CTRL.EC_OP_MODE is set to 1, the external interface logic accesses the memory buffer using the external interface clock (SPI SCLK). This configuration enables EZ and CMD_RESP mode functionality in both Active and Deep Sleep power modes.

Memory buffer arbitration in Active system power mode

In Active power mode, arbitration is required between the external interface logic (operating on SPI SCLK) and the CPU interface logic (operating on the system peripheral clock). The arbitration mechanism always gives the highest priority to the external interface logic (host accesses). The external interface logic takes two serial interface clock/bit periods for SPI. During this time, internal logic is temporarily denied access to the memory buffer.

Behavior based on SCBx_CTRL.BLOCK

- **If SCBx_CTRL.BLOCK = 1**:
  - Internal logic access to the memory buffer is blocked until the external interface logic completes its access and the memory buffer is granted
  - While this ensures normal SCB register functionality, the blocking introduces additional internal bus wait states
- **If SCBx_CTRL.BLOCK = 0**:
  - Internal logic access to the memory buffer is not blocked but fails when it conflicts with an external interface logic access
  - A read access failure:
    - Returns the value 0xFFFF:FFFF
    - This value is unique, as successful memory buffer reads return unsigned byte values in the range 0x0000:0000 to 0x0000:00FF
  - A write access failure: Can be detected by reading back the target memory buffer location to confirm that the read value matches the intended write value
  - No additional internal bus wait states are introduced, but accesses to the memory buffer may not take effect

Failure detection for conflicting accesses

For both BLOCK = 1 and BLOCK = 0, conflicting internal logic access to the memory buffer sets the following interrupt fields:

- SCBx_INTR_TX.BLOCKED = 1 for write access failures
- SCBx_INTR_RX.BLOCKED = 1 for read access failures

These fields can serve as status indicators or interrupt causes when their associated mask fields are enabled.

If a series of read or write operations is performed when SCBx_CTRL.BLOCK = 0, failures can be detected by:

1. Performing a logical OR operation on all read values and comparing the result to 0xFFFF:FFFF
2. Checking the SCBx_INTR_TX.BLOCKED and SCBx_INTR_RX.BLOCKED fields to confirm whether a failure occurred during the series of operations

24.3.5.2 Using SPI master to clock slave

In normal SPI master mode transmissions, the SCLK is generated only when the SCB is enabled and data is being transmitted. However, this behavior can be modified to always generate a clock on the SCLK line while the SCB is enabled.

This functionality is useful when the slave device relies on the SCLK for operations other than SPI communication. To enable continuous clock generation, write 1 to the SCBx_SPI_CTRL.SCLK_CONTINUOUS register.

24.3.5.3 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 using the internal clock. The internal clock runs at a higher frequency than the interface clock (SCLK), allowing the master to oversample its input signals (MISO).

- **Oversampling**:
  - The SCBx_CTRL.OVS register specifies the oversampling rate
  - The oversampling rate is calculated as: SCBx_CTRL.OVS value + 1
  - In SPI master mode:
    - When MISO is used, the valid range for oversampling is 4 to 16
    - When MISO is not used, the valid range is 2 to 16

Bit rate calculation: The bit rate is calculated using the formula:

```
Bit rate = Input clock/SCBx_CTRL.OVS
```

- For an input clock of 100 MHz:
  - The maximum bit rate is 25 Mbps when MISO is used
  - The maximum bit rate is 50 Mbps when MISO is not used

Note: *To use a bit rate of 50 Mbps, ensure the following parameters are configured to avoid potential metastability:*

- *CLK_SCB = 200 MHz*
- *OVS = 3*
- *MEDIAN = 1*

These values determine the fastest SCLK frequency the SCB hardware can support. However, they do not guarantee that the master can correctly receive data from the slave at these speeds.

Determining path delay for MISO: To ensure reliable communication, calculate the MISO path delay using the following equation:

```
1/2 * tSCLK ≥ tSCLK_PCB_D + tDSO + tMISO_PCB_D + tDSI
```

Where:

- `tSCLK` is the period of the SPI clock
- `tSCLK_PCB_D` is the delay of SCLK from the master to the slave on the PCB
- `tDSO` is the total internal slave delay (time from the SCLK edge at the slave pin to the MISO edge at the slave pin)
- `tMISO_PCB_D` is the delay of MISO from the slave to the master on the PCB
- `tDSI` is the master setup time

Key points to consider:

- Most slave datasheets list `tDSO` (this may have a different name; look for "MISO output valid after SCLK edge")
- Most master datasheets include `tDSI` or "master setup time"
- Calculate `tSCLK_PCB_D` and `tMISO_PCB_D` based on the specific PCB geometries

If the desired speed is not achieved after performing the calculations, consider using the MISO late sample feature of the SCB. This feature instructs the SCB to sample the incoming MISO signal on the next SCLK edge, providing an additional one-half cycle of timing margin.

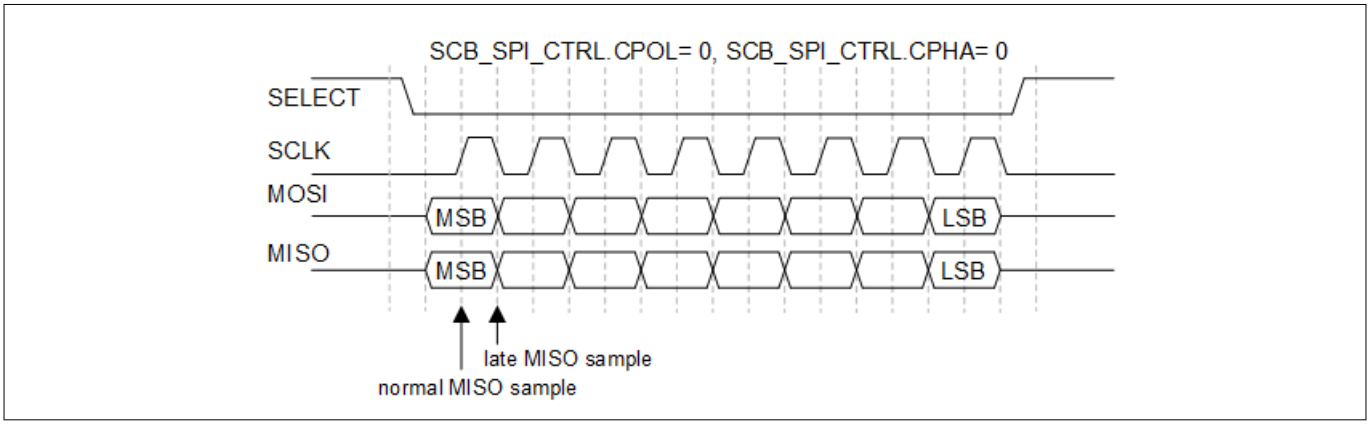


Figure 247 MISO sampling timing

The equation changes to:

```
tSCLK ≥ tSCLK_PCB_D + tDSO + tMISO_PCB_D + tDSI
```

Since the late sample feature improves timing, it is recommended to leave it enabled at all times. The `tDSI` specification in the device datasheet assumes that the late sample is enabled.

***Note**: The `SCBx_SPI_CTRL.LATE_MISO_SAMPLE` register is set to 1 by default.*

SPI slave mode

In 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 dictated by the external master and is not controlled by the `SCBx_SPI_CTRL.LATE_MISO_SAMPLE` setting.

In the internally-clocked mode, the timing is proportional to the frequency of the internal clock. For example:

```
TDSO_EXT = TDSO_INT + 3 * Tclk_scb
```

Where:

- `TDSO_EXT` is the output data valid time after the SCLK driving edge (external SPI master clock)
- `TDSO_INT` applies when the external SPI master does not use the late-sample feature

***Note**: To use a bit rate of 50 Mbps, set the following parameters to ensure metastability is avoided:*

- *`CLK_SCB = 200 MHz`*
- *`OVS = 4`*
- *`MEDIAN = 1`*

24.3.6 SPI master SELECT output timing control

The SPI master SELECT output signal's timing parameters are variable and apply to the following:

- **SELECT setup time**: The time from the SELECT signal becoming active to the start of the SPI clock
- **SELECT hold time**: The time from the SPI clock's last edge to the SELECT signal becoming inactive
- **Inter-data frame deselect time**: The time from the SELECT signal becoming inactive to it becoming active again

The following options are available for configuring these delays:

- **SELECT setup time**:
  - **When `SCBx_SPI_CTRL.CPHA = 0`:**
    - 0.75 SPI clock cycles
    - 1.75 SPI clock cycles (for SPI slaves requiring a longer setup time)
  - **When `SCBx_SPI_CTRL.CPHA = 1`:**
    - 0.25 SPI clock cycles
    - 1.25 SPI clock cycles (for SPI slaves requiring a longer setup time)
- **SELECT hold time**:
  - Configured using the `SCBx_SPI_CTRL.SSEL_HOLD_DEL` register
  - **When `SCBx_SPI_CTRL.CPHA = 0`:**
    - 0.25 SPI clock cycles
    - 1.25 SPI clock cycles (for SPI slaves requiring a longer hold time)
  - **When `SCBx_SPI_CTRL.CPHA = 1`:**
    - 0.75 SPI clock cycles
    - 1.75 SPI clock cycles (for SPI slaves requiring a longer hold time)

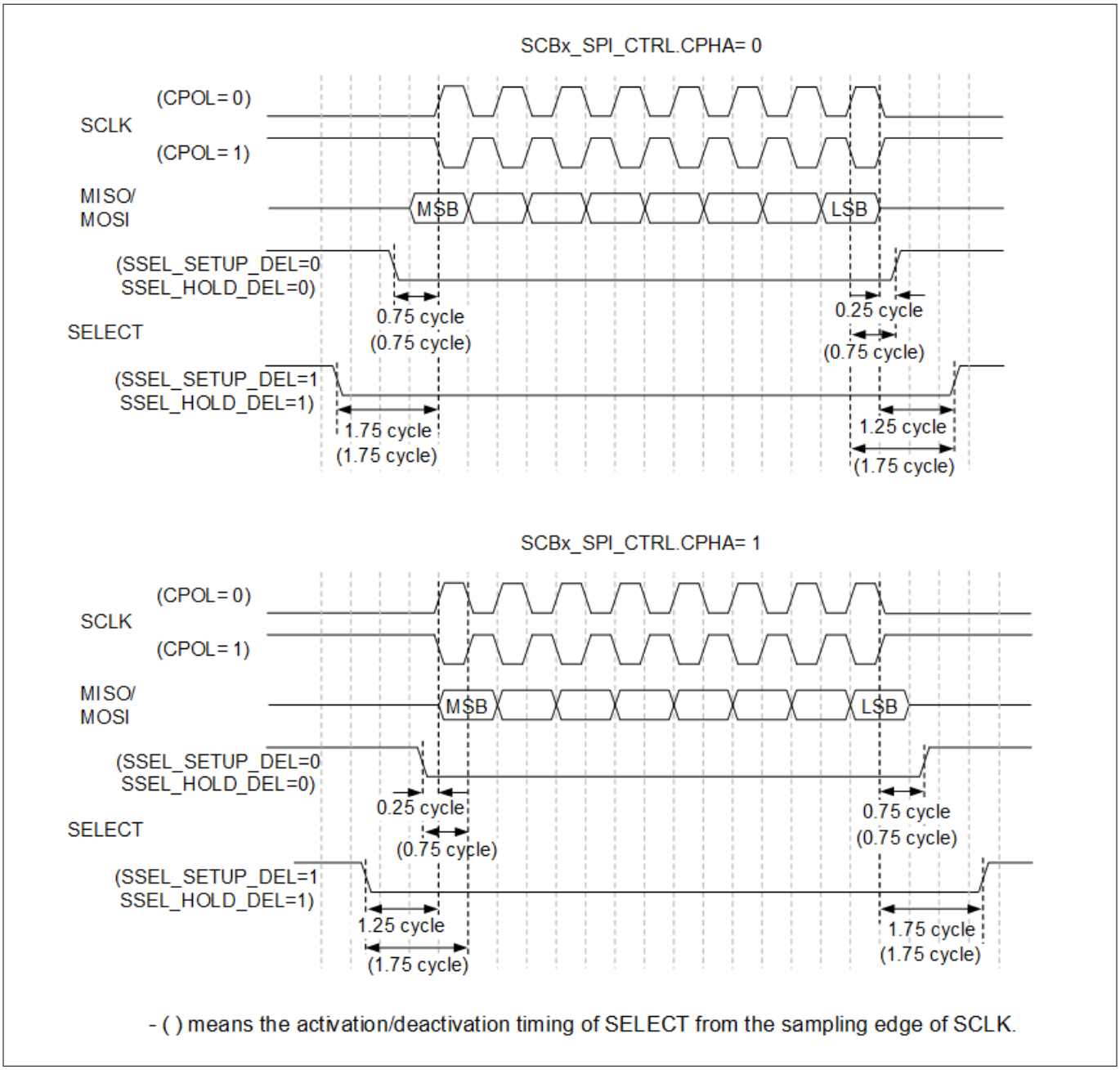


Figure 248 SELECT setup/hold delay

INTER-FRAME deselect time is configurable through the `SCBx_SPI_CTRL.SSEL_INTER_FRAME_DEL` register and can be set to the following:

- 1.5 SPI clock cycles
- 2.5 SPI clock cycles (for SPI slaves requiring a longer deselect time)

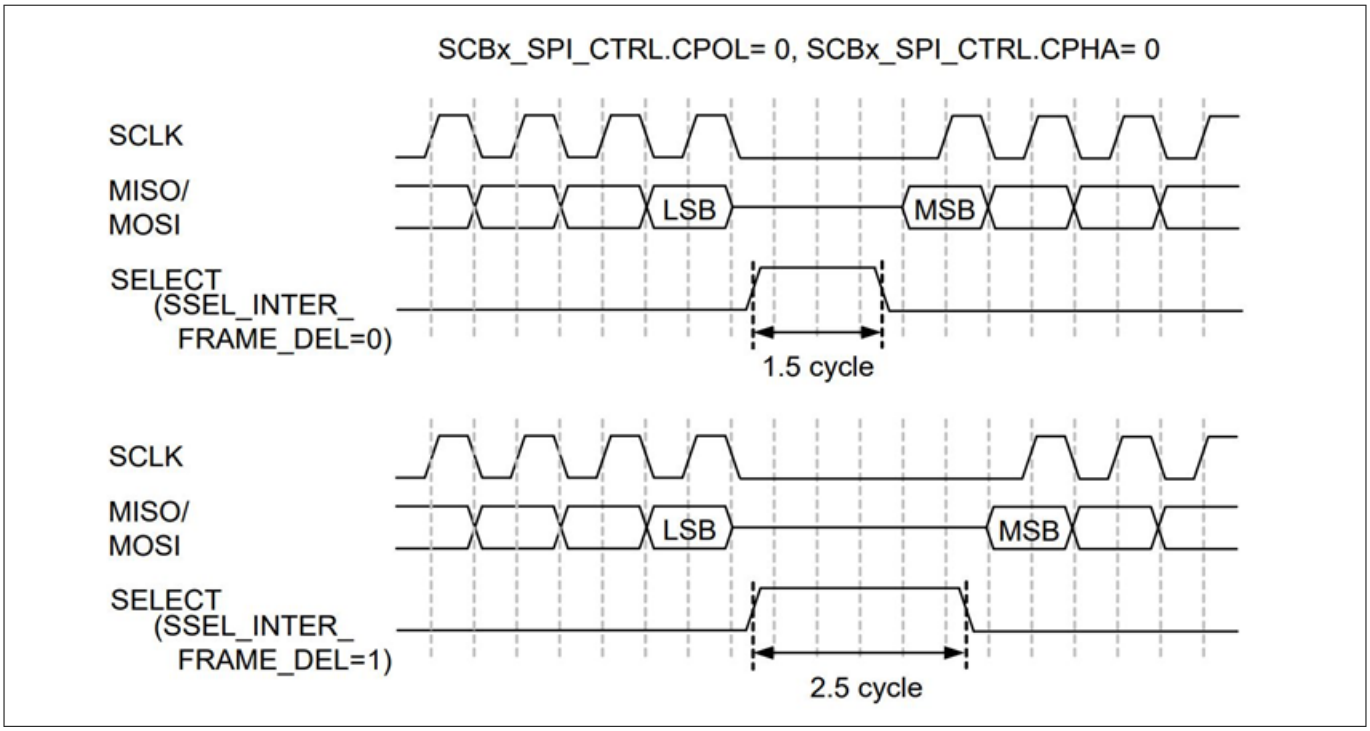


Figure 249 SELECT INTER-FRAME deselect time

24.3.7 SPI parity functionality

The features of the parity functionality in SPI mode are as follows:

- Applies to both SPI master and SPI slave modes in internally-clocked operation
- Adds a parity bit to the data frame, which is used to identify single-bit data frame errors. The parity bit directly follows the data frame bits
- Can be enabled or disabled using the SCBx_SPI_TX_CTRL.PARITY_ENABLED and SCBx_SPI_RX_CTRL.PARITY_ENABLED registers independently
- When transmitting, a parity bit can be inserted. When receiving, the parity bit can be checked. If the parity check fails:
  - The received data can either be sent to the Rx FIFO or dropped, based on the DROP_ON_PARITY_ERROR register
  - The PARITY_ERROR bit in the SCBx_INTR_RX register is set
- Supports even and odd parity, configured through the SCBx_SPI_TX_CTRL.PARITY and SCBx_SPI_RX_CTRL.PARITY registers

24.3.8 Loopback

In SPI Master mode, the SCB supports internal loopback functionality, which connects the MOSI output signal to the MISO input signal without affecting the data on the physical pins. This functionality is configured using the SCBx_SPI_CTRL.LOOPBACK register.

24.3.9 Enabling and initializing SPI

To program the SPI, do the following:

1. Program protocol-specific information using the SCBx_SPI_CTRL register:
   a. Select the sub-modes of the protocol (MODE)

   b. Configure master-slave functionality (MASTER_MODE)
   c. Select one of the four chip-select lines (SSEL)
   d. Specify whether the SELECT line stays active for an entire transfer or only for each data frame width (SSEL_CONTINUOUS)
   e. Set the SELECT polarity (SSEL_POLARITY0-3)
   f. Note that EZSPI and CMD_RESP modes can only be used in Slave mode
2. Program generic transmitter and receiver information using the SCBx_TX_CTRL and SCBx_RX_CTRL registers:
   a. Specify the data frame width, which must always be 8 for EZSPI and CMD_RESP
   b. Specify whether the most significant bit (MSb) or least significant bit (LSb) is the first to be transmitted or received. For EZSPI and CMD_RESP, it must always be MSb first
3. Program the transmitter and receiver FIFOs using the SCBx_TX_FIFO_CTRL and SCBx_RX_FIFO_CTRL registers:
   a. Set the trigger level (TRIGGER_LEVEL)
   b. Clear the transmitter and receiver FIFO and shift registers (CLEAR)
   c. Enable these settings only in FIFO mode
4. Enable the SPI block by writing 1 to the SCBx_CTRL.ENABLED register:
   a. After enabling the block, do not modify the control bits. To change settings, such as the operation mode (for example, switching from Motorola mode to TI mode) or switching from externally-clocked to internally-clocked operation, first disable the block, make the changes, and then re-enable the block
   b. Re-enabling the block causes reinitialization, which results in the loss of the associated state (for example, FIFO contents)

24.3.10 I/O pad connection

24.3.10.1 SPI master

In SPI master mode, the SCB provides both data transmit and data receive functionality. Figure 250 and Table 170 show 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 the SCB is disabled, the respective out_en signals are set to 0, causing the output to enter the High-Z state. To avoid the High-Z state, do one of the following:

- Use GPIO to drive the output to an idle level
- Use pull-up or pull-down drive modes with an internal pull-up or pull-down resistor (with a fixed resistance value)
- Use strong drive mode with an external pull-up or pull-down resistor (to allow a flexible resistance value)

***Note**: Both internal and external pull-up/pull-down resistors negatively affect the maximum data rate.*

MISO pull-up in full-duplex mode: For the MISO input in normal full-duplex mode:

- When the SPI slave device is not selected, its MISO output will be in a High-Z state
- If no SPI slave devices connected to the SPI master are selected, the MISO line will remain in a High-Z state

A pull-up resistor is required on the MISO line to avoid the High-Z state. Available options for configuring the MISO line include:

- **High-impedance (High-Z) drive mode**: With an external pull-up or pull-down resistor
- **Pull-up or pull-down drive mode**: With an internal pull-up or pull-down resistor

***Note**: Both internal and external pull-up/pull-down resistors negatively affect the maximum data rate.*

Half-duplex mode is not supported because it requires driving both "strong pull-up" and "strong pull-down" in the normal functional mode when `spi_mosi_out_en = 1`.

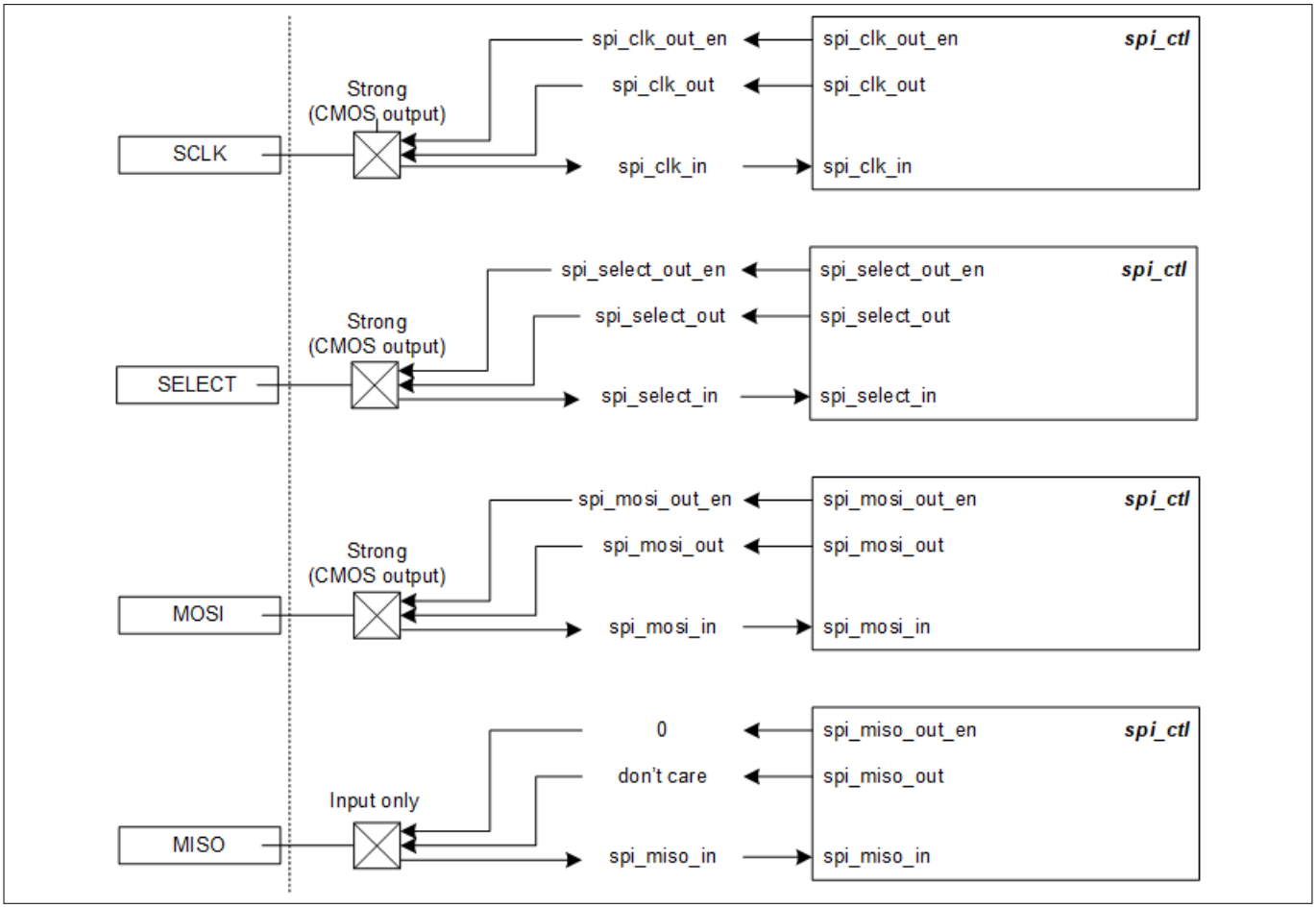


Figure 250 SPI master I/O pad connection usage

Table 170 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

24.3.10.2 SPI slave

In SPI slave mode, the SCB provides both data transmit and data receive functionality. Figure 13 and Table 3 show 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 the SCB is disabled, the respective out_en signals are set to 0, causing the output to enter the High-Z state. To avoid this High-Z state, do one of the following:

- Use GPIO to drive the output to an idle level
- Use pull-up or pull-down drive modes with an internal pull-up or pull-down resistor (fixed resistance value)
- Use strong drive mode with an external pull-up or pull-down resistor (for a flexible resistance value)

***Note**: Both internal and external pull-up/pull-down resistors negatively affect the maximum data rate.*

Half-duplex mode and open drain configuration

- When SCBx_TX_CTRL.OPEN_DRAIN = 1, the MOSI and MISO lines can be shorted to operate in half-duplex mode
- The drive mode of the MISO output must be configured as open drain drives low only
- An external pull-up resistor must be added to the MISO line

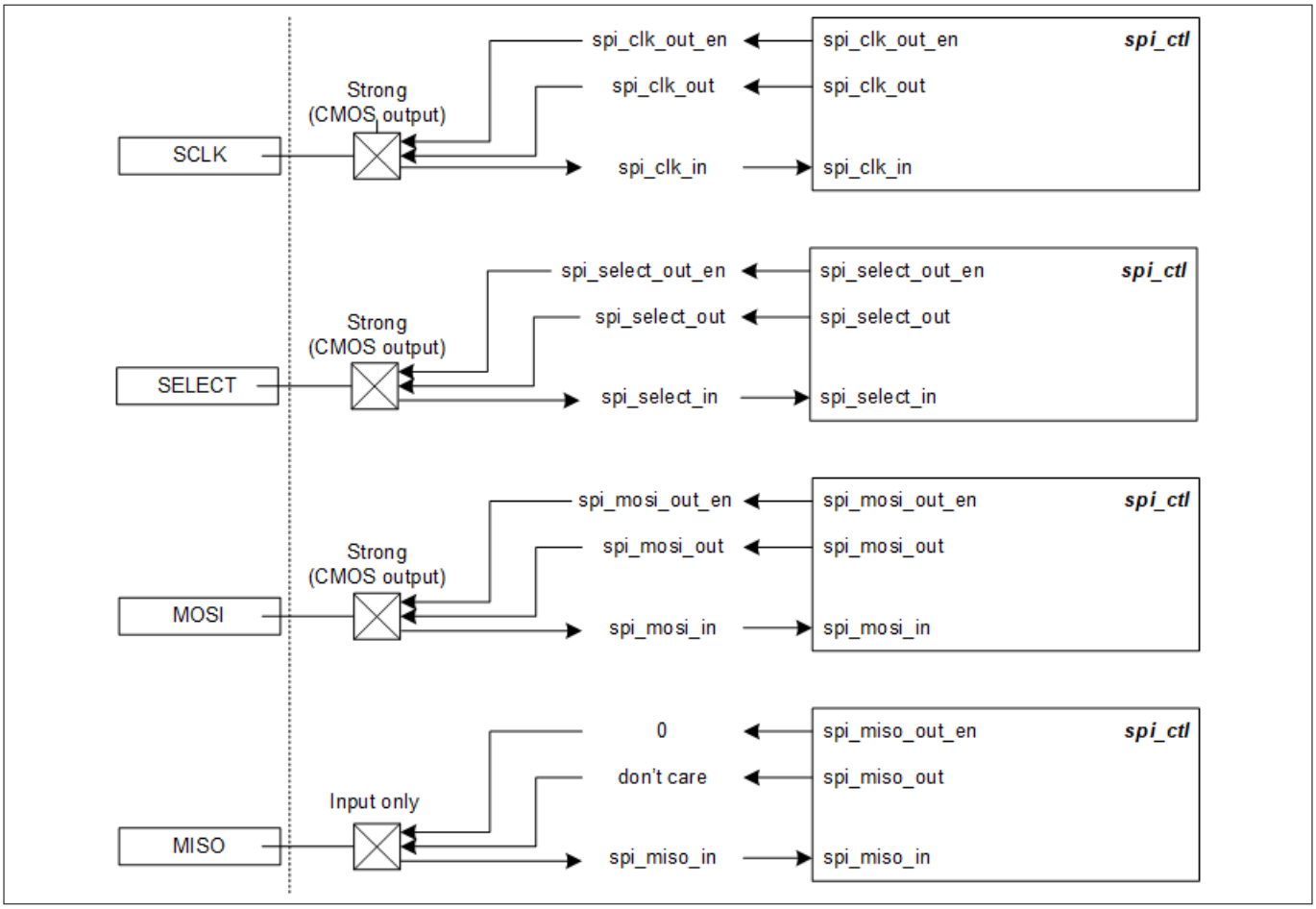


Figure 251 SPI master I/O pad connection usage

Table 171 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

(table continues...)

Table 171 (continued) SPI slave I/O signal description

I/O pads	Drive mode	On-chip I/O signals	Usage
SELECT	Input only	spi_select_out_en spi_select_out spi_select_in	Receive a select signal
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_ic_block_ec spi_miso_in spi_ec_miso_out_en spi_ec_miso_out	Transmit a data element

24.3.11 Registers summary

The SPI interface is managed through a set of 32-bit control and status registers. For more information on these registers, see the CYW20829 registers reference manual.

24.4 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 primary signals:

- **Tx**: Transmitter output
- **Rx**: Receiver input

Additionally, two side-band signals are used to implement flow control in UART. Flow control applies only to the Tx functionality:

- **Clear to Send (CTS)**:
  - Input signal to the transmitter
  - Active state signals that the receiver is ready to receive data and the transmitter can send the data
- **Request to Send (RTS)**:
  - Output signal from the receiver
  - Active state indicates 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, it activates its RTS signal, which in turn activates the CTS of the transmitting device, signaling it to send data.

Note: *Flow control is only available in the UART standard protocol mode and is not supported in the following modes:*

- *UART Local Interconnect Network (LIN)*
- *SmartCard (ISO7816)*
- *Infrared Data Association (IrDA)*
- *UART multiprocessor mode*

Not all SCBs support UART mode. See the device datasheet for details.

24.4.1 Features

The SCB supports the following features:

- UART protocol
  - Standard UART
  - Multi-processor mode
  - SmartCard (ISO7816) reader
  - Infrared Data Association (IrDA)
  - Local Interconnect Network (LIN)
- Break detection
- Baud rate detection
- Collision detection, which identifies when a driven bit value is not reflected on the bus, indicating another component is driving the same bus
- Programmable data frame size from 4 to 9 bits
- Programmable number of STOP bits, configurable in terms of half-bit periods between 1 and 4
- Parity support, including odd and even parity
- Programmable oversampling
- Start skipping
- Hardware flow control

24.4.2 General description

Figure 252 shows an example of UART Tx and Rx transfers.

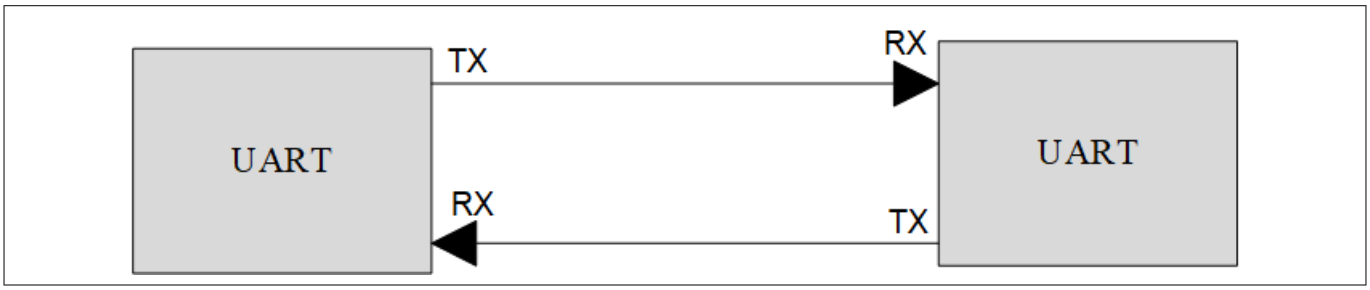


Figure 252 UART example

A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit, and completed by one or more stop bits. The start and stop bits indicate the beginning and end of the data transmission. The parity bit is sent by the transmitter and is used by the receiver to detect single-bit errors. Since the interface is asynchronous and does not use a clock, the transmitter and receiver operate on their own clocks. Therefore, the transmitter and receiver must 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 the start, stop, and parity bits. The number of stop bits can range from 1 to 4. The parity bit can be enabled or disabled. If enabled, the parity type can be set to even parity or odd parity. Using the parity bit is available only in Standard UART and SmartCard UART modes. In the IrDA UART mode, the parity bit is automatically disabled.

Note:
- *The UART interface does not support external clocking. It operates only in Active and Sleep system power modes*
- *UART supports only FIFO buffer mode*

24.4.3 UART modes of operation

24.4.3.1 Standard protocol

A typical UART transfer consists of a start bit followed by multiple data bits, optionally a parity bit, and one or more stop bits.

- The start bit value is always `0`
- The data bit values depend on the data being transferred
- The parity bit value ensures even or odd parity across the data bits
- The stop bit value is always `1`

The transmitter generates the parity bit, and the receiver can use it to detect single-bit transmission errors. When data is not being transmitted, the Tx line remains at `1`, the same value as the stop bits.

Since the interface operates without a clock, the transmitter and receiver must agree on the baud rate. The transmitter and receiver use their own internal clocks. The receiver clock operates at a higher frequency than the bit transfer frequency, allowing it to oversample the incoming signal.

The transition from a stop bit to a start bit is represented by a change from `1` to `0` on the Tx line. This transition is used by the receiver to synchronize with the transmitter clock. Synchronization at the start of each data transfer enables error-free communication even if frequency drift occurs between the transmitter and receiver clocks. Clock accuracy requirements depend on the size of the data transfer.

The stop period, or the number of stop bits between successive data transfers, is typically agreed upon between the transmitter and receiver. It usually ranges from 1 to 3-bit transfer periods.

Figure 253 illustrates the UART protocol.

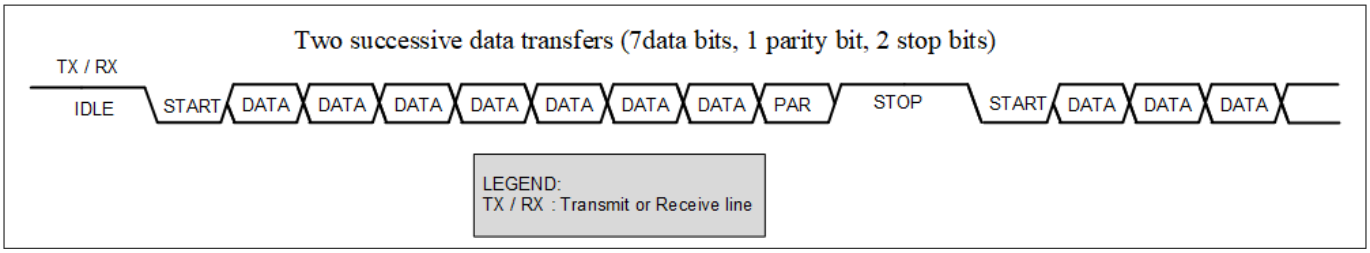


Figure 253 UART standard protocol example

The receiver oversamples the incoming signal. The value of the sample point in the middle of the bit transfer period, as determined by the receiver's clock, is used. Figure 254 illustrates this process. Note the slight drift between the transmitter and receiver clocks.

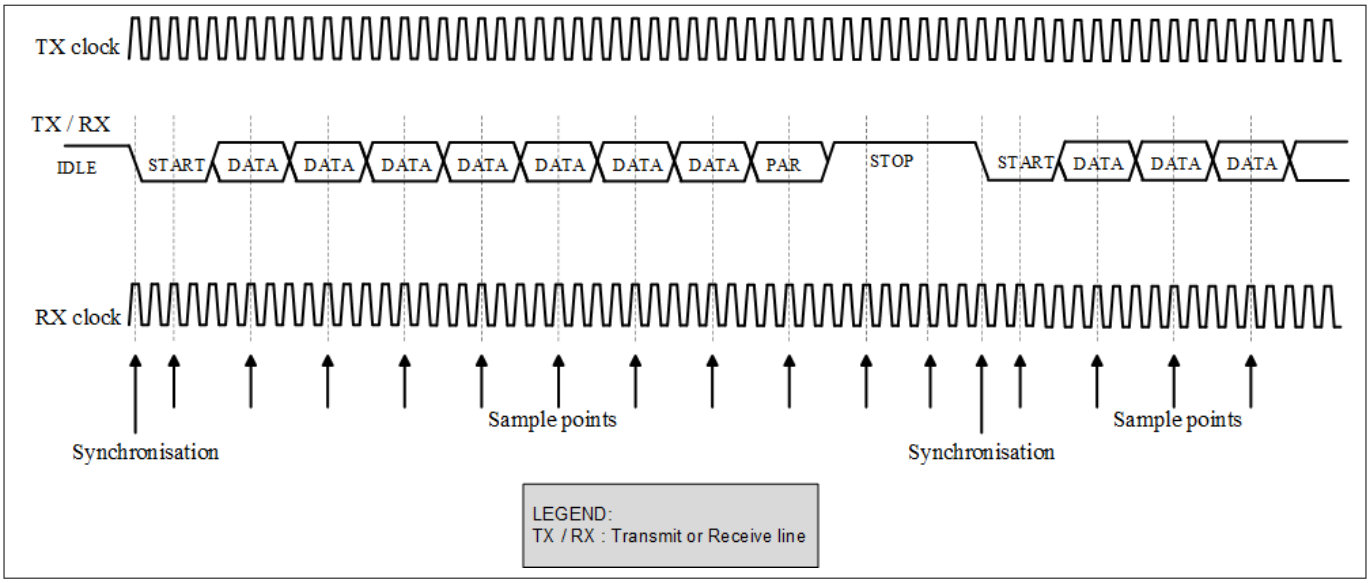


Figure 254 UART standard protocol oversampling (Rx clock faster than Tx clock)

Alternatively, three samples around the middle of the bit transfer period, based on the receiver's clock, can be used for a majority vote. This method reduces noise sensitivity and increases accuracy. It is enabled by setting the MEDIAN filter in the SCBx_RX_CTRL register.

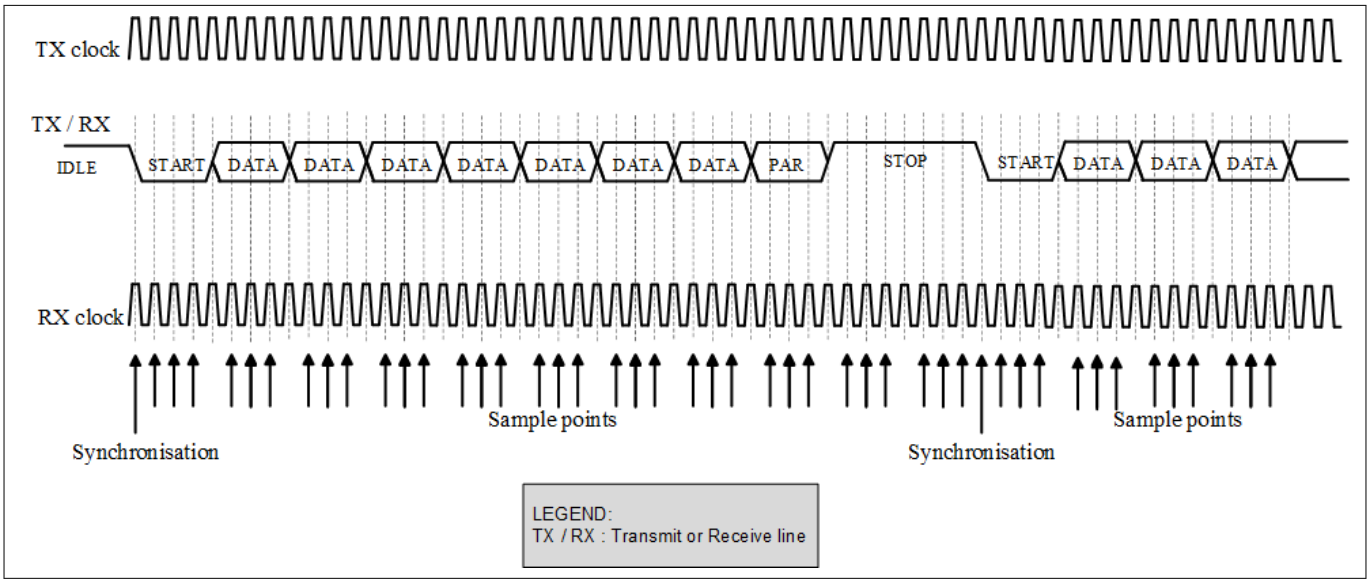


Figure 255 UART standard protocol oversampling (Rx clock equal to Tx clock)

Parity

This functionality adds a parity bit to the data frame, which is used to identify single-bit data frame errors. The parity bit is always inserted directly after the data frame bits.

- The transmitter calculates the parity bit (when SCBx_UART_TX_CTRL.PARITY_ENABLED is 1) based on the data frame bits:
  - For even parity, SCBx_UART_TX_CTRL.PARITY is 0
  - For odd parity, SCBx_UART_TX_CTRL.PARITY is 1
- The receiver checks the parity bit (when SCBx_UART_RX_CTRL.PARITY_ENABLED is 1) based on the received data frame bits:
  - For even parity, SCBx_UART_RX_CTRL.PARITY is 0
  - For odd parity, SCBx_UART_RX_CTRL.PARITY is 1

Parity functionality applies to both Tx and Rx operations. Dedicated control fields are available for configuration:

- **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 parity error is detected by the receiver:

- If SCBx_UART_RX_CTRL.DROP_ON_PARITY_ERROR is 0, the data frame is placed in the Rx FIFO
- If SCBx_UART_RX_CTRL.DROP_ON_PARITY_ERROR is 1, the data frame is dropped

Figure 256 illustrates the parity functionality for an 8-bit data frame.

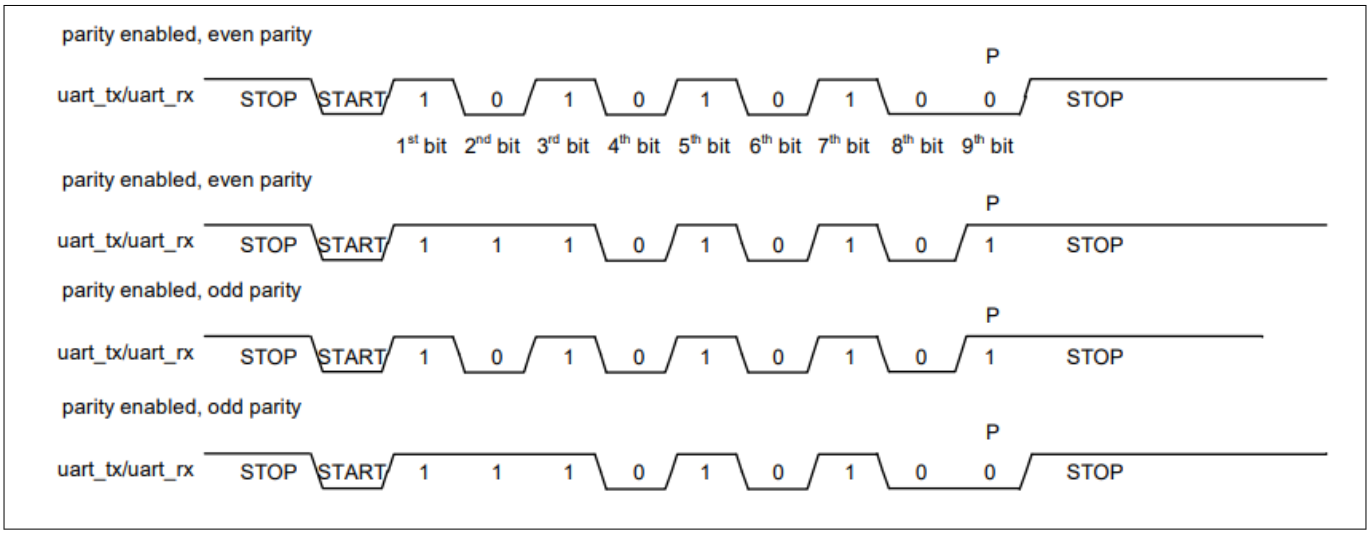


Figure 256 UART parity example

Start skipping

Start skipping applies only to receive functionality and is supported in standard UART mode. In regular receive operation, synchronization occurs during the START bit period (a 1 to 0 transition on the UART Rx line). In start skipping mode, synchronization occurs on the first received data frame bit, which must be a 1 (a 0 to 1 transition on the UART Rx line).

Start skipping is typically used to allow wake-up from the System Deep Sleep power mode for UART Rx functionality. The process follows these steps:

- Before entering System Deep Sleep power mode:
  - UART Rx functionality is disabled
  - IOSS/GPIO is programmed to trigger an interrupt when a 1 to 0 transition (START bit) is detected on the UART Rx line
- While in System Deep Sleep power mode, UART Rx functionality is not operational
- The IOSS/GPIO interrupt is activated on detecting the START bit, and the system transitions from System Deep Sleep to CPU Active power mode. This transition takes approximately 20 µs
- The CPU enables UART Rx functionality and sets SCBx_UART_RX_CTRL.SKIP_START to 1

- The UART receiver synchronizes data frame reception on the next `0` to `1` transition. If UART Rx functionality is enabled in time, this will correspond to the transition from the START bit to the first received data frame bit
- The UART receiver resumes normal operation, where successive data frame synchronization occurs during the START bit period

Figure 257 illustrates the start skipping process.

