EZ-USB™ FX10 (USB 10 Gbps) device controller architecture

Reference manual

About this document

Scope and purpose

The Infineon EZ-USB™ FX10 is a family of USB 3.2 Gen 2x1 device controllers targeting the emerging USB 10 Gbps applications in video, audio, and other generic data acquisition markets. EZ-USB™ FX10 contains Cortex®-M4 and M0+ MCUs, 512 KB flash, 128 KB SRAM, 128 KB ROM, serial communication blocks (SCB) and a CRYPTO engine to support various security features.

A high-bandwidth data subsystem provides DMA data transfers between LVDS/LVCMOS and USB ports at speeds up to 10 Gbps. An additional 1-MB SRAM is included in the high-bandwidth data subsystem to provide buffering for USB data. EZ-USB™ FX10 supports USB Type-C plug orientation detection and flip-MUX function without the need for an external high-speed MUX.

Intended audience

The document is intended for users of EZ-USB™ FX10 USB 3.2 Gen 2x1 device controller.

Table of contents

About this document 1

Scope and purpose 1

Intended audience 1

Table of contents 2

Section A: Overview 15

1 Introduction 16
1.1 Features 16
1.2 Applications 18
1.3 Block diagram 18

2 Functional overview 19
2.1 USB interface 19
2.1.1 USB 3.2 Gen 2×1 device 19
2.1.2 USB 2.0 High-Speed device 19
2.1.3 ReNumeration 20
2.1.4 Precision time measurement 20
2.2 High-bandwidth data subsystem 20
2.2.1 GPIF III interface 20
2.2.2 High-bandwidth direct memory access (HB-DMA) 21
2.3 CPU subsystem 21
2.4 System resources 22
2.4.1 Power system 22
2.4.2 Power modes 22
2.4.3 Booting options 24
2.4.4 Clock system 24
2.4.5 Internal main oscillator (IMO) 24
2.4.6 Internal low-speed oscillator (ILO) 24
2.5 System-level ESD 25
2.5.1 External crystal oscillators 26
2.5.2 Clock dividers 26
2.5.3 Trigger routing 27
2.5.4 Reset 27
2.6 Fixed-function digital 28
2.6.1 Timer/counter/pulse-width modulator (TCPWM) 28
2.6.2 Serial Communication Block (SCB) 28
2.6.3 USB full-speed device interface 29
2.6.4 Quad-SPI/serial memory interface (SMIF) 29
2.7 GPIO 29
2.8 Special-function peripherals 30
2.8.1 Audio subsystem 30

3 Document construction 31
3.1 Major sections 31
3.2 Documentation conventions 31
3.2.1 Register conventions 31
3.2.2 Numeric naming 31
3.2.3 Unit of measure 32

Table of contents

3.2.4 Acronyms and abbreviations ... 33

Section B: USB interface 37

4 USB interface ... 38
4.1 Supported modes ... 38
4.2 Introduction to USB 3.2 ... 38
4.3 Features ... 40
4.4 Block diagram ... 40
4.5 Overview of USB 3.2 interface block ... 41
4.5.1 USB3.2 PHY ... 41
4.5.2 USB3.2 device ... 41
4.5.3 Stream support for USB 3.2 ... 45
4.5.4 USB precision time measurement (PTM) support ... 45
4.5.5 Endpoint memory (EPM) ... 48
4.5.6 DMA controller ... 48
4.6 Overview of USB 2.0 interface block ... 49
4.6.1 Features ... 49
4.6.2 Block diagram ... 49
4.6.3 Overview ... 50
4.6.4 USB 2 device controller – Low power modes ... 51
4.7 USB programming model ... 52
4.7.1 USB Controller Abstraction Layer (CAL) ... 53
4.7.2 USB device (USBD) Layer ... 54

Section C: CPU subsystem 61

5 CPU subsystem (CPUSS) ... 63
5.1 Features ... 63
5.2 Architecture ... 64
5.2.1 Address and memory maps ... 65
5.2.2 Registers ... 66
5.2.3 Operating modes and privilege levels ... 68
5.2.4 Instruction set ... 68

6 SRAM controller ... 70
6.1 Features ... 70
6.2 Architecture ... 70
6.3 Wait states ... 71

7 Inter-processor communication ... 72
7.1 Features ... 72
7.2 Architecture ... 73
7.2.1 IPC channel ... 73
7.2.2 IPC interrupt ... 74
7.2.3 IPC channels and interrupts ... 75
7.2.4 Message passing ... 76
7.2.5 Implementing locks ... 77
7.2.6 Message passing ... 77
7.2.7 Typical usage models ... 79

8 Fault monitoring ... 81
8.1 Features ... 81
8.2 Architecture ... 81

Table of contents

8.2.1 Fault report . . . 82
8.2.2 Signaling interface . . . 83
8.2.3 Monitoring . . . 84
8.2.4 Low-power mode operation . . . 85
8.2.5 Using a fault structure . . . 85
8.2.6 CPU exceptions versus fault monitoring . . . 86
8.3 Fault sources . . . 86
8.4 Register list . . . 88

9 Interrupts . . . 89
9.1 Features . . . 89
9.2 Architecture . . . 89
9.3 Interrupts and exceptions - operation . . . 90
9.3.1 Interrupt/exception handling . . . 90
9.3.2 Level and pulse interrupts . . . 91
9.3.3 Exception vector table . . . 92
9.4 Exception sources . . . 94
9.4.1 Reset exception . . . 94
9.4.2 Non-maskable interrupt exception . . . 94
9.4.3 HardFault exception . . . 95
9.4.4 Memory management fault exception . . . 95
9.4.5 Bus fault exception . . . 95
9.4.6 Usage fault exception . . . 96
9.4.7 Supervisor Call (SVCall) exception . . . 96
9.4.8 PendSupervisory (PendSV) exception . . . 96
9.4.9 System Tick (SysTick) exception . . . 96
9.5 Interrupt sources . . . 97
9.6 Interrupt/exception priority . . . 104
9.7 Enabling and disabling interrupts . . . 105
9.8 Interrupt/exception states . . . 106
9.8.1 Pending interrupts/exceptions . . . 106
9.9 Stack usage for interrupts/exceptions . . . 107
9.10 Interrupts and low-power modes . . . 107
9.11 Interrupt/exception – initialization/ configuration . . . 108
9.12 Register list . . . 109

10 Protection units . . . 110
10.1 Architecture . . . 110
10.2 EZ-USB™ FX10 protection architecture . . . 111
10.3 Register architecture . . . 115
10.3.1 Protection structure and attributes . . . 115
10.4 Bus master protection attributes . . . 117
10.5 Protection context . . . 118
10.6 Protection contexts 0, 1, 2, 3 . . . 118
10.7 Protection structure . . . 120
10.7.1 Protection violation . . . 120
10.7.2 MPU . . . 120
10.7.3 SMPU . . . 120
10.7.4 PPU . . . 121
10.7.5 Protection of protection structures . . . 129
10.7.6 Protection structure types . . . 129

Table of contents

11 DMA controller (DMAC) ... 131
11.1 Features ... 131
11.2 Architecture ... 131
11.3 Channels ... 132
11.3.1 Channel interrupts ... 133
11.4 Descriptors ... 133
11.4.1 Address configuration ... 136
11.4.2 Transfer size ... 138
11.4.3 Descriptor chaining ... 139
11.5 DMAC controller ... 139
11.5.1 Trigger selection ... 140
11.5.2 Channel logic ... 140
11.5.3 Output triggers ... 140

12 Data wire controller (DW) ... 141
12.1 Features ... 141
12.2 Architecture ... 141
12.3 Channels ... 142
12.3.1 Channel interrupts ... 143
12.4 Descriptors ... 144
12.4.1 Address configuration ... 146
12.4.2 Transfer size ... 148
12.4.3 Descriptor chaining ... 149
12.5 DMA controller ... 150
12.5.1 Trigger selection ... 150
12.5.2 Pending triggers ... 150
12.5.3 Output triggers ... 151
12.5.4 Status registers ... 151
12.5.5 DMA performance ... 151

13 Cryptographic function block (CRYPTO) ... 154
13.1 Features ... 154
13.2 Architecture ... 154
13.3 Instruction controller ... 155
13.3.1 Instructions ... 156
13.3.2 Instruction operands ... 156
13.3.3 Load and store FIFO instructions ... 157
13.3.4 Register buffer instructions ... 159
13.4 Hash algorithms ... 164
13.4.1 SHA1 and SHA2 ... 164
13.4.2 SHA3 ... 166
13.5 DES and TDES ... 167
13.6 AES ... 169
13.7 CRC ... 171
13.8 PRNG ... 174
13.9 TRNG ... 175
13.10 Vector unit ... 181
13.10.1 VU register file ... 183
13.10.2 Stack ... 184
13.10.3 Memory operands ... 185
13.10.4 Datapath ... 185

Table of contents

13.10.5 Status register . . . 186
13.10.6 Instructions . . . 187
13.10.7 Instruction set . . . 188

14 Nonvolatile memory . . . 205
14.1 Flash memory . . . 205
14.1.1 Features . . . 205
14.1.2 Configuration . . . 205
14.1.3 Flash geometry . . . 206
14.1.4 Flash controller . . . 207
14.1.5 Read while write (RWW) support . . . 208
14.2 Flash memory programming . . . 209
14.2.1 Features . . . 209
14.2.2 Architecture . . . 209
14.3 System call implementation . . . 210
14.3.1 System call via CM0+ or CM4 . . . 210
14.3.2 System call via DAP . . . 210
14.3.3 Exiting from a system call . . . 210
14.3.4 SRAM usage . . . 210
14.4 SROM API library . . . 211
14.5 System calls . . . 213
14.5.1 Infineon ID . . . 213
14.5.2 Write row . . . 217
14.5.3 Program row . . . 219
14.5.4 Erase all . . . 221
14.5.5 Checksum . . . 223
14.5.6 Compute hash . . . 225
14.5.7 ConfigureRegionBulk . . . 226
14.5.8 DirectExecute . . . 227
14.5.9 Erase sector . . . 227
14.5.10 Soft Reset . . . 229
14.5.11 Erase row . . . 230
14.5.12 Erase subsector . . . 231
14.5.13 GenerateHash . . . 233
14.5.14 ReadUniqueID . . . 234
14.5.15 CheckFactoryHash . . . 235
14.5.16 TransitionToRMA . . . 236
14.5.17 ReadFuseByteMargin . . . 237
14.5.18 TransitionToSecure . . . 238
14.6 System Call Status . . . 239

15 Boot code . . . 240
15.1 Features . . . 240
15.2 ROM boot . . . 240
15.2.1 Data integrity checks . . . 240
15.2.2 Life-cycle stages and protection states . . . 244
15.2.3 Secure boot in ROM boot . . . 247
15.2.4 Protection setting . . . 247
15.2.5 SWD/JTAG repurposing . . . 249
15.2.6 Disable WDT . . . 249
15.2.7 ROM boot flow chart . . . 250

Table of contents

15.3 Flash boot . . . 251
15.3.1 Overview . . . 251
15.3.2 Features of flash boot . . . 251
15.3.3 Using flash boot . . . 251
15.3.4 Flash boot layout . . . 252
15.3.5 Flash boot flow chart . . . 253

16 Device security . . . 258
16.1 Features . . . 258
16.2 Architecture . . . 258
16.2.1 Life cycle stages and protection sates . . . 258
16.2.2 Flash security . . . 262
16.2.3 Hardware-based encryption . . . 262

17 Program and debug interface . . . 263
17.1 Features . . . 263
17.2 Architecture . . . 263
17.2.1 Debug access port (DAP) . . . 265
17.2.2 ROM tables . . . 266
17.2.3 Trace . . . 266
17.2.4 Embedded cross triggering . . . 266
17.3 Serial wire debug (SWD) interface . . . 267
17.3.1 SWD timing details . . . 268
17.3.2 ACK details . . . 269
17.3.3 Turnaround (Trn) period details . . . 269
17.4 JTAG interface . . . 270
17.5 Programming the EZ-USB™ FX10 MCU . . . 272
17.5.1 SWD port acquisition . . . 272
17.5.2 SWD programming mode entry . . . 272
17.5.3 SWD programming routine executions . . . 272
17.6 Registers . . . 273

Section D: System resources subsystem (SRSS) 274

18 Power supply and monitoring . . . 275
18.1 Features . . . 275
18.2 Architecture . . . 275
18.3 Power supply . . . 276
18.3.1 Regulators summary . . . 276
18.3.2 Power pins and rails . . . 277
18.3.3 Power sequencing requirements . . . 277
18.3.4 Power supply sources . . . 278
18.4 Voltage monitoring . . . 278
18.4.1 Power-on reset (POR) . . . 278
18.4.2 Brownout detect (BOD) . . . 278
18.4.3 Low-voltage detect (LVD) . . . 278
18.4.4 Over voltage protection (OVP) . . . 279
18.5 Register list . . . 280

19 Device power modes . . . 281
19.1 Features . . . 281
19.2 Architecture . . . 281
19.2.1 CPU Power modes . . . 282

Table of contents

19.2.2 CPU Active mode . . . 282
19.2.3 CPU Sleep mode . . . 282
19.2.4 System Power modes . . . 283
19.2.5 System Deep Sleep mode . . . 283
19.2.6 Other Operation modes . . . 283
19.3 Power mode transitions . . . 284
19.3.1 Power-up transitions . . . 285
19.3.2 Power mode transitions . . . 286
19.3.3 Wakeup transitions . . . 287
19.4 Summary . . . 288
19.5 Register list . . . 289

20 Clocking system . . . 290
20.1 Block diagram . . . 290
20.2 Clock sources . . . 291
20.2.1 Internal main oscillator (IMO) . . . 291
20.2.2 External crystal oscillator (ECO) . . . 291
20.2.3 External clock (EXT_CLK) . . . 293
20.2.4 Internal low-speed oscillator (ILO) . . . 293
20.2.5 ECO prescaler . . . 294
20.3 Clock generation . . . 294
20.3.1 PLL without SSCG and fractional operation . . . 294
20.3.2 Frequency locked loop (FLL) . . . 295
20.4 Clock trees . . . 298
20.4.1 Path clocks . . . 298
20.4.2 High-frequency root clocks . . . 299
20.4.3 Low-frequency root clocks . . . 299
20.4.4 Timer clock . . . 300
20.4.5 Clock output function . . . 300
20.5 CLK_HF distribution . . . 300
20.5.1 CLK_MEM . . . 300
20.5.2 CLK_SLOW . . . 300
20.5.3 CLK_FAST_0 and CLK_FAST_1 . . . 300
20.5.4 CLK_PERI . . . 301
20.5.5 PCLK . . . 301
20.5.6 CLK_GR . . . 301
20.5.7 CLK_TRC_DBG . . . 301
20.6 Peripheral clock dividers . . . 301
20.6.1 Fractional clock dividers . . . 302
20.6.2 Peripheral clock divider configuration . . . 302
20.7 Clock calibration counters . . . 304
20.8 Clock supervision (CSV) . . . 305
20.8.1 Overview . . . 305
20.8.2 CSV operation . . . 306
20.9 Registers . . . 309

21 Reset system . . . 313
21.1 Features . . . 313
21.2 Architecture . . . 313
21.2.1 Power-on reset (POR) . . . 315
21.2.2 Brownout reset . . . 315

Table of contents

21.2.3 Watchdog timer reset ... 315
21.2.4 Software initiated reset ... 315
21.2.5 External reset ... 315
21.2.6 Logic protection fault reset ... 316
21.2.7 Clock-supervision logic reset ... 316
21.3 Identifying reset sources ... 316
21.4 Register list ... 317

22 Watchdog timer ... 318
22.1 Features ... 318
22.2 Architecture ... 318
22.3 Free-running WDT ... 319
22.3.1 Overview ... 319
22.3.2 Watchdog reset ... 321
22.3.3 Watchdog interrupt ... 322
22.4 Multi-counter WDTs ... 323
22.4.1 Overview ... 323
22.4.2 Enabling and disabling WDT ... 327
22.5 Watchdog Cascade options ... 328
22.5.1 MCDWT Reset ... 330
22.5.2 MCWDT Interrupt ... 330
22.6 Reset cause detection ... 331
22.7 Register list ... 331

23 I/O system ... 332
23.1 Features ... 332
23.2 Architecture ... 332
23.2.1 I/O cell architecture ... 333
23.2.2 Digital input buffer ... 334
23.2.3 Digital output driver ... 334
23.3 High-speed I/O matrix ... 338
23.4 I/O state on power up ... 339
23.5 Behavior in Low-power modes ... 340
23.6 Interrupt ... 340
23.7 Peripheral connections ... 342
23.7.1 Firmware-controlled GPIO ... 342
23.8 Registers ... 343

24 Trigger multiplexer block ... 344
24.1 Features ... 344
24.2 Architecture ... 344
24.2.1 Trigger multiplexer group ... 345
24.2.2 One-to-one trigger ... 345
24.2.3 Trigger multiplexer block ... 345
24.2.4 Software triggers ... 353
24.3 Register list ... 354

Section E: High-bandwidth subsystem (HBWSS) 355

25 LVDS/LVCMOS IO subsystem ... 356
25.1 Features ... 356
25.2 Architecture ... 357
25.2.1 LVDS/LVCMOS ports vs. LVDS/LVCMOS links ... 358

Table of contents

25.2.2 LVDS and LVCMOS port combinations . . . 359
25.2.3 LVCMOS master mode configuration . . . 362
25.2.4 LVDS Link configurations . . . 362
25.2.5 LVCMOS link configurations . . . 365
25.3 Functional operating modes . . . 366
25.3.1 Training mode . . . 366
25.3.2 Data transmission mode . . . 367
25.4 LVDS control byte encoding . . . 367
25.5 Internal pattern generator . . . 377
25.5.1 Programming flow . . . 378
25.6 Clocks . . . 380
25.7 Power modes . . . 382
25.7.1 Entry to L1 from L0 . . . 383
25.7.2 Exit from L1 to L0 . . . 383
25.7.3 Entry to L3 to L0 . . . 383
25.7.4 Procedure to exit L3 and enter L0 . . . 384

26 GPIF III interface . . . 385
26.1 Typical GPIF III interfaces . . . 386
26.2 Initialization and configuration of GPIF III block . . . 387
26.2.1 GPIF III state machine control . . . 387
26.2.2 LVDS flow control . . . 387
26.2.3 Known limitations . . . 387

27 High-bandwidth DMA subsystem . . . 390
27.1 Operating modes . . . 392
27.1.1 Functional mode . . . 392
27.1.2 Power modes . . . 392
27.2 DMA concepts . . . 392
27.2.1 Possible data flow in EZ-USB™ FX10 . . . 393
27.2.2 Decentralized DMA . . . 399
27.2.3 Sockets, descriptors, buffers and packets . . . 399
27.2.4 Socket operation . . . 402
27.2.5 DMA adapter . . . 406
27.2.6 DMA register definitions . . . 407
27.3 Operation . . . 407
27.4 Link layer . . . 409
27.5 DMA transfer management . . . 417
27.5.1 DMA configuration . . . 417
27.5.2 DMA operation . . . 423
27.5.3 CPUSS DMA operation . . . 426
27.5.4 Handling short packets on the USB 2.0 device . . . 427
27.6 HBWSS DMA application programming interface . . . 428
27.6.1 HBWSS DMA data structures . . . 428
27.6.2 HBWSS DMA functions . . . 431
27.7 CPUSS DMA functions . . . 437

Section F: Digital subsystem 438

28 Serial communications block (SCB) . . . 439
28.1 Features . . . 439
28.2 Architecture . . . 439

Table of contents

28.2.1 Buffer modes . . . 439
28.2.2 FIFO mode . . . 440
28.2.3 Clocking modes . . . 440
28.3 Serial peripheral interface (SPI) . . . 442
28.3.1 Features . . . 442
28.3.2 General description . . . 443
28.3.3 SPI modes of operation . . . 444
28.3.4 SPI Buffer modes . . . 449
28.3.5 Clocking and oversampling . . . 454
28.3.6 Enabling and initializing SPI . . . 458
28.3.7 I/O pad connection . . . 459
28.3.8 SPI registers . . . 461
28.4 UART . . . 462
28.4.1 Features . . . 462
28.4.2 General description . . . 463
28.4.3 UART modes of operation . . . 463
28.4.4 Clocking and oversampling . . . 475
28.4.5 Enabling and initializing the UART . . . 475
28.4.6 I/O pad connection . . . 476
28.4.7 UART registers . . . 478
28.5 Inter integrated circuit (I2C) . . . 479
28.5.1 Features . . . 479
28.5.2 General description . . . 479
28.5.3 External electrical connections . . . 480
28.5.4 Terms and definitions . . . 481
28.5.5 I2C modes of operation . . . 482
28.5.6 I2C Buffer modes . . . 484
28.5.7 Clocking and oversampling . . . 489
28.5.8 Enabling and initializing the I2C . . . 493
28.5.9 I/O pad connections . . . 494
28.5.10 I2C registers . . . 495
28.6 SCB interrupts . . . 496
28.6.1 SPI interrupts . . . 497
28.6.2 UART interrupts . . . 501
28.6.3 I2C interrupts . . . 504

29 Serial memory interface (SMIF) . . . 506
29.1 Features . . . 506
29.2 Architecture . . . 506
29.2.1 TX and RX FIFOs . . . 508
29.2.2 MMIO mode . . . 510
29.2.3 XIP mode . . . 510
29.2.4 Cache . . . 511
29.2.5 Arbitration . . . 512
29.2.6 Deselect Delay . . . 512
29.2.7 Cryptography . . . 512
29.3 Memory device signal interface . . . 513
29.3.1 Specifying memory devices . . . 513
29.3.2 Connecting SPI memory devices . . . 514
29.3.3 SPI Data Transfer . . . 520

Table of contents

29.3.4 Example of Setting up SMIF . . . 521
29.4 Triggers . . . 523
29.5 Interrupts . . . 523
29.6 Sleep operation . . . 524
29.7 Performance . . . 524

30 CAN FD controller . . . 525
30.1 Overview . . . 525
30.1.1 Features . . . 525
30.1.2 Features not supported . . . 525
30.2 Configuration . . . 526
30.2.1 Block diagram . . . 526
30.2.2 Dual clock sources . . . 526
30.2.3 Interrupt lines . . . 526
30.3 Functional description . . . 527
30.3.1 Operation modes . . . 527
30.3.2 Timestamp generation . . . 534
30.3.3 Timeout counter . . . 534
30.3.4 RX handling . . . 535
30.3.5 TX handling . . . 542
30.3.6 FIFO acknowledge handling . . . 547
30.3.7 Configuring the CAN bit timing . . . 547
30.4 Message RAM . . . 550
30.4.1 Message RAM configuration . . . 550
30.4.2 RX buffer and FIFO element . . . 551
30.4.3 TX buffer element . . . 554
30.4.4 TX event FIFO element . . . 557
30.4.5 Standard Message ID Filter Element . . . 559
30.4.6 Extended Message ID filter element . . . 561
30.4.7 Trigger memory element . . . 563
30.4.8 RAM watchdog (RWD) . . . 565
30.5 TTCAN operation . . . 565
30.5.1 Reference message . . . 565
30.5.2 TTCAN configuration . . . 566
30.5.3 TTCAN gap control . . . 574
30.5.4 Stop watch . . . 575
30.5.5 Local time, cycle time, global time, and external clock synchronization . . . 575
30.5.6 TTCAN error level . . . 577
30.5.7 TTCAN message handling . . . 578
30.5.8 TTCAN interrupt and error handling . . . 580
30.5.9 Level 0 . . . 581
30.5.10 Synchronization to external time schedule . . . 583
30.6 Setup procedures . . . 584
30.6.1 General program flow . . . 584
30.6.2 Clock stop request . . . 584
30.6.3 MRAM OFF operation . . . 585
30.6.4 MRAM ON operation . . . 585
30.6.5 Procedures specific to M_TTCAN channel . . . 586
30.7 Registers . . . 596

31 Timer, counter, and PWM (TCPWM) . . . 600

Table of contents

31.1 Features . . . 600
31.2 Architecture . . . 600
31.2.1 Enabling and disabling counters in TCPWM block . . . 601
31.2.2 Clocking . . . 601
31.2.3 Trigger inputs . . . 602
31.2.4 Trigger outputs . . . 603
31.2.5 Interrupts . . . 604
31.2.6 PWM outputs . . . 604
31.2.7 Power modes . . . 605
31.3 Operation modes . . . 605
31.3.1 Timer mode . . . 606
31.3.2 Capture mode . . . 613
31.3.3 Quadrature Decoder mode . . . 616
31.3.4 Pulse-width modulation mode . . . 620
31.3.5 Pulse width modulation with Dead Time mode . . . 632
31.3.6 Pulse width modulation Pseudo-Random mode (PWM_PR) . . . 634
31.4 TCPWM registers . . . 638

32 Inter-IC sound bus . . . 639
32.1 Features . . . 639
32.2 Architecture . . . 639
32.3 Digital audio interface formats . . . 640
32.3.1 Standard I2S format . . . 640
32.3.2 Left Justified (LJ) format . . . 642
32.3.3 Time division multiplexed (TDM) format . . . 643
32.4 Clocking polarity and delay options . . . 644
32.5 Interfacing with Audio Codecs . . . 644
32.6 Clocking features . . . 645
32.7 FIFO buffer and DMA support . . . 646
32.8 Interrupt support . . . 649
32.9 Watchdog timer . . . 650

33 PDM-PCM converter . . . 651
33.1 Features . . . 651
33.2 Architecture . . . 651
33.2.1 Enable/disable converter . . . 652
33.2.2 Clocking features . . . 652
33.2.3 Over-sampling ratio . . . 653
33.2.4 Mono/stereo microphone support . . . 653
33.2.5 Hardware FIFO buffers and DMA controller support . . . 655
33.2.6 Interrupt support . . . 656
33.2.7 Digital volume gain . . . 657
33.2.8 Smooth gain transition . . . 657
33.2.9 Soft mute . . . 658
33.2.10 Word length and sign bit extension . . . 658
33.2.11 High-pass filter . . . 658
33.2.12 Enable/disable streaming . . . 658
33.2.13 Power modes . . . 658
33.3 Operating procedure . . . 659
33.3.1 Initial configuration . . . 659
33.3.2 Interrupt service routine (ISR) configuration . . . 659

Table of contents

33.3.3 Enabling / disabling streaming 660

Revision history **661**

Section A: Overview

This section encompasses the following chapters:

- **"Introduction"** on page 16
- **"Functional overview"** on page 19
- **"Document construction"** on page 31

1 Introduction

The Infineon EZ-USB™ FX10 is a family of USB 3.2 Gen 2×1 device controllers targeting the emerging USB 10 Gbps applications in video, audio, and other generic data acquisition markets. EZ-USB™ FX10 contains Cortex®-M4 and M0+ MCUs, 512 KB flash, 128 KB SRAM, 128 KB ROM, serial communication blocks (SCB) and a CRYPTO engine to support various security features. A high-bandwidth data subsystem provides DMA data transfers from LVDS / LVCMOS input to USB output at speeds up to 10 Gbps for USB 3.2 Gen 2x1 transfer rates. A 1 MB SRAM is included in the high-bandwidth data subsystem to provide buffering for USB data. EZ-USB™ FX10 supports USB-C connectors with USB-C plug orientation detection and correction without the need for an external high-speed MUX.

1.1 Features

- USB 3.2 Gen 2x1 device port
  - Integrated USB 10 Gbps and Hi-Speed PHYs: 10 Gbps, 5 Gbps, 480 Mbps
  - USB Type-C plug orientation detection and correction
  - Up to 32 USB endpoints
- 32-bit dual-core CPU subsystem
  - 150 MHz Arm® Cortex®-M4F (CM4) CPU with single-cycle multiply, floating point, and memory protection unit (MPU)
  - 100 MHz Arm® Cortex®-M0+ (CM0+) CPU with single-cycle multiply and MPU
- Memory subsystem
  - 512 KB application flash, read-while-write (RWW) support
  - Two 8 KB flash caches, one for each CPU
  - 128 KB SRAM with power and data retention control
  - 128 KB ROM for device initialization, flash write, security, eFuse programming
  - 1024 KB SRAM for LVDS / LVCMOS to USB data buffer
  - 1 KB one-time programmable (OTP) eFuse array
- Dual-port General Programmable Interface (GPIF III)
  - LVDS/sub-LVDS multi-lane serial-data bus receiver mode
    - Each port consists of eight data lanes, one control lane, one clock lane, and seven LVCMOS control signals
    - Two ports can be combined into one link of 16 data lanes or as two independent links of up to eight lanes each
    - Maximum data rate: 1.25 Gbps per lane RX in DDR mode and 625 Mbps per lane RX in SDR mode
  - LVCMOS parallel-data bus transceiver mode
    - Each port consists of sixteen data, one clock, and ten control signals
    - Two ports can be combined into one 32-bit wide data bus or as two independent links of up to 16 lines each
    - Maximum 160 MHz DDR or SDR mode for receiver (RX)
    - 100 MHz SDR and 80 MHz DDR master mode for transmitter (TX)

Introduction

- Peripheral IO subsystem - total of 48 shared IOs
  - Dual QSPI configurable as single, dual, quad, dual-quad, and octal interfaces
  - Seven serial communication blocks (SCBs) configurable as I2C, UART, or SPI
  - Two-channel configurable audio interfaces
    - I2S master or slave, RX or TX
    - PDM to PCM converter for microphone
  - One USB full-speed (FS) device for virtual COM function
  - Eight TCPWMs (timers / counters / pulse-width modulators)
  - One CAN FD interface
  - GPIOs: each peripheral IO can be configured as GPIO
- Flexible clocking options
  - 8 MHz internal main oscillator (IMO) with ±2% accuracy
  - 32 kHz internal low-speed oscillator (ILO)
  - 24 MHz external crystal oscillator
  - Phase-locked loop (PLL) for multiplying clock frequencies
  - Frequency-locked loop (FLL) for multiplying IMO frequency
  - Integer and fractional peripheral clock dividers
- Low-power support
  - 1.7 V to 3.6 V operation
  - Four power modes for fine-grained power management
  - Deep Sleep mode SRAM retention
- Security built into platform architecture
  - ROM-based root of trust via uninterrupted “Secure Boot”
  - Stepwise authentication of execution images
  - Secured execution of code in execute-only mode for protected routines
  - All debug and test ingress paths can be disabled
  - Eight protection contexts
- Cryptography accelerator
  - Hardware acceleration for symmetric and asymmetric cryptographic methods and hash functions
  - True Random Number Generation (TRNG) function
- Package options
  - 169-ball, 10 × 10 × 1.2 mm, 13 × 13 ball array, 0.75 mm pitch FBGA
- Development tools
  - ModusToolbox™ software enables cross platform code development with a robust suite of tools and software libraries.
  - Supports UVC, UAC, HID, CDC vendor class protocols
  - Compatible with USB3Vision (U3V) standard
  - EZ-USB™ FX10 Explorer Kit (DVK)
  - EZ-USB™ FX10 HDMI 4K60 Capture Solution Demo Kit
  - EZ-USB™ FX10 CAD libraries provide footprint and schematic
  - Support for common tools - BSDL files and IBIS models

Infineon developer community enables connection with fellow USB developers around the world, 24 hours a day, 7 days a week.

1.2 Applications

- HDMI capture cards
- SDI USB video capture cards
- Industrial cameras (machine vision)
- Video conference cameras
- Document cameras
- Surveillance cameras
- Medical Imaging cameras
- USB time-of-flight (TOF) cameras
- USB oscilloscopes
- USB logic analyzers
- USB data acquisition systems

1.3 Block diagram

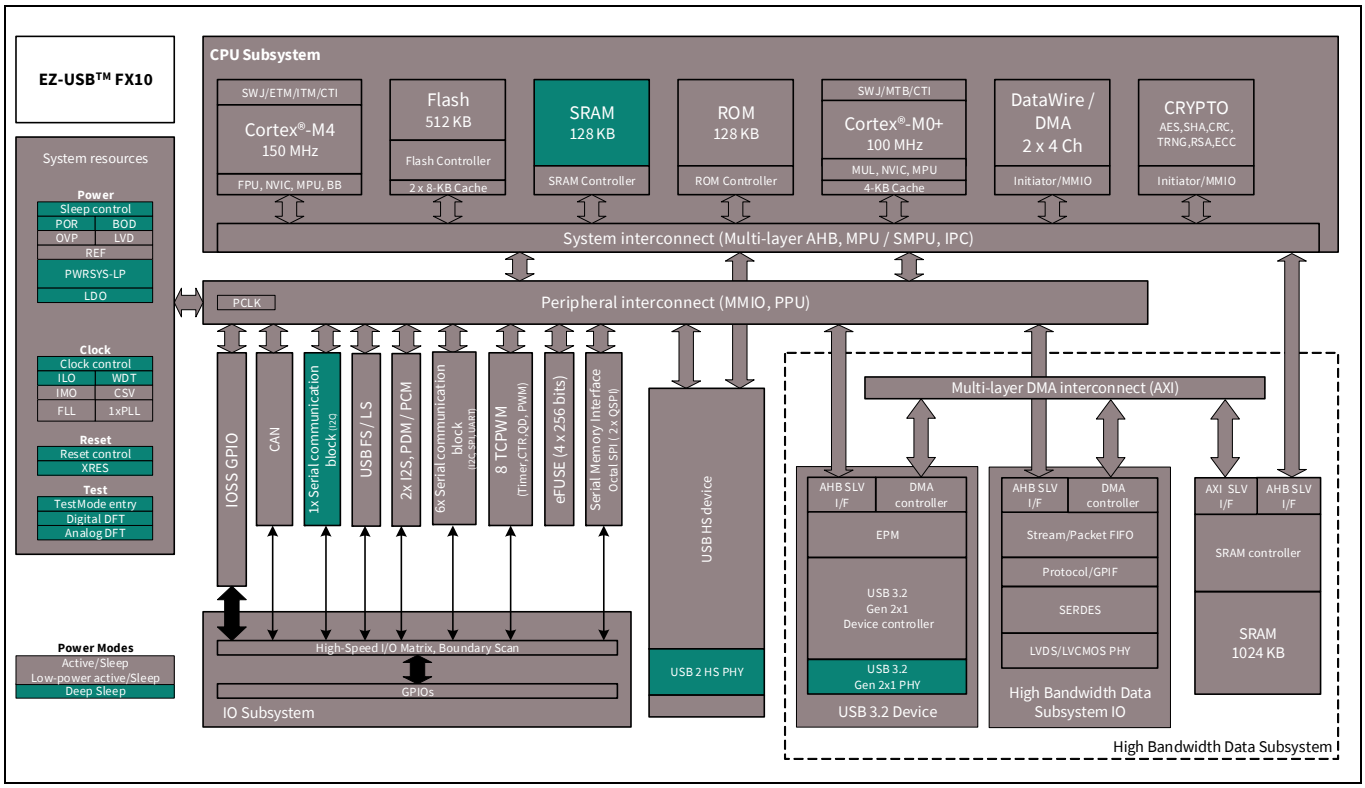


Figure 1-1. EZ-USB™ FX10 architecture diagram

2 Functional overview

2.1 USB interface

2.1.1 USB 3.2 Gen 2×1 device

EZ-USB™ FX10 supports the following features:

- Configurable to operate in USB 3.2 Gen 2 (10 Gbps) or USB 3.2 Gen 1 (5 Gbps)
- USB 2.0 High-Speed (HS) at 480 Mbps
- Up to 32 endpoints (EPs), 16 IN, 16 OUT, each endpoint can support BULK, ISOCHRONOUS, or INTERRUPT transfer
- Supports simultaneous transactions across multiple IN EPs
- Supports all USB low-power states
- Supports UVC, UAC, HID, CDC, U3V, and vendor class protocols
- Supports USB precision time measurement (PTM)
- Supports USB bus-powered and self-powered mode
- Detect USB-C connector orientation using CC lines. External Rd resistors are required on CC lines

2.1.2 USB 2.0 High-Speed device

EZ-USB™ FX10 USB interface block consists of the USB 3.2 Gen 2 device block and USB High-Speed device block.

The following features are supported:

- Supports USB 2.0 high-speed (480 Mbps) and full speed (12 Mbps). No support for USB low-speed
- Supports low-power options
- Supports up to sixteen IN and OUT physical endpoints
- Supports remote wake-up capabilities
- Compliant to USB 2.0 specifications and USB 2.0 DC resistance ECN

2.1.2.1 Compliance to USB 2.0 DC resistance engineering change notice (ECN)

The ECN updates the USB 2.0 specification to account for DC resistance (DCR) on the D+ and D- data paths between the USB transceiver device and the connector. This increased resistance could be due to multiple crossbar switches and/or passive components such as common-mode chokes. The ECN updates affect the following high-speed parameters: VHSOH transmit voltage, VHSDSC disconnect level, and VHSSQ squelch level.

2.1.2.1.1 Allowable transmit output voltage (VHSOH)

The ECN expands the allowable transmit output voltage from 400 mV ± 10% to 360 mV to 625 mV. The Part Number supports the default transmit voltage of 400 mV and a programmable nominal range of 370 mV to 520 mV.

2.1.2.1.2 Disconnect envelope detector threshold voltage (VHSDSC)

The ECN changes the disconnect envelope detector threshold range from 525 mV to 625 mV to a much wider range. The design guide shows a minimum threshold range of 550 mV to 640 mV to a maximum of ~820 mV to ~950 mV. EZ-USB™ FX10 supports programmable threshold levels that cover the original 525 mV to 625 mV range up to the maximum range of ~820 mV to ~950 mV.

2.1.2.1.3 Squelch threshold voltage (VHSSQ)

The ECN changes the squelch threshold from 100 mV to 150 mV to a lowered range depending on the DCR. EZ-USB™ FX10 supports the original threshold range of 100 mV to 150 mV and can be programmed to a lowered threshold range of 85 mV to 135 mV.

EZ-USB™ FX10 also supports multiple programmable options for pre-emphasis which can be used to compensate for increased DCR and other components that cause channel degradation.

2.1.3 ReNumeration

Because of EZ-USB™ FX10's soft configuration, one chip can take on the identities of multiple distinct USB devices. When first plugged into USB, EZ-USB™ FX10 enumerates automatically with the vendor ID (0x04B4) and downloads firmware and USB descriptors over the USB interface. The downloaded firmware executes an electrical disconnect and reconnect. The Part Number enumerates again, this time as a device defined by the downloaded information. This patented two-step process, called "ReNumeration", happens instantly when the device is plugged in.

2.1.4 Precision time measurement

When multiple devices are connected to the same host, it is necessary for them to have a common sense of time based on the host specified timestamp to avoid deviations due to different propagation delays. Precision time measurement (PTM) provides a mechanism for the device to estimate the link and hub propagation delays to compensate for this deviation. For more details, see **"USB precision time measurement (PTM) support"** on page 45.

2.2 High-bandwidth data subsystem

The high-bandwidth data subsystem provides DMA data transfers between LVDS / LVCMOS IO subsystem and the USB 3.2 Gen 2 device at speeds up to 10 Gbps. A 1-MB SRAM is included in the high-bandwidth data subsystem to provide buffering for data. The data is transferred over an AXI3 interconnect. This subsystem interfaces to the System Interconnect for data transfer to other low-bandwidth peripherals and the Memory-mapped IO (MMIO) Interconnect.

The high-bandwidth data subsystem IOs can be configured as LVDS or LVCMOS interfaces. It has two links and each link can be configured independently to support the LVDS interface or LVCMOS parallel interface.

For more details, see **"LVDS/LVCMOS IO subsystem"** on page 356.

2.2.1 GPIF III interface

The GPIF III is a programmable state machine that enables a flexible interface that may function as a master or slave in industry-standard or proprietary interfaces. Only parallel interfaces are implemented with GPIF III.

The following are the GPIF III features:

- Functions as master or slave
- Provides 256 firmware programmable states
- Supports 8-bit, 16-bit, 24-bit, and 32-bit parallel data bus
- Supports two independent instances of GPIF III state machines, each connected to a link, or a single GPIF III state machine (first instance) connected to a wide link

The GPIF III state transitions are based on control input signals. The control output signals are driven as a result of the GPIF III state transitions. For more details, see **"GPIF III interface"** on page 385.

2.2.2 High-bandwidth direct memory access (HB-DMA)

The DMA controller includes the following features:

- Supports multiple simultaneous streams
- Supports maximum throughput of 20 Gbps
- DMA is configurable by the CPU. Once configured, the DMA operates automatically and transfers the video stream from the LVDS/LVCMOS interface to the USB 3.2 device without CPU intervention
- Adds header and footer with every data packet (payload) without CPU intervention

For more details, see **"High-bandwidth DMA subsystem"** on page 390.

2.3 CPU subsystem

The CPU subsystem has the following major components:

- 150 MHz Cortex®-M4 core
- 100 MHz Cortex®-M0+ core
- 512 KB flash memory
- 128 KB SRAM
- 128 KB ROM
- DMA
- Cryptography accelerator

Following are the multiple bus masters in EZ-USB™ FX10 (see block diagram in **Figure 1-1**):

- CPUs
- DMA controllers
- QSPI
- USB
- Cryptography accelerator

Generally, all memory and peripherals can be accessed and shared by all bus masters through multi-layer Arm® AMBA high-performance bus (AHB) arbitration. Accesses between CPUs can be synchronized using an interprocessor communication (IPC) block. For more details see **"CPU subsystem"** on page 61.

2.4 System resources

2.4.1 Power system

The power system assures that voltage levels are as required for each respective mode and will either delay mode entry (for example, power-on reset (POR)) until voltage levels are as required for proper function or generate resets (brownout detect (BOD)) when the power supply drops below specified levels. The design guarantees safe chip operation between the power supply voltage drops below-specified levels (for example, below 1.7 V) and the reset occurs. There are no voltage sequencing requirements.

The VDDD supply (1.7 to 3.6 V) powers a low-dropout regulator (LDO). The System Low-power (LP) operates the core operating voltage (VCCD) at 1.1 V and offers high performance with no restrictions on device configuration.

2.4.2 Power modes

EZ-USB™ FX10 can operate in four systems and three CPU power modes. These modes are intended to minimize the average power consumption in an application.

The EZ-USB™ FX10 supports the following power modes, in order to decrease power consumption:

- System Low-power (LP) – All peripherals and CPU power modes are available at maximum speed
- CPU Active – CPU is executing code in system LP
- CPU Sleep – CPU code execution is halted in system LP
- CPU Deep Sleep – CPU code execution is halted and system Deep Sleep is requested in system LP mode
- System Deep Sleep – Only low-frequency peripherals are available after both CPUs enter CPU Deep Sleep mode
- Normal mode – This is the full-functional operating mode. The internal CPU clock and the internal PLLs are enabled in this mode.
  - USB3 operation at 5 Gbps or 10 Gbps (U0 mode)
  - USB3 low-power modes (U1, U2)
  - High-bandwidth data subsystem IOs port in LVCMOS SDR or DDR mode
  - High-bandwidth data subsystem IOs port in LVDS 625 Mbps or 1.25 Gbps mode

The I/O power supplies VDDIOP0, VDDIOP1, VDDIO_P0CTRL, and VDDIO_P1CTRL can be turned off when the corresponding interface is not in use. These supplies cannot be turned OFF at any time if the GPIF III interface is used in the application.

- Other Low-power modes (see **Table 2-1**):
  - Suspend mode with USB 3.0 mode
  - Deep Sleep mode
  - Core power-down mode

CPU Active, Sleep, and Deep Sleep are standard Arm®-defined power modes supported by the Arm® CPU instruction set architecture (ISA). The System LP mode is an additional low-power mode supported by the EZ-USB™ FX10.

Table 2-1. Entry and exit methods for low-power modes

Low-power modes	Characteristics	Methods of entry	Methods of exit
Suspend mode with USB 3.2 PHY enabled	The power consumption in this mode does not exceed 1.8 mW. USB 3.2 PHY is enabled and is in U3 mode (one of the suspend modes defined by the USB 3.2 specification). This one block alone is operational with its internal clock, while all other clocks are shut down. All I/Os maintain their previous state. The power supply for the wakeup source and core power must be retained. All other power domains can be turned on/off individually. CPU shall be in Deep Sleep mode The states of the configuration registers, buffer memory, and all internal RAM are maintained. All transactions must be completed before Part Number enters Suspend mode (the state of outstanding transactions is not preserved). The firmware resumes operation from where it was suspended (except when woken up by an XRES assertion) because the program counter does not reset.	Firmware executing on the Arm® core can put Part Number into Suspend mode. For example, on a USB suspend condition, the firmware may decide to put Part Number into suspend mode.	Resume condition on SSRX± D+ transitioning to LOW or HIGH D- transitioning to LOW or HIGH Detection of VBUS GPIO or I2C activity GPIF III interface assertion of any LVCMOS CTL pins Assertion of XRES
Deep Sleep mode	The power consumption in this mode does not exceed 120 µW. All configuration register settings and program/data RAM contents are preserved. All blocks are powered off including the USB PHY. GPIO pins maintain their configuration. Crystal oscillator is turned off Internal PLL is turned off. USB transceiver is turned off Arm® core is in the Deep Sleep state. Upon wakeup, the core restarts and runs the program stored in the program/data RAM. Power supply for the wakeup source and core power must be retained. All other power domains can be turned on/off individually.	Firmware executing on Arm® core configures the appropriate register	GPIO or I2C activity GPIF III interface assertion of LVCMOS I/Os Assertion of XRES
Core power-down mode	The power consumption in this mode does not exceed 15 µW. Core power is turned OFF along with all other power supplies to Part Number. All buffer memory, configuration registers and the program RAM do not maintain their state. After exiting this mode, the firmware will be reloaded. In this mode, all other power domains can be turned on/off individually.	Turn OFF all power domains	Reapply power supplies and assertion of XRES

2.4.3 Booting options

EZ-USB™ FX10 can load boot images from various sources, selected by the configuration of the PMODE pin. Following are the EZ-USB™ FX10 boot options:

- Boot from USB
- Boot from internal application flash

Table 2-2. EZ-USB™ FX10 booting options

PMODE_P13.0	Boot from
LOW	Internal application flash
HIGH	USB

*Note: The part CYUSB3081-FCAXI has a preprogrammed firmware on the internal application flash and do not provide any bootloader. The preprogrammed firmware is configured through data read from the external QSPI flash memory. All the other FX5/FX10 parts support the bootloader and follow the booting mode defined in **Table 2-2**.*

2.4.4 Clock system

Figure 2-1 shows that the clock system of this product line consists of the following:

- Internal main oscillator (IMO)
- Internal low-speed oscillator (ILO)
- External 24-MHz crystal oscillator (ECO)
- External clock input
- Two phase-locked loops (PLLs)
- One frequency-locked loop (FLL)

Clocks may be buffered and brought out to a pin on a smart I/O port.

2.4.5 Internal main oscillator (IMO)

The IMO is the primary source of internal clocking. It is trimmed at the factory to achieve the specified accuracy. The IMO frequency is 8 MHz and the tolerance is ±2%.

2.4.6 Internal low-speed oscillator (ILO)

The ILO is a very low-power oscillator, nominally 32 kHz, which operates in all power modes. The ILO can be calibrated against a higher accuracy clock for better accuracy.

2.5 System-level ESD

All pins of EZ-USB™ FX10 support ±2.2-kV Human Body Model (HBM) based on JESD22-A114 specification. IEC protection has to be provided at the system level.

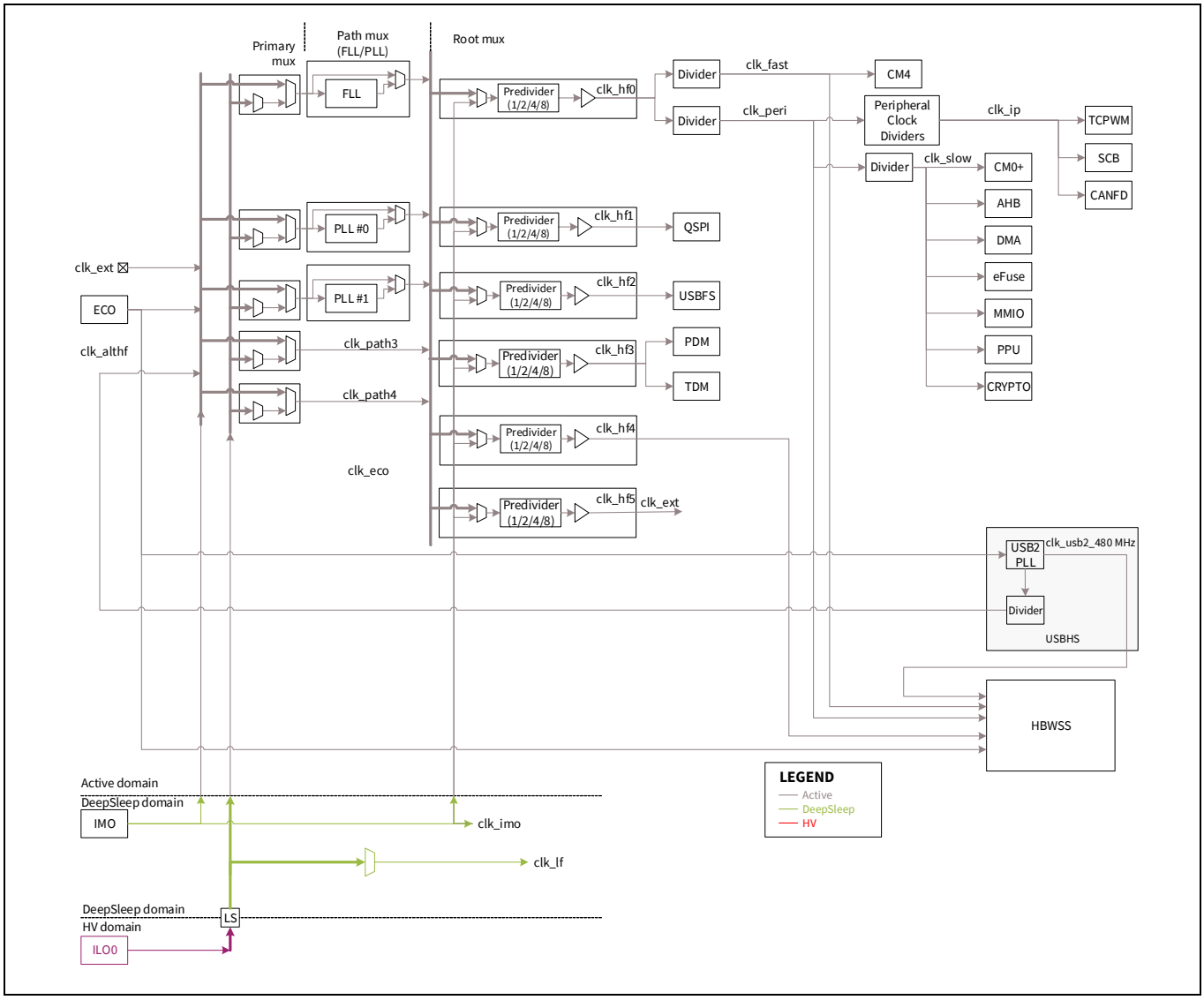


Figure 2-1. Clocking diagram

2.5.1 External crystal oscillators

When an external crystal is used, the start-up time for the clock output from the crystal driver circuit is determined by the following:

- External crystal oscillator's intrinsic characteristics
- Internal settings of the crystal driver
- Load capacitance on the XIP and XOP pins

Figure 2-2 shows the crystal model components, the load capacitors, the XIP and XOP pins of Part Number, and the crystal driver circuit. The model consists of the following parameters:

- Crystal motional capacitance (Cs)
- Crystal motional inductance (Ls)
- Crystal equivalent series resistance (ESR)
- Crystal shunt capacitance (C0)
- Load capacitors CL1 and CL2

Check the ECO specifications for the crystal values and the crystal datasheet for the load capacitor values. The ECO requires balanced external load capacitors.

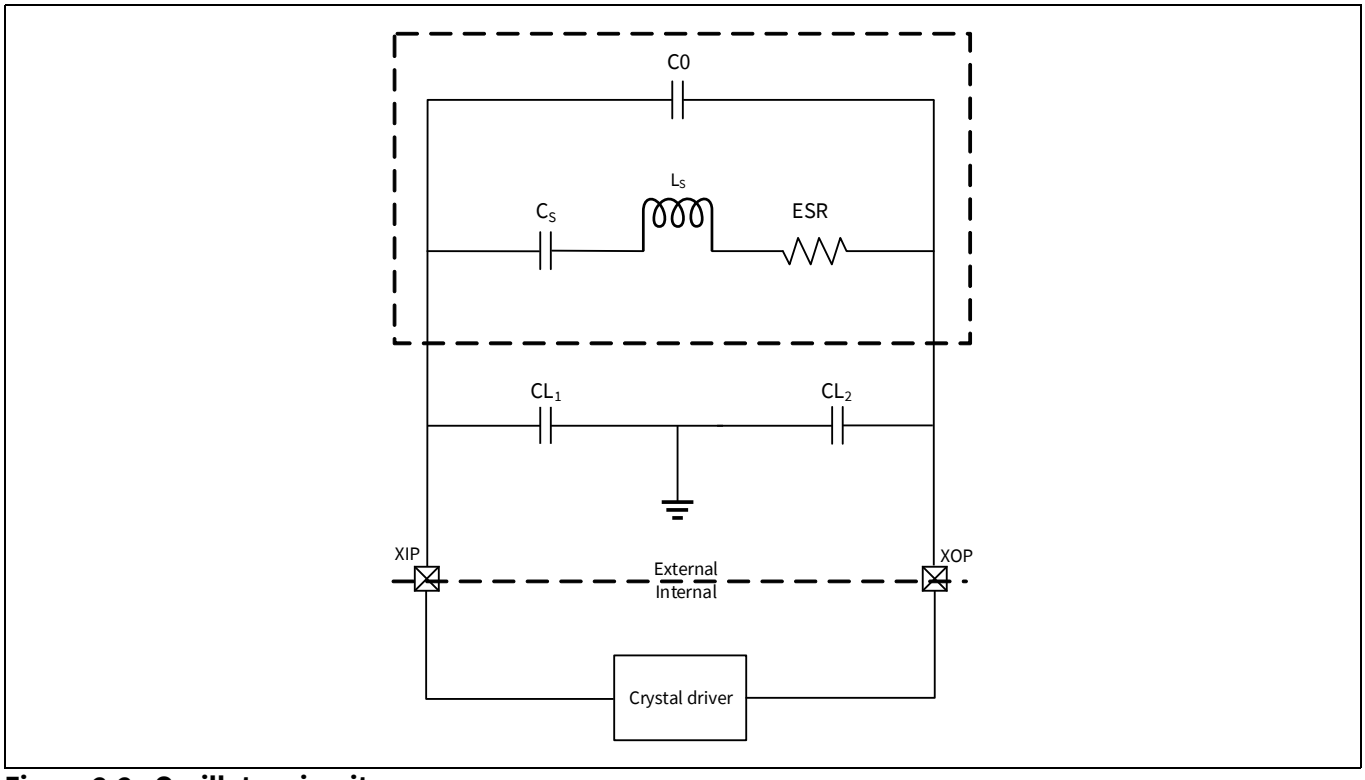


Figure 2-2. Oscillator circuits

2.5.2 Clock dividers

The following integer and fractional clock dividers are provided for peripheral use and timing purposes.

- Eight 8-bit clock dividers
- Sixteen 16-bit integer clock dividers
- Four 16.5-bit fractional clock dividers
- One 24.5-bit fractional clock divider

2.5.3 Trigger routing

EZ-USB™ FX10 contains a trigger multiplexer block. This is a collection of digital multiplexers and switches that are used for routing trigger signals between peripheral blocks and between GPIOs and peripheral blocks.

There are two types of trigger routing. Trigger multiplexers have reconfigurability in the source and destination. There are also hardwired switches called “one-to-one triggers”, which connect a specific source to a destination. The user can enable or disable the route.

2.5.4 Reset

EZ-USB™ FX10 can be reset from a variety of sources:

- The POR to hold the device in reset while the power supply ramps up to the level required for the device to function properly. POR activates automatically at power-up.
- The BOD reset to monitor the digital voltage supply VDDD and generate a reset if VDDD falls below the minimum required logic operating voltage.
- The external reset dedicated pin (XRES) to reset the device using an external source. The XRES pin is active LOW. It can be connected either to a pull-up resistor to VDDD, or to an active drive circuit, as **Figure 2-3** shows. If a pull-up resistor is used, select its value to minimize current draw when the pin is pulled LOW; 4.7 kΩ is typical.

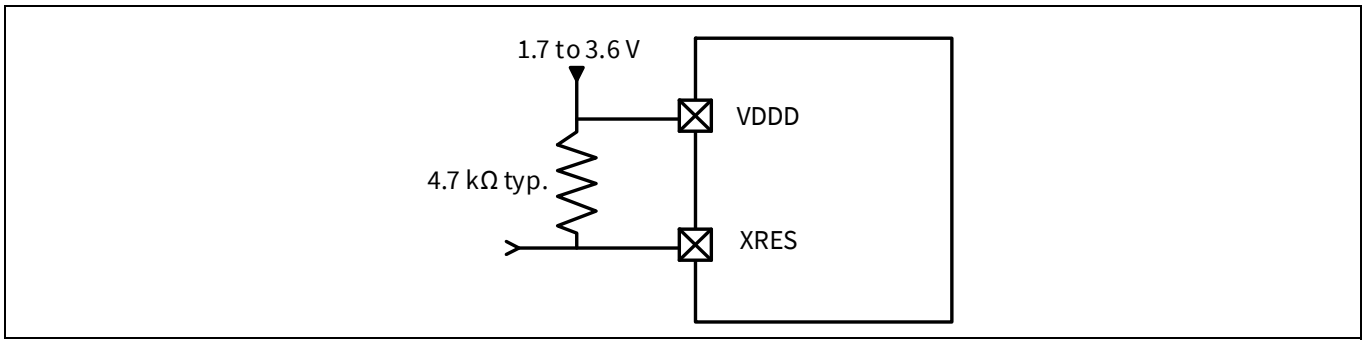


Figure 2-3. XRES connection diagram

- Software-initiated reset to reset the device on demand using the firmware.
- Logic-protection fault can trigger an interrupt or reset the device if unauthorized operating conditions occur; for example, reaching a debug breakpoint while executing privileged code.

The reset events are asynchronous and guarantee reversion to a known state. Some of the reset sources are recorded in a register, which is retained through reset and allows the software to determine the cause of the reset.

2.6 Fixed-function digital

2.6.1 Timer/counter/pulse-width modulator (TCPWM)

- The TCPWM 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, 128)
- Double buffering of compare/capture and period values
- Underflow, overflow, and capture/compare output signals
- Supports interrupt on:
  - Terminal count – Depends on the mode; typically occurs on overflow or underflow
  - Capture/compare – The count is captured to the capture register or the counter value equals the value in the compare register
- Complementary output for PWMs
- Selectable start, reload, stop, count, and capture event signals for each TCPWM; with the rising edge, falling edge, both edges, and level trigger options. The TCPWM has a Kill input to force outputs to a predetermined state.

In this device, there are 8×32-bit TCPWMs.

2.6.2 Serial Communication Block (SCB)

EZ-USB™ FX10 has seven SCBs. Each SCB can be configured in any of the following modes:

- **I²C mode:** The SCB can implement a full multi-master and slave interface (it is capable of multi-master arbitration). This block can operate at speeds of up to 1 Mbps (Fast Mode Plus). It also supports EZI2C, which creates a mailbox address range and effectively reduces I2C communication to reading from and writing to an array in memory. The SCB supports a 256-byte FIFO for receiving and transmitting.
  The I2C peripheral is compatible with I2C standard-mode, Fast Mode, and Fast Mode Plus devices as defined in the I2C-bus specification and user manual (UM1024). The I2C bus I/O is implemented with GPIO in open-drain modes.
- **UART mode:** This is a full-featured UART operating at up to 8 Mbps. It supports automotive single-wire interface (LIN), infrared interface (IrDA), and SmartCard (ISO 7816) protocols, all of which are minor variants of the basic UART protocol. In addition, it supports the 9-bit multiprocessor mode that allows the addressing of peripherals connected over common RX and TX lines. Common UART functions such as parity error, break detect, and frame error are supported. A 256-byte FIFO allows much greater CPU service latencies to be tolerated.
- **SPI mode:** The SPI mode supports full Motorola SPI, TI secure simple pairing (SSP) (which essentially adds a start pulse that is used to synchronize SPI Codecs), and National Microwire (a half-duplex form of SPI). The SPI block supports an EZSPI mode in which the data interchange is reduced to reading and writing an array in memory. The SPI interface operates with a 25 MHz clock.

2.6.3 USB full-speed device interface

This product line incorporates a full-speed USB device interface. The device can have up to eight endpoints. A 512-byte SRAM buffer is provided and DMA is supported.

Note that if the USB-FS pins are not used, connect VDDUSB to ground and leave the USBDP and USBDM pins unconnected.

2.6.4 Quad-SPI/serial memory interface (SMIF)

A serial memory interface is provided, running at up to 80 MHz. It supports single, dual, quad, dual-quad, and octal SPI configurations and supports up to four external memory devices. It supports two modes of operation:

- Memory-mapped I/O (MMIO), a command mode interface that provides data access via registers and FIFOs
- Execute In Place (XIP), in which AHB reads and writes are directly translated to SPI read and write transfers.

In XIP mode, the external memory is mapped into the Part Number internal address space, enabling code execution directly from the external memory. To improve performance, a 4-KB cache is included. XIP mode also supports AES-128 on-the-fly encryption and decryption, enabling secured storage and access of code and data in the external memory.

2.7 GPIO

EZ-USB™ FX10 has 48 GPIOs including the pins used for low-speed peripheral interface, which implement the following:

- Eight drive modes:
  - Input only
  - 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
  - In strong drive mode, there are four selectable drive strengths for IOL and IOH current drive
- Input threshold select (CMOS or LVTTL)
- Selectable slew rates for dV/dt-related noise control to improve EMI

The pins are organized into logical entities called ports, which are up to eight pins wide. Data output and pin state register store, respectively, the values to be driven on the pins and the input states of the pins.

Every pin can generate an interrupt if enabled, and each port has an interrupt request (IRQ) associated with it.

The port's ten pins are capable of overvoltage-tolerant (OVT) operation, where the input voltage may be higher than VDDD. OVT pins are commonly used with I2C to allow powering the chip off while maintaining a physical connection to an operating I2C bus without affecting its functionality.

The GPIO pins can be ganged to source or sink higher values of current. GPIO pins, including OVT pins, may not be pulled up higher than the absolute maximum; see Electrical specifications section in the device datasheet.

2.8 Special-function peripherals

2.8.1 Audio subsystem

This subsystem consists of the following hardware blocks:

- Two Inter-IC sound (I2S) interfaces
- Two PDM to PCM decoder channels

Each of the I2S interfaces implements two independent hardware FIFO buffers – TX and RX, which can operate in master or slave mode. The following features are supported:

- Multiple data formats – I2S, left-justified, time division multiplexed (TDM) mode A, and TDM mode B
- Programmable channel/word lengths – 8/16/18/20/24/32 bits
- Internal/external clock operation up to 192 ksps
- Interrupt mask events – trigger, not empty, full, overflow, underflow, and watchdog
- Configurable FIFO trigger level with DMA support

The I2S interface is commonly used to connect with audio codecs, simple DACs, and digital microphones.

The PDM-to-PCM decoder implements a single hardware Rx FIFO that decodes a stereo or mono 1-bit PDM input stream to PCM data output. The following features are supported:

- Programmable data output word length – 16/18/20/24 bits
- Programmable gain amplifier (PGA) for volume control from –12 dB to +10.5 dB in 1.5 dB steps
- Configurable PDM clock generation. Range from 384 kHz to 3.072 MHz
- Droop correction and configurable decimation rate for sampling; up to 48 ksps
- Programmable high-pass filter gain
- Interrupt mask events – not empty, overflow, trigger, and underflow
- Configurable FIFO trigger level with DMA support

The PDM-to-PCM decoder is commonly used to connect to digital PDM microphones. Up to two microphones can be connected to the same PDM data line. For more details, see **“Digital subsystem”** on page 438.

3 Document construction

This document includes the following main chapters:

- **"USB interface"** on page 37
- **"CPU subsystem"** on page 61
- **"System resources subsystem (SRSS)"** on page 274
- **"High-bandwidth subsystem (HBWSS)"** on page 355
- **"Digital subsystem"** on page 438

3.1 Major sections

For ease of use, information is organized into sections and chapters that are divided according to device functionality.

- Section – Presents the top-level architecture, how to get started, and conventions and overview information of the product.
- Chapter – Presents the chapters specific to an individual aspect of the section topic. These are the detailed implementation and information for some aspect of the integrated circuit.
- Registers reference manual – Supplies all device register details summarized in the reference manual. This is an additional document.

3.2 Documentation conventions

This document uses only three distinguishing font types, besides those found in the headings.

- The first is the use of *italics* when referencing a document title or file name
- The third is the use of `Courier New` font, distinguishing code examples

3.2.1 Register conventions

The register conventions are detailed in the EZ-USB™ FX10 registers reference manual.

3.2.2 Numeric naming

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

3.2.3 Unit of measure

This table lists the unit of measure used in this document.

Table 3-1. Unit of measure

Abbreviation	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 (1000)
kΩ	kilohms
MHz	megahertz
MΩ	megaohms
µA	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
V	volts

3.2.4 Acronyms and abbreviations

This table lists the acronyms used in this document.

Table 3-2. Acronyms and abbreviations

Acronym	Description
A/D	analog to digital
ABS	absolute
AES	Advanced Encryption Standard
AHB	AMBA (advanced microcontroller bus architecture) high-performance bus, Arm® data transfer bus
Arm®	Advanced RISC Machine, a CPU architecture
BD	buffer descriptors
BOD	brownout detection
CAN FD	controller area network with flexible data rate
CBS	credit-based shaping
CFI	canonical format indicator
CMOS	complementary metal-oxide-semiconductor
CPHA	(SPI) clock phase
CPOL	(SPI) clock polarity
CPU	central processing unit
CPUSS	CPU subsystem
CRC	cyclic redundancy check, an error-checking protocol
CSV	clock supervisor
DDR	double data rate (also see SDR)
DES	data encryption standard
DMA	direct memory access
DW	data wire, same as P-DMA
ECC	error correcting code
ECO	external crystal oscillator
EEE	Energy Efficient Ethernet (IEEE Std 802.3az)
EOF	end of frame
ETM	embedded trace macrocell
FCS	frame check sequence
FIFO	first in first out
FLL	frequency locked loop
FPU	floating point unit
GHS	Green Hills tool chain with IDE
GPIO	general-purpose input/output
HSIOM	high-speed I/O matrix
HSM	hardware security module
IF	interface

Table 3-2. Acronyms and abbreviations (continued)

Acronym	Description
I/O	input/output
IP	Internet protocol
IPG	inter-packet gap
I^2C	Inter-Integrated Circuit, a communications protocol
ILO	internal low-speed oscillator
IMO	internal main oscillator
IPC	inter-processor communication
IrDA	infrared interface
IRQ	interrupt request
JTAG	Joint Test Action Group
LAN	local area network (IEEE Std 802)
LLDP	link layer discovery protocol (IEEE Std 802.1AB)
LPI	low-power idle (IEEE Std 802.3az)
LVD	low-voltage detection
MAC	media access control (IEEE Std 802)
MCU	microcontroller unit
MCWDT	multi-counter watchdog timer
M-DMA	memory-direct memory access
MDC	management data clock
MII	media independent interface
MISO	master-in slave-out
MMIO	memory mapped I/O
MOSI	master-out slave-in
MPU	memory protection unit
NSP	non-standard preamble
NVIC	nested vectored interrupt controller
OTA	over-the-air programming
OTP	one-time programmable
OVD	overvoltage detection
P-DMA	peripheral-direct memory access same as DW
PCS	physical coding sublayer
PFC	priority-based flow control (IEEE Std 802.1Qbb)
PLL	phase-locked loop
PHY	physical sublayer
POR	power-on reset
PPB	private peripheral bus
PPPoE	point-to-point protocol over ethernet
PPU	peripheral protection unit

Table 3-2. Acronyms and abbreviations (continued)

Acronym	Description
PRNG	Pseudo-Random Number Generator
PTP	precision time protocol (IEEE Std 1588)
PWM	pulse-width modulation
RAM	random access memory
RISC	reduced-instruction-set computing
ROM	read-only memory
RTC	real-time clock
RX	reception
SAR	successive approximation register
SCB	serial communication block
SCL	I^2C serial clock
SDA	I^2C serial data
SDR	single data rate (also see DDR)
SECDED	single error correction double error detection
SerDes	serializer/deserializer
SHA	secure hash algorithm
SHE	secure hardware extension
SFD	start of frame delimiter
SGMII	serial Gigabit media independent interface
SMIF	serial memory interface
SMPU	shared memory protection unit
SNAP	subnetwork access protocol
SOF	start of frame
SPI	serial peripheral interface, a communications protocol
SRAM	static random-access memory
SWD	single wire debug
Tbit	bit period
TCP	transfer control protocol
TCPWM	timer/counter pulse-width modulator
TTL	transistor-transistor logic
TRNG	True Random Number Generator
TS	timestamp
TSU	timestamp unit
TX	transmission
UART	Universal Asynchronous Transmitter Receiver, a communications protocol
UDP	user datagram protocol
VLAN	Virtual LAN (IEEE Std 802.1Q)
WCO	watch crystal oscillator

Table 3-2. Acronyms and abbreviations (continued)

Acronym	Description
WDT	watchdog timer reset
XIP	execute-in-place
XRES_L	external reset I/O pin (active LOW)
XTAL	crystal

Section B: USB interface

This section encompasses the following chapter:

- **"USB interface"** on page 38

Top level architecture

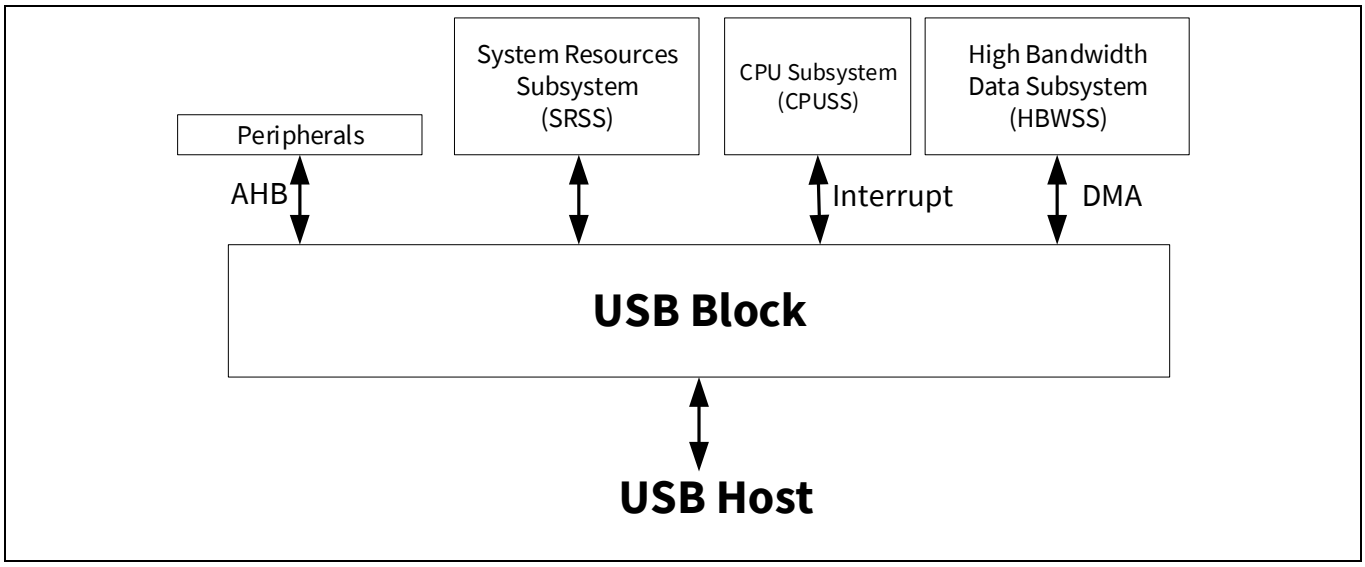


Figure B-1. USB interface block diagram

4 USB interface

4.1 Supported modes

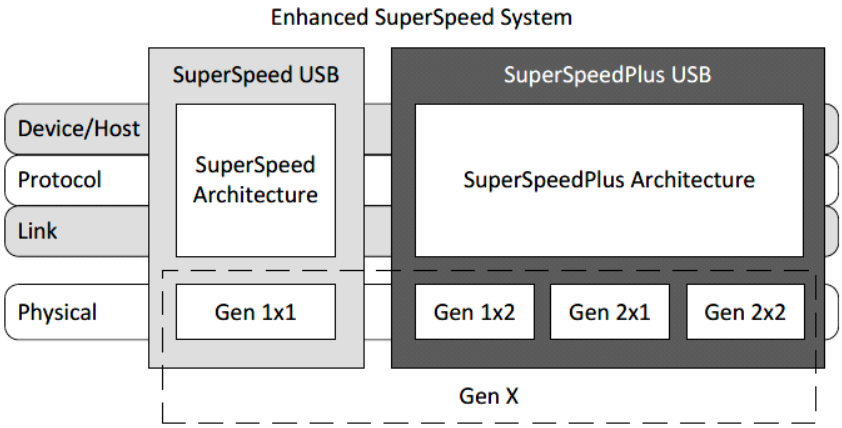
The USB interface supports the following modes:

- USB 3.2 Gen 2 (10 Gbps)
- USB 3.2 Gen 1 (5 Gbps)
- USB 2.0 Mode (480 Mbps)

4.2 Introduction to USB 3.2

USB 3.2 is similar to earlier versions of USB in that it is a cable bus supporting data exchange between a host computer and a wide range of simultaneously accessible peripherals. The attached peripherals share bandwidth through a host-scheduled protocol. The bus allows peripherals to be attached, configured, used, and detached while the host and other peripherals are in operation.

USB 3.2 is a dual-bus architecture that provides backward compatibility with USB 2.0. One bus is a USB 2.0 bus and the other is an Enhanced USB 5 Gbps bus. USB 3.2 specifically adds dual-lane support.



*Note: This image is from USB 3.2 specification. For more information, see **USB** webpage.*

Figure 4-1. USB 3.2 bus architecture

Table 4-1. Comparing enhanced USB 5 Gbps Bus to USB 2.0 Bus

Characteristics	USB 3.2	USB 2.0
Data rate	Gen 1 (5.0 Gbps), Gen 1x2 and Gen 2x1 (10 Gbps), Gen 2x2 (20 Gbps)	Low-speed (1.5 Mbps), full-speed (12 Mbps), and high-speed (480 Mbps)
Data interface	Dual-simplex, four-wire differential signaling separate from USB 2.0 signaling Simultaneous bi-directional data flows	Half-duplex two-wire differential signaling Unidirectional data flow with negotiated directional bus transitions
Cable signal count	Four for USB 3.2 data path and two for USB 2.0 data path	Two: Two for low-speed/full-speed/highspeed (USB 2.0) data path
Bus transaction protocol	Host directed, asynchronous traffic flow Packet traffic is explicitly routed	Host directed, polled traffic flow Packet traffic is broadcast to all devices
Power management	Multi-level link power management supporting idle, sleep, and suspend states. Link-, Device-, and Function-level power management.	Port-level suspend with two levels of entry/exit latency device-level power management
Bus power	For one lane operation: Support for LOW (150 mA)/HIGH (900 mA) bus-powered devices with lower power limits for un-configured and suspended devices. For two lane operation: Support for LOW (250 mA)/high (1,500 mA) bus-powered devices with lower power limits for un-configured and suspended devices.	Support for LOW (100 mA)/HIGH (500 mA) bus-powered devices with lower power limits for un-configured and suspended devices
Port state	Port hardware detects connect events and brings the port into operational state ready for USB 3.2 data communication.	Port hardware detects connect events. System software uses port commands to transition the port into an enabled state (i.e., can do USB data communication flows).
Data transfer types	USB 2.0 types with USB 3.2 constraints. Bulk has streams capability.	Four data transfer types: control, bulk, Interrupt, and Isochronous

4.3 Features

- USB 3.2 device support – 32 endpoints
  - USB 3.2 Gen 2 (10 Gbps)
  - USB 3.2 Gen 1 (5 Gbps)
  - USB 2.0 Mode (480 Mbps)
- USB protocol layer features
  - Default control endpoint 0 support
  - Function endpoints supported up to 15 IN and up to 15 OUT
  - Each function endpoint can be configured to BULK, ISO or INTERRUPT
  - Supports BULK end point streaming
  - Supports simultaneous transactions across multiple IN endpoints
  - Pipelined isochronous IN
  - Transaction reordering
  - Precise time measurement (PTM) support
  - Configurable burst size for each function endpoint
  - Supports standard PIPE 5.1 interface without message bus compliant (for USB 3.2) to interface to embedded PHY
- USB link layer features
  - Timers are configurable
  - Low power state support
  - Compliance pattern, loopback and debug
  - Error and event monitoring through interrupts
  - Lane bonding and de-skewing in x2 configuration
- Special USB events are decoded and communicated to the processor through the register interface and interrupts

4.4 Block diagram

Figure 4-2 shows the top-level block diagram of the EZ-USB™ FX10's USB block. This block consists of USB3.2 device, USB3.2 PHYs, EPM Logic, and DMA logic. The data received from or sent out to the USB interface is from the End-point memory (EPM). The other side of the EPM is connected to the DMA logic.

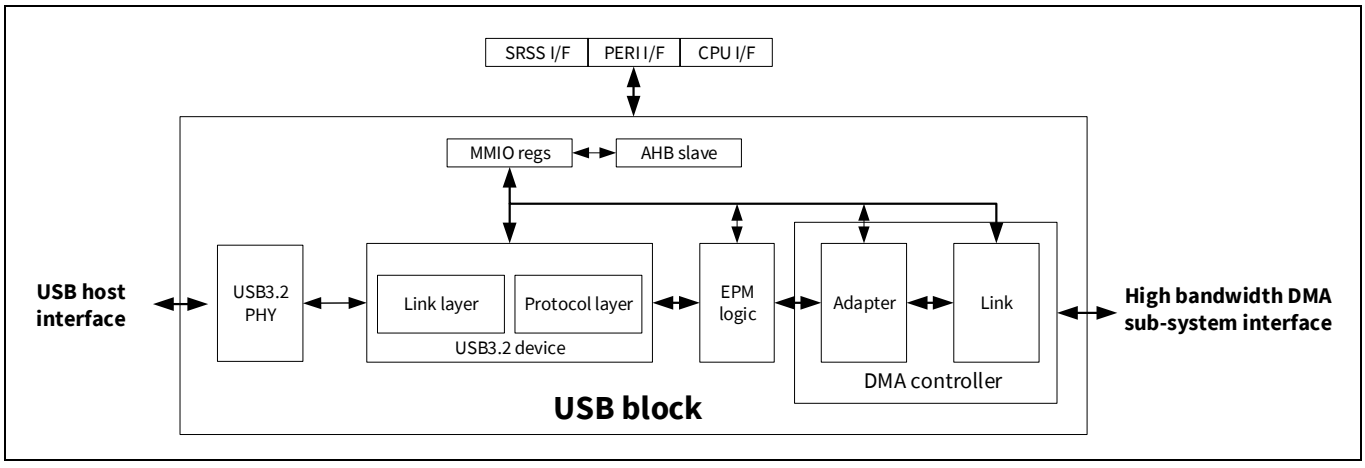


Figure 4-2. USB top level block diagram

4.5 Overview of USB 3.2 interface block

4.5.1 USB3.2 PHY

The PHY supports 10 Gbps (USB 3.2 Gen 2x1) and 5 Gbps (USB 3.2 Gen 1) speed for USB mode. The PHY implements a SERDES architecture. The sub-system communicates to the controller through a 40-bit PMA interface compliant to PIPE specification.

4.5.2 USB3.2 device

The USB3.2 device is implemented in two sub-blocks as shown in **Figure 4-3**.

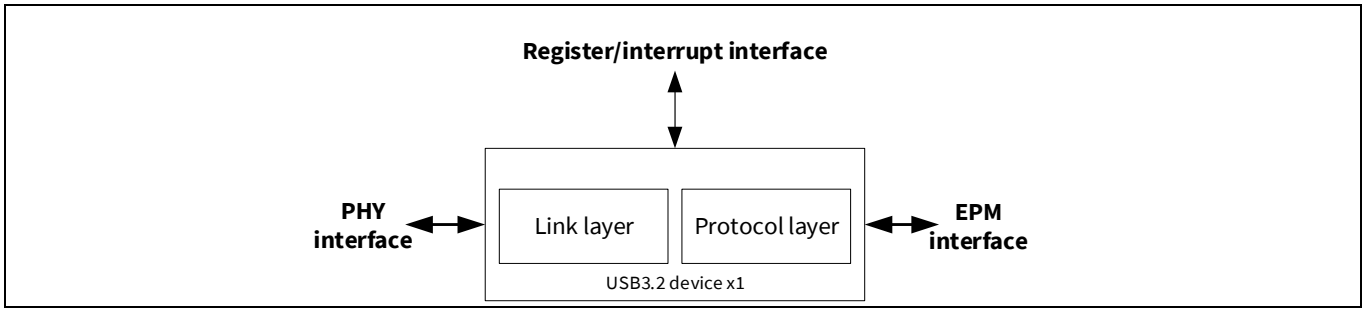


Figure 4-3. USB3.2 Gen 2 device

4.5.2.1 Link layer

The link layer block interacts with protocol layer and USB 3.2 PHY blocks. This block processes the packets and handles point-to-point transactions as well as power management. The Link Layer in a USB 3.2 system is responsible for managing the communication between two USB devices. For any given segment of the USB bus, both the downstream port of a host or a hub and the upstream port of a device have Link layer handler.

The Link Layer duties include:

- USB connection establishment and detection
- Power state management
- Packet processing and transmission between the Protocol Layer and the USB Physical Layer
- USB Link test mode management

The USB 3.2 bus is a dual simplex bus, with separate transmit and receive channels that carry data simultaneously. Since the data transmitted can be data packet and control information, the receive and transmit logic blocks need to work together, especially for handling the control flow.

In addition to the standard Protocol-level command structure, in the USB 3.2 bus there is additional, lower-level Link Commands operating beneath that which handle the link management and flow control

The Link Layer Block sits between the USB 3.2 PHY and the Protocol Layer Block. The Link Layer is responsible for direct control of the PHY and also for overall communication with the USB Device on the other end of the cable. The PHY Interface is compliant with the PIPE 5.1 specification (without message bus). The basic data path is 32 bits wide, which will mean that the clock frequency of the interface is 125 MHz–312.5 MHz. The transmit and receive data buses include per-byte indicators of whether the value is a data byte or a special character.

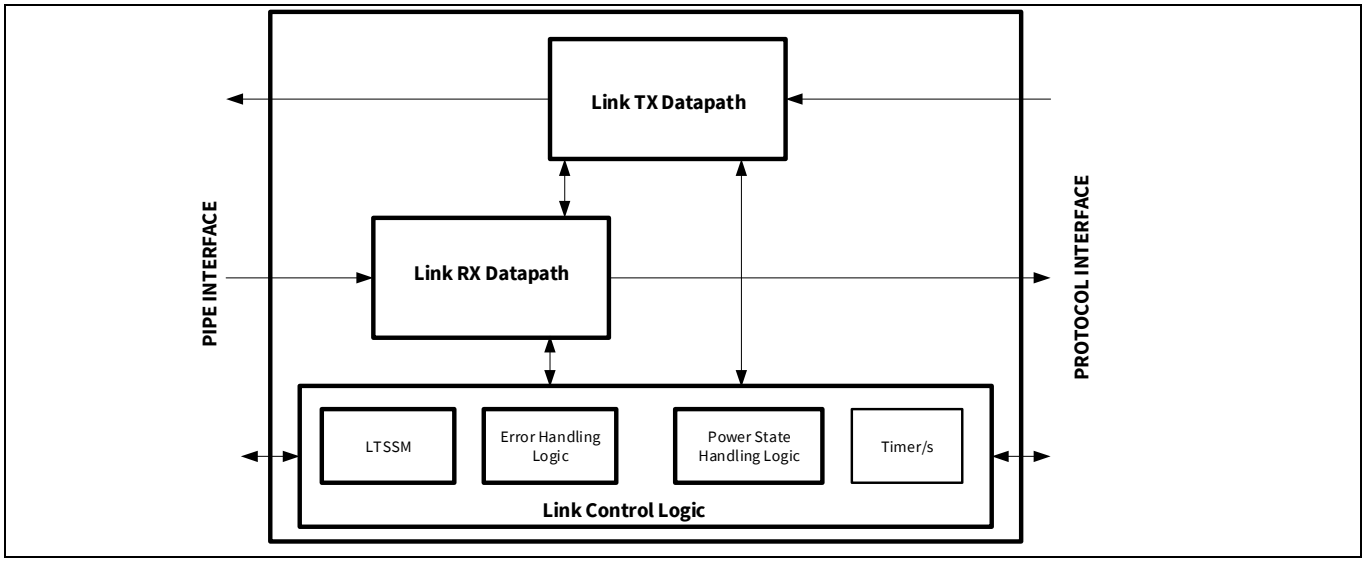


Figure 4-4. Link layer block diagram

All USB packets that goes to the protocol layer goes through the link layer. For each of these packets, the link layer will pass the first three DWORDs of the header packet between the PHY and the protocol block, and then process the fourth DWORD which contains the CRC-16 value and the link control word.

For received packets, the three DWORDs will be buffered for the Protocol Block if the CRC-16 and the header sequence is correct. The protocol layer block should be able to accept the data as it comes from the link layer Block.

For transmission, the link layer will create the CRC-16 value to be transmitted as well as the link control word. The link layer will also add or detect the packet starting ordered sets. The protocol layer block will need to provide the data to be transmitted to the link layer as needed.

4.5.2.1.1 Link training and status state machine (LTSSM)

The LTSSM block represents the overall state of the USB 3.2 link layer, including its operational, reset, training, and power-saving states. The link layer block will handle the state transitions by itself automatically.

4.5.2.1.2 Error detection logic

The error detection logic block is responsible for collecting all the Link Layer errors in the RX and the TX path and communicating them to the LTSSM.

- Header packet errors
- Missing of header packet:
  - Invalid header packet due to CRC errors: Detected in RX link header block
  - Mismatch of a RX header sequence number: Detected in RX link header block
- Packet framing error: Corrupted HPSTART: Detected in the RX link header block
- Header packet error: Detected in RX Link Header block
- RX header sequence number error: Detected in the RX link header block
- Link command error: RX link command logic block
- ACK TX header sequence number error: Detected in the ACK TX header sequence number block
- Header sequence number advertisement error: Detected in RX link header block (or PENDING_HP_TIMER timeout)
- Rx header buffer credit advertisement error: Detected in RX link header block (or CREDIT_HP_TIMER timeout)
- Training sequence error: Detected in ordered set checker
- 8b/10b error: Detected in USB 3.2 PHY
- Gen 2x1 block header errors: There are two types of block header errors, a correctable single bit block header error, and a detectable but not correctable two-bit block header error.

4.5.2.1.3 Power state handling logic

The power state handling logic implements three timers for link power management:

- PM_LC_TIMER is used when the port is requesting a transition to a low power state. If this timer times-out before a LAU or LXU link command is received, the port transitions to the recovery state.
- PM_ENTRY_TIMER is used when the port has received a request to transition to a low-power state. If this timer times-out before an LPMA or TS1 training sequence is received, the port transitions to the Recovery state.
- Ux_EXIT_TIMER is used to ensure that the port transitions from the U1 and U2 low-power states to the Active U0 state before losing any header packets. If it does not do so, the port transitions to the SS.Inactive state.

4.5.2.2 Protocol layer

The protocol layer block interacts with EPM and Link Layer blocks. It processes the data packets from the Link Layer. It is responsible for packet and handshake processing per USB 3.2 spec.

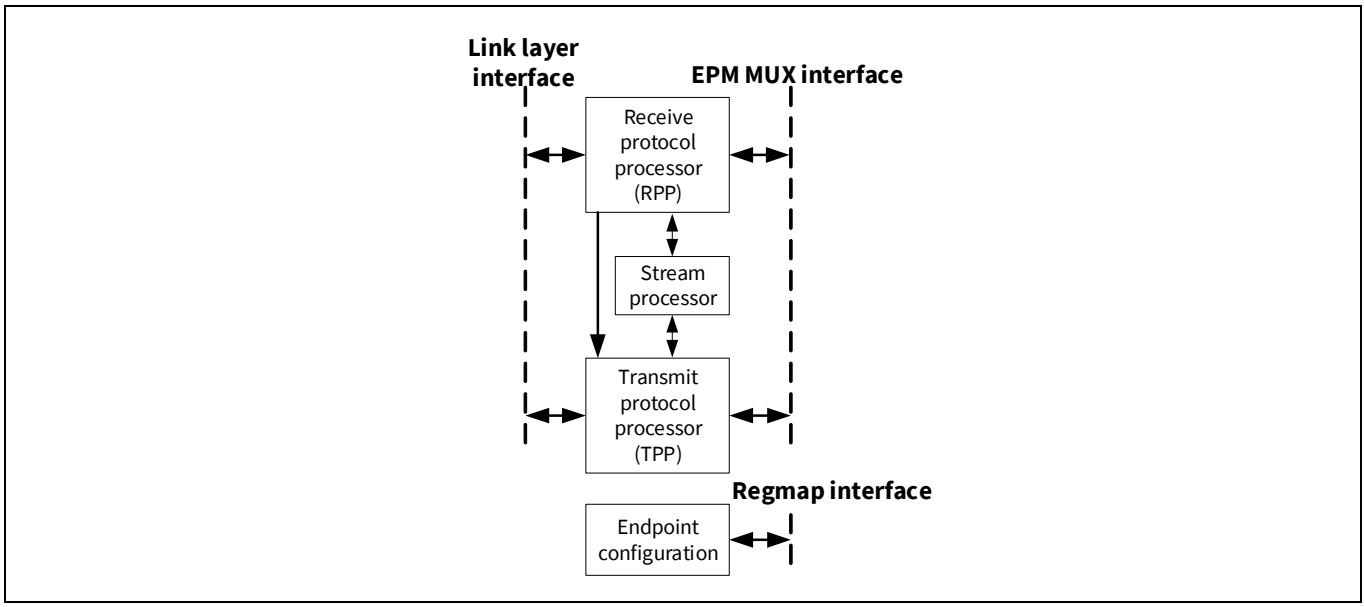


Figure 4-5. Protocol Layer block

It is responsible to move and control the flow of data between EPM and host. It handles Transaction Packet (TP), Data Packet (DP), Link Management Packet (LMP) and Isochronous Timestamp Packet (ITP). It supports Control, Interrupt, Isochronous, and Bulk (with stream).

Link management packets contain parameters that control the link layer block. The protocol block decodes these packets and passes the parameters back to the link layer block.

Table 4-2. LMP parameters

Parameter	Description
Force_LinkPM_Accept	Force acceptance of LGO_Ux Link commands
U2 Inactivity Timeout[7:0]	Defines the timeout for going from U1 to U2
Vendor Test[8:0]	Vendor-specific test
Vendor Test Data[63:0]	Vendor-specific test data

In addition, the protocol layer will indicate to the link layer which power states to which the link layer can transition. Force_LinkPM_Accept will override this.

4.5.3 Stream support for USB 3.2

In case of USB 3.2, a stream logic mapper is used to map the active stream and the End-point number to the appropriate socket in the DMA. This allows sixteen active streams at a time in the OUT direction. Similarly, it also allows sixteen IN streams at a time. These streams can be mapped to a single EP or multiple EPs. In case the 17th stream needs to be mapped, it will be done under the Firmware control. The Hardware upon seeing the StreamID that does not match one of the current active 16 StreamIDs, raises the Interrupt to the CPU and sends the NRDY to the Host. The Firmware then requests the HW to evict one of the currently active StreamID. The HW ensures that the current StreamID can be evicted by sending the NRDY for that StreamID to the host. Once it establishes that Host and the Device states are in sync, it evicts the active StreamID and replaces it with the new StreamID. It then sets the StreamID switch bit that indicates to the firmware the new StreamID is ready for transfer and the FW can switch the descriptors in the adapter. Upon updating the Descriptors, the Firmware sets an enable bit to indicate to the HW that the StreamID socket is ready. The controller HW then sends the ERDY to the Host for that StreamID.

4.5.4 USB precision time measurement (PTM) support

4.5.4.1 Isochronous timestamp packet (ITP)

The isochronous timestamp packets (ITPs) are the main mechanism used to maintain a shared sense of time across all devices connected to a host controller.

The host controller multicasts ITPs to all devices in the U0 link state about once in 125 µs. The host shall send ITPs within tTimestampWindow (0 to 8 µs) of the start of the bus interval boundary.

ITP sending shall be started within tIsochronousTimestampStart (250 µs) of the link entering U0 from polling or U3. Devices shall ignore any ITPs received with the Delayed (DL) bit set in the LCW (due to the packet being delayed by a hub).

Fields in the ITP:

- Isochronous timestamp: 27-bit field which is divided into:
  - Bus interval counter: 14-bit field which provides the current value of the 125 s counter. Rolls over to 0 after reaching max. value of 0x3FFF (2.048 s).
  - Delta: 13-bit field specifying the delay between bus interval boundary and the start of the ITP in tIsochTimestampGranularity (8 HSUI = 16.667 ns) units
- Bus interval adjustment control: 7-bit field which provides the address of the device which controls the bus interval adjustment (deprecated in USB 3.2)
- Correction: 14-bit field specifying the negative delay accumulated by the ITP while passing through PTM supported hubs, in tIsochTimestampGranularity (16.667 ns) units

4.5.4.2 Precision time measurement (PTM)

While the ITPs allow multiple devices connected to the same host to have a common sense of time-based on the host specified timestamp, there can be deviations due to different propagation delays taken by the ITP to reach the devices. Precision Time Measurement (PTM) provides a mechanism for the device to estimate the link and hub propagation delays to compensate for this deviation.

PTM consists of two parts:

- Link delay measurement (LDM)
- Hub delay measurement (HDM)

4.5.4.3 PTM bus interval boundary counters

The bus interval boundary is defined using a pair of counters which use a format similar to the 27-bit isochronous timestamp field from the ITP. These counters work on the basis of the PTM clock which has a period of tIsochTimestampGranularity (16.667 ns).

- 13-bit PTM delta counter is incremented every PTM clock cycle to measure the delay from the bus interval boundary. This is a modulo-7500 counter so that the counter wraps to 0 at the end of a 125 µs bus interval.
- 14-bit PTM bus interval counter is incremented every time the PTM delta counter wraps. This counter wraps after reaching the maximum value of 0x3FFF.

Synchronization of the PTM clock to the bus interval on the device side is implementation dependent.

4.5.4.4 LDM protocol

A USB device which supports PTM will initiate LDM requests to the downstream hub or roothub port that it is connected to. The downstream port will send an LDM response once the request has been received.

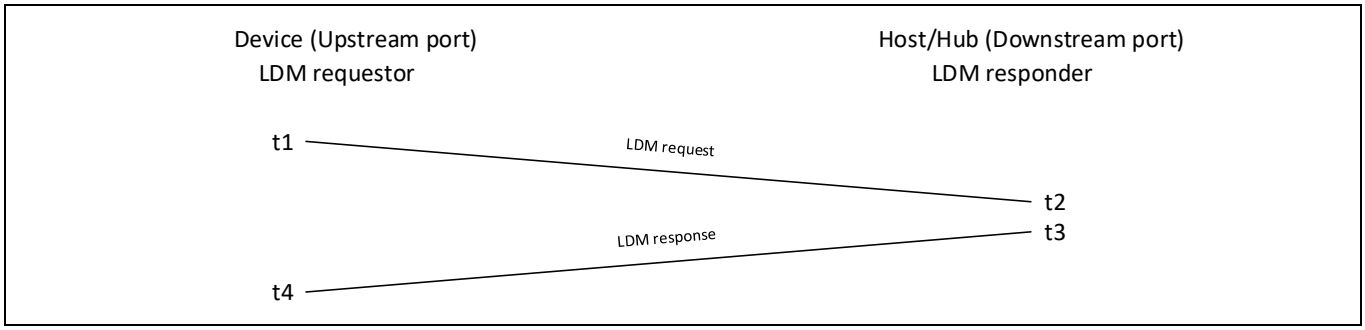


Figure 4-6. Link delay calculation

As shown in **Figure 4-6**, the USB device is expected to take timestamp (t1) at the instance where the LDM request is being sent and timestamp (t4) at the instance where the LDM response has been received. Similarly, the downstream port responding to the LDM request will take timestamps t2 and t3 representing when the LDM request was received and the LDM response is being sent. The time difference (t3 - t2) will be provided through the Response Delay field in the LDM response LMP.

The link delay can be calculated by the device as:

((t4 - t1) - Response Delay) / 2

This delay has to be adjusted by the difference between the actual and nominal LMP transmission time.

The nominal LMP transmission time is *tLMPTransmissionDelay* or 40 ns.

The actual LMP transmission time depends on the time of USB 3 Link as shown in **Table 4-3**.

Table 4-3. Link management packet (LMP) transmission times

USB3 link speed	LMP transmission time
Gen1x1	40 ns
Gen2x1	16.5 ns

Link delay for the connection from device to corresponding downstream port = *((t4 - t1) - Response Delay) / 2) + LMP Transmission Time - 40 ns*

4.5.4.5 PTM bus interval boundary calculation

The USB devices need to calculate the Bus Interval Boundary when an ITP is received.

The Bus Interval Counter(RxITP), Delta(RxITP) and Correction(RxITP) values are taken from the ITP packet that is received.

PTM Delta Counter is calculated as follows from these values and the Link Delay measured previously:

PTM Delta Counter = MOD((ISOCH_DELAY + LDM Link Delay + Delta(RxITP) - Correction(RxITP)) / 7500)

where,

ISOCH_DELAY = Delay between the transmission of a packet from the host and its reception by the device. This delay is provided by the host through the SET_ISOCH_DELAY control request.

PTM bus interval counter = Bus Interval Counter(RxITP) + FLOOR((LDM Link Delay + Delta(RxITP)) / 7500)

4.5.4.6 Timestamp calculation on EZ-USB™ FX10

The above calculations can be performed on the EZ-USB™ FX10 device to calculate the timestamp corresponding to the instant when an ITP is received by the device.

The USB3.2 block provides the following capabilities to enable these calculations:

- Provides all the fields from the ITP packets that are received (RxITP)
- LDM LMP exchange support:
  - Send an LDM Request LMP on firmware command
  - Timestamp of the instant when the LDM request LMP is transmitted by the device
  - Timestamp of the instant when the LDM response LMP is received by the device
- Fields from the LDM response LMP that is received

With the above information, PTM Bus Interval Counter and PTM Delta Counter values corresponding to the time when the ITP was received can be calculated. If these calculations must be performed by firmware, consider the time delta between the ITP being received and the time at which the delay calculations are being made by the firmware (from an ISR).

This can be done if a counter exists which is continuously running at 60 MHz
(period = tIsochTimestampGranularity) and a register which stores the counter value at the time that the ITP was received.

All the timestamps can be provided using this 60 MHz counter. The USB block has dedicated registers which provide snapshots of the counter value at the following times:

- When an ITP is received
- When an LDM request is sent (t1)
- When an LDM response is received (t4)

Firmware can use the above timestamp values and the ITP information to calculate the PTM bus interval boundary at any instant of time.

Since all calculations are being done on the basis of the ITPs sent from the host controllers, multiple EZ-USB™ FX10 devices connected to the same host will be synchronized to each other.

4.5.4.7 Using the PTM timestamps

4.5.4.7.1 Synchronous trigger of image capture

EZ-USB™ FX10 needs to send a signal (GPIO, I2C command etc.) at a desired time to trigger a frame capture by the image sensor.

We can define a command from the host controller to the EZ-USB™ FX10 specifying the timestamp at which the image capture should be triggered. Firmware can use the current value of the PTM interval counters to identify the delay after which the image capture has to be triggered. The TCPWM timer can then be used to trigger the GPIO or a DMA transfer from SCB to activate the image sensor.

4.5.4.7.2 Timestamp in video metadata

The UVC data headers include optional timestamp fields which can be filled with the image source timestamp at the time when the video frame was captured. If we can have an interrupt raised when the frame is captured by the sensor, firmware can calculate the corresponding timestamp and fill this value into a header register maintained in the LVDS block. The LVDS block can then copy the timestamp into the UVC headers which are interleaved with the video data.

4.5.5 Endpoint memory (EPM)

The Endpoint memory (EPM) stores the data packets that are sent on the USB Bus. Each USB Device can have a maximum of sixteen endpoints, with endpoint 0 being reserved as a command-and-control pipe from the host to the device, leaving fifteen endpoints for main data transfer. It also includes DMA through the Interconnect fabric to efficiently transfer data to and from this block.

4.5.6 DMA controller

The USB Block has a DMA controller to handle data transfer between USB block and the system memory. Two separate paths, one for reading data from system memory to USB Bus (Egress Path) and another to write USB Data to system memory (ingress path), are required to support the 10 Gbps bandwidth for USB3.2 in each direction.

The Ingress Path writes data from USB to system memory (or 512 × 64 (4 KB) FIFOs). The data after it has been written in the FIFO has to be committed by the USB3.2 device. Once the data is committed to the FIFO, it moves to the system memory through the DMA.

The DMA adapter will keep track of the space available in the system memory. It will provide information in advance to the controller logic to send NRDY to the Host once the system memory does not have space to accept more data.

The Egress Path reads data from system memory to USB. A retry buffer of 16 KB is used to support burst up to the spec max limit of 16. The Device can send the max burst of 16 packets before getting the ACK back from the Host for the first packet. The Retry buffer stores the packets from the Burst. Once the Ack for the burst is received, it clears the Retry buffer and starts reading from the system memory. In case the Host retries the previously sent packet, the Retry buffer Logic starts the retry from the appropriate packet.

4.6 Overview of USB 2.0 interface block

EZ-USB™ FX10 USB2.0 device controller supports High-speed (480 Mbps) and Full-Speed (12 Mbps) modes. It supports up to 32 endpoints. Each endpoint can be independently configured to support any transfer types – control, interrupt, bulk, or isochronous transfers.

4.6.1 Features

- Compliant to USB2.0
- Supports link power management (LPM) of USB2.0
  - L1-Sleep and L2-Suspend
- Supports high-speed (480Mbps) and full-speed (12Mbps) transmission
- Support up to sixteen IN / OUT endpoints.
- Supports all USB2 transfer types – Bulk, Isochronous, Control and Interrupt
- Supports remote wake-up capabilities
- Supports max packet size of up to 1024 bytes
- Supports UTMI+ Level 3 interface to the PHY
- Uses 16-bit UTMI interface to PHY
- Supports endpoint data transfer through DMA controller

4.6.2 Block diagram

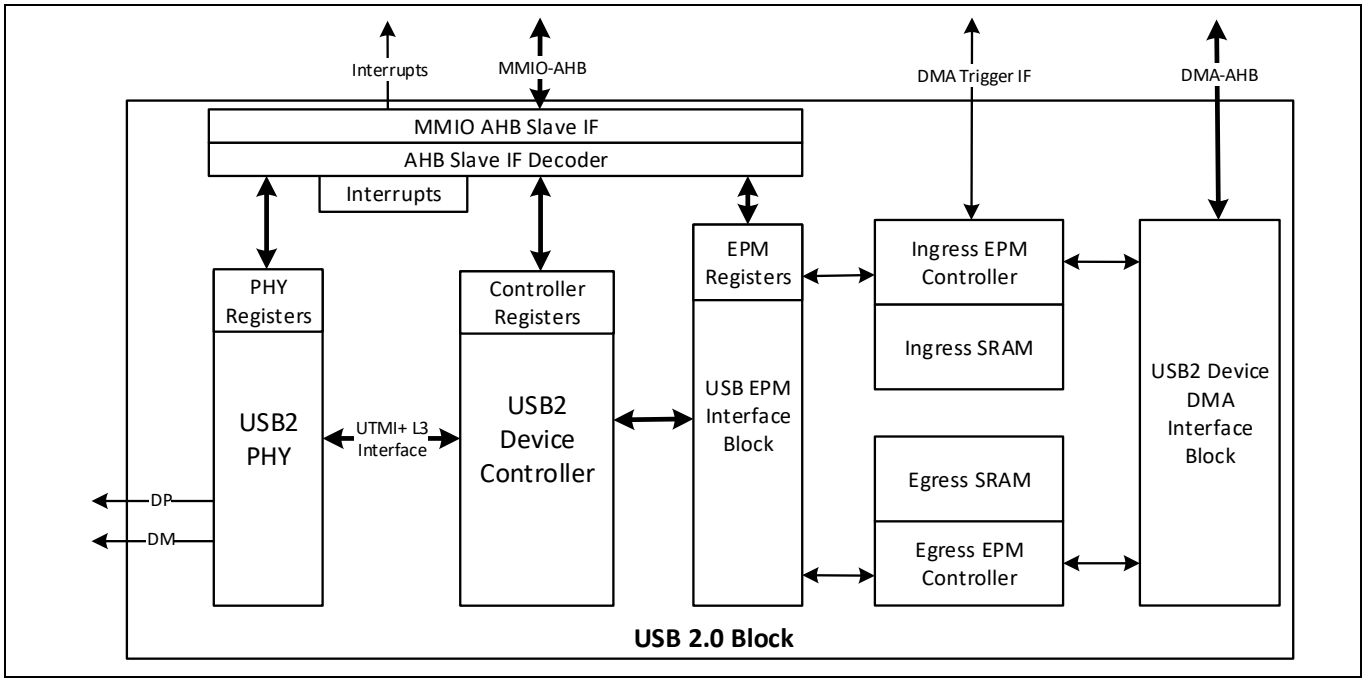


Figure 4-7. USB 2.0 interface block diagram

4.6.3 Overview

4.6.3.1 USB2 PHY

The USB2.0 PHY performs low-level protocol and signaling functions. The PHY, while transmitting, serializes data, generates, and appends SYNC and EOP packet fields, performs bit stuffing and NRZI encoding operations. While receiving data, PHY recovers incoming data and clock, de-serializes data, removes SYNC and EOP packet fields, performs bit de-stuffing and NRZI decoding operations.

4.6.3.2 USB2 device controller

The block is the peripheral controller that enables communication with the external USB host. It can be configured as either a full-speed or a high-speed peripheral.

4.6.3.3 Endpoint memory (EPM) USB interface block

This block interfaces with USB2 device controller on one side and connects to EPM controller on the other size. Based on the direction of the transaction, it is partitioned into two sub-blocks.

4.6.3.4 Endpoint memory (EPM) USB write interface block

The Host's request for OUT/CONTROL transfer from host to device are handled by this block. The DATA stage of the transactions is converted from UTMI clock domain to AHB clock domain. Unavailability of memory space for storage would result in a flow control to be reported back from AHB clock domain to UTMI clock domain. The device controller then issues NAK on the USB interface. The USB packet data is converted to 32-bit words and forwarded to the Ingress controller for further processing.

4.6.3.5 Endpoint memory (EPM) USB read interface block

The Host's request for IN/CONTROL transfer from device to host, are handled by this block. The DATA stage of the transactions is converted from AHB clock domain to UTMI clock domain. The endpoint data and response from Egress controller is forwarded to the device controller block as 16-bit words. The data is forwarded in response to host requests.

4.6.3.6 Ingress EPM controller

This block manages the ingress path during the data stage of a USB transfer. The primary functions of the block are summarized below. The functions are common across all endpoints.

- Keeps track of the number of bytes received during each data phase of a USB transfer
- Tracks the number of ingress data requests which arrives on the USB ingress interface
- Arbitrates memory access between the USB write interface and the AHB read request to the Ingress SRAM
- Communicates and handshakes with the external DMA controller at packet boundaries
  - Generates trigger to external DMA control after arrival and storing of a complete packet. It increments ingress data request count
  - Decrements ingress data request count after receiving a trigger from DMA indicating transfer completion. A transfer completion by DMA involves reading data corresponding to one USB packet. This involves multiple AHB read requests.
- Upon memory full condition, the incoming data from USB interface is dropped and is NAK'ed
- Short packets and ZLPs interrupt the CPU. Additionally, SLPs triggers an external DMA controller. The CPU is expected to handle the processing of these packets by appropriate means.

4.6.3.7 Egress EPM controller

The block manages the egress path during the data stage of a USB transfer. The primary functions of the egress EPM core are listed below. These functions are carried out on per-endpoint basis.

- Arbitrates memory access between the AHB write interface and the USB read request to the Egress SRAM
- Data intended for each endpoint is moved to the corresponding egress SRAM by the external DMA controller
- Manages retry on the USB Interface - Flushes current transaction after it is successfully transmitted on the USB interface
- ISO tokens: Flush happens after the last byte is sent on interface
- BULK tokens: Flush happens after an ACK is received on the interface

4.6.3.8 Ingress EPM SRAM

The memory block allows storing of up to two full-sized USB packets in the buffer. This memory is shared between all endpoints.

- Data stage of OUT and CONTROL transfers are stored
- Stores multiple transactions for different endpoints in the same memory

4.6.3.9 Egress EPM SRAM

The egress memory block allows storing of two full-sized USB transactions in the buffer. The information generated on the Egress path and stored in the buffer are summarized below.

- Pre-loaded data to be sent on the USB interface in current micro-frame for any IN and CONTROL requests
- Stores multiple short transactions to be transmitted on USB interface
- Stores multiple transactions for different endpoints in the same memory

4.6.4 USB 2 device controller – Low power modes

The table below summarizes the low-power modes supported.

Table 4-4. Low-power modes

Power modes	Description
ACTIVE	All the logic is powered on with all clocks running
DEEPSLEEP	ACTIVE domain logic is powered down. Clocks are turned off, except for clocks required during Deep Sleep. USB PHY is in L2 suspend state

These power states are entered depending on the USB2.0 bus power states.

The USB device controller is partitioned into two power domains – Active and Deep Sleep, in order to save power. During normal operation, the IP is in Active power mode allowing packet transfers on interface. Deep Sleep can be entered on a request to enter L2 suspend state on USB interface. The L1-sleep state is an optional low power state with faster entry and exit time requirements. The controlled remains in Active state during L1-sleep. An exit from these low power modes is based on one of the two conditions – host-initiated wake-up or resume request or a device-initiated resume request which in turn wakes up the host.

4.7 USB programming model

This section explains about overall EZ-USB™ FX10 firmware architecture, different layers and its responsibility along with how they communicate with each other. **Figure 4-8** gives information about firmware's layered architecture.

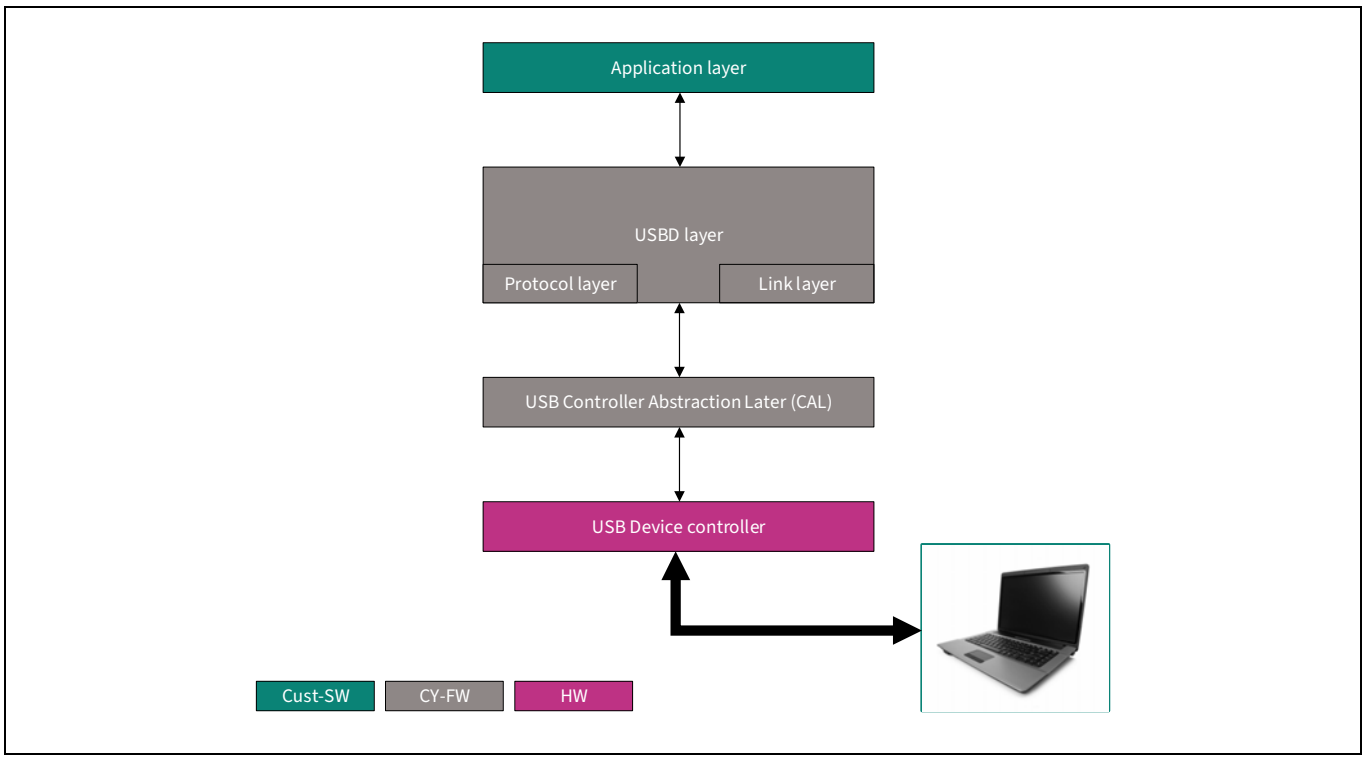


Figure 4-8. USB programming model

USB device controller: Hardware where USB device controller functionality is implemented. It supports USB 3.x specification.

Controller abstraction layer (CAL): This layer implements the register access functions and ISRs.

USBD layer: This layer is responsible for maintaining various state machines, handling enumeration process, providing various API and callback to application layer, maintaining end to end connectivity along with port-port connectivity. As part of 3.x architecture, this layer is divided into two sub layer called protocol layer and link layer.

- Protocol Layer: Generic USB Device end-to-end functionality is implemented in this layer. Maintaining device state, handling enumeration process are responsibility of this layer. It also provides APIs and callback to upper layers.
- Link Layer: Most of Link layer state machine handling and error recovery link training will be taken care in hardware. This layer provides few APIs to manage link power state changes to upper layer.
- Application Layer: USB device's functionality (like UVC/HID/MSD) will be defined and addressed in this layer. This layer also communicates with other hardware devices at another end (non-USB end) which makes device functionality complete. Class driver at host side will communicate with this layer in the device.

4.7.1 USB Controller Abstraction Layer (CAL)

The USB CAL layer hides all hardware complexity from above layers. It provides set of APIs to read/write different hardware blocks within USB controller. It also handles critical part of interrupts and inform upper layers through various events and messages to perform rest of the operations.

CAL layer implementation can be divided into two major components:

- Interrupt handling
- APIs for upper layers i.e., USB Device (USBD) protocol layer and USBD link layer

4.7.1.1 Interrupt handling

One of the major responsibilities of CAL layer is to handle USB device related interrupts. As a result, various Interrupt Service Routines (ISRs) are implemented as part of CAL layer. ISR will do only critical task required for the raised interrupt and rest of the operations should be done in the respective threads. Each ISR will access different registers to identify cause of the interrupt and if required, then it will read some more registers to collect data related to raised interrupt. Once ISR identifies the cause and collects the required data, it will inform to USBD thread through a interrupt- safe MessageQueue implementation.

Interrupts are divided into two major sub-sections:

- Interrupts related to protocol layer
- Interrupt related to link layer

Figure 4-9 gives information about high level interrupt hierarchy in the CAL layer.

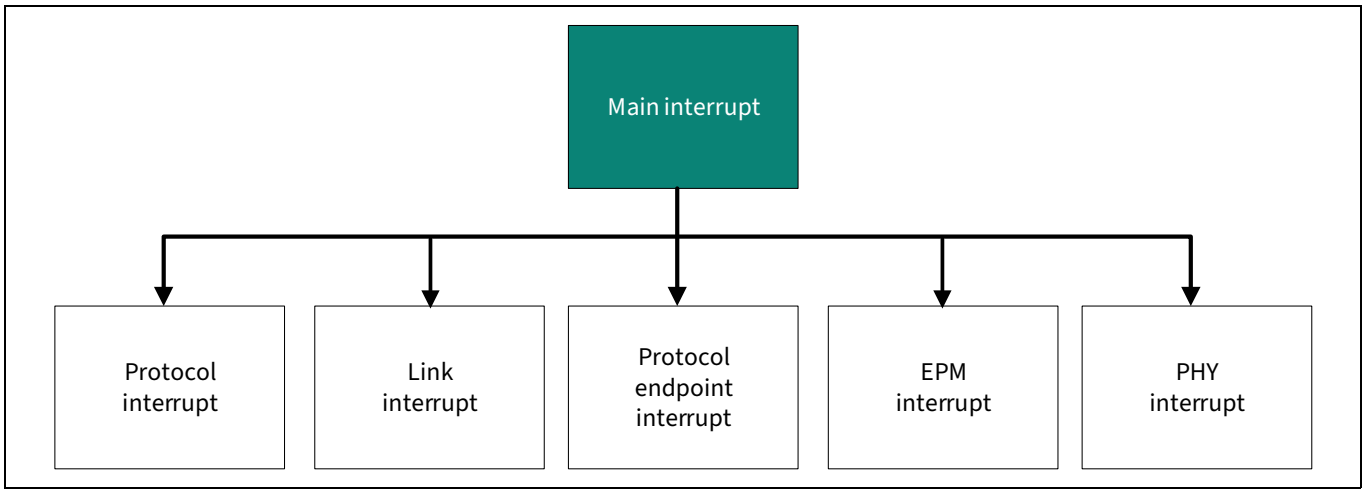


Figure 4-9. Interrupt hierarchy

4.7.2 USB device (USBD) Layer

As part of 3.x architecture, this layer is divided into two sub layer called protocol layer and link layer. From firmware point of view, end to end functionality will be considered as protocol layer and port-port functionality will be considered as link layer.

USB device (USBD) layer is responsible for the following:

- Handles enumeration process.
- Provides API for application layer
- Provides callback mechanism for application layer
- Maintains USB device state machine
- Handles events/messages coming from lower layer which includes generic USBD messages, protocol layer specific messages, and link layer specific messages.
- Maintain end to end flow control (protocol sub-layer)
- Maintain port-port flow control (link layer)
- Power management (link sub-layer mostly).
- Suspend will be handled by protocol sub-layer

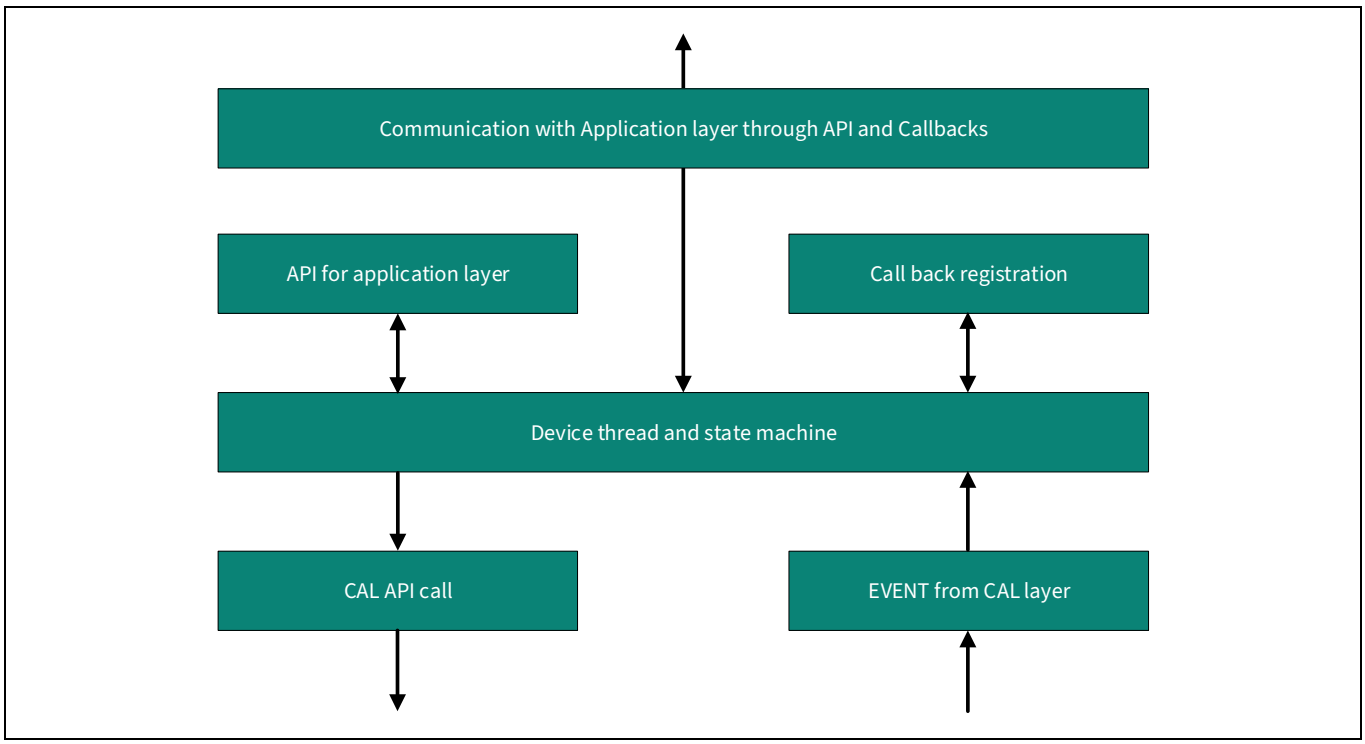


Figure 4-10. Module within USBD layer

4.7.2.1 USB device layer initialization

The first API that needs to be called for USB initialization is Cy_USB_USBD_Init. This handles device level initialization for both USB Hi-Speed and USB 10 Gbps functionalities by calling Cy_USBHS_Cal_Init and Cy_USBSS_Cal_Init respectively.

All associated data structures, endpoint memories, interrupts, callback registrations are cleared and endpoints and required DMA channels for the Default Control Pipe (endpoint 0) are initialized.

USB high-speed device implementation uses the CPU DMA and USB 5 Gbps and above uses the High-Bandwidth DMA (HBDMA) respectively. These are initialized by Cy_USB_USBD_cpuDmaInit and Cy_USB_Create_Ep0HbDmaChannels API calls.

At this stage, a RTOS task message queue and USBD task is also created and is tasked with handling various USBD messages passed on by the EZ-USB™ FX10 Controller Abstraction Layer (CAL). See cy_en_usb_cal_msg_type_t enumeration in cy_usb_common.h for details on various CAL layer messages that the task handles.

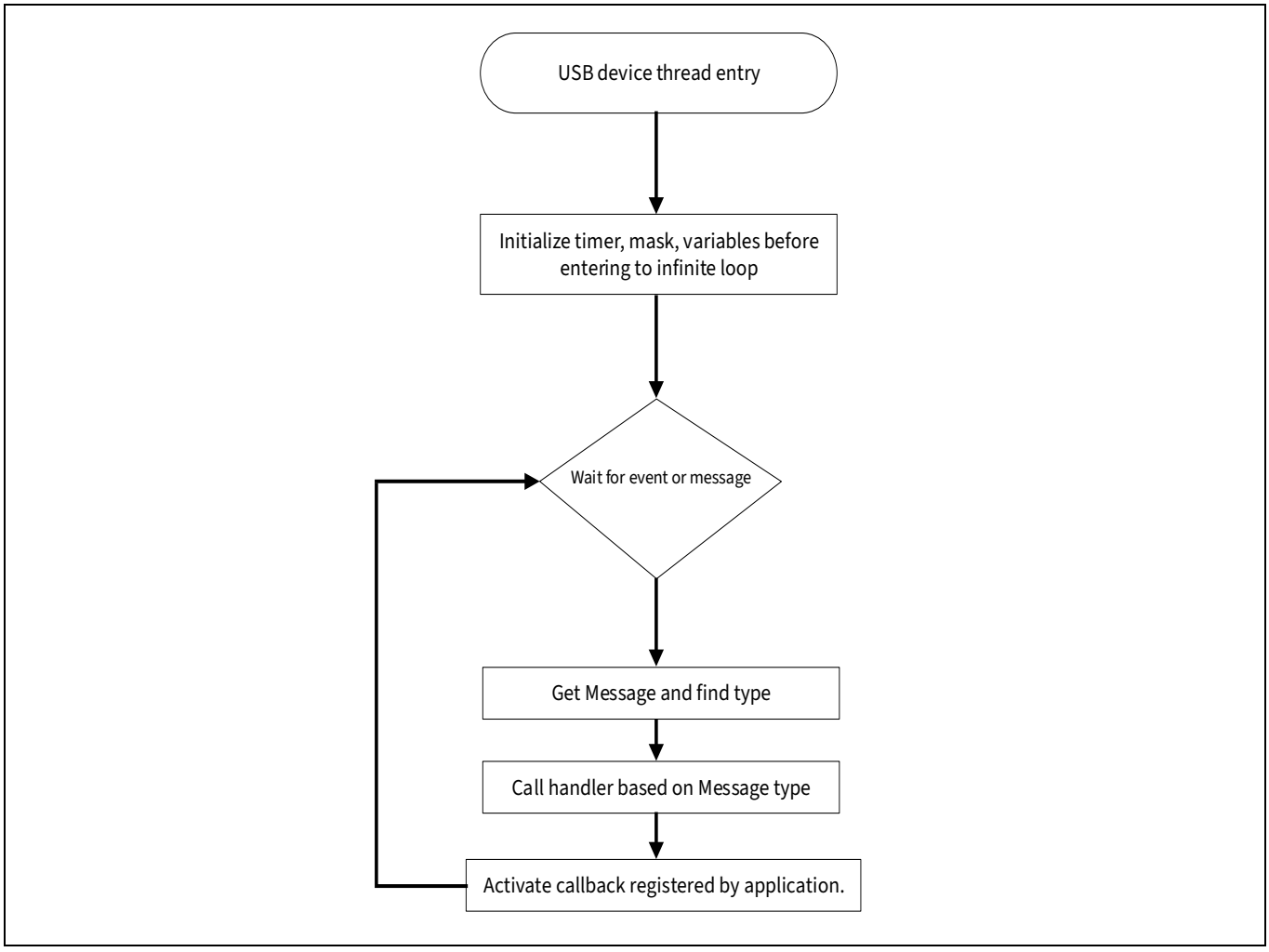


Figure 4-11. USBD task

4.7.2.2 USB enumeration

The USB device needs the following to be configured before the device is ready for connection to the USB host for enumeration.

- Register USB Callbacks using `Cy_USBD_RegisterCallback` API. See `cy_en_usb_usbd_cb_t` enumeration in `cy_usb_usbd.h` for various callback events provided to the application.
- Register USB Descriptors using `Cy_USBD_SetDscr` API. For USB 5 Gbps and above, the populated USB descriptors need to be present in the high-bandwidth DMA region.
- Configure VBUS detection GPIO (P4.0) to signal an interrupt for both rising and falling edges. The detected interrupt is used to signal an event to the application task handler thread to enable the USB IP block and try for a USB connection.

On receiving a VBUS detected event, the device should try for a USB connection by calling `Cy_USBD_ConnectDevice` API. The application can call this API with the desired connection speed (i.e., USB High-Speed, USB 5 Gbps or above). Based on the selected speed, the appropriate CAL layer API will be invoked. Note that the final speed will be decided after negotiation.

`Cy_USBSS_Cal_Connect` API handles initiating the connection for USB 5 Gbps and above speeds. This API does the following:

- Performs orientation detection using CC pins and choose the appropriate PHY for connection
- Enables USB link and protocol level interrupts
- Configures various link layer timeouts as per USB specification (For example, LFPS durations for Low-power Mode exits etc.)
- Initialises the active USB PHY by calling the `Cy_USBSS_Cal_InitSSPhy` API. This API enables the PHY Sub System and configures the PHY PLL by calling the `Cy_USBSS_Phy_PllInit` API. It also initialises the transmitter and receiver section of the PHY by calling the respective APIs.
  - `Cy_USBSS_Phy_TxInit` configures the electrical parameters of the transmitter. See **"USB PHY initialization"** on page 57 section for more details.
  - `Cy_USBSS_Phy_RxInit` enables the USB3 terminations and configures the receiver analog front end (AFE)
- Configure the USB block to use the PHY clock (312.5 MHz)
- Enable clock by writing to the `CTRL_CLK_EN` bit in `USB32DEV_MAIN` register. It is at this stage that the device host starts the enumeration process for the device and queries the supported USB descriptors.

`Cy_USBD_ConnectHsDevice` API handles initiation of connection for USB high-speed device. This API enables the required interrupts and make the device visible on the bus. Once the device is visible, host will issue reset and start enumeration process.

The above sections provide a brief overview on EZ-USB™ FX10 USB initialization and usage flow. Users can also refer to `cy_usb_usbd.c` and `cy_usb_usbxx_cal_drv.c` files in `fx3g2_usb_common` folder of the EZ-USB™ FX10 firmware stack for exact implementation details of the APIs referred in the above sections.

4.7.2.2.1 USB PHY initialization

Configurable parameters

- **Signal amplitude**

The amplitude (swing) of the differential signal transmitted by the EZ-USB™ FX10 device in mV is set using the macro called FX3G2_TXSWING_MV. By default, the maximum amplitude supported by the device is 1.15 V (1150 mV) and the amplitude is set to a value of 1.10 V.

- **De-emphasis and pre-shoot**

Please refer to section 6.7.5.2 of the USB 3.2 Specification for details of how a 3-tap transmit equalizer is used to select the desired de-emphasis and pre-shoot values for the 10 Gbps signal.

The tap coefficient values C_{-1}, C_1 and C_0 corresponding to the desired de-emphasis and pre-shoot settings must be calculated and then plugged into the `PhyTxCursorPoints` array used in the driver code in the order of C_1, C_0 and C_{-1}.

The tap coefficients can be calculated as shown below.

1. Select the desired De-emphasis (DE) and Pre-shoot values (PS).
2. Calculate $k = 10^{(PS / 20)}$
3. Calculate $l = 10^{(DE / 20)}$
4. $C_{-1} = (l * (1 - k)) / (2 * (k * l + 1 - l))$
5. $C_1 = (l - 1) / (2 * (k * l + 1 - l))$
6. $C_0 = 1 - abs(C_{-1}) - abs (C_1)$
7. Convert the three coefficients to the desired range:
   a) PhyTxCursorPoints[0] = (uint32_t)(1000 * abs(C_1))
   b) PhyTxCursorPoints[1] = (uint32_t)(1000 * abs(C_0))
   c) PhyTxCursorPoints[2] = (uint32_t)(1000 * abs(C_{-1}))

Pre-shoot (dB)	De-emphasis (dB)	k	l	C_{-1}	C_1	C_0	PhyTxCursorPoints[0]	PhyTxCursorPoints[1]	PhyTxCursorPoints[2]
1.2	-2.1	1.1481536	0.7852356	-0.052	-0.096	0.852	96	851	52
2.2	-3.1	1.2882496	0.699842	-0.084	-0.125	0.791	791	791	83
3.2	-7	1.4454398	0.4466836	-0.083	-0.231	0.686	230	686	82
2.5	-5	1.3335214	0.5623413	-0.079	-0.184	0.737	184	736	78

- **EZ-USB™ FX10 PHY configuration procedure**

The configuration of the USB 3.2 transmitter electrical parameters is done in the `Cy_USBSS_Phy_TxInit` function.