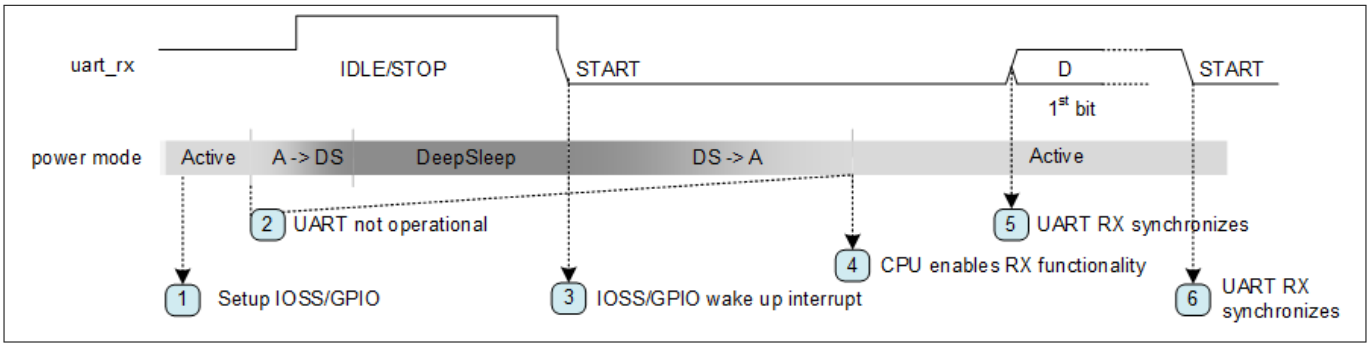


Figure 257 UART start skip and wakeup from System Deep Sleep

Note that the process described above works only for lower baud rates. The transition from System Deep Sleep to CPU Active power mode and the CPU enabling the UART receive functionality must take less than one-bit period. This ensures that the UART receiver is active in time to detect the `0` to `1` transition.

In step 4 of the process, firmware requires some time to complete the wake-up interrupt routine and enable the UART receive functionality before the block can detect the rising edge on the UART Rx line.

If the above steps cannot be completed within one-bit time, a "dummy" byte should be sent to the device to wake it up before transmitting actual UART data. In this case, the `SKIP_START` bit can remain set to `0`.

Note:
- *This feature works only if the GPIO interrupt is triggered on the falling edge of the START bit (and not on other data bits)*
- *If the GPIO interrupt is triggered incorrectly, the UART may receive corrupted data bytes. Correct operation depends on higher-level protocols to detect data corruption and determine when it is safe for the UART to enter Sleep mode*

Break detection

Break detection is supported in standard UART mode. This functionality detects when the UART Rx line remains low (`0`) for more than `SCBx_UART_RX_CTRL.BREAK_WIDTH` bit periods. The break width must be larger than the maximum number of low (`0`) bit periods in a regular data transfer, plus an additional 1-bit period. This additional bit period is a minimum requirement and should preferably be larger to account for clock inaccuracies between the transmitter and receiver.

For example, for an 8-bit data frame size with parity support:

- The maximum number of low (`0`) bit periods is 10 (1 START bit, 8 `0` data frame bits, and 1 `0` parity bit)
- Therefore, the break width should be larger than `10 + 1 = 11`
- In this case, `SCBx_UART_RX_CTRL.BREAK_WIDTH` can be set to 11
- Break detection applies only 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 frame is used by the transmitter to indicate a special condition to the receiver, which may result in actions such as a reset, shutdown, or initialization sequence at the receiver

Note:
- *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 258 illustrates a regular data frame and a break frame (8-bit data frame, parity support, and a break width of 12-bit periods.

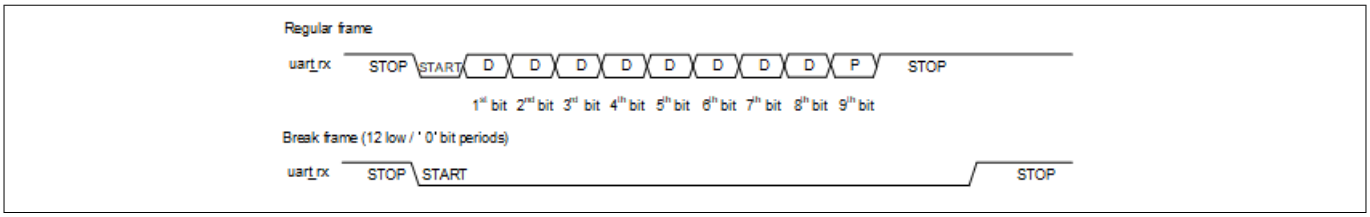


Figure 258 UART – Regular frame and break frame

When SCBx_UART_RX_CTRL.BREAK_LEVEL is set to 1, idle-line detection can be performed. For example, after successive transfers of several UART data frames, an idle (high) level lasting longer than the length of a normal data frame (start + 8 data bits + 1 parity bit + 1 stop bit) indicates the end of the successive transfer.

Flow control

The standard UART mode supports flow control, which regulates the pace at which the transmitter sends data to the receiver. Flow control is enabled through the SCBx_UART_FLOW_CTRL.CTS_ENABLED register field:

- When this field is set to 0, the transmitter sends data whenever its Tx FIFO is not empty
- When this field is set to 1, the transmitter sends data only when the UART CTS line is active and the Tx FIFO is not empty

***Note**: Flow control applies only to Tx functionality.*

Two UART side-band signals are used to implement flow control:

- **UART RTS (uart_rts_out)**:
  - Output signal from the receiver
  - Active state indicates that the receiver is ready to receive data (RTS: Request to Send)
- **UART CTS (uart_cts_in)**:
  - Input signal to the transmitter
  - Active state indicates that the transmitter can send data (CTS: Clear to Send)

The receiver's uart_rts_out signal connects to the transmitter's uart_cts_in signal. The uart_rts_out signal is activated based on the number of used Rx FIFO entries compared with the SCBx_UART_FLOW_CTRL.TRIGGER_LEVEL field. If the number of used Rx FIFO entries is less than SCBx_UART_FLOW_CTRL.TRIGGER_LEVEL, uart_rts_out becomes active.

Polarity control:

- UART side-band signals are typically active low but can sometimes be active high
- The polarity of the side-band signals is controlled using:
  - SCBx_UART_FLOW_CTRL.RTS_POLARITY for the RTS signal
  - SCBx_UART_FLOW_CTRL.CTS_POLARITY for the CTS signal

Figure 259 provides an overview of the flow control functionality.

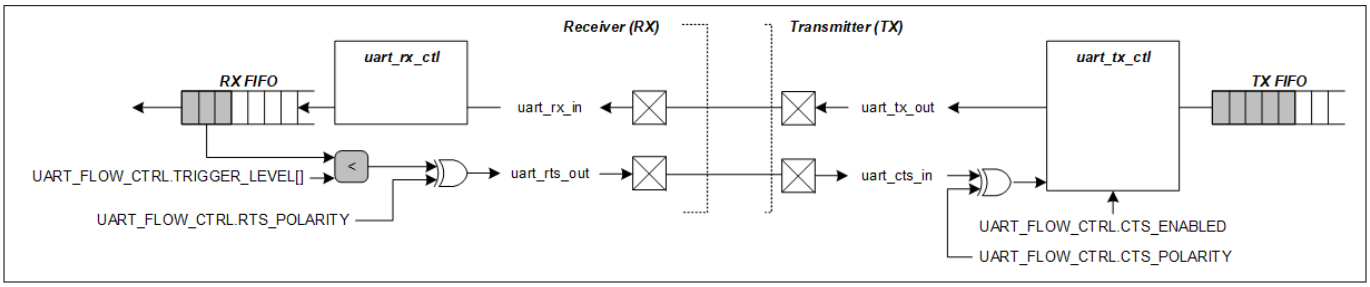


Figure 259 UART flow control connection

UART multi-processor mode

The UART_MP (multi-processor) mode operates in a single-master, multi-slave topology, as shown in Figure 260. This mode is also referred to as the UART 9-bit protocol because the data field is nine bits wide. UART_MP is a part of Standard UART mode.

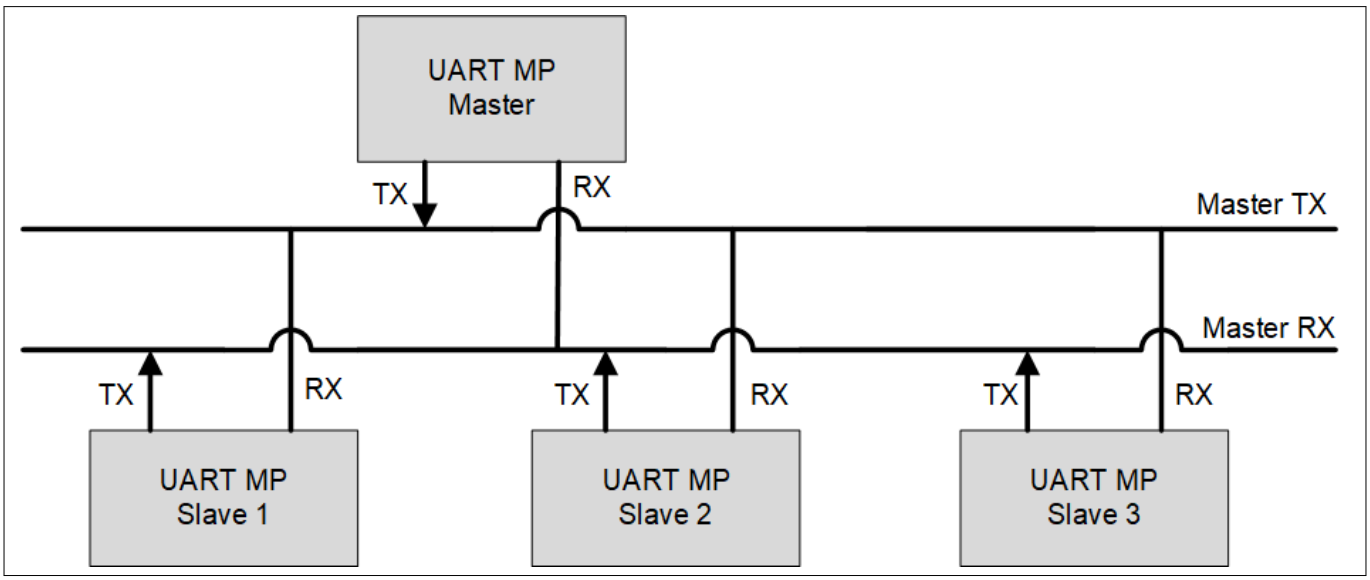


Figure 260 UART MP mode bus connections

The main properties of the UART_MP mode are:

- Single-master, multi-slave configuration (multi-drop network)
- Each slave is identified by a unique address
- Uses a 9-bit data field, with the ninth bit serving as the address/data flag (MP bit):
  - When the MP bit is set high, it indicates an address byte
  - When the MP bit is set low, it indicates a data byte
  - A data frame is shown in Figure 261
- Parity bit is disabled

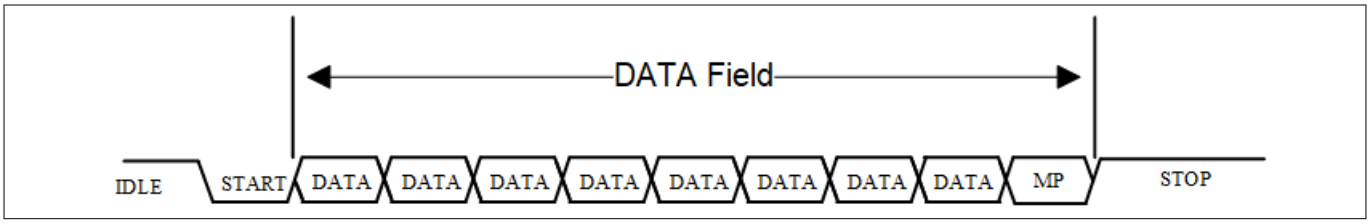


Figure 261 UART MP address and data frame

The SCB can operate as either a master or a slave device in UART_MP mode. Both the SCBx_TX_CTRL and SCBx_RX_CTRL registers must be configured for a 9-bit data frame size.

- When operating as a UART_MP master device, the firmware updates the MP flag for each address or data frame
- When operating as a UART_MP slave device:
  - The MP_MODE field of the SCBx_UART_RX_CTRL register must be set to 1
  - The SCBx_RX_MATCH register must be configured with the slave address and address mask
  - If the ADDR_ACCEPT field of the SCBx_CTRL register is set to 1, the matched address is written into the Rx FIFO

If the received address does not match the device's address, the interface ignores the subsequent data until another address is received and compared.

Configuring the SCB as a standard UART interface

To configure the SCB as a standard UART interface, follow these steps:

1. Configure the SCB as a UART interface by writing `10b` to the `MODE` field (bits [25:24]) of the `SCBx_CTRL` register
2. Configure the UART interface to operate in Standard protocol mode by writing `00` to the `MODE` field (bits [25:24]) of the `SCBx_UART_CTRL` register
3. To enable UART MP mode or UART LIN mode:
   a. Write `1` to the `MP_MODE` field (bit 10) of the `SCBx_UART_RX_CTRL` register to enable UART MP mode
   b. Write `1` to the `LIN_MODE` field (bit 12) of the `SCBx_UART_RX_CTRL` register to enable UART LIN mode
4. Follow 2. through 4. in the Enabling and initializing UART section

24.4.3.2 SmartCard (ISO7816)

ISO7816 is an asynchronous serial interface based on a single-master, single-slave topology. It defines both Reader (master) and Card (slave) functionality. For details, refer to the ISO7816 specification.

The SCB supports both master (Reader) and slave (Card) functions. It provides basic physical-layer support with asynchronous character transmission. The `UART_TX` line is internally multiplexed between the UART_TX and UART_RX control modules and connected to the SmartCard I/O line.

A SmartCard transfer is similar to a UART transfer but includes the additional feature of a negative acknowledgment (NACK), which may be sent from the receiver to the transmitter.

- A NACK is always represented as `0`. Both the master and slave can drive the same line, but never simultaneously

In a SmartCard transfer:

- The transmitter drives the START bit, data bits, and optionally a parity bit
- After transmitting these bits, the transmitter releases the bus, entering the stop period. This causes the line to idle at `1` (the value of a stop bit)
- After half a bit transfer period into the stop period, the receiver may drive a NACK on the line (a value of `0`) for half a bit transfer period

When a NACK is driven:

- The transmitter observes the NACK and responds by extending the stop period by one bit transfer period
- For this protocol to work, the stop period must be a minimum of two bit transfer periods

***Note**: A data transfer with a NACK takes one bit transfer period longer than a transfer without a NACK.*

Typically, implementations use a tristate driver with a pull-up resistor, ensuring that when the line is not transmitting data or transmitting the stop bit, its value is `1`.

Figure 262 illustrates the SmartCard protocol.

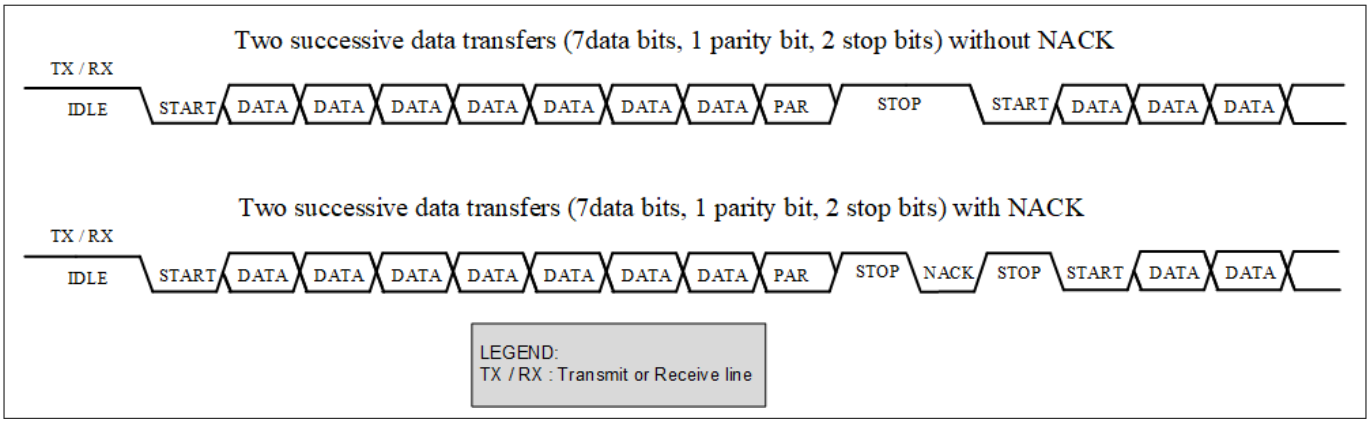


Figure 262 SmartCard example

The communication baud rate when using the SmartCard is defined as: `Baud rate = Fscb_clk/Oversample`

Where:

1. `Fscb_clk` is the frequency of the `scb_clk` clock
2. Oversample is the oversampling factor

Configuring SCB as UART SmartCard interface

To configure the SCB as a UART SmartCard interface, follow these steps:

1. Configure the SCB as a UART interface by writing `10` to the `MODE` field (bits [25:24]) of the `SCB_CTRL` register
2. Configure the UART interface to operate in SmartCard protocol mode by writing `01` to the `MODE` field (bits [25:24]) of the `SCB_UART_CTRL` register
3. Follow 2. through 4. in the Enabling and initializing UART section

For more information on these registers, refer to the UART registers.

24.4.3.3 Infrared data association (IrDA)

The SCB supports the IrDA protocol for data rates up to 115.2 kbps using the UART interface. It provides support for only the basic physical layer of the IrDA protocol for rates below 115.2 kbps. Therefore, the system implementing this block must incorporate additional system resources to create a complete IrDA communication system.

The IrDA protocol adds a modulation scheme to UART signaling. At the transmitter, the bits are modulated, and at the receiver, the 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
- A bit value of `1` is signaled by holding the line to `0`

For data rates up to 115.2 kbps, the RZI modulation scheme is used with a pulse duration of 3/16 of the bit period. The sampling clock frequency must be set to 16 times the selected baud rate by configuring the `SCB_OVS` field in the `SCB_CTRL` register. Additionally, the CYW20829 MCU supports a low-power IrDA receiver mode, which can detect pulses with a minimum width of 1.41 µs.

Bit rates below 115.2 kbps can be achieved by configuring the `clk_scb` frequency. Supported rates include:

- 2.4 kbps
- 9.6 kbps
- 19.2 kbps
- 38.4 kbps
- 57.6 kbps

Figure 263 shows how a UART transfer is modulated using IrDA.

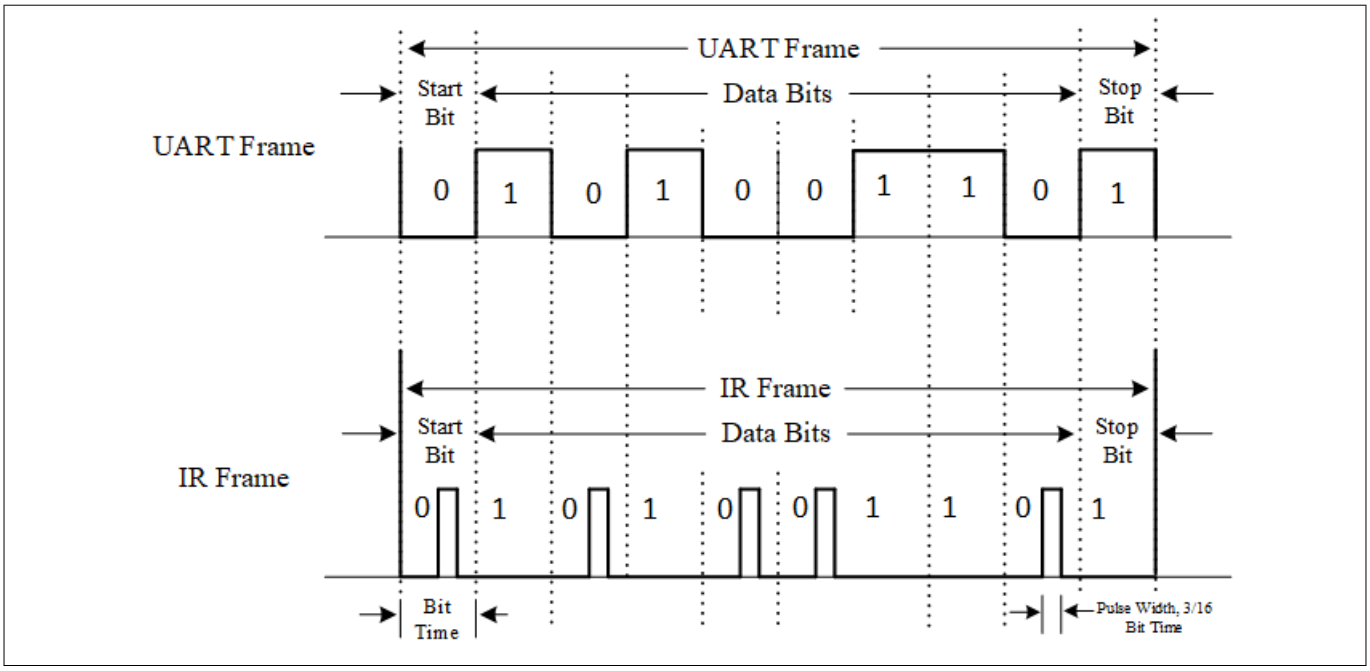


Figure 263 IrDA example

Configuring the SCB as a UART IrDA interface

To configure the SCB as a UART IrDA interface, follow these steps:

1. Configure the SCB as a UART interface by writing `10b` to the `MODE` field (bits [25:24]) of the `SCB_CTRL` register
2. Configure the UART interface to operate in IrDA protocol mode by writing `10b` to the `MODE` field (bits [25:24]) of the `SCB_UART_CTRL` register
3. Enable the Median filter on the input interface line by writing `1` to the `MEDIAN` field (bit 9) of the `SCB_RX_CTRL` register
4. Configure the SCB as described in Enabling and initializing UART

For more information on these registers, refer to the UART registers.

24.4.3.4 Local interconnect network (LIN) mode

The SCB supports the LIN protocol as part of the standard UART. LIN is designed with a single-master, multi-slave topology, where one master node and multiple slave nodes communicate on the LIN bus.

The SCB UART provides support only for LIN slave functionality. The LIN specification defines both the physical layer (layer 1) and the data link layer (layer 2).

Figure 264 illustrates the UART_LIN interface and the LIN transceiver.

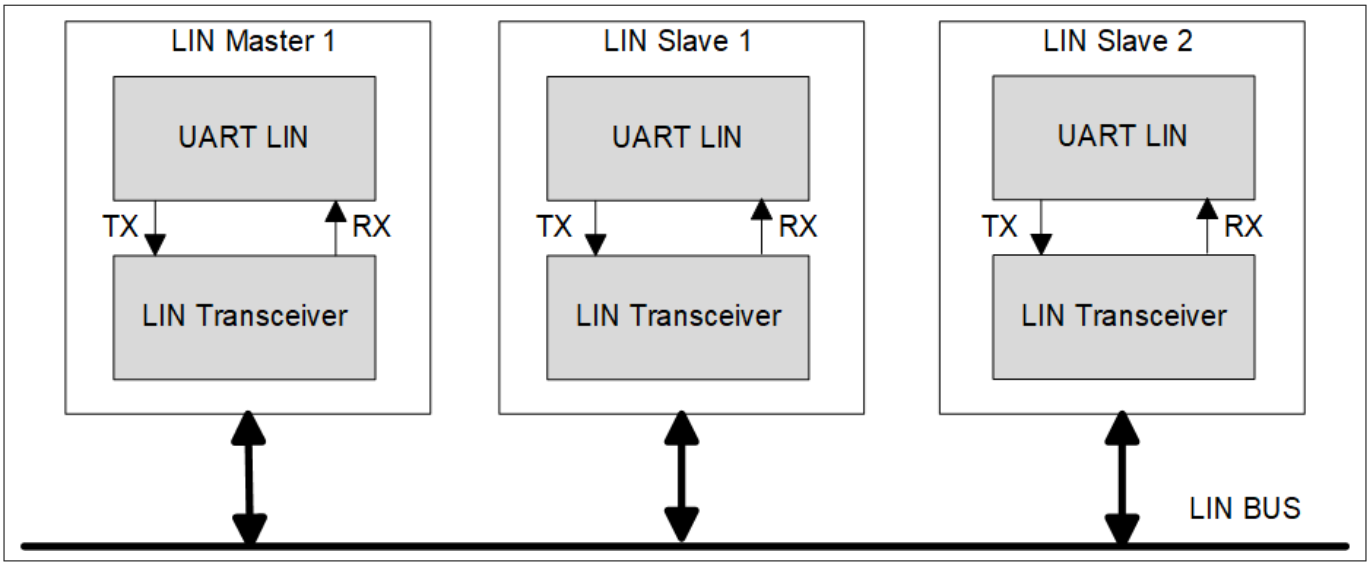


Figure 264 UART_LIN and LIN transceiver

The LIN protocol defines two tasks:

- **Master task**: Involves sending a header packet to initiate a LIN transfer
- **Slave task**: Involves transmitting or receiving a response

The master node supports both the master task and the slave task, while the slave node supports only the slave task, as shown in Figure 265.

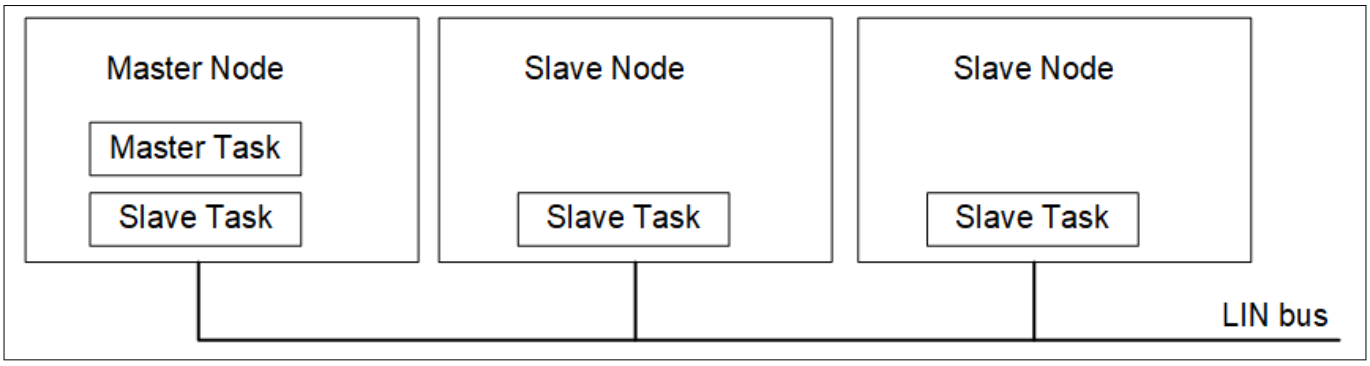


Figure 265 LIN bus nodes and tasks

LIN frame structure

LIN operates by transmitting frames at predetermined time intervals. A frame is divided into two fields: header and response, as shown in Figure 266.

The header field consists of the following:

- **Break field**: At least 13 bit periods with a value of `0`
- **Sync field**: A `0x55` byte frame used to synchronize the clock of the slave task with the master task
- **Identifier field**: A frame that specifies a specific slave device

The response field consists of:

- Data
- Checksum

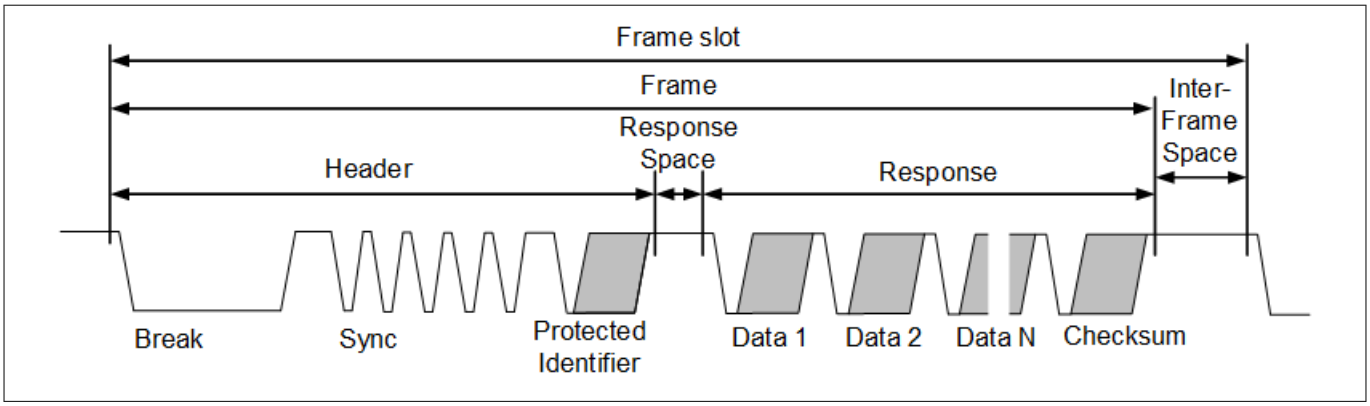


Figure 266 LIN frame structure

In LIN protocol communication, the least significant bit (LSb) of the data is transmitted first, and the most significant bit (MSb) is transmitted last. The start bit is encoded as `0`, and the stop bit is encoded as `1`.

The following sections describe all byte fields in the LIN frame.

Break field

Every new frame in the LIN protocol begins with a break field, which is always generated by the master. The break field consists of logical zero for a minimum of 13 bit times, followed by a break delimiter.

The structure of the break field is shown in Figure 267.

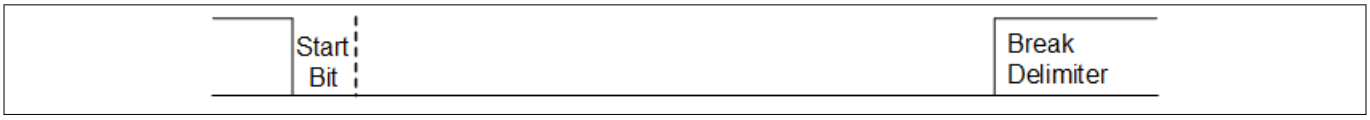


Figure 267 LIN break field

Sync field

The sync field is the second field transmitted by the master in the header. Its value is `0x55`. The sync field is used to synchronize the clock of the slave task with the master task, enabling automatic baud rate detection.

Figure 268 shows the structure of the LIN sync field.

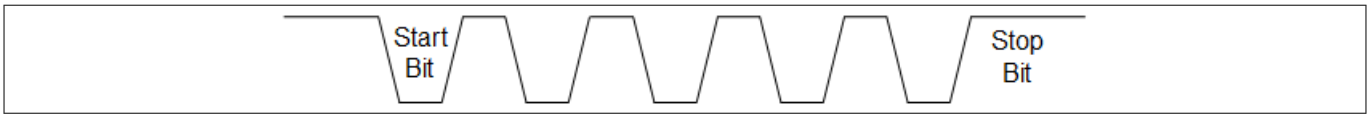


Figure 268 LIN sync field

Protected identifier (PID) field

A protected identifier (PID) field consists of two sub-fields: the frame identifier (bits 0–5) and the parity (bits 6 and 7). The structure of the PID field is shown in Figure 269.

- **Frame identifier**:
  - Values `0` to `59` (`0x3B`) are used for signal-carrying frames
  - Values `60` (`0x3C`) and `61` (`0x3D`) are used to carry diagnostic and configuration data
  - Values `62` (`0x3E`) and `63` (`0x3F`) are reserved for future protocol enhancements
- **Parity**: Parity is calculated using the frame identifier bits

Note: *The parity calculation of the PID will also be handled by firmware. The SCB block's UART processes the PID as a standard data byte.*

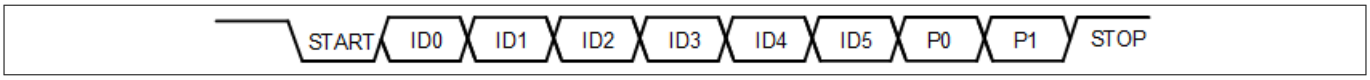


Figure 269 PID field

Data

In LIN, each frame can carry a minimum of one byte and a maximum of eight bytes of data. The least significant bit (LSb) of the data byte is transmitted first, and the most significant bit (MSb) is transmitted last.

Checksum

The checksum is the last byte field in the LIN frame. It is calculated by inverting the 8-bit sum (including the carryover) of either:

- All data bytes only
- All data bytes along with the PID field

There are two types of checksums in LIN frames:

- **Classic checksum**: Calculated over all data bytes only (used in LIN 1.x slaves)
- **Enhanced checksum**: Calculated over all data bytes and the protected identifier (used in LIN 2.x slaves)

***Note**: The checksum calculation is handled by firmware, and the SCB block's UART processes it as a standard data byte.*

24.4.4 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` can be set to 1 MHz, and the oversample factor set to `10`. The oversampling value is configured using the `SCB_CTRL.OVS` register field, where the oversampling value is `SCB_CTRL.OVS + 1`.

Oversampling in different UART sub-modes:

1. **Standard UART and SmartCard sub-modes**:
   - Valid `OVS` field range: [7,15]
   - This includes protocols like LIN
2. **UART transmit IrDA sub-mode**:
   - This field indirectly specifies the oversampling
   - Oversampling determines the interface clock per bit cycle and the pulse width
   - Only one valid `OVS` value: 16 (written as `0` in the `OVS` field of the `SCB_CTRL` register)
   - Pulse width is approximately 3/16 of the bit period for all bit rates
3. **UART receive IrDA sub-mode**: Supported baud rates: 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, the pulse is approximately 3/16 of the bit period for all bit rates
   - In low-power transmission mode (for rates less than 115.2 kbps), the pulse width is smaller:
     - Typical pulse width: 1.62 µs
     - Minimum pulse width: 1.41 µs

Pulse detection in UART receive IrDA sub-mode:

- Pulse widths greater than or equal to two SCB input clock cycles are reliably detected by the receiver
- Pulse widths less than two but 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
- `SCB_RX_CTRL.MEDIAN` must be set to `1` to enable IrDA receiver functionality

The SCB input clock and the oversampling factor determine the IrDA bit rate. Refer to the register documentation for more details about OVS values for different baud rates.

24.4.5 Loopback

In UART mode, the SCB supports an internal loopback mechanism, allowing it to receive data from the Rx pad and transmit the same data through the Tx pad without CPU involvement. This feature is configured using the SCBx_UART_CTRL.LOOPBACK register.

24.4.6 Enabling and initializing UART

The UART must be programmed in the following order:

1. Program protocol-specific information using the UART_TX_CTRL, UART_RX_CTRL, and UART_FLOW_CTRL registers:
   a. Select sub-modes of the protocol
   b. Configure transmitter and receiver functionality
2. Program generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers:
   a. Specify the data frame width
   b. Specify whether the MSb (most significant bit) or LSb (least significant bit) 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:
   a. Set the trigger level
   b. Clear the transmitter and receiver FIFOs and Shift registers
4. Enable the block by writing 1 to the ENABLE bit of the SCB_CTRL register:
   a. After the block is enabled, do not change the control bits. Any changes, such as modifying the operation mode (for example, from SmartCard to IrDA), must be made after disabling the block
   b. The change takes effect only after the block is re-enabled
   c. Re-enabling the block causes re-initialization, which results in loss of the associated state, such as FIFO content

24.4.7 I/O pad connection

24.4.7.1 Standard UART mode

Figure 270 and Figure 271, along with Table 172, show the use of the I/O pads for the Standard UART mode.

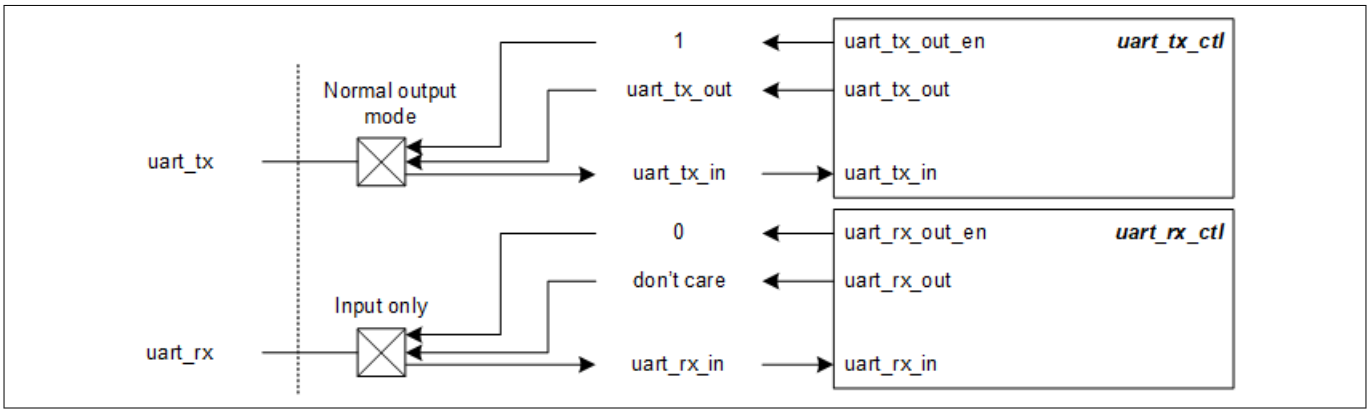


Figure 270 Standard UART mode I/O pad connections

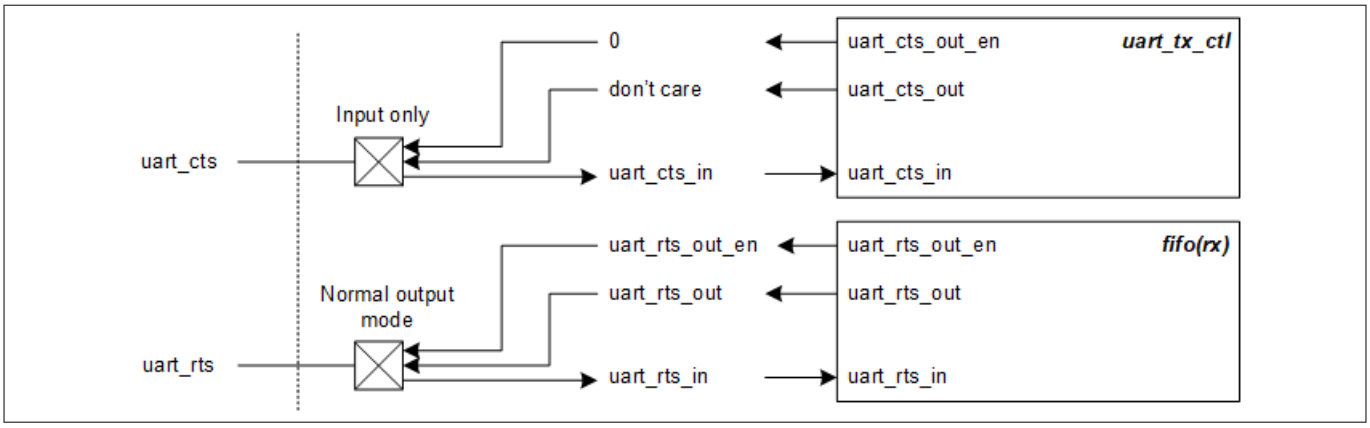


Figure 271 Standard UART mode and flow control I/O pad connections

Table 172 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
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

24.4.7.2 SmartCard mode

Figure 272 and Table 173 show the use of the I/O pads for SmartCard mode.

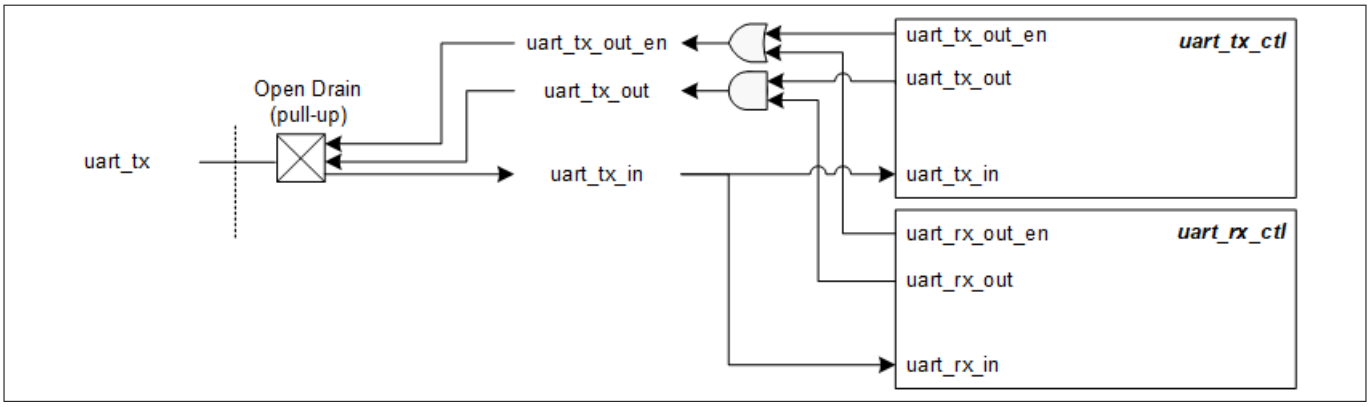


Figure 272 SmartCard mode I/O pad connections

Table 173 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 • Used to receive a negative acknowledgment of a transmitted data element
		uart_tx_out_en uart_tx_out	• Used to transmit a data element • Used to transmit a negative acknowledgment for a received data element

24.4.7.3 IrDA mode

Figure 273 and Table 174 show the use of the I/O pads for IrDA mode.

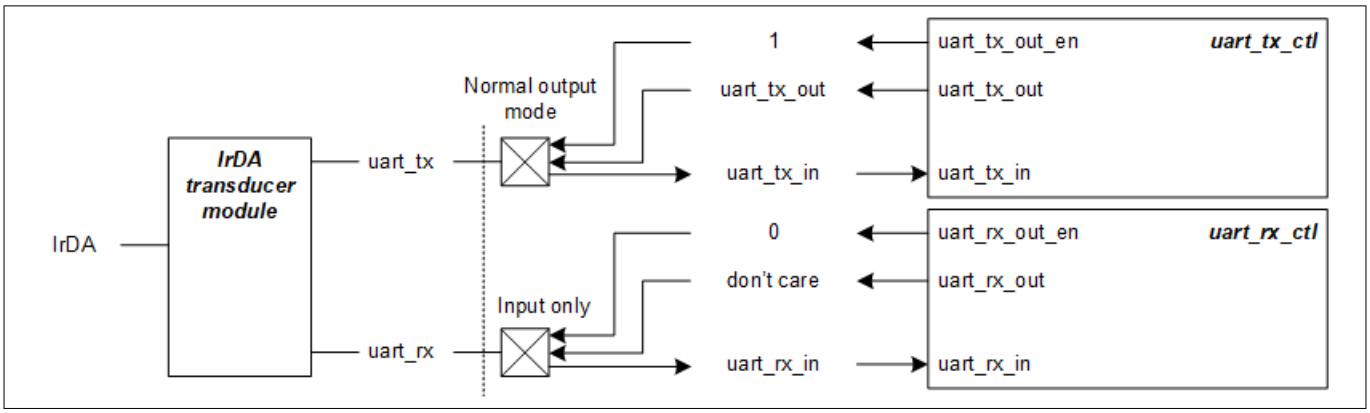


Figure 273 IrDA mode I/O pad connections

Table 174 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

24.4.7.4 LIN mode

Figure 274 and Table 175 show the use of the I/O pads for LIN mode.

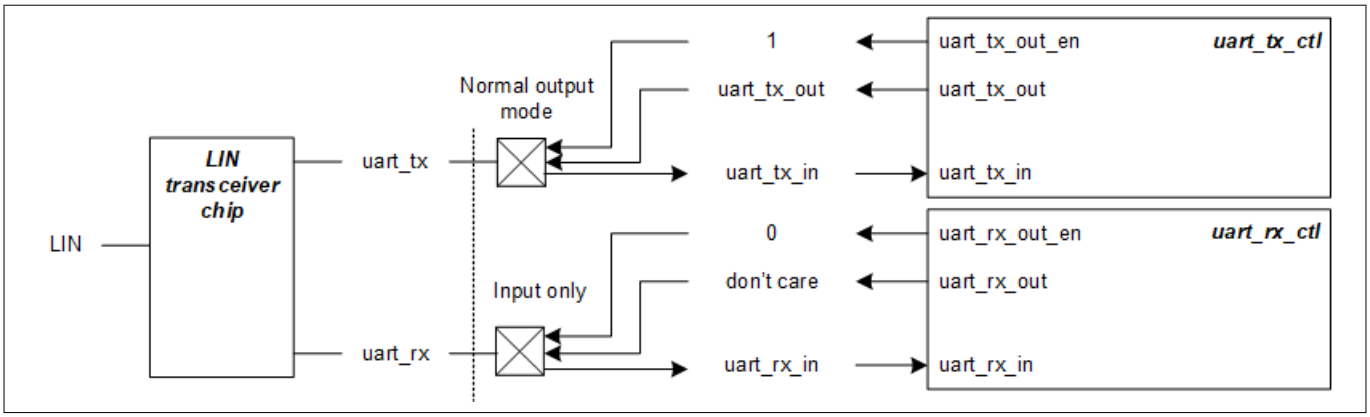


Figure 274 LIN mode I/O pad connections

Table 175 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

24.4.8 Registers summary

The UART interface is controlled using a set of 32-bit registers. For detailed information on these registers, see the CYW20829 registers reference manual.

24.5 Inter-integrated circuit (I^2C)

This section explains the implementation of I^2C in CYW20829. For detailed information on the I^2C protocol specification, refer to the I^2C-bus specification available on NXP's Website.

In CYW20829:

- `SCB[0]` and `SCB[2]` blocks support both master and slave modes
- Only `SCB[0]` is available in Deep Sleep power mode and supports externally-clocked operations

24.5.1 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 filtering
- Clock stretching
- Collision detection
- Programmable oversampling of the I^2C clock signal (SCL)
- FIFO, EZ, and CMD_RESP buffer modes
- Local loopback control

- General-call address detection
- Wakeup from Sleep modes

24.5.2 General description

Figure 275 shows an example of an I²C communication network.

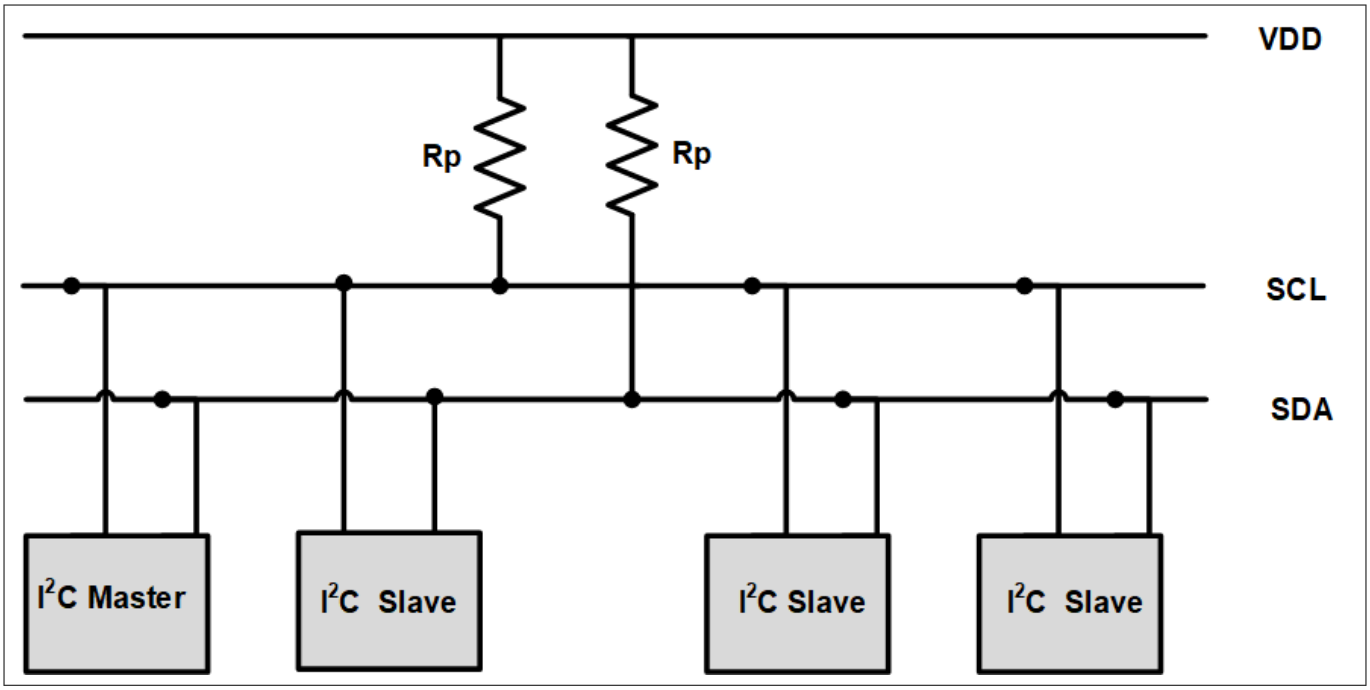


Figure 275 I²C interface block diagram

The standard I²C bus is a two-wire interface with the following lines:

- Serial Data (SDA)
- Serial Clock (SCL)

I²C devices connect 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 transmitters or receivers. Each slave device connected to the bus is software-addressable using a unique 7-bit address.

24.5.3 Terms and definitions

Table 176 describes the commonly used terms in an I²C communication network.

Table 176 Definition of I²C bus terminology

Term	Description
Transmitter	The device that sends data to the bus.
Receiver	The device that receives data from the bus.
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.

(table continues...)

Table 176 (continued) Definition of I²C bus terminology

Term	Description
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.

24.5.3.1 Clock stretching

When a slave device is not 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 remains '0', regardless of the values that any other master or slave device may be driving on the SCL line. This mechanism is known as clock stretching and represents the only situation in which a slave device drives the SCL line.

The master device monitors the SCL line and detects when it cannot generate a positive clock pulse '1' on the SCL line. In response, the master delays the generation of a positive edge on the SCL line, effectively synchronizing with the slave device that is stretching the clock.

The SCB on CYW20829 supports clock stretching and will use this functionality when necessary.

24.5.3.2 Bus arbitration

The I²C protocol operates as a multi-master, multi-slave interface. Master devices implement bus arbitration by monitoring the SDA line. Bus collisions occur when the master detects an SDA line value that differs from the value it is driving on the SDA line.

For example:

- Master 1 drives the value '1' on the SDA line while Master 2 drives the value '0' on the SDA line
- The actual SDA line value becomes '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 retains control of the bus

24.5.4 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 transitions from master to slave mode when it is addressed. Only one master may be active during a data transfer. The active master drives the clock signal on the SCL line. Table 177 shows the I²C modes of operation.

Table 177 I²C modes

Mode	Description
Slave	Slave only operation (default)
Master	Master only operation
Multi-master	Supports more than one master on the bus

Table 178 lists common bus events that occur during an I²C data transfer. The Write transfer and Read transfer sections describe the I²C bus bit format used during data transfer.

Table 178 I²C bus event 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.
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)

All modes support two types of transfer: read and write. In write transfer, the master sends data to the slave. In read transfer, the master receives data from the slave.

24.5.4.1 Write transfer

A typical write transfer begins with the master generating a START condition on the I²C bus. The master then writes a 7-bit I²C slave address followed by a write indicator (‘0’) 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 any connected slave devices or if the addressed device does not wish to acknowledge the request, it transmits a no acknowledgment (NACK) by not pulling the SDA line low. The absence of acknowledgment results in an SDA line value of ‘1’ due to the pull-up resistor implementation.

When the slave transmits no acknowledgment, the master may end the write transfer by generating a STOP condition. Alternatively, the master may generate a repeated START condition to retry the transfer.

If the master receives an acknowledgment, it transmits data to the bus. The addressed slave responds with an acknowledgment after receiving each data byte to confirm receipt. Upon receiving this acknowledgment, the master may continue transmitting additional data bytes.

When the transfer is complete, the master generates a STOP condition to terminate the operation.

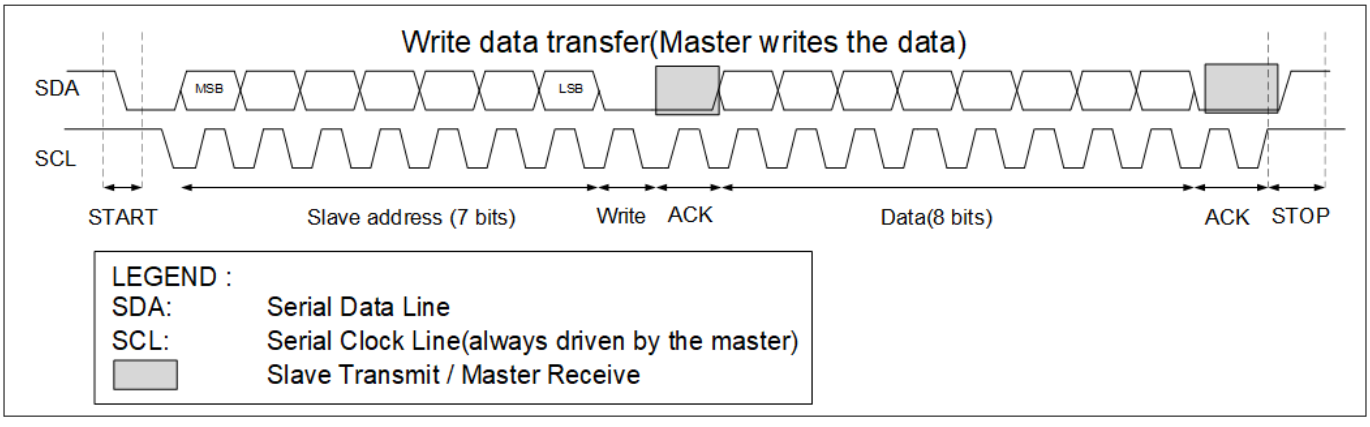


Figure 276 Master write data transfer

24.5.4.2 Read transfer

A typical read transfer begins with the master generating a START condition on the I²C bus. The master then writes a 7-bit I²C slave address followed by a read indicator (‘1’) after the START condition. The addressed slave transmits an acknowledgment by pulling the SDA line low during the ninth bit time.

If the slave address does not match any connected slave devices or if the addressed device does not desire to acknowledge the request, a no acknowledgment (NACK) is transmitted by not pulling the SDA line low. The absence of acknowledgment results in an SDA line value of ‘1’ due to the pull-up resistor implementation.

When the slave transmits no acknowledgment, the master may terminate the read transfer by generating a STOP condition. Alternatively, the master may generate a repeated START condition to retry the transfer.

If the slave acknowledges the address, it begins transmitting data immediately after the acknowledgment signal. The master transmits an acknowledgment to confirm the receipt of each data byte sent by the slave. Upon receiving this acknowledgment, the slave transmits additional data bytes.

The master may send a NACK signal to instruct the slave to stop sending data bytes. This concludes the read transfer.

When the transfer is complete, the master generates a STOP condition to terminate the operation.

Individual data transfers, consisting of one or more data elements, begin with a START event and end with a STOP condition. Combined data transfers consist of multiple individual transfers that are separated by repeated START events rather than STOP conditions.

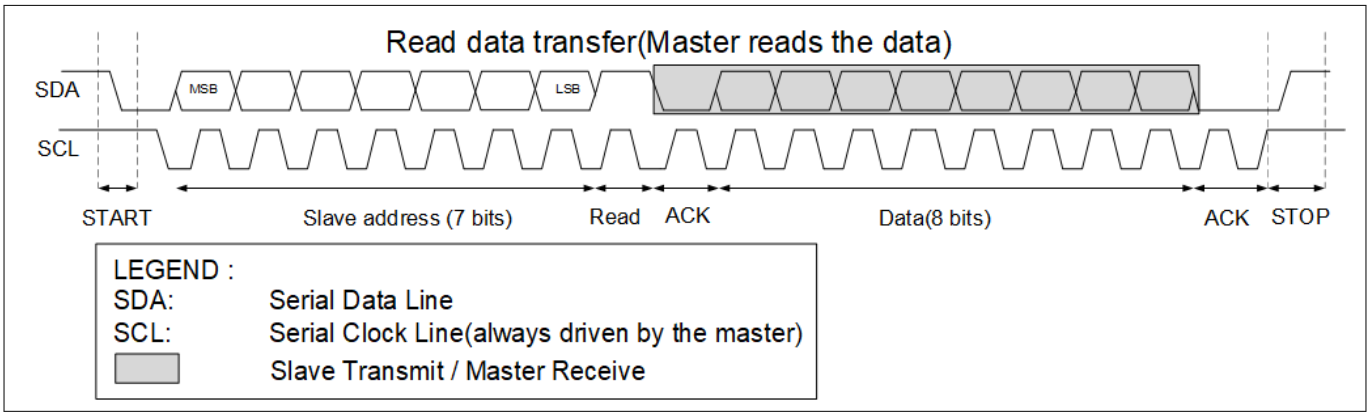


Figure 277 Master read data transfer

24.5.5 I²C buffer modes

I²C operates in three different buffered modes: FIFO, EZ, and CMD_RESP modes. Each mode utilizes the buffer in a unique way. The following subsections provide detailed explanations of these buffered modes.

24.5.5.1 FIFO mode

The FIFO mode provides a Tx FIFO for transmitting data and an Rx FIFO for receiving data. Each FIFO is constructed from the SRAM buffer. The FIFOs can be configured as 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 through the `SCBx_CTRL.MEM_WIDTH` register. For I²C, it is recommended to use the FIFO in BYTE mode because all transactions are byte-wide.

The FIFO mode operates in Active and Sleep power modes. In Deep Sleep power mode, only a Deep Sleep-capable SCB can wake the device from sleep through a slave address match. For more information, see Deep Sleep to Active mode transition in FIFO buffer mode.

A write operation to the transmit FIFO uses the `SCBx_TX_FIFO_WR` register, while a read operation 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. In debug mode, a read operation from the status register behaves as a read from the SCBx_RX_FIFO_RD_SILENT register. In this case, data is not removed from the FIFO.

A programmable threshold can be defined to trigger events based on the number of FIFO entries. Triggers are generated under the following conditions:

- For the 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 receive FIFO exceeds TRIGGER_LEVEL

Several interrupt status bits indicate FIFO states, such as whether the FIFOs are full, empty, or have reached other critical conditions.

24.5.5.1.1 Deep Sleep to Active mode transition in FIFO buffer mode

When SCBx_CTRL.EC_AM_MODE = 1, SCBx_CTRL.EC_OP_MODE = 0, and FIFO mode is enabled, the following descriptions apply only 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 control. Once the logic takes over, the address is acknowledged (ACK'd), and the master can begin writing data. Before entering Deep Sleep, disable CLK_SCB. Upon waking from Deep Sleep, re-enable CLK_SCB. At this point, the clock stretch is 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 control and the CPU writes either SCBx_I2C_S_CMD.S_ACK or SCBx_I2C_S_CMD.S_NACK. Before entering Deep Sleep, disable CLK_SCB. Upon waking from Deep Sleep, re-enable CLK_SCB 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 not acknowledged (NACK'd) until the internally-clocked logic takes control. When the internally-clocked logic takes over, the internal clock may not be at the correct frequency due to PLL/FLL locking times. This can result in incorrect timing on the I^2C bus for the ACK/NACK. To prevent this, disable CLK_SCB before entering Deep Sleep and re-enable it 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 control and the CPU writes data into the Tx FIFO. Before entering Deep Sleep, disable CLK_SCB. Upon waking from Deep Sleep, re-enable CLK_SCB 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 not acknowledged (NACK'd) until the internally-clocked logic takes control. When the internally-clocked logic takes over, there is no guarantee that the internal clock will operate at the correct frequency due to PLL/FLL locking times. This can result in incorrect timing on the I^2C bus for the ACK/NACK. To prevent this, disable CLK_SCB before entering Deep Sleep and re-enable it only after the PLL/FLL have stabilized

***Note**: When performing a repeated START after a write operation, 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. Failing to do so will result in the address in the FIFO being sent as data rather than as an address.*

24.5.5.2 EZI²C mode

The EZI²C protocol is a unique communication scheme developed by Infineon, based on the I²C protocol. It implements a meta-protocol on top of the standard I²C protocol to enable indexed memory transfers between an I²C master and slave without requiring CPU intervention.

The EZI²C protocol defines a single memory buffer on the slave device, indexed with an 8-bit address. This buffer supports up to 256 entries, with each entry being 8 bits wide. The EZ address is used to access these 256 locations. The CPU interacts with the memory buffer using the EZ_DATA registers, which handle word accesses but only use the least significant byte of the word.

The slave interface accesses the memory buffer using the current address. At the start of a transfer (I²C START or RESTART), the base address is copied to the current address. Each data-element write or read operation targets the current address location, and after each access, the current address is incremented by one.

When the current address reaches the last memory buffer address (255), it is not incremented further. Any subsequent write operations overwrite the last address's data, while subsequent read operations return the same value as the last address. The I²C bus master must be aware of the memory buffer capacity in EZ mode to avoid unintended behavior.

The I²C base and current addresses are available through the I2C_STATUS register. At the end of a transfer, the difference between the base and current addresses indicates the number of read or write accesses performed. Interrupt cause fields are provided to identify the end of a transfer.

The EZI²C protocol can be implemented using firmware or hardware. All SCBs support EZI²C through a firmware-based implementation in both Active and Sleep power modes. SCBs capable of Deep Sleep mode support a hardware-and firmware-based EZI²C implementation in Deep Sleep power mode.

The EZI²C protocol includes three operation phases:

- **Address phase**: The master transmits an 8-bit address to the slave. This address becomes the slave's base and current address
- **Write phase**: The master writes one or more 8-bit data elements to the slave's memory buffer. The slave sets its current address to the base address, writes received data elements to the current address location, and increments the current address after each write
- **Read phase**: The master reads one or more 8-bit data elements from the slave's memory buffer. The slave sets its current address to the base address, retrieves data elements from the current address location, and increments the current address after each read

***Note**: A slave's base address is updated by the master, not by the CPU.*

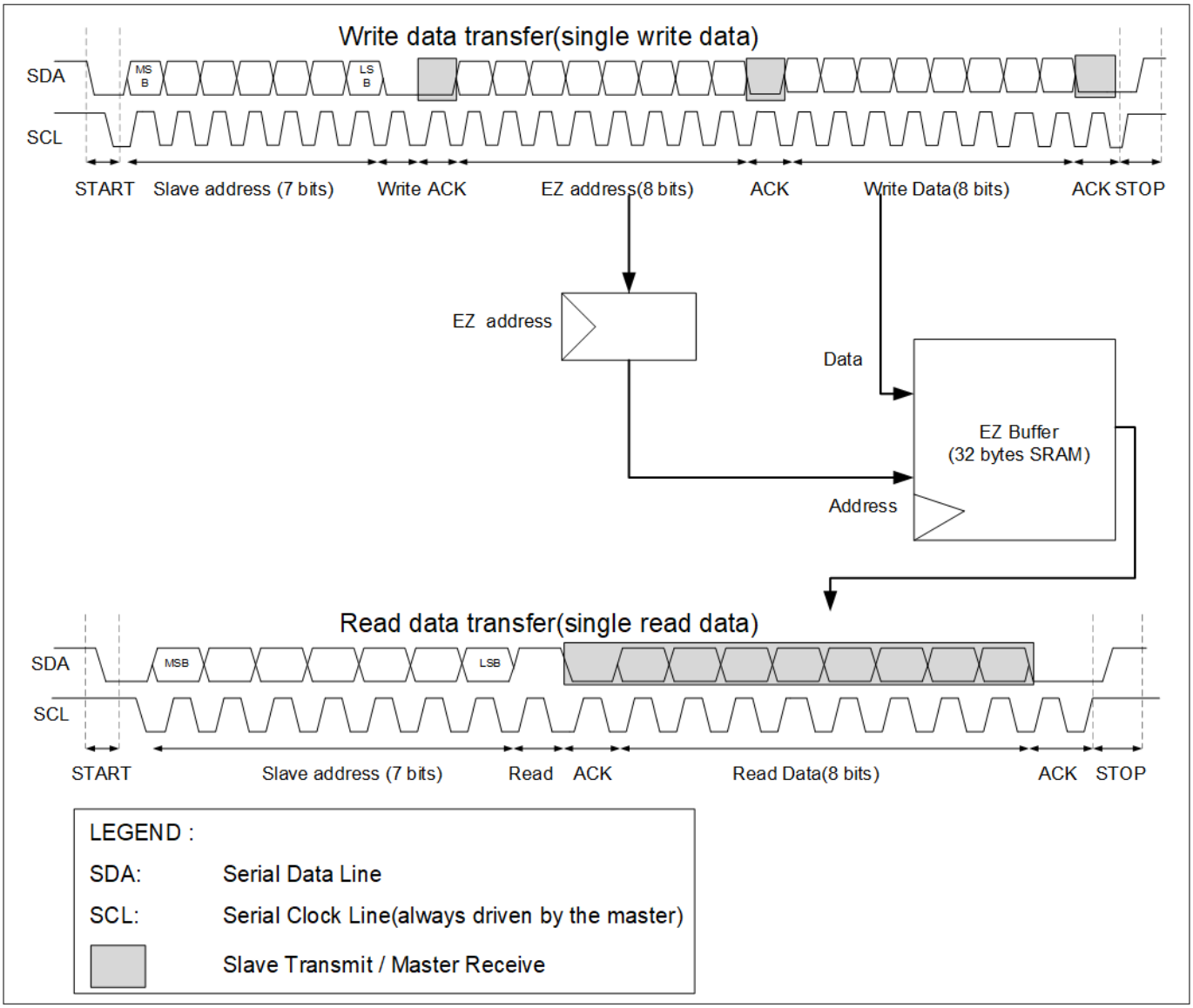


Figure 278 EZI²C write and read data transfer

24.5.5.2.1 Deep Sleep to 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 control. Once the logic takes over, the address is acknowledged (ACK'd), and the master can begin writing data. Before entering Deep Sleep, disable `CLK_SCB`. Upon waking from Deep Sleep, re-enable `CLK_SCB`. At this point, the clock stretch is released
- `S_NOT_READY_ADDR_NACK = 1`, `SCBx_I2C_CTRL.S_READY_ADDR_ACK = x`: The incoming address is not acknowledged (NACK'd) until the internally-clocked logic takes control. When the internally-clocked logic takes over, there is no guarantee that the internal clock will operate at the correct frequency due to PLL/FLL locking times. To avoid timing issues on the I²C bus for the ACK/NACK, disable `CLK_SCB` before entering Deep Sleep and re-enable it after the PLL/FLL have stabilized

24.5.5.3 Command-response mode

This mode uses a single memory buffer with a base read address, a current read address, a base write address, and a current write address to index the memory buffer. The base addresses are provided by the CPU.

The slave uses the current addresses to index the memory buffer for sequential access. The memory buffer holds 256 8-bit data elements. Both the base and current addresses are within the range [0 to 255].

The CPU reads from and writes to the memory buffer using the SCBx_EZ_DATA registers. These are word accesses, but only the least significant byte of the word is used.

The slave interface uses the current addresses to access the memory buffer. At the start of a write transfer (I^2C START or RESTART), the base write address is copied to the current write address. A data-element write is performed at the current write address location, and the current write address is incremented by 1 after the operation. During a read transfer, the base read address is copied to the current read address at the start of the transaction. A data-element read is performed from the current read address location, and after the data element is transmitted, the current read address is incremented by 1.

When the current addresses reach the last memory buffer address (255), they are not incremented further. Subsequent write operations overwrite any previously written value at the last address. Subsequent read operations continue to return the same value from the last address. The bus master must be aware of the memory buffer's capacity in Command-Response mode to avoid unintended behavior.

Base addresses are configured through the SCBx_CMD_RESP_CTRL register, and current addresses can be viewed in the SCBx_CMD_RESP_STATUS register. At the end of a transfer (I^2C STOP condition), the difference between the base and current addresses indicates the number of read/write operations performed. Interrupts to signal the end of a transfer are available and can be found in the SCBx_INTR_I2C_EC and SCBx_INTR_SPI_EC registers.

The command-response mode operates in both Active and Deep Sleep power modes. It comprises two distinct phases of operation:

- **Write phase**:
  - The write phase begins with a START or RESTART condition, followed by the slave address with the read/write bit set to ‘0’ to indicate a write operation
  - The slave’s current write address is set to the base write address
  - Received data elements are written to the current write address location
  - After each write operation, the current write address is incremented
- **Read phase**:
  - The read phase begins with a START or RESTART condition, followed by the slave address with the read/write bit set to ‘1’ to indicate a read operation
  - The slave’s current read address is set to the base read address
  - Transmitted data elements are read from the current read address location
  - After each transmitted data element, the current read address is incremented

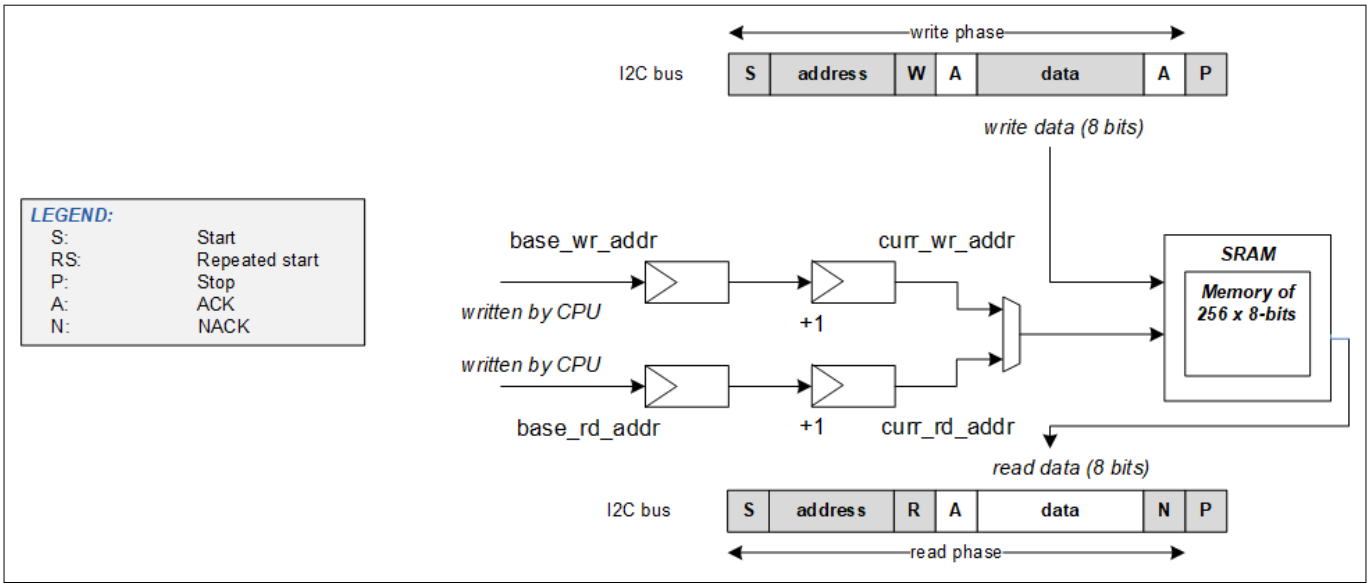


Figure 279 I²C command response mode

***Note**: A slave's base address is updated by the CPU, not by the master.*

24.5.6 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. The SCBx_CTRL.EC_AM_MODE bitfield indicates whether I²C address matching is internally clocked (0) or externally clocked (1). Address matching represents the first phase of the I²C protocol. The SCBx_CTRL.EC_OP_MODE bitfield indicates whether the rest of the I²C protocol operation, apart from address matching, is internally clocked (0) or externally clocked (1). Externally clocked operation is supported only in I²C slave mode.

Internally clocked operation uses programmable clock dividers. For I²C, an integer clock divider must be used for both master and slave modes. For additional details on system clocking, see Clocking system. The internally clocked mode does not support command-response mode.

***Note**: In CYW20829, only one SCB supports the externally clocked mode of operation.*

The configurations of SCBx_CTRL.EC_AM_MODE and SCBx_CTRL.EC_OP_MODE are as follows:

- **SCBx_CTRL.EC_AM_MODE = 0 and SCBx_CTRL.EC_OP_MODE = 0**: Use this configuration for Active mode only
  - **FIFO mode**: Supported
  - **EZ mode**: Supported
  - **CMD_RESP mode**: Not supported. The slave NACKs every address
- **SCBx_CTRL.EC_AM_MODE = 1 and SCBx_CTRL.EC_OP_MODE = 0**: Use this configuration for both Active and Deep Sleep functionalities. This setting uses externally clocked functionality for I²C address matching and internally clocked functionality to access the memory buffer. A "hand over" from external to internal clocking occurs using either an ACK/NACK or clock stretching scheme.
  - The ACK/NACK scheme may terminate the current transfer and requires the master to retry the transfer
  - The clock stretching scheme stretches the current transfer after receiving a matching address. This approach requires the master to support either NACK generation (and retry) or clock stretching. When the I²C address is matched, the SCBx_INTR_I2C_EC.WAKE_UP field is set to 1. The associated Deep Sleep functionality interrupt brings the system into Active power mode
  - **FIFO mode**: See FIFO mode

- **EZ mode**: See EZI²C mode
- **CMD_RESP mode**: Not supported. The slave NACKs every address

- **SCBx_CTRL.EC_AM_MODE = 1 and SCBx_CTRL.EC_OP_MODE = 1**: Use this configuration for both Active and Deep Sleep functionalities. When the slave is selected, the SCBx_INTR_I2C_EC.WAKE_UP field is set to 1, and the associated Deep Sleep functionality interrupt brings the system into Active power mode. When the slave is deselected, the SCBx_INTR_I2C_EC.EZ_STOP and/or SCBx_INTR_I2C_EC.EZ_WRITE_STOP fields are set to 1
  - **FIFO mode**: Not supported
  - **EZ mode**: Supported
  - **CMD_RESP mode**: Supported

Table 179 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

An externally clocked operation uses a clock provided by the serial interface. The externally clocked mode does not support FIFO mode. When SCBx_CTRL.EC_OP_MODE is set to '1', the external interface logic accesses the memory buffer on the external interface clock (I²C SCL). This enables functionality for EZ and CMD_RESP modes in both Active and Deep Sleep power modes.

In Active power mode, the memory buffer requires arbitration between the external interface logic (on the I²C SCL) and the CPU interface logic (on the system peripheral clock). Arbitration always gives the highest priority to the external interface logic (host accesses). For I²C, the external interface logic takes one serial interface clock period per bit. During this period, internal logical access to the memory buffer is denied service. CYW20829 provides two programmable options to handle this "denial of service":

- **SCBx_CTRL.BLOCK = 1**: Internal logical access to the memory buffer is blocked until the buffer is granted access and the external interface logic completes its operation
  - For a 100-kHz I²C interface, the maximum blocking period for one serial interface bit period is approximately 10 µs (208 clock cycles on a 48 MHz SCB input clock)
  - This option maintains normal SCB register functionality, but the blocking time introduces additional internal bus wait states
- **SCBx_CTRL.BLOCK = 0**: Internal logical access to the memory buffer is not blocked but fails if it conflicts with access from the external interface logic
  - A read access failure returns the value 0xFFFF:FFFF
  - A write access failure is ignored. This option does not introduce additional internal bus wait states, but memory buffer access may not take effect. Failure detection is handled as follows:
    - **Read failure**: Easily detected, as failed memory buffer reads return a unique value of 0xFFFF:FFFF. Non-failed accesses always return an unsigned byte value in the range 0x0000:0000 to 0x0000:00FF
    - **Write failure**: Detected by reading the memory buffer location and verifying that the read value matches the written value

For both options, any conflicting internal logical access to the memory buffer sets the SCBx_INTR_TX.BLOCKED field to '1' for write accesses, and the SCBx_INTR_RX.BLOCKED field to '1' for read accesses. These fields can be used as status indicators or as interrupt cause fields when their associated mask fields are enabled.

When a series of read or write accesses is performed and SCBx_CTRL.BLOCK = 0, failure detection can be handled as follows:

- Perform a logical OR operation on all read values and compare the result to 0xFFFF:FFFF to detect a read failure
- Check the SCBx_INTR_TX.BLOCKED and SCBx_INTR_RX.BLOCKED fields to determine whether a failure occurred during the series of write or read operations

24.5.6.1 Glitch filtering

The CYW20829 SCB I²C includes both analog and digital glitch filters to ensure signal integrity.

- **Analog glitch filters**:
  - Analog glitch filters are applied to 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 to the i2c_sda_out output signal (AF_out)
  - Analog glitch filters are enabled and disabled using the SCBx_I2C_CFG register
  - Only the _SEL bitfields in this register should be modified. The _TRIM bitfields must not be changed
- **Digital glitch filters**:
  - Digital glitch filters are applied to the i2c_scl_in and i2c_sda_in input signals (DF_in)
  - The digital glitch filter is enabled using the SCBx_RX_CTRL.MEDIAN register

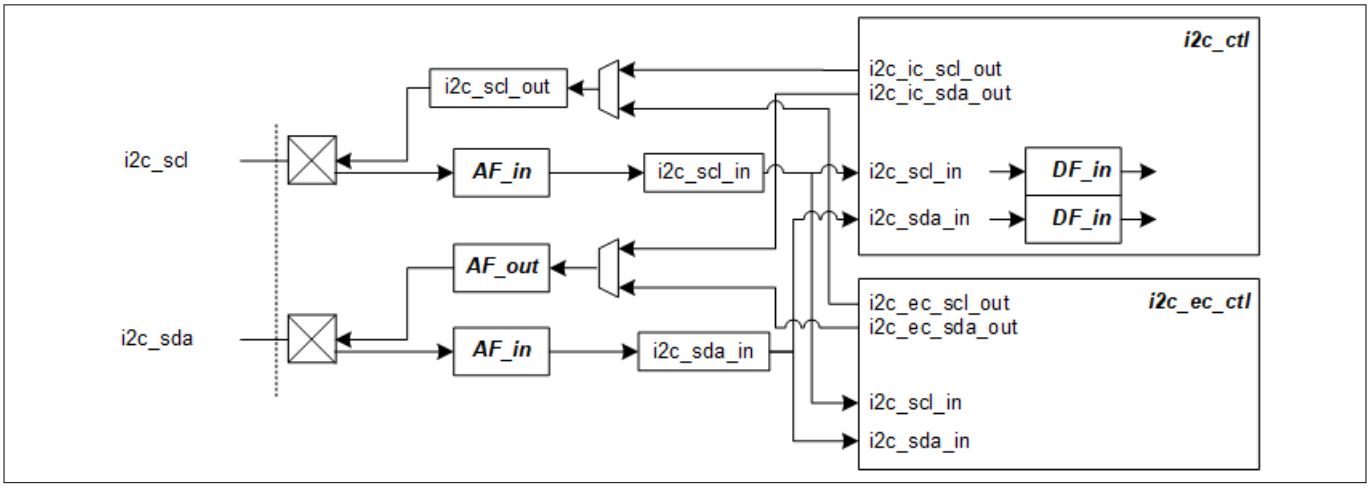


Figure 280 I²C glitch filtering connection

Table 180 and Table 181outlines the recommended combinations of glitch filters.

Table 180 Glitch filter combination 1

AF_in	AF_out	DF_in	Comments
0	0	1	Used in internally clocked mode while operating as master in fast-mode plus (1-MHz speed mode)
1	0	0	Used in internally clocked mode when SCBx_CTRL.EC_OP_MODE is set to '0'
1	1	0	Used in externally clocked mode when SCBx_CTRL.EC_OP_MODE is set to '1'. Applicable only in 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:

Table 181 Glitch filter combination 2

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

24.5.6.2 Oversampling and bit rate

Internally-clocked master

CYW20829 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, within the device, and in the block (including analog and digital glitch filters) contribute to the signal interface timing.

In master mode, the block operates using CLK_SCB and utilizes programmable oversampling factors for the SCL high and low phases. These are configured using the SCBx_I2C_CTRL.HIGH_PHASE_OVS and SCBx_I2C_CTRL.LOW_PHASE_OVS registers, respectively.

Table 182 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 182 assumes worst-case conditions on the I²C bus. The following equations can be used to calculate the settings for a specific system. These calculations require measuring the rise and fall times on the SCL and SDA lines in the system.

- tCLK_SCB(Min):
  tCLK_SCB(Min) = (tLOW + tRF)/SCBx_I2C_CTRL.LOW_PHASE_OVS
  - If CLK_SCB operates faster than this value, the tLOW of the I²C specification will be violated
  - The fall time (tF) must be measured in the system
- tCLK_SCB(Max) (when analog and digital filters are configured as follows):
  - **Analog filter enabled and digital filter disabled**:
    tCLK_SCB(Max) = (tVD – tRF – 100 ns)/3
  - **Analog filter disabled and digital filter enabled**:
    tCLK_SCB(Max) = (tVD – tRF)/4
    - tRF is the maximum of either the rise time or the fall time
    - If CLK_SCB operates slower than this frequency, the tVD requirement will be violated

Internal-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, slave mode does not use programmable oversampling factors. Table 183 assumes worst-case conditions on the I²C bus, including the chip's internal delay.

Table 183 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) (Maximum clock period for CLK_SCB):
  - **When the analog filter is enabled and the digital filter is disabled**:
    tCLK_SCB(Max) = (tVD – tRF – 100 ns)/3
  - **When the analog filter is disabled and the digital filter is enabled**:
    tCLK_SCB(Max) = (tVD – tRF)/4
    - tRF is the greater of the rise time or fall time
    - If CLK_SCB operates slower than this frequency, the tVD requirement will be violated
- tCLK_SCB(Min) (Minimum clock period for CLK_SCB): The minimum period of CLK_SCB is determined using the following equation:
  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 requirements provided in Table 182 and Table 183.

24.5.7 Loopback

In master-slave mode, the SCB supports internal SCL and SDA lines, which are routed internally within the peripheral. As a result, the internal lines are unaffected by other I²C devices on the bus. This mode is configured using the SCBx_I2C_CTRL.LOOPBACK register.

24.5.8 Enabling and initializing I²C

24.5.8.1 Configuring for I²C FIFO mode

The I²C interface must be programmed in the following order:

1. Program protocol-specific information using the SCBx_I2C_CTRL register:
   a. Select master or slave functionality (MASTER_MODE, SLAVE_MODE)

2. Program 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 the most significant bit (MSb) is the first bit to be transmitted or received (MSB_FIRST = 1)
3. Set the SCBx_CTRL.MEM_WIDTH to 1 to enable byte mode
4. Program the transmitter and receiver FIFOs using the SCBx_TX_FIFO_CTRL and SCBx_RX_FIFO_CTRL registers:
   a. Set the trigger level (TRIGGER_LEVEL)
   b. Clear the transmitter and receiver FIFOs and shift registers (CLEAR)
5. Program the SCBx_CTRL register to:
   a. Enable the I²C block
   b. Select the I²C mode

24.5.8.2 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 1 to the SCBx_CTRL.CMD_RESP register
3. Configure the following bits in the SCBx_I2C_CTRL register:
   a. Set SCBx_I2C_CTRL.S_READY_ADDR_ACK (bit 12)
   b. Set SCBx_I2C_CTRL.S_READY_DATA_ACK (bit 13)

24.5.9 I/O pad connection

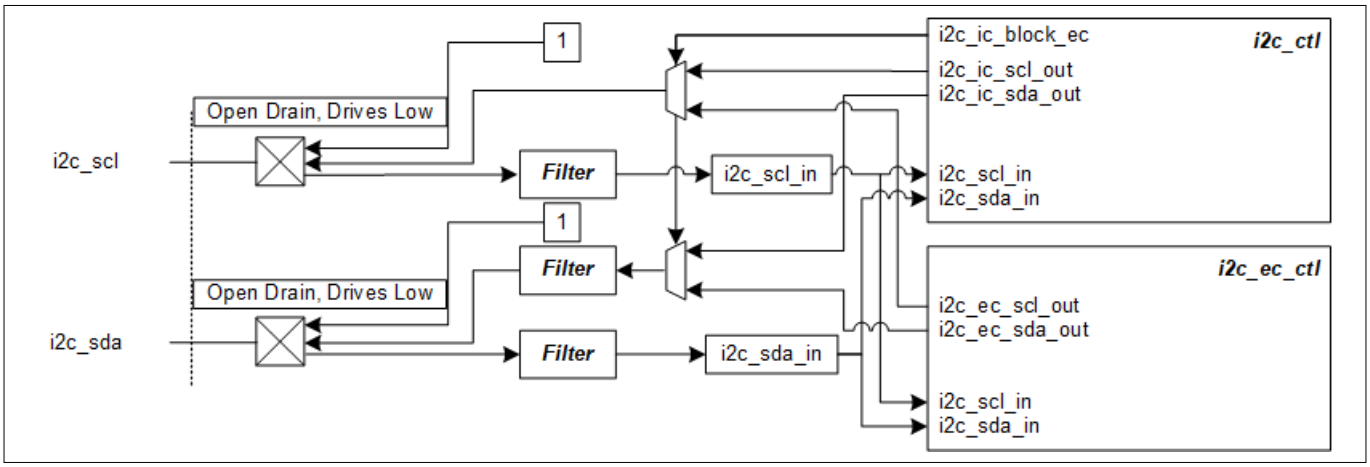


Figure 281 I²C I/O pad connection

Table 184 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_sda	Open drain drives low	i2c_sda_in	Receive data
		i2c_sda_out	Transmit data

24.5.10 Registers summary

The I²C interface is controlled through a set of configuration, control, and status registers. For detailed information about these registers, see the CYW20829 registers reference manual.

24.6 SCB interrupts

The SCB supports interrupt generation for various events. The interrupts generated by the SCB block depend on the mode of operation.

Table 185 SCB interrupts

Interrupt	Functionality	Active/Deep Sleep	Registers
interrupt_master	Master and SPI master functionality	Active	• SCBx_INTR_M • SCBx_INTR_M_SET • SCBx_INTR_M_MASK • SCBx_INTR_M_MASKED
interrupt_slave	I²C slave and SPI slave functionality	Active	• SCBx_INTR_S • SCBx_INTR_S_SET • SCBx_INTR_S_MASK • SCBx_INTR_S_MASKED
interrupt_tx	UART transmitter and Tx FIFO functionality	Active	• SCBx_INTR_TX • SCBx_INTR_TX_SET • SCBx_INTR_TX_MASK • SCBx_INTR_TX_MASKED
interrupt_rx	UART receiver and Rx FIFO functionality	Active	• SCBx_INTR_RX • SCBx_INTR_RX_SET • SCBx_INTR_RX_MASK • SCBx_INTR_RX_MASKED
interrupt_i2c_ec	Externally-clocked I²C slave functionality	Deep Sleep	• SCBx_INTR_I2C_EC • SCBx_INTR_I2C_EC_MASK • SCBx_INTR_I2C_EC_MASKED
interrupt_spi_ec	Externally-clocked SPI slave functionality	Deep Sleep	• SCBx_INTR_SPI_EC • SCBx_INTR_SPI_EC_MASK • SCBx_INTR_SPI_EC_MASKED

24.6.1 SPI interrupts

SPI interrupts are classified into master interrupts, slave interrupts, Tx interrupts, Rx interrupts, and externally-clocked (EC) mode interrupts. Each interrupt output represents the logical OR of all possible interrupt sources within its group. For example, the Tx interrupt output is the logical OR of all possible Tx interrupt sources. This signal goes high when any of the enabled Tx interrupt sources are active.

The SCB provides an interrupt cause register (SCBx_INTR_CAUSE) that can be used to determine the interrupt source. Interrupt registers are cleared by writing 1 to the corresponding bit fields. Certain interrupt sources may be triggered repeatedly as long as the triggering condition is active, even if the interrupt source has been cleared. For instance, the TX_FIFO_EMPTY interrupt is asserted as long as the transmit FIFO is empty, even after the interrupt source is cleared. For additional details on interrupt registers, see the CYW20829 registers reference manual.

The SPI supports interrupts for 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 (SPI_EZ_WRITE_STOP)
  - SPI slave deselected after any EZSPI transfer (SPI_EZ_STOP)
  - SPI bus error slave deselected unexpectedly during an SPI transfer. Firmware may choose to clear the Tx and Rx FIFOs for this error (SPI_BUS_ERROR
- **SPI Tx interrupts (SCBx_INTR_TX)**:
  - Tx FIFO contains 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 Rx interrupts (SCBx_INTR_RX)**:
  - Rx FIFO contains 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
- **SPI externally-clocked interrupts (SCBx_INTR_SPI_EC)**:
  - Wakeup request on slave select (WAKE_UP)
  - SPI STOP detection at the end of any transfer (EZ_STOP)
  - SPI STOP detection at the end of a write transfer (EZ_WRITE_STOP)
  - SPI STOP detection at the end of a read transfer (EZ_READ_STOP)

24.6.2 UART interrupts

UART interrupts are classified into Tx interrupts and Rx interrupts. Each interrupt output represents the logical OR of all possible interrupt sources within its group. For example, the Tx interrupt output is the logical OR of all possible Tx interrupt sources. This signal goes high when any of the enabled Tx interrupt sources are active.

The SCB provides an interrupt cause register (SCBx_INTR_CAUSE) that identifies the interrupt source. Interrupt registers are cleared by writing 1 to the corresponding bit fields. Certain interrupt sources may be triggered repeatedly as long as the triggering condition is active, even if the interrupt source has been cleared. For instance, the Tx_FIFO_EMPTY interrupt is asserted as long as the Tx FIFO is empty, even after the interrupt source is cleared. For more details on interrupt registers, see the CYW20829 registers reference manual.

The UART block generates interrupts for the following events:

UART Tx interrupts (SCBx_INTR_TX)

- Tx FIFO contains 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 operation completed when the UART finishes transferring all data in the Tx FIFO and the last stop bit 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 interrupts (SCBx_INTR_RX)

- Rx FIFO contains 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 the received data frame (FRAME_ERROR)
- Parity error in the received data frame (PARITY_ERROR)
- LIN baud rate detection completed (BAUD_DETECT)
- Successful detection of a LIN break (BREAK_DETECT)

24.6.3 I²C interrupts

I²C interrupts are classified into master interrupts, slave interrupts, Tx interrupts, Rx interrupts, and externally-clocked (EC) mode interrupts. Each interrupt output represents the logical OR of all possible interrupt sources within its group. For example, the Tx interrupt output is the logical OR of all possible Tx interrupt sources. This signal goes high when any of the enabled Tx interrupt sources are active.

The SCB provides an interrupt cause register (SCBx_INTR_CAUSE) that can be used to determine the interrupt source. Interrupt registers are cleared by writing 1 to the corresponding bit field. Certain interrupt sources may be triggered repeatedly as long as the triggering condition is met, even if the interrupt source has been cleared. For instance, the TX_FIFO_EMPTY interrupt is asserted as long as the Tx FIFO is empty, even after the interrupt source is cleared. For more detailed information on interrupt registers, see the CYW20829 registers reference manual.

The I²C block generates interrupts for the following conditions:

- **I²C master interrupts (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 or START condition detected (I2C_BUS_ERROR)
- **I²C slave interrupts (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 received general call address (I2C_GENERAL)
  - I²C bus error, unexpected STOP or START condition detected (I2C_BUS_ERROR)

- **I²C Tx interrupts (SCBx_INTR_TX)**:
  - Tx FIFO contains 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) is blocked (BLOCKED)
- **I²C Rx interrupts (SCBx_INTR_RX)**:
  - Rx FIFO contains 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 interrupts (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)

25 PDM-PCM converter

25.1 Overview

The pulse density modulation–pulse code modulation (PDM–PCM) converter converts PDM data into PCM data. The PDM–PCM converter receives the serial data stream from external digital PDM microphones.

PDM is best described as oversampled 1-bit audio, as it consists of a high-sampling-rate, single-bit digital signal. In contrast, most modern digital audio systems use multi-bit PCM to represent audio signals. PCM offers the advantage of being easy to manipulate, enabling signal processing operations such as mixing, filtering, and equalization on the audio stream.

PDM, which uses only one bit to convey audio data, is simpler in both concept and execution compared to PCM. This simplicity makes PDM a popular method for delivering audio from microphones to the signal processor.

25.2 Features

- Supports up to two PDM receivers
- Supports mono and stereo configurations
- Half-rate sampling to reduce system power consumption
- Provides multiple filtering options, including CIC filter, FIR filter, and DC-blocking filter, in the signal processing path to reduce noise and achieve the desired decimation
- Features a programmable interface clock
- Allows configuration of programmable FIR filter coefficients
- Enables programmable CIC and FIR filter decimation
- Includes a programmable DC-blocking coefficient
- Supports programmable PCM sample sizes of 8, 10, 12, 14, 16, 18, 20, 24, and 32 bits
- Offers programmable sampling delay to accommodate varying master-slave-master round-trip delays
- Includes a 64-entry Rx FIFO with interrupt and trigger support
- Implements a common data path with SRAM, FIR filter, and DC-blocking filter logic shared among all PDM receivers

25.3 Architecture

The PDM-PCM converter features two PDM receivers or channels, allowing two PDM sources or microphones to interface with the two PDM receivers of the converter. It simplifies the conversion process by exposing various configuration settings as programmable registers, which users can adjust to meet application-specific requirements.

The entire PDM-PCM conversion process is managed in hardware, ensuring efficient operation. Figure 282 shows the block diagram of the PDM-PCM converter. See the device datasheet for details on port pin-assignments for the PDM block signals, electrical specifications, and the interrupt vector number.