- Perform impedance calibration on the SSTX pads. This operation is started by setting the ZCAL_EN bit in the TX_AFE_ZTRIM register and then waiting until the `reg_int_tx_imp_cal_done` bit in the INTR0 register gets set. Once this calibration done interrupt is received, the calibration code will be available in the ZCAL_CODE field of the TX_AFE_STATUS register.

```
/* Start impedance calibration */
USB_PHY_TOP->TX_AFE_ZTRIM |= USB32DEV_PHYSS_USB40PHY_TOP_TX_AFE_ZTRIM_ZCAL_EN_Msk;
Cy_SysLib_DelayUs(1);
/* Wait for the calibration to complete. */
do {
i = USB_PHY_TOP->INTR0;
} while ((i & USB32DEV_PHYSS_USB40PHY_TOP_INTR0_REG_INT_TX_IMP_CAL_DONE_Msk) == 0);
/* Clear the calibration done interrupt. */
USB_PHY_TOP->INTR0 = USB_PHY_TOP->INTR0;
```

USB interface

- Calculate the total number of half-rate driver slices that should be used by adding 56 to the calibration code obtained in the previous step

```
nTotal = 56UL + ((USB_PHY_TOP->TX_AFE_STATUS &
USB32DEV_PHYSS_USB40PHY_TOP_TX_AFE_STATUS_ZCAL_CODE_Msk) >>
USB32DEV_PHYSS_USB40PHY_TOP_TX_AFE_STATUS_ZCAL_CODE_Pos);
```

- Calculate the number of slices to be dedicated for swing (TX signal amplitude) control

```
/* nVSwing = (1 - (vSwing / vTx)) * nTotal = (1 - (1.1 V / 1.15 V)) * nTotal. */
nVSwing = (nTotal * (FX3G2_VPTX_MV - FX3G2_TXSWING_MV)) / FX3G2_VPTX_MV;
```

- Calculate the number of slices to be dedicated for pre- and post-shoot

```
/*
* nPost = C[1] * (nTotal - nVSwing)
* nPre = C[-1] * (nTotal - nVSwing)
*/
nPost = (PhyTxCursorPoints[0] * (nTotal - nVSwing)) / 1000;
nPre = (PhyTxCursorPoints[2] * (nTotal - nVSwing)) / 1000;
```

- The transmitter has a maximum of 88 half-rate driver slices available. Each driver slice can be configured in one of 4 modes using a pair of configuration bits. Out of the 88 slices,
  - (88 - nTotal) slices are to be configured with a setting of 'b00 as they are unused
  - nVSwing slices are to be configured with a setting of 'b02 as they are used for transmit amplitude control
  - (nPre + nPost) slices are to be configured with a setting of 'b01 as they are used for pre- or post-shoot control
  - The remaining slices (nTotal - (nVSwing + nPre + nPost)) are to be configured with a setting of 'b03.

A total of 176 bits (88 slices * 2 bits) are calculated as described above and programmed into 8 shift registers which hold 22 bits each. Once the values have been loaded into the shift registers, they are copied into the actual driver control by setting the `reg_tx_cfg_write_done` bit in the `TX_AFE_CFG` register. Once the driver configuration is complete, the PHY will clear this request bit as acknowledgment of the process.

```
/* Program the write done bit to start shifting the data. */
USB_PHY_TOP->TX_AFE_CFG =
USB32DEV_PHYSS_USB40PHY_TOP_TX_AFE_CFG_REG_TX_CFG_WRITE_DONE_Msk;
Cy_SysLib_DelayUs(5);
/* Wait until shift register update is complete. */
do {
i = USB_PHY_TOP->TX_AFE_CFG;
} while ((i & USB32DEV_PHYSS_USB40PHY_TOP_TX_AFE_CFG_REG_TX_CFG_WRITE_DONE_Msk) !=
0);
```

4.7.2.2.2 USB link initialization

When `Cy_USBD_ConnectDevice()` is called to enable the USB connection, the USB stack attempts to initialize the USB link as defined in the USB specification.

When making a USB 3.x data connection, EZ-USB™ FX device detects the USB-C cable plug orientation using the voltage on the CC1 and CC2 pins. The voltage on the CC1 and CC2 pins are measured using the internal ADC and the connection polarity is determined based on the pin on which a valid V_{Rd} voltage is detected. The orientation information is used to choose either the USB3TX1 and USB3RX1 pairs or the USB3TX2 and USB3RX2 pairs for operation in case of USB single-lane modes (USB Gen 1x1, USB Gen 2x1).

Once the USB-C cable orientation has been detected, the corresponding USB PHY instance is initialized as described in the previous section.

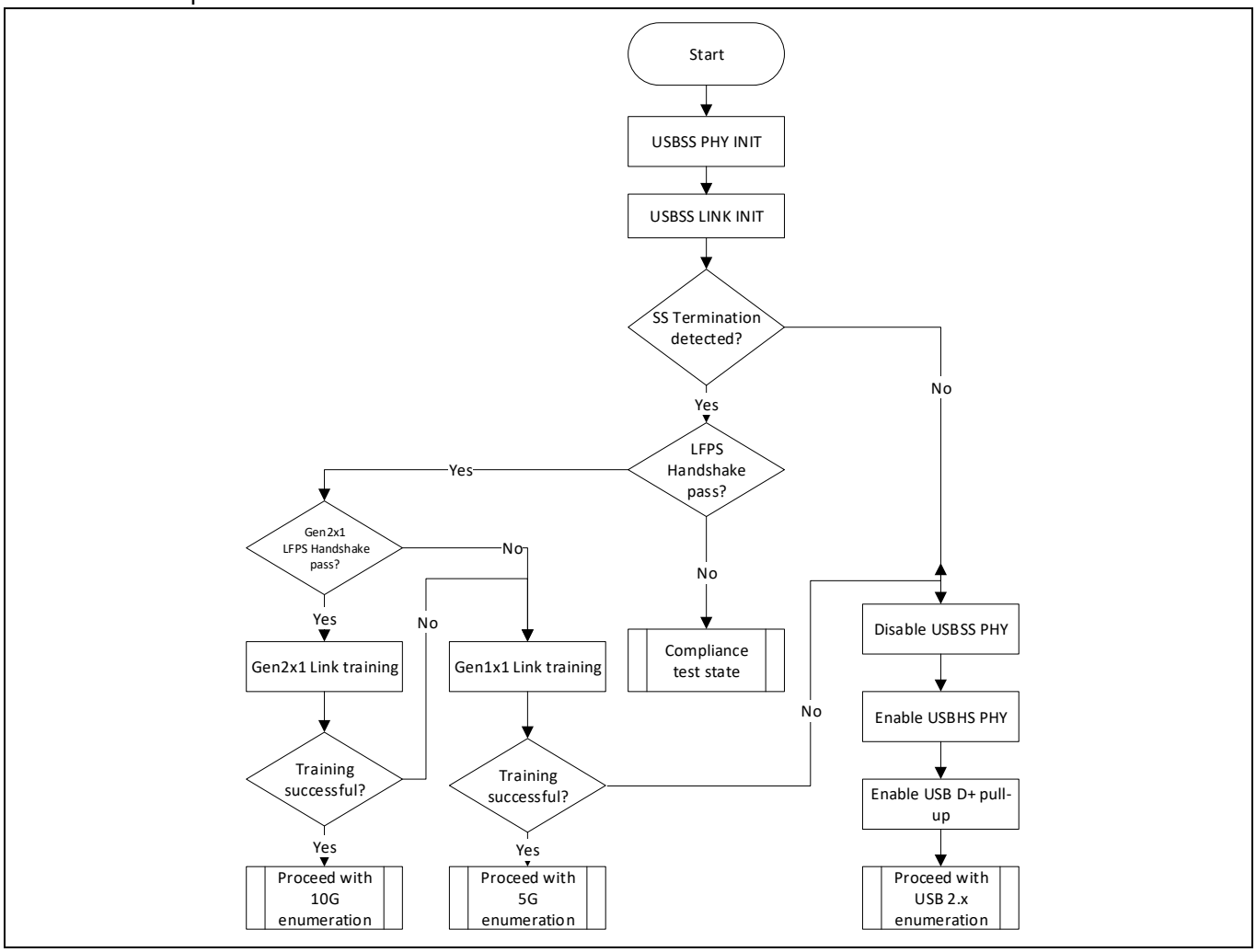


Figure 4-12. USB link initialization flow chart

Figure 4-12 shows a simplified flow chart of the USB link initialization sequence. Link will get initialized in any one of 10 Gbps, 5 Gbps, 480 Mbps or 12 Mbps modes based on the type of upstream port that EZ-USB™ FX is connected to.

4.7.2.2.3 Receiver equalization

Receiver equalization is done in the EZ-USB™ FX10 USB PHY receiver block when the device reaches polling. RxEQ LTSSM state during its link training. EZ-USB™ FX10 has 16 predefined Look Up Table (LUT) entries that hold configuration values for the Continuous Time Linear Equalization (CTLE) resistors, capacitors and boost settings. Variable Gain Amplifier (VGA), Decision Feedback Equalizer (DFE) and InputCap Gain values are adjusted to optimize signal amplitude for each of the available LUT setting. The eye height for each LUT setting is measured and stored during the Polling. RxEQ stage. The best eye height and DFE equalization setting is identified among the LUTs, and this configuration is used to configure the CTLE.

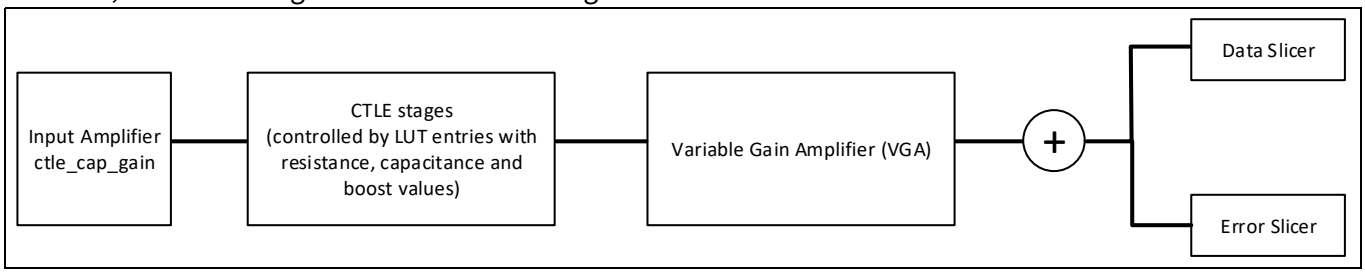


Figure 4-13. Receiver equalization

Section C: CPU subsystem

This section encompasses the following chapters:

- **"CPU subsystem (CPUSS)"** on page 63
- **"SRAM controller"** on page 70
- **"Inter-processor communication"** on page 72
- **"Fault monitoring"** on page 81
- **"Interrupts"** on page 89
- **"Protection units"** on page 110
- **"DMA controller (DMAC)"** on page 131
- **"Data wire controller (DW)"** on page 141
- **"Cryptographic function block (CRYPTO)"** on page 154
- **"Nonvolatile memory"** on page 205
- **"Boot code"** on page 240
- **"Device security"** on page 258
- **"Program and debug interface"** on page 263

Top level architecture

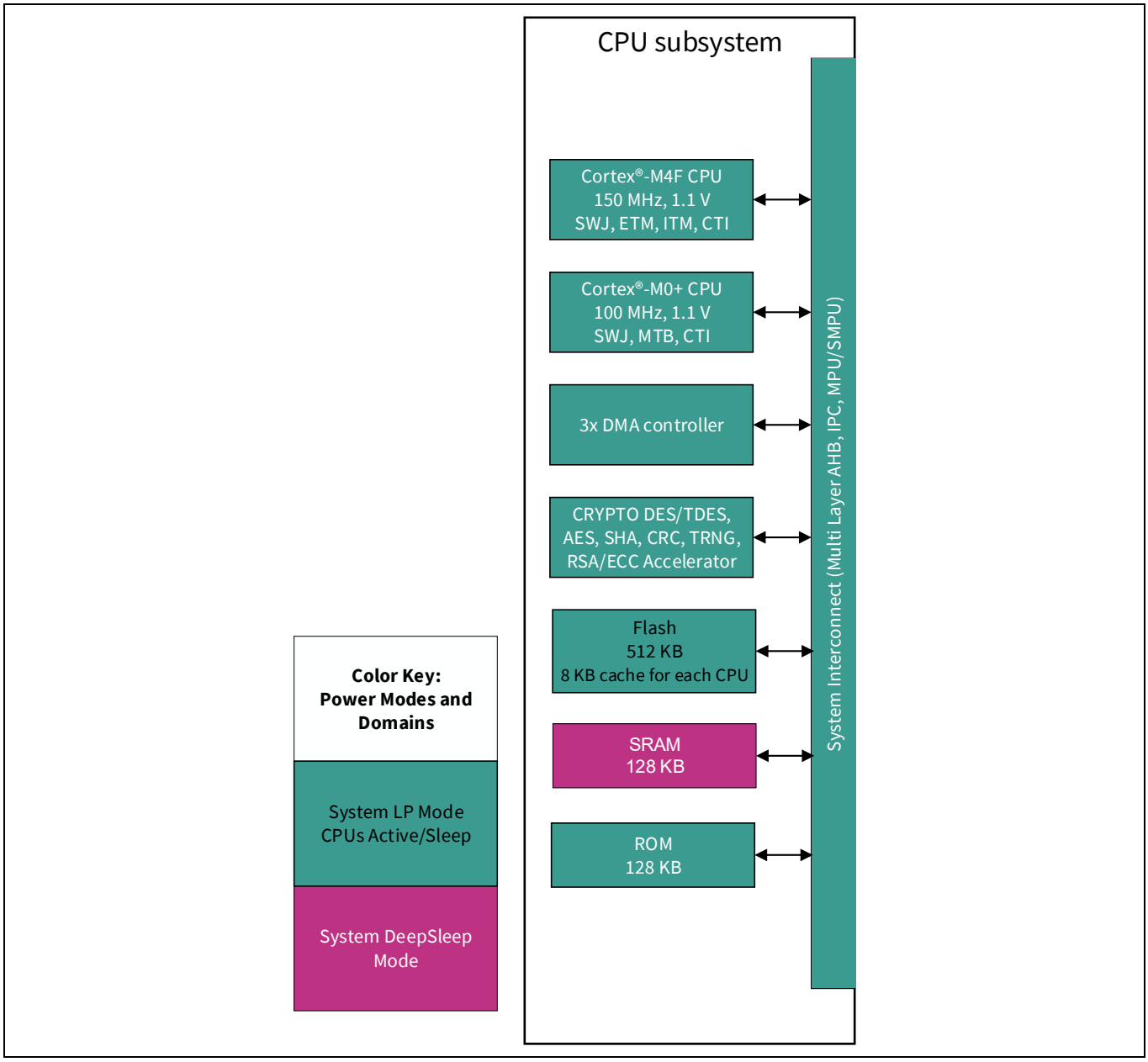


Figure C-1. CPU system block diagram

5 CPU subsystem (CPUSS)

The CPU subsystem is based on dual 32-bit Arm® Cortex® CPUs, as **Figure 5-1** shows. The Cortex®-M4 is the main CPU. It is designed for short interrupt response time, high code density, and high 32-bit throughput while maintaining a strict cost and power consumption budget. A secondary Cortex®-M0+ CPU implements security, safety, and protection features.

This section provides only an overview of the Arm® Cortex® CPUs in EZ-USB™ FX10. For details, see the Arm® documentation sets for **Cortex®-M4** and **Cortex®-M0+**.

5.1 Features

EZ-USB™ FX10 Arm® Cortex® CPUs have the following features:

- Cortex®-M4 has a floating-point unit (FPU) that supports single-cycle digital signal processing (DSP) instructions, and a **memory protection unit** (MPU). Cortex®-M0+ has an **MPU**.
- Both CPUs have 8-KB instruction caches with four-way set associativity.[1)]
- Maximum clock frequency of 150 MHz for the Cortex®-M4 and 100 MHz for the Cortex®-M0+
- The Cortex®-M4 implements a version of the Thumb instruction set based on Thumb-2 technology (defined in the *Armv7-M Architecture Reference Manual*). The Cortex®-M0+ supports the Armv6-M Thumb instruction set (defined in the *Armv6-M Architecture Reference Manual*). See **"Instruction set"** on page 68.
- Both CPUs have Cortex®-M4 devices generic user guide for rapid and deterministic interrupt response. For details, see the **"Interrupts"** on page 89
- Both CPUs have extensive debug support. For details, see the **"Program and debug interface"** on page 263.
  - SWJ: combined serial wire debug (SWD) and Joint Test Action Group (JTAG) ports
  - Serial wire viewer (SWV): provides real-time trace information through the serial wire output (SWO) interface
  - Breakpoints
  - Watchpoints
  - Trace: Cortex®-M4: embedded trace macrocell (ETM). Cortex®-M0+: 4-KB micro trace buffer (MTB)
- Inter-processor communication (IPC) hardware – see the **"Inter-processor communication"** on page 72

1) EZ-USB™ FX10 does not support cache coherency. As a result when a particular row of flash that executes instructions is written/updated, the updated information will not be reflected in the cache. The cache should be cleared in the firmware during such instances. This is applicable for both CM4 and CM0+ cache – the appropriate (CM0+ and/or CM4) cache should be invalidated.

5.2 Architecture

Each CPU is a 32-bit processor with its own 32-bit datapath and a 32-bit memory interface. Each CPU has its own set of 32-bit registers. They support a wide variety of instructions in the Thumb instruction set. They support two operating modes (see **"Operating modes and privilege levels"** on page 68).

The Cortex®-M4 instruction set includes:

- Signed and unsigned, 32×32 → 32-bit and 32×32 → 64-bit, multiply and multiply-accumulate, all single-cycle
- Signed and unsigned 32-bit divides that take two to 12 cycles
- DSP instructions, including single instruction multiple data (SIMD) instructions
- Complex memory-load and store access
- Complex bit manipulation; see the bitfield instructions in **Table 5-5**

The Cortex®-M4 FPU has its own set of registers and instructions. It is compliant with the ANSI/IEEE Std 754-2008, IEEE Standard for Binary Floating-Point Arithmetic.

The Cortex®-M0+ has a single cycle 32×32 → 32-bit signed multiplication instruction.

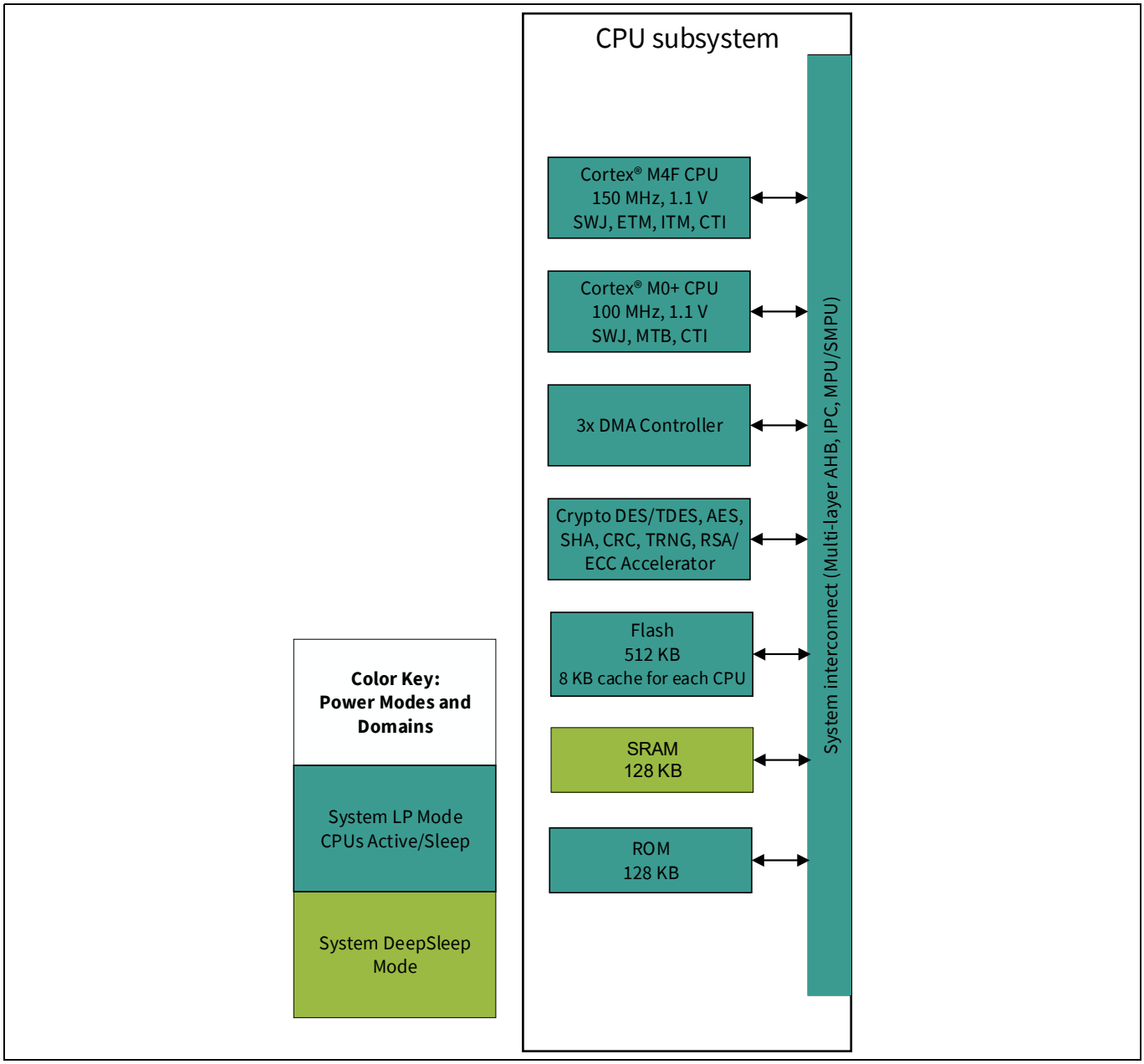


Figure 5-1. CPU subsystem block diagram

5.2.1 Address and memory maps

Both CPUs have a fixed address map, with shared access to memory and peripherals. The 32-bit (4 GB) address space is divided into the regions shown in **Table 5-1**. Note that code can be executed from the code and SRAM regions.

Table 5-1. Address map for CM4 and CM0+

Address range	Name	Use
0x0000 0000 – 0x1FFF FFFF	Code	Program code region. Data can also be placed here. It includes the exception vector table, which starts at address 0.
0x2000 0000 – 0x3FFF FFFF	SRAM	Reserved.
0x4000 0000 – 0x5FFF FFFF	Peripheral	All peripheral registers. Code cannot be executed from this region. CM4 bit-band in this region is not supported in Part Number.
0x6000 0000 – 0x9FFF FFFF	External RAM	Quad SPI (see **"Serial memory interface (SMIF)"** on page 506). Code can be executed 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.

The device memory map shown in **Table 5-2** applies to both CPUs. That is, the CPUs share access to all EZ-USB™ FX10 Memory and Peripheral registers.

Table 5-2. Internal memory address map for CM4 and CM0+

Address range	Memory type	Size
0x0000 0000 – 0x0001 FFFF	ROM	128 KB
0x0800 0000 – 0x0801 FFFF	SRAM	128 KB
0x1000 0000 – 0x1007 FFFF	Application flash	Up to 512 KB
0x1400 0000 – 0x1400 7FFF	Auxiliary flash, can be used for EEPROM emulation	32 KB
0x1600 0000 – 0x1600 7FFF	Supervisory flash	32 KB
0x1C00 0000 – 0x1C0F FFFF	High-bandwidth DMA buffer SRAM	Up to 1 MB

SRAM is located in the code region for both CPUs (see **Table 5-1**). This facilitates executing code out of SRAM. There is no physical memory located in the CPUs' SRAM region.

Note: *The CPUSS_CM0_VECTOR_TABLE_BASE and CPUSS_CM4_VECTOR_TABLE_BASE registers determine the location of the vector table for each CPU. A number of LS bits in each register are set to '0'. As a result, there are restrictions on the location of vector tables – they must be on a 256-byte boundary for CM0+ and a 1024-byte boundary for CM4.*

5.2.2 Registers

Both CPUs have sixteen 32-bit registers, as **Table 5-3** shows.

- **R0 to R12 - General-purpose registers**: R0 to R7 can be accessed by all instructions; the other registers can be accessed by a subset of the instructions
- **R13 - Stack Pointer (SP)**: There are two stack pointers, with only one available at a time. In thread mode, the CONTROL register indicates the stack pointer to use – Main Stack Pointer (MSP) or Process Stack Pointer (PSP)
- **R14 - Link register**: Stores the return program counter during function calls
- **R15 - Program Counter**: This register can be written to control program flow

Table 5-3. Cortex®-M4 and Cortex®-M0+ registers

Name	Type[a)]	Reset value	Description
R0 – R12	RW	Undefined	R0–R12 are 32-bit general-purpose registers for data operations.
MSP (R13) PSP (R13)	RW	[0x0000 0000]	The Stack Pointer (SP) is register R13. In thread mode, bit[1] of the CONTROL register indicates the stack pointer to use: 0 = Main stack pointer (MSP). This is the reset value 1 = Process stack pointer (PSP) On reset, the processor loads the MSP with the value from the vector address
LR (R14)	RW	See note[b)]	The Link Register (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions
PC (R15)	RW	[0x0000 0004]	The Program Counter (PC) is register R15. It contains the current program address. On reset, the processor loads the PC with the value from the vector address plus 0x0000 0004. Bit[0] of the value is loaded into the EPSR T-bit (see **Table 5-4**) at reset; it must always be '1'.
PSR	RW	Undefined	The Program Status Register (PSR) combines: • Application Program Status register (APSR) • Execution Program Status register (EPSR) • Interrupt Program Status register (IPSR)
APSR	RW	Undefined	The APSR contains the current state of the condition flags from previous instruction executions
EPSR	RO	0x0100 0000	On reset, the EPSR Thumb state bit is loaded with the value bit[0] of the register [0x0000 0004]. It must always be '1'. In Cortex®-M4, other bits in this register control the state of interrupt-continuable instructions and the if-then (IT) instruction.
IPSR	RO	0	The IPSR contains the current exception number.
PRIMASK	RW	0	The PRIMASK register prevents activation of all exceptions with configurable priority.
CONTROL	RW	0	The CONTROL register controls: • The privilege level in Thread mode; see **"Operating modes and privilege levels"** on page 68 • The currently active stack pointer, MSP or PSP • Cortex®-M4 only: whether to preserve the floating-point state when processing an exception
FAULTMASK	RW	0	Cortex®-M4 only. Bit 0 = 1 prevents the activation of all exceptions except NMI.
BASEPRI	RW	0	Cortex®-M4 only. When set to a nonzero value, prevents processing any exception with a priority greater than or equal to the value.

a) Describes access type during program execution in thread mode and handler mode. Debug access can differ.

b) LR reset value is 0xFFFF FFFF in Cortex®-M4, undefined in Cortex®-M0+.

The Cortex®-M4 floating-point unit (FPU) also has the following registers:

- Thirty-two 32-bit single-precision registers, S0 to S31. These registers can also be addressed as sixteen 64-bit double-precision registers, D0 to D15
- Five FPU control and status registers:
  - CPACR – Coprocessor Access Control Register
  - FPCCR – Floating-point Context Control Register
  - FPCAR – Floating-point Context Address Register
  - FPSCR – Floating-point Status Control Register
  - FPDSCR – Floating-point Default Status Control Register

For more information on how these registers are used, see the Cortex®-M4 devices generic user guide.

Use the MSR and MRS instructions to access the PSR, PRIMASK, CONTROL, FAULTMASK, and BASEPRI registers. **Table 5-4** and **Table 5-5** show how the PSR bits are assigned.

Table 5-4. Cortex®-M0+ PSR bit assignments

Bit	PSR register	Name	Usage
31	APSR	N	Negative flag
30	APSR	Z	Zero flag
29	APSR	C	Carry or borrow flag
28	APSR	V	Overflow flag
27–25	–	–	Reserved
24	EPSR	T	Thumb state bit. Must always be ‘1’. Attempting to execute instructions when the T bit is 0 results in a HardFault exception.
23–6	–	–	Reserved
5–0	IPSR	Exception Number	Exception number of current ISR: 0 = thread mode 1 = reserved 2 = NMI 3 = HardFault 4 – 10 = reserved 11 = SVCall 12, 13 = reserved 14 = PendSV 15 = SysTick 16 = IRQ0 … 47 = IRQ31

5.2.3 Operating modes and privilege levels

Both CPUs support two operating modes and two privilege levels:

- **Operating modes**
  - **Thread mode**: Used to execute application software. The processor enters Thread mode when it comes out of reset.
  - **Handler mode**: Used to handle exceptions. The processor returns to Thread mode when it has finished all exception processing.
- **Privilege levels**
  - Unprivileged: The software has limited access to the MSR and MRS instructions, and cannot use the CPSID and CPSIE instructions. It cannot access the system timer, NVIC, or system control block. It may have restricted access to memory or peripherals.
  - Privileged: The software can use all the instructions and has access to all resources

In Thread mode, the CONTROL register controls whether software execution is privileged or unprivileged. In Handler mode, software execution is always privileged.

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

In Handler mode, the MSP is always used. The exception entry and return mechanisms automatically update the CONTROL register, which may change whether MSP/PSP is used.

In Thread mode, use the MSR instruction to set the stack pointer bit in the CONTROL register. When changing the stack pointer, use an ISB instruction immediately after the MSR instruction. This ensures that instructions after the ISB execute using the new stack pointer.

5.2.4 Instruction set

Both CPUs implement subsets of the Thumb instruction set, as **Table 5-5** shows. The table does not show the large number of variants and conditions of the instructions. For details, see one of the Arm® Cortex® Generic **User Guides** or **Reference Manuals**.

An instruction operand can be a register, a constant, or another instruction-specific parameter. Instructions act on the operands and often store the result in a destination register. Many instructions have restrictions on using the PC or SP for the operands or destination register. See the Arm® Cortex® documentation for more details.

Table 5-5. Instruction set summary – Cortex®-M4 and Cortex®-M0+

Functional group	Cortex®-M4	Cortex®-M0+	Brief list of instruction mnemonics
Memory access	–	–	LDR, STR, ADR, PUSH, POP
General data processing	–	–	Cortex®-M0+: ADD, ADC, AND, ASR, BICS, CMN, CMP, EOR, LSL, LSR, MOV, MVNS, ORR, REV, ROR, RSB, SBC, SUB, SXT, UXT, TST Cortex®-M4 has all of the above plus: CLZ, ORN, RRX, SADD, SAS, SSA, SSUB, TEQ, UADD, UAS, USA, USUB
Multiply and divide	–	MUL only	MLA, MLS, MUL, SDIV, SMLA, SMLS, SMMLA, SMMLS, SMUA, SMUL, SMUS, UDIV, UMAAL, UMLAL, UMULL
Saturating	–	–	SSAT, USAT, QADD, QSUB, QASX, QSAX, QDADD, QDSUB, UQADD, UQASX, UQSAX, UQSUB
Packing and unpacking	–	–	PKH, SXT, SXTA, UXT, UXTA
Bitfield	–	–	BFC, BFI, SBFX, UBFX
Branch and control	–	4	Cortex®-M0+: B{cc}, BL, BLX, BX Cortex®-M4 has all of the above plus: CBNZ, CBZ, IT, TB
Miscellaneous	–	4	CPSID, CPSIE, DMB, DSB, ISB, MRS, MSR, NOP, SEV, SVC, WFE, WFI Cortex®-M4 has all of the above plus BKPT
Floating-point	–	–	VABS, VADD, VCMP, VCVT, VDIV, VFMA, VFNMA, VFMS, VFNMS, VLD, VLMA, VLMS, VMOV, VMRS, VMSR, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VPOP, VPUSH, VSQRT, VST, VSUB

6 SRAM controller

This chapter explains the EZ-USB™ FX10 MCU SRAM controller, its features, architecture, and wait states. The SRAM controller enables the CPU to read and write parts of the EZ-USB™ FX10 SRAM.

6.1 Features

The CPUSS has up to three identical SRAM controllers. The SRAM controller has the following features:

- Consists of two AHB-Lite interfaces:
  - An AHB-Lite bus interface on clk_fast that connects to the fast bus infrastructure
  - An AHB-Lite bus interface on clk_slow that connects to the slow bus infrastructure
- Supports programmable number of clk_hf wait states
- Supports 8-, 16-, and 32-bit accesses

6.2 Architecture

The design has two AHB-Lite interfaces that connect to the AHB-Lite infrastructure. Each AHB-Lite interface is connected to a synchronization component that translates between the interface clock (either clk_fast or clk_slow) and the high-frequency clock (clk_hf).

Arbitration is performed on the AHB-Lite transfers from the two ports (AHB-Lite interface). Arbitration uses device-wide bus master specific arbitration priorities. Therefore, although two AHB-Lite interfaces are provided, only one AHB-Lite transfer is accepted by the port arbitration component.

The AHB-Lite transfers are the origin for all SRAM accesses; that is, the write buffer and SRAM repair requests result from AHB-Lite transfers. The SRAM controller differentiates between the following three types of AHB-Lite transfers:

- AHB-Lite read transfers
- 32-bit AHB-Lite write transfers
- 8-bit and 16-bit AHB-Lite write transfers (also referred to as partial AHB-Lite write transfers)

Each type is described in detail:

AHB-Lite read transfers

An AHB-Lite read transfer is translated into an SRAM read access. If the read address matches in the write buffer, the SRAM has stale data and the write data provides the requested read data (this functionality is provided by the read merge component).

32-bit AHB-Lite write transfers

A 32-bit AHB-Lite write transfer is translated into an SRAM write access. If the write address matches in the write buffer, the matching write buffer entries have stale data and these entries are invalidated.

Partial AHB-Lite write transfers

A partial AHB-Lite write transfer is translated into an SRAM read access and an SRAM write access. The SRAM read access is the direct result of the partial write transfer and the SRAM write access is the result of a write buffer request. A partial write transfer requires an SRAM read access to retrieve the “missing” data bytes from the SRAM. If the read address matches in the write buffer, the SRAM has stale data and the write data provides the requested read data (this functionality is provided by the read merge component). The requested read data is merged with the partial write data to provide a complete 32-bit data word (this functionality is provided by the write merge component). The address and the merged write data are written to the write buffer. A future write buffer request results in an SRAM write access with the merged write data.

Only the partial AHB-Lite write transfers use the write buffer.

Write buffer

The write buffer is a temporary holding station for future SRAM write accesses.

The buffer allows SRAM write accesses to be postponed. This allows for more performance critical AHB-Lite requests to “overtake” write buffer requests. Memory consistency is guaranteed by matching the SRAM access address with the write buffer entries’ addresses: a “matching” SRAM read access uses the read merge component and a matching SRAM write access invalidates the matching write buffer entries.

When the write buffer is full, an entry needs to be freed to accommodate future partial AHB-Lite write transfers. Therefore, a full write buffer raises the priority of the write buffer request path.

The write buffer is constructed as a FIFO with four entries (the order in which entries are written is the same as the order in which entries are read). Each entry consists of:

- A valid field
- An invalidated field
- A word address
- A 32-bit data word

Note that the merged write data written to the write buffer is always a 32-bit data word. Therefore, no byte mask is required.

When the write buffer is written (an entry is added): the entry valid field is set to ‘1’ and the invalidated field is set to ‘0’.

When the write buffer is read (an entry is removed): the entry valid field is set to ‘0’. If the entry invalidate field is ‘1’, the write buffer request path is selected for an SRAM write access. If the entry valid field is ‘0’, no SRAM access is performed.

On an SRAM read access, a matching entry provides write buffer merge data for the read merge component.

On an SRAM write access resulting from a 32-bit AHB-Lite write transfer, a matching entry invalidated field is set to ‘1’.

The state of the write buffer is reflected by RAMi_STATUS.WB_EMPTY. The write buffer is not retained in Deep Sleep power mode. Therefore, when transitioning to system Deep Sleep power mode, the write buffer should be empty. Note that this requirement is typically met, because a transition to Deep Sleep power mode also requires that there are no outstanding AHB-Lite transfers. If there are no outstanding AHB-Lite transfers, the write buffer gets SRAM access.

6.3 Wait states

The programmable wait states represent the number of clk_hf cycles for a read path through the SRAM memory to flipflops in either the fast domain (CM4 CPU) or slow domain (such as CM0+ CPU, DataWire, and DMA controller).

As the wait states are represented in clk_hf cycles, the wait states do not have to be reprogrammed when the fast clock domain frequency (clk_fast) or slow clock domain frequency (clk_slow) is changed. However, it may be necessary to reprogram the wait states when the high-frequency clock domain (clk_hf) is changed. This means the required number of wait states is a function of the clk_hf frequency.

The fast clock domain is timing closed at a higher frequency than the slow clock domain. Therefore, the read path through the SRAM memory to flipflops in the fast domain is faster than the read path through the SRAM memory to flipflops in the slow domain. In other words, the required number of “fast” wait states (RAMi_CTL.FAST_WS) should be less than or equal to the required number of “slow” wait states (RAMi_CTL.SLOW_WS).

The SRAM controller also has internal SRAM read paths. These paths are to flipflops in the SRAM controller in the high-frequency clock domain (clk_hf). For these SRAM accesses (for example, an SRAM read access to support a partial AHB-Lite write transfer), the fast wait states are used. This is because the maximum fast domain frequency (clk_fast) equals the high-frequency domain frequency (clk_hf).

7 Inter-processor communication

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

- **IPC channel**: Communication and synchronization between processors is achieved using this structure
- **IPC interrupt**: Each interrupt structure configures an interrupt line, which can be triggered by a 'notify' or 'release' event of any IPC channel.

The Channel and Interrupt structures are independent and have no correlation to each other as shown in **Figure 7-1**.

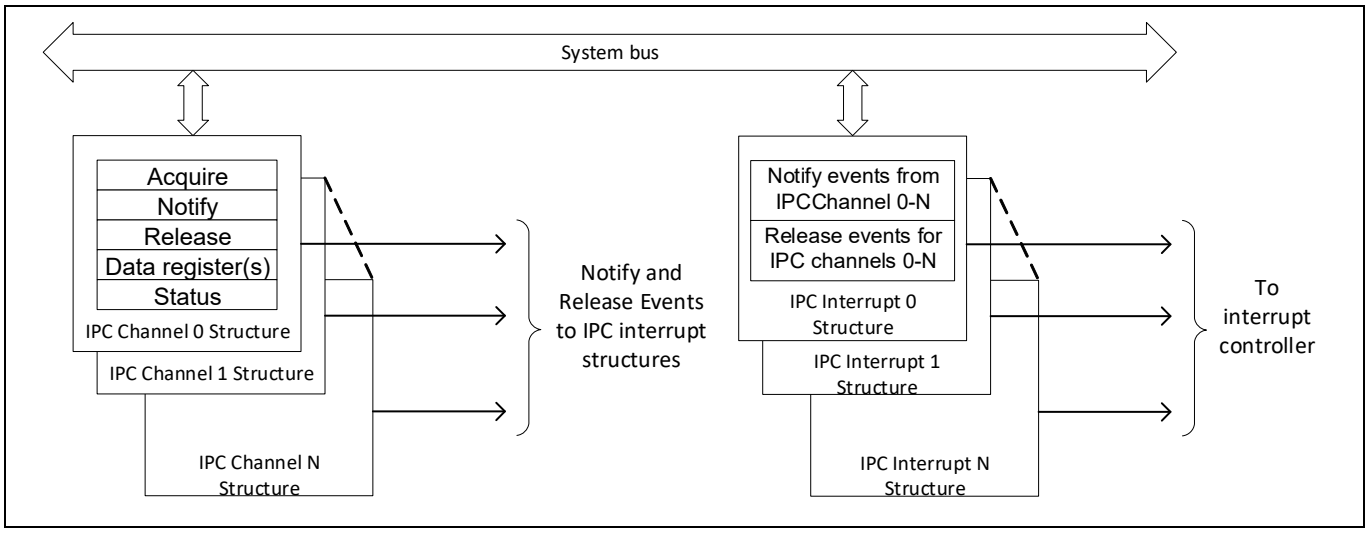


Figure 7-1. IPC register architecture

7.1 Features

The features of IPC are as follows:

- Implements locks for mutual exclusion between processors
- Allows sending messages between processors
- Supports up to sixteen channels for communication
- Supports up to sixteen interrupts, which can be triggered using notify or release events from the channels

7.2 Architecture

7.2.1 IPC channel

An IPC channel is implemented as six hardware registers, as shown in **Figure 7-2**. The IPC channel registers are accessible to all the 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 a '1', the read acquired the lock.

  If the SUCCESS field returns a '0', the read did not acquire the lock.
  Note that a single read access performs two functions:
  - The attempt to acquire a lock
  - Return the result of the acquisition attempt (SUCCESS field)
    The atomicity of these two functions is essential in a CPU with multiple tasks that can preempt each other. The register also has bitfields that provide information about the processor that acquired it. When acquired, this register is released by writing any value into the IPC_STRUCTx_RELEASE register. If the register was already in an acquired state another attempt to read the register will not be able to acquire it.
- **IPC_STRUCTx_NOTIFY**: This register is used to generate an IPC notify event. Each bit in this register corresponds to an IPC interrupt structure. The notify event generated from an IPC channel can trigger any or multiple interrupt structures.
- **IPC_STRUCTx_RELEASE**: Any write to this register will release the IPC channel. This register also has a bit that corresponds to each IPC interrupt structure. The release event generated from an IPC channel can trigger any or multiple interrupt structures. To only release the IPC channel and not generate an interrupt, you can write a zero into the IPC release register.
- **IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1**: These are two 32-bit registers with a combined size of 64 bits that are meant to hold data. These registers can be considered as the shared data memory for the channel. Typically, these registers will hold messages that need to be communicated between processors. If the messages are larger than the combined 64-bit size, place pointers in one or both of these registers.
- **IPC_STRUCTx_LOCK_STATUS**: This register provides the instantaneous lock status for the IPC channel. The register provides details if the channel is acquired. If acquired, it provides the processor's ID, protection context, and other details. The reading of lock status provides only an instantaneous status, which can be changed in the next cycle based on the activity of other processors on the channel.

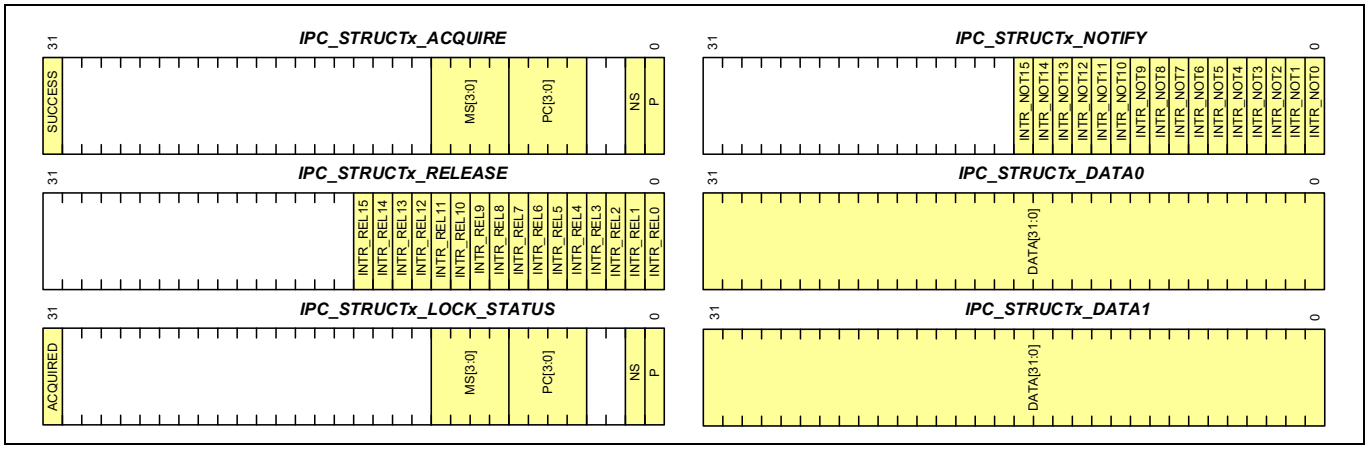


Figure 7-2. IPC channel structure

7.2.2 IPC interrupt

- Each IPC interrupt line in the system has a corresponding IPC interrupt structure. An IPC interrupt can be triggered by a notify or a release event from any of the IPC channels in the system. You can choose to mask any of the sources of these events using the IPC interrupt registers. **Figure 7-3** shows the registers in an IPC Interrupt structure.
- **IPC_INTR_STRUCTx_INTR:** This register provides the instantaneous status of the interrupt sources. Note that there are 16 notify and 16 release event bits in this register. These are the notify and release events corresponding to the 16 IPC channels. When a notify event is triggered in the IPC channel 0, the corresponding Notify0 bit is activated in the interrupt registers. A write of '1' to a bit will clear the interrupt.
- **IPC_INTR_STRUCTx_INTR_MASK:** The bit in this register masks the interrupt sources. Only the interrupt sources with their masks enabled can trigger the interrupt.
- **IPC_INTR_STRUCTx_INTR_SET:** A write of '1' into this register will set the interrupt.
- **IPC_INTR_STRUCTx_INTR_MASKED:** This register provides the instantaneous value of the interrupts after they are masked. The value in this register is (IPC_INTR_STRUCTx_INTR AND IPC_INTR_STRUCTx_INTR_MASK).

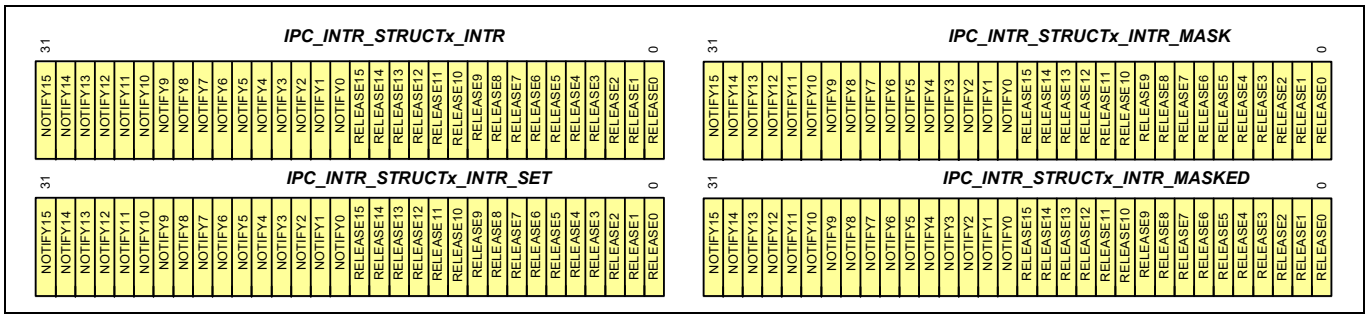


Figure 7-3. IPC interrupt structure

7.2.3 IPC channels and interrupts

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

Each IPC channel has a release and notify register, which can drive events on any of the IPC interrupts. An illustration of this relation between the IPC channels and the IPC interrupt structure is shown in **Figure 7-4**.

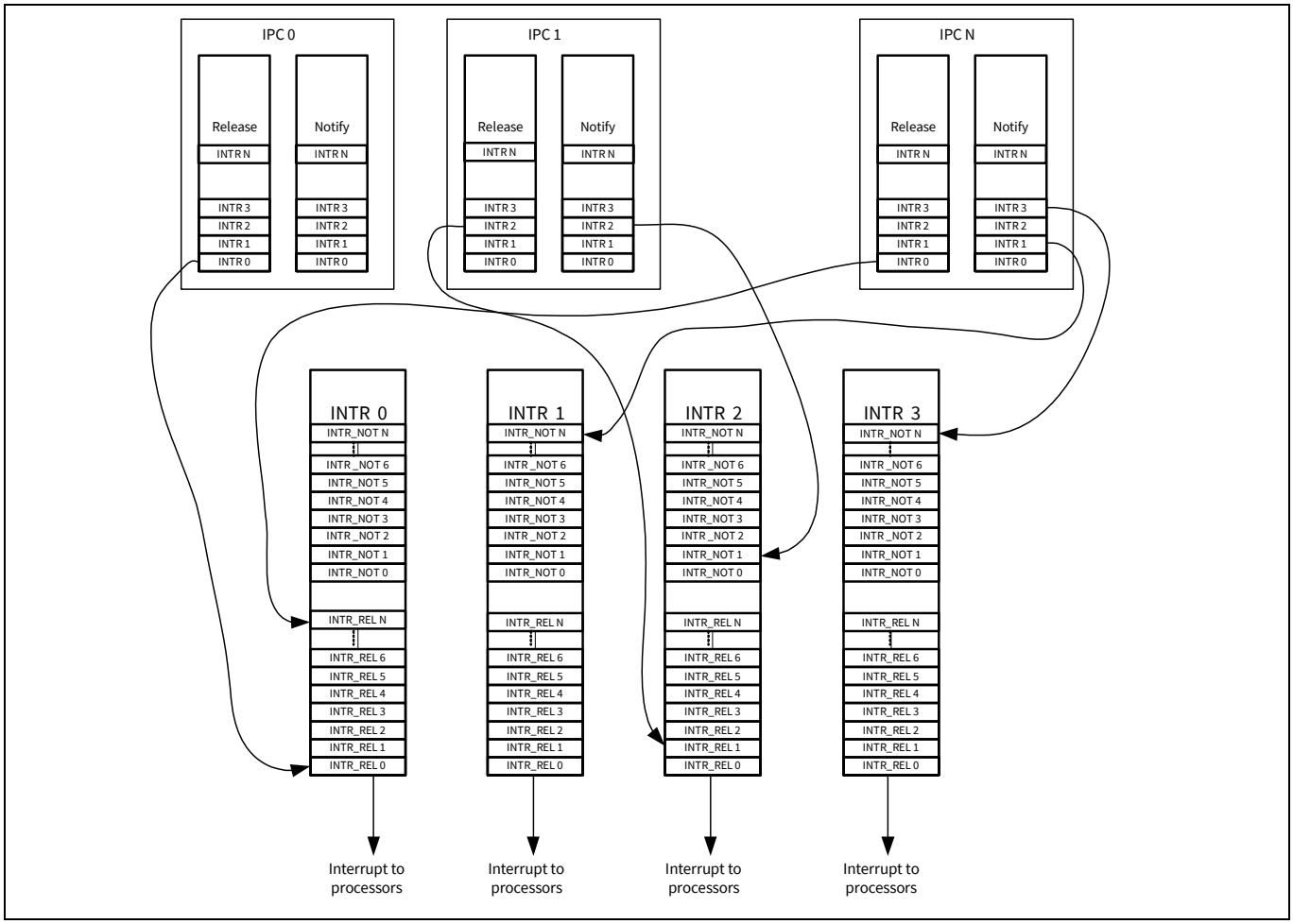


Figure 7-4. IPC channels and interrupts

7.2.4 Message passing

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

Figure 7-5 portrays an example of a sender (Processor A) sending data to a receiver (Processor B). IPC interrupt A is configured to interrupt Processor A. IPC interrupt B is configured to interrupt Processor B.

1. The sender will attempt to acquire the IPC channel by reading the IPC_STRUCTx_ACQUIRE register. If the channel was acquired, the sender has ownership of the channel for data transmission. This also changes the status of the channel and its corresponding IPC_STRUCTx_LOCK_STATUS register. If the channel was not acquired, the processor should wait until the channel is free for acquisition. This can be done by polling the IPC channel's IPC_STRUCTx_LOCK_STATUS register.
2. After the IPC channel is acquired, the sender has control of the channel for communication and places the 64-bit message data in the IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1 registers.
3. Now that the message is placed in the IPC channel, the sender generates a notify event on the receiver's interrupt line. It does this by setting the corresponding bit in the IPC channel's IPC_STRUCTx_NOTIFY register. This event creates a notify event at IPC interrupt B (IPC_INTR_STRUCTx_INTR). If the IPC channel's notify event was enabled by setting the mask bit (IPC_INTR_STRUCTx_INTR_MASK [31:23]) in the IPC interrupt B, this will generate an interrupt in the receiver.
4. When it receives IPC interrupt B, the receiver can poll the IPC_INTR_STRUCTx_INTR_MASKED register to understand which IPC channel had triggered the notify event. Based on this, the receiver identifies the channel to read and reads from the IPC channel's IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1 registers. The receiver has now received the data sent by the sender. It needs to release the channel so that other processors/processes can use it.
5. The receiver releases the channel. It also optionally generates a release event on the sender's IPC interrupt A. This will generate a release event interrupt on the sender if the corresponding channel release event was masked.

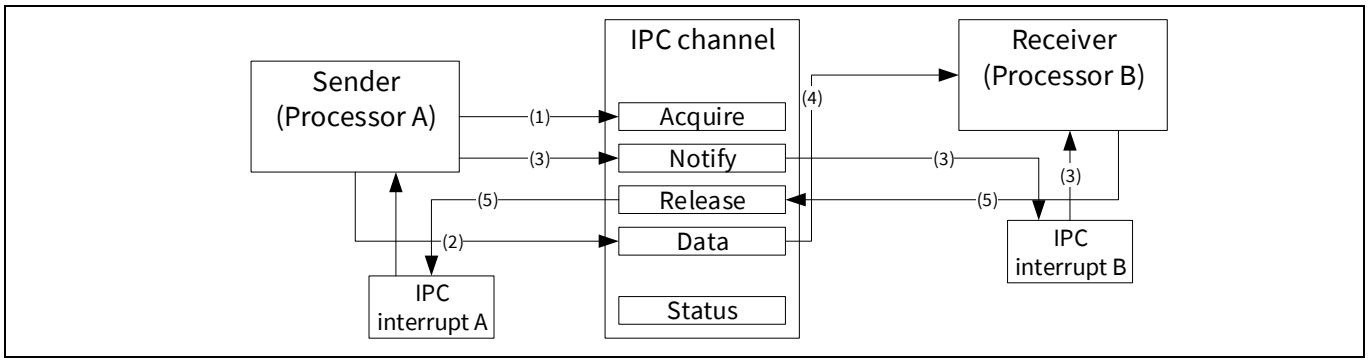


Figure 7-5. Sending messages using IPC

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

7.2.5 Implementing locks

The IPC channels can be used to implement locks. Locks are typically used in multi-core systems to implement some form of mutually exclusive access to a shared resource. When multiple processors share a resource, the processors are capable of acquiring and releasing the IPC channel. Therefore, the processor can assume an IPC channel as a lock. The semantics of this code is that access to the shared resource is gated by the processor's ownership of the channel. So the processors will need to acquire the IPC channel before they access the shared resource.

Failure to acquire the IPC channel signifies a lock on the shared resource because another processor has control of it. Note that the IPC channel will not enforce which processor acquires or releases the channel. All processors can acquire or release the IPC channel and the semantics of the code must make sure that the processor that acquires the channel is the one that releases it.

7.2.6 Message passing

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

Figure 7-6 portrays an example of a sender (Processor A) sending data to a receiver (Processor B). IPC interrupt A is configured to interrupt Processor A. IPC interrupt B is configured to interrupt Processor B.

1. The sender will attempt to acquire the IPC channel by reading the IPC_STRUCTx_ACQUIRE register. If the channel was acquired, the sender has ownership of the channel for data transmission. This also changes the status of the channel and its corresponding IPC_STRUCTx_LOCK_STATUS register. If the channel was not acquired, the processor should wait until the channel is free for acquisition. This can be done by polling the IPC channel's IPC_STRUCTx_LOCK_STATUS register.
2. After the IPC channel is acquired, the sender has control of the channel for communication and places the 64-bit message data in the IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1 registers.
3. Now that the message is placed in the IPC channel, the sender generates a notify event on the receiver's interrupt line. It does this by setting the corresponding bit in the IPC channel's IPC_STRUCTx_NOTIFY register. This event creates a notify event at IPC interrupt B (IPC_INTR_STRUCTx_INTR). If the IPC channel's notify event was enabled by setting the mask bit (IPC_INTR_STRUCTx_INTR_MASK [31:23]) in the IPC interrupt B, this will generate an interrupt in the receiver.
4. When it receives IPC interrupt B, the receiver can poll the IPC_INTR_STRUCTx_INTR_MASKED register to understand which IPC channel had triggered the notify event. Based on this, the receiver identifies the channel to read and reads from the IPC channel's IPC_STRUCTx_DATA0 and IPC_STRUCTx_DATA1 registers. The receiver has now received the data sent by the sender. It needs to release the channel so that other processors/processes can use it.
5. The receiver releases the channel. It also optionally generates a release event on the sender's IPC interrupt A. This will generate a release event interrupt on the sender if the corresponding channel release event was masked.

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

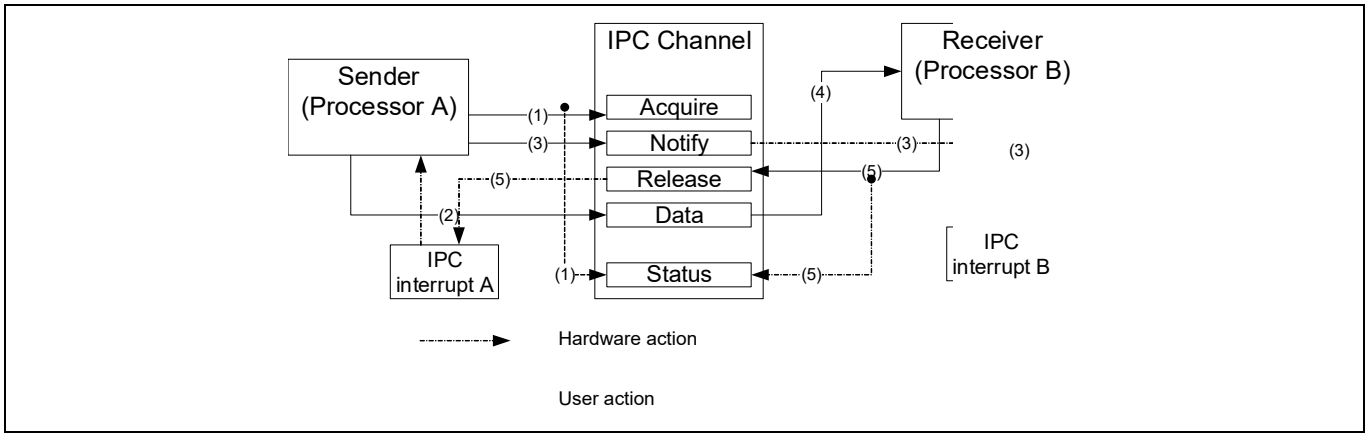


Figure 7-6. Sending messages using IPC

In the previous example, the size of the data being transmitted was just 64 bits. Larger messages can be sent as pointers. The sender can allocate a larger message structure in memory and pass the pointers in the data registers. **Figure 7-7** shows the usage. Note that the user code must implement the synchronization of the message read process.

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

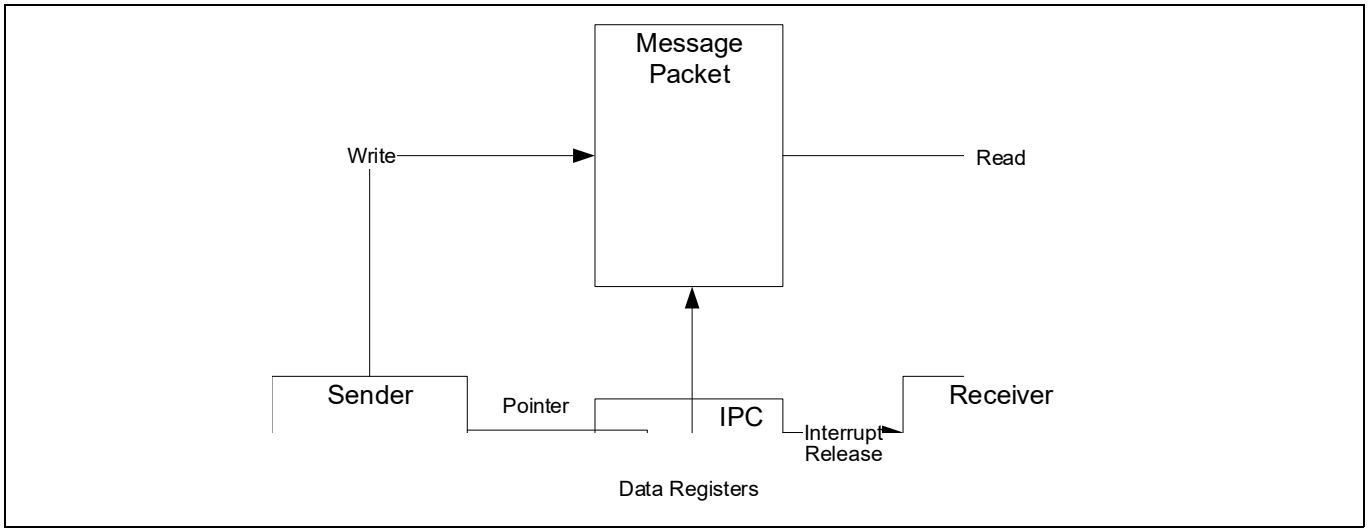


Figure 7-7. Communicating larger messages

7.2.7 Typical usage models

The unique channel and interrupt architecture of the EZ-USB™ FX10 IPC allows for a range of usage models for multicore communication. Some of these are listed here as an example. Note that the communication models possible based on the IPC architecture are not restricted to the ones listed in this document.

Note: This document only provides a high-level usage model and does not go into details of data management in the communication. This will need to be determined based on the specific application use case.

7.2.7.1 Full duplex communication

In this usage model, an IPC channel is used according to the direction of communication between cores. For managing events an IPC interrupt is used per core. In a dual core system this will translate to what is shown in **Figure 7-8**.

In this example, the IPC channel X is dedicated to data communication from Core 0 to Core 1 and IPC channel Y is for data communication from Core 1 to Core 0. The IPC interrupt X will signal events on Core 1. Hence its interrupt output is connected to Core 1's system interrupts. The events are triggered by writing into the IPC interrupts register structure over the system bus. Similarly, IPC interrupt Y is dedicated to Core 0.

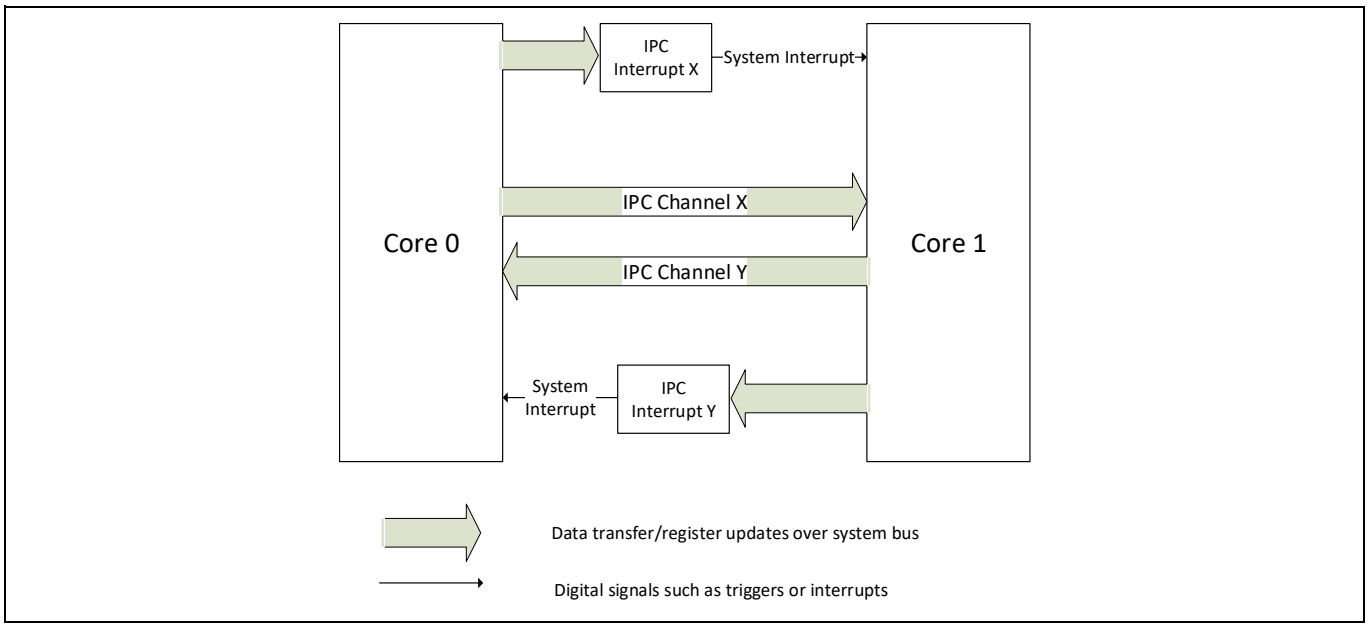


Figure 7-8. Full duplex IPC for dual core communication

7.2.7.2 Half duplex with independent event handling

In this model, only one IPC channel is used to set up the transfer between the two cores. This means that only one side controls data transfer at a time. The channels Lock register must be used to avoid contention of the one shared IPC channel. Two independent events are supported due to the two IPC interrupt structures being used. This model is shown in **Figure 7-9**.

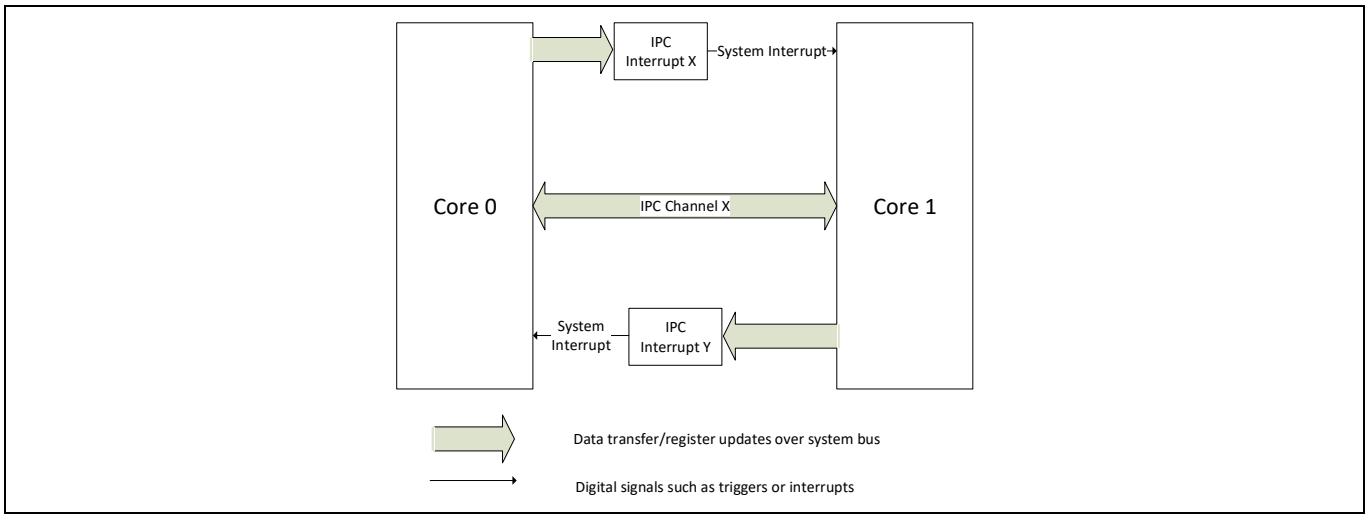


Figure 7-9. Half duplex with independent event handling

7.2.7.3 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, the access to the interrupt registers must be managed using the IPC lock of the channel. As shown in **Figure 7-10**, the IPC interrupt will be set up to trigger interrupts in both cores. Hence the individual core interrupts should have logic in its ISR to check which core is in control of the IPC and determine if the message and event was for that core.

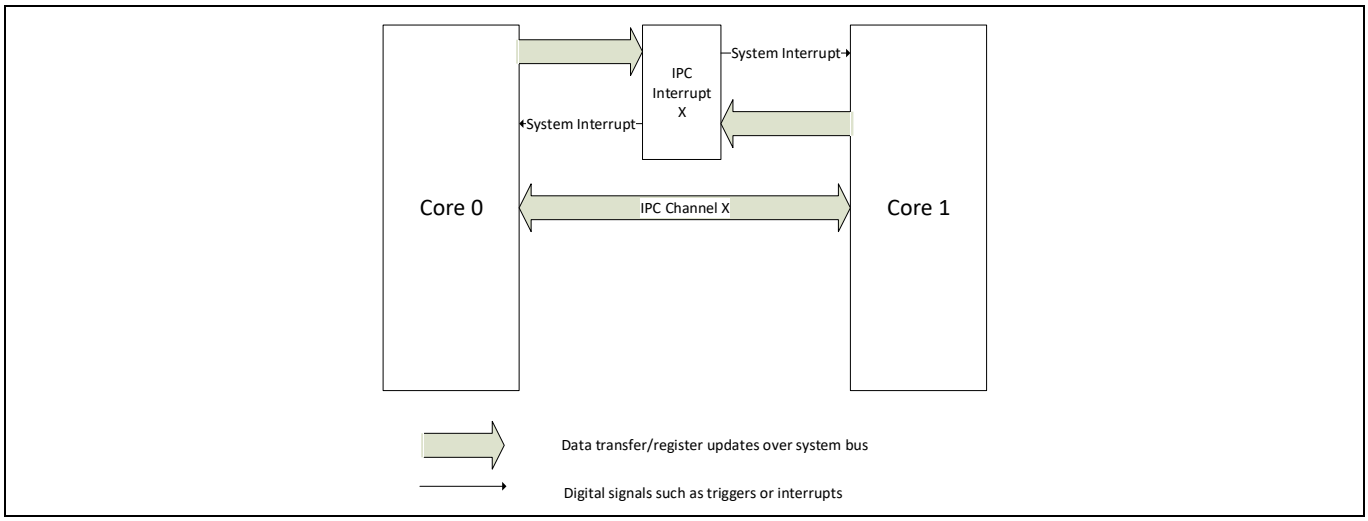


Figure 7-10. Half duplex with shared event handling

*Note: Some IPC channel and interrupt structures are reserved as part of the SROM code. See the SROM architecture and API in **"Flash memory programming"** on page 209 for a list IPC channels and interrupts being used by this API.*

8 Fault monitoring

The fault monitoring allows you to monitor various faults generated within the device and take actions based on the fault reported. The fault structures present in the EZ-USB™ FX10 MCU monitor access violation faults at protection units (MPU, SMPU, or PPU) and flash controller bus error/fault. In addition to reporting faults, the fault structures in EZ-USB™ FX10 MCUs provide a mechanism to log data from the fault sources and optionally perform soft reset.

The EZ-USB™ FX10 MCU family supports two centralized fault report/monitoring structures that monitor faults generated within the device. Each fault report structure can monitor and report faults from up to 96 sources.

8.1 Features

Each EZ-USB™ FX10 MCU fault report structure supports:

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

8.2 Architecture

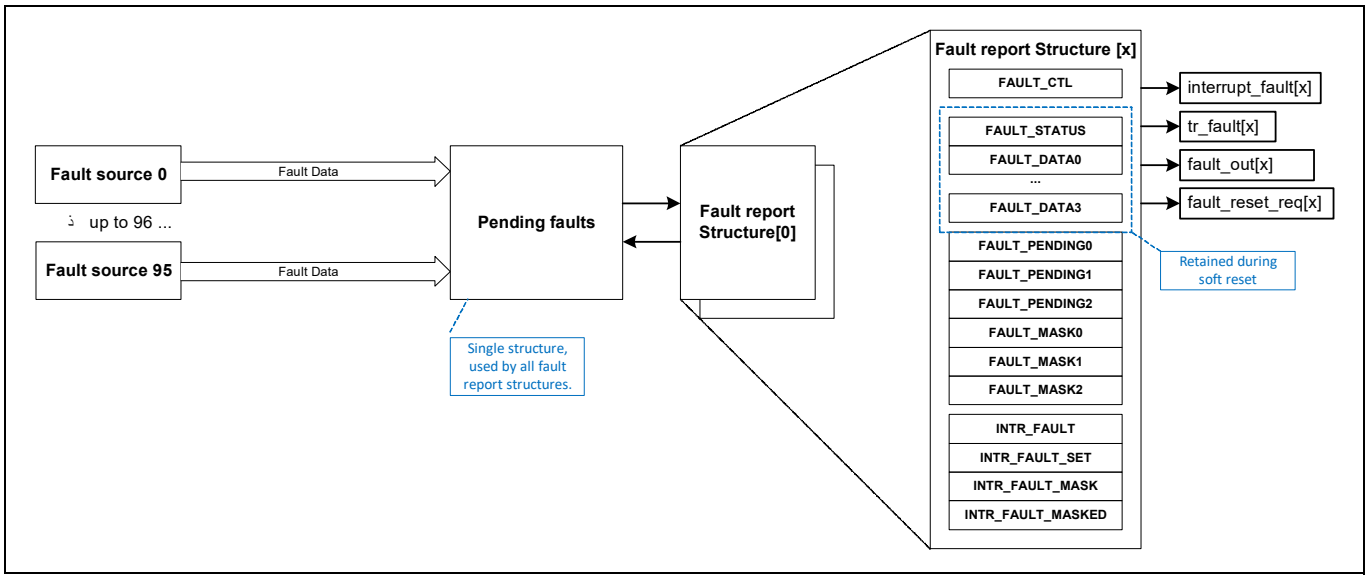


Figure 8-1. Fault report structure

The EZ-USB™ FX10 MCU family uses centralized fault report structures. This centralized nature allows for a system wide handling of faults simplifying firmware development. Only a single fault interrupt handler is required to monitor multiple faults. The fault report structure provides the fault source and additional fault specific information through a single set of registers; no iterative search for the fault source and fault information is required.

The fault structure can be configured to capture one or more faults as listed in **Table 8-2**. When a fault structure is configured to capture a specific fault, an occurrence of that fault will be recorded as a pending fault. If the fault structure has finished processing all other faults or if there are no other pending faults, the fault data will be captured into the fault structure registers. In addition, a successful capture can trigger an interrupt and be processed by either Cortex®-M4 or Cortex®-M0+ depending on the application requirement.

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

8.2.1 Fault report

The EZ-USB™ FX10 MCU family supports two fault report structures. Each fault report structure has a dedicated set of control and status registers. Each fault report structure captures a single fault. The captured fault information includes:

- Fault validity bit that indicates a fault is captured (VALID bit [31] of the FAULT_STRUCTx_STATUS register). This bit is set whenever a fault is captured. The bit should be cleared after processing the fault information. New faults are captured only when this bit is '0'.
- Fault index, as shown in **Table 8-2**, identifies the fault source (IDX bits [6:0] of FAULT_STRUCTx_STATUS)
- Additional fault information describing fault specifics (FAULT_STRUCTx_DATA0 through FAULT_STRUCTx_DATA3 registers). This additional information is fault source specific. For example, an MPU protection violation provides information on the violating bus address, the bus master identifier, and bus access control information in only two FAULT_DATA registers. The details of the fault information for various faults is explained in **Table 8-1**.

Table 8-1. Fault information

Fault source	Fault information
MPU/SMPU violation	DATA0[31:0]: Violating address DATA1[0]: User read DATA1[1]: User write DATA1[2]: User execute DATA1[3]: Privileged read DATA1[4]: Privileged write DATA1[5]: Privileged execute DATA1[6]: Non-secure DATA1[11:8]: Master identifier DATA1[15:12]: Protection context identifier DATA1[31]: '0' MPU violation; '1': SMPU violation
Master interface PPU violation	DATA0[31:0]: Violating address DATA1[0]: User read DATA1[1]: User write DATA1[2]: User execute DATA1[3]: Privileged read DATA1[4]: Privileged write DATA1[5]: Privileged execute DATA1[6]: Non-secure DATA1[11:8]: Master identifier DATA1[15:12]: Protection context identifier DATA1[31]: '0': PPU violation, '1': peripheral bus error

Table 8-1. Fault information (continued)

Fault source	Fault information
Peripheral group PPU violation	DATA0[31:0]: Violating address DATA1[0]: User read DATA1[1]: User write DATA1[2]: User execute DATA1[3]: Privileged read DATA1[4]: Privileged write DATA1[5]: Privileged execute DATA1[6]: Non-secure DATA1[11:8]: Master identifier DATA1[15:12]: Protection context identifier DATA1[31:30]: '0': PPU violation, '1': timeout detected, '2': peripheral bus error
Flash controller bus error	FAULT_DATA0[31:0]: Violating address FAULT_DATA1[31]: '0': FLASH macro interface bus error; '1': memory hole FAULT_DATA1[15:12]: Protection context identifier FAULT_DATA1[11:8]: Master identifier

8.2.2 Signaling interface

In addition to captured fault information, each fault report structure supports a signaling interface to notify the system about the captured fault. The interface of fault report structure 'x' supports the following:

- **A fault interrupt (interrupt_fault[x]):** Use the FAULT_STRUCTx_INTR, FAULT_STRUCTx_INTR_SET, FAULT_STRUCTx_INTR_MASK and FAULT_STRUCTx_INTR_MASKED registers to monitor, set, and mask the FAULT_STRUCTURE[x]'s interrupt. Only a single interrupt cause is available, which indicates that a fault is detected. The fault report registers can be read in the interrupt handler to deduce the fault. The FAULT bit [0] of the FAULT_STRUCTx_INTR_MASK register provides a mask/enable for the interrupt. The FAULT bit [0] of the FAULT_STRUCTx_INTR register is set to '1' when a fault is captured. Setting this bit in firmware clears the interrupt.
- **A DMA trigger (tr_fault[x]):** The fault structure generates a DMA trigger when VALID bit [31] of the FAULT_STRUCTx_STATUS register is set. To enable the trigger, set the TR_EN bit [0] of the FAULT_STRUCTx_CTL register. The trigger can be connected to a DMA controller, which can transfer captured fault information from the fault report structure to memory and can clear the VALID bit [31] of the FAULT_STRUCTx_STATUS register. See the **"Trigger multiplexer block"** on page 344 for more details.
- **A chip output signal (fault_out[x]):** The fault structure generates an output signal, which is set when VALID bit [31] of the FAULT_STRUCTx_STATUS register is set. This signal can be routed out of the device through the HSIOM (refer to the device datasheet. The output signal is enabled by setting the OUT_EN bit [1] of the FAULT_STRUCTx_CTL register. The output signal can be used to communicate non-recoverable faults to off-chip components (possibly resulting in a reset of the chip).
- **A fault reset request signal (fault_reset_req[x]):** The fault structure generates a soft reset when VALID bit [31] of the FAULT_STRUCTx_STATUS register is set. The reset capability is enabled by setting RESET_REQ_EN bit [2] of FAULT_STRUCTx_CTL. The reset request performs a soft reset. This reset is captured as RESET_ACT_FAULT in the SRSS_RES_CAUSE register. The fault information in FAULT_STRUCTx_STATUS and FAULT_STRUCTx_DATA registers is retained through this reset.

Fault monitoring

Because the device has a single fault_reset_req signal, the individual fault_reset_req[x] signals from the fault structures are combined into a single fault_reset_req signal as shown in **Figure 8-2**.

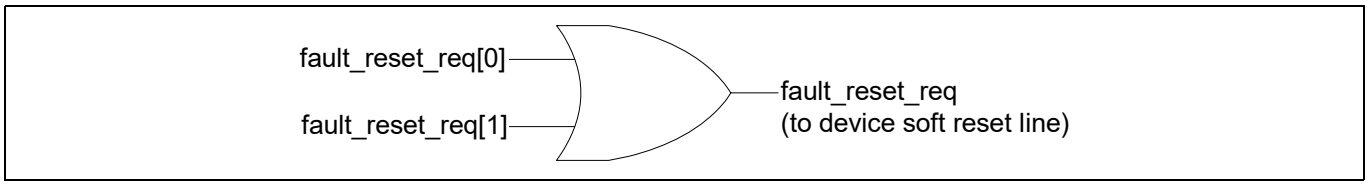


Figure 8-2. Fault reset request

8.2.3 Monitoring

A central structure, which is shared by all fault report structures, keeps track of all pending faults in the system. The FAULT_STRUCTx_PENDINGx registers reflect what fault sources are pending and provide a single pending bit for up to 96 fault sources. The registers are mirrored in all the fault report structures; that is, they read the same value in all fault structures. The bit indexing in the registers follow the fault index captured in **Table 8-2**. For instance, bit [0] of FAULT_STRUCTx_PENDING0 captures a CM0+ MPU/SMPU violation and bit [1] of FAULT_STRUCTx_PENDING1 captures a peripheral group#1 PPU violation.

The pending faults are faults that are not yet captured by a fault structure. When a pending fault is captured by a fault structure, the associated pending bit is cleared to '0'.

Each fault report structure is selective in the faults it captures. The FAULT_STRUCTx_MASK0, FAULT_STRUCTx_MASK1, FAULT_STRUCTx_MASK2 registers of a fault structure decide the pending faults that it captures. These faults are referred to as "enabled" faults. The FAULT_STRUCTx_MASK registers are unique to each fault structure. This allows for the following:

- One fault report structure is used to capture recoverable faults and one fault report structure is used to capture non-recoverable faults. The former can be used to generate a fault interrupt and the latter can be used to activate a chip output signal and/or activate a reset request.
- Two fault report structures are used to capture the same faults. The first fault is captured by the structure with the lower index (for example, fault structure 0) and the second fault is captured by the structure with the higher index (for example, fault structure 1). Note that both structures cannot capture the same fault at the same time. As soon as a fault is captured, the pending bit is cleared and the other structure will not be aware of the fault. Fault structure 0 has precedence over fault structure 1.

The fault structure captures "enabled" faults only when VALID bit [31] of FAULT_STRUCTx_STATUS register is '0'. When a fault is captured, hardware sets the VALID bit [31] of the FAULT_STRUCTx_STATUS register. In addition, hardware clears the associated pending bit to '0'. When a fault structure is processed, firmware or a DMA transfer should clear the VALID bit [31] of the FAULT_STRUCTx_STATUS register. Note that fault capturing does not consider FAULT bit [0] of FAULT_STRUCTx_INTR register and firmware should clear the bit after servicing the interrupt, if the interrupt is enabled.

8.2.4 Low-power mode operation

The fault report structure functionality is available in Active and Sleep (and the LP counterparts) power modes only. The interfaces between the fault sources and fault report structures are reset in the Deep Sleep power mode. Because the fault report structure is an active functionality, pending faults (in the FAULT_STRUCTx_PENDING registers) are not retained when transitioning to Deep Sleep power mode. The fault structure's registers can be partitioned based on the reset domain and their retention capability as follows:

- Active reset domain: FAULT_STRUCTx_PENDING, FAULT_STRUCTx_INTR, FAULT_STRUCTx_INTR_SET, and FAULT_STRUCTx_INTR_MASKED registers. These registers are not retained in Deep Sleep power mode.
- Deep Sleep reset domain: FAULT_STRUCTx_CTL, FAULT_STRUCTx_MASK, and FAULT_STRUCTx_INTR_MASK registers. These registers are retained in Deep Sleep power mode but any system reset will reset these registers to the default state.
- Hard reset domain: FAULT_STRUCTx_STATUS and FAULT_STRUCTx_DATA registers. These registers are retained through soft resets (detectable in SRSS_RES_CAUSE registers). However, hard resets such as XRES/POR/BOD will reset the registers.

8.2.5 Using a fault structure

Follow these steps to configure and use a fault structure:

1. Identify the faults from **Table 8-2** to be monitored in the system.
2. For firmware fault handling through interrupts:
   a) Set the FAULT bit [0] of the FAULT_STRUCTx_INTR_MASK register
   b) Set the FAULT bit [0] of the FAULT_STRUCTx_INTR register to clear any pending interrupt
   c) Enable the FAULTx interrupt to the CPU by configuring the appropriate ISER register. Refer to the **"Interrupts"** on page 89
3. For fault handling through DMA:
   a) Set the TR_EN bit [0] of the FAULT_STRUCTx_CTL register
   b) Route the tr_fault[x] signal to the trigger input DMA controller. See the **"Trigger multiplexer block"** on page 344
   c) Configure and enable the DMA controller to transfer FAULT_STRUCTx_STATUS and FAULT_STRUCTx_DATA registers to memory and write back '0' to FAULT_STRUCTx_STATUS register after the transfer is complete. Refer to the **"Data wire controller (DW)"** on page 141
4. For fault handling outside the device:
   a) Set the OUT_EN bit [1] of FAULT_STRUCTx_CTL register
   b) Route the fault_out[x] signal to a pin through HSIOM. Refer to the device datasheet for more details
   c) Use the signal externally for processing the fault – generate external reset, power cycle, or log fault information
5. Set the RESET_REQ_EN bit [2] of the FAULT_STRUCTx_CTL register, if a soft reset is required on any fault detection in the structure.
6. Clear VALID bit [31] of the FAULT_STRUCTx_STATUS register to clear any fault captured.
7. Set the fault index bits in the FAULT_STRUCTx_MASK registers for faults that need to be captured by the fault structure as explained in **"Monitoring"** on page 84.

8.2.6 CPU exceptions versus fault monitoring

Some faults captured in **Table 8-2** also result in bus errors or CPU exceptions (Cortex®-M4 Bus/Usage/Memory/Hard faults). The faults can be communicated in two ways:

- As a bus error to the master of the faulting bus transfer. This will result in Bus, Usage, Memory, or Hard fault exceptions in the CPU.
- As a fault in a fault report structure. This fault can be communicated as a fault interrupt to any processor in the system. This allows fault handling on a processor that is not the master of the faulting bus transfer. It is useful for faults that cause the master of the faulting transfer to become unresponsive or unreliable.

8.3 Fault sources

The fault sources can vary between device families. **Table 8-2** provides the list of fault sources available in EZ-USB™ FX10 MCUs.

Table 8-2. Fault sources

Fault index	Source	Description
0	cpuss.mpu_vio[0]	CM0+ MPU/SMPU violation
1	cpuss.mpu_vio[1]	CRYPTO MPU/SMPU violation
2	cpuss.mpu_vio[2]	DW0 MPU/SMPU violation
3	cpuss.mpu_vio[3]	DW1 MPU/SMPU violation
4	cpuss.mpu_vio[4]	DMAC MPU/SMPU violation
6	–	–
7 to 15	Reserved	–
16	cpuss.mpu_vio[16]	CM4 MPU/SMPU violation (System bus)
17	cpuss.mpu_vio[17]	CM4 code error
18	cpuss.mpu_vio[18]	CM4 code flash controller errors
19 to 27	Reserved	–
28	peri.ms_vio[0]	CM0+ Peripheral Master Interface PPU violation
29	peri.ms_vio[1]	CM4 Peripheral Master Interface PPU violation
30	peri.ms_vio[2]	DW0 Peripheral Master Interface PPU violation
31	peri.ms_vio[3]	DW1 Peripheral Master Interface PPU violation
32	peri.group_vio[0]	Peripheral group #0 (Peripheral Clock dividers, Trigger Mux etc) PPU violation Register address range: 0x40000000 to 0x40100000
33	peri.group_vio[1]	Peripheral group #1 (Crypto block) PPU violation Register address range: 0x40100000 to 0x40200000
34	peri.group_vio[2]	Peripheral group #2 (CPUSS, SRSS, EFUSE) PPU violation Register address range: 0x40200000 to 0x40300000
35	peri.group_vio[3]	Peripheral group #3 (IOSS, LPCOMP, CSD, TCPWM, LCD, BLE) PPU violation Register address range: 0x40300000 to 0x40400000
36	peri.group_vio[4]	Peripheral group #4 (SMIF) PPU violation Register address range: 0x40400000 to 0x40500000
37	–	–

Table 8-2. Fault sources (continued)

Fault index	Source	Description
38	peri.group_vio[6]	Peripheral group #6 (SCB) PPU violation Register address range: 0x40600000 to 0x40700000
39	Reserved	–
40	Reserved	–
41	peri.group_vio[9]	Peripheral group #9 (PASS) PPU violation Register address range: 0x41000000 to 0x42000000
42	–	–
43 to 47	Reserved	–
48	cpuss.flashc_main_bus_err	Flash controller bus error
49 to 95	Reserved	–

8.4 Register list

Name	Description
FAULT_STRUCTx_CTL	Fault control register for enabling DMA trigger, fault output, and fault reset signals
FAULT_STRUCTx_STATUS	Fault status register that stores the validity and fault index of the currently captured fault
FAULT_STRUCTx_DATA0	Fault data register 0 that stores fault information associated with the currently captured fault
FAULT_STRUCTx_DATA1	Fault data register 1 that stores fault information associated with the currently captured fault
FAULT_STRUCTx_DATA2	Fault data register 2 that stores fault information associated with the currently captured fault
FAULT_STRUCTx_DATA3	Fault data register 3 that stores fault information associated with the currently captured fault
FAULT_STRUCTx_PENDING0	Fault pending register 0 that stores pending (not captured) faults with fault index from 0 to 31
FAULT_STRUCTx_PENDING1	Fault pending register 1 that stores pending (not captured) faults with fault index from 32 to 63
FAULT_STRUCTx_PENDING2	Fault pending register 2 that stores pending (not captured) faults with fault index from 64 to 95
FAULT_STRUCTx_MASK0	Fault mask register 0 that enables the capture of pending faults with fault index from 0 to 31 by the fault structure
FAULT_STRUCTx_MASK1	Fault mask register 1 that enables the capture of pending faults with fault index from 32 to 63 by the fault structure
FAULT_STRUCTx_MASK2	Fault mask register 2 that enables the capture of pending faults with fault index from 64 to 95 by the fault structure
FAULT_STRUCTx_INTR	Fault interrupt register that stores the unmasked status of the fault structure’s interrupt
FAULT_STRUCTx_INTR_SET	Fault interrupt set register used to set the fault structure's interrupt through firmware
FAULT_STRUCTx_INTR_MASK	Fault interrupt mask register that masks fault interrupt
FAULT_STRUCTx_INTR_MASKED	Fault interrupt register that stores the masked status of the fault structure's interrupt

9 Interrupts

The EZ-USB™ FX10 MCU family supports interrupts and CPU exceptions on both Cortex®-M4 and Cortex®-M0+ cores. Any condition that halts normal execution of instructions is treated as an exception by the CPU. Thus an interrupt request is treated as an exception. However, in the context of this chapter, interrupts refer to those events generated by peripherals external to the CPU such as timers, serial communication block, and port pin signals; exceptions refer to those events that are generated by the CPU such as memory access faults and internal system timer events. Both interrupts and exceptions result in the current program flow being stopped and the exception handler or interrupt service routine (ISR) being executed by the CPU. Both Cortex®-M4 and Cortex®-M0+ cores provide their own unified exception vector table for both interrupt handlers/ISR and exception handlers.

9.1 Features

The EZ-USB™ FX10 MCU supports the following interrupt features:

- Supports 133 system interrupts
  - 133 Arm® Cortex®-M4 interrupts
  - Eight Arm® Cortex®-M0+ external interrupts and eight Arm® Cortex®-M0+ internal (software only) interrupts. The CPU supports up to 32 interrupts, but only 16 interrupts are used by EZ-USB™ FX10 interrupt infrastructure. The eight external CPU interrupts support Deep Sleep (WIC) functionality.
  - Four system interrupts can be mapped to each of the CPU non-maskable interrupt (NMI)
  - Up to 37 interrupt sources capable of waking the device from Deep Sleep power mode
- Nested vectored interrupt controller (NVIC) integrated with each CPU core, yielding low interrupt latency
- Wakeup interrupt controller (WIC) enabling interrupt detection (CPU wakeup) in Deep Sleep power mode
- Vector table may be placed in either flash or SRAM
- Configurable priority levels (eight levels for Cortex®-M4 and four levels for Cortex®-M0+) for each interrupt
- Level-triggered and pulse-triggered interrupt signals

9.2 Architecture

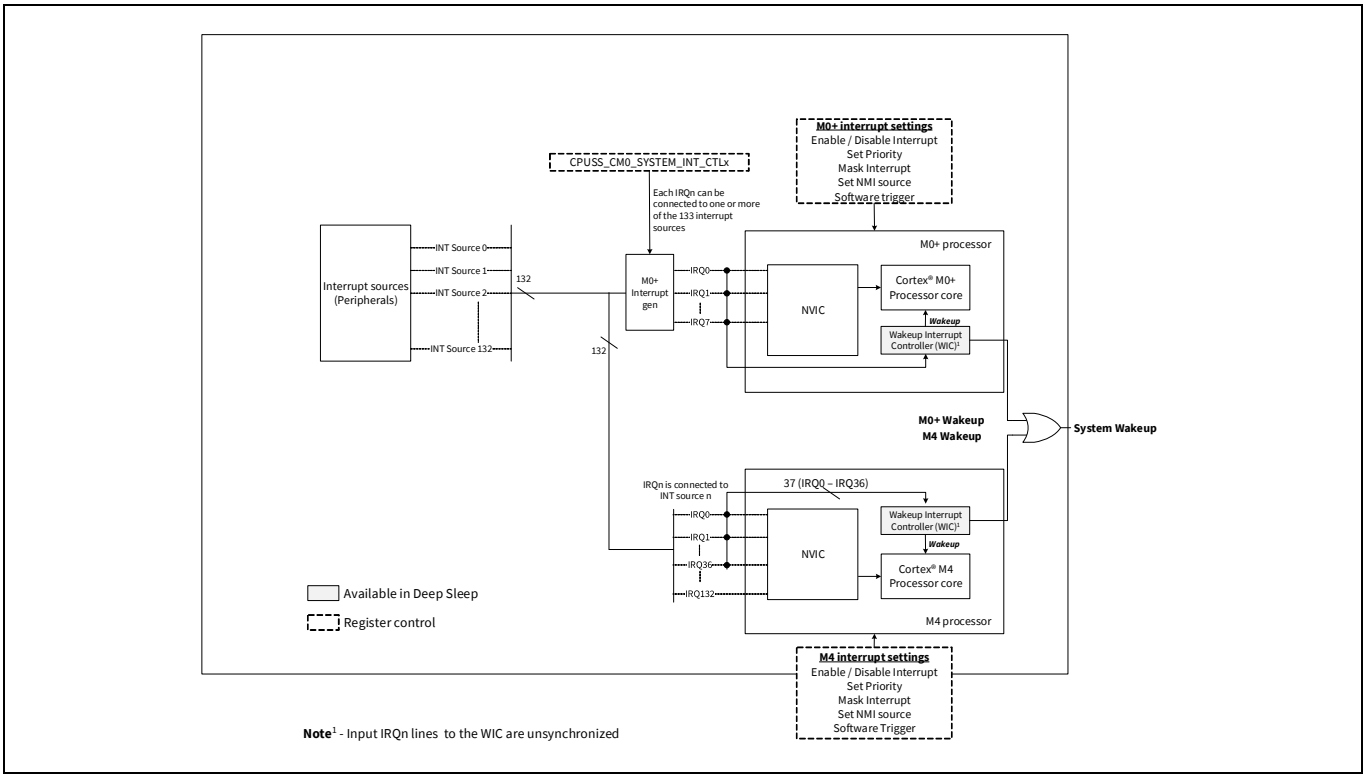


Figure 9-1. Interrupts block diagram

Figure 9-1 shows the EZ-USB™ FX10 MCU interrupt architecture. The EZ-USB™ FX10 MCU has 133 system interrupts that are generated by various peripherals. These interrupt signals are processed by the NVIC of the individual core. In the Cortex®-M4 core, the system interrupt source 'n' is directly connected to IRQn. The Cortex®-M0+ interrupt architecture uses eight CPU interrupts IRQ[7:0] out of the 32 available IRQn lines of the core. In the Cortex®-M0+ core, the system interrupt source connection to a particular IRQn of the core is configurable and any of the 133 system interrupts can be mapped to any of the IRQ[7:0]. This ensures that all the system interrupts can be mapped onto any CPU interrupt simultaneously. See the **"Interrupt sources"** on page 97 for more details about the system interrupt to CPU interrupt mapping. The NVIC takes care of enabling/disabling individual interrupt IRQs, priority resolution, and communication with the CPU core. The other exceptions such as NMI and hard faults are not shown in **Figure 9-1** because they are part of CPU core generated events, unlike interrupts, which are generated by peripherals external to the CPU.

In addition to the NVIC, the EZ-USB™ FX10 MCU supports wakeup interrupt controllers (WIC) for each CPU and a shared interrupt synchronization block that synchronizes the interrupts to CLK_HF domain (adds two CLK_HF cycles delay for synchronization). The WIC provides detection of Deep Sleep interrupts in the Deep Sleep CPU power mode. Each CPU can individually be in Deep Sleep mode; the device is said to be in Deep Sleep mode only when both the CPUs are in Deep Sleep mode. See the **"Device power modes"** on page 281 for details. The Cortex®-M4 WIC block supports up to 37 interrupts that can wake up the CPU from Deep Sleep power mode. The Cortex®-M0+ WIC block supports all eight interrupts. The device exits Deep Sleep mode (system wakeup signal in **Figure 9-1**) as soon as one CPU wakes up. The synchronization blocks synchronize the interrupts to the CPU clock frequency as the peripheral interrupts can be asynchronous to the CPU clock frequency.

9.3 Interrupts and exceptions - operation

9.3.1 Interrupt/exception handling

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

1. Assuming that all the interrupt and exception signals are initially LOW (idle or inactive state) and the processor is executing the main code, a rising edge on any one of the signals is registered by the NVIC, if the interrupt or exception is enabled to be serviced by the CPU. The signal is now in a pending state waiting to be serviced by the CPU.
2. On detecting the signal from the NVIC, the CPU stores its current context by pushing the contents of the CPU registers onto the stack.
3. The CPU also receives the exception number of the triggered interrupt from the NVIC. All interrupts and exceptions have a unique exception number, as given in **Table 9-1**. By using this exception number, the CPU fetches the address of the specific exception handler from the vector table.
4. The CPU then branches to this address and executes the exception handler that follows.
5. Upon completion of the exception handler, the CPU registers are restored to their original state using stack pop operations; the CPU resumes the main code execution.

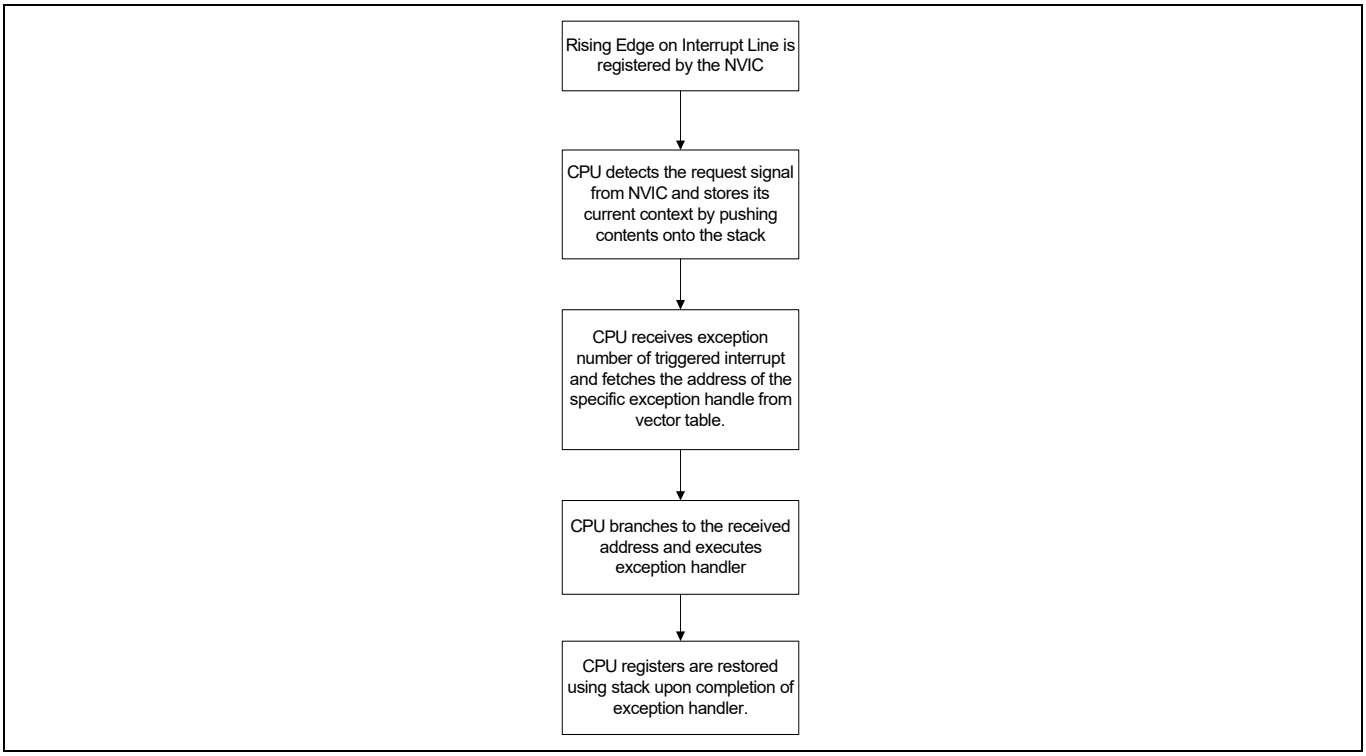


Figure 9-2. Interrupt handling when triggered

When the NVIC receives an interrupt request while another interrupt is being serviced or receives multiple interrupt requests at the same time, it evaluates the priority of all these interrupts, sending the exception number of the highest priority interrupt to the CPU. Thus, a higher priority interrupt can block the execution of a lower priority ISR at any time.

Exceptions are handled in the same way that interrupts are handled. Each exception event has a unique exception number, which is used by the CPU to execute the appropriate exception handler.

9.3.2 Level and pulse interrupts

Both CM0+ and CM4 NVICs support level and pulse signals on the interrupt lines (IRQn). The classification of an interrupt as level or pulse is based on the interrupt source.

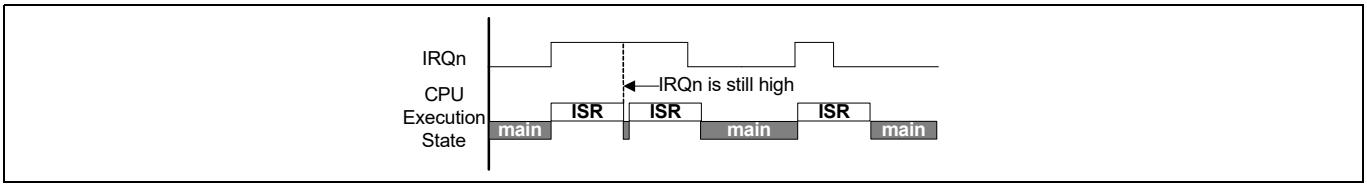


Figure 9-3. Level interrupts

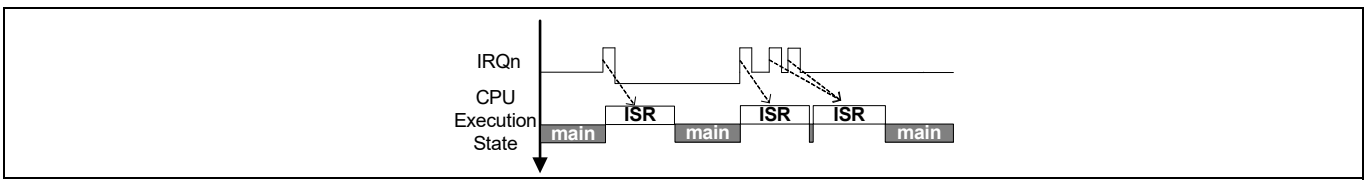


Figure 9-4. Pulse interrupts

Interrupts

Figure 9-3 and **Figure 9-4** show the working of level and pulse interrupts, respectively. Assuming the interrupt signal is initially inactive (logic LOW), the following sequence of events explains the handling of level and pulse interrupts:

1. On a rising edge event of the interrupt signal, the NVIC registers the interrupt request. The interrupt is now in the pending state, which means the interrupt requests have not yet been serviced by the CPU.
2. The NVIC then sends the exception number along with the interrupt request signal to the CPU. When the CPU starts executing the ISR, the pending state of the interrupt is cleared.
3. For pulse interrupts, when the ISR is being executed by the CPU, one or more rising edges of the interrupt signal are logged as a single pending request. The pending interrupt is serviced again after the current ISR execution is complete (see **Figure 9-4** for pulse interrupts).
4. For level interrupts, if the interrupt signal is still high after completing the ISR, it will be pending and the ISR is executed again. **Figure 9-3** illustrates this for level triggered interrupts, where the ISR is executed as long as the interrupt signal is HIGH.

9.3.3 Exception vector table

The exception vector tables (**Table 9-1** and **Table 9-2**) store the entry point addresses for all exception handlers in Cortex®-M0+ and Cortex®-M4 cores. The CPU fetches the appropriate address based on the exception number.

Table 9-1. M0+ exception vector table

Exception number	Exception	Exception priority	Vector address
–	Initial Stack Pointer Value	Not applicable (NA)	Start_Address = 0x0000 or CM0P_SCS_VTOR[a)]
1	Reset	–3, the highest priority	Start_Address + 0x04
2	Non Maskable Interrupt (NMI)	–2	Start_Address + 0x08
3	HardFault	–1	Start_Address + 0x0C
4-10	Reserved	NA	Start_Address + 0x10 to Start_Address + 0x28
11	Supervisory Call (SVCall)	Configurable (0 – 3)	Start_Address + 0x2C
12-13	Reserved	NA	Start_Address + 0x30 to Start_Address + 0x34
14	PendSupervisory (PendSV)	Configurable (0 – 3)	Start_Address + 0x38
15	System Timer (SysTick)	Configurable (0 – 3)	Start_Address + 0x3C
16	External Interrupt (IRQ0)	Configurable (0 – 3)	Start_Address + 0x40
…	…	Configurable (0 – 3)	…
31	External Interrupt (IRQ15)	Configurable (0 – 3)	Start_Address + 0x7C

a) Start Address = 0x0000 on reset and is later modified by firmware by updating the CM0P_SCS_VTOR register.

Table 9-2. Cortex®-M4 exception vector table

Exception number	Exception	Exception priority	Vector address
–	Initial stack pointer value	–	Start_Address = 0x0000 or CM4_SCS_VTOR[a)]
1	Reset	–3, highest priority	Start _Address + 0x0004
2	Non Maskable Interrupt (NMI)	–2	Start _Address + 0x0008
3	Hard fault	–1	Start _Address + 0x000C
4	Memory management fault	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)	Configurable (0–7)	Start _Address + 0x002C
12–13	Reserved	–	–
14	Pend Supervisory (PendSV)	Configurable (0–7)	Start _Address + 0x0038
15	System Tick timer (SysTick)	Configurable (0–7)	Start _Address + 0x003C
16	External interrupt (IRQ0)	Configurable (0–7)	Start _Address + 0x0040
….	….	….	….
156	External interrupt (IRQ140)	Configurable (0–7)	Start _Address + 0x02D8
157	External interrupt (IRQ141)	Configurable (0–7)	Start _Address + 0x02DC

a) Start Address = 0x0000 on reset and is later modified by firmware by updating CM4_SCS_VTOR register.

In **Table 9-1** and **Table 9-2**, the first word (4 bytes) is not marked as exception number zero. This is because the first word in the exception table is used to initialize the main stack pointer (MSP) value on device reset; it is not considered as an exception. In the EZ-USB™ FX10 MCU, both the vector tables can be configured to be located either in flash memory or SRAM. The vector table offset register (VTOR) present as part of Cortex®-M0+ and Cortex®-M4 system control space registers configures the vector table offset from the base address (0x0000). The CM0P_SCS_VTOR register sets the vector offset address for the CM0+ core and CM4_SCS_VTOR sets the offset for the M4 core. The VTOR value determines whether the vector table is in flash memory (0x10000000 to 0x10100000) or SRAM (0x08000000 to 0x08048000). Note that the VTOR registers can be updated only in privilege CPU mode. The advantage of moving the vector table to SRAM is that the exception handler addresses can be dynamically changed by modifying the SRAM vector table contents. However, the nonvolatile flash memory vector table must be modified by a flash memory write. Note that the exception table must be 256 byte-aligned for Cortex®-M0+ and 1024 byte-aligned for Cortex®-M4.

The exception sources (exception numbers 1 to 15) are explained in **“Exception sources”** on page 94. The exceptions marked as Reserved in **Table 9-1** are not used, although they have addresses reserved for them in the vector table. The interrupt sources (exception numbers 16 to 183) are explained in **“Interrupt sources”** on page 97.

9.4 Exception sources

This section explains the different exception sources listed in **Table 9-1** and **Table 9-2** (exception numbers 1 to 15).

9.4.1 Reset exception

Device reset is treated as an exception in EZ-USB™ FX10 MCUs. Reset exception is always enabled with a fixed priority of –3, the highest priority exception in both the cores. When the device boots up, only the Cortex®-M0+ core is available. The CM0+ core executes the ROM boot code and can enable Cortex®-M4 core from the application code. The reset exception of the CM0+ core is tied to the device reset or startup. When the Cortex®-M0+ core releases the Cortex®-M4 reset, the M4 reset exception is executed. A device reset can occur due to multiple reasons, such as power-on-reset (POR), external reset signal on XRES pin, or watchdog reset. When the device is reset, the initial boot code for configuring the device is executed by the Cortex®-M0+ out of supervisory read-only memory (SROM). The boot code and other data in SROM memory are programmed by Infineon, and are not read/write accessible to external users. After completing the SROM boot sequence, the Cortex®-M0+ code execution jumps to flash memory. Flash memory address 0x10000004 (Exception#1 in **Table 9-1**) stores the location of the startup code in flash memory. The CPU starts executing code out of this address. Note that the reset exception address in the SRAM vector table will never be used because the device comes out of reset with the flash vector table selected. The register configuration to select the SRAM vector table can be done only as part of the startup code in flash after the reset is de-asserted. Note that the reset exception flow for Cortex®-M4 is the same as Cortex®-M0+. However, Cortex®-M4 execution begins only after CM0+ core de-asserts the M4 reset.

9.4.2 Non-maskable interrupt exception

Non-maskable interrupt (NMI) is the highest priority exception next to reset. It is always enabled with a fixed priority of –2. Both the cores have their own NMI exception. There are three ways to trigger an NMI exception in a CPU core:

- **NMI exception from a system interrupt:** Both Cortex®-M0+ and Cortex®-M4 provide an option to trigger an NMI exception from up to four of the 133 system interrupts. The NMI exception triggered due to the interrupt will execute the NMI handler pointed to by the active exception vector table. The CPUSS_CM4_NMI_CTLx and CPUSS_CM0P_NMI_CTLx registers select the interrupt source that triggers the NMI from hardware. The registers have a default value of 1023; that is, if the register is set to 1023, then that particular register does not map any interrupt source to the NMI. There are four such registers and each can map one interrupt vector to the NMI. NMI is triggered when any of the four interrupts are triggered; that is, the interrupts are logically ORed. See **Figure 9-5**.

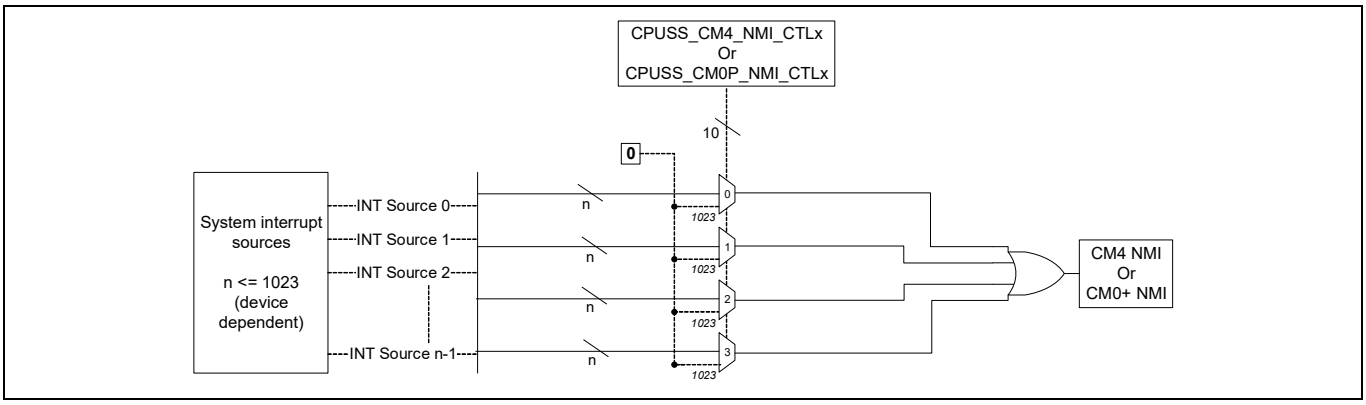


Figure 9-5. NMI trigger

- **NMI exception by setting NMIPENDSET bit (user NMI exception):** An NMI exception can be triggered in software by setting the NMIPENDSET bit in the interrupt control state registers (CM0P_SCS_ICSR and CM4_SCS_ICSR). Setting this bit will execute the NMI handler pointed to by the active vector table in the respective CPU cores.
- **System Call NMI exception:** This exception is used for nonvolatile programming and other system call operations such as flash write operation and flash checksum operation. Inter processor communication (IPC) mechanism is used to implement a system call in EZ-USB™ FX10 MCUs. A dedicated IPC mailbox is associated with each core (M0+ and M4) and the debug access port (DAP) to trigger a system call. The CPU or DAP acquires this dedicated mailbox, writes the system call opcode and argument to the mailbox, and notifies a dedicated IPC structure. Typically, the argument is a pointer to a structure in SRAM. This results in an NMI interrupt in the CM0+ core. Note that all the system calls are serviced by Cortex®-M0+ core. A Cortex®-M0+ NMI exception triggered by this method executes the NMI exception handler code that resides in SROM. Note that the NMI exception handler address is automatically initialized to the system call API located in SROM (at 0x0000000D) by the boot code. The value should be retained during vector table relocations; otherwise, no system call will be executed. The NMI handler code in SROM is not read/write accessible because it contains nonvolatile programming routines that cannot be modified by the user. The result of the system call is passed through the same IPC mechanism. For details, see to the **"Inter-processor communication"** on page 72.

9.4.3 HardFault exception

Both CM0+ and CM4 cores support HardFault exception. HardFault is an always-enabled exception that occurs because of an error during normal or exception processing. HardFault has a fixed priority of –1, meaning it has higher priority than any exception with configurable priority. A HardFault exception is a catch-all exception for different types of fault conditions, which include executing an undefined instruction and accessing an invalid memory addresses. The CPU does not provide fault status information to the HardFault exception handler, but it does permit the handler to perform an exception return and continue execution in cases where software has the ability to recover from the fault situation.

9.4.4 Memory management fault exception

A memory management fault is an exception that occurs because of a memory protection-related fault. The fixed memory protection constraints determine this fault, for both instruction and data memory transactions. This fault is always used to abort instruction accesses to Execute Never (XN) memory regions. The memory management fault is only supported by the M4 core. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

9.4.5 Bus fault exception

A Bus fault is an exception that occurs because of a memory-related fault for an instruction or data memory transaction. This might be from an error detected on a bus in the memory system. The bus fault is supported only by the M4 core. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

9.4.6 Usage fault exception

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

- an undefined instruction
- an illegal unaligned access
- invalid state on instruction execution
- an error on exception return

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

- an unaligned address on word and halfword memory access
- division by zero

The usage fault is supported only by the M4 core. The priority of the exception is configurable from 0 (highest) to 7 (lowest).

9.4.7 Supervisor Call (SVCall) exception

Both CM0+ and CM4 cores support SVCall exception. Supervisor Call (SVCall) is an always-enabled exception caused when the CPU executes the SVC instruction as part of the application code. Application software uses the SVC instruction to make a call to an underlying operating system and provide a service. This is known as a supervisor call. The SVC instruction enables the application to issue an SVCall that requires privileged access to the system.

The priority of an SVCall exception can be configured to a value between 0 and 3 for CM0+ and 0 to 7 for CM4 core by writing to the bitfields PRI_11 of the System Handler Priority Register 2 (CM0P_SCS_SHPR2 and CM4_SCS_SHPR2). When the SVC instruction is executed, the SVCall exception enters the pending state and waits to be serviced by the CPU. The SVCALLPENDED bit in the System Handler Control and State Register (CM0P_SCS_SHCSR and CM4_SCS_SHCSR) can be used to check or modify the pending status of the SVCall exception.

9.4.8 PendSupervisory (PendSV) exception

Both CM0+ and CM4 cores support PendSV exception. PendSV is another supervisor call related exception similar to SVCall, normally being software-generated. PendSV is always enabled and its priority is configurable similar to SVCall. The PendSV exception is triggered by setting the PENDSVSET bit in the Interrupt Control State register (CM0P_SCS_ICSR and CM4_SCS_ICSR). On setting this bit, the PendSV exception enters the pending state, and waits to be serviced by the CPU. The pending state of a PendSV exception can be cleared by setting the PENDSVCLR bit in the Interrupt Control State register. The priority of a PendSV exception can be configured to a value between 0 and 3 for CM0+ and 0 to 7 for M4 by writing to the bitfields PRI_14 of the System Handler Priority Register 3. See the Armv6-M Architecture Reference Manual for more details.

9.4.9 System Tick (SysTick) exception

Both CM0+ and CM4 cores in EZ-USB™ FX10 MCUs support a system timer, referred to as SysTick, as part of their internal architecture. SysTick provides a simple, 24-bit decrementing counter for various timekeeping purposes such as an RTOS tick timer, high-speed alarm timer, or simple counter. The SysTick timer can be configured to generate an interrupt when its count value reaches ‘0’, which is referred to as a SysTick exception. The exception is enabled by setting the TICKINT bit in the SysTick Control and Status register (CM0P_SCS_SYST_CSR and CM4_SCS_SYST_CSR). The priority of a SysTick exception can be configured to a value between 0 and 3 for CM0+ and 0 to 7 for M4 by writing to the bitfields PRI_15 of the System Handler Priority Register 3 (SHPR3). The SysTick exception can always be generated in software at any instant by writing a one to the PENDSTSET bit in the Interrupt Control State Register. Similarly, the pending state of the SysTick exception can be cleared by writing a one to the PENDSTCLR bit in the Interrupt Control State register.

9.5 Interrupt sources

The EZ-USB™ FX10 MCU supports 133 interrupts from peripherals. The source of each interrupt is listed in **Table 9-3**. These system interrupts are mapped directly to Cortex®-M4 core (IRQ0 to IRQ141 or exception 16 to 157). For Cortex®-M0+ core, any of the 133 interrupts can be routed to any of the available eight interrupts (IRQ0 to IRQ7 or exception 16 to 23). The CPUSS_CM0_SYSTEM_INT_CTLx registers are used to make this interrupt selection in CM0+.

There are 133 CPUSS_CM0_SYSTEM_INT_CTLx registers. The CPU_INT_IDX bits [2:0] of the register selects which IRQn line of CM0+, the x^{th} system interrupt is mapped to. The CPU_INT_VALID bit [31] of the register decides whether the x^{th} interrupt is enabled for the core. For Cortex®-M0+, multiple system interrupts can be mapped on the same CPU interrupt. Therefore, an active CPU interrupt may indicate one or multiple active system interrupts. For instance to connect Port 0 (IDX = 0) and Port 2 (IDX = 2) GPIO interrupts to IRQ0 of CM0+, CPU_INT_IDX bits should be set to '0' and CPU_INT_VALID bit should be set to '1' in CPUSS_CM0_SYSTEM_INT_CTL0 and CPUSS_CM0_SYSTEM_INT_CTL2 registers. Note that this only connects and enables the interrupts; configuration and masking of the interrupt to the NVIC should be done as part of source peripheral configuration.

As a result of the reduction functionality in the Cortex®-M0+ core, multiple system interrupts share a CPU interrupt handler as provided by the CPU's vector table. Each CPU interrupt has an associated CPUSS_CM0_INTx_STATUS register:

- SYSTEM_INT_VALID bit[31] of the register specifies if any system interrupt is active for the CPU interrupt. This bit remains set until all the system interrupts connected to the CPU interrupts are cleared at the source.
- SYSTEM_INT_IDX bits[9:0] of the register specifies the index (a number in the range [0, 1022]) of the lowest active system interrupt mapped to the corresponding CPU interrupt.

For instance, say CM0+ IRQ0 is connected to Port 0 and Port 2 GPIO interrupts. When an interrupt in both Port 0 and Port 2 are triggered simultaneously, IRQ0 in CM0+ will be triggered with CPUSS_CM0_INT0_STATUS reading 0 (Port 0 interrupt) in SYSTEM_INT_IDX bit and SYSTEM_INT_VALID bit set. After Port 0 interrupt is serviced and cleared, IRQ0 will be triggered again with SYSTEM_INT_IDX bit set to '2' (Port 2 interrupt) in the CPUSS_CM0_INT0_STATUS register. Only after servicing Port 2 interrupt and clearing, the SYSTEM_INT_VALID bit will be cleared.

The CPU interrupt handler should use the SYSTEM_INT_IDX field to index a system interrupt lookup table and jump to the system interrupt handler. The lookup table is typically located in one of the system memories. The following code illustrates the approach:

Interrupts

```
typedef void (* SystemIntr_Handler)(void);
void CM0_CpuIntr0_Handler (void)
{
uint32_t system_int_idx;
SystemIntr_Handler handler;
if(CPUSS_CM0_INT0_STATUS.SYSTEM_INT_VALID)
{
system_int_idx = CPUSS_CM0_INT0_STATUS.SYSTEM_INT_IDX;
handler = SystemIntr_Table[system_int_idx];
handler(); // jump to system interrupt handler
}
else
{
// Triggered by SW or due to SW clear error (SW cleared a peripheral
// interrupt flag but didn't clear the Pending flag at NVIC)
}
}
…
void CM0_CpuIntr7_Handler (void)
{
uint32_t system_int_idx;
SystemIntr_Handler handler;
if(CPUSS_CM0_INT7_STATUS.SYSTEM_INT_VALID)
{
system_int_idx = CPUSS_CM0_INT7_STATUS.SYSTEM_INT_IDX;
handler = SystemIntr_Table[system_int_idx];
handler(); // jump to system interrupt handler
}
else
{
// Triggered by SW or due to SW clear error (SW cleared a peripheral
// interrupt flag but didn't clear the Pending flag at NVIC)
}
}
void CM0_SystemIntr0_Handler (void)
{
// Clear the peripheral interrupt request flag by register write
// Read back the register, to ensure the completion of register write access
// Handle system interrupt 0.
}
…
void CM0_SystemIntr167_Handler (void)
{
// Clear the peripheral interrupt request flag by register write
// Read back the register, to ensure the completion of register write access
// Handle system interrupt 167.
}
```

Interrupts

The interrupts include standard interrupts from the on-chip peripherals such as TCPWM, serial communication block, CSD block, watchdog, and so on. The interrupt generated is usually the logical OR of the different peripheral states. The peripheral interrupt status register should be read in the ISR to detect which condition generated the interrupt. These interrupts are usually level interrupts. The appropriate interrupt registers should be cleared in the ISR to deassert the interrupt. Usually a write '1' is required to clear the registers. If the interrupt register is not cleared in the ISR, the interrupt will remain asserted and the ISR will be executed continuously. See the **"I/O system"** on page 332 for details on GPIO interrupts.

As seen from **Table 9-3**, 39 interrupts (IRQ0 to IRQ38) are capable of waking up the device from Deep Sleep power mode. For Cortex®-M4, IRQ0 to IRQ38 directly map to these sources. However, in the Cortex®-M0+, all the eight IRQ lines support Deep Sleep wakeup if they are connected to a Deep Sleep capable interrupt. This means the 39 Deep Sleep wakeup-capable interrupts can be connected to any of the eight IRQ lines of Cortex®-M0+, if such a wakeup is desired.

Table 9-3. List of EZ-USB™ FX10 interrupts

System interrupt	Cortex®-M4 exception number	Power mode	Description
0	16	Deep Sleep	GPIO Port Interrupt #0
1	17	Deep Sleep	GPIO Port Interrupt #1
2	18	Deep Sleep	GPIO Port Interrupt #4
3	19	Deep Sleep	GPIO Port Interrupt #5
4	20	Deep Sleep	GPIO Port Interrupt #6
5	21	Deep Sleep	GPIO Port Interrupt #7
6	22	Deep Sleep	GPIO Port Interrupt #8
7	23	Deep Sleep	GPIO Port Interrupt #9
8	24	Deep Sleep	GPIO Port Interrupt #10
9	25	Deep Sleep	GPIO Port Interrupt #11
10	26	Deep Sleep	GPIO Port Interrupt #12
11	27	Deep Sleep	GPIO Port Interrupt #13
12	28	Deep Sleep	GPIO All Ports
13	29	Deep Sleep	GPIO Supply Detect Interrupt
14	30	Deep Sleep	Serial Communication Block #6 (DeepSleep capable)
15	31	Deep Sleep	Multi Counter Watchdog Timer interrupt
16	32	Deep Sleep	Multi Counter Watchdog Timer interrupt
17	33	Deep Sleep	USBHS DEV interrupt
18	34	Deep Sleep	Other combined Interrupts for SRSS (LVD, WDT, CLKCAL)
19	35	Deep Sleep	CPUSS Inter Process Communication Interrupt #0
20	36	Deep Sleep	CPUSS Inter Process Communication Interrupt #1
21	37	Deep Sleep	CPUSS Inter Process Communication Interrupt #2
22	38	Deep Sleep	CPUSS Inter Process Communication Interrupt #3
23	39	Deep Sleep	CPUSS Inter Process Communication Interrupt #4
24	40	Deep Sleep	CPUSS Inter Process Communication Interrupt #5
25	41	Deep Sleep	CPUSS Inter Process Communication Interrupt #6
26	42	Deep Sleep	CPUSS Inter Process Communication Interrupt #7
27	43	Deep Sleep	CPUSS Inter Process Communication Interrupt #8
28	44	Deep Sleep	CPUSS Inter Process Communication Interrupt #9
29	45	Deep Sleep	CPUSS Inter Process Communication Interrupt #10
30	46	Deep Sleep	CPUSS Inter Process Communication Interrupt #11
31	47	Deep Sleep	CPUSS Inter Process Communication Interrupt #12
32	48	Deep Sleep	CPUSS Inter Process Communication Interrupt #13
33	49	Deep Sleep	CPUSS Inter Process Communication Interrupt #14

Table 9-3. List of EZ-USB™ FX10 interrupts (continued)

System interrupt	Cortex®-M4 exception number	Power mode	Description
34	50	Deep Sleep	CPUSS Inter Process Communication Interrupt #15
35	51	Deep Sleep	USB32DEV wakeup interrupt
36	52	Deep Sleep	LVDS wakeup interrupt
46	62	Active	Serial Communication Block #1
47	63	Active	Serial Communication Block #2
48	64	Active	Serial Communication Block #3
49	65	Active	Serial Communication Block #4
50	66	Active	Serial Communication Block #5
51	67	Active	Serial Communication Block #6
52	68	Active	CPUSS DMAC, Channel #0
53	69	Active	CPUSS DMAC, Channel #1
54	70	Active	CPUSS DMAC, Channel #2
55	71	Active	CPUSS DMAC, Channel #3
56	72	Active	CPUSS DMAC, Channel #4
57	73	Active	CPUSS DMAC, Channel #5
58	74	Active	CPUSS DataWire #0, Channel #0
59	75	Active	CPUSS DataWire #0, Channel #1
60	76	Active	CPUSS DataWire #0, Channel #2
61	77	Active	CPUSS DataWire #0, Channel #3
62	78	Active	CPUSS DataWire #0, Channel #4
63	79	Active	CPUSS DataWire #0, Channel #5
64	80	Active	CPUSS DataWire #0, Channel #6
65	81	Active	CPUSS DataWire #0, Channel #7
66	82	Active	CPUSS DataWire #0, Channel #8
67	83	Active	CPUSS DataWire #0, Channel #9
68	84	Active	CPUSS DataWire #0, Channel #10
69	85	Active	CPUSS DataWire #0, Channel #11
70	86	Active	CPUSS DataWire #0, Channel #12
71	87	Active	CPUSS DataWire #0, Channel #13
72	88	Active	CPUSS DataWire #0, Channel #14
73	89	Active	CPUSS DataWire #0, Channel #15
74	90	Active	CPUSS DataWire #0, Channel #16
75	91	Active	CPUSS DataWire #0, Channel #17
76	92	Active	CPUSS DataWire #0, Channel #18
77	93	Active	CPUSS DataWire #0, Channel #19
78	94	Active	CPUSS DataWire #0, Channel #20
79	95	Active	CPUSS DataWire #0, Channel #21

Table 9-3. List of EZ-USB™ FX10 interrupts (continued)

System interrupt	Cortex®-M4 exception number	Power mode	Description
80	96	Active	CPUSS DataWire #0, Channel #22
81	97	Active	CPUSS DataWire #0, Channel #23
82	98	Active	CPUSS DataWire #1, Channel #0
83	99	Active	CPUSS DataWire #1, Channel #1
84	100	Active	CPUSS DataWire #1, Channel #2
85	101	Active	CPUSS DataWire #1, Channel #3
86	102	Active	CPUSS DataWire #1, Channel #4
87	103	Active	CPUSS DataWire #1, Channel #5
88	104	Active	CPUSS DataWire #1, Channel #6
89	105	Active	CPUSS DataWire #1, Channel #7
90	106	Active	CPUSS DataWire #1, Channel #8
91	107	Active	CPUSS DataWire #1, Channel #9
92	108	Active	CPUSS DataWire #1, Channel #10
93	109	Active	CPUSS DataWire #1, Channel #11
94	110	Active	CPUSS DataWire #1, Channel #12
95	111	Active	CPUSS DataWire #1, Channel #13
96	112	Active	CPUSS DataWire #1, Channel #14
97	113	Active	CPUSS DataWire #1, Channel #15
98	114	Active	CPUSS DataWire #1, Channel #16
99	115	Active	CPUSS DataWire #1, Channel #17
100	116	Active	CPUSS DataWire #1, Channel #18
101	117	Active	CPUSS DataWire #1, Channel #19
102	118	Active	CPUSS DataWire #1, Channel #20
103	119	Active	CPUSS DataWire #1, Channel #21
104	120	Active	CPUSS DataWire #1, Channel #22
105	121	Active	CPUSS DataWire #1, Channel #23
106	122	Active	CPUSS Fault Structure Interrupt #0
107	123	Active	CPUSS Fault Structure Interrupt #1
108	124	Active	CRYPTO Accelerator Interrupt
109	125	Active	FLASH Macro Interrupt
110	126	Active	Floating Point operation fault
111	127	Active	CM0+ CTI #0
112	128	Active	CM0+ CTI #1
113	129	Active	CM4 CTI #0
114	130	Active	CM4 CTI #1
115	131	Active	TCPWM #0, Counter #0
116	132	Active	TCPWM #0, Counter #1

Table 9-3. List of EZ-USB™ FX10 interrupts (continued)

System interrupt	Cortex®-M4 exception number	Power mode	Description
117	133	Active	TCPWM #0, Counter #2
118	134	Active	TCPWM #0, Counter #3
119	135	Active	TCPWM #0, Counter #4
120	136	Active	TCPWM #0, Counter #5
121	137	Active	TCPWM #0, Counter #6
122	138	Active	TCPWM #0, Counter #7
123	139	Active	TDM0 Audio interrupt RX
124	140	Active	TDM0 Audio interrupt TX
125	141	Active	Serial Memory Interface interrupt
126	142	Active	USBFS High Priority Interrupt Group
127	143	Active	USBFS Medium Priority Interrupt Group
128	144	Active	USBFS Low Priority Interrupt Group
129	145	Active	USBHSDEV active interrupt
130	146	Active	CAN #0, Consolidated interrupt #0
131	147	Active	CAN #0, Interrupt #0, Channel #0
132	148	Active	CAN #0, Interrupt #1, Channel #0
133	149	Active	PDM interrupt #0
134	150	Active	PDM interrupt #1
135	151	Active	LVDS active interrupt
136	152	Active	HBWSS trigger assist interrupt
137	153	Active	HBWSS USB egress DMA adapter interrupt
138	154	Active	HBWSS USB ingress DMA adapter interrupt
139	155	Active	USB32DEV active interrupt
140	156	Active	LVDS DMA adapter interrupt #0
141	157	Active	LVDS DMA adapter interrupt #1

9.6 Interrupt/exception priority

Exception priority is useful for exception arbitration when there are multiple exceptions that need to be serviced by the CPU. Both M4 and M0+ cores in EZ-USB™ FX10 MCUs provide flexibility in choosing priority values for different exceptions. All exceptions other than Reset, NMI, and HardFault can be assigned a configurable priority level. The Reset, NMI, and HardFault exceptions have a fixed priority of –3, –2, and –1, respectively. In EZ-USB™ FX10 MCUs, lower priority numbers represent higher priorities. This means that the Reset, NMI, and HardFault exceptions have the highest priorities. The other exceptions can be assigned a configurable priority level between 0 and 3 for Cortex®-M0+ and 0 to 7 for Cortex®-M4.

Both M0+ and M4 support nested exceptions in which a higher priority exception can obstruct (interrupt) the currently active exception handler. This pre-emption does not happen if the incoming exception priority is the same as or lower than the active exception. The CPU resumes execution of the lower priority exception handler after servicing the higher priority exception. The CM0+ core in the EZ-USB™ FX10 MCU allows nesting of up to four exceptions; the CM4 core allows up to eight exceptions. When the CPU receives two or more exceptions requests of the same priority, the lowest exception number is serviced first.

The registers to configure the priority of exception numbers 1 to 15 are explained in **“Exception sources”** on page 94.

The priority of the 32 CM0+ and 133 CM4 interrupts can be configured by writing to the respective Interrupt Priority registers (CM0P_SCS_IPR and CM4_SCS_IPR). This is a group of eight (CM0+) and 60 (CM4) 32-bit registers with each register storing the priority values of four interrupts, as given in **Table 9-4** and **Table 9-5**. For CM0+, the first 16 bits of the ISER and ICER registers are valid. Refer to **“Interrupt sources”** on page 97 for details on how to map a CPU interrupt to a system interrupt in CM0+.

Table 9-4. Interrupt priority register bit definitions for Cortex®-M0+ (CM0P_SCS_IPR)

Bits	Name	Description
7:6	PRI_N0	Priority of interrupt number N
15:14	PRI_N1	Priority of interrupt number N+1
23:22	PRI_N2	Priority of interrupt number N+2
31:30	PRI_N3	Priority of interrupt number N+3

Table 9-5. Interrupt priority register bit definitions for Cortex®-M4 (CM4_SCS_IPR)

Bits	Name	Description
7:5	PRI_N0	Priority of interrupt number N
15:13	PRI_N1	Priority of interrupt number N+1
23:21	PRI_N2	Priority of interrupt number N+2
31:29	PRI_N3	Priority of interrupt number N+3

9.7 Enabling and disabling interrupts

The NVICs of both CM0+ andCM4 core provide registers to individually enable and disable the interrupts in software. If an interrupt is not enabled, the NVIC will not process the interrupt requests on that interrupt line. The Interrupt Set-Enable Register (CM0P_SCS_ISER and CM4_SCS_ISER) and the Interrupt Clear-Enable Register (CM0P_SCS_ICER and CM4_SCS_ICER) are used to enable and disable the interrupts respectively. These registers are 32-bit wide and each bit corresponds to the same numbered interrupt in CM0+. For CM4 core, there are eight ISER/ICER registers. These registers can also be read in software to get the enable status of the interrupts. **Table 9-6** shows the register access properties for these two registers. Note that writing zero to these registers has no effect.

For CM0+, the first 16 bits of the ISER and ICER registers are valid. Refer to **"Interrupt sources"** on page 97 for details on how to map a CPU interrupt to a system interrupt in CM0+.

Table 9-6. Interrupt enable/disable registers

Register	Operation	Bit value	Comment
Interrupt Set Enable register	Write	1	To enable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled
Interrupt Clear Enable register	Write	1	To disable the interrupt
		0	No effect
	Read	1	Interrupt is enabled
		0	Interrupt is disabled

The ISER and ICER registers are applicable only for the interrupts. These registers cannot be used to enable or disable the exception numbers 1 to 15. The 15 exceptions have their own support for enabling and disabling, as explained in **"Exception sources"** on page 94.

The PRIMASK register in the CPUs (both CM0+ and CM4) can be used as a global exception enable register to mask all the configurable priority exceptions irrespective of whether they are enabled. Configurable priority exceptions include all the exceptions except Reset, NMI, and HardFault listed in **Table 9-1**. When the PM bit (bit 0) in the PRIMASK register is set, none of the configurable priority exceptions can be serviced by the CPU, though they can be in the pending state waiting to be serviced by the CPU after the PM bit is cleared.

9.8 Interrupt/exception states

Each exception can be in one of the following states.

Table 9-7. Exception states

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

The Interrupt Control State Register (CM0P_SCS_ICSR and CM4_SCS_ICSR) contains status bits describing the various exceptions states.

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

9.8.1 Pending interrupts/exceptions

When a peripheral generates an interrupt request signal to the NVIC or an exception event occurs, the corresponding exception enters the pending state. When the CPU starts executing the corresponding exception handler routine, the exception is changed from the pending state to the active state. The NVIC allows software pending of the 32 (CM0+) or 133 (CM4) interrupt lines by providing separate register bits for setting and clearing the pending states of the interrupts. The Interrupt Set-Pending register (CM0P_SCS_ISPR and CM4_SCS_ISPR) and the Interrupt Clear-Pending register (CM0P_SCS_ICPR and CM4_SCS_ICPR) are used to set and clear the pending status of the interrupt lines. These registers are 32 bits wide, and each bit corresponds to the same numbered interrupt. In the case of CM4, there are eight sets of such registers to accommodate all 168 interrupts. Table 9-8 shows the register access properties for these two registers. Note that writing zero to these registers has no effect.

Table 9-8. Interrupt set Pending/Clear Pending registers

Register	Operation	Bit value	Comment
Interrupt Set-Pending Register (ISPR)	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 (ICPR)	Write	1	To clear a pending interrupt
		0	No effect
	Read	1	Interrupt is pending
		0	Interrupt is not pending

Setting the pending bit when the same bit is already set results in only one execution of the ISR. The pending bit can be updated regardless of whether the corresponding interrupt is enabled. If the interrupt is not enabled, the interrupt line will not move to the pending state until it is enabled by writing to the ISER register.

Note that the ISPR and ICPR registers are used only for the peripheral interrupts. These registers cannot be used for pending the exception numbers 1 to 15. These 15 exceptions have their own support for pending, as explained in **"Exception sources"** on page 94.

9.9 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 starts executing the corresponding exception handler (in handler mode). The CPU pushes the contents of the eight 32-bit internal registers into the stack. These registers are the Program and Status Register (PSR), ReturnAddress, Link Register (LR or R14), R12, R3, R2, R1, and R0. Both Cortex®-M4 and Cortex®-M0+ have two stack pointers - MSP and PSP. Only one of the stack pointers can be active at a time. When in thread mode, the Active Stack Pointer bit in the Control register is used to define the current active stack pointer. When in handler mode, the MSP is always used as the stack pointer. The stack pointer always grows downwards and points to the address that has the last pushed data.

When the CPU is in thread mode and an exception request comes, the CPU uses the stack pointer defined in the control register to store the general-purpose register contents. After the stack push operations, the CPU enters handler mode to execute the exception handler. When another higher priority exception occurs while executing the current exception, the MSP is used for stack push/pop operations, because the CPU is already in handler mode. See the **"CPU subsystem (CPUSS)"** on page 63 for details.

9.10 Interrupts and low-power modes

The EZ-USB™ FX10 MCU family allows device (CPU) wakeup from low-power modes when certain peripheral interrupt requests are generated. The Wakeup Interrupt Controller (WIC) block generates a wakeup signal that causes the CPU to enter Active mode when one or more wakeup sources generate an interrupt signal. After entering Active mode, the ISR of the peripheral interrupt is executed.

The Wait For Interrupt (WFI) or Wait For Event (WFE) instruction, executed by the CPU, triggers the transition into Sleep, and Deep Sleep modes. Both the WFI and WFE instructions are capable of waking up on interrupts. However, the WFE requires the interrupts to be unmasked in the CPU's Priority Mask register. Refer to the PRIMASK register definition on the **Arm website**. In addition, the WFE instruction puts the CPU to sleep based on the status of an event bit and wakes up from an event signal, typically sent by the other CPU. WFI does not require PRIMASK unmasking and can wake up the CPU from any pending interrupt masked to the NVIC or WIC. However, WFI cannot wake up the CPU from event signals from other CPUs. The sequence of entering the different low-power modes is detailed in the **"Device power modes"** on page 281.

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

- Interrupt sources that are available in the Active, Sleep, and Deep Sleep modes (watchdog timer interrupt, GPIO interrupts
- Interrupt sources that are available only in the Active and Sleep modes

When using the WFE instruction in CM4, make sure to call the WFE instruction twice to properly enter and exit Sleep/Deep Sleep modes. This behavior comes from the event register implementation in Arm® v7 architecture used in Cortex®-M4. According to the ARM V7 architecture reference manual (section B1.5.18 Wait For Event and Send Event):

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

Therefore, the first WFE instruction puts CM4 to Sleep and second WFE clears the event register after a WFE wakeup, which sets the event register. So the next WFE will put the core to Sleep.

Note that this behavior is not present in Arm® v6 architecture used in Cortex®-M0+. Therefore, in CM0+ only one WFE instruction is sufficient to successfully enter or exit Sleep and Deep Sleep modes.

9.11 Interrupt/exception – initialization/ configuration

This section covers the different steps involved in initializing and configuring exceptions in the EZ-USB™ FX10 MCU.

- Configuring the Exception Vector Table Location: The first step in using exceptions is to configure the vector table location as required - either in flash memory or SRAM. This configuration is done as described in **"Exception vector table"** on page 92.
  The vector table should be available in SRAM if the application must change the vector addresses dynamically. If the table is located in flash, then a flash write operation is required to modify the vector table contents. The Eclipse IDE for ModusToolbox™ uses the vector table in SRAM by default.
- Configuring Individual Exceptions: The next step is to configure individual exceptions required in an application, as explained in earlier sections.
  - Configure the exception or interrupt source; this includes setting up the interrupt generation conditions. The register configuration depends on the specific exception required. Refer to the respective peripheral chapter to know more about the interrupt configuration supported by them.
  - Define the exception handler function and write the address of the function to the exception vector table. **Table 9-1** gives the exception vector table format; the exception handler address should be written to the appropriate exception number entry in the table.
  - For Cortex®-M0+, define and enable the additional system interrupt handler table and functions as explained in **"Interrupt sources"** on page 97.
  - Set up the exception priority, as explained in **"Interrupt/exception priority"** on page 104.
  - Enable the exception, as explained in **"Enabling and disabling interrupts"** on page 105.

9.12 Register list

Table 9-9. Register list

Register name	Description
CPUSS_CM0_NMI_CTLx	Cortex®-M0+ NMI control registers (4 registers)
CPUSS_CM0_SYSTEM_INT_CTLx	Cortex®-M0+ interrupt control registers for xth system interrupt (168 registers)
CPUSS_CM0_SYSTEM_INTx_STATUS	Cortex®-M0+ interrupt status registers for xth CPU interrupt (8 registers)
CPUSS_CM4_NMI_CTLx	Cortex®-M4 NMI control registers (4 registers)
SYSTEM_CM0P_SCS_ISER	Cortex®-M0+ interrupt set-enable register
SYSTEM_CM0P_SCS_ICER	Cortex®-M0+ interrupt clear enable register
SYSTEM_CM0P_SCS_ISPR	Cortex®-M0+ interrupt set-pending register
SYSTEM_CM0P_SCS_ICPR	Cortex®-M0+ interrupt clear-pending register
SYSTEM_CM0P_SCS_IPR	Cortex®-M0+ interrupt priority register
SYSTEM_CM0P_SCS_ICSR	Cortex®-M0+ interrupt control state register
SYSTEM_CM0P_SCS_VTOR	Cortex®-M0+ vector table offset register
SYSTEM_CM0P_SCS_AIRCR	Cortex®-M0+ application interrupt and reset control register
SYSTEM_CM0P_SCS_SHPR2	Cortex®-M0+ system handler priority register 2
SYSTEM_CM0P_SCS_SHPR3	Cortex®-M0+ system handler priority register 3
SYSTEM_CM0P_SCS_SHCSR	Cortex®-M0+ system handler control and state register
SYSTEM_CM4_SCS_ISER	Cortex®-M4 interrupt set-enable register
SYSTEM_CM4_SCS_ICER	Cortex®-M4 interrupt clear enable register
SYSTEM_CM4_SCS_ISPR	Cortex®-M4 interrupt set-pending register
SYSTEM_CM4_SCS_ICPR	Cortex®-M4 interrupt clear-pending register
SYSTEM_CM4_SCS_IPR	Cortex®-M4 interrupt priority registers
SYSTEM_CM4_SCS_ICSR	Cortex®-M4 interrupt control state register
SYSTEM_CM4_SCS_VTOR	Cortex®-M4 vector table offset register
SYSTEM_CM4_SCS_AIRCR	Cortex®-M4 application interrupt and reset control register
SYSTEM_CM4_SCS_SHPR2	Cortex®-M4 system handler priority register 2
SYSTEM_CM4_SCS_SHPR3	Cortex®-M4 system handler priority register 3
SYSTEM_CM4_SCS_SHCSR	Cortex®-M4 system handler control and state register

10 Protection units

Protection units are implemented in the EZ-USB™ FX10 MCU to enforce security based on different operations. A protection unit allows or restricts bus transfers. The rules are enforced based on specific properties of a transfer. The rules that determine protection are implemented in protection structures (a register structure). A protection structure defines the protected address space and the protection attributes. The hardware that evaluates these protection structures, to restrict or permit access, is the protection unit. The EZ-USB™ FX10 device has different types of protection units such as MPU, SMPU, and PPU. Each have a distinct set of protection structures, which helps define different protection regions and their attributes.

10.1 Architecture

Figure 10-1 shows a conceptual view of implementation of the EZ-USB™ FX10 protection system.

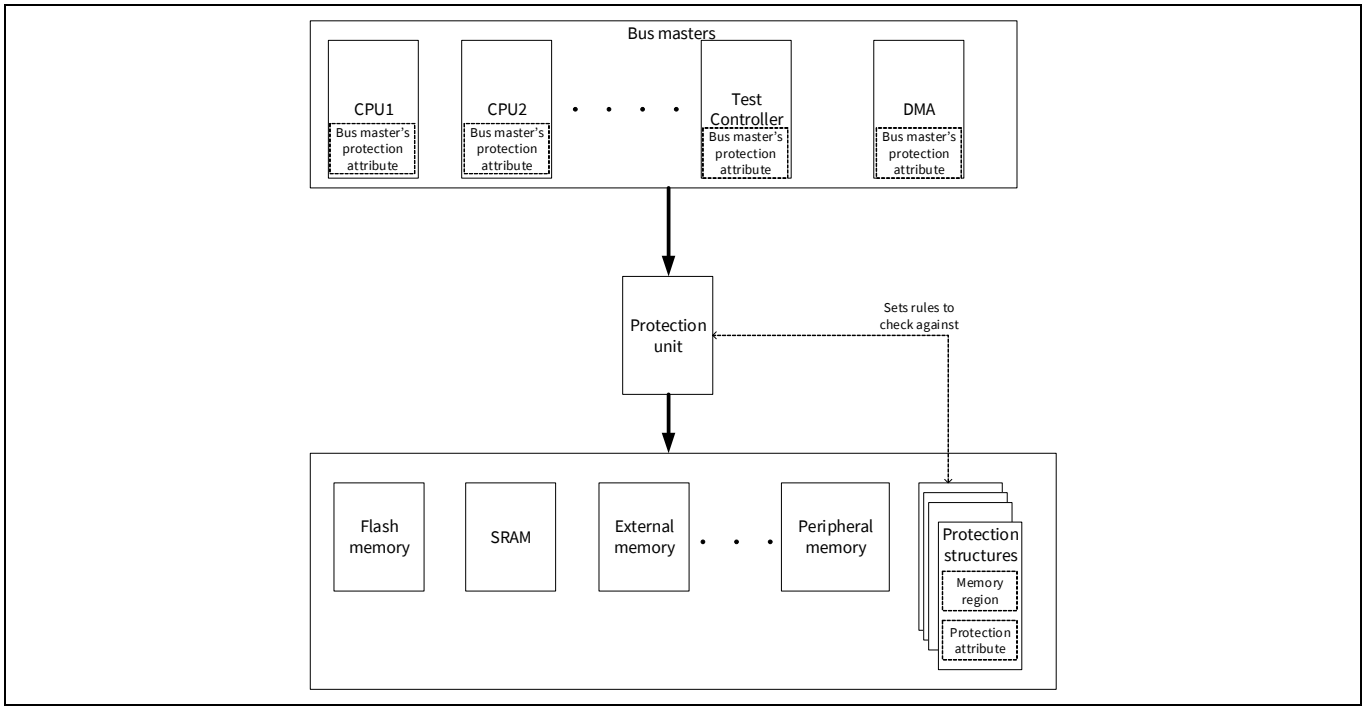


Figure 10-1. Conceptual view of FX10 protection system

The functioning of a secure system is based on the following:

- Bus masters: This term refers to the bus masters in the architecture. In a EZ-USB™ FX10 device, an example of a bus master is a Cortex®-M core, DMA, or a test controller.
- Protection units: Protection units are the hardware engines that enforce the protection defined by protection structures. There are three types of protection units, acting at different levels of memory access with different precedence and priority of protection – MPU, SMPU, and PPU.
- Protection structure: A protection structure is a register structure in memory that sets up the rules based on which each protection unit will evaluate a transfer. Each protection unit associates itself to multiple protection structures. The protection structure associated with a protection unit are evaluated in the order starting with the protection structure with the largest index. For example, if there are 16 protection structures associated with a protection unit, then the evaluation of a transfer starts from protection structure 15 and counts down. Physically a protection structure is a register structure in the memory map that defines a protection rule. Each protection structure constitutes the following:
  - Defines a memory region on which the rule is applied. It designates what the bus transfer needs to be evaluated against this protection structure.

    - Base address
    - Size of memory block
  - A set of protection attributes
    - R/W/X
    - User/privilege
    - Secure/non Secure
    - Protection context
- Protection attributes: These are properties based on which a transfer is evaluated. There are multiple protection attributes. The set of protection attributes available for a protection structure depends on the protection unit it is associated with. Protection attributes appear in two places:
  - Protection structures: Protection attributes associated with a protection structure set the rules for access based on these attributes.
  - Bus master's protection attribute: Each bus master has its own access attributes, which define the bus master's access privileges. Some of these attributes, such as secure/non-secure, are set for a master. Other attributes such as protection context and user/privilege attribute are dynamic attributes, which change based on bus master's context and state.

In summary, a EZ-USB™ FX10 device has protection units that act as a gate for any access to the EZ-USB™ FX10 memory map. The rules for protection are set by the protection structures. Each bus master is qualified by its own protection attribute. For every bus transfer, the protection unit compares the bus master's protection attribute and accessed address against the rules set in the protection structures and decides on providing or denying access.

10.2 EZ-USB™ FX10 protection architecture

When there is a memory (SRAM/flash/peripheral) access by a bus master, the access is evaluated by a protection unit against the protection attributes set in protection structures for the memory location being accessed. If the bus master's protection attributes satisfy the protection attributes set in the protection structures, then access is allowed by the protection unit. If there is an access restriction, a fault condition is triggered and a bus error occurs. Thus protection units secure bus transfer address range either in memory locations (SRAM/flash) or peripheral registers. From an architectural perspective, there is no difference between memory protection and peripheral protection. However, from an implementation perspective, separate memory and peripheral protection is provided.

Two types of protection units (memory protection units (MPU) and shared memory protection units (SMPU)) are provided in the CPU subsystem (CPUSS) to protect memory locations. A separate protection unit type is provided for peripheral protection (PPU) in the PERI:

- A bus master may have a dedicated MPU. In a CPU bus master, the MPU is typically implemented as part of the CPU and is under control of the OS/kernel. In a non-CPU bus master, the MPU is typically implemented as part of the bus infrastructure and under control of the OS/kernel of the CPU that "owns or uses" the bus master. If a CPU switches tasks or if a non-CPU switches ownership, the MPU settings are typically updated by OS/kernel software. The different MPU types are:
  - An MPU that is implemented as part of the CPU. This type is found in the Arm® CM0+ and CM4 CPUs.
  - An MPU that is implemented as part of the bus infrastructure. This type is found in bus masters such as crypto and test controller. The definition of this MPU type follows the Arm® MPU definition (in terms of memory region and access attribute definition) to ensure a consistent software interface.
- SMPUs are intended for implementing protection in a situation with multiple bus masters. These protection units implement a concept called Protection Context. A protection context is a pseudo state of a bus master, which can be used to determine access attributes across multiple masters. The protection context is a protection attribute not specific to a bus master. The SMPUs can distinguish between different protection

contexts; they can also distinguish secure from non-secure accesses. This allows for an effective protection in a multi-core scenario.

Protection units

- PPUs are protection units provided in the PERI register space for peripheral protection. The PPU attributes are similar to the SMPU, except that they are intended for protecting the peripheral space. Refer to the registers reference manual for details. The PPUs are intended to distinguish between different protection contexts and to distinguish secure from non-secure accesses and user mode accesses from privileged mode accesses. There are two types of PPU structures:
  - Fixed PPUs implement protection for fixed address regions that typically correspond to a specific peripheral
  - Programmable PPUs allows the user to program the address region to be protected

The platform's DMA controller does not have an MPU. Instead, a DMA controller channel inherits the access control attributes of the bus transfer that programmed the channel.

The definition of SMPU and PPU follows the MPU definition and adds the capability to distinguish accesses from different protection contexts (the MPU does not include support for a protection context). If security is required, the SMPU and possibly PPUs MMIO registers must be controlled by a secure CPU that enforces system-wide protection.

Figure 10-2 gives an overview of the location of MPUs, SMPUs, and PPUs in the system. Note that a peripheral group PPU needs to provide access control only to the peripherals within a peripheral group (group of peripherals with a shared bus infrastructure).

As mentioned, the MPU, SMPU, and PPU protection functionality follows the Arm® MPU definition:

- Multiple protection structures are supported
- Each structure specifies an address range in the unified memory architecture and access attributes. An address range can be as small as 32 bytes.

A protection violation is caused by a mismatch between a bus master's access attributes and the protection structure and access attributes for the memory region configured in the protection structure.

A bus transfer that violates a protection structure results in a bus error.

For AXI transfers, the complete address range is matched. If a transfer references multiple 32-byte regions (the smallest protection structure address range is 32 bytes), multiple cycles are required for matching – one cycle per 32-byte region.

Protection violations are captured in the fault report structure to allow for failure analysis. The fault report structures can generate an interrupt to indicate the occurrence of a fault. This is useful if the violating bus master cannot resolve the bus error by itself, but requires another CPU bus master to resolve the bus error on its behalf.

For a buffered mode of transfer (CPUSS_BUFF_CTL[WRITE_BUFF]), the behavior during protection violation is different. When CPUSS_BUFF_CTL[WRITE_BUFF] is set to '1', the write transfers on the bus are buffered. So the transfer is first acknowledged when the buffer receives the transfer. A protection violation will be only evaluated when the actual write happens at the destination register. This leads to the write transfer not generating a bus error for buffered mode. However, a fault will be registered as soon as the transfer tries to write the destination location. Therefore, for buffered writes, the user must verify the fault structure to make sure no violations have occurred.

A protection violation results in a bus error and the bus transfer will not reach its target. An MPU or SMPU violation that targets a peripheral will not reach the associated protection evaluation (PPU). In other words, MPU and SMPU have a higher priority over PPU.

Protection units

Protection unit addresses the following:

- Security requirements. This includes prevention of malicious attacks to access secure memory or peripherals. For example, a non-secure master should not be able to access key information in a secure memory region.
- Safety requirements. This includes detection of accidental (non-malicious) software errors and random hardware errors. Enabling failure analysis is important so the root cause of a safety violation can be investigated. For example, analyzing a flash memory failure on a device that is returned from the field should be possible.

To address security requirements, the Cortex®-M0+ is used as a 'secure CPU'. This CPU is considered a trusted entity. Any access by the CPU tagged as "secure" will be called "secure access".

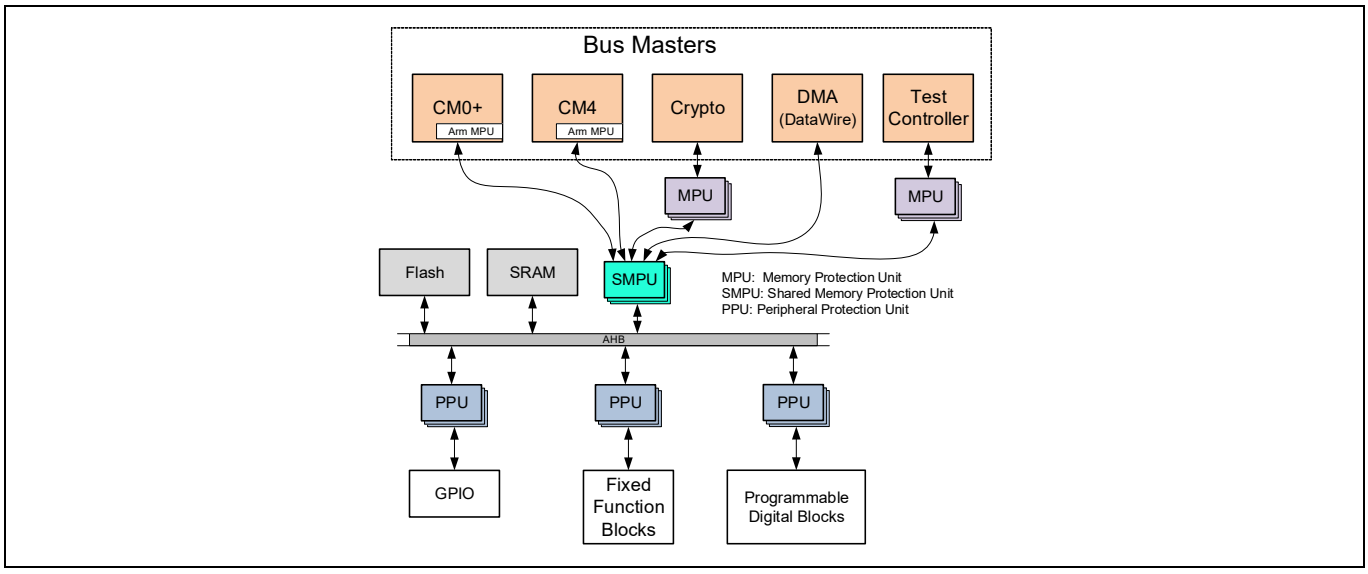


Figure 10-2. Protection architecture

The different types of protection units cater to different use cases for protection.

Table 10-1. Protection use cases

Protection unit type	Use
Arm® MPU	Used to protect memory between tasks within in a single Arm® core. A task in one of the Arm® cores can protect its memory from access by another task in the same core.
MPU	Same as the Arm® MPU, but for other bus masters such as the test controller or crypto, which do not have a built-in MPU in their block IP.
SMPU	Used to protect memory addresses that are shared between multiple bus masters.
Fixed PPU protection structures	These protect specific peripheral memory space. The protection structures have a preprogrammed memory region and can be used only to protect the peripheral it was intended for.
Programmable PPU protection structures	These protect the peripheral space but the memory region is not fixed. So users can easily program it to protect any space in the peripheral memory region.

10.3 Register architecture

The protection architecture has different conceptual pieces and different sets of registers correspond to each of these concepts.

10.3.1 Protection structure and attributes

The MPU, SMPU, and PPU protection structure definition follows the Arm® definition. Each protection structure is defined by:

- An address region
- Access control attributes

A protection structure is always aligned on a 32-byte boundary in the memory space. Two registers define a protection structure: ADDR (address register) and ATT (attribute register). This structure alignment and organization allow straightforward protection of the protection structures by the protection scheme. This is discussed later in this chapter.

Address region: The address region is defined by:

- The base address of a region as specified by ADDR.ADDR
- The size of a region as specified by ATT.REGION_SIZE
- Individual disables for eight subregions within the region, as specified by ADDR.SUBREGION_DISABLE.

The REGION_SIZE field specifies the size of a region. The region size is a power of 2 in the range of [256 B, 4 GB]. The base address ADDR specifies the start of the region, which must be aligned to the region size. A region is partitioned into eight equally sized sub-regions. The SUBREGION_DISABLE field specifies individual enables for the sub-regions within a region. For example, a REGION_SIZE of “0x08” specifies a region size of 512 bytes. If the start address is 0x1000:5400 (512-byte aligned), the region ranges from 0x1000:5400 to 0x1000:55ff. This region is partitioned into the following eight 64-byte subregions:

subregion 0 from 0x1000:5400 to 0x1000:543f

subregion 1 from 0x1000:5440 to 0x1000:547f

…

subregion 7 from 0x1000:55c0 to 0x1000:55ff.

If the SUBREGION_DISABLE is 0x82 (bitfields 1 and 7 are ‘1’), subregions 1 and 7 are disabled; subregions 0, 2, 3, 4, 5, and 6 are enabled.

In addition, an ATT.ENABLED field specifies whether the region is enabled. Only enabled regions participate in the protection “matching” process. Matching identifies if a bus transfer address is contained within an enabled subregion (SUBREGION_DISABLE) of an enabled region (ENABLED).

Protection attributes: The protection attributes specify access control to the region (shared by all subregions within the region). Access control is performed by comparing against a bus master's protection attributes of the bus master performing the transfer. The following access control fields are supported:

- Control for read accesses in user mode (ATT.UR field)
- Control for write accesses in user mode (ATT.UW field)
- Control for execute accesses in user mode (ATT.UX field)
- Control for read accesses in privileged mode (ATT.PR field)
- Control for write accesses in privileged mode (ATT.PW field)
- Control for execute accesses in privileged mode (ATT.PX field)
- Control for secure access (ATT.NS field)
- Control for individual protection contexts (ATT.PC_MASK[15:0], with MASK[0] always constant at 1). This protection context control field is present only for the SMPU and PPU.

Protection units

The execute and read access control attributes are orthogonal. Execute transfers are typically read transfers. To allow execute and read transfers in user mode, both ATT.UR and ATT.UX must be set to '1'. To allow data and read transfers in user mode, only ATT.UR must be set to '1'. In addition, the ATT.PC_MATCH control field is supported, which controls "matching" and "access evaluation" processes. This control field is present only for the SMPU and PPU protection structures.

For example, only protection context 2 can access a specific address range. These accesses are restricted to read and write secure accesses in privileged mode. The access control fields are programmed as follows:

- ATT.UR is 0: read accesses in user mode not allowed
- ATT.UW is 0: write accesses in user mode not allowed
- ATT.UX is 0: execute accesses in user mode not allowed
- ATT.PR is 1: read accesses in privileged mode allowed
- ATT.PW is 1: write accesses in privileged mode allowed
- ATT.PX is 0: execute accesses in privileged mode not allowed
- ATT.NS is 0: secure access required
- ATT.PC_MASK is 0x0005: protection context 1 and 3 accesses enabled (all other protection contexts are disabled)
- ATT.PC_MATCH is 0: the PC_MASK field is used for access evaluation. Three separate access evaluation subprocesses are distinguished:
  - A subprocess that evaluates the access based on read/write, execute, and user/privileged access attributes
  - A subprocess that evaluates the access based on the secure/non-secure attribute
  - A subprocess that evaluates the access based on the protection context index (used only by the SMPU and PPU when ATT.PC_MATCH is 0)

If all access evaluations are successful, access is allowed. If any process evaluation is unsuccessful, access is not allowed. Matching the bus transfer address and access evaluation of the bus transfer (based on access attributes) are two independent processes:

- Matching process. For each protection structure, the process identifies whether a transfer address is contained within the address range. This identifies the "matching" regions.
- Access evaluation process. For each protection structure, the process evaluates the bus transfer access attributes against the access control attributes.

A protection unit typically has multiple protection structures and evaluates the protection structures in decreasing order. The first matching structure provides the access control attributes for the evaluation of the transfer's access attributes. In other words, higher-indexed structures take precedence over lower-indexed structures.

The following pseudo code illustrates the process.

```
match = 0;
for (i = n-1; i >= 0; i--)// n: number of protection regions
   if (Match ("transfer address", "protection context"
              "MMIO registers ADDR and ATT of protection structure i")) {
      match = 1; break;
   }
|

if (match)
   AccessEvaluate ("transfer access attributes", "protection context"
                   "MMIO register ATT of protection structure i");
else
   "access allowed"
```

Note:

- If no protection structure provides a match, the access is allowed
- If multiple protection structures provide a match, the access control attributes for the access evaluation are provided by the protection structure with the highest index

An example of using the PC_MATCH feature is as follows. Two SMPU structures are configured to protect the same address range:

- Case 1: SMPU#2: PC = 3, PC_MATCH = 0 SMPU#1: PC = 2, PC_MATCH = 0
  To access the master of protection context 2, SMPU#2 has the highest index and address match, but attributes do not match; therefore, access is restricted. The SMPU#1 is not evaluated because the PC_MATCH is 0.
- Case 2: SMPU#2: PC = 3, PC_MATCH = 1 SMPU#1: PC = 2, PC_MATCH = 0
  The SMPU#2 address matches but PC does not match and is skipped because PC_MATCH is 1. SMPU#1 is evaluated and the address and attributes match; therefore, access is allowed.

As mentioned, the protection unit evaluates the protection structures in decreasing order. From a security requirements perspective, this is of importance: a non-secure protection context must not be able to add protection structures that have a higher index than the protection structures that provide secure access. The protection structure with a higher index can be programmed to allow non-secure accesses. Therefore, in a secure system, the higher programmable protection structures are protected to only allow restricted accesses

10.4 Bus master protection attributes

The protection structures set up the rules for different memory regions and their access attributes. The bus master's own protection attributes are used by the protection units to regulate access, based on rules set by the protection structures. Not all bus masters provide all protection attributes that are associated with a bus transfer. Some examples are:

- None of the bus masters has a native protection context attribute. This must be set dynamically based on the task being executed by the bus master.
- The Arm® Cortex®-M4 and Arm® Cortex®-M0+ CPUs provide a user/privilege attribute, but do not provide a secure/non-secure attribute natively. This must be set at a system level.

To ensure system-wide restricted access, missing attributes are provided by register fields. These fields may be set during the boot process or by the secure CPU.

- The SMPU MS_CTL.PC_MASK[] and MPU MS_CTL.PC[] register fields provide protection context functionality
- The SMPU MS_CTL.P register field provides the user/privileged attribute for those masters that do not provide their own attribute
- The SMPU MS_CTL.NS register provides the secure/non-secure attribute for those masters that do not provide their own attribute
- Masters that do not provide an execute attribute have the execute attribute set to '0'

The DMA controller channels inherit the access control attributes of the bus transfers that configured the DMA channel.

- All the bus masters in the system have SMPU and MPU MS_CTL registers associated with them
- The MPU MS_CTL.PC_SAVED field (and associated protection context 0 functionality, which is discussed later in the chapter) is only present for the CM0+ master
- The SMPU MS_CTL.P, MS_CTL.NS, and MS_CTL.PC_MASK fields are not present for the DMA. The bus transfer attributes are provided through "inheritance": the bus transfer attributes are from the master that owns the DMA channel that initiated the bus transfer
- The MPU MS_CTL register is not present for the DMA masters. The protection context (PC) bus transfer attribute is provided through inheritance

10.5 Protection context

Each bus master has a MPU MS_CTL.PC[3:0] protection context field. This protection context is used as the protection context attribute for all bus transfers that are initiated by the master. The SMPUs and PPUs allow or restrict bus transfers based on the protection context attribute.

Multiple masters can share a protection context. For example, a CPU and a crypto controlled by the CPU may share a protection context (the CPU and crypto PC[] fields are the same). Therefore, the CPU and crypto share the SMPU and PPU access restrictions.

A bus master protection context is changed by reprogramming the master's PC[] field. Changing a protection context is required for CPU bus masters that may transition between multiple tasks, each catering to different protection contexts. As the protection context allows or restricts bus transfers, changes to the protection context should be controlled and should not compromise security. Furthermore, changes to the protection context should incur limited CPU overhead to allow for frequent protection context changes. Consider a case in which a CPU executes two software stacks with different protection contexts. To this end, each bus master has an SMPU MS_CTL.PC_MASK[15:0] protection context mask field that identifies what protection contexts can be programmed for the bus master:

- The protection context field MS_CTL.PC[3:0]. This register is controlled by the associated bus master and has the same access restrictions as the bus master's MPU registers.
- The protection context mask field MS_CTL.PC_MASK[15:0]. This register is controlled by the secure CPU and has the same access restrictions as the SMPU registers.

The PC_MASK[] field is a "hot-one" field that specifies whether the PC[] field can be programmed with a specific protection context. Consider an attempt to program PC[] to '3':

- If PC_MASK[3] is '1', PC[] is set to "3"
- If PC_MASK[3] is '0', PC[] is not changed

10.6 Protection contexts 0, 1, 2, 3

The EZ-USB™ FX10 MCU supports protection contexts to isolate software execution for security and safety purposes. Protection contexts are used to restrict access to memory and peripheral resources.

Out of a maximum of 16 protections contexts (PCs), four PCs are treated special: the entry to special PCs 0, 1, 2, and 3 is hardware-controlled. For each PC i, a programmable exception handler address is provided (CM0_PCi_HANDLER.ADDR [31:0]). A CPU exception handler fetch, which returns a handler address that matches the programmed CM0_PCi_HANDLER.ADDR[31:0] address value, causes the CM0+ PC to be changed to PC i by hardware. However, if the current PC is already 0, 1, 2, or 3, the current PC is not changed (an attempt to change the PC results in an AHB-Lite bus error). This ensures that CPU execution in PC 0, 1, 2, or 3 cannot be interrupted/preempted by CPU execution in another PC 0, 1, 2, or 3. In other words, CPU execution in PC 0, 1, 2, or 3 requires cooperative multi-tasking between the different PCs. This means that hand-overs between different PCs are software-scheduled/controlled. A security implementation requires PC software to clean/zeroize information that it wants to keep confidential from other PC software.

Protection units

The following pseudo code gives a description of the hardware control over the CM0+ current PC “pc”:

```
if      (CM0_PC_CTL.VALID[0] & (addr == CM0_PC0_HANDLER. ADDR)) {match_new, pc_new} =
{1, 0};
else if (CM0_PC_CTL.VALID[1] & (addr == CM0_PC1_HANDLER.ADDR)) {match_new, pc_new} =
{1, 1};
else if (CM0_PC_CTL.VALID[2] & (addr == CM0_PC2_HANDLER.ADDR)) {match_new, pc_new} =
{1, 2};
else if (CM0_PC_CTL.VALID[3] & (addr == CM0_PC3_HANDLER.ADDR)) {match_new, pc_new} =
{1, 3};
else                                                          match_new          =  0;

if ("exception handler fetch")
begin
   if (match_new)
   begin
      if      (pc == pc_new)          ;
      else if (CM0_PC_CTL.VALID[pc]) "AHB-Lite bus error";
      else                           {pc, pc_saved} = {pc_new,   pc};     // "push"
   end
   else
   begin
                                    {pc, pc_saved} = {pc_saved, pc_saved};   // "pop"
   end
end
```

Note that each of the protection special PCs 0, 1, 2, and 3 have a dedicated CM0_PC_CTL.VALID[i] field to specify that the PC's exception handler address is provided through CM0_PCi_HANDLER.ADDR[31:0]. If a PC's exception handler address is not provided, the PC is treated as an ordinary PC (PCs 4, 5, ..., 15). Note that the current PC “pc” and a saved PC “pc_saved” implement a two-entry stack. Hardware pushes the current PC to the stack upon entry of a special exception handler. Hardware then pushes the saved PC from the stack upon entry of an ordinary exception handler. An attempt to enter a special exception handler from a special exception handler with a different PC results in an AHB-Lite bus error (which causes the CPU to enter the bus fault exception handler). This scenario should not occur in a carefully designed cooperative multi-tasking software implementation.

Of the four special PCs, PC 0 is treated differently:

- It is the default PC value after a Deep Sleep reset
- It has unrestricted access; that is, it is not affected by the CPUSS and PERI protection schemes. Therefore, the Infineon boot code software always starts execution in PC 0. The boot code software initializes the protection structures and initializes the CM0_PCi_HANDLER registers. The boot code is considered “trusted” software and its unrestricted access in PC 0 is not considered a protection concern. After initialization of the protection information, the access to the protection information itself is typically restricted for all other PCs (the boot code software deploys the restrictions) and the protection information provides specific restricted access to the other special PCs and ordinary PCs (1, 2, ..., 15). As part of the boot process configuration, the following things happen:
  - The boot process sets MPUn.MS_CTL.PC_SAVED to a context that is not a special. context. This is required to prevent accidental/malicious incorrect entry into a special context.
  - If the system configuration does not use PC=0 after boot is complete, the boot process disables all future use of PC=0 by making ineffective CPUSS.CM0_PC0_HANDLER by clearing CPUSS.CM0_PC_CTL.VALID[0]. Furthermore, the protection region CPUSS.BOOT is assigned to be accessible from PC=0 only, which ensures these settings can no longer be changed after boot.

- The boot process then changes the current PC by changing MPUn.MS_CTL.PC to a protection context that is not a special context. This is done for all currently active masters (which typically is only the CM0+).
- When the above is complete, the system executes a regular (not special) context and can enter into special contexts only through the use of interrupts using the special handler addresses.

10.7 Protection structure

10.7.1 Protection violation

If an MPU, SMPU, or PPU detects a not-allowed transfer, the bus transfer results in a bus error. The bus transfer does not reach its target memory location or peripheral register. In addition, information on the violating bus transfer is communicated to the fault report structure.

10.7.2 MPU

The MPUs are situated in the CPUSS and are associated to a single master. An MPU distinguishes user and privileged accesses from a single bus master. However, the capability exists to perform access control on the secure/non-secure attribute.

As an MPU is associated to a single master, the MPU protection structures do not provide protection context control attributes.

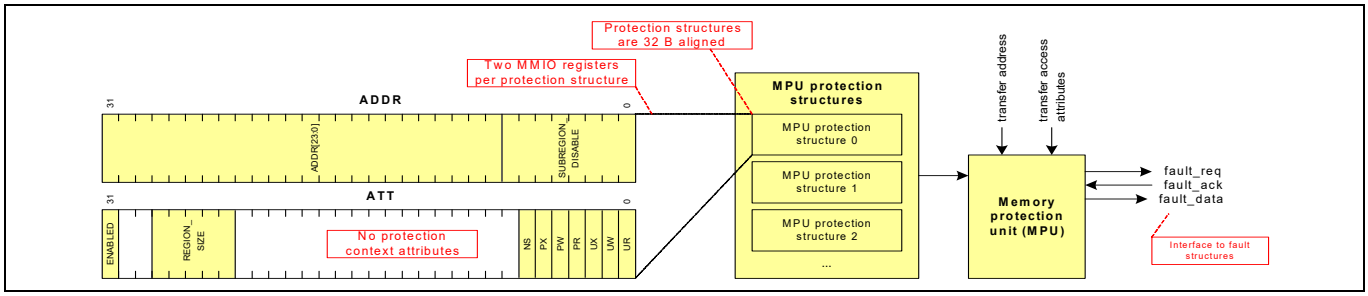


Figure 10-3. MPU functionality

10.7.3 SMPU

The SMPU is situated in the CPUSS and is shared by all bus masters. The SMPU distinguishes between different protection contexts and distinguishes secure from non-secure accesses. However, the capability exists to perform access control on the user/privileged mode attribute.

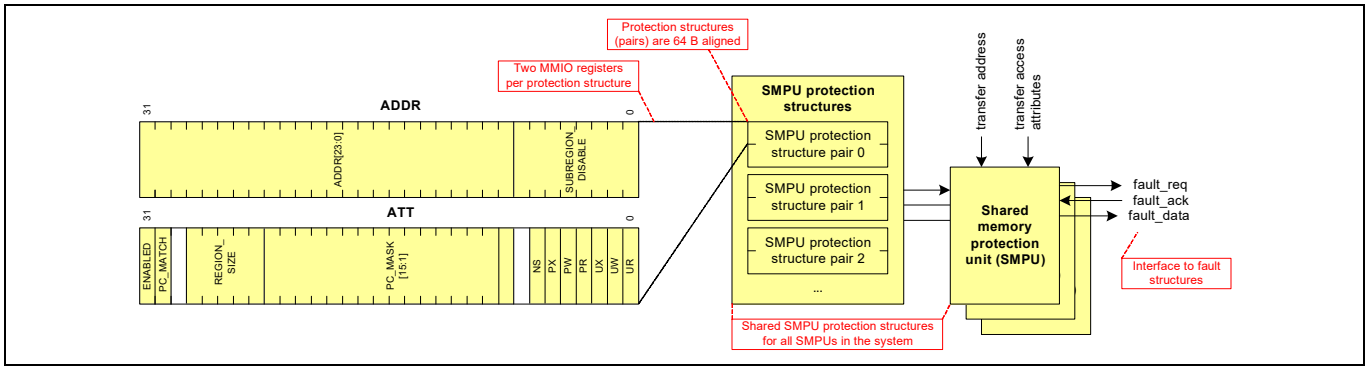


Figure 10-4. SMPU functionality

Note that a single set of SMPU region structures provides the same protection information to all SMPUs in the systems.

10.7.4 PPU

- The PPUs are situated in the PERI block and are associated with a peripheral group (a group of peripherals with a shared AHB-Lite bus infrastructure). A PPU is shared by all bus masters. The PPU distinguishes between different protection contexts; it also distinguishes secure from non-secure accesses and user mode from privileged mode accesses.

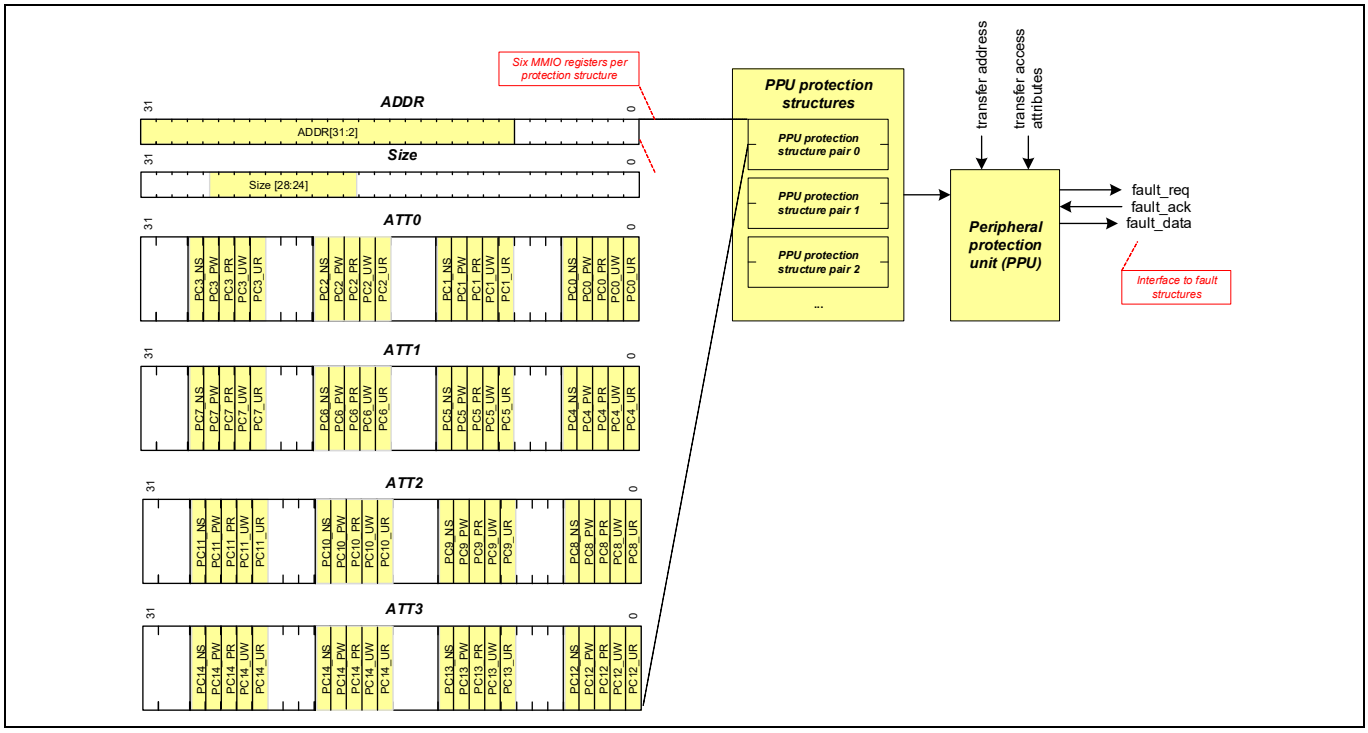


Figure 10-5. PPU functionality

There are two types of PPU structures: Fixed and Programmable.

- The fixed PPU structures protect fixed areas of memory and hence a specific predetermined peripheral region. In other words, the ADDR, SUBREGION_DISABLE, and REGION_SIZE fields are fixed for a specific device. See the registers reference manual for a definition of fixed PPUs and the address regions they protect. Their protection attributes can be configured for each protection context. **Table 10-2** summarizes the different fixed PPU structures in EZ-USB™ FX10 devices and details the regions they protect.

Table 10-2. Fixed PPU structures

Protection structure	Base address of protected region	Size of protected region	Protected peripheral/block(s)
PERI_MS_PPU_FX0	0x40000000	8 KB	PERI
PERI_MS_PPU_FX1	0x40004010	4 B	PERI_GR0
PERI_MS_PPU_FX2	0x40004030	4 B	PERI_GR1
PERI_MS_PPU_FX3	0x40004050	4 B	PERI_GR2
PERI_MS_PPU_FX4	0x40004060	32 B	PERI_GR3
PERI_MS_PPU_FX5	0x40004080	32 B	PERI_GR4
PERI_MS_PPU_FX6	0x400040C0	32 B	PERI_GR6
PERI_MS_PPU_FX7	0x40004120	32 B	PERI_GR9
PERI_MS_PPU_FX8	0x40004140	32 B	PERI_GR10
PERI_MS_PPU_FX9	0x40008000	32 KB	PERI_TR
PERI_MS_PPU_FX10	0x40100000	1 KB	CRYPTO
PERI_MS_PPU_FX11	0x40101000	2 KB	CRYPTO
PERI_MS_PPU_FX12	0x40102000	256 B	CRYPTO
PERI_MS_PPU_FX13	0x40102100	4 B	CRYPTO
PERI_MS_PPU_FX14	0x40102120	4 B	CRYPTO
PERI_MS_PPU_FX15	0x40108000	4 KB	CRYPTO
PERI_MS_PPU_FX16	0x40200000	1 KB	CPUSS
PERI_MS_PPU_FX17	0x40201000	4 KB	CPUSS
PERI_MS_PPU_FX18	0x40202000	512 B	CPUSS
PERI_MS_PPU_FX19	0x40208000	1 KB	CPUSS_CM0_SYSTEM_INT
PERI_MS_PPU_FX20	0x4020A000	1 KB	CPUSS_CM4_SYSTEM_INT
PERI_MS_PPU_FX21	0x40210000	256 B	FAULT_STRUCT0
PERI_MS_PPU_FX22	0x40210100	256 B	FAULT_STRUCT1
PERI_MS_PPU_FX23	0x40220000	32 B	IPC_STRUCT0
PERI_MS_PPU_FX24	0x40220020	32 B	IPC_STRUCT1
PERI_MS_PPU_FX25	0x40220040	32 B	IPC_STRUCT2
PERI_MS_PPU_FX26	0x40220060	32 B	IPC_STRUCT3
PERI_MS_PPU_FX27	0x40220080	32 B	IPC_STRUCT4
PERI_MS_PPU_FX28	0x402200A0	32 B	IPC_STRUCT5
PERI_MS_PPU_FX29	0x402200C0	32 B	IPC_STRUCT6
PERI_MS_PPU_FX30	0x402200E0	32 B	IPC_STRUCT7
PERI_MS_PPU_FX31	0x40220100	32 B	IPC_STRUCT8
PERI_MS_PPU_FX32	0x40220120	32 B	IPC_STRUCT9
PERI_MS_PPU_FX33	0x40220140	32 B	IPC_STRUCT10
PERI_MS_PPU_FX34	0x40220160	32 B	IPC_STRUCT11
PERI_MS_PPU_FX35	0x40220180	32 B	IPC_STRUCT12
PERI_MS_PPU_FX36	0x402201A0	32 B	IPC_STRUCT13
PERI_MS_PPU_FX37	0x402201C0	32 B	IPC_STRUCT14

Table 10-2. Fixed PPU structures (continued)

Protection structure	Base address of protected region	Size of protected region	Protected peripheral/block(s)
PERI_MS_PPU_FX38	0x402201E0	32 B	IPC_STRUCT15
PERI_MS_PPU_FX39	0x40221000	16 B	IPC_INTR_STRUCT0
PERI_MS_PPU_FX40	0x40221020	16 B	IPC_INTR_STRUCT1
PERI_MS_PPU_FX41	0x40221040	16 B	IPC_INTR_STRUCT2
PERI_MS_PPU_FX42	0x40221060	16 B	IPC_INTR_STRUCT3
PERI_MS_PPU_FX43	0x40221080	16 B	IPC_INTR_STRUCT4
PERI_MS_PPU_FX44	0x402210A0	16 B	IPC_INTR_STRUCT5
PERI_MS_PPU_FX45	0x402210C0	16 B	IPC_INTR_STRUCT6
PERI_MS_PPU_FX46	0x402210E0	16 B	IPC_INTR_STRUCT7
PERI_MS_PPU_FX47	0x40221100	16 B	IPC_INTR_STRUCT8
PERI_MS_PPU_FX48	0x40221120	16 B	IPC_INTR_STRUCT9
PERI_MS_PPU_FX49	0x40221140	16 B	IPC_INTR_STRUCT10
PERI_MS_PPU_FX50	0x40221160	16 B	IPC_INTR_STRUCT11
PERI_MS_PPU_FX51	0x40221180	16 B	IPC_INTR_STRUCT12
PERI_MS_PPU_FX52	0x402211A0	16 B	IPC_INTR_STRUCT13
PERI_MS_PPU_FX53	0x402211C0	16 B	IPC_INTR_STRUCT14
PERI_MS_PPU_FX54	0x402211E0	16 B	IPC_INTR_STRUCT15
PERI_MS_PPU_FX55	0x40230000	64 B	PROT_SMPU
PERI_MS_PPU_FX56	0x40234000	4 B	PROT_MPU0
PERI_MS_PPU_FX57	0x40235400	1 KB	PROT_MPU5
PERI_MS_PPU_FX58	0x40235800	1 KB	PROT_MPU6
PERI_MS_PPU_FX59	0x40237800	4 B	PROT_MPU14
PERI_MS_PPU_FX60	0x40237C00	1 KB	PROT_MPU15
PERI_MS_PPU_FX61	0x40240000	8 B	FLASHC_FLASH
PERI_MS_PPU_FX62	0x40240008	4 B	FLASHC_FLASH
PERI_MS_PPU_FX63	0x40240200	256 B	FLASHC
PERI_MS_PPU_FX64	0x40240400	128 B	FLASHC_CM0
PERI_MS_PPU_FX65	0x40240480	128 B	FLASHC_CM4
PERI_MS_PPU_FX66	0x40240500	4 B	FLASHC_CRYPTO
PERI_MS_PPU_FX67	0x40240580	4 B	FLASHC_DW0
PERI_MS_PPU_FX68	0x40240600	4 B	FLASHC_DW1
PERI_MS_PPU_FX69	0x40240680	4 B	FLASHC_DMAC
PERI_MS_PPU_FX70	0x40240700	4 B	FLASHC_EXT_MS0
PERI_MS_PPU_FX71	0x40240780	4 B	FLASHC_EXT_MS1
PERI_MS_PPU_FX72	0x4024F000	4 KB	FLASHC_FM
PERI_MS_PPU_FX73	0x40260000	256 B	PWR
PERI_MS_PPU_FX74	0x40260100	16 B	
PERI_MS_PPU_FX75	0x40260180	16 B	WDT
PERI_MS_PPU_FX76	0x40260200	128 B	MCWDT

Table 10-2. Fixed PPU structures (continued)

Protection structure	Base address of protected region	Size of protected region	Protected peripheral/block(s)
PERI_MS_PPU_FX77	0x40260300	256 B	CLK
PERI_MS_PPU_FX78	0x40260400	1 KB	CLK
PERI_MS_PPU_FX79	0x40260800	8 B	RES
PERI_MS_PPU_FX80	0x40267000	4 KB	PWR_TRIM, CLK_TRIM
PERI_MS_PPU_FX81	0x4026FF00	128 B	PWR_TRIM, CLK_TRIM
PERI_MS_PPU_FX83	0x40280000	128 B	DW00
PERI_MS_PPU_FX84	0x40290000	128 B	DW10
PERI_MS_PPU_FX85	0x40280100	128 B	DW0_CRC
PERI_MS_PPU_FX86	0x40290100	128 B	DW1_CRC
PERI_MS_PPU_FX87	0x40288000	64 B	DW0_CH_STRUCT0
PERI_MS_PPU_FX88	0x40288040	64 B	DW0_CH_STRUCT1
PERI_MS_PPU_FX89	0x40288080	64 B	DW0_CH_STRUCT2
PERI_MS_PPU_FX90	0x402880C0	64 B	DW0_CH_STRUCT3
PERI_MS_PPU_FX91	0x40288100	64 B	DW0_CH_STRUCT4
PERI_MS_PPU_FX92	0x40288140	64 B	DW0_CH_STRUCT5
PERI_MS_PPU_FX93	0x40288180	64 B	DW0_CH_STRUCT6
PERI_MS_PPU_FX94	0x402881C0	64 B	DW0_CH_STRUCT7
PERI_MS_PPU_FX95	0x40288200	64 B	DW0_CH_STRUCT8
PERI_MS_PPU_FX96	0x40288240	64 B	DW0_CH_STRUCT9
PERI_MS_PPU_FX97	0x40288280	64 B	DW0_CH_STRUCT10
PERI_MS_PPU_FX98	0x402882C0	64 B	DW0_CH_STRUCT11
PERI_MS_PPU_FX99	0x40288300	64 B	DW0_CH_STRUCT12
PERI_MS_PPU_FX100	0x40288340	64 B	DW0_CH_STRUCT13
PERI_MS_PPU_FX101	0x40288380	64 B	DW0_CH_STRUCT14
PERI_MS_PPU_FX102	0x402883C0	64 B	DW0_CH_STRUCT15
PERI_MS_PPU_FX103	0x40288400	64 B	DW0_CH_STRUCT16
PERI_MS_PPU_FX104	0x40288440	64 B	DW0_CH_STRUCT17
PERI_MS_PPU_FX105	0x40288480	64 B	DW0_CH_STRUCT18
PERI_MS_PPU_FX106	0x402884C0	64 B	DW0_CH_STRUCT19
PERI_MS_PPU_FX107	0x40288500	64 B	DW0_CH_STRUCT20
PERI_MS_PPU_FX108	0x40288540	64 B	DW0_CH_STRUCT21
PERI_MS_PPU_FX109	0x40288580	64 B	DW0_CH_STRUCT22
PERI_MS_PPU_FX110	0x402885C0	64 B	DW0_CH_STRUCT23
PERI_MS_PPU_FX111	0x40288600	64 B	DW0_CH_STRUCT24
PERI_MS_PPU_FX112	0x40288640	64 B	DW0_CH_STRUCT25
PERI_MS_PPU_FX113	0x40288680	64 B	DW0_CH_STRUCT26
PERI_MS_PPU_FX114	0x402886C0	64 B	DW0_CH_STRUCT27
PERI_MS_PPU_FX115	0x40288700	64 B	DW0_CH_STRUCT28
PERI_MS_PPU_FX116	0x40298000	64 B	DW1_CH_STRUCT0

Table 10-2. Fixed PPU structures (continued)

Protection structure	Base address of protected region	Size of protected region	Protected peripheral/block(s)
PERI_MS_PPU_FX117	0x40298040	64 B	DW1_CH_STRUCT1
PERI_MS_PPU_FX118	0x40298080	64 B	DW1_CH_STRUCT2
PERI_MS_PPU_FX119	0x402980C0	64 B	DW1_CH_STRUCT3
PERI_MS_PPU_FX120	0x40298100	64 B	DW1_CH_STRUCT4
PERI_MS_PPU_FX121	0x40298140	64 B	DW1_CH_STRUCT5
PERI_MS_PPU_FX122	0x40298180	64 B	DW1_CH_STRUCT6
PERI_MS_PPU_FX123	0x402981C0	64 B	DW1_CH_STRUCT7
PERI_MS_PPU_FX124	0x40298200	64 B	DW1_CH_STRUCT8
PERI_MS_PPU_FX125	0x40298240	64 B	DW1_CH_STRUCT9
PERI_MS_PPU_FX126	0x40298280	64 B	DW1_CH_STRUCT10
PERI_MS_PPU_FX127	0x402982C0	64 B	DW1_CH_STRUCT11
PERI_MS_PPU_FX128	0x40298300	64 B	DW1_CH_STRUCT12
PERI_MS_PPU_FX129	0x40298340	64 B	DW1_CH_STRUCT13
PERI_MS_PPU_FX130	0x40298380	64 B	DW1_CH_STRUCT14
PERI_MS_PPU_FX131	0x402983C0	64 B	DW1_CH_STRUCT15
PERI_MS_PPU_FX132	0x40298400	64 B	DW1_CH_STRUCT16
PERI_MS_PPU_FX133	0x40298440	64 B	DW1_CH_STRUCT17
PERI_MS_PPU_FX134	0x40298480	64 B	DW1_CH_STRUCT18
PERI_MS_PPU_FX135	0x402984C0	64 B	DW1_CH_STRUCT19
PERI_MS_PPU_FX136	0x40298500	64 B	DW1_CH_STRUCT20
PERI_MS_PPU_FX137	0x40298540	64 B	DW1_CH_STRUCT21
PERI_MS_PPU_FX138	0x40298580	64 B	DW1_CH_STRUCT22
PERI_MS_PPU_FX139	0x402985C0	64 B	DW1_CH_STRUCT23
PERI_MS_PPU_FX140	0x40298600	64 B	DW1_CH_STRUCT24
PERI_MS_PPU_FX141	0x40298640	64 B	DW1_CH_STRUCT25
PERI_MS_PPU_FX142	0x40298680	64 B	DW1_CH_STRUCT26
PERI_MS_PPU_FX143	0x402986C0	64 B	DW1_CH_STRUCT27
PERI_MS_PPU_FX144	0x40298700	64 B	DW1_CH_STRUCT28
PERI_MS_PPU_FX145	0x402A0000	16 B	DMAC
PERI_MS_PPU_FX146	0x402A1000	256 B	DMAC_CH0
PERI_MS_PPU_FX147	0x402A1100	256 B	DMAC_CH1
PERI_MS_PPU_FX148	0x402A1200	256 B	DMAC_CH2
PERI_MS_PPU_FX149	0x402A1300	256 B	DMAC_CH3
PERI_MS_PPU_FX150	0x402C0000	128 B	EFUSE
PERI_MS_PPU_FX151	0x402C0800	512 B	EFUSE
PERI_MS_PPU_FX152	0x402D0000	64 KB	PROFILE
PERI_MS_PPU_FX153	0x40300000	8 B	HSIOM_PRT0_PORT
PERI_MS_PPU_FX154	0x40300010	8 B	HSIOM_PRT1_PORT
PERI_MS_PPU_FX155	0x40300020	8 B	HSIOM_PRT2_PORT

Table 10-2. Fixed PPU structures (continued)

Protection structure	Base address of protected region	Size of protected region	Protected peripheral/block(s)
PERI_MS_PPU_FX156	0x40300030	8 B	HSIOM_PRT3_PORT
PERI_MS_PPU_FX157	0x40300040	8 B	HSIOM_PRT4_PORT
PERI_MS_PPU_FX158	0x40300050	8 B	HSIOM_PRT5_PORT
PERI_MS_PPU_FX159	0x40300060	8 B	HSIOM_PRT6_PORT
PERI_MS_PPU_FX160	0x40300070	8 B	HSIOM_PRT7_PORT
PERI_MS_PPU_FX161	0x40300080	8 B	HSIOM_PRT8_PORT
PERI_MS_PPU_FX162	0x40300090	8 B	HSIOM_PRT9_PORT
PERI_MS_PPU_FX163	0x403000A0	8 B	HSIOM_PRT10_PORT
PERI_MS_PPU_FX164	0x403000B0	8 B	HSIOM_PRT11_PORT
PERI_MS_PPU_FX165	0x403000C0	8 B	HSIOM_PRT12_PORT
PERI_MS_PPU_FX166	0x403000D0	8 B	HSIOM_PRT13_PORT
PERI_MS_PPU_FX167	0x403000E0	8 B	HSIOM_PRT14_PORT
PERI_MS_PPU_FX168	0x40302000	32 B	HSIOM_AMUX_SPLIT
PERI_MS_PPU_FX169	0x40302200	16 B	
PERI_MS_PPU_FX170	0x40310000	64 B	GPIO_PRT0
PERI_MS_PPU_FX171	0x40310080	64 B	GPIO_PRT1
PERI_MS_PPU_FX172	0x40310100	64 B	GPIO_PRT2
PERI_MS_PPU_FX173	0x40310180	64 B	GPIO_PRT3
PERI_MS_PPU_FX174	0x40310200	64 B	GPIO_PRT4
PERI_MS_PPU_FX175	0x40310280	64 B	GPIO_PRT5
PERI_MS_PPU_FX176	0x40310300	64 B	GPIO_PRT6
PERI_MS_PPU_FX177	0x40310380	64 B	GPIO_PRT7
PERI_MS_PPU_FX178	0x40310400	64 B	GPIO_PRT8
PERI_MS_PPU_FX179	0x40310480	64 B	GPIO_PRT9
PERI_MS_PPU_FX180	0x40310500	64 B	GPIO_PRT10
PERI_MS_PPU_FX181	0x40310580	64 B	GPIO_PRT11
PERI_MS_PPU_FX182	0x40310600	64 B	GPIO_PRT12
PERI_MS_PPU_FX183	0x40310680	64 B	GPIO_PRT13
PERI_MS_PPU_FX184	0x40310700	64 B	GPIO_PRT14
PERI_MS_PPU_FX185	0x40310040	16 B	GPIO_PRT0
PERI_MS_PPU_FX186	0x403100C0	16 B	GPIO_PRT1
PERI_MS_PPU_FX187	0x40310140	16 B	GPIO_PRT2
PERI_MS_PPU_FX188	0x403101C0	16 B	GPIO_PRT3
PERI_MS_PPU_FX189	0x40310240	16 B	GPIO_PRT4
PERI_MS_PPU_FX190	0x403102C0	16 B	GPIO_PRT5
PERI_MS_PPU_FX191	0x40310340	16 B	GPIO_PRT6
PERI_MS_PPU_FX192	0x403103C0	16 B	GPIO_PRT7
PERI_MS_PPU_FX193	0x40310440	16 B	GPIO_PRT8
PERI_MS_PPU_FX194	0x403104C0	16 B	GPIO_PRT9

Protection units

Table 10-2. Fixed PPU structures (continued)

Protection structure	Base address of protected region	Size of protected region	Protected peripheral/block(s)
PERI_MS_PPU_FX195	0x40310540	16 B	GPIO_PRT10
PERI_MS_PPU_FX196	0x403105C0	16 B	GPIO_PRT11
PERI_MS_PPU_FX197	0x40310640	16 B	GPIO_PRT12
PERI_MS_PPU_FX198	0x403106C0	16 B	GPIO_PRT13
PERI_MS_PPU_FX199	0x40310740	8 B	GPIO_PRT14
PERI_MS_PPU_FX200	0x40314000	64 B	GPIO_INTR
PERI_MS_PPU_FX201	0x40315000	8 B	
PERI_MS_PPU_FX202	0x40320800	256 B	SMARTIO_PRT8
PERI_MS_PPU_FX203	0x40320900	256 B	SMARTIO_PRT9
PERI_MS_PPU_FX204	0x40350000	64 KB	LPCOMP
PERI_MS_PPU_FX206	0x40380000	64 KB	TCPWM0
PERI_MS_PPU_FX207	0x40390000	64 KB	TCPWM1
PERI_MS_PPU_FX209	0x403F0000	64 KB	USBFS0
PERI_MS_PPU_FX210	0x40420000	64 KB	SMIF0
PERI_MS_PPU_FX213	0x40600000	64 KB	SCB0
PERI_MS_PPU_FX214	0x40610000	64 KB	SCB1
PERI_MS_PPU_FX215	0x40620000	64 KB	SCB2
PERI_MS_PPU_FX216	0x40630000	64 KB	SCB3
PERI_MS_PPU_FX217	0x40640000	64 KB	SCB4
PERI_MS_PPU_FX218	0x40650000	64 KB	SCB5
PERI_MS_PPU_FX219	0x40660000	64 KB	SCB6
PERI_MS_PPU_FX220	0x40670000	64 KB	SCB7
PERI_MS_PPU_FX221	0x40680000	64 KB	SCB8
PERI_MS_PPU_FX222	0x40690000	64 KB	SCB9
PERI_MS_PPU_FX223	0x406A0000	64 KB	SCB10
PERI_MS_PPU_FX224	0x406B0000	64 KB	SCB11
PERI_MS_PPU_FX225	0x406C0000	64 KB	SCB12
PERI_MS_PPU_FX226	0x40A00000	4 KB	PDM0
PERI_MS_PPU_FX227	0x40A10000	4 KB	I2S0
PERI_MS_PPU_FX228	0x40A11000	4 KB	I2S1

Protection units

- The programmable PPU structures can have configurable address regions. Similar to fixed-PPU structures, the protection attributes of programmable PPU structures can be configured for each protection context. Programmable PPU structures are similar to SMPU structures but are intended to be used with the peripheral register space. These protection structures are typically used to protect registers in a specific block, which are not covered by the resolution of fixed PPU structures.

  Note that the memory regions of the fixed master structures, fixed slave structures, and programmable master structures are fixed by hardware and are mutually exclusive; that is, they do not overlap. The memory regions of the programmable slave structure are software-programmable and can potentially overlap. Therefore, it is important to assign priority to the protection structure matching process. The order in which these are evaluated are as follows:
  - The fixed master structures are evaluated in decreasing order
  - The fixed slave structures are evaluated in decreasing order
  - The programmable master structures are evaluated in decreasing order
  - The programmable slave structures are evaluated in decreasing order

    The programmable slave structures are evaluated last. These structures are software-programmable and can potentially overlap (overlapping should not allow software to circumvent the protection as provided by the fixed protection structure pairs).

    Each peripheral group has a dedicated PPU. The protection information is provided by peripheral group MMIO registers. A peripheral group PPU uses fixed protection structure pairs for two purposes.
  - Fixed protection structure pairs protect peripherals (one pair for each peripheral). The master structure protects the MMIO registers of the pair (the memory region encompasses the MMIO registers of the pair's master structure and slave structure). The slave structure protects the peripheral (the memory region encompasses the peripheral address region).
  - Fixed protection structure pairs protect specific peripheral subregions (one pair for each subregion). The master structure protects the MMIO registers of the pair. The slave structure protects the peripheral subregion. These pairs can be used to protect, for example, individual DW channels in the DW peripheral or individual IPC structures in the IPC peripheral.

    Note that the memory regions of the fixed peripheral master structures, fixed peripheral slave structures, and fixed peripheral subregion master structures are fixed by hardware and are mutually exclusive; that is, they do not overlap. The memory regions of the fixed peripheral subregion slave structures are fixed by hardware and typically are a subset of a peripheral address region, and therefore overlap with a fixed peripheral slave structure. Therefore, it is important to assign priority to how the protection structure matching process:
  - The fixed peripheral subregion master structures are evaluated in decreasing order
  - The fixed peripheral subregion slave structures are evaluated in decreasing order
  - The fixed peripheral master structures are evaluated in decreasing order
  - The fixed peripheral slave structures are evaluated in decreasing order

    It is important to evaluate the fixed peripheral subregion master structures first. This allows software to assign different protection for a subregion of a peripheral.

10.7.5 Protection of protection structures

The MPU, SMPU, and PPU-based protection architecture is consistent and provides the flexibility to implement different system-wide protection schemes. Protection structures can be set once at boot time or can be changed dynamically during device execution. For example, a CPU RTOS can change the CPU's MPU settings; a secure CPU can change the SMPU and PPUs settings. But such a system will be left insecure if there is no way to protect the protection structures themselves. There must be a way to restrict access to the protection structures.

The protection of protection structures is achieved using another protection structure. For this reason, protection structures are defined in pairs of master and slave. We refer to the slave and master protection structures as a protection pair. Note that the address range of the master protection structure is known, that is, it is constant.

The first (slave) protection structure protects the resource and the second (master) protection structure protects the protection (address range of the second protection structure includes both the master and slave protection structures).

The protection architecture is flexible enough to allow for variations:

- Exclusive peripheral ownership can be shared by more than two protection contexts
- The ability to change ownership is under control of a single protection context, and exclusive or non-exclusive peripheral ownership is shared by multiple protection contexts

Note that in secure systems, typically a single secure CPU is used. In these systems, the ability to change ownership is assigned to the secure CPU at boot time and not dynamically changed. Therefore, you must assign the secure CPU its own, dedicated protection context.

Both PPU and SMPU is intended to distinguish between different protection contexts and to distinguish secure from non-secure accesses. Therefore, both PPU and SMPU protection use protection structure pairs. In the SMPU, the slave protection structure provides SMPU protection information and the master protection structure provides PPU protection information (the master and slave protection structures are registers).

10.7.6 Protection structure types

Different protection structure types are used because some resources, such as peripheral registers, have a fixed address range. Protection of protection structures requires pairs of neighboring protection structures.

Three types of protection structures with a consistent register interface are described:

- **Programmable protection structures:** These are 32-byte protection structures with a programmable address range. These structures are used by the MPUs
- **Fixed protection structure pairs.:**These are 64-byte master/slave protection structure pairs, consisting of two 32-byte protection structures. These structures are used by the PPUs. Both structures have a fixed, constant address region. The master structure has the UX and PX attributes as constant '0' (execution is never allowed) and the UR and PR attributes as constant '1' (reading is always allowed). The slave structure has the UX and PX attributes as constant '1'.
- **Programmable protection structure pairs:** These are 64-byte master/slave protection structure pairs, consisting of two 32-byte protection structures. These structures are used by the PPU and SMPU. The master structure has a fixed, constant address region. The slave structure has a programmable address region. The master structure has the UX and PX attributes as constant '0' (execution is never allowed) and the UR and PR attributes as constant '1' (reading is always allowed). The PPU slave structure has the UX and PX attributes as constant '1'. The SMPU slave structure has programmable UX and PX attributes.

Note that the master protection structure in a protection structure pair is required only to address security requirements. The distinction between the three protection structure types is an implementation optimization. From an architectural perspective, all PPU protection structures are the same, with the exception that for some protection structures the address range is fixed and not programmable.

Protection units

As mentioned earlier, a protection unit evaluates the protection regions in decreasing protection structure index order. The protection structures are evaluated in the following order:

- Fixed protection structures for specific peripherals or peripheral register address ranges
- Programmable protection structures

Therefore, the fixed structures precede programmable structures.

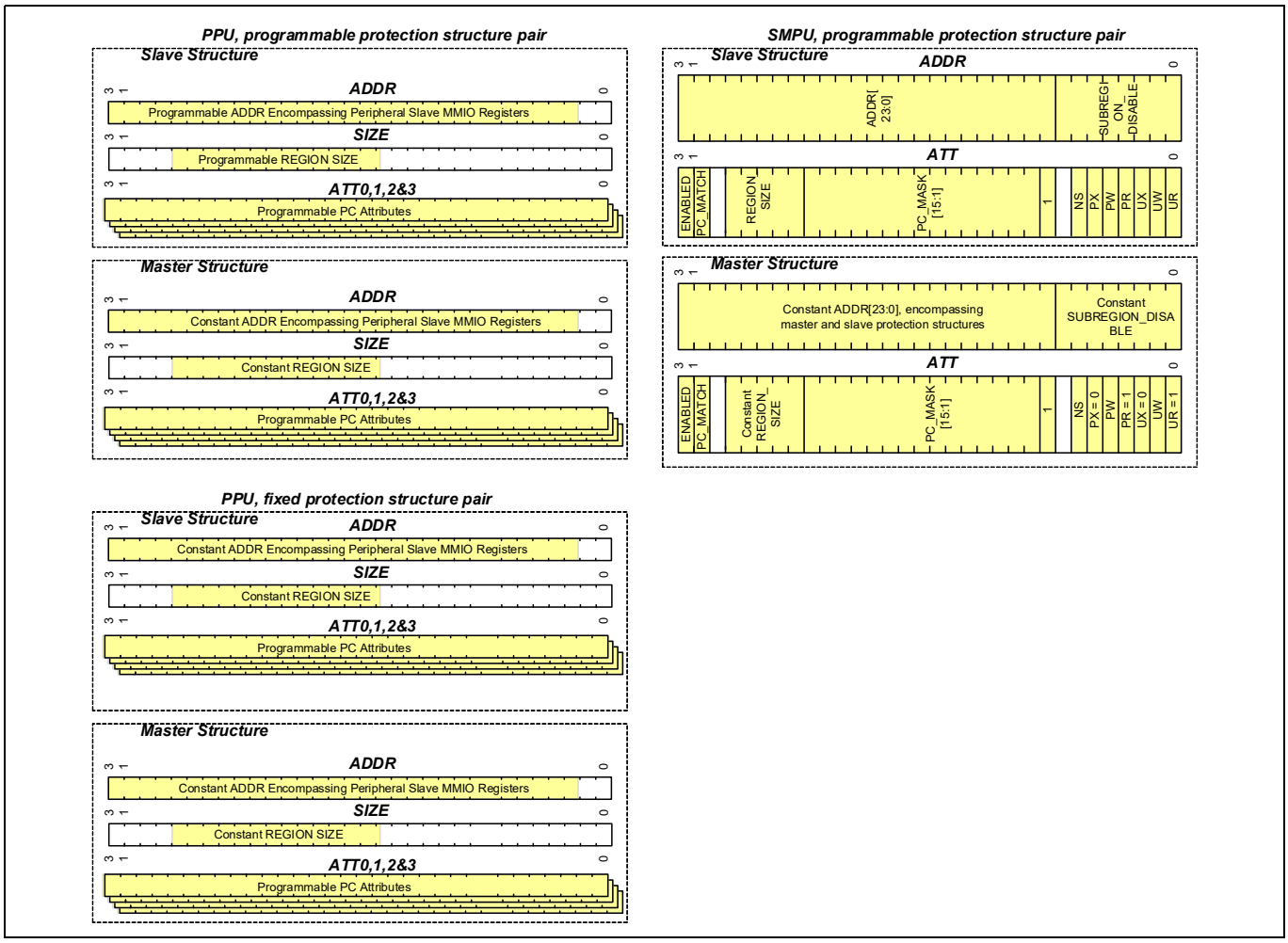


Figure 10-6. PPU and SMPU pairs

Note: *By default, both CPUs (CM0+ and CM4) are in protection context 0 when they come out of reset. In protection context 0, the master is able to access all memory regardless of its protection settings. The master's protection context will need to be changed from protection context 0 to make any protection structure configuration effective. Multiple protection structures may be preconfigured as part of the boot code, which sets up a secure environment at boot time. See the* ***"Boot code"*** *on page 240 for details of these configurations.*

11 DMA controller (DMAC)

The DMAC transfers data to and from memory, peripherals, and registers. These transfers occur independent from the CPU. The DMAC can be configured to perform multiple independent data transfers. All data transfers are managed by a channel. There can be up to 32 channels in the DMAC. The number of channels in the DMAC controller can vary with devices. Refer to the device datasheet for the number of channels supported in the device. A channel has an associated priority; channels are arbitrated according to their priority. The main difference between the DMA (DW) and DMAC relate to their usage. The DMA (DW) is meant as a small data size, transactional DMA, which would typically be used to transfer bytes between peripherals. Using the DMA (DW) for large transaction is expensive on a system due to its relatively low performance. The DMAC is a transitional DMA. It is more efficient than the DMA (DW) and should be used to transfer large amounts of data. The DMAC has dedicated channel logic for all channels. Furthermore, the DMAC also includes a 12-byte FIFO for temporary data storage. This results in increased memory bandwidth for the DMAC. The DMAC also supports an additional transfer mode called memory copy and scatter transfer.

11.1 Features

The DMAC controller has the following features:

- Supports up to 32 channels per DMAC controller; see the device datasheet for details
- Four levels of priority for each channel
- Descriptors are defined in memory and referenced to the respective channels
- Supports single, 1D, 2D, Memory-copy, or Scatter-transfer modes for a descriptor
- Supports transfer of up to 2^32 data elements per descriptor
- Configurable source and destination address increments
- Supports 8-bit, 16-bit, and 32-bit data widths at both source and destination
- Configurable input trigger behavior for each descriptor
- Configurable interrupt generation in each descriptor
- Configurable output trigger generation for each descriptor
- Descriptors can be chained to other descriptors in memory

11.2 Architecture

A data transfer is initiated by an input trigger. This trigger may originate from the source peripheral of the transfer, the destination peripheral of the transfer, CPU software, or from another peripheral. Triggers provide Active/Sleep functionality and are not available in Deep Sleep power mode.

The data transfer details are specified by a descriptor. Among other things, this descriptor specifies:

- The source and destination address locations and the size of the transfer
- The actions of a channel; for example, generation of output triggers and interrupts. See the **“Interrupts”** on page 89 for more details
- Data transfer types can be single, 1D, or 2D as defined in the descriptor structure. These types define the address sequences generated for source and destination. 1D and 2D transfers are used for “scatter gather” and other useful transfer operations.

11.3 Channels

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

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

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

The data transfer associated with a trigger is made up of one or more ‘atomic transfers’ or ‘single transfers’; see **Table 11-1** for a better understanding. A single trigger could be configured to transfer multiple “single transfers”.

Channel registers:

- CH_CTL. This register provides generic channel control information
- CH_CURR_PTR. This register provides the memory location address where the current descriptor is located. Software needs to initialize this register. Hardware sets this register to the current descriptor’s next descriptor pointer, when advancing from the current descriptor to the next descriptor in a chained descriptor list. When this field is ‘0’, there is no valid descriptor.
- CH_IDX. This register provides the current X and Y indices of the channel into the current descriptor. Software needs to initialize this register. Hardware sets the X and Y indices to ‘0’, when advancing from the current descriptor to the next descriptor in a descriptor list.
- CH_SRC. This register provides the current address of the source location.
- CH_DST. This register provides the current address of the destination location.
- CH_DESCR_STATUS. This register provides the validity of other CH_DESCR registers.
- CH_DESCR_CTL. This register contains a copy of DESCR_CTL of the currently active descriptor.
- CH_DESCR_SRC. This register contains a copy of DESCR_SRC of the currently active descriptor.
- CH_DESCR_DST. This register contains a copy of DESCR_DST of the currently active descriptor.
- CH_DESCR_X_INCR. This register contains a copy of DESCR_X_INCR of the currently active descriptor.
- CH_DESCR_X_SIZE. This register contains a copy of DESCR_X_SIZE of the currently active descriptor.
- CH_DESCR_Y_INCR. This register contains a copy of DESCR_Y_INCR of the currently active descriptor.
- CH_DESCR_Y_SIZE. This register contains a copy of DESCR_Y_SIZE of the currently active descriptor.
- CH_DESCR_NEXT_PTR. This register contains a copy of DESCR_NEXT_PTR of the currently active descriptor.
- INTR. This register contains the interrupts that are currently activated for this channel
- INTR_SET. Writing ‘1’ to the appropriate bit in this register sets the corresponding INTR field to ‘1’.
- INTR_MASK. Mask for corresponding field in the INTR register.
- INTR_MASKED. Logical AND of corresponding INTR and INTR_MASK fields.

Note that channel state is retained in Deep Sleep power mode.

A channel has three access control attributes that are used by the shared memory protection units (SMPUs) and peripheral protection units (PPUs) for access control. These fields are typically inherited from the master that modified the channel’s control register.

- The Privileged Mode (CHi_CTL.P) attribute can be set to privileged or user
- The Non-secure (CHi_CTL.NS) attribute can be set to secure or non-secure
- The Protection context (CHi.CTL.PC) attribute can be set to one of the protection contexts

11.3.1 Channel interrupts

Every DMAC channel has an interrupt line associated with it. The INTR_TYPE parameter in the descriptor determines the event that will trigger the interrupt for the channel. In addition, each DMAC channel has INTR, INTR_SET, INTR_MASK, and INTR_MASKED registers to control their respective interrupt lines. INTR_MASK can be used to mask the interrupt from the DMA channel. The INTR and INTR_SET can be used to clear and set the interrupt, respectively, for debug purposes.

The DMAC_CHx_INTR registers allow the user to set different interrupt causes. The following causes are available to configure:

- bit 0: Interrupt based on transfer completion configured based on INTR_TYPE field in the descriptor
- bit 1: Source bus error
- bit 2: Destination bus error
- bit 3: Misaligned source address
- bit 4: Misaligned destination address
- bit 5: Current descriptor pointer is null
- bit 6: Active channel is in disabled state
- bit 7: Descriptor bus error

For error-related interrupt causes (every state other than Completion), the channel is disabled (hardware sets CH_CTL.ENABLED to '0').

11.4 Descriptors

The data transfer between a source and destination in a channel is configured using a descriptor. Descriptors are stored in memory. The descriptor pointer is specified in the DMAC channel registers. The DMAC controller does not modify the descriptor and treats it as read only. A descriptor is a set of up to six 32-bit registers that contain the configuration for the transfer in the associated channel.

There are three types of descriptors: Single transfer, 1D transfer, and 2D transfer.

Table 11-1. Descriptor types

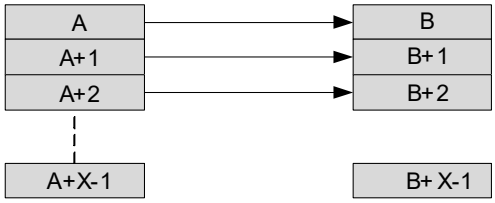
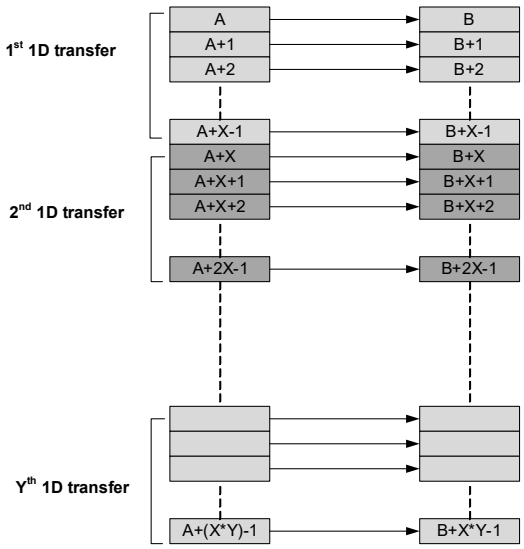
Descriptor type	Description
Single transfer	Transfers a single data element 
1D transfer	Performs a one-dimensional "for loop". This transfer is made up of X number of single transfers 

Table 11-1. Descriptor types (continued)

Descriptor type	Description
2D transfer	Performs a two-dimensional “for loop”. This transfer is made up of Y number of 1D transfers 
Memory copy	This is a special case of 1D transfer; DESCR_X_INCR.SRC_X_INCR and DESCR_X_INCR.DST_X_INCR are implicitly set to 1 and not as part of the descriptor (source and destinations data are consecutive). The descriptor type assumes that both source and destination locations are memory locations. Therefore, 8-bit, 16-bit, and 32-bit transfers can be used (as long as alignment requirements are met), rather than using transfer of a specific type (as specified by SRC_TRANSFER_SIZE and DST_TRANSFER_SIZE). The size of the descriptor is five 32-bit words. DESCR_CTL, DESCR_SRC, DESCR_DST, DESCR_X_SIZE, and DESCR_NEXT_PTR (note that the descriptor size is one word less than the 1D transfer descriptor size). Scatter: This descriptor type is intended to write a set of 32-bit data elements, whose addresses are “scattered” around the address space. The size of the descriptor is four 32-bit words. DESCR_CTL, DESCR_SRC, DESCR_X_SIZE, and DESCR_NEXT_PTR.
Scatter transfer	This descriptor type is intended to write a set of 32-bit data elements, whose addresses are “scattered” around the address space. The size of the descriptor is four 32-bit words. DESCR_CTL, DESCR_SRC, DESCR_X_SIZE, and DESCR_NEXT_PTR.

- **Single transfer**

The following pseudo code illustrates a single transfer.

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

- **1D transfer**

The following pseudo code illustrates a 1D transfer. Note that the 1D transfer is represented by a loop with each iteration executing a single transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
       DST_ADDR[X_IDX * DST_X_INCR] =
          (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR];
}
```

- **2D transfer**

The following pseudo code illustrates a 2D transfer. Note that the 2D transfer is represented by a loop with each iteration executing an inner loop, which is the 1D transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
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];
    }
}
```

- **Memory copy**

The following pseudo code illustrates a memory copy.

```
// DST_ADDR is a pointer to an object of type uint8_t
// SRC_ADDR is a pointer to an object of type uint8_t
// This transfer type uses 8-bit, 16-bit an 32-bit transfers. The HW ensures that
// alignment requirements are met.
for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
       DST_ADDR[X_IDX] = SRC_ADDR[X_IDX];
}
```

- **Scatter**

The following pseudo code illustrates the scatter transfer

```
// SRC_ADDR is a pointer to an object of type uint32_t
for (X_IDX = 0; X_IDX < X_COUNT; X_IDX += 2) {
           address = SRC_ADDR[X_IDX];
           data      = SRC_ADDR[X_IDX + 1];
           *address = data;
}
```

The parameters in the descriptor help configure the different aspects of the transfers explained.

Figure 11-2 shows the structure of a descriptor.

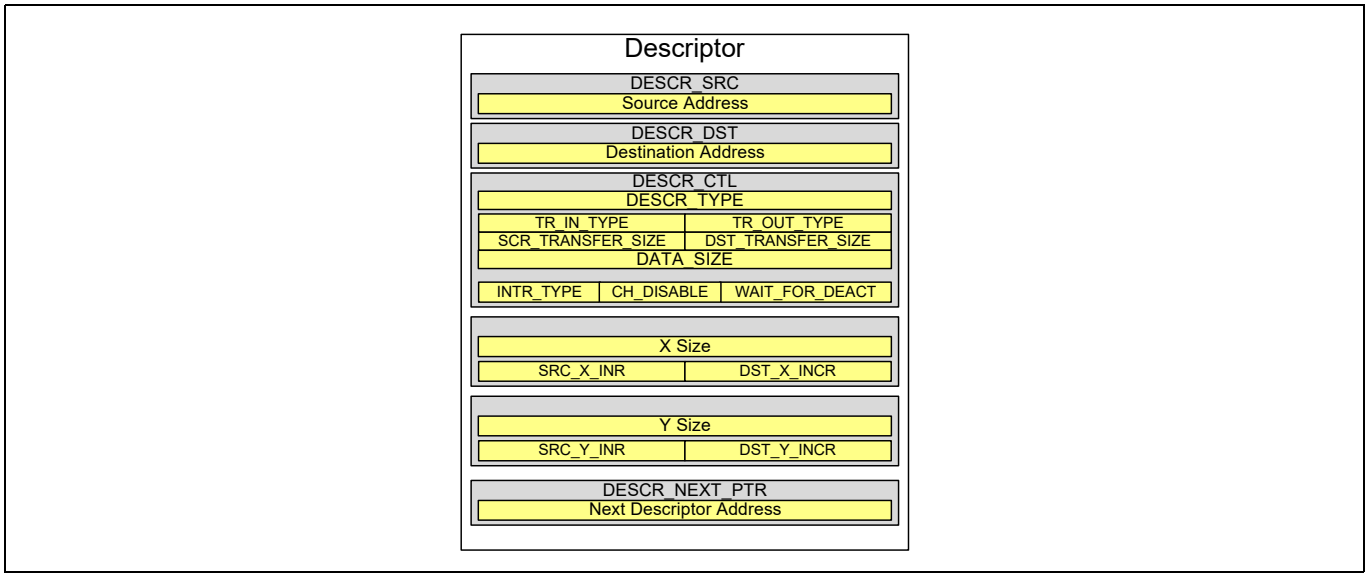


Figure 11-1. Descriptor structure

11.4.1 Address configuration

- **Source and destination address:** The source and destination addresses are set in the respective registers in the descriptor. These set the base addresses for the source and destination location for the transfer. If the descriptor is configured to transfer a single element, this field holds the source/destination address of the data element. If the descriptor is configured to transfer multiple elements with source address or destination address or both in an incremental mode, this field will hold the address of the first element that is transferred.
- **DESCR_TYPE:** This field configures whether the descriptor has a single, 1D, 2D type, memory copy, or scatter transfer.
- **Source data prefetch, DATA_PREFETCH:** When enabled, source data transfers are initiated as soon as the channel is enabled; the current descriptor pointer is not “0” and there is space available in the channel's data FIFO. When the input trigger is activated, the trigger can initiate destination data transfers with data that is already in the channel's data FIFO. This effectively shortens the initial delay of the data transfer.

Note: Data prefetch should be used with care to ensure that data coherency is guaranteed and that prefetches do not cause undesired side effects.

- **Trigger input type, TR_IN_TYPE:** This field determines how the DMAC engine responds to input trigger signal. This field can be configured for one of the following modes:
  - Type 0: A trigger results in execution of a single transfer. Regardless of the DESCR_TYPE setting, a trigger input will trigger only a single element transfer. For example, in a 1D transfer, the DMAC will transfer only one data element in every trigger.
  - Type 1: A trigger results in the execution of a single 1D transfer. If the DESCR_TYPE was set to single transfer, the trigger signal will trigger the single transfer specified by the descriptor. For a DESCR_TYPE set to 1D transfer, the trigger signal will trigger the entire 1D transfer configured in the descriptor. For a 2D transfer, the trigger signal will trigger only a single iteration of the Y loop transfer. If the descriptor type is “memory copy”, the trigger results in the execution of a memory copy transfer.
  - Type 2: A trigger results in execution of the current descriptor. Regardless of DESCR_TYPE, the trigger will execute the entire descriptor. If there was a next descriptor configured for the current descriptor, this trigger setting will not automatically trigger the next descriptor.

  - Type 3: A trigger results in execution of the current descriptor and also triggering the next descriptor. The execution of the next descriptor from this point will be determined by the TR_IN_TYPE setting of the next descriptor.
- **Trigger out type, TR_OUT_TYPE:** This field determines what completion event will generate the output trigger signal. This field can be configured to one of the following modes:
  - Type 0: Generates a trigger output for completion of every single element transfer
  - Type 1: Generates a trigger output for completion of a 1D transfer. If the descriptor type is “memory copy”, the output trigger is generated after the execution of a memory copy transfer. If the descriptor type is “scatter”, the output trigger is generated after the execution of a scatter transfer.
  - Type 2: Generates a trigger output for completion of the current descriptor. This trigger output is generated independent of the state of the DESCR_NEXT_PTR.
  - Type 3: Generates a trigger output on completion of the current descriptor, when the current descriptor is the last descriptor in the descriptor chain. This means a trigger is generated when the descriptor execution is complete and the DESCR_NEXT_PTR is ‘0’.
- **Interrupt Type, INTR_TYPE:** This field determines which completion event will generate the output interrupt signal. This field can be configured to one of the following modes:
  - Type 0: Generates an interrupt output for completion of every single element transfer
  - Type 1: Generates an interrupt output for completion of a 1D transfer. If the descriptor type is “memory copy”, the interrupt is generated after the execution of a memory copy transfer. If the descriptor type is “scatter” the interrupt is generated after the execution of a scatter transfer.
  - Type 2: Generates an interrupt output for completion of the current descriptor. This interrupt output is generated independent of the state of the DESCR_NEXT_PTR.
  - Type 3: Generates an interrupt output on completion of the current descriptor, when the current descriptor is the last descriptor in the descriptor chain. This means an interrupt is generated when the descriptor execution is complete and the DESCR_NEXT_PTR is ‘0’.
- **WAIT_FOR_DEACT:** When the DMAC transfer based on the TR_IN_TYPE is completed, the data transfer engine checks the state of trigger deactivation. The data transfer on the second trigger is initiated only after deactivation of the first. The WAIT_FOR_DEACT parameter will determine when the trigger signal is considered deactivated. The first DMAC transfer is activated when the trigger is activated, but the transfer is not considered complete until the trigger is deactivated. This field is used to synchronize the controller’s data transfers with the agent that generated the trigger. This field has four settings:
  - 0 – Pulse Trigger: Do not wait for deactivation
  - 1 – Level-sensitive waits four SYSCLK cycles: The DMAC trigger is deactivated after the level trigger signal is detected for four cycles.
  - 2 – Level-sensitive waits 16 SYSCLK cycles: The DMAC transfer is initiated if the input trigger is seen to be activated after 16 clock cycles
  - 3 – Pulse trigger waits indefinitely for deactivation. The DMAC transfer is initiated after the trigger signal deactivates. The next transfer is initiated only if the trigger goes low and then high again.
- **X Size:** This field determines the number of single element transfers present in the X loop (inner loop). This field is valid when the DESCR_TYPE is set to 1D or 2D transfer. For the “memory copy” descriptor type, (X_COUNT + 1) is the number of transferred Bytes. For the “scatter” descriptor type, ceiling (X_COUNT/2) is the number of (address, write data) initialization pairs processed.
- **Source Address Increment (X loop) (SCR_X_INCR):** This field configures the index by which the source address is to be incremented for every iteration in an X loop. The field is expressed in multiples of SRC_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the source address does not need to be incremented, you can set this parameter to ‘0’.
- **Destination Address Increment (X loop) (DST_X_INCR):** This field configures the index by which the destination address is to be incremented, for every iteration in an X loop. The field is expressed in multiples of DST_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the destination address does not need to be incremented, you can set this parameter to ‘0’.

- **Y Size:** This field determines the number of 1D transfers present in the Y loop (outer loop). This field is valid when the DESCR_TYPE is set to 2D transfer.
- **Source Address Increment (Y loop) (SCR_Y_INCR):** This field configures the index by which the source address is to be incremented, for every iteration in a Y loop. The field is expressed in multiples of SRC_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the source address does not need to be incremented, you can set this parameter to '0'.
- **Destination Address Increment (X loop) (DST_Y_INCR):** This field configures the index by which the destination address is to be incremented, for every iteration in a Y loop. The field is expressed in multiples of DST_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the destination address does not need to be incremented, you can set this parameter to '0'.
- **Channel Disable (CH_DISABLE):** This field specifies whether the channel is disabled or not after completion of the current descriptor (independent of the value of the DESCR_NEXT_PTR). A disabled channel will ignore its input triggers.

11.4.2 Transfer size

The word width for a transfer can be configured using the transfer/data size parameter in the descriptor. The settings are diversified into source transfer size, destination transfer size, and data size. The data size parameter (DATA_SIZE) sets the width of the bus for the transfer. The source and destination transfer sizes set by SCR_TRANSFER_SIZE and DST_TRANSFER_SIZE can have a value either the DATA_SIZE or 32 bit. DATA_SIZE can have a 32-bit, 16-bit, or 8-bit setting.

The source and destination transfer size for the DMAC must match the addressable width of the source and destination, regardless of the width of data that must be moved. The DATA_SIZE parameter will correspond to the width of the actual data. For example, if a 16-bit PWM is used as a destination for DMAC data, the DST_TRANSFER_SIZE must be set to 32-bit to match the width of the PWM register, because the peripheral register width for the TCPWM block (and most EZ-USB™ FX10 MCU peripherals) is always 32-bit wide. However, in this example the DATA_SIZE for the destination may still be set to 16-bit because the 16-bit PWM only uses two bytes of data. SRAM and Flash are 8-bit, 16-bit, or 32-bit addressable and can use any source and destination transfer sizes to match the needs of the application.

Table 11-2 summarizes the possible combinations of the transfer size settings and its description.

Table 11-2. Transfer size settings

DATA_SIZE	SCR_TRANSFER_SIZE	DST_TRANSFER_SIZE	Description
8-bit	8-bit	8-bit	No data manipulation
8-bit	32-bit	8-bit	Higher 24 bits from the source dropped
8-bit	8-bit	32-bit	Higher 24 bits zero padded at destination
8-bit	32-bit	32-bit	Higher 24 bits from the source dropped and higher 24 bits zero padded at destination
16-bit	16-bit	16-bit	No data manipulation
16-bit	32-bit	16-bit	Higher 16 bits from the source dropped
16-bit	16-bit	32-bit	Higher 16 bits zero padded at destination
16-bit	32-bit	32-bit	Higher 16 bits from the source dropped and higher 16-bit zero padded at destination
32-bit	32-bit	32-bit	No data manipulation

11.4.3 Descriptor chaining

Descriptors can be chained together. The DESCR_NEXT_PTR field contains a pointer to the next descriptor in the chain. A channel executes the next descriptor in the chain when it completes executing the current descriptor. The last descriptor in the chain has DESCR_NEXT_PTR set to '0' (NULL pointer). A descriptor chain is also referred to as a descriptor list. It is possible to have a circular list; in a circular list, the execution continues indefinitely until there is an error or the channel or the controller is disabled by user code.

11.5 DMAC controller

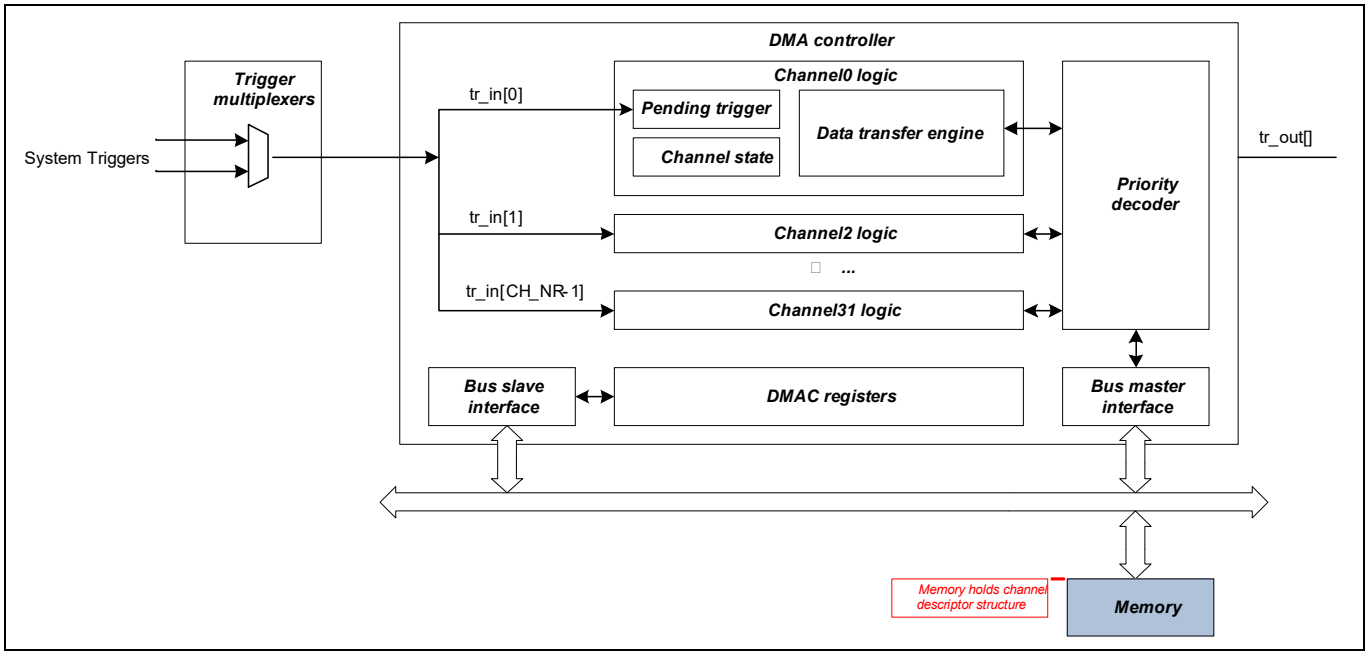


Figure 11-2. DMAC controller overview

11.5.1 Trigger selection

Trigger signals can be generated from different sections of the chips. A trigger multiplexer block helps route these trigger signals to the destination. The DMAC is one such destination of triggers. The trigger multiplexer block is outside the DMAC block and is discussed in the **"Trigger multiplexer block"** on page 344.

11.5.2 Channel logic

The channel logic keeps track of pending triggers for each channel and initiates the transfer corresponding to the active descriptor, based on availability of the bus and arbitration by priority decoder.

The **priority decoder** determines the highest priority pending channel.

Master I/F is an AHB-Lite bus master that allows the DMAC controller to initiate AHB-Lite data transfers to the source and destination locations as well as to read the descriptor from memory.

Slave I/F is an AHB-Lite bus slave that allows the main CPU to access DMAC controller control/status registers.

11.5.3 Output triggers

Each channel has an output trigger. This trigger is high for two slow clock cycles. The trigger is generated on the completion of a data transfer. At the system level, these output triggers can be connected to the trigger multiplexer component. This connection allows a DMAC controller output trigger to be connected to a DMAC controller input trigger. In other words, the completion of a transfer in one channel can activate another channel or even reactivate the same channel.

DMAC output triggers routing to other DMAC channels or other peripheral trigger inputs is achieved using the trigger multiplexer. Refer to the **"Trigger multiplexer block"** on page 344.

12 Data wire controller (DW)

The DMA transfers data to and from memory, peripherals, and registers. These transfers occur independent from the CPU. The DMA can be configured to perform multiple independent data transfers. All data transfers are managed by a channel. There can be up to 32 channels in the DMA. The number of channels in the DMA controller can vary with devices. Refer to the device datasheet for the number of channels supported in the device. A channel has an associated priority; channels are arbitrated according to their priority.

12.1 Features

The DMA controller has the following features:

- Supports up to 29 channels per DMA controller; see the device datasheet for details
- Supports multiple DMA controller instances in a device
- Four levels of priority for each channel
- Descriptors are defined in memory and referenced to the respective channels
- Supports single, 1D, or 2D transfer modes for a descriptor
- Supports transfer of up to 65536 data elements per descriptor
- Configurable source and destination address increments
- Supports 8-bit, 16-bit, and 32-bit data widths at both source and destination
- Configurable input trigger behavior for each descriptor
- Configurable interrupt generation in each descriptor
- Configurable output trigger generation for each descriptor
- Descriptors can be chained to other descriptors in memory

12.2 Architecture

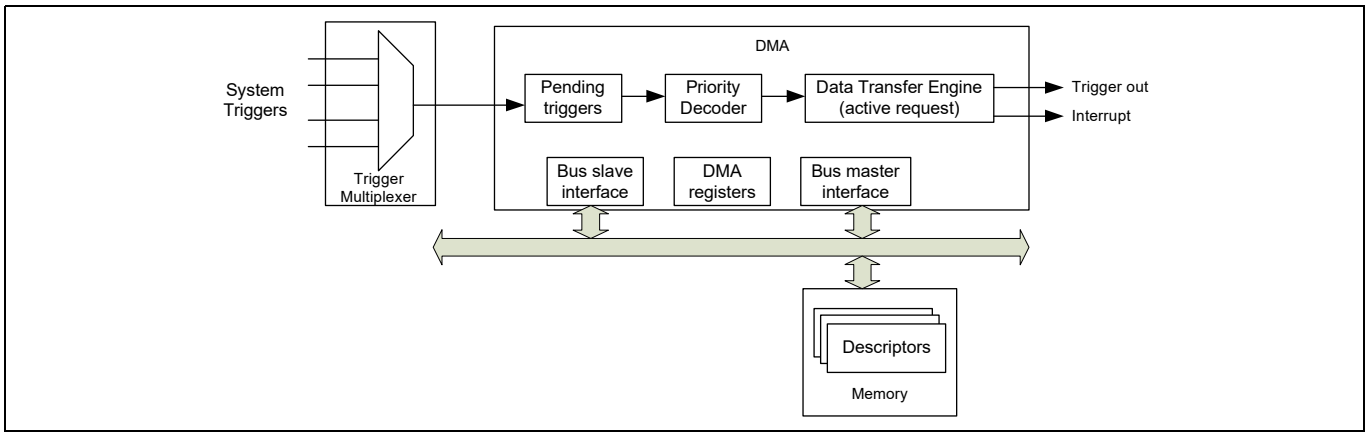


Figure 12-1. DMA controller

A data transfer is initiated by an input trigger. This trigger may originate from the source peripheral of the transfer, the destination peripheral of the transfer, CPU software, or from another peripheral. Triggers provide Active/Sleep functionality and are not available in Deep Sleep power mode.

The data transfer details are specified by a descriptor. Among other things, this descriptor specifies:

- The source and destination address locations and the size of the transfer
- The actions of a channel; for example, generation of output triggers and interrupts. See the **“Interrupts”** on page 89 for more details.
- Data transfer types can be single, 1D, or 2D as defined in the descriptor structure. These types define the address sequences generated for source and destination. 1D and 2D transfers are used for “scatter gather” and other useful transfer operations.

12.3 Channels

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

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

- The highest priority group with pending channels is identified first
- Within this priority group, round-robin arbitration is applied

Channel state: At any given time, one channel actively performs a data transfer. This channel is called the active channel. A channel can be in one of four channel states. The active channel in a DW controller can be determined by reading the DWx_STATUS[ACTIVE] and DWx.STATUS[CH_IDX].

Pending state of a channel is determined by reading the DW_CH_STRUCT_CH_STATUS[PENDING] associated with that channel. If a channel is enabled and is not in the Pending or Active state, then it is considered blocked.

Table 12-1. Channel states

Channel state	Description
Disabled	The channel is disabled by setting CHi_CTL.ENABLED to '0'. The channel trigger is ignored in this state.
Blocked	The channel is enabled and is waiting for a trigger to initiate a data transfer.
Pending	The channel is enabled and has received an active trigger. In this state, the channel is ready to initiate a data transfer but waiting for it to be scheduled.
Active	The channel is enabled, has received an active trigger and has been scheduled. It is actively performing data transfers. If there are multiple channels pending, the highest priority pending channel is scheduled.

The data transfer associated with a trigger is made up of one or more 'atomic transfers' or 'single transfers'; see **Table 12-2** for a better understanding. A single trigger could be configured to transfer multiple "single transfers".

A channel can be marked preemptable (CHi_CTL.PREEMPTABLE). If preemptable, and there is a higher priority pending channel, then that higher priority channel can preempt the current channel between single transfers. If a channel is preempted, the existing single transfer is completed; the current channel goes to pending state and the higher priority channel is serviced. On completion of the higher priority channel's transfer, the pending channel is resumed. Note that preemption has an impact on the data transfer rates of the channel being preempted. Refer to **"DMA performance"** on page 151 for these performance implications.

A channel has two access control attributes that are used by the shared memory protection units (SMPUs) and peripheral protection units (PPUs) for access control. These fields are typically inherited from the master that modified the channel's control register.

- The Privileged Mode (CHi_CTL.P) attribute can be set to privileged or user
- The Non-secure (CHi_CTL.NS) attribute can be set to secure or non-secure

A descriptor associated with each channel describes the data transfer. The descriptor is stored in memory and CHi_CURR_PTR provides the descriptor address associated with channel "i" and Chi_IDX provides the current X and Y indices into the descriptor.

A channel's descriptor state is encoded as part of the channel's register state. The following registers provide a channel's descriptor state:

- CH_CTL. This register provides generic channel control information.
- CH_CURR_PTR. This register provides the address of the memory location where the current descriptor is located. The user firmware must initialize this register. If the descriptors are chained, the DMA hardware automatically sets this register to the next descriptor pointer.
- CH_IDX. This register provides the current X and Y indices of the channel into the current descriptor. User firmware must initialize this register. DMA hardware sets the X and Y indices to 0, when advancing from the current descriptor to the next descriptor in a descriptor list.

Note that channel state is retained in Deep Sleep power mode.

12.3.1 Channel interrupts

Every DMA channel has an interrupt line associated with it. The INTR_TYPE parameter in the descriptor determines the event that will trigger the interrupt for the channel. In addition each DMA channel has INTR, INTR_SET, INTR_MASK, and INTR_MASKED registers to control their respective interrupt lines. INTR_MASK can be used to mask the interrupt from the DMA channel. The INTR and INTR_SET can be used to clear and set the interrupt, respectively, for debug purposes.

The DW_CH_STRUCT_CH_STATUS[INTR_CAUSE] field provides the user a means to determine the cause of the interrupt being generated. The following are different values for this register:

- 0: No interrupt generated
- 1: Interrupt based on transfer completion configured based on INTR_TYPE field in the descriptor
- 2: Source bus error
- 3: Destination bus error
- 4: Misaligned source address
- 5: Misaligned destination address
- 6: Current descriptor pointer is null
- 7: Active channel is in disabled state
- 8: Descriptor bus error
- 9–15: Not used

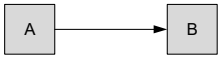
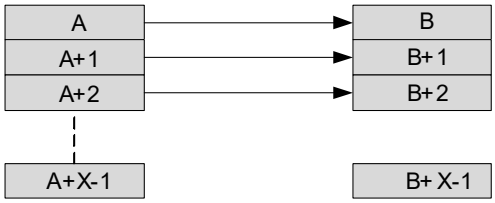
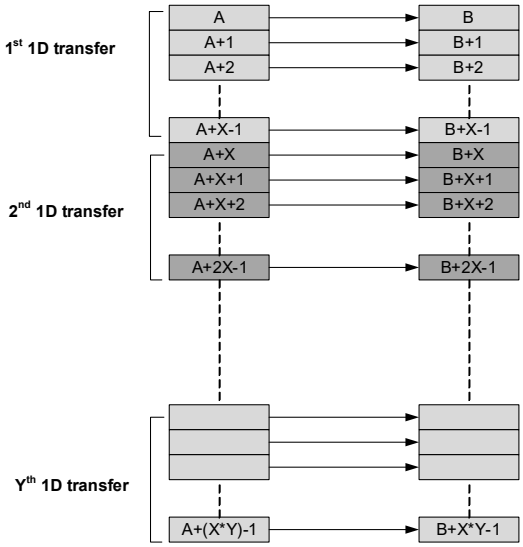
For error related interrupt causes (INTR_CAUSE is 2, 3,..., 8), the channel is disabled (hardware sets CH_CTL.ENABLED to '0').

The bus errors are typically caused by incompatible accesses to the addresses in question. This may be due to those addresses being protected or having read or write restrictions. Source and destination bus errors can also occur due to mismatch in data sizes (see **"Transfer size"** on page 148).

12.4 Descriptors

The data transfer between a source and destination in a channel is configured using a descriptor. Descriptors are stored in memory. The descriptor pointer is specified in the DMA channel registers. The DMA controller does not modify the descriptor and treats it as read only. A descriptor is a set of up to six 32-bit registers that contain the configuration for the transfer in the associated channel. There are three types of descriptors.

Table 12-2. Descriptor types

Descriptor type	Description
Single transfer	Transfers a single data element. 
1D transfer	Performs a one-dimensional “for loop”. This transfer is made up of X number of single transfers. 
2D transfer	Performs a two-dimensional “for loop”. This transfer is made up of Y number of 1D transfers. 
CRC transfer	This performs a one-dimensional “for loop” similar to the 1D transfer. However, the source data is not transferred to a destination. Instead, a CRC is calculated over the source data. The CRC configuration is provided through a set of registers that is shared by all DMA channels. The assumption is that the DMA channels use the CRC functionality mutually exclusively in time. These registers are: CRC_CTL, CRC_DATA_CTL, CRC_POL_CTL, CRC_LFSR_CTL, CRC_REM_CTL, and CRC_REM_RESULT. Note that the CRC configuration is the same as the Crypto CRC configuration.

- **Single transfer**

The following pseudo code illustrates a single transfer.

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

- **1D transfer**

The following pseudo code illustrates a 1D transfer. Note that the 1D transfer is represented by a loop with each iteration executing a single transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
       DST_ADDR[X_IDX * DST_X_INCR] =
          (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR];
}
```

- **2D transfer**

The following pseudo code illustrates a 2D transfer. Note that the 2D transfer is represented by a loop with each iteration executing an inner loop, which is the 1D transfer.

```
// DST_ADDR is a pointer to an object of type defined by DST_TRANSFER_SIZE
// SRC_ADDR is a pointer to an object of type defined by SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
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];
    }
}
```

- **CRC transfer**

The following psuedo code illustrates CRC transfer.