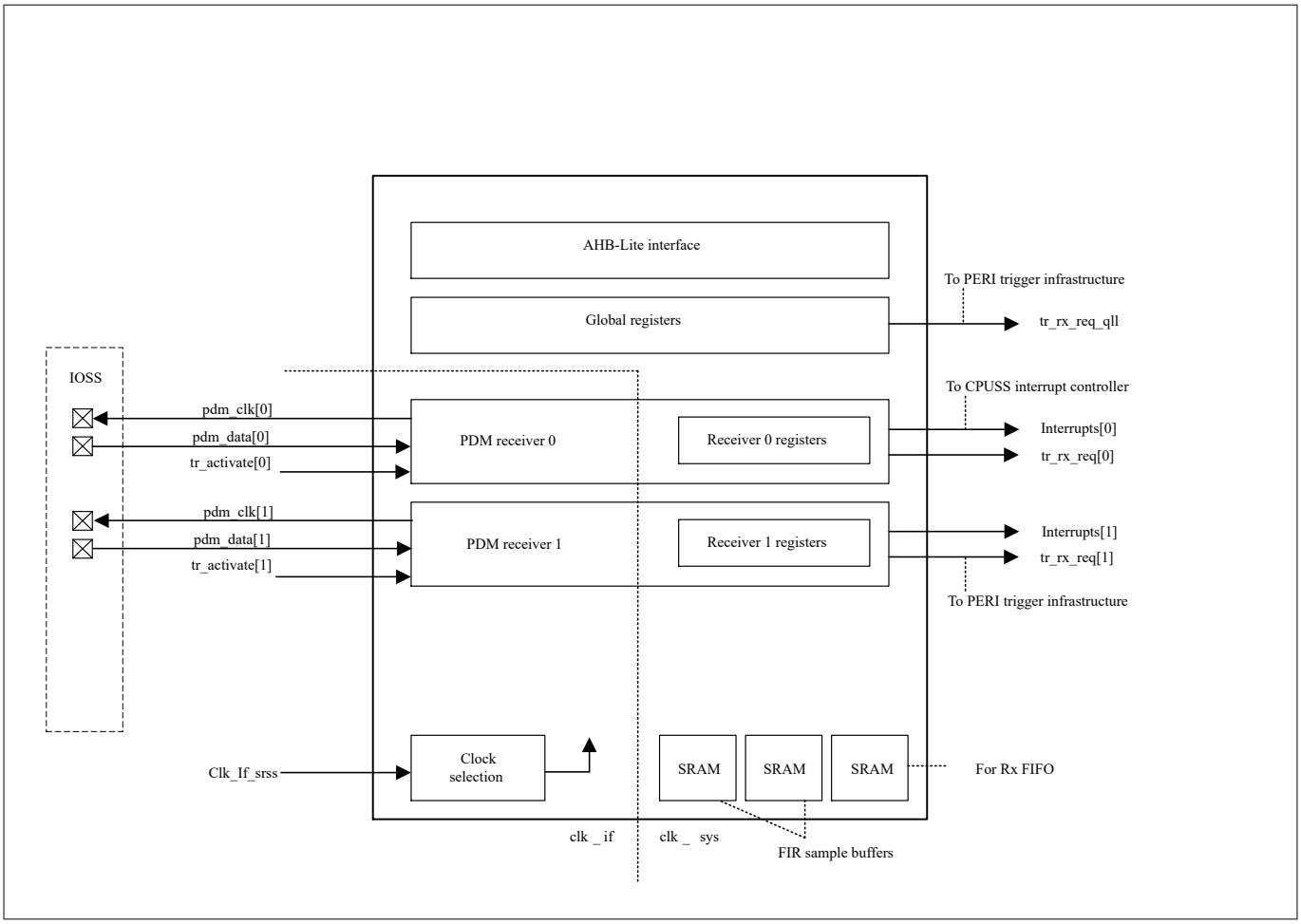


Figure 282 Block diagram

The PDM bits are oversampled, and the received PDM bits are processed to generate PCM samples. The converted PCM samples are further processed using a series of filters, which can be enabled through the CIC_CTL, FIR0_CTL, and FIR1_CTL registers specific to each PDM receiver. The processed PCM samples are stored in the Rx FIFO of each PDM receiver. A CPU or DMA can then read the PCM samples from the Rx FIFO.

The PDM-PCM receiver functions as a master receiver that generates the clock (pdm_clk) and receives PDM data on the pdm_data line. The PDM sources, acting as interface slaves, drive the pdm_data line. Typically, the PDM sources are digital microphones.

In the PDM-PCM converter, the block registers are divided into the following categories:

- **Global registers**: Common to all PDM receivers, these registers control the activation of specific PDM receivers, the common pdm_clk, the operating mode, and the FIR coefficient used by every PDM receiver
- **Receiver-specific registers**: These registers control filtering, Rx FIFO configuration, and PCM sample formatting

The PDM receiver is enabled using the ENABLED bit in the receiver-specific CTL register. It can be disabled when not in use to save power. When the CTL.ENABLED bit is set to 0 (disabled), all SRAMs enter a low-power retention state, and the non-retention registers lose their values. When the PDM receiver is enabled again, the non-retention registers are reset to their default values. Before enabling the receiver and activating PDM-to-PCM conversion, user firmware must ensure that all non-retention registers are configured as required.

The PDM receiver is activated by either:

- Setting the receiver-specific activation bit in the CTL or CTL_SET registers
- A rising edge (0 to 1 transition) on a receiver's tr_activate input trigger, which sets the ACTIVE bit specific to the PDM receiver in the global CTL register

The default reset condition for a PDM receiver is disabled. For proper operation, the receiver must be fully configured before enabling. The typical steps required to configure and start a PDM receiver are as follows:

- Configure the global registers, such as CLOCK_CTL, ROUTE_CTL, TEST_CTL, and FIR*_COEFF*, with the desired settings
- Program the FIFO trigger level, PCM word size, sample delay, CIC decimation, FIR decimation and scale, and DC-blocking filter coefficient
- Enable the receiver
- Activate the receiver

25.3.1 Operating modes

The CYW20829 device supports two PDM receivers, allowing a maximum of two PDM microphones to interface with the PDM-PCM converter. These receivers are identified as:

- PDM receiver 0
- PDM receiver 1

The two PDM receivers can be considered as a pair, enabling flexibility in the connection of pdm_data[] lines. The pdm_data[] lines can either be swapped or shared within this pair:

- **Swapped scenario**: The pdm_data[0] line can be connected to PDM receiver 1, while the pdm_data[1] line can be connected to PDM receiver 0
- **Shared scenario**: The pdm_data[] line is shared between the two receivers, requiring only a single IOSS pin

25.3.1.1 Mono mode

The PDM-PCM converter block supports two operating modes: mono and stereo. The mono mode includes the following subtypes:

- Single mono mode
- Dual mono mode

The operating modes are easily understood when considering the PDM receiver pair. The user can configure the desired mode through the ROUTE_CTL register.

- **Single mono mode**: A single PDM microphone is interfaced with a single PDM receiver, N, using either the pdm_data[N] or pdm_data[N^1] line. Here, N is any number from 0 to 1

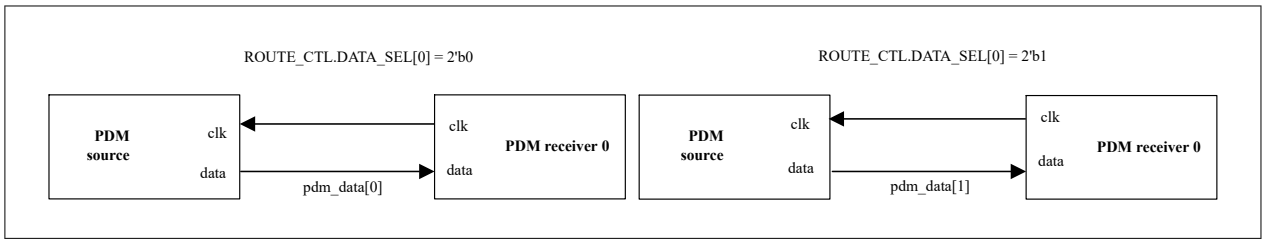


Figure 283 Single mono mode

- **Dual mono mode**: Two PDM microphones are interfaced with PDM receivers 0 and 1. In this configuration, as the pdm_data[] line is not shared between PDM receivers 0 and 1, separate IOSS pins are required
  - **Normal mode**: The pdm_data[0] and pdm_data[1] data lines are connected to PDM receivers 0 and 1, respectively
  - **Swapped mode**: The pdm_data[1] and pdm_data[0] data lines are connected to PDM receivers 0 and 1, respectively. In this configuration, the data lines are swapped between PDM receivers 0 and 1

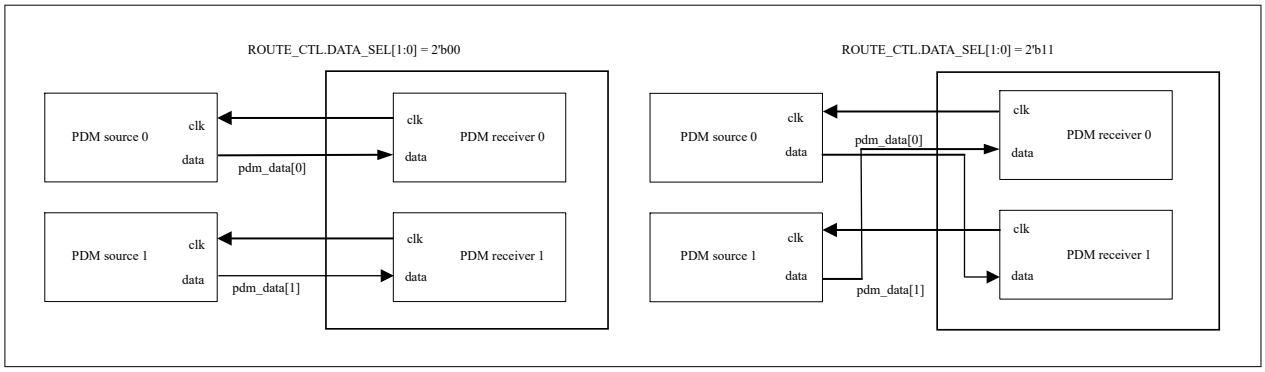


Figure 284 Dual mono mode

25.3.1.2 Stereo mode

In stereo mode, both PDM receivers within the pair are active, allowing two PDM microphones to interface with PDM receivers 0 and 1. In this configuration, the data lines are shared between the two PDM receivers in a pair, as shown in Figure 285. In a stereo configuration, the PDM sources drive the same data line in a time-sharing fashion. Based on the `pdm_data[]` line used, the stereo mode is further categorized into the following two modes:

- **Stereo mode 1**: The `pdm_data[0]` line is shared by PDM receivers 0 and 1
- **Stereo mode 2**: The `pdm_data[1]` line is shared by PDM receivers 0 and 1

To configure the PDM–PCM converter for the desired mode, use the `ROUTE_CTL` global register. This register specifies which IOSS data input signal (`pdm_data[]` line) is routed to each PDM receiver within the pair. Each PDM receiver `N` has a dedicated 1-bit control field in the `ROUTE_CTL` global register. The PDM receiver `N` uses the `DATA_SEL[N]` bit of the `ROUTE_CTL` global register for selection. The 1-bit `DATA_SEL[N]` field functions as follows:

- '0': PDM receiver `N` uses the data input signal `pdm_data[N]`. This is the default configuration
- '1': PDM receiver `N` uses the data input signal `pdm_data[N ^ 1]`, where the lower bit of the PDM receiver index is inverted

For example:

- If `DATA_SEL` is `0b10` and only the first pair is enabled, both PDM receivers 0 and 1 use the `pdm_data[0]` data line
- If `DATA_SEL` is `0b11`, PDM receivers 0 and 1 use the `pdm_data[1]` and `pdm_data[0]` data lines, respectively
- In the default configuration, where `DATA_SEL` is `0b000000`, PDM receivers 0 and 1 use the `pdm_data[0]` and `pdm_data[1]` data lines, respectively

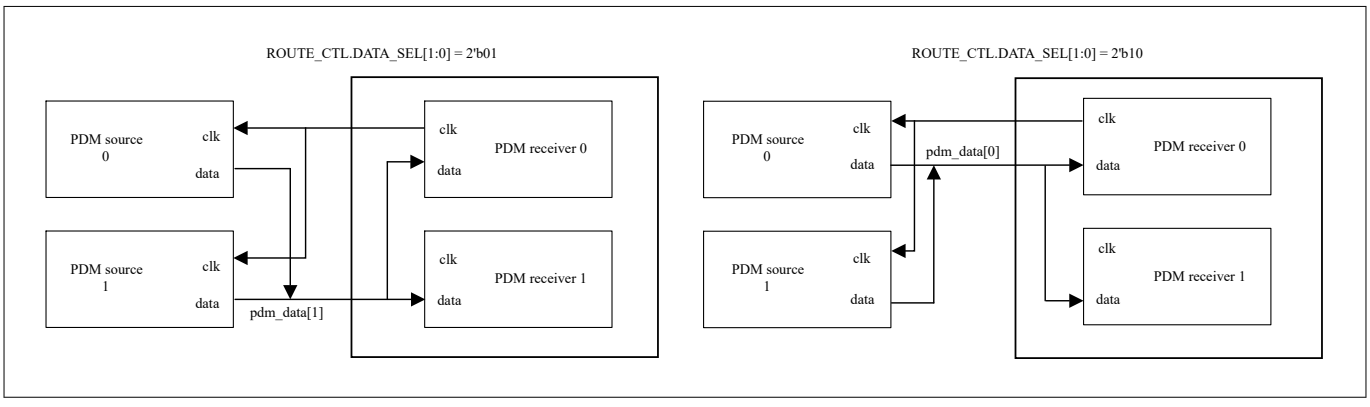


Figure 285 Stereo mode

25.3.2 Clocking

The PDM-PCM converter can be clocked using either an internal clock or an external clock. The PDM interface clock (clk_if) is derived from one of the following sources:

- An internal clock source, the SRSS clock (clk_if_srss)
- An external clock source provided on the pdm_data[0] or pdm_data[1] line

Figure 286 shows all available options. All PDM receivers share the same PDM interface clock (clk_if). The interface clock is first derived and then divided to produce the PDM clock (pdm_clk).

Key configuration details are as follows:

- The clock source for clk_if is selected using the CLOCK_SEL bits in the CLOCK_CTL global register
- The clock divider is configured using the CLOCK_DIV bits in the CLOCK_CTL global register

The legal range for the CLOCK_DIV is 3 to 255. For optimal operation, an odd value is preferred for the CLOCK_DIV, as this ensures a 50% duty cycle for the pdm_clk.

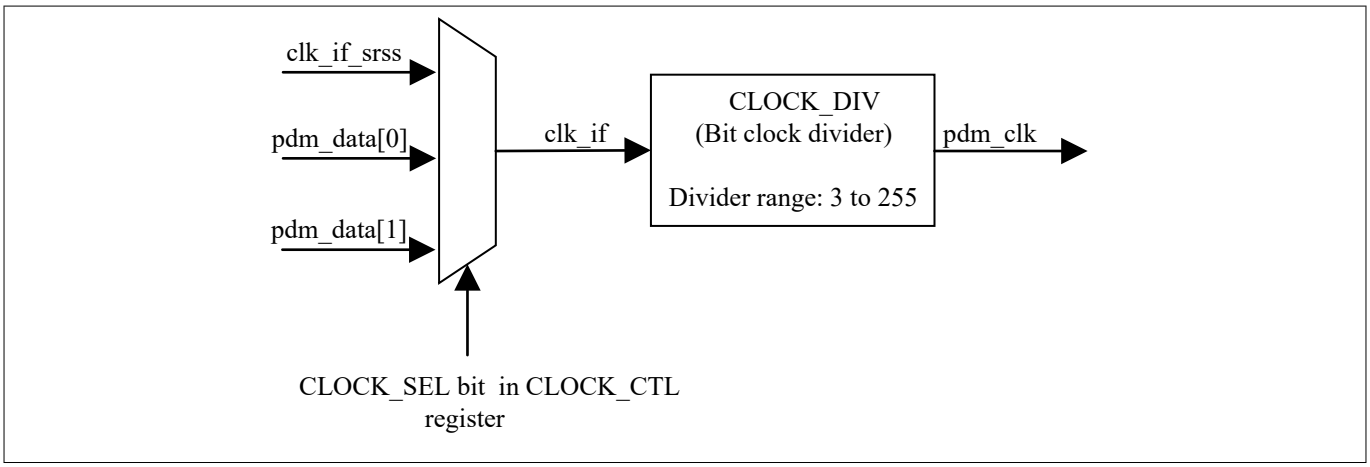


Figure 286 Clocking options

Typically, PDM data is oversampled at a 64x rate. Therefore, the PDM clock (pdm_clk) is defined as:

```
pdm_clk = PCM sample rate (Fs) * 64
```

The PDM source clock is oversampled to improve the resolution of the digital signal, enhancing the accuracy of the analog-to-digital conversion process. Oversampling is achieved by sampling the PDM bit-stream at a rate higher than the Nyquist rate (twice the highest frequency in the signal), typically at 64 times or more. The high oversampling ratio results in shorter bit durations in the PDM bit-stream, producing a signal with finer resolution.

After oversampling, the PDM bit-stream is downsampled using decimation. Decimation reduces the sampling rate by filtering out high-frequency components that are not pertinent and then discarding a portion of the remaining samples. Decimation is necessary because the high oversampling ratio used in PDM microphones generates a large amount of data, much of which may be redundant. By combining oversampling and decimation, PDM microphones achieve high accuracy and resolution in a compact form factor. The decimation factor of 64 is achieved using the CIC, FIR0, and FIR1 filters.

The pdm_clk operates the PDM interface logic, while all other logic runs on the clk_sys. The clk_if and clk_sys clocks are asynchronous. The maximum clock frequencies supported for clk_if and clk_sys (System clock) are 50 MHz and 100 MHz, respectively.

The PDM interface clock is defined as: `clk_if = pdm_clk * (CLOCK_DIV + 1)`

The relationship between the SRSS clock and the PCM sampling rate (Fs) is expressed as: `clk_if_srss = (CLOCK_DIV + 1) * Fs * 64`

Table 186 shows example values for typical PCM sampling rates and the common oversampling rate of M = 64.

Table 186 Example values for typical PCM sampling rates with an oversampling rate of M = 64

Sampling rate Fs [kHz]	Oversampling rate	pdm_clk [MHz]	Clock divider	clk_if_srss [MHz]
8	64	0.512	91	47.104
16	64	1.024	45	47.104
32	64	2.048	22	47.104
44.1	64	2.8224	16	47.9808
48	64	3.072	15	49.152

When any of the receivers is enabled, the CLOCK_SEL and CLOCK_DIV fields must not be modified.

With full-rate and half-rate sampling, users can dynamically change the digital microphone clock (pdm_clk) without affecting the PDM sampling frequency at the input to the CIC filter. Reducing the microphone clock to half its rate results in lower system power consumption but also decreases audio quality.

25.3.2.1 Full rate sampling

By default, full-rate sampling is enabled. The pdm_clk is determined by the value of CLOCK_DIV and is defined as: `pdm_clk = clk_if / (CLOCK_DIV + 1)`. In this mode, each captured PDM value is passed directly to the CIC filter for further processing.

25.3.2.2 Halve-rate sampling

The pdm_clk can be divided by two (frequency halved) by setting the HALVE bit in the CLOCK_CTL global register to 1. In halve-rate sampling mode, two PDM values are produced for each PDM period, meaning each PDM value is captured twice and sent to the CIC filter twice, as shown in Figure 287. Effectively, the same number of PDM values are generated during the PDM period as in full-rate sampling, ensuring that the sample rate (Fs) remains constant. However, the digital microphone operates at half the frequency, reducing system power consumption at the cost of lower audio quality.

In halve-rate sampling mode, the pdm_clk is defined as: `pdm_clk = 2 * clk_if / (CLOCK_DIV + 1)`. Halve-rate sampling allows the digital microphone's clock (pdm_clk) to be dynamically adjusted without affecting the PDM sample frequency sent to the CIC filter.

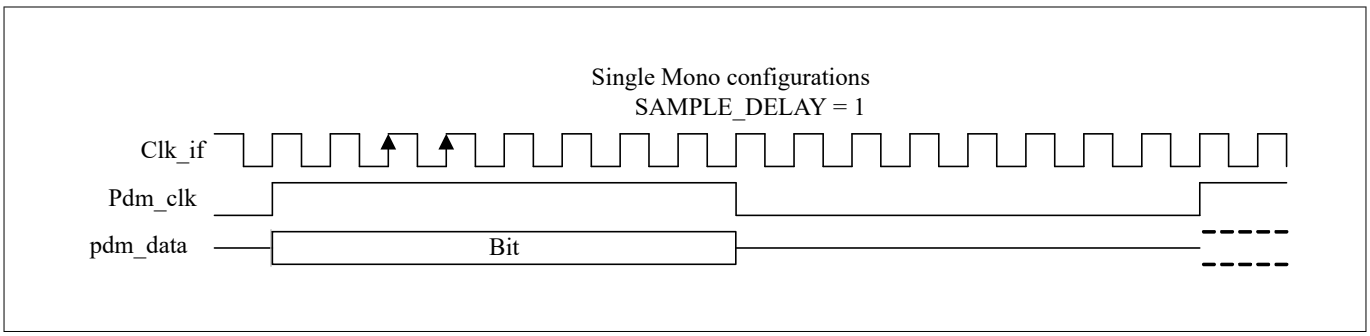


Figure 287 Halve-rate sampling

25.3.3 Sampling delay

The sampling delay allows flexibility to accommodate different master–slave–master round-trip delays. Each PDM receiver can configure the sampling delay using the SAMPLE_DELAY bit in the receiver-specific IF_CTL register. The sampling delay is measured in full cycles of clk_if, starting from the rising edge of the pdm_clk.

In stereo mode:

- The PDM bit transmission for PDM receiver N begins on the rising edge of pdm_clk
- The PDM bit transmission for PDM receiver (N+1) begins on the consecutive falling edge of pdm_clk

In mono mode:

- Only the PDM bit for PDM receiver N is transmitted
- The data line remains idle for PDM receiver (N+1)

This is shown in Figure 288, where N must be an even value.

For both PDM receivers N and (N+1), the PDM bit is sampled on the rising edge of clk_if, taking into account the sampling delay configured using the SAMPLE_DELAY bit in the IF_CTL register.

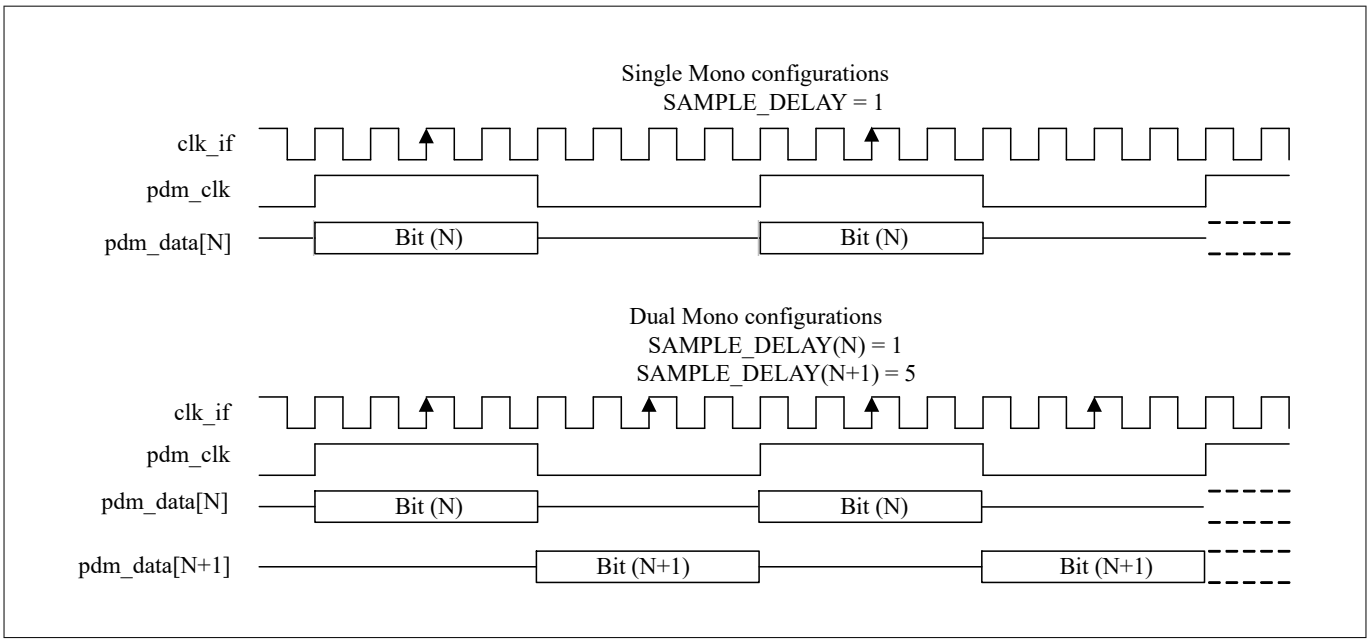


Figure 288 Mono configurations

In stereo configurations, configure the sampling delay for PDM receivers N and (N+1) so that the clk_if capture edge aligns with the midpoint between pdm_clk edges, as shown in Figure 289. Under ideal conditions:

- The midpoint for the first PDM receiver (N) in the pair is at 1/4 of the PDM interface period
- The midpoint for the second PDM receiver (N+1) in the pair is at 3/4 of the PDM interface period

The sampling delays for N and (N+1) are defined as follows:

- `SAMPLE_DELAY(N) = ((CLOCK_DIV + 1) / 4) - 1`
- `SAMPLE_DELAY(N+1) = (3 * (CLOCK_DIV + 1) / 4) - 1`

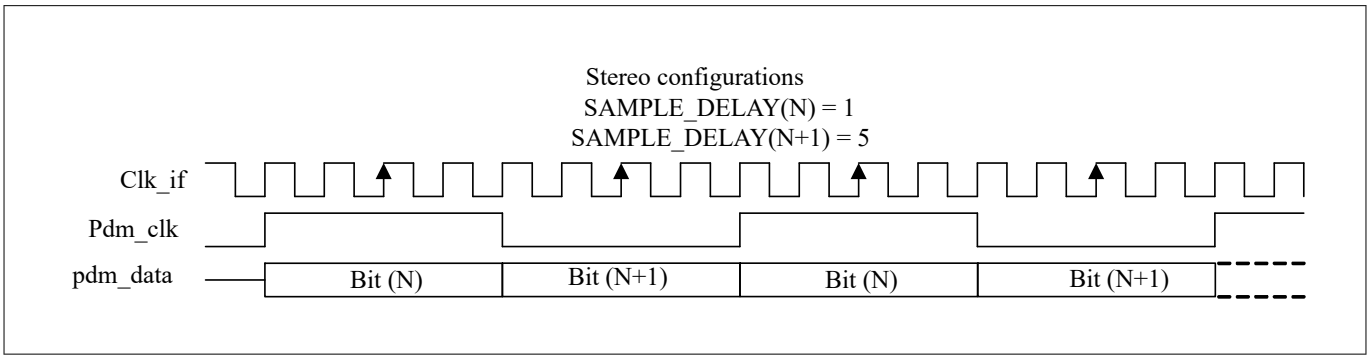


Figure 289 Stereo configurations

In practice, due to round-trip delays, the SAMPLE_DELAY value may need to be set to a later point relative to the ideal midpoint, as shown in Figure 290. For both mono and stereo configurations, avoid setting a SAMPLE_DELAY value that positions the capture edge too close to the pdm_clk edges. Doing so may result in inaccurate data capture.

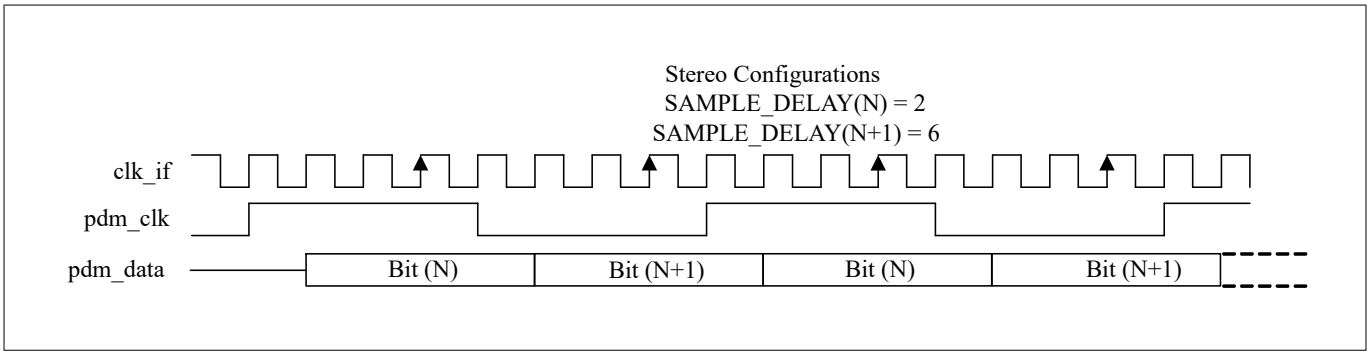


Figure 290 Practical use case

25.3.4 Filter options

The PDM-PCM converter block provides multiple filtering options for processing PDM data from the PDM source. The available filters include one cascaded integrator comb (CIC) filter, two finite impulse response (FIR) filters, and one DC-blocking filter.

- Dedicated CIC decimation filters are available for each PDM receiver. The CIC filter's integrators and differentiators reuse a single adder for efficient operation
- The FIR and DC-blocking filters are implemented using a common filtering path shared by all PDM receivers

This common filter path enables the following memory-sharing features:

- **Common SRAM memory for FIR 0 filter data**:
  - Each PDM receiver uses 64 entries of 26 bits
  - Two PDM receivers require 128 x 26-bit SRAM
- **Common SRAM memory for FIR 1 filter data**:
  - Each PDM receiver uses 64 entries of 24 bits
  - Two PDM receivers require 128 x 24-bit SRAM
- **Common SRAM memory for Rx FIFOs**:
  - Each Rx FIFO uses 64 entries of 24 bits
  - Two PDM receivers require 128 x 24-bit SRAM

The common filter path is shown in Figure 291.

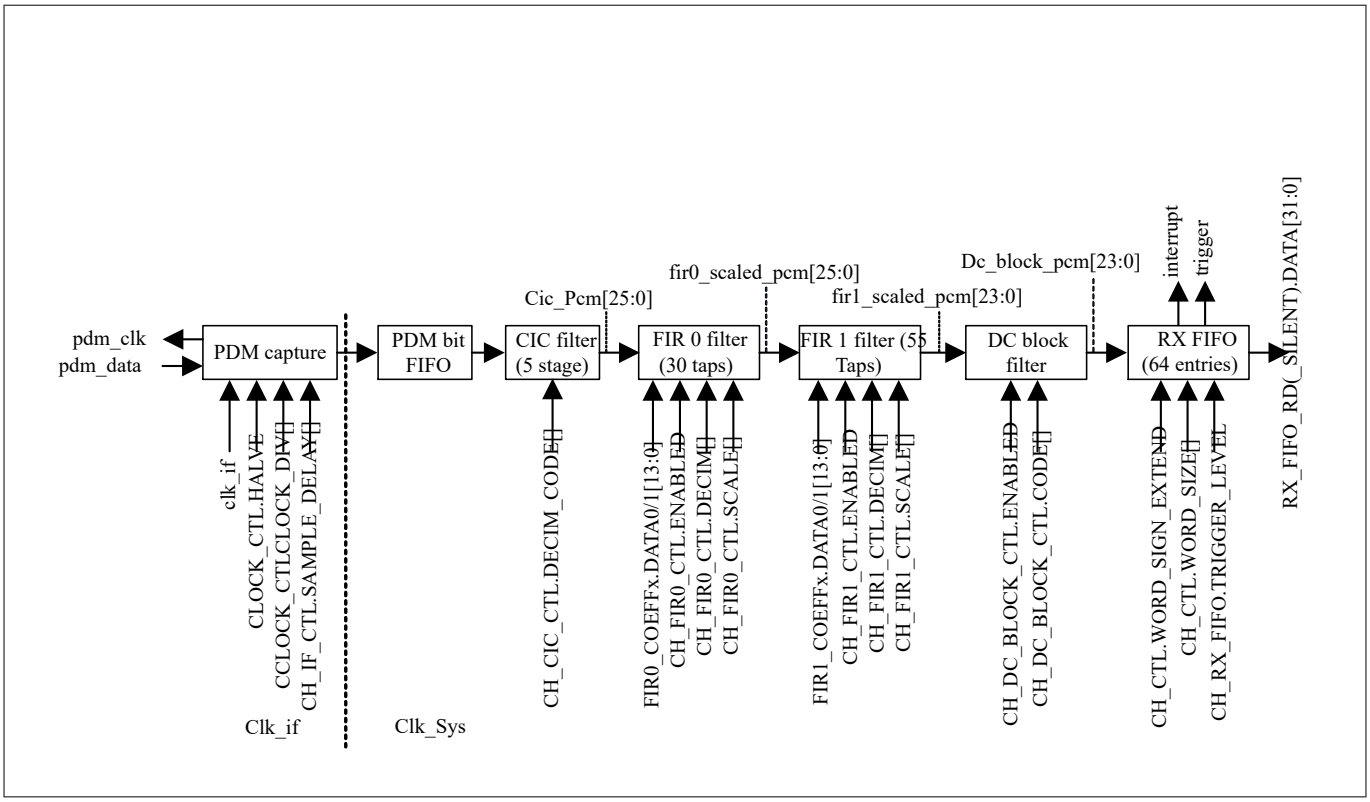


Figure 291 PDM receivers common path

25.3.4.1 Cascaded integrator comb (CIC) filter

CIC filters are used to reduce the sampling rate of a signal. These filters combine integrator and comb filter stages in a cascaded structure to efficiently remove high-frequency components from the input signal.

- The integrator stage accumulates the input signal over time, acting as a low-pass filter
- The comb stage decimates the signal by selecting every N^{th} sample

The cascaded structure of the filter enables a high degree of downsampling while maintaining low computational complexity. The PDM-PCM converter uses a 5-stage CIC filter. The input to the CIC filter is oversampled PDM bits, and the output is a two's-complement 26-bit integer, `cic_pcm[25:0]`. The frequency of the output PCM samples is decimated by a factor specified by the `DECIM_CODE` bit field in the `CIC_CTL` register.

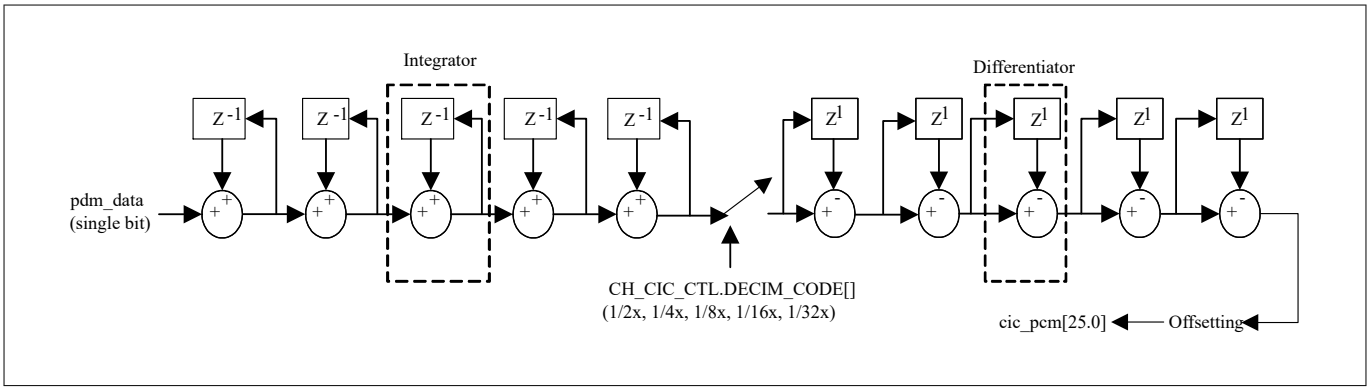


Figure 292 CIC filter

The offsetting functionality in the CIC filter converts unsigned data into signed data within a predefined range, determined by the `DECIM_CODE` value in the `CIC_CTL` register. This offsetting logic ensures that specific input patterns on the PDM interface correspond to predictable CIC filter PCM values:

- A digital silence pattern on the PDM interface (alternating '0', '1', '0', '1' values) results in a CIC PCM value of 0
- A continuous pattern of '0', '0', '0' PDM values produces the minimum CIC PCM value of `-0x1000000` (when `DECIM_CODE` is set to 4)
- A continuous pattern of '1', '1', '1' PDM values produces the maximum CIC PCM value of `0x1000000` (when `DECIM_CODE` is set to 4)

25.3.4.2 Finite impulse response (FIR) filters

25.3.4.2.1 FIR0

This is the first FIR filter in the common filter path. The FIR 0 filter is a 30-tap symmetric filter with 14-bit signed coefficients in the range [-8192, 8191]. By default, the FIR 0 filter is disabled and can be enabled by setting the `ENABLED` bit in the `FIR0_CTL` register for each PDM receiver. The FIR filter coefficients are programmed through the `FIR0_COEFFx.DATA0/1` registers, where x ranges from 0 to 7. The coefficients do not have any default values and must be explicitly programmed by the user before enabling the filter.

The FIR filter processes 30 successive 26-bit signed CIC filter PCM values, `cic_pcm[25:0]`, to produce intermediate 45-bit signed values, `fir0_pcm[44:0]`. The extra 19 bits accommodate the 30 14-bit coefficients. The resulting 45-bit PCM values are then scaled to 26-bit signed PCM values, `fir0_scaled_pcm[25:0]`. The scaling operation is defined by the `SCALE` bit field in the `FIR0_CTL` register, which specifies the right-shift amount and performs rounding on the PCM values. Finally, the PCM values are clipped to 26 bits.

The decimation code determines the output frequency of the FIR 0 filter. This code is configured using the `DECIM3` bit field in the `FIR0_CTL` register for each PDM receiver. The frequency of the output PCM samples is reduced by the factor specified in `DECIM3`. Typically, this filter is used for sample frequencies (`Fs`) of 8 kHz and 16 kHz.

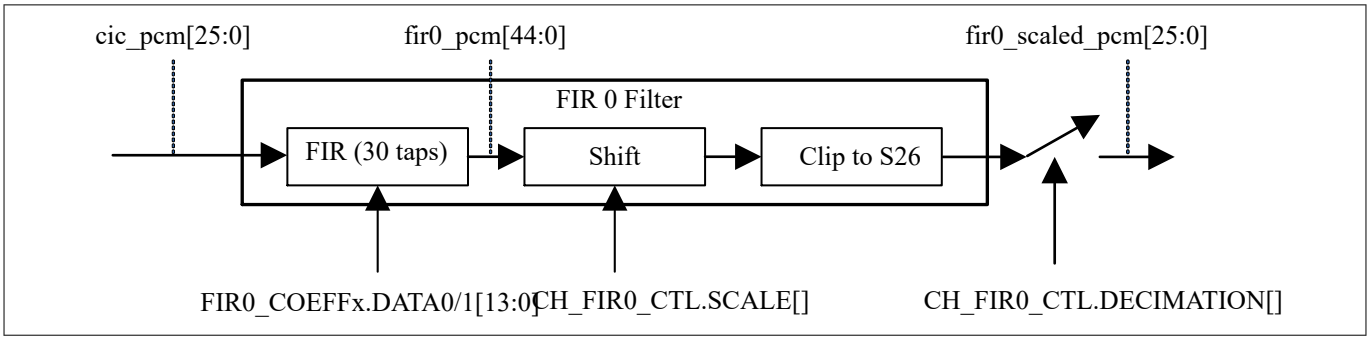


Figure 293 FIR0

25.3.4.2.2 FIR1

This is the second FIR filter in the common filter path. The FIR 1 filter is a 55-tap symmetric filter with 14-bit signed coefficients in the range [-8192, 8191]. By default, the FIR 1 filter is disabled and can be enabled by setting the `ENABLED` bit in the `FIR1_CTL` register for each PDM receiver. The FIR filter coefficients are programmed through the `FIR1_COEFFx.DATA0/1` registers, where x ranges from 0 to 13. These coefficients have default values

and include built-in droop correction. The total filter gain (sum of the coefficients) is 13,921. Default coefficient values are as follows:

```
-2, 21, 26, -17, -41, 25, 68, -33,
-107, 41, 160, -48, -230, 54, 325, -56,
-453, 51, 631, -31, -894, -21, 1326, 172,
-2191, -770, 4859, 8191, 4859, -770, -2191, 172,
1326, -21, -894, -31, 631, 51, -453, -56,
325, 54, -230, -48, 160, 41, -107, -33,
68, 25, -41, -17, 26, 21, -2
```

These default values can be reprogrammed based on user requirements.

The FIR 1 filter processes 55 successive 26-bit signed PCM values from the FIR 0 filter (`fir0_scaled_pcm[25:0]`). The intermediate output is a 46-bit signed value, `fir1_pcm[45:0]`. The additional 20 bits account for the fifty-five 14-bit coefficients. These 46-bit PCM values are scaled to 24-bit signed PCM values, `fir1_scaled_pcm[23:0]`, as specified by the SCALE bit field in the FIR1_CTL register for each PDM receiver. The SCALE value determines the right-shift amount and performs rounding of the PCM values. Finally, the PCM values are clipped to 24 bits.

The decimation code determines the output frequency of the FIR 1 filter. This code is configured using the DECIM2 bit field in the FIR1_CTL register for each PDM receiver. The output PCM samples are decimated by the factor indicated in DECIM2. The resulting `fir1_scaled_pcm[23:0]` values are then input to the DC-blocking filter.

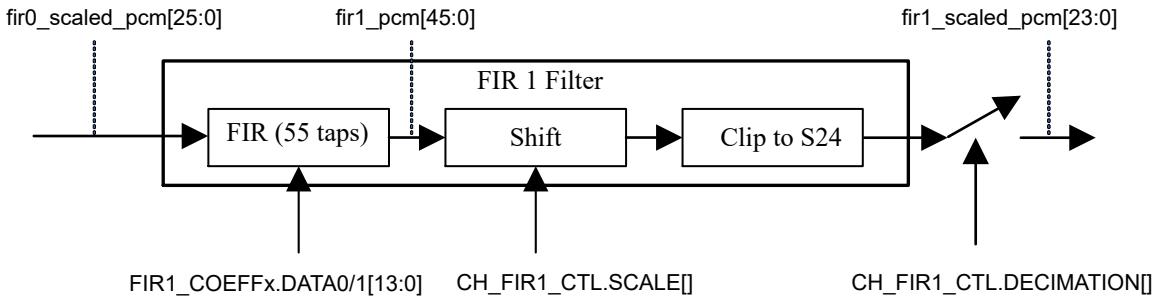


Figure 294 FIR1

25.3.4.3 DC-blocking filter

The DC-blocking filter removes the DC component from the FIR filter output, `fir1_scaled_pcm[23:0]`. By default, the DC-blocking filter is enabled. It has a programmable gain coefficient, `dc_block_coeff[12:0]`, which is indirectly defined by the CODE bit field in the DC_BLOCK_CTL register. The output of the DC-blocking filter is scaled to 24-bit values, `dc_block_pcm[23:0]`, and these outputs are written to the Rx FIFO.

For debugging purposes, the DC-blocking filter can be disabled by clearing the ENABLED bit in the DC_BLOCK_CTL register. The definition of `dc_block_pcm[23:0]` varies based on whether the filter is enabled or disabled:

- **Disabled case**: `dc_block_pcm(n) = fir1_scaled_pcm(n)`
- **Enabled case**:

```
dc_block_state_scaled(n-1) = dc_block_state(n-1) - (dc_block_state(n-1) >> (12 - CODE))
dc_block_state(n) = CLIP37 (2^13 * (fir1_scaled_pcm(n) - fir1_scaled_pcm(n-1)) +
dc_block_state_scaled(n-1))
dc_block_pcm(n) = dc_block_state(n) >> 13
```

The first step (scaling) adjusts the DC-block state, effectively multiplying it by a variable `alpha`, which is close to 1. The value of `alpha` is calculated as: `alpha = 1 - (1/2^(12 - CODE))`.

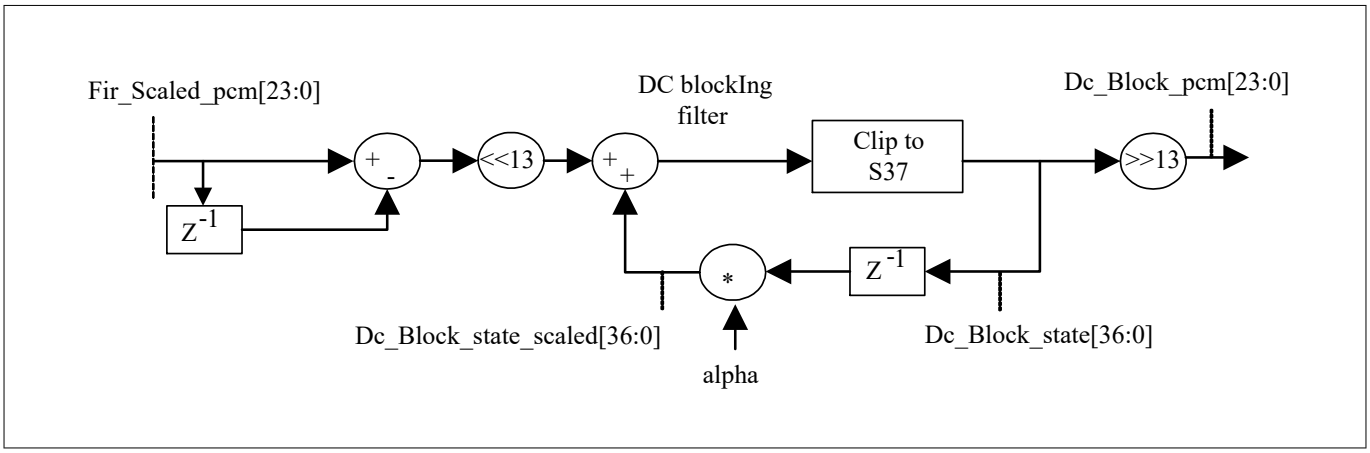


Figure 295 DC blocking filter

25.3.5 Rx FIFO

The PDM-PCM converter hardware writes data into the Rx FIFO, while the software reads it. When the number of FIFO entries exceeds a programmable trigger level, a trigger interrupt is generated. The trigger level is configured using the `TRIGGER_LEVEL` bits in the `Rx_FIFO_CTL` register. Upon a trigger interrupt, the PCM data must be transferred from the Rx FIFO buffer to SRAM, either by the CPU or the DMA.

In addition to the trigger interrupt, the Rx FIFO supports both underflow and overflow interrupts:

- An overflow interrupt is activated when the hardware writes data to an almost full Rx FIFO
- An underflow interrupt is activated when the software reads data from an empty Rx FIFO

A common SRAM is shared among all Rx FIFOs. Since each Rx FIFO contains 64 entries of 24 bits, the required memory for `N` PDM receivers is: `N * 64 * 24 bits`.