```
// 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 SRC_TRANSFER_SIZE
// t_DATA_SIZE is the type associated with the DATA_SIZE
CRC_STATE = CRC_LFSR_CTL;
for (X_IDX = 0; X_IDX <= X_COUNT; X_IDX++) {
   Update_CRC (CRC_STATE, (t_DATA_SIZE) SRC_ADDR[X_IDX * SRC_X_INCR];
}
DST_ADDR = CRC_STATE;
```

The parameters in the descriptor help configure the different aspects of the transfers explained.

Figure 12-2 shows the structure of a descriptor.

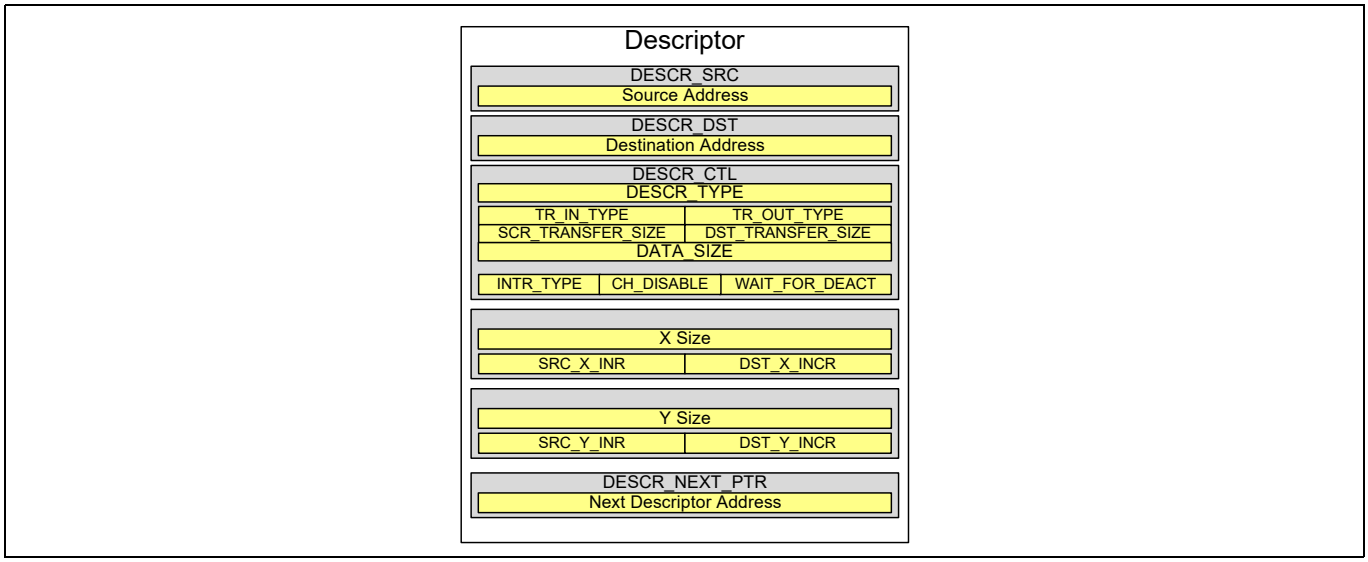


Figure 12-2. Descriptor structure

12.4.1 Address configuration

- **Source and Destination Address:** The source and destination addresses are set in the respective registers in the descriptor. These set the base addresses for the source and destination location for the transfer. In case the descriptor is configured to transfer a single element, this field holds the source/destination address of the data element. If the descriptor is configured to transfer multiple elements with source address or destination address or both in an incremental mode, this field will hold the address of the first element that is transferred.
- **DESCR_TYPE:** This field configures whether the descriptor has a single, 1D, or 2D type.
- **Trigger input type, TR_IN_TYPE:** This field determines how the DMA engine responds to input trigger signal. This field can be configured for one of the following modes:
  - Type 0: A trigger results in execution of a single transfer. Regardless of the DESCR_TYPE setting, a trigger input will trigger only a single element transfer. For example, in a 1D transfer, the DMA will transfer only one data element in every trigger.
  - Type 1: A trigger results in the execution of a single 1D transfer. If the DESCR_TYPE was set to single transfer, the trigger signal will trigger the single transfer specified by the descriptor. For a DESCR_TYPE set to 1D transfer, the trigger signal will trigger the entire 1D transfer configured in the descriptor. For a 2D transfer, the trigger signal will trigger only a single iteration of the Y loop transfer.
  - Type 2: A trigger results in execution of the current descriptor. Regardless of DESCR_TYPE, the trigger will execute the entire descriptor. If there was a next descriptor configured for the current descriptor, this trigger setting will not automatically trigger the next descriptor.
  - Type 3: A trigger results in execution of the current descriptor and also triggering the next descriptor. The execution of the next descriptor from this point will be determined by the TR_IN_TYPE setting of the next descriptor.

- **Trigger out type, TR_OUT_TYPE:** This field determines what completion event will generate the output trigger signal. This field can be configured to one of the following modes:
  - Type 0: Generates a trigger output for completion of every single element transfer
  - Type 1: Generates a trigger output for completion of a 1D transfer
  - Type 2: Generates a trigger output for completion of the current descriptor. This trigger output is generated independent of the state of the DESCR_NEXT_PTR.
  - Type 3: Generates a trigger output on completion of the current descriptor, when the current descriptor is the last descriptor in the descriptor chain. This means a trigger is generated when the descriptor execution is complete and the DESCR_NEXT_PTR is '0'.
- **Interrupt Type, INTR_TYPE:** This field determines which completion event will generate the output interrupt signal. This field can be configured to one of the following modes:
  - Type 0: Generates an interrupt output for completion of every single element transfer
  - Type 1: Generates an interrupt output for completion of a 1-D transfer
  - Type 2: Generates an interrupt output for completion of the current descriptor. This interrupt output is generated independent of the state of the DESCR_NEXT_PTR.
  - Type 3: Generates an interrupt output on completion of the current descriptor, when the current descriptor is the last descriptor in the descriptor chain. This means an interrupt is generated when the descriptor execution is complete and the DESCR_NEXT_PTR is '0'.
- **WAIT_FOR_DEACT:** When the DMA transfer based on the TR_IN_TYPE is completed, the data transfer engine checks the state of trigger deactivation. The data transfer on the second trigger is initiated only after deactivation of the first. The WAIT_FOR_DEACT parameter will determine when the trigger signal is considered deactivated. The first DMA transfer is activated when the trigger is activated, but the transfer is not considered complete until the trigger is deactivated. This field is used to synchronize the controller's data transfers with the agent that generated the trigger. This field has four settings:
  - 0 – Pulse Trigger: Do not wait for deactivation. When a trigger is detected, the transfer is initiated. After completing the transfer, if the trigger is still active then it is considered as another trigger and the subsequent transfer is initiated immediately.
  - 1 – Level-sensitive waits four slow clock cycles after the transfer to consider as a deactivation. When a trigger is detected, the transfer is initiated. After completing the transfer, if the trigger is still active then it is considered as another trigger after waiting for four cycles. Then, a subsequent transfer is initiated. The transfer corresponding to the trigger is considered complete only at the end of the four additional cycles. Even trigger output events will be affected based on this delay. This parameter adds a four-cycle delay in each trigger transaction and hence affects throughput.
  - 2 – Level-sensitive waits 16 slow clock cycles after the transfer to consider as a deactivation. When a trigger is detected, the transfer is initiated. After completing the transfer, if the trigger is still active then it is considered as another trigger after waiting for 16 cycles. Then, a subsequent transfer is initiated. The transfer corresponding to the trigger is considered complete only at the end of the 16 additional cycles. Even trigger output events will be affected based on this delay. This parameter adds a 16-cycle delay in each trigger transaction and hence affects throughput.
  - 3 – Pulse trigger waits indefinitely for deactivation. The DMA transfer is initiated after the trigger signal deactivates. The next transfer is initiated only if the trigger goes low and then high again. A trigger signal that remains active or does not transition to zero between two transaction will simply stall the DMA channel.

    The WAIT_FOR_DEACT field is used in a system to cater to delayed response of other parts of the system to actions of the DMA. Consider an example of a TX FIFO that has a trigger going to the DMA when its not full. Free space in FIFO will trigger a DMA transfer to the FIFO, which in turn will deactivate the trigger. However, there can be a delay in this deactivation by the agent, which may cause the DMA to have initiated another transfer that can cause a FIFO overflow. This can be avoided by using the four or 16 clock cycle delays.
- **X Count:** This field determines the number of single element transfers present in the X loop (inner loop). This field is valid when the DESCR_TYPE is set to 1D or 2D transfer.

- **Source Address Increment (X loop) (SCR_X_INCR):** This field configures the index by which the source address is to be incremented for every iteration in an X loop. The field is expressed in multiples of SRC_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the source address does not need to be incremented, you can set this parameter to zero.
- **Destination Address Increment (X loop) (DST_X_INCR):** This field configures the index by which the destination address is to be incremented, for every iteration in an X loop. The field is expressed in multiples of DST_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the destination address does not need to be incremented, you can set this parameter to zero.
- **Y Count:** This field determines the number of 1D transfers present in the Y loop (outer loop). This field is valid when the DESCR_TYPE is set to 2D transfer.
- **Source Address Increment (Y loop) (SCR_Y_INCR):** This field configures the index by which the source address is to be incremented, for every iteration in a Y loop. The field is expressed in multiples of SRC_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the source address does not need to be incremented, you can set this parameter to zero.
- **Destination Address Increment (X loop) (DST_Y_INCR):** This field configures the index by which the destination address is to be incremented, for every iteration in a Y loop. The field is expressed in multiples of DST_TRANSFER_SIZE. This field is a signed number and hence may be decrementing or incrementing. If the destination address does not need to be incremented, you can set this parameter to zero.
- **Channel Disable (CH_DISABLE):** This field specifies whether the channel is disabled or not after completion of the current descriptor (independent of the value of the DESCR_NEXT_PTR). A disabled channel will ignore its input triggers.

12.4.2 Transfer size

The word width for a transfer can be configured using the transfer/data size parameter in the descriptor. The settings are diversified into source transfer size, destination transfer size, and data size. The data size parameter (DATA_SIZE) sets the width of the bus for the transfer. The source and destination transfer sizes set by SCR_TRANSFER_SIZE and DST_TRANSFER_SIZE can have a value either the DATA_SIZE or 32 bit. DATA_SIZE can have a 32-bit, 16-bit, or 8-bit setting.

The source and destination transfer size for the DMA must match the addressable width of the source and destination, regardless of the width of data that must be moved. The DATA_SIZE parameter will correspond 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, because the peripheral register width for the TCPWM block (and most EZ-USB™ FX10 MCU peripherals) is always 32-bit wide. However, in this example the DATA_SIZE for the destination may still be set to 16 bit because the 16-bit PWM only uses two bytes of data. SRAM and Flash are 8-bit, 16-bit, or 32-bit addressable and can use any source and destination transfer sizes to match the needs of the application.

Table 12-3 summarizes the possible combinations of the transfer size settings and its description.

Table 12-3. Transfer size settings

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

12.4.3 Descriptor chaining

Descriptors can be chained together. The DESCR_NEXT_PTR field contains a pointer to the next descriptor in the chain. A channel executes the next descriptor in the chain when it completes executing the current descriptor. The last descriptor in the chain has DESCR_NEXT_PTR set to ‘0’ (NULL pointer). A descriptor chain is also referred to as a descriptor list. It is possible to have a circular list; in a circular list, the execution continues indefinitely until there is an error or the channel or the controller is disabled by user code.

12.5 DMA controller

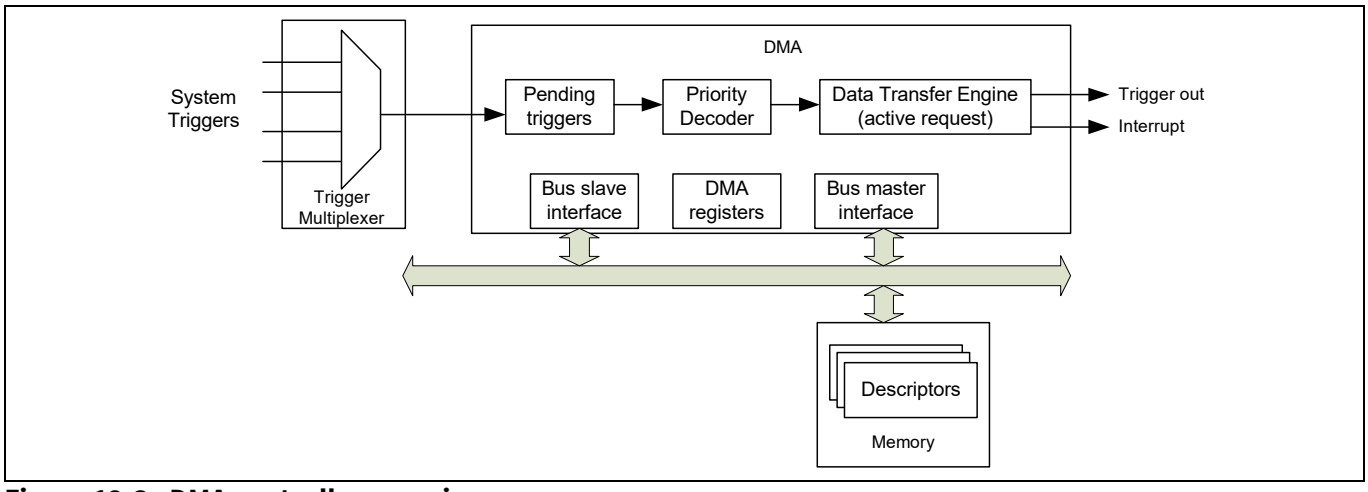


Figure 12-3. DMA controller overview

12.5.1 Trigger selection

Trigger signals can be generated from different sections of the chips. A trigger multiplexer block helps route these trigger signals to the destination. The DMA is one such destination of triggers. The trigger multiplexer block is outside the DMA block and is discussed in the **"Trigger multiplexer block"** on page 344.

12.5.2 Pending triggers

Pending triggers keep track of activated triggers by locally storing them in pending bits. This is essential because multiple channel triggers may be activated simultaneously, whereas only one channel can be served by the data transfer engine at a time. This component enables the use of both level-sensitive (high/'1') and pulse-sensitive (two high/'1' clk_slow cycles) triggers.

- Level-sensitive triggers are associated with a certain state, for example, a FIFO being full. These triggers remain active as long as the state is maintained. It is not required to track pending level-sensitive triggers in the DMA controller because the triggers are maintained outside the controller.
- Pulse-sensitive triggers are associated with a certain event, for example, sample has become available. It is essential to track these triggers in the DMA controller because the trigger pulse may disappear before it is served by the data transfer engine. Pulse triggers should be high/'1' for two clk_slow cycles.

The **priority decoder** determines the highest priority pending channel.

The **data transfer engine** is responsible for the data transfer from a source location to a destination location. When idle, the data transfer engine is ready to accept the highest priority activated channel. It is also responsible for reading the channel descriptor from memory.

Master I/F is an AHB-Lite bus master that allows the DMA controller to initiate AHB-Lite data transfers to the source and destination locations as well as to read the descriptor from memory.

Slave I/F is an AHB-Lite bus slave that allows the main CPU to access DMA controller control/status registers.

12.5.3 Output triggers

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

The DMA output triggers routing to other DMA channels or other peripheral trigger inputs is achieved using the trigger multiplexer. Refer to the **"Trigger multiplexer block"** on page 344.

12.5.4 Status registers

The controller status register (DWx_STATUS) contains the following information.

- ACTIVE – Active channel present, yes/no
- P – Active channel's access control user/privileged
- NS – Active channel's access control secure/non-secure
- CH_IDX – Active channel index if there is an active channel
- PRIO – Active channel priority
- PREEMPTABLE – Active channel pre-emptable or not
- STATE – State of the DW controller state machine. The following states are specified:
  - Default/inactive state
  - Loading descriptor: This state is when the controller has recognized the channel that was triggered and become active and is now loading its respective descriptor.
  - Loading data element: Reading data from source address
  - Storing data element: Writing data into the destination address
  - Update of active channel control information
  - Waiting for input trigger deactivation

12.5.5 DMA performance

The DMA block works on the clk_slow domain and hence all clocks described in this section are in clk_slow units.

Every time a DMA channel is triggered the DMA hardware goes through the following steps:

- Trigger synchronization
- Detection, priority decoding, and making channel pending
- Start state machine and load channel configuration
- Load DMA descriptor
- Load next DMA descriptor pointer
- Moving first element of data from source to destination.

Each of these steps involve multiple cycles for completion. **Table 12-4** shows the number of cycles needed for each step.

Table 12-4. DMA steps and performance

Operation	Cycles
Trigger Synchronization	2
Detection, priority decoding and making channel pending	1
Start state machine and load channel config	3
Load descriptors	4 for single transfer 5 for 1-D transfer 6 for 2-D transfer
Load next pointer	1
Moving data from source to destination	3
Total	14 for single transfer

For subsequent transfers on a preloaded descriptor, cycles are needed only to move the data from source to destination. Therefore, transfers such as 1D and 2D, which are not preempted, incurs all the cycles only for the first transfer; subsequent transfers will cost three cycles.

Based on the configuration of TRIG_IN_TYPE, the trigger synchronization cycles may be incurred for each single element transfer or for each 1D transfer.

The descriptor is four words long for a single transfer type, five words for 1D transfer, and six words for a 2D transfer. Hence, the number of cycles needed to fetch a descriptor will vary based on its type.

Another factor to note is the latency in data or descriptor fetch due to wait states or bus latency.

The DMA performance for different types of transfers can be summarized as follows.

- Single transfer
  - Fourteen cycles per transfer + latency due to wait states or bus latency
- 1D transfer
  - To transfer n data elements
    Number of cycles = 12 + n * 3 + m

    m is the total number of wait states seen by DMA while loading or storing descriptors or data. An additional cycle is required for the first transfer, to load the X-Loop configuration register.
- 2D transfer
  - If the 2 D transfer is transferring n elements then
    Number of cycles = 13 + n * 3 + m

    m is total number of wait states seen by DMA while loading or storing descriptors or data. Two additional cycles are required for the first transfer, to load the X-loop and Y-Loop configuration register.

Note: Descriptors in memory and memory wait states will also affect the descriptor load delay.

- Wait states: Memory accesses can have a wait state associated with them. These wait states need to be accounted into the calculation of throughput.
- Channel arbitration: Some time channels are not immediately made active after reception of trigger. This is due to other active channels in the system. This can lead to multiple cycles being lost before the channel is even made active.
- Preemption: The choice of making a DMA channel preemptable impacts its performance. This is because every time a channel is prempted:
  - The channel is in a pending state for as long as the higher priority channel is running
  - On resumption, the channel descriptor needs to be fetched again. This is additional cycles for every resume. So if there are a large number of high-priority channels, making a low-priority channel preemptible can have adverse effects on its throughput. On the other hand, if there is a low-priority channel that is transferring a large amount of data, then not making it preemptable can starve other high-priority channels for too long.
    Sometimes, users can also distribute channels across multiple DW blocks to avoid conditions of preemption and deal with the contention at the bus arbitration level
- Bus arbitration: Several bus masters access the bus, including the CPU cores and multiple DMA (DW) and DMAC. This makes any access to data movement over the bus subject to arbitration with other masters. Actions such as fetching the descriptor or data can be stalled by arbitration. The arbitration of the bus is based on the arbitration scheme configured in PROT_SMPU_MSx_CTL[PRIO]
- Transfer width: The width of the transfer configured by the Data_size parameter in the descriptor is important in the transfer throughput calculation. 32-bit transfers are four times faster than 8-bit transfers.

13 Cryptographic function block (CRYPTO)

The Cryptographic block (CRYPTO) provides hardware implementation and acceleration of cryptographic functions. Implementation in hardware takes less time and energy than the equivalent firmware implementation. In addition, the block provides True Random Number Generation functionality in silicon, which is not available in firmware.

13.1 Features

- Advanced encryption standard (AES)
- Data encryption and triple data encryption standards (DES, TDES)
- Secure hash algorithm (SHA)
- Cyclic redundancy checking (CRC)
- Pseudo random number generator (PRNG)
- True random number generator (TRNG)
- Vector unit (VU) to support asymmetric key cryptography, such as RSA and ECC.

13.2 Architecture

Figure 13-1 gives an overview of the CRYPTO block.

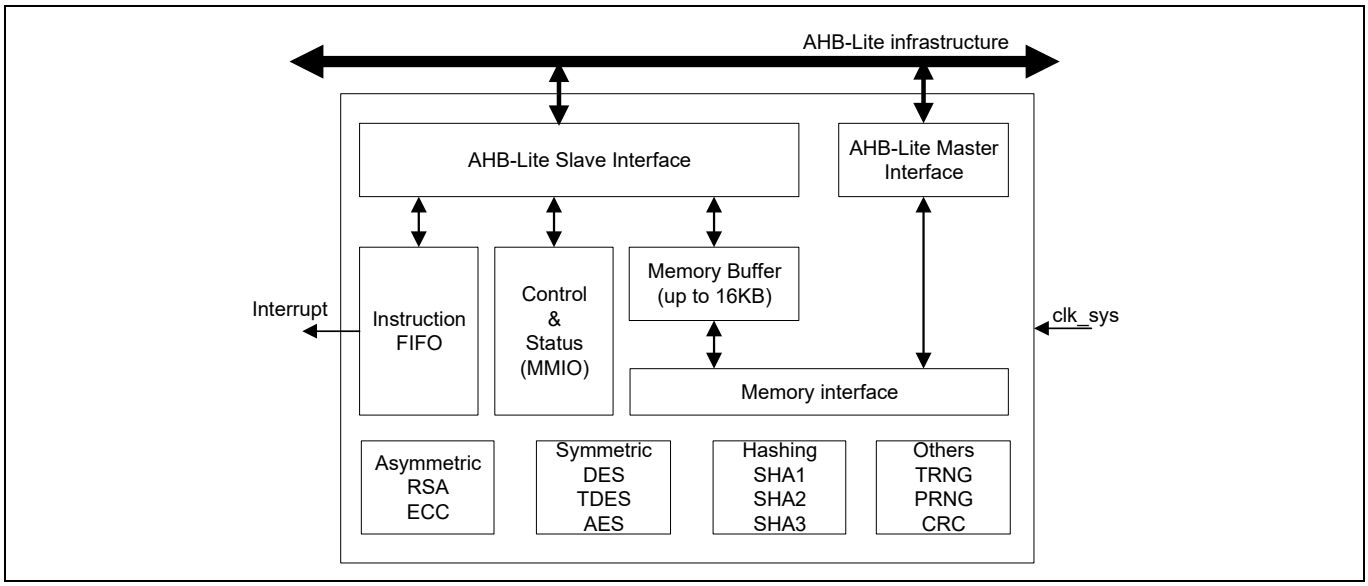


Figure 13-1. CRYPTO block diagram

The CRYPTO block has the following interfaces:

- An AHB-Lite slave interface connects the block to the AHB-Lite infrastructure. This interface supports 8/16/32-bit AHB-Lite transfers. MMIO register accesses are 32-bit only. Memory buffer accesses can be 8/16/32-bit.
- An AHB-Lite master interface connects the block to the AHB-Lite infrastructure. This interface supports 8/16/32-bit AHB-Lite transfers. The interface enables the CRYPTO block to access operation operand data from system memories, such as Flash or SRAM.
- A single interrupt signal “interrupt” is used to signal the completion of an operation
- A clock signal “clk_sys” interface connects to the SRSS

The block has the following components:

- An AHB-Lite slave interface
- An AHB-Lite master interface
- A component that contains MMIO control and status registers
- A memory buffer, for internal operation operand data
- A memory interface that directs operation operand data requests to either the block internal memory buffer or to the AHB-Lite master interface
- An instruction controller component that decodes instructions from an instruction FIFO. The controller issues the instructions to the function specific components
- Cryptographic function specific components

The following sections explain each component in detail.

13.3 Instruction controller

The instruction controller consists of an instruction FIFO, an instruction decoder, and a general-purpose register file.

- The instruction FIFO is software-programmable through MMIO registers that are accessed through the AHB-Lite slave interface. Software writes instructions and instruction operand data to the instruction FIFO. The FIFO consists of eight 32-bit FIFO entries.
- The instruction decoder decodes the instructions (and the associated instruction operand data) from the instruction FIFO. The instruction decoder issues the decoded instruction to a specific functional component. The functional component is responsible for instruction execution.
- The general-purpose register file consists of sixteen 32-bit registers. An instruction specifies the specific use of register-file registers. Registers are used to specify instruction operand data, such as memory locations (for example, for DES, AES, and SHA instructions) or immediate data operations (for example, for vector unit instructions).

13.3.1 Instructions

An instruction consists of a sequence of one, two, or three instruction words. Most instructions are encoded by a single instruction word.

The instruction FIFO can hold up to eight 32-bit instruction words. A CPU writes instruction words to the instruction FIFO (INSTR_FF_WR register) and the CRYPTO block decodes the instruction words to execute the instructions. INSTR_FF_STATUS.USED specifies how many of the eight instruction FIFO entries are used. The instruction FIFO decouples the progress of CPU execution from the CRYPTO block execution: the CPU can write new instruction words to the FIFO, while the block executes previously written instructions.

There are multiple interrupt causes associated with the instruction FIFO and the instruction decoder:

- The INTR.INSTR_FF_OVERFLOW interrupt cause is activated on a write to a full instruction FIFO
- THE INTR.INSTR_FF_LEVEL interrupt cause is activated when the number of used FIFO entries (INSTR_FF_STATUS.USED) is less than a specified number of FIFO entries (INSTR_FF_CTL.LEVEL)
- The INTR.INSTR_OPC_ERROR interrupt cause is activated when an instruction's operation code is not defined
- The INTR.INSTR_CC_ERROR interrupt cause is activated when a vector unit instruction has an undefined condition code

Most instructions perform specific cryptographic functionality. For example, the AES instruction performs and Advanced Encryption Standard (AES) block cipher operation. Some instructions perform more generic functionality; most generic instructions move operand data between different locations. Higher level symmetric cipher and hash functionality is implemented using a combination of cryptographic instructions and generic instructions. Higher level asymmetric cipher functionality is implemented using a set of vector unit (VU) instructions.

13.3.2 Instruction operands

The instruction operands are found in one of the following locations:

- System memory
- Memory buffer
- Instruction FIFO instruction words
- Load and store FIFOs
- Register buffer
- Vector unit register-file

More details of the above list of items are mentioned below:

- **System memory:** The system memory includes all memory-mapped memories attached to the bus infrastructure that are accessible by the CRYPTO block through the master bus interface
- **Memory buffer:** The CRYPTO block memory buffer is an internal SRAM with a capacity of up to 16 KB. This internal SRAM provides better latency and bandwidth characteristics than the system memory. Therefore, frequently accessed vector unit instruction operand data is typically located in the memory buffer.
- Both the block's external system memory and internal memory buffer are accessed through the same memory interface component. The access address specifies if the access is to the system memory or the internal memory buffer (also see VU_CTL.ADDR[31:14]).
- External bus masters can access both the system memory and the CRYPTO's internal memory. The external bus masters access the internal memory through the slave bus interface.
- **Instruction FIFO:** For some instructions, immediate operand data is provided by the instruction words. The limited 32-bit instruction words only allow for limited immediate operand data.
- **Load and store FIFOs:** Most instructions have stream-like operand data: sequences of bytes that are specified by the access address of the start byte. The two load FIFOs provide access to source operand data and the single store FIFO provides access to destination operand data. Typically, vector unit instruction operand data is "streamed" from the CRYPTO block memory buffer.

- **Register buffer:** Most symmetric and hash cryptographic instructions benefit from a large (2048-bit) register buffer. This register buffer provides access flexibility that is not provided by the load and store FIFOs. The register buffer is shared by different instructions to amortize its cost (silicon area). After an Active reset or a CRYPTO block reset (CTL.ENABLED), the register buffer is set to '0'.
- **Vector unit register file:** Most vector unit instructions perform large integer arithmetic functionality. For example, the VU ADD instruction can add two 4096-bit numbers. Typically, operand data is "streamed" from the memory buffer. In addition, a vector unit register file with sixteen registers is provided. Each register specifies the location of a number (start word access address) and the size of the number (the size is in bits).

13.3.3 Load and store FIFO instructions

The load and store FIFOs provide access to operand data in a "streaming" nature. Operand data is streamed through the memory interface from or to either the internal memory buffer or the system memory.

Two independent load FIFOs provide access to streamed source operand data. Each FIFO has a multi-byte buffer to prefetch operand data.

One store FIFO provides access to streamed destination operand data. The FIFO has a multi-byte buffer to temporarily hold the data before it is written to the memory interface.

Streamed operand data is specified by a memory start address and an operand size in bytes.

Three FIFO instructions are supported: FF_START, FF_STOP, and FF_CONTINUE. The FF_START and FF_STOP instructions are supported for both the load and store FIFOs. The FF_CONTINUE instruction is only supported for the load FIFOs. The FF_START and FF_CONTINUE instructions consist of three instruction words. The FF_STOP instruction consists of a single instruction word.

Table 13-1. FF_START instruction

Instruction format	**FF_START (ff_identifier, address[31:0], size[31:0])**	
Encoding	IW0[31:24] = "operation code" IW0[3:0] = ff_identifier // "8": load FIFO 0, // "9": load FIFO 1, // "12": store FIFO IW1[31:0] = address[31:0] IW2[31:0] = size[31:0]	
Mnemonic	**Operation code**	**Functionality**
FF_START	0x70	Clear the FIFO multi-byte buffer. Start streaming size[31:0] operand data bytes, starting at address[31:0]. This instruction is supported by both load and store FIFOs. For load FIFOs, data bytes are read through the memory interface. For the store FIFO, data bytes are written to the memory interface. The INSTR_OPC_ERROR interrupt cause is set when the ff_identifier is not a legal value.

Table 13-2. FF_STOP instruction

Instruction format	**FF_STOP (ff_identifier)**
Encoding	IW0[31:24] = "operation code" IW0[3:0] = ff_identifier // "8": load FIFO 0, // "9": load FIFO 1, // "12": store FIFO

Table 13-2. FF_STOP instruction

Instruction format	**FF_STOP (ff_identifier)**	
Mnemonic	**Operation code**	**Functionality**
FF_STOP	0x72	Stop streaming This instruction is supported by both load and store FIFOs. For load FIFOs, the multi-byte buffer is cleared. For the store FIFO, the multi-byte buffer is written to the memory interface; that is, the buffer is flushed. The INSTR_OPC_ERROR interrupt cause is set when the ff_identifier is not a legal value.

Table 13-3. FF_CONTINUE Instruction

Instruction format	**FF_CONTINUE (ff_identifier, address[31:0], size[31:0])**	
Encoding	IW0[31:24] = “operation code” IW0[3:0] = ff_identifier // “8”: load FIFO 0, // “9”: load FIFO 1, IW1[31:0] = address[31:0] IW2[31:0] = size[31:0]	
Mnemonic	**Operation code**	**Functionality**
FF_CONTINUE	0x71	Do not clear the FIFO multi-byte buffer. Continue streaming size[31:0] operand data bytes, starting at address[31:0]. This instruction can only be started when a previous FF_START or FF_CONTINUE instruction for the same load FIFO has read all its operand data bytes from the memory interface (see LOAD01/_FF_STATUS.BUSY). This instruction is only supported by the load FIFOs. The INSTR_OPC_ERROR interrupt cause is set when the ff_identifier is not a legal value.

The status of the load and store FIFOs is provided through the LOAD0_FF_STATUS, LOAD1_FF_STATUS, and STORE_FF_STATUS registers.

13.3.4 Register buffer instructions

The 2048-bit register buffer has two 1024-bit partitions: reg_buff[1023:0] and reg_buff[2047:1024].

The reg_buff[1023:0] partition has eight 128-bit subpartitions:

- block0[127:0] = reg_buff[0*128+127:0*128]
- block1[127:0] = reg_buff[1*128+127:1*128]
- block2[127:0] = reg_buff[2*128+127:2*128]
- block3[127:0] = reg_buff[3*128+127:3*128]
- block4[127:0] = reg_buff[4*128+127:4*128]
- block5[127:0] = reg_buff[5*128+127:5*128]
- block6[127:0] = reg_buff[6*128+127:6*128]
- block7[127:0] = reg_buff[7*128+127:7*128]

The reg_buff[1023:0] partition consists of 128 bytes: Byte offset 0 identifies reg_buff[7:0] and Byte offset 127 identifies reg_buff[1023:1016].

Some instructions work on the complete register buffer and some instructions work on 128-bit subpartitions.

Table 13-4. CLEAR instruction

Instruction format	**CLEAR ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
CLEAR	0x64	`reg_buff[2047:0] = 0;` This instruction is used to set the register buffer to ‘0’. This instruction is useful to prevent information leakage from the register buffer. The instruction also sets DEV_KEY_STATUS.LOADED to '0'.

Table 13-5. SWAP instruction

Instruction format	**SWAP ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
SWAP	0x65	`temp = reg_buff[1023:0];` `reg_buff[1023:0] = reg_buff[2047:1024];` `reg_buff[2047:1024] = temp;` This instruction swaps/exchanges the two register buffer partitions.

Table 13-6. XOR instruction

Instruction format	**XOR (offset[6:0], size[7:0])**	
Encoding	`IW[31:24] = “operation code”` `IW[14:8] = offset` `IW[7:0] = size // in the range [0,128]`	
Mnemonic	**Operation code**	**Functionality**
XOR	0x66	`data = GetFifoData (LOAD0_FIFO, size);` `data = data << (offset*8);` `reg_buff[1023:0] = reg_buff[1023:0] ^ data;` This instruction always uses load FIFO 0. *Note: This instruction can only access the lower register buffer partition.*

Table 13-7. STORE instruction

Instruction format	**STORE (offset[6:0], size[7:0])**	
Encoding	IW[31:24] = “operation code” IW[14:8] = offset IW[7:0] = size // in the range [0,128]	
Mnemonic	**Operation code**	**Functionality**
STORE	0x67	data = reg_buff[1023:0] data = data >> (offset*8); SetFifoData (STORE_FIFO, size, data)

Table 13-8. BYTE_SET instruction

Instruction format	**BYTE_SET (offset[6:0], byte[7:0])**	
Encoding	IW[31:24] = “operation code” IW[14:8] = offset IW[7:0] = byte	
Mnemonic	**Operation code**	**Functionality**
BYTE_SET	0x68	reg_buff[offset*8 + 7: offset*8] = byte;

Some instructions work on (up to) 128-bit subpartitions or blocks. In addition, these instructions can work on the load and store FIFOs. The instructions' source and destination operand identifiers are encoded as follows:

- 0: block0[127:0] = reg_buff[0*128+127:0*128]
- 1: block1[127:0] = reg_buff[1*128+127:1*128]
- 2: block2[127:0] = reg_buff[2*128+127:2*128]
- 3: block3[127:0] = reg_buff[3*128+127:3*128]
- 4: block4[127:0] = reg_buff[4*128+127:4*128]
- 5: block5[127:0] = reg_buff[5*128+127:5*128]
- 6: block6[127:0] = reg_buff[6*128+127:6*128]
- 7: block7[127:0] = reg_buff[7*128+127:7*128]
- 8: load FIFO 0
- 9: load FIFO 1
- 12: store FIFO

Table 13-9. BLOCK_MOV instruction

Instruction format	**BLOCK_MOV (reflect[1], size[3:0], dst[3:0], src0[3:0])**	
Encoding	IW[31:24] = “operation code” IW[23] = reflect IW[19:16] = size IW[15:12] = dst IW[3:0] = src0	
Mnemonic	**Operation code**	**Functionality**
BLOCK_MOV	0x40	size = (size == 0) ? 16 : size; data1 = data0 = GetBlock (src0, size); if (reflect) { // assume size of 16 B / 128 bit temp = data0; for (i = 0; i < 16; i += 1) { for (j = 0; j < 8; j += 1) { // reflection of bits in a byte data1[8*i + j] = data0[8*i + 7-j]; } } } SetBlock (dst, size, data1);

Table 13-10. BLOCK_XOR instruction

Instruction format	**BLOCK_XOR (size[3:0], dst[3:0], src1[3:0], src0[3:0])**	
Encoding	IW[31:24] = “operation code” IW[19:16] = size IW[15:12] = dst IW[7:4] = src1 IW[3:0] = src0	
Mnemonic	**Operation code**	**Functionality**
BLOCK_XOR	0x41	size = (size == 0) ? 16 : size; data0 = GetBlock (src0, size); data1 = GetBlock (src1, size); SetBlock (dst, size, data0 ^ data1);

Table 13-11. BLOCK_SET instruction

Instruction format	**BLOCK_SET (size[3:0], dst[3:0], byte[7:0])**	
Encoding	IW[31:24] = “operation code” IW[19:16] = size IW[15:12] = dst IW[7:0] = byte	
Mnemonic	**Operation code**	**Functionality**
BLOCK_SET	0x42	size = (size == 0) ? 16 : size; SetBlock (dst, size, {16{byte[7:0]}})

Table 13-12. BLOCK_CMP instruction

Instruction format	**BLOCK_CMP (size[3:0], src1[3:0], src0[7:0])**	
Encoding	IW[31:24] = “operation code” IW[19:16] = size IW[7:4] = src1 IW[3:0] = src0	
Mnemonic	**Operation code**	**Functionality**
BLOCK_CMP	0x43	size = (size == 0) ? 16 : size; data0 = GetSourceBlock (src0, size); data1 = GetSourceBlock (src1, size); RESULT.DATA |= (data0[size*8-1:0] != data1[size*8-1:0]);

Table 13-13. BLOCK_GCM instruction

Instruction format	**BLOCK_GCM (when GCM parameter is '1')**	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation code**	**Functionality**
BLOCK_GCM	0x43	Performs GCM multiplication: • {block2, block1} = XMUL (block1, block3) • block1 = GCM_Reduce ({block2, block1}) The instruction uses three specific 128-bit blocks. The XMUL operation is part of the BLOCK_GCM instruction and is a carry-less multiplication. The reduction is specific for the GCM cipher mode.

GetBlock is defined as follows:

```
GetBlock (src, size) {
   switch (src) {
      0: return block0[127:0];
      1: return block1[127:0];
      2: return block2[127:0];
      3: return block3[127:0];
      4: return block4[127:0];
      5: return block5[127:0];
      6: return block6[127:0];
      7: return block7[127:0];
      default: return GetFifoData (src, size);
   }
}
```

SetBlock is defined as follows:

```
SetBlock (dst, size, data) {
   switch (dst) {
      0: block0[127:0] = data;
      1: block1[127:0] = data;
      2: block2[127:0] = data;
      3: block3[127:0] = data;
      4: block4[127:0] = data;
      5: block5[127:0] = data;
      6: block6[127:0] = data;
      7: block7[127:0] = data;
      default: SetFifoData (dst, size, data);
   }
}
```

The GetFifoData function is defined as loading “size” bytes from a load FIFO. Up to 16 bytes can be loaded as data[127:0]. The first (top) loaded FIFO entry is mapped on data[7:0], the second loaded FIFO entry is mapped on data[15:8], and so on. The SetFifoData function is defined as storing size bytes from a load FIFO. Up to 16 bytes can be stored as data[127:0]. The first stored FIFO entry is mapped on data[7:0], the second stored FIFO entry is mapped on data[15:8], and so on.

13.4 Hash algorithms

13.4.1 SHA1 and SHA2

The SHA1 and SHA2 functionality includes three hash instructions per the SHA standard (FIPS 180-4).

- The SHA1 instruction provides all SHA1 functionality (SHA1 always uses 512-bit blocks)
- The SHA2_256 instruction provides SHA2 functionality for 512-bit blocks
- The SHA2_512 instruction provides SHA2 functionality for 1024-bit blocks

The instructions support different block sizes and hash sizes:

Instruction	Block size	Hash size
SHA1	512 bits	160 bits
SHA2_256	512 bits	256 bits
SHA2_512	1024 bits	512 bits

The SHA1 instruction supports a single algorithm with a specific message digest size. The SHA2_256 and SHA2_512 instructions support multiple algorithms with different message digest sizes.

Instruction	Algorithm	Hash size	Message digest size
SHA1	SHA-1	160 bits	160 bits
SHA2_256	SHA-224	256 bits	224 bits
	SHA-256	256 bits	256 bits
SHA2_512	SHA-384	512 bits	384 bits
	SHA-512	512 bits	512 bits
	SHA-512/224	512 bits	224 bits
	SHA-512/256	512 bits	256 bits

A SHA algorithm calculates a fixed-length hash value from a variable length message. The hash value is used to produce a message digest or signature. It is computationally impossible to change the message without changing the hash value. The algorithm is stateless: a given message always produces the same hash value. To prevent “replay attacks”, a counter may be included in the message.

The variable length message must be preprocessed: a ‘1’ bit must be appended to the message followed by ‘0’s and a bit size field. The preprocessed message consists of an integer multiple of 512 bit or 1024 bit blocks. The SHA component processes a single block at a time:

- The first SHA instruction on the first message block uses an initial hash value as defined by the standard (each SHA algorithm has a specific initial hash value).
- Subsequent SHA instructions on successive message blocks use the produced hash value of the previous SHA operation.

The SHA instruction of the last message block produces the final hash value. The message digest is a subset of this final hash value.

The SHA instructions do not perform the following functionality:

- Preprocessing of a message.
- Initialization of the register buffer with the algorithm’s specific initial hash value.
- Copy the algorithm’s message digest to memory.

Software is required to preprocess the message. The FIFO_START instruction can be used to load (load FIFO) the register buffer with the initial hash value and to store (store FIFO) the message digest.

Cryptographic function block (CRYPTO)

A SHA instruction uses “round weights” that are derived from the message block. Each SHA round uses a dedicated round weight. The “round weights” are derived on-the-fly (a new round weight is calculated when needed, and replaces a round weight from a previous round). The following table provides the number of rounds.

Instruction	Rounds
SHA1	80
SHA2_256	64
SHA2_512	80

The instructions use register buffer operands. Specifically, the instructions use reg_buff[2047:0]:

- reg_buff[1023:0] is used for the round weights. Before an instruction, this region is written with the message block. The SHA1 and SHA2_256 instructions use reg_buff[511:0] and the SHA2_512 instruction uses reg_buff[1023:0]
- reg_buff[1535:1024] is used for the hash value. Before the first SHA instruction, this region is written with the algorithm’s initial hash value. The algorithms with hash values smaller than 512 bits only use the lower bits of this region.
- reg_buff[2047:1536] is used as a working copy of the hash value. This working copy is updated during the SHA rounds and copied to reg_buff[1535:1024] at the end of the instruction.

The instructions are described in **Table 13-14** through **Table 13-16**.

Table 13-14. SHA1instruction

Instruction format	**SHA1 ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
SHA1	0x69	Perform a SHA1 function on a 512-bit message block in reg_buff[511:0] with the current 160-bit hash value in reg_buff[1183:1024]. The resulting hash value is provided in reg_buff[1183:1024]. At the end of the instruction, reg_buff[1023:0] is set to ‘0’.

Table 13-15. SHA2_256 instruction

Instruction format	**SHA2_256 ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
SHA2_256	0x6a	Perform a SHA2 function on a 512-bit message block in reg_buff[511:0] with the current 256-bit hash value in reg_buff[1279:1024]. The resulting hash value is provided in reg_buff[1279:1024]. At the end of the instruction, reg_buff[1023:0] is set to ‘0’.

Table 13-16. SHA2_512 instruction

Instruction format	**SHA2_512 ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
SHA2_512	0x6b	Perform a SHA2 function on a 1024-bit message block in reg_buff[1023:0] with the current 512-bit hash value in reg_buff[1535:1024]. The resulting hash value is provided in reg_buff[1535:1024]. At the end of the instruction, reg_buff[1023:0] is set to ‘0’.

13.4.2 SHA3

The Secure Hash Algorithm-3 (SHA-3) is a family of six algorithms:

- SHA3-224
- SHA3-256
- SHA3-384
- SHA3-512
- SHAKE128
- SHAKE256

Each of these algorithms relies on a specific instance of the Keccak-p[b, nr] permutation, with b = 1600 and nr = 24. The parameter b specifies the permutation bit width (1600 bits) and the parameter nr specifies the number of permutation rounds (24 rounds).

The permutation bit width b is the sum of:

- The rate r, which is the number of consumed message bits or produced digest bits per application of the Keccak permutation (SHA3 instruction).
- The capacity c, which is defined as b-r

All six hash algorithms are constructed by padding a message M and applying the Keccak-p[1600, 24] permutation repeatedly. The algorithms differ in terms of the rate r and the padding.

The permutation’s rate r determines the speed of the algorithm: a higher rate requires less applications of the permutation function (SHA3 instruction).

The permutation’s capacity c determines the security of the algorithm: a higher capacity provides higher security.

Table 13-17 lists the algorithms’ rate and capacity. In addition, it lists the size of the message digest. **Note:** The SHA3 hash algorithms have fixed-length digests and the SHAKE extendable output functions have variable-length digests.

Table 13-17. Algorithm rate and capacity

Algorithm	**Rate, capacity**	**Digest length**
SHA3-224	r = 1152 b, c = 448 b	224 b (28 B)
SHA3-256	r = 1088 b, c = 512 b	256 b (32 B)
SHA3-384	r = 832 b, c = 768 b	384 b (48 B)
SHA3-512	r = 576 b, c = 1024 b	512 b (64 B)
SHAKE128	r = 1344 b, c = 256 b	Variable length
SHAKE256	r = 1088 b, c = 512 b	Variable length

The padded message has a length that is an integer multiple of the rate r. The message is processed by repeatedly applying the SHA3 permutation instruction. Each instruction application uses a message block of r bits. The permutation function combines the message block of r bits with the current permutation state of b bits and calculates a new permutation state of b bits.

- The first SHA3 instruction on the first message block uses the first message block as the initial permutation state.
- Subsequent SHA3 instructions on successive message blocks first combine the message block with an exclusive or (XOR) with the current state and calculate a new permutation state.

The message digest is produced in a similar way as the message is consumed: each application of the permutation instruction produces a message digest of r bits.

The SHA3 functionality includes a SHA3 instruction. The instruction performs a permutation on the 1600-bit state in reg_buf[1599:0].

The instruction implements the permutation, all other functionality is implemented in sofftware (combination of message block with the permutation state and copying the message digest to memory).

Table 13-18 describes the instruction.

Table 13-18. SHA3 instruction

Instruction format	**SHA3 ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
SHA3	0x6c	Perform a SHA3 permutation on a 1600-bit state reg_buff[1599:0]. The resulting state is provided in reg_buff[1599:0].

13.5 DES and TDES

The (T)DES functionality includes block ciphers and inverse block ciphers according to the DES and triple DES standard (FIPS 46-3). Note that this standard was withdrawn, because it no longer provides the security that is needed to protect federal government information.

(T)DES is supported for backward compatibility.

- The DES block cipher encrypts a 64-bit block of plaintext data into a 64-bit block of ciphertext data.
- The DES inverse block cipher decrypts a 64-bit block of ciphertext data into a 64-bit block of plaintext data.

The DES symmetric key consists of 64 bit. Only 56 bits are used by the algorithm, the other 8 bits are used for parity checking.

Tripple-DES applies the DES block cipher three times. Each application uses a different key, which results in a 192-bit TDES key bundle consisting of three 64-bit keys SK1, SK2 and SK3.

- The TDES block cipher uses the DES block cipher with key SK1 to encrypt a 64-bit plaintext into a 64-bit block T1. Next, it uses the DES inverse block cipher with key SK2 to decrypt the 64-bit block T1 into a 64-bit block T2. Next, it uses the DES block cipher with key SK3 to encrypt the 64-bit block T2 into a 64-bit ciphertext.
- The TDES inverse block cipher uses the DES inverse block cipher with key SK3 to decrypt a 64-bit ciphertext into a 64-bit block T2. Next, it uses the DES block cipher with key SK2 to encrypt the 64-bit block T2 into a 64-bit block T1. Next, it uses the DES inverse block cipher with key SK1 to decrypt the 64-bit block T1 into a 64-bit plaintext.

There are four instructions: DES, DES_INV, TDES and TDES_INV, which use register buffer operands. Specifically, the instructions use the 128-bit subpartitions block0, block1, block4, block5, block6, and block7.

- Subpartitions block0 and block1 are used for plaintext and ciphertext data
- Subpartitions block4, block5, block6, and block7 are used for key information (block5 and block7 are only used by the TDES and TDES_INV instructions)

Cryptographic function block (CRYPTO)

Unlike the BLOCK_MOV instruction, the (T)DES instructions use predetermined/fixed subpartitions.
The instructions are described in **Table 13-19** through **Table 13-22**.

Table 13-19. DES instruction

Instruction format	**DES ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
DES	0x52	Perform a DES block cipher. • The 64-bit plaintext is in block0[63:0] • The resulting 64-bit ciphertext is in block1[63:0] • The 64-bit input key is in block4[63:0] • The 64-bit output key (round key of the final cipher round) is in block6[63:0] The instruction is non-destructive: block0 and block4 are not changed by the instruction.

Table 13-20. DES_INV instruction

Instruction format	**DES_INV ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
DES_INV	0x53	Perform a DES inverse block cipher. • The 64-bit ciphertext is in block0[63:0] • The resulting 64-bit plaintext is in block1[63:0] • The 64-bit input key is in block4[63:0] • The 64-bit output key (round key of the final inverse cipher round) is in block6[63:0] The instruction is non-destructive: block0 and block4 are not changed by the instruction.

Table 13-21. TDES instruction

Instruction format	**TDES ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
TDES	0x54	Perform a TDES block cipher. • The 64-bit plaintext is in block0[63:0]. • The resulting 64-bit ciphertext is in block1[63:0]. • The 192-bit input key is in {block5[63:0], block4[127:0]} (= {SK3[63:0], SK2[63:0], SK1[63:0]}). • The 192-bit output key (round key of the final cipher round) is in {block7[63:0], block6[127:0]}. The instruction is non-destructive: block0, block4 and block5 are not changed by the instruction.

Table 13-22. TDES_INV instruction

Instruction format	**TDES_INV ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
TDES_INV	0x55	Perform a TDES inverse block cipher. • The 64-bit ciphertext is in block0[63:0]. • The resulting 64-bit plaintext is in block1[63:0]. • The 192-bit input key is in {block5[63:0], block4[127:0]} (= {SK3[63:0], SK2[63:0], SK1[63:0]}). • The 192-bit output key (round key of the final cipher round) is in {block7[63:0], block6[127:0]}. The instruction is non-destructive: block0, block4 and block5 are not changed by the instruction.

The (T)DES instructions are used with the FF_START, FF_CONTINUE, FF_STOP, BLOCK_MOV, BLOCK_XOR, and BLOCK_SET instructions to implement different block cipher modes, such as EBC, CBC, OFB, CTR, and CMAC.

13.6 AES

The AES functionality includes a block cipher and an inverse block cipher per the AES standard (FIPS 197):

- The block cipher (AES instruction) encrypts a 128-bit block of plaintext data into a 128-bit block of ciphertext data
- The inverse block cipher (AES_INV instruction) decrypts a 128-bit block of ciphertext data into a 128-bit block of plaintext data

AES is a symmetric block cipher: it uses the same symmetric key for the block cipher and inverse block cipher. The (inverse) block cipher consists of multiple rounds. Each round uses a round key that is generated from the symmetric key.

The block cipher uses the symmetric key as the (start) round key for the first cipher round. The round key for second cipher round is generated from the symmetric key. The round key for the third cipher round is generated from the round key of the second cipher round, and so forth. The round key for the last cipher round is generated from the round key of the one-before-last cipher round.

The inverse block cipher uses the round key of the final block cipher round as the (start) round key for the first inverse cipher round. The round key for the second inverse cipher round is generated from the round key of the first inverse cipher round, and so forth. The round key for the last inverse cipher round is generated from the round key of the one-before-last inverse cipher round. The round key of the last inverse cipher round is the same as the round key of the first cipher round (the symmetric key).

Round key generation is independent of the plaintext data or ciphertext data. The AES instruction requires the symmetric key as input to the block cipher. The AES_INV instruction requires the round key of the final cipher round as input to the inverse block cipher.

The component supports 128 bit, 192 bit, and 256 bit keys. The key size is specified by AES_CTL.KEY_SIZE[1:0]. The key size determines the number of rounds. **Table 13-23** gives the number of rounds.

Table 13-23. AES cipher rounds

AES key size	Rounds
AES128	10 rounds
AES192	12 rounds
AES256	14 rounds

AES and AES_INV are the two different AES instructions; they use register buffer operands. Specifically, the instructions use the 128-bit subpartitions block0, block1, block4, block5, block6, and block7.

- Subpartitions block0 and block1 are used for plaintext and ciphertext data
- Subpartitions block4, block5, block6 and block7 are used for key information

Unlike, for example, the BLOCK_MOV instruction, the AES and AES_INV instructions use predetermined/fixed subpartitions.

The instructions are described in **Table 13-24** and **Table 13-25**.

Table 13-24. AES instruction

Instruction format	**AES ( )**	
Encoding	`IW[31:24] = “operation code”`	
Mnemonic	**Operation code**	**Functionality**
AES	0x50	Perform an AES block cipher. • The 128-bit plaintext is in block0. • The resulting 128-bit ciphertext is in block1. • The input key is in block4 and block5. For a 128-bit key, the key is in block4[127:0]. For a 192-bit key, the key is in {block5[63:0], block4[127:0]}. For a 256-bit key, the key is in {block5[127:0], block4[127:0]}. • The output key (round key of the final cipher round) is in block6 and block7. For a 128-bit key, the key is in block6[127:0]. For a 192-bit key, the key is in {block7[63:0], block6[127:0]}. For a 256-bit key, the key is in {block7[127:0], block6[127:0]}. Note that the output key is the input key for an AES inverse block cipher. The instruction is non-destructive: block0, block4, and block5 are not changed by the instruction. Note that the AES instruction can be used to derive the AES_INV input key from the symmetric key.

Table 13-25. AES_INV instruction

Instruction format	**AES_INV ( )**	
Encoding	IW[31:24] = “operation code”	
Mnemonic	**Operation code**	**Functionality**
AES_INV	0x51	Perform an AES block cipher. • The 128-bit ciphertext is in block0. • The resulting 128-bit plaintext is in block1. • The input key is in block4 and block5. For a 128-bit key, the key is in block4[127:0]. For a 192-bit key, the key is in {block5[63:0], block4[127:0]}. For a 256-bit key, the key is in {block5[127:0], block4[127:0]}. • The output key (round key of the final inverse cipher round) is in block6 and block7. For a 128-bit key, the key is in block6[127:0]. For a 192-bit key, the key is in {block7[63:0], block6[127:0]}. For a 256-bit key, the key is in {block7[127:0], block6[127:0]}. Note that the output key is the input key for an AES block cipher. The instruction is non-destructive: block0, block4 and block5 are not changed by the instruction.

The AES and AES_INV instructions are used with the FF_START, FF_CONTINUE, FF_STOP, BLOCK_MOV, BLOCK_XOR, and BLOCK_SET instructions to implement different block cipher modes, such as EBC, CBC, OFB, CTR, and CMAC.

13.7 CRC

The CRC functionality performs a cyclic redundancy check with a programmable polynomial of up to 32 bits.

The load FIFO 0 provides the data (and the size of the data) on which the CRC is performed. The data must be laid out in little endian format (least significant byte of a multi-byte word should be located at the lowest memory address of the word).

CRC_DATA_CTL.DATA_XOR[7:0] specifies a byte pattern with which each data byte is XOR’d. This allows for inversion of the data byte value.

CRC_CTL.DATA_REVERSE allows for bit reversal of the data byte (this provides support for serial interfaces that transfer bytes in most-significant-bit first and least-significant bit first configurations).

CRC_POL_CTL.POLYNOMIAL[31:0] specifies the polynomial. The polynomial specification omits the high order bit and should be left aligned. For example, popular 32-bit and 16-bit CRC polynomials are specified as follows:

CRC32: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

CRC_POL_CTL.POLYNOMIAL[31:0] = 0x04c11db7

CRC16-CCITT: $x^{16} + x^{12} + x^5 + 1$

CRC_POL_CTL.POLYNOMIAL[31:0] = (0x1021 << 16)

CRC16: $x^{16} + x^{15} + x^2 + 1$

CRC_POL_CTL.POLYNOMIAL[31:0] = (0x8005 << 16)

RESULT.DATA[31:0] holds the LFSR state of the CRC calculation. Before the CRC operation, this field should be initialized with the CRC seed value.

CRC_REM_CTL.REM_XOR[31:0] specifies a 32-bit pattern with which the RESULT.DATA[31:0] LFSR state is XOR'd.

CRC_CTL.REM_REVERSE allows for bit reversal of the XOR'd state.

CRC_REM_RESULT.REM[31:0] holds the result of the CRC calculation, and is derived from the end state of the CRC calculation (RESULT.DATA[31:0]).

The CRC instruction is described in **Table 13-26**.

Table 13-26. CRC instruction

Instruction format	**CRC ( )**	
Encoding	`IW[31:24] = "operation code"`	
Mnemonic	**Operation code**	**Functionality**
CRC	0x58	Perform the CRC function on all data in load FIFO 0. The result of the CRC function is provided through CRC_REM_RESULT.REM[31:0]. If the data is scattered in memory, a single FF_START and multiple FF_CONTINUE instructions should be used to gather the data in a load FIFO 0 Byte stream. Each FF_START and FF_CONTINUE instruction should be followed by a CRC instruction, which processes all the bytes of the associated FIFO instruction (the CRC instruction does not have a "size" operand).

Figure 13-2 illustrates the CRC functionality.

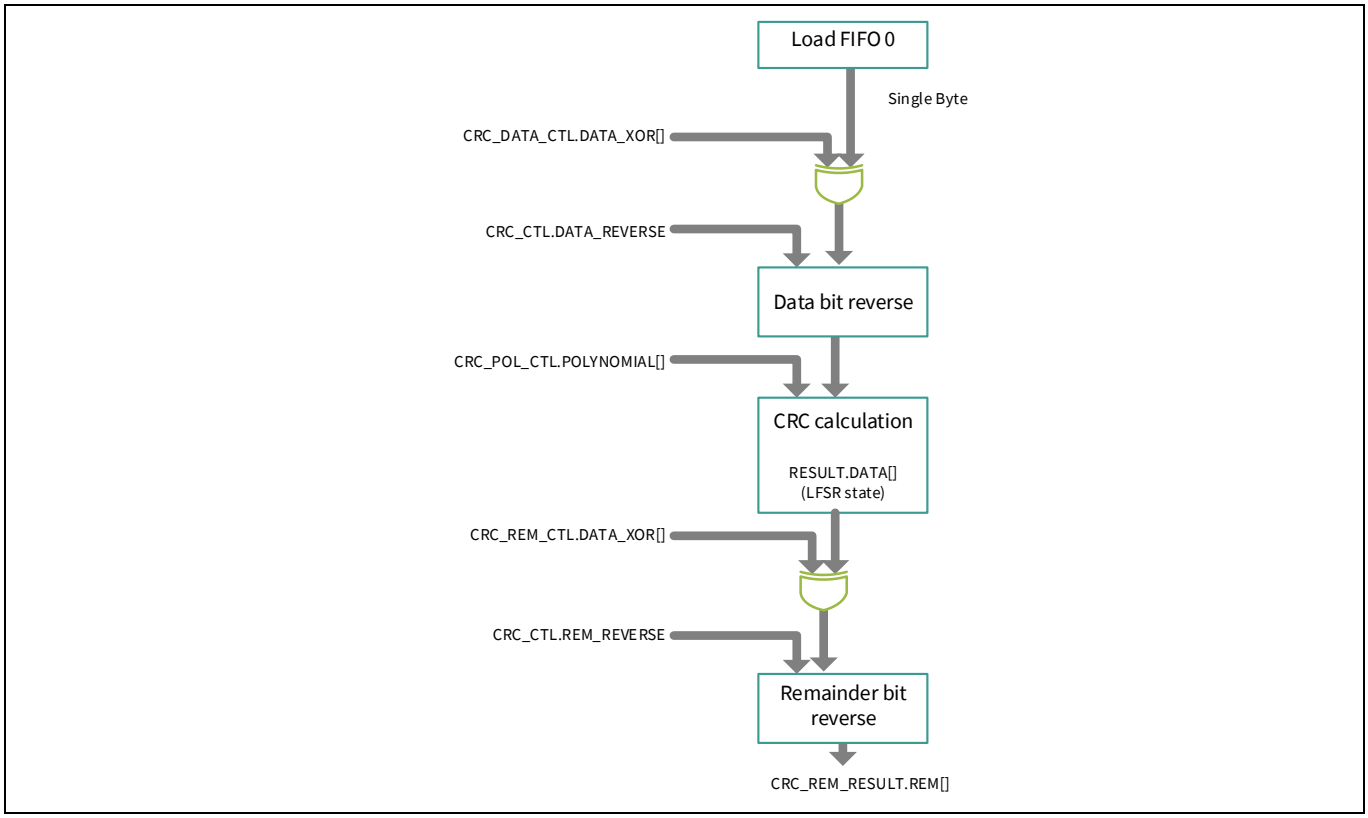


Figure 13-2. CRC functionality

The Linear Feedback Shift Register functionality operates on the LFSR state. It uses the programmed polynomial and consumes a data bit for each iteration (eight iterations are performed per cycle to provide a throughput of one data byte per cycle). **Figure 13-3** illustrates the functionality for the CRC32 polynomial ($x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$).

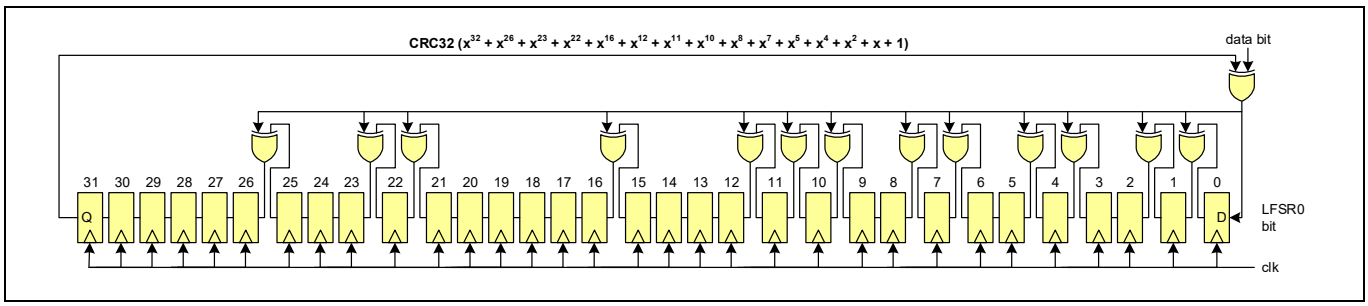


Figure 13-3. CRC32 functionality

Different CRC algorithms require different seed values and have different requirements for XOR functionality and bit reversal. **Table 13-27** provides the proper settings for the CRC32, CRC16-CCITT, and CRC16 algorithms. The table also provides the remainder after the algorithms are performed on a five-byte array {0x12, 0x34, 0x56, 0x78, 0x9a}.

Table 13-27. CRC32, CRC16-CCITT, and CRC16 algorithm settings

MMIO register field	CRC32	CRC16-CCITT	CRC16
CRC_POL_CTL.POLYNOMIAL	0x04c11db7	0x10210000	0x80050000
CRC_CTL.DATA_REVERSE	1	0	1
CRC_DATA_CTL.DATA_XOR	0x00	0x00	0x00
RESULT.DATA (seed)	0xffffffff	0xffff0000	0xffff0000
CRC_CTL.REM_REVERSE	1	0	1
CRC_REM_CTL.REM_XOR	0xffffffff	0x00000000	0x00000000
CRC_REM_RESULT.REM	0x3c4687af	0xf8a00000	0x000048d0

13.8 PRNG

The pseudo random number generation (PRNG) component generates pseudo random numbers in a fixed range [0, PR_MAX_CTL.DATA[31:0]]. The generator is based on three Fibonacci-based Linear Feedback Shift Registers (LFSRs). The following three irreducible polynomials (with minimum feedback) are used:

- 32-bit polynomial: $x^{32} + x^{30} + x^{26} + x^{25} + 1$
- 31-bit polynomial: $x^{31} + x^{28} + 1$
- 29-bit polynomial: $x^{29} + x^{27} + 1$

Figure 13-4 illustrates the LFSR functionality.

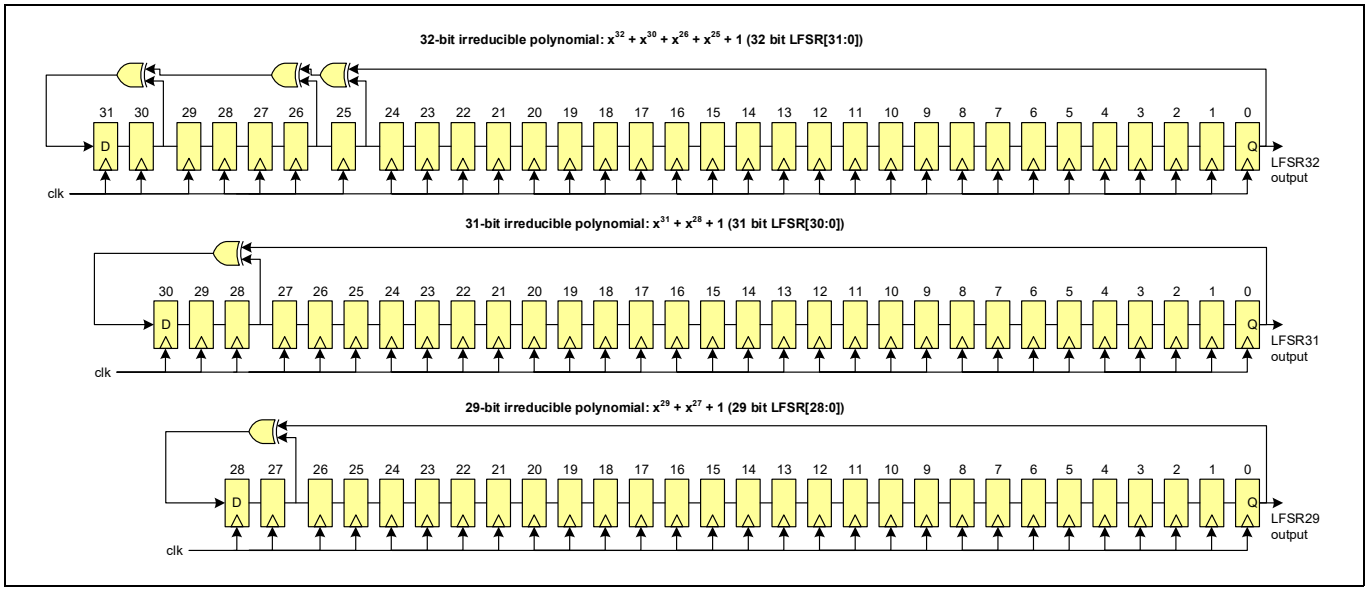


Figure 13-4. Fixed fibonacci-based LFSRs

Software initializes the LFSRs with non-zero seed values. The PR_LFSR_CTL0, PR_LFSR_CTL1 and PR_LFSR_CTL2 registers are provided for this purpose. At any time, the state of these registers can be read to retrieve the state of the LFSRs. The 32-bit LFSR generates a repeating bit sequence of 232 – 1 bits, the 31-bit LFSR generates a repeating bit sequence of 231 – 1 and the 29-bit LFSR generates a repeating bit sequence of 229 – 1. As the numbers 232-1, 231-1, and 229-1 are relatively prime, the XOR output is a repeating bit sequence of roughly 232+31+29.

The final pseudo random bit is the XOR of the three bits that are generated by the individual LFSRs.

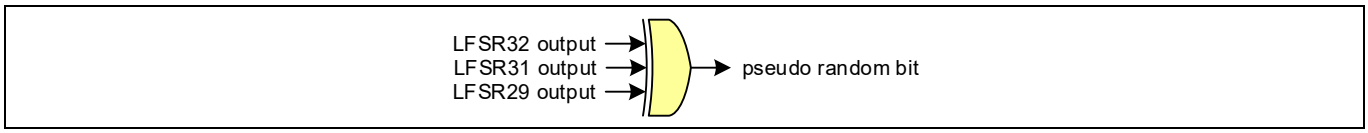


Figure 13-5. XOR reduction logic

The pseudo random number generator uses a total of 33 pseudo random bits to generate a result in the range [0, PR_MAX_CTL.DATA[31:0]].

To generate a pseudo random number result, the following calculation is performed.

MAX[31:0] = PR_MAX_CTL.DATA[31:0]

MAX_PLUS1[32:0] = MAX[31:0] + 1;

product[63:0] = MAX_PLUS1[32:0] * pr[32:1] + MAX[31:0] * pr[0];

result = product[63:32];

PR_CMD.START is the PR command. The maximum value of the generated random number (in PR_RESULT.DATA) is specified by PR_MAX_CTL.DATA. The PR command can be executed in parallel with the instruction FIFO instructions and the TR command.

13.9 TRNG

The true random number generator component (TRNG) generates true random numbers. The bit size of these generated numbers is programmable (TR_CTL.SIZE is in the range [0,32]).

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 (temperature) sensitivity of the inverter delays, jitter is introduced on a ring's oscillating signal. The jittered oscillating signal is sampled to produce a “digitized analog signal” (DAS). This is done for all multiple ring oscillators.

To increase entropy and to reduce bias in DAS bits, the DAS bits are further postprocessed. Post-processing involves two steps:

- An optional reduction step (over up to six ring oscillator DAS bits and over one or multiple DAS bit periods) to increase entropy.
- An optional “von Neumann correction” step to reduce a ‘0’ or ‘1’ bias.
  This correction step processes pairs of reduction bits as produced by the previous step. Given two reduced bits r0 and r1 (with r0 being produced before r1), the correction step is defined as follows:
  - {r0, r1} = {0, 0}: no bit is produced
  - {r0, r1} = {0, 1}: a ‘0’ bit is produced (bit r0)
  - {r0, r1} = {1, 0}: a ‘1’ bit is produced (bit r0)
  - {r0, r1} = {1, 1}: no bit is produced

In other words the correction step only produces a bit on a ‘0’ to ‘1’ or ‘1’ to ‘0’ transition. Note that for a random input bit sequence, the correction step produces an output bit sequence of roughly one-quarter the frequency of the input bit sequence (the input reduction bits are processed in non-overlapping pairs and only half of the pair encodings result in an output bit).

Post-processing produces bit samples that are considered true random bit samples. The true random bit samples are shifted into a register, to provide random values of up to 32 bits.

As a result of high-switching activity, ring oscillators consume a significant amount of power. Therefore, when the TRNG functionality is disabled, the ring is “broken” to prevent switching. When the TRNG functionality is enabled, a ring oscillator initially has predictable behavior. However, over time, infinitesimal environmental (temperature) changes cause an increasing deviation from this predictable behavior.

- During the initial delay, the ring oscillator is not a reliable physical noise source.
- After an initial delay, the same ring oscillator will show different oscillation behavior and provides a reliable physical noise source.

Therefore, the DAS bits can be dropped during an initialization period (TR_CTL.INIT_DELAY[]).

Figure 13-6 gives an overview of the TRNG component.

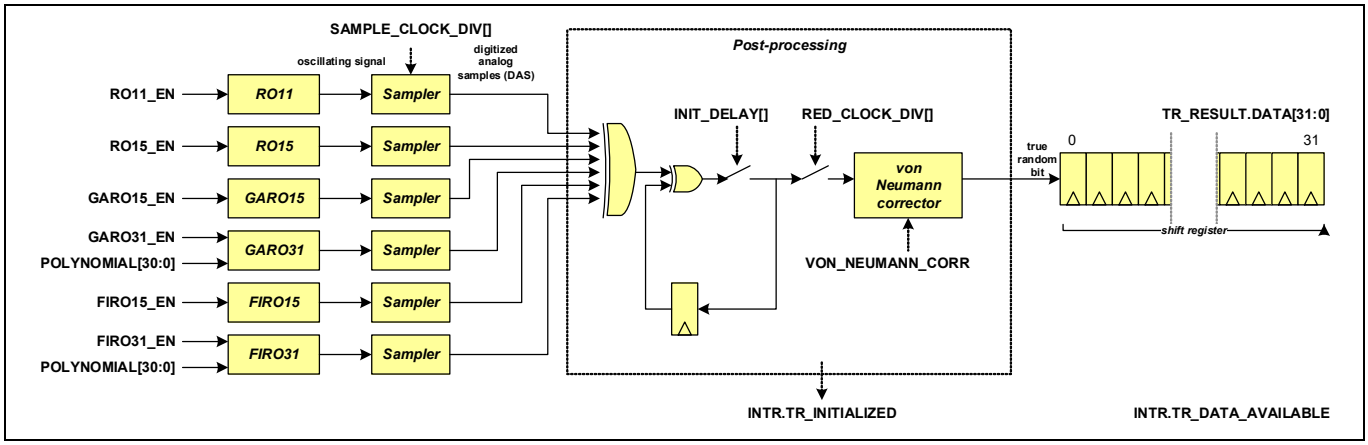


Figure 13-6. TRNG overview

The “Sampler” logic digitizes an oscillating signal. **Figure 13-7** illustrates the functionality (the complete logic is implemented using standard platform toolkit components).

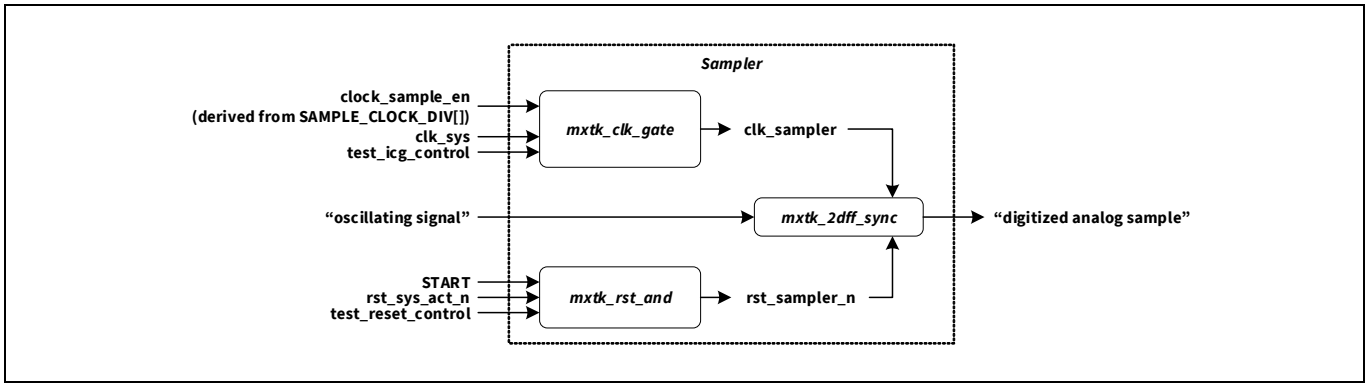


Figure 13-7. Sampler logic

Note that when a ring oscillator is disabled, the synchronization logic is reset. Therefore, the ring oscillator contributes a constant ‘0’ to the reduction step of the post-processing. As mentioned, the TRNG relies on up to six ring oscillators:

- RO11: A fixed ring oscillator consisting of 11 inverters
- RO15: A fixed ring oscillator consisting of 15 inverters
- GARO15: A fixed Galois based ring oscillator of 15 inverters
- GARO31: A flexible Galois based ring oscillator of up to 31 inverters. A programmable polynomial of up to order 31 provides the flexibility in the oscillator feedback.
- FIRO15: A fixed Fibonacci based ring oscillator of 15 inverters
- FIRO31: A flexible Fibonacci based ring oscillator of up to 31 inverters. A programmable polynomial of up to order 31 provides the flexibility in the oscillator feedback

Each ring oscillator can be enabled or disabled. When disabled, the ring is “broken” to prevent switching. The following Figures illustrate the schematics of the fixed ring oscillators.

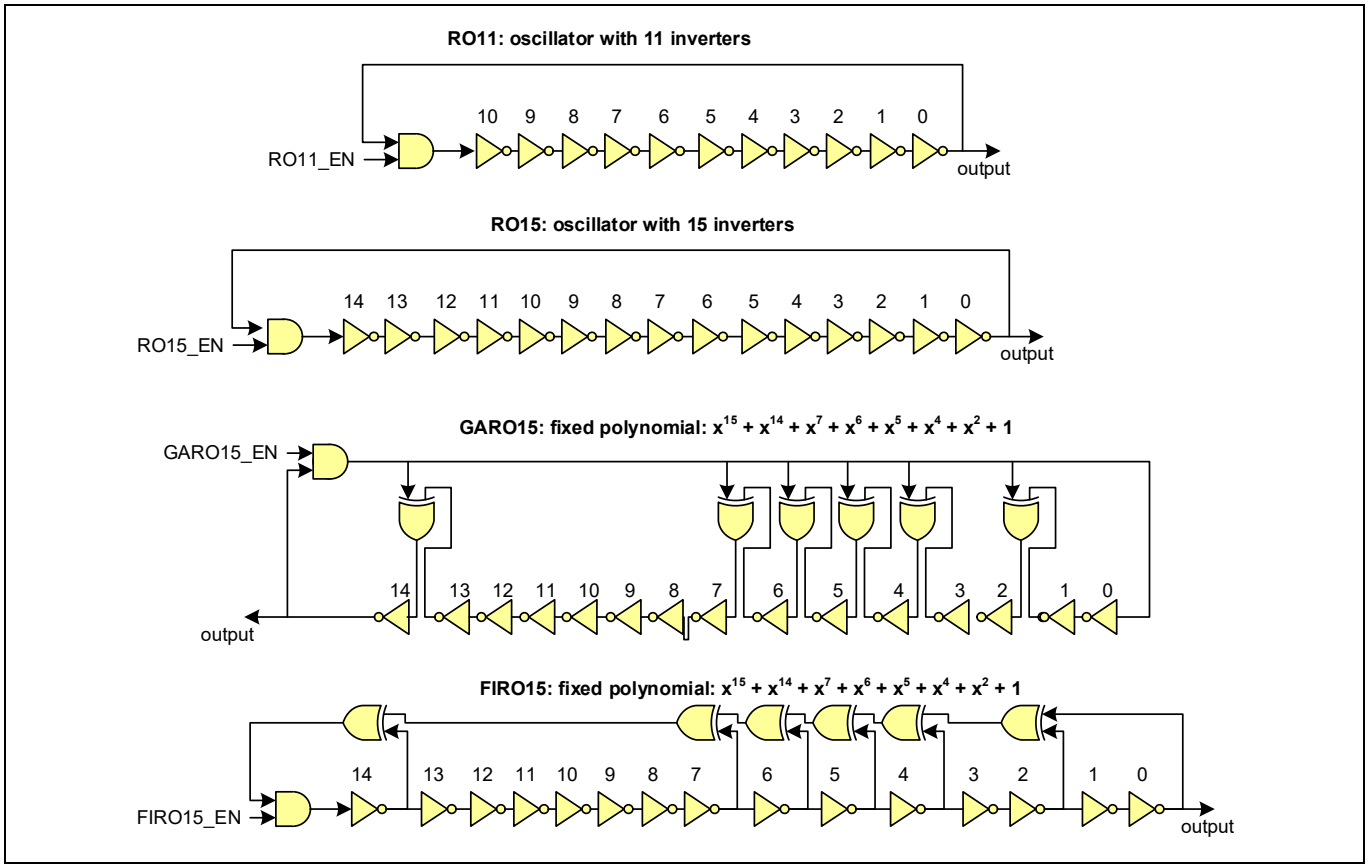


Figure 13-8. Four fixed ring oscillators: RO11, RO15, GARO15, FIRO15

The XXX_EN enable signals originate from a MMIO register field.

The flexible Galois- and Fibonacci-based ring oscillators rely on programmable polynomials to specify the oscillator feedback. This allows for rings of 1, 3, 5, …, 31 inverters (an odd number is required to generate an oscillating signal). **Figure 13-9** gives an overview of the Galois-based ring oscillator.

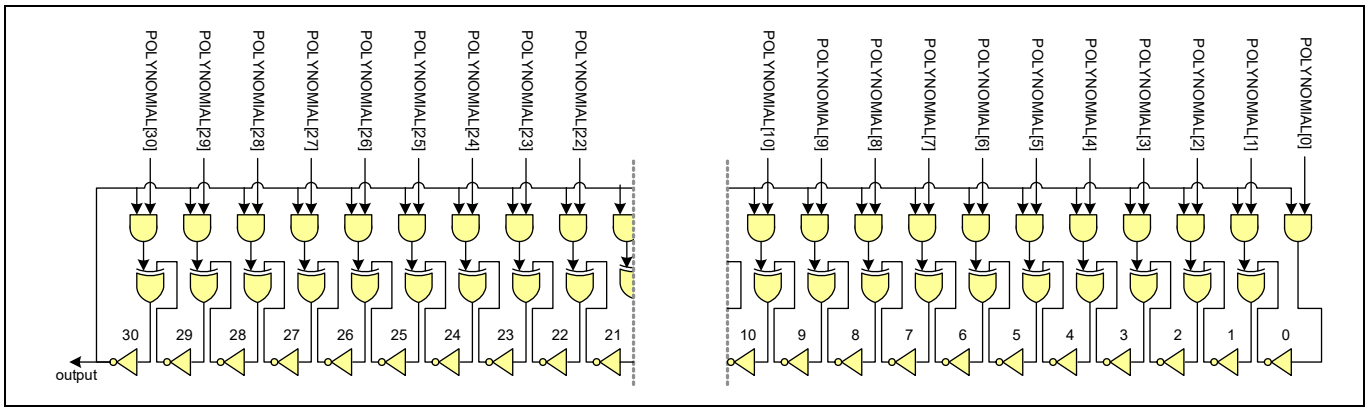


Figure 13-9. Flexible Galois-based ring oscillators: GARO31

Cryptographic function block (CRYPTO)

When the ring oscillator is disabled (GARO31_EN is '0'), the polynomial is forced to "0" and the ring is broken as illustrated by **Figure 13-10**.

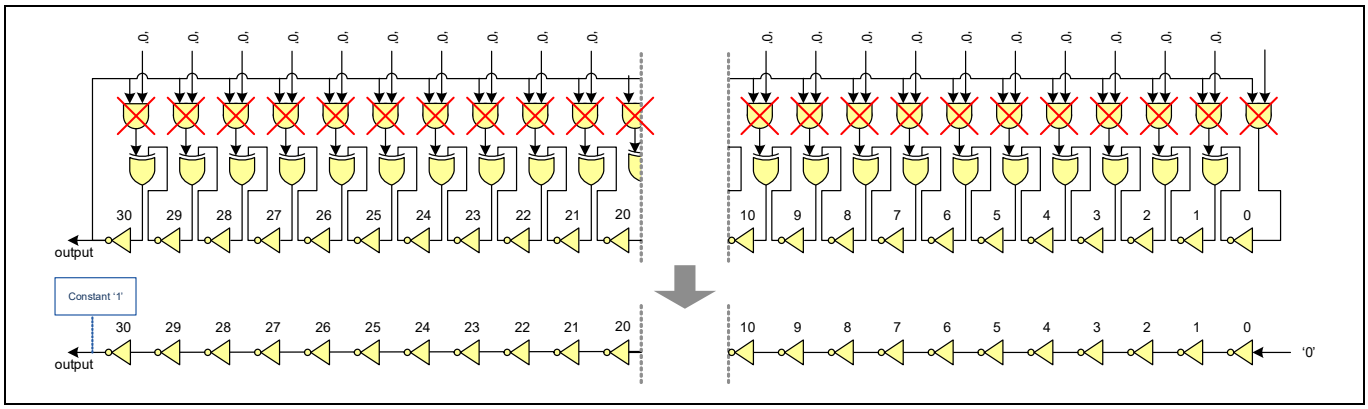


Figure 13-10. GARO31 "stopped"

The programmable polynomial specifies the oscillator feedback. **Figure 13-11** illustrates two examples.

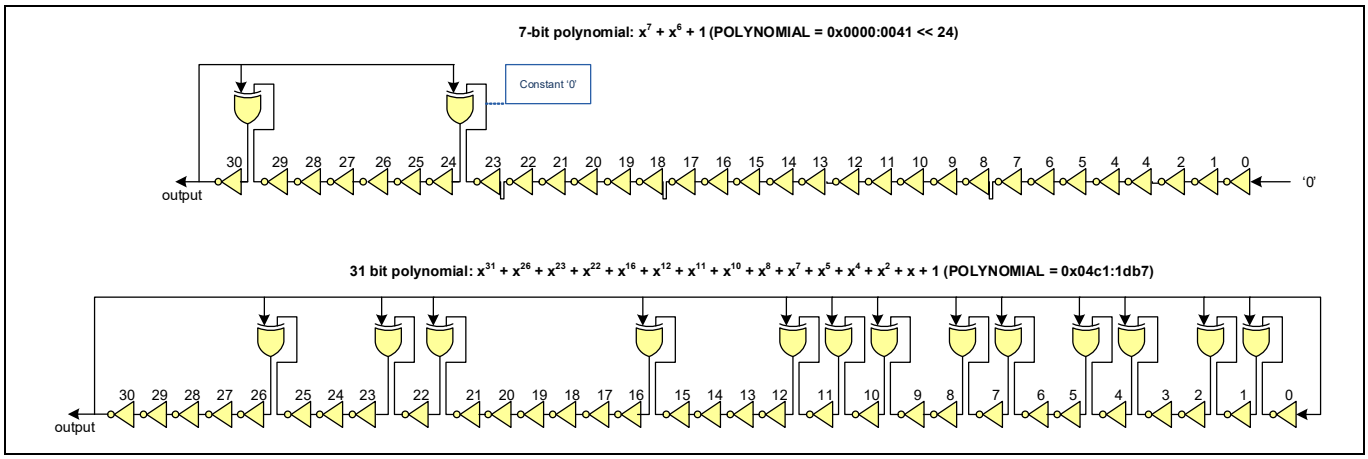


Figure 13-11. GARO31 – Two examples

Figure 13-12 gives an overview of the Fibonacci-based ring oscillator.

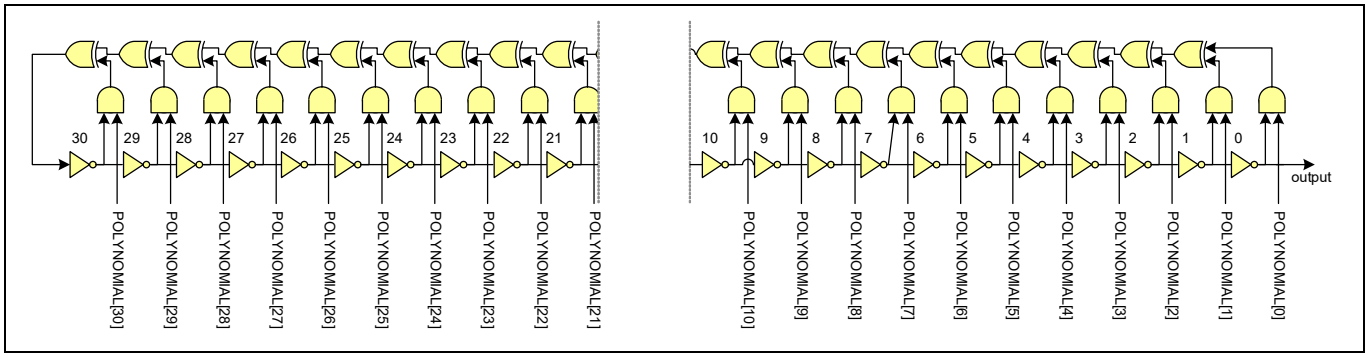


Figure 13-12. Flexible Fibonacci-based Ring Oscillators: FIRO31

When the ring oscillator is disabled (FIRO31_EN is '0'), the polynomial is forced to "0" and the ring is broken as illustrated by **Figure 13-13**.

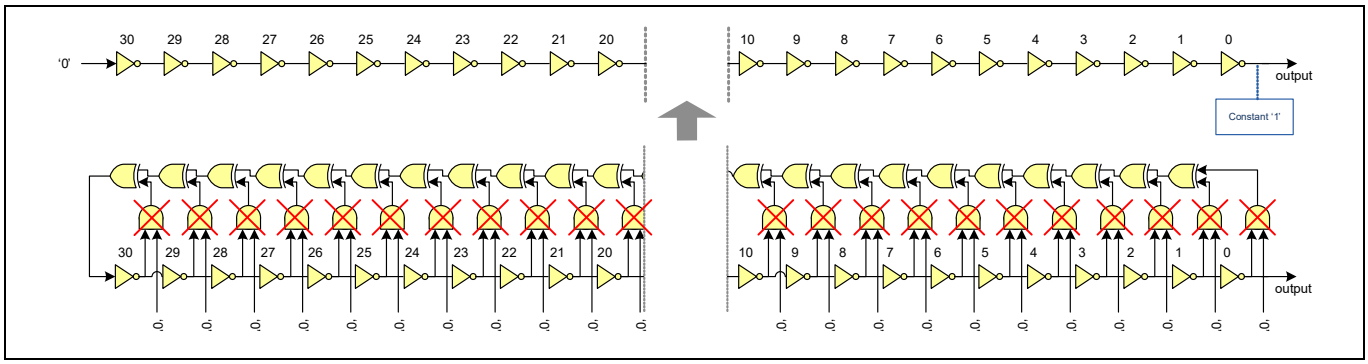


Figure 13-13. FIRO31 "stopped"

The programmable polynomial specifies the oscillator feedback. **Figure 13-14** illustrates two examples.

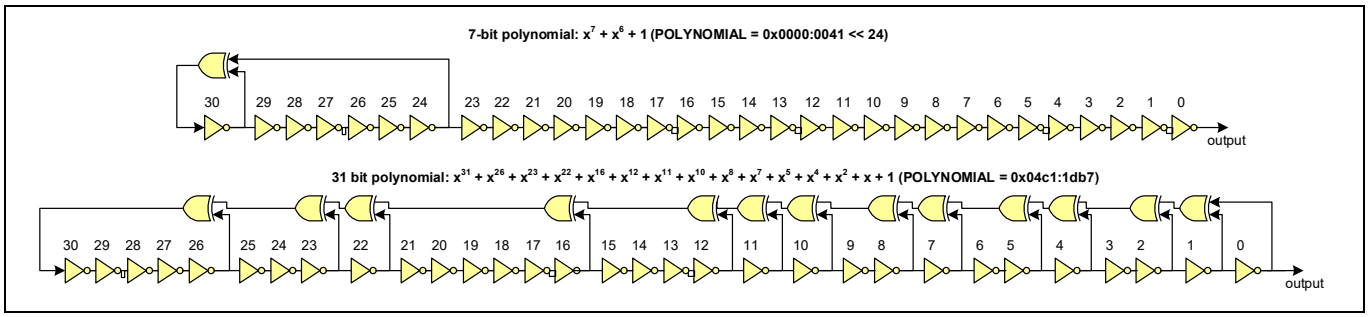


Figure 13-14. FIRO31 – Two examples

There is one TR command, TR_CMD.START, which can be executed in parallel with the instruction FIFO instructions and the PR command. The size of the generated random number (in TR_RESULT.DATA) is specified by TR_CTL.SIZE.

The TRNG has a built-in health monitor that performs tests on the digitized noise source to detect deviations from the intended behavior. For example, the health monitor detects "stuck at" faults in the digitized analog samples. The health monitor tests one out of three selected digitized bit streams:

- DAS bitstream. This is XOR of the digitized analog samples
- RED bitstream. This is the bitstream of reduction bits. Note that each reduction bit may be calculated over multiple DAS bits.
- TR bitstream. This is the bitstream of true random bits (after the "von Neumann reduction" step)

The health monitor performs two different tests:

The repetition count test. This test checks for the repetition of the same bit value ('0' or '1') in a bitstream. A detection indicates that a specific active bit value (specified by a status field BIT) has repeated for a pre-programmed number of bits (specified by a control field CUTOFF_COUNT[7:0]). The test uses a counter to maintain the number of repetitions of the active bit value (specified by a status field REP_COUNT[7:0]).

If the test is started (specified by START_RC field) and a change in the bitstream value is observed, the active bit value BIT is set to the new bit value and the repetition counter REP_COUNT[] is set to '1'. If the bitstream value is unchanged, the repetition counter REP_COUNT[] is incremented by "1".

A detection stops the repetition count test (the START_RC field is set to '0'), sets the associated interrupt status field to '1' and ensures that hardware does not modify the status fields. When the test is stopped, REP_COUNT[] equals CUTOFF_COUNT[].

A detection stops the TRNG functionality (all TR_CTL.XXX_EN fields are set to '0') if TR_CTL.STOP_ON_RC_DETECT is set to '1'.

The adaptive proportion test. This test checks for a disproportionate occurrence of a specific bit value ('0' or '1') in a bit stream. A detection indicates that a specific active bit value (specified by a status field BIT) has occurred a pre-programmed number of times (specified by a control field CUTOFF_COUNT[15:0]) in a bit sequence of a specific bit window size (specified by a control field WINDOW_SIZE[15:0]). The test uses a counter to maintain an index in the current window (specified by WINDOW_INDEX[15:0]) and a counter to maintain the number of occurrences of the active bit value (specified by a status field OCC_COUNT[15:0]).

If the test is started (specified by START_AP field), the bitstream is partitioned in bit sequences of a specific window size. At the first bit of a bit sequence, the active bit value BIT is set to the first bit value, the counter WINDOW_INDEX is set to "0" and the counter OCC_COUNT is set to "1". For all other bits of a bit sequence, the counter WINDOW_INDEX is incremented by "1". If the new bit value equals the active bit value BIT, the counter OCC_COUNT[15:0] is incremented by "1". Note that the active bit value BIT is only set at the first bit of a bit sequence.

A detection stops adaptive proportion test (the START_AP field is set to '0'), sets the associated interrupt status field to '1' and ensures that hardware does not modify the status fields. When the test is stopped, OCC_COUNT[] equals CUTOFF_COUNT[] and the WINDOW_INDEX identifies the bit sequence index on which the detection occurred.

A detection stops the TRNG functionality (all TR_CTL.XXX_EN fields are set to '0') if TR_CTL.STOP_ON_AP_DETECT is set to '1'.

Figure 13-15 illustrates the health monitor functionality.

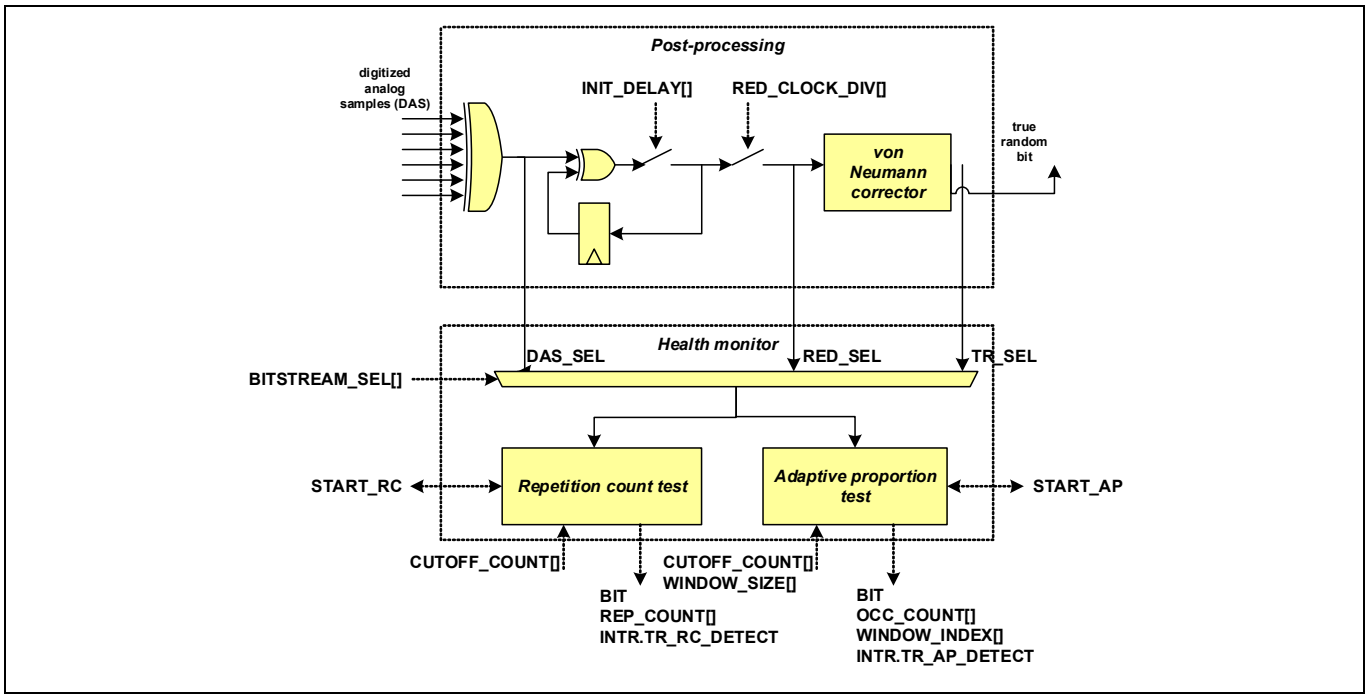


Figure 13-15. Health Monitor Functionality

Implementation note. The ring oscillators have special design requirements. They are not synthesized logic, but manually constructed from selected cells from the standard cell library. Three types of standard cells are required:

- An inverter cell. The selected cell should have similar rise and fall time requirements. Clock tree inverter cells tend to have this property and are the preferred implementation of the inverter cell.
- A two-input XOR cell
- A two-input AND cell

The design uses three Verilog design modules to instantiate the selected cells. This allows easy porting from one standard cell library to another standard cell library. The three Verilog design modules should not be changed by tools (similar to how the platform toolkit components are treated in the design flow).

A ring oscillator should be placed and routed in a self-contained rectangular area. This area should preferably be as small as possible.

13.10 Vector unit

The vector unit (VU) addresses the requirements of asymmetric key cryptography. Similar to other cryptography functionality, the VU connects to the memory interface through the load and store FIFOs. The memory buffer or system memory contains memory operand data for the vector unit instructions.

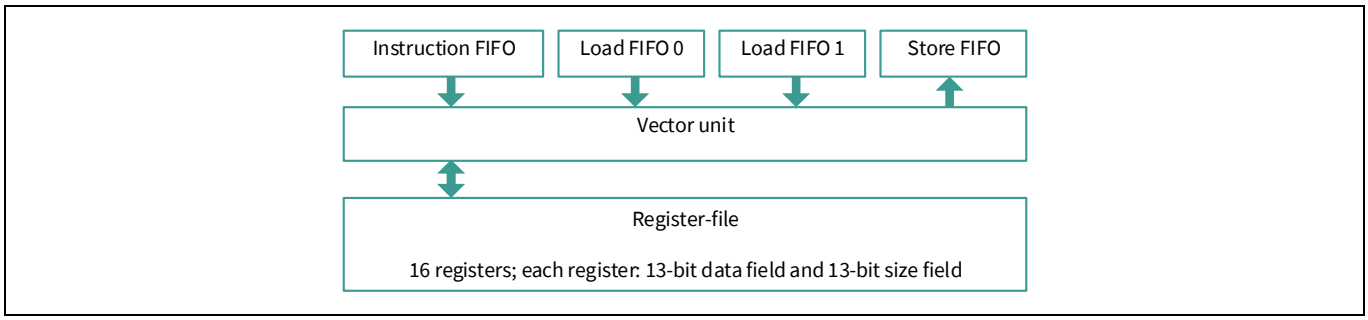


Figure 13-16. Vector unit

The VU addresses asymmetric key cryptography. Asymmetric key cryptography includes RSA, Diffie-Hellman key exchange, digital signature authentication, and elliptic curve cryptography (ECC). These algorithms share the requirement to efficiently perform computations in a Galois field on large integers of up to 1000's of bits.

The VU performs instructions on operand data.

- The VU instructions are provided by the instruction FIFO
- The VU operand data is specified by any of the following (all encoded as part of the 32-bit instruction word):
  - instruction operation code
  - instruction register indices
  - instruction immediate values

The VU uses the VU register file. A VU register is a (data, size) pair: a 13-bit data field (bits 28 down to 16) and a 13-bit size field (bits 12 down to 0). The data field is either used as:

- Instruction operand data: the complete 13-bit data is used
- An offset into memory: the 13-bit data value is a word offset wrt. a base address into memory

The size field is only used when the data field is used as an offset into memory. In this case, the size field specifies the bit size (minus 1) of the memory operand (the 13-bit field specifies a size in the range of [1, 8192] bits).

Cryptographic function block (CRYPTO)

The VU is a “trimmed down” application domain specific CPU:

- “Trimmed down”. It does not have a CPU instruction fetch unit that supports non-sequential program flow. This functionality is typically not required for asymmetric key cryptography functions or needs to be provided by the external AHB-Lite bus master (typically a CPU). This design decision simplifies the VU.
- Note that the instruction FIFO provides the mechanism by which instructions are provided. This mechanism only supports a sequential VU program flow.
- Application domain specific. The VU instructions and operand data are tuned for asymmetric key cryptography. For example, instructions are provided for arithmetic over binary extension fields (GF(2m)) or prime fields (GF(p)), and memory operands allow for a large operand length of up to 8192 bits (although the VU internal data path is limited to 32 bit)
- CPU. The VU has CPU-like design components, such as an instruction decoder, a register file, and data path with multiple functional units

The VU architecture is the result of a tradeoff between silicon area, design complexity, performance efficiency, and algorithmic flexibility. Note that most asymmetric key algorithms can be performed by the regular CPU. However, this typically comes at a lower performance level, possibly large code footprint (due to loop unrolling of code to improve performance when operating on large operand data), and possibly large data footprint (due to lookup tables to improve performance).

The improved performance of the VU (over a regular CPU) for asymmetric key algorithms also allows for more generic algorithmic implementations, as opposed to implementations targeting, such as specific primes or irreducible polynomials.

Before describing the architecture, a short C example is illustrates the VU functionality.

```
#define SIZE 4096
void Example ()
{
int a = 0; int b = 1; int c = 2; // assign register indices
uint8_t c_data[SIZE/8];
ALLOC_MEM (a, SIZE); // allocate 4096 bits of data
ALLOC_MEM (b, SIZE); // allocate 4096 bits of data
ALLOC_MEM (c, SIZE); // allocate 4096 bits of data
Crypto_WriteMemNumber (a, “0x21542555:fed35532: … :ffea2345”);
Crypto_writeMemNumber (b, “0xef45ac2a:34a312bc: … :000003ab”);
ADD (c, a, b);
Crypto_ReadMemNumber (c, c_data);
}
```

The example adds two 4096-bit numbers and produces a 4096-bit number. The example is explained as follows:

- Three local variables are assigned VU register indices:
  - local variable a uses register 0
  - local variable b uses register 1
  - local variable c used register 2
- Memory is allocated in the memory buffer for each variable: each variable needs 4096 bits
- A function “Crypto_WriteMemNumber” is called to initialize the memory operand data for source variables a and b
- A VU ADD instruction (long integer addition) is executed
- A function “Crypto_WriteMemNumber” is called to read the result of the instruction from the memory

The example illustrates that the full expressive power of the C language is available when writing software for the VU.

The example illustrates that relatively complex functionality can be expressed in only a few lines of C code (consider implementing the same functionality on a 32-bit Arm® processor).

The “Crypto_” functions copy data to and from the memory. These functions are executed on the regular CPU and may take a significant number of cycles. Asymmetric key algorithm typically requires little copy functions and a lot of VU instructions. As a result, the cycle overhead of the copy functions is negligible. As an example, RSA requires exponentiation of a number by a second number modulo a third number. This RSA functionality requires only four copy functions (three to initialize three memory operands and one to read the result), but requires thousands of VU instructions.

The following sections provide an architectural overview of the VU.

13.10.1 VU register file

The register file has sixteen registers (r0 through r15). Each register consists of a 13-bit data field and a 13-bit size field. The data field is either used as:

- Instruction operand data: the 13-bit data is used
- An offset into memory: the 13-bit data value is a word offset for a base address into memory. Typically, memory operand data is located in the crypto memory buffer (this provides better latency/performance than memory operand data in system memory).

The size field is used only when the data field is used as an offset into memory. In this case, the size field specifies the bit size (minus 1) of the memory operand (the 13-bit field specifies a size in the range of [1, 8192] bits).

The CPU software decides what instructions use what registers. There is no compiler with specific conventions for register usage or to perform register allocation. To ensure some consistency and interoperability of the functions in a software library, the software programmer should introduce its own conventions. Some examples of conventions are:

- Each non-leaf function saves all registers to the stack on function entry and restores all registers from the stack on function exit. This makes all registers available for use within the function.
- Each non-leaf function restricts its register use to registers r4 through r14
- Each leaf function either restricts its register use to registers r0 through r3 or follows the same save/restore convention as non-leaf functions

The register-file register r15 serves a specific purpose. Similar to the Arm® architecture, register r15 is used as a stack pointer.

Figure 13-17 illustrates the register file and how the data fields are used as offsets in a 16 KB memory region (either in the crypto memory buffer or the system memory). The base address of the memory region is provided by VU_CTL.ADDR[31:8] and the memory region size is specified by VU_CTL.MASK[].

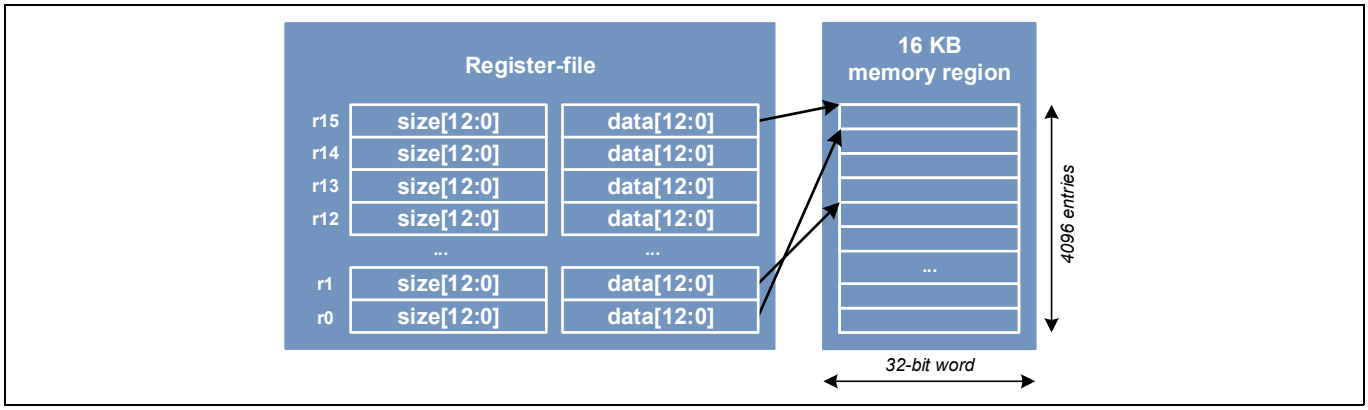


Figure 13-17. Register file

13.10.2 Stack

The VU stack resides in the memory region. Register r15 is used as a stack pointer. The stack has two purposes:

- It is used to save/restore registers r0 through r14. Each register (data field and size field) uses a single 32-bit word
- The instruction PUSH_REG saves/pushes all registers r0 through r14 on the stack (and register r15 is decremented by 15)
- The instruction POP_REG restores/pops all registers r0 through r14 from the stack (and register r15 is incremented by 15)
- The use of the stack pointer (register r15) is implied by the PUSH_REG and POP_REG instructions.
- It is used to allocate memory operands
- The instruction ALLOC_MEM (rx, size-1) allocates enough 32-bit words to hold a memory operand of "size" bits for register rx. The stack pointer (register r15) is decremented by the number of allocated 32-bit words. The data field of register rx is updated with the new stack pointer value.
- The instruction FREE_MEM ("15 bit pattern") frees the 32-bit words that hold memory operand data associated with the registers identified by the bit pattern. Note that a register's size field specifies the number 32-bit words that hold its associated memory operand. The stack pointer is incremented by the number of freed 32-bit words.
- The use of the stack pointer (register r15) is implied by the instruction. Note that to allocate multiple memory operands, multiple ALLOC_MEM instructions are required. However, to free multiple memory buffer operands, only a single FREE_MEM instruction is required.

Figure 13-18 illustrates the VU register-file state before and after the execution of instructions ALLOC_MEM (r0, 32-1) (32 bits requiring one 32-bit stack element) and ALLOC_MEM (r1, 80-1) (80 bits requiring three 32-bit stack elements).

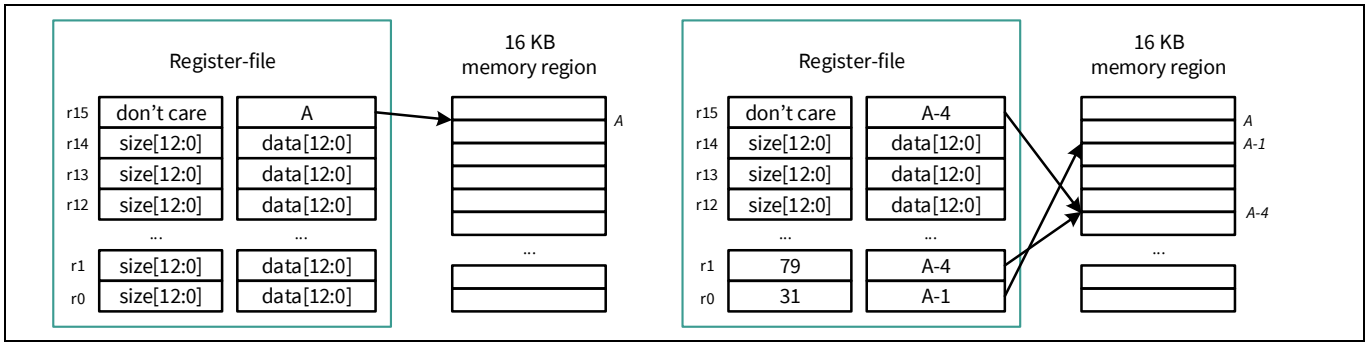


Figure 13-18. Left: Register file state before ALLOC_MEM instructions; right: register file after ALLOC_MEM instructions

Figure 13-19 illustrates the VU component state after the execution of the instruction FREE_MEM ((1 << r1) | (1 << r0)).

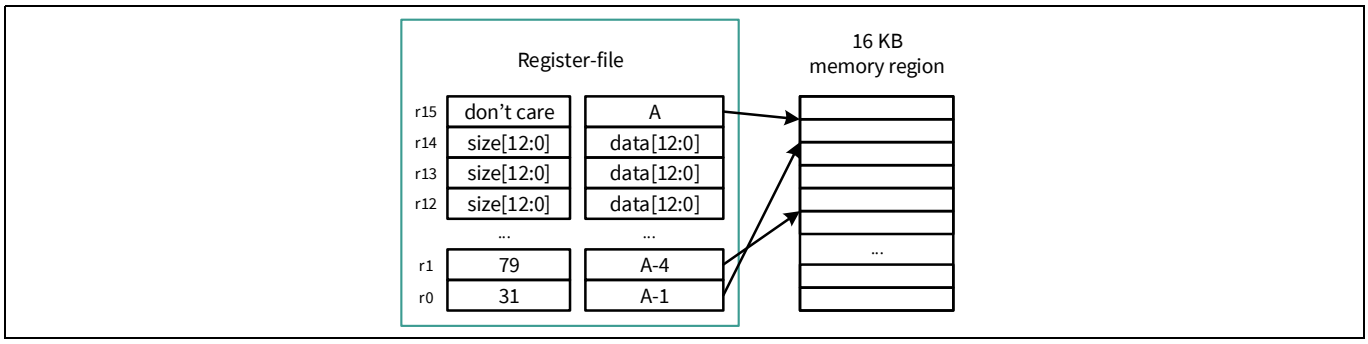


Figure 13-19. Register file state after FREE_MEM instruction

Freeing of stack elements should be in the reverse order of allocation of stack elements. Also see the description of the FREE_MEM instruction for the order in which registers are freed (lower registers are freed before higher registers).

13.10.3 Memory operands

Asymmetric key cryptography requires computations on large integers of up to 1000's of bits. These integers are implemented using memory operands. The instruction registers' data values provide offsets in a memory region. The base address of the memory region is provided by VU_CTL.ADDR[31:8]. The memory region is located in either the IP memory buffer or the system memory. The memory region is accessed through the load and store FIFOs.

Memory operands are typically located in the crypto memory buffer (better access latency). Memory operand require one or multiple 32-bit words. The register size field specifies the bit size of a memory operand. This size is limited to the range [1, 8192] bits. Note that there is no size restriction within this range. A memory operand of n bits requires (n+31)/32 32-bit words.

13.10.4 Datapath

The VU instructions can operate on 13-bit register data values and/or large memory operands. The VU datapath is limited to 32 bits (the width of a single memory word).

To operate on large memory operands, multiple datapath iterations are required. Dependent on the instruction opcode, these iterations may be independent of each other (for example, a OR instruction) or may be dependent of each other (for example, an ADD instruction requires a carry). This complexity is completely hidden from software.

- The instruction opcode specifies whether datapath iterations are dependent or independent
- The register size field provides the VU decoder and datapath with all the information it needs to determine the number of datapath iterations

The VU datapath consists of three functional units:

- An ALU that performs addition, subtraction, and logical instructions
- A barrel shifter that performs shift instructions
- A multiplier that performs multiplication and squaring instructions

In addition, a state machine is present for instruction decoding and controlling multiple datapath iterations. The state machine also performs administration instructions such as PUSH_REG, POP_REG, ALLOC_MEM, and FREE_MEM.

Note that the instruction execution time (in clock cycles) is typically independent of the instruction operand data values. However, the execution time is dependent on the size of memory operands (as specified by the register size field). This is an important characteristic, as the data value independence complicates differential power or execution time attacks.

13.10.5 Status register

Similar to the Arm® architecture, the VU has a STATUS register:

- It is affected by instruction execution
- It controls instruction execution

The STATUS register is a 4-bit field, with the following fields:

- CARRY field (bit 0). This field is set to '1' if the result of an addition exceeds the destination operand size, if the result of a subtraction is positive or zero, or as the result of a shift instruction.
- EVEN field (bit 1). This field is set to '1' (and cleared to '0' otherwise) if the result of an instruction is even (bit 0 of the destination operand is '0').
- ZERO field (bit 2). This field is set to '1' (and cleared to '0' otherwise) if the result of an instruction is "0".
- ONE field (bit 3). This field is set to '1' (and cleared to '0' otherwise) if the result of an instruction is "1".

The STATUS register fields are set and cleared as a result of instruction execution. Not all instructions affect all STATUS register fields.

Conditional execution. As mentioned, the STATUS register controls instruction execution. It does so through conditional instruction execution (again, similar to the Arm® architecture). Almost all instructions (the SET_REG instruction is the exception) support conditional execution. Conditional execution only executes an instruction when a specific condition, as specified by a condition code in the instruction word, is met. The condition code is dependent on the STATUS register fields. The following table lists the condition codes and the conditions under which the conditional instruction is executed.

Table 13-28. Condition codes

Abbreviation	Condition code value	Description	Condition
ALWAYS	0x0	Always	'1'
EQ	0x1	Equal	ZERO
NE	0x2	Not equal	!ZERO
CS	0x3	Carry set/higher or same	CARRY
CC	0x4	Carry clear/lower	!CARRY
HI	0x5	Higher	CARRY & !ZERO
LS	0x6	Lower or same	!CARRY | ZERO
EVEN	0x7	Even	EVEN
ODD	0x8	Odd	!EVEN
ONE	0x9	One	ONE
NOT_ONE	0xa	Not one	!ONE

If a conditional instruction has a condition code that evaluates to '0'/FALSE, it does not change the VU functional state. This means that:

- a destination operand is not updated
- the STATUS register is not updated

13.10.6 Instructions

The VU instructions operate on either register operand data or memory operand data.

- **Operand types.** Typically, VU instruction operands all have the same type: either register or memory operands. However, some exceptions do exist. For example, consider the following CTSAME (count trailing same) instruction.

```
CTSAME (r0, r1, r2)
```

  This instruction counts the number of trailing bits (least significant bits), which are the same as the two memory operands that are identified by register r1 and r2. The instruction result is stored in the data field of register r0. This instruction has two source operands of the memory type and one destination operand of the register type.

- **Conditional execution.** Conditional execution makes the execution of an instruction dependent on a condition. For example, consider the COND_CTSAME (HI, r0, r1, r2) instruction. This instruction uses the “HI” condition code. The CTSAME instruction is only executed when the “HI” condition code evaluates to ‘1’/TRUE. This is the case when the CARRY field of the STATUS register is ‘1’/TRUE and the ZERO field of the STATUS register is ‘0’/FALSE.
  Unconditional execution uses the “ALWAYS” condition code, which always evaluates to ‘1’/TRUE. For brevity, and unconditional instruction XXX is not written as COND_XXX, but simply as XXX.
  Note that a not executed conditional instruction does not update the STATUS register.
- **Memory operand data extension.** Register operand data always has the same 13-bit size. Memory operand data has a size that is specified by a register’s size field. Therefore, it is possible to have memory operands with different sizes. This is intentional, but requires some rules in terms of how to deal with instructions that operate on memory operands with different sizes. These rules are as follows:
  - If an instruction has a destination memory operand (for example, an ADD instruction), the size of this destination operand specifies the size of the instruction execution
  - If an instruction has no destination memory operand (for example, a CTSAME instruction), the maximum size of all source memory operands specifies the size of the instruction
  - If the instruction size (as determined by the previous two rules) is larger than a source memory operand, the memory operand is extended with leading ‘0’ bits (most significant ‘0’ bits)
  - If the instruction size is smaller than a source memory operand, the memory operand’s leading bits (most significant bits) are dropped/ignored (the LSR, LSR1, and LSR1_WITH_CARRY are an exception to this rule)

  To illustrate memory operand extension, we give a few examples.

```
CTSAME (r0, r1, r2)
r1 specifies a 40-bit memory operand 0x11:12345678
r2 specifies a 64-bit memory operand 0x11111111:ffffff78
```

  The instruction size is 64 bits (maximum size of the two source memory operands). The memory operand 0x11:12345678 is extended to 0x00000011:12345678. The CTSAME instruction produces the result 8 in the register r0 data field.

```
ADD (r0, r1, r2)
r0 specifies a 48-bit memory operand
r1 specifies a 40-bit memory operand 0x11:12345678
r2 specifies a 64-bit memory operand 0xffffffff:ffffff78
```

  The instruction size is 48 bits (size of the destination memory operand). The memory operand 0x11:12345678 is extended to 0x0011:12345678. The memory operand 0xffffffff:ffffff78 is reduced to 0xffff:ffffff78. The ADD instruction produces the 40-bit result 0x0011:123455f0 in the destination memory operand. Note that this instruction generates a carry and will set the CARRY field of the STATUS register to ‘1’/TRUE.

13.10.7 Instruction set

This section describes all VU instructions. For each instruction, the following is described:

- The instruction format
  - Source operand types (register or memory operands)
  - Support for conditional execution
  - Encoding of the 32-bit instruction word (IW[31:0])
- Mnemonic
- Operation code
- Functionality

For instructions with memory operands, the source and destination memory operands may overlap, unless otherwise mentioned (the USQUARE, XSQUARE, UMUL, and XMUL instructions do not allow memory operand overlap).

Table 13-29. ALLOC_MEM instruction

Instruction format	**Mnemonic (rdst, imm13[12:0])** **rdst: register operand** **imm13: 13-bit immediate date**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[19:16] = rdst IW[12:0] = imm13	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
ALLOC_MEM	0x12	r15.data[12:0] = r15_data[12:0] – (imm13 >> 5) - 1; rdst.data[12:0] = r15_data[12:0]; rdst.size[12:0] = imm13; // bit size minus ‘1’

Table 13-30. Instructions with memory operands, category III

Instruction format	**Mnemonic (imm16[15:0])** **imm16: 16-bit immediate date**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:0] = imm16	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
FREE_MEM	0x13	for (idx = 0; idx <= 15; idx++) { if (imm16[idx]) { imm13 = r[idx].size[12:0] r15.data[12:0] = r15_data[12:0] + (imm13 >> 5) + 1; } }

Table 13-31. Instructions with Register Operand Only, Category I

Instruction format	**Mnemonic (rdst, imm13_0[13:0], imm13_1[11:0])** **rdst: register operand** **imm13_0: 13 bit immediate (for register data field)** **imm13_1: 13-bit immediate (for register size field)**	
Encoding	IW[31:30] = “operation code” IW[29:26] = rdst IW[25:13] = imm13_1 IW[12:0] = imm13_0	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
SET_REG	0x2	rdst.data[12:0] = imm13_1; rdst.size[12:0] = imm13_0; // bit size minus ‘1’

Table 13-32. MOV_REG instruction

Instruction format	**Mnemonic (rdst, rsrc) or COND_Mnemonic (cc, rdst, rsrc)** **rdst: register operand** **rsrc: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[3:0] = rsrc	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
MOV_REG	0x02	rdst.data = rsrc.data; rdst.size = rsrc.size;

Table 13-33. LD_REG Instruction

Instruction format	**Mnemonic (rsrc1, rsrc0) or COND_Mnemonic (cc, rsrc1, rsrc0)** **rsrc1: register operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
LD_REG	0x00	word = GetMemData (r15.data+rsrc0, 32-1); // r15: stack pointer rsrc1.data[12:0] = word >> 16; rsrc1.size[12:0] = word;

Table 13-34. ST_REG instruction

Instruction format	**Mnemonic (rsrc1, rsrc0) or COND_Mnemonic (cc, rsrc1, rsrc0)** **rsrc1: register operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
ST_REG	0x01	word = rsrc1.data[12:0] << 16 | rsrc1.size[12:0] SetMemBuffData (r15.data+rsrc0, word, 32-1);

Table 13-35. Instructions with register operand only, category III

Instruction format	**Mnemonic (rsrc1, rsrc0) or COND_Mnemonic (cc, rsrc1, rsrc0)** **rsrc0: register operand** **rsrc1: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
SWAP_REG	0x03	data = rsrc1.data; size = rsrc1.size; rsrc1.data = rsrc0.data; rsrc1.size = rsrc0.size; rsrc0.data = data; rsrc0.size = size;

Table 13-36. Instructions with register operand only, category IV

Instruction format	**Mnemonic () or COND_Mnemonic (cc)** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
PUSH_REG	0x10	r15.data -= 15; // r15: stack pointer word = r0.data[12:0] << 16 | r0.size[12:0]; MemBuff[r15.data+0] = word; word = r1.data[12:0] << 16 | r1.size[12:0]; MemBuff[r15.data+1] = word; … word = r14.data[12:0] << 16 | r14.size[12:0]; MemBuff[r15.data+14] = word;
POP_REG	0x11	word = GetMemData (r15.data+0, 32-1); r0.data = word >> 16; r0.size = word; word = MemBuff[r15.data+1]; r1.data[12:0] = word >> 16; r1.size[12:0] = word; … word = MemBuff[r15.data+14]; r14.data[12:0] = word >> 16; r14.size[12:0] = word; r15.data += 15;

Table 13-37. Instructions with Register Operand Only, Category V

Instruction format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: register operand** **rsrc1: register operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
ADD_REG	0x06	rdst.data[12:0] = rsrc1.data[12:0] + rsrc0.data[12:0];
SUB_REG	0x07	rdst.data[12:0] = rsrc1.data[12:0] – rsrc0.data[12:0];
OR_REG	0x08	rdst.data[12:0] = rsrc1.data[12:0] | rsrc0.data[12:0];
AND_REG	0x09	rdst.data[12:0] = rsrc1.data[12:0] & rsrc0.data[12:0];
XOR_REG	0x0a	rdst.data[12:0] = rsrc1.data[12:0] ^ rsrc0.data[12:0];
NOR_REG	0x0b	rdst.data[12:0] = ~(rsrc1.data[12:0] | rsrc0.data[12:0]);
NAND_REG	0x0c	rdst.data[12:0] = ~(rsrc1.data[12:0] & rsrc0.data[12:0]);
MIN_REG	0x0d	rdst.data[12:0] = Minimum (rsrc1.data[12:0], rsrc0.data[12:0]);
MAX_REG	0x0e	rdst.data[12:0] = Maximum (rsrc1.data[12:0], rsrc0.data[12:0]);

Table 13-38. MOV_REG_TO_STATUS

Instruction format	**Mnemonic (rsrc) or COND_Mnemonic (cc, rsrc)** **rsrc: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[3:0] = rsrc	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
MOV_REG_TO_STATUS	0x04	STATUS.CARRY = rsrc.data[0]; STATUS.EVEN = rsrc.data[1]; STATUS.ZERO = rsrc.data[2]; STATUS.ONE = rsrc.data[3];

Table 13-39. MOV_STATUS_TO_REG

Instruction format	**Mnemonic (rdst) or COND_Mnemonic (cc, rdst)** **rdst: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
MOV_STATUS_TO_REG	0x05	rdst.data= 0; rdst.data[0] = STATUS.CARRY; rdst.data[1] = STATUS.EVEN; rdst.data[2] = STATUS.ZERO; rdst.data[3] = STATUS.ONE;

Table 13-40. MOV_IMM_TO_STATUS

Instruction format	**Mnemonic (imm4) or COND_Mnemonic (cc, imm4)** **imm4: immediate value** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[3:0] = imm4	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
MOV_IMM_TO_STATUS	0x01	STATUS.CARRY = imm[0]; STATUS.EVEN = imm[1]; STATUS.ZERO = imm[2]; STATUS.ONE = imm[3];

Table 13-41. Instructions with mixed operands, category I

Instruction format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: memory buffer operand** **rsrc1: memory buffer operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Shared Functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
LSL	0x20	src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_data = SizeAdjust (src1_data, rdst.size); dst_data = src1_data << rsrc0.data[12:0]; if (rsrc0.data[12:0] != 0) STATUS.CARRY = dst_data[rdst.size +1]; else STATUS.CARRY = 0 SetMemBuffData (rdst.data, dst_data, rdst.size);
LSR	0x23	src1_data = GetMemData (rsrc1.data, rsrc1.size); dst_data = src1_data >> rsrc0.data[12:0]; if (rsrc0.data[12:0] != 0) STATUS.CARRY = dst_data[-1]; else STATUS.CARRY = 0; SetMemBuffData (rdst.data, dst_data, rdst.size);

Table 13-42. Instructions with mixed operands, category II

Instruction format	**Mnemonic (rdst, rsrc) or COND_Mnemonic (cc, rdst, rsrc)** **rdst: memory buffer operand** **rsrc: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc	
Shared functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
LSL1	0x21	src_data = GetMemData (rsrc.data, rsrc.size); src_data = SizeAdjust (src_data, rdst.size); dst_data = (src_data << 1); STATUS.CARRY = dst_data[rdst.size +1]; SetMemBuffData (rdst.data, dst_data, rdst.size);
LSL1_WITH_CARRY	0x22	src_data = GetMemData (rsrc.data, rsrc.size); src_data = SizeAdjust (src_data, rdst.size); dst_data = (src_data << 1) | STATUS.CARRY; STATUS.CARRY = dst_data[rdst.size +1]; SetMemBuffData (rdst.data, dst_data, rdst.size);
LSR1	0x24	src_data = GetMemData (rsrc.data, rsrc.size); dst_data = src_data >> 1; STATUS.CARRY = dst_data[-1]; SetMemBuffData (rdst.data, dst_data, rdst.size);
LSR1_WITH_CARRY	0x25	src_data = GetMemData (rsrc.data, rsrc.size); dst_data = src_data >> 1; dst_data[rdst.size] = STATUS.CARRY; STATUS.CARRY = dst_data[-1]; SetMemBuffData (rdst.data, dst_data, rdst.size);

Table 13-43. Instructions with mixed operands, category III

Instruction format	**Mnemonic (rdst, rsrc) or COND_Mnemonic (cc, rdst, rsrc)** **rdst: memory buffer operand** **rsrc: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[3:0] = rsrc	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
SET_BIT	0x28	dst_data = GetMemData (rdst.data, rdst.size); dst_data[rsrc.data[12:0]] = 1; SetMemBuffData (rdst.data, dst_data, rdst.size);
CLR_BIT	0x29	dst_data = GetMemData (rdst.data, rdst.size); dst_data[rsrc.data[12:0]] = 0; SetMemBuffData (rdst.data, dst_data, rdst.size);
INV_BIT	0x2a	dst_data = GetMemData (rdst.data, rdst.size); dst_data[rsrc.data[12:0]] = !dst_data[rsrc.data[12:0]]; SetMemBuffData (rdst.data, dst_data, rdst.size);

Table 13-44. Instructions with mixed operands, category IV

Instruction format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: register operand** **rsrc1: memory buffer operand** **rsrc0: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
GET_BIT	0x2b	src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_bit = src1_data[rsrc0.data[12:0]]; rdst.data = src1_bit; STATUS.CARRY = src1_bit;

Table 13-45. Instructions with mixed operands, category V

Instruction format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: register operand** **rsrc1: memory buffer operand** **rsrc0: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Shared functionality	STATUS.ZERO = (src0_data[instr_size:0] == src1_data[instr_size:0]) Note that for 8192-bit operands, two equal operands result in rdst.data[12:0] = “0”. The ZERO cause field can be used to identify this situation.	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
CLSAME	0x26	instr_size = Maximum (rsrc1.size, rsrc0.size); src0_data = GetMemData (rsrc0.data, rsrc0.size); src0_data = SizeAdjust (src0_data, instr_size); src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_data = SizeAdjust (src1_data, instr_size); temp_data = src1_data ^ src0_data; for (idx = instr_size-1; idx >= 0; idx--) { if (temp_data[idx] == 1) break; } rdst.data[12:0] = instr_size - idx - 1;
CTSAME	0x27	instr_size = Maximum (rsrc1.size, rsrc0.size); src0_data = GetMemData (rsrc0.data, rsrc0.size); src0_data = SizeAdjust (src0_data, instr_size); src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_data = SizeAdjust (src1_data, instr_size); temp_data = src1_data ^ src0_data; for (idx = 0; idx < instr_size; idx++) { if (temp_data[idx] == 1) break; } rdst.data[12:0] = idx;

Table 13-46. Instructions with memory operands, category I

Instruction format	**Mnemonic (rdst) or COND_Mnemonic (cc, rdst)** **rdst: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst	
Shared functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
SET_TO_ZERO	0x34	instr_size = rdst.size; dst_data = 0; SetMemBuffData (rdst.data, dst_data, instr_size);
SET_TO_ONE	0x35	instr_size = rdst.size; dst_data = 1; SetMemBuffData (rdst.data, dst_data, instr_size);

Table 13-47. Instructions with memory operands, category II

Instruction format	**Mnemonic (rdst, rsrc) or COND_Mnemonic (cc, rdst, rsrc)** **rdst: memory buffer operand** **rsrc: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[3:0] = rsrc	
Shared functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
MOV	0x30	instr_size = rdst.size; src_data = GetMemData (rsrc.data, rsrc.size); src_data = SizeAdjust (src_data, instr_size); dst_data = src_data; SetMemBuffData (rdst.data, dst_data, instr_size);
XSQUARE	0x31	instr_size = rdst.size; src_data = GetMemData (rsrc.data, rsrc.size); src_data = SizeAdjust (src_data, instr_size); dst_data = PolynomialMultiplication (src_data, src_data); SetMemBuffData (rdst.data, dst_data, instr_size); *Note: The source and destination memory operands must not overlap.*
USQUARE	0x2f	instr_size = rdst.size; src_data = GetMemData (rsrc.data, rsrc.size); src_data = SizeAdjust (src_data, instr_size); dst_data = src_data * src_data; SetMemBuffData (rdst.data, dst_data, instr_size); *Note: The source and destination memory operands must not overlap.*

Table 13-48. Instructions with memory operands, category III

Instruction format	**Mnemonic (rsrc1, rsrc0) or COND_Mnemonic (cc, rsrc1, rsrc0)** **rsrc1: memory buffer operand** **rsrc0: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
CMP_SUB	0x3d	instr_size = Maximum (rsrc1.size, rsrc0.size); src0_data = GetMemData (rsrc0.data, rsrc0.size); src0_data = SizeAdjust (src0_data, instr_size); src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_data = SizeAdjust (src1_data, instr_size); dst_data = src1_data – src0_data; STATUS.CARRY = (src1_data >= src0_data); STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)
CMP_DEGREE	0x3e	instr_size = Maximum (rsrc1.size, rsrc0.size); src0_data = GetMemData (rsrc0.data, rsrc0.size); src0_data = SizeAdjust (src0_data, instr_size); src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_data = SizeAdjust (src1_data, instr_size); STATUS.CARRY = PolynomialDegree (src1_data) >= PolynomialDegree (src0_data); STATUS.ZERO = PolynomialDegree (src1_data) == PolynomialDegree (src0_data);

Table 13-49. Instructions with memory operands, category IV

Instruction format	**Mnemonic (rsrc) or COND_Mnemonic (cc, rsrc)** **rsrc: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[3:0] = rsrc	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
TST	0x3f	instr_size = rsrc.size; src_data = GetMemData (rsrc.data, rsrc.size); dst_data = src_data; STATUS.CARRY = 0; // always set to ‘0’!!! STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)

Table 13-50. Instructions with memory operands, category V

Instruction format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: memory buffer operand** **rsrc1: memory buffer operand** **rsrc0: memory buffer operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[15:12] = rdst IW[7:4] = rsrc1 IW[3:0] = rsrc0	
Shared functionality	STATUS.EVEN = (dst_data[0] == 0) STATUS.ZERO = (dst_data[instr_size:0] == 0) STATUS.ONE = (dst_data[instr_size:0] == 1)	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
XMUL	0x32	instr_size = rdst.size; src0_data = GetMemData (rsrc0.data, rsrc0.size); src0_data = SizeAdjust (src0_data, instr_size); src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_data = SizeAdjust (src1_data, instr_size); dst_data = PolynomialMultiplication (src1_data, src0_data); SetMemBuffData (rdst.data, dst_data, instr_size); **Note:** The source and destination memory operands must not overlap. When one of the operands is 32 or less bits in size, performance is best when rsrc0 is used for this operand.

Table 13-50. Instructions with memory operands, category V (continued)

Instruction format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: memory buffer operand** **rsrc1: memory buffer operand** **rsrc0: memory buffer operand** **cc: condition code**	
UMUL	0x33	`instr_size = rdst.size;` `src0_data = GetMemData (rsrc0.data, rsrc0.size);` `src0_data = SizeAdjust (src0_data, instr_size);` `src1_data = GetMemData (rsrc1.data, rsrc1.size);` `src1_data = SizeAdjust (src1_data, instr_size);` `dst_data = src1_data * src0_data;` `SetMemBuffData (rdst.data, dst_data, instr_size);` *Note: The source and destination memory operands must not overlap.* When one of the operands is 32 or less bits in size, performance is best when rsrc0 is used for this operand.
ADD	0x36	`instr_size = rdst.size;` `src0_data = GetMemData (rsrc0.data, rsrc0.size);` `src0_data = SizeAdjust (src0_data, instr_size);` `src1_data = GetMemData (rsrc1.data, rsrc1.size);` `src1_data = SizeAdjust (src1_data, instr_size);` `dst_data = src1_data + src0_data;` `STATUS.CARRY = (dst_data >= (1 << instr_size));` `SetMemBuffData (rdst.data, dst_data, instr_size);`
SUB	0x37	`instr_size = rdst.size;` `src0_data = GetMemData (rsrc0.data, rsrc0.size);` `src0_data = SizeAdjust (src0_data, instr_size);` `src1_data = GetMemData (rsrc1.data, rsrc1.size);` `src1_data = SizeAdjust (src1_data, instr_size);` `dst_data = src1_data - src0_data;` `STATUS.CARRY = (src1_data >= src0_data);` `SetMemBuffData (rdst.data, dst_data, instr_size);`

Table 13-50. Instructions with memory operands, category V (continued)

Instruction format	**Mnemonic (rdst, rsrc1, rsrc0) or COND_Mnemonic (cc, rdst, rsrc1, rsrc0)** **rdst: memory buffer operand** **rsrc1: memory buffer operand** **rsrc0: memory buffer operand** **cc: condition code**	
ADD_WITH_CARRY	0x14	instr_size = rdst.size; src0_data = GetMemData (rsrc0.data, rsrc0.size); src0_data = SizeAdjust (src0_data, instr_size); src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_data = SizeAdjust (src1_data, instr_size); dst_data = src1_data + src0_data + STATUS.CARRY; STATUS.CARRY = (dst_data >= (1 << instr_size)); SetMemBuffData (rdst.data, dst_data, instr_size);
SUB_WITH_CARRY	0x15	instr_size = rdst.size; src0_data = GetMemData (rsrc0.data, rsrc0.size); src0_data = SizeAdjust (src0_data, instr_size); src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_data = SizeAdjust (src1_data, instr_size); dst_data = src1_data - src0_data – !STATUS.CARRY; STATUS.CARRY = (src1_data >= src0_data); SetMemBuffData (rdst.data, dst_data, instr_size);
OR, AND, XOR, NOR, NAND	0x38, 0x39, 0x3a, 0x3b, 0x3c	instr_size = rdst.size; src0_data = GetMemData (rsrc0.data, rsrc0.size); src0_data = SizeAdjust (src0_data, instr_size); src1_data = GetMemData (rsrc1.data, rsrc1.size); src1_data = SizeAdjust (src1_data, instr_size); OR: dst_data = src1_data | src0_data; AND: dst_data = src1_data & src0_data; XOR: dst_data = src1_data ^ src0_data; NOR: dst_data = ~(src1_data | src0_data); NAND: dst_data = ~(src1_data & src0_data); SetMemBuffData (rdst.data, dst_data, instr_size);

Table 13-51. Instructions with memory operands, category VI

Instruction format	**Mnemonic (rdst, imm12) or COND_Mnemonic (cc, rdst, imm12)** **rdst: memory buffer operand** **rsrc: register operand** **cc: condition code**	
Encoding	IW[31:24] = “operation code” IW[23:20] = cc IW[19:16] = rdst IW[12:0] = imm13	
Mnemonic	**Operation code**	**Functionality (if “cc” evaluates to ‘1’/TRUE)**
SET_BIT_IMM	0x2c	dst_data = GetMemData (rdst.data, rdst.size); dst_data[imm13] = 1; SetMemBuffData (rdst.data, dst_data, rdst.size);
CLR_BIT_IMM	0x2d	dst_data = GetMemData (rdst.data, rdst.size); dst_data[imm13] = 0; SetMemBuffData (rdst.data, dst_data, rdst.size);
INV_BIT_IMM	0x2e	dst_data = GetMemData (rdst.data, rdst.size); dst_data[imm13] = !dst_data[imm13]; SetMemBuffData (rdst.data, dst_data, rdst.size);

14 Nonvolatile memory

Nonvolatile memory refers to the flash and SROM memory in the EZ-USB™ FX10 MCU. This chapter explains the geometry and capability of the flash memory. It also lists the SROM API functions that are used to program the flash memory.

14.1 Flash memory

The EZ-USB™ FX10 flash offers high bulk program performance and supports ultra-low-power operation. Flash is typically used to store CPU instructions and data when the device power is off. Flash may be written, but the process is much slower and more restrictive than for SRAM.

14.1.1 Features

This section lists the features of EZ-USB™ FX10 flash.

- 512-byte row size; minimum programmable unit
- Supports the Read While Write (RWW) feature with a RWW sector size of 256 KB
- 10-year retention
- Endurance of 100k program cycles

14.1.2 Configuration

14.1.2.1 Block diagram

Flash is part of the CPU subsystem. The Cortex®-M4 and Cortex®-M0+, as well as other bus masters, can access flash via the AHB.

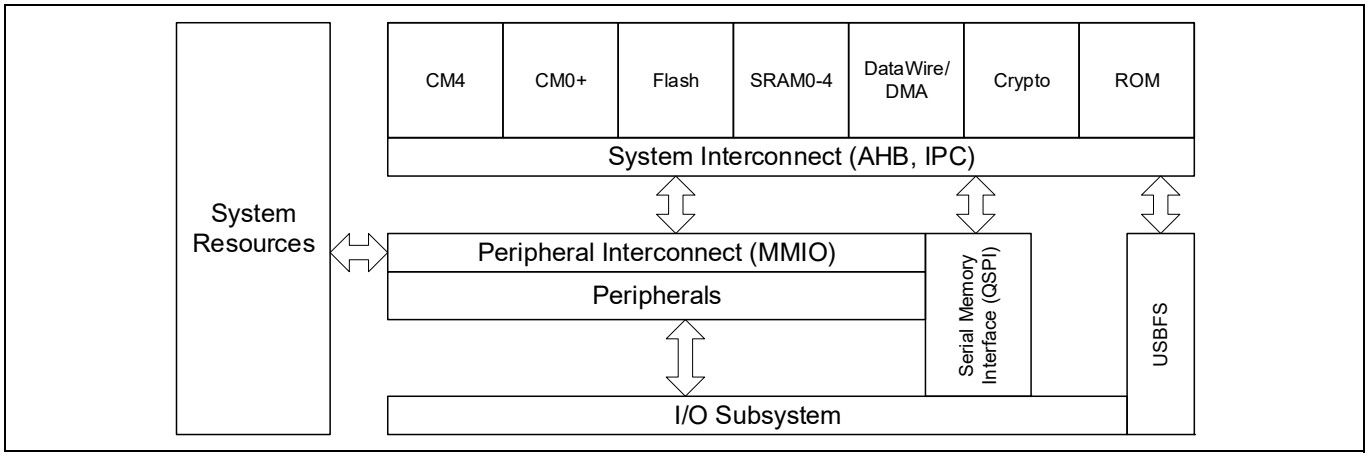


Figure 14-1. Block diagram

14.1.3 Flash geometry

The flash is divided into three regions: the application region, and the supervisory flash region (SFlash), and the EE emulation flash region. The flash has an Erase Disturb mechanism in which writes to rows affect the endurance of other rows in the same sector. It is recommended that the EE emulation flash region is used for frequently-updated data. For data that changes infrequently or code images, the application flash region can be used.

The SFlash region is used to store trim parameters, system configuration parameters, protection and security settings, boot code, and other Infineon proprietary information. Read access to this region is permitted, but program/erase access is limited. The application region is used to store code images or data.

Each region divides into sectors and pages. The sector is the largest division of the region and consists of a number of 512 byte rows.

Table 14-1. Flash geometry

Application flash						SFlash and EE emulation flash				
Read width	KB	Sectors	KB/ sector	Rows/ sector	KB/row	KB	Sectors	KB/ sector	Rows/ sector	KB/ row
128	512	2	256	512	0.5	64	2	32	64	0.5

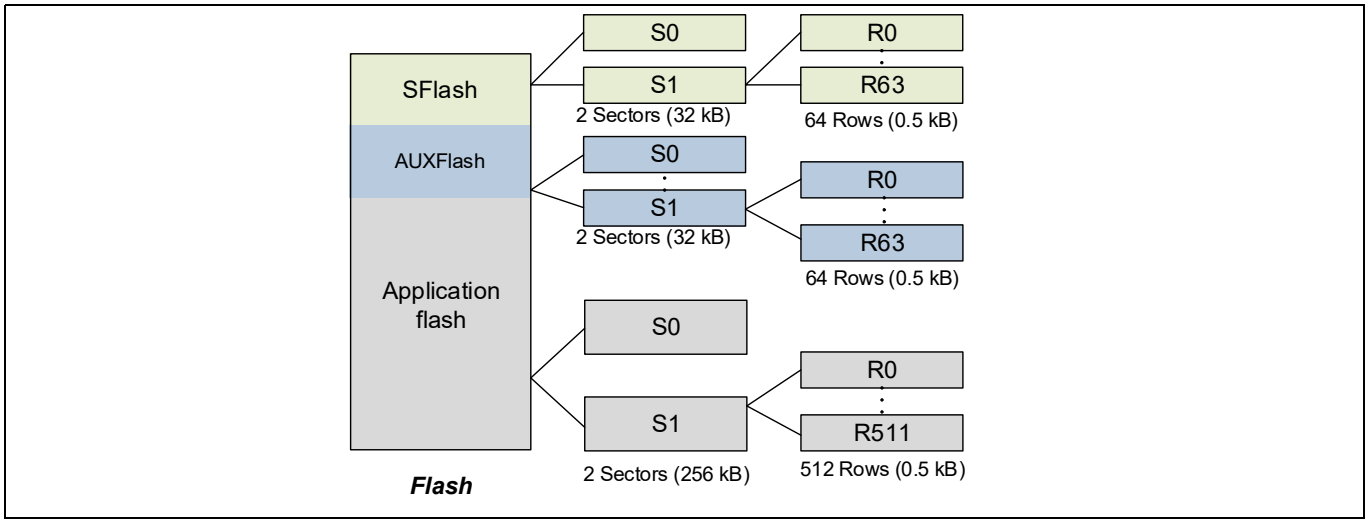


Figure 14-2. Flash geometry organization

14.1.4 Flash controller

Access to the flash memory is enabled through the flash controller. The flash controller interfaces with the AHB-Lite bus and provides flash access for the CM0+, CM4, Crypto, DataWire, and debug. The flash controller generates a bus error if:

- A flash read access to a sector that is currently being programmed/erased
- A read access to a memory hole in the flash memory region. A memory hole is defined as a location that is not occupied by the application flash, EE Emulation flash, or SFlash. Note that the EE Emulation flash space and SFlash space can be non-powers of 2 for some flash macros

The flash controller also provides the registers which support configuration of flash accesses.

Table 14-2. Flash controller retention registers

Register name	Description
FLASHC_FLASH_CTL	Flash controller control register
FLASHC_FLASH_PWR_CTL	Flash enable control
FLASHC_FLASH_CMD	Flash commands for cache/buffer invalidation
FLASHC_CM0/CM4_STATUS	CM0/4 interface status
FLASHC_CM0/CM4_CA_CTL	CM0/4 cache control
FLASHC_*Peripheral*_BUFF_CTL	Buffer control register where *peripheral* may be Crypto, DMA, DWx (DataWire x = 0, 1), or EXT_MSx (External master; x = 0, 1)

14.1.4.1 Wait state count

FLASHC_FLASH_CTL supports configuration of flash wait cycles. If clk_hf is greater than the maximum operating frequency of the flash memory, it is necessary to insert wait cycles when accessing the flash memory by setting the appropriate value in the FLASHC_FLASH_CTL.MAIN_WS register. Set the wait cycles as follows.

HF c	FLASH_CTL.MAIN_WS[3:0]
clk_hf ≤ 33 MHz	0
33 < clk_hf ≤ 66 MHz	1
66 < clk_hf ≤ 99 MHz	2
99 < clk_hf ≤ 133 MHz	3
133 < clk_hf ≤ 150 MHz	4

14.1.4.2 Power modes

FLASHC_FLASH_PWR_CTL provides enable bits for the flash memory. Software can turn off all regions of the flash memory by setting the ENABLE and ENABLE_HV fields to 0.

The wakeup time of flash memory is 10 µs. Software should wait at least 10 µs before reading from flash after it is re-enabled through the FLASHC_FLASH_PWR_CTL register.

The flash controller provides functionality down to the Deep Sleep power mode. In the Deep Sleep power mode, the following flash controller information is retained:

- The retention MMIO registers (listed in **Table 14-2**)
- The cache data structure

Note that buffer information (in the AHB-Lite buffer interfaces and in the synchronization logic) is not retained. Losing buffer information after Deep Sleep transition has limited performance impact.

14.1.4.3 CPU caches

The flash controller provides 8 kB caches for both the CM0+ and CM4 CPUs. Each cache is a four-way set associative with a least recently used (LRU) replacement scheme. Four-way set associativity means that each cache has four ways per set, with each way containing a valid bit, tag, and data. The cache looks at all of the ways in a selected set, checking for validity and a matching tag. These caches can be enabled/disabled through the FLASHC_CM0/4_CA_CTL.CA_EN registers.

Table 14-3. CM0/4 cache control register values and bit field

Register	Bit field and bit name	Description
CM0_CA_CTL[32:0]	CA_EN[31]	Cache enable 0: Disabled 1: Enabled
CM4_CA_CTL[32:0]	CA_EN[31]	Cache enable 0: Disabled 1: Enabled

The cache supports faster flash memory reads when enabled. On a read transfer “miss”, however, a normal flash controller access will occur.

Cache prefetch. The caches support pre-fetching through the CM0/4_CA_CTL.PREF_EN register.

Table 14-4. CM0/4 cache control prefetch enable register values and bit field

Register	Bit field and bit name	Description
CM0_CA_CTL[32:0]	PREF_EN[30]	Prefetch enable 0: Disabled 1: Enabled
CM4_CA_CTL[32:0]	PREF_EN[30]	Prefetch enable 0: Disabled 1: Enabled

If prefetch is enabled, a cache miss results in a 16 B refill for the missing data and a 16 B prefetch for the next sequential data. The prefetch data is stored in a temporary buffer and is only copied to the cache when a read transfer “misses” and requires that data.

14.1.5 Read while write (RWW) support

The EZ-USB™ FX10 MCU supports read operations on one area while programming/erasing in another area. This is implemented to support firmware upgrades and parallel tasks in the dual-core system. The application flash contains eight RWW sectors, UFLASH0 to UFLASH7, each 256KB in size. The EE emulation and SFlash are additional RWW sectors apart from the main flash.

The RWW feature is available between sectors – you can read/execute from one sector while there is an ongoing write/erase operation in another sector. However, when the code execution/read is in the last 16 bytes of a given sector (say sector 0) and the flash write/erase operation is in the next sector (sector 1), an RWW violation may occur if prefetch is enabled. This is because prefetch will fetch the next 16 bytes of data, which is part of sector 1 while a write operation is underway in the same sector. This will result in a fault and should be considered during firmware design. Firmware can be designed to place dead code in the last 16 bytes of every sector making sure the last 16 bytes of a sector are never accessed or can disable prefetch during a flash write/erase operation.

14.2 Flash memory programming

14.2.1 Features

- SROM API library for flash management through system calls such as Program Row, Erase Flash, and Blow eFuse
- System calls can be performed using CM0+, CM4, or DAP

14.2.2 Architecture

Flash programming operations are implemented as system calls. System calls are executed out of SROM in Protection Context 0. System calls are executed inside CM0+ NMI. The system call interface makes use of IPC to initiate an NMI to CM0+.

System calls can be performed by CM0+, CM4, or DAP. Each of them have a reserved IPC structure (used as a mailbox) through which they can request CM0+ to perform a system call. Each one acquires the specific mailbox, writes the opcode and argument to the data field of the mailbox, and notifies a dedicated IPC interrupt structure. This results in an NMI interrupt in CM0+. The following diagram illustrates the system call interface using IPC.

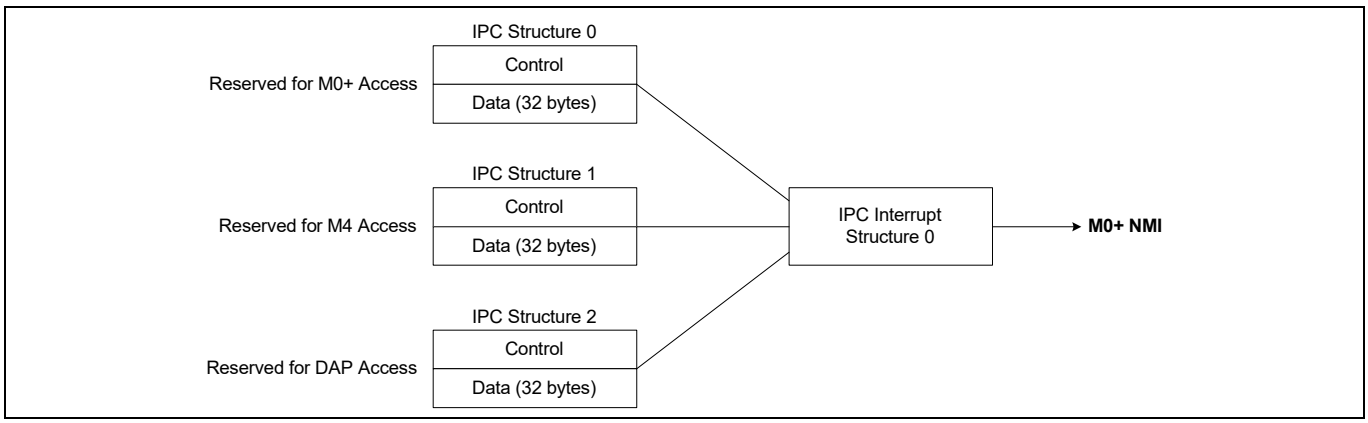


Figure 14-3. System call interface using IPC

Table 14-5. IPC structure

Master	**IPC structure resource used**
M0+	IPC_STRUCT0
M4	IPC_STRUCT1
DAP	IPC_STRUCT2
Access point	**IPC structure resource used**
CM0+ NMI interrupt	IPC_INTR_STRUCT0

The EZ-USB™ FX10 MCU's IPC component carries only a single 32-bit argument. This argument is either a pointer to SRAM or a formatted opcode or argument value that cannot be a valid SRAM address. The encoding used for DAP and the CM4 or CM0+ is slightly different.

DAP: If (opcode + argument) is less than or equal to 31 bits, store them in the data field and set the LSb of the data field as '1'. Upon completion of the call, a return value is passed in the IPC data register. For calls that need more argument data, the data field is a pointer to a structure in SRAM (aligned on a word boundary) that has the opcode and the argument. So it is a pointer if and only if the LSb is 0.

CM4 or CM0+: A pointer is always used to structure SRAM. Commands that are issued as a single word by DAP can still be issued by CM0+ or CM4, but use an SRAM structure instead.

The NMI interrupt handler for system calls works as follows.

- If the ROM boot process code is not initialized in the protection state (PROTECTION is still at its default/reset value UNKOWN), the NMI calls have no effect and the handler returns
- A jump table is used to point to the code in ROM or flash. This jump table is located in ROM or flash (as configured in SFlash).

The IPC mechanism is used to return the result of the system call. Two factors must be considered.

- The result is to be passed in SRAM: CM0+ writes the result in SRAM and releases the IPC structure. The requester knows that the result is ready from the RELEASE interrupt.
- The result is scalar (≤ 32 bits) and there is no SRAM to pass the result: in this case, the CM0+ writes the result to the data field of the IPC structure and releases it. The requester can read the data when the IPC structure lock is released. The requester polls the IPC structure to know when it is released.

External programmers program the EZ-USB™ FX10 MCU flash memory using the JTAG or SWD protocol by sending the commands to the DAP. Flash memory can also be programmed by the CM4/CM0+ CPU by accessing the IPC interface. This type of programming is typically used to update a portion of the flash memory as part of a bootload operation, or other application requirement, such as updating a lookup table stored in the flash memory. All write operations to flash memory, whether from the DAP or from the CPU, are done through the CM0+.

14.3 System call implementation

14.3.1 System call via CM0+ or CM4

System calls can be made from the CM0+ or CM4 at any point during code execution. CM0+ or CM4 should acquire the IPC_STRUCT reserved for them and provide arguments in either of the methods described above and notify IPC interrupt 0 to trigger a system call.

14.3.2 System call via DAP

If the device is acquired, then the boot ROM enters “busy-wait loop” and waits for commands issued by the DAP.

14.3.3 Exiting from a system call

When the API operation is complete, CM0+ will release the IPC structure that initiated the system call. If an interrupt is required upon release, the corresponding mask bit should be set in IPC_INTR_STRUCT_INTR_MASK.RELEASE[i]. The exit code must also restore the CM0+ protection context (PROT_MPU_MS_CTL.PC) to the one that was backed up in PROT_MPU_MS_CTL.PC_SAVED.

14.3.4 SRAM usage

2KB of SRAM [TOP_OF_SRAM – 2KB, TOP_OF_SRAM] is reserved for system calls and 1KB of SRAM [TOP_OF_SRAM – 3KB, TOP_OF_SRAM – 2KB] is used by SROM boot and should not be part of noinit RAM. TOP_OF_SRAM is the last address of SRAM.

14.4 SROM API library

SROM has two categories of APIs:

- Flash management APIs – These APIs provide the ability to program, erase, and test the flash macro.
- System management APIs – These APIs provide the ability to perform system tasks such as checksum and blowing eFuse.

Nonvolatile memory

Table 14-6 shows a summary of the APIs.

Table 14-6. List of system calls

System call	Opcode	Description	API category	Access allowed		
				Normal[a]	Secure	Dead
"Infineon ID" on page 213	0x00	Returns die ID, major/minor ID, and protection state	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
"Write row" on page 217	0x01	Blows the addressed eFuse bit	SYS	DAP	None	None
"Write row" on page 217	0x05	Pre-program, erase, and program the addressed flash row	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
"Program row" on page 219	0x06	Programs the addressed flash page	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
"Erase all" on page 221	0x0A	Erases all flash	FLS	CM0+, CM4, DAP	None	DAP
"Checksum" on page 223	0x0B	Reads either the whole flash or a row of flash, and returns the sum of each byte read	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
"Compute hash" on page 225	0x0D	Computes the hash value of the mentioned flash region	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
"ConfigureRegionBulk" on page 226	0x0E	Copies data from addressed source to addressed destination	SYS	None	None	None
"DirectExecute" on page 227	0x0F	Executes the code located at the provided address	SYS	DAP[b]	None	None
"Erase sector" on page 227	0x14	Erases the addressed flash sector	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
"Soft Reset" on page 229	0x1B	Provides system reset to either or both cores	SYS	CM0+, CM4, DAP	DAP[c]	CM0+, CM4
"Erase row" on page 230	0x1C	Erases the addressed flash row	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
"Erase subsector" on page 231	0x1D	Erases the addressed flash sector	FLS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
"GenerateHash" on page 233	0x1E	Generates the hash of the objects indicated by the table of contents	SYS	CM0+, CM4, DAP	None	None
"ReadUniqueID" on page 234	0x1F	Returns the unique ID of the die from SFlash	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP	CM0+, CM4, DAP
"CheckFactoryHash" on page 235	0x27	Checks if FACTORY HASH is valid	SYS	CM0+, CM4, DAP	None	None
"TransitionToRMA" on page 236	0x28	Convert part to RMA life cycle	SYS	None	CM0+, CM4, DAP	None
"ReadFuseByteMargin" on page 237	0x2B	Marginally reads eFuse	SYS	CM0+, CM4, DAP	CM0+, CM4, DAP[d]	None
"TransitionToSecure" on page 238	0x2F	Blows required eFuses to transition to SECURE or SECURE_WITH_DEBUG	SYS	CM0+, CM4, DAP[e]	None	None

a) See "Device security" on page 258.
b) Allowed only if SFLASH.DIRECT_EXECUTE_DISABLE = 0.
c) DAP has no access if secure eFuse is blown.
d) DAP has no access if secure eFuse is blown, but access is allowed in SECURE_WITH_DEBUG life cycle
e) Only allowed in the NORMAL_PROVISIONED life-cycle stage.

14.5 System calls

Table 14-6 lists all the system calls supported in EZ-USB™ FX10 MCUs along with the function description and availability in device protection modes. See the "Device security" on page 258 for more information on the device protection settings. Note that some system calls cannot be called by the CM4, CM0+, or DAP as given in the table. The following sections provide detailed information on each system call.

14.5.1 Infineon ID

This function returns a 12-bit family ID, 16-bit silicon ID, 8-bit revision ID, and the current device protection mode. These values are returned to the IPC_STRUCT_DATA register if invoked with IPC_STRUCT_DATA[0] set to '1'. Parameters are passed through the IPC_STRUCT_DATA register.

Note that only 32 bits are available to store the return value in the IPC structure. Therefore, the API takes a parameter ID type based on which it will return family ID and revision ID if the ID type is set to '0', and silicon ID and protection state if the ID type is set to '1'.

Table 14-7. Infineon ID

Infineon IDs	Memory location	Data
Family ID [7:0]	0xF0000FE0	Part number [7:0]
Family ID [11:8]	0xF0000FE4	Part number [3:0]
Major Revision	0xF0000FE8	Revision [7:4]
Minor Revision	0xF0000FEC	Rev and Minor Revision Field [7:4]
Silicon ID	SFlash	Silicon ID [15:0]
Protection state	MMIO	Protection [3:0]

Parameters if DAP is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x00	Silicon ID opcode
Bits [15:8]	0 - returns 0. Read family ID and revision ID from SFlash 1 - returns 16-bit silicon ID and protection state	ID type.
Bits [0]	0x1	Indicates that all the arguments are passed in DATA.

Parameters if CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x00	Silicon ID opcode.

Nonvolatile memory

Address	Value to be written	Description
Bits[15:8]	0 - returns 12-bit family ID and revision ID 1 - returns 16-bit silicon ID and protection state	ID type.
Bits[0]	0xXX	Not used (don't care).

Nonvolatile memory

Return if DAP invoked the system call

Address	Return value	Description
IPC_STRUCT_DATA Register		
Bits [7:0]	If ID type = 0, Family ID Lo If ID type = 1, Silicon ID Lo	See the device datasheet for silicon ID values for different part numbers.
Bits [15:8]	If ID type = 0, Family ID Hi If ID type = 1, Silicon ID Hi	
Bits [19:16]	If ID type = 0, Minor Revision ID If ID type = 1, Protection state 0: UNKNOWN 1: VIRGIN 2: NORMAL 3: SECURE 4: DEAD	See the device datasheet for these values.
Bits [23:20]	If ID Type = 0 Major Revision ID If ID Type = 1 Life-cycle stage 0: VIRGIN 1: NORMAL 2: SEC_W_DBG 3: SECURE 4: RMA	
Bits [27:24]	0xXX	Not used (don't care).
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

Nonvolatile memory

Return if CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [7:0]	If ID type = 0, Family ID Lo If ID type = 1, Silicon ID Lo	See the device datasheet for silicon ID values for different part numbers.
Bits [15:8]	If ID type = 0, Family ID Hi If ID type = 1, Silicon ID Hi	
Bits [19:16]	If ID type = 0, Minor Revision ID If ID type = 1, Protection state 0: UNKNOWN 1: VIRGIN 2: NORMAL 3: SECURE 4: DEAD	
Bits [23:20]	If ID Type = 0 Major Revision ID If ID Type = 1 Life-cycle stage 0: VIRGIN 1: NORMAL 2: SEC_W_DBG 3: SECURE 4: RMA	
Bits [27:24]	0xXX	Not used (don’t care).
Bits [31:28]	0xA	Success status code.

14.5.2 Write row

This function is used to program the flash. You must provide data to be loaded and the flash address to be programmed. The WriteRow parameter performs pre-program and erase, and then programs the flash row with contents from the row latch.

The EZ-USB™ FX10 MCU supports the Read While Write (RWW) feature, which allows flash to be read from a RWW sector that is not programmed/erased during a program/erase of another RWW sector. Each row is of 512 bytes in size.

The API is implemented in three phases to make it non-blocking. The first phase sets up the flash for pre-program and erase operations and returns to the user code by exiting from NMI without releasing the IPC structure that invoked the API. Now, user code and interrupts can be handled but no NMI can be invoked.

Upon completion of the erase, an interrupt is generated by the flash macro, which will invoke the second phase of the WriteRow API to complete the ongoing erase operation successfully and start the program operation. API returns from NMI to user code after it sets up for program operation.

Upon completion of the program, an interrupt is generated by the flash macro, which will invoke the third phase of the WriteRow API to complete the ongoing program operation successfully; this completes the WriteRow API. SROM API will now return the pass or fail status and releases the IPC structure.

This API can also be called in blocking mode by setting the blocking parameter as '1', in which case the API will return only after all flash operation completes.

The API returns a fail status if you do not have write access to flash according to SMPU settings. See the **"CPU subsystem (CPUSS)"** on page 63 for more details. After flash program operation is complete, the API will optionally compare the flash row with the contents in row latch for data integrity check. The function returns 0xF0000022 if the data integrity check fails.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to **"Power supply"** on page 276 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Note that the device should not be reset or transitioned into Deep Sleep power mode until the flash write is complete.

Nonvolatile memory

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x05	Write Row opcode
Bits[23:16]	0xXX	Not used
Bits[15:8]	Blocking: 0x01 – API blocks CM0+ Other values - Non-blocking	–
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/programmed 1 - Read operation is stalled until the erase/program operation is complete	–
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	–
SRAM_SCRATCH Register + 0x04		
Bits [23:16]	Verify row: 0 - Data integrity check is not performed 1 - Data integrity check is performed	–
Bits [31:24], Bits [15:0]	0xXX	Not used (don’t care)
SRAM_SCRATCH Register + 0x08		
Bits [31:0]	–	Flash address to be programmed. This should be provided in 32-bit system address format. For example, to program the second half-word, provide either of the byte address 0x1000003 or 0x1000004.
SRAM_SCRATCH Register + 0x0C		
Bits [31:0]	Data word 0 (data provided should be proportional to the data size provided, data to be programmed into LSbs)	–
SRAM_SCRATCH Register + 0x0C + n*0x04		
Bits [31:0]	Data word n (data to be programmed)	–

Nonvolatile memory

Return if DAP/CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS/Program command ongoing in background 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.5.3 Program row

This function programs the addressed flash row. You must provide the data to be loaded and flash address to be programmed. The flash row should be in the erased state before calling this function.

The function is implemented in two phases to make it non-blocking. The first phase sets up the flash for program operation and returns to user code by exiting from NMI without releasing the IPC structure that invoked the API. Now user code and interrupts can be handled but no NMI can be invoked.

Upon completion of the program operation, an interrupt is generated by the flash macro, which will invoke the second phase of the ProgramRow API to complete the ongoing program operation successfully. The SROM API will return the pass or fail status and releases the IPC structure.

After flash program operation is complete, the API will optionally compare the flash row with the contents in the row latch for data integrity check. It returns STATUS_PL_ROW_COMP_FA if data integrity check fails. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to **"Power supply"** on page 276 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Note that the device should not be reset or transitioned into Deep Sleep power mode until the flash write is complete.

Nonvolatile memory

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits[31:24]	0x06	Program Row opcode
Bits[23:16]	Skip blank check: 0x01 – Skips the blank check step Other – Perform blank check	–
Bits[15:8]	Blocking: 0x01 – API blocks CM0+ Other values - Non-blocking	–
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/programmed 1 - Read operation is stalled until the erase/program operation is complete	–
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	–
SRAM_SCRATCH Register + 0x04		
Bits [23:16]	Verify row: 0 - Data integrity check is not performed 1 - Data integrity check is performed	–
Bits [15:8]	Data location: 0 - row latch 1 - SRAM	–
Bits[7:0]	0xXX	Not used (don't care)
SRAM_SCRATCH Register + 0x08		
Bits [31:0]	–	Flash address to be programmed. This should be provided in 32-bit system address format. For example, to program the second half-word, provide either of the byte address 0x1000003 or 0x1000004.
SRAM_SCRATCH Register + 0x0C		
Bits [31:0]	Data word 0 (data provided should be proportional to data size provided, data to be programmed into LSbs)	–
SRAM_SCRATCH Register + 0x0C + n*0x04		
Bits [31:0]	Data word n (data to be programmed)	–

Nonvolatile memory

Return if DAP/CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS/Program command ongoing in background 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.5.4 Erase all

This function erases the entire flash macro specified. This API will erase only the main flash array. The API will check whether all data is '0' to confirm whether the erase is successful. It will return CHECKSUM_NON_ZERO error status if a non-zero word is encountered in the available flash.

The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operation.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to **"Power supply"** on page 276 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Parameters if DAP is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x0A	Erase All opcode
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/ programmed 1 - Read operation is stalled until the erase/program operation is complete	–
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	–
Bits [0]	1	Indicates that all the arguments are passed in DATA.

Nonvolatile memory

Parameters if CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x0A	Erase All opcode
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	–

Return if DAP invoked the system call

Address	Return value	Description
IPC_STRUCT_DATA Register		
Bits [23:20]	0x00	–
Bits [27:24]	0xXX	Not used (don’t care)
Bits [31:28]	0xA	Success status code

Return if CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [23:20]	0x00	–
Bits [27:24]	0xXX	Not used (don’t care)
Bits [31:28]	0xA	Success status code

14.5.5 Checksum

This function reads either the entire flash or a row of flash, and returns the sum of each byte read. Bytes 1 and 2 of the parameters select whether the checksum is performed on the entire flash or on a row of flash. This function will inherit the identity of the master that called the function. Hence if a non-secure master requests for either the whole or page checksum of a secured flash, then the fault exception will be raised by the hardware.

The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Parameters if DAP is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x0B	Checksum opcode
Bits [23:22]	0 - application 1 - EE emulation other - supervisory	Flash region
Bits [21]	0 - row 1 - whole flash	Whole flash
Bits [20:8]	–	Row ID
Bits [0]	1	Indicates that all the arguments are passed in DATA.

Parameters if CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x0B	Checksum opcode
Bits [23:22]	0 - application 1 - EE emulation 2 - supervisory	Flash region
Bits [21]	0 - row 1 - whole flash	Whole flash
Bits [20:8]	–	Row ID
Bits [0]	0	–

Nonvolatile memory

Return if DAP invoked the system call

Address	Return value	Description
IPC_STRUCT_DATA Register		
Bits [23:0]	Checksum	Checksum if status is SUCCESS
Bits [27:24]	0xXX	Not used (don't care)
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

Return if CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Checksum	Checksum if status is SUCCESS
Bits [27:24]	0xXX	Not used (don't care)
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details)

14.5.6 Compute hash

This function generates the hash of the flash region provided using the formula:

H(n+1) = {H(n)*2+Byte}% 127; where H(0) = 0

This function returns an invalid address status if called on an out-of-bound flash region. Note that CM0+ will inherit the protection context of the master, which invoked it before performing hash. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x0D	Compute hash opcode
Bits[23:16]	0xXX	Not used (don’t care)
Bits[15:8]	0x01 - CRC8 SAE others - Basic hash	Type
Bits[7:0]	0xXX	Not used (don’t care)
SRAM_SCRATCH Register + 0x04		
Bits [31:0]		Start address (32-bit system address of the first byte of the data).
SRAM_SCRATCH Register + 0x08		
Bits [31:0]	0 – 1 byte 1 – 2 bytes, and so on	Number of bytes

Return if DAP/CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Hash of the data	Hash of data if status is SUCCESS.
Bits [27:24]	0xXX	Not used (don’t care)
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **“System Call Status”** on page 239 for details).

14.5.7 ConfigureRegionBulk

This API writes a 32-bit data value to a set of contiguous addresses. It cannot be used to configure protected registers or flash. The Start and End addresses of the region are configurable but must be within a writable area. The region must also be 32-bit aligned. The data will be written to the region starting at the start address up to and including the memory at the end address. The start address must be lower than the end address or the API will return an error status.

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x0E	Configure region bulk opcode.
Bits [23:0]	0xXX	Not used (don't care)
SRAM_SCRATCH Register + 0x04		
Bits [31:0]	Start address	32-bit system address of the first byte of the data
SRAM_SCRATCH Register + 0x08		
Bits [31:0]	End address	32-bit system address of the first byte of the data
SRAM_SCRATCH Register + 0x0C		
Bits [31:0]	Data	Data to be written to each 32-bit address from start to end

Return if DAP/CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Error Code (if any)	Error Code
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.5.8 DirectExecute

This function directly executes code located at a configurable address. This function is only available in normal life-cycle state if the DIRECT_EXECUTE_DISABLE bit is 0.

Parameters if DAP is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x0F	Direct Execute Opcode
Bits [23:2]	Address [21:0]	Address of function to execute
Bit [1]	0 - SRAM 1 - Flash	Location of function to execute
Bit [0]	1	Indicates that all arguments are passed through DATA

Return if DAP invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Error Code (if any)	Error Code
Bits [31:28]	0xF = ERROR	Does not return any status on success.

14.5.9 Erase sector

This function erases the specified sector. Each sector consists of eight rows. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to **"Power supply"** on page 276 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Note that the device should not be reset or transitioned into Deep Sleep power mode until the erase is complete.

Nonvolatile memory

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x14	Erase Sector opcode.
Bits[23:16]	0x01 - Set FM interrupt mask Other - Do not set FM interrupt mask	Interrupt mask
Bits[15:8]	Blocking: 0x01 – API blocks CM0+ Other values – Non-blocking	–
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/programmed 1 - Read operation is stalled until the erase/program operation is complete	–
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	–
SRAM_SCRATCH Register + 0x04		
Bits [31:0]	–	Flash address to be erased. Should be provided in 32-bit system address format. For example, to erase the second sector you need to provide the 32-bit system address of any of the bytes in the second sector.

Return if DAP/CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.5.10 Soft Reset

This function resets the system by setting CM0+ AIRCR system reset bit. This will result in a system-wide reset, except debug logic. This API can also be used for selective reset of only the CM4 core based on the 'Type' parameter. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Parameters if DAP is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x1B	Soft Reset opcode
Bits [7:1]	0 - System reset 1 - Only CM4 resets	Type of reset
Bits[0]	0x1	Indicates all arguments are passed in DATA.

Parameters if CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x1B	Soft Reset opcode
Bits [7:1]	0 - System reset 1 - Only CM4 resets	Type of reset
Bits [0]	0xXX	Not used (don't care)

Return if DAP invoked the system call

Address	Return value	Description
IPC_STRUCT_DATA Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

Return if CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.5.11 Erase row

This function erases the specified row. You must provide the address of the row that needs to be erased. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to **"Power supply"** on page 276 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
5. Write VCC_SEL = 0.

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x1C	Erase row opcode
Bits[23:16]	0xXX	–
Bits[15:8]	0x01 - API blocks CM0+ Other - Non-blocking	–
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/programmed 1 - Read operation is stalled until the erase/program operation is complete	–
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	–
SRAM_SCRATCH Register + 0x04		
Bits [31:0]	Address	Flash address to be erased. This should be provided in the 32-bit system address format. For example, to erase the second row you need to provide the 32-bit system address of any of the bytes in the second row.

Return if DAP/CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.5.12 Erase subsector

This function erases the specified subsector. The values are returned to the IPC_STRUCT_DATA register. Parameters are passed through the IPC_STRUCT_DATA register.

Note that to be able to perform flash writes, the V_{CCD} should be more than 0.99 V. If the device operating voltage is less than 0.99 V (the 0.9-V mode of operation), follow this sequence to perform any flash write operations.

1. Write the appropriate registers to increase voltage from 0.9 V to 1.1 V. Refer to **"Power supply"** on page 276 for details on how to switch voltages.
2. Write VCC_SEL = 1.
3. Perform the flash write operations.
4. Write the appropriate registers to drop the regulated voltage from 1.1 V to 0.9 V.
1. Write VCC_SEL = 0.

Nonvolatile memory

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x1D	Erase subsector opcode
Bits [23:16]	0xXX	Not used (don't care)
Bits [15:8]	0x01- API blocks CM0+ Other - Non-blocking	–
Bit [2]	0 - Read operation is allowed on the sector that is not being erased/programmed 1 - Read operation is stalled until the erase/program operation is complete	–
Bit [1]	0 - HV cycles are firmware controlled 1 - HV cycles are hardware controlled	–
SRAM_SCRATCH Register + 0x04		
Bits [31:0]	Address	Flash address to be erased. This should be provided in 32-bit system address format. For example, to erase the second subsector you need to provide the 32-bit system address of any of the bytes in the second subsector.

Return if DAP/CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [23:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.5.13 GenerateHash

This function returns the truncated SHA-256 of the Flash boot programmed in SFlash. This function gets the Flash Boot size from the TOC. This function is typically called to confirm that the hash value to be blown into eFuse matches what the ROM Boot expects.

This function returns the number of zeros of the SECURE_HASH

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits[31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits[31:24]	0x1E	Opcode for GenerateHash
Bits[15:8]	0x1 - Return the factory hash Other - Return hash of all objects per TOC1 and TOC2	–

Return if DAP/CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits[23:0]	Error code (if any)	See **"System Call Status"** on page 239 for details
Bits[31:28]	0xA = SUCCESS 0xF = ERROR	Status code (See **"System Call Status"** on page 239 for details)
SRAM_SCRATCH Register + 0x04		
Bits[31:0]	HASH_WORD0	–
SRAM_SCRATCH Register + 0x08		
Bits[31:0]	HASH_WORD1	–
SRAM_SCRATCH Register + 0x0C		
Bits[31:0]	HASH_WORD2	–
SRAM_SCRATCH Register + 0x10		
Bits[31:0]	HASH_WORD3	–
SRAM_SCRATCH Register + 0x14		
Bits[31:0]	HASH_ZEROS	–

14.5.14 ReadUniqueID

This function returns the unique ID of the die from SFlash.

Parameters if DAP/ CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits[31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits[31:24]	0x1F	Opcode for ReadUniqueID
Bits[15:0]	Not Used (Don't care)	–

Return if DAP/ CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits[23:0]	DIE_LOT if SUCCESS	See **"System Call Status"** on page 239 for details
Bits[31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details)
SRAM_SCRATCH Register + 0x04		
Bits[31:0]	DIE_ID0	–
SRAM_SCRATCH Register + 0x08		
Bits[31:0]	DIE_ID1	–

14.5.15 CheckFactoryHash

This function generates the FACTORY_HASH according to the TOC1 and compares the value with the FACTORY1_HASH eFuse value.

Parameters if DAP is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x27	CheckFactoryHash Opcode
Bit [0]	1	Indicates all arguments are passed in DATA.

Parameters if CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x27	CheckFactoryHash Opcode

Return if DAP invoked the system call

Address	Return value	Description
IPC_STRUCT_DATA Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

Return if CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.5.16 TransitionToRMA

This function converts the part from SECURE, SECURE_WITH_DEBUG, or NORMAL to the RMA life-cycle stage. This API performs eFuse programming, VDD should be set to 2.5 V for successful programming.

This function uses Flash Boot functions. The stack consumed by these functions is around 700 bytes. This function returns STATUS_EMB_ACTIVE failure code if any active embedded flash operations are going on.

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:0]	0x28	Transition to RMA opcode
SRAM_SCRATCH + 0x04		
Bits [31:0]	Object Size in Bytes (including itself)	–
SRAM_SCRATCH + 0x08		
Bits [31:0]	Command ID (0x120028F0)	–
SRAM_SCRATCH + 0x0C		
Bits [31:0]	Unique ID word 0	–
SRAM_SCRATCH + 0x10		
Bits [31:0]	Unique ID word 1	–
SRAM_SCRATCH + 0x14		
Bits [31:0]	Unique ID word 2	–
SRAM_SCRATCH + 0x18		
Bits [31:0]	Signature of RMA certificate	–

Return if DAP invoked the system call

Address	Return value	Description
IPC_STRUCT_DATA Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

Return if CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.5.17 ReadFuseByteMargin

This API returns the eFuse contents of the addressed byte read marginally. The read value of a blown bit is '1' and of a not blown bit is '0'.

Parameters if DAP is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:24]	0x2B	Opcode for ReadFuseByteMargin
Bits [23:20]	0 - Low resistance, –50% from nominal 1 - Nominal resistance (default read condition) 2 - High resistance (+50% from nominal) Other - 100% from nominal	Margin control
Bits [19:8]	eFuse address (0, 1… 511)	–
Bit [0]	1 - Indicates all arguments are passed from DATA	–

Parameters if CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x2B	Opcode for ReadFuseByteMargin
Bits [23:20]	0 - Low resistance, –50% from nominal 1 - Nominal resistance (default read condition) 2 - High resistance (+50% from nominal) Other - 100% from nominal	Margin control
Bits [19:8]	eFuse address (0, 1… 511)	–
Bit [0]	1 - Indicates all arguments are passed from DATA	–

Return if DAP invoked the system call

Address	Return value	Description
IPC_STRUCT_DATA Register		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).
Bits [23:0]	Byte read from eFuse	See **"System Call Status"** on page 239 for details.

Nonvolatile memory

Return if CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).
Bits [23:0]	Byte read from eFuse	See **"System Call Status"** on page 239 for details.

14.5.18 TransitionToSecure

This API validates the FACTORY_HASH value and programs the SECURE_HASH, secure access restrictions, and dead access restrictions into eFuse. This function programs the SECURE or SECURE_WITH_DEBUG eFuse to transition to the SECURE or SECURE_WITH_DEBUG life-cycle stage. If the FACTORY_HASH or the TOC1 magic number is invalid, then the API fails and returns status code 0xF00000b5. Note that if the Factory_HASH in eFuse is invalid, then the TransitionToSecure API fails and the error code is written to the IPC[2].DATA register.

Parameters if DAP/CM0+/CM4 is master

Address	Value to be written	Description
IPC_STRUCT_DATA Register		
Bits [31:0]	SRAM_SCRATCH_ADDR	SRAM address where the API parameters are stored. This must be a 32-bit aligned address.
SRAM_SCRATCH Register		
Bits [31:24]	0x2F	Transition to Secure opcode
Bits [15:8]	1 - Blow Debug Fuse Other - Blow Secure Fuse	Select SECURE_WITH_DEBUG or SECURE life-cycle Stage
SRAM_SCRATCH + 0x04		
Bits [31:0]	SECURE_ACCESS_RESTRICT	–
SRAM_SCRATCH + 0x08		
Bits [31:0]	DEAD_ACCESS_RESTRICT	–

Return if DAP/CM0+/CM4 invoked the system call

Address	Return value	Description
SRAM_SCRATCH Register		
Bits [27:0]	Error code (if any)	See **"System Call Status"** on page 239 for details.
Bits [31:28]	0xA = SUCCESS 0xF = ERROR	Status code (see **"System Call Status"** on page 239 for details).

14.6 System Call Status

At the end of every system call, a status code is written over the arguments in the IPC_DATA register or the SRAM address pointed by IPC_DATA. A success status is 0xAXXXXXXX, where X indicates don't care values or return data for system calls that return a value. A failure status is indicated by 0xF00000XX, where XX is the failure code.

Note that for system calls that require access to restricted memory regions, if the client does not have access to the restricted region, the system call may return 0xF000_0013 rather than the expected 0xF000_0008. Because of this, it is important to ensure that arguments are passed correctly and that the memory region to access is available for read or write access.

Table 14-8. System call status

Status code	Description
0xAXXXXXXX	Success – The X denotes a don't care value, which has a value of '0' returned by the SROM.
0xF0000001	Invalid protection state – This API is not available in current protection state.
0xF0000002	Invalid eFuse address.
0xF0000003	Invalid flash page latch address.
0xF0000004	Wrong or out-of-bound flash address.
0xF0000005	Row is write protected.
0xF0000006	Client did not use its reserved IPC structure for invoking the system call.
0xF0000008	Client does not have access to the addressed memory location.
0xF0000009	Command in progress.
0xF000000A	Checksum of flash resulted in non-zero.
0xF000000B	The opcode is not valid.
0xF0000013	An argument to the system call is not in a valid location.
0xF0000021	Sector erase requested on SFlash region.
0xF0000041	Bulk erase failed.
0xF0000042	Sector erase failed.
0xF0000043	Subsector erase failed.
0xF0000044	Verification of Bulk, Sector, or Subsector fails after program/erase.
0xF000000F	Returned when invalid arguments are passed to the API. For example, calling Silicon ID API with ID type of 0x5.
0xF000000b5	Invalid FACTORY_HASH

15 Boot code

System boot is defined as the process of validating and starting the product firmware. EZ-USB™ FX10 has 128 KB of embedded SROM, which stores the firmware to begin the boot process. This firmware is called ROM boot. The second stage of the boot process takes place in SFlash and is called Flash boot. The main function of the boot process is to configure the system (apply trims, configure access, and set protection settings according to the product life-cycle stage), authenticate the application, and transfer control to the application.

15.1 Features

The EZ-USB™ FX10 boot code supports the following features:

- After reset, the boot code starts execution from ROM on the CM0+
- The boot process consists of two parts – ROM boot process and Flash boot process
- The ROM boot code applies life-cycle stage and protection state
- The ROM boot code validates the integrity of the Flash boot process before starting it
- The Flash boot code validates the integrity of the user application before starting it when the device in the SECURE life-cycle stage. User application validation may optionally be enabled in the NORMAL life-cycle stage

15.2 ROM boot

The boot code starts execution from ROM. The ROM boot code applies trims and configurations, and validates the integrity of the flash boot process before starting it.

15.2.1 Data integrity checks

The ROM boot uses a table of contents (TOC) data structure to locate various objects that are stored in both application flash and SFlash. The location of some of these objects is fixed in the factory while other object locations are fixed at the original equipment manufacturer (OEM). The TOC is split into two parts – TOC1 and TOC2 – that correspond to objects fixed in the factory and objects fixed at OEM, respectively. TOC1 and TOC2 are stored in SFlash.

The data integrity of selected objects listed at the beginning of TOC1 must be verified at the OEM before transitioning the part to the SECURE life-cycle stage. The FACTORY_HASH is generated using SHA-256, which is the concatenation of the objects to be verified in TOC1. The FACTORY_HASH is stored in eFuse before the device leaves Infineon. The device validates the FACTORY_HASH as part of the TransitionToSecure system call.

The data integrity of selected objects in both TOC1 and TOC2 must be verified as part of authentication of Flash boot by ROM boot in the SECURE life-cycle stage. A 128-bit truncated SHA-256 value known as the SECURE_HASH is used to check the integrity of these objects. If the TOC structures fail integrity checks, then the RTOC structures are checked for validity. If the RTOC structures are valid, then the boot process uses the RTOC structures in place of the TOC structures. The SECURE_HASH is programmed automatically during the TransitionToSecure system call. If the TransitionToSecure system call fails, it will exit before programming the SECURE_HASH in eFuse. See **“System Call Status”** on page 239 for more information.

A summary of the objects used in the data integrity checks is shown in **Figure 15-1**. The TOC structure formats are outlined in **Table 15-1** to **Table 15-3**.

Boot code

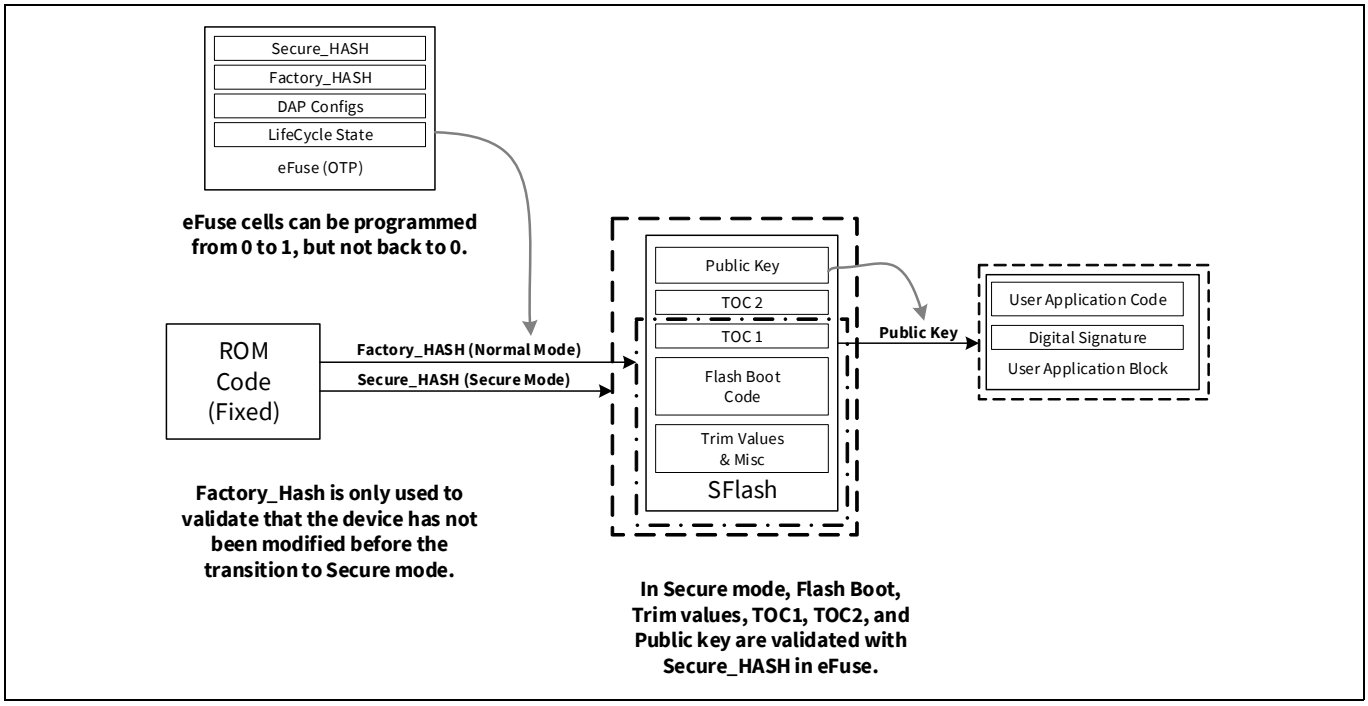


Figure 15-1. Objects, memory regions, and stages of boot process for data integrity checks

Table 15-1. TOC1 format

Offset from beginning of row	Purpose	Comment
0x00	Object size in bytes for CRC calculation starting from offset 0x00	TOC1 filled in VIRGIN protection state by Infineon
0x04	Magic number (0x01211219)	
0x08	Number of objects starting from offset 0xC to be verified for FACTORY_HASH	
0x0C	Unused	
0x10	Address of Unique ID (fixed size of 12 bytes) stored in SFlash	
0x14	Address of Flash boot object that includes Flash Patch object stored in SFlash	
0x18	Unused	
0x1C-0x1F8	Unused	
0x1FC	CRC16-CCITT (the upper two bytes contain the CRC little endian value and the lower two bytes are 0)	

Table 15-2. TOC2 format

Offset from beginning of row	Absolute address	SFlash register	Purpose
0x00	0x16007C00	TOC2_OBJECT_SIZE	Object size in bytes for CRC calculation starting from offset 0x00
0x04	0x16007C04	TOC2_MAGIC_NUMBER	Magic number (0x01211220)
0x08	0x16007C08	TOC2_KEY_BLOCK_ADDR	Address of Key Storage Flash Blocks. Note that this also marks the top of flash available.
0x0C	0x16007C0C	TOC2_SMIF_CFG_STRUCT_ADDR	Null terminated table of pointers representing the SMIF configuration structure
0x10	0x16007C10	TOC2_FIRST_USER_APP_ADDR	Address of first user application object
0x14	0x16007C14	TOC2_FIRST_USER_APP_FORMAT	First application object format (4 bytes). ‘0’ means basic application format, ‘1’ means Infineon secure application format, and ‘2’ means simplified secure application format
0x18	0x16007C18	TOC2_SECOND_USER_APP_ADDR	Address of second user application object (0’s if none)
0x1C	0x16007C1C	TOC2_SECOND_USER_APP_FORMAT	Second application object format (4 bytes).
0x20	0x16007C20	TOC2_SHASH_OBJECT	Number of additional objects (in addition to the object included in the FACTORY_HASH) starting from offset 0x24 to be verified for SECURE_HASH. The maximum number of additional objects is 15.
0x24	0x16007C24	TOC2_SIGNATURE_VERIF_KEY	Address of signature verification key (0 if none). The object is signature scheme specific. It is the public key in case of RSA.
0x1F8	0x16007DF8	TOC2_FLAGS	Flash boot parameters. These fields are listed in **Table 15-3**.
0x1FC	0x16007DFC	TOC2_CRC_ADDR	CRC16-CCITT (the upper 2 bytes contain the CRC little endian value and the lower 2 bytes are ‘0’

Table 15-3. TOC2_FLAGS bits, default settings, and descriptions

TOC2_FLAGS bits	Name	Default	Description
1:0	CLOCK_CONFIG	2	Indicates the CM0+ CPU clock frequency configuration. The clock will remain at this setting after flash boot execution. 0 = 8 MHz IMO with no FLL 1 = 25 MHz, IMO with FLL 2 = 50 MHz, IMO with FLL (Default) 3 = Use ROM boot clocks configuration (100 MHz)
4:2	LISTEN_WINDOW	0	Determines the wait window to allow sufficient time to acquire the debug port. 0 = 20 ms (Default) 1 = 10 ms 2 = 1 ms 3 = 0 ms (No listen window) 4 = 100 ms
6:5	SWJ_PINS_CTL	2	Determines if the SWJ pins are configured in SWJ mode by flash boot. SWJ pins can be enabled later in the application code. 0 = Do not enable SWJ pins in flash boot. Listen window is skipped. 1 = Do not enable SWJ pins in flash boot. Listen window is skipped. 2 = Enable SWJ pins in flash boot (default) 3 = Do not enable SWJ pins in flash boot. Listen window is skipped.
8:7	APP_AUTH_DISABLE	0	Determines if the application digital signature verification is performed. 0 = Authentication is enabled (default) 1 = Authentication is disabled 2 = Authentication is enabled (recommended) 3 = Authentication is enabled
10:9	FB_BOOTLOADER_DISABLE	0	Reserved

The wait window time is configurable; however, the CRC value used to validate the TOC2 must be updated after changing the value. New entries that need HASH can grow from the top starting at offset 0x28; entries that do not need HASH can grow from the bottom starting at offset 0x1F8. When the device is in the SECURE life-cycle state, the TOC2 can no longer be modified.

15.2.2 Life-cycle stages and protection states

SECURE and SECURE_WITH_DEBUG life-cycle stages are governed by eFuse. All life-cycle stages are irreversible once set. **Table 15-4** shows the eFuse locations that store information, access restriction settings, and life-cycle stage settings.

Table 15-4. Object location in eFuse

Offset	# of bytes	Name	Description
0x00	20	Reserved	Reserved for EZ-USB™ FX10 MCU system usage
0x14	16	SECURE_HASH	Secure objects 128-bit hash
0x24	2	Reserved	Reserved
0x26	1	SECURE_HASH_ZEROS	Number of zeros in SECURE_HASH
0x27	2	DEAD_ACCESS_RESTRICT	Access restrictions in Dead life-cycle stage
0x29	2	SECURE_ACCESS_RESTRICT	Access restrictions in Secure life-cycle stage
0x2B	1	LIFECYCLE_STAGE	Normal, Secure, and Secure with Debug fuse bits
0x2C	16	FACTORY_HASH	FACTORY_HASH value
0x40	64	CUSTOMER_DATA	Customer data

Normal mode access restrictions to be applied on DAP are stored in SFlash.

Table 15-5. Normal life-cycle stage DAP Restriction register in SFlash

Offset	# of bytes	Name	Description
0x1A00	2	NORMAL_ACCESS_RESTRICTIONS	Access restrictions applied to DAP in NORMAL life-cycle state.

The LIFECYCLE_STAGE object in eFuse is used to set the life-cycle state of the device. **Table 15-6** shows the bits and the associated settings of the LIFECYCLE_STAGE location.

Table 15-6. eFuse bits and life-cycle states

Bits	Field name	Description
0	NORMAL	A ‘1’ indicates that all factory trimming and testing is complete. The device is in NORMAL mode.
1	SECURE_WITH_DEBUG	A ‘1’ indicates that the device is in SECURE_WITH_DEBUG mode. Before the device is transitioned to this stage, the SECURE_HASH must have been programmed in eFuse and valid application code must have been programmed in the main flash.
2	SECURE	A ‘1’ indicates that the device is in SECURE mode. The SECURE_HASH must have been programmed in eFuse and valid application code has been programmed in the main flash.
3	RMA	This life-cycle stage allows Failure Analysis (FA). The part is transitioned to the RMA life-cycle stage when the customer wants Infineon to perform failure analysis. The customer erases all sensitive data before invoking the system call that transitions the part to RMA.

Boot code

The DAP settings for Secure and Dead protection states are set in the SECURE_ACCESS_RESTRICT0 and DEAD_ACCESS_RESTRICT0 registers. The DAP settings for the Normal protection state is set in NORMAL_ACCESS_RESTRICTIONS in SFlash. **Table 15-7** shows the format used to set the DAP settings for these states. For more information about the protection states, see the **"Device security"** on page 258.

Table 15-7. DAP access restriction registers for normal, secure, and dead protection states (all default to 0)

Bits	Name	Description
0	CM0_AP_DISABLE	A '1' indicates that this device does not allow access to the M0+ debug access port
1	CM4_AP_DISABLE	A '1' indicates that this device does not allow access to the M4 debug access port
2	SYS_AP_DISABLE	A '1' indicates that this device does not allow access to the system debug access port
3	SYS_AP_MPU_ENABLE	A '1' indicates that the MPU on the system debug access port must be programmed and locked according to the settings in the next four fields. The SYS_DISABLE bit must be left at '0' for this setting to matter. If the SYS_DISABLE bit is set to '1', then the next four fields are invalid. This affects only the SYS_AP. It does not affect the CM0/4 AP.
5:4	SFLASH_ALLOWED	This field indicates what portion of Supervisory Flash is accessible through the system debug access port. Only a portion of Supervisory Flash starting at the bottom of the area is exposed. Encoding is as follows: 0: Entire region 1: One-half 2: One-quarter 3: Nothing For example, for an encoding of "2: one-quarter", the valid code region starts at address 0x16000000 and will go up to one-quarter of the SFlash memory region.
7:6	MMIO_ALLOWED	This field indicates what portion of the MMIO region is accessible through the system debug access port. Encoding is as follows: 0: All MMIO registers 1: Only IPC MMIO registers accessible (system calls) 2, 3: No MMIO access

The SECURE_ACCESS_RESTRICT1, DEAD_ACCESS_RESTRICT1, and NORMAL_ACCESS_RESTRICTIONS [15:8] (SFlash) registers control debug access restriction to specific memory regions. **Table 15-8** shows the format used to set these restrictions.

Table 15-8. SECURE_ACCESS_RESTRICT1, DEAD_ACCESS_RESTRICT1, and NORMAL_ACCESS_RESTRICTIONS [15:8] registers (all default to '0')

Bits	Name	Description
2:0	FLASH_ALLOWED	This field indicates what portion of the main flash is accessible through the system debug access port. Only the portion starting at the bottom of flash (0x1000_0000) is exposed to the system DAP. The encoding is as follows: 0: entire region 1: seven-eighth 2: three-fourth 3: one-half 4: one-quarter 5: one-eighth 6: one-sixteenth 7: nothing See **Figure 15-2** for an example of the encoding for FLASH_ALLOWED.
5:3	SRAM_ALLOWED	This field indicates what portion of SRAM 0 is accessible through the system debug access port. Only the portion starting at the bottom of the area is exposed. Encoding is the same as FLASH_ALLOWED.
6	UNUSED	UNUSED
7	DIRECT_EXECUTE_DISABLE	Disables DirectExecute system call functionality (implemented in software). See the **"Nonvolatile memory"** on page 205 for more information.

As an example, if you want to limit the lower one-quarter of flash of a device with 2MB of flash, a "4" is written into the FLASH_ALLOWED field. Only the flash area between 0x1000_0000 and 0x1007_FFFF will be accessible via the SYS_DAP. The remaining areal between 0x1008_0000 to 0x101F_FFFF is not accessible. This means that only the lower quarter of flash can be reprogrammed, erased, or read by the debug port.

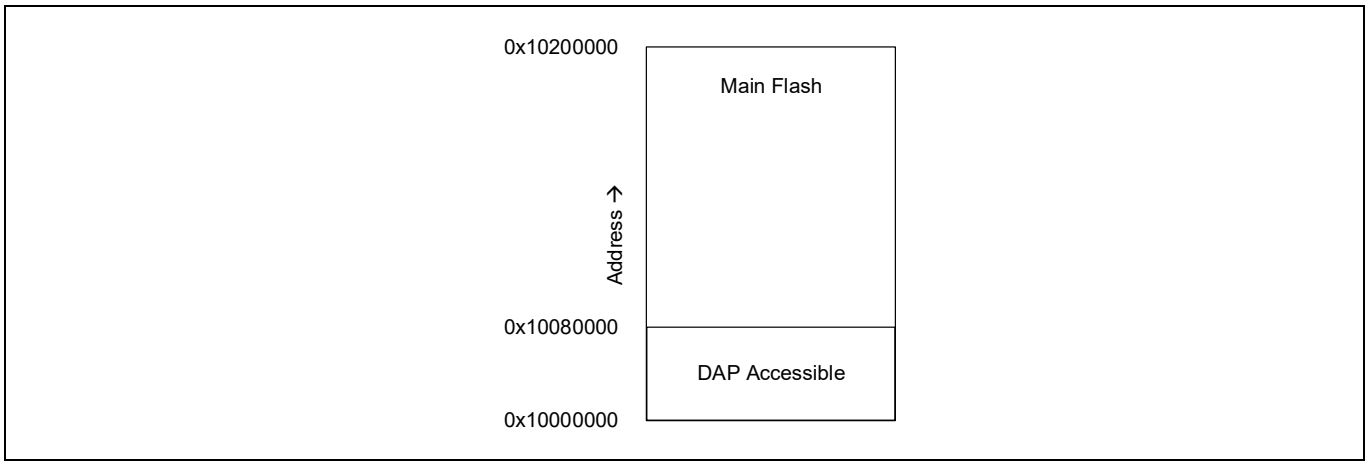


Figure 15-2. FLASH_ALLOWED with encoding '4'

As shown in **Figure 15-2**, if you want to limit the lower fourth of flash of a device with 2MB of flash, write a '4' into the FLASH_ALLOWED field. Only the flash area between 0x10000000 and 0x1007FFFF will be accessible via the SYS_DAP. The remaining area between 0x10080000 to 0x101FFFFF would not be accessible. This means that only the lower fourth of flash can be reprogrammed, erased, or read by the debug port.

15.2.3 Secure boot in ROM boot

At the end of the ROM boot process, the SECURE eFuse bit is checked. If the life-cycle is SECURE, then ROM boot reads the SECURE access restrictions in eFuse and deploys the settings using the SYS_AP MPU. If the APP_AUTH_DISABLE field of the TOC2 is not disabled, then ROM boot calculates the SECURE_HASH value from the contents of Flash boot, the TOC1 and TOC2 structures, and the public key and compares it to the stored hash value (SECURE_HASH) in eFuse. If they match, the Flash boot code and data are authentic. If the Flash boot code is authentic, then ROM boot will hand over the boot process to Flash boot, which resides in SFlash. If the Flash boot is not authentic, then the boot code sets the device into the DEAD protection state.

15.2.4 Protection setting

ROM boot sets some of the protections units (SMPU, PPU, and MPU) during the boot process. It is recommended that these settings are not modified.

SYS_AP DAP access restrictions are deployed by MPU15 based on access restrictions encoding stored in SFlash or eFuse. Eight MPU structures implement the SYS_AP DAP access restrictions. If the SYS_AP MPU is disabled, the following regions are available for access through the system debug access port.

Table 15-9. Protection setting

Protection structure	Attribute settings	Protected region	Region base	Region size (bytes)
PROT_MPU15_MPU_STRUCT0	ADDR: 0x00000000 ATT: 0x9E000000	ALL	0x00000000	2G
PROT_MPU15_MPU_STRUCT1	ADDR: 0x402200F8 ATT: 0x8700007F	IPC	0x40220000	96
PROT_MPU15_MPU_STRUCT2	–	Work flash region protection option is not present in EZ-USB™ FX10	–	–
PROT_MPU15_MPU_STRUCT3	ADDR: 0x16000000 ATT: 0x8E00007F	SFlash	0x16000000	32K (SFlash size)
PROT_MPU15_MPU_STRUCT4	ADDR: 0x10000000 ATT: 0x9100007F	Flash	0x10000000	256K (Flash Size)
PROT_MPU15_MPU_STRUCT6	ADDR: 0x8000000 ATT: 0x9000007F	RAM0	0x08000000	128K (SRAM0 size)

Other protection units are set up during ROM boot to restrict access to various regions to ensure device security. The following tables list those structures, the regions they protect, and the protection settings applied to those regions. The protection settings are applied in all life-cycle stages except VIRGIN.

Table 15-10. Region access

Region	Description
CPUSS_BOOT	Contains CM0+ protection context 0-3, debug port control, protection status, RAM, and ROM trim control registers.
EFUSE_CTL	EFUSE_CTL register. Specifies retention for eFuse registers.
EFUSE_DATA	Contains registers that support lifecycle setting, key storage, and other one-time-programmable data.
FM_CTL	Specifies flash hardware block control settings including direct memory cell access addressing, write enable, and interface selection.
Bottom 2KB of SRAM	Contains the system call stack.
SROM	Contains system firmware that supports ROM boot and system calls.
CRYPTO MMIO	Contains registers that support error correction, random number generation, device key locations, and other general cryptography information.
IPC MMIO	Contains registers that support inter-processor communication

Table 15-11. Protection structures

Protection structure	Protection (Read/write)	Attribute settings	Protected region	Region base	Region size (bytes)
PERI_MS_PPU_FX15	No Access	ATT0: 0x00000000	CRYPTO Memory buffer, used to store operand data for crypto operation	0x40108000	4K
PERI_MS_PPU_FX18	Write only in PC0 Read Always	ATT0: 0x1515151F ATT1: 0x15151515	CPUSS_BOOT	0x40202000	512
PERI_MS_PPU_FX58	Write only in PC0 Read Always	ATT0: 0x1515151F ATT1: 0x15151515	MPU15_MAIN DAP MPU	0x40237C00	1K
PERI_MS_PPU_FX68	R/W only in PC0	ATT0: 0x0000001F	FM_CTL	0x4024F000	4K
PERI_MS_PPU_FX148	R/W only in PC0	ATT0: 0x0000001F	EFUSE_CTL Registers	0x402C0000	128
PERI_MS_PPU_FX149	R/W only in PC0	ATT0: 0x0000001F	EFUSE_DATA Registers	0x402C0800	512
PROT_SMPU_SMPU_STRUCT15	Read only in PC0	ATT0: 0x8AFFFE00 ATT1: 0x80FFFE40	Last 2KB of SRAM	0x0801F7D0	2K

Table 15-11. Protection structures

Protection structure	Protection (Read/write)	Attribute settings	Protected region	Region base	Region size (bytes)
PROT_SMPU_SMPU_STRUCT14	Read only in PC0	ATT0: 0x8E00FF00 ATT1: 0x8700FF49	SROM area containing boot code, Crypto, and Flash macro controls.	0x00001000	24K
PERI_MS_PPU_PR2	Disable access during system calls with Crypto enabled (all access allowed otherwise)	ATT0: 1F1F1F1F ATT1: 1F1F1F1F	Entire Crypto MMIO region during any crypto operation	0x40100000	64K
PERI_MS_PPU_PR3	Disable access during system calls (all access allowed otherwise)	ATT0: 1F1F1F1F ATT1: 1F1F1F1F	IPC MMIO (during system calls)	0x40220000	32

15.2.5 SWD/JTAG repurposing

When the user-configured access restrictions disable access to the debug access ports, the boot process does not access or change the SWD/JTAG pins. User firmware can, at any time, change the configuration of the SWD/JTAG pins to another mode, peripheral, configuration, or purpose. To allow debugging of such applications, a configurable ‘listen window’ is provided in the TOC2 at offset 0x1F8 bits [4:2]. The boot process will connect and enable the JTAG/SWD interface and wait for a specified time before starting application firmware. It is expected that application firmware checks the CPUSS DP_STATUS.SWJ_CONNECTED bit and repurposes the pins only when no SWD/JTAG connection is available.

15.2.6 Disable WDT

The ROM boot code will disable the WDT if the eFuse.WDT_DISABLE bit is set.

15.2.7 ROM boot flow chart

Figure 15-3 shows the ROM boot flow chart.

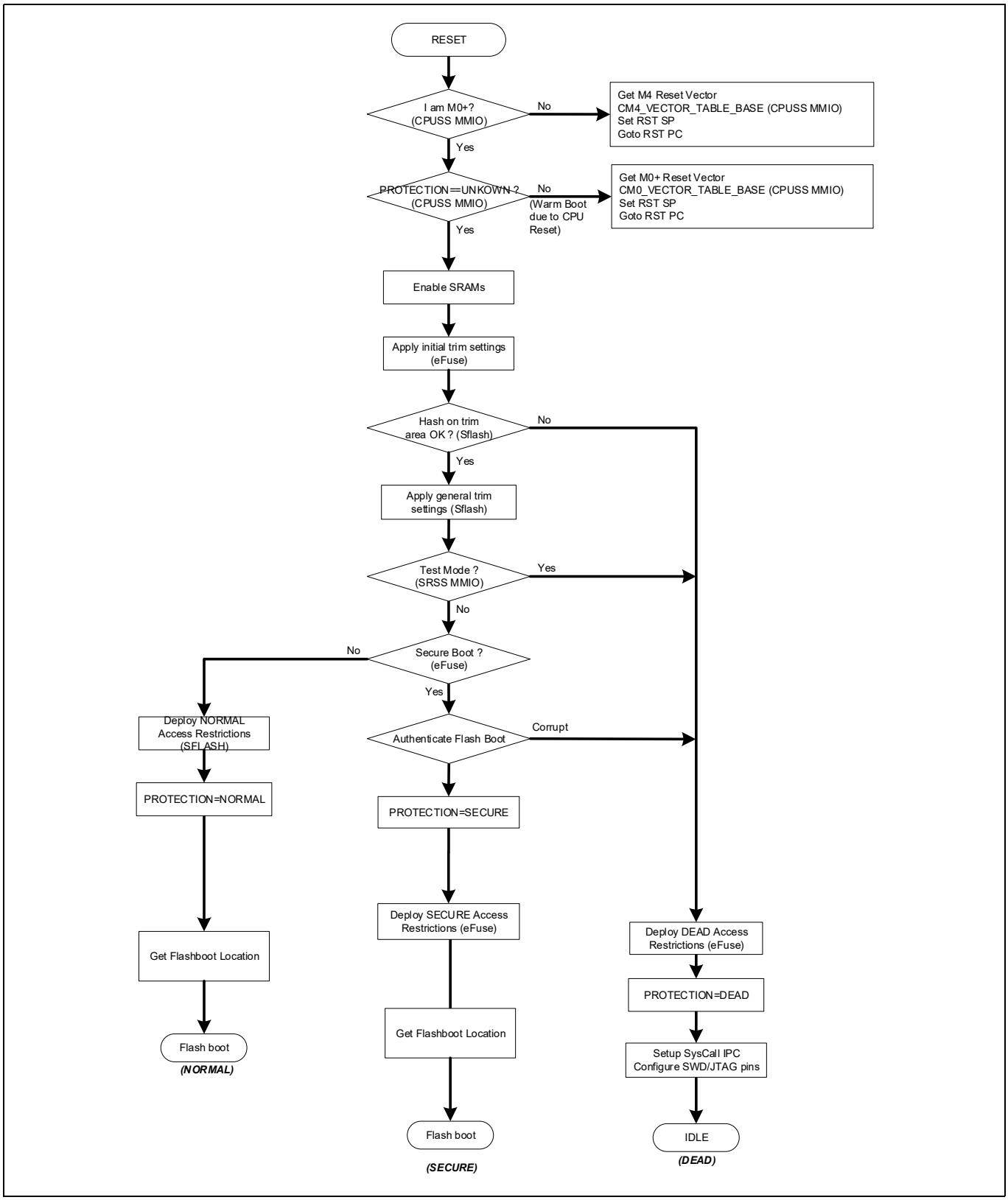


Figure 15-3. ROM boot flow chart

15.3 Flash boot

15.3.1 Overview

The second stage of boot, Flash boot, is entered after ROM boot has authenticated the Flash boot image if the device is in the SECURE or SECURE_WITH_DEBUG life-cycle stage, or if the APP_AUTH_DISABLE field is not set in TOC2. In the SECURE or SECURE_WITH_DEBUG life-cycle states, flash boot prevents an application that does not pass integrity checks from running. The integrity checks can optionally be enabled in the NORMAL mode by setting the APP_AUTH_DISABLE field in TOC2 to any 2-bit value other than 1. This ensures that the application originates from a valid source and has not been altered.

The Flash boot firmware:

- Runs on the security processor (Cortex-M0+)
- Validates the contents of the TOC2
- Validates the user application in the Secure life-cycle state or optionally in the Normal life-cycle state
- Sets up the debug access port (DAP)
- Enables system calls
- Hands control to the user application

15.3.2 Features of flash boot

- RSASSA PKCS1 V1.5 with SHA-256 User Application Signature Verification
- 2048-bit RSA public key with pre-calculated key coefficients for RSA calculation speed increase
- Support for all device life-cycle stages
- Configurable debug listen window

15.3.3 Using flash boot

Flash boot can be used to allow only authenticated firmware to run on a given platform.

Flash boot determines the life-cycle stage, validates the TOC2, and optionally validates the firmware before transferring execution to the user's CM0+ application code. The user's CM0+ application enables the CM4 core.

The user firmware will not execute if it is modified by third-party malicious software. The user firmware in flash is validated using the RSA algorithm. The 2048-bit public key in SFlash and the digital signature stored with the user application are located using the table of contents, which is located at the end of SFlash. The public key is hashed by SECURE_HASH and is validated during boot time. If the code is defined as insecure, the device will go to dead mode with limited debug capabilities determined by the eFuse settings.

15.3.4 Flash boot layout

Flash boot consists of a Header field and a Code Segment field. For hash calculation, the object size must be a multiple of 16B so zero padding is applied in the Code Segment.

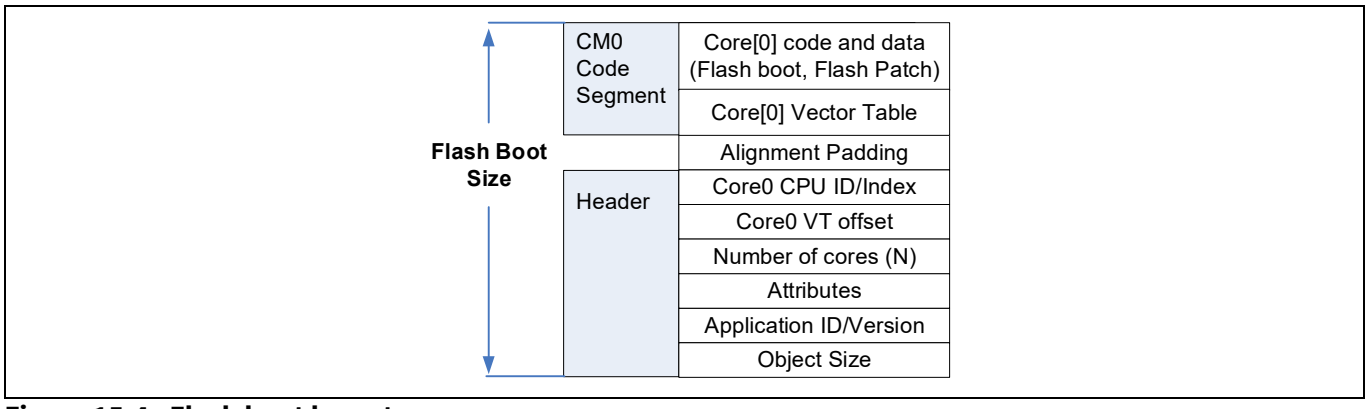


Figure 15-4. Flash boot layout

15.3.4.1 Header

The Flash boot header consists of:

- The Flash boot object size
- The Flash boot application version
- The number of cores (set to '1' for Flash boot)
- CM0+ vector table offset
- Infineon ID and CPU Core Index

15.3.4.2 Code segment

The CM0 code segment consists of:

- CM0+ vector table
- CM0+ Flash boot code and data
- Zero padding to make the Flash boot object a multiple of 16B

15.3.5 Flash boot flow chart

The Flash boot program flow is shown in **Figure 15-5**. The entry point is a fixed offset in SFlash. Each section of the flow chart is labeled with an index number. In the sections following the flow chart, explanations of each step are provided with the associated index number.

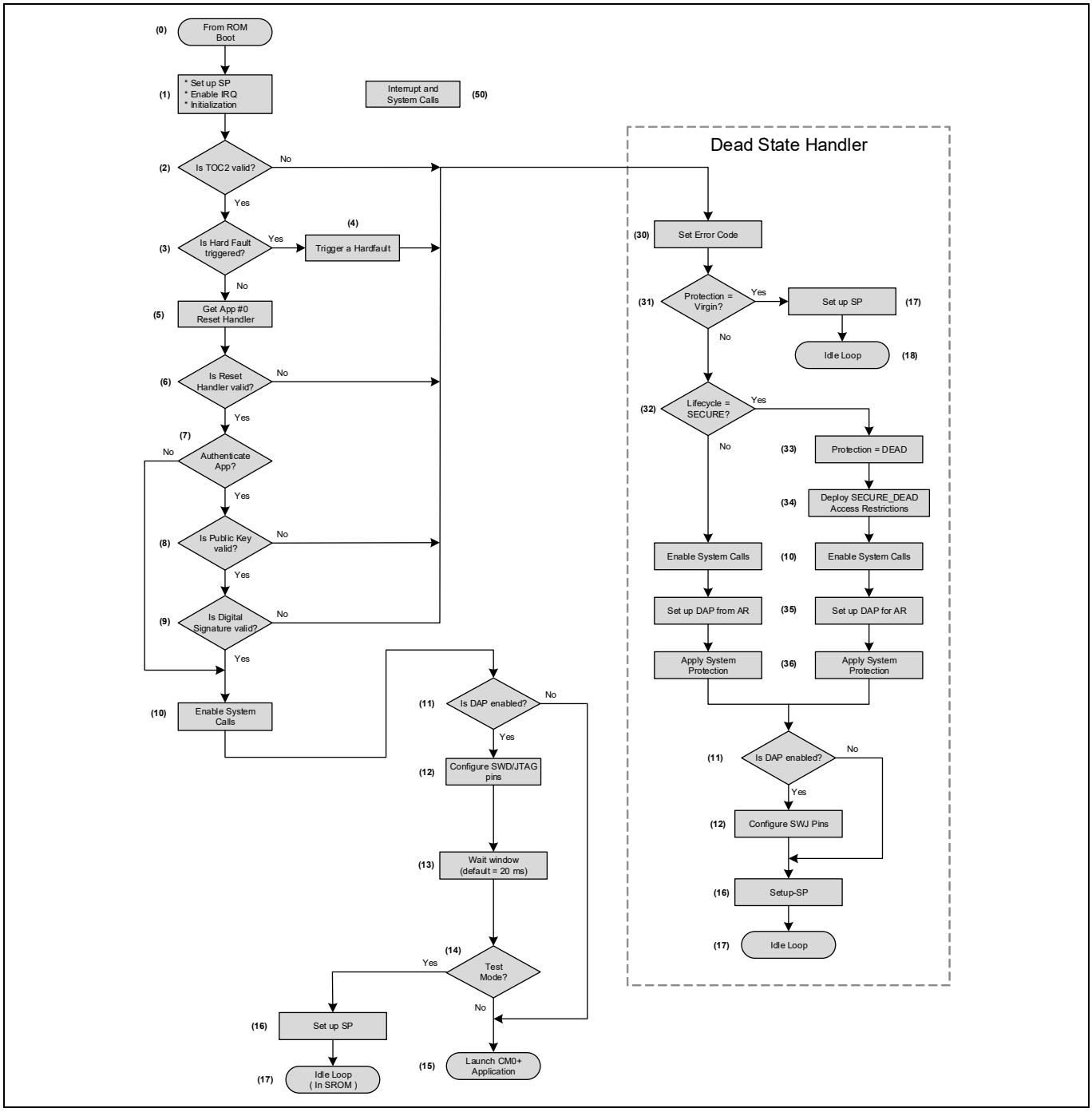


Figure 15-5. Flash boot flow chart

- Entry from ROM Boot (0)

ROM boot transfers control to Flash boot after it validates the SFlash block and TOC1 in the user flash.

- Basic Initialization (1)

This stage sets the value of the stack pointer during runtime. To support recovery from Hard-Fault exceptions during Flash boot, this stage also enables interrupts.

Boot code

- Is TOC2 Valid? (2)

The TOC2 may be in three states:

- VALID: The TOC2 structure and CRC are valid. This is the default state of the TOC2
- CORRUPTED: Either the TOC2 structure or CRC value are incorrect
- ERASED: The first two 32-bit words at the start of TOC2 are equal to the SFlash erase value. The erased value is 0x0000_0000.

If the PROTECTION is SECURE then ERASED state is a part of CORRUPTED state.

If the PROTECTION is not SECURE then ERASED state is a special case of VALID state. In this case, Flash boot treats all the TOC2 entries as having a default value:

- SFLASH_TOC2_FIRST_USER_APP_ADDR is 0x1000_0000 (the start of Flash)
- SFLASH_TOC2_FIRST_USER_APP_FORMAT is 0 (Basic Application Format)
- SFLASH_TOC2_FLAGS = 0x0000_0000

The other TOC2 entries will not be used when TOC2 is in ERASED or corrupted states.

- Is Hard Fault Triggered? (3)

The Flash boot test system requires a way to trigger a hard fault in the Flash boot code. The condition to trigger a hard-fault is as follows:

1. SFLASH_FLAGS bit FB_HARDFAULT is set.
2. A 32-bit word in TOC2 at offset + 500 contains an address to a 32-bit word named HardFaultTrigger.
3. HardFaultTrigger value is 0x0000_0001.

- Trigger a Hard Fault (4)

Flash boot can be used to trigger a hard fault for a testing purposes.

This can be done, for example, by a 32-bit word read or write to an unaligned address, or to an invalid memory region.

The hard fault handler recovers the MCU into the DEAD state branch. In this branch, an error code is set and the DAP is enabled.

- Get App #0 Reset Handler (5)

Flash boot reads the application start address from the TOC2 entry:

- SFLASH_TOC2_FIRST_USER_APP_ADDR for App#0

The application format is stored in the TOC2 entry:

- SFLASH_TOC2_FIRST_USER_APP_FORMAT for App #0

The address of reset handler inside the application depends on the application format.

- Is Reset Handler Valid? (6)

The Flash boot checks if the address of the reset handler for the user application is in RAM, SFlash, application flash, or EE emulation flash.

- Authenticate App? (7)

The Flash boot optionally authenticates a digital signature for the application image if the TOC2_FLAGS bit APP_AUTH_DISABLE = 1.

If this field is set then the authentication step is skipped.

- Is Public Key Valid (8)

The public key structure is filled by the user. Thus, it must be validated to ensure the correctness of the its entries before being used.

- Is Digital Signature Valid? (9)

The user application located in main flash is validated using the standard RSA 2048-bit algorithm (RSASSA-PKCS1-v1_5-2048). The public key used for this operation is stored in the user public key area of SFlash.

Boot code

The application can be in one of three formats, only two of which are validated using a public key.

The Crypto block is used for validation and is only enabled during the application validation process to conserve power.

Figure 15-6 shows the application validation flow.

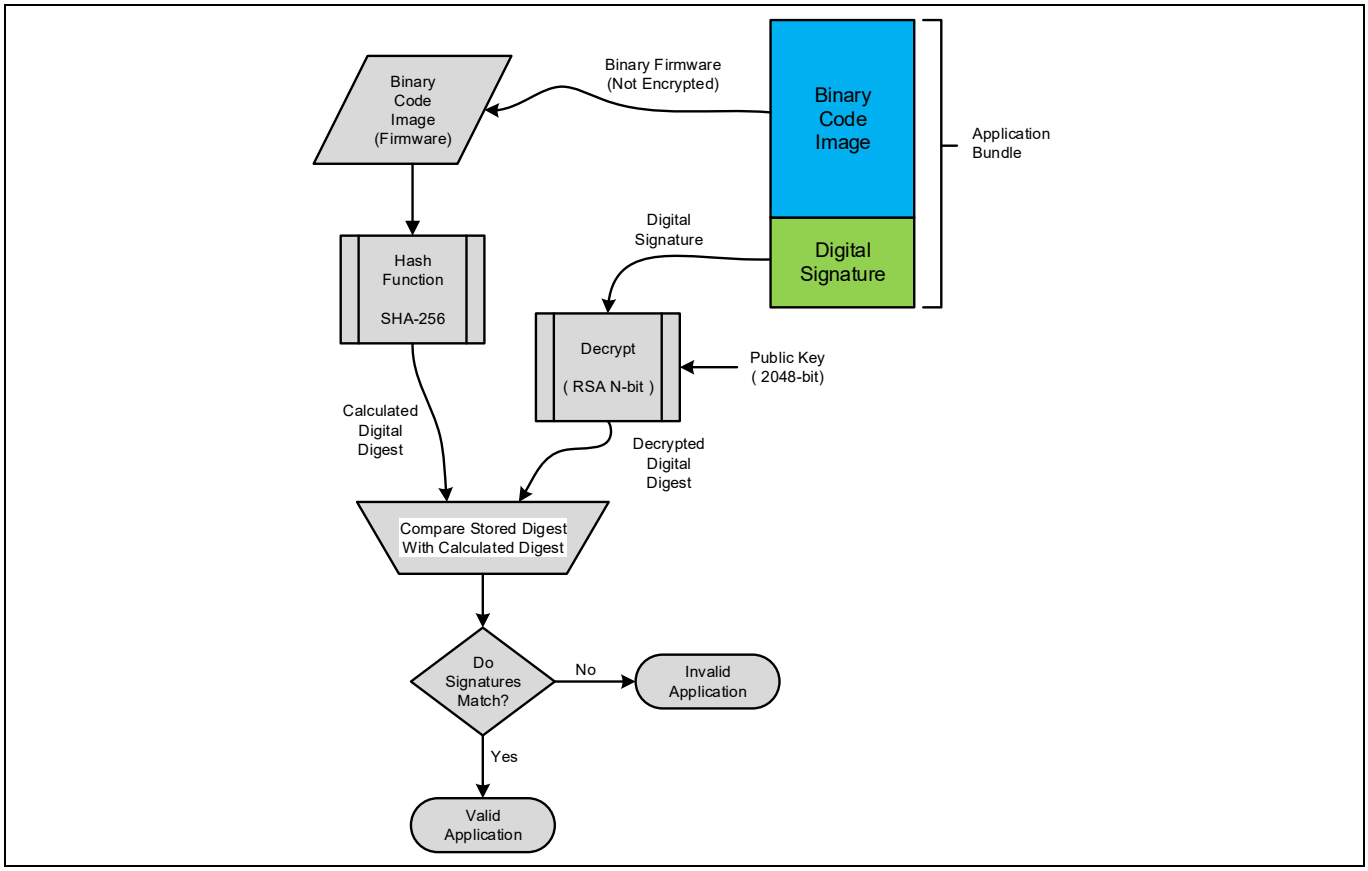


Figure 15-6. Application validation