Each PDM receiver has a dedicated FIFO trigger, which is typically connected to a DMA channel. In addition, a 'unified' trigger is provided. The unified trigger activates when all enabled PDM receivers' triggers are active. This allows a single DMA channel to service all enabled PDM receivers, instead of requiring separate DMA channels for each receiver. System access to the Rx FIFOs is achieved through an AHB-Lite interface. When a PDM receiver is disabled (`CTL.ENABLED = 0`) or when the Rx FIFO is empty, reading the Rx FIFO register returns '0' and does not generate a bus error. The Rx FIFO register is software read-only; any software write to the register is ignored and does not generate a bus error.

Reading PCM samples behaves like a POP operation, removing the data from the Rx FIFO. Specifically:

- `Rx_FIFO_STATUS.RD_PTR` is incremented
- `Rx_FIFO_STATUS.USED` is decremented

The read data (`DATA`) is right-aligned, with unused bit positions extended as specified by `CTL.WORD_SIGN_EXTEND`. Reading from an empty Rx FIFO triggers an `INTR_Rx.FIFO_UNDERFLOW` event.

Software can handle Rx FIFO data in two ways:

- **Read (pop)**: PCM data from the top of the FIFO can be read using the Rx_FIFO_RD register
- **Peek**: PCM data at the top of the FIFO can be observed (without removing it) using the Rx_FIFO_RD_SILENT register

The FREEZE bit in the Rx_FIFO_CTL register offers debugging functionality:

- When the FREEZE bit is set, hardware writes have no effect on the Rx FIFO, and the FIFO write pointer does not advance
- When FREEZE is cleared (0), hardware writes resume, and the FIFO write pointer advances as expected

The Rx_FIFO_STATUS register provides information about the Rx FIFO status, which includes:

- **Number of entries in the FIFO**: Read using the USED bits
- **FIFO read location**: The location from which data is read, using the RD_PTR bits
- **FIFO write location**: The location where new data is written by the hardware, using the WR_PTR bits

***Note**: Reading the FIFO's read and write pointers provides functionality for debugging purposes only.*

25.3.6 Interrupts

The PDM-PCM converter block provides one interrupt for each PDM receiver, which is routed to the interrupt controller in the CPU. For details on the interrupt vector number, as well as the procedure to configure interrupt priority, vector address, and enabling or disabling the interrupt, refer to the Interrupts chapter of this manual.

The PDM-PCM converter interrupts can be triggered by any of the following events:

- FIFO_TRIGGER
- FIFO_OVERFLOW
- FIFO_UNDERFLOW
- FIR_OVERFLOW
- IF_OVERFLOW

For FIFO-related events such as FIFO_TRIGGER, FIFO_OVERFLOW, and FIFO_UNDERFLOW, see Rx FIFO.

- The hardware sets the FIR_OVERFLOW field to 1 when the CIC filter produces PCM samples faster than the FIR filter can process them. This indicates that the PDM-PCM converter's system frequency is too low
- The hardware sets the IF_OVERFLOW field to 1 when the PDM samples are generated too quickly by the interface logic, leading to an interface overflow. This may indicate that the PDM-PCM converter's system frequency is too low relative to the interface frequency

The interface overflow is a non-recoverable error and requires the following corrective actions:

1. The software must disable the PDM receiver by clearing the CTL.ENABLED field
2. The INTR_RX.IF_OVERFLOW field must be cleared to 0

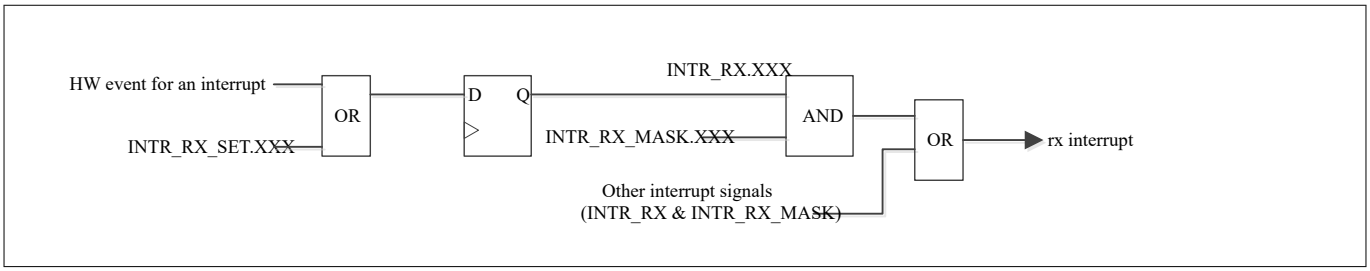


Figure 296 Interrupts

Each interrupt event can be individually enabled or disabled to generate an interrupt condition. The INTR_RX_MASK register is used to enable specific events by writing 1 to the corresponding bit for the desired interrupt event.

Regardless of the INTR_RX_MASK settings, if any event occurs, the hardware sets the corresponding event status bit in the INTR_RX register. The INTR_RX_MASKED register is the result of a bitwise AND operation between the INTR_RX_MASK and INTR_RX registers: INTR_RX_MASKED = INTR_RX_MASK AND INTR_RX

The final interrupt signal is the logical OR of all the bits in the INTR_RX_MASKED register. Only the events enabled in the INTR_RX_MASK register are propagated as interrupt events to the interrupt controller. Interrupts can also be manually triggered in software by writing to the corresponding bits in the INTR_TX_SET register. Figure 296 shows the interrupt signal generation for a single PDM receiver, and all receivers follow this same architecture.

In the interrupt service routine (ISR):

- The INTR_RX_MASKED register should be read to identify the events that triggered the interrupt
- Multiple events can trigger the interrupt, as the final interrupt signal is the logical OR output of all enabled events
- The ISR should perform tasks corresponding to each triggered interrupt event

At the end of the ISR:

1. The value previously read from the INTR_RX_MASKED register should be written to the INTR_TX or INTR_RX register to clear the bits corresponding to the processed interrupt events
2. If the bits are not cleared by writing 1 to the INTR_RX register, the interrupt signal will remain high
3. A dummy read of the INTR_TX or INTR_RX register is required to ensure the earlier write operation takes effect

26 Serial memory interface (SMIF)

The serial memory interface (SMIF) is designed to facilitate convenient and efficient communication with serial flash memories, enabling the CYW20829 SoC to perform read, write, and erase operations. By utilizing the SMIF interface, developers can seamlessly integrate serial flash memories into their applications, offering more versatile and scalable memory solutions.

The SMIF in CYW20829 is an IP block for external serial connectivity, providing high-performance communication with external memory devices. It uses an SPI-based communication interface to connect with external quad SPI (QSPI) high-speed memory devices, ensuring efficient and reliable data transfer for a wide range of applications.

26.1 Features

The SMIF in the CYW20829 platform includes the following features:

- Supports interfacing with up to four memory devices (slaves) simultaneously
- Follows the SPI protocol
- Operates in SPI mode 0, where clock polarity (CPOL) and clock phase (CPHA) are both set to ‘0’
- Supports single, dual, quad, and octal SPI protocols
- Provides dual-quad SPI mode, allowing the use of two quad SPI memory devices to increase data bandwidth for SPI read and write transfers
- Includes configurable MISO sampling time and a programmable receiver clock
- Supports device capacities ranging from 64 KB to 128 MB
- Offers a memory-mapped mode of operation, Execute in Place, which maps external memory into an internal memory address
- Provides a command mode (MMIO mode) that enables using the SMIF block as simple communication hardware
- Includes a 4-KB read cache for memory-mapped (XIP) mode operations
- Supports on-the-fly 128-bit encryption and decryption

26.2 Description

The SMIF hardware block diagram provides a high-level view of the SMIF hardware in CYW20829. The block is divided into multiple clock domains, enabling various domains to access the SMIF while maintaining an asynchronous clock for the communication interface.

The SMIF block can generate DMA triggers and interrupt signals, allowing events within the SMIF block to initiate actions in other parts of the system.

The SMIF interface is implemented with the following components:

- Eight data lines
- Four slave select lines
- A clock, which can be differential or single-ended
- A read/write data strobe (RWDS) signal line

Access to the SMIF block is available in two modes: MMIO mode and XIP mode.

MMIO mode:

- The MMIO mode provides access to the SMIF’s peripheral registers and internal FIFOs
- In this mode, the user code is responsible for constructing the command structure for the external memory

- Typically, this mode is used for writing data to external flash memory
- The MMIO interface is also used to configure the SMIF hardware block, including setting up the device registers required for XIP operation

Execute in Place mode:

- The Execute in Place mode maps the external memory space to a range of addresses in the CYW20829 address space
- When this address range is accessed, the hardware automatically generates the necessary commands to initiate transfers from the external memory
- The Execute in Place mode is typically used for executing code stored in external memory. Through this mode, the external memory is mapped into the internal address space of the CYW20829 MCU, ensuring seamless code execution directly from external memory

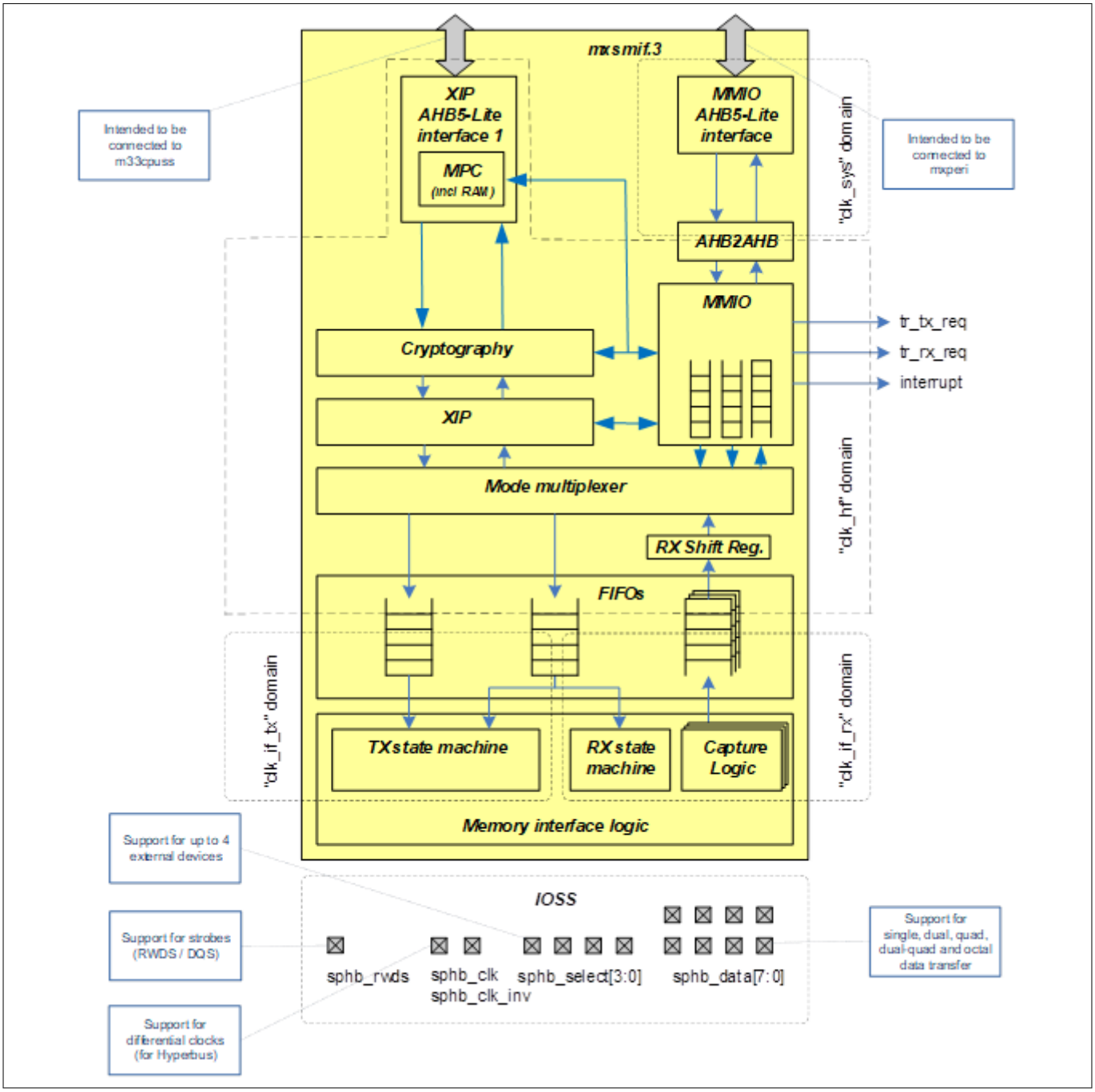


Figure 297 SMIF hardware block diagram

CYW20829 includes two AHB-Lite interfaces for accessing the SMIF: one for accessing the MMIO registers and another for supporting Execute in Place operations. Both interfaces provide access to external memory devices. At any given time, either the MMIO AHB-Lite interface or the XIP AHB-Lite interface has access to the memory interface logic and the external memory devices. The operation mode is determined by the `SMIFn_CTL.XIP_MODE` register and must not be modified when the SMIF is busy (`STATUS.BUSY = 1`).

MMIO AHB-Lite interface:

- In MMIO mode, access is facilitated through software writes to the transmit (Tx) FIFOs and software reads from the receive (Rx) FIFOs
- The Tx and Rx FIFOs are mapped to the SMIF registers
- This interface offers flexibility to execute various types of external memory device transfers, such as setup, programming, or erasing operations

XIP AHB-Lite interface:

- In XIP mode, access is supported through AHB-Lite read and write transfers, which are automatically translated by the hardware into memory device read and write transfers
- This interface is optimized for efficient memory device read and write transfers but does not support other types of memory operations
- To enhance performance, the XIP AHB-Lite interface includes a 4-KB read cache

Mutually exclusive modes:

- MMIO mode and XIP mode are mutually exclusive
- Both modes share the Tx and Rx FIFOs, as well as the memory interface logic
- In MMIO mode, the Tx and Rx FIFOs are accessed through SMIF registers and are under software control
- In XIP mode, the Tx and Rx FIFOs are controlled by hardware

The memory interface logic is controlled through the Tx and Rx FIFOs and operates independently of the mode of operation, making it agnostic to whether MMIO or XIP mode is being used. This architecture ensures efficient and flexible access to external memory devices for different application requirements.

26.2.1 Tx and Rx FIFOs

The SMIF block contains two Tx FIFOs and one Rx FIFO. These FIFOs provide asynchronous clock-domain transfer between clk_hf logic and clk_if_tx/clk_if_rx memory-interface logic. The memory-interface logic is entirely controlled through the Tx and Rx FIFOs.

- The Tx command FIFO transmits memory commands to the memory-interface logic
- The Tx data FIFO transmits write data to the memory-interface transmit logic
- The Rx data FIFO receives read data from the memory-interface receive logic

26.2.1.1 Tx command FIFO

The Tx command FIFO consists of four 20-bit entries. Each entry holds a command. A memory transfer is composed of a series of commands. In other words, a command specifies a phase of a memory transfer. Four types of commands are supported:

- **Tx command**

  A memory transfer must start with a Tx command. The Tx command includes a byte to be transmitted over the memory interface. It specifies the width of the data transfer (single, dual, quad, or octal data transfer). It also indicates whether the command is for the last phase of the memory transfer (explicit “last command” indication). Additionally, the Tx command determines which of the four external devices are selected (multiple devices can be selected simultaneously). The device selection encoded by the Tx command is used for the entire memory transfer. The Tx command asserts the corresponding slave-select lines, which is why every memory transfer must start with this command

- **Tx_count command**

  The Tx_count command specifies the number of bytes to be transmitted from the Tx data FIFO. This command uses the Tx data FIFO to provide the bytes to be transmitted over the memory interface. The Tx_count command specifies the width of the data transfer and always serves as the last phase of the memory transfer (implicit “last command” indication - de-asserts the slave select). The Tx_count command does not assert the slave-select lines; this must be done by a preceding Tx command
- **Rx_count command**

  The Rx_count command specifies the number of bytes to be received into the rx data FIFO. This command uses the rx data FIFO to accept the bytes received over the memory interface. The Rx_count command specifies the width of the data transfer and always serves as the last phase of the memory transfer (implicit “last command” indication - de-asserts the slave select). The Rx_count command does not assert the slave-select lines; this must be done by a preceding Tx command
- **Dummy_count command**

  The Dummy_count command specifies a number of dummy cycles. Dummy cycles implement a turn-around (TAR) time, during which the memory master switches from transmitting (driving the data lines) to receiving (listening on the same data lines). The Dummy_count command never serves as the last phase of the memory transfer (implicit NOT “last command” indication - de-asserts the slave select). It must always be followed by another command. The Dummy_count command does not assert the slave-select lines; this must be done by a preceding Tx command

Together, these four command types can construct any SPI transfer. The Tx command FIFO is used by both the memory-interface transmit and receive logic to ensure lockstep operation. The Tx command FIFO represents a queue of commands to be processed.

Software writes the sequence of commands into the Tx command FIFO to initiate communication with the slave device. The software can read the number of used Tx command FIFO entries through the `TX_CMD_FIFO_STATUS.USED[2:0]` register field.

Software writes to the Tx command FIFO using the MMIO `TX_CMD_FIFO_WR` register. If software attempts to write to a full Tx command FIFO, the MMIO `CTL.BLOCK` field specifies the behavior:

- If `CTL.BLOCK` is set to ‘0’, an AHB-Lite bus error is generated
- If `CTL.BLOCK` is set to ‘1’, the AHB-Lite write transfer is extended until an entry becomes available, increasing latency

26.2.1.2 Tx data FIFO

The Tx data FIFO consists of eight 8-bit entries. A Tx command FIFO `TX_COUNT` command specifies the number of bytes to be transmitted, which corresponds to the number of Tx data FIFO entries used. The Tx data FIFO is utilized by the memory-interface transmit logic.

Software can read the number of used Tx data FIFO entries through the `TX_DATA_FIFO_STATUS.USED[3:0]` register field.

Software can write to the Tx data FIFO using the `TX_DATA_FIFO_WR1`, `TX_DATA_FIFO_WR2`, and `TX_DATA_FIFO_WR4` registers:

- The `TX_DATA_FIFO_WR1` register supports writing a single byte to the FIFO
- The `TX_DATA_FIFO_WR2` register supports writing two bytes to the FIFO
- The `TX_DATA_FIFO_WR4` register supports writing four bytes to the FIFO

If software attempts to write more bytes than the available entries in the Tx data FIFO, the MMIO `CTL.BLOCK` field specifies the behavior:

- If `CTL.BLOCK` is set to ‘0’, an AHB-Lite bus error is generated
- If `CTL.BLOCK` is set to ‘1’, the AHB-Lite write transfer is extended until the required entries become available

26.2.1.3 Rx data FIFO

The Rx data FIFO consists of eight 8-bit entries. A Tx command FIFO RX_COUNT command specifies the number of bytes to be received, which corresponds to the number of Rx data FIFO entries used. The memory-interface transmit logic stops generating the SPI clock when the Rx data FIFO is full. This mechanism ensures flow control.

Software can read the number of used Rx data FIFO entries through the RX_DATA_FIFO_STATUS.USED[3:0] register field.

Software can read from the Rx data FIFO using the RX_DATA_FIFO_RD1, RX_DATA_FIFO_RD2, and RX_DATA_FIFO_RD4 registers:

- The RX_DATA_FIFO_RD1 register supports reading a single byte from the FIFO
- The RX_DATA_FIFO_RD2 register supports reading two bytes from the FIFO
- The RX_DATA_FIFO_RD4 register supports reading four bytes from the FIFO

If software attempts to read more bytes than are available in the Rx data FIFO, the MMIO CTL.BLOCK field specifies the behavior:

- If BLOCK is set to '0', an AHB-Lite bus error is generated, and a hard fault occurs
- If BLOCK is set to '1', the AHB-Lite read transfer is extended until the required bytes become available

Software can also read the first byte of the Rx data FIFO without changing the status of the FIFO using the RX_DATA_FIFO_RD1_SILENT register.

26.2.2 MIMO mode

If CTL.XIP_MODE is set to '0', the SMIF operates in MMIO mode. In this mode, software generates SPI transfers by accessing the Tx FIFOs and Rx FIFO. Software writes data to the Tx FIFOs and reads data from the Rx FIFO. The Tx command FIFO contains formatted commands (Tx, TX_COUNT, RX_COUNT, and DUMMY_COUNT). Software must ensure that it generates valid memory transfers and accesses the FIFOs correctly.

For example:

- If a memory transfer is generated to read four bytes from a memory device, software must read those four bytes from the Rx data FIFO
- Similarly, if a memory transfer is generated to write four bytes to a memory device, software must write those four bytes to the Tx command FIFO or Tx data FIFO

Incorrect software behavior can cause the memory interface to lock up. For instance, if a memory transfer is initiated to read 32 bytes from a memory device but software does not read from the Rx data FIFO, the memory transfer will lock up. This happens because the Rx data FIFO can only hold eight bytes (as it has eight entries), and the memory interface cannot provide additional bytes while the FIFO is full. This condition prevents any subsequent memory transfers from occurring.

To avoid such issues, software must read from the FIFOs to prevent congestion. Note that a memory transfer lock-up due to the state of the Tx or Rx FIFOs is still compliant with the memory bus protocol, although it is undesirable. The SPI protocol allows the interface clock to shut down in the middle of a memory transfer, which effectively pauses the operation until the issue is resolved.

26.2.3 XIP mode

If CTL.XIP_MODE is set to '1', the SMIF operates in XIP mode. In this mode, hardware automatically generates memory transfers by accessing the Tx FIFOs and Rx FIFO without software intervention. Hardware supports only memory read and write transfers, while other functionality, such as status reads, is not supported. For example, writing into a flash memory device may not be supported in XIP mode because the operation involves not only a write command but also a status check to verify the operation's completion.

- Hardware generates a memory read transfer for an AHB-Lite read transfer (specifically for AHB-Lite read transfers that miss in the cache)
- Hardware generates a memory write transfer for an AHB-Lite write transfer

Each slave device slot has a set of associated device configuration registers. To access a memory device in XIP mode, the corresponding device configuration registers (`SMIFn_DEVICEn`) must be initialized. These device configuration registers set up the following parameters for the memory:

- **Write enable (WR_EN)**: Used to disable writes in XIP mode
- **Crypto enable (CRYPTO_EN)**: When enabled, all read accesses to the memory are decrypted and all write accesses are encrypted automatically
- **Data select (DATA_SEL)**: Selects the data lines to be used
- **Base address and size**: Configures the mapped memory space, where any access in this space is automatically converted into an external memory access
- **Read and write commands**: Used to communicate with the external memory device. These commands are defined through multiple settings

Since different memory devices support different types of read and write commands, hardware must be provided with device-specific details to perform automatic translations. For this purpose, each memory device has a set of MMIO registers that define its memory read and write transfers. These specifications include:

- Presence and value of the SPI command byte
- Number of address bytes
- Presence and value of the mode byte
- Number of dummy cycles
- Specified data transfer widths

The XIP interface logic generates an AHB-Lite bus error under the following conditions:

- The SMIF is disabled (`SMIFn_CTL.ENABLED` is '0')
- The SMIF is not in XIP mode (`SMIFn_CTL.XIP_MODE` is '0')
- The transfer request falls outside a valid memory region
- The transfer is a write, and the identified memory region does not support writes (`SMIFn_DEVICEn_CTL.WR_EN` is '0')
- In XIP mode (`CTL.XIP_MODE` is '1') and dual-quad SPI mode (`ADDR_CTL.DIV2` is '1'), the transfer address is not a multiple of 2
- In XIP mode (`CTL.XIP_MODE` is '1') and dual-quad SPI mode (`ADDR_CTL.DIV2` is '1'), the transfer size is not a multiple of 2

26.2.4 Cache

To improve XIP performance, the XIP AHB-Lite interface includes a cache. The cache is defined as follows:

- 4 KB capacity
- Read-only cache: write transfers bypass the cache. A write to an address already prefetched in the cache invalidates the associated cache subsector. If a write to memory occurs in MMIO mode, you must invalidate the cache before switching back to XIP mode
- Four-way set associative, with a least recently used (LRU) replacement scheme

Each XIP interface implements a 4 KB cache memory, which is enabled by default. Any XIP access can be cached if the cache is enabled. There are separate cache registers for the slow cache (in the `clk_slow` domain) and the fast cache (in the `clk_fast` domain). The cache can be enabled using the `SLOW_CA_CTL[ENABLED]` or `FAST_CA_CTL[ENABLED]` registers.

- Read transfers that result in a cache "hit" are processed directly by the cache
- Read transfers that result in a cache "miss" trigger an XIP memory read transfer

If CA_CTL.PREF_EN is set to '1', prefetching is enabled. If CA_CTL.PREF_EN is set to '0', prefetching is disabled. When prefetching is enabled, a cache miss initiates a 16 B (subsector) refill for the missing data and a 16 B prefetch for the next sequential data, regardless of whether this data is already in the cache.

Cache coherency is not supported by the hardware. For example:

- A write to an address in the XIP interface 0 cache invalidates (clears) the associated cache subsector in the XIP interface 0 cache, but the corresponding data in the XIP interface 1 cache remains outdated
- Users must manually invalidate the cache in such cases by using the SLOW_CA_CMD[INV] or FAST_CA_CMD[INV] register

Caches should also be invalidated during mode transitions. For example, in MMIO mode, a write to an address in the cache interface causes the data in the cache interface to become outdated. To ensure valid data, the cache should be invalidated before transitioning back to XIP mode.

26.2.5 Arbitration

The SMIF provides two AHB-Lite slave interfaces to the CPUSS: one fast interface and one slow interface. Both interfaces include a cache (as described in section 30.2.4, Cache) and can generate XIP requests to external memory devices. An arbitration component, as shown in Figure 30-1, manages arbitration between the two interfaces. Arbitration is based on the master identifier of the AHB-Lite transfer.

The arbitration priority is determined by a system-wide priority scheme. Each master identifier is assigned a 2-bit priority level, where "0" represents the highest priority level, and "3" represents the lowest priority level. Master identifiers with the same priority level are grouped into the same priority group. Within a priority group, arbitration is performed using a round-robin scheme.

26.2.6 Deselect delay

The SMIF allows configuration of the deselect delay between transfers. The SMIFn_CTL.DESELECT_DELAY field specifies the minimum number of interface cycles during which the chip-select line remains inactive.

26.2.7 Cryptography

In XIP mode, a cryptography component provides on-the-fly encryption for write data and on-the-fly decryption for read data. The use of on-the-fly cryptography is controlled by a device's MMIO CTL.CRYPTO_EN field. In MMIO mode, the cryptography component is accessible through a register interface to support offline encryption and decryption.

The cryptography functionality is designed for scenarios where data is stored in external memory devices in an encrypted format. In such cases:

- Memory read transfers require decryption functionality
- Memory write transfers require encryption functionality

By encrypting data in external memory devices (nonvolatile devices), the risk of sensitive data leakage is mitigated.

Encryption and decryption are based on the AES-128 forward block cipher, which uses the advanced encryption standard with a 128-bit key. The KEY[127:0] is a secret (private) key programmed into the CRYPTO_KEY3 to CRYPTO_KEY0 registers. These registers are write-only, and any software read operation returns "0." In SMIF hardware, applying AES-128 with KEY[127:0] to a plaintext PT[127:0] produces a ciphertext CT[127:0].

- In XIP mode, the XIP address is used as the plaintext PT[]. The resulting ciphertext CT[] is used on-the-fly and is not accessible to software. The XIP address is extended using the CRYPTO_INPUT3 to CRYPTO_INPUT0 registers
- In MMIO mode, the MMIO CRYPTO_INPUT3 to CRYPTO_INPUT0 registers provide the plaintext PT[]. The resulting ciphertext CT[] is accessible through the MMIO CRYPTO_OUTPUT3 to CRYPTO_OUTPUT0 registers

Figure 298 shows the cryptographic functionality in XIP and MMIO modes.

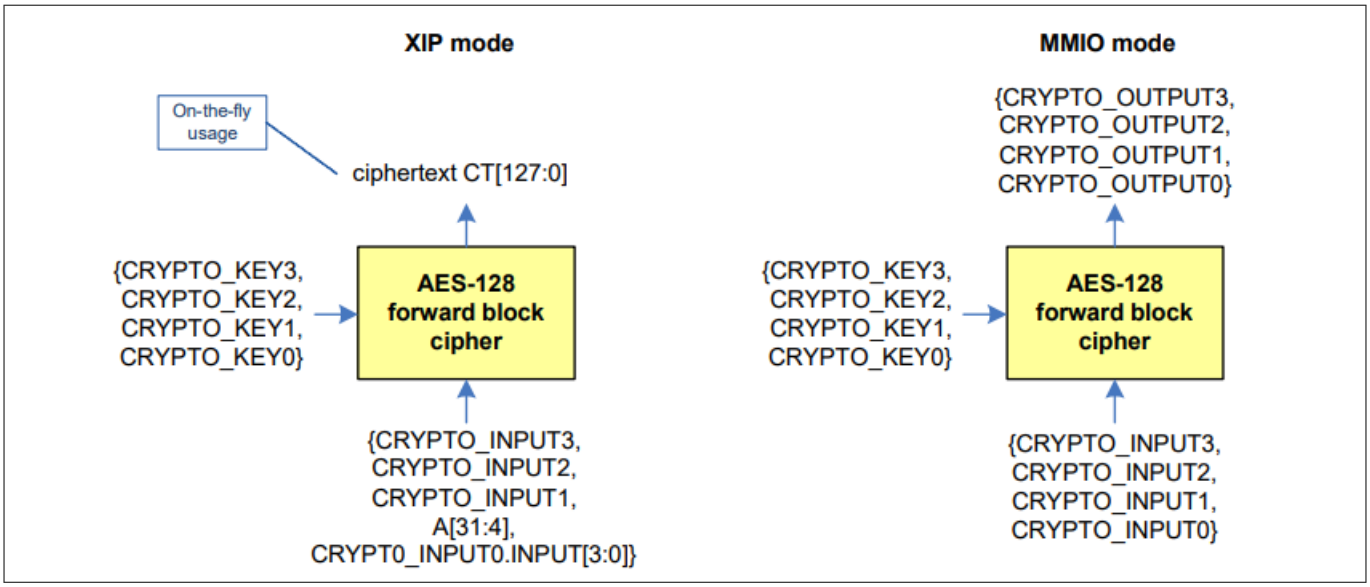


Figure 298 Cryptography in XIP and MMIO modes

In XIP mode, the resulting ciphertext `CT[]` (generated by encrypting the address) is XORed with the memory transfer's read or write data. It is important to note that the AES-128 block cipher operates on the address of the data, not the data itself.

For memory read transfers, this means that as long as the latency of the memory transfer's read data is greater than the latency of the AES-128 block cipher operation, the on-the-fly decryption does not introduce any additional delay. Figure 299 provides an illustration of the complete XIP mode functionality.

XIP mode encrypts only the address and XORs the result with the data. To replicate the same functionality in MMIO mode, you must provide the address as the `PT[]` input in the `CRYPTO_INPUTx` registers.

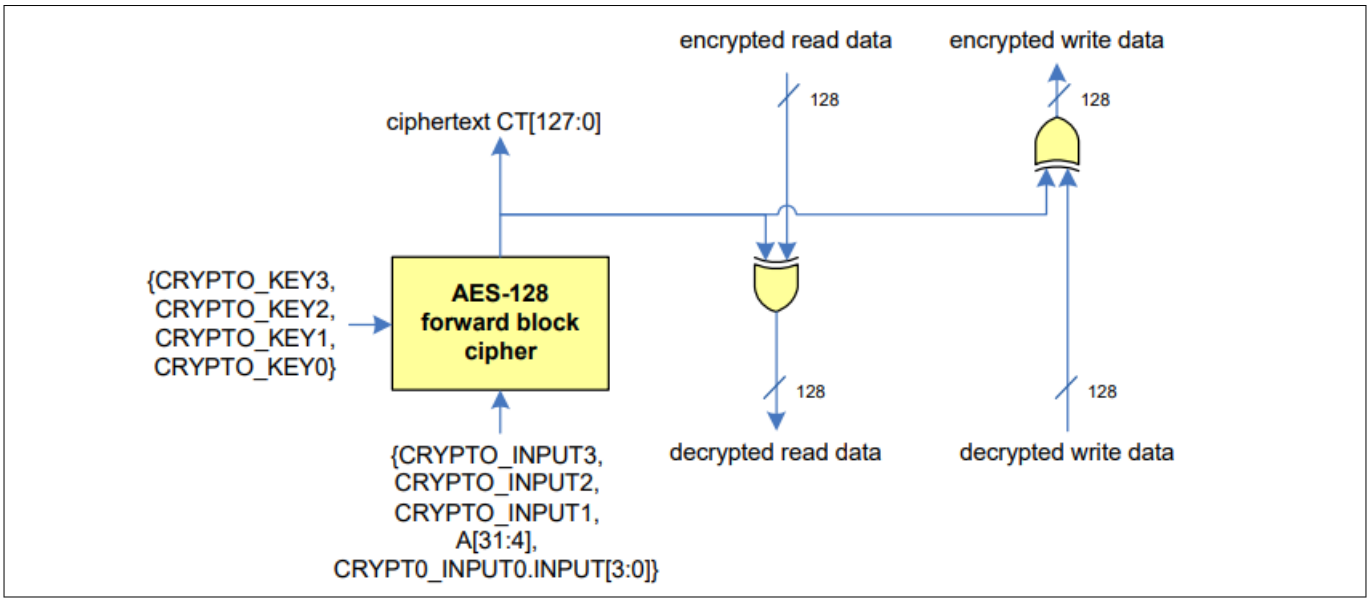


Figure 299 XIP mode functionality

26.3 Memory device signal interface

The SMIF functions as a master for SPI applications. SPI requires the definition of clock polarity and phase. In SPI mode, the SMIF supports a single clock polarity and phase configuration:

- **Clock polarity (CPOL) is '0'**: The base value of the clock is 0
- **Clock phase (CPHA) is '0'**: Data is driven on the falling edge of the clock, and data capture is specified by `CTL.CLOCK_IF_RX_SEL`

This configuration is commonly referred to as SPI configuration 0 and is supported by SPI memory devices.

26.3.1 Specifying memory devices

The SMIF requires memory devices to be defined for their operation in XIP mode. The SMIF supports up to four memory devices. Each memory device is defined through a set of registers. The memory device-specific register structure includes:

- **Device base address and capacity**: The `ADDR` register specifies the memory device's base address in the CYW20829 address space, and the `MASK` register specifies the memory device's size or capacity. If a memory device is not present or is disabled, the `ADDR` and `MASK` registers define a memory device with 0 B capacity. Typically, the devices' address regions in the CYW20829 address space are non-overlapping (except in dual-quad SPI mode) to ensure that the activation of select signals is mutually exclusive
- **Device data signal connections**: This is described in Connecting SPI memory devices
- **Definition of a read transfer**: This supports XIP mode functionality
- **Definition of a write transfer**: This also supports XIP mode functionality

Each memory device uses a dedicated device-select signal. For example, memory device 0 uses `select[0]`, memory device 1 uses `select[1]`, and so on. This establishes a fixed one-to-one connection between a memory device, its register set, and its select signal connection.

In XIP mode, the XIP AHB-Lite bus transfer address is compared with the device region. If the address falls within the device region, the corresponding device-select signal is activated. If an XIP AHB-Lite bus transfer address falls within multiple regions (this is possible if the device regions overlap), all associated device-select signals are activated. This overlap enables XIP operation in dual-quad SPI mode, where the command, address, and mode bytes are driven to two quad SPI devices simultaneously.

In XIP mode, dual-quad SPI mode requires the `ADDR_CTL.DIV2` field of the selected memory devices to be set to '1'. When this field is set to '1':

- The transfer address is divided by 2 and provided to the memory devices as the divided-by-2 address
- Each memory device contributes a 4-bit nibble for every 8-bit byte. Because the memory devices are quad SPI memories with a byte interface, the transfer size must be a multiple of 2

The `XIP_ALIGNMENT_ERROR` interrupt cause is set under the following conditions (in XIP mode and when `ADDR_CTL.DIV2` is '1'):

- **Transfer address is not a multiple of 2**: In this case, the divided-by-2 address provided to the memory devices is incorrect
- **Transfer size is not a multiple of 2**: In this case, the memory devices contribute only a nibble of a byte. This is unsupported because the memory devices have a byte interface

26.3.2 Connecting SPI memory devices

Memory device I/O signals (SCK, CS, SI/IO0, SO/IO1, IO2, IO3, IO4, IO5, IO6, IO7) are connected to the SMIF I/O signals (`clk`, `select[3:0]`, and `spi_data[7:0]`). Not all memory devices provide the same number of I/O signals.

Table 187 Memory device I/O signals

Memory device	I/O signals
Single SPI memory	SCK, CS, SI, SO. This memory device has two data signals (SI and SO).

(table continues...)

Table 187 (continued) Memory device I/O signals

Memory device	I/O signals
Dual SPI memory	SCK, CS, IO0, IO1. This memory device has two data signals (IO0, IO1).
Quad SPI memory	SCK, CS, IO0, IO1, IO2, IO3. This memory device has four data signals (IO0, IO1, IO2, IO3).
Octal SPI memory	SCK, CS, IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7. This memory device has eight data signals (IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7).

Each memory device has a single clock signal (SCK), a single active-low select signal (CS), and multiple data signals (IO0, IO1, …).

- Each memory device is assigned a fixed select signal connection (select[3:0])
- Each memory device has programmable data signal connections (data[7:0]), configured using the CTL.DATA_SEL[1:0] field

The CTL.DATA_SEL[1:0] field specifies how a device's data signals are connected. It is responsible for configuring the data lines used by a slave device. This configuration is distinct from the select lines, which are used to address the four slaves of the SMIF master.

The SMIF interface uses this information to:

- Drive data on the correct spi_data[] outputs
- Capture data from the correct spi_data[] inputs

If multiple device-select signals are activated, the same data is driven to all selected devices simultaneously.

Not all data signal connections are valid or supported. The supported connections depend on the type of memory device.

Table 188 Data signal connections

DATA_SEL[1:0]	Single SPI device	Dual SPI device	Quad SPI device	Octal SPI device
1	spi_data[2] = SIspi_data[3] = SO	spi_data[2] = IO0spi_data[3] = IO1	Ilegal	Illegal
2	spi_data[4] = SIspi_data[5] = SO	spi_data[4] = IO0spi_data[5] = IO1	spi_data[4] = IO0… spi_data[7] = IO3	Illegal
3	spi_data[6] = SIspi_data[7] = SO	spi_data[6] = IO0spi_data[7] = IO1	Illegal	Illegal
0	spi_data[0] = SIspi_data[1] = SO	spi_data[0] = IO0spi_data[1] = IO1	spi_data[0] = IO0… spi_data[3] = IO3	spi_data[0] = IO0… spi_data[7] = IO7

Memory devices can:

- Use shared data signal connections
- Use dedicated data signal connections, which reduce the load on the data lines. This enables faster signal-level changes, allowing for a faster I/O interface

Note that dual-quad SPI mode requires dedicated data signals to enable simultaneous read and/or write data transfers to and from two quad SPI devices.

Figure 300 shows memory Device 0, which is a single SPI memory with data signal connections to spi_data[1:0].

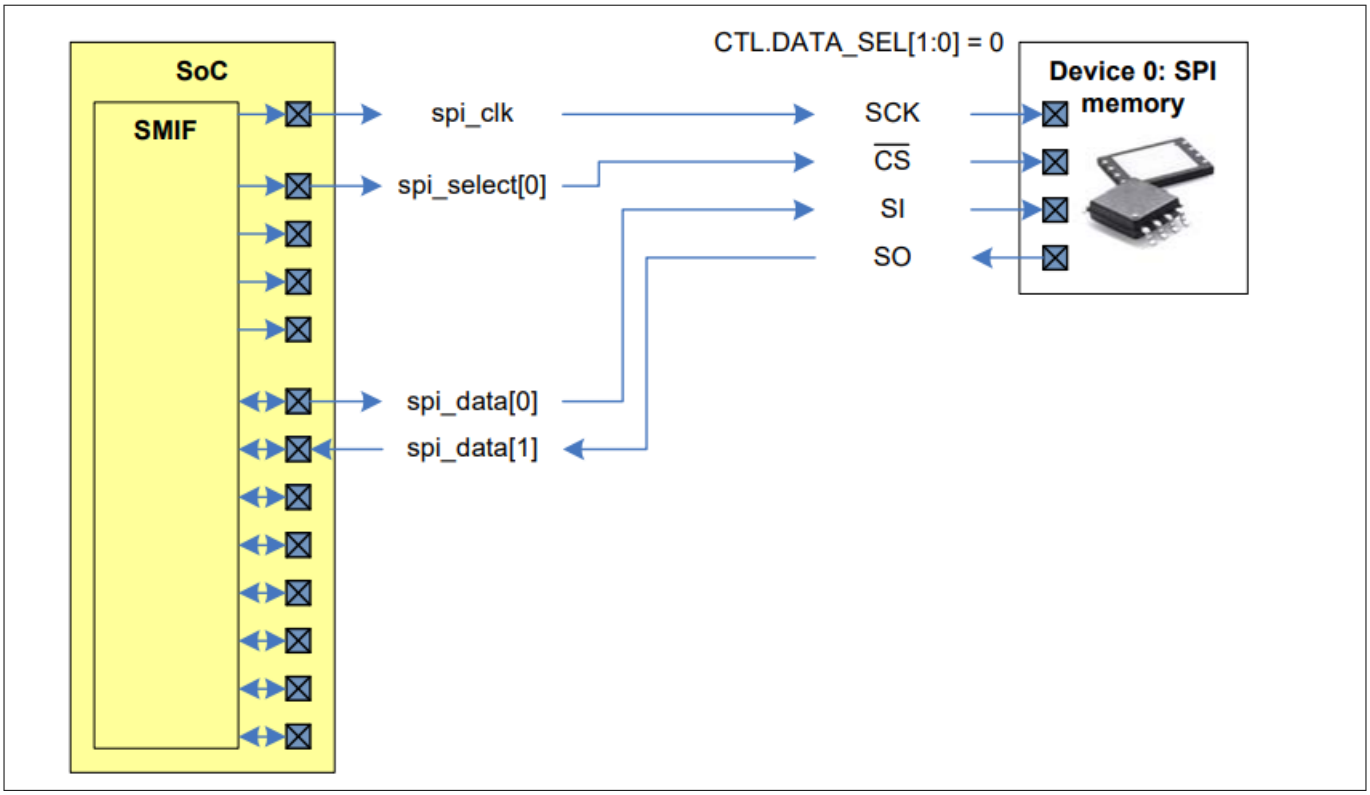


Figure 300 Single SPI memory Device 0 connected to spi_data[1:0]

Due to the pin layout, you may choose to connect a memory device to specific data lines. Figure 30-5 illustrates memory Device 0, which is a single SPI memory with data signal connections to `spi_data[7:6]`.

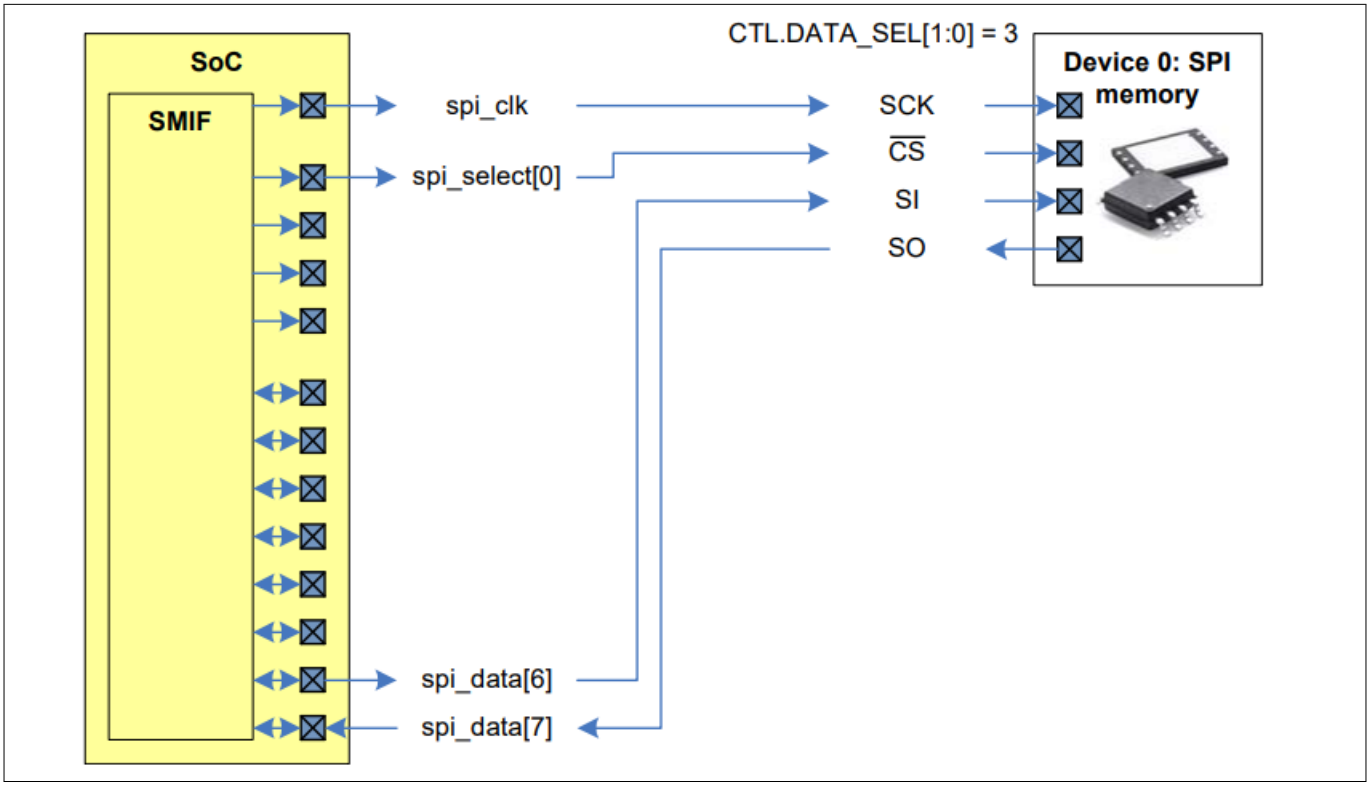


Figure 301 Single SPI memory Device 0 connected to spi_data[7:6]

Figure 302 illustrates memory devices 0 and 1, both configured as single SPI memories. Each device uses dedicated data-signal connections. The device-address regions within the CYW20829 address space must be non-overlapping to ensure that the activation of select[0] and select[1] remains mutually exclusive.