- Enable System Calls (10)

At this point the system calls are enabled. A table of the system calls is found in **"SROM API library"** on page 211. The function calls are access via IPC that communicate to the CM0+ with interrupts. The SROM function EnableSystemCall() is called to enable these system calls.

- Is DAP Enabled (11)

Determines if the Debug Access Port (DAP) is enabled by reading the CPUSS_AP_CTL register, that was set during ROM BOOT. The DAP consist of three sections: CM4 AP, CM0+ AP, and System AP. If any of these APs are enabled, then the DAP is considered to be enabled.

- Configure SWD/JTAG Pins (12)

The SWD and JTAG GPIOs are configured to work with the DAP. The CPUSS_AP_CTL register will be used to set DAP.

- Wait Window (14)

The CPU delays execution to allow the debug hardware to acquire the CM0+. The default delay time is 20 ms, but the user can set other delay options. For a SECURE mode device, it is recommended to set the delay to 0 as debug will not be enabled.

This delay allows the debug hardware to acquire the debug interface and set the operation to test mode.

- Test Mode Enable? (15)

Boot code

After the listen window delay, the firmware checks if the SRSS_TST_MODE register has either TEST_MODE or TEST_KEY_DFT_EN bit set. If either bit is set, execution is transferred to an endless loop in SROM. This is done by calling the BusyWaitLoop() ROM boot function.

- Launch CM0+ Application (16)

If the TOC2 is erased and the life-cycle stage is NORMAL, Flash boot assumes that an application is in the “Basic Application Format” and the application start address is the start of the user flash.

If the TOC2 is valid and the life-cycle stage is NORMAL, then the application can be in either the Basic or Cypress Standard Secure application formats.

Otherwise, the application is stored in the Cypress Secure application format.

The procedure to launch a user application is:

1. Set CPUSS_CM0_VECTOR_TABLE_BASE to the start of the user application interrupt vector table.
2. Set CPUSS_CM4_VECTOR_TABLE_BASE to 0xFFFF_0000.
3. Perform a core reset.
4. After a core reset is performed ROM boot is launched (on CM0+).
5. ROM boot checks if CPUSS_PROTECTION ! = 0, which means ROM boot is launched on CM0+ after a core reset.
6. If (5) is true, ROM boot sets SP and PC register values from the user interrupt vector table. The address of a user application interrupt vector table is stored at step (1) to CPUSS_CM0_VECTOR_TABLE_BASE.
7. When ROM boot sets PC register value with the user reset handler address, user code starts executing.

- Set Up SP (17)

The SP register value for Flash boot is at the top of user RAM.

- Idle Loop (18)

Before going to an idle loop the Flash boot sets the CPUSS_CM0_VECTOR_TABLE_BASE MMIO register to 0xFFFF_0000.

In this state, a NMI interrupt can wake up the device. To prevent unwanted firmware access, device access restrictions must be set up correctly for the appropriate life cycle stage. The protection settings for each life-cycle stage are listed in **“Life-cycle stages and protection states”** on page 244.

- Set Error Code (30)

If the user application flash or the TOC2 was determined to be invalid, an error code will be written to IPC.DATA (structure 2); this will enable to detect the cause during debug.

Table 15-12. Error code

Error name	Value	Description
CY_FB_STATUS_SUCCESS	0xA100_0100	Success status value
CY_FB_STATUS_BUSY_WAIT_LOOP	0xA100_0101	Debugger probe acquired the device in Test Mode. The Flash boot entered a busy wait loop.
CY_FB_ERROR_INVALID_APP_SIGN	0xF100_0100	Application signature validation failed for the device families where flash boot launches only one application from TOC2. Either the application structure or a digital signature is invalid for the device families in which Flash boot may launch either of two applications in TOC2.
CY_FB_ERROR_INVALID_TOC	0xF100_0101	Empty or invalid TOC
CY_FB_ERROR_INVALID_KEY	0xF100_0102	Invalid public key
CY_FB_ERROR_UNREACHABLE	0xF100_0103	Unreachable code
CY_FB_ERROR_TOC_DATA_CLOCK	0xF100_0104	TOC contains invalid CM0+ clock attribute

Boot code

Table 15-12. Error code

Error name	Value	Description
CY_FB_ERROR_TOC_DATA_DELAY	0xF100_0105	TOC contains invalid listen window delay
CY_FB_ERROR_FLL_CONFIG	0xF100_0106	FLL configuration failed
CY_FB_ERROR_INVALID_APP0_DATA	0xF100_0107	Application structure is invalid for the device families in which Flash boot may launch only one application from TOC2
CY_FB_ERROR_CRYPTO	0xF100_0108	Error in Crypto operation
CY_FB_ERROR_INVALID_PARAM	0xF100_0109	Invalid parameter value
CY_FB_ERROR_BOOT_HARD_FAULT	0xF100_010a	A hard fault exception had happened in the Flash boot
CY_FB_ERROR_UNEXPECTED_INTERRUPT	0xF100_010B	Any unexpected interrupt had happened in the Flash boot
CY_FB_ERROR_BOOTLOADER	0xF100_0140	Any bootloader error
CY_FB_ERROR_BOOT_LIN_INIT	0xF100_0141	Bootloader error, LIN initialization failed
CY_FB_ERROR_BOOT_LIN_SET_CMD	0xF100_0142	Bootloader error, LinSetCmd() failed

- Protection = Virgin? (31)

The CPUSS_PROTECTION MMIO register value is compared to the desired protection mode.

- Life Cycle = SECURE (32)

The life-cycle stage value is stored in eFuse. To determine this value, Flash boot reads the corresponding eFuse bits. If the SECURE life-cycle bit is set in eFuse, then the device is in the SECURE life-cycle stage. Otherwise, the device is in another life-cycle stage.

Note: Life-cycle stage is not the same as protection mode. In this case, SECURE_WITH_DEBUG is not equal to SECURE life-cycle stage; however, the protection state equals to SECURE for both these stages.

- Protection = DEAD (33)

In DEAD protection state, the system will not attempt to start any flash firmware and will restrict access to system resources to shield secure/sensitive information. The intention is to allow a controlled set of failure analysis capabilities only. By default, DEAD mode is open to all debug functions and users should change to the level of security required.

- Deploy Access Restrictions (34)

SECURE_DEAD access restrictions are applied for entering DEAD branch from the SECURE life-cycle stage.

Assess restrictions are applied by calling the RestrictAccess() ROM boot function.

- Set Up DAP from AR (35)

The ROM boot function GetAccessRestrictStruct() is called to determine which APs (access points, one of CM0+ AP, CM4, and TC) are enabled. Based on this information the proper values are written to the CPUSS_AP_CTL register.

- Apply System Protection (36)

System protection settings are usually applied by ROM boot code before entering Flash boot. If the device enters the DEAD branch, the system protection settings are changed. Thus, Flash boot calls the ROM boot function ApplyProtectionSettings() to reconfigure them.

16 Device security

The EZ-USB™ FX10 MCU offers several features to protect user designs from unauthorized access or copying. Selecting a secure life cycle stage, enabling flash protection, and using hardware-based encryption can provide a high level of security.

16.1 Features

The EZ-USB™ FX10 MCU provides the following device security features:

- Nonvolatile and irreversible life cycle stages that can limit program and debug access
- Shared memory protection unit (SMPU) that provides programmable flash, SRAM, and register protection
- Cryptographic function block that provides hardware-based encryption and decryption of data and code

16.2 Architecture

16.2.1 Life cycle stages and protection sates

The EZ-USB™ FX10 MCUs have configurable, nonvolatile life cycle stages. Life cycle stages follow a strict, irreversible progression governed by writing to eFuse – a 1024-bit nonvolatile memory with each bit being one time programmable (OTP). The eFuse can hold unalterable keys and trim information. This chapter discusses hardware blocks used to implement device security.

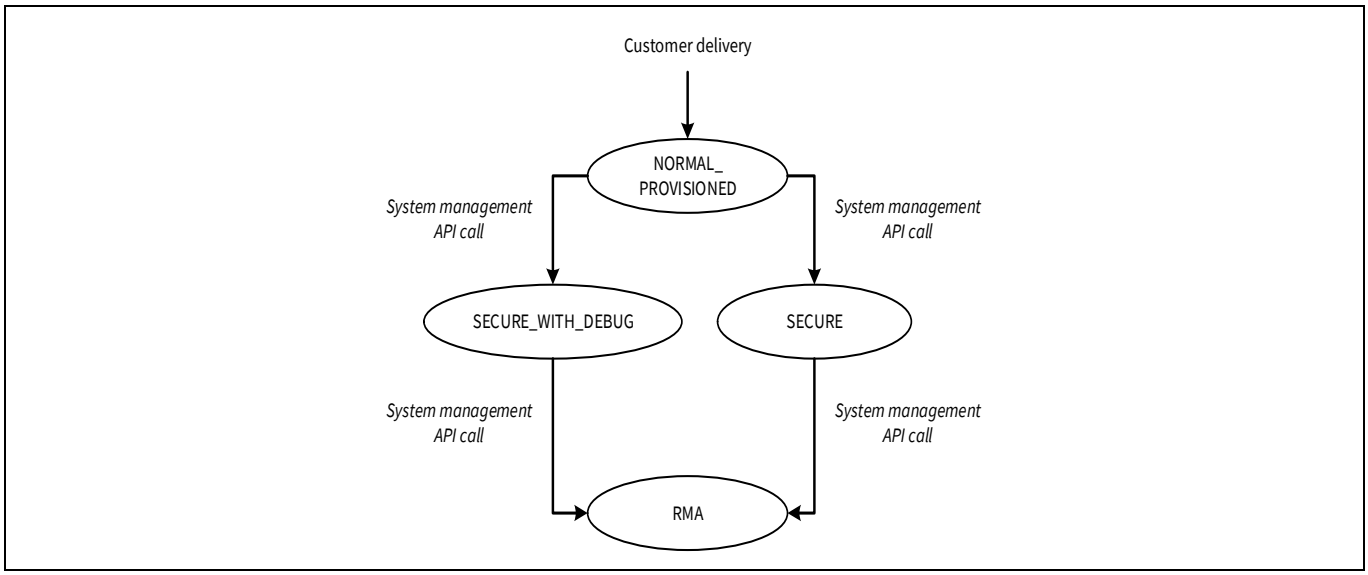


Figure 16-1. Life cycle stage transitions

Device security

The EFUSE_DATA_LIFECYCLE_STAGE eFuse governs the life-cycle stages.

The EZ-USB™ FX10 MCU supports the following life cycle stages:

- VIRGIN – This stage is used by Infineon during assembly and testing. During this stage, trim values and flash boot are written into SFlash. Devices that are in this stage never leave the factory. In this stage, the boot ROM assumes that no other eFuse data or flash data is valid. Devices are transitioned to the Normal stage before they leave the factory.
- NORMAL – This is the life cycle stage of a device after trimming and testing is complete in the factory. All configuration and trimming information is complete. Valid flash boot code is programmed in the SFlash. To allow the OEM to check data integrity of trims, flash boot, and other objects from the factory, a hash (SHA-256 truncated to 128 bits) of these objects is stored in eFuse. This hash is referred to as Factory_HASH. In Normal mode, by default, there is full debug access to the CPUs and system. The device may be erased and programmed as needed.
- SECURE – This is the stage of a secure device. Before transitioning to this stage, ensure that the following steps are complete:
  - Secure access restrictions are programmed into the EFUSE_DATA_SECURE_ACCESS_RESTRICT0 and EFUSE_DATA_SECURE_ACCESS_RESTRICT1 registers
  - The TOC2 should optionally be programmed with the correct address of the user key storage blocks in flash
  - The TOC2 should be programmed with the user application format, the starting addresses of the user application, and the address of the public key in SFlash
  - Protection units should be set up to protect sensitive code regions
  - Public and private keys must be generated

    In this stage, the protection state is set to Secure, and Secure DAP access restrictions are deployed. A secure device will boot only after successful authentication of its flash boot and application code.

    After an MCU is in the Secure life cycle stage, it cannot go back to the Normal stage. SFlash cannot be programmed in this stage. The debug ports may be disabled depending on user preferences, so it is not possible to reprogram or erase the device with a hardware programmer/debugger. At this point, the firmware can only be updated by invoking a bootloader, which must be provided as part of device firmware. Access restrictions in the SECURE state can be controlled by writing to the following eFuses: EFUSE_DATA_SECURE_ACCESS_RESTRICT0 and EFUSE_DATA_DEAD_ACCESS_RESTRICT1.

    Code should be tested in Normal and Secure with Debug stages before advancing to this stage. This is to prevent a configuration error that can cause the device to be inaccessible for programming and therefore, become unusable.
- SECURE_WITH_DEBUG – This is similar to the Secure stage, except that Normal access restrictions are applied to enable debugging. Devices that are in the Secure with Debug stage cannot be changed back to either Secure or Normal stage.

Secure applications may also be developed in the Normal life-cycle stage with code validation enabled. This requires the TOC2_FLAGS.APP_AUTH_DISABLE bits at offset 0x1F8 to be set. See the **"Boot code"** on page 240 for more information. This method has the advantage of being able to transition to the SECURE life-cycle stage after testing is complete.

- RMA – Customers can transition the device to the RMA stage (from Secure) when they want Infineon to perform failure analysis on the device. The customer should erase all sensitive data (including firmware) before invoking the system call that transitions the device to RMA. Flash erase should be performed by calling EraseRow/EraseSubSector/EraseSector/EraseAll SROM APIs five times repeatedly on the secure FLASH region before converting the device to RMA. Erasing the device five times eliminates any possibility of determining the previous state of any flash cell. See the **“Nonvolatile memory”** on page 205 for more details on these SROM APIs.

  When invoking the system call to transition to RMA, the customer must provide a certificate that authorizes Infineon to transition the device with a specific Unique ID to the RMA life cycle stage. The certificate will be signed by the customer using the same private key that is used to sign the user application image. The verification of the signature uses the same algorithm used by flash boot to authenticate the user application. The same public key (injected by the OEM) stored in SFlash is used for the verification. Note that the signature is unique to a specific device. A signature will only work with the specific device for which it was generated because it uses the unique device ID.

  When a device is reset in the RMA life cycle stage, the boot will set access restrictions such that the DAP has access only to System AP and IPC MMIO registers for making system calls, and adequate RAM for communication. It will then wait for the “Open for RMA” system call from the DAP along with the certificate of authorization. This is the same certificate used in transitioning the device to RMA. The boot process will not initiate any firmware until it successfully executes the “Open for RMA” command. After the command is successful, the device will behave as though it were in the Virgin life cycle stage, but it cannot be transitioned to Virgin or any other stage. The life cycle stage stored in eFuse cannot be changed from RMA. Every time the device is reset, it must execute the “Open for RMA” command successfully before the device can be used

Device security

A powered EZ-USB™ FX10 MCU also has a volatile protection state that reflects its life cycle stage. The protection state is determined on boot by reading the eFuse values. **Figure 16-2** illustrates the protection state transitions.

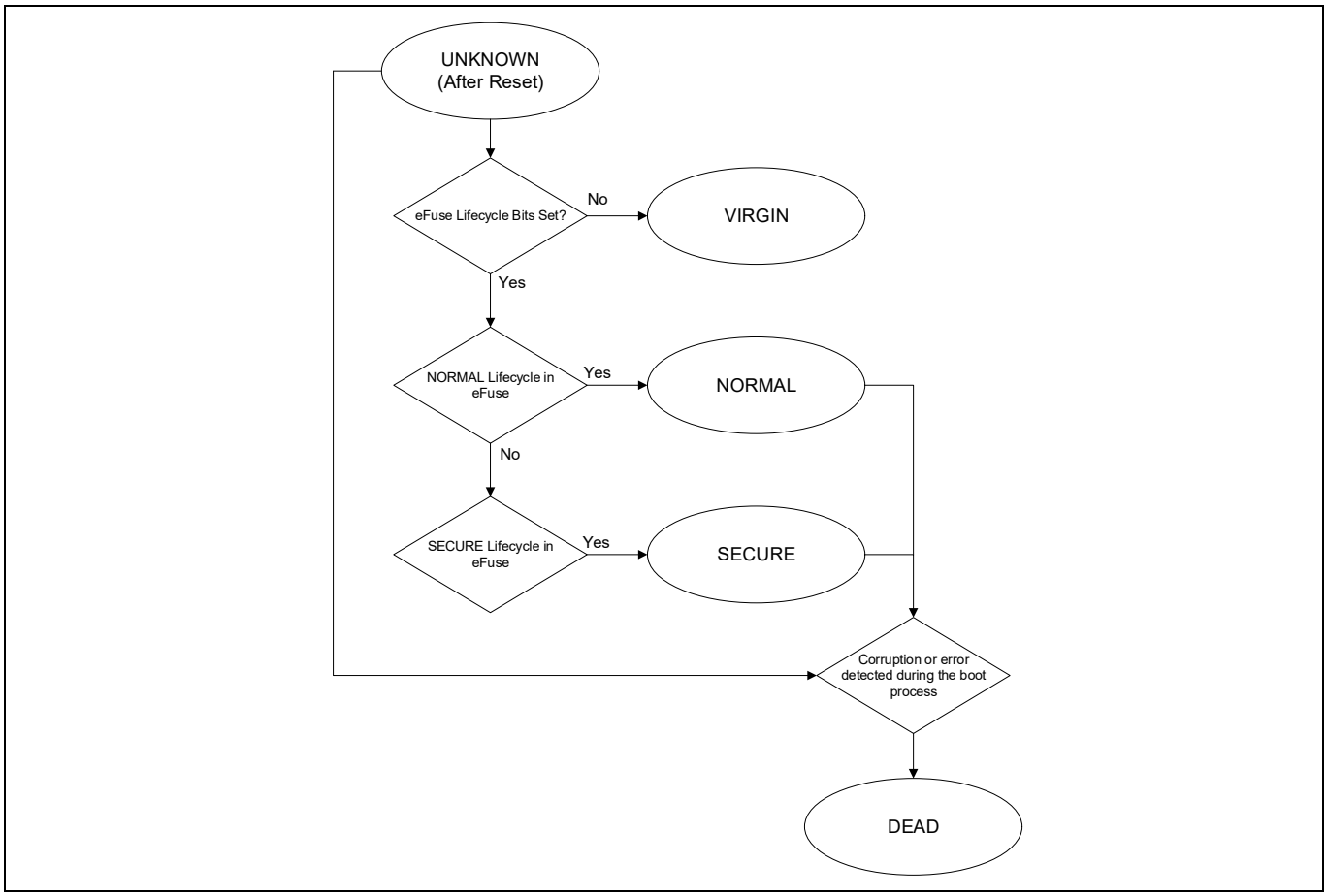


Figure 16-2. Protection state transitions

Protection state is defined by the STATE field of the CPUSS_PROTECTION register.

- STATE is 0: UNKNOWN state
- STATE is 1: VIRGIN state
- STATE is 2: NORMAL state
- STATE is 3: SECURE state
- STATE is 4: DEAD state

The CPUSS_PROTECTION register can be written only to affect the transitions defined in **Figure 16-2**. Any value written to this register that does not represent a transition in **Figure 16-2** is rejected in hardware. A life cycle stage change (by writing to eFuse) does not immediately affect a protection state change. After writing the NORMAL, SECURE_WITH_DEBUG, or SECURE state in the eFuse, a reboot is required for the change of life cycle stage to take effect on the protection state.

There are more protection states than there are lifecycle stages. The additional protection states are:

- UNKNOWN – A device comes out of reset in this state. The ROM boot code needs to run through its courses (reading eFuses and checking integrity) to determine the life cycle stage of the device and select the proper protection state.
- DEAD – A device will enter the DEAD protection state when a corruption/error is detected in the boot process. In DEAD protection state, the system will not attempt to start any flash firmware and may restrict access to system resources to shield secure/sensitive information. Access restrictions in DEAD mode are controlled by EFUSE_DATA_DEAD_ACCESS_RESTRICT0 and EFUSE_DATA_DEAD_ACCESS_RESTRICT1 eFuses.

16.2.2 Flash security

The EZ-USB™ FX10 MCUs include a flexible flash-protection system that controls access to flash memory. This feature is designed to secure proprietary code, but it can also be used to protect against inadvertent writes to the bootloader portion of flash.

Flash memory is organized in sectors. Flash protection is provided by a shared memory protection unit (SMPU). The SMPU is intended to distinguish between different protection contexts and to distinguish secure from non-secure accesses. The system function that performs flash programming first looks at the SMPU settings and will not allow to program or erase flash blocks protected by the SMPUs.

For more details, see the **“Protection units”** on page 110.

16.2.3 Hardware-based encryption

The EZ-USB™ FX10 MCU has a cryptographic block (Crypto) that provides hardware implementation and acceleration of cryptographic functions. It implements symmetric key encryption and decryption, hashing, message authentication, random number generation (pseudo and true), and cyclic redundancy checking. It can work with internal as well as external memory. See the **“Cryptographic function block (CRYPTO)”** on page 154 for more details.

17 Program and debug interface

The EZ-USB™ FX10 MCU Program and Debug interface provides a communication gateway for an external device to perform programming or debugging. The external device can be a Infineon-supplied programmer and debugger, or a third-party device that supports programming and debugging. The serial wire debug (SWD) or the JTAG interface can be used as the communication protocol between the external device and EZ-USB™ FX10 MCUs.

17.1 Features

- Supports programming and debugging through the JTAG or SWD interface
- CM4 supports 4-bit ETM tracing, serial wire viewer (SWV), and printf() style debugging through the single-wire output (SWO) pin. CM0+ supports Micro Trace Buffer (MTB) with 4 KB dedicated RAM.
- Supports Cross Triggering Interface (CTI) and Cross Triggering Matrix (CTM)
- CM0+ supports four hardware breakpoints and two watchpoints. CM4 supports six hardware breakpoints and four watchpoints.
- Provides read and write access to all memory and registers in the system while debugging, including the Cortex-M4 and Cortex-M0+ register banks when the core is running or halted

17.2 Architecture

Figure 17-1 shows the block diagram of the program and debug interface in the EZ-USB™ FX10 MCU. The debug and access port (DAP) acts as the program and debug interface. The external programmer or debugger, also known as the “host”, communicates with the DAP of the EZ-USB™ FX10 MCU “target” using either the SWD or JTAG interface. The debug physical port pins communicate with the DAP through the high-speed I/O matrix (HSIOM). See the **“I/O system”** on page 332 for details on HSIOM.

The debug infrastructure is organized in the following four groups:

- DAP (provides pin interfaces through which the debug host can connect to the chip)
- Cortex-M0+ core debug components
- Cortex-M4 core debug components
- Other debug infrastructure (includes the CM4 tracing, the CTM, and the System ROM table)

Program and debug interface

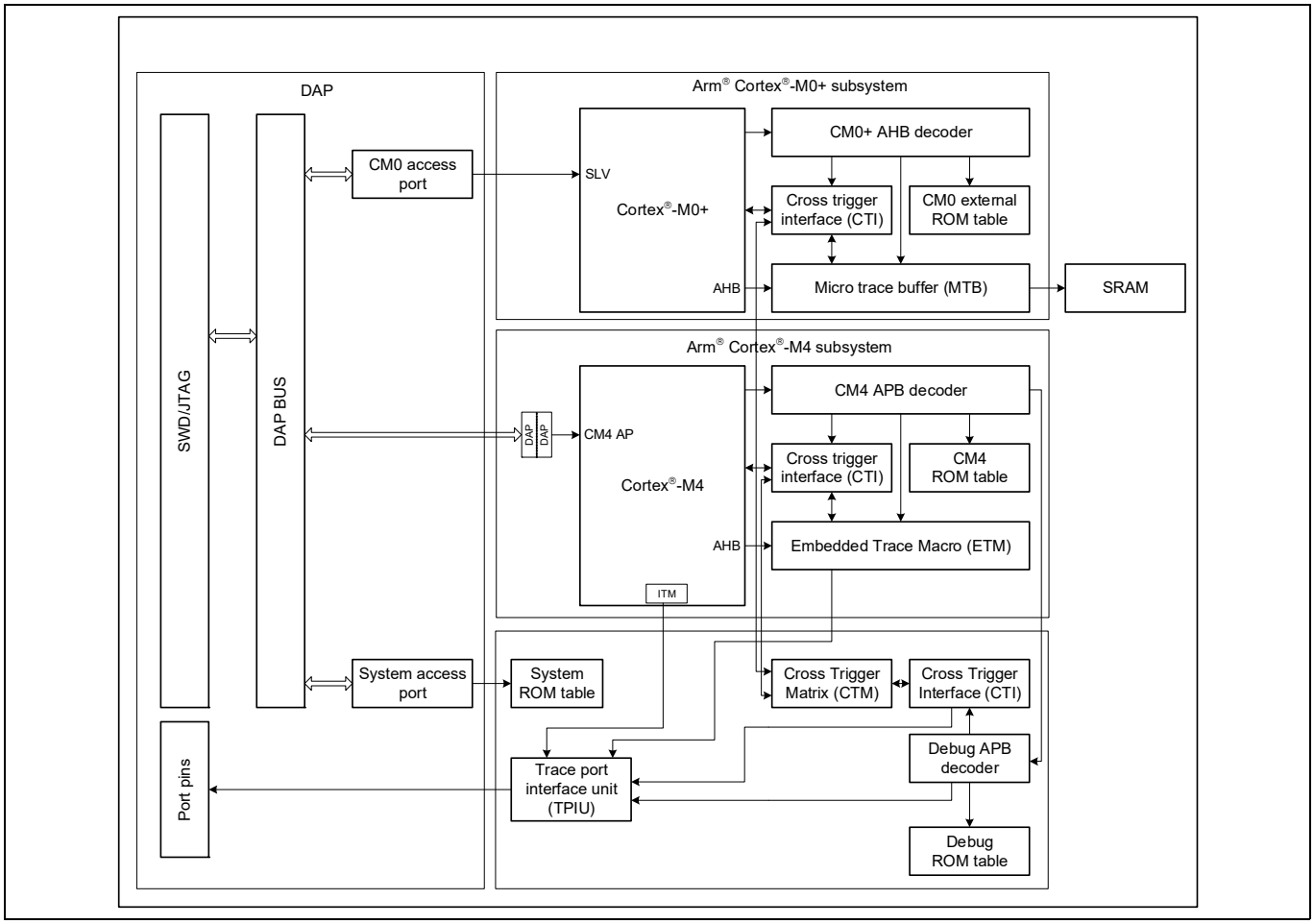


Figure 17-1. Program and debug interface

The DAP communicates with the Cortex®-M0+ CPU using the Arm®-specified advanced high-performance bus (AHB) interface. AHB is the systems interconnect protocol used inside the device, which facilitates memory and peripheral register access by the AHB master. The EZ-USB™ FX10 MCU has six AHB masters – Arm® CM4 CPU core, Arm® CM0 CPU core, Datawire0, Datawire1, Crypto, and DAP. The external host can effectively take control of the entire device through the DAP to perform 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
  - Micro trace buffer (MTB-M0+) for tracing Cortex-M0+ program execution
  - Embedded trace macrocell (ETM-M4) for tracing Cortex-M4 program execution
- Trace sink components
  - Trace port interface unit (TPIU) to drive the trace information out of the chip to an external trace port analyzer
- Cross-triggering components
  - Cross-trigger interface (CTI)
  - Cross-trigger matrix (CTM)
- ROM tables

17.2.1 Debug access port (DAP)

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

The debug port (DP) connects to the DAP bus, which in turn connects to one of three Access Ports (AP), namely:

- The CM0-AP, which connects directly to the AHB debug slave port (SLV) of the CM0+ and gives access to the CM0+ internal debug components. This also allows access to the rest of the system through the CM0+ AHB master interface. This provides the debug host the same view as an application running on the CM0+. This includes access to the MMIO of other debug components of the Cortex®-M0+ subsystem. These debug components can also be accessed by the CM0+ CPU, but cannot be reached through the other APs or by the CM4 core.
- The CM4-AP located inside the CM4 gives access to the CM4 internal debug components. The CM4-AP also allows access to the rest of the system through the CM4 AHB master interfaces. This provides the debug host the same view as an application running on the CM4 core. Additionally, the CM4-AP provides access to the debug components in the CM4 core through the External Peripheral Bus (EPB). These debug components can also be accessed by the CM4 CPU, but cannot be reached through the other APs or by the CM0+ core.
- The System-AP, which through an AHB mux gives access to the rest of the system. This allows access to the System ROM table, which cannot be reached any other way. The System ROM table provides the chip ID but is otherwise empty.

17.2.1.1 DAP security

For security reasons all three APs each can be independently disabled. Each AP disable is controlled by two MMIO bits. One bit, CPUSS_AP_CTL.xxx_DISABLE (where xxx can be CM0 or CM4 or SYS), can be set during boot, before the debugger can connect, based on eFuse settings. After this bit is set it cannot be cleared.

The second bit, CPUSS_AP_CTL.xxx_ENABLE, is a regular read/write bit. This bit also resets to zero and is set to '1' by either the ROM boot code or the flash boot code depending on the life-cycle stage. This feature can be used to block debug access during normal operation, but re-enable some debug access after a successful authentication.

In addition to the above, the System AP is also protected by an MPU. This can be used to give the debugger limited access to the rest of the system. Allow access to the System ROM table for chip identification. If debug access is restored after successful authentication, this MPU must be configured to allow authentication requests.

Note: The debug slave interfaces of both the CPUs bypass the internal CPU MPU.

17.2.1.2 DAP power domain

Almost all the debug components are part of the Active power domain. The only exception is the SWD/JTAG-DP, which is part of the Deep Sleep power domain. This allows the debug host to connect during Deep Sleep, while the application is 'running' or powered down. This enables in-field debugging for low-power applications in which the chip is mostly in Deep Sleep.

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

Note that only the two SWD pins (SWCLKTCK and SWDIOTMS) are operational during the Deep Sleep mode – the JTAG pins are operational only in Active mode. The JTAG debug and JTAG boundary scan are not available when the system is in Deep Sleep mode. JTAG functionality is available only after a chip power-on-reset.

17.2.2 ROM tables

The ROM tables are organized in a tree hierarchy. Each AP has a register that contains a 32-bit address pointer to the base of the root ROM table for that AP. For EZ-USB™ FX10 MCUs, there are three such root ROM tables.

Each ROM table contains 32-bit entries with an address pointer that either points to the base of the next level ROM table. Each ROM table also contains a set of ID registers that hold JEDEC compliant identifiers to identify the manufacturer, part number, and major and minor revision numbers. For all ROM tables in EZ-USB™ FX10 MCUs, these IDs are the same. Each ROM table and CoreSight compliant component also contains component identification registers.

17.2.3 Trace

The micro trace buffer (MTB-M0+) component captures the program execution flow from Cortex-M0+ CPU and stores it in a local SRAM memory. This information can be read by an external debug tool through JTAG/SWD interface to construct the program execution flow.

The embedded trace macro (ETM) component connected to Cortex-M4 captures the program execution flow from Cortex-M4 CPU and generates trace output on its advanced trace bus (ATB) interface. The instrumentation trace macrocell (ITM), which is inside Cortex-M4, also generates trace output on its ATB interface. These two ATB interfaces (from ETM-M4 and ITM) are connected a trace port interface unit (TPIU).

The TPIU drives the external pins of a trace port (through IOSS interface), so that the trace can be captured by an external trace port analyzer (TPA). For more details, refer to the Arm® Debug Interface Architecture Specification ADIv5.0 to ADIv5.2.

17.2.4 Embedded cross triggering

The Arm® CoreSight includes embedded cross triggering (ECT) to communicate events between debug components. These events are particularly useful with tracing and multicore platforms. 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 trace buffer being full or not or based on other events

CoreSight uses two components to support ECT, namely a CTI and a CTM, both of which are used in EZ-USB™ FX10 MCUs.

The CTI component interfaces with other debug components, sending triggers back and forth and synchronizing them as needed. The CTM connects several CTIs, thus allowing events to be communicated from one CTI to another.

The EZ-USB™ FX10 MCU has three CTIs, one for each CPU and one for the trace components in the debug structure. These three CTIs are connected together through the CTM. The CM4 CTI is located in the fast clock domain and the other two CTIs and the CTM are all located in the same slow-frequency clock domain. For more details, refer to the Arm® documentation.

17.3 Serial wire debug (SWD) interface

The EZ-USB™ FX10 MCU supports programming and debugging through the SWD interface. The SWD protocol is a packet-based serial transaction protocol. At the pin level, it uses a single bidirectional data signal (SWDIO) and a unidirectional clock signal (SWDCK). The host programmer always drives the clock line, whereas either the host or the target drives the data line. A complete data transfer (one SWD packet) requires 46 clocks and consists of three phases:

- **Host packet request phase** – The host issues a request to the EZ-USB™ FX10 MCU target
- **Target acknowledge response phase** – The EZ-USB™ FX10 MCU target sends an acknowledgement to the host
- **Data transfer phase** – The host or target writes data to the bus, depending on the direction of the transfer

When control of the SWDIO line passes from the host to the target, or vice versa, there is a turnaround period (Trn) where neither device drives the line and it floats in a high-impedance (Hi-Z) state. This period is either one-half or one and a half clock cycles, depending on the transition.

Figure 17-2 shows the timing diagrams of read and write SWD packets.

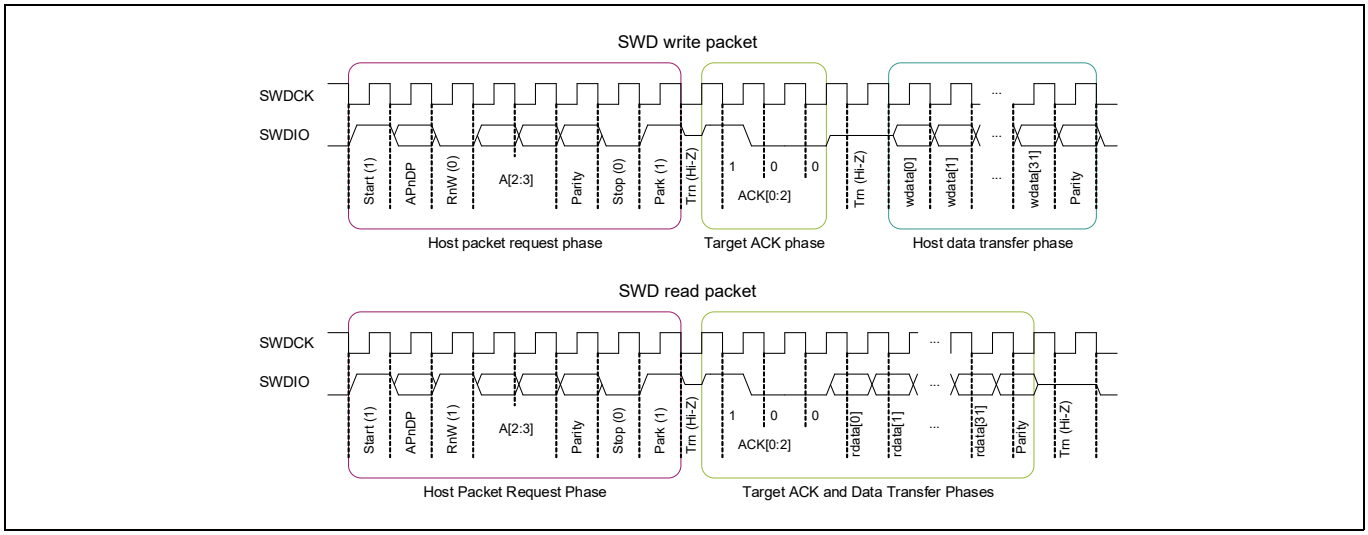


Figure 17-2. SWD write and read packet timing diagrams

The sequence to transmit SWD read and write packets are as follows:

- Host packet request phase: SWDIO driven by the host
  - The start bit initiates a transfer; it is always logic 1
  - The AP not DP (APnDP) bit determines whether the transfer is an AP access – 1b or a DP access – 0b
  - The Read not Write bit (RnW) controls which direction the data transfer is in. 1b represents a 'read from' the target, or 0b for a 'write to' the target.
  - The Address bits (A[3:2]) are register select bits for AP or DP, depending on the APnDP bit value.
    Note: Address bits are transmitted with the LSb first.
  - The parity bit contains the parity of APnDP, RnW, and ADDR bits. It is an even parity bit; this means, when XORed with the other bits, the result will be 0.
    If the parity bit is not correct, the header is ignored by the EZ-USB™ FX10 MCU; there is no ACK response (ACK = 111b). The programming operation should be aborted and retried again by following a device reset.
  - The stop bit is always logic 0.
  - The park bit is always logic 1.
- Target Acknowledge Response Phase: SWDIO driven by the target

  - The ACK[2:0] bits represent the target to host response, indicating failure or success, among other results. See **Table 17-1** for definitions.
    Note: ACK bits are transmitted with the LSb first.
- Data Transfer Phase: SWDIO driven by either target or host depending on direction
  - The data for read or write is written to the bus, LSb first.
  - The data parity bit indicates the parity of the data read or written. It is an even parity; this means when XORed with the data bits, the result will be '0'.
    If the parity bit indicates a data error, corrective action should be taken. For a read packet, if the host detects a parity error, it must abort the programming operation and restart. For a write packet, if the target detects a parity error, it generates a FAULT ACK response in the next packet.

According to the SWD protocol, the host can generate any number of SWDCK clock cycles between two packets with SWDIO low. Three or more dummy clock cycles should be generated between two SWD packets if the clock is not free-running or to make the clock free-running in IDLE mode.

The SWD interface can be reset by clocking the SWDCK line for 50 or more cycles with SWDIO high. To return to the idle state, clock the SWDIO low once.

17.3.1 SWD timing details

The SWDIO line is written to and read at different times depending on the direction of communication. The host drives the SWDIO line during the host packet request phase and, if the host is writing data to the target, during the Data Transfer phase as well. When the host is driving the SWDIO line, each new bit is written by the host on falling SWDCK edges, and read by the target on rising SWDCK edges. The target drives the SWDIO line during the Target Acknowledge Response phase and, if the target is reading out data, during the Data Transfer phase as well. When the target is driving the SWDIO line, each new bit is written by the target on rising SWDCK edges, and read by the host on falling SWDCK edges.

Table 17-1 and **Figure 17-2** illustrate the timing of SWDIO bit writes and reads.

Table 17-1. SWDIO bit write and read timing

SWD packet phase	SWDIO edge	
	Falling	**Rising**
Host packet request	Host write	Target read
Host data transfer		
Target ack response	Host read	Target write
Target data transfer		

17.3.2 ACK details

The acknowledge (ACK) bitfield is used to communicate the status of the previous transfer. OK ACK means that previous packet was successful. A WAIT response requires a data phase. For a FAULT status, the programming operation should be aborted immediately. **Table 17-2** shows the ACK bit-field decoding details.

Table 17-2. SWD transfer ACK response decoding

Response	ACK[2:0]
OK	001b
WAIT	010b
FAULT	100b
NO ACK	111b

Details on WAIT and FAULT response behaviors are as follows:

- For a WAIT response, if the transaction is a read, the host should ignore the data read in the data phase. The target does not drive the line and the host must not check the parity bit as well.
- For a WAIT response, if the transaction is a write, the data phase is ignored by the EZ-USB™ FX10 MCU. But, the host must still send the data to be written to complete the packet. The parity bit corresponding to the data should also be sent by the host.
- A WAIT response means that the EZ-USB™ FX10 MCU is processing the previous transaction. The host can try for a maximum of four continuous WAIT responses to see whether an OK response is received. If it fails, then the programming operation should be aborted and retried again.
- For a FAULT response, the programming operation should be aborted and retried again by doing a device reset

17.3.3 Turnaround (Trn) period details

There is a turnaround period between the packet request and the ACK phases, as well as between the ACK and the data phases for host write transfers, as shown in **Figure 17-2**. According to the SWD protocol, the Trn period is used by both the host and target to change the drive modes on their respective SWDIO lines. During the first Trn period after the packet request, the target starts driving the ACK data on the SWDIO line on the rising edge of SWDCK. This action ensures that the host can read the ACK data on the next falling edge. Thus, the first Trn period lasts only one-half cycle. The second Trn period of the SWD packet is one and a half cycles. Neither the host nor the EZ-USB™ FX10 MCU should drive the SWDIO line during the Trn period.

17.4 JTAG interface

In response to higher pin densities on ICs, the Joint Test Action Group (JTAG) proposed a method to test circuit boards by controlling the pins on the ICs (and reading their values) via a separate test interface. The solution, later formalized as IEEE Standard 1149.1-2001, is based on the concept of a serial shift register routed across all of the pins of the IC – hence the name "boundary scan." The circuitry at each pin is supplemented with a multipurpose element called a boundary scan cell. In EZ-USB™ FX10 MCUs, most GPIO port pins have a boundary scan cell associated with them (see the GPIO block diagrams in the **"I/O system"** on page 332). The interface used to control the values in the boundary scan cells is called the Test Access Port (TAP) and is commonly known as the JTAG interface. It consists of three signals: Test Data In (TDI), Test Data Out (TDO), and Test Mode Select (TMS). Also included is a clock signal (TCK) that clocks the other signals. TDI, TMS, and TCK are all inputs to the device and TDO is the output from the device. This interface enables testing multiple ICs on a circuit board, in a daisy-chain fashion, as shown in **Figure 17-3**.

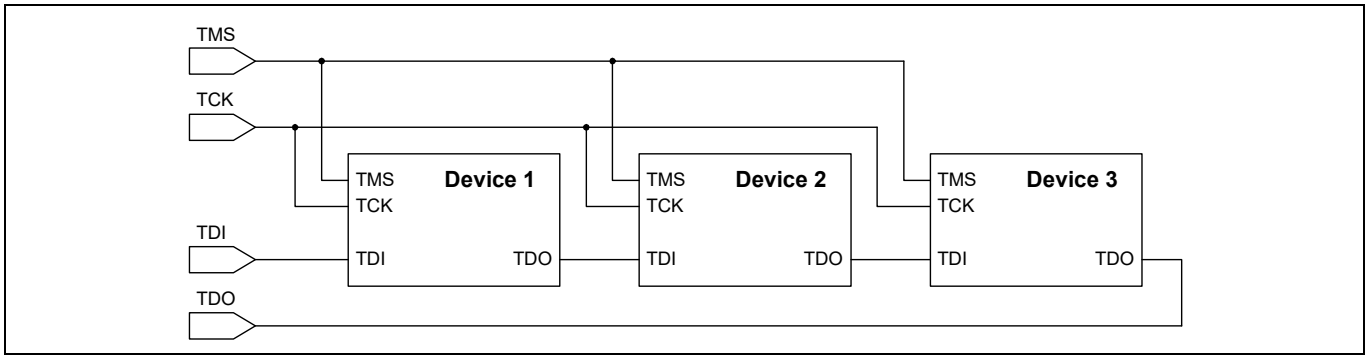


Figure 17-3. JTAG interface to multiple ICs on a circuit board

The JTAG interface architecture within each device is shown in **Figure 17-4**. Data at TDI is shifted in, through one of several available registers, and out to TDO.

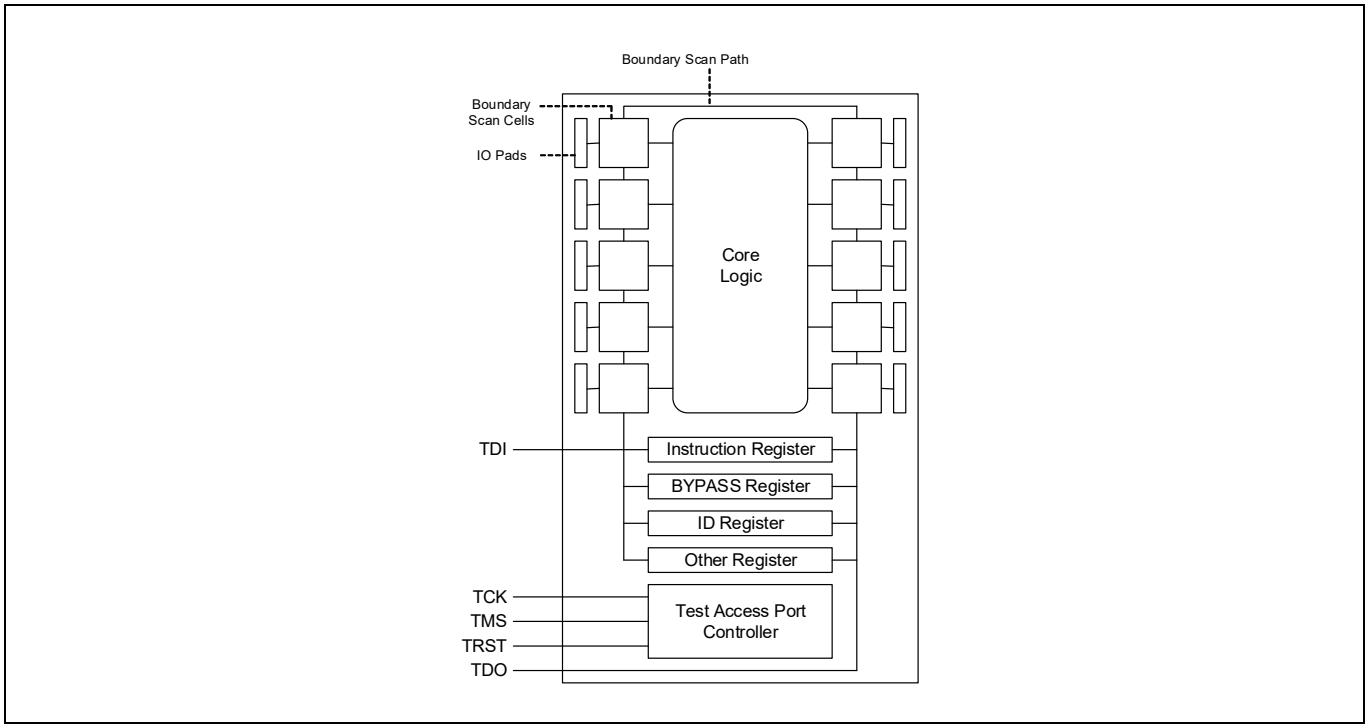


Figure 17-4. JTAG interface architecture

Program and debug interface

The TMS signal controls a state machine in the TAP. The state machine controls which register (including the boundary scan path) is in the TDI-to-TDO shift path, as shown in **Figure 17-5**. The following terms apply:

- IR - the instruction register
- DR - one of the other registers (including the boundary scan path), as determined by the contents of the instruction register
- capture - transfer the contents of a DR to a shift register, to be shifted out on TDO (read the DR)
- update - transfer the contents of a shift register, shifted in from TDI, to a DR (write the DR)

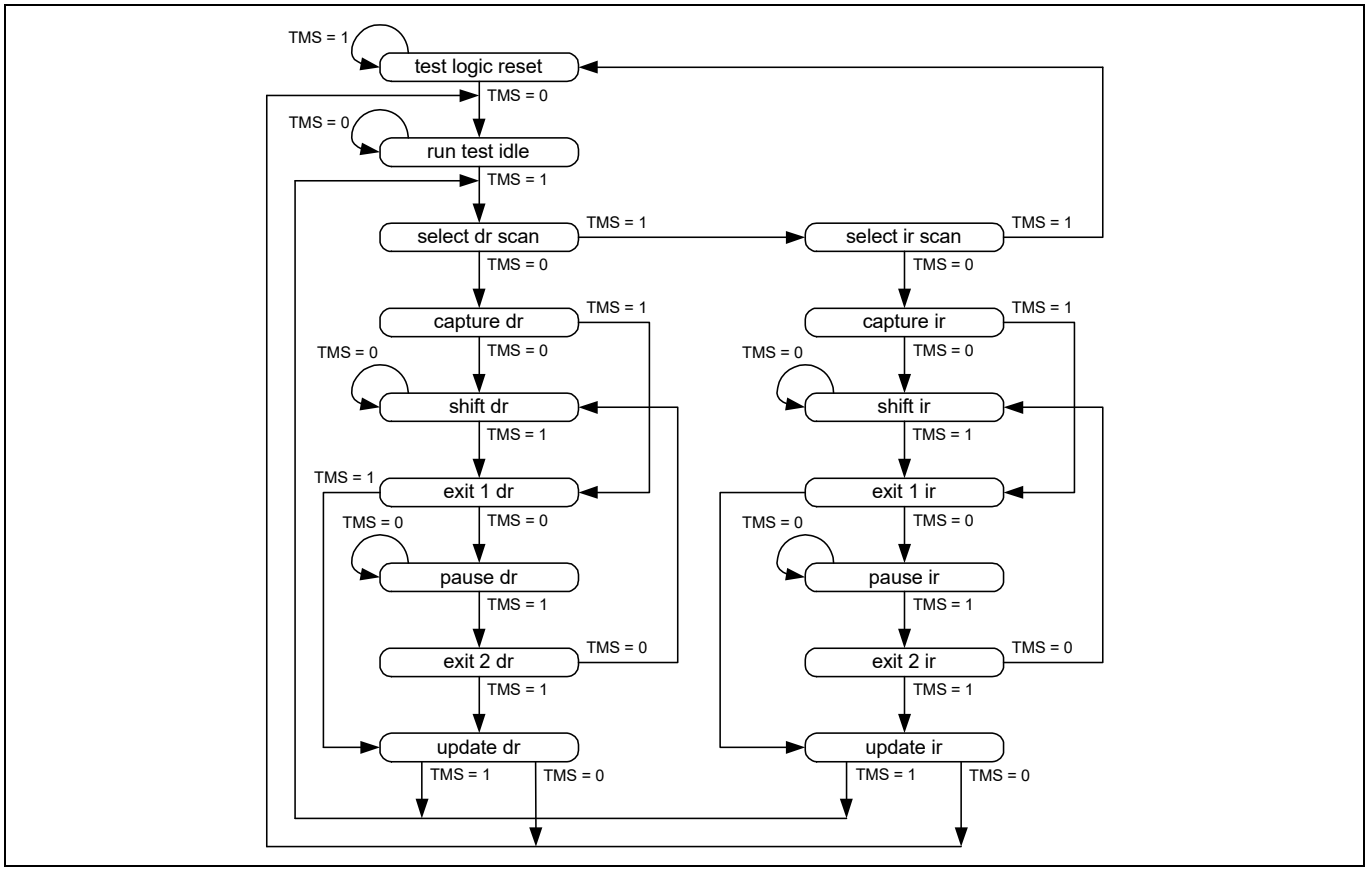


Figure 17-5. TAP state machine

The registers in the TAP are:

- Instruction – Typically two to four bits wide, holds the current instruction that defines which data register is placed in the TDI-to-TDO shift path
- Bypass – one bit wide, directly connects TDI with TDO, causing the device to be bypassed for JTAG purposes
- ID – 32 bits wide, used to read the JTAG manufacturer/part number ID of the device
- Boundary Scan Path (BSR) – Width equals the number of I/O pins that have boundary scan cells, used to set or read the states of those I/O pins

Other registers may be included in accordance with the device manufacturer specifications. The standard set of instructions (values that can be shifted into the instruction register), as specified in IEEE 1149, are:

- EXTEST – Causes TDI and TDO to be connected to the BSR. The device is changed from its normal operating mode to a test mode. Then, the device's pin states can be sampled using the capture dr JTAG state, and new values can be applied to the pins of the device using the update dr state.
- SAMPLE – Causes TDI and TDO to be connected to the BSR, but the device remains in its normal operating mode. During this instruction, the BSR can be read by the capture dr JTAG state to take a sample of the functional data entering and leaving the device.

- PRELOAD – Causes TDI and TDO to be connected to the BSR, but the device is left in its normal operating mode. The instruction is used to preload test data into the BSR before loading an EXTEST instruction.

Optional, but commonly available, instructions are:

- IDCODE – Causes TDI and TDO to be connected to an IDCODE register
- INTEST – Causes TDI and TDO to be connected to the BSR. While the EXTEST instruction allows access to the device pins, INTEST enables similar access to the corelogic signals of a device

For more information, see the IEEE Standard, available at **www.ieee.org**.

17.5 Programming the EZ-USB™ FX10 MCU

The EZ-USB™ FX10 MCU is programmed using the following sequence.

1. Acquire the SWD port in the EZ-USB™ FX10 MCU.
2. Enter the programming mode.
3. Execute the device programming routines such as Silicon ID check, flash programming, flash verification, and checksum verification.

17.5.1 SWD port acquisition

17.5.1.1 SWD port acquire sequence

The first step in device programming is for the host to acquire the target's SWD port. The host first performs a device reset by asserting the external reset (XRES) pin. After removing the XRES signal, the host must send an SWD connect sequence for the device within the acquire window to connect to the SWD interface in the DAP.

The debug access port must be reset using the standard Arm® command. The DAP reset command consists of more than 49 SWDCK clock cycles with SWDIO asserted HIGH. The transaction must be completed by sending at least one SWDCK clock cycle with SWDIO asserted LOW. This sequence synchronizes the programmer and the chip. Read_DAP() refers to the read of the IDCODE register in the debug port. The sequence of line reset and IDCODE read should be repeated until an OK ACK is received for the IDCODE read or a timeout (2 ms) occurs. The SWD port is said to be in the acquired state if an OK ACK is received within the time window and the IDCODE read matches with that of the Cortex®-M0+ DAP.

17.5.2 SWD programming mode entry

After the SWD port is acquired, the host must enter the device programming mode within a specific time window. This is done by setting the TEST_MODE bit (bit 31) in the test mode control register (MODE register). The debug port should also be configured before entering the device programming mode.

17.5.3 SWD programming routine executions

When the device is in programming mode, the external programmer can start sending the SWD packet sequence for performing programming operations such as flash erase, flash program, checksum verification, and so on. The programming routines are explained in the **"Nonvolatile memory"** on page 205.

17.6 Registers

Table 17-3. List of registers

Register name	Description
CM0P_DWT	Cortex® M0+ Data Watchpoint and Trace (DWT) registers
CM0P_BP	Cortex® M0+ BreakPoint (BP) registers
CM0P_ROM	Cortex® M0+ CPU Coresight ROM table
CM0P_CTI	Cortex® M0+ Cross-Trigger Interface (CTI) registers
CM0P_MTB	Cortex® M0+ Micro Trace Buffer (MTB) registers
CM4_ITM	Cortex® M4 Instrumentation Trace Macrocell (ITM) registers
CM4_DWT	Cortex® M4 Data Watchpoint and Trace (DWT) registers
CM4_FPB	Cortex® M4 Flash Patch and Breakpoint (FPB) registers
CM4_SCS	Cortex® M4 System Control Space (SCS) registers
CM4_ETM	Cortex® M4 Embedded Trace Macrocell (ETM) registers
CM4_CTI	Cortex® M4 Cross-Trigger Interface (CTI) registers
CM4_ROM	Cortex® M4 CPU Coresight ROM table
TRC_TPIU	System Trace Coresight Trace Port Interface Unit (TPIU) registers

Section D: System resources subsystem (SRSS)

This section encompasses the following chapters:

- **"Power supply and monitoring"** on page 275
- **"Device power modes"** on page 281
- **"Clocking system"** on page 290
- **"Reset system"** on page 313
- **"Watchdog timer"** on page 318
- **"I/O system"** on page 332
- **"Trigger multiplexer block"** on page 344

Top level architecture

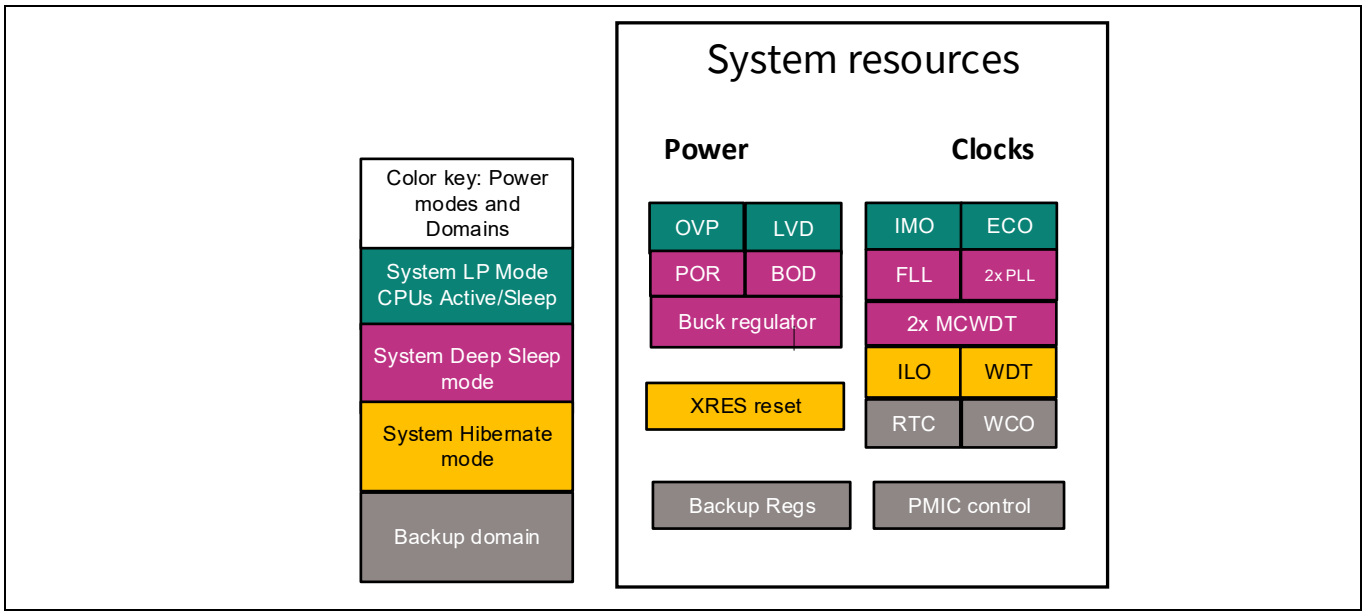


Figure D-1. System-wide resources block diagram

18 Power supply and monitoring

The EZ-USB™ FX10 MCU family supports an operating voltage range of 1.7 V to 3.6 V. It integrates multiple regulators to power the blocks within the device in various power modes. The device supports multiple power supply rails – V_{DDD}, V_{DDA}, and V_{DDIO} – enabling the application to use a dedicated supply for different blocks within the device.

The EZ-USB™ FX10 MCU family supports power-on-reset (POR), brownout detection (BOD), over-voltage protection (OVP), and low-voltage-detection (LVD) circuit for power supply monitoring and failure protection purposes.

18.1 Features

The power supply subsystem of the EZ-USB™ FX10 MCU supports the following features:

- Operating voltage range of 1.7 V to 3.6 V
- User-selectable core logic operation at either 1.1 V (LP)
- Three independent supply rails (V_{DDD}, V_{DDA}, and V_{DDIO}) for EZ-USB™ FX10 core peripherals
- Multiple on-chip regulators
- One low-dropout (LDO) regulator to power peripherals in Active power mode
- Multiple low-power regulators to power peripherals operating in different low-power modes
- Two BOD circuit (V_{DDD} and V_{CCD}) in all power modes
- LVD circuit to monitor V_{DDD}, V_{AMUXA}, V_{AMUXB}, or V_{DDIO}
- One OVP block monitoring V_{CCD}

18.2 Architecture

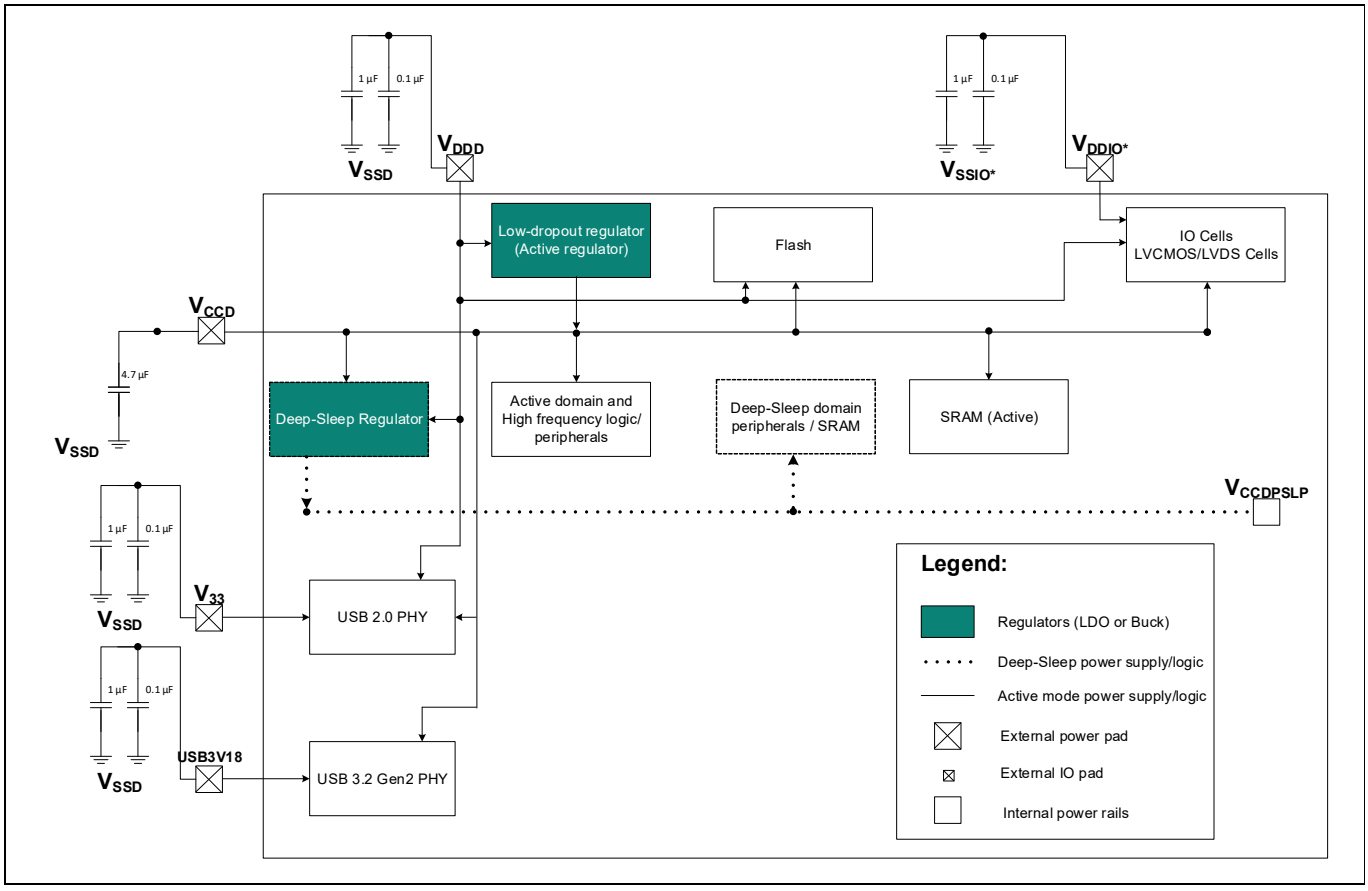


Figure 18-1. Power system block diagram

See the device datasheet for the values to be used for the capacitors and inductor shown in **Figure 18-1**.

The regulators and supply pins/rails shown in **Figure 18-1** power various blocks inside the device. The availability of various supply rails/pins for an application will depend on the device package selected. Refer to the device datasheet for details.

All the core regulators draw their input power from the V_{DDD} supply pin. The on-chip buck uses the V_{DD_NS} supply pin as its input. V_{CCD} supply is used to power all active domain and high frequency peripherals. The output of the buck is connected to the V_{CCD} pin and in firmware the V_{CCD} supply can be switched to the buck output. A dedicated Deep Sleep regulator powers all the Deep Sleep peripherals. The Deep Sleep regulator switches its output to V_{CCD} when available and to its regulated output when V_{CCD} is not present (System Deep Sleep power mode).

When the V_{DDA} pin is not present, analog peripherals run directly from the V_{DDD}.

The I/O cells operate from various V_{DDx} (V_{DDA}, V_{DDD}, or V_{DDIO}) pins depending on the port where they are located. V_{CCD} supply is used to drive logic inside the I/O cells from core peripherals. To know which I/Os operate from which supply, refer to the device datasheet.

In addition to the power rails and regulators, the device provides options to monitor supply rails and protection against potential supply failures. These include a POR circuit, a BOD circuit, an OVP circuit, and a LVD circuit.

18.3 Power supply

18.3.1 Regulators summary

18.3.1.1 Core regulators

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

- **Linear core regulator**

The device includes a linear LDO regulator to power the Active and Sleep mode peripherals. This regulator generates the core voltage (V_{CCD}) required for Active mode operation of the peripherals from V_{DDD}. The regulator is capable of providing 0.9 V and 1.1 V for core operation. See **"Core operating voltage"** on page 277. The regulator is available in Active and Sleep power modes. This regulator implements two sub-modes – high-current and low-current modes. LINREG_LPMODE bit [24] of the PWR_CTL register selects between the two modes of operation. The high-current mode is the normal operating mode, that is, the device operates to its full capacity in Active or Sleep power modes. In the low-current mode or the minimum regulator current mode, the current output from the regulator is limited. This mode implements the Low-Power Active and Sleep power modes. The low-current mode sets a limitation on the capabilities and availability of resources in the Low-Power Active and Sleep modes. For details, see the **"Device power modes"** on page 281.

By default, the linear regulator is powered on reset. The regulator can be disabled by setting the LINREG_DIS bit [23] of the PWR_CTL register. Note that the linear regulator should be turned OFF only when the following condition is satisfied:

- The load current requirement of the device from the V_{CCD} supply does not exceed 20 mA. This should be ensured by the firmware by disabling power consuming high-frequency peripherals and reducing the system clock frequency.

If the linear regulator is turned OFF without the above conditions satisfied, it will result in V_{CCD} brownout and the device will reset.

Power supply and monitoring

- **Core operating voltage**

EZ-USB™ FX10 MCUs can operate at 0.9 V LP mode (nominal) core voltage. On reset, the core is configured to operate at 1.1 V by default. At 0.9 V, power consumption is less, but there are some limitations. The maximum operating frequency for all HFCLK paths should not exceed 50 MHz, whereas the peripheral and slow clock should not exceed 25 MHz.

Follow these steps to change the EZ-USB™ FX10 MCU core voltage:

1. Set the ACT_REG_TRIM bits[4:0] of the PWR_TRIM_PWRSYS_CTL register to '0x07' for 0.9 V or '0x17' for 1.1 V.
2. In the case of 1.1 V to 0.9 V transition, the time it takes to discharge or settle to the new voltage may depend on the load. So the system can continue to operate while the voltage discharges.
3. In the case of 0.9 V to 1.1 V transition, wait 9 µs for the regulator to stabilize to the new voltage. The clock frequency can be increased after the settling delay.

Note: When changing clock frequencies, make sure to update wait states of RAM/ROM/FLASH. Refer to the ***"CPU subsystem (CPUSS)"*** *on page 63 for details on the wait states.*

- **Other low-power regulators**

In addition to the core active regulators, the device includes multiple low-power regulators for powering Deep Sleep peripherals/logic ($V_{CCDPSLP}$), digital retention logic/SRAM (V_{CCRET}), and N-wells (V_{NWELL}) in the device. Note that V_{NWELL} is not shown in **Figure 18-1** because this rail is used across the device for powering all the N-wells in the chip. These rails are shorted to V_{CCD} in Active and Sleep power modes. V_{CCRET} powers all the Active mode logic that needs to be in retention in Deep Sleep mode.

18.3.2 Power pins and rails

Table 18-1 lists all the power supply pins available in the device. The supply rails running inside the device ($V_{CCDPSLP}$, V_{CCRET}, V_{DDBAK}, and V_{NWELL}) are derived from these external supply pins/rails.

Table 18-1. Supply pins

Supply pin	Ground pin	Voltage range supported[a)]	Description
V_{DD} or V_{DDD}	V_{SS} or V_{SSD}	1.7 V to 3.6 V	V_{DD} is a single supply input for multiple supplies and V_{DDD} is a digital supply input. Either of the pins will be present in a given package.
V_{CCD}	V_{SS} or V_{SSD}	Capacitor (0.9 V or 1.1 V)	Core supply or bypass capacitor for the internal core regulator (LDO).
V_{DDA}	V_{SS} or $V_{SSA\ (if\ available)}$	1.7 V to 3.6 V	Analog supply voltage.
V_{DDIO}	V_{SS} or $V_{SSIO\ (if\ available)}$	1.7 V to 3.6 V	Additional I/O supply voltage.

a) Refer to the device datasheet for exact range and recommended connections.

18.3.3 Power sequencing requirements

V_{DDD}, V_{DDIO}, and V_{DDA} do not have any sequencing limitation and can establish in any order. The presence of V_{DDA} without V_{DD} or V_{DDD} can cause some leakage from V_{DDA}. However, it will not drive any analog or digital output. All the V_{DDA} pins in packages that offer multiple V_{DDA} supply pins, must be shorted externally (on the PCB). Note that the system will not exit reset until both V_{DDD} and V_{DDA} are established. However, it will not wait for other supplies to establish.

18.3.4 Power supply sources

The EZ-USB™ FX10 MCU offers power supply options that support a wide range of application voltages and requirements. V_{DDD} input supports a voltage range of 1.7 V to 3.6 V. If the application voltage is in this range, then the EZ-USB™ FX10 MCU (V_{DDD}) can be interfaced directly to the application voltage. In applications that have voltage beyond this range, a suitable PMIC (Buck or Boost or Buck-Boost) should be used to bring the voltage to this range.

Other supply rails and pins such as V_{DDA} and V_{DDIO} exist independent of V_{DDD} and V_{CCD}.

18.4 Voltage monitoring

The EZ-USB™ FX10 MCU offers multiple voltage monitoring and supply failure protection options. This includes POR, BOD, LVD, and OVP.

18.4.1 Power-on reset (POR)

The POR circuits provide a reset pulse during the initial power ramp. POR circuits monitor V_{DDD} voltage. Typically, the POR circuits are not very accurate about the trip-point.

POR circuits are used during initial chip power-up and then disabled. Refer to the device datasheet for details on the POR trip-point levels.

18.4.2 Brownout detect (BOD)

The BOD circuit protects the operating or retaining logic from possibly unsafe supply conditions by applying reset to the device. The EZ-USB™ FX10 MCU offers two BOD circuits – high-voltage BOD (HVBOD) and low-voltage BOD (LVBOD). The HVBOD monitors the V_{DDD} voltage and LVBOD monitors the V_{CCD} voltage. Both BOD circuits generate a reset if a voltage excursion dips below the minimum V_{DDD}/V_{CCD} voltage required for safe operation (see the device datasheet for details). The system will not come out of RESET until the supply is detected to be valid again.

The HVBOD circuit guarantees a reset in System LP and Deep Sleep power modes before the system crashes, provided the V_{DDD} supply ramp satisfies the datasheet maximum supply ramp limits in that mode.

The LVBOD, operating on V_{CCD}, is not as robust as the HVBOD. The limitation is because of the small voltage detection range available for LVBOD on the minimum allowed V_{CCD}.

For details on the BOD trip-points, supported supply ramp rate, and BOD detector response time, refer to the device datasheet.

18.4.3 Low-voltage detect (LVD)

An LVD circuit monitors external supply voltage and accurately detects depletion of the energy source. The LVD generates an interrupt to cause the system to take preventive measures.

The LVD can be configured to monitor V_{DDD}, V_{AMUXA}, V_{AMUXB}, or V_{DDIO}. The HVLVD1_SRCSEL bits [6:4] of the PWR_LVD_CTL register selects the source of the LVD. The LVD support up to 15 voltage levels (thresholds) to monitor between 1.2 V to 3.1 V. The HVLVD1_TRIPSEL bits [3:0] of the PWR_LVD_CTL register select the threshold levels of the LVD. LVD should be disabled before selecting the threshold. The HVLVD1_EN bit [7] of the PWR_LVD_CTL register can be used to enable or disable the LVD.

Whenever the voltage level of the supply being monitored drops below the threshold, the LVD generates an interrupt. This interrupt status is available in the SRSS_INTR register. HVLVD1 bit [1] of the SRSS_INTR register indicates a pending LVD interrupt. The SRSS_INTR_MASK register decides whether LVD interrupts are forwarded to the CPU or not.

Note that the LVD circuit is available only in Active, LPACTIVE, Sleep, and LPSLEEP power modes. If an LVD is required in Deep Sleep mode, then the device should be configured to periodically wake up from Deep Sleep mode using a Deep Sleep wakeup source. This makes sure an LVD check is performed during Active/LPACTIVE mode.

When enabling the LVD circuit, it is possible to receive a false interrupt during the initial settling time. The firmware can mask this by waiting for 8 µs after setting the HVLVD1_EN bit in the PWR_LVD_CTL register. The recommended firmware procedure to enable the LVD function is:

1. Ensure that the HVLVD1 bit in the SRSS_INTR_MASK register is 0 to prevent propagating a false interrupt.
2. Set the required trip-point in the HVLVD1_TRIPSEL field of the PWR_LVD_CTL register.
3. Configure the LVD edge (falling/rising/both) that triggers the interrupt by configuring HVLVD1_EDGE_SEL bits[1:0] of SRSS_INTR_CFG register. By default, the configuration disables the interrupt.
   Note: LVD logic may falsely detect a falling edge during Deep Sleep entry. This applies only when HVLVD1_EDGE_SEL is set to FALLING(2) or BOTH(3). Firmware can workaround this condition by disabling falling edge detection before entering Deep Sleep, and re-enabling it after exiting Deep Sleep.
4. Enable the LVD by setting the HVLVD1_EN bit in the PWR_LVD_CTL register. This may cause a false LVD event.
5. Wait at least 8 µs for the circuit to stabilize.
6. Clear any false event by setting the HVLVD1 bit in the SRSS_INTR register. The bit will not clear if the LVD condition is truly present.
7. Unmask the interrupt by setting the HVLVD1 bit in SRSS_INTR_MASK.

LVD is used to detect potential brownouts. When the supply being monitored drops below the threshold, the interrupt generated can be used to save necessary data to flash, dump logs, trigger external circuits, and so on.

For details on supported LVD thresholds, refer to the device datasheet and the PWR_LVD_CTL register definition in the registers reference manual.

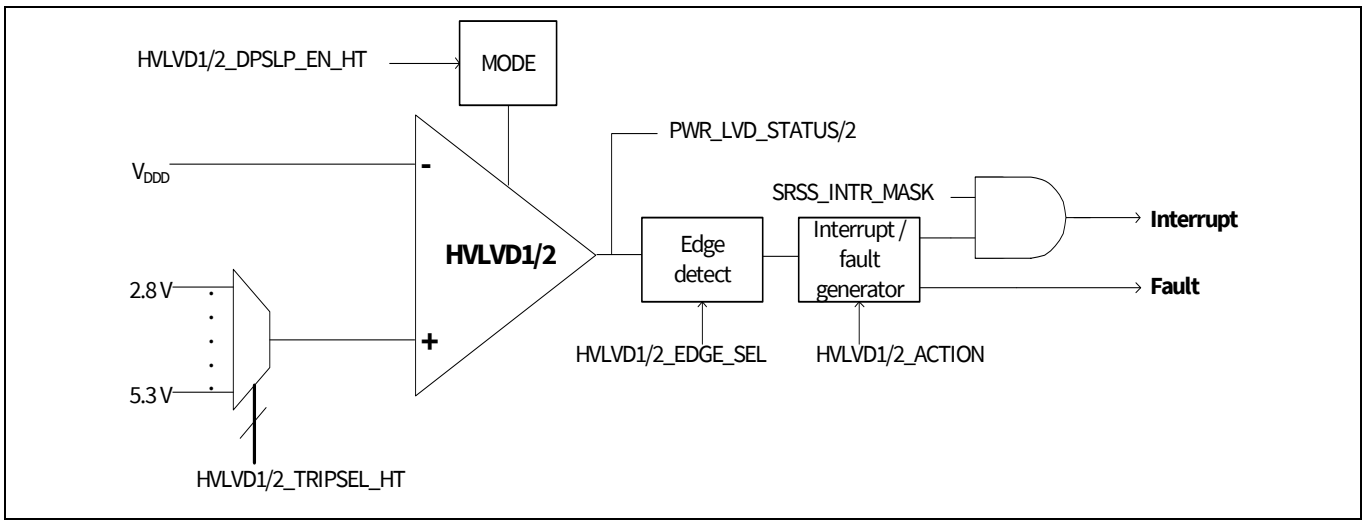


Figure 18-2. LVD block

18.4.4 Over voltage protection (OVP)

The EZ-USB™ FX10 MCU offers an over voltage protection circuit that monitors the V_{CCD} supply. Similar to the BOD circuit, the OVP circuit protects the device from unsafe supply conditions by applying a reset. As the name suggests, the OVP circuit applies a device reset, when the V_{CCD} supply goes above the maximum allowed voltage.

18.5 Register list

Name	Description
PWR_CTL	Power Mode Control register - controls the device power mode options and allows observation of current state
PWR_LVD_CTL	LVD Configuration register
SRSS_INTR	SRSS Interrupt register - shows interrupt requests from the SRSS peripheral
SRSS_INTR_MASK	SRSS Interrupt Mask register - controls forwarding of the interrupt to CPU
SRSS_INTR_SET	SRSS Interrupt Set register - sets interrupts; this register is used for firmware testing
SRSS_INTR_MASKED	SRSS Interrupt Masked register - logical AND of corresponding SRSS interrupt request (SRSS Interrupt register) and mask bits (SRSS Interrupt Mask register)

19 Device power modes

The EZ-USB™ FX10 MCU can operate in four system and three CPU power modes. These modes are intended to minimize the average power consumption in an application. The power modes supported by EZ-USB™ FX10 MCUs, in the order of decreasing power consumption, are:

- System Low Power (LP) – All peripherals and CPU power modes are available at maximum speed and current
- CPU Active – CPU is executing code in system LP mode
- CPU Sleep – CPU code execution is halted in system LP mode
- CPU Deep Sleep – CPU code execution is halted and system Deep Sleep is requested while in system LP mode
- System Deep Sleep – Entered only after both CPUs enter CPU Deep Sleep mode. Only low-frequency peripherals are available

CPU Active, Sleep, and Deep Sleep are standard Arm®-defined power modes supported by the Arm® CPU instruction set architecture (ISA). System LP and Deep Sleep modes are additional low-power modes supported by EZ-USB™ FX10.

19.1 Features

The EZ-USB™ FX10 MCU power modes have the following features:

- Four system and three CPU power modes aimed at optimizing power consumption in an application
- System Deep Sleep mode with support for multiple wakeup sources and configurable amount of SRAM retention

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

- Enabling and disabling clocks to peripherals
- Powering on/off clock sources
- Powering on/off peripherals and resources inside the EZ-USB™ FX10 device

19.2 Architecture

The EZ-USB™ FX10 device supports multiple power modes. Some modes only affect the CPUs (CPU power modes) and others affect the whole system (system power modes). The system and CPU power modes are used in combination to control the total system performance and power. CPU power modes are entered separately for each CPU on the device.

The SysPm peripheral driver library (PDL) driver supports all device power mode transitions and is the recommended method of transition and configuration of EZ-USB™ FX10 MCU power resources.

Device power modes

Table 19-1 summarizes the power modes available in EZ-USB™ FX10 MCUs, their description, and details on their entry and exit conditions.

Table 19-1. EZ-USB™ FX10 MCU power modes

System Power mode	MCU Power mode	Description	Entry conditions	Wakeup sources	Wakeup action
LP	Active	System – Primary mode of operation. 1.1 V core voltage. All peripherals are available (programmable). Maximum clock frequencies CPU – Active mode	Reset from external reset, brownout, power on reset system. Wakeup from CPU Sleep or CPU Deep Sleep while in system LP mode. Wakeup from system Deep Sleep after entered from LP mode.	Not applicable	N/A
	Sleep	1.1 V core voltage. One or more CPUs in Sleep mode (execution halted). All peripherals are available (programmable). Maximum clock frequencies	In system LP mode, CPU executes WFI/WFE instruction with Deep Sleep disabled	Any interrupt to CPU	Interrupt
	Deep Sleep	1.1 V core voltage. One CPU in Deep Sleep mode (execution halted). Other CPU in Active or Sleep mode. All peripherals are available (programmable). Maximum clock frequencies	In system LP mode, CPU executes WFI/WFE instruction with Deep Sleep enabled	Any interrupt to CPU	Interrupt
Deep Sleep	Deep Sleep	All high-frequency clocks and peripherals are turned off. Low-frequency clock (32 kHz) and low-power analog and digital peripherals are available for operation and as wakeup sources. SRAM is retained (programmable).	Both CPUs simultaneously in CPU Deep Sleep mode.	GPIO interrupt, SCB, watchdog timer	Interrupt

19.2.1 CPU Power modes

The CPU Active, Sleep, and Deep Sleep modes are the standard Arm®-defined power modes supported by both Cortex®-M4 and Cortex®-M0+ CPUs. All Arm® CPU power modes are available in system LP power mode. CPU power modes affect each CPU independently.

19.2.2 CPU Active mode

In CPU Active mode, the CPU executes code and all logic and memory is powered. The firmware may decide to enable or disable specific peripherals and power domains depending on the application and power requirements. All the peripherals are available for use in Active mode. The device enters CPU Active mode upon any device reset or wakeup.

19.2.3 CPU Sleep mode

In CPU Sleep mode, the CPU clock is turned off and the CPU halts code execution. Note that in the EZ-USB™ FX10 MCU, Cortex®-M4 and Cortex®-M0+ both support their own CPU Sleep modes and each CPU can be in sleep independent of the other CPU state. All peripherals available in Active mode are also available in Sleep mode. Any peripheral interrupt, masked to the CPU, will wake the CPU to Active mode. Only the CPU(s) with the interrupt masked will wake.

19.2.3.1 CPU Deep Sleep mode

In CPU Deep Sleep mode, the CPU requests the device to go into system Deep Sleep mode. When the device is ready, it enters Deep Sleep mode as detailed in **"System Deep Sleep mode"** on page 283.

Because EZ-USB™ FX10 has more than one CPU, both CPUs must independently enter CPU Deep Sleep before the system will transition to system Deep Sleep.

19.2.4 System Power modes

System power modes affect the whole device and may be combined with CPU power modes.

19.2.4.1 System Low Power mode

System Low Power (LP) mode is the default operating mode of the device after reset and provides maximum system performance. In LP mode all resources are available for operation at their maximum power level and speed.

While in system LP mode the CPUs may operate in any of the standard Arm® defined CPU modes detailed in **"CPU Power modes"** on page 282.

19.2.5 System Deep Sleep mode

In system Deep Sleep mode, all the high-speed clock sources are off. This in turn makes high-speed peripherals unusable in system Deep Sleep mode. However, low-speed clock sources and peripherals may continue to operate, if configured and enabled by the firmware. In addition, the peripherals that do not need a clock or receive a clock from their external interface (I^2C or SPI slave) may continue to operate, if configured for system Deep Sleep operation. The EZ-USB™ FX10 MCU provides an option to configure the amount of SRAM, in blocks of 32 KB, that are retained during Deep Sleep mode.

Both Cortex®-M0+ and Cortex®-M4 can enter CPU Deep Sleep mode independently. However, the entire device enters system Deep Sleep mode only when both the CPUs are in CPU Deep Sleep. On wakeup, the CPU that woke up enters CPU Active mode and the other CPU remains in CPU Deep Sleep mode. On wakeup, the system will return to LP mode based on what mode was Active before entering system Deep Sleep. Both CPUs may wake up to CPU Active simultaneously from the same wakeup source if so configured.

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

- The LPM_READY bit of the PWR_CTL register should read '1'. This ensures the system is ready to enter Deep Sleep. If the bit reads '0', then the device will wait in system LP mode instead of system Deep Sleep until the bit is set, at which instant the device automatically enters Deep Sleep mode, if requested.
- Both Cortex®-M0+ and Cortex®-M4 must be in CPU Deep Sleep. This is achieved by setting the SLEEPDEEP bit [2] of the SCR register of both Cortex®-M0+ and Cortex®-M4 and then executing WFI or WFE instruction.

In system Deep Sleep mode, the LP mode regulator is turned off and a lower power, Deep Sleep regulator sources all the peripherals enabled in system Deep Sleep mode. See the **"Power supply and monitoring"** on page 275 for details. **Table 19-5** provides the list of resources available in system Deep Sleep mode.

Interrupts from low-speed, asynchronous, or low-power analog peripherals can cause a CPU wakeup from system Deep Sleep mode. Note that when a debugger is running on either core, the device stays in system LP mode and the CPUs enter CPU Sleep mode instead of CPU Deep Sleep mode. EZ-USB™ FX10 uses buffered writes. Therefore, writes to an MMIO register or memory can take few a 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.

19.2.6 Other Operation modes

In addition to the power modes discussed in the previous sections, there are three other states the device can be in Reset and OFF states. These states are determined by the external power supply and XRES connections. No firmware action is required to enter these modes nor an interrupt or wakeup event to exit them.

19.2.6.1 Reset state

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

19.2.6.2 Off state

The off state simply represents the device state with no power applied. The reset and off states are discussed for completeness of all possible modes and states the device can be in. These states can be used in a system to further optimize power consumption. For instance, the system can control the supply of the EZ-USB™ FX10 MCU by enabling or disabling the regulator output powering the device using the PMIC interface.

19.3 Power mode transitions

Figure 19-1 shows various states the device can be in along with possible power mode transition paths.

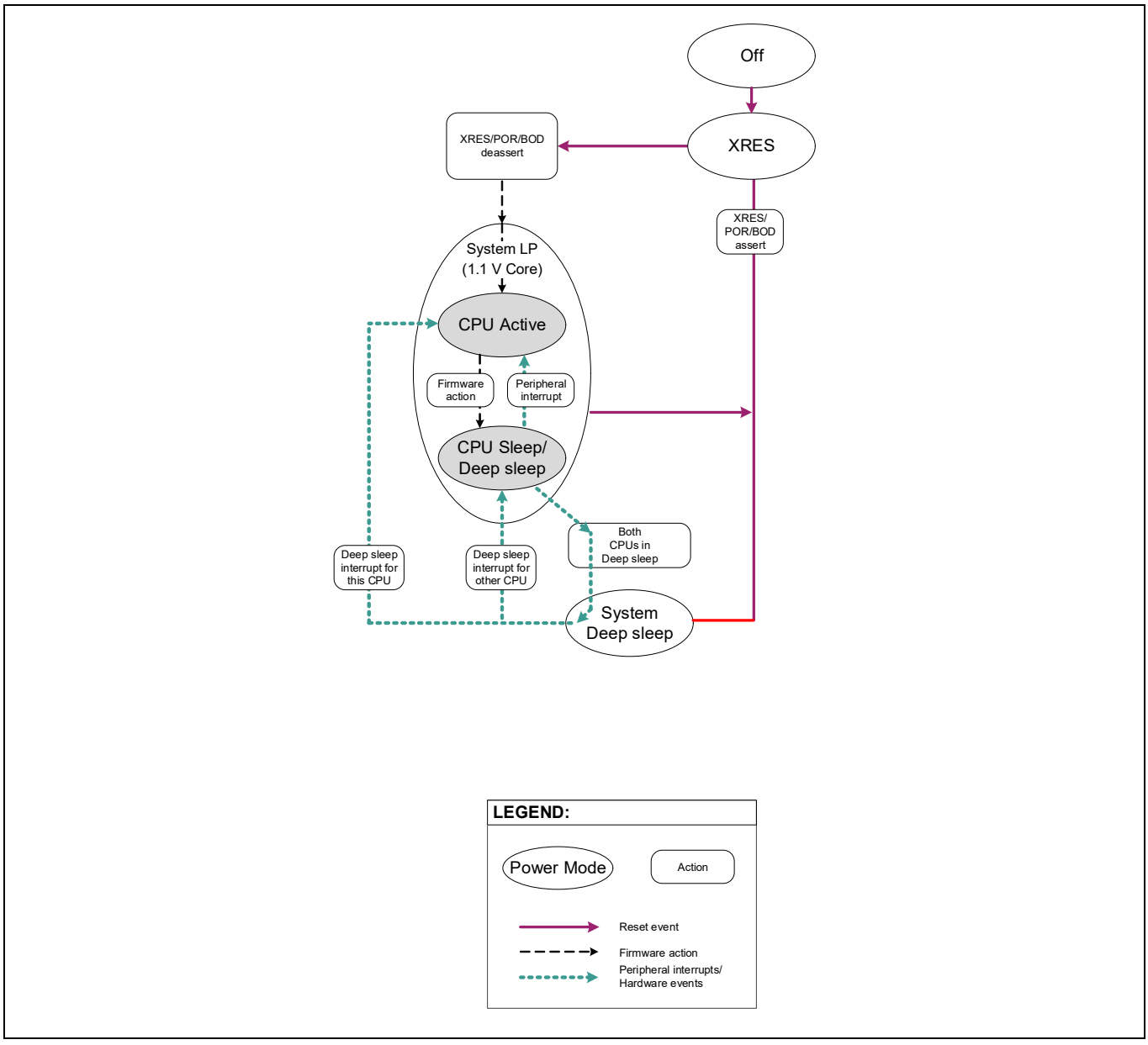


Figure 19-1. Power mode transitions in EZ-USB™ FX10 MCU

19.3.1 Power-up transitions

Table 19-2 summarizes various power-up transitions, their type, triggers, and actions.

Table 19-2. Power mode transitions

Initial state	Final state	Type	Trigger	Actions
Off	XRES	External	Power rail (V_{DDD}) ramps up above POR voltage level with XRES pin asserted.	1. All high-voltage logic is reset
Off	Reset	External	Power rail (V_{DDD}) ramps up above POR voltage level with XRES pin de-asserted.	1. All high-voltage logic is reset 2. Low-voltage (internal core and Deep Sleep mode) regulators and references are ramped up 3. All low-voltage logic (logic operating from internal regulators) is reset 4. IMO clock is started
XRES	Reset	External	XRES is de-asserted with V_{DDD} present and above POR level.	1. Low-voltage regulators and references are ramped up 2. All low-voltage logic is reset 3. IMO clock is started
Reset	Active	Internal	Reset sequence completes. This transition can also be caused by internal resets.	1. Clock is released to the system 2. System reset is de-asserted 3. CPU starts execution

19.3.2 Power mode transitions

Table 19-3. Power mode transitions

Initial state	Final state	Type	Trigger	Actions
System LP and CPU Active	System LP and CPU Sleep	Internal	Firmware action 1. Clear the SLEEPDEEP bit [2] of the SCR register for both Cortex-M0+ and Cortex-M4. 2. Optionally, set the SLEEPONEXIT bit [1] of the SCR register, if the CPU runs only on interrupts. When this bit is set, the CPU does not return to application code after the WFI/WFE instruction is executed. The CPU wakes up on any enabled (masked to CPU) interrupt or event and enters CPU Sleep mode as soon as it exits the interrupt or services the event. 3. Optionally, set the SEVONPEND bit [4] of the SCR register if the application must wake up the CPU from any pending interrupt. If this bit is set, any interrupt that enters a pending state wakes up the CPU. This includes all the disabled (unmasked) interrupts to CPU. 4. Execute WFI/WFE instruction on both CPUs.	1. CPU clocks are gated off 2. CPU waits for an interrupt or event to wake it up.
System LP and CPU Active	System LP and CPU Deep Sleep	Internal	Firmware action Perform these steps to enter Deep Sleep mode (LPM_READY bit [5] of the PWR_CTL register should read '1' before performing these steps): 1. Set the SLEEPDEEP bit [2] of the SCR register for one or both Cortex-M0+ and Cortex-M4. 2. Optionally, set the SLEEPONEXIT bit [1] of the SCR register if the CPU runs only on interrupts. When this bit is set, the CPU does not return to application code after the WFI/WFE instruction is executed. The CPU wakes up on any enabled (masked to CPU) interrupt or event and enters CPU Deep Sleep mode as soon as it exits the interrupt or services the event. 3. Optionally, set the SEVONPEND bit [4] of the SCR register if the application needs to wake up the CPU from any pending interrupt. If this bit is set, any interrupt that enters a pending state wakes up the CPU. This includes all the disabled (unmasked) interrupts to CPU. 4. Read the SCR register before executing a WFI/WFE instruction to ensure the write operation is complete. EZ-USB™ FX10 uses buffered writes and any write transfer just before executing WFI/WFE instruction should be followed by a read to the same memory location. This ensures that the write operation has taken effect before entering Deep Sleep mode. Execute WFI/WFE instruction on both CPUs. 5. CPU in Deep Sleep mode generates a hardware request for the whole device to enter system Deep Sleep. A CPU waiting in CPU Deep Sleep state is functionally identical to CPU Sleep mode with the exception of the hardware request. 6. If only one CPU is in Deep Sleep mode the system will remain in system LP mode until the other CPU also enters CPU Deep Sleep. While waiting, a masked interrupt can wake the CPU to Active mode.	1. CPU clocks are gated off 2. CPU waits for a Deep Sleep interrupt to wake it up.
System LP and CPU Deep Sleep	System Deep Sleep	Internal	Hardware action 1. When both CPUs enter CPU Deep Sleep mode and the LPM_READY bit [5] of the PWR_CTL register reads '1', the device will automatically transition to system Deep Sleep power mode.	1. High-frequency clocks are shut down. 2. Retention is enabled and non-retention logic is reset. 3. Active regulator is disabled and Deep Sleep regulator takes over.

19.3.3 Wakeup transitions

Table 19-4. Wakeup transitions

Initial state	Final state	Type	Trigger	Actions
CPU Sleep	CPU Active	Internal/ External	Any peripheral interrupt masked to CPU	1. Clock to CPU is ungated. 2. Peripheral interrupt is serviced by CPU. 3. Device remains in current system LP power mode.
System Deep Sleep	System LP Active and CPU Active	Internal/ External	Any Deep Sleep interrupt	1. Active regulator and references are enabled. 2. Retention is disabled and non-retention reset is de-asserted. 3. High-frequency clocks are turned on. 4. CPU exits low-power mode and services the interrupt. 5. Returns to previous system LP power mode. *Note: If only one CPU wakes up from the system Deep Sleep interrupt, then the other CPU remains in the CPU Deep Sleep state until its Deep Sleep interrupt wakes it up. However, the system will wake up to LP mode. Note that a Deep Sleep interrupt can wake up either one or both CPUs depending on the WIC configuration for the CPU. See the **"Interrupts"** on page 89.*

19.4 Summary

Table 19-5. Resources available in Different Power modes

Component	Power modes			
	LP		Deep Sleep	XRES
	CPU Active	CPU Sleep/ Deep Sleep		
Core functions				
CPU	On	Sleep	Retention	Off
SRAM	On	On	Retention	Off
Flash	Read/Write	Read/Write	Off	Off
High-Speed Clock (IMO, ECO, PLL, FLL)	On	On	Off	Off
LVD	On	On	Off	Off
ILO	On	On	On	Off
Peripherals				
SMIF/SHDC	On	On	Retention	Off
LPCMP	On	On	On[a]	Off
TCPWM	On	On	Off	Off
SCB	On	On	Retention (I^2C/SPI wakeup available)[a)]	Off
GPIO	On	On	On	Off
Watchdog timer	On	On	On	Off
Multi-Counter WDT	On	On	On	Off
Resets				
XRES	On	On	On	On
POR	On	On	On	Off
BOD	On	On	On	Off
Watchdog reset	On	On	On	Off

a) Only the SCB with system Deep Sleep support is available in the Deep Sleep power mode; other SCBs are not available in the Deep Sleep power mode.

19.5 Register list

Name	Description
PWR_CTL	Power Mode Control register – controls the device power mode options and allows observation of current state.
CM4_SCS_SCR	Cortex®-M4 System Control register – controls the CM4 CPU sleep and Deep Sleep decisions on the WFI/WFE instruction execution. This register is detailed in the ARMv7-M Architecture Reference Manual available from Arm®.
CM0P_SCS_SCR	Cortex®-M0+ System Control register – controls the CM0+ CPU sleep and Deep Sleep decisions on the WFI/WFE instruction execution. This register is detailed in the ARMv7-M Architecture Reference Manual available from Arm®.
CPUSS_CM0_CTL	Controls the CM0+ power state. Note that this register may only be modified by the CM4 while the CM0+ is in CPU Deep Sleep mode.
CPUSS_CM0_STATUS	Specifies if the CM0+ is in CPU Active, Sleep, or Deep Sleep power mode.
CPUSS_CM4_PWR_CTL	Controls the CM4 power state. Note that this register may only be modified by the CM0+ while the CM4 is in CPU Deep Sleep mode.
CPUSS_CM4_STATUS	Specifies if the CM4 is in CPU Active, Sleep, or Deep Sleep power mode.

20 Clocking system

The EZ-USB™ FX10 family clocking system includes these resources:

- Three internal clock sources:
  - 8 MHz internal main oscillator (IMO)
  - Internal low-speed oscillators (ILO0/ILO1)
- Four external clock sources
  - External clock connected to one of two EXT_CLK inputs
  - External crystal oscillator (ECO)
  - External watch crystal oscillator (WCO)
- One frequency-locked loop (FLL)
- Several phase-locked loops (PLL)

20.1 Block diagram

Figure 20-2 gives a generic view of the clocking system in EZ-USB™ FX10 family devices.

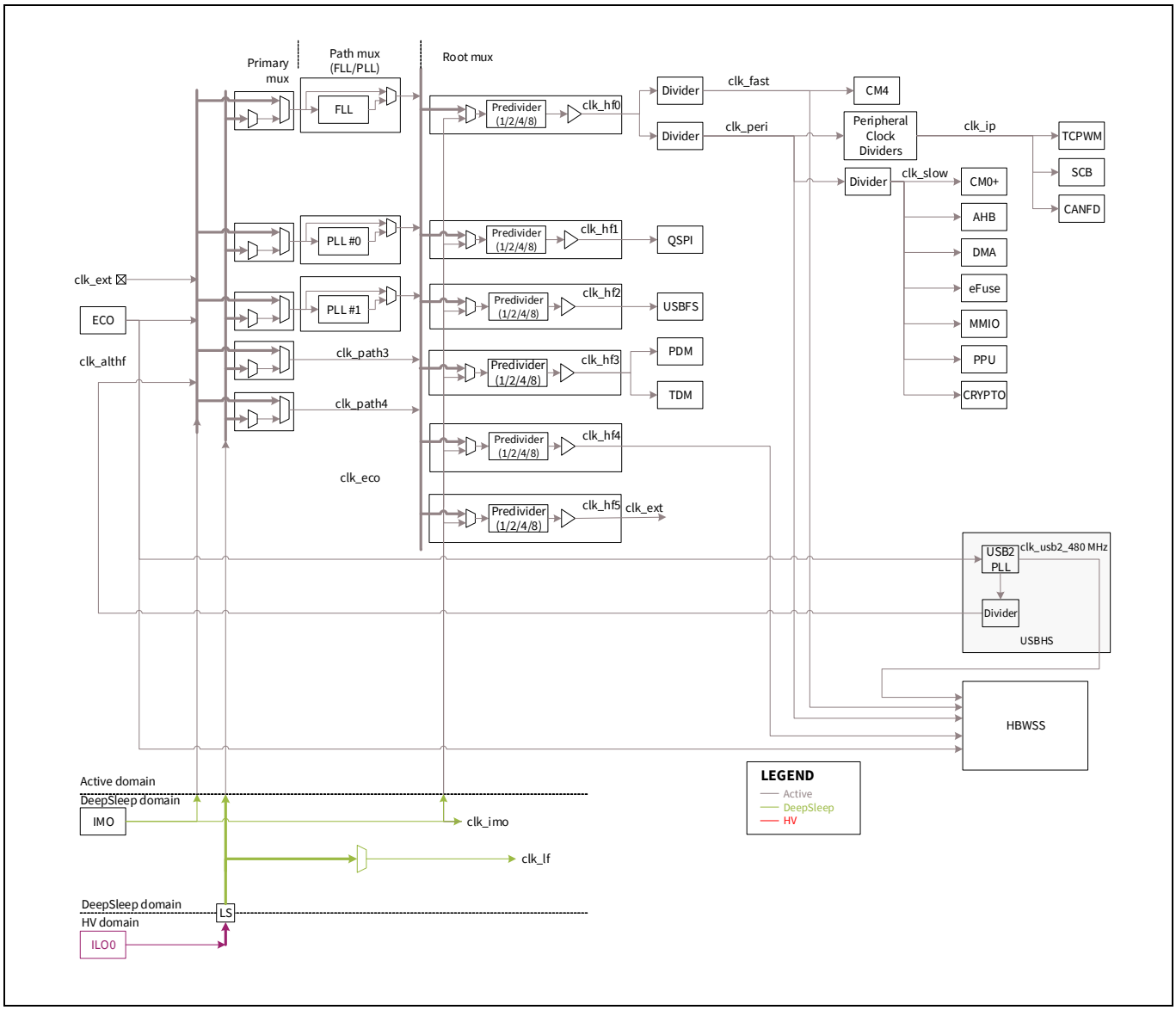


Figure 20-2. Clocking system block diagram

20.2 Clock sources

20.2.1 Internal main oscillator (IMO)

The IMO is an accurate, high-speed internal (crystal-less) oscillator that produces a fixed frequency. Refer to the datasheet for the IMO frequency. The IMO output can be used by the PLL or FLL to generate a wide range of higher frequency clocks, or it can be used directly by the high-frequency root clocks. The IMO is enabled and disabled with CLK_IMO_CONFIG.ENABLE.

The IMO should not be disabled if it is the source of the clock path feeding high-frequency clock zero (CLK_HF0). CLK_HF0 is the source clock for the CPU. Therefore, if the IMO is in the source path of CLK_HF0, disabling the IMO disables the CPU.

The IMO is only available in Active and Sleep modes.

20.2.2 External crystal oscillator (ECO)

The EZ-USB™ FX10 family contains an oscillator to drive an external crystal. Refer to the datasheet for the frequency of ECO. This clock source is built using an oscillator circuit. The circuit employs an external crystal that needs to be populated on the external crystal pins of the EZ-USB™ FX10 device.

The ECO can be enabled using the CLK_ECO_CONFIG register fields.

20.2.2.1 ECO trimming

The ECO supports a wide variety of crystals and ceramic resonators with the nominal frequency range specification described in the datasheet. The crystal manufacturer typically provides numerical values for parameters, namely the maximum drive level (D_L), the equivalent series resistance (ESR), the ideal shunt capacitance (C_0) and the parallel load capacitance (C_L). These parameters can be used to calculate the transconductance (g_m) and the maximum peak oscillation voltage across the crystal (V_P).

The formula of V_P is as follows. ECO does not support V_P less than 0.3 V.

$$\text{Max peak value:Vp} = \frac{\sqrt{\frac{D_L}{2ESR}}}{\pi f(C_0 + C_L)}$$

The formula of Transconductance (g_m) is as follows. The ECO block can deliver a maximum Transconductance (g_m) of 17.6 mA/V.

$$\text{Transconductance:} \qquad g_m > 20 \times ESR \times (2\pi \times f)^2 \times (C_0 + C_L)^2$$

The formula of Negative resistance (R_{neg}) is as follows. To guarantee crystal start up, negative resistance needs to be at least five times larger than ESR.

$$\text{Negative Registance:} \qquad |R_{neg}| = \frac{g_m \times 4 \times {C_L}^2}{(2\pi \times f)^2 \times (4 \times {C_L}^2 + 4 \times C_L \times C_0)^2}$$

The ATRIM, WDTRIM, and FTRIM fields can be found in the CLK_ECO_CONFIG2 register. The ATRIM and WDTRIM settings control the trim for amplitude of the oscillator output. The FTRIM setting controls the filter used to prevent the third harmonic oscillation.

Amplitude trim (ATRIM) sets the crystal drive level when automatic gain control (AGC) is enabled (CLK_ECO_CONFIG.AGC_EN = 1). AGC must be enabled for V_P < 1.1 V and disabled for all other cases.

Clocking system

Watchdog trim (WDTRIM) sets the threshold on XO magnitude where the ECO block releases the clock to the system.

Filter trim (FTRIM) tunes the low-pass filter between the ECO_IN pin and the amplifier, which is used to prevent amplification of harmonics of the intended crystal frequency. The FTRIM value at 0x03 can be used; no other value is required.

WARNING: The V_p setting is critical for reliable system performance. If the V_p settings are too large (or AGC is disabled), the crystal can be damaged or suffer premature aging due to excessive power dissipation. If the V_p settings are too small, the oscillation will be more susceptible to system noise.

Based on the V_P value, the ATRIM, WDTRIM, and FTRIM values are set as shown in **Table 20-6**.

Table 20-6. ATRIM, WDTRIM, and FTRIM settings based on V_P

V_P [V]	AGC_EN	ATRIM	WDTRIM	FTRIM
$0.5 \le V_P < 0.55$	0x1	0x4	0x2	0x3
$0.55 \le V_P < 0.60$	0x1	0x5		
$0.60 \le V_P < 0.65$	0x1	0x6	0x3	
$0.65 \le V_P < 0.70$	0x1	0x7		
$0.70 \le V_P < 0.75$	0x1	0x8	0x4	
$0.75 \le V_P < 0.80$	0x1	0x9		
$0.80 \le V_P < 0.85$	0x1	0xA	0x5	
$0.85 \le V_P < 0.90$	0x1	0xB		
$0.90 \le V_P < 0.95$	0x1	0xC	0x6	
$0.95 \le V_P < 1.00$	0x1	0xD		
$1.00 \le V_P < 1.05$	0x1	0xE	0x7	
$1.05 \le V_P < 1.10$	0x1	0xF		
$1.10 \le V_P$	0x0	0x0-0xE[a)]		

a) It is acceptable to select any value from 0x0 to 0xE.

The GTRIM sets up the trim for amplifier gain based on the calculated g_m, as shown in **Table 20-7**.

Table 20-7. GTRIM settings

g_m [mA/V]	GTRIM
$0 \le g_m < 2.2$	0x00
$2.2 \le g_m < 4.4$	0x01
$4.4 \le g_m < 6.6$	0x02
$6.6 \le g_m < 8.8$	0x03
$8.8 \le g_m < 11$	0x04
$11 \le g_m < 13.2$	0x05
$13.2 \le g_m < 15.4$	0x06
$15.4 \le g_m \le 17.6$	0x07

RTRIM should be oscillator feedback resistor, as shown in **Table 20-8**.

Table 20-8. RTRIM settings

Nominal frequency f [MHz]	RTRIM
28.6 < f	0x00
23.33 < f ≤ 28.6	0x01
16.5 < f ≤ 23.33	0x02
f ≤ 16.5	0x03

First, set up the trim values based on **Table 20-6** through **Table 20-8** and then enable the ECO. After the ECO is enabled, the CLK_ECO_STATUS register can be checked to ensure it is ready.

20.2.3 External clock (EXT_CLK)

The external clock can be sourced from a signal on a designated I/O pin. This clock can be used as the source clock for either the PLL or FLL, or can be used directly by the high-frequency clocks.

When manually configuring a pin as the input to the EXT_CLK, the drive mode of the pin must be set to high impedance digital to enable the digital input buffer. See the**"I/O system"** on page 332 for more details. See the device datasheet to determine the specific pin used for EXT_CLK.

The EXT_CLK function is bi-directional. See **"Clock generation"** on page 294 for more details.

20.2.4 Internal low-speed oscillator (ILO)

The two ILO blocks operate with no external components and output a stable clock. Refer to the datasheet for the frequency of the two ILOs. The ILO block is relatively low power and low accuracy. It is available in all power modes.

The ILO blocks can be used as the clock source for:

- DSI_Mux: While the ILOs are routable through the DSI_MUX, there are no supported use cases for doing so
- ILO0 is the clock for the watchdog timer (WDT) and Deep Sleep CSV (clock supervision)
- ILO1 is used only if all of the following are true
  - CLK_LF must be available in Deep Sleep mode (otherwise use ECO output)
  - Clock supervision of CLK_LF is necessary (otherwise, use ILO0)
  - WCO is not available (otherwise, use WCO)

The ILO0 and the ILO1 are enabled/disabled with CLK_ILO0_CONFIG.ENABLE and CLK_ILO1_CONFIG.ENABLE respectively. It is recommended to always leave ILO0 enabled as it is the source of the WDT.

User must ensure not to turn off the ILO0 (CLK_ILO0) before initiating any soft reset.

If the WDT is enabled, the only way to disable the ILO0 is to first clear WDT_CTL.WDT_LOCK and then clear CLK_ILO0_CONFIG.ENABLE. If the WDT_CTL.WDT_LOCK is set, any register write to disable the ILO0 will be ignored. Enabling the WDT will automatically enable the ILO0.

The MCU provides an opportunity to calibrate the ILOx by using the calibration counter, described in **"Clock calibration counters"** on page 304. For instance, the ECO can be used as a reference clock. This result can be used to determine how the ILOx needs to be adjusted. The ILO0 and ILO1 can be trimmed using the CLK_TRIM_ILO0_CTL and CLK_TRIM_ILO1_CTL registers.

20.2.5 ECO prescaler

ECO prescaler divides the ECO and creates a clock that can be used with CLF_LF clock. This feature is only available during active and sleep mode. It cannot be used during Deep Sleep mode.

The division function has a 10-bit integer divider and 8-bit fractional divider. This function is configured using the CLK_ECO_PRESCALE register.

20.3 Clock generation

This section explains phase locked loop (PLL) and frequency locked loop (FLL) implemented in the EZ-USB™ FX10 family.

EZ-USB™ FX10 family has two types of PLLs; PLL without SSCG and fractional operation, and PLL with SSCG and fractional operation. EZ-USB™ FX10 family has one type of FLL.

20.3.1 PLL without SSCG and fractional operation

This makes it possible to use the IMO or other clock to generate much higher clock frequencies for the rest of the system. **Figure 20-3** shows the block diagram of a PLL without SSCG and fractional operation.

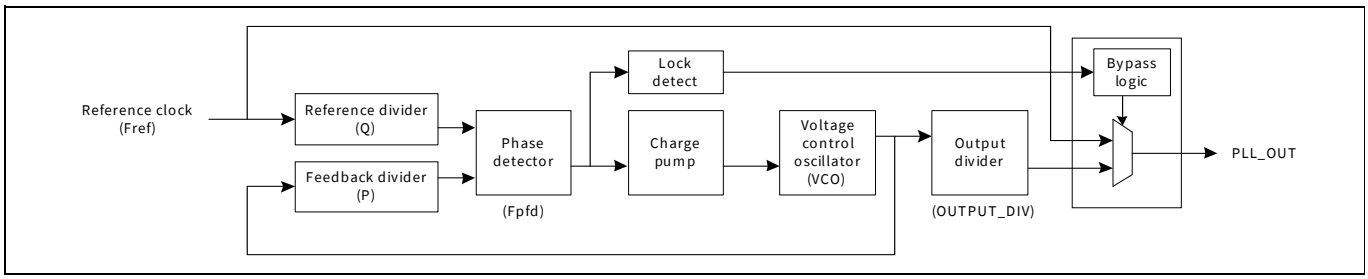


Figure 20-3. PLL without SSCG and fractional operation

The PLL is configured following these steps:

Note: Fref is the input frequency of the PLL; that is, the frequency of the input clock such as 8 MHz for the IMO.

1. Determine the desired reference clock frequency (Fref) and desired output frequency (PLL_OUT). Calculate the reference (Q), feedback (P), and output (OUTPUT_DIV) dividers subject to the following constraints:
   a) PFD frequency (phase detector frequency). Fpfd = Fref / Q. There may be multiple reference divider values that meet this constraint.
   b) VCO frequency. VCO = Fpfd × P. There may be multiple feedback divider values that meet this constraint with different REFERENCE_DIV choices.
   c) Output frequency. PLL_OUT = VCO / OUTPUT_DIV. It may not be possible to get the exact desired frequency due to granularity; therefore, consider the frequency error of the two closest choices.
   d) Choose the best combination of divider parameters depending on the application.
2. Program the divider settings in the appropriate CLK_PLL_CONFIGx register. Do not enable the PLL on the same cycle as configuring the dividers. Do not change the divider settings while the PLL is enabled.
3. Enable the PLL (CLK_PLL_CONFIGx.ENABLE = 1). Wait at least 1 µs for PLL circuits to start.
4. Wait until the PLL is locked before using the output. By default, the PLL output is bypassed to its reference clock and will automatically switch to the PLL output when it is locked. This behavior can be changed using CLK_PLL_CONFIGx.BYPASS_SEL. The status of the PLL can be checked by reading CLK_PLL_STATUSx. This register contains a bit indicating the PLL has locked. It also contains a bit indicating if the PLL lost the lock status.

20.3.2 Frequency locked loop (FLL)

EZ-USB™ FX10 family device contains one FLL, which resides on CLK_PATH0. Refer to the datasheet for the frequency range that can be input to the FLL and the frequency range that the FLL can output. This makes it possible to use the IMO to generate much higher clock frequencies for the rest of the system. **Figure 20-4** shows the block diagram of FLL.

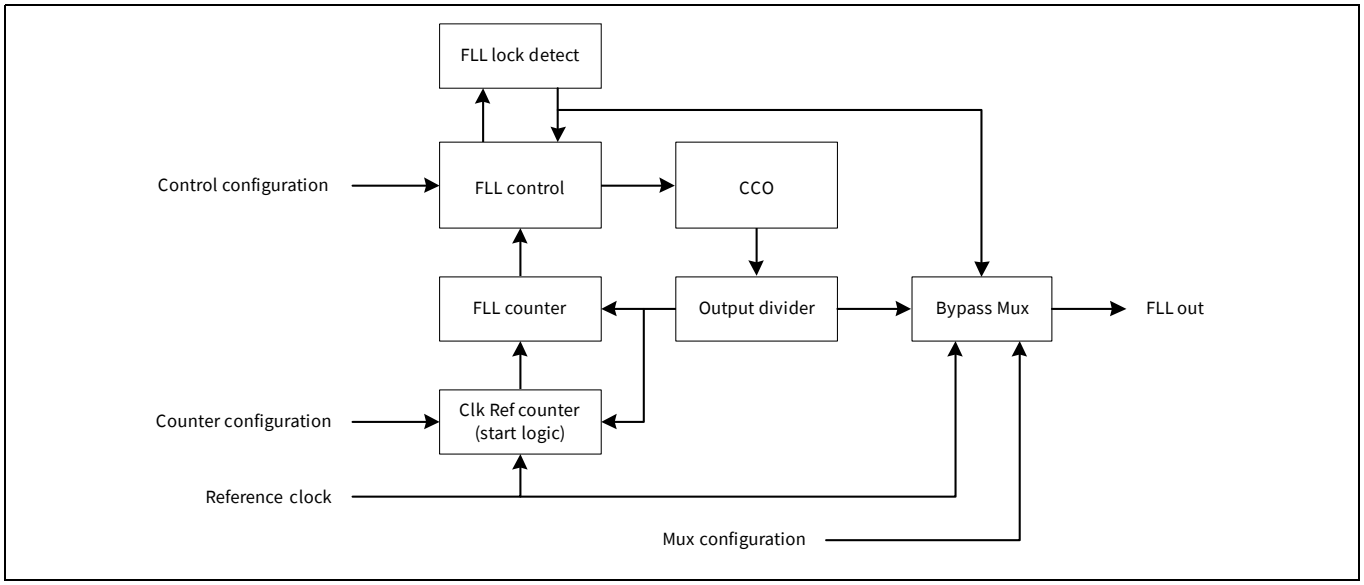


Figure 20-4. FLL block diagram

The FLL is similar in purpose to a PLL but is not equivalent; there are some differences:

- FLL can start up (lock) much faster than the PLL
- It consumes less current than the PLL
- FLL does not lock the phase. At the heart of the FLL is a current-controlled oscillator (CCO). The output frequency of this CCO is controlled by adjusting the trim of the CCO; this is done in hardware.
- FLL can produce a clock with good duty cycle through its divided clock output
- FLL reference clock can be the WCO, IMO (8 MHz), or any other periodic clock source

Clocking system

The CCO can output a stable frequency in the 48 MHz to 160 MHz range. This range is divided into five sub-ranges as shown by **Table 20-9**.

Table 20-9. CCO frequency range

CCO range	0	1	2	3	4
Fmin	48 MHz	64 MHz	85 MHz	113 MHz	150 MHz
Fmax	64 MHz	85 MHz	113 MHz	150 MHz	200 MHz

*Note: The output of the CCO has an option to enable a divide by two or not. For this device, the divide by two must always be enabled. The output range of the FLL is shown in **Table 20-9**.*

Within each range, the CCO output is controlled via a 9-bit trim field. This trim field is updated via hardware based on the control algorithm described here.

A reference clock must be provided to the FLL. This reference clock is typically the IMO, but can be many different clock sources. The FLL compares the reference clock against the CCO clock to determine how to adjust the CCO trim. Specifically, the FLL will count the number of CCO clock cycles inside a specified window of reference clock cycles. The number of reference clock cycles to count is set by CLK_FLL_CONFIG2.FLL_REF_DIV.

After the CCO clocks are counted, they are compared against an ideal value and an error is calculated. The ideal value is programmed into CLK_FLL_CONFIG.FLL_MULT.

As an example, the reference clock is the IMO (8 MHz), the desired CCO frequency is 100 MHz, the value for CLK_FLL_CONFIG2.FLL_REF_DIV is set to 146. This means that the FLL will count the number of CCO clocks within 146 clock periods of the reference clock. In one clock cycle of the reference clock (IMO), there should be 100 / 8 = 12.5 clock cycles of the CCO. Multiply this number by 146 and the value of CLK_FLL_CONFIG.FLL_MULT should be 1825.

If the FLL counts a value different from 1825, it attempts to adjust the CCO such that it achieves 1825 the next time it counts. This is done by scaling the error term with CLK_FLL_CONFIG3.FLL_LF_IGAIN and CLK_FLL_CONFIG3.FLL_LF_PGAIN. **Figure 20-5** shows how the error (err) term is multiplied by FLL_LF_IGAIN and FLL_LF_PGAIN and then summed with the current trim to produce a new trim value for the CCO. CLK_FLL_CONFIG4.CCO_LIMIT can be used to put an upper limit on the trim adjustment; this is not needed for most situations.

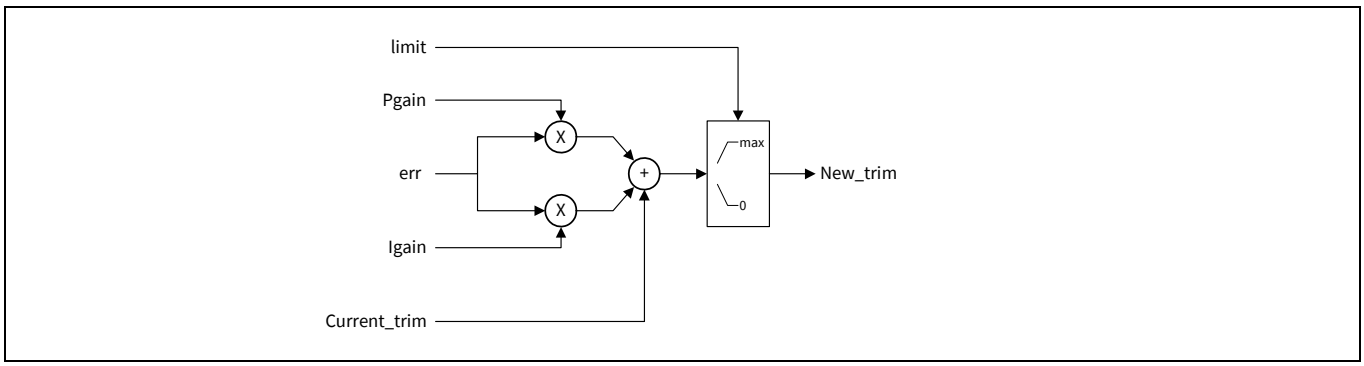


Figure 20-5. FLL error correction diagram

Clocking system

The FLL determines if it is “locked” by comparing the error term with CLK_FLL_CONFIG2.LOCK_TOL.

When the error is less than LOCK_TOL the FLL is considered locked.

After each adjustment to the trim the FLL can be programmed to wait a certain number of reference clocks before doing a new measurement. The number of reference clocks to wait is set in the CLK_FLL_CONFIG3.SETTLING_COUNT. It is recommended to set this such that the FLL waits ~1 µs before a new count. Therefore, if the 8 MHz IMO is used as the reference this field should be programmed to ‘8’.

When configuring the FLL there are two important factors that must be considered: lock time and accuracy. Accuracy is the closeness to the intended output frequency. These two numbers are inversely related to each other via the value of CLK_FLL_CONFIG2.FLL_REF_DIV.

Higher CLK_FLL_CONFIG2.FLL_REF_DIV values lead to higher accuracy, whereas lower CLK_FLL_CONFIG2.FLL_REF_DIV values lead to faster lock times.

In the example used previously the 8 MHz IMO was used as the reference, and the desired FLL output was 100 MHz. For that example, there are 12.5 CCO clocks in one reference clock. If the value for CLK_FLL_CONFIG2.FLL_REF_DIV is set to ‘1’ then CLK_FLL_CONFIG.FLL_MULT must be set to either ‘13’ or ‘12’. This will result in a CCO output of either 96 MHz or 104 MHz, and an error of 4 percent from the desired 100 MHz. Therefore, the best way to improve this is to increase CLK_FLL_CONFIG2.FLL_REF_DIV. However, the larger CLK_FLL_CONFIG2.FLL_REF_DIV is, the longer each measurement cycle takes, thus increasing the lock time. In this example, CLK_FLL_CONFIG2.FLL_REF_DIV was set to 146. This means each measurement cycle takes 146 × (1/8 MHz) = 18.25 µs, whereas when CLK_FLL_CONFIG2.FLL_REF_DIV is set to ‘1’, each measurement cycle takes 1 × (1/ 8 MHz) 0.125 µs.

Another issue with lower CLK_FLL_CONFIG2.FLL_REF_DIV values is that the minimum CLK_FLL_CONFIG2.LOCK_TOL is 1, so the output of the CCO can have an error of ± 1. In the example where CLK_FLL_CONFIG2.FLL_REF_DIV = 1 and CLK_FLL_CONFIG.FLL_MULT = 13, the MULT value can really be 12, 13, or 14 and still be locked.

This means the output of the FLL may vary between 96 and 112 MHz, which may not be desirable.

A choice must be made between faster lock times and more accurate FLL outputs. The biggest change is the value of REF_DIV. The CLK_FLL_STATUS register checks the status of FLL.

20.4 Clock trees

EZ-USB™ FX10 family clocks are distributed throughout the device, as shown in **Figure 20-2**. The clock trees are described in this section:

- Path clocks (CLK_PATH)
- High-frequency root clocks (CLK_HF)
- Low-frequency clock (CLK_LF)
- Timer clock (CLK_TIMER)

20.4.1 Path clocks