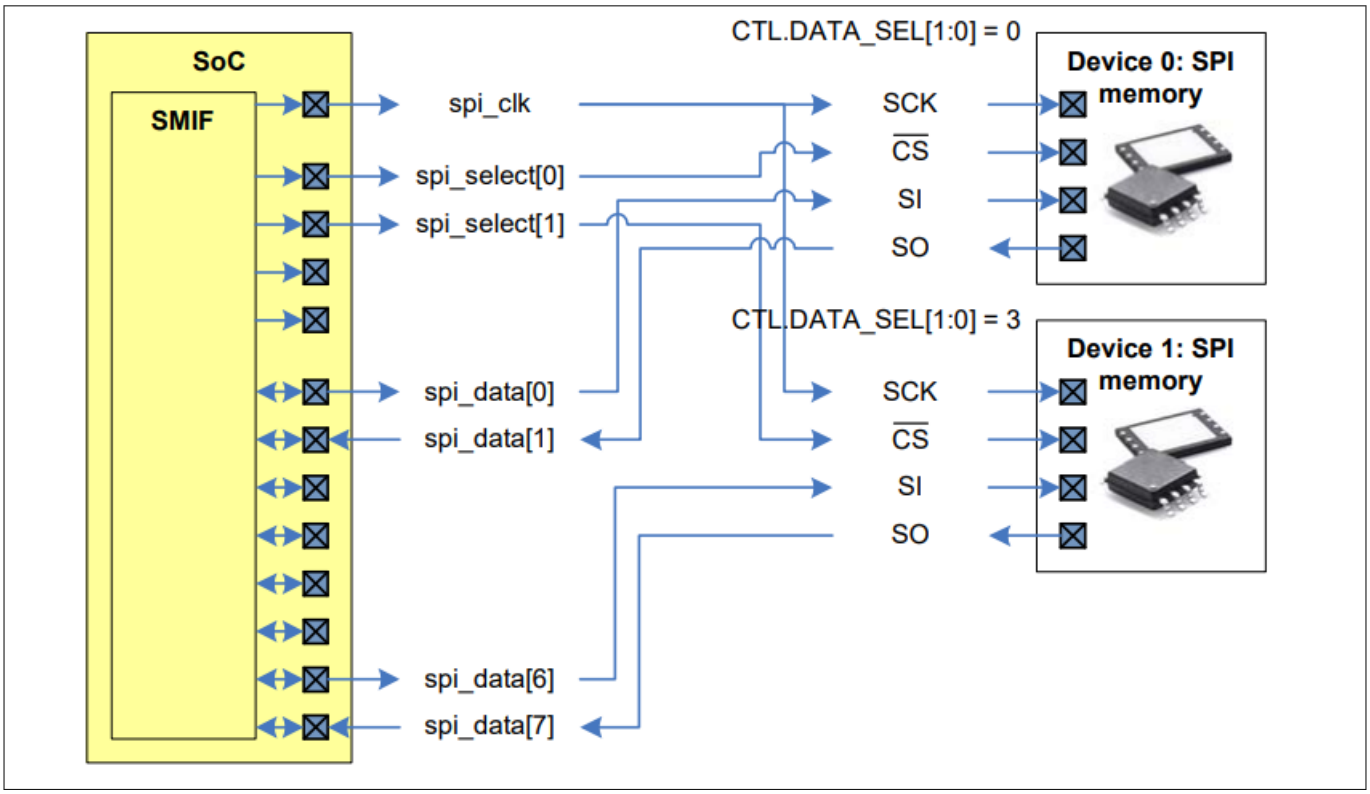


Figure 302 Single SPI memory devices 0 and 1 - Dedicated data signal

Figure 303 illustrates memory devices 0 and 1, both configured as single SPI memories. Both devices use shared data-signal connections. The device-address regions in the CYW20829 address space must be non-overlapping to ensure that the activation of select[0] and select[1] remains mutually exclusive. This configuration increases the load on the data lines, which may result in a slower I/O interface.

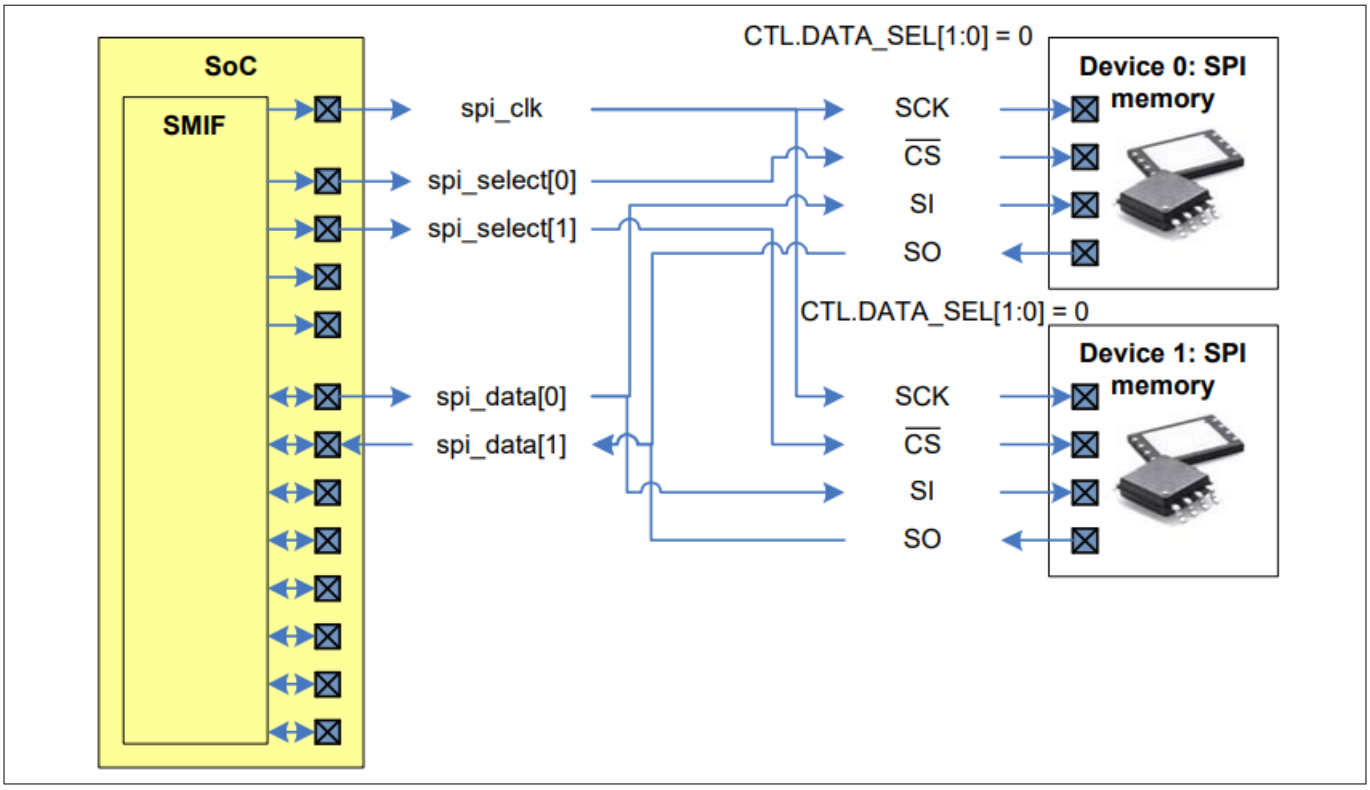


Figure 303 Single SPI memory devices 0 and 1 - Shared data signal

Figure 304 illustrates memory Device 0, a quad SPI memory with data-signal connections to spi_data[7:4].

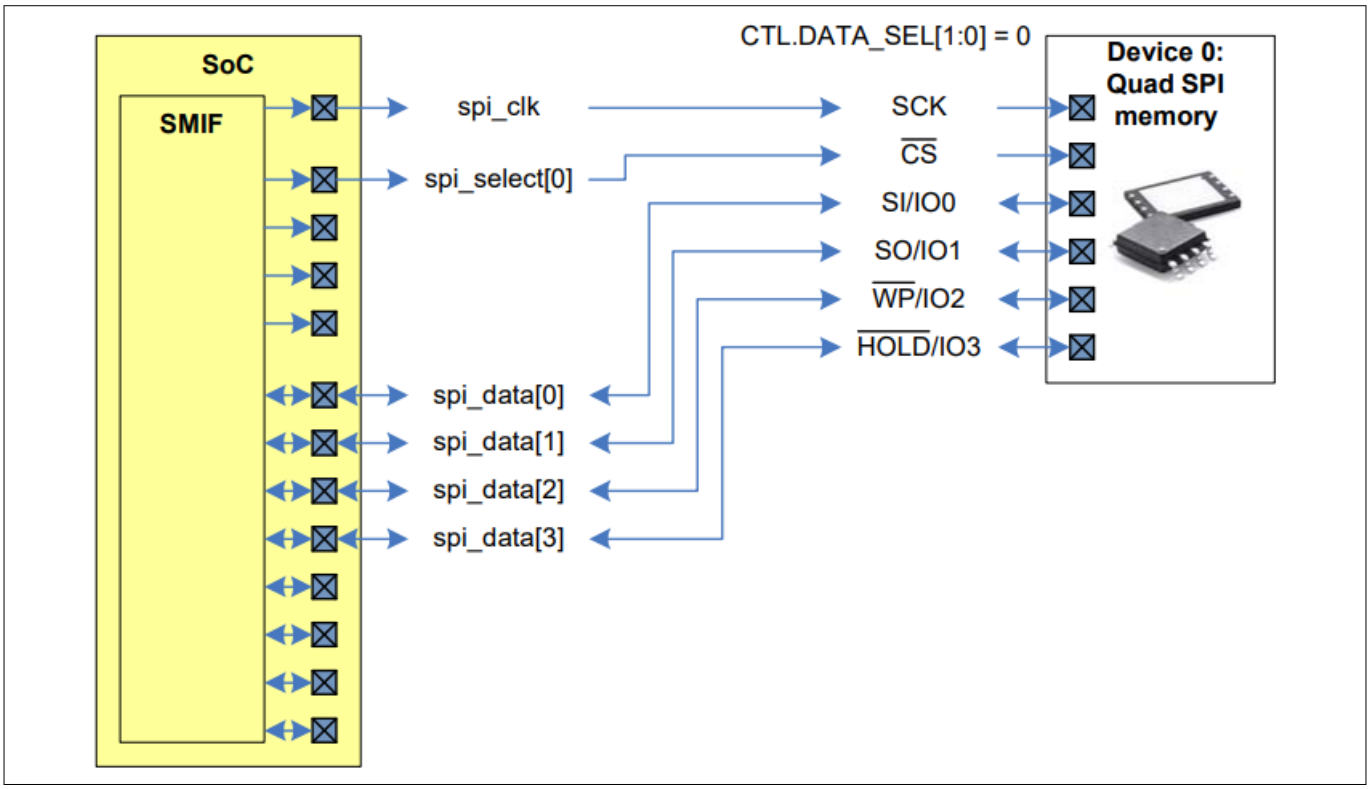


Figure 304 Quad SPI memory Device 0

Figure 305 illustrates memory devices 0 and 1. Device 0 is a single SPI memory, and Device 1 is a quad SPI memory. Each device uses dedicated data-signal connections. The device-address regions in the CYW20829

address space must be non-overlapping to ensure that the activation of select[0] and select[1] remains mutually exclusive.

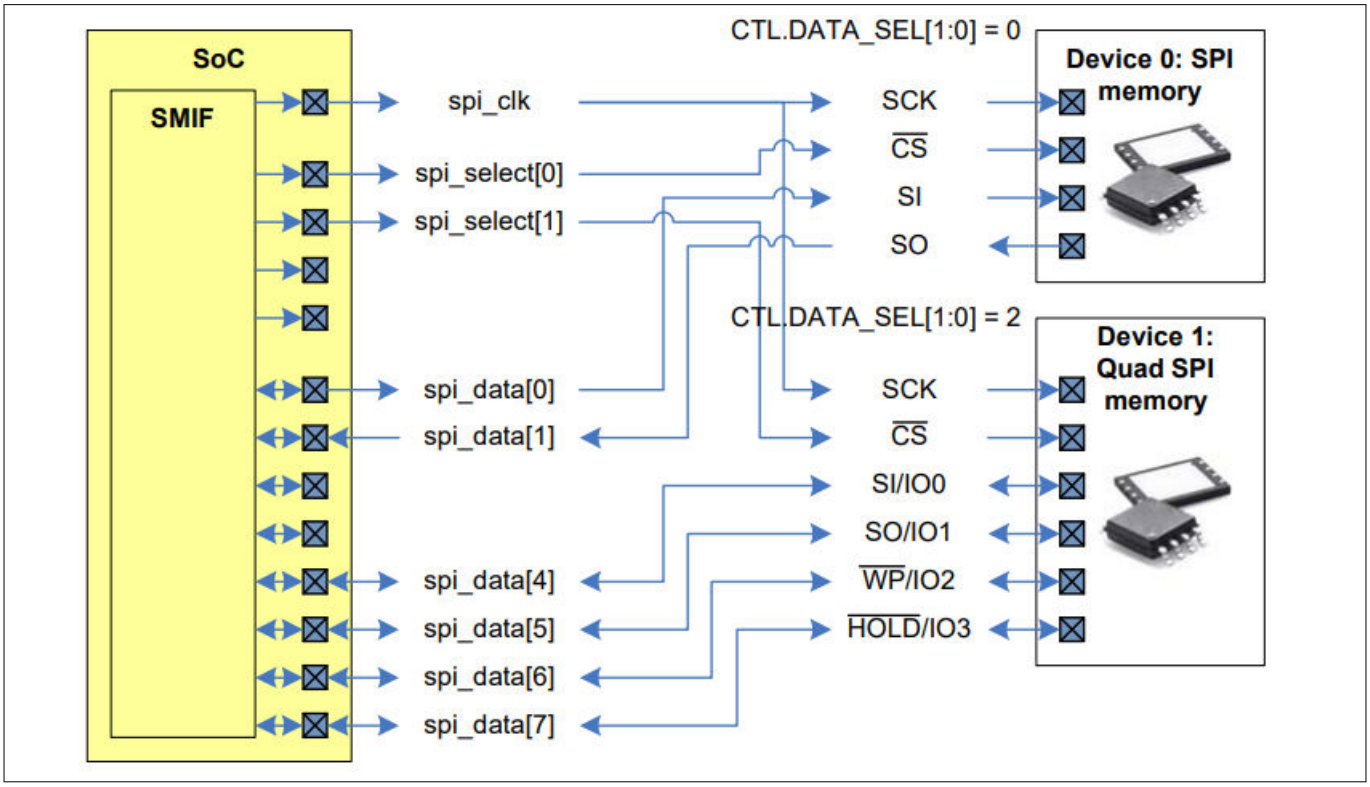


Figure 305 Single SPI memory 0 and Quad SPI memory 1 - Dedicated data signal

Figure 303 illustrates memory devices 0 and 1. Device 0 is a single SPI memory, and Device 1 is a quad SPI memory. Both devices use shared data-signal connections. The device-address regions in the CYW20829 address space must be non-overlapping to ensure that the activation of select[0] and select[1] remains mutually exclusive.

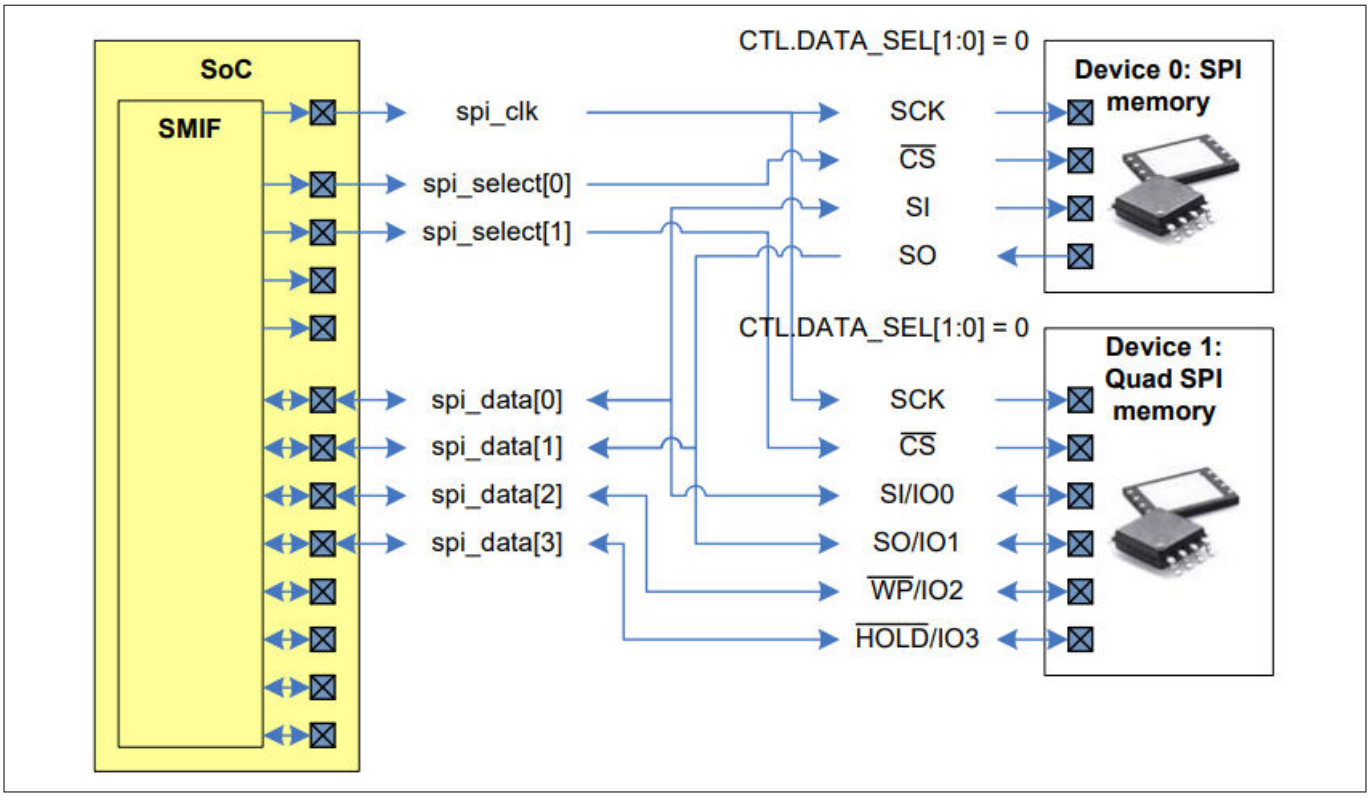


Figure 306 Single SPI memory Device 0 and Quad SPI memory Device 1 - Shared data signal

Figure 307 illustrates memory devices 0 and 1, both of which are quad SPI memories. Each device uses dedicated data-signal connections. The device-address regions in the CYW20829 address space are identical to ensure that the activation of select[0] and select[1] occurs simultaneously in XIP mode. This configuration is known as a dual-quad setup. During SPI read and write transfers, each device provides a nibble of a byte.

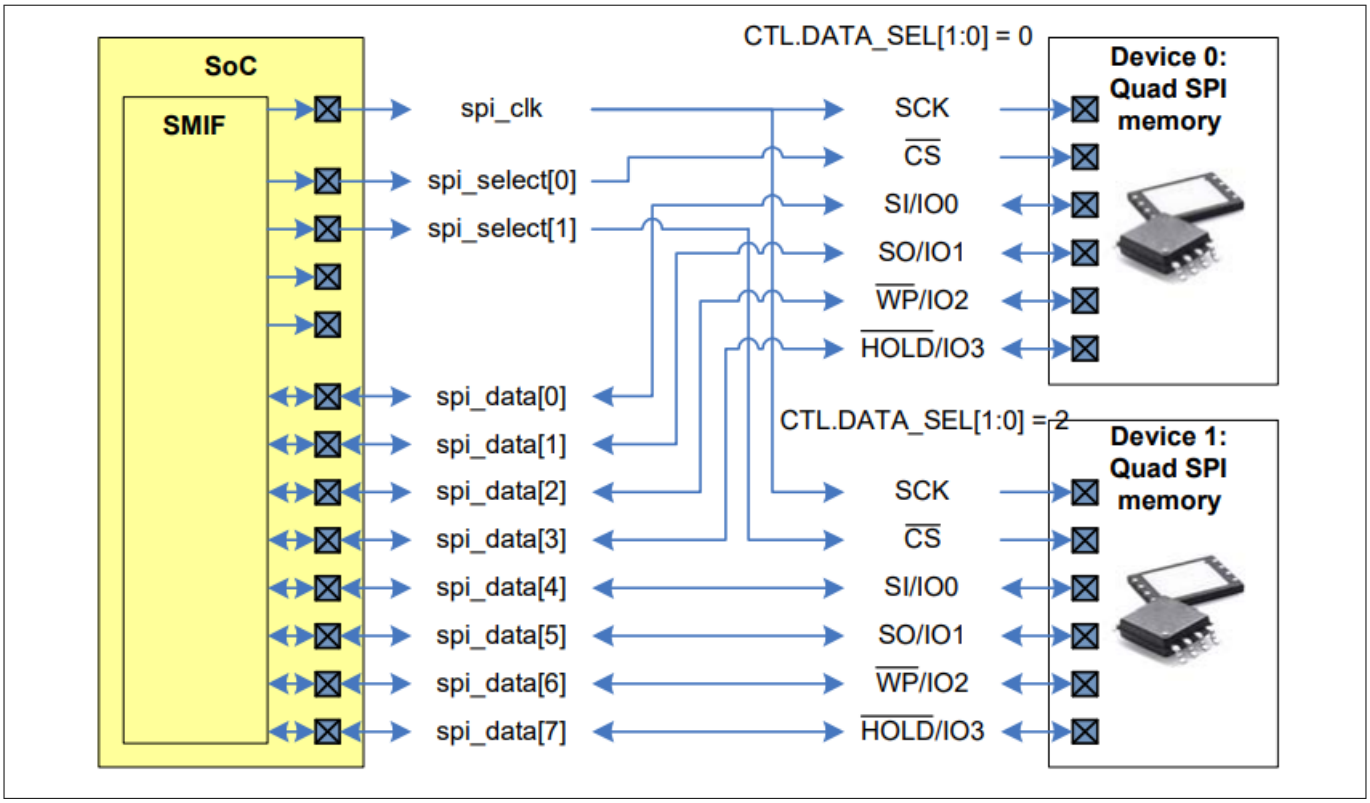


Figure 307 Quad SPI memory Devices 0 and 1

Figure 308 illustrates memory device 0, an octal SPI memory with data-signal connections to spi_data[7:0].

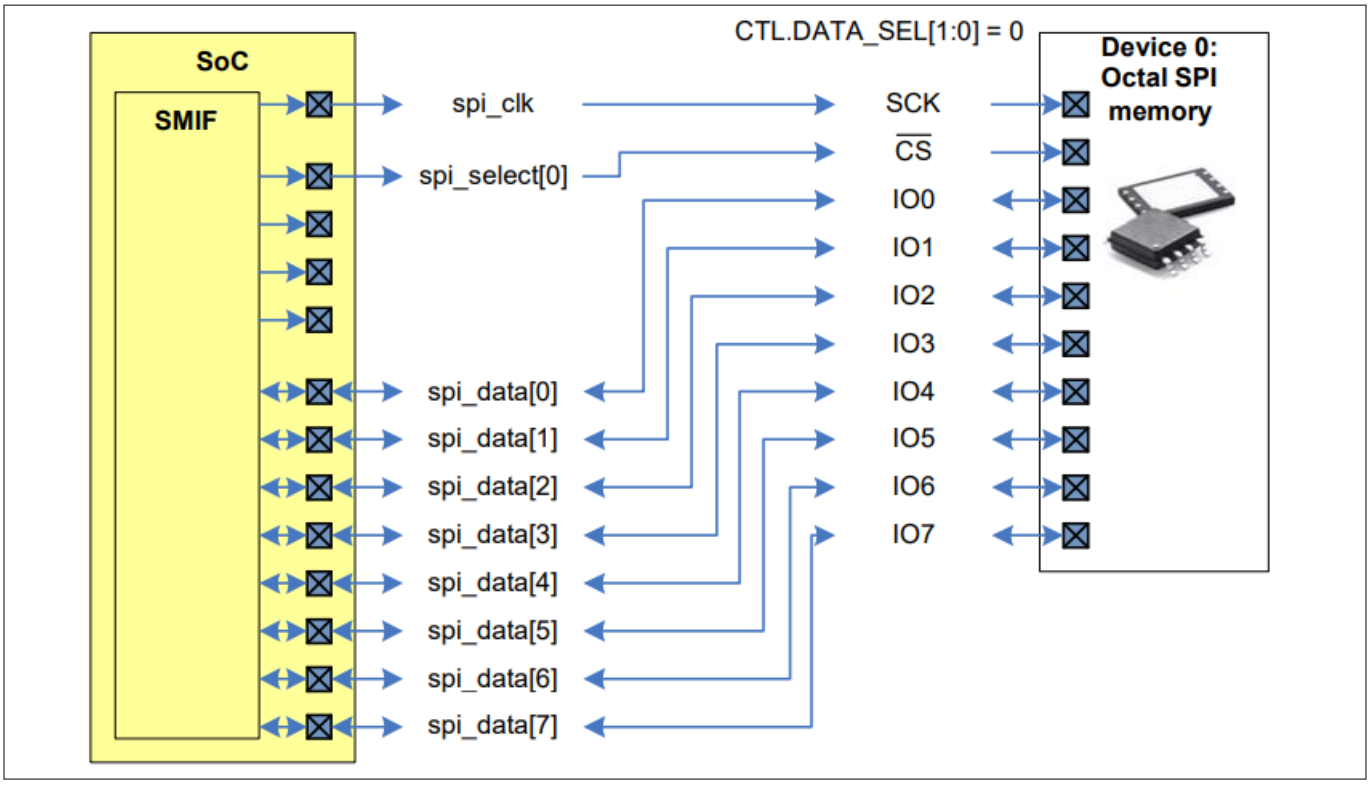


Figure 308 Octal SPI memory Device 0

26.3.3 SPI data transfer

SPI data transfer uses the most-significant bit (MSb) for the first data transfer. For a byte B, consisting of bits b7, b6, …, b0, bit b7 is transferred first, followed by bit b6, and so on. In dual, quad, dual-quad, and octal SPI transfers, multiple bits are transferred per cycle. For a single SPI device with data-signal connections to spi_data[1:0] (DATA_SEL set to “0”), Table 189 summarizes the transfer of byte B.

Table 189 Transfer of byte B

Cycle	Data transfer
0	For a write transfer: b7 is transferred on data[0] and SI/IO0.For a read transfer: b7 is transferred on data[1] and SO/IO1.
1	For a write transfer: b6 is transferred on data[0] and SI/IO0.For a read transfer: b6 is transferred on data[1] and SO/IO1.
2	For a write transfer: b5 is transferred on data[0] and SI/IO0.For a read transfer: b5 is transferred on data[1] and SO/IO1.
3	For a write transfer: b4 is transferred on data[0] and SI/IO0.For a read transfer: b4 is transferred on data[1] and SO/IO1.
4	For a write transfer: b3 is transferred on data[0] and SI/IO0.For a read transfer: b3 is transferred on data[1] and SO/IO1.
5	For a write transfer: b2 is transferred on data[0] and SI/IO0.For a read transfer: b2 is transferred on data[1] and SO/IO1.
6	For a write transfer: b1 is transferred on data[0] and SI/IO0.For a read transfer: b1 is transferred on data[1] and SO/IO1.
7	For a write transfer: b0 is transferred on data[0] and SI/IO0.For a read transfer: b0 is transferred on data[1] and SO/IO1.

In single SPI data transfer, the data signals are uni-directional. In the table, data[0] is exclusively used for write data and is connected to the device's SI input signal, while data[1] is exclusively used for read data and is connected to the device's SO output signal.

For a dual SPI device with data-signal connections to data[1:0] (DATA_SEL set to “0”), Table 190 summarizes the transfer of byte B.

Table 190 Dual data transfer

Cycle	Data transfer
0	b7, b6 are transferred on data[1:0] and IO1, IO0.
1	b5, b4 are transferred on data[1:0] and IO1, IO0.
2	b3, b2 are transferred on data[1:0] and IO1, IO0.
3	b1, b0 are transferred on data[1:0] and IO1, IO0.

For a quad SPI device with data-signal connections to data[3:0] (DATA_SEL set to “0”), Table 191 summarizes the transfer of byte B.

Table 191 Quad data transfer

Cycle	Data transfer
0	b7, b6, b5, b4 are transferred on data[3:0] and IO3, IO2, IO1, IO0.
1	b3, b2, b1, b0 are transferred on data[3:0] and IO3, IO2, IO1, IO0.

For an octal SPI device with data-signal connections to data[7:0] (DATA_SEL set to “0”), Table 192 summarizes the transfer of byte B.

Table 192 Octal data transfer

Cycle	Data transfer
0	b7, b6, b5, b4, b3, b2, b1, b0 are transferred on data[7:0] and IO7, IO6, IO5, IO4, IO3, IO2, IO1, IO0.

In dual-quad SPI mode, two quad SPI devices are used.

- The first device (the device with the lower device structure index) must have data-signal connections to data[3:0] (DATA_SEL set to 0)
- The second device (the device with the higher device structure index) must have data-signal connections to data[7:4] (DATA_SEL set to 2)

The command and data phases of the SPI transfer use different data-transfer widths:

- The command, address, and mode bytes use quad SPI data transfer
- The read data and write data use octal SPI data transfer

Each device provides a nibble of each data byte: the first device provides the lower nibble, while the second device provides the higher nibble. Table 193 summarizes the transfer for read and write data byte B.

Table 193 Dual-quad SPI mode, Octal data transfer

Cycle	Data transfer
0	b7, b6, b5, b4 are transferred on data[7:4] and second device IO3, IO2, IO1, IO0. b3, b2, b1, b0 are transferred on data[3:0] and first device IO3, IO2, IO1, IO0.

26.3.4 Example of setting up SMIF

Devices 0 and 1 are used to implement dual-quad SPI mode. Both devices are 1 MB (8 Mb) in size, and the address requires 3 bytes. Device 0 has data-signal connections to data[3:0], while Device 1 has data-signal connections to data[7:4].

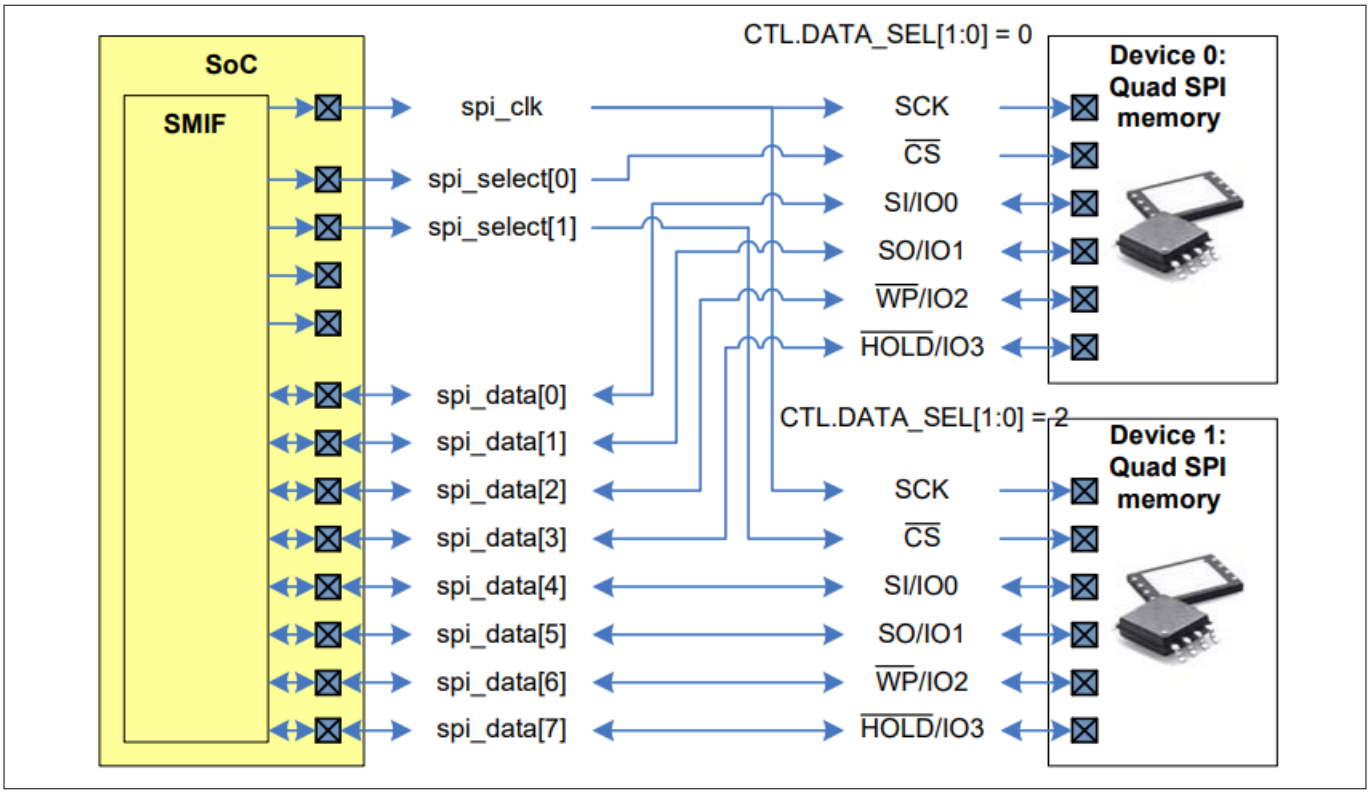


Figure 309 Setting up SMIF

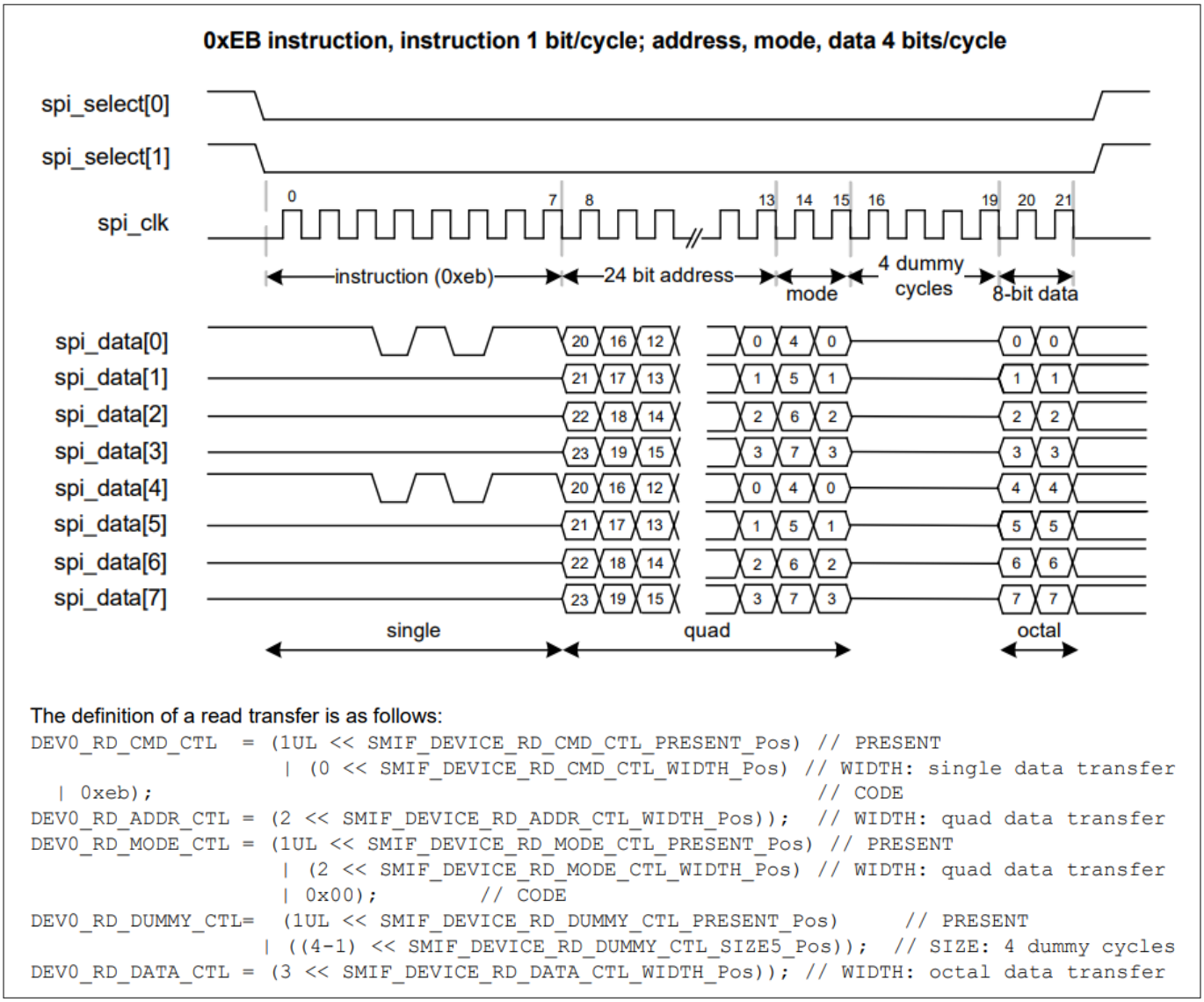
In dual-quad SPI mode, the AHB-Lite bus transfer address is divided by two. Cryptography and write functionality are disabled in the following example configuration:

```
#define MASK_1MB 0xfff00000

DEV0_ADDR = CPUSS_SMIF_BASE;
DEV0_MASK = MASK_1MB; // MASK: 1 MB region
DEV0_CTL = (0 << SMIF_DEVICE_CTL_DATA_SEL_Pos) // DATA_SEL: data[3:0]
         | (0 << SMIF_DEVICE_CTL_CRYPTO_EN_Pos) // CRYPTO_EN: disabled
         | (0 << SMIF_DEVICE_CTL_WR_EN_Pos); // WR_EN: disabled
DEV0_ADDR_CTL = (1 << SMIF_DEVICE_ADDR_CTL_DIV2_Pos) // DIV2: enabled
              | ((3 - 1) << SMIF_DEVICE_ADDR_CTL_SIZE2_Pos); // SIZE: 3-byte address

DEV1_ADDR = CPUSS_SMIF_BASE;
DEV1_MASK = 0xfff00000; // MASK: 1 MB region
DEV1_CTL = (2 << SMIF_DEVICE_CTL_DATA_SEL_Pos) // DATA_SEL: data[7:4]
         | (0 << SMIF_DEVICE_CTL_CRYPTO_EN_Pos) // CRYPTO_EN: disabled
         | (1 << SMIF_DEVICE_CTL_WR_EN_Pos); // WR_EN: enabled
DEV1_ADDR_CTL = (1 << SMIF_DEVICE_ADDR_CTL_DIV2_Pos) // DIV2: enabled
              | ((3 - 1) << SMIF_DEVICE_ADDR_CTL_SIZE2_Pos); // SIZE: 3-byte address
```

For XIP read transfers, the 0xEB command/instruction is used. Figure 310 illustrates a two-byte transfer from Devices 0 and 1 in dual-quad SPI mode.



The definition of a read transfer is as follows:

```
DEV0_RD_CMD_CTL   = (1UL << SMIF_DEVICE_RD_CMD_CTL_PRESENT_Pos) // PRESENT
                  | (0 << SMIF_DEVICE_RD_CMD_CTL_WIDTH_Pos) // WIDTH: single data transfer
  | 0xeb);                                                  // CODE
DEV0_RD_ADDR_CTL = (2 << SMIF_DEVICE_RD_ADDR_CTL_WIDTH_Pos));  // WIDTH: quad data transfer
DEV0_RD_MODE_CTL = (1UL << SMIF_DEVICE_RD_MODE_CTL_PRESENT_Pos) // PRESENT
                  | (2 << SMIF_DEVICE_RD_MODE_CTL_WIDTH_Pos) // WIDTH: quad data transfer
                  | 0x00);         // CODE
DEV0_RD_DUMMY_CTL=  (1UL << SMIF_DEVICE_RD_DUMMY_CTL_PRESENT_Pos)      // PRESENT
                 | ((4-1) << SMIF_DEVICE_RD_DUMMY_CTL_SIZE5_Pos));  // SIZE: 4 dummy cycles
DEV0_RD_DATA_CTL = (3 << SMIF_DEVICE_RD_DATA_CTL_WIDTH_Pos)); // WIDTH: octal data transfer
```

Figure 310 Two-byte transfer in dual-quad SPI mode

***Note**: The command uses single-data transfer, the address and mode byte use quad-data transfer, and the read data byte uses octal-data transfer.*

26.4 Triggers

The SMIF has two level-sensitive triggers:

- tr_tx_req is associated with the Tx data FIFO
- tr_rx_req is associated with the Rx data FIFO

The trigger functionality depends on the SMIF configuration:

- If the SMIF is enabled (CTL.ENABLED is 1) and MMIO operation mode is selected (CTL.XIP_MODE is 0), the trigger functionality is enabled
- If the SMIF is disabled (CTL.ENABLED is 0) or XIP operation mode is selected (CTL.XIP_MODE is 1), the trigger functionality is disabled

The behavior of the triggers is defined as follows:

1. The MMIO TX_DATA_FIFO_CTL.TRIGGER_LEVEL field specifies a number of FIFO entries. The tr_tx_req trigger becomes active when the number of used Tx data FIFO entries is less than or equal to the specified number.
   - This condition is expressed as: TX_DATA_FIFO_STATUS.USED ≤ TRIGGER_LEVEL
2. The MMIO RX_DATA_FIFO_CTL.TRIGGER_LEVEL field specifies a number of FIFO entries. The tr_rx_req trigger becomes active when the number of used Rx data FIFO entries exceeds the specified number.
   - This condition is expressed as: RX_DATA_FIFO_STATUS.USED > TRIGGER_LEVEL

26.5 Interrupts

The SMIF has a single interrupt output with six interrupt causes:

- **INTR.TR_TX_REQ**: This interrupt is triggered in MMIO mode when the tr_tx_req trigger is activated
- **INTR.TR_RX_REQ**: This interrupt is triggered in MMIO mode when the tr_rx_req trigger is activated
- **INTR.XIP_ALIGNMENT_ERROR**: This interrupt is triggered in XIP mode when the selected device's ADDR_CTL.DIV2 field is set to 1 and either the AHB-Lite bus address is not a multiple of 2, or the requested transfer size is not a multiple of 2. This interrupt identifies erroneous behavior in dual-quad SPI mode when the ADDR_CTL.DIV2 field is set to 1
- **INTR.TX_CMD_FIFO_OVERFLOW**: This interrupt is triggered in MMIO mode during an AHB-Lite write transfer to the Tx command FIFO (TX_CMD_FIFO_WR) when there are insufficient free entries
- **INTR.TX_DATA_FIFO_OVERFLOW**: This interrupt is triggered in MMIO mode during an AHB-Lite write transfer to the Tx data FIFO (TX_DATA_FIFO_WR1, TX_DATA_FIFO_WR2, and TX_DATA_FIFO_WR4) when there are insufficient free entries
- **INTR.RX_DATA_FIFO_OVERFLOW**: This interrupt is triggered in MMIO mode during an AHB-Lite read transfer from the Rx data FIFO (RX_DATA_FIFO_RD1, RX_DATA_FIFO_RD2, and RX_DATA_FIFO_RD4) when there are insufficient free entries

26.6 Sleep operation

The SMIF hardware operates in both Sleep and Active power modes. In Sleep power mode, only DMA-driven transactions can be performed because the CPU, which acts as the SPI master, is not active.

26.7 Performance

Accessing external memory introduces some latency, which depends on the SMIF operation mode, the amount of data being transferred, caching, and cryptography. In MMIO mode, the number of interface clock cycles per transfer is determined by the following equation:

$$\underset{\textit{(in interface clock cycles)}}{\textit{Time to Transfer N bytes}} = \left(\left(\frac{\textit{opcode size}}{\textit{opcode width}}\right) + \left(\frac{\textit{address size}}{\textit{address width}}\right) + \left(\frac{\textit{mode size}}{\textit{mode width}}\right) + \textit{dummycycles} + \left(\frac{\textit{data size}}{\textit{data width}}\right)\right)$$

Figure 311 Equation

For example, in Figure 311, the number of interface clock cycles can be calculated using the following equation:

22 cycles = [8-bit instruction/1 single width] + [24-bit address/4 width] + [8-bit mode/4 width] + 4 dummy cycles + [16-bit data/8 width]

In XIP mode, performance is influenced by the cache.

- For data reads that hit in the cache, the read operation does not incur any interface clock cycles
- For data reads that miss in the cache, the operation proceeds as a normal SMIF read, and if prefetching is enabled, it results in two 16 B cache sub-sector refills
- Writes to external memory in XIP mode occur as normal SMIF write operations

Enabling cryptography may also impact SMIF performance.

- The AES-128 block cipher typically introduces a latency of 13 `clk_hf` cycles
  - For transfers lasting more than 13 cycles, on-the-fly decryption does not add any delay
  - For transfers lasting fewer than 13 cycles, the transfer latency is extended to 13 cycles
- When the cache is enabled, the 13-cycle encryption latency is incurred only once for every 16 B fetched for the cache

27 Inter-IC sound bus (I²S)

The inter-IC sound bus (I²S) is a serial-bus interface standard used to connect digital-audio devices. In addition to the standard I²S format, the I²S block supports the left-justified (LJ) format and the time-division-multiplexed (TDM) format.

27.1 Features

- Supports standard I²S, left-justified (LJ), and eight-channel time-division-multiplexed (TDM) digital-audio-interface formats
- Supports both master and slave mode operation in all digital-audio formats
- Supports independent operation of receive (Rx) and transmit (Tx) directions
- Operates from an external master clock provided through an external IC such as an audio codec
- Provides configurable clock-divider registers to generate the required sample rates
- Supports data-word lengths of 8-bit, 16-bit, 18-bit, 20-bit, 24-bit, and 32-bit per channel
- Supports channel lengths of 8-bit, 16-bit, 18-bit, 20-bit, 24-bit, and 32-bit per channel (channel length is fixed at 32-bit in TDM format)
- Provides two hardware FIFO buffers, one for the Tx block and one for the Rx block
- Supports both DMA- and CPU-based data transfers

27.2 Architecture

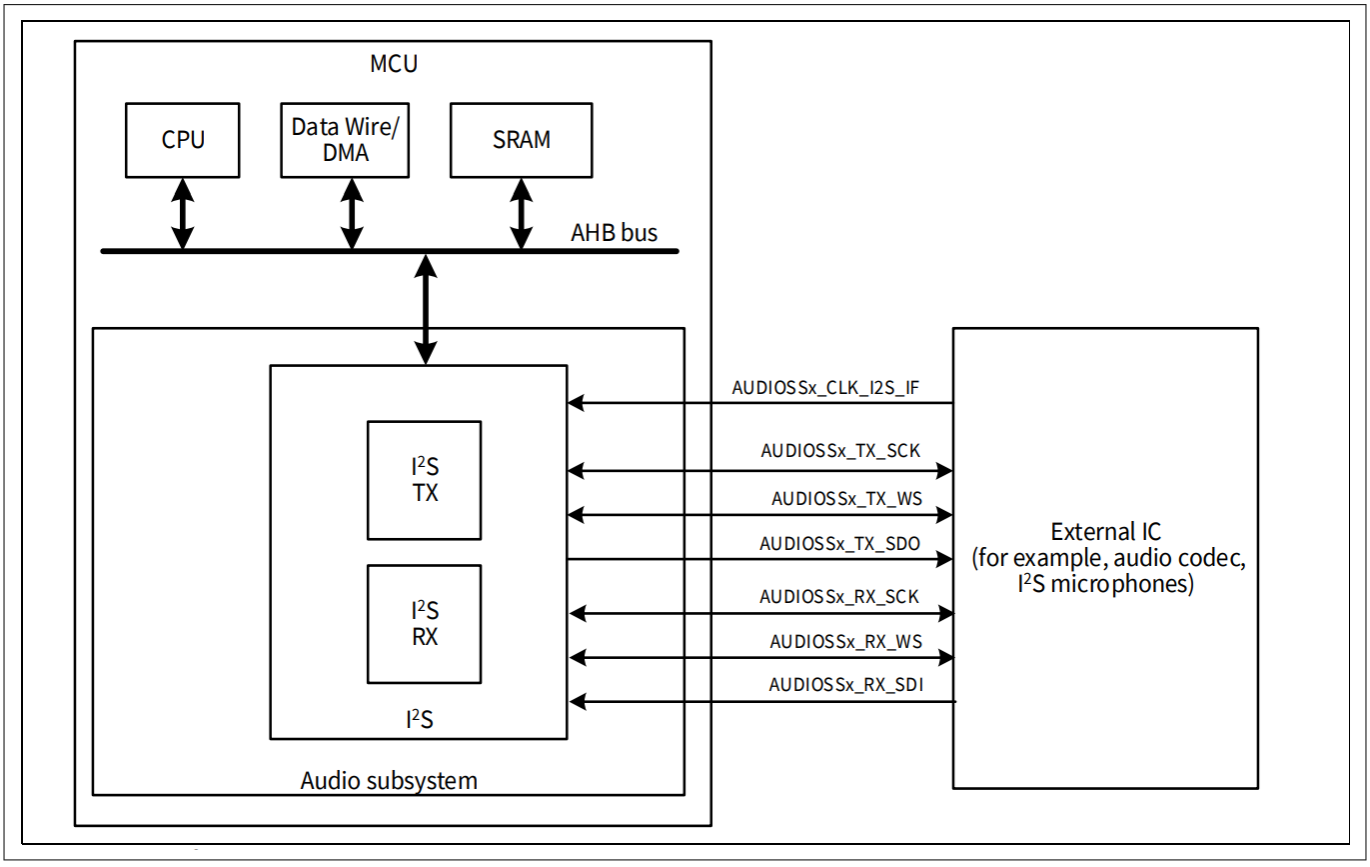


Figure 312 Audio subsystem I²S block diagram

Figure 312 shows the high-level block diagram of the I²S block, which consists of two sub-blocks – I²S transmitter (Tx) and I²S receiver (Rx). The digital-audio-interface format and master/slave mode configuration can be performed independently for the Tx and Rx blocks.

- In master mode, the word-select (WS) and serial-data clock (SCK) signals are generated by the I²S block in the MCU
- In slave mode, the WS and SCK signals are input signals to the MCU and are generated by an external master device

The I²S block configuration, control, and status registers, along with the FIFO data buffers, are accessible through the AHB bus. AHB-bus masters such as the CPU and DMA can access the I²S registers through the AHB interface. See the device datasheet for information on port-pin assignments of the I²S block signals and AC/DC electrical specifications.

27.3 Digital audio interface formats

The I²S block supports the following digital-audio-interface formats:

- Standard I²S format
- Left-justified (LJ) format
- Time-division-multiplexed (TDM) format

The Tx and Rx sub-blocks can be independently configured to support one of the above formats in either master or slave mode. The I2S_MODE bits in the I2Sx_TX_CTL and I2Sx_RX_CTL registers are used to configure the digital-audio-interface format for the Tx and Rx blocks, respectively. The MS (master/slave) bit in the I2Sx_TX_CTL and I2Sx_RX_CTL registers is used to configure the blocks to operate in either master or slave mode.

27.3.1 Standard I²S format

Figure 313 shows the timing diagrams for the different word-length and channel-length combinations in the standard I²S digital-audio format. In the standard I²S format, the word-select (WS) signal is low for left-channel data and high for right-channel data. The WS signal transitions one bit clock (SCK) early, relative to the start of the left or right channel data. All the serial data (SD) and WS signal transitions occur on the falling edge of the SCK signal, while the read operations on the WS and SD lines are typically performed on the rising edge of the SCK signal.

The I²S Tx block writes to the serial-data line (TX_SDO) on the falling edge of TX_SCK, while the I²S Rx block reads data (RX_SDI) on the rising edge of RX_SCK. The serial data is transmitted most significant bit (MSb) first. Depending on whether the block is in master or slave mode, the WS and SCK signals are either generated by the block (master mode) or serve as input signals to the block (slave mode).

The I²S block supports configurable word-length and channel-length selection options:

- The word length for the Tx and Rx blocks can be configured using the WORD_LEN bits in the I2Sx_TX_CTL and I2Sx_RX_CTL registers, respectively
- The channel length for the Tx and Rx blocks can be configured using the CH_LEN bits in the I2Sx_TX_CTL and I2Sx_RX_CTL registers, respectively
- The channel-length configuration must always be greater than or equal to the word-length configuration

When the I²S Rx block operates in slave mode, the master Tx device must ensure its channel-length configuration matches the I²S Rx block's channel-length setting. A mismatch in channel length will prevent proper operation of the I²S Rx block in slave mode.

In the Tx block, when the channel length is greater than the word length, the unused bits can be transmitted as either 0 or 1. This is configured using the OVHDATA bit in the I2Sx_TX_CTL register. In the Rx block, when the word length is less than 32 bits, the unused most significant bits written to the 32-bit RX_FIFO register can either be set to 0 or extended using the sign bit. This is configured using the BIT_EXTENSION bit in the I2Sx_RX_CTL register.

27 Inter-IC sound bus (I²S)

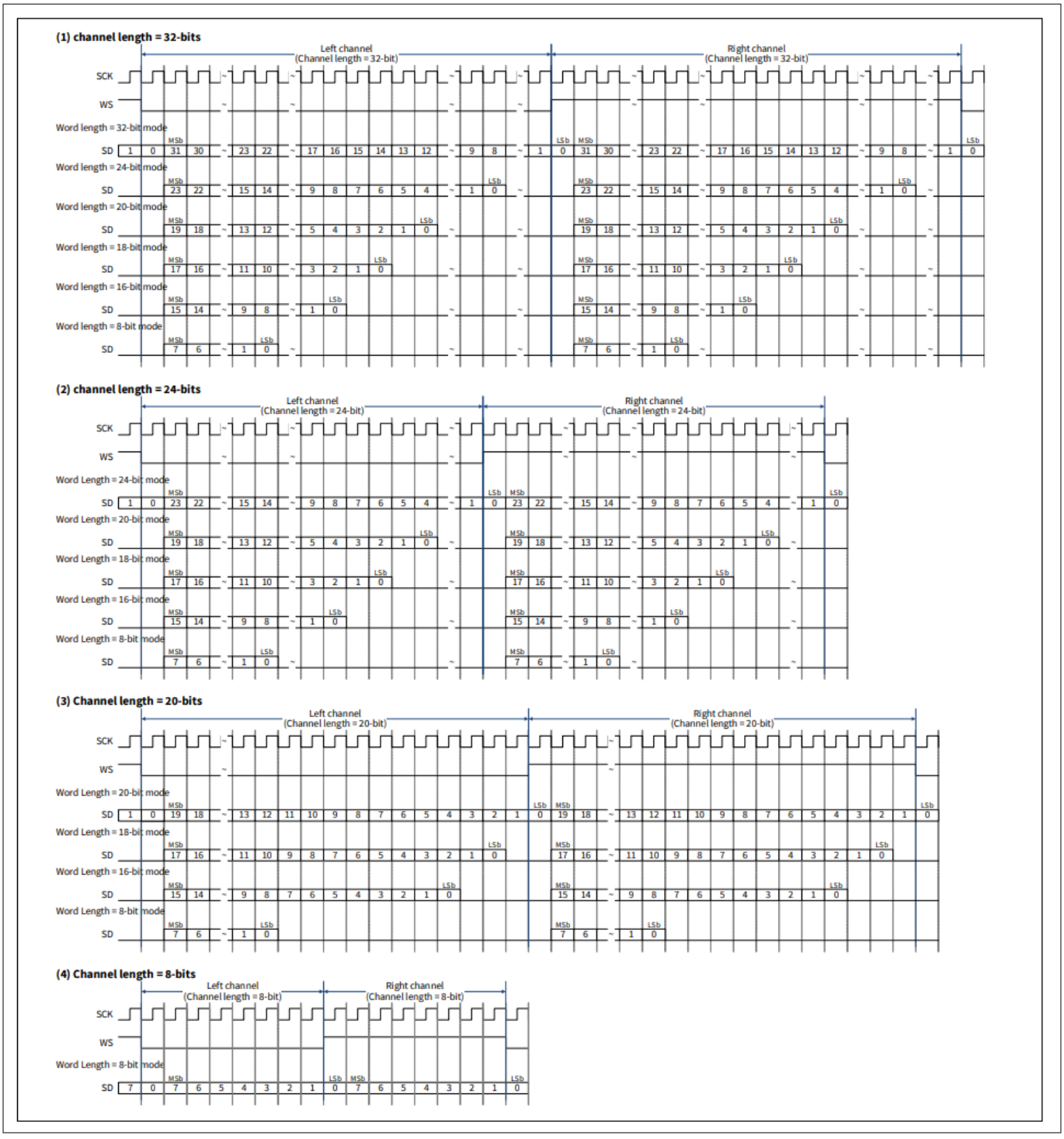


Figure 313 Standard I²S format (word-length and channel-length combination timing diagrams)

Table 194 lists the supported word-length and channel-length combinations.

Table 194 Word-length and channel-length combinations

Channel length	Word length					32-bit
	8-bit	16-bit	18-bit	20-bit	24-bit	
32-bit	Valid	Valid	Valid	Valid	Valid	Valid

(table continues...)

Table 194 (continued) Word-length and channel-length combinations

Channel length	Word length					32-bit
	8-bit	16-bit	18-bit	20-bit	24-bit	
24-bit	Valid	Valid	Valid	Valid	Valid	Invalid
30-bit	Valid	Valid	Valid	Valid	Invalid	Invalid
18-bit	Valid	Valid	Valid	Invalid	Invalid	Invalid
16-bit	Valid	Valid	Invalid	Invalid	Invalid	Invalid
8-bit	Valid	Invalid	Invalid	Invalid	Invalid	Invalid

27.3.2 Left-justified (LJ) format

Figure 314 shows the timing diagrams for the left-justified (LJ) interface format, using the 32-bit channel-length and 32-bit word-length configuration as an example. The only differences between the standard I²S and LJ formats are as follows:

- In the standard I²S format, the WS signal is low for left-channel data and high for right-channel data. In the LJ format, the WS signal is high for left-channel data and low for right-channel data
- In the standard I²S format, the WS signal transitions one bit clock (SCK) early, relative to the start of the channel data (coinciding with the least-significant bit (LSb) of the previous channel). In the LJ format, there is no early transition, and the WS signal transitions coincide with the start of the channel data

Apart from these differences, all the features explained in the Standard I²S format section apply to the LJ format as well.

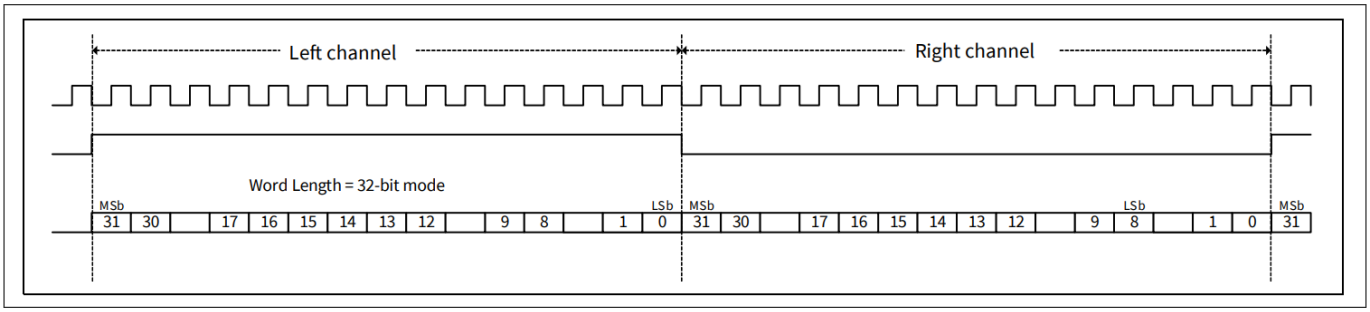


Figure 314 Left-justified digital audio format

27.3.3 Time-division-multiplexed (TDM) format

Figure 4 shows the timing diagrams for the two types of time-division-multiplexed (TDM) formats supported by the I²S block. The differences between the standard I²S/left-justified (LJ) formats and the TDM format are as follows:

- The standard I²S/LJ formats support only two channels (left/right) per frame, while the TDM format supports up to eight channels per frame
- In the TDM format, the channel length for all eight channels is fixed at 32 bits. In contrast, the channel length is configurable in the standard I²S/LJ formats. The word length per channel is configurable in the TDM format, similar to the standard I²S, and the data is transmitted in most significant bit (MSb) first order. In the I²S Tx block, when the word length per channel is less than the 32-bit channel length, the OVHDATA bit in the `I2Sx_TX_CTL` register is used to fill the unused least-significant channel data bits with either all zeros or all ones

- In the TDM format, all eight channels of data are always present in a frame, so the frame width is fixed at 256 bits. You can configure the number of active channels in a frame by setting the CH_NR bits in the I2Sx_TX_CTL and I2Sx_RX_CTL registers. In the standard I²S/LJ formats, CH_NR must always be configured for two channels. In the TDM format, the number of active channels can be less than or equal to eight. The unused (inactive) channels always follow the active channels in a frame. For example, if CH_NR is set for four channels, CH0 to CH3 are active channels, while CH4 to CH7 are unused channels. The OVHDATA bit in the I2Sx_TX_CTL register determines whether the unused channels are filled with all zeros or all ones
- The pulse width of the word-select (WS) signal in the TDM format can be configured to be either one bit clock (SCK) wide or one channel wide. The WS_PULSE bit in the I2Sx_TX_CTL and I2Sx_RX_CTL registers selects the pulse width. In contrast, the pulse width is fixed to one channel width in the I²S/LJ formats
- Two types of TDM formats are supported:
  - In TDM mode A, the rising edge of the WS signal, which signifies the start of a frame, coincides with the start of CH0 data
  - In TDM mode B, the rising edge of the WS signal, which signifies the start of a frame, is one bit clock (SCK) early, relative to the start of CH0 data (coinciding with the last bit of the previous frame)

The selection between the two TDM formats (mode A or mode B) is made using the I²S_MODE bits in the I2Sx_TX_CTL and I2Sx_RX_CTL registers.

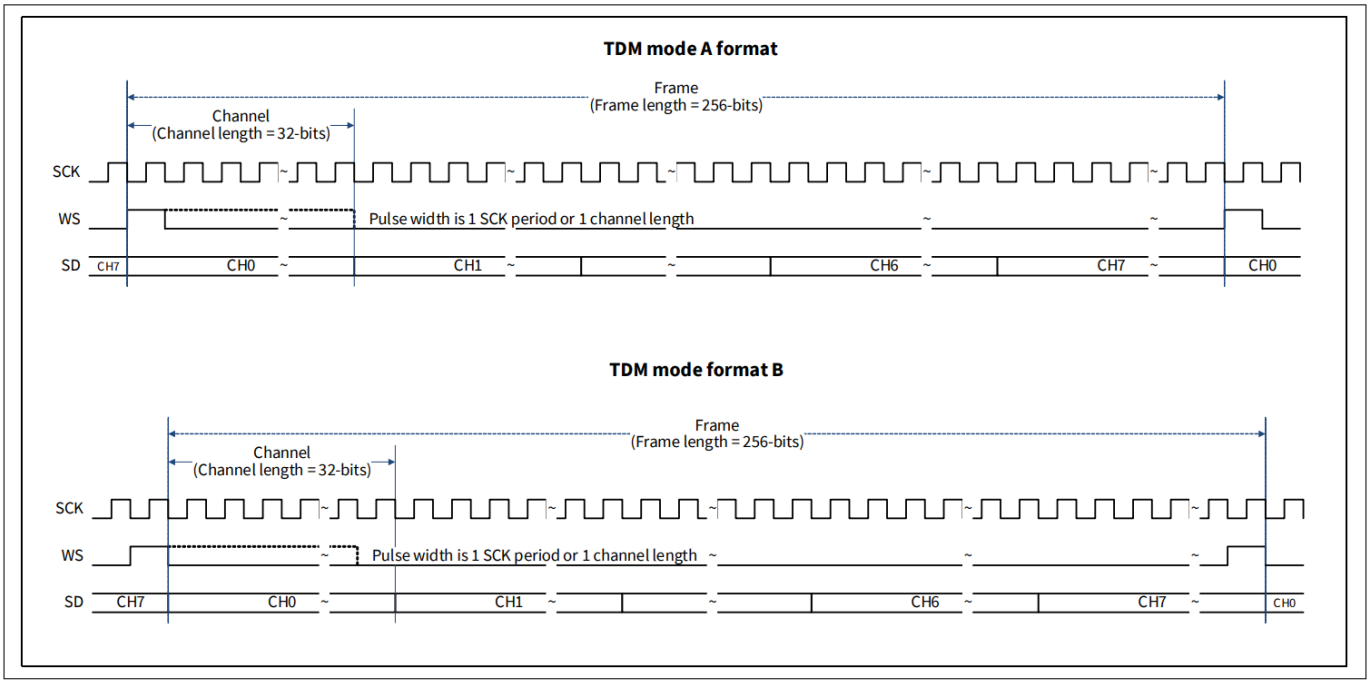


Figure 315 TDM digital audio interface format

27.3.4 Clock polarity and delay options

The I²S block supports configurable clock-polarity and delay options to address timing issues arising from PCB signal-propagation delays and delays associated with internal device signal routing.

When the I²S Tx block operates in slave mode:

- The TX_SCK and TX_WS signals are input signals to the MCU
- The TX_SDO output signal is transmitted on the falling edge of TX_SCK
- The external master device Rx block samples the TX_SDO signal on the subsequent rising edge of TX_SCK

s

Timing issues may occur when the TX_SDO signal does not meet the setup and hold time requirements for input data on the master device's Rx block. To mitigate such issues, the I²S Tx block in the CYW20829 MCU provides

an option to advance the serial-data transmission by 0.5 SCK cycles. This feature is enabled by setting the B_CLOCK_INV bit in the I2Sx_TX_CTL register. It can be used to resolve timing issues when the I²S Tx block operates in slave mode.

Table 195 Tx block configuration for master mode

Tx block configuration	Clock polarity register	Description
	I2Sx_TX_CTL.SCKO_POL	
(1)	0	Serial data is transmitted off the SCK falling edge
(2)	1	Serial data is transmitted off the SCK rising edge

Table 196 Tx block configuration for slave mode

Tx block configuration	Clock polarity register	Delay option register	Description
	I2Sx_TX_CTL.SCKI_POL	I2Sx_TX_CTL.B_CLOCK_INV	
(1)	0	0	Serial data is transmitted off the SCK falling edge
(2)	0	1	Serial data is transmitted off the SCK rising edge that is 0.5 SCK cycles before (1)
(3)	1	0	Serial data is transmitted off the SCK rising edge
(4)	1	1	Serial data is transmitted off the SCK falling edge that is 0.5 SCK cycles bore (3)

When the I²S Rx block operates in master mode:

- The RX_SCK and RX_WS signals are output signals from the CYW20829 MCU
- The RX_SDI signal is transmitted by the external device on the falling edge of RX_SCK
- The I²S Rx block samples the RX_SDI signal on the subsequent rising edge of RX_SCK

Timing issues may occur if the RX_SDI signal reaching the MCU block does not meet the setup and hold time requirements for input data. To address this issue, the I²S Rx block provides an option to delay serial-data capture by 0.5 SCK cycles. This feature is enabled by setting the B_CLOCK_INV bit in the I2Sx_RX_CTL register and can be used when timing issues arise while operating the I²S Rx block in master mode.

In addition to these clock-delay options, the I²S block supports bit-clock (SCK) inversion options:

- In master mode, the outgoing bit clock (SCK) can be inverted by setting the SCKO_POL bit in the I2Sx_TX_CTL and I2Sx_RX_CTL registers
- In slave mode, the incoming bit clock (SCK) can be inverted by setting the SCKI_POL bit in the I2Sx_TX_CTL and I2Sx_RX_CTL registers

Table 197 Rx block configuration for master mode

Rx block configuration	Clock polarity register	Delay option register	Description
	I2Sx_RX_CTL.SCKO_POL	I2Sx_RX_CTL.B_CLOCK_INV	
(1)	0	0	Serial data is captured by the SCK rising edge
(2)	0	1	Serial data is captured by the SCK falling edge that is 0.5 SCK cycles after (1)
(3)	1	0	Serial data is captured by the SCK falling edge
(4)	1	1	Serial data is captured by the SCK rising edge that is 0.5 SCK cycles after (3)

Table 198 Rx block configuration for slave mode

Rx block configuration	Clock polarity register	Description
	I2Sx_RX_CTL_SCKI_POL	
(1)	0	Serial data is captured by the SCK rising edge
(2)	1	Serial data is captured by the SCK falling edge

27.4 Interfacing with audio codecs

The I²S block in the CYW20829 MCU interfaces with an audio codec device based on the selected codec device and the end-application requirements. Some scenarios and their respective connection diagrams are as follows:

- **Codecs with separate WS and SCK signals for the Rx and Tx directions**:
  - To interface with these codecs, the connections between the I²S block and the codec device are as shown in Figure 316. The I²S Tx signals (TX_SCK, TX_WS, TX_SDO) are connected to the codec Rx signals, while the I²S Rx signals (RX_SCK, RX_WS, RX_SDI) are connected to the codec Tx signals
  - The direction of the SCK signals (TX_SCK, RX_SCK) and WS signals (TX_WS, RX_WS) depends on which device is configured as the master and which is configured as the slave
- **Codecs with shared WS and SCK signals for both Rx and Tx directions**:
  - There are two possible configurations for interfacing these codecs with the MCU, as shown in Figure 316. In both configurations, the SCK signals (TX_SCK, RX_SCK, codec_sck) are externally shorted. Similarly, the WS signals (TX_WS, RX_WS, codec_ws) are also externally shorted
  - Ensure that only one block drives the SCK and WS lines
    - When the codec acts as the slave device:
      - The I²S Rx block should be in master mode, and the I²S Tx block should be in slave mode (or vice versa, with the I²S Rx block as the slave and the I²S Tx block as the master)
    - When the codec acts as the master device:
      - Both the I²S Rx and Tx blocks should be configured in slave mode

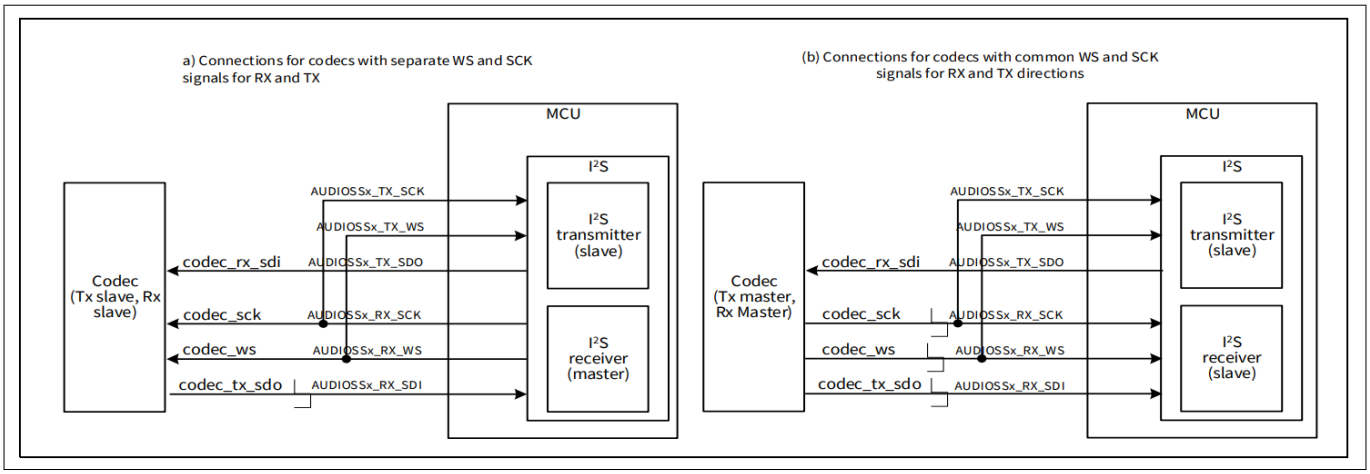


Figure 316 Interfacing with codecs having common WS and SCK signals

27.5 Clocking features

The I²S unit has three clock inputs:

Table 199 Clock inputs

Signal	Description
CLK_SYS_I2S	**System clock**: This clock is used for the AHB slave interface, control, status, and interrupt registers. It also clocks the DMA-trigger control logic.
CLK_AUDIO_I2S	**I²S internal clock**: This clock is used for the I²S transmitter (Tx) and receiver (Rx) blocks. It operates asynchronously with the CLK_SYS_I2S. This clock is connected to the CLK_HF[1] high-frequency clock in the device. For more details on high-frequency clocks, refer to the Clocking system chapter.
CLK_I2S_IF	**I²S external clock**: This clock is provided by an external I²S bus host through a port pin. It is used in place of the CLK_AUDIO_I2S clock to synchronize I²S data with the clock used by the external I²S bus host.

Figure 317 shows the clock-divider structure in the I²S block. In master mode, the SCK and WS signals are generated using either the CLK_AUDIO_I2S internal clock or the CLK_I2S_IF external clock. See the device datasheet for the port-pin assignment of the CLK_I2S_IF clock. The CLOCK_SEL bit in the I2S_CLOCK_CTL register controls the selection between the internal and external clocks.

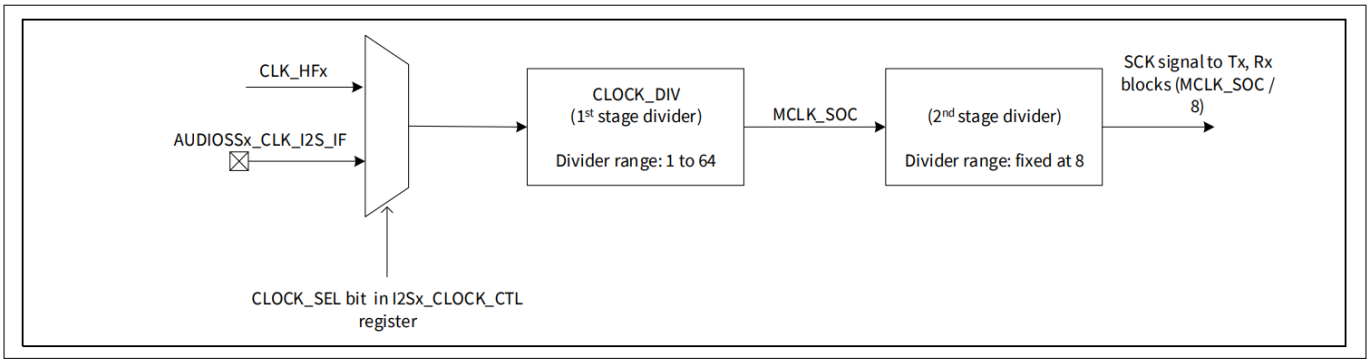


Figure 317 Clocking divider structure

There are two stages of clock dividers in the I²S block, as described below:

- The first-stage clock divider generates the internal I²S master clock (MCLK_SOC). The input clock for the first stage is either CLK_AUDIO_I2S or CLK_I2S_IF. This divider is configured using the CLOCK_DIV bits in the I2S_CLOCK_CTL register. Supported divider values range from 1 to 64
- The second-stage clock divider generates the SCK signals. Its input clock is the output of the first-stage clock divider. This divider value is fixed at 8 (FTX_SCK = FRX_SCK = FMCLK_SOC/8). The word-select (WS) signal frequency depends on the SCK frequency and the configured channel-length value

When operating in slave mode, the internal clock (MCLK_SOC) frequency must always be eight times the frequency of the input serial clock. To ensure this condition is met in slave mode, select the appropriate clock source and adjust the CLOCK_DIV divider value accordingly. Typically, when the I²S block operates in slave mode, the host sends a master clock that is an integral multiple of the sampling rate. This master clock can be routed to the CLK_I2S_IF port pin. The CLOCK_DIV divider value must then be adjusted to maintain MCLK_SOC at eight times the input SCK frequency.

Table 200 provides examples of clock divider settings for standard sampling rates using the standard I²S format. In Table 200, the first-stage divider values correspond to the register field values, the actual divider values are one more than the configured register values, as explained in the clock divider section. See the device datasheet for details on the maximum supported SCK frequency and output sampling rates.

Table 200 I²S divider values for standard audio sampling rates in standard I²S format

Sampling rate (SR) (kHz)	WORD_LEN (bits)	SCK (2*WORD_LEN*SR) (MHz)	CLK_HFx (MHz)	CLK_HFx/SCK (total divider ratio)	CLK_CLOCK_DIV (first divider)	Second stage divider (fixed at 8)
8	32	0.512	49.152	96	11	8
16	32	1.024	49.152	48	5	8
32	32	2.048	49.152	24	2	8
48	32	3.072	49.152	16	1	8
44.1	32	1.4112	45.1584	32	3	8

27.6 FIFO buffer and DMA support

The I²S block has two FIFO buffers: one each for the Tx block and the Rx block, respectively. The channel data ordering format in both the Tx and Rx FIFOs depends on the configured digital audio format. This ordering format must be considered when writing to the Tx FIFO or reading from the Rx FIFO.

- In the standard I²S and LJ digital audio formats, the data ordering is (L, R, L, R, L, ...), where L refers to left-channel data and R refers to right-channel data

- In the TDM format, with the number of active channels set to four, the data ordering is `(CH0, CH1, CH2, CH3, CH0, CH1, CH2, CH3, CH0, ...)`
- If the number of active channels is set to eight, the cycle repeats after CH0–CH7 data

Table 201 I²S FIFO buffers

Feature	Tx FIFO	Rx FIFO
Architecture	256 depth FIFOs for up to 32-bit data elements	
Data register	`I2S_TX_FIFO_WR`	`I2S_RX_FIFO_RD`
Data format	Right-aligned	Right-aligned. Receive data is extended by zeros or the sign-bit.
Trigger Control register for DMA	`I2S_TR_CTL.TX_REQ_EN`	`I2S_TR_CTL.RX_REQ_EN`
Trigger level	When the Tx FIFO has less entries than `I2S_TX_CTL.TRIGGER_LEVEL`, a transmitter trigger event is generated.	When the Rx FIFO has more entries than `I2S_RX_CTL.TRIGGER_LEVEL`, a receiver trigger event is generated.

27.6.1 I²S Tx FIFO

The I²S Tx block has a hardware FIFO with a depth of 256 elements, where each element is 32 bits wide. In addition to this 256-element FIFO, the I²S block includes an internal transmit buffer that can store four 32-bit data elements to be transmitted. This four-element buffer serves as an intermediary to hold data for transfer on the I²S bus. It is not exposed to the AHB bus interface.

- The `I2Sx_TX_FIFO_CTL` register is used for FIFO control operations. The `TRIGGER_LEVEL` bits in the `I2Sx_TX_FIFO_CTL` register can generate a transmit trigger event when the Tx FIFO has fewer entries than the value configured in the `TRIGGER_LEVEL` bits
- FIFO freeze can be enabled by setting the `FREEZE` bit in the `I2Sx_TX_FIFO_CTL` register. When this bit is set and the Tx block is operational (with the `TX_START` bit in `I2S_CMD` set), hardware reads from the Tx FIFO do not remove the FIFO entries, and the Tx FIFO read pointer does not advance. Writes to the `I2S_TX_FIFO` register increment the Tx FIFO write pointer. If the Tx FIFO becomes full, the internal write pointer stops incrementing. Freeze functionality is intended for firmware debugging, not normal operation. To resume normal operation after using the freeze feature, reset the I²S by clearing the `TX_ENABLED` bit in the `I2S_CTL` register and then setting it again
- The `CLEAR` bit in the `I2Sx_TX_FIFO_CTL` register is used to clear the Tx FIFO by resetting the FIFO's read and write pointers. Write access to the Tx FIFO through the `I2Sx_TX_FIFO_WR` register is disabled when the `CLEAR` bit is set
- The `I2Sx_TX_FIFO_STATUS` register provides FIFO status information, such as the number of used entries in the Tx FIFO and the current values of the Tx FIFO read and write pointers. This register can be used for debugging purposes. The Tx FIFO read pointer is updated when data is transferred from the Tx FIFO to the internal transmit buffer. The Tx FIFO write pointer is updated when data is written to the `I2Sx_TX_FIFO_WR` register via the CPU or the DMA controller

For Tx FIFO data writes using the CPU, the hardware can trigger an interrupt for various FIFO conditions, such as `TX_TRIGGER`, `TX_NOT_FULL`, and `TX_EMPTY`. As part of the interrupt handler, the CPU can write to the `I2S_TX_FIFO_WR` register. The recommended method is to write `(256 – TRIGGER_LEVEL)` words to the `I2Sx_TX_FIFO_WR` register whenever the `TX_TRIGGER` interrupt event is triggered. Additionally, interrupt events can be generated for FIFO overflow or underflow conditions.

For DMA-based Tx data transfers, the I²S Tx DMA trigger signal (`tr_i2s_tx_req`) can be enabled by setting the `TX_REQ_EN` bit in the `I2S_TR_CTL` register. The trigger signal becomes active whenever the Tx FIFO has fewer entries than the value configured in the `TRIGGER_LEVEL` field. The DMA channel can be configured to transfer up

to (256 – TRIGGER_LEVEL) words from the source address (such as flash or SRAM regions). The destination address for the DMA must always be the I2Sx_TX_FIFO_WR register address, with the destination address increment feature disabled in the DMA channel configuration. The FIFO address increment is handled internally to adjust the write pointer, so the DMA must not increment the destination address. For more details on DMA channel configuration, see Direct memory access (DMA DW).

Data in the I2Sx_TX_FIFO_WR register is always right-aligned. The format for different word-length configurations is provided in Figure 318.

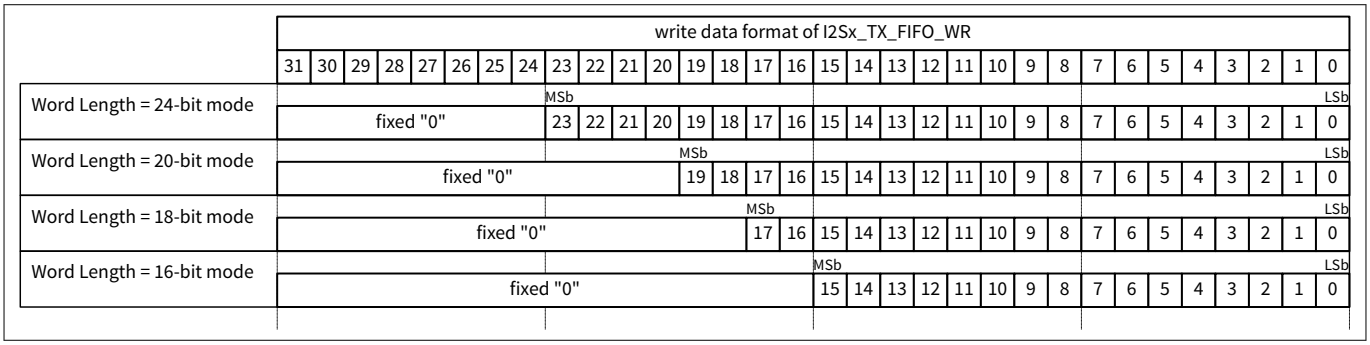


Figure 318 I2S_RX_FIFO_RD register format for different word lengths

27.6.2 I²S Rx FIFO

The I²S Rx block has a hardware FIFO with a depth of 256 elements, where each element is 32 bits wide. In addition to this 256-element FIFO, the I²S block has an internal receive buffer capable of storing four 32-bit data elements. This four-element buffer serves as an intermediary to hold data received from the I²S bus and is not exposed to the AHB bus interface.

- The I2Sx_RX_FIFO_CTL register is used for FIFO control operations. The TRIGGER_LEVEL bits in the I2Sx_RX_FIFO_CTL register are used to generate a receive trigger event when the Rx FIFO has more entries than the value configured in the TRIGGER_LEVEL bits. In the standard I²S and LJ formats, the TRIGGER_LEVEL bits can be configured up to a maximum value of 253. In the TDM format, the maximum value for TRIGGER_LEVEL is (254 - CH_NR), where CH_NR is the number of active channels in the TDM frame
- The FIFO freeze operation can be enabled by setting the FREEZE bit in the I2Sx_RX_FIFO_CTL register. When this bit is set and the Rx block is operational (with the RX_START bit in the I2Sx_CMD register set), hardware will not write to the Rx FIFO, and the Rx FIFO write pointer will not advance. Reads from the I2Sx_RX_FIFO register increment the Rx FIFO read pointer. When the Rx FIFO becomes empty, the internal read pointer stops incrementing. Freeze functionality is intended for firmware debugging and not for normal operation. To restore normal operation after using the freeze operation, reset the I²S by clearing the RX_ENABLED bit in the I2Sx_CTL register and setting it again
- The CLEAR bit in the I2Sx_RX_FIFO_CTL register clears the Rx FIFO by resetting the FIFO's read and write pointers. Read access to the Rx FIFO using the I2Sx_RX_FIFO_RD or I2Sx_RX_FIFO_RD_SILENT registers is not allowed while the CLEAR bit is set
- The I2Sx_RX_FIFO_STATUS register provides FIFO status information, including the number of used entries in the Rx FIFO and the current values of the Rx FIFO read and write pointers. This register is useful for debugging purposes. The Rx FIFO write pointer is updated whenever data is transferred into the Rx FIFO from the internal receive buffer. The Rx FIFO read pointer is updated whenever data is read from the I2Sx_RX_FIFO_RD register by the CPU or DMA controller. The I2Sx_RX_FIFO_RD_SILENT register is available for debugging and always returns the top element of the Rx FIFO without advancing the read pointer

For Rx FIFO data reads using the CPU, the hardware can trigger an interrupt for various FIFO conditions such as RX_TRIGGER, RX_NOT_EMPTY, and RX_FULL. As part of the interrupt handler, the CPU can read from the I2Sx_RX_FIFO_RD register. The recommended approach is to read (TRIGGER_LEVEL + 1) words from the

I2Sx_RX_FIFO_RD register each time the RX_TRIGGER interrupt event is triggered. Interrupt events can also be generated for FIFO overflow or underflow conditions.

For DMA-based Rx data transfers, the I²S Rx DMA trigger signal (tr_i2s_rx_req) can be enabled by setting the RX_REQ_EN bit in the I2Sx_TR_CTL register. The trigger signal becomes active whenever the Rx FIFO has more entries than the value configured in the TRIGGER_LEVEL field. The DMA channel can be configured to transfer up to (TRIGGER_LEVEL + 1) words to the destination address, such as an SRAM region. The source address for the DMA must always be the I2Sx_RX_FIFO_RD register address, with the source address increment feature disabled in the DMA channel configuration. FIFO address incrementing is handled internally to adjust the read pointer, so the DMA must not increment the source address. For details on DMA channel configuration, see Direct memory access (DMA DW).

Data in the I2Sx_RX_FIFO_RD register is always right-aligned. The format of the I2Sx_RX_FIFO_RD register for different word-length configurations is shown in Figure 319. Unused most-significant bits are either set to 0 or sign-bit extended, depending on the BIT_EXTENSION bit in the I2Sx_RX_CTL register.

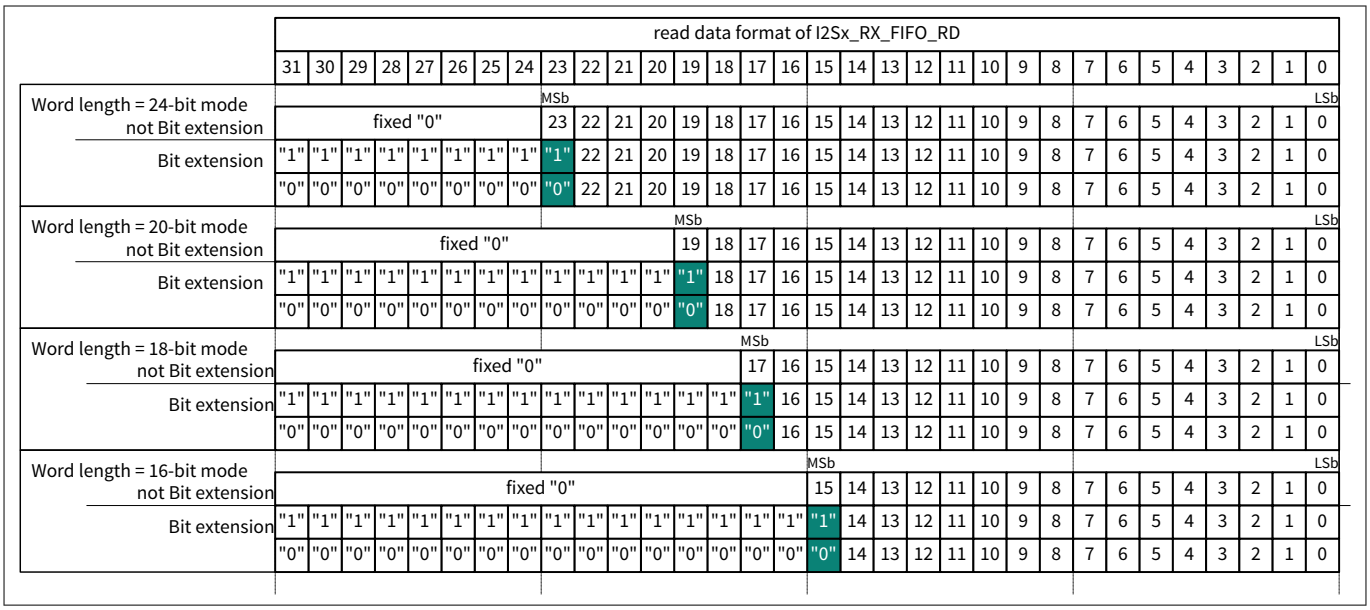


Figure 319 I2Sx_RX_FIFO_RD register format for different word lengths

27.7 Interrupt support

The I²S block has one interrupt output signal that connects to the interrupt controller in the CPU. For details on the vector number of the I²S interrupt and the procedure to configure the interrupt priority, vector address, and enabling or disabling the interrupt, see Interrupts.

The I²S interrupt can be triggered by any of the following events:

- TX_TRIGGER
- TX_NOT_FULL
- TX_EMPTY
- TX_OVERFLOW
- TX_UNDERFLOW
- TX_WD
- RX_TRIGGER
- RX_NOT_EMPTY
- RX_FULL
- RX_OVERFLOW

- RX_UNDERFLOW
- RX_WD

Each interrupt event can be individually enabled or disabled to generate an interrupt condition. The I2Sx_INTR_MASK register is used to enable the required events by writing 1 to the corresponding bit. Regardless of the INTR_MASK settings, if any of the events occur, the hardware sets the corresponding event status bit in the I2Sx_INTR register.

The I2Sx_INTR_MASKED register provides a bitwise AND operation between the I2Sx_INTR_MASK and I2Sx_INTR registers. The final I²S interrupt signal is generated as the logical OR of all bits in the I2Sx_INTR_MASKED register. Therefore, only the events enabled in the I2Sx_INTR_MASK register are propagated as interrupt events to the interrupt controller.

Interrupts can also be triggered in software by writing to the corresponding bits in the I2Sx_INTR_SET register. Figure 320 shows how the interrupt signal is generated.

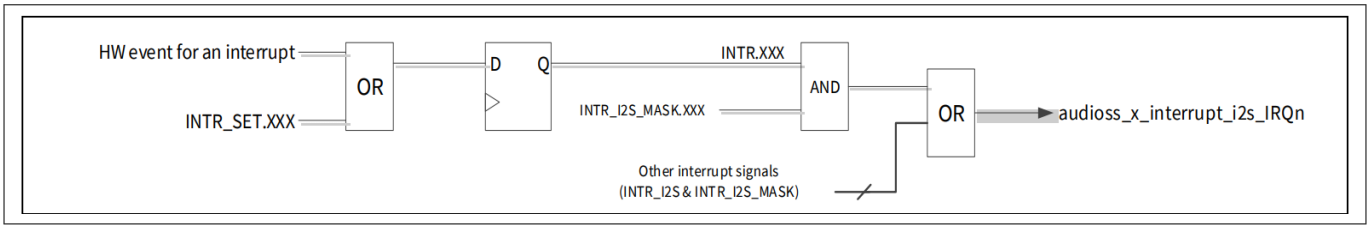


Figure 320 Interrupt signal generation

In the interrupt service routine (ISR), the I2Sx_INTR_MASKED register should be read to identify the events that triggered the interrupt. Multiple events can trigger the interrupt because the final interrupt signal is the logical OR output of all events.

- The ISR should handle the tasks corresponding to each interrupt event that was triggered
- At the end of the ISR, the value read earlier from the I2Sx_INTR_MASKED register should be written to the I2Sx_INTR register. This clears the bits corresponding to the interrupt events that have been processed in the ISR
- If the bits are not cleared by writing 1 to the I2Sx_INTR register, the interrupt signal will remain high
- A dummy read of the I2Sx_INTR register should be performed after the write operation to ensure the register update takes effect

27.8 Watchdog timer

The Tx and Rx blocks have independent watchdog timers, which can generate an interrupt event if the word select (WS) input remains idle for longer than the configured time period. This feature is available only in slave mode, where an external master drives the WS input lines (TX_WS, RX_WS). The feature can detect signal transmission issues, problems with the master device, or if the master has halted communication.