EZ-USB™ FX10 family device has several clock paths: CLK_PATH0 contains the FLL, CLK_PATH1 to CLK_PATH<P+1> contains the PLLs, and CLK_PATH<P+D> is a direct connection to the high-frequency root clocks. Note that the FLL and PLL can be bypassed if they are not needed. These paths are the input sources for the high-frequency clock roots.

Each clock path has a mux to determine the source clock for that path. This configuration is done in the CLK_PATH_SELECTx register. **Table 20-10** shows the clock path source selections.

Table 20-10. Clock path source selections

Name	Description
PATH_MUX	Selects the source for CLK_PATH 0: IMO 1: EXT_CLK 2: ECO 3: Reserved. Do not use. 4: DSI_MUX 5: Reserved 6-7: Reserved. Do not use.

The DSI mux is configured through the CLK_DSI_SELECTx register. **Table 20-11** shows the DSI mux source selections.

Table 20-11. DSI mux source selections

Name	Description
DSI_MUX	Selects the source for DSI_MUX 0-15: Reserved. Do not use. 16: ILO0 17: WCO 18: Reserved. Do not use. 19: Reserved. Do not use. 20: ILO1 21-31: Reserved. Do not use.

20.4.2 High-frequency root clocks

EZ-USB™ FX10 family has several high-frequency root clocks (CLK_HF). Each CLK_HF has a particular destination on the device; refer to the datasheet for details.

Each high-frequency root clock has a mux to determine its source. This configuration is done in the CLK_ROOT_SELECTx register. The number of CLK_PATH depends on the device. Refer to the datasheet for details.

Each CLK_HF has a predivider, which is set in the CLK_ROOT_SELECTx register.

CLK_HF0 is always enabled because it is the source of the CPU clock. Other CLK_HF can be enabled or disabled using CLK_ROOT_SELECTx.ENABLE.

Table 20-12. CLK_HF divider selection

Name	Description
ROOT_DVI	Selects predivider value for the clock root 0: No Divider 1: Divide clock by 2 2: Divide clock by 4 3: Divide clock by 8

20.4.3 Low-frequency root clocks

The low-frequency clock (CLK_LF) in EZ-USB™ FX10 family has four input options: ILO0, ILO1, and ECO_Prescaler. The source of CLK_LF is set in CLK_SELECT.LFCLK_SEL.

Table 20-13. LFCLK input selection bits LFCLK_SEL

Name	Description
LFCLK_SEL	LFCLK input clock selection 0: ILO0 2: Reserved. Do not use. 3: Reserved. Do not use. 4: ILO1 5: ECO Prescaler 6: Reserved 7: Reserved. Do not use.

20.4.4 Timer clock

The timer clock (CLK_TIMER) can be used as a clock source for the CPU SYSTICK timer. The source for CLK_TIMER can only be the IMO.

The CPU SYSTICK timer operation is configured by the CPUSS_SYSTICK_CTL register.

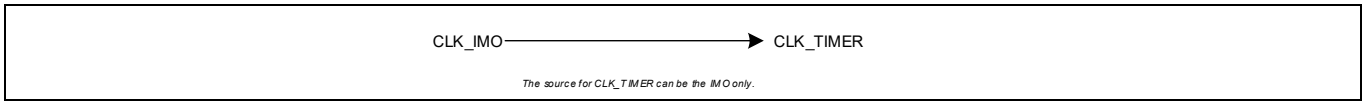


Figure 20-6. Connection to CLK_TIMER

Table 20-14. Timer clock selection

Name	Description
TIMER_SEL	Timer clock selection 0: CLK_IMO 1: CLK_HF0

20.4.5 Clock output function

The EXT_CLK terminal is bi-directional. The EXT_CLK terminal can input clock to EZ-USB™ FX10; it is also possible to output the EZ-USB™ FX10 internal clock. To use EXT_CLK as an input, configure the HSIOM to select EXT_CLK on a supported pin, and set that GPIO to the High Impedance digital drive mode. To use EXT_CLK as an output, configure the HSIOM to select EXT_CLK on a supported pin, and set that GPIO to a mode setting capable of driving the selected clock frequency.

The clock source available on EXT_CLK is CLK_HF3, which can choose any internal clock source including ECO. Note that CLK_HF3 is a shared resource; changing the predivider setting impacts all the connections.

20.5 CLK_HF distribution

EZ-USB™ FX10 has several CLK_HFs, which connect CLK_FAST_0, CLK_FAST_1, CLK_MEM, CLK_SLOW, CLK_PERI, CLK_GR and PCLK. CLK_FAST_0, CLK_FAST_1, CLK_MEM, CLK_SLOW, CLK_PERI and CLK_GR are source clocks of the CPU subsystem, SRSS, and some peripheral functions. Also, a part of CLK_HF connects to CSV. Refer to the device datasheet for the connection relationship between CLK_HF and the source clocks.

20.5.1 CLK_MEM

CLK_MEM clocks the CPUSS fast infrastructure. This clock is a divided version of CLK_HF. The divider for this clock is set in the CPUSS_MEM_CLOCK_CTL register.

20.5.2 CLK_SLOW

CLK_SLOW is the source clock for the Cortex®-M0+. This clock is a divided version of CLK_MEM. The divider for this clock is set in CPUSS_SLOW_CLOCK_CTL.INT_DIV.

20.5.3 CLK_FAST_0 and CLK_FAST_1

CLK_FAST_x clocks the CM4. This clock is a divided version of CLK_HF1; the divider for this clock is set in the CPUSS_FAST_0_CLOCK_CTL and CPUSS_FAST_1_CLOCK_CTL registers.

20.5.4 CLK_PERI

CLK_PERI is the source clock for all programmable peripheral clock dividers. It is a divided version of CLK_HF. The divider for this clock is set in CPUSS_PERI_CLOCK_CTL.INT_DIV.

20.5.5 PCLK

PCLK is the source of peripheral functions: CAN FD, TCPWM, SCB. For details, **“Peripheral clock dividers”** on page 301.

PCLK are of different kinds; it connects to various peripheral functions. Refer to the peripheral clocks section in the datasheet for more information.

20.5.6 CLK_GR

CLK_GR is a clock input to peripheral functions. It is grouped by the clock gater. Each GR_CLK is divided by the source clock and generated. The divider for the clock is configured in PERI_GRx_CLOCK_CTL.INT_DIV. CLK_GR can be enabled or disabled. The configuration is configured using the PERI_GRx_SL_CTL register.

20.5.7 CLK_TRC_DBG

CLK_TRC_DBG is a clock used for debugging. This clock is generated by dividing a clock source. See the device datasheet for the clock source. Clock division uses the CPUSS_TRC_DBG_CLOCK_CTL register.

20.6 Peripheral clock dividers

EZ-USB™ FX10 family peripherals such as SCBs and TCPWMs require a clock. These peripherals can be clocked only by a peripheral clock divider.

EZ-USB™ FX10 family has several peripheral clocks (PCLK) and peripheral clock dividers. Peripheral clock dividers can be 8-bit, 16-bit, 16.5-bit (16 integer bits and five fractional bits) and 24.5-bit (24 integer bits and five fractional bits). The output of any of these dividers can be routed to any peripheral. The divider also outputs a signal to divide (enable) the clock signal.

Refer to the datasheet for the number of dividers and assignment of peripheral clocks. **Figure 20-7** shows the peripheral clock divider diagram.

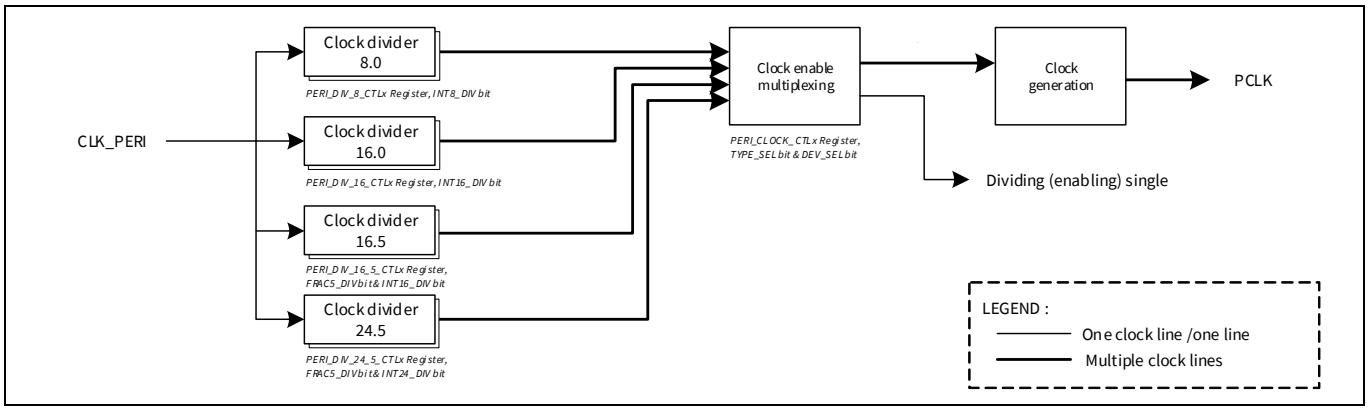


Figure 20-7. Peripheral clock divider

20.6.1 Fractional clock dividers

Fractional clock dividers allow the clock divisor to include a fraction of 0..31/32. For example, a 16.5-bit divider with an integer divide value of 3 generates a 16-MHz clock from a 48-MHz CLK_PERI. A 24.5-bit divider with an integer divide value of 4 generates a 12-MHz clock from a 48-MHz CLK_PERI. A 24.5-bit divider with an integer divide value of 3 and a fractional divider of 16 generates a 48 / (3 + 16/32) = 48 / 3.5 = 13.7 MHz clock from a 48-MHz CLK_PERI. Not all 13.7 MHz clock periods are equal in size; half of them will be three CLK_PERI cycles and half of them will be two CLK_PERI cycles.

Fractional dividers are useful when a high-precision clock is required (such as a UART/SPI serial interface). Fractional dividers are not used when a low jitter clock is required, because the clock periods have a jitter of one CLK_PERI cycle.

20.6.2 Peripheral clock divider configuration

The peripheral clock dividers are configured using registers from the Peripheral block; specifically PERI_PCLK_GRx_CLOCK_CTLy, PERI_PCLK_GRx_DIV_CMD, PERI_PCLK_GRx_DIV_8_CTLy, PERI_PCLK_GRx_DIV_16_CTLy, PERI_PCLK_GRx_DIV_16_5_CTLy, and PERI_PCLK_GRx_DIV_24_5_CTLy registers.

First the clock divider needs to be configured. This is done via the PERI_PCLK_GRx_DIV_8_CTLy, PERI_PCLK_GRx_DIV_16_CTLy PERI_PCLK_GRx_DIV_16_5_CTLy, and PERI_PCLK_GRx_DIV_24_5_CTLy registers. The number of each divider can be found in the Peripheral Clock Dividers section of the datasheet. Dividers not listed in “Divider Type” are not implemented. The divider selection is determined by the group number and the instance number. For example, use the PERI_PCLK_GR1_DIV_16_CTL3 register to configure the third 16-bit divider with Group 1.

After the divider is configured, use the PERI_PCLK_GRx_DIV_CMD register to enable the divider. This is done by setting the PERI_PCLK_GRx_DIV_CMD.DIV_SEL to the divider number you want to enable, and setting the PERI_PCLK_GRx_DIV_CMD.TYPE_SEL to the divider type. For example, if you wanted to enable the 0th 24.5-bit divider, write '0' to PERI_PCLK_GRx_DIV_CMD.DIV_SEL and ‘3’ to PERI_PCLK_GRx_DIV_CMD.TYPE_SEL. If you wanted to enable the tenth 16-bit divider, write ‘10’ to PERI_PCLK_GRx_DIV_CMD.DIV_SEL and ‘1’ to PERI_PCLK_GRx_DIV_CMD.TYPE_SEL.

To connect a peripheral to a specific divider, the PERI_PCLK_GRx_CLOCK_CTLy register is used. There is a PERI_PCLK_GRx_CLOCK_CTLy register corresponding to each PCLK mentioned in the Peripheral Clock section of the datasheet. For example, to select the twelfth 16-bit divider for PCLK with group 1 and output 29, configure DIV_SEL to ‘12’ and TYPE_SEL to ‘1’ in the PERI_PCLK_GR1_CLOCK_CTL29 register. Also, the ‘Output’ order matches the order of y of the PERI_PCLK_GRx_CLOCK_CTLy register.

20.6.2.1 Phase aligning dividers

For specific use cases, it is required to generate clocks that are phase aligned. For example, consider the generation of two gated clocks at 24 and 12 MHz, both of which are derived from a 48-MHz CLK_PERI. If phase alignment is not considered, the generated gated clocks can appear as follows.

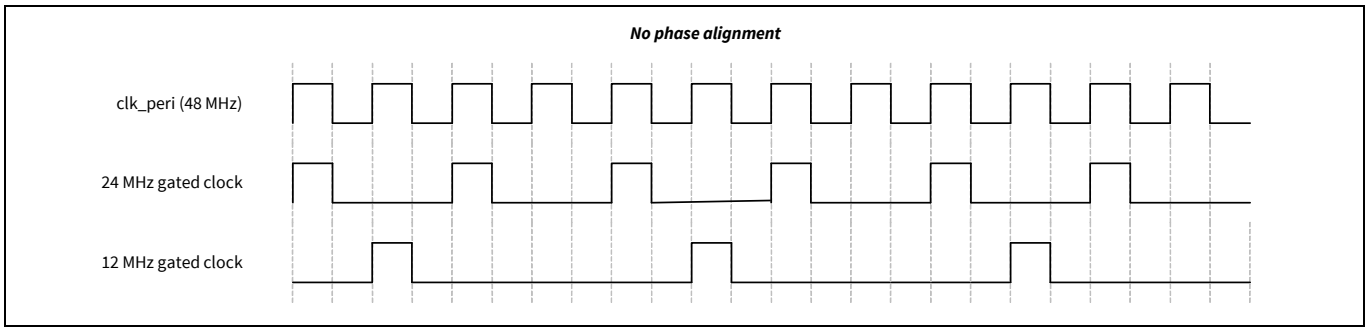


Figure 20-8. Non phase-aligned clock dividers

These clock signals may or may not be acceptable, depending on the logic functionality implemented on these two clocks. If the two clock domains communicate with each other, and the slower clock domain (12 MHz) assumes that each high/'1' pulse on its clock coincides with a high/'1' phase pulse in the higher clock domain (24 MHz), the phase misalignment is not acceptable. To address this, it is possible to have PCLK dividers produce clock signals that are phase-aligned with any of the other (enabled) clock dividers. Therefore, if (enabled) divider x is used to generate the 24-MHz clock, divider y can be phase-aligned to divider x and used to generate the 12 MHz clock. The generated gated clocks can appear as follows.

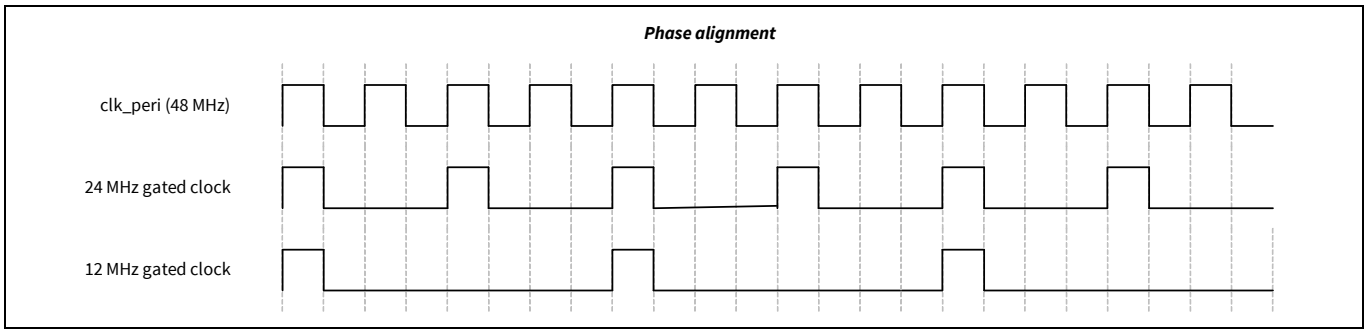


Figure 20-9. Phase-aligned clock dividers

Phase alignment also works for fractional divider values. If (enabled) divider x is used to generate the 38.4 MHz clock (divide by 1 8/32), divider y can be phase-aligned to divider x and used to generate the 19.2-MHz clock (divide by 2 16/32). The generated gated clocks can appear as follows.

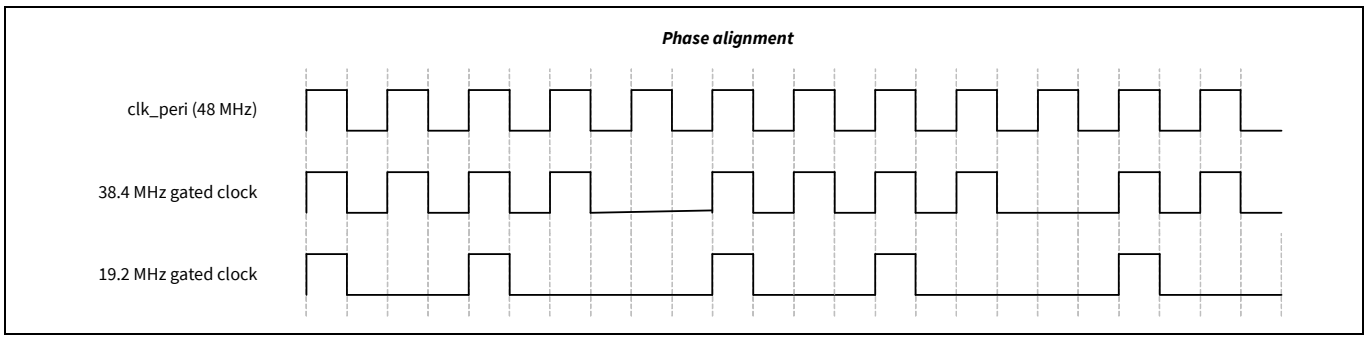


Figure 20-10. Phase-aligned fractional dividers

Divider phase alignment requires that the divider to which it is phase aligned is already enabled. This requires the dividers to be enabled in a specific order. This order can be represented by a divider dependency graph.

Phase alignment is implemented by controlling the start moment of the divider counters in hardware. When a divider is enabled, the divider counters are set to '0'. The divider counters will only start incrementing from '0' to the programmed integer and fractional divider values when the divider to which it is phase aligned has an integer counter value of '0'.

Note that the divider and clock multiplexer control register fields are all retained during Deep Sleep power mode. However, the divider counters that are used to implement the integer and fractional clock dividers are not. These counters are set to '0' during Deep Sleep power mode. Therefore, when transitioning from Deep Sleep to active power mode, all dividers (and clock signals) are enabled and phase-aligned by design.

Phase alignment is accomplished by setting the PERI_PCLK_GRx_DIV_CMD.PA_DIV_SEL and PERI_PCLK_GRx_DIV_CMD.PA_DIV_TYPE before enabling the clock. For example, to align the fourth 8-bit divider to the third 16-bit divider, set PERI_PCLK_GRx_DIV_CMD.DIV_SEL to '4', PERI_PCLK_GRx_DIV_CMD.TYPE_SEL to '0', PERI_PCLK_GRx_DIV_CMD.PA_DIV_SEL to '3', and PERI_PCLK_GRx_DIV_CMD.PA_TYPE_SEL to '1'.

20.7 Clock calibration counters

A feature of the clocking system in EZ-USB™ FX10 family is built-in hardware calibration counters. These counters can be used to compare the frequency of two clock sources against one another. The primary use case is to take a higher accuracy clock such as the ECO and use it to measure a lower accuracy clock such as the ILOx. The result of this measurement can then be used to trim the ILOx.

There are two counters: Calibration counter 1 is clocked off of calibration clock 1 (generally the high-accuracy clock) and it counts down; calibration counter 2 is clocked off of calibration clock 2 and it counts up. When calibration counter 1 reaches 0, calibration counter 2 stops counting up and its value can be read. From that value the frequency of calibration clock 2 can be determined with the following equation.

For example, if calibration clock 1 = 8 MHz, counter 1 = 1000, and counter 2 = 5

Calibration clock 2 Frequency = (5/1000) × 8 MHz = 40 kHz.

Calibration clock 1 and calibration clock 2 are selected with the CLK_OUTPUT_FAST register. All clock sources are available as a source for these two clocks. CLK_OUTPUT_SLOW is also used to select the clock source.

Calibration counter 1 is programmed in CLK_CAL_CNT1. Calibration counter 2 can be read in CLK_CAL_CNT2.

When calibration counter 1 reaches 0, CLK_CAL_CNT1.CAL_COUNTER_DONE is set.

$$\text{CalibrationClock2frequency} = \frac{\text{Counter2FinalValue}}{\text{Counter1InitialValue}} ¥ \text{CalibrationClock1Frequency}$$

20.8 Clock supervision (CSV)

20.8.1 Overview

This section provides an overview of the clock supervision features.

- The CSV circuit checks whether the frequency of the monitored clock is within the allowed frequency window If the monitored clock stops, or fails to start, it is detected as a low frequency
- All CLK_HFs have the CSV. All CLK_HF CSVs use the same reference clock (CLK_REF_HF). The reference clock is a selection of one of the active clock sources in the CSV_REF_SEL register. Typically, the IMO is selected (default).
- Note that ILO0 is supervised both by CLK_LF and CLK_REF_HF, CLK_LF is needed for Deep Sleep supervision and CLK_REF_HF is needed for accuracy (while active)
- CSV_REF monitors CLK_REF_HF with CLK_ILO0
- All CSV_HFs and CSV_REF are in the active domain
- There are two CSVs (CSV_LF and CSV_ILO) in the Deep Sleep domain. CSV_LF is used to monitor the selected CLK_LF clock with ILO0. CSV_ILO is used to monitor ILO0 with CLK_LF. ILO1 is provided to enable clock supervision of ILO0 during Deep Sleep when the WCO is not being used.

Figure 20-11 gives an overview of the location of CSVs for the EZ-USB™ FX10.

Refer to the datasheet for details about the relationship between the monitored clock and reference clock for each CSV component.

20.8.2 CSV operation

The basic operation principle of the CSV circuit is as follows:

Note: Period is the monitored clock count. Target is the reference clock count. Their time periods are the same in an ideal situation.

- The monitored clock generates a monitor event (period), and reference clock generates a lower and upper limit
- The Monitor event is compared against a lower or upper limit
- An error is reported if Monitor event ≤ lower limit, or monitor event > upper limit

Figure 20-12 shows an example of the CSV operation signal.

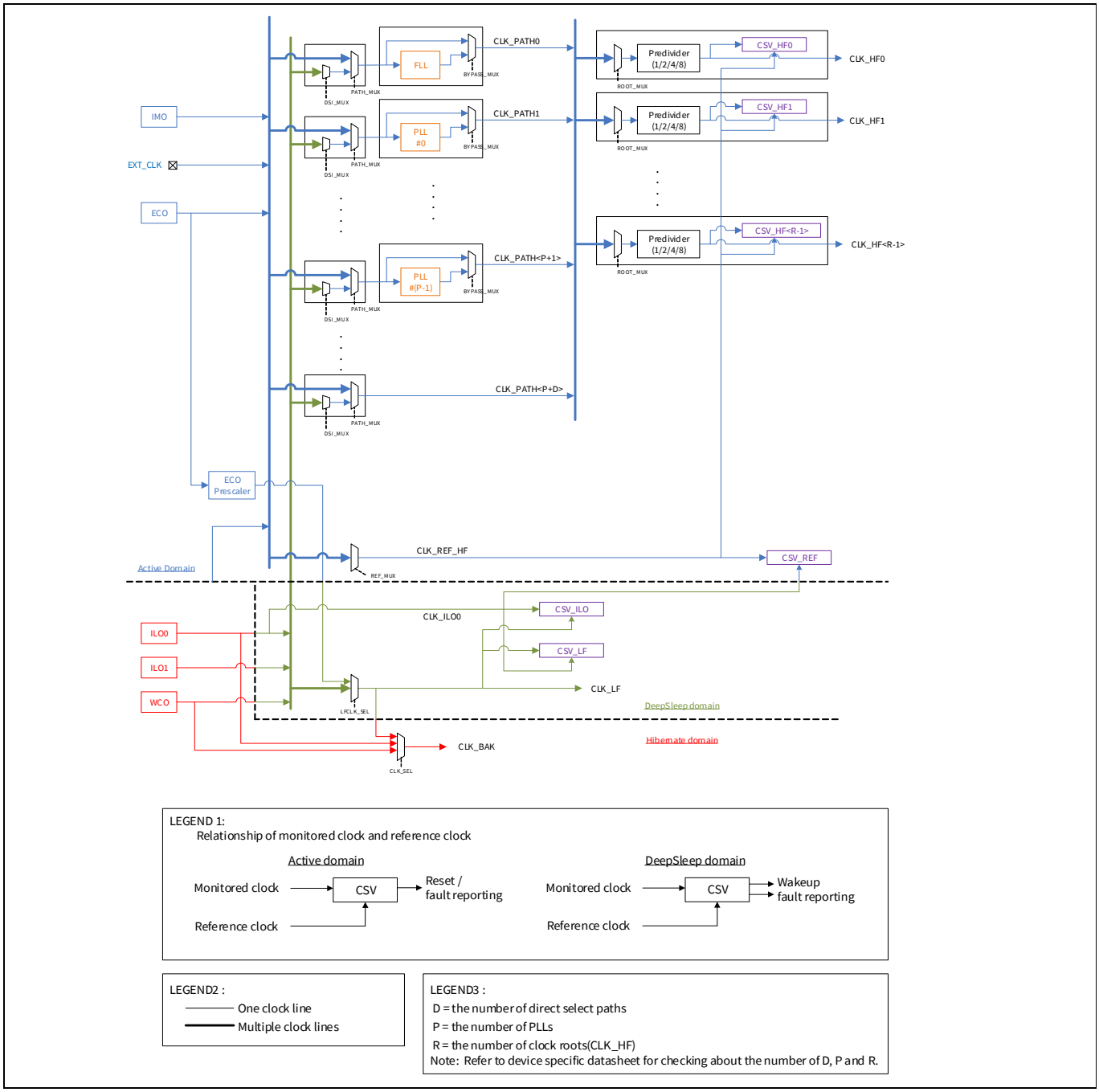
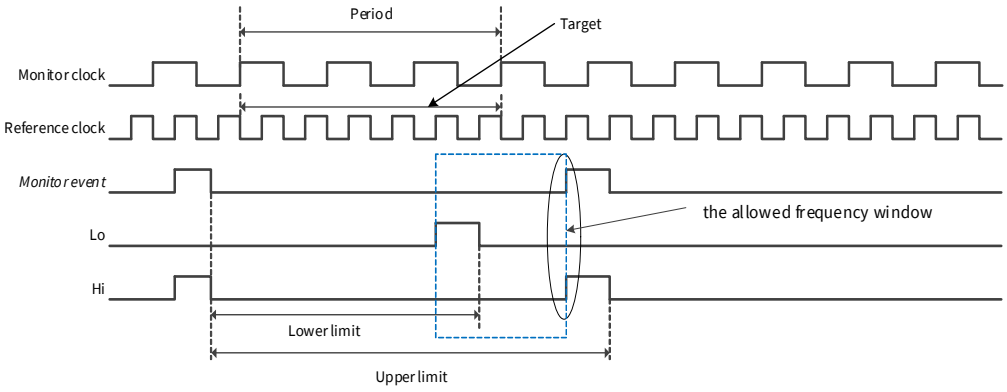
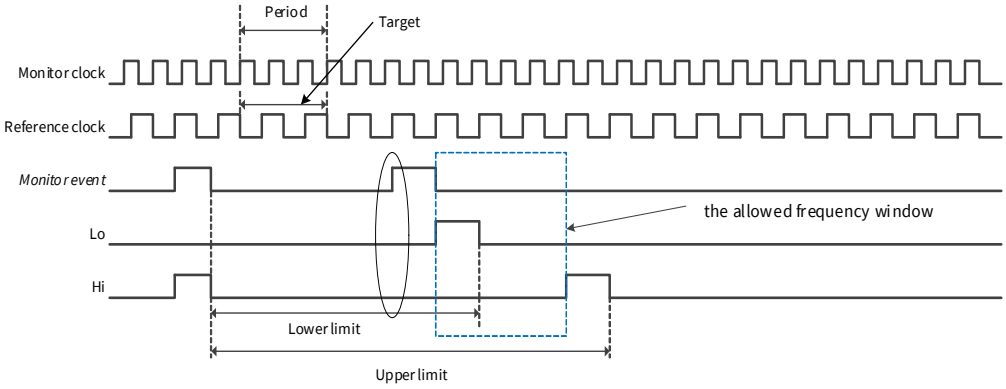


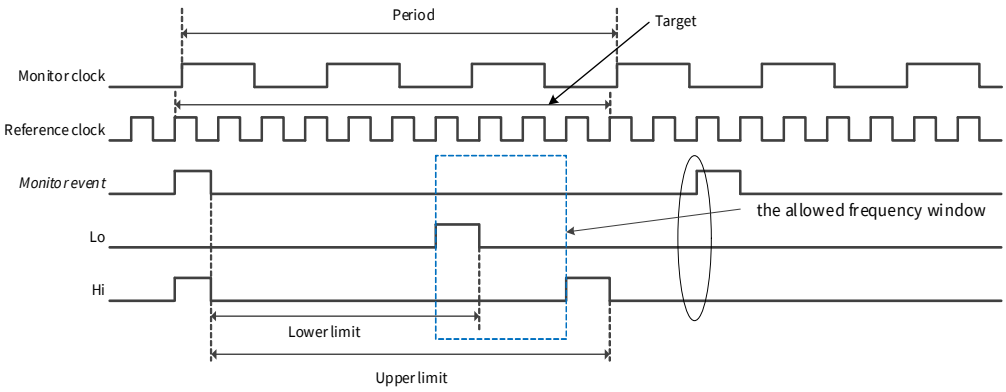
Figure 20-11. CSV diagram



(1) The error is not reported: Monitor event > Lower limit, and Monitor event <= Upper limit



(2) The error is reported: Monitor event <= Lower limit



(3) The error is reported: Monitor event > Upper limit

Figure 20-12. An example of CSV operation signal

Clocking system

Key points for CSV operation:

- Check the following parameters:
  - Reference clock frequency and tolerance (integer%)
  - Monitor clock frequency and the required tolerance (integer%)
    The required monitor clock tolerance should be equal to or larger than the reference clock tolerance.
- Determine “Target” with the following formula:
- Minimum Target = 200 / Reference clock tolerance
- For example, for a tolerance of one percent, the target must be at least 200. Increasing the target increases CSV accuracy and latency
- Determine “Period” with the following formula:
- Period = Target / (Reference clock frequency / Monitor clock frequency)
- Determine Lower_limit and Upper_limit with the following formula
  - Lower_limit = Period * ((Reference clock frequency * (1 – tolerance/100)) / (Monitor clock frequency * (1 + tolerance/100)))
  - Upper_limit = Period * ((Reference clock frequency * (1 + tolerance/100)) / (Monitor clock frequency * (1 – tolerance/100)))
- If the two clocks are asynchronous (typical) then there will be a one-cycle variation of Monitor Event periods
- The frequency window needs to account for the maximum clock tolerance on both clocks
- Lower_limit must be at least one less than Upper_limit
- All CSVs are initially off and require configuration before enabling
- The active domain CSVs are automatically stopped during Deep Sleep
  - After wakeup they will automatically restart
  - Each CSV has a software programmable startup time
- All active domain CSVs can either generate a reset or fault report
  - The CLK_HF0 CSV must use reset because the fault structure runs on CLK_HF0
  - All other CSVs should use a fault report to allow software to shut down
  - A fault report will result in an interrupt
- The Deep Sleep domain CSVs operate during active and Deep Sleep
- Deep Sleep domain CSVs can only report faults (no reset option)
  - A CSV error detection will wake up the system (if needed), which enables fault reporting
  - The fault report will result in an interrupt (no direct interrupt from CSV)
  - In Active/sleep/Deep Sleep it can generate a fault. If the device is in Deep Sleep, the fault causes a wakeup.
- All CSVs are enabled independent of the monitored clock; therefore:
  - Software should disable the CSV before stopping or reconfiguring the monitored clock (to avoid a false error detection)
  - The CSV needs to be reconfigured accordingly and restarted after the monitored clock is re-started
- The CSV_REF_SEL register elects a source to be used as the reference clock for CSV in the active domain. The registers to configure the CSV function are as follows. These registers can enable CSV, and can configure an action when CSV is activated.
  - CSV_HF_CSVx_REF_CTL
  - CSV_REF_CSVx_REF_CTL
  - CSV_LF_CSVx_REF_CTL
  - CSV_ILO_CSVx_REF_CTL
  - CSV_BAK_CSVx_REF_CTL

The following registers can configure upper limit and lower limit. Set the Lower_limit and Upper_limit as -1.

- CSV_HF_CSVx_REF_LIMIT
- CSV_REF_CSVx_REF_LIMIT
- CSV_LF_CSVx_REF_LIMIT
- CSV_ILO_CSVx_REF_LIMIT
- CSV_BAK_CSVx_REF_LIMIT

The following registers can configure PERIOD (time period). Set the Period as -1.

- CSV_HF_CSVx_MON_CTL
- CSV_REF_CSVx_MON_CTL
- CSV_LF_CSVx_MON_CTL
- CSV_ILO_CSVx_MON_CTL
- CSV_BAK_CSVx_REF_MON_CTL

- When CSV violation occurs, SRSS_CSV (0x18) is indicated in the IDX bit of the FAULT_STRUCT_STATUS register. The CSV fault report is captured in FAULT_STRUCT_DATA.

20.9 Registers

Registers related to the clock system are shown in **Table 20-15** and **Table 20-16**.

Table 20-15. Clock System registers in the PERI

Register	Name	Description
PERI_GRx_CLOCK_CTL	Divider Control register	This register configures division of CLK_GR.
PERI_GRx_SL_CTL	Slave Control	This register controls whether CLK_GR is enabled or disabled.
PERI_PCLK_GRx_CLOCK_CTLy	Divider Clock Control register	This register configures DIV_SEL and TYPE_SEL of PCLKs.
PERI_PCLK_GRx_DIV_8_CTLy	Divider Control register (for 8.0 divider)	This register controls the 8-bit divider.
PERI_PCLK_GRx_DIV_16_CTLy	Divider Control register (for 16.0 divider)	This register controls the 16-bit divider.
PERI_PCLK_GRx_DIV_16_5_CTLy	Divider Control register (for 16.5 divider)	This register controls the 16.5-bit divider.
PERI_PCLK_GRx_DIV_24_5_CTLy	Divider Control register (for 24.5 divider)	This register controls the 24.5-bit divider.
PERI_PCLK_GRx_DIV_CMD	Divider Command register	This register controls whether each divider is enabled or disabled.

Table 20-16. Clock System registers in the SRSS

Register	Name	Description
CLK_DSI_SELECTx	Clock DSI Select register	Configures DSI mux in the clock generation path. Each path has its own copy of this register. See **"Path clocks"** on page 298 for DSI signal connectivity list.
CLK_OUTPUT_FAST	Fast Clock Output Select register	Two signals can be selected to enable comparison of clocks. Fast clock output is observable only in (LP)Active/(LP)Sleep and are clamped low during Deep Sleep.
CLK_OUTPUT_SLOW	Slow Clock Output Select register	Two signals can be selected to enable comparison of clocks. Slow clock output is observable in (LP)Active/(LP)Sleep/Deep Sleep.
CLK_CAL_CNT1	Clock Calibration Counter 1	This register is a calibration counter that counts down by CLK_FAST selected by the CLK_OUTPUT_FAST register.
CLK_CAL_CNT2	Clock Calibration Counter 2	This register is a calibration counter that is counted up by CLK_FAST selected by the CLK_OUTPUT_FAST register.
CLK_PATH_SELECTx	Clock Path Select register	Selects a source for clock path. The output of this mux can be used as the root of a clock tree. If there is a PLL on the path, this mux output is the PLL reference clock. The related PLL register contains a mux to select whether the clock path uses the PLL output or is bypassed to the PLL reference clock.
CLK_ROOT_SELECTx	Clock Root Select register	Selects a root for a high-frequency clock tree and DSI input. Each clock root has a copy of this register.
CLK_SELECT	Clock Selection register	Clock source selection register.
CLK_ILO0_CONFIG	ILO0 Configuration	Configuration register for ILO0.
CLK_ILO1_CONFIG	ILO1 Configuration	Configuration register for ILO1.
CLK_IMO_CONFIG	IMO Configuration	Internal high-speed R/C Oscillator Configuration register. Note that this oscillator comes up active on power up. The oscillator provides the primary system clock (HFCLK) on power up until firmware configures differently. This oscillator is also used before system start to count out power up delays.
CLK_ECO_CONFIG	ECO Configuration register	Internal High-Speed Oscillator Configuration register for the external-crystal.
CLK_ECO_PRESCALE	ECO Prescaler Configuration register	Fractional prescaler value to bring down the ECO frequency to 32768 Hz if used as CLK_LF. Do not divider settings while ECO prescaler is enabled or enabling.
CLK_ECO_STATUS	ECO Status register	Error and status indications.
CLK_FLL_CONFIG	FLL Configuration register	This register contains frequency lock loop (FLL) configuration. FLL circuit settings should not be changed while it is a selected clock (connected to logic). This prevents clock glitches that can crash the logic.
CLK_FLL_CONFIG2	FLL Configuration register2	–
CLK_FLL_CONFIG3	FLL Configuration register3	–

Table 20-16. Clock System registers in the SRSS (continued)

Register	Name	Description
CLK_FLL_CONFIG4	FLL Configuration register4	–
CLK_FLL_STATUS	FLL Status register	This register indicates status for the FLL. This register is synchronized during an AHB read transaction. This causes a number of wait-states to be inserted in the transaction depending on the frequency ration between system and FLL frequency.
CLK_ECO_CONFIG2	ECO Configuration register 2	Internal High-Speed Oscillator Configuration register for the external-crystal.
CLK_PLL_CONFIGx	PLL Configuration register	This register contains PLL configuration. Each PLL has a copy of this register. PLL circuit settings should not be changed while it is a selected clock (connected to logic).
CLK_PLL_STATUSx	PLL Status register	This register indicates status for the PLL. Each PLL has a copy of this register.
CLK_ILO0_TRIM_CTL	ILO0 TRIM register	This register configures trim of ILO0 frequency.
CLK_ILO1_TRIM_CTL	ILO1 TRIM register	This register configures trim of ILO1 frequency.
CSV_REF_SEL	Select CSV Reference Clock register for active domain.	–
Active domain	Selects a source to be used as the reference clock for CSV in the active domain.	–
CSV_HF_CSVx_REF_CTL	Clock Supervision Reference Control register for CLK_HF	This register sets the control of CSV function.
CSV_REF_CSVx_REF_CTL	Clock Supervision Reference Control register for CLK_REF_LF	–
CSV_LF_CSVx_REF_CTL	Clock Supervision Reference Control register for CLK_LF	–
CSV_ILO_CSVx_REF_CTL	Clock Supervision Reference Control register for clk_ilo0	–
CSV_HF_CSVx_REF_LIMIT	Clock Supervision Reference Limits register for CLK_HF	This register sets LOWER and UPEER to be used for the CSV function.
CSV_REF_CSVx_REF_LIMIT	Clock Supervision Reference Limits register for CLK_REF_LF	–
CSV_LF_CSVx_REF_LIMIT	Clock Supervision Reference Limits register for CLK_LF	–
CSV_ILO_CSVx_REF_LIMIT	Clock Supervision Reference Limits register for clk_ilo0	–
CSV_HF_CSVx_MON_CTL	Clock Supervision Monitor Control register for CLK_HF	This register sets PERIOD to be used for the CSV function.
CSV_REF_CSVx_MON_CTL	Clock Supervision Monitor Control register for CLK_REF_LF	–
CSV_LF_CSVx_MON_CTL	Clock Supervision Monitor Control register for CLK_LF	–
CSV_ILO_CSVx_MON_CTL	Clock Supervision Monitor Control register for clk_ilo0	–

Table 20-16. Clock System registers in the SRSS (continued)

Register	Name	Description
CLK_PLL400Mx_CONFIG	400-MHz PLL Configuration Register	This register contains PLL configuration for 400-MHz PLL. Each PLL 400 has a copy of this register. Configuration settings should not be changed while it is a selected clock (connected to logic). This prevents clock glitches that can crash the logic.
CLK_PLL400Mx_CONFIG2	400-MHz PLL Configuration Register2	This register configures the fractional divider.
CLK_PLL400Mx_CONFIG3	400-MHz PLL Configuration Register3	This register configures the SSCG.
CLK_PLL400Mx_STATUS	400-MHz PLL Status Register	This register indicates status for a PLL400. Each PLL400 has a copy of this register.
CPUSS_FAST_0_CLOCK_CTL	Fast 0 Clock Control Register	This register configures CLK_FAST_0.
CPUSS_FAST_1_CLOCK_CTL	Fast 1 Clock Control Register	This register configures CLK_FAST_0.
CPUSS_SLOW_CLOCK_CTL	Slow Clock Control Register	This register configures CLK_SLOW.
CPUSS_PERI_CLOCK_CTL	Peripheral Interconnect Clock Control Register	This register configures CLK_PERI.
CPUSS_MEM_CLOCK_CTL	Memory Clock Control Register	This register configures CLK_MEM.
CPUSS_TRC_DBG_CLOCK_CTL	Trace and Debug Clock Control Register	This register configures CLK_TRC_DBG.

21 Reset system

The EZ-USB™ FX10 MCU family supports several types of resets that guarantee error-free operation during power up and allow the device to reset based on user-supplied external hardware or internal software reset signals. The EZ-USB™ FX10 MCU also contains hardware to enable the detection of certain resets.

21.1 Features

The EZ-USB™ FX10 MCU has these reset sources:

- Power-on reset (POR) to hold the device in reset while the power supply ramps up to the level required for the device to function properly
- Brownout reset (BOD) to reset the device if the power supply falls below the device specifications during normal operation
- External reset (XRES) to reset the device using an external input
- Watchdog timer (WDT) reset to reset the device if the firmware execution fails to periodically service the watchdog timer
- Software initiated reset to reset the device on demand using firmware
- Logic-protection fault resets to reset the device if unauthorized operating conditions occur
- Clock-supervision logic resets to reset the device when clock-related errors occur

21.2 Architecture

The following sections provide a description of the reset sources available in the EZ-USB™ FX10 MCU family. **Table 21-1** lists all the reset sources.

Table 21-1. MCU reset sources

Reset source	Reset conditions	Availability in System Power modes	Cause detection
POR	This reset condition occurs during device power-up. POR holds the device under reset until the V_{DDD} reaches the threshold voltage as specified in the device datasheet.	All	Not inferable using the reset cause registers.
BOR	V_{DDD} falls below the minimum logic operating voltage specified in the device datasheet.	All	Not inferable using the reset cause registers.
WDT Reset	WDT is not reset by firmware within the configured reset time period of the WDT.	All	RESET_HWWDT bit or RESET_SWWDT0 to RESET_SWWDT3 status bits of the RES_CAUSE register is set when a watchdog reset occurs. This bit remains set until cleared by the firmware or until a POR, XRES, or BOD reset occurs. All other resets leave this bit unaltered.
Software Reset	SYSRESETREQ bit is set by the firmware in the CM0_AIRCR/CM4_AIRCR register.	Active	RESET_SOFT bit of the RES_CAUSE register is set

Table 21-1. MCU reset sources (continued)

Reset source	Reset conditions	Availability in System Power modes	Cause detection
External Reset	Logic LOW input to XRES pin	All	Not inferable using the reset cause registers.
Logic Protection Fault Reset	Unauthorized protection violation	Active, Deep Sleep modes	RESET_ACT_FAULT or RESET_DPSLP_FAULT bits of the RES_CAUSE register is set.
Clock-Supervision Logic Reset	Loss of a high-frequency clock or watch-crystal clock, or due to a high-frequency clock error.	Active, Deep Sleep modes	RESET_CSV_WCO_LOSS bit of the RES_CAUSE register is set when WCO clock is lost. The RESET_CSV_HF_LOSS of RES_CAUSE2 register can be used to identify resets caused by the loss of a high-frequency clock. RESET_CSV_HF_FREQ field can be used to identify resets caused by the frequency error of a high-frequency clock.

21.2.1 Power-on reset (POR)

The POR is provided to keep the system in a reset state during power-up. POR holds the device in reset until the supply voltage, V_{DDD} reaches the datasheet specification. The POR activates automatically at power-up. Refer to the device datasheet for details on the POR trip-point levels.

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, then the reset is caused by POR, BOD, or XRES.

21.2.2 Brownout reset

The brownout reset monitors the chip digital voltage supply V_{DDD} and generates a reset if V_{DDD} falls below the minimum logic operating voltage specified in the device datasheet. See the **"Power supply and monitoring"** on page 275 for more details.

BOD events do not set a reset cause status bit, but in some cases they can be detected. In some BOD events, V_{DDD} will fall below the minimum logic operating voltage specified by the datasheet, but remain above the minimum logic retention voltage.

21.2.3 Watchdog timer reset

The watchdog timer reset causes a reset if the WDT is not serviced by the firmware within a specified time limit.

The RESET_WDT bit or RESET_MCWDT0 to RESET_MCWDT3 status bits of the RES_CAUSE register is set when a watchdog reset occurs. This bit remains set until cleared by the firmware or until a POR, XRES, or BOD reset occurs. All other resets leave this bit unaltered.

21.2.4 Software initiated reset

The software initiated reset is a mechanism that allows the CPU to request a reset. The Cortex-M0+ and Cortex-M4 Application Interrupt and Reset Control registers (CM0_AIRCR and CM4_AIRCR, respectively) can request a reset by writing a '1' to the SYSRESETREQ bit of the respective registers.

Note that a value of 0x5FA should be written to the VECTKEY field of the AIRCR register before setting the SYSRESETREQ bit; otherwise, the processor ignores the write. See the **"CPU subsystem (CPUSS)"** on page 63 and **Arm® documentation** on AIRCR for more details.

The RESET_SOFT status bit of the RES_CAUSE register is set when a software reset occurs. This bit remains set until cleared by firmware or until a POR, XRES, or BOD reset occurs. All other resets leave this bit unaltered.

21.2.5 External reset

External reset (XRES) is a reset triggered by an external signal that causes immediate system reset when asserted. The XRES pin is active low – a logic '1' on the pin has no effect and a logic '0' causes reset. The pin is pulled to logic '1' inside the device. XRES is available as a dedicated pin. For detailed pinout, refer to the pinout section of the device datasheet.

The XRES pin holds the device in reset as long as the pin input is '0'. When the pin is released (changed to logic '1'), the device goes through a normal boot sequence. The logical thresholds for XRES and other electrical characteristics are listed in the Electrical Specifications section of the device datasheet. The XRES is available in all power modes.

An XRES event does 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, then the reset is caused by POR, BOD, or XRES.

21.2.6 Logic protection fault reset

The logic protection fault reset detects any unauthorized protection violations and causes the device to reset if they occur. One example of a protection fault is reaching a debug breakpoint while executing privileged code.

The RESET_ACT_FAULT or RESET_DPSLP_FAULT bits of the RES_CAUSE register is set when a protection fault occurs in Active or Deep Sleep modes, respectively. These bits remain set until cleared or until a POR, XRES, or BOD reset. All other resets leave this bit unaltered.

21.2.7 Clock-supervision logic reset

The clock-supervision logic initiates a reset due to the loss of a high-frequency clock or watch-crystal clock, or due to a high-frequency clock error.

The RESET_CSV_WCO_LOSS bit of the RES_CAUSE register is set when the clock supervision logic requests a reset due to the loss of a watch-crystal clock (if enabled).

The RESET_CSV_HF_LOSS is a 16-bit field in the RES_CAUSE2 register that can be used to identify resets caused by the loss of a high-frequency clock. Similarly, the RESET_CSV_HF_FREQ field can be used to identify resets caused by the frequency error of a high-frequency clock. For more information on clocks, see the **"Clocking system"** on page 290.

21.3 Identifying reset sources

When the device comes out of reset, it is often useful to know the cause of the most recent or even older resets. This is achieved through the RES_CAUSE and RES_CAUSE2 registers. These registers have specific status bits allocated for some of the reset sources. These registers record the occurrences of WDT reset, software reset, logic-protection fault, and clock-supervision resets. However, these registers do not record the occurrences of POR, BOD, and XRES resets. The bits in these registers are set on the occurrence of the corresponding reset and remain set after the reset, until cleared by the firmware or a loss of retention, such as a POR, XRES, or BOD.

The reset sources can be inferred to some extent by the status of the RES_CAUSE and RES_CAUSE2 registers, as shown in **Table 21-2**.

Table 21-2. Reset cause bits to detect reset source

Register	Bitfield	Number of bits	Description
RES_CAUSE	RESET_WDT	1	A hardware WDT reset has occurred since the last power cycle.
RES_CAUSE	RESET_ACT_FAULT	1	Fault logging system requested a reset from its Active logic.
RES_CAUSE	RESET_DPSLP_FAULT	1	Fault logging system requested a reset from its Deep Sleep logic.
RES_CAUSE	RESET_CSV_WCO_LOSS	1	Clock supervision logic requested a reset due to loss of a watch-crystal clock.
RES_CAUSE	RESET_SOFT	1	A CPU requested a system reset through its SYSRESETREQ. This can be done via a debugger probe or in firmware.
RES_CAUSE	RESET_MCWDT0	1	Multi-counter WDT reset #0 has occurred since the last power cycle.
RES_CAUSE	RESET_MCWDT1	1	Multi-counter WDT reset #1 has occurred since the last power cycle.
RES_CAUSE	RESET_MCWDT2	1	Multi-counter WDT reset #2 has occurred since the last power cycle.
RES_CAUSE	RESET_MCWDT3	1	Multi-counter WDT reset #3 has occurred since the last power cycle.
RES_CAUSE2	RESET_CSV_HF_LOSS	16	Clock supervision logic requested a reset due to loss of a high-frequency clock. Each bit index K corresponds to a clk_hf<K>. Unimplemented clock bits return zero.
RES_CAUSE2	RESET_CSV_HF_FREQ	16	Clock supervision logic requested a reset due to frequency error of a high-frequency clock. Each bit index K corresponds to a clk_hf<K>. Unimplemented clock bits return zero.

For more information, see the RES_CAUSE and RES_CAUSE2 registers in the registers reference manual.

If these methods cannot detect the cause of the reset, then it can be one of the non-recorded and non-retention resets: BOD, POR, or XRES. These resets cannot be distinguished using on-chip resources.

21.4 Register list

Table 21-3. Reset System register list

Register	Description
RES_CAUSE	Reset cause observation register
RES_CAUSE2	Reset cause observation register 2
CM4_AIRCR	Application interrupt and reset control register of Cortex®-M4
CM0_AIRCR	Application interrupt and reset control register of Cortex®-M0+

22 Watchdog timer

The watchdog timer (WDT) is a hardware timer that automatically resets the device in the event of an unexpected firmware execution path. The WDT, if enabled, must be serviced periodically in firmware to avoid a reset. Otherwise, the timer elapses and generates a device reset. In addition, the WDT can be used as an interrupt source or a wakeup source in low-power modes.

The EZ-USB™ FX10 MCU family includes one free-running WDT and two multi-counter WDTs (MCWDT). The WDT has a 16-bit counter. Each MCWDT has two 16-bit counters and one 32-bit counter. Thus, the watchdog system has a total of seven counters – five 16-bit and two 32-bit. All 16-bit counters can generate a watchdog device reset. All seven counters can generate an interrupt on a match event.

22.1 Features

The EZ-USB™ FX10 MCU WDT supports these features:

- One 16-bit free-running WDT with:
  - ILO as the input clock source
  - Device reset generation if not serviced within a configurable interval
  - Periodic Interrupt/wakeup generation in LP Active, LP Sleep, and Deep Sleep power modes
- Two MCWDTs, each supporting:
  - Device reset generation if not serviced within a configurable interval
  - LFCLK (ILO or WCO) as the input clock source
  - Periodic interrupt/wake up generation in LP Active, LP Sleep, and Deep Sleep power modes
  - Two 16-bit and one 32-bit independent counters, which can be configured as a single 64-bit or 48-bit (with one 16-bit independent counter), or two 32-bit cascaded counters

22.2 Architecture

The EZ-USB™ FX10 MCU supports 174 system interrupts. The interrupts are routed to both the CPU cores. In the case of CM4 only the first 39 interrupts are routed to WIC while all 174 interrupts are routed to NVIC. The CM0 has access to only eight interrupts of the maximum supported 32 interrupts. The 174 interrupt sources are multiplexed and at a time eight interrupt sources can be connected to the CM0. The CPUSS_CM0_SYSTEM_INT_CTLx register decides which interrupts are connected to the CM0. See the **“Interrupts”** on page 89 for details on how to configure the interrupt for the free-running WDT/MCWDT to the required CPU. The free-running WDT/MCWDT resource must be used by one CPU only; it is not intended for simultaneous use by both CPUs because of the complexity involved in coordination.

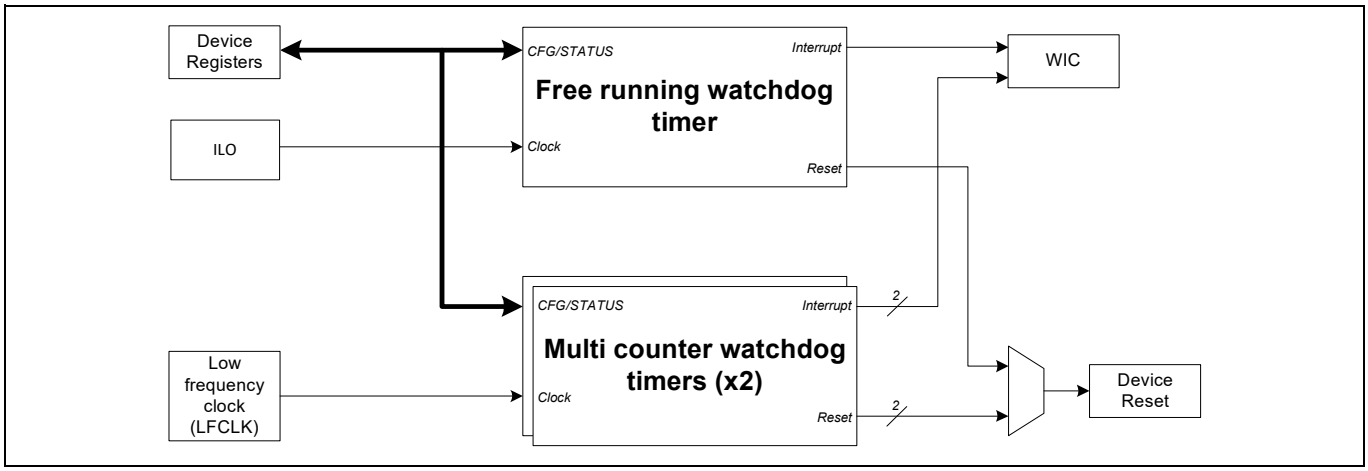


Figure 22-1. Watchdog timer block diagram

22.3 Free-running WDT

22.3.1 Overview

Figure 22-2 shows the functional overview of the free-running WDT. The WDT has a free-running wraparound up-counter with a maximum of 16-bit resolution. The counter is clocked by the ILO. The timer can generate an interrupt on match and a reset event on the third unhandled interrupt. The number of bits used for a match comparison is configurable as depicted in Figure 22-2.

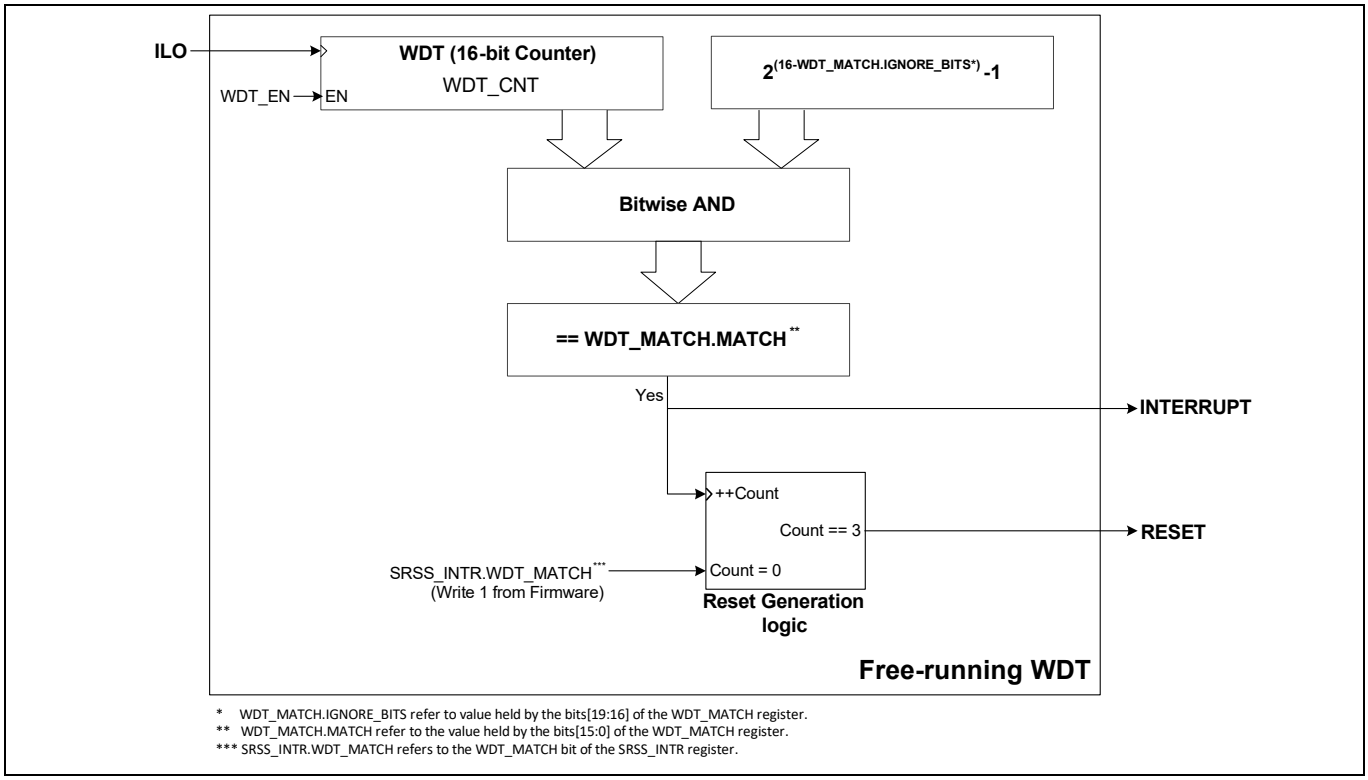


Figure 22-2. Free-running WDT Functional Diagram

When enabled, the WDT counts up on each rising edge of the ILO. When the counter value (WDT_CNT register) equals the match value stored in MATCH bits [15:0] 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 16-bit boundary (65535) at which point, it wraps around to 0 and counts up. The match interrupt is generated every time the counter value equals the match value.

The WDT_MATCH bit of the SRSS_INTR register is set whenever a WDT match interrupt occurs. This interrupt must be cleared by writing a '1' to the same bit. Clearing the interrupt resets the watchdog. If the firmware does not clear the interrupt for two consecutive occasions, the third interrupt generates a device reset.

In addition, the WDT provides an option to set the number of bits to be used for comparison. The IGNORE_BITS (bits [19:16] of the WDT_MATCH register) is used for this purpose. These bits configure the number of MSbs to ignore from the 16-bit count value while performing the match. For instance, when the value of these bits equals 3, the MSb 3 bits are ignored while performing the match and the WDT counter behaves similar to a 13-bit counter. Note that these bits do not reduce the counter size – the WDT_CNT register still counts from 0 to 65535 (16-bit).

Watchdog timer

The WDT can be enabled or disabled using the WDT_EN bit [0] of the WDT_CTL register. The WDT_CTL register provides a mechanism to lock the WDT configuration registers. The WDT_LOCK bits [31:30] control the lock status of the WDT registers. These bits are special bits, which can enable the lock in a single write; to release the lock, two different writes are required. The WDT_LOCK bits protect the WDT_EN bit, WDT_MATCH register, CLK_ILO_CONFIG register, and LFCLK_SEL bits [1:0] of the CLK_SELECT register. Note that the WDT_LOCK bits are not retained in Deep Sleep mode and reset to their default (LOCK) state after a Deep Sleep wakeup. As a result, to update any register protected by the WDT_LOCK bits after a Deep Sleep wakeup, a WDT UNLOCK sequence should be issued before the register update.

Table 22-1 explains various registers and bitfields used to configure and use the WDT.

Table 22-1. Free-running WDT Configuration options

Register [Bit_Pos]	Bit_Name	Description
WDT_CTL[0]	WDT_EN	Enable or disable the watchdog reset 0: WDT reset disabled 1: WDT reset enabled
WDT_CTL[31:30]	WDT_LOCK	Lock or unlock write access to the watchdog configuration and clock related registers. When the bits are set, the lock is enabled. 0: No effect 1: Clear bit 0 2: Clear bit 1 3: Set both bit 0 and 1 (lock enabled) WDT will lock on a reset. This field is not retained in Deep Sleep mode, so the WDT will be locked after wakeup from these modes.
WDT_CNT[15:0]	COUNTER	Current value of WDT counter
WDT_MATCH[15:0]	MATCH	Match value to a generate watchdog match event
WDT_MATCH[19:16]	IGNORE_BITS	Number of MSbs of the WDT_CNT register to ignore for comparison with the MATCH value. Up to 12 MSbs can be ignored; settings above 12 act same as a setting of 12.
SRSS_INTR[0]	WDT_MATCH	WDT interrupt request This bit is set whenever a watchdog match event happens. The WDT interrupt is cleared by writing a ‘1’ to this bit
SRSS_INTR_MASK[0]	WDT_MATCH	Mask for the WDT interrupt 0: WDT interrupt is blocked 1: WDT interrupt is forwarded to CPU

22.3.2 Watchdog reset

A watchdog is typically used to protect the device against firmware/system crashes or faults. When the WDT is used to protect against system crashes, 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 function of the firmware crashes or is in an endless loop, the WDT interrupt vector can still be intact and feed the WDT periodically. Note that when the debug probe is connected, the device reset is blocked and an interrupt is generated instead.

The safest way to use the WDT against system crashes is to:

- Configure the watchdog reset period such that firmware is able to reset the watchdog at least once during the period, even along the longest firmware delay path. If both CM4 and CM0 are being used then, use the free-running WDT for one of the CPU and one MCWDT for the other CPU to prevent firmware crashes on either core.
- Reset (feed) the watchdog by clearing the interrupt bit regularly in the main body of the firmware code by writing a '1' to the WDT_MATCH bit in the SRSS_INTR register. Note that this does not reset the watchdog counter, it feeds only the watchdog so that it does not cause a reset for the next two match events.

Do not reset the watchdog (clear interrupt) in the WDT interrupt service routine (ISR) if WDT is being used as a reset source to protect the system against crashes. Therefore, do not use the WDT reset feature and ISR together.

The recommended steps to use WDT as a reset source are as follows:

1. Make sure the WDT configuration is unlocked by clearing the WDT_LOCK bits[31:30] of the WDT_CTL register. Note that clearing the bits requires two writes to the register with each write clearing one bit as explained in **Table 22-1**.
2. Write the desired IGNORE_BITS in the WDT_MATCH register to set the counter resolution to be used for the match.
3. Write any match value to the WDT_MATCH register. The match value does not control the period of watchdog reset as the counter is not reset on a match event. This value provides an option to control only the first interrupt interval, after that the successive interrupts' period is defined by the IGNORE_BITS. Approximate watchdog period (in seconds) is given by the following equation:

$$\text{Watchdog reset period} = \text{ILO}_{\text{period}} \times (2 \times 2^{16 - \text{IGNORE\_BITS}} + \text{WDT\_MATCH}) \quad (22.1)$$

4. Set the WDT_MATCH bit in the SRSS_INTR register to clear any pending WDT interrupt.
5. Enable ILO by setting the ENABLE bit [31] of the CLK_ILO_CONFIG register.
6. Enable the WDT by setting the WDT_EN bit in WDT_CTL register.
7. Lock the WDT and ILO configuration by writing '3' to the WDT_LOCK bits. This also locks the LFCLK_SEL bits of the CLK_SELECT register.
8. In the firmware, write '1' to the WDT_MATCH bit in the SRSS_INT register to feed (clear interrupt) the watchdog.

22.3.3 Watchdog interrupt

In addition to generating a device reset, the WDT can be used to generate interrupts. Note that interrupt servicing and watchdog reset cannot be used simultaneously using the free-running WDT.

The watchdog counter can send interrupt requests to the CPU in CPU Active power modes and to the wakeup interrupt controller (WIC) in CPU Sleep and Deep Sleep power modes. It works as follows:

- **CPU Active mode**: In Active power mode, the WDT can send the interrupt to the CPU. The CPU acknowledges the interrupt request and executes the ISR. Clear the interrupt in the ISR.
- **CPU Sleep or Deep Sleep mode**: In this mode, the CPU is powered down. Therefore, the interrupt request from the WDT is directly sent to the WIC, which then wakes up the CPU. The CPU acknowledges the interrupt request and executes the ISR. Clear the interrupt in the ISR.

For more details on device power modes, see the **"Device power modes"** on page 281.

Because of its free-running nature, the WDT should not be used for periodic interrupt generation. Use the MCWDT instead; see **"Multi-counter WDTs"** on page 323. The MCWDT counters can be used to generate periodic interrupts. If absolutely required, follow these steps to use the WDT as a periodic interrupt generator:

1. Unlock the WDT if this is the first update to the WDT registers after a Deep Sleep wakeup, or a device reset.
2. Write the desired IGNORE_BITS in the WDT_MATCH register to set the counter resolution to be used for the match.
3. Write the desired match value to the WDT_MATCH register.
4. Set the WDT_MATCH bit in the SRSS_INTR register to clear any pending WDT interrupt.
5. Enable the WDT interrupt to CPU by setting the WDT_MATCH bit in SRSS_INTR_MASK.
6. Enable SRSS interrupt to the CPU by configuring the appropriate ISER register (see the **"Interrupts"** on page 89 for details).
7. In the ISR, unlock the WDT; clear the WDT interrupt and add the desired match value to the existing match value. By doing so, another interrupt is generated when the counter reaches the new match value (period).

22.4 Multi-counter WDTs

22.4.1 Overview

Figure 22-3 shows the functional overview of a single multi-counter WDT block. EZ-USB™ FX10 MCU has two MCWDT blocks. Each MCWDT block includes two 16-bit counters (MCWDTx_WDT0 and MCWDTx_WDT1) and one 32-bit counter (MCWDTx_WDT2). These counters can be configured to work independently or in cascade (up to 64-bit). The 16-bit counters can generate an interrupt or reset the device. The 32-bit counter can only generate an interrupt. All the counters are clocked by LFCLK.

Note: Because EZ-USB™ FX10 MCU includes two CPUs (Cortex®-M0+ and Cortex®-M4), associate one MCWDT block to only one CPU during runtime. Although both the MCWDT blocks are available for both CPUs, a single MCWDT is not intended to be used by multiple CPUs simultaneously; however, a CPU can be associated with 0 or more MCWDTs.

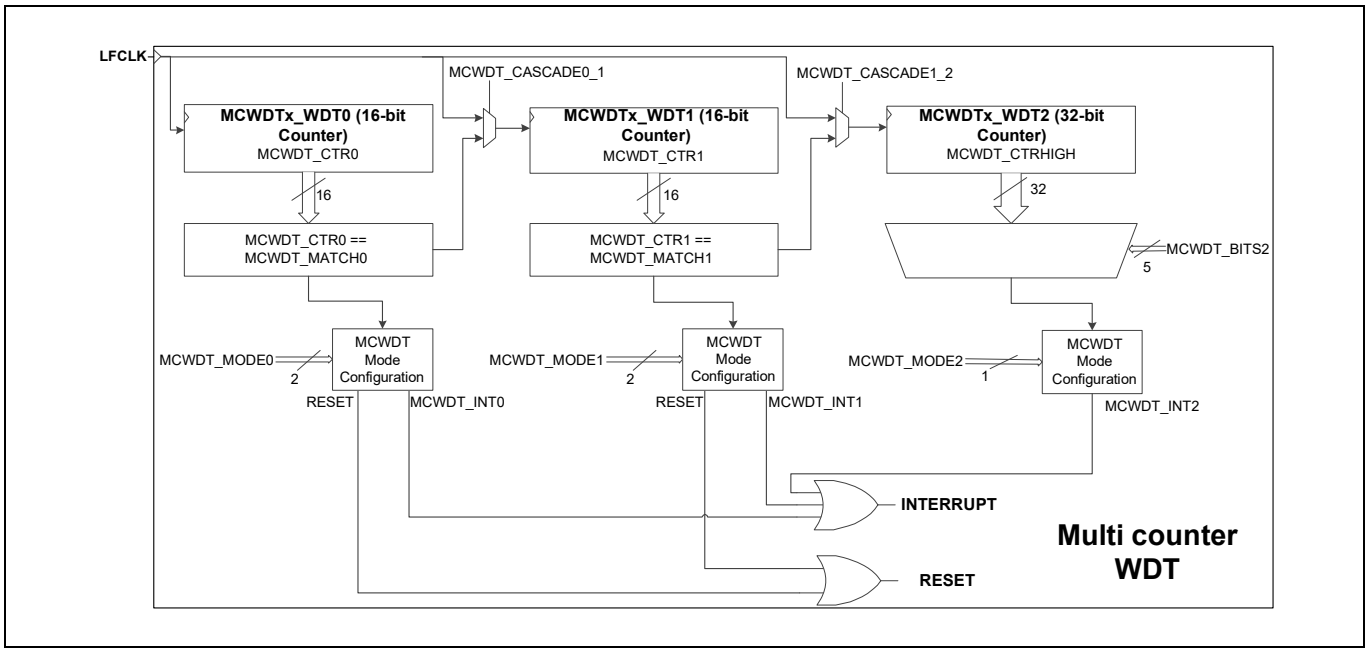


Figure 22-3. Multi-counter WDT functional diagram

22.4.1.1 MCWDTx_WDT0 and MCWDTx_WDT1 counters operation

MCWDTx_WDT0 and MCWDTx_WDT1 are 16-bit up counters, which can be configured to be a 16-bit free-running counter or a counter with any 16-bit period. These counters can be used to generate an interrupt or reset.

The WDT_CTR0 bits [15:0] and WDT_CTR1 bits [16:31] of the MCWDTx_CNTLOW register hold the current counter values of MCWDTx_WDT0 and MCWDTx_WDT1 respectively. The WDT_MATCH0 bits [15:0] and WDT_MATCH1 bits [16:31] of the MCWDTx_MATCH register store the match value for MCWDTx_WDT0 and MCWDTx_WDT1 respectively. The WDT_MODEx bits of the MCWDTx_CONFIG register configure the action the watchdog counter takes on a match event (WDT_MATCHx == WDT_CTRx). The MCWDTx_WDT0/WDT1 counters perform the following actions:

- Assert interrupt (WDT_INTx) on match
- Assert a device reset on match
- Assert an interrupt on match and a device reset on the third unhandled interrupt

In addition to generating reset and interrupt, the match event can be configured to clear the corresponding counter. This is done by setting the WDT_CLEARx bit of the MCWDTx_CONFIG register. MCWDTx_WDT0/WDT1 counter operation is shown in **Figure 22-4**.

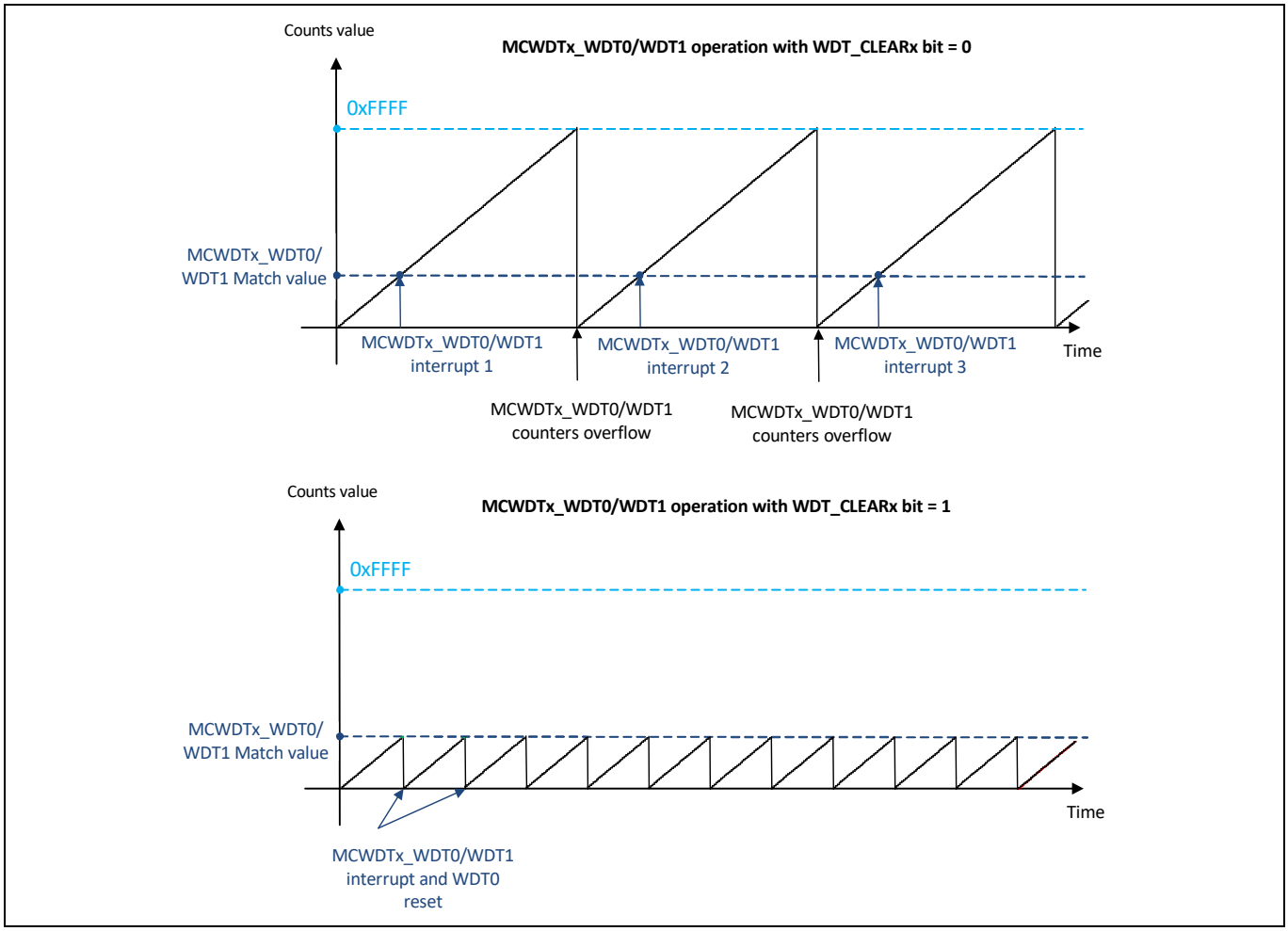


Figure 22-4. MCWDTx_WDT0/WDT1 operation

Table 22-2. MCWDTx_WDT0 and MCWDTx_WDT1 configuration options

Register [Bit_Pos]	Bit_Name	Description
MCWDTx_CONFIG[1:0] MCWDTx_CONFIG[9:8]	WDT_MODE0 WDT_MODE1	WDT action on a match event (WDT_CTRx == WDT_MATCHx) 0: Do nothing 1: Assert interrupt (WDT_INTx) 2: Assert device reset 3: Assert interrupt on match and a device reset on the third unhandled interrupt
MCWDTx_CONFIG[2] MCWDTx_CONFIG[10]	WDT_CLEAR0 WDT_CLEAR1	Clear the MCWDTx_WDT0/WDT1 counter on match. In other words, (WDT_MATCHx + 1) acts similar to a period for the MCWDTx_WDT0/WDT1 counter. 0: Free-running counter 1: Clear WDT_CTRx bits on match
MCWDTx_CNTLOW[15:0] MCWDTx_CNTLOW[16:31]	WDT_CTR0 WDT_CTR1	Current counter values. Bits[15:0] contain the current value of the MCWDTx_WDT0 counter and bits[31:16] contain the current value of MCWDTx_WDT1 counter.
MCWDTx_MATCH[15:0] MCWDTx_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.

22.4.1.2 MCWDTx_WDT2 counter operation

The MCWDTx_WDT2 is a 32-bit free-running counter, which can be configured to generate an interrupt. The MCWDTx_CNTHIGH register holds the current value of the MCWDTx_WDT2 counter. MCWDTx_WDT2 does not support a match feature. However, it can be configured to generate an interrupt when one of the counter bits toggle. The WDT_BITS2 bits [28:24] of the MCWDTx_CONFIG register selects the bit on which the MCWDTx_WDT2 interrupt is asserted. WDT_MODE2 bit [16] of the MCWDTx_CONFIG register decides whether to assert an interrupt on bit toggle or not. **Figure 22-5** shows the MCWDTx_WDT2 counter operation.

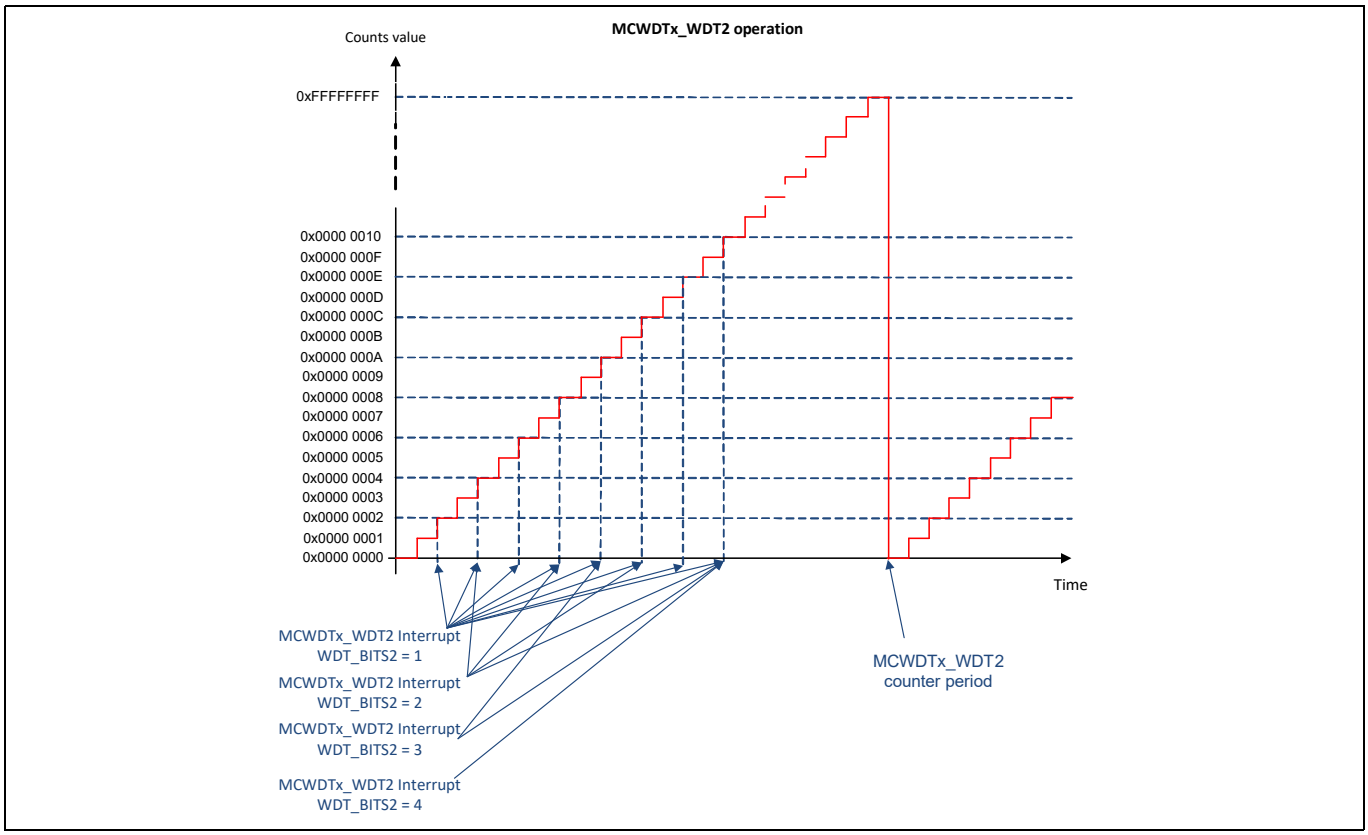


Figure 22-5. MCWDTx_WDT2 operation

Table 22-3. MCWDTx_WDT2 configuration options

Register [Bit_Pos]	Bit_Name	Description
MCWDTx_CONFIG[16]	WDT_MODE2	MCWDTx_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 MCWDTx_CNTHIGH register.
MCWDTx_CONFIG[28:24]	WDT_BITS2	Bit to monitor for MCWDTx_WDT2 interrupt assertion 0: Asserts when bit [0] of MCWDTx_CNTHIGH register toggles (interrupt every tick) …… 31: Asserts when bit [31] of MCWDTx_CNTHIGH register toggles (interrupt every 2^{31} ticks)
MCWDTx_CNTHIGH[31:0]	WDT_CTR2	Current counter value of MCWDTx_WDT2

22.4.2 Enabling and disabling WDT

The MCWDT counters are enabled by setting the WDT_ENABLEx bit in the MCWDTx_CTL register and are disabled by clearing it. Enabling or disabling a MCWDT requires 1.5 LFCLK cycles to come into effect. Therefore, the WDT_ENABLEx bit value must not be changed more than once in that period and the WDT_ENABLEDx bit of the MCWDTx_CTL register can be used to monitor enabled/disabled state of the counter.

The WDT_RESETx bit of the MCWDTx_CTL register clears the corresponding MCWDTx counter when set in firmware. The hardware clears the bit after the counter resets. This option is useful when the MCWDTx_WDT0 or MCWDTx_WDT1 is configured to generate a device reset on a match event. In such cases, the device resets when the counter reaches the match value. Thus, setting the WDT_RESET0 or WDT_RESET1 bit resets MCWDTx_WDT0 or MCWDTx_WDT1 respectively, preventing device reset.

After the MCWDT is enabled, do not write to the MCWDT configuration (MCWDTx_CONFIG) and control (MCWDTx_CTL) registers. Accidental corruption of these registers can be prevented by setting the WDT_LOCK bits [31:30] of the MCWDTx_CTL register. If the application requires updating the match value (WDT_MATCH) when the MCWDT is running, the WDT_LOCK bits must be cleared. The WDT_LOCK bits require two different writes to clear both the bits. Writing a '1' to the bits clears bit 0. Writing a '2' clears bit 1. Writing a '3' sets both the bits and writing '0' does not have any effect. Note that the WDT_LOCK bits protects only MCWDTx_CTL (except the WDT_LOCK bits), MCWDTx_CONFIG, and MCWDTx_MATCH registers. The LFCLK select registers are protected by the free-running WDT lock bits.

Table 22-4. Watchdog configuration options

Register [Bit_Pos]	Bit_Name	Description
MCWDTx_CTL[0] MCWDTx_CTL[8] MCWDTx_CTL[16]	WDT_ENABLE0 WDT_ENABLE1 WDT_ENABLE2	Enable MCWDT counter x 0: Counter is disabled 1: Counter is enabled
MCWDTx_CTL[1] MCWDTx_CTL[9] MCWDTx_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 receive an acknowledgment of the change
MCWDTx_CTL[3] MCWDTx_CTL[11] MCWDTx_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
MCWDTx_CTL[31:30]	WDT_LOCK	Locks or unlocks write access to the MCWDTx_CTL (except the WDT_LOCK bits), MCWDTx_CONFIG, and MCWDTx_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)

Note: When the watchdog counters are configured to generate an interrupt every LFCLK cycle, make sure you read the MCWDTx_INTR register after clearing the watchdog interrupt (setting the WDT_INTx bit in the MCWDTx_INTR register). Failure to do this may result in missing the next interrupt. Hence, the interrupt period becomes LFCLK/2.

22.5 Watchdog Cascade options

The cascade configuration shown in **Figure 22-3** provides an option to increase the MCWDT counter resolution. The WDT_CASCADE0_1 bit [3] of the MCWDTx_CONFIG register cascades MCWDTx_WDT0 and MCWDTx_WDT1 and the WDT_CASCADE1_2 bit [11] of the MCWDTx_CONFIG register cascades MCWDTx_WDT1 and MCWDTx_WDT2. Note that cascading two 16-bit counters does not provide a 32-bit counter; instead, you get a 16-bit period counter with a 16-bit prescaler. For example, when cascading MCWDTx_WDT0 and MCWDTx_WDT1, MCWDTx_WDT0 acts as a prescaler for MCWDTx_WDT1 and the prescaler value is defined by the WDT_MATCH0 bits [15:0] in the MCWDTx_MATCH register. The MCWDTx_WDT1 has a period defined by WDT_MATCH1 bits [31:16] in the MCWDTx_MATCH register. The same logic applies to MCWDTx_WDT1 and MCWDTx_WDT2 cascading.

Table 22-5. Watchdog Cascade options

Register [Bit_Pos]	Bit_Name	Description
MCWDTx_CONFIG[3]	WDT_CASCADE0_1	Cascade MCWDTx_WDT0 and MCWDTx_WDT1 0: MCWDTx_WDT0 and MCWDTx_WDT1 are independent counters 1: MCWDTx_WDT1 increments two cycles after WDT_CTR0 == WDT_MATCH0
MCWDTx_CONFIG[11]	WDT_CASCADE1_2	Cascade MCWDTx_WDT1 and MCWDTx_WDT2 0: MCWDTx_WDT1 and MCWDTx_WDT2 are independent counters 1: MCWDTx_WDT2 increments two cycles after WDT_CTR1 == WDT_MATCH1

When using cascade (WDT_CASCADE0_1 or WDT_CASCADE1_2 set), resetting the counters when the prescaler or lower counter is at its match value with the counter configured to clear on match, results in the upper counter incrementing to 1 instead of remaining at 0. This behavior can be corrected by issuing a second reset to the upper counter after approximately 100 µs from the first reset. Note that the second reset is required only when the first reset is issued while the prescaler counter value is at its match value. **Figure 22-6** illustrates the behavior when MCWDTx_WDT0 and MCWDTx_WDT1 are cascaded along with the second reset timing.

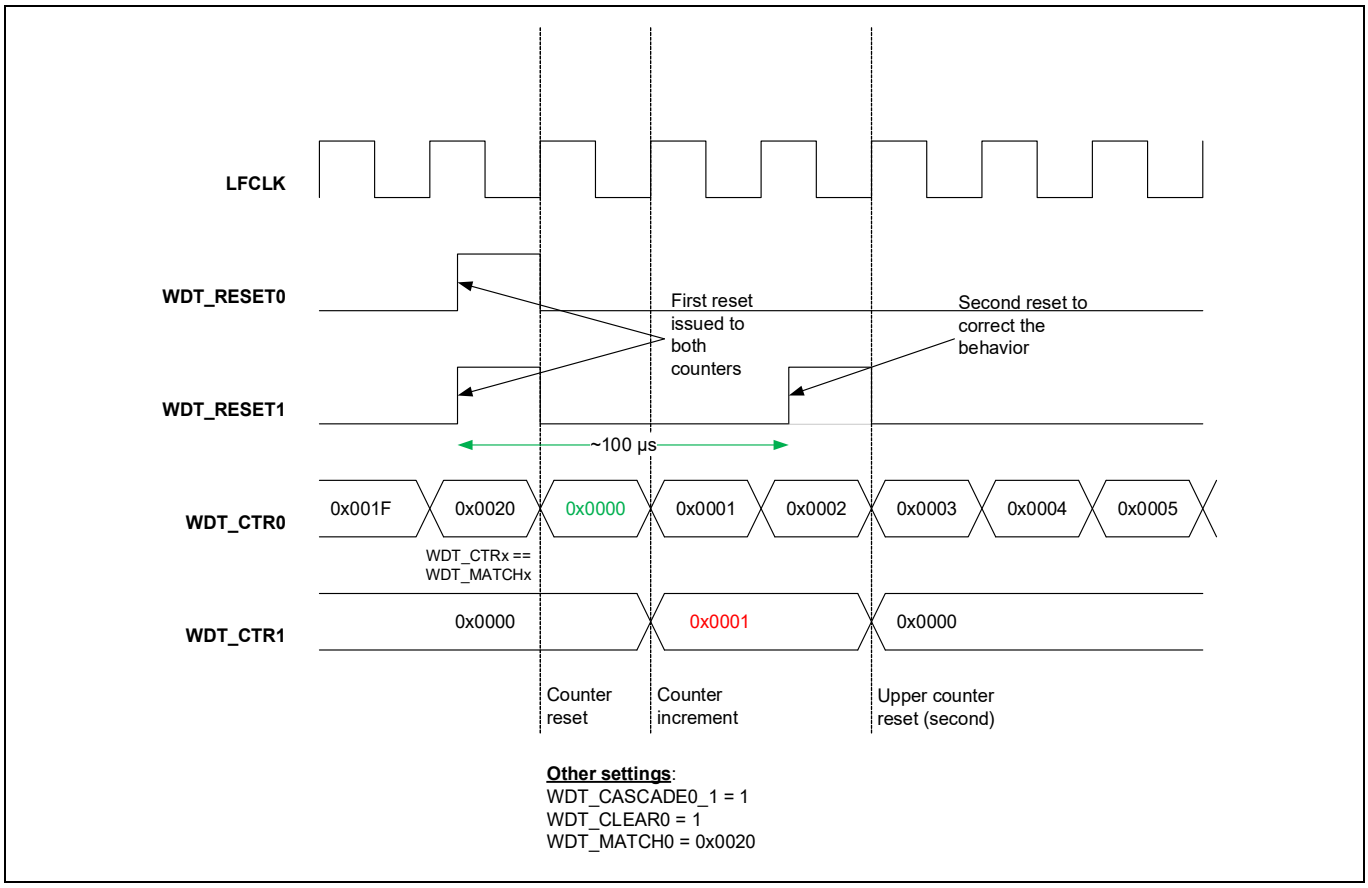


Figure 22-6. MCWDT Reset Behavior in Cascaded mode

In addition, the counters exhibit non-monotonicity in the following cascaded conditions:

- If WDT_CASCADE0_1 is set, then WDT_CTR1 does not increment the cycle after WDT_CTR0 = WDT_MATCH0.
- If WDT_CASCADE1_2 is set, then WDT_CTR2 does not increment the cycle after WDT_CTR1 = WDT_MATCH1.
- If both WDT_CASCADE0_1 and WDT_CASCADE1_2 are set, then WDT_CTR2 does not increment the cycle after WDT_CTR1 = WDT_MATCH1 and WDT_CTR1 does not increment the cycle after WDT_CTR0 = WDT_MATCH0.

When cascading is enabled, always read the WDT_CTR1 or WDT_CTR2 counter value only when the prescaler counter (WDT_CTR0 or WDT_CTR1) value is not ‘0’. This makes sure the upper counter is incremented after a match event in the prescaler counter.

22.5.1 MCDWT Reset

MCWDTx_WDT0 and MCWDTx_WDT1 can be configured to generate a device reset similar to the free-running WDT reset. Note that when the debug probe is connected, the device reset is blocked but an interrupt is generated if configured. Follow the steps to use the MCWDTx_WDT0 or MCWDTx_WDT1 counter of a MCWDTx block to generate a system reset:

1. Configure the MCWDT to generate a reset using the WDT_MODEx bits in MCWDTx_CONFIG. Configure the WDT_MODE0 or WDT_MODE1 bits in MCWDTx_CONFIG to '2' (reset on match) or '3' (interrupt on match and reset on the third unhandled interrupt).
2. Optionally, set the WDT_CLEAR0 or WDT_CLEAR1 bit in the MCWDTx_CONFIG register for MCWDTx_WDT0 or MCWDTx_WDT1 to reset the corresponding watchdog counter to '0' on a match event. Otherwise, the counters are free running. See **Table 22-2** for details.
3. Calculate the watchdog reset period such that firmware is able to reset the watchdog at least once during the period, even along the longest firmware delay path. For WDT_MODEx == 2, match value is same as the watchdog period. For WDT_MODEx == 3, match value is one-third of the watchdog period. Write the calculated match value to the WDT_MATCH register for MCWDTx_WDT0 or MCWDTx_WDT1. Optionally, enable cascading to increase the interval.

Note: The legal value for the WDT_MATCH field is 1 to 65535.

4. For WDT_MODEx == 2, set the WDT_RESETx bit in the MCWDTx_CONFIG register to reset the WDTx counter to 0. For WDT_MODEx == 3, set the WDT_INTx bit in MCWDTx_INTR to clear any pending interrupts.
5. Enable WDTx by setting the WDT_ENABLEx bit in the MCWDTx_CTL register. Wait until the WDT_ENABLEDx bit is set.
6. Lock the MCWDTx configuration by setting the WDT_LOCK bits of the MCWDTx_CTL register.
7. In the firmware, feed (reset) the watchdog as explained in step 4.

Do not reset watchdog in the WDT ISR. It is also not recommended to use the same watchdog counter to generate a system reset and interrupt. For example, if MCWDTx_WDT0 is used to generate system reset against crashes, then MCWDTx_WDT1 or MCWDTx_WDT2 should be used for periodic interrupt generation.

22.5.2 MCWDT Interrupt

When configured to generate an interrupt, the WDT_INTx bits of the MCWDTx_INTR register provide the status of any pending watchdog interrupts. The firmware must clear the interrupt by setting the WDT_INTx. The WDT_INTx bits of the MCWDTx_INTR_MASK register mask the corresponding WDTx interrupt of the MCWDTx block to the CPU.

Follow these steps to use WDT as a periodic interrupt generator:

1. Write the desired match value to the WDT_MATCH register for MCWDTx_WDT0/WDT1 or the WDT_BITS2 value to the MCWDTx_CONFIG register for MCWDTx_WDT2.

Note: The legal value for the WDT_MATCH field is 1 to 65535.

2. Configure the WDTx to generate an interrupt using the WDT_MODEx bits in MCWDTx_CONFIG. Configure the WDT_MODE0 or WDT_MODE1 bits in MCWDTx_CONFIG for MCWDTx_WDT0 or MCWDTx_WDT1 to '1' (interrupt on match) or '3' (interrupt on match and reset on third unhandled interrupt). For MCWDTx_WDT2, set the WDT_MODE2 bit in the MCWDTx_CONFIG register.
3. Set the WDT_INT bit in MCWDTx_INTR to clear any pending interrupt.
4. Set the WDT_CLEAR0 or WDT_CLEAR1 bit in the MCWDTx_CONFIG register for MCWDTx_WDT0 or MCWDTx_WDT1 to reset the corresponding watchdog counter to '0' on a match event.
5. Mask the WDTx interrupt to the CPU by setting the WDT_INTx bit in the MCWDTx_INTR_MASK register

6. Enable WDTx by setting the WDT_ENABLEx bit in the MCWDTx_CTL register. Wait until the WDT_ENABLEDx bit is set.
7. Enable MCWDTx interrupt to the CPU by configuring the appropriate ISER register. Refer to the **"Interrupts"** on page 89.
8. In the ISR, clear the WDTx interrupt by setting the WDT_INTx bit in the MCWDTx_INTR register.

Note that interrupts from all three WDTx counters of the MCWDT block are mapped as a single interrupt to the CPU. In the interrupt service routine, the WDT_INTx bits of the MCWDTx_INTR register can be read to identify the interrupt source. However, each MCWDT block has its own interrupt to the CPU. For details on interrupts, see the **"Interrupts"** on page 89.

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 works similar to the free-running WDT.

22.6 Reset cause detection

The RESET_WDT bit [0] in the RES_CAUSE register indicates the reset generated by the free-running WDT. The RESET_MCWDTx bit in the RES_CAUSE register indicates the reset generated by the MCWDTx block. These bits remain set until cleared or until a power-on reset (POR), brownout reset (BOD), or external reset (XRES) occurs. All other resets leave this bit unaltered. For more details, see the **"Reset system"** on page 313.

22.7 Register list

Table 22-6. WDT registers

Name	Description
WDT_CTL	Watchdog Counter Control Register
WDT_CNT	Watchdog Counter Count Register
WDT_MATCH	Watchdog Counter Match Register
MCWDTx_MCWDT_CNTLOW	Multi-counter WDT Sub-counters 0/1
MCWDTx_MCWDT_CNTHIGH	Multi-counter WDT Sub-counter 2
MCWDTx_MCWDT_MATCH	Multi-counter WDT Counter Match Register for counters 0 and 1
MCWDTx_MCWDT_CONFIG	Multi-counter WDT Counter Configuration, including bit toggle interrupt generation for counter 2
MCWDTx_MCWDT_CTL	Multi-counter WDT Counter Control
MCWDTx_MCWDT_INTR	Multi-counter WDT Counter Interrupt Register
MCWDTx_MCWDT_INTR_SET	Multi-counter WDT Counter Interrupt Set Register
MCWDTx_MCWDT_INTR_MASK	Multi-counter WDT Counter Interrupt Mask Register
MCWDTx_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

23 I/O system

This chapter explains the EZ-USB™ FX10 MCU I/O system, its features, architecture, operating modes, and interrupts. The I/O system provides the interface between the CPU core and peripheral components to the outside world. The flexibility of EZ-USB™ FX10 MCUs and the capability of its I/O to route most signals to most pins greatly simplifies circuit design and board layout. The GPIO pins in the EZ-USB™ FX10 MCU family are grouped into ports; a port can have a maximum of eight GPIO pins.

23.1 Features

The EZ-USB™ FX10 MCU GPIOs have these features:

- Digital input and output capabilities
- Eight drive strength modes
- Separate port read and write registers
- Overvoltage tolerant (OVT-GPIO) pins
- Separate I/O supplies and voltages for up to six groups of I/O
- Edge-triggered interrupts on rising edge, falling edge, or on both edges, on all GPIO
- Slew rate control
- Selectable CMOS and low-voltage LVTTL input buffer mode
- Smart I/O provides the ability to perform Boolean functions in the I/O signal path

23.2 Architecture

The EZ-USB™ FX10 MCU is equipped with analog and digital peripherals. **Figure 23-1** shows an overview of the routing between the peripherals and pins.

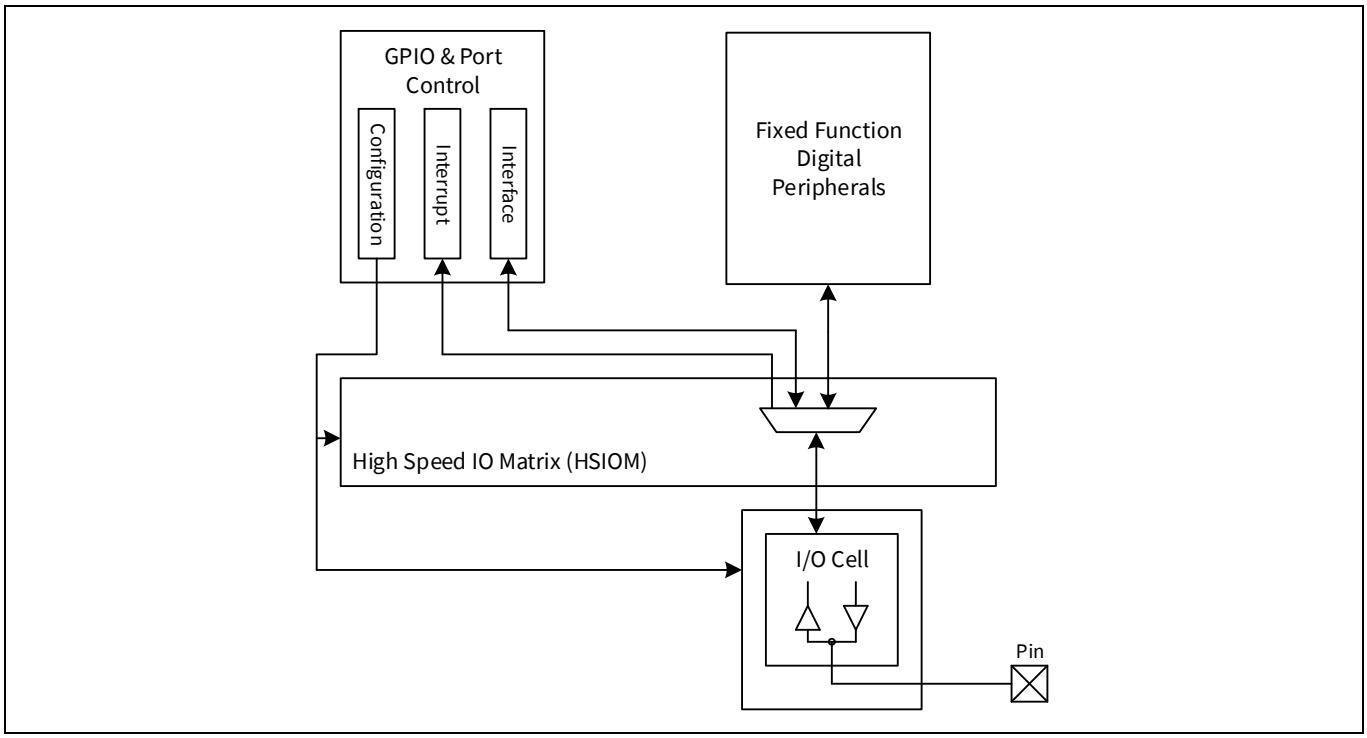


Figure 23-1. GPIO interface overview

I/O system

GPIO pins are connected to I/O cells. These cells are equipped with an input buffer for the digital input, providing high input impedance and a driver for the digital output signals. The digital peripherals connect to the I/O cells via the high-speed I/O matrix (HSIOM). The HSIOM for each pin contains multiplexers to connect between the selected peripheral and the pin.

23.2.1 I/O cell architecture

Figure 23-2 shows the I/O cell architecture present in every GPIO cell. It comprises an input buffer and an output driver that connect to the HSIOM multiplexers for digital input and output signals. Analog peripherals connect directly to the pin for point to point connections or use the AMUXBUS.

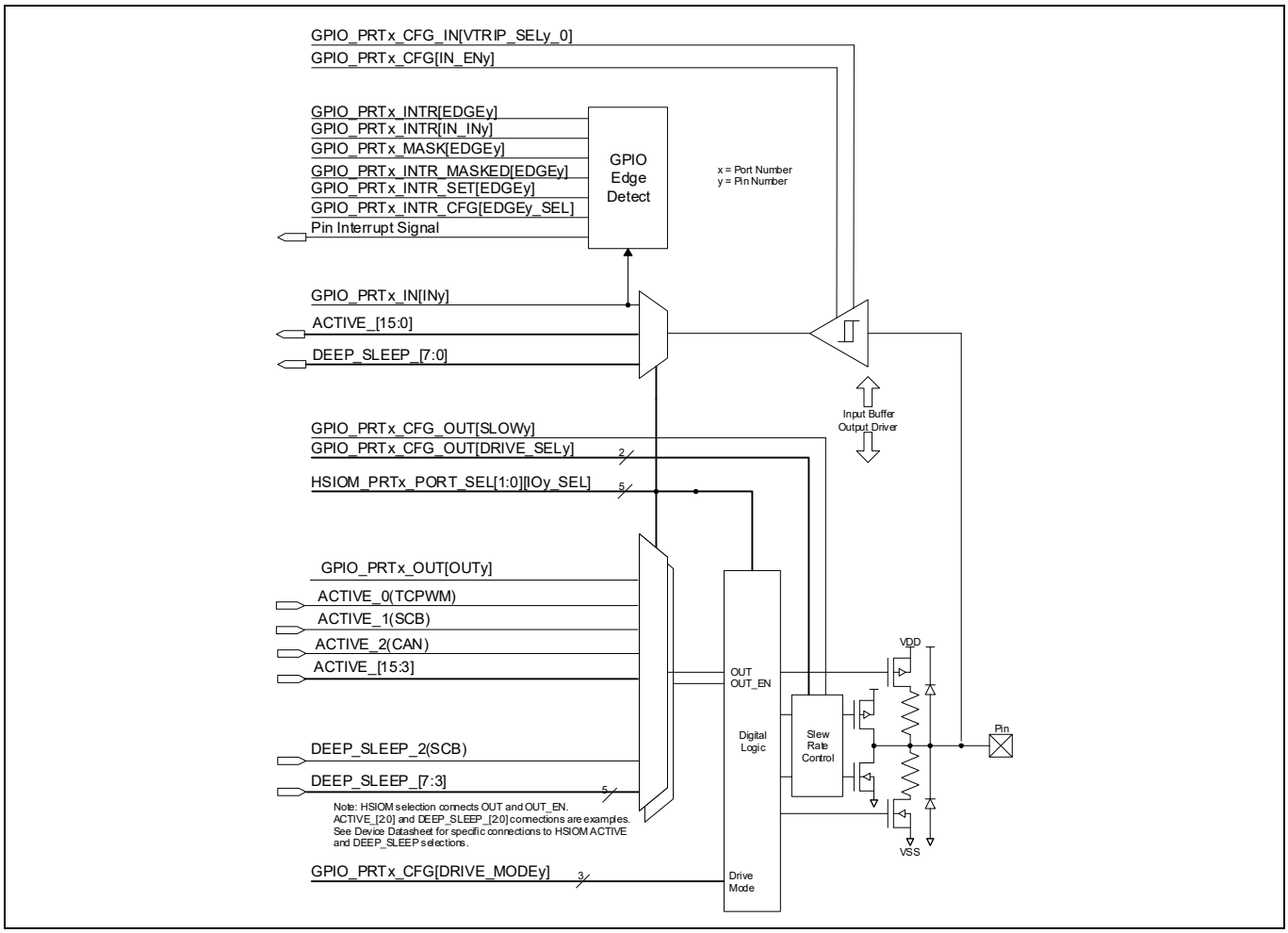


Figure 23-2. GPIO and GPIO_OVT cell architecture

23.2.2 Digital input buffer

The digital input buffer provides a high-impedance buffer for the external digital input. The buffer is enabled or disabled by the IN_EN[7:0] bit of the Port Configuration Register (GPIO_PRTx_CFG, where x is the port 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_PORT_SELx) selects the pin connection. See the device datasheet for the specific connections available for each pin.

If a pin is connected only to an analog signal, the input buffer should be disabled to avoid crowbar currents.

Each pin's input buffer trip point and hysteresis are configurable for the following modes:

- CMOS + I^2C
- TTL

These buffer modes are selected by the VTRIP_SEL[7:0]_0 bit of the Port Input Buffer Configuration register. (GPIO_PRTx_CFG_IN).

23.2.3 Digital output driver

Pins are driven by the digital output driver. It consists of circuitry to implement different drive modes and slew rate control for the digital output signals. The HSIOM selects the control source for the output driver. The two primary types of control sources are CPU registers and fixed-function digital peripherals. A particular HSIOM connection is selected by writing to the HSIOM port select register (HSIOM_PORT_SELx).

I/O ports are powered by different sources. The specific allocation of ports to supply sources can be found in the Pinout section of the device datasheet.

Each GPIO pin has ESD diodes to clamp the pin voltage to the I/O supply source. Ensure that the voltage at the pin does not exceed the I/O supply voltage $V_{DDIO}/V_{DDD}/V_{DDA}$ or drop below $V_{SSIO}/V_{SSD}/V_{SSA}$. For the absolute maximum and minimum GPIO voltage, see the device datasheet.

23.2.3.1 Drive modes

Each I/O is individually configurable to one of eight drive modes by the DRIVE_MODE[7:0] field of the Port Configuration register, GPIO_PRTx_CFG. **Table 23-1** lists the drive modes. Drive mode '1' is reserved and should not be used in most designs. CPU register and AMUXBUS connections support seven discrete drive modes to maximize design flexibility. Fixed-function digital peripherals, such as SCB and TCPWM blocks, support modified functionality for the same seven drive modes compatible with fixed peripheral signaling. **Figure 23-3** shows simplified output driver diagrams of the pin view for CPU register control on each of the eight drive modes. **Figure 23-4** is a simplified output driver diagram that shows the pin view for fixed-function-based peripherals for each of the eight drive modes.

Table 23-1. Drive mode settings

Drive mode	Value	CPU register, AMUXBUS				Fixed-function digital peripheral			
		OUT_EN = 1		OUT_EN = 0		OUT_EN = 1		OUT_EN = 0	
		OUT = 1	OUT = 0	OUT = 1	OUT = 0	OUT = 1	OUT = 0	OUT = 1	OUT = 0
High Impedance	0	HI-Z	HI-Z	HI-Z	HI-Z	HI-Z	HI-Z	HI-Z	HI-Z
Reserved	1	Strong 1	Strong 0	HI-Z	HI-Z	Strong 1	Strong 0	Weak 1 & 0	Weak 1 & 0
Resistive Pull Up	2	Weak 1	Strong 0	HI-Z	HI-Z	Strong 1	Strong 0	Weak 1	Weak 1
Resistive Pull Down	3	Strong 1	Weak 0	HI-Z	HI-Z	Strong 1	Strong 0	Weak 0	Weak 0
Open Drain, Drives Low	4	HI-Z	Strong 0	HI-Z	HI-Z	Strong 1	Strong 0	HI-Z	HI-Z
Open Drain, Drives High	5	Strong 1	HI-Z	HI-Z	HI-Z	Strong 1	Strong 0	HI-Z	HI-Z
Strong	6	Strong 1	Strong 0	HI-Z	HI-Z	Strong 1	Strong 0	HI-Z	HI-Z
Resistive Pull Up and Down	7	Weak 1	Weak 0	HI-Z	HI-Z	Strong 1	Strong 0	Weak 1	Weak 0

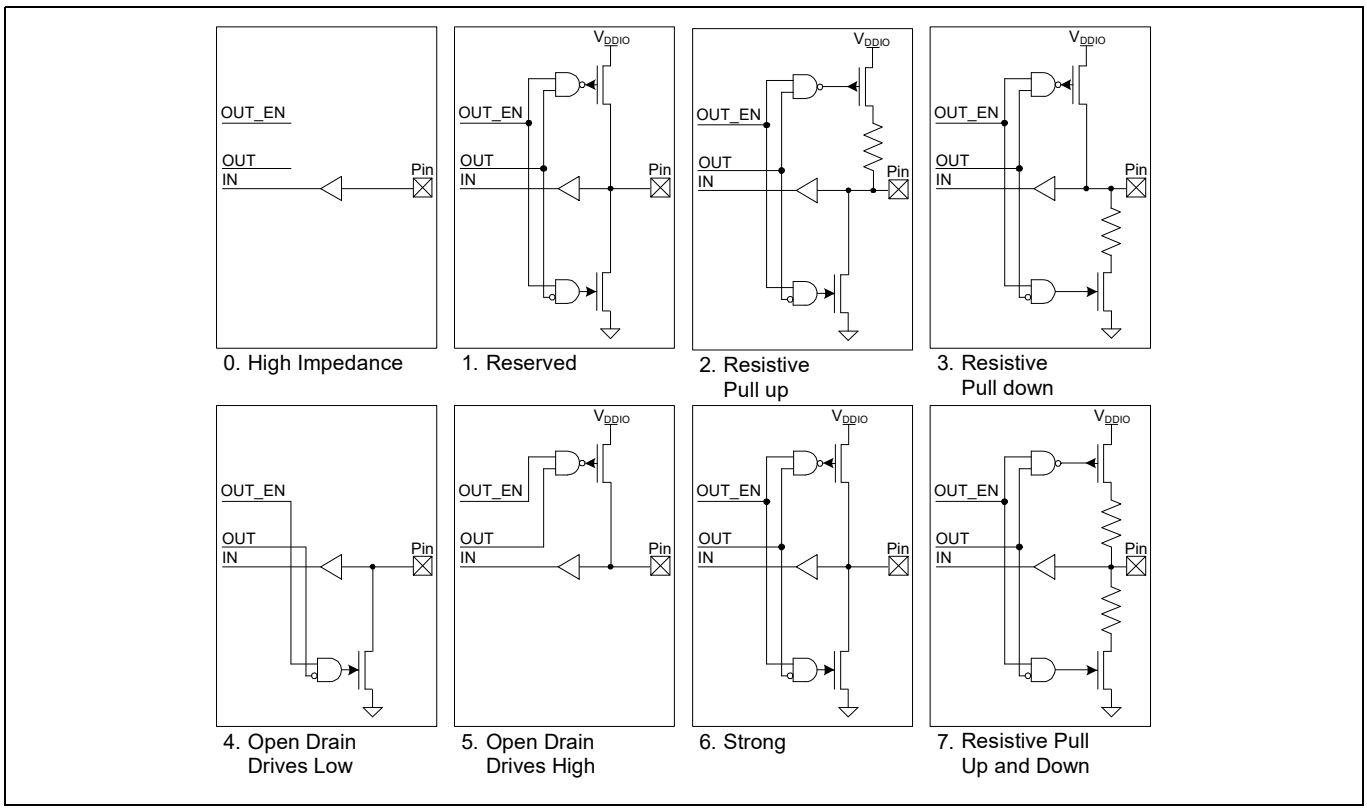


Figure 23-3. CPU I/O drive mode block diagram

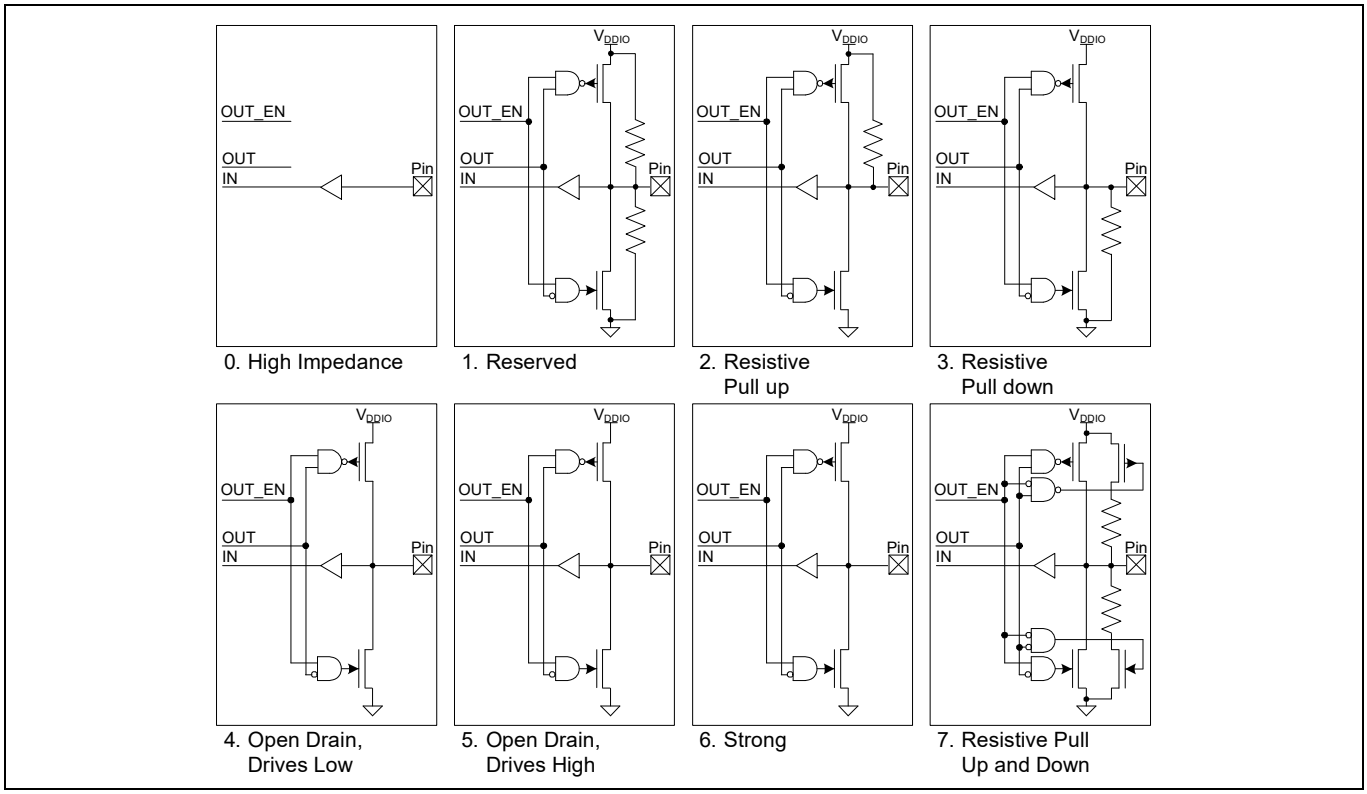


Figure 23-4. Peripheral I/O drive mode block diagrams

- High-impedance

This is the standard high-impedance (HI-Z) state recommended for analog and digital inputs. For digital signals, the input buffer is enabled; for analog signals, the input buffer is typically disabled to reduce crowbar current and leakage in low-power designs. To achieve the lowest device current, unused GPIOs must be configured to the high-impedance drive mode with input buffer disabled. High-impedance drive mode with input buffer disabled is also the default pin reset state.

- Resistive pull-up or resistive pull-down

Resistive modes provide a series resistance in one of the data states and strong drive in the other. Pins can be used for either digital input or digital output in these modes. If resistive pull-up is required, a '1' must be written to that pin's Data Register bit. If resistive pull-down is required, a '0' must be written to that pin's Data Register. Interfacing mechanical switches is a common application of these drive modes. The resistive modes are also used to interface EZ-USB™ FX10 with open drain drive lines. Resistive pull-up is used when the input is open drain low and resistive pull-down is used when the input is open drain high.

- Open drain drives high and open drain drives low

Open drain modes provide high impedance in one of the data states and strong drive in the other. Pins are useful as digital inputs or outputs in these modes. Therefore, these modes are widely used in bidirectional digital communication. Open drain drive high mode is used when the signal is externally pulled down and open drain drive low is used when the signal is externally pulled high. A common application for the open drain drives low mode is driving I^2C bus signal lines.

- Strong drive

The strong drive mode is the standard digital output mode for pins; it provides a strong CMOS output drive in both high and low states. Strong drive mode pins should not be used as inputs under normal circumstances. This mode is often used for digital output signals or to drive external devices.

- Resistive pull-up and resistive pull-down

In the resistive pull-up and pull-down mode, the GPIO will have a series resistance in both logic 1 and logic 0 output states. The high data state is pulled up while the low data state is pulled down. This mode is useful when the pin is driven by other signals that may cause shorts.

23.2.3.2 Slew rate control

GPIO pins have fast and slow output slew rate options for the strong drivers configured using the SLOW bit of the port output configuration register (GPIO_PRTx_CFG_OUT). By default, this bit is cleared and the port works in fast slew mode. This bit can be set if a slow slew rate is required. Slower slew rate results in reduced EMI and crosstalk and are recommended for low-frequency signals or signals without strict timing constraints.

When configured for fast slew rate, the drive strength can be set to one of four levels using the DRIVE_SEL field of the port output configuration register (GPIO_PRTx_CFG_OUT). The drive strength field determines the active portion of the output drivers used and can affect the slew rate of output signals. Drive strength options are full drive strength (default), one-half strength, one-quarter strength, and one-eighth strength. Drive strength must be set to full drive strength when the slow slew rate bit (SLOW) is set.

23.2.3.3 GPIO-OVT pins

Select device pins are overvoltage tolerant (OVT) and are useful for interfacing to busses or other signals that may exceed the pin's V_{DDIO} supply, or where the whole device supply or pin V_{DDIO} may not be always present. They are identical to regular GPIOs with the additional feature of being overvoltage tolerant. GPIO-OVT pins have hardware to compare V_{DDIO} to the pin voltage. If the pin voltage exceeds V_{DDIO}, the output driver is disabled and the pin driver is tristated. This results in negligible current sink at the pin.

Note that in overvoltage conditions, the input buffer data will not be valid if the external source's specification of V_{OH} and V_{OL} do not match the trip points of the input buffer defined by the current V_{DDIO} voltage.

23.3 High-speed I/O matrix

The high-speed I/O matrix (HSIOM) is a set of high-speed multiplexers that route internal CPU and peripheral signals to and from GPIOs. HSIOM allows GPIOs to be shared with multiple functions and multiplexes the pin connection to a user-selected peripheral. The HSIOM_PRTx_PORT_SEL[1:0] registers allow a single selection from up to 32 different connections to each pin as listed in **Table 23-2**.

Table 23-2. HSIOM connections

SELy_SEL	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	AMUXA	OUT Register	1	IN Register	1	0	Analog mux bus A connected to pin
5	AMUXB	OUT Register	1	IN Register	0	1	Analog mux bus B connected to pin
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
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

Table 23-2. HSIOM connections (continued)

SELy_SEL	Name	Digital driver signal source		Digital input signal destination	Analog switches		Description
		OUT	OUT_EN		AMUXA	AMUXB	
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
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
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. See the "Pinouts" section of the device datasheet for more details on the features supported by each pin.

23.4 I/O state on power up

During power up, all the GPIOs are in high-impedance analog state and the input buffers are disabled. During runtime, GPIOs can be configured by writing to the associated registers. Note that the pins supporting debug access port (DAP) connections (SWD lines) are always enabled as SWD lines during power up. The DAP connection does not provide pull-up or pull-down resistors; therefore, if left floating some crowbar current is possible. The DAP connection can be disabled or reconfigured for general-purpose use through the HSIOM only after the device boots and starts executing code.

23.5 Behavior in Low-power modes

Table 23-3 shows the status of GPIOs in low-power modes.

Table 23-3. 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 can operate in CPU Sleep mode. • Inputs buffers are active; thus an interrupt on any I/O can be used to wake the CPU.
System Deep Sleep	• GPIO, GPIO-OVT, and SIO pins, connected to System Deep Sleep domain peripherals, are functional. All other pins maintain the last output driver state and configuration. • Pin interrupts are functional on all I/Os and can be used to wake the device.

23.6 Interrupt

All port pins have the capability to generate interrupts. Figure 23-5 shows the routing of pin signals to generate interrupts.

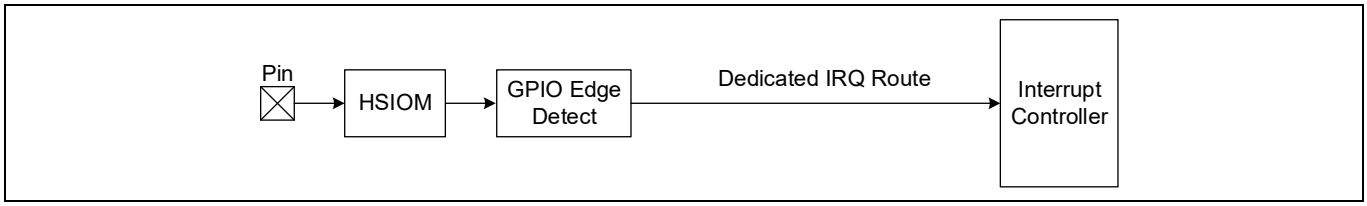


Figure 23-5. Interrupt signal routing

Pin signal through the “GPIO Edge Detect” block with direct connection to the CPU interrupt controller.

Figure 23-6 shows the GPIO Edge Detect block architecture.

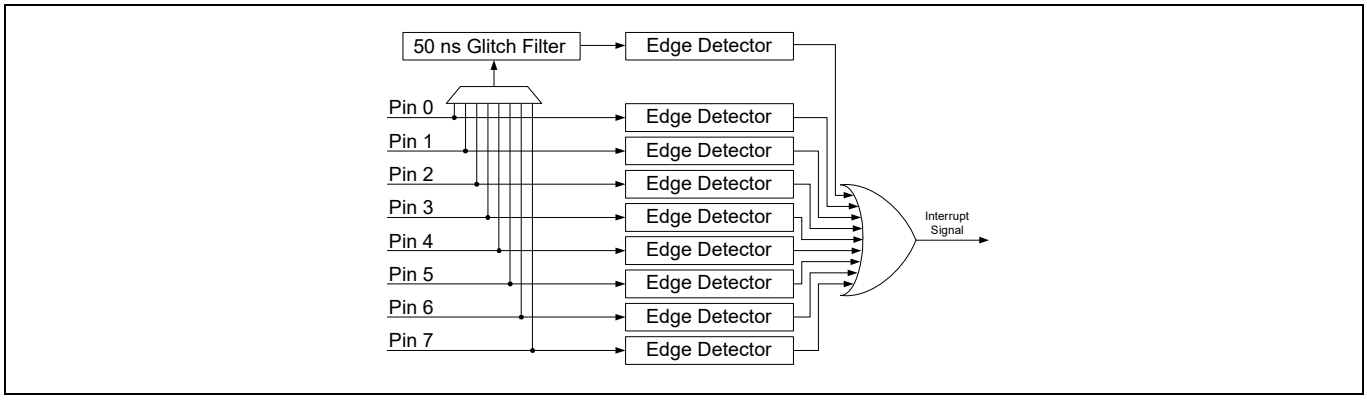


Figure 23-6. GPIO Edge Detect Block architecture

I/O system

An edge detector is present at each pin. It is capable of detecting rising edge, falling edge, and both edges without any reconfiguration. The edge detector is configured by writing into the EDGEv_SEL bits of the Port Interrupt Configuration register, GPIO_PRTx_INTR_CFG, as shown in **Table 23-4**.

Table 23-4. Edge detector configuration

EDGE_SEL	Configuration
00	Interrupt is disabled
01	Interrupt on rising edge
10	Interrupt on falling edge
11	Interrupt on both edges

Writing '1' to the corresponding status bit clears the pin edge state. Clearing the edge state status bit is important; otherwise, an interrupt can occur repeatedly for a single trigger or respond only once for multiple triggers, which is explained later in this section. When the Port Interrupt Control Status register is read at the same time an edge is occurring on the corresponding port, it can result in the edge not being properly detected. Therefore, when using GPIO interrupts, read the status register only inside the corresponding interrupt service routine and not in any other part of the code.

Firmware and the debug interface are able to trigger a hardware interrupt from any pin by setting the corresponding bit in the GPIO_PRTx_INTR_SET register.

In addition to the pins, each port provides a glitch filter connected to its own edge detector. This filter can be driven by one of the pins of a port. The selection of the driving pin is done by writing to the FLT_SEL field of the GPIO_PRTx_INTR_CFG register as shown in **Table 23-5**.

Table 23-5. Glitch filter input selection

FLT_SEL	Selected pin
000	Pin 0 is selected
001	Pin 1 is selected
010	Pin 2 is selected
011	Pin 3 is selected
100	Pin 4 is selected
101	Pin 5 is selected
110	Pin 6 is selected
111	Pin 7 is selected

When a port pin edge occurs, you can read the Port Interrupt Status register, GPIO_PRTx_INTR, to know which pin caused the edge. This register includes both the latched information on which pin detected an edge and the current pin status. This allows the CPU to read both information in a single read operation. This register has an additional use – to clear the latched edge state.

The GPIO_PRTx_INTR_MASK register enables forwarding of the GPIO_PRTx_INTR edge detect signal to the interrupt controller when a '1' is written to a pin's corresponding bitfield. The GPIO_PRTx_INTR_MASKED register can then be read to determine the specific pin that generated the interrupt signal forwarded to the interrupt controller. The masked edge detector outputs of a port are then ORed together and routed to the interrupt controller (NVIC in the CPU subsystem). Thus, there is only one interrupt vector per port.

The masked and ORed edge detector block output is routed to the Interrupt Source Multiplexer shown in **Figure 10-3**, which gives an option of Level and Rising Edge detection. If the Level option is selected, an interrupt is triggered repeatedly as long as the Port Interrupt Status register bit is set. If the Rising Edge detect option is selected, an interrupt is triggered only once if the Port Interrupt Status register is not cleared. Thus, the interrupt status bit must be cleared if the Edge Detect block is used.

All of the port interrupt vectors are also ORed together into a single interrupt vector for use on devices with more ports than there are interrupt vectors available. To determine the port that triggered the interrupt, the GPIO_INTR_CAUSEx registers can be read. '1' present in a bit location indicates that the corresponding port has a pending interrupt. The indicated GPIO_PRTx_INTR register can then be read to determine the pin source.

23.7 Peripheral connections

23.7.1 Firmware-controlled GPIO

For standard firmware-controlled GPIO using registers, the GPIO mode must be selected in the HSIOM_PORT_SELx register.

The GPIO_PRTx_OUT register is used to read and write the output buffer state for GPIOs. A write operation to this register changes the GPIO's output driver state to the written value. A read operation reflects the output data written to this register and the resulting output driver state. It does not return the current logic level present on GPIO pins, which may be different. Using the GPIO_PRTx_OUT register, read-modify-write sequences can be safely performed on a port that has both input and output GPIOs.

In addition to the data register, three other registers – GPIO_PRTx_SET, GPIO_PRTx_CLR, and GPIO_PRTx_INV – are provided to set, clear, and invert the output data respectively on specific pins in a port without affecting other pins. This avoids the need for read-modify- write operations in most use cases. Writing '1' to these register bitfields will set, clear, or invert the respective pin; writing '0' will have no affect on the pin state.

GPIO_PRTx_IN is the port I/O pad register, which provides the actual logic level present on the GPIO pin when read. Writes to this register have no effect.

23.8 Registers

Table 23-6. I/O registers

Name	Description
GPIO_PRTx_OUT	Port output data register reads and writes the output driver data for I/O pins in the port.
GPIO_PRTx_OUT_CLR	Port output data clear register clears output data of specific I/O pins in the port.
GPIO_PRTx_OUT_SET	Port output data set register sets output data of specific I/O pins in the port.
GPIO_PRTx_OUT_INV	Port output data invert register inverts output data of specific I/O pins in the port.
GPIO_PRTx_IN	Port input state 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.
GPIO_PRTx_INTR_MASKED	Port interrupt masked status register reads the masked interrupt status forwarded to the CPU interrupt controller.
GPIO_PRTx_INTR_SET	Port interrupt set register allows firmware to set pin interrupts.
GPIO_PRTx_INTR_CFG	Port interrupt configuration register selects the edge detection type for each pin interrupt.
GPIO_PRTx_CFG	Port configuration register selects the drive mode and input buffer enable for each pin.
GPIO_PRTx_CFG_IN	Port input buffer configuration register configures the input buffer mode for each pin.
GPIO_PRTx_CFG_OUT	Port output buffer configuration register selects the output driver slew rate for each pin.
HSIOM_PORT_SELx	High-speed I/O Mux (HSIOM) port selection register selects the hardware peripheral connection to I/O pins.

Note: The 'x' in the GPIO register name denotes the port number. For example, GPIO_PTR1_OUT is the Port 1 output data register.

24 Trigger multiplexer block

Every peripheral in the EZ-USB™ FX10 MCU is interconnected using trigger signals. Trigger signals are means by which peripherals denote an occurrence of an event or a state. These triggers are used as means to affect or initiate some action in other peripherals. The trigger multiplexer block helps to route triggers from a source peripheral block to a destination.

24.1 Features

- Ability to connect any trigger signal from one peripheral to another
- Supports a software trigger, which can trigger any signal in the block
- Supports multiplexing of triggers between peripherals
- One-to-one trigger paths for dedicated triggers that are more commonly routed
- Ability to configure a trigger multiplexer with trigger manipulation features in hardware such as inversion and edge/level detection
- Ability to block triggers in debug mode

24.2 Architecture

The trigger signals in the EZ-USB™ FX10 MCU are digital signals generated by peripheral blocks to denote a state such as FIFO level, or an event such as the completion of an action. These trigger signals typically serve as initiator of other actions in other peripheral blocks. An example is a PDM peripheral block that receives data from an external PDM device. When the amount of data reaches a preset level, a trigger signal is generated which triggers a DMA channel that in turn transfers the received data from the PDM peripheral to a memory buffer. This example is shown in **Figure 24-1**.

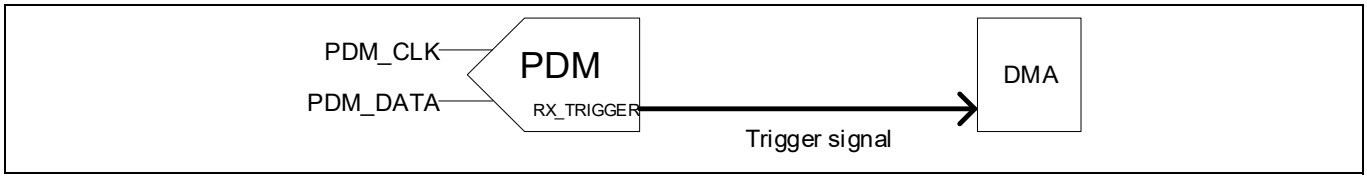


Figure 24-1. Trigger signal example

A EZ-USB™ FX10 MCU has multiple peripheral bocks; each of these blocks can be connected to other blocks through trigger signals, based on the system implementation. To support this, the EZ-USB™ FX10 MCU has hardware, which is a series of multiplexers used to route the trigger signals from potential sources to destinations. This hardware is called the trigger multiplexer block. The trigger multiplexer can connect to any trigger signal emanating out of any peripheral block in the EZ-USB™ FX10 MCU and route it to any other peripheral to initiate or affect an operation at the destination peripheral block.

24.2.1 Trigger multiplexer group

The trigger multiplexer block is implemented using several trigger multiplexers. A trigger multiplexer selects a signal from a set of trigger output signals from different peripheral blocks to route it to a specific trigger input of another peripheral block. The multiplexers are grouped into trigger groups. All the trigger multiplexers in a trigger group have similar input options and are designed to feed similar destination signals. Hence the trigger group can be considered as a block that multiplexes multiple inputs to multiple outputs. This concept is illustrated in **Figure 24-2**. Both the trigger multiplexers and the one-to-one triggers offer trigger manipulation, which includes inversion or specifies the output signal as edge or level triggered. For edge triggering, the trigger is synchronized to the consumer block's (the peripheral block that is receiving the trigger) clock and a two-cycle pulse is generated on this clock.

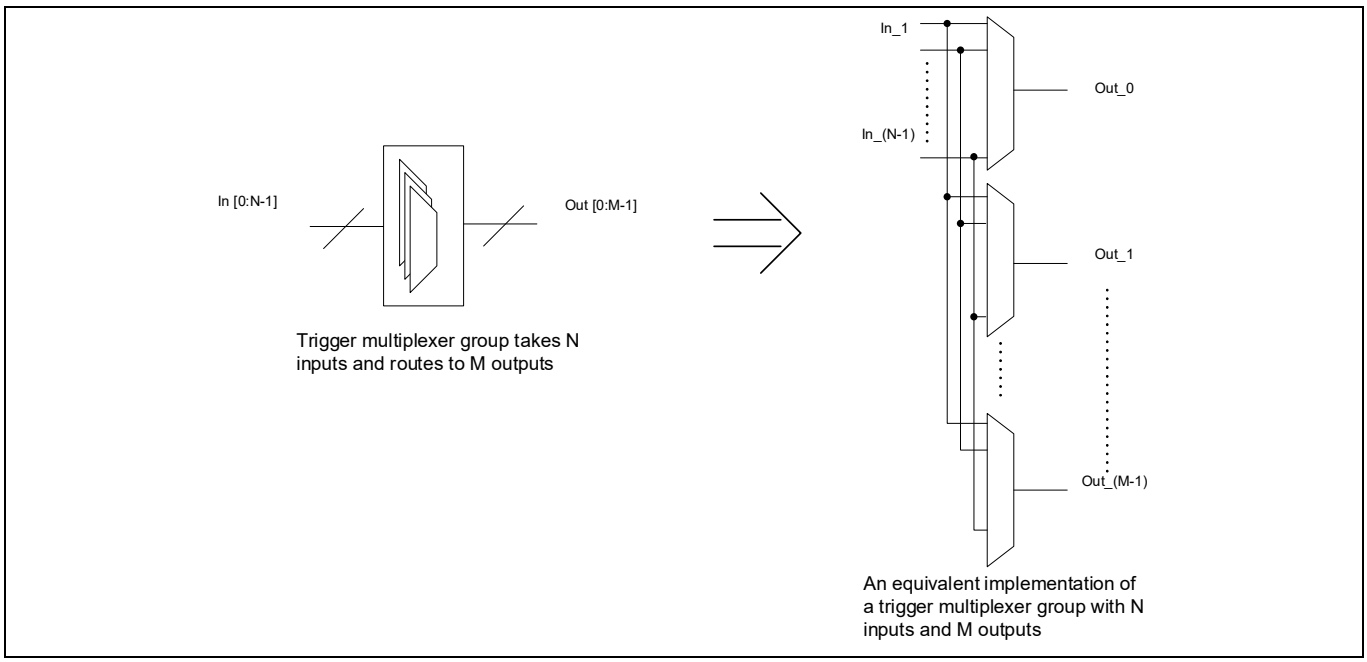


Figure 24-2. Trigger multiplexer groups

24.2.2 One-to-one trigger

In addition to trigger multiplexers, there are dedicated trigger paths called one-to-one triggers. These trigger paths/routes are between fixed source and destination peripherals and cannot be multiplexed. These paths can be enabled or disabled. A group of registers in the format PERI_TR_1TO1_GR[X]_TR_CTL[Y], with X being the trigger group and Y being the output trigger number from the one-to-one trigger, can be used to enable or disable the triggers path.

24.2.3 Trigger multiplexer block

The trigger multiplexer block has two types of trigger routings: trigger multiplexers that are capable of selecting from multiple trigger sources to a specific trigger destination and one-to-one triggers that are predetermined fixed trigger paths that can be enabled or disabled. See **Table 24-2** for more information on each trigger group and its description.

Note: The triggers output into different peripherals, which may have more routing than is shown on the trigger routing diagram. For more information on this routing, go to the trigger destination peripheral block.

Trigger multiplexer block

Figure 24-3 to **Figure 24-11** show the trigger multiplexer architecture.

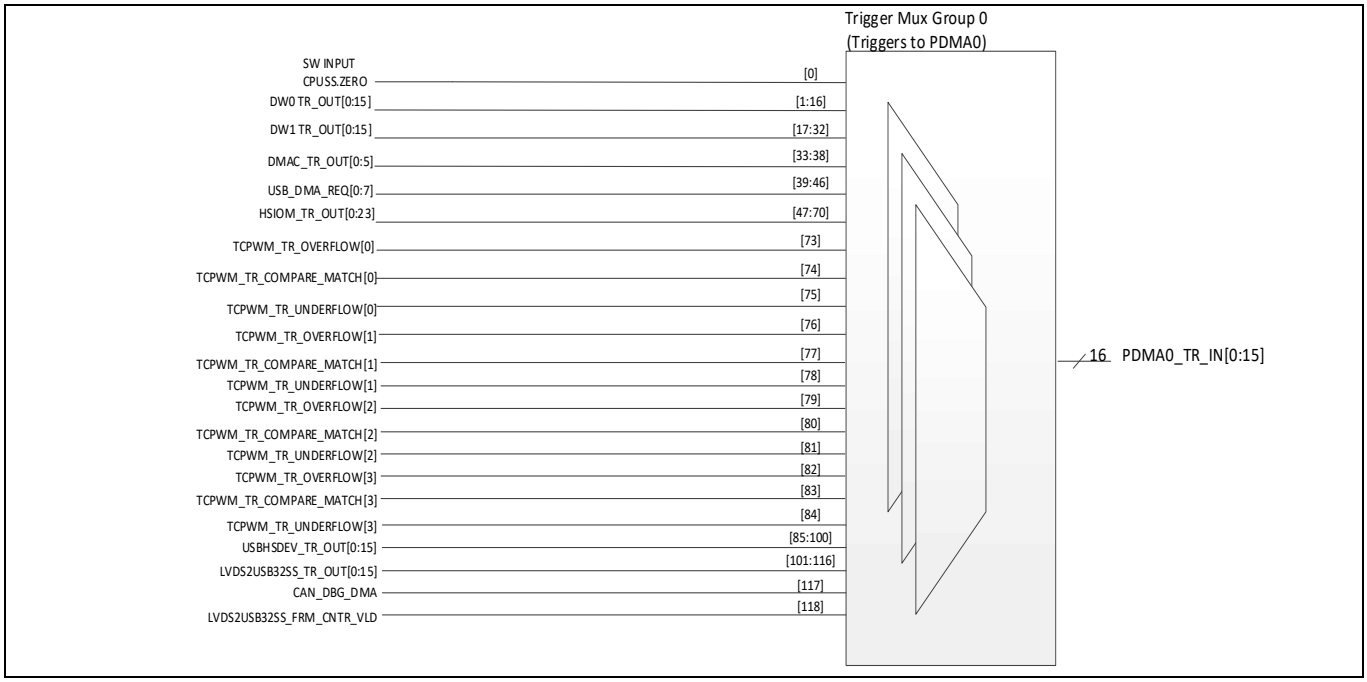


Figure 24-3. Trigger multiplexer block architecture (1 of 9)

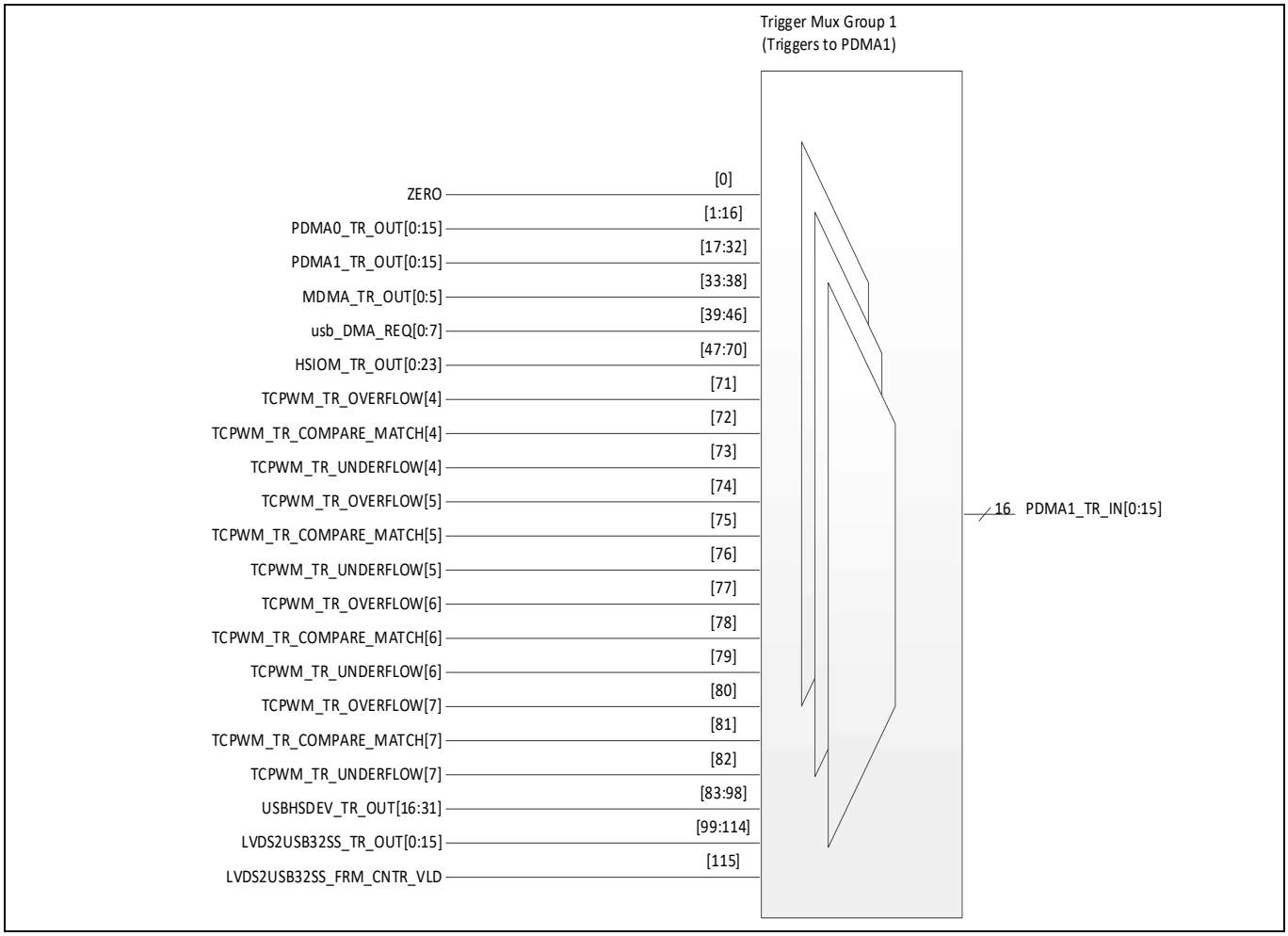


Figure 24-4. Trigger multiplexer block architecture (2 of 9)

Trigger multiplexer block

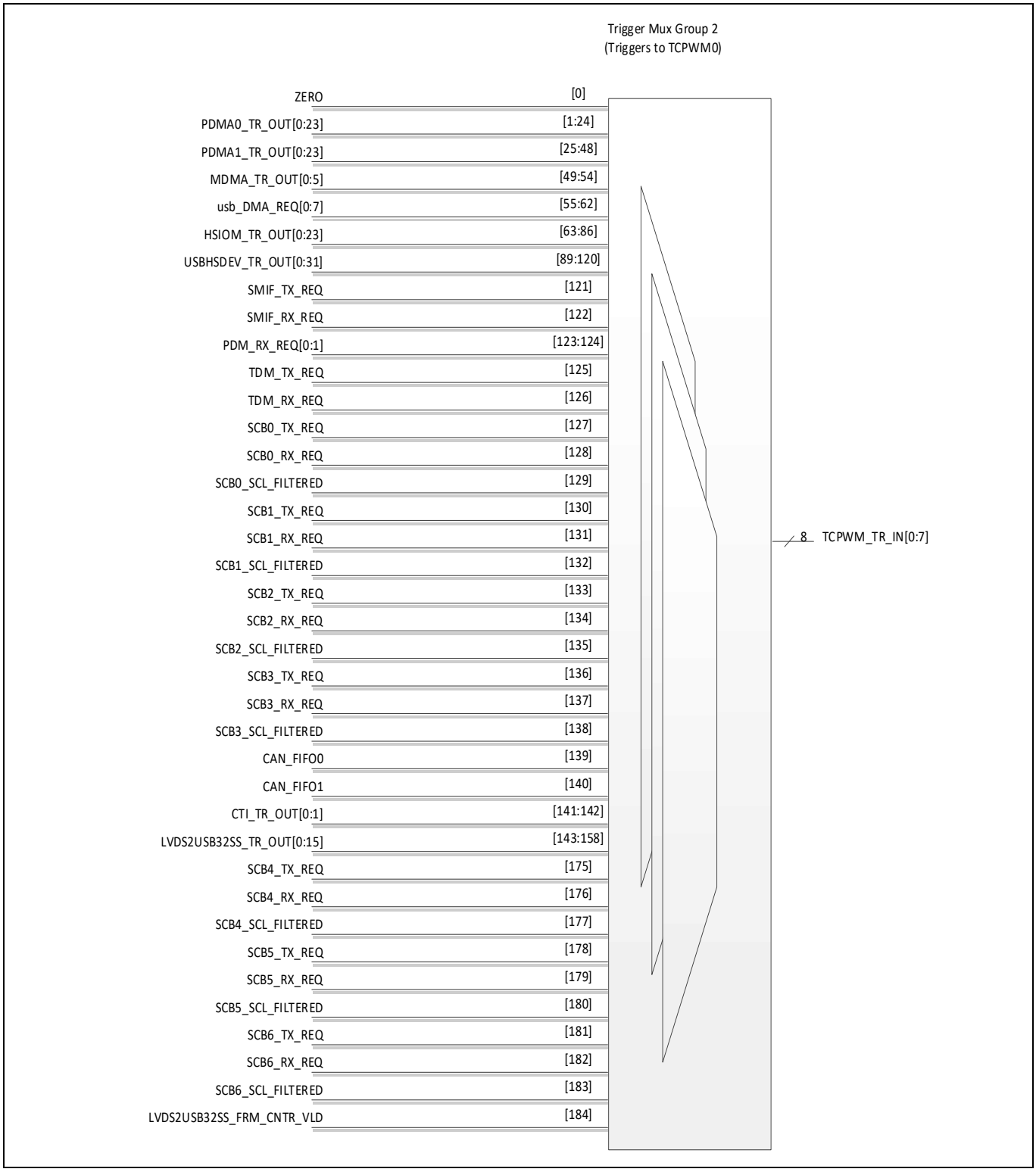


Figure 24-5. Trigger multiplexer block architecture (3 of 9)

Trigger multiplexer block

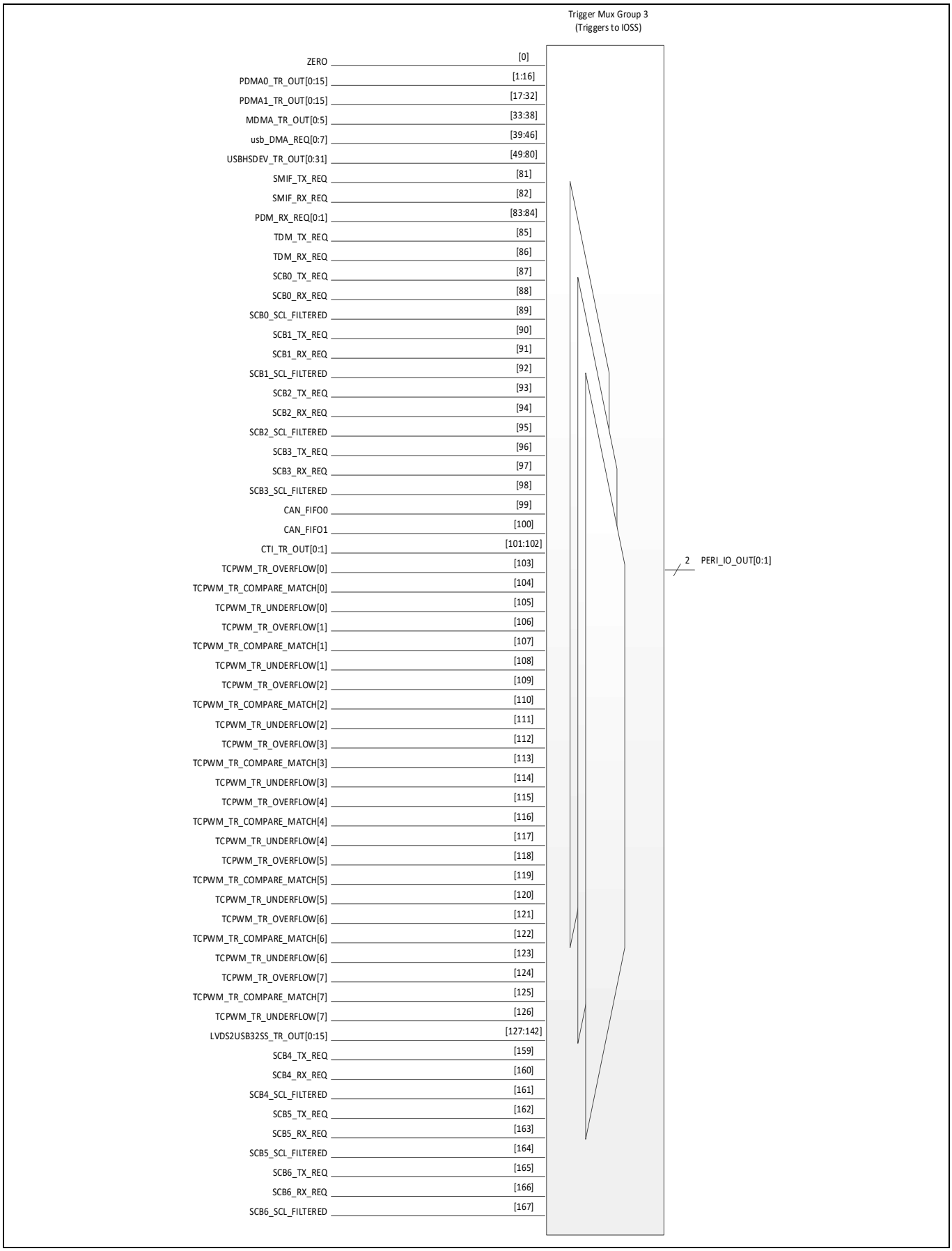


Figure 24-6. Trigger multiplexer block architecture (4 of 9)

Trigger multiplexer block

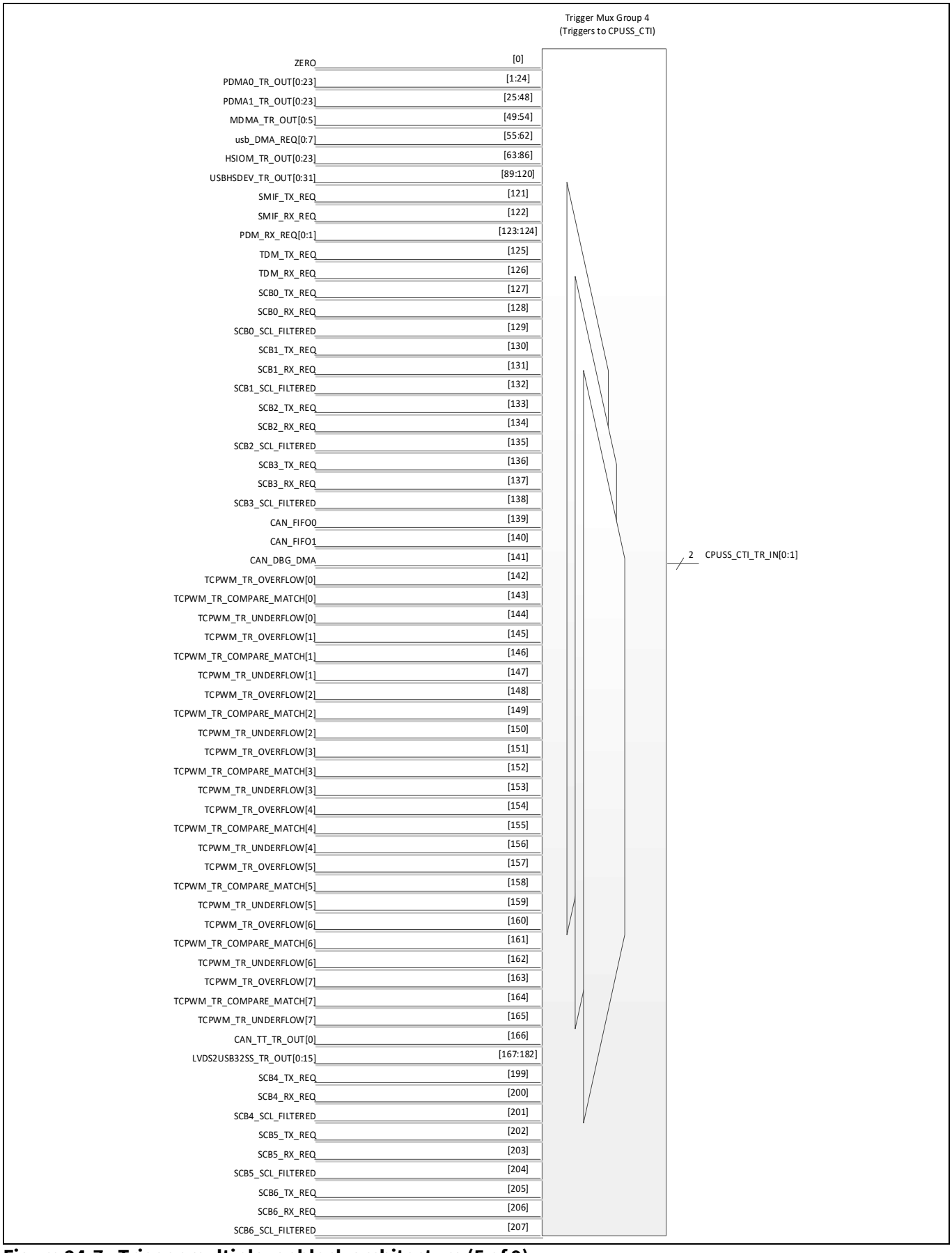


Figure 24-7. Trigger multiplexer block architecture (5 of 9)

Trigger multiplexer block

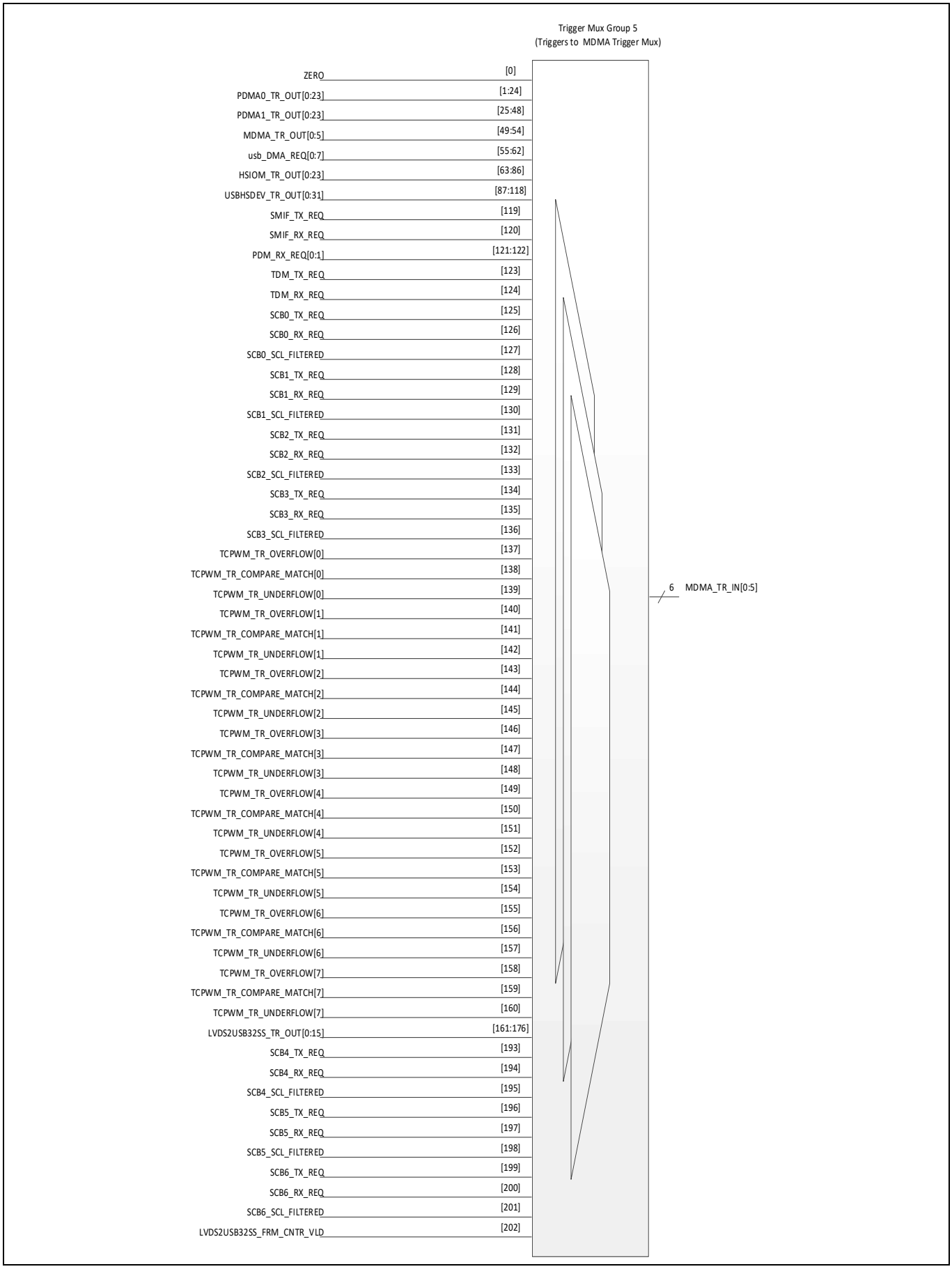


Figure 24-8. Trigger multiplexer block architecture (6 of 9)

Trigger multiplexer block

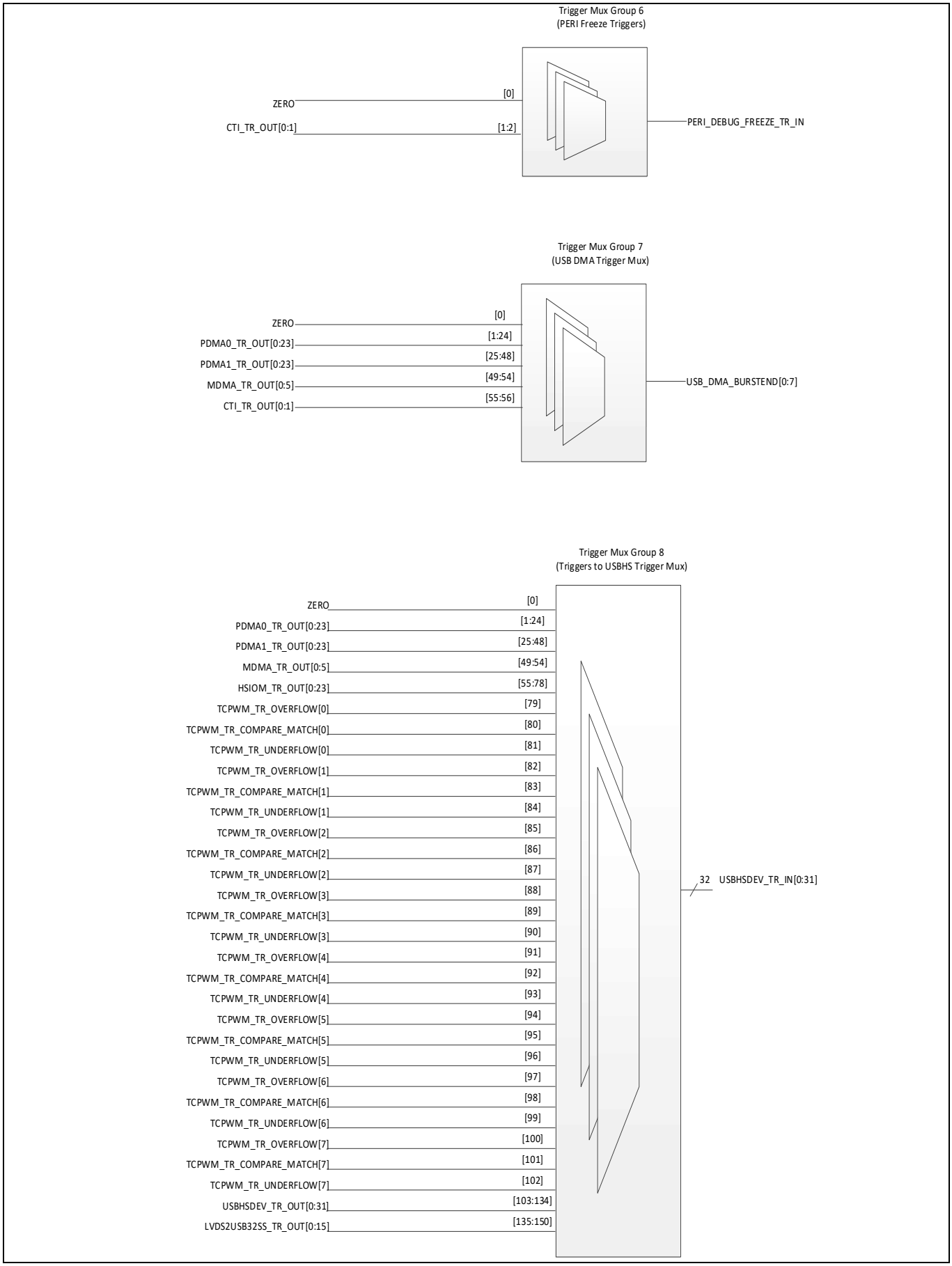


Figure 24-9. Trigger multiplexer block architecture (7 of 9)

Trigger multiplexer block

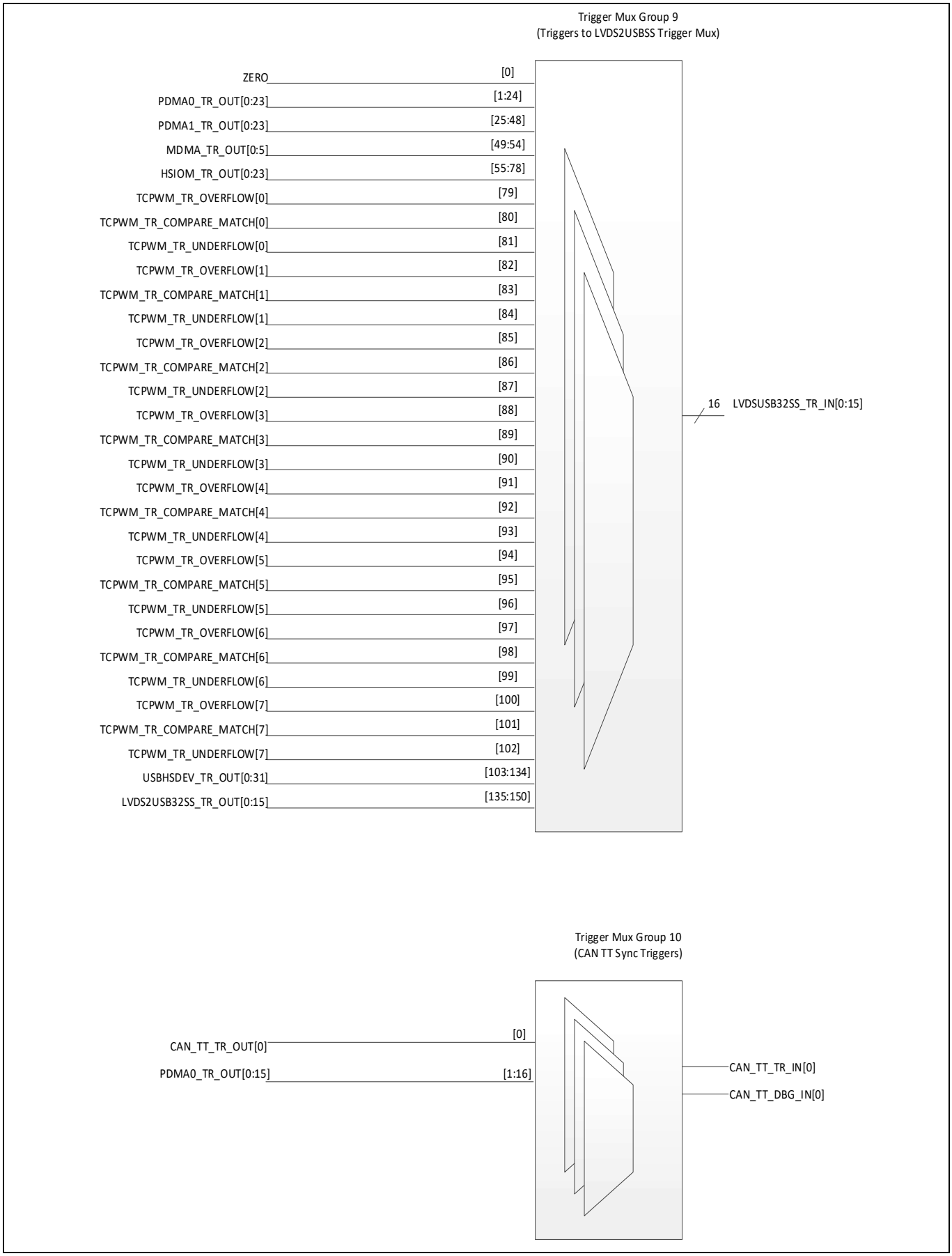


Figure 24-10. Trigger multiplexer block architecture (8 of 9)

Trigger multiplexer block

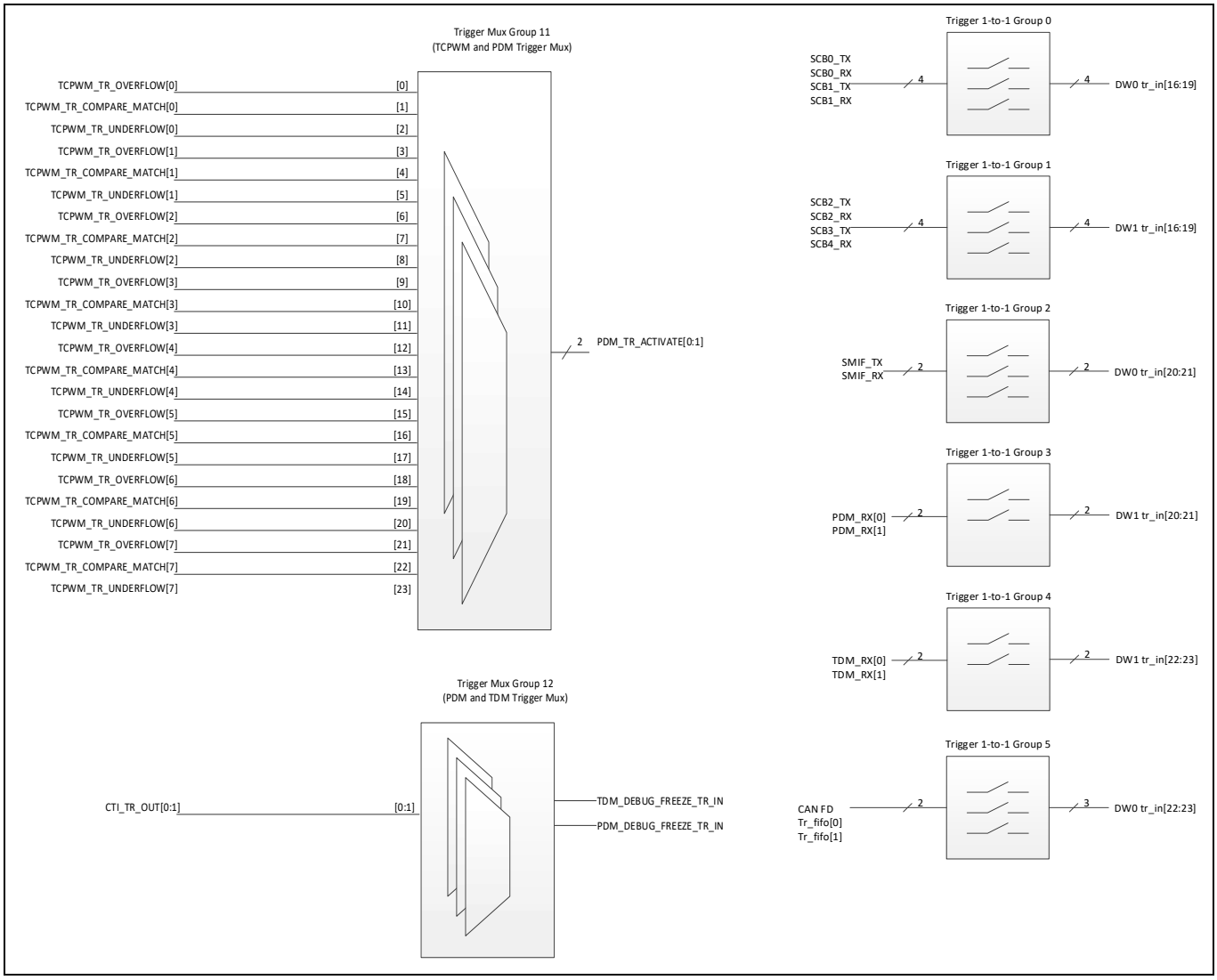


Figure 24-11. Trigger multiplexer block architecture (9 of 9)

24.2.4 Software triggers

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

The PERI_TR_CMD[GROUP_SEL] bitfield selects the trigger group of the signal being activated. The PERI_TR_CMD[OUT_SEL] bitfield determines whether the trigger signal is in output or input of the multiplexer. PERI_TR_CMD[TR_SEL] selects the specific line in the trigger group.

The PERI_TR_CMD[COUNT] bitfield sets up the number of peripheral clocks the trigger will be activated.

The PERI_TR_CMD[ACTIVATE] bitfield is set to '1' to activate the trigger line specified. The hardware resets this bit after the trigger is deactivated after the number of cycles set by the PERI_TR_CMD[COUNT].

24.3 Register list

Table 24-1. Register list

Register name	Description
PERI_TR_CMD	Trigger command register. The control enables software activation of a specific input trigger or output trigger of the trigger multiplexer structure.
PERI_TR_GR[X]_TR_OUT_CTL[Y]	This register specifies the input trigger for a specific output trigger in a trigger group. It can also invert the signal and specify if the output signal should be treated as a level-sensitive or edge-sensitive trigger. Every trigger multiplexer has a group of registers, the number of registers being equal to the output bus size from the multiplexer. In the register format, X is the trigger group and Y is the output trigger line number from the multiplexer.
PERI_TR_1TO1_GR[X]_TR_OUT_CTL[Y]	This register specifies the input trigger for a specific output trigger in a one-to-one trigger group. It also includes trigger manipulation that can invert the signal and specify if the output signal should be treated as a level-sensitive or edge-sensitive trigger. Every one-to-one trigger multiplexer has a group of registers, the number of registers being equal to the output bus size from the multiplexer. The registers are formated with X as the trigger group and Y as the output trigger line number from the multiplexer.

Table 24-2. Trigger group

Trigger group number	Description
0	Routes all input trigger signals to DMA0
1	Routes all input trigger signals to DMA1
2	Routes all input trigger signals to TCPWM0(32-bit)
3	Routes all input trigger signals to IOSS
4	Routes all input trigger signals to CPUSS CTI
5	Routes all input trigger signals to MDMA
6	Routes all input trigger signals to PERI freeze trigger
7	Routes all input trigger signals to USB trigger
8	Routes all input trigger signals to USBHS DNA trigger
9	Routes all input trigger signals to LVDS to USB sub system
10	Routes all input trigger signals to CAN
11	Routes all input trigger signals to TCPWM PDM
12	Routes all input trigger signals to PDM TDM Freeze
Trigger 1-to-1 group	
0	Connects SCB triggers to DMA0
1	Connects SCB triggers to DMA1
2	Connects SMIF triggers to DMA0
3	Connects PDM triggers to DMA1
4	Connects TDM triggers to DMA1
5	Connects CAN triggers to DMA0

Section E: High-bandwidth subsystem (HBWSS)

This section encompasses the following chapters:

- **"LVDS/LVCMOS IO subsystem"** on page 356
- **"GPIF III interface"** on page 385
- **"High-bandwidth DMA subsystem"** on page 390

Top level architecture

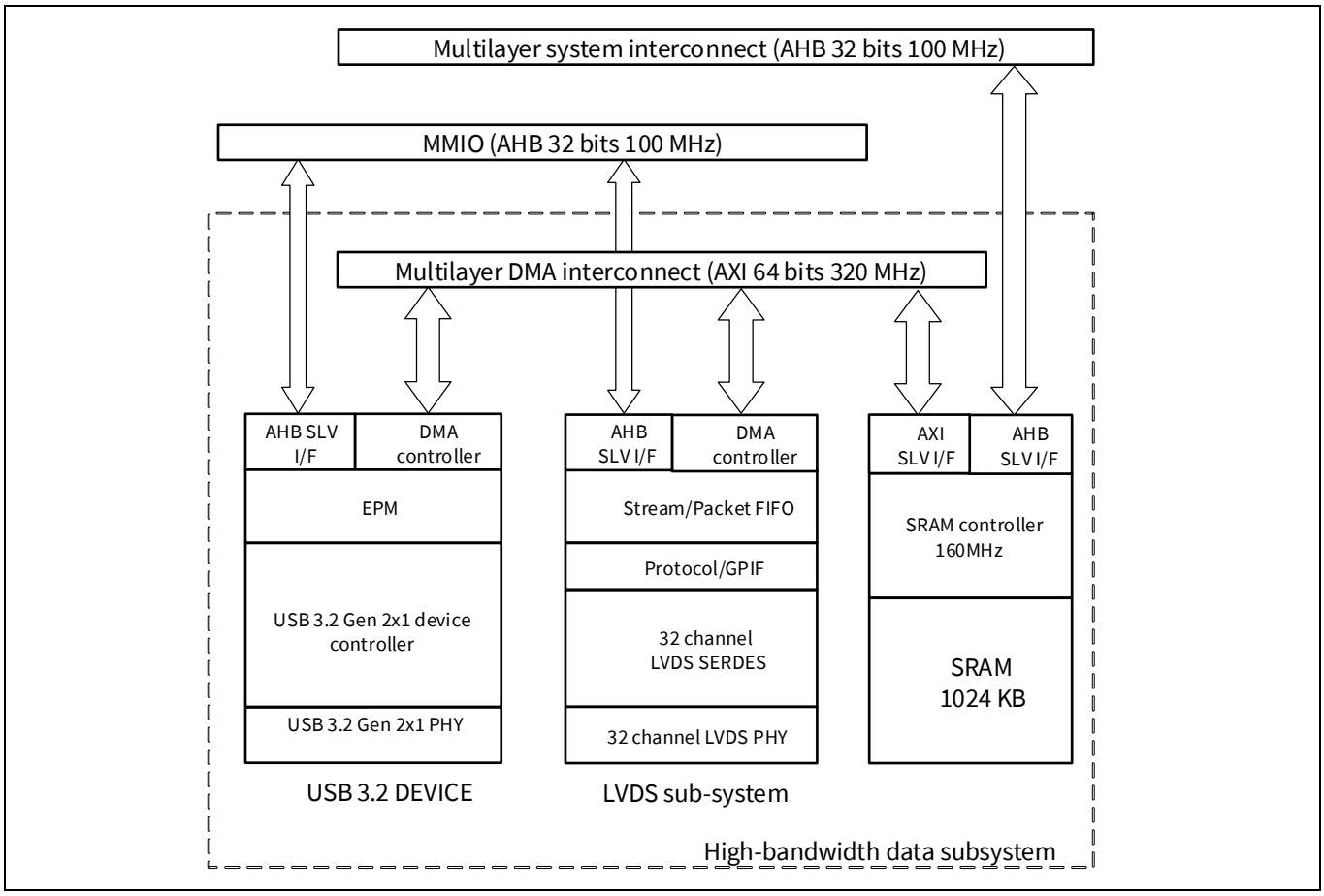


Figure E-1. High-bandwidth sub system block diagram

25 LVDS/LVCMOS IO subsystem

25.1 Features

- LVDS/LVCMOS physical layer
  - Supports two LVDS ports, two LVCMOS ports or one LVDS and one LVCMOS ports configurations
  - The ports form links that have data lanes, a clock lane and control lanes
  - Both ports can be independent links or can be merged to form a wider link
  - LVDS receiver interface
    - Each port can be configured for 1, 2, 4 or 8 data lanes and one clock and control lane
    - Supports gearing ratios[1.] of 1:1, 2:1, 4:1 and 8:1 on each LVDS link
    - Each LVDS link has seven LVCMOS control signals of which three signals are input only and the other four signals are input/output control signals
    - Supports a clock range from 74.25 MHz to 625 MHz
    - Maximum data rate per lane is 1250 Mbps (for example, 625 MHz clock and 2:1 gearing ratio)
    - Supports a programmable pattern for LVDS link training
    - Control word decoding on the control lane to support special features
    - LVDS interface is compliant to TIA-644
  - LVCMOS mode
    - Supports both transmitter and receiver mode of operation
    - Each port supports up to sixteen data lanes, one clock line and up to ten control signals
    - Data lanes support SDR and DDR mode of operation
    - Supports master mode operation where the interface clock is output by the EZ-USB™ FX10
    - The maximum data rate on each LVCMOS IO is:
      160 MHz for single data rate (SDR) and double data rate (DDR) in Receiver mode
  - Transmitter mode:
    - 100 MHz SDR (max 3.2 Gbps)
    - 80 MHz DDR in Master mode (max 5 Gbps)
  - Low power mode entry under firmware control
- GPIF III link layer
  - Configurable state machine that enables a flexible interface that may function as a master or slave in industry-standard or proprietary interfaces
  - Supports 256 firmware programmable states
  - Supports 8-bit, 16-bit, 24-bit, and 32-bit parallel data bus
  - Supports two independent instances of GPIF III state machines each connected to a link or single GPIF III state machine connected to wide link
  - Supports Short Length Packets and Zero Length Packets
  - Supports an Idle timer to indicate absence of link activity
  - Supports accurate time stamp addition and addition of header and footer to incoming data stream using control byte encoding feature
- High-Bandwidth DMA subsystem
  - Supports multiple simultaneous bidirectional streams
  - DMA is configurable by the CPU
  - After configuration, the DMA operates automatically and transfers the data stream between the LVDS/LVCMOS interface and the USB 10 Gbps block without CPU intervention

Note:

1. *Gearing ratio indicates the number of bits transferred, per data lane, per clock on the interface. For example, gearing ratio 2:1 means that two bits of data are transferred per lane in each clock.*

25.2 Architecture

LVDS/LVCMOS interface is used to connect to external data sources to transfer data over EZ-USB™ FX10's LVDS or LVCMOS IOs to other peripherals over DMA. This block is internally split into a physical layer (LVDS/LVCMOS Ports), a link layer (GPIF III Links) and a high-bandwidth DMA layer. The following sections contain the architectural details of each of these layers.

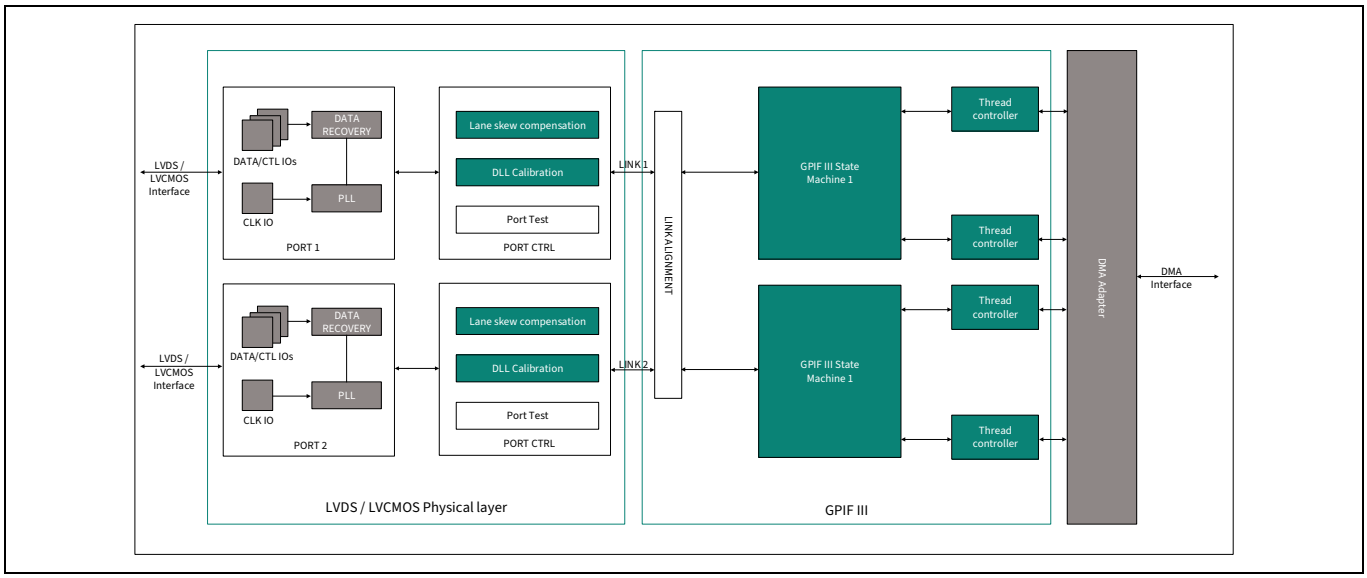


Figure 25-1. LVDS/LVCMOS IO subsystem block diagram

The physical layer (PHY) for the LVDS mode controls the data recovery and SERializer-DESerialzer (SERDES) functionality of the interface. It also has a phase-locked loop (PLL) to multiply the input clock to generate the serial clock (at-speed clock) and frame clock (end-of-packet clock). The deserializer uses the serial clock to sample the incoming data, which are then output every period of the frame clock.

The primary function of the PHY is skew compensation between the lanes in LVDS and LVCMOS DDR modes and the delay-locked loops (DLL) calibration.

A deskewing algorithm is performed in order to adjust the phase relationship between each one of the incoming data lanes and the frame clock and the serial clock.

Deskewing is performed after the DLL locking procedure has been completed.

A monitor algorithm starts after the deskewing algorithm has finished. This monitor algorithm is constantly running in background during normal operation in order to adjust the optimal serial clock sampling point found by the deskewing algorithm over temperature and voltage variations. This algorithm operates independently on all data lanes and on the frame clock.

In LVCMOS mode, the data is directly forwarded to the digital part of the PHY along with the sampling clock with no deserialization. However, in LVCMOS DDR mode, de-skewing is present for the input signals.

The PHY also implements built-in self-test functionality for each of the constituent IP blocks.

The PHY transfers the received serial data at each IO port in parallel form to the link layer. The link layer for EZ-USB™ FX10's LVDS / LVCMOS interface consists of two independent GPIF III state machines and four GPIF thread controllers. Each thread controller act as an interface between the core and the DMA adapters. GPIF state machine 1 can access all thread controllers (in Wide link mode), whereas GPIF State Machine 2 can access only Threads 2 and 3. The link layer also compensates for lane skew and aligns data bytes of all lanes of a port. It also does link alignment to align the data of both ports in Wide link mode.

The high-bandwidth DMA layer handles data transfers between the LVDS / LVCMOS interface and USB Interface Block or other system peripherals. This layer has a distributed DMA controller that uses the AXI3 bus architecture.

25.2.1 LVDS/LVCMOS ports vs. LVDS/LVCMOS links

The port terminology refers to the physical pins of the LVDS/LVCMOS I/O interface. These pins are named with a prefix of the port number (P0CP or P1CN, where P0 = Port 0 and P1 = Port 1).

The data flow path between the ports and the GPIF III interface is called links.

In EZ-USB™ FX10, two ports can be configured to be independent data paths (or links), or two ports can be merged to form a single link to route data from two ports to the GPIF III interface as a single link (wide link).

In summary, the following are the two possibilities for port-to-link mapping:

- Port 0 mapped to Link 0 and Port 1 mapped to Link 1 (Narrow Link mode)

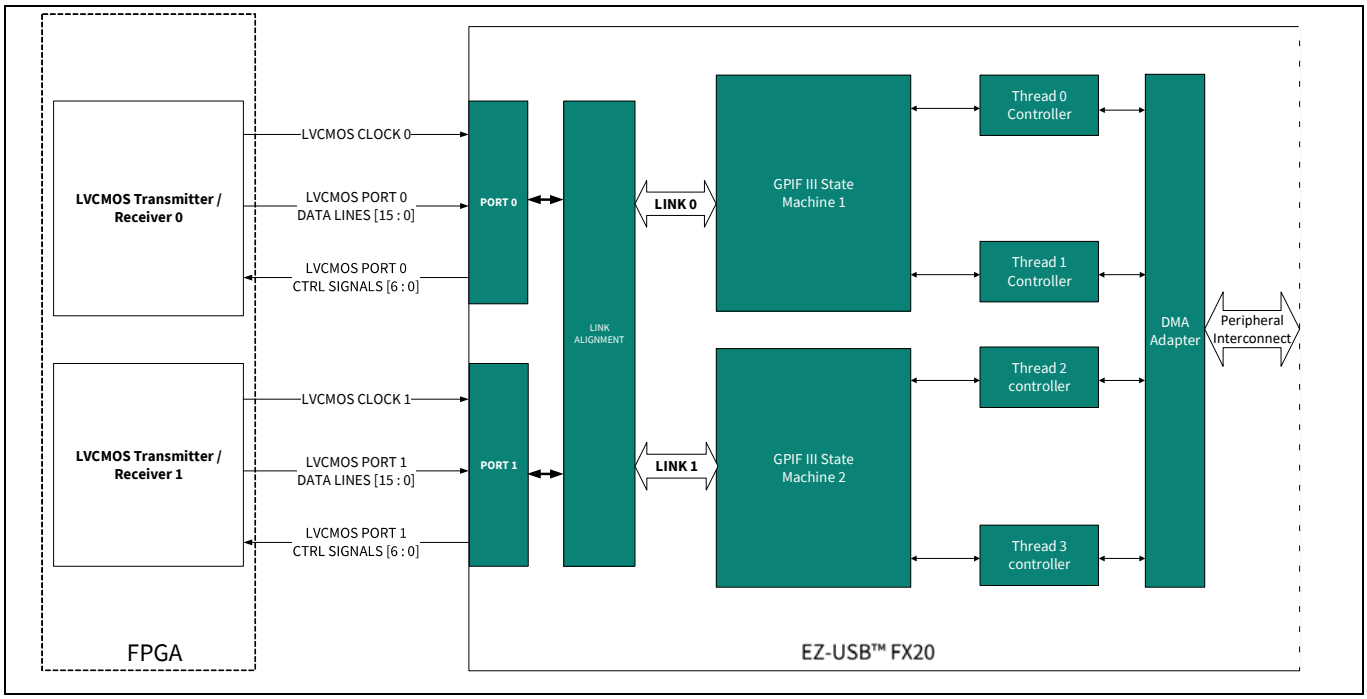


Figure 25-2. Ports in Narrow Link mode

- Port 0 and Port 1 are mapped to Link 0 (Wide Link mode)

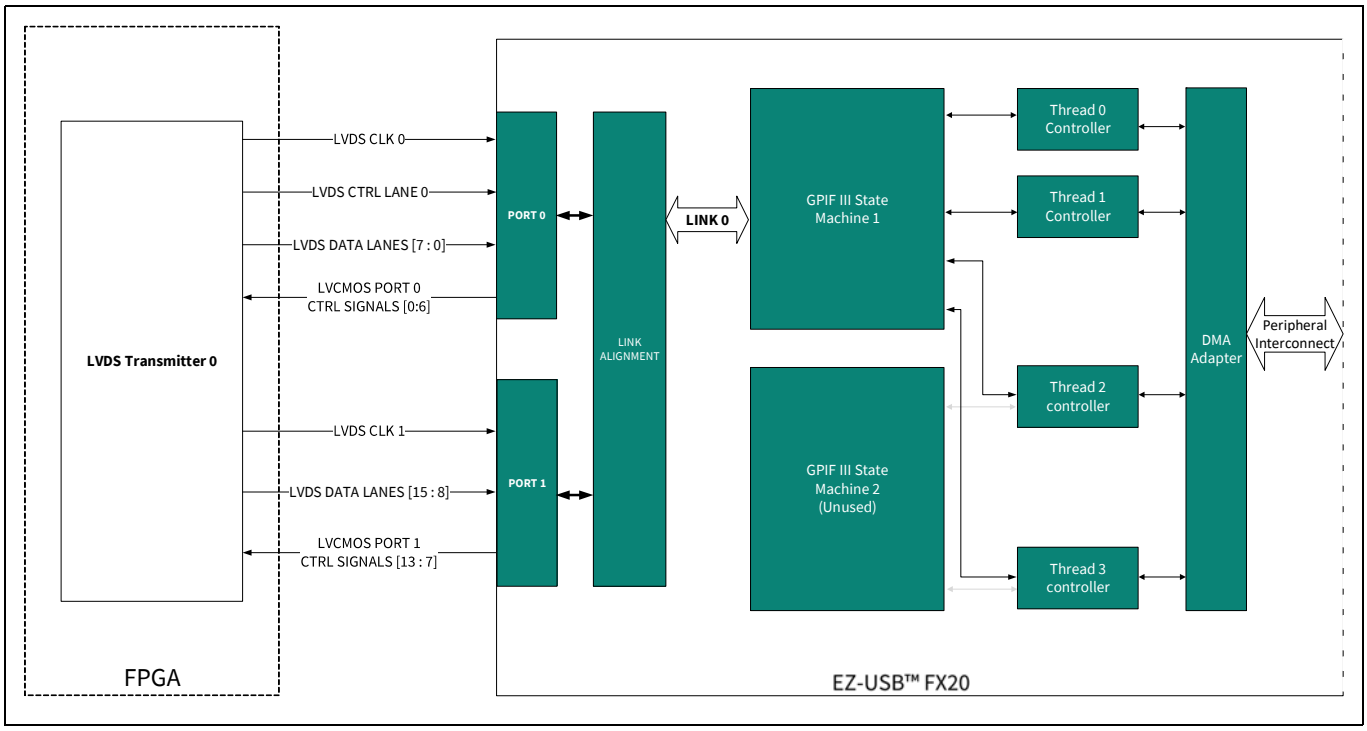


Figure 25-3. Ports in Wide Link mode

25.2.2 LVDS and LVCMOS port combinations

One port or a combination of two ports can be configured as a single link in either LVDS mode or LVCMOS mode. Port signals can be combined to form wider LVDS link or LVCMOS bus as shown in **Table 25-1**.

Table 25-1. Possible port combinations on the LVDS/LVCMOS interface

Configuration	Port0	Port1
Narrow Link	LVDS input link with 1, 2, 4 or 8 data lanes, 1 clk lane, 1 ctrl lane, 3 input and 4 output LVCMOS ctrl signals	LVDS input link with 1, 2, 4 or 8 data lanes, 1 clk lane, 1 ctrl lane, 3 input and 4 output LVCMOS ctrl signals
	LVCMOS input/output bus with 8 or 16 data, 1 clk, and 10 input/output ctrl signals	LVCMOS input/output bus with 8 or 16 data, 1 clk, and 10 input/output ctrl signals
	LVCMOS input/output bus with 8 or 16 data, 1 clk, and 10 input/output ctrl signals	LVDS input link with 1, 2, 4 or 8 data lanes, 1 clk lane, 1 ctrl lane, 3 input and 4 output LVCMOS ctrl signals
	LVDS input link with 1, 2, 4 or 8 data lanes, 1 clk lane, 1 ctrl lane, 3 input and 4 output LVCMOS ctrl signals	LVCMOS input/output bus with 8 or 16 data, 1 clk, and 10 input/output ctrl signals
Wide Link	LVDS input link with 16 data lanes, 2 clk lane, 1 ctrl lane (of Port 0) and 14 input/output LVCMOS ctrl signals	
	LVDS input link with 12 data lanes, 2 clk lane, 1 ctrl lane (of Port 0) and 14 input/output LVCMOS ctrl signals	
	LVCMOS input/output bus with 24 or 32 data, 1 clk, and 20 input/output ctrl signals	

LVDS/LVCMOS IO subsystem

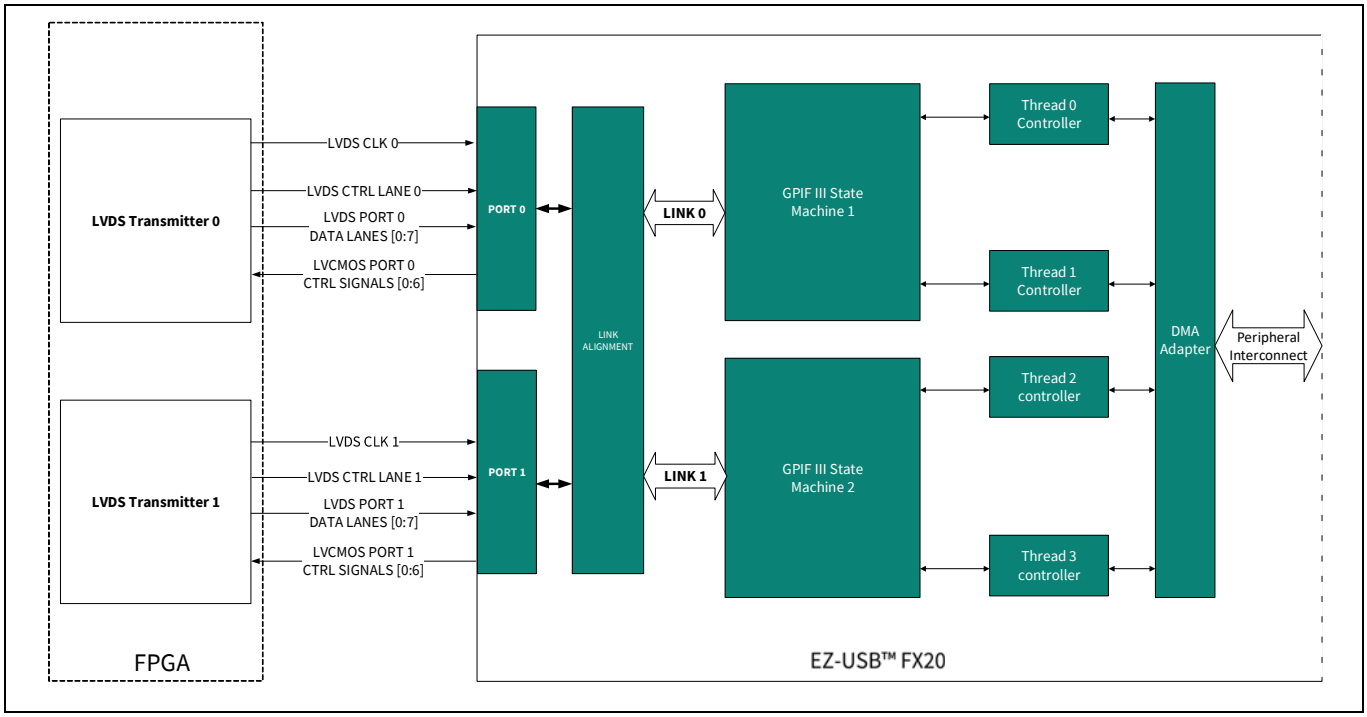


Figure 25-4. LVDS Narrow Link configuration

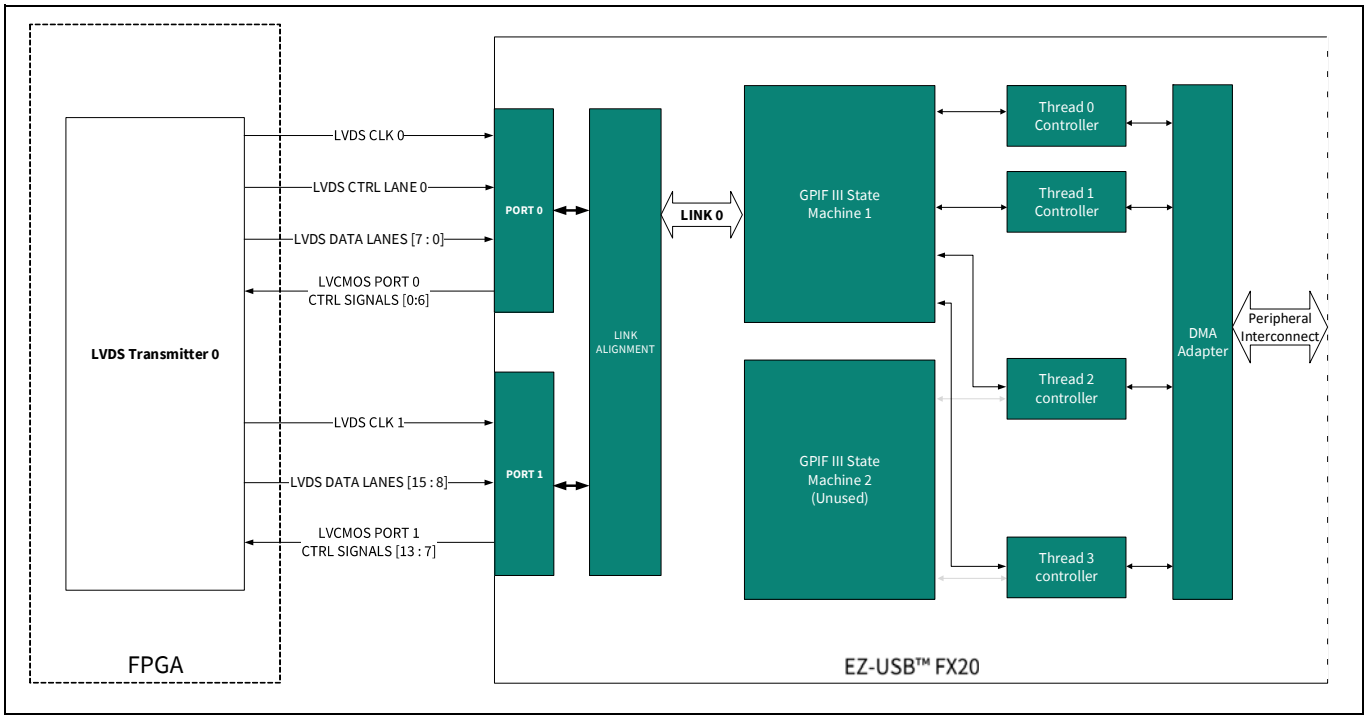


Figure 25-5. LVDS Wide Link configuration

LVDS/LVCMOS IO subsystem

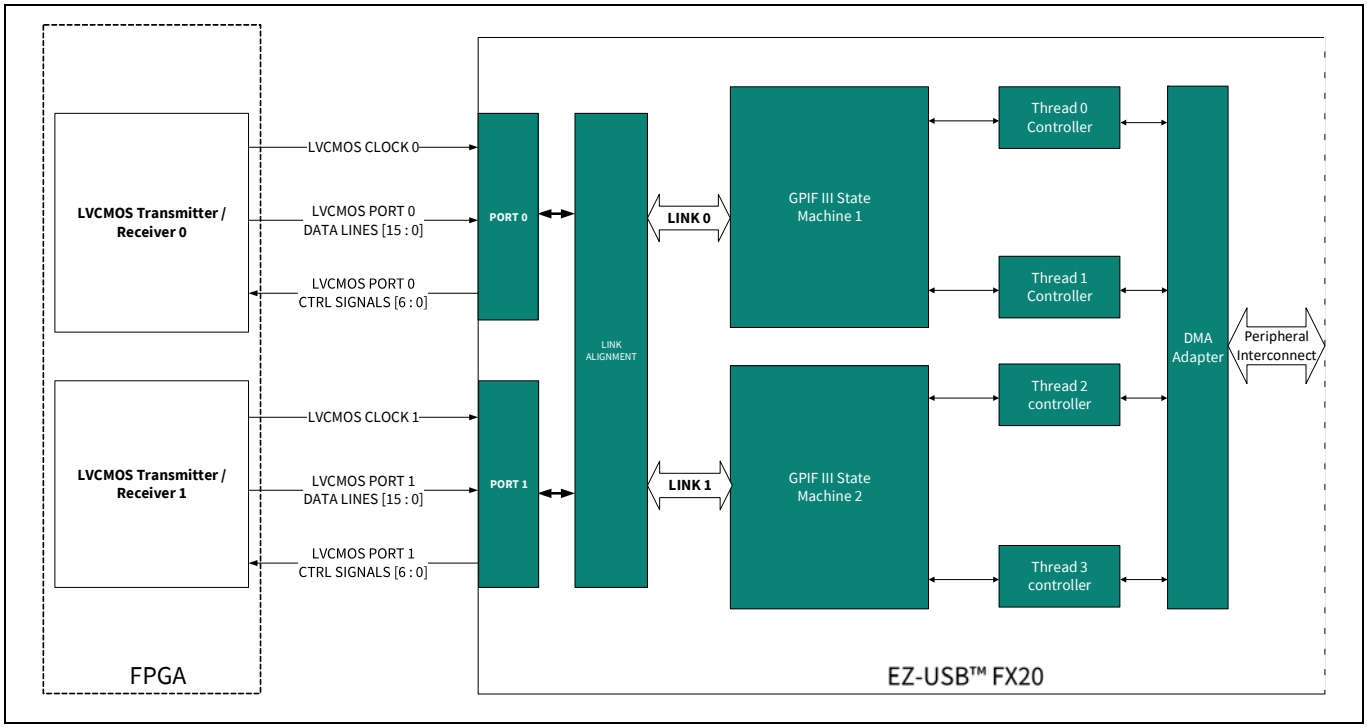


Figure 25-6. LVCMOS Narrow Link configuration

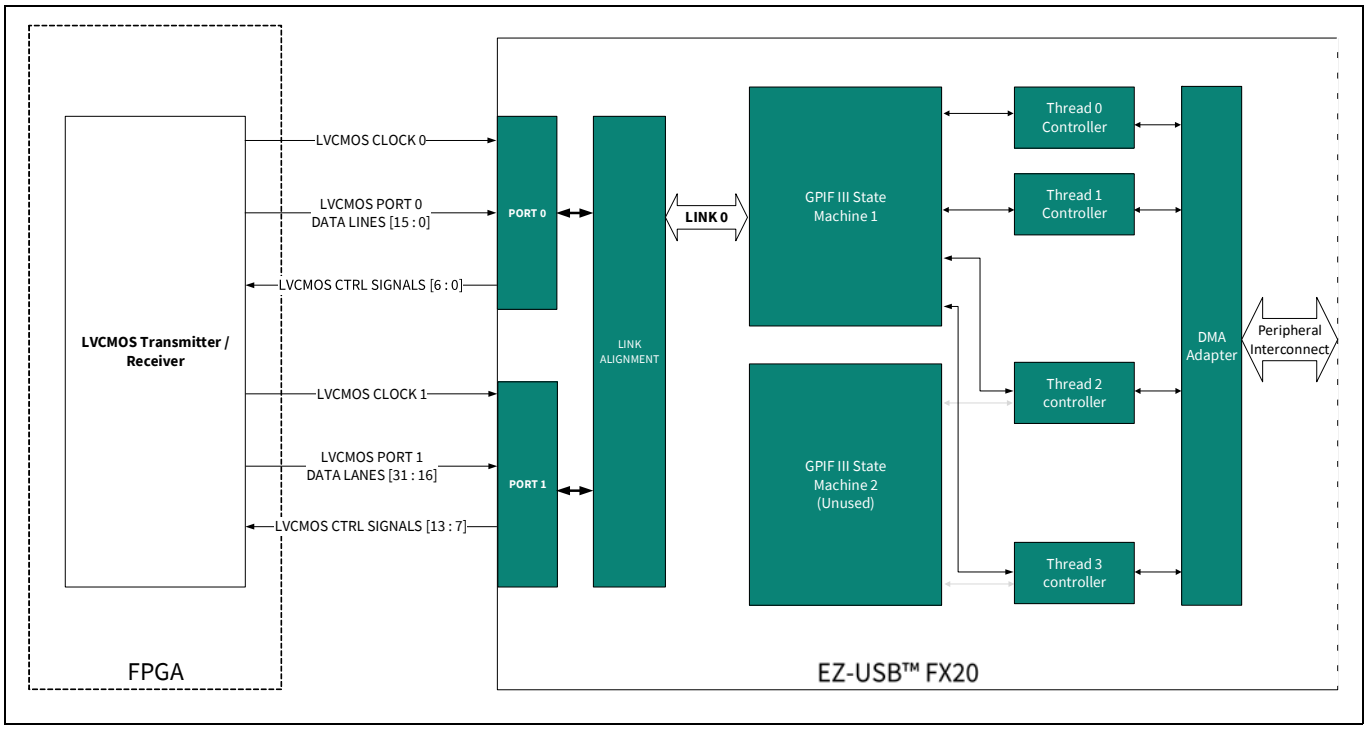


Figure 25-7. LVCMOS Wide Link configuration

25.2.3 LVCMOS master mode configuration

The LVCMOS block of the EZ-USB™ FX10 also has the capability to operate in a mode where the interface clock is transmitted from the EZ-USB™ FX10 itself. This mode of operation is called as Master mode. Master mode can only be used in cases where the LVCMOS block is used as transmitter (i.e. data is sent from EZ-USB™ FX10 to FPGA), and data lines cannot be driven by FPGA in this mode.

Currently, the LVCMOS master mode operation supports 80 MHz DDR and 100 MHz SDR clock configurations. Hence, the maximum transmit throughput in EZ-USB™ FX10's LVCMOS transmit mode is 5 Gbps (32 lines * 80 MHz * 2). The direct state switching feature (DSS) of GPIF FSM is not supported in this mode.

25.2.4 LVDS Link configurations

1. The gearing ratios (see note below) 1:1, 2:1, 4:1 and 8:1 are supported on each configuration of link.

Note: Gearing ratio indicates the number of bits transferred, per data lane, per clock on the interface. For example, gearing ratio 2:1 means that two bits of data are transferred per lane in each clock.

2. The number of data lanes per link and gearing ratios are configured at the time of this block's initialization.
3. LVDS transmitter or FPGA must send a minimum of one byte on each data and ctrl lane. Partial byte transfers are not allowed.
4. The FPGA should transmit the Most Significant bit (MSb) of the byte first.
5. One full byte in control lane identifies various types of control information such as training sequence, idle, full-width data, partial-data, frame-start, frame-end, line-start, line-end, and so on. Different control bytes supported, their encoding and functionality are captured in **Table 25-5**.
6. Each data lane always brings in a minimum of 1 byte of data. If partial data is present on the link, they shall be present on the least significant lanes. Partial transfers will be needed if the number of bytes to be transferred is less than the number of lanes i.e., in a 8-lane configuration with gearing ratio 1:1, if the FPGA only has 4 bytes in the last transfer. In this case, these 4 bytes should be transferred in lane 0 to lane 3 and the valid data lanes should be signaled to the receiver using the "p0001000" control byte to sample the 4 least significant lanes (lanes 0-3).
7. After firmware enables the block, LVDS transmitter or FPGA is expected to send a training sequence on all data lanes and ctrl lane of a link - no other control codes are expected for first (Training Sequence Time (TS_Time = 50 µs).

Table 25-2 shows control and data bits present on 8, 4, 2, and 1 lane links for each of gearing ratios 1:1, 2:1, 4:1, 8:1. The clk pulse column in **Table 25-2** shows the clock count.

Note:

1. *PxD0-7 signifies Port x, where x is either 0 or 1.*
2. *D[x] signifies received data (64-bit, 32-bit, 16-bit or 8-bit).*

Table 25-2. LVDS Link configurations

Lanes	Gearing [Data Bit : Clock]	Clock Pulse	Ctrl lane bits	PxD7 lane	PxD6 lane	PxD5 lane	PxD4 lane	PxD3 lane	PxD2 lane	PxD1 lane	PxD0 lane
8	1:1	0	C[7]	D[63]	D[55]	D[47]	D[39]	D[31]	D[23]	D[15]	D[7]
		1	C[6]	D[62]	D[54]	D[46]	D[38]	D[30]	D[22]	D[14]	D[6]
		2	C[5]	D[61]	D[53]	D[45]	D[37]	D[29]	D[21]	D[13]	D[5]
		3	C[4]	D[60]	D[52]	D[44]	D[36]	D[28]	D[20]	D[12]	D[34
		4	C[3]	D[59]	D[51]	D[43]	D[35]	D[27]	D[19]	D[11]	D[3]
		5	C[2]	D[58]	D[50]	D[42]	D[34]	D[26]	D[18]	D[10]	D[2]
		6	C[1]	D[57]	D[49]	D[41]	D[33]	D[25]	D[17]	D[9]	D[1]
		7	C[0]	D[56]	D[48]	D[40]	D[32]	D[24]	D[16]	D[8]	D[0]
8	2:1	0	C[7:6]	D[63:62]	D[55:54]	D[47:46]	D[39:38]	D[31:30]	D[23:22]	D[15:14]	D[7:6]
		1	C[5:4]	D[61:60]	D[53:52]	D[45:44]	D[37:36]	D[29:28]	D[21:20]	D[13:12]	D[5:4]
		2	C[3:2]	D[59:58]	D[51:50]	D[43:42]	D[35:34]	D[27:26]	D[19:18]	D[11:10]	D[3:2]
		3	C[1:0]	D[57:56]	D[49:48]	D[41:40]	D[33:32]	D[25:24]	D[17:16]	D[9:8]	D[1:0]
8	4:1	0	C[7:4]	D[63:60]	D[55:52]	D[47:44]	D[39:38]	D[31:28]	D[23:20]	D[15:12]	D[7:4]
		1	C[3:0]	D[59:56]	D[51:48]	D[43:40]	D[35:32]	D[27:24]	D[19:16]	D[11:8]	D[3:0]
8	8:1	0	C[7:0]	D[63:56]	D[55:48]	D[47:40]	D[39:32]	D[31:24]	D[23:16]	D[15:8]	D[7:0]
4	1:1	0	C[7]	–	–	–	–	D[31]	D[23]	D[15]	D[7]
		1	C[6]	–	–	–	–	D[30]	D[22]	D[14]	D[6]
		2	C[5]	–	–	–	–	D[29]	D[21]	D[13]	D[5]
		3	C[4]	–	–	–	–	D[28]	D[20]	D[12]	D[4]
		4	C[3]	–	–	–	–	D[27]	D[19]	D[11]	D[3]
		5	C[2]	–	–	–	–	D[26]	D[18]	D[10]	D[2]
		6	C[1]	–	–	–	–	D[25]	D[17]	D[9]	D[1]
		7	C[0]	–	–	–	–	D[24]	D[16]	D[8]	D[0]
4	2:1	0	C[7:6]	–	–	–	–	D[31:30]	D[23:22]	D[15:14]	D[7:6]
		1	C[5:4]	–	–	–	–	D[29:28]	D[21:20]	D[13:12]	D[5:4]
		2	C[3:2]	–	–	–	–	D[27:26]	D[19:18]	D[11:10]	D[3:2]
		3	C[1:0]	–	–	–	–	D[25:24]	D[17:16]	D[9:8]	D[1:0]
4	4:1	0	C[7:4]	–	–	–	–	D[31:28]	D[23:20]	D[15:12]	D[7:4]
		1	C[3:0]	–	–	–	–	D[27:24]	D[19:16]	D[11:8]	D[3:0]
4	8:1	0	C[7:0]	–	–	–	–	D[31:24]	D[23:16]	D[15:8]	D[7:0]

Table 25-2. LVDS Link configurations (continued)

Lanes	Gearing [Data Bit : Clock]	Clock Pulse	Ctrl lane bits	PxD7 lane	PxD6 lane	PxD5 lane	PxD4 lane	PxD3 lane	PxD2 lane	PxD1 lane	PxD0 lane
2	1:1	0	C[7]	–	–	–	–	–	–	D[15]	D[7]
		1	C[6]	–	–	–	–	–	–	D[14]	D[6]
		2	C[5]	–	–	–	–	–	–	D[13]	D[5]
		3	C[4]	–	–	–	–	–	–	D[12]	D[4]
		4	C[3]	–	–	–	–	–	–	D[11]	D[3]
		5	C[2]	–	–	–	–	–	–	D[10]	D[2]
		6	C[1]	–	–	–	–	–	–	D[9]	D[1]
		7	C[0]	–	–	–	–	–	–	D[8]	D[0]
2	2:1	0	C[7:6]	–	–	–	–	–	–	D[15:14]	D[7:6]
		1	C[5:4]	–	–	–	–	–	–	D[13:12]	D[5:4]
		2	C[3:2]	–	–	–	–	–	–	D[11:10]	D[3:2]
		3	C[1:0]	–	–	–	–	–	–	D[9:8]	D[1:0]
2	4:1	0	C[7:4]	–	–	–	–	–	–	D[15:12]	D[7:4]
		1	C[3:0]	–	–	–	–	–	–	D[11:8]	D[3:0]
2	8:1	0	C[7:0]	–	–	–	–	–	–	D[15:8]	D[7:0]
1	1:1	0	C[7]	–	–	–	–	–	–	–	D[7]
		1	C[6]	–	–	–	–	–	–	–	D[6]
		2	C[5]	–	–	–	–	–	–	–	D[5]
		3	C[4]	–	–	–	–	–	–	–	D[4]
		4	C[3]	–	–	–	–	–	–	–	D[3]
		5	C[2]	–	–	–	–	–	–	–	D[2]
		6	C[1]	–	–	–	–	–	–	–	D[1]
		7	C[0]	–	–	–	–	–	–	–	D[0]
1	2:1	0	C[7:6]	–	–	–	–	–	–	–	D[7:6]
		1	C[5:4]	–	–	–	–	–	–	–	D[5:4]
		2	C[3:2]	–	–	–	–	–	–	–	D[3:2]
		3	C[1:0]	–	–	–	–	–	–	–	D[1:0]
1	4:1	0	C[7:4]	–	–	–	–	–	–	–	D[7:4]
		1	C[3:0]	–	–	–	–	–	–	–	D[3:0]
1	8:1	0	C[7:0]	–	–	–	–	–	–	–	D[7:0]

25.2.5 LVCMOS link configurations

Table 25-3 shows how an LVCMOS transmitter or FPGA should transmit 16- or 8-bits of data to EZ-USB™ FX10's LVCMOS interface. D[0] signifies bit 0 of Data D.

Table 25-3. LVCMOS link configurations

Bus width	Gearing	Clock pulse	Ctrl Bus[9:0]	PXD15	PXD14	PXD13	PXD12	PXD11	PXD10	PXD9	PXD8	PXD7	UPXD6	PXD5	PXD4	PXD3	PXD2	PXD1	PxD0
16	1:1	0	C[9:0]	D[15]	D[14]	D[13]	D[12]	D[11]	D[10]	D[9]	D[8]	D[7]	D[6]	D[5]	D[4]	D[3]	D[2]	D[1]	D[0]
16	2:1	0	C[19:0]	D [31:30]	D [29:28]	D [27:26]	D [25:24]	D [23:22]	D [21:20]	D [19:18]	D [17:16]	D [15:14]	D [13:12]	D [11:10]	D [9:8]	D [7:6]	D [5:4]	D [3:2]	D[1:0]
8	1:1	0	C[9:0]									D[7]	D[6]	D[5]	D[4]	D[3]	D[2]	D[1]	D[0]
8	2:1	0	C[19:0]									D [15:14]	D [13:12]	D [11:10]	D [9:8]	D [7:6]	D [5:4]	D [3:2]	D [1:0]

25.3 Functional operating modes

25.3.1 Training mode

This mode is entered when the LVDS receiver block is enabled by firmware when the CPU is ready to receive data. This mode is also entered when the block exits from LVDS low power modes. A link-ready indication should be sent to the LVDS transmitter device (LVDS-TX) via I2C, SPI or control signals over LVCMOS interface.

Table 25-4. LVDS training

Mode	Width	PHY training required?	Link training required?
LVCMOS SDR	Narrow link	N	N
	Wide link	N	N
LVCMOS DDR	Narrow link	N	N
	Wide link	N	Y
LVDS	Narrow link	Y	Y
	Wide link	Y	Y

On receiving the Link Ready indication from EZ-USB™ FX10, the LVDS-TX should start the training sequence. The training sequence consists of two stages:

- PHY training to align byte boundaries on each lane
- Link training to enable lane and link alignment

The description for the sequence are as follows:

25.3.1.1 PHY Training

- After LVDS block is enabled or after it exits from LVDS lower-power modes, the LVDS-TX must send the link training sequence on all the lanes of the link before sending data. The control lane is used to identify the training sequence using the TSEQ control byte, from other types of data transmitted on the data lanes.
- The training sequence should be sent by LVDS-TX as repeated 8-bit training pattern (TS_Pattern). The 8-bit training pattern is programmable through configuration registers (LVDSSS_LVDS_AFEx_PHY_TRAIN_CONFIG register). The supported PHY training pattern values are 0xB3, 0x4C, 0xCA, 0x35, 0xBA, 0x45, 0x9C, 0x63, 0x39, and 0xC6.
- LVDS-TX should send the training sequence for a minimum training time (TS_Time) of around 50 microseconds from Link-ready.
- Once trained, the link does not need to be re-trained until
  - clock is stopped by LVDS-TX or
  - the link comes out of low power mode (L1 or L3 exit) or
  - the block is re-enabled
- This block does not provide acknowledgment to indicate training completion back to LVDS-TX. After sending the training sequence for TS-time, LVDS-Tx should transmit the link training blocks. The PLL-lock, lane- and link-alignment is achieved within “TS_Time”.

There is no bit-error rate on LVDS links. LVDS link is susceptible to abnormal EMI.

25.3.1.2 Link Training

The Link Training is done after completion of PHY training sequence. The goal is to align data bytes of all active lanes in LVDS mode. In case of LVCMOS, link training is needed only in LVCMOS DDR Wide Link mode.

The Link training block consists of a 4-byte sequence on each data lanes from data lane 15 through Lane 0 and the control lane. The training block bytes are configurable using LVDSSS_LVDS_TRAINING_BLK register in EZ-USB™ FX10.

The FPGA should wait for 3 µs after sending the link training pattern before moving to data transmission.

During normal operation, when skew of lanes drift so much that it is not possible for hardware to find the optimal clock phase to sample data lanes, then hardware will raise LVDS_AFE_PHY_INTR.MONITOR_ADJUST_FAIL interrupt. On seeing this interrupt firmware and FPGA has to re-initiate training.

25.3.2 Data transmission mode

This mode is entered once the link training is complete. The LVDS-TX indicates presence of data on the link using LVDS control lane, for the duration of data transmission. LVDS-TX can perform full/partial width data transfers. The number of bytes valid in a transfer should be indicated by the data control lane byte.

In case of LVCMOS, the LVDS-TX indicates the data presence on the interface using LVCMOS control pins (for example: SLCS, SLWR, SLRD, and so on).

For both LVDS and LVCMOS modes, LVCMOS control pins can also be used as status signals (such as buffer_ready to accept data, transmit data, and so on).

25.4 LVDS control byte encoding

EZ-USB™ FX10 LVDS/LVCMOS I/O interface accepts 8-bit control bytes to handle the received data either as a special command byte or actual data. In case of the LVDS Interface, these control bytes are received over EZ-USB™ FX10 control channel (PxCP and PxCN, where x = 0 and 1 for Port 0 and 1 respectively). In case of LVCMOS, the control bytes are supported in Enhanced Metadata mode. In this mode, the control bytes are transferred using data pins D15-D8. An 8-bit control byte received on the ctrl lane indicates control information such as Training sequence, Frame/Line/Packet start or End, etc. and also indicates how many bytes of the data lanes are to be pushed into the buffer.

The control byte on ctrl lane, is interpreted as control-code if the LS-bit is ‘1’. If the LS-bit is ‘0’, data is pushed into data buffers and the remaining bits of control byte indicate how many bytes in the link are to be pushed into data buffers.

Table 25-5 lists the control byte encoding supported by FX10 in LVDS and LVCMOS enhanced mode. The encoding given below is an example and is expected to be modified during design.

Table 25-5. ControlByteEncoding

Control-byte encoding (Most-Significant-bit is parity *p*)	Control-byte name	Parameter in Control Byte	Parameter on Data lane0	Control-byte valid in LVCMOS enhanced mode?	Available to GPIF III as input? (not available as output control)	Description
*p*xxx xxx1						Control byte, with a parity bit *p*. Odd parity is used. Parity check is performed, when enabled by EZ-USB™ FX10 CPU. Parity bit is not applicable for Training Sequence.
xxxx xxxx	TSEQ	N/A	N/A	N	Y	Training Sequence is programmable using EZ-USB™ FX10's LVDSSS_LVDS_AFEx_PHY_TRAIN_CONFIG Register. This training sequence will be expected on all Data and Control lanes, after Link-ready is communicated to LVDS-TX for TS_time (50 usecs). After TS_time, link will be trained. Parity-bit and check is not applicable for Training sequence on control lane.
*p*000 0001	IDLE	N/A	N/A	N	Y	Idle condition- data on data lanes and IDLE control bytes on control lane are dropped.
*p*000 1**tt**1	STAD[**tt**]	2-bit thread address	N/A	CPU - Configurable	Y	Set Thread Address for a link; **tt** specifies the 2-bit thread number for wide link with 16 data lanes. In narrow link mode, on Port 0, only thread 00 and 01 may be specified and on Port 1, only thread 10 and 11 may be specified. For wide-link configuration *tt* specifies thread number, which can be 00, 01, 10 or 11. The FPGA should sample the Buffer-ready and Buffer-watermark signals for the thread after 100 Fclk cycles of the LVDS interface clock

Table 25-5. ControlByteEncoding (continued)

Control-byte encoding (Most-Significant-bit is parity *p*)	Control-byte name	Parameter in Control Byte	Parameter on Data lane0	Control-byte valid in LVCMOS enhanced mode?	Available to GPIF III as input? (not available as output control)	Description
*p*001 **kkk**1	SSAD[**kkk**]	3-bit socket address bits[3:1]	N/A	CPU - Configurable	Y	Set Socket Address for current thread of the link. Current thread is 00 by default on Port 0, Port 1 and in Wide-Link mode if not previously set. The ***kkk*** specifies the socket's 3 msb bits of socket number. In narrow link, the actual socket number (3'b*kkk* x 2) + *tt[0], tt* is the current thread on Port 0 and Port 1. On Wide-link, the actual socket number is (3'b*kkk* x 2) + *tt[0] on* adapter *tt[1].* The LVDX-Tx should sample the Buffer-ready and Buffer-watermark for the thread after 100 Fclk cycles of the LVDS interface clock
*p*010 0001	SPTSSS	N/A	N/A	Y	N	Set Precision Time Stamp (PTS) snapshot in PTSSS register. A snapshot of lower 32-bits of TS counter is stored in PTSSS variable on receiving SPTSSS control-byte command
*p*010 0011	SSCRSS	N/A	N/A	Y	N	Set SCR snapshot in SCRSS.SCRSS is a 6-byte variable. Whenever SSCRSS control-byte command is received •32-bits of Time stamp (TS) counter value is stored in lower 32-bits of SCRSS register. The LVDSSS_LVDS_SCRSS_VALUE_CFG0 config register will decide which 32-bits of TS nanosecond counter has to be used: Value 0 = 31:0 – time represented will be in terms of ns Value 1 = 39:8 – time represented will be in terms of 256ns Value 2 = 41:10 – time represented is closer to usec (1024ns) •msb 11-bits/lsb 11-bits/14-bits(based on config register) of bus interval counter is stored in next 14-bits of SCRSS variable.

Table 25-5. ControlByteEncoding (continued)

Control-byte encoding (Most-Significant-bit is parity *p*)	Control-byte name	Parameter in Control Byte	Parameter on Data lane0	Control-byte valid in LVCMOS enhanced mode?	Available to GPIF III as input? (not available as output control)	Description
*p*010 01**i**1	SVARWL[i]	1-bit count i	Byte-value	Y except in 8-bit mode	N	EZ-USB™ FX10 has two 16-bit metadata variable registers for each GPIF Thread, namely Variable Word 0 (VARW0) and Variable Word 1 (VARW1). Reference to i in the control byte is be used to select between VARW0 and VARW1. Set Variable Word Lower (SVARWL) control byte is used to set LSB of Metadata Variable Word register i (LVDSSS_LVDS_THREAD0_VARW) with byte value in lane0; This can be used for specifying UVC header flag byte or any other metadata value.
*p*010 10**i**1	SVARWH[i]	1-bit count i	Byte-value	Y except in 8-bit mode	N	EZ-USB™ FX10 has two 16-bit metadata variable registers for each GPIF Thread, namely Variable Word 0 (VARW0) and Variable Word 1 (VARW1). Reference to i in the control byte is be used to select between VARW0 and VARW1. Set Variable Word Lower (SVARWL) control byte is used to set MSB of Metadata Variable Word register i (LVDSSS_LVDS_THREAD0_VARW) with byte value in lane0; This can be used for specifying UVC header flag byte or any other metadata value.
*p*011 00**i**1	UEVC[i]	1-bit count i	N/A	Y	Y	Update event count EVC[i]. Each GPIF thread has one event counter register. Event counts are visible to GPIF and CPU. Events are counted using 8-byte counters EVC[i]. As an example, this counter can be used for counting frames and the flag to indicate to CPU that a new frame is starting.
*p*011 01**i**1	CEVC[i]	1-bit count i	N/A	Y	Y	Clear event counter EVC[i].
*p*100 0**ff**1	SFLG[ff]	2-bit count f	N/A	Y	Y	Set flag FLG[f]. Each GPIF thread has 4 flags. Flags are visible to GPIF and CPU. A Maskable interrupt will be raised to CPU, whenever these flags get set or cleared by FPGA. No counters are associated with flags.
*p*100 1**ff**1	CFLG[ff]	2-bit count f	N/A	Y	Y	Clear FLG[f].

Table 25-5. ControlByteEncoding (continued)

Control-byte encoding (Most-Significant-bit is parity *p*)	Control-byte name	Parameter in Control Byte	Parameter on Data lane0	Control-byte valid in LVCMOS enhanced mode?	Available to GPIF III as input? (not available as output control)	Description
*p*101 0001	EPC	N/A	N/A	Y	Y	Enable payload count for selected thread. Indicates that data is to be counted for payload data stream and the corresponding 64-bit payload-count (PLC) register LVDSSS_LVDS_THREAD_PLC_DW0 and LVDSSS_LVDS_THREAD_PLC_DW1 is to be updated every time data is transferred to the thread.
*p*101 0011	HPC	N/A	N/A	Y	Y	Halt (or suspend) payload Count for flow . This control-byte stops updating payload count. This could be useful for not counting databytes while metdata is being transferred. After HPC, a EPC control byte has to be sent before actual data starts getting transferred, to ensure that PLC gets updated for every data transfer.
*p*101 0101	CPC	N/A	N/A	Y	Y	Clear 64-bit Payload Count for flow and halt the counting. EPC control-byte need to be sent to allow
*p*101 0111	INITDCRC	N/A	N/A	N	Y	Initialize the CRC to default value for the thread. The initial value is programmed via THREAD<N>.DCRC.INITIAL_VALUE register by EZ-USB™ FX10 CPU.

Table 25-5. ControlByteEncoding (continued)

Control-byte encoding (Most-Significant-bit is parity *p*)	Control-byte name	Parameter in Control Byte	Parameter on Data lane0	Control-byte valid in LVCMOS enhanced mode?	Available to GPIF III as input? (not available as output control)	Description
*p*101 1001	CHKDCRC	N/A	N/A	N	Y	Check Data CRC (CHKDCRC) Control byte indicates that 16-bit Link CRC is present in least significant lanes (lane 0 and lane 1). This control byte causes the computed 16-bit CRC within LVDS link layer, in the LVDSSS_LVDS_THREAD0_DCRC variable for the thread, to be checked against the CRC bytes presented on the data lanes. For a link with single data-lane, the CHKDCRC control byte will be presented twice and CRC check is done after the second CRC byte is presented. On a single data-lane configuration, the two CHKDCRC control bytes must be presented consecutively, with no other control or data bytes in between. Errors are flagged as CPU interrupts if check results in mismatch. The Data CRC is computed with all data bytes presented on the data lanes. Data bytes are identified by a 0 on lsb of control byte. Note that training sequences, training block for wide link are excluded in Data CRC computation. The Data CRC presented on the data lanes is consumed by LVDS link layer and does not get inserted into data buffers when CHKDRCR control byte(s) are received. The DCRC metadata variable may be inserted via insert metata (INMD) control, with DCRC embedded in metadata.
*p*101 1011	INITMCRC	N/A	N/A	Y	Y	Initialize 16-bit Metadata CRC (MCRC) for the thread. The initial value is programmed via THREAD<N>.MCRC.INITIAL_VALUE register by EZ-USB™ FX10 CPU. The MCRC field is updated when Metadata words are inserted on the thread's data buffer. MCRC may be used as a metadata field.
*p*101 1101	Reserved					Reserved

Table 25-5. ControlByteEncoding (continued)

Control-byte encoding (Most-Significant-bit is parity *p*)	Control-byte name	Parameter in Control Byte	Parameter on Data lane0	Control-byte valid in LVCMOS enhanced mode?	Available to GPIF III as input? (not available as output control)	Description
*p*110 0mm1	INMD[mm]	mm: 2-bit metadata to insert	N/A	Y	Y	Insert MetaData *mm* [3-0] to data stream. EZ-USB™ FX10 supports four metadata register regions per thread. FPGA can choose which metadata to be inserted by specifycing the mm bits in control byte. LVDS-TX should send 30 Idle-control bytes for Metadata insertion. This feature can be usedfor insertion of Frame/Line/Packet header/leader/footer/trailer - as needed for UVC, U3V
*p*110 1111	EOP	N/A	N/A	N	Y	End of packet (EOP) or data : This control byte can result in either of the below scenarios: If buffer is available and empty, a Zero Length Packet (ZLP) is issued If buffer is partially filled, wrap-up the buffer. To send a Zero Length Buffer (ZLB) FPGA needs to wait, by sending IDLEs on control lane until a fresh buffer and then issue a EOP. A fresh buffer is available when buffer ready is signaled by EZ-USB™ FX10.

Table 25-5. ControlByteEncoding (continued)

Control-byte encoding (Most-Significant-bit is parity *p*)	Control-byte name	Parameter in Control Byte	Parameter on Data lane0	Control-byte valid in LVCMOS enhanced mode?	Available to GPIF III as input? (not available as output control)	Description
*p*111 0011	L1ENTER	N/A	N/A	N	N	Enter L1: Control byte causes link to enter low power L1 state. All control bytes except L1EXITare ignored on control lane. L1 – A low power state in which EZ-USB™ FX10 LVDS receivers on data lanes are powered down (clock- and control-lanes continue to be active in L1). The time to enter L1 state, L1ENTER_TIME, would be around 20x interface clock cycles. The time to exit L1 and get back to L0 would be around L1EXIT_Time (expected to be less than 50 usecs). During the L1ENTER_TIME (L1 entry time) period, FPGA shall continue to transmit L1ENTER control-bytes on control lane while data and clock lane(s) continue to be active. The data received along with L1ENTER control bytes are discarded by LVDS link layer. After L1ENTER_TIME, FPGA may place the LVDS data lanes in low-power state, while continuing to transmit L1ENTERs on control lane with active clock lane.
*p*111 0101	L1EXIT	N/A	N/A	N	N	Exit L1: This control byte causes link exit L1 low power state for link.
*p*111 x111						Reserved
*p*xxx xxx1						Reserved
Xxxx xxx0						All or the indicated least significant (LS) bytes of a link gets into data buffer
*p*000 0000				N		All Data bytes gets into data buffer. These data-transfers are called full-width-transfers. Non full-width-transfers are called partial-transfers. Partial transfers can only occur as the last transfer into a buffer. An EOP is required after last partial transfer to commit the buffer, if buffer is not filled.

Table 25-5. ControlByteEncoding (continued)

Control-byte encoding (Most-Significant-bit is parity *p*)	Control-byte name	Parameter in Control Byte	Parameter on Data lane0	Control-byte valid in LVCMOS enhanced mode?	Available to GPIF III as input? (not available as output control)	Description
*p*000 0010				N		1 LS Byte - gets into data buffer. Data sent on lane 0 is sampled and gets into data buffer
*p*000 0100				N		2 LS Bytes - get into data buffer. Data sent on lane 0 and lane 1 are sampled and gets into data buffer
p000 0110				N		3 LS Bytes - get into data buffer. Data sent on lane 0, lane 1, lane 2 are sampled and gets into data buffer
*p*000 1000				N		4 LS Bytes - get into data buffer. Data sent on lane 0, lane 1, lane 2, lane 3 are sampled and gets into data buffer
*p*000 1010				N		5 LS Bytes - get into data buffer. Data sent on lane 0, lane 1, lane 2, lane 3, lane 4 are sampled and gets into data buffer
*p*000 1100				N		6 LS Bytes - get into data buffer. Data sent on lane 0, lane 1, lane 2, lane 3, lane 4, lane 5 are sampled and gets into data buffer
*p*000 1110				N		7 LS Bytes - get into data buffer. Data sent on lane 0, lane 1, lane 2, lane 3, lane 4, lane 5, lane 6 are sampled and gets into data buffer
*p*00*y yyy*0				N		(*4*'b*yyyy*) LS Bytes - gets into data buffer. Data sent on lanes 0 to lane yyyy-1 are sampled and gets into data buffer
*p*001 1110				N		15 LS Bytes - get into data buffer. Data sent on lanes 0 to lane 14 are sampled and gets into data buffer

Table 25-6. Socket thread mapping in EZ-USB™ FX10

Thread # (decimal)	Thread number (binary) STAD[tt]	Socket number SSAD[kkk]	Actual socket address = (3'bkkk x 2) + tt[0] on adapter tt[1]	DMA adapter
0	00	000	0	0
		001	2	
		010	4	
		011	6	
		100	8	
		101	10	
		110	12	
		111	14	
1	01	000	1	
		001	3	
		010	5	
		011	7	
		100	9	
		101	11	
		110	13	
		111	15	
2	10	000	16	1
		001	18	
		010	20	
		011	22	
		100	24	
		101	26	
		110	28	
		111	30	
3	11	000	17	
		001	19	
		010	21	
		011	23	
		100	25	
		101	27	
		110	29	
		111	31	

25.5 Internal pattern generator

EZ-USB™ FX10EZ-USB™ FX10 can support a internal loopback mode on its LVDS / LVCMOS link layer. In this mode one GPIF State machine will act as a receiver and another GPIF state machine will act as a transmitter. The cross connections between Port 0 and Port 1 are done at the link layer for this mode. This mode can be utilized to validate GPIF to DMA data flow without applying stimulus on external IO pins.

Control signals and data to be driven on to the link interface is controlled by loopback program stored in HBWSS SRAM. Format and fields of loopback programming stored in HBWSS SRAM is captured in **Table 25-7**.

Table 25-7. Loopback fields

Field	Width	Bit position	Comments
VLD	1	0	Indicates that loopback program entry in HBWSS SRAM is valid
START	1	1	Indicates start of loopback program. This should be the first line of loopback program. When this is set, only DATA_MODE field is valid.
END	1	2	Last entry of loopback program
Reserved	1	3	Reserved
DATA_MODE	4	7:4	This field is valid only when START field is set. This indicates the interface mode of this loopback program. 3'b000 – lvds mode 3'b010 – lvcmos sdr mode 3'b011 – lvcmos ddr mode 3'b100 – enhanced lvcmos sdr mode 3'b101 – enhanced lvcmos ddr mode
REPEAT_CNT	12	19:8	Decides for how many interface clock cycles, the control and data read from current instruction has to be repeated. Fixed to '1'.
DATA_SRC	2	21:20	This field is relevant only when REPEAT_CNT is greater than '1'. 2'b00 – repeat data of current instruction for REPEAT_CNT times 2'b01 – increment LSB 8-bits of data read from current instruction, every cycle by 1 and apply on interface in next cycle. Repeat this for REPEAT_CNT cycles. Here same 8-bits of data is applied on all valid data bus width.
Reserved	10	31:22	Reserved
CTRL_BYTE	8	39:32	Control byte to be applied on link layer interface. Valid only for LVDS or enhanced LVCMOS mode
Reserved	4	43:40	Reserved
CTRL	20	63:44	Control signals to be applied on link layer interface. Out of 20-bits, LSB 10-bits are for port-0 and MSB 10-bits are for port-1. In case of wide link, all 20-bits are meant for port-0 only.
Data	64	127:64	Maximum 64-bits of data can be stored in a single instruction, which is sufficient for a narrow port. In case of wide link, 128-bits of data is needed, hence same 64-bit data is repeated 2x and applied on link layer data interface. If bus width is <64-bits, then only lsb 'bus width' bits of 64-bit are used and rest of data bits are ignored.

127 65	64	63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44	43	42	41	40	39 38 37 36 35 34 33 32	31	30	29	28	27	26	25	24	23	22	21 20	19 18 17 16 15 14 13 12 11 10 9 8	7 6 5 4	3	2	1	0
Data		CTRL	x	x	x	x	CTRL_BYTE	x	x	x	x	x	x	x	x	x	x	DATA_SRC	REPEAT_CNT	DATA_MODE	x	END	START	VLD

Figure 25-8. Loopback format

- Each instruction of loopback program is of 128-bits or 16-bytes
- Loopback program shall be stored in HBWSS SRAM
- CPU populates SRAM with required loopback program
- CPU programs any relevant configuration registers, GPIF FSMs and triggers loopback testing
- Thread 1 or Thread 3 shall execute loopback program. Hence in loopback mode thread 1 and 3 can't be used by GPIF FSM

There are different combinations of link layer loopback

- Port-0 under test
  - CPU shall setup Thread 3/Socket 31 as consumer socket and associate loopback program to this socket
  - GPIF-1 FSM will be setup to read loopback program and apply loopback stimulus to port-0 link layer interface
  - GPIF-0 FSM will be setup to work on loopback stimulus and push data to producer socket/descriptor/buffer
- Port-1 under test
  - CPU shall setup Thread 1/Socket 15 as consumer socket and associate loopback program to this socket
  - GPIF-0 FSM will be setup to read loopback program and apply loopback stimulus to port-1 link layer interface
  - GPIF-1 FSM will be setup to work on loopback stimulus and push data to producer socket/descriptor/buffer
- Loopback mode is not supported in Wide link mode.

25.5.1 Programming flow

The `fx3g2_uvc` and `fx3g2_u3v` code examples in the SDK supports streaming a fixed color bar pattern using the internal pattern generator. LVDS_LB_EN makefile argument can be used to enable this feature.

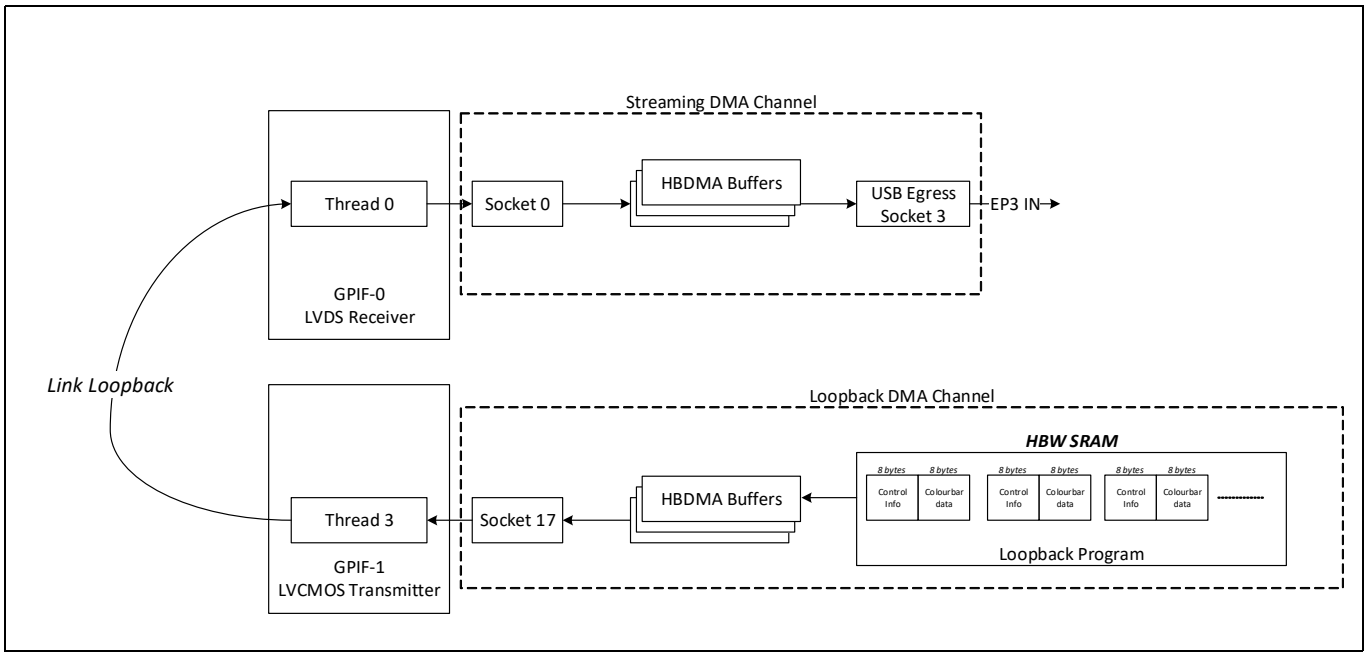


Figure 25-9. Internal pattern generator example implementation

LVDS/LVCMOS IO subsystem

- GPIF-0 is configured as a LVDS receiver and GPIF-1 is configured as a LVCMOS transmitter
- Two DMA channels are created - one from high-bandwidth (HBW) SRAM to Socket 17 of GPIF 1 and another from Socket-0 to USB IN Endpoint#3's socket
- When the host requests for data from the device (for example, UVC Commit Control request), the device populates the required pattern data in the HBW SRAM in the form of multiple loopback program register entries
  - Each loopback program register entry is of 16 bytes, where lower 8 bytes store the control information such as Start, End, Control Byte and so on and the upper 8 bytes store pattern data
  - For a UYVY 3840 × 2160 color bar frame that has 8 bands, each color will be 3840/8=480 pixels wide i.e. 960 bytes. Each loopback program register entry can hold 8 bytes of pattern data. As a result, each color will have 960/8=120 entries
  - The loopback program also includes IDLE cycles and other control bytes parameters
  - The Loopback program entry for multiple lines of the pattern are populated in the HBW SRAM and committed to the GPIF-1 socket. This process is repeated for every consumer event received from the Loopback DMA channel
- The pattern data is forwarded to the GPIF-0's streaming DMA channel by the link loopback feature of EZ-USB™ FX10

25.6 Clocks

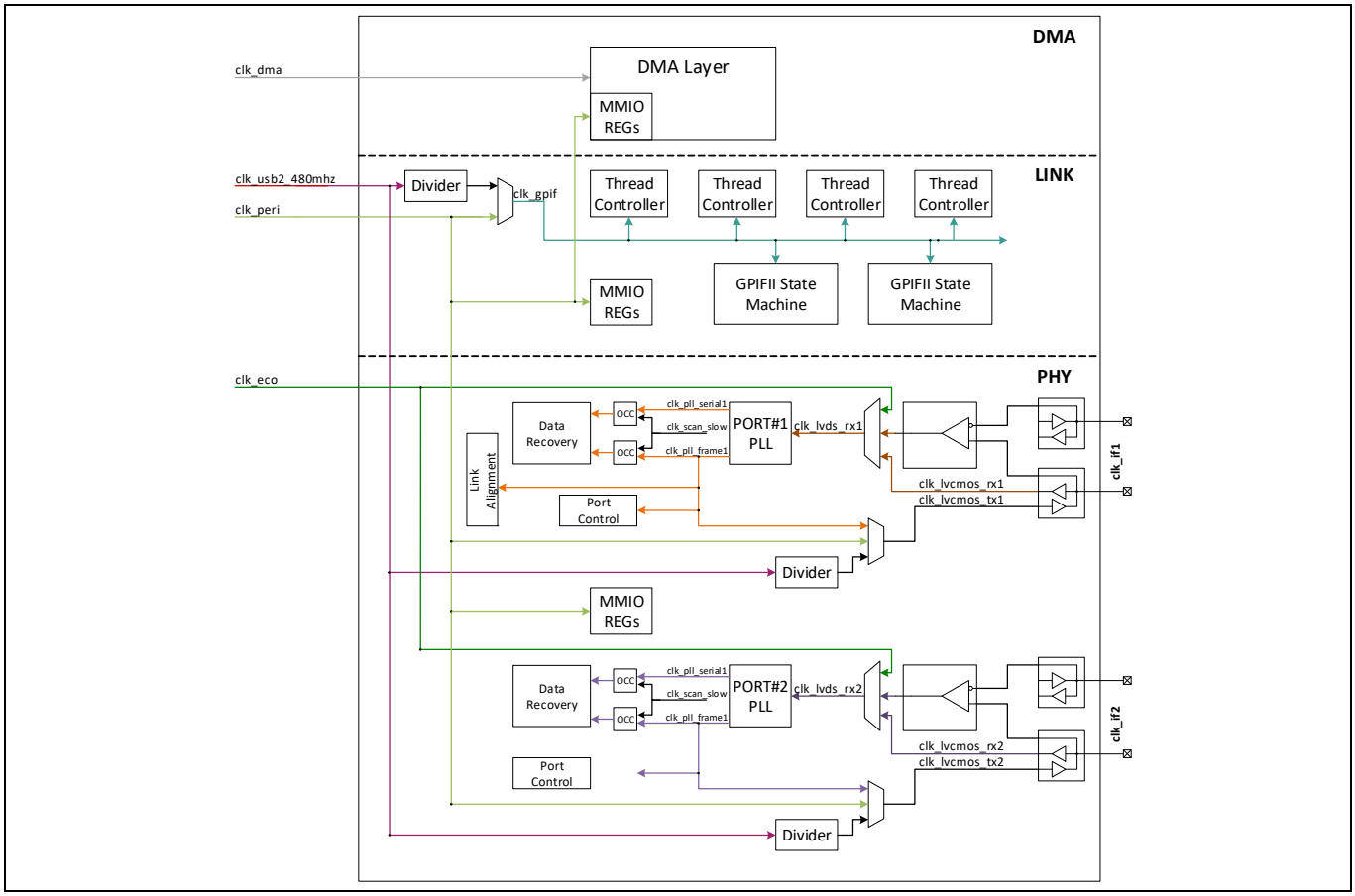


Figure 25-10. LVDS/LVCMOS interface clocks

DMA: The AXI Interface operates on the DMA clock and is used as the clock for the GPIF Interface. The clk_dma could be 480 MHz or 240 MHz.

MMIO: The AHB interface operates on the clk_peri at 100 MHz.

GPIF: The GPIF clock is generated from 2 clock sources. The first source is CLK_PERI at 100 MHz and can be used for lower bandwidth links. The second source is the 480 MHz USB2 clock. A divider is used to generate 240 MHz, 160 MHz clocks for high-speed LVDS links.

PHY: The input clock on the link interface is either a direct LVCMOS input for LVCMOS link or is a differential input for an LVDS link as shown in **Figure 25-10**.

LVDS operation frequencies and gearing ratios supported on EZ-USB™ FX10 are provided in **Table 25-8**.

Table 25-8. LVDS clock frequencies, ranges, gearing and bit rates

Row#	Clock frequency (MHz)	Gearing	Max bandwidth per lane (Mbps)
1	625	1:1	625
2	625	2:1	1250
3	156.25	1:1	156.25
4	156.25	2:1	312.5
5	156.25	4:1	625
6	156.25	8:1	1250
7	148.5	1:1	148.5
8	148.5	2:1	297
9	148.5	4:1	594
10	148.5	8:1	1188
11	74.25	2:1	149
12	74.25	4:1	298
13	74.25	8:1	596

LVCMOS operation modes and frequencies are documented in **Table 25-9**.

Table 25-9. LVCMOS clock frequencies, ranges and bit rates

Max Clk (MHz)	DDR / SDR	Direction	Data to clock alignment	Max bandwidth for 16-bit / 32-bit bus (Mbps)
160	DDR	Input	Edge	5,120 / 10,240
160	SDR	Input	Positive edge	2560 / 5,120
100	SDR	Output	positive edge	1,600 / 3,200
80	DDR	Output	Edge	2560 / 5120

25.7 Power modes

LVDS-TX can enter Low power mode to save power when there is no data to transmit. LVDS/ LVCMOS block has two ports. Each port can be in different power modes independent of the other port.

Table 25-10 lists the available power modes for the LVDS / LVCMOS interface.

Table 25-10. LVDS power modes

Power mode	Description
L0-Active	This mode corresponds to the active data transmission mode. The LVDS clock is running and the PLL is active. The system power mode must be in Active for this IP to be in L0 power mode. In this active state, when FPGA has no valid data to send, FPGA shall send IDLE control bytes and data lanes can carry any valid data. The data on data lanes will be discarded by LVDS port when control lane has IDLE control byte. In LVCMOS, FPGA uses one or more of the control pins (depending on interface protocol used by application) to indicate presence of valid data on data lanes.
L1-Clock off	This mode is LVDS-Tx initiated low power mode. It is entered by stopping the LVDS clock. This mode does not involve the firmware. The link needs to be trained again once the clock is restarted by LVDS-Tx. The system power mode must be in Active for this IP to be in L1 power mode. The time to enter L1 state, L1ENTER_TIME would be around 200x interface cycles. The time to exit L1 and get back to L0 would be around L1EXIT_Time (expected to be less than 50 µs). L1 mode is supported only in LVDS mode. No need to support L1 low power state in LVCMOS mode, as it does not result in any significant power savings.
L3-IO disabled	This mode is a firmware initiated low power mode. The LVDS/LVCMOS interface senses the idle state of the link and the signals to firmware, via interrupt, which then disables the LVDS IOs and link-not-ready state is indicated to LVDS-Tx by the firmware via I2C, SPI or control signals over LVCMOS control interface. The always-on logic monitors activity on clock lane and signals wakeup to the firmware, which then enables the IOs and exits this power mode. The time to enter L3 state and L3ENTER_TIME would be around 200x interface cycles (time taken for HW to disable LVDS IOs). The time to exit L3 and get back to L0 would be around L3EXIT_Time (expected to be 5 ms). FPGA uses a LINKx_GOL3 control to signal entry and exit to and from L3 state. LINKx_GOL3 will be one of LVCMOS control pins (default is P0LVCMOSIO3 for port 0, wideport. P1LVCMOSIO3 for port 1) associated with each Port and will be a selectable. This GPIO will be capable of waking EZ-USB™ FX10 from Deep Sleep. LINKx_GOL3 can also be any one of the generic GPIOs of EZ-USB™ FX10. The system may enter Deep Sleep power mode when this block is in L3 power mode.

The LVDS-TX and EZ-USB™ FX10 need to follow certain sequence of steps, before entering low power modes, in order not to damage the LVDS IOs. EZ-USB™ FX10 LVDS IOs should be tristated before LVDS-TX IOs are turned-off. The below sections describe the sequence to be followed for each of these power mode entry/exits.

25.7.1 Entry to L1 from L0

1. Link is in L0 state.
2. FPGA may choose go to low power L1 state, by transmitting L1ENTER control-byte. This causes the EZ-USB™ FX10 to initiate entry to L1.
3. Link layer signals this event to CPU with an interrupt.
4. EZ-USB™ FX10 turns off its LVDS Data Lane IOs
5. During the L1ENTER_TIME (L1 entry time) period, which could take up to 200x interface cycles, FPGA shall continue to transmit L1ENTER control-bytes on control lane while data and clock lane(s) continue to be active. The data received along with L1ENTER control bytes are discarded by LVDS link layer.
6. After L1ENTER_TIME, FPGA may place the LVDS data lanes in low-power state, while continuing to transmit L1ENTERs on control lane with active clock lane.
7. There is no any feedback from EZ-USB™ FX10 to FPGA, to indicate that data lanes are turned OFF. It is time based, where 300 ns (L1ENTER_TIME) is needed for EZ-USB™ FX10 to enter L1 power mode, after receiving the L1ENTER command.
8. L1 power mode entry is taken care by hardware and no firmware involvement is needed.

25.7.2 Exit from L1 to L0

1. FPGA shall send a L1EXIT control-byte on the control lane. The LVDS link layer provides an interrupt to firmware when it detects L1EXIT control byte.
2. EZ-USB™ FX10 is configured by firmware to re-initialize PHY training.
3. FPGA shall send PHY Training Sequence for TS_Time (50 µs) on control and data lanes while keeping the clock lanes active. When the LVDS PHY has done training it provides training-done interrupt to firmware.
4. FPGA shall follow PHY training sequence, immediately with link training block on control and data lanes. An interrupt is provided to CPU if link training block detection is failed/passed by link layer.
5. After sending the Training Sequence (or IDLEs), FPGA may resume sending valid data or other valid control bytes.
6. Along with L1EXIT command, FPGA should start driving LVDS data lanes too. By doing so, it is ensured that EZ-USB™ FX10EZ-USB™ FX10 IO inputs are not in HiZ, when EZ-USB™ FX10 IOs are turned ON, because of L1EXIT command. LVDS IO circuit can get damaged if its inputs are floating (or HiZ), when IOs are active.

25.7.3 Entry to L3 to L0

1. When there is no valid data to transmit FPGA transmits IDLE control bytes on control lane and data (that is discarded by EZ-USB™ FX10) on all configured data-lanes on the link, along with active clock lane(s).
2. FPGA may choose to go to low power L3 state, by asserting LINKx_GOL3. For wide link configuration LINK0_GOL3 is used.
3. EZ-USB™ FX10 disables its LVDS IOs, on receiving LINKx_GOL3 and raises LINKx_L3_ENTRY interrupt to CPU.
4. EZ-USB™ FX10 firmware does below steps on seeing LINKx_L3_ENTRY interrupt asserted.
   a) Disables all thread's by programming GPIF_THREAD_CONFIG.ENABLE_THREAD_CTLR to '0'
   b) Suspend the sockets
   c) Disables DLL first, followed by PLL
   d) Asserts active reset
5. During the L3ENTER_TIME (L3 entry time), FPGA shall continue to transmit IDLE control-bytes on control lane while data and clock lane(s) continue to be active.
6. After L3ENTER_TIME, FPGA can place the LVDS data, control and clock lanes in low-power state or power them down.
7. FPGA may maintain this L3 state indefinitely.

Even in case of LVCMOS mode too, procedure to enter L3 is same as LVDS mode.

Note that there is no feedback from EZ-USB™ FX10 to FPGA, to indicate LVDS/LVCMOS block has entered L3 low power mode.

25.7.4 Procedure to exit L3 and enter L0

1. FPGA shall decide to exit L3 and deasserts LINKx_GOL3.
2. Before de-asserting LINKx_GOL3, FPGA activates the clock lane(s) and sends PHY Training Sequence for L3EXIT_Time (expected to be around 5 milliseconds) on data and control lanes.
3. LINKx_GOL3 deassert causes EZ-USB™ FX10 to get out of Deep Sleep, and asserts LINKx_L3_EXIT interrupt.
4. EZ-USB™ FX10 firmware does below steps on seeing LINKx_L3_EXIT interrupt
   a) Deassert active reset
   b) Enable socket
   c) Enable thread
   d) Enable clocks (enabling PLL and DLL)
   e) Re-initiates link training
   f) Enables the GPIF_WAVEFORM_SWITCH.SWITCH_NOW
5. When the LVDS PHY has done training it provides training-done interrupt to firmware.
6. FPGA shall send a link training block following phy training sequence. An interrupt is provided to CPU if link training block detection is failed/passed by link layer.
7. After the sending the Training Sequence, FPGA shall resume sending valid data or other valid control bytes.

In case of LVCMOS mode too, L3 exit sequence is same as in LVDS mode, except for the absence of PHY training sequence.

26 GPIF III interface

The GPIF III is a programmable state machine that enables a flexible interface that may function as a master or slave in industry-standard or proprietary interfaces. Only parallel interfaces may be implemented with GPIF III.

Following are the GPIF III features:

- Provides 256 firmware programmable states
- Supports 8-bit, 16-bit, 24-bit and 32-bit parallel data bus
- Supports two independent instances of GPIF III state machines each connected to a link or single GPIF III state machines connected to wide link

The GPIF III state transitions are based on control input signals. The control output signals are driven as a result of the GPIF III state transitions. See the GPIF III Designer tool documentation for more details. The GPIF III state machine's behavior is defined by a GPIF III descriptor. The GPIF III descriptor is designed such that the required interface specifications are met. 8 KB of memory (separate from the 128 KB data and 256/512/1024 KB buffer SRAM) is dedicated to the GPIF III waveform where the GPIF III descriptor is stored in a specific format.

EZ-USB™ FX10 can support up to two independent GPIF III interfaces in its LVDS/LVCMOS Subsystem as shown in **Figure 26-1**.

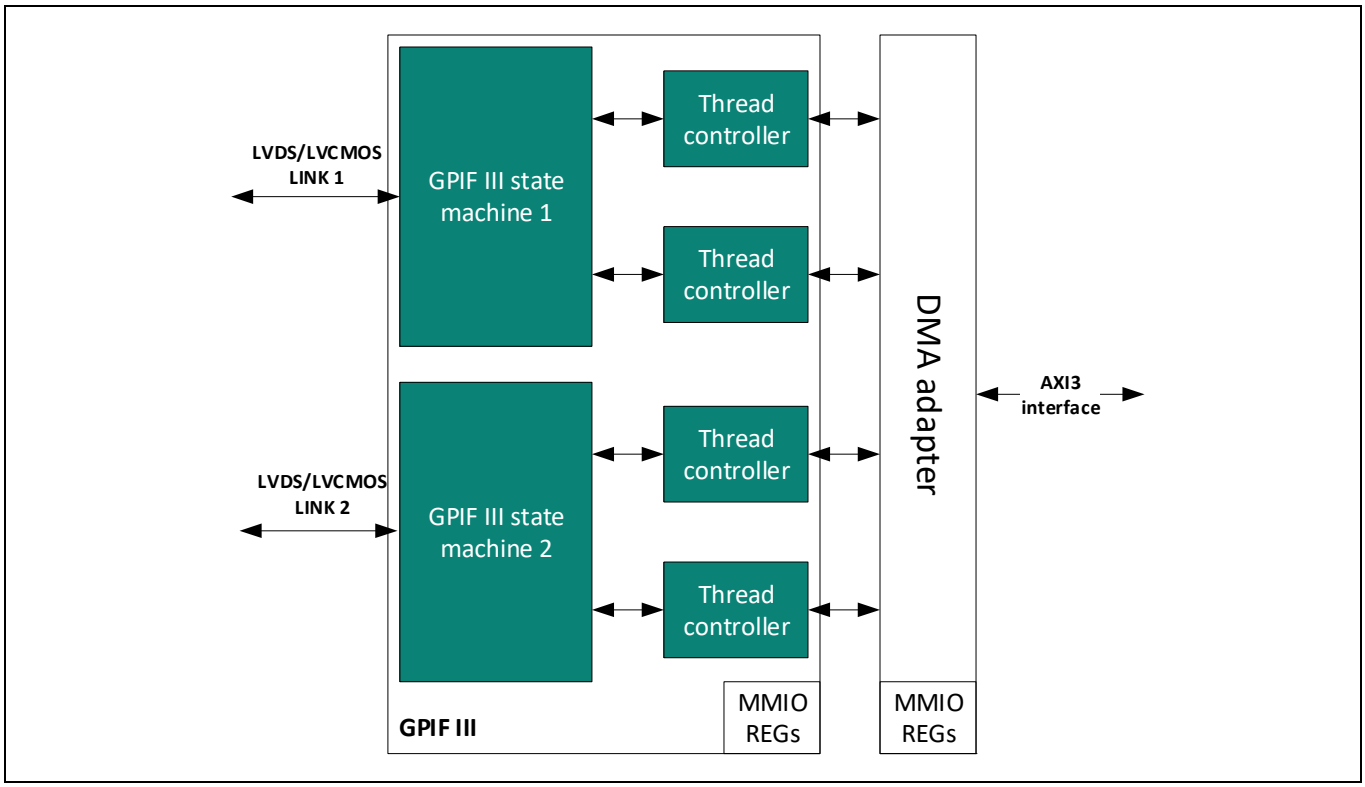


Figure 26-1. GPIF III interface block diagram

26.1 Typical GPIF III interfaces

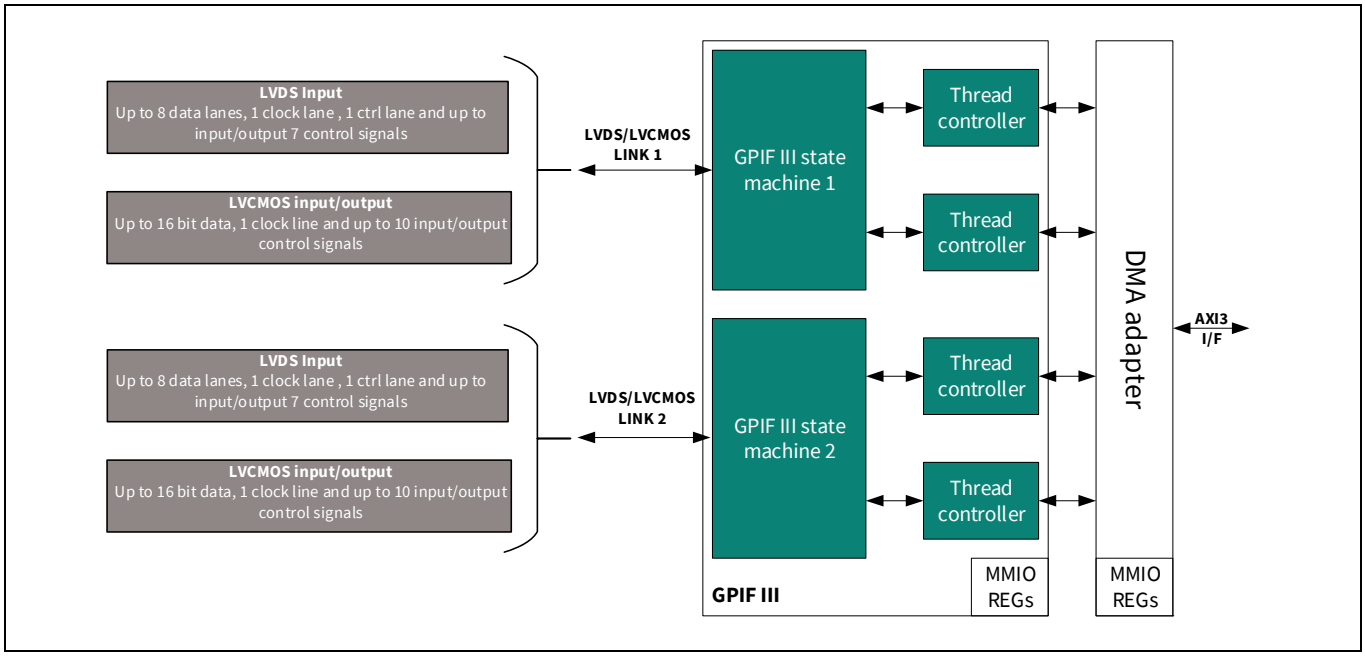


Figure 26-2. Dual link interface

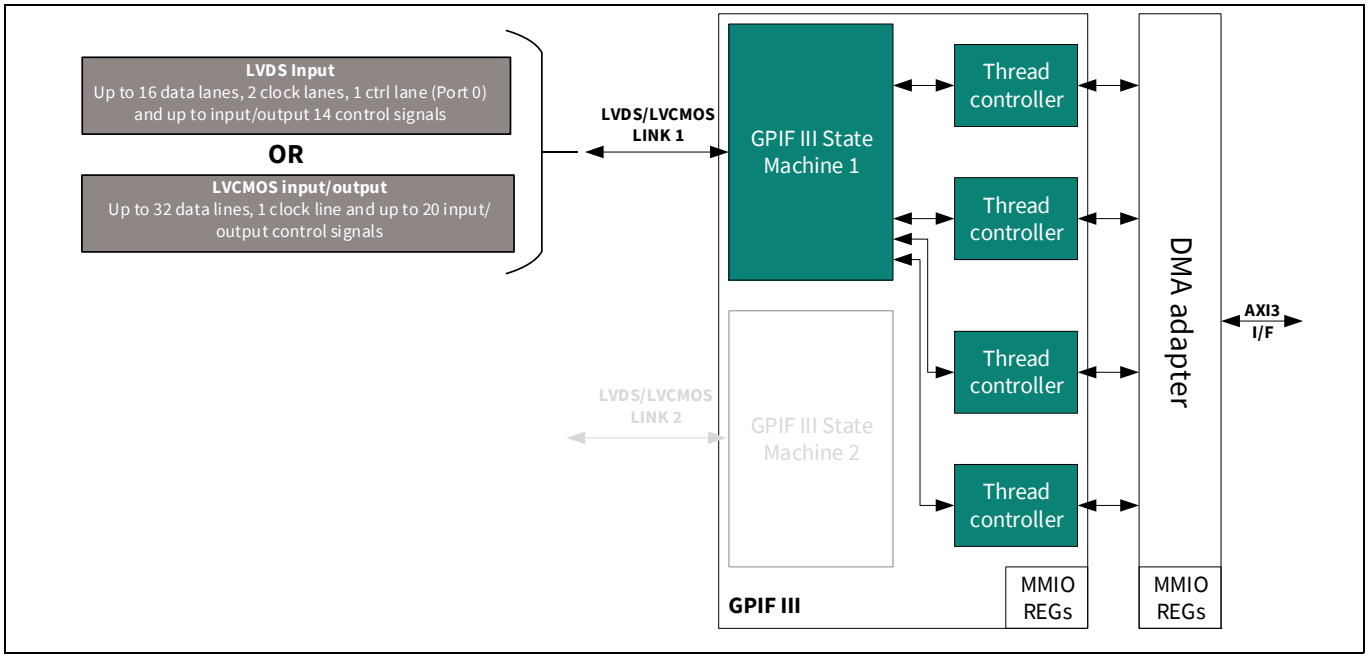


Figure 26-3. Wide link interface

26.2 Initialization and configuration of GPIF III block

The GPIF III block is configured by writing to a set of device registers and configuration memories. The GPIF III Designer utility generates the configuration data corresponding to the communication protocol to be implemented based on the graphical state machine. This tool generates the GPIF III configuration information in the form of a C header file that defines a set of data structures. The EZ-USB™ FX10 software development ecosystem and the GPIF III Designer installation also include a set of header files that have the configuration data for a number of commonly used protocols.

The CyU3PGpifLoad() API is used by the firmware application to load the configuration generated by the GPIF III Designer tool. This API, in turn, internally uses the CyU3PGpifWaveformLoad(), CyU3PGpifInitTransFunctions(), and CyU3PGpifConfigure() APIs to complete the configuration. These APIs can be used directly as a replacement for the CyU3PGpifLoad() call with the constraint that the CyU3PGpifConfigure() call should be made after the CyU3PGpifWaveformLoad() and CyU3PGpifInitTransFunctions() calls.

26.2.1 GPIF III state machine control

The firmware application may require the GPIF III state machine to be started and stopped multiple times during run time. In most cases where the EZ-USB™ FX10 device is functioning as a slave device, there will be a single GPIF III state machine, and the application needs to start it as soon as the configuration has been loaded. The CyU3PGpifSMStart() API can be used to start the state machine operation in this case.

If the application uses EZ-USB™ FX10 as a GPIF III master, there may be multiple disjointed state machines that implement different parts of the communication protocol. The CyU3PGpifSMSwitch() API can be used to switch between different state machines in this case.

If the GPIF III state machine operation is to be suspended and later resumed, the CyU3PGpifSMControl() API can be used to pause or resume state machine operation.

If the state machine operation is to be stopped at some point and restarted from the reset state later, the CyU3PGpifDisable() API can be used with the forceReload parameter set to CyFalse(0). The CyU3PGpifSMStart() API can then be used to restart the state machine.

If the active GPIF III configuration is to be changed, the CyU3PGpifDisable() API can be used with the forceReload parameter set to CyTrue(1). The CyU3PGpifLoad() API can then be used to load a fresh configuration.

26.2.2 LVDS flow control

The flow control will be available for each LVDS link. The flow control (buffer-ready) signal assertion by this block indicates that the LVDS-TX side can transmit a packet without interruption. For fixed size data packets, the packet size shall be pre-negotiated between this block and LVDS-TX over I2C/SPI or other interfaces.

The buffer-ready will be de-asserted before end of the current packet if buffer is not available.

LVDS-TX shall have the capability to send short packets and zero-length packets by sending suitable sequences on the control lane.

26.2.3 Known limitations

26.2.3.1 General

There is one-cycle delay between consumption of data as dictated by the update_dout and production of data as governed by read_pop. This means that a pre-fetch buffer of at least two samples is needed.

Once ready, the socket along with associated prefetch buffer must remain ready throughout the transfer. If this cannot be guaranteed, the state-machine becomes complex and latency increases as states need to split so that each state has only two branches.

26.2.3.2 Asynchronous protocols

The asynchronous protocols need at least one sampling event during the set-up + hold window of data and address. At 200 MHz, this means that the set-up + hold time has to be greater than 5 ns.

Minimum delay from OE to data valid is explained in the timing scheme and is 22.5 ns with all guard-bands.

26.2.3.3 Synchronous protocols

The synchronous protocols that expect next clock cycle response must ensure that no-more than four states are reachable from a given state. If five transitions are possible from a state, three transition will be made in one clock cycle, however the other two transitions will require two clock cycles to complete. During these two clock cycles, if the inputs change again, GPIFII may exhibit behavior that is not compliant with the protocol under consideration.

The synchronous protocols register their inputs signals once to break timing arch. One more cycle is needed to push read-data into output flop. Therefore, protocols need a minimum read latency of two cycles. It is possible to burst data afterwards continuously.

26.2.3.4 Slave FIFO

Since slave FIFO is likely to be the most important interface and since there is a large legacy of use, this interface is treated separately. Registers were added to provide FX3 like interface. GPIF_THREAD_CONFIG register provides facility to define watermarks.

The GPIF has four threads and partial and non partial (full or empty) flags for the active sockets in each thread are available simultaneously, any combination of these flags can be routed to any control pin through GPIF_CTRL_BUS_SELECT register.

The overall scheme of slave FIFO interface is as follows:

- The GPIF state machine is responsible for interpreting addresses to provide index in the indexed mode. The GPIF state machine is also responsible for all the data movement.
- The hardware is responsible for selecting various flags according to the index selection provided by the state machine and driving them to the output.

The following limitations exist:

- Due to delays in buffer switching a notion of burst size is introduced and for transfers that are less than a buffer size, partial flags are rounded to the nearest burst size. However, in the current design, in PP mode 0 (slave FIFO mode) partial flags are not rounded to nearest burst size. For partial-flags to be effectively used buffer size and data should be constrained to be multiples of burst-size. Alternate ways of implementing flow control is described in **“Flow control design considerations for master side of a Slave FIFO interface”** on page 389.
- Non-partial flags lag by the number of clock cycles required in the synchronization logic. So, it is recommended to use partial flags that match the total round-trip latency of the system.
- When switching to a non-active socket the interface will indicate empty until an empty buffer for write or buffer with data is available for read.

26.2.3.5 Flow control design considerations for master side of a Slave FIFO interface

The master side of the slave-FIFO should take care of the following:

- The programmable partial-flag provides an early indication of the end of buffer or valid data. Master should use the partial flag to flow control data transfer. The full/empty flag indicates availability of buffer space or data in the buffer. Master should use partial-flag to pause transfers and should use full/empty flag to start or resume transfers. Master should program the partial-flag to deassert system latency cycles (L) before full/empty flag. The value L should be programmed in the GPIF_THREAD_CONFIG[thread#].WATERMARK register.
- Master should select burst size in cycles as 2^n where n is in range [x - 15]; where 2^x> L. Typical value for x is 4, implying a burst of 16 cycles.
- The master should truncate the burst a fixed number of cycles after master detects partial-flag. The fixed number of extra access (K) in the truncated burst should be a value that exactly empties/fills the buffer. K is a constant and depends on the programmed watermark and the system latency. Alternatively, master may truncate the burst a fixed number of cycles after master detects partial-flag and switch to single cycle accesses until full/empty flag deasserts. This will ensure that all bytes of a short packet are read out without buffer underrun. During write this mechanism prevents buffer overflow.
- For read transfers after a buffer switch to a buffer having byte count less than (watermark cycles * interface-width in bytes), the partial-flag will remain deasserted while full/empty flag asserted. In this case master should do single cycle access to empty the read buffer until full/empty flag deasserts.
- When switching sockets the full/empty flag will transition from deassert to assert when the addressed socket is ready for transfer. If buffer is not ready at the addressed socket full/empty flag will remain deasserted. The partial flag should be checked only when full/empty flag is asserted.
- During writes, when a zero length or partial buffer length packet is written to the buffer the master full/empty flag indicates when the next empty buffer is available. The partial flag should be checked only when full/empty flag is asserted.
- During reads the zero length buffers (buffer carrying a ZLP without any valid data bytes) will be consumed by the socket and not indicated to the master.
- If master does not have the capability to truncate the bursts, higher level applications should ensure that data moved on the slave-FIFO interface is always a multiple of burst sizes. The buffer-sizes should be multiple of burst sizes.

27 High-bandwidth DMA subsystem

High-Bandwidth DMA sub system supports up to 20 Gbps interconnect between USB3.2 device and LVDS/LVCMOS Interface. The main features of this sub-system are:

- DMA Interconnect capable of 20 Gbps throughput
- 64-bit wide AXI3 at 240 MHz
- 32-bit wide AHB at 100 MHz
  - Distributed multi-channel DMA
- Supports multiple simultaneous streams
  - Allows data transfer to other lower bandwidth peripherals in the chip
  - Includes a 1 MB SRAM operating at 160 MHz
  - Provides data store for the USB3.2 and LVDS/LVCMOS subsystem

The DMA descriptor is the unit of transfer over the interconnect

- DMA descriptor transfers in one AXI3 Interconnect burst
- Provides storage for DMA descriptors
- SRAM bandwidth shared between AXI3 (High BW) and AHB (System Access)
- Supports retention during Deep Sleep

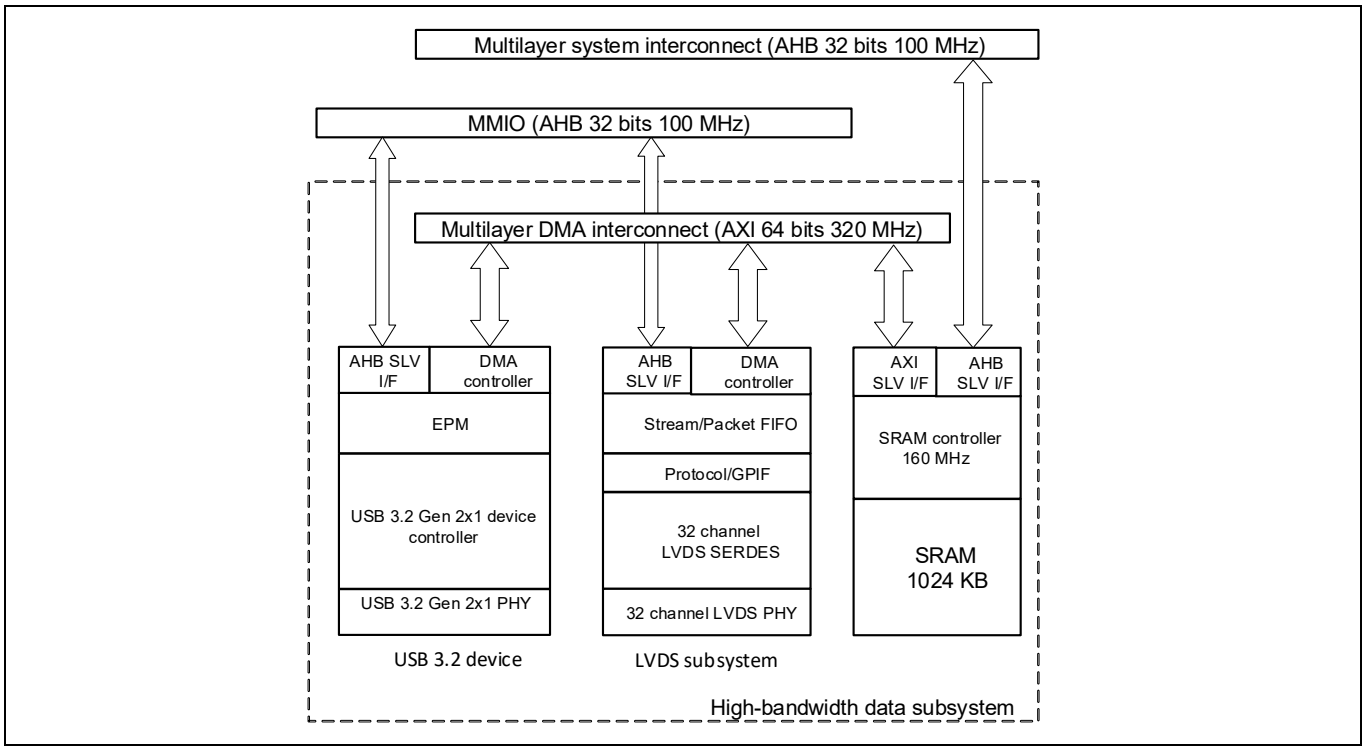


Figure 27-1. Block diagram

High-bandwidth DMA subsystem

The high-bandwidth DMA sub system integrates the USB 3.2 device controller and the LVDS/LVCMOS subsystem through the high-bandwidth AXI Interconnect.

The subsystem includes a 1 MB SRAM that acts as the data storage for the high-bandwidth subsystem. The high-bandwidth data between the USB 3.2 and LVDS/LVCMOS subsystem is transferred through the SRAM. The SRAM provides the storage to account for USB 3.2 host latencies. The data transfer towards other lower bandwidth IPs also happens from this SRAM. This SRAM is exposed to the external DMA controller on the system-interconnect to facilitate this data transfer. The SRAM controller implements an arbiter between the transfers from the high-bandwidth sources and the external DMA controller.

This DMA sub-system is the high-bandwidth interconnect that facilitates the 20Gbps data throughput between the embedded SRAM and the LVDS/LVCMOS or USB3.2 controllers. This interconnect is based on the AXI3 address-merge bus and can run at 320-MHz and is 64-bit wide.

In addition to the high-bandwidth DMA, the EZ-USB™ FX10 also supports the standard CPUSS including the DataWire and DMA controllers. The CPUSS DW/DMA controllers will be used for all DMA transfers from/to the blocks outside the HBWSS (i.e., SCB, CANFD, MXPDM, I2S, SMIF, Crypto, and so on) as well as the USB HS Device block.

The CPUSS DMA can perform transfers between all peripherals and memories connected to the System Interconnect AHB. As the 1 MB SRAM inside the HBWSS is also accessible on the system level AHB, the CPUSS DMA controller can be used to transfer data between this SRAM and outside peripherals. However, the CPUSS DMA does not have access to the distributed DMA controller interfaces in the USB 3.2 and LVDS blocks and hence cannot transfer data from/to these blocks directly.

The multi-level DMA controller inside the HBWSS does not have access to the system AHB and hence can only handle transfers within the HBWSS itself (between the USB 3.2 device controller, the LVDS subsystem and the 1 MB SRAM).

The CPUSS DMA controller has a centralized architecture and supports peripheral to peripheral, peripheral to memory, memory to peripheral and memory to memory data transfers outside the HBWSS. The controller can be configured to support a device-specific number of DMA channels and executes these transfers based on priority levels set by the firmware.

The CPUSS supports Data Wire (DW) functionality which is meant for low latency transfers across a large number of channels and DMA functionality which is meant for high memory bandwidth for a small number of channels.

27.1 Operating modes

27.1.1 Functional mode

The high-bandwidth DMA sub system follows the modes of the embedded USB 3.2 and LVDS/LVCMOS subsystems. The sub system is clock gated when active until the embedded blocks request for data transfer.

The power state of the HBWSS is determined by the USB and LVDS blocks. The embedded USB 3.2 and LVDS blocks are Deep Sleep peripherals. The system can choose to enter Deep Sleep when there no data to transfer and the USB 3.2 block requests low power entry. The embedded SRAM in the sub-system can be configured to retain in Deep Sleep mode. The retention can be only restricted to the descriptor memory. **Table 27-1** documents the supported power modes.

27.1.2 Power modes

Table 27-1. Power modes

Power mode	Description
Active	All the logic is powered on and all the clocks are running. Clocks gated when no data transfer is Active.
DeepSleep	Only Deep Sleep logic is ON. All high-speed clocks are off. Some memories and logic have retention.

27.2 DMA concepts

A DMA transfer can be any one of the following:

A transfer or a sequence of transfers:

1. to memory from a peripheral.
2. from memory to a peripheral.
3. from producer block to consumer block through memory.

In case of EZ-USB™ FX10, the following data flows are possible:

1. High-bandwidth (10 Gbps) data flow between USB32 and LVDS IP.
2. Medium-bandwidth (up to 80 Mbps) data flow between the HBWSS and the rest of the system.
3. Low-bandwidth (up to 1 Mbps) data flow between the HBWSS and the rest of the system.

The data movement within the HBWSS is completely managed by the DMA controllers embedded in the USB 3.2 and LVDS/LVCMOS blocks.

Table 27-2. Possible data flows

Data flow	Source controller	Destination controller
USB 10 Gbps to LVDS	USB 10 Gbps controller moves data from USB 10 Gbps to SRAM	LVDS controller moves data from SRAM to LVDS.
LVDS to USB 10 Gbps	LVDS controller moves data from LVDS to SRAM	USB 10 Gbps controller moves data from SRAM to USB 10 Gbps block.
USB10 Gbps to external peripheral	USB 10 Gbps block controller moves data from USB 10 Gbps block to SRAM	The external DMA controller moves data from SRAM to the external component.
External to USB 10 Gbps	The external DMA controller moves data from external component to SRAM	USB 10 Gbps controller moves data from SRAM to USB 10 Gbps block.
LVDS to external	LVDS controller moves data from LVDS to SRAM	The external DMA controller moves data from SRAM to the external component.
External to LVDS	The external DMA controller moves data from external component to SRAM	LVDS controller moves data from SRAM to LVDS

27.2.1 Possible data flow in EZ-USB™ FX10

- **LVDS to USB 3.2 device:** In the EZ-USB™ FX10 video applications, video data will be received through the LVDS interface and transferred to the USB host controller through a bulk or isochronous endpoint. Performing these transfers at expected speeds up to 10 Gbps is the most important function in EZ-USB™ FX10 applications.

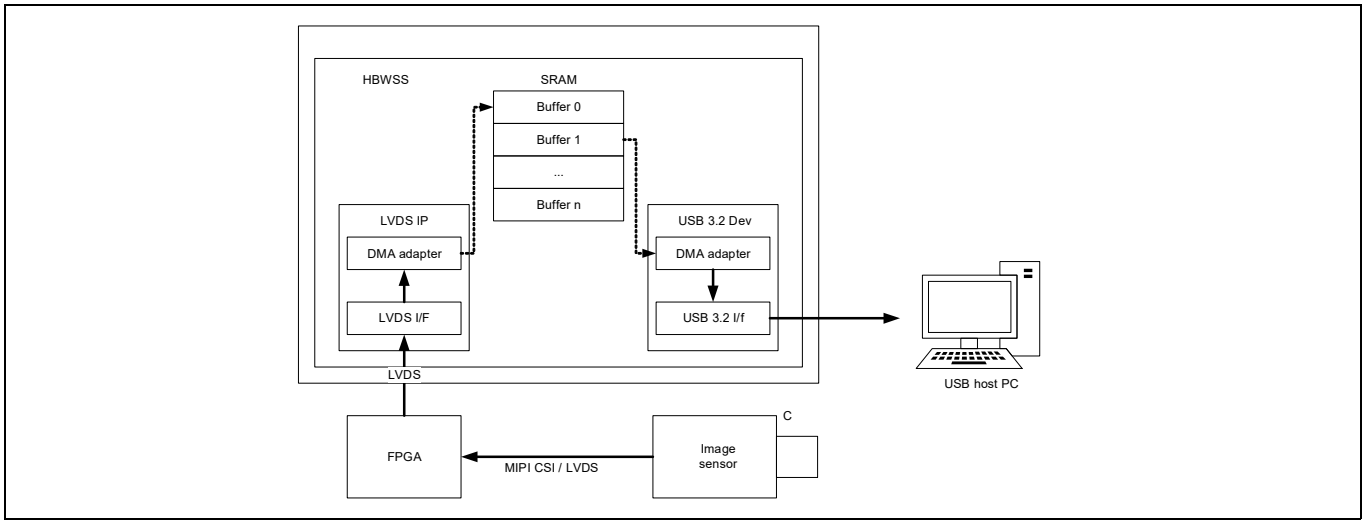


Figure 27-2. LVDS to USB 3.2 transfer using HBWSS DMA

- **USB 3.2 to LVCMOS transfer:** The LVDS sub-system on the EZ-USB™ FX10 device can be configured to operate in a single ended LVCMOS mode as well. While only data input is supported in the LVDS use case, both data input and output are supported when this block is used in LVCMOS mode. Such an implementation can be used to exchange data between a USB host controller and an FPGA/processor connected to the EZ-USB™ FX10. HBWSS DMA can be used in the case to move data received from the USB host controller through a bulk OUT endpoint to the FPGA or processor.

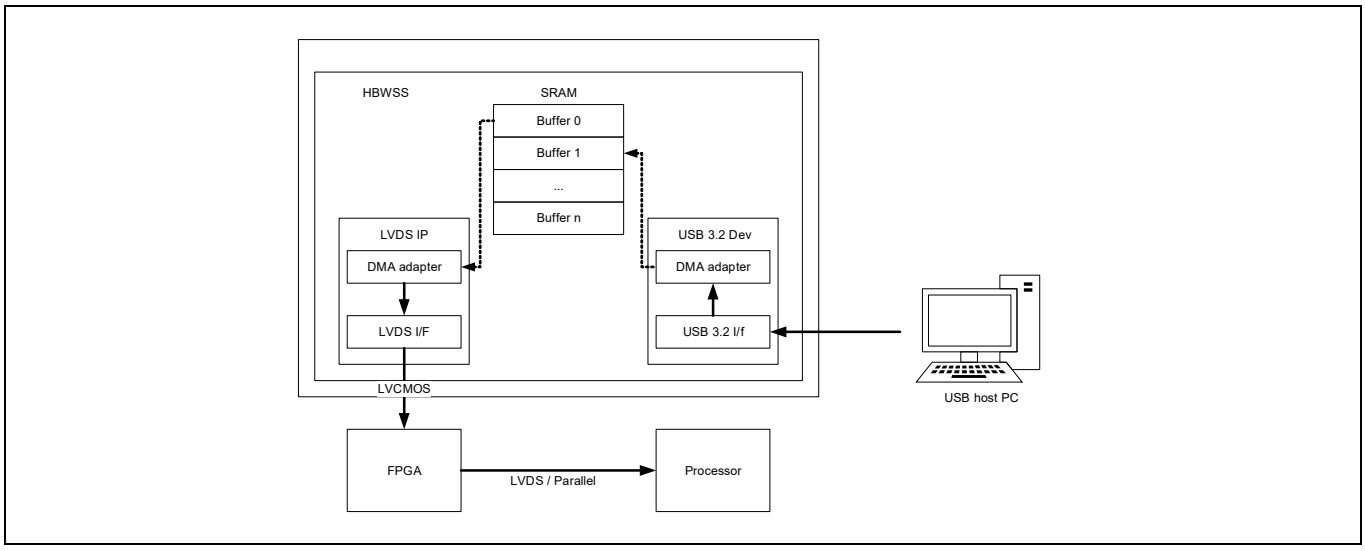


Figure 27-3. USB 3.2 to LVCMOS transfer using HBWSS DMA

- **FLASH/RAM to USB 3.2 transfer:** When EZ-USB™ FX10 is connected to a USB host controller, it needs to respond to a set of descriptor queries which help to identify the type of USB peripheral implemented. As the response to these descriptor queries is typically stored as constant arrays in the device firmware, handling them requires copying the data from the internal flash to the USB 3.2 device IP.

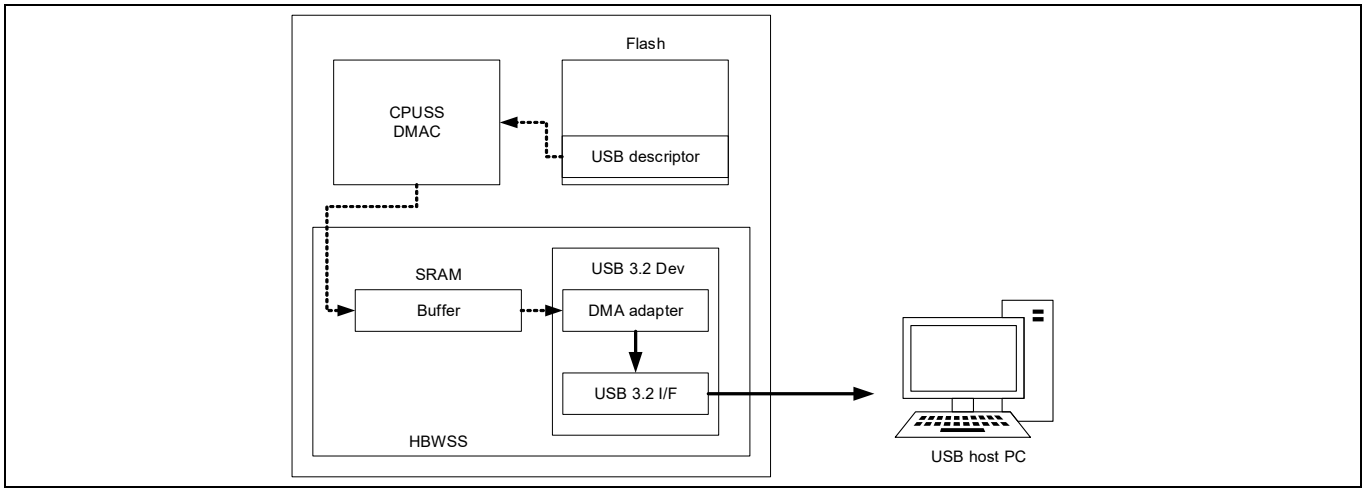


Figure 27-4. Flash to USB 3.2 transfer using CPUSS and HBWSS DMA

Since HBWSS DMA can only handle transfers from the HBWSS SRAM, the data needs to copied to a temporary buffer in this SRAM first. This transfer can be done by the CPU manually copying the data or using the CPUSS DMA controller. **Figure 27-4** shows the case where the data is first copied into the HBWSS SRAM using CPUSS DMAC and then transferred to the USB host using HBWSS DMA. Since the amount of data to be transferred on this path is limited and CPU intervention is needed in any case to parse the request and identify the appropriate response, it is acceptable to have firmware involvement to handle these data transfers.

- **PDM/I2S to USB 3.2 transfer:** Some EZ-USB™ FX10 application require the transfer of audio data in addition to the video stream coming from an image sensor. The audio data will be received by the EZ-USB™ FX10 device through peripherals like PDM or I2S and needs to be transferred to the USB Host Controller through an Isochronous IN endpoint. As shown in **Figure 1-1**, the PDM and I2S peripherals are connected to the System AHB and the USB 3.2 peripheral is connected to the HBWSS AXI bus. This means that the data needs to be copied from the PDM/I2S peripheral to the HBWSS SRAM using CPUSS DMA and then from there to the host controller using HBWSS DMA.

  Figure 27-5 illustrates the data flow for a transfer from a microphone to the USB host controller through the MXPDM and USB 3.2 device IP blocks. The CPUSS DMA is used to copy the data from the MXPDM receive FIFO to the HBWSS SRAM. The HBWSS DMA is then used to transfer the data from the SRAM to the USB endpoint. The same flow applies for other peripherals such as I2S and SCB.

  Firmware intervention is required to trigger the HBWSS DMA once per buffer based on transfer completion interrupts received from the CPUSS DMAC. As the effective transfer rate is only in the range of a few Mbps and DMA transfer buffer sizes in the order of a few KB can be used, the frequency of CPU interrupts will be less than once per ms, which can be easily handled.

High-bandwidth DMA subsystem

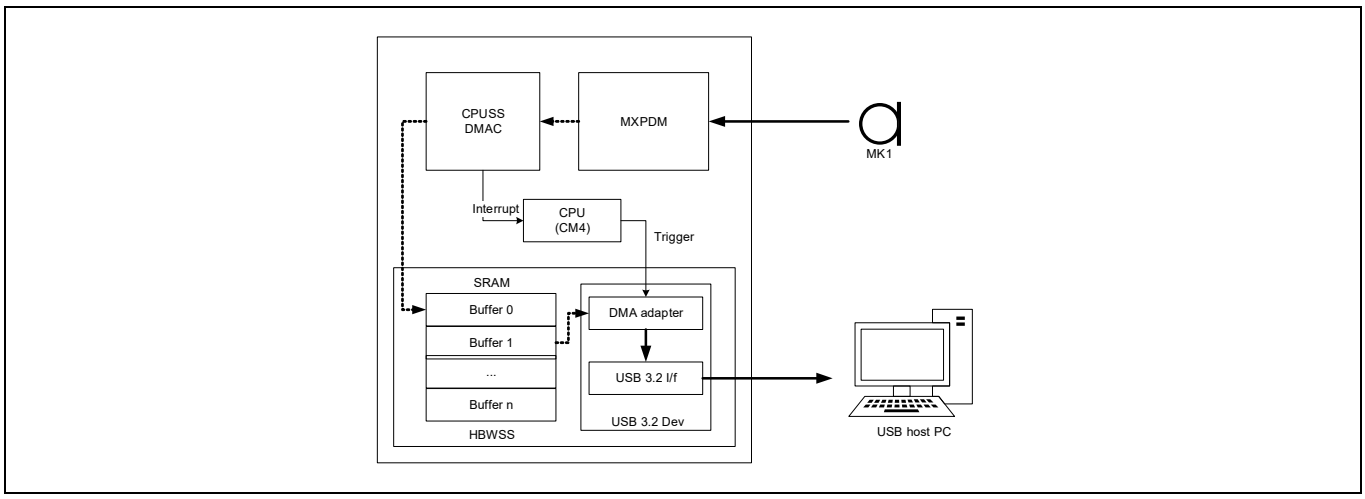


Figure 27-5. PDM to USB 3.2 transfer

- **LVDS to USB 2.0 device:** In this case, the data is received from the LVDS IP and copied to the HBWSS SRAM by the HBWSS DMA. It is then copied from the SRAM to the USB 2.0 device IP by the CPUSS DMA.

  Figure 27-6 illustrates the data flow for this use case. Since both HBWSS and CPUSS DMA is involved in the data path, firmware intervention is required once per buffer of data transferred. The CPU will be interrupted once a buffer has been filled with data received from the LVDS interface and will need to enable the CPUSS DMA channel. The CPUSS DMA will be programmed to use the trigger from the USB 2.0 endpoint memory block so that a complete data buffer can be transferred without additional firmware intervention.

  The target data rate for this data path is of the order of 50 Mbps. Hence, DMA data buffer sizes greater than 50 KB need to be used to ensure that the frequency of interrupts to the CPU is kept in the order of once per ms. DMA buffer size of 64 KB is recommended for use in this case.

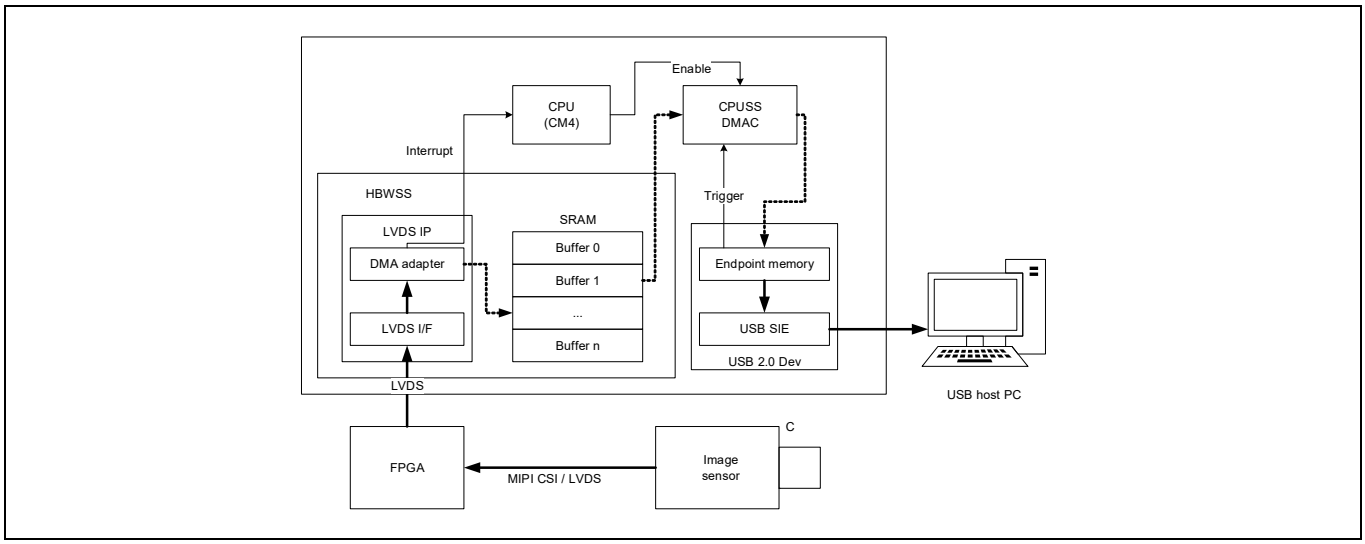


Figure 27-6. LVDS to USB 2.0 transfer using HBWSS and CPUSS DMA

- **USB 2.0 device to LVCMOS transfer:** This case is similar to the LVDS to USB 2.0 device case described above, except that the direction of transfer is reversed. Data will be received from the USB endpoint and copied into the HBWSS SRAM buffers by the CPUSS DMA. The trigger from the USB endpoint memory will be used to initiate the transfer and CPU will be interrupted once a data buffer has been filled (transfer count for the descriptor has been completed). The CPU will then send a trigger to the HBWSS DMA which will copy the data from the SRAM buffer to the LVCMOS interface.

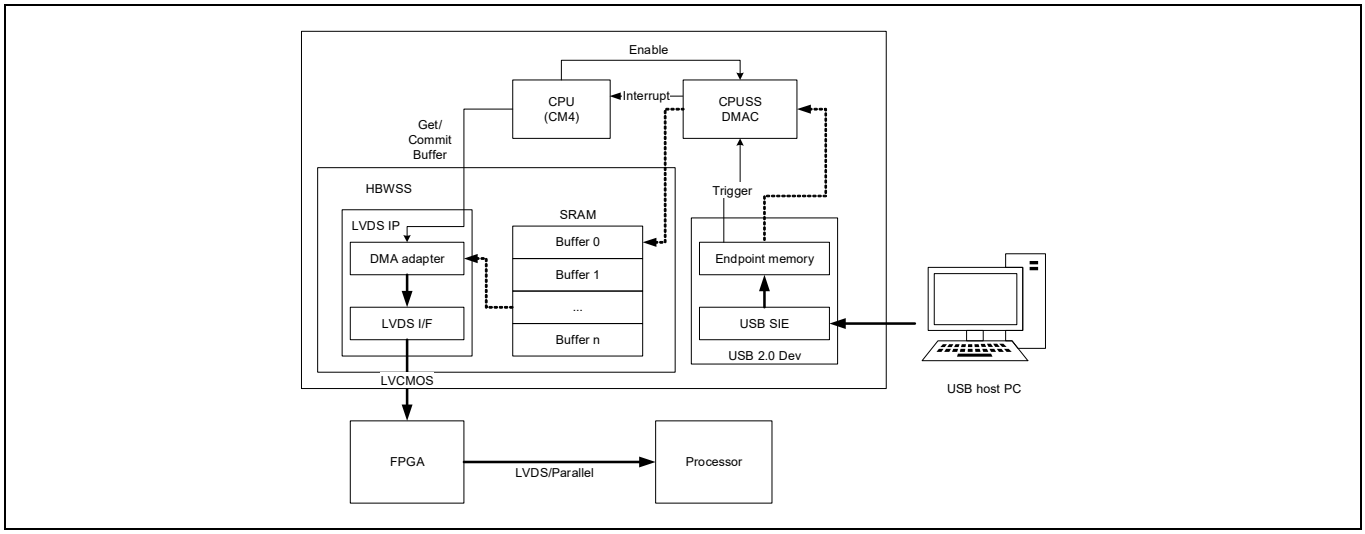


Figure 27-7. USB 2.0 to LVCMOS transfer using CPUSS and HBWSS DMA

- **FLASH/RAM to USB 2.0 device:** This case is similar to the FLASH/RAM to USB 3.2 case, except that the CPUSS DMA shall be used for transfer instead of the HBWSS DMA. Since the CPUSS DMA is capable of sourcing the data from any memory region, the data can be transferred directly from the flash without copying it into the RAM.

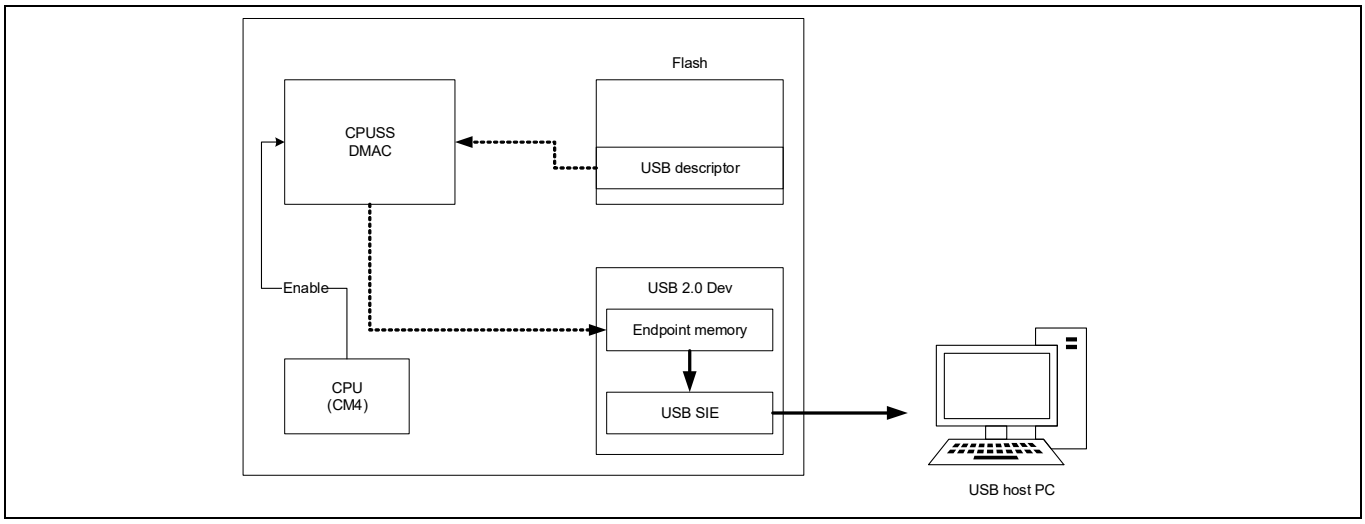


Figure 27-8. FLASH to USB 2.0 transfer using CPUSS DMA

- **I2S/PDM to USB 2.0 device:** This case is a simple application of peripheral-to-peripheral data transfer using the CPUSS DMA. While this can be implemented as a direct peripheral to peripheral data transfer, it is preferable to implement this as a peripheral to memory transfer followed by a memory to peripheral transfer as shown in **Figure 27-9**.

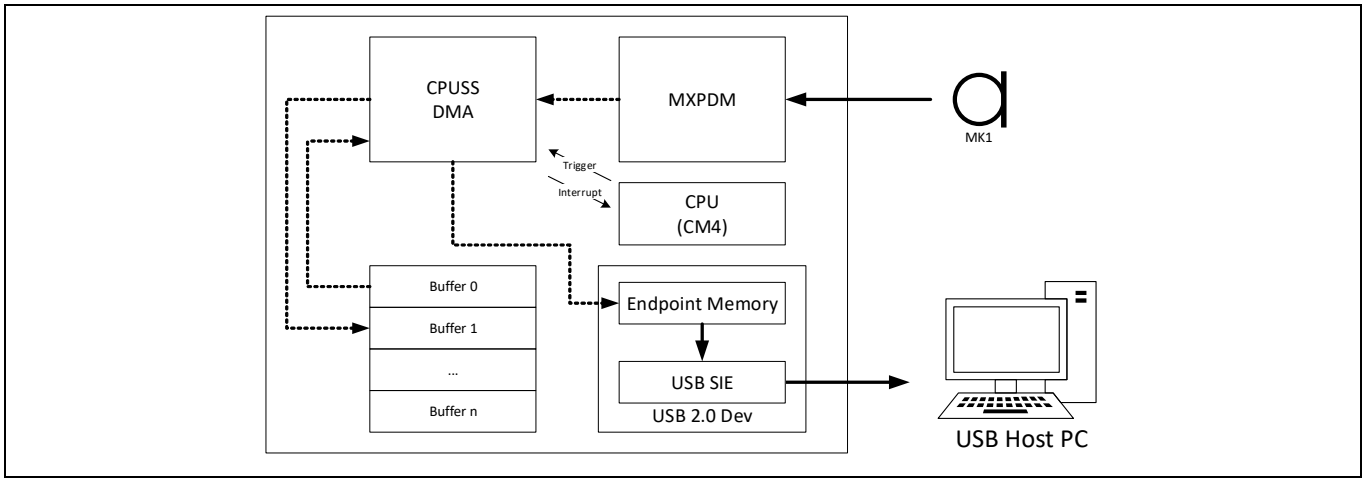


Figure 27-9. PDM to USB 2.0 device transfer

27.2.2 Decentralized DMA

The HBWSS DMA mechanism is implemented without a central DMA controller to allow for reuse of software for existing FX3 customers.

As the name indicates, these devices uses a distributed DMA architecture where the responsibility of moving data in to or out from a peripheral remained with a DMA adapter block associated with the peripheral. Since the data transfers are performed by the peripherals directly, there is no support for memory to memory transfers.

This block is designed to support packet-based transfers for USB or LVDS applications. As the USB and LVDS interfaces are asynchronous and cannot handle delays in the middle of a packet of data, direct peripheral to peripheral data transfers are not used. Instead all data transfers are routed through the 1 MB System RAM inside the HBWSS. The data is copied one packet at a time from the source peripheral to the memory buffer and from there to the sink peripheral.

The architecture supports direct trigger signaling between the DMA adapters in the USB and LVDS peripherals and hence does not require firmware intervention at the packet or transfer level.

27.2.3 Sockets, descriptors, buffers and packets

DMA is implemented through the socket concept. The following properties hold:

- A socket (**Figure 27-10**) represents either the consuming or the producing half of a transfer
- Each DMA transfer involves 1 or 2 sockets. For a transfer from one peripheral block to another two sockets are involved
- A peripheral can have multiple sockets. For example, for the USB 3.2 block, each endpoint direction is associated with one socket
- A socket is either a consuming or a producing socket at any point in time. Not both at the same time
- A socket is associated with a list of tasks called descriptors (**Figure 27-10**)
  - This list can be either circular or of finite length. The CPU can extend this list on demand. Descriptor lists can be statically defined and allocated or setup on demand for each transfer.
  - The same descriptor list can be used by a producing and a consuming socket. This is not mandatory, but required to synchronize automatically between a producing and consuming peripheral. Conversely, a single descriptor can appear in at most two lists - one consuming and one producing socket's list. It is possible, however, for a producing or consuming list to appear in more than one socket (explained in detail later).
- Descriptors are associated with buffers(**Figure 27-10**) and have a producer, a consumer or both or neither. A descriptor without a producer and consumer is called free. Synchronization between producers and consumers happens at the level of descriptors and buffers.
- Buffers can have any size starting from 16 bytes up to 65535 bytes (64 KB - 1) and are required to be aligned to 32-byte boundaries to maximize throughput. Typically, they are statically allocated for each use-case (or even at boot), but this is not mandated.
  - A packet may be smaller than one buffer or even of zero size. A packet may be split over multiple buffers. There can be multiple packets in one buffer.
  - Descriptors contain meta-data about the data in the buffer (in addition to location and size). In particular the number of valid bytes.
- Multiple descriptors may refer to the same buffer. When more than one descriptor is used to describe the same buffer, it is the responsibility of software to make sure that the descriptors are coherent and synchronized with respect to each other.
- There are a fixed number of fixed-size descriptors in a fixed location in the embedded SRAM. Because of this, hardware can directly access a descriptor given its descriptor number. Descriptor numbers are 16-bit, allowing for up to 64K descriptors to exist. The number of descriptors is limited by the number of buffers that can be dedicated in the 1 MB SRAM.
- The default state of the socket is disabled and holds no descriptor

- Sockets can stall when waiting for data to consume or for free space in memory to produce into. They do this independent of other sockets in the same peripheral and independent of the peer socket it is synchronized to. Typically, this occurs when synchronizing to another peripheral that is slower. Other sockets on the same peripheral may continue to transfer data. Using the stall mechanism for a socket only makes sense if the peripheral using the socket is flow controlled at least at buffer/descriptor granularity. If a stalling condition occurs on a socket connected to a peripheral without flow control, this will cause an underflow/overflow error in the peripheral (the socket will still stall of course).
- Sockets will automatically suspend when they have completed their descriptor list. This state typically leads to a software interrupt. It is a software task to extend the descriptor list and resume or to terminate the suspended socket.
- Sockets can also automatically suspend when they have completed their transaction count (measured in either bytes or packets). This state also typically leads to a software interrupt. It is a software task to re-up the transaction size and resume or to terminate the suspended socket.
- Sockets can also be suspended by the peripheral, typically because of an error condition that requires CPU intervention. This state also typically leads to a software interrupt. It is a software task to either correct the error condition and resume or terminate the socket.
- Sockets can only be terminated by software (see above points). The peripheral block and DMA adapter can only suspend the socket and indicate the reason why. This guarantees the CPU the opportunity to inspect the status of the active descriptor and reason for the suspend state. Software can then decide to either resume or terminate.

These concepts are depicted in **Figure 27-10** and **Figure 27-11**.

Figure 27-10 shows the concepts of socket, descriptor and buffer.

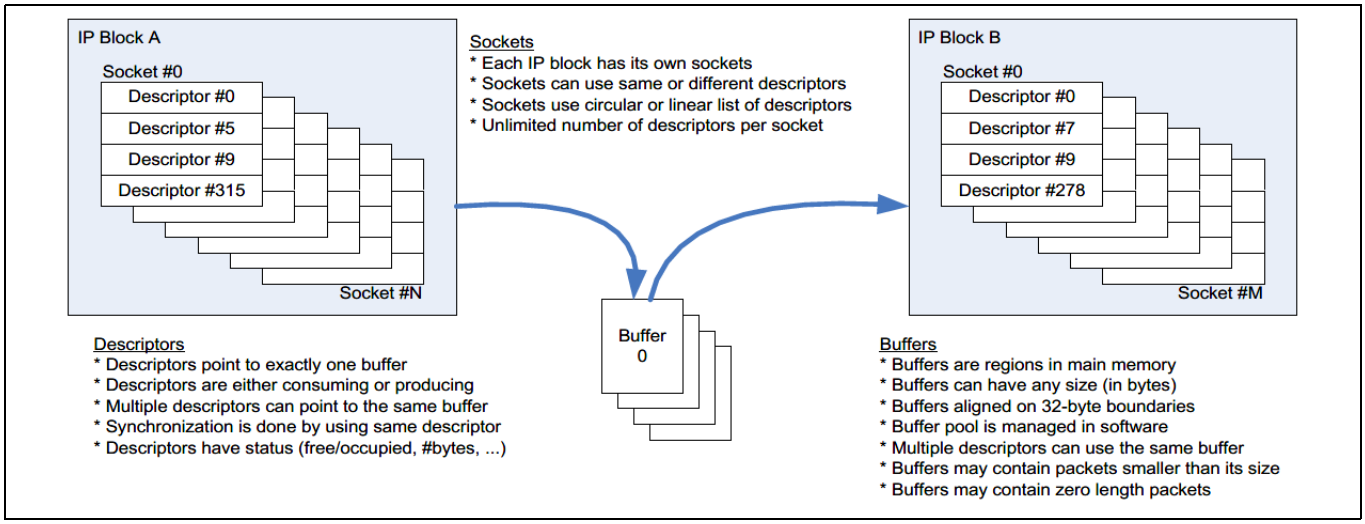


Figure 27-10. Sockets, descriptors, buffers

Figure 27-11 indicates where descriptors are stored and introduces the concept of an active descriptor. Descriptors contain information about the buffer.

High-bandwidth DMA subsystem

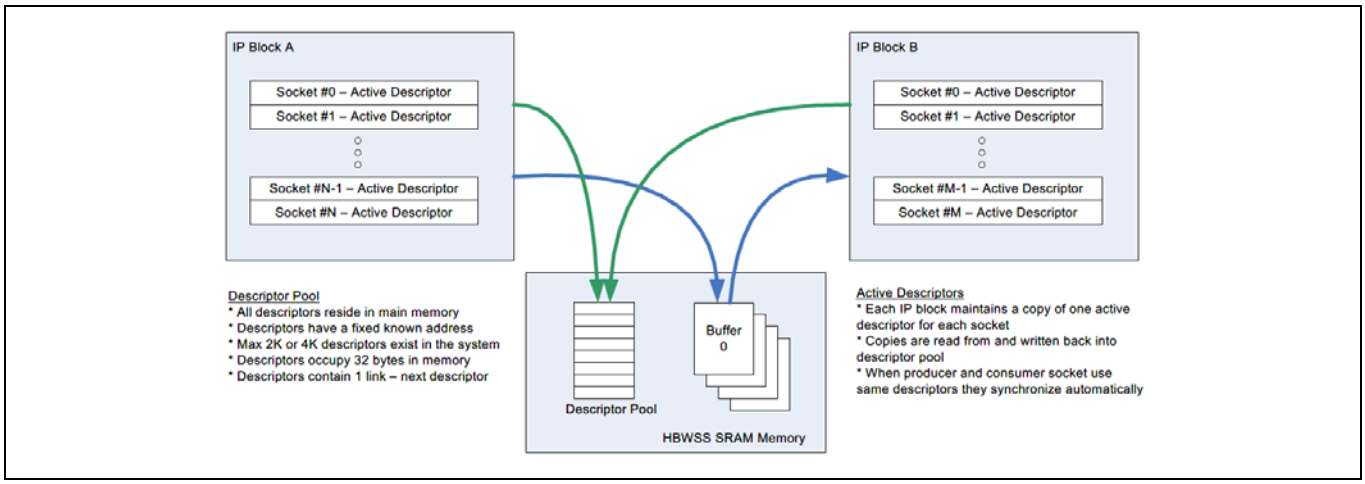


Figure 27-11. Active descriptors

27.2.4 Socket operation

The state diagram for a socket is shown in **Figure 27-12**. The socket implements a synchronization mechanism to allow for 'stalled' sockets to be woken up by a buffer produce event or consume event in the partner-peripheral. This synchronization is done with hardware triggers.

The behavior of a producing socket is as follows:

1. When a producer has filled a buffer (an end-of-packet or buffer-full occurred - note that packets can be empty), it updates the descriptor associated with the buffer to indicate the buffer is full (DMA Adapter -to- memory write transaction).
2. The producer then generates a produce event by asserting the corresponding output trigger associated with this socket. Trigger routing between the producer and the consumer socket is setup by firmware through the trigger multiplexer.
3. The producer then loads the next descriptor for the socket and makes it active (DMA Adapter -to- memory read transaction).
4. If the descriptor indicates its buffer space is occupied, the socket stalls else the producer starts filling the buffer with data.
5. If a producer is stalled, it waits for a consume event for further execution.
6. A consume event from the socket corresponding to the stalled descriptor is indicated as the assertion of the corresponding input trigger associated with this socket. Trigger routing between the producer and the consumer socket is setup by firmware through the trigger multiplexer. The descriptor is reloaded on receiving this trigger and data write transfers into the buffer may resume.
7. If no descriptor is available (next_dscr=0xFFFF) the socket suspends. When software extends the descriptor list and explicitly 'resumes' the IP block operation continues.

The behavior of the consumer socket is exactly symmetric to the producer as follows:

1. When a consumer has emptied a buffer (an end-of-packet or buffer- empty occurred), it updates the descriptor associated with the buffer to indicate the buffer is empty and available (DMA Adapter -to- memory write transaction).
2. The consumer then generates a consume event by asserting the corresponding output trigger associated with this socket. Trigger routing between the producer and the consumer socket is setup by firmware through the trigger multiplexer.
3. The consumer then loads the next descriptor for the socket and makes it active (DMA adapter -to- memory read transaction).
4. If the descriptor indicates its buffer is empty, the socket stalls else the socket starts reading from the buffer
5. If the consumer is stalled it waits for a produce event for further execution.
6. A produce event from the socket related to the stalled descriptor is indicated as the assertion of the corresponding input trigger associated with this socket. Trigger routing between the producer and the consumer socket is setup by firmware through the trigger multiplexer. The descriptor is reloaded on receiving this trigger and data read transfers from the buffer may resume.
7. If no descriptor is available the socket suspends. When software extends the descriptor list and explicitly 'resumes' the peripheral block operation continues.

The state diagram for a socket is shown in **Figure 27-12** (this diagram is identical for consumer and producer sockets).

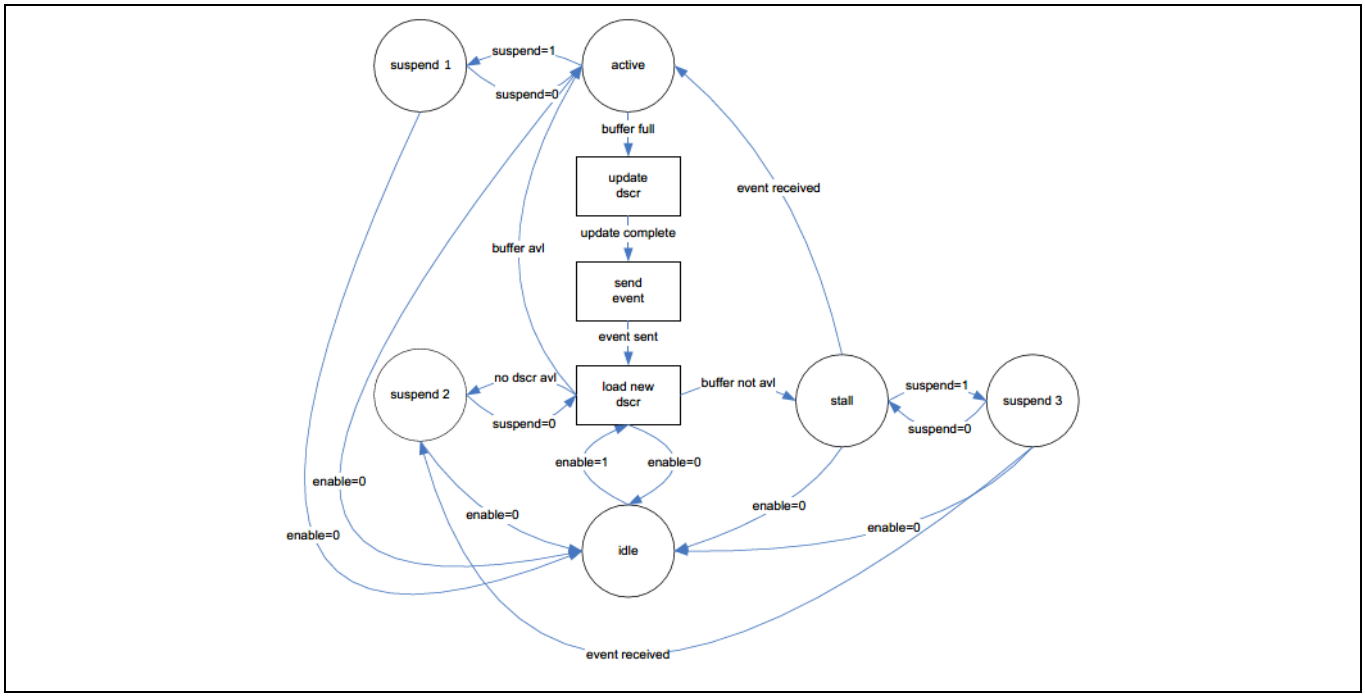


Figure 27-12. Socket diagram

27.2.4.1 Multiple sockets using the same descriptor chain

In addition to a producer and consumer of data using the same descriptor to synchronize their traffic, there are cases where multiple sockets use the same descriptor or descriptor chain. In these cases the sockets are of the same direction (producing or consuming) and must be in the same DMA Adapter. Note that, as discussed earlier, a descriptor can participate in two lists only: one consuming and one producing list.

These uses include, but are not limited to the following examples,

- ECC correction of data: One socket writes the data, but that data needs to be corrected before it is ready for consumption. This correction happens one or more buffers later than the original writing. In this case two producing sockets use the same descriptor chain, one is a few buffers ahead and is writing data, the other lags behind and issues the correction. In this case, the sockets are configured such that only the last socket issues produce events.

27.2.4.2 Software manipulation of sockets

Sockets are initialized, inspected and modified by software. The procedures by which to do this are described here.

27.2.4.3 Initializing a socket

Sockets can be initialized only when in IDLE state, i.e. SCK_STATUS.enable=0. If this is not the case, the socket must first be terminated. The general procedure is as follows

- Descriptors are allocated or located and initialized in memory. The descriptors are chained appropriately.
- The socket registers are initialized with proper configuration values. This includes SCK_CHAIN, which contains the number of the first descriptor to be loaded. The register space for every socket includes a copy of the descriptor registers (DSCR