- If the master drives the same Word Select signal to both the TX_WS and RX_WS lines, only one of the watchdog timers can be enabled to generate an interrupt event
- Although the following explanation describes the Tx watchdog, the same information applies to the Rx watchdog

Configuring the Tx watchdog timer:

1. To enable the Tx watchdog timer, set the WD_EN bit in the I2Sx_TX_CTL register
2. Configure the watchdog timer reload value (32-bit timer) by writing to the I2Sx_TX_WATCHDOG register. Writing a value of 0 to this register will also disable the watchdog timer
3. The timer runs on the CLK_PERI system clock. For details on CLK_PERI generation, see Clocking system

4. The watchdog timer starts running when both the WD_EN and TX_START bits are set
5. The timer reloads in the following conditions:
   - A rising-edge event occurs on the TX_WS input signal
   - The timer value reaches 0

When the timer value reaches 0, the TX_WD interrupt event is generated. To enable interrupt generation by the watchdog timer, set the TX_WD bit in the I2Sx_INTR_MASK register. The TX_WD interrupt event can be cleared by writing 1 to the TX_WD bit in the I2Sx_INTR register. Figure 321 the watchdog behavior when the timer is enabled.

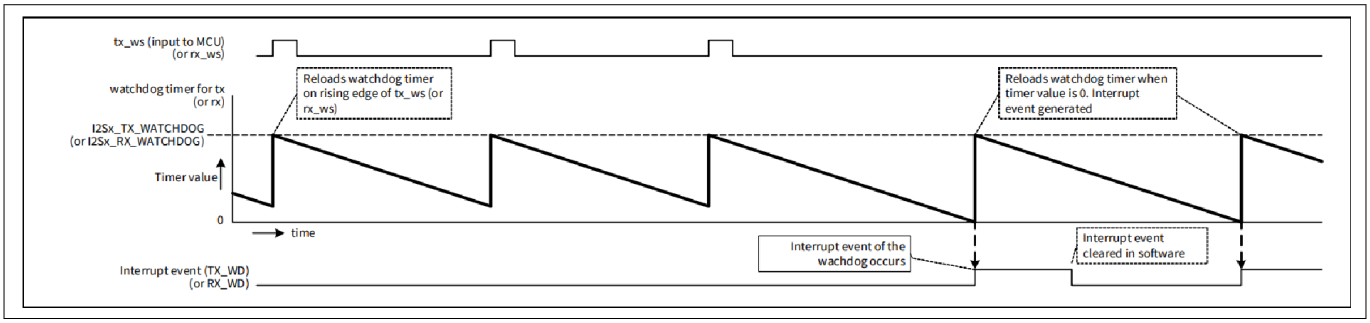


Figure 321 Watchdog timer working

27.9 MCLK output function

The I²S unit generates the MCLK output signal for an external audio codec. The MCLK output signal is generated only when all the following conditions are met:

- I2Sx_CTL.TX_ENABLE or RX_ENABLE = 1 (I²S is enabled)
- I2Sx_CLOCK_CTL.CLOCK_SEL = 0 (I²S clock is sourced from the internal clock: CLK_HFx)
- I2Sx_CLOCK_CTL.MCLK_DIV = 0, 1, 2, or 3 (Division ratio: 1, 2, 4, or 8)
- I2Sx_CLOCK_CTL.MCLK_EN = 1 (MCLK output is enabled)

Figure 322 shows the use of the MCLK output signal for the external audio codec.

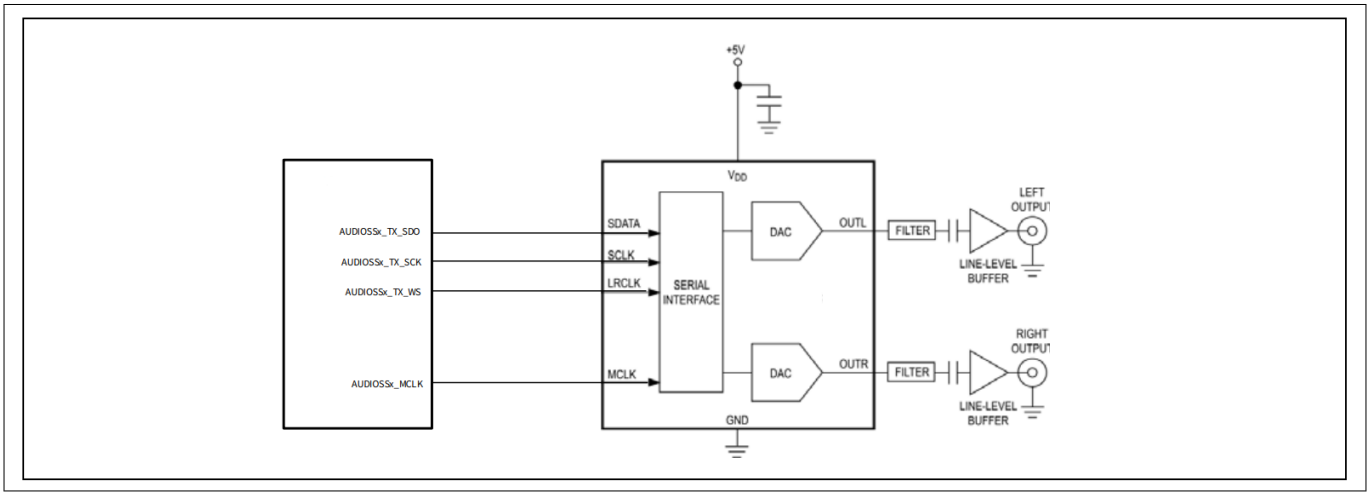


Figure 322 MCLK output signal for the external audio codec

27.10 Registers summary

Table 202 Registers summary

Register	Name	Description
I2Sx_CTL	I²S Control register	This register is used to control the I²S block.
I2Sx_CLOCK_CTL	I²S Clock Control register	This register can configure frequencies for I²S block and master clock. It should not be changed during communication.
I2Sx_CLOCK_STAT	I²S Clock Status register	This register indicates the status of the MCLK divider and is used to allow a controlled shutdown of the clock.
I2Sx_CMD	I²S Command register	This register can control I²S functions.
I2Sx_TR_CTL	I²S Trigger Control register	This register can enable DMA for I²S. It should not be changed during communication.
I2Sx_TX_CTL	I²S Transmitter Control register	This register can configure the transmitter for I²S functions. It should not be changed during communication.
I2Sx_TX_WATCHDOG	I²S Transmitter Watchdog register	This register can configure counter value for the Tx watchdog. It should not be changed during the operation.
I2Sx_RX_CTL	I²S Receiver Control register	This register can configure the receiver for I²S functions. It should not be changed during communication.
I2Sx_RX_WATCHDOG	I²S Receiver Watchdog register	This register can configure counter value for the Rx watchdog. It should not be changed during the operation.
I2Sx_TX_FIFO_CTL	I²S Tx FIFO Control register	This register can configure Tx FIFO for I²S.
I2Sx_TX_FIFO_STATUS	I²S Tx FIFO Status register	This register displays the status of Tx FIFO for I²S.
I2Sx_TX_FIFO_WR	I²S Tx FIFO Write register	This register stores transmission data for I²S transfer.
I2Sx_RX_FIFO_CTL	I²S Rx FIFO Control register	This register can configure Rx FIFO for I²S.

(table continues...)

Table 202 (continued) Registers summary

Register	Name	Description
I2Sx_RX_FIFO_STATUS	I²S Rx FIFO Status register	This register displays the status of Rx FIFO for I²S.
I2Sx_RX_FIFO_RD	I²S Rx FIFO Read register	This register stores received data for I²S transfer.
I2Sx_RX_FIFO_RD_SILENT	I²S Rx FIFO Silent Read register	This register stores received data for I²S transfer.
I2Sx_INTR	I²S Interrupt register	This register displays the interrupt status.
I2Sx_INTR_SET	I²S Interrupt Set register	This register can be used to trigger an interrupt for firmware testing.
I2Sx_INTR_MASK	I²S Interrupt Mask register	This register controls whether the I²S interrupt is forwarded to the corresponding processor.
I2Sx_INTR_MASKED	I²S Interrupt Masked register	Bitwise AND between the Interrupt Request and Mask registers, so firmware can read the status of all mask enabled interrupt causes with a single load operation.

28 Analog-to-digital converter/microphone

28.1 Features

- Single switched-capacitor sigma-delta ADC core for both audio and DC measurement
- 12 MHz clocking rate
- MIC PGA gain range: 0–42 dB with gain steps smaller than 1 dB
- Programmable decimation filter chain (CIC decimation)
- Fixed finite impulse response (FIR) chains and infinite impulse response (IIR) chain for equalization and audio sample processing
- Programmable interface to analog ADC
- Generates interrupts when the FIFO threshold is reached
- PDM clock support for receiving PDM data

28.2 System diagram

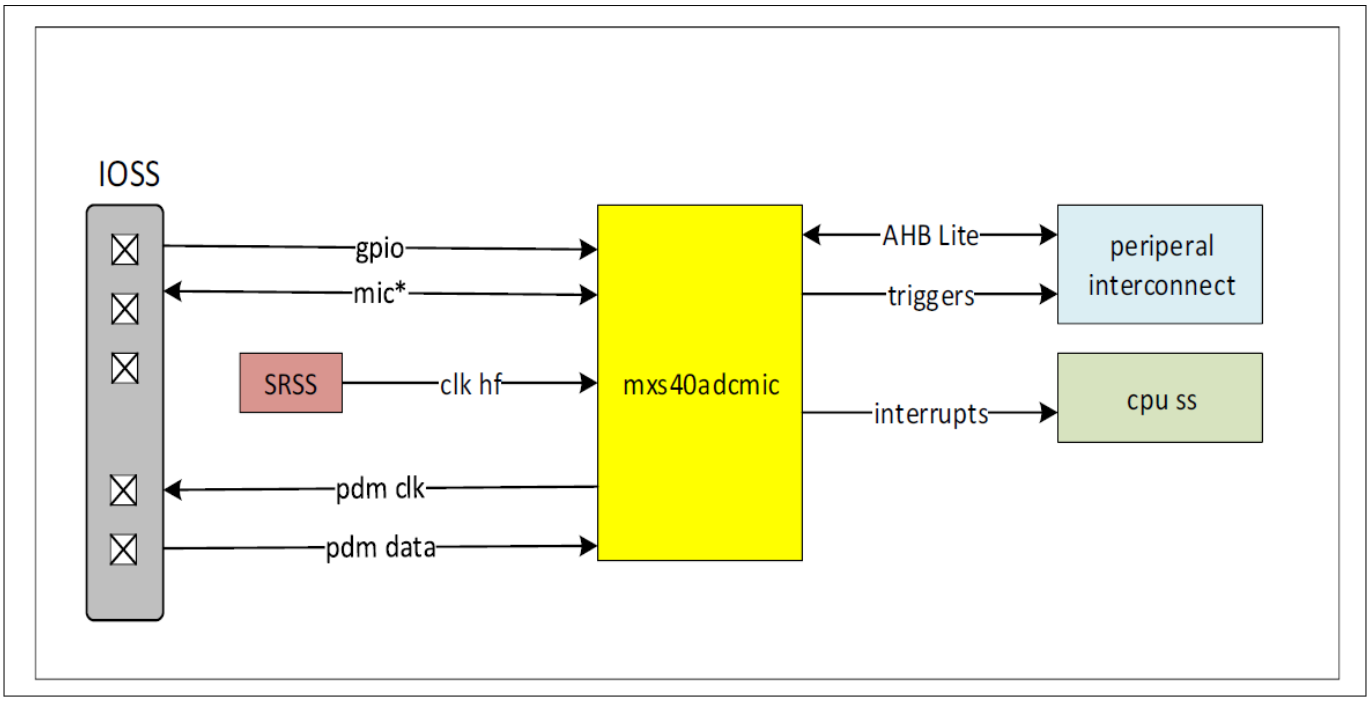


Figure 323 System diagram

The mxs40adcmic block interfaces with the analog ADC block and outputs PDM data from analog microphones.

28.3 Block diagram

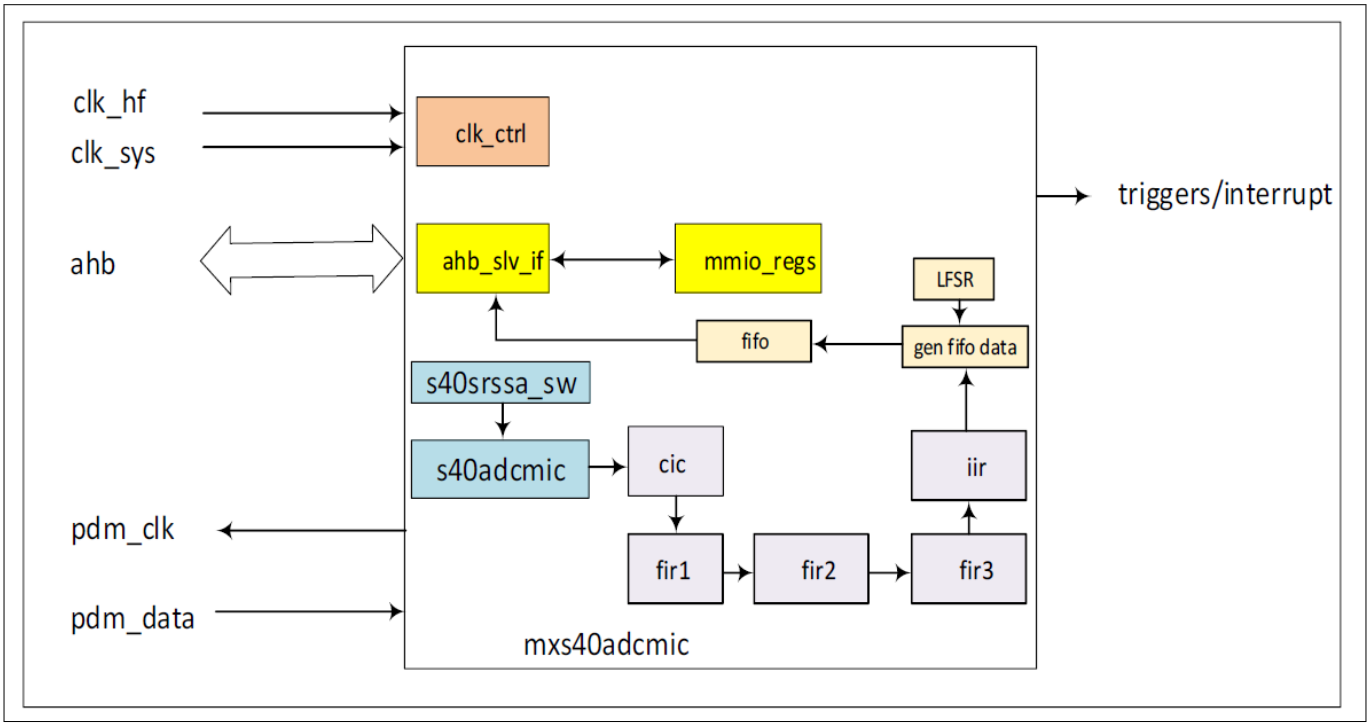


Figure 324 mxs40adcmic high-level block diagram

28.3.1 Functional description

The mxs40adcmic block serves as the primary interface to the delta-sigma modulator (DSM) analog ADC block. The PDM samples generated by the ADC are decimated using CIC and FIR filters to extract the audio component. The extracted audio data is stored in a FIFO and retrieved by the CPU subsystem when a threshold is reached. The block also supports DC measurement, where the output of the CIC is stored in the MMIO register.

28.3.1.1 Operating modes

The mxs40adcmic operates exclusively in active mode and does not support a Deep Sleep mode.

28.3.1.2 Interfaces

- AHB slave interface for MMIO register access
- Two clock inputs:
  - **Clk_hf**: 24 MHz clock that drives the ADC and internal logic
  - **Clk_sys**: PERI bus access clock for MMIO registers
- MIC I/Os: MIC_N, MIC_P, and MIC_BIAS
- 8 GPIO inputs for DC measurements
- PDM clock and data interface

28.3.1.3 Interrupts and triggers

The mxs40adcmic supports a single interrupt lane that aggregates two sources:

1. FIFO threshold interrupt
2. DC output ready interrupt

This block provides two triggers, coupled with the same interrupt sources, which can control the Datawire.

28.3.1.4 Recovery

The only way to recover from an unexpected state is to reset the block.

28.3.1.5 Initialize

The default state after a reset is active mode, used for processing PDM samples from the ADC.

28.3.1.6 Pin status

The only externally facing pins are the PDM interface and GPIO, which are held low when not in use.

29 Bluetooth® Low Energy subsystem (BLESS)

29.1 Features

The Bluetooth® LE subsystem has the following features:

- 48-MHz Arm® Cortex®-M33 CPU with a 2.4 GHz RF transceiver and 50 Ω antenna drive
- Digital PHY
- Link layer engine supporting both master and slave modes
- Programmable TX power: up to 10 dBm
- Link layer engine supports up to 16 simultaneous connections in any combination of central and peripheral devices:
  - For example, 13 central devices and 3 peripheral devices, or 3 central devices and 13 peripheral devices
- LE 2 Mbps and LE Long Range modes
- LE Advertising Extensions
- LE Isochronous Channels
- Periodic Advertising with Responses (PAwR)
- Encrypted Advertising Data
- LE GATT security levels, characteristic and advertising coding selection

29.2 Architecture

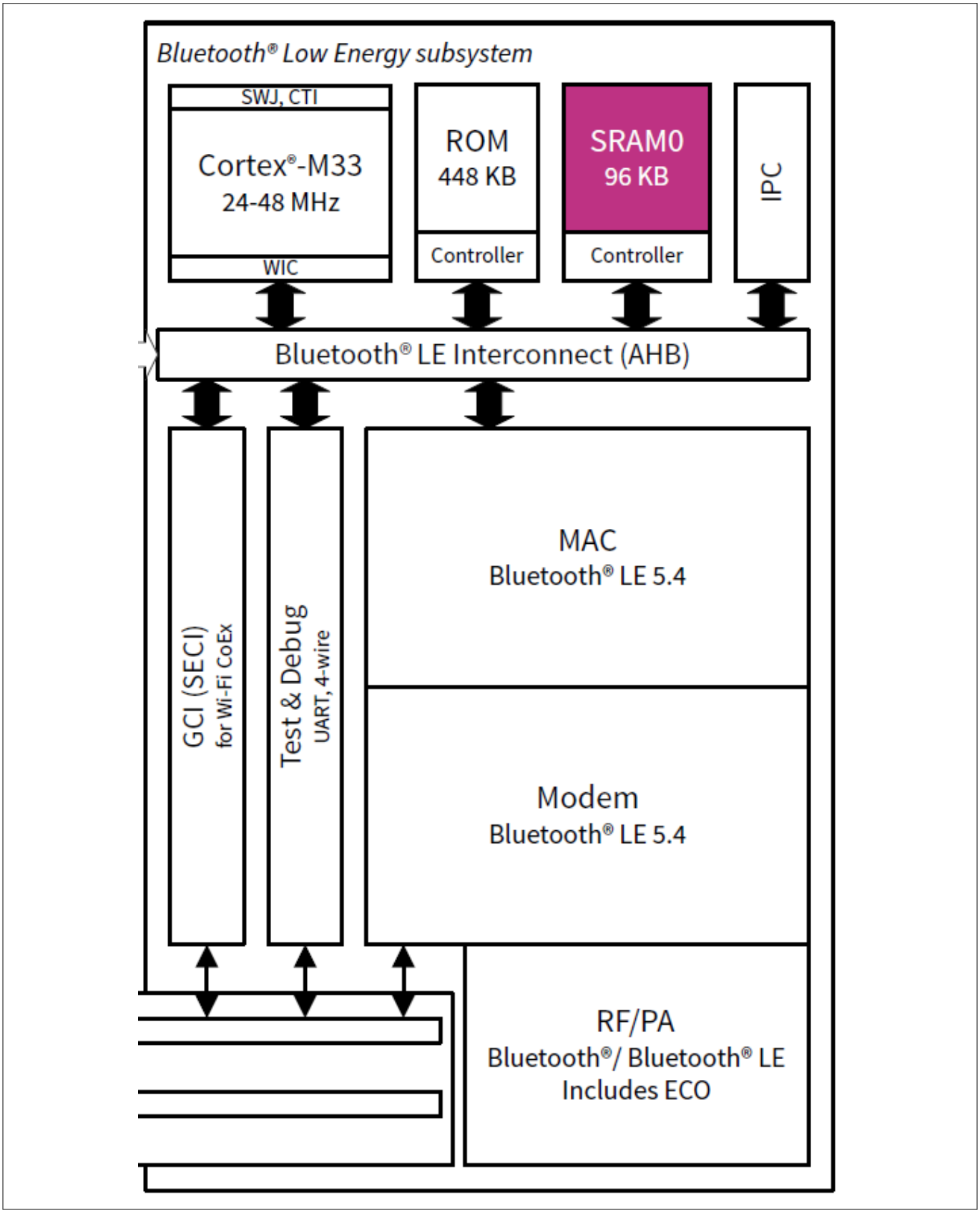


Figure 325 Bluetooth® LE subsystem block diagram

The Bluetooth® LE subsystem includes the following major blocks:

- **Link layer (LL) controller**: The link layer controller implements the sleep state machines for various Bluetooth® LE procedures (advertiser, scanner, initiator, and connection) and roles (master and slave). It also handles timing-critical functions such as packet framing/deframing, CRC generation/checking, encryption/decryption, packet transmission, and protocol time reference management. Additionally, it supports direct test mode and includes Host Control Interface (HCI) functionality
- **System control**: This block consists of the AHB bus interface logic, the interrupt control logic, and the subsystem controller. The subsystem controller manages the Bluetooth® LE subsystem clocks, reset, and power modes
- **Radio controller**: This block contains the mode transition finite state machine (FSM), which controls the low-dropout regulator (LDO) and power supply switches for the radio and PHY

29.2.1 Link layer controller

This section outlines the high-level functional modes and configuration options of the link layer.

29.2.1.1 Bluetooth® LE functional modes and configuration

This section describes the various modes of Bluetooth® LE functionality. The operation of the link layer is defined using a state machine with the following primary modes:

- Standby
- Advertising
- Scanning (active or passive)
- Initiating
- Connection (master or slave)
- Direct Test Mode (DTM)
- Encryption/Decryption
- **Standby state**: The standby state is the default functional state. In this state, the link layer neither transmits nor receives any packets. Transitions from this state to the advertising, scanning, or other states are possible based on the functional requirements. Additionally, the standby state can be entered from any other state
- **Advertising state**: In the advertising state, the link layer transmits advertising packets and can optionally respond to responses from peer devices. A device in this state, referred to as an advertiser, transitions to it from the standby state. Devices enter this state to become discoverable, connectable, or to broadcast data to other nearby devices
- **Scanning state**: In the scanning state, the link layer enables the device to listen for advertising packets from peer devices. This state supports two types of scanning:
  - **Active scanning**: Active scanning allows the device to send scan requests to advertising devices. This helps gather additional information about peer devices and can potentially lead to establishing a connection
  - **Passive scanning**: Passive scanning is used to simply listen for the presence of other devices without sending any responses

  A device in the scanning state, referred to as a scanner, transitions to this state from the standby state
- **Initiating state**: In the initiating state, the link layer listens for advertising packets from a specific peer device and responds to initiate a connection. A device in this state, referred to as an initiator, transitions to this state from the standby state
- **Connection state**: The connection state is the final procedural state in the link layer protocol. A device enters this state from either the advertising or initiating state and is considered to be "in connection." This

state supports two roles: master and slave. If the connection state is entered from the advertising state, the device assumes the role of a slave. If it is entered from the initiating state, the device assumes the role of a master

- **Direct-test mode (DTM)**: Direct-test mode enables testing of the physical layer (radio digital/RF). It allows a tester to command the controller's physical layer to transmit or receive a sequence of predefined test packets. The tester can analyze the received packets to verify whether the physical layer operates according to specifications. Additionally, the tester can measure various RF parameters from the received packets to assess the compliance of the physical layer with RF specifications
- **Encryption/Decryption**: This mode enables the encryption of plaintext data and decryption of encrypted data. Bluetooth® LE employs AES-128 encryption and decryption using CCM (counter with CBC-MAC). Additionally, this module supports encryption through the electronic codebook (ECB) mode

29.2.1.2 Firmware reset

The link layer can be reset through a firmware soft reset request. This operation resets all registers and state machines to their default values and states, respectively. However, the soft reset does not modify the contents of the FIFOs. Instead, the existing content becomes invalid, and the FIFOs must be reinitialized.

29.2.2 Clocks

This subsystem operates with three primary clocks:

- **Clk_althf**: A 24 MHz crystal oscillator clock generated by the radio block and consumed locally. An output clock is also provided for system resource consumption. The Clk_althf request is controlled by the system resource using the Q-channel interface
- **Clk_lf**: A 32 kHz clock sourced from either the watch crystal oscillator (WCO) or precision internal low oscillator (PILO). This clock drives the internal counters during the Deep Sleep power mode. The clock source is selected by the system software
- **Clk_pwr, IMO**: An 8 MHz clock that drives power transition sequences
- **Clk_hf**: A 48 MHz clock sourced from the internal high oscillator (IHO). This clock drives the compute subsystem

29.2.3 Power states

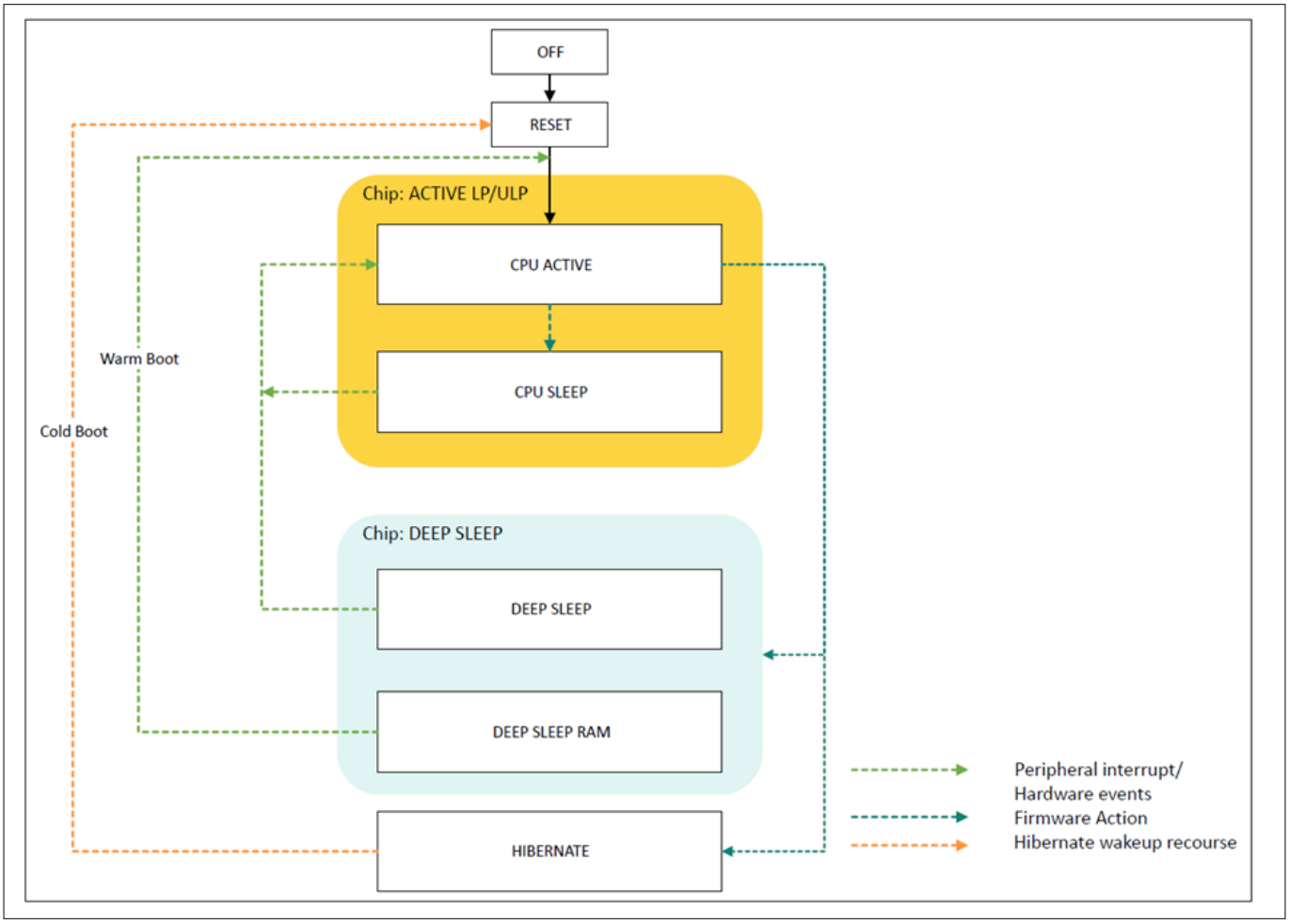


Figure 326 Bluetooth® LE subsystem power state machine

Power mode switching can be controlled at the MCU subsystem (MCUSS) level, while the Bluetooth® subsystem (BTSS) manages its own power mode state transitions internally. For the device to enter a low-power mode, such as Deep Sleep, both CPUs must be in the Deep Sleep state. Figure 327 shows the transitions between different power modes starting from the power-on reset (POR).

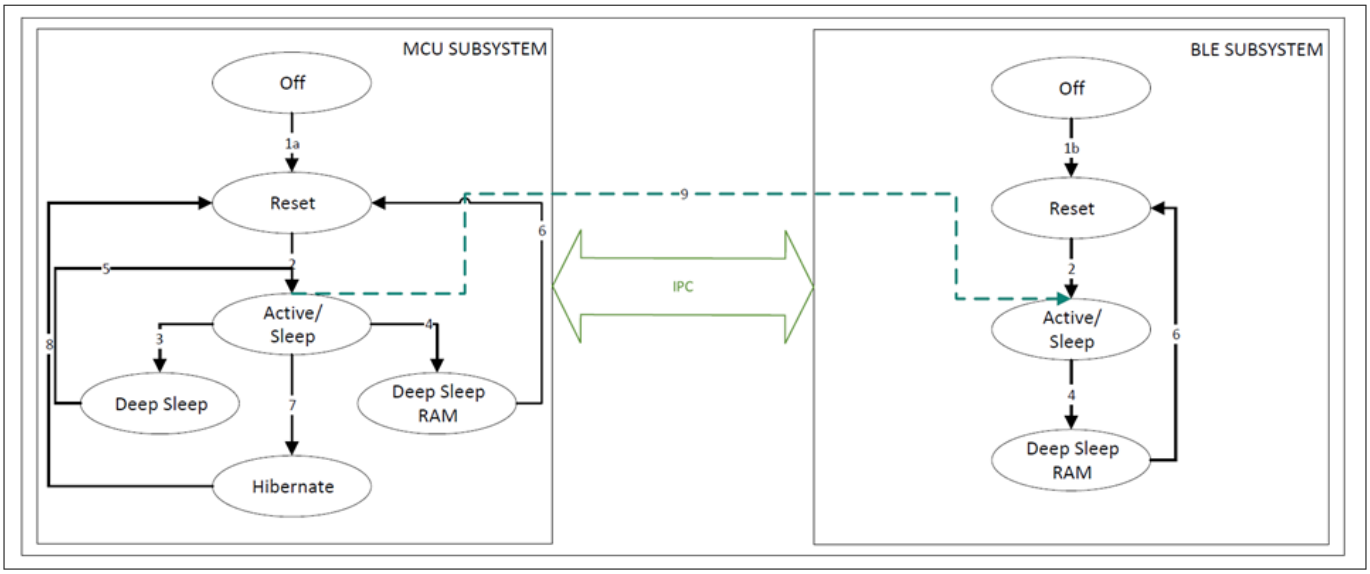


Figure 327 Power mode transitions of MCUSS and Bluetooth® subsystem

Table 203 Power mode transitions and triggers for MCUSS and Bluetooth® subsystem

Transition	Description	Transit triggers	Power mode
1a	MCUSS OFF to reset transition	Power supply applied to CYW20829	Active
1b	The Bluetooth® subsystem transitions from the OFF state to the reset state, where it remains until the MCU subsystem (MCUSS) releases it.	Power supply applied to CYW20829	Active
2	The MCU subsystem (MCUSS) and Bluetooth® subsystem transition from the reset state to the active mode independently. The Bluetooth® subsystem transitions to active mode only after being released from reset by the MCUSS. Each subsystem can move between active and sleep modes independently.	During firmware boot or startup, the system transitions from the reset state to the active mode. The MCU subsystem (MCUSS) downloads configurations to the Bluetooth® subsystem via inter-processor communication (IPC). Once configured, both subsystems transition to the Active mode.	Active

(table continues...)

Table 203 (continued) Power mode transitions and triggers for MCUSS and Bluetooth® subsystem

Transition	Description	Transit triggers	Power mode
3	The MCU subsystem (MCUSS) transitions to the Deep Sleep state, and the Bluetooth® subsystem enters the Deep Sleep RAM state. There is no operational difference for the Bluetooth® subsystem between the Deep Sleep and Deep Sleep RAM power modes.	Firmware (OS idle task) transitions from Active to Deep Sleep RAM mode.	Deep Sleep
4	Both the MCU subsystem (MCUSS) and the Bluetooth® subsystem transition to the Deep Sleep RAM state.	Firmware (OS idle task) transitions from Active to Deep Sleep RAM mode	Deep Sleep RAM
5	The MCU subsystem (MCUSS) transitions from Deep Sleep to Active mode and resumes execution from the point where it entered Deep Sleep. The Bluetooth® subsystem transitions to Active mode via the warm boot handler (Path 6).	Deep Sleep wakeup sources	Active
6	Both the MCU subsystem (MCUSS) and the Bluetooth® subsystem transition independently from the Deep Sleep RAM state to active mode and execute their respective warm boot handlers.	Deep Sleep wakeup sources	Active
7	The MCU subsystem (MCUSS) initiates the transition to Hibernate mode, which the Bluetooth® subsystem cannot ignore. The device then enters Hibernate mode.	Firmware transitions from Active to Hibernate mode	Hibernate
8	The MCUSS transitions from Hibernate to Active mode.	Hibernate wakeup sources	Active

(table continues...)

Table 203 (continued) Power mode transitions and triggers for MCUSS and Bluetooth® subsystem

Transition	Description	Transit triggers	Power mode
9	The MCUSS releases the Bluetooth® subsystem from reset.	Bluetooth® LE stack initialization	Active

***Note**: Power mode transitions for the Bluetooth® subsystem (for example, from Active to Sleep) are managed internally by the radio controller firmware. These transitions do not require any handling by the user application.*

29.2.4 Bluetooth® LE radio

CYW20829 incorporates a Bluetooth® 5.4 LE subsystem (BLESS) that includes the physical layer (PHY) and link layer (LL) engines, along with an embedded security engine. The Bluetooth® LE subsystem supports all Bluetooth® LE 5.4 features, including:

- LE 2 Mbps
- LE Long Range
- LE Advertising Extensions
- LE Isochronous Channels
- Periodic Advertising with Responses (PAwR)
- Encrypted Advertising Data
- LE GATT Security Levels Characteristic
- Advertising Coding Selection

Infineon also provides an extensive driver library and middleware support for Bluetooth® LE. For details, see the Eclipse IDE for ModusToolbox™ software.

The physical layer consists of the digital PHY and an RF transceiver capable of transmitting and receiving Gaussian frequency shift keying (GFSK) packets at 1 Mbps or 2 Mbps over the 2.4 GHz ISM band. The device also supports Bluetooth® LE long-range operation at 500 kbps and 125 kbps speeds.

The baseband controller is a composite of hardware and firmware that supports both master and slave modes. Key protocol elements, such as the host control interface (HCI) and link control, are implemented in firmware. Time-critical functional blocks, including encryption, CRC, data whitening, and access code correlation, are implemented in hardware within the LL engine.

The RF transceiver features an integrated balun, providing a single-ended RF port pin to drive a 50 Ω antenna via a matching and filtering network. In the receive direction, it converts the RF signal from the antenna into a digital bit stream after performing GFSK demodulation. In the transmit direction, it performs GFSK modulation and converts the digital baseband signal to a radio frequency before transmission through the antenna.

29.3 Infineon Host Bluetooth® Stack (BTSTACK)

The Infineon Host Bluetooth® Stack (BTSTACK) provides an API for creating Bluetooth® applications. BTSTACK implements the protocols and profiles required by the Bluetooth® SIG, enabling the creation and certification of Bluetooth® applications and components. It is designed to support multiple platforms and is built and distributed as the BTSTACK Library.

Table 204 provides details of the common API references discussed in this chapter. For additional information, see the Bluetooth® Host Stack Library .

Table 204 Bluetooth® Stack (BTSTACK) library

API reference	Data structures	Description
Logical Link Control and Adaptation Protocol (L2CAP)	wiced_bt_l2cap_le_appl_information_t	Defines the structure that applications use to register with the Bluetooth® LE Logical Link Control and Adaptation Layer Protocol (L2CAP). L2CAP provides both connection-oriented and connectionless data services to upper-layer protocols. It supports protocol multiplexing capabilities as well as segmentation and reassembly operations.
Common LE Advertisement, Scan, and Connection defines	wiced_bt_ble_advert_elem_t	Advertisement element
LE Extended Mode Advertising and Scanning	wiced_ble_ext_adv_duration_config_t	Contains definitions, structures, and functions related to Bluetooth® LE extended advertising and scanning.
	wiced_ble_ext_scan_results_t	
	wiced_ble_ext_adv_params_t	
	wiced_ble_ext_conn_cfg_phy_options_t	
	wiced_ble_ext_conn_cfg_t	
	wiced_ble_ext_scan_req_received_event_data_t	
	wiced_ble_biginfo_adv_report_t	
	wiced_ble_ext_scan_phy_params_t	
	wiced_ble_ext_scan_params_t	
	wiced_ble_ext_scan_enable_params_t	
	wiced_ble_ext_adv_set_terminated_event_data_t	
	wiced_ble_cmd_cmplt_event_data_t	
	wiced_ble_ext_adv_event_data_t	
LE Periodic Mode Advertising and Scanning	wiced_ble_padv_subevent_data_req_event_data_t	Contains definitions, structures, and functions related to Bluetooth® LE periodic advertising and scanning. These include Periodic Advertising with Response (PAwR) event data, periodic advertising parameters, periodic advertising
	wiced_ble_padv_rsp_report_event_data_t	
	wiced_ble_padv_subevent_data_t	

(table continues...)

Table 204 (continued) Bluetooth® Stack (BTSTACK) library

API reference	Data structures	Description
	wiced_ble_padv_subevent_rsp_data_t	synchronization parameters, and more.
	wiced_ble_padv_params_t	
	wiced_ble_padv_create_sync_params_t	
	wiced_ble_padv_sync_established_event_data_t	
	wiced_ble_padv_sync_transfer_event_data_t	
	wiced_ble_padv_report_event_data_t	
	wiced_ble_padv_sync_transfer_param_t	
LE connection	wiced_bt_ble_phy_preferences_t	Contains connection-related definitions and data structures used in Bluetooth® LE link management. These include the Connection List, Filter Accept List, and PHY settings.
	wiced_bt_ble_conn_params_t	
	wiced_bt_ble_pref_conn_params_t	
	wiced_bt_ble_key_material_t	
Bluetooth® Stack Initialize and Configuration	wiced_bt_cfg_ble_scan_settings_t	Describes the APIs and data structures required to initialize and configure the Bluetooth® Stack.
	wiced_bt_cfg_ble_advert_settings_t	
	wiced_bt_cfg_gatt_t	
	wiced_bt_cfg_avdt_t	
	wiced_bt_cfg_avrc_t	
	wiced_bt_cfg_rfcomm_t	
	wiced_bt_cfg_ble_t	
	wiced_bt_cfg_isoc_t	
	wiced_bt_cfg_settings_t	
Generic Attribute (GATT)	For full data structures, see the Bluetooth® Host Stack Library .	The Generic Attribute Profile (GATT) defines a service framework that enables Bluetooth® Low Energy (LE) applications to function as either a client or server device. This profile also provides capabilities for service discovery as well as reading, writing, notifications, and indications of characteristics defined on a server.

Glossary

ABUS
Analog Output Bus

AC
Alternating current

ADC
Analog-to-digital converter

ADV
Advertising

AES
Advanced Encryption Standard

AHB
AMBA (Advanced Microcontroller Bus Architecture) High-Performance Bus, an Arm® data transfer bus

API
Application programming interface

APOR
Analog power-on reset

BC
Broadcast clock

BCD
Binary coded decimal

BESL
Best effort service latency

Bluetooth® LESS (BLESS)
Bluetooth® LE subsystem

BOD
Brownout detect

BOM
Bill of materials

BR
Bit rate

BRA
Bus request acknowledge

BRQ
Bus request

Glossary

CAN
Controller area network

CI
Carry in

CIC
Cascaded Integrator Comb

CMAC
Cipher-based message authentication code

CMP
Compare

CO
Carry out

COM
LCD common signal

CPHA
Clock phase

CPOL
Clock polarity

CPU
Central processing unit

CPUSS
CPU subsystem

CRC
Cyclic redundancy check

CSD
Capacitive sigma delta

CT
Cipher text

CTI
Cross triggering interface

CTM
Cross triggering matrix

DAC
Digital-to-analog converter

DAP
Debug access port

Glossary

DC
Direct current

DES
Data Encryption Standard

DFF
D flip-flop

DI
Digital or data input

DL
Drive level

DMA
Direct memory access

DMIPS
Dhrystone million instructions per second

DNL
Differential nonlinearity

DO
Digital or data output

DSI
Digital system interconnect

DSM
Deep Sleep mode

DSP
Digital signal processing

DU
Data unit

DW
Datawire

ECO
External crystal oscillator

EEPROM
Electrically erasable programmable read only memory

EMIF
External memory interface

ESR
Equivalent series resistance

Glossary

ETM
Embedded trace macrocell

FB
Feedback

FIFO
First In, First Out

FPU
Floating point unit

FSR
Full-scale range

GAP
Generic access profile

GATT
Generic attribute profile

GFSK
Gaussian Frequency-Shift Keying

GPIO
General-purpose I/O

HCI
Host Controller Interface (Bluetooth® LE Stack)

HFCLK
High-frequency clock

HMAC
Hashed message authentication code

HPF
High-pass filter

HSIOM
High-speed I/O matrix

I/O
Input/Output

I2C
Inter-integrated circuit

I2S
Inter-IC sound

IDE
Integrated development environment

ILO
Internal low-speed oscillator

IMO
Internal main oscillator

INL
Integral nonlinearity

IOR
I/O Read

IOW
I/O Write

IPC
Inter-processor communication

IPTAT
Proportional to absolute temperature

IRA
Interrupt request acknowledge

IRES
Initial power-on reset

IRK
Identity resolution key

IRQ
Interrupt request

ISA
Instruction set architecture

ISR
Interrupt service routine

ITM
Instrumentation trace macrocell

ITO
Indium tin oxide

IVR
Interrupt vector read

IZTAT
Zero dependency to absolute temperature

JWT
JSON web token

Glossary

L2CAP
Logical link control and adaptation protocol

LCD
Liquid crystal display

LFCLK
Low-frequency clock

LFSR
Linear Feedback Shift Register

LIN
Local interconnect network

LJ
Left justified

LL
Link layer

LNA
Low-noise amplifier

LP
System Low-Power mode

LPM
Link power management

LR
Link register

LRb
Last received bit

LRB
Last received byte

LSb
Least significant bit

LSB
Least significant byte

LUT
Lookup table

MAC
Message authentication code

MISO
Master-in, Slave-out

MMIO
Memory-mapped input/output

MOSI
Master-out, Slave-in

MPU
Memory protection unit

MSb
Most significant bit

MSB
Most significant byte

MSP
Main stack pointer

MTB
Micro trace buffer

NI
Next instant

NMI
Non-maskable interrupt

NVIC
Nested vectored interrupt controller

OE
Output enable

OSR
Over-sampling ratio

OVP
Over-voltage protection

PA
Power amplifier

PC
Program counter

PCB
Printed circuit board

PCH
Program counter high

PCL
Program counter low

PD
Power down

PDU
Protocol data unit

PGA
Programmable gain amplifier

PHY
Physical layer

PLD
Programmable logic device

PM
Power management

POR
Power-on reset

PPU
Peripheral protection units

PRNG
Pseudo random number generator

PRS
Pseudo random sequence

PSA
Platform security architecture

PSP
Process stack pointer

PSR
Program Status register

PSRR
Power supply rejection ratio

PSSDC
Power System Sleep duty cycle

PWM
Pulse-width modulator

RAM
Random access memory

RETI
Return from interrupt

RF
Radio frequency

RMS
Root mean square

RNG
Random number generator

ROM
Read only memory

RoT
Root of Trust

RPA
Resolvable private address

RW
Read/Write

SC
Switched capacitor

SCB
Serial Communication Block

SEG
LCD segment signal

SEO
Single-ended zero

SHA-256
Secure Hash Algorithm

SIE
Serial interface engine

SIMO
Single input multiple output

SIO
Special I/O

SMPU
Shared memory protection units

SNR
Signal-to-noise ratio

SOF
Start of frame

SOI
Start of instruction

SP
Stack pointer

SPD
Sequential phase detector

SPI
Serial peripheral interconnect

SPIM
Serial peripheral interconnect master

SPIS
Serial peripheral interconnect slave

SRAM
Static random access memory

SROM
Supervisory read only memory

SRSS
System resources subsystem

SSADC
Single slope ADC

SSC
Supervisory system call

SVCall
Supervisor call

SWD
Single wire debug

SWV
Serial wire viewer

SYSCLK
System clock

TAT
Turnaround time

TC
Terminal count

TCPWM
Timer/counter pulse-width modulator

TD
Transaction descriptors

TDM
Time division multiplexing

TFFF
Toggle flip-flop

TIA
Transimpedance amplifier

TPIU
Trace port interface unit

TRNG
True random number generator

UART
Universal Asynchronous Receiver/Transmitter

UDB
Universal digital block

ULP
System Ultra Low-Power mode

USB
Universal serial bus

USBIO
USB I/O

VTOR
Vector Table Offset register

WCO
Watch crystal oscillator

WDR
Watchdog reset

WDT
Watchdog timer

WIC
Wakeup interrupt controller

XRES
External reset

XRES_N
External reset, active low

Revision history

Document version	Date of release	Description of changes
**	2026-02-05	Initial release
*A	2026-04-09	Publish to web.

Trademarks

The Bluetooth® word mark and logos are registered trademarks owned by Bluetooth® SIG, Inc., and any use of such marks by Infineon is under license.

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2026-04-09

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-xrp1712827926036

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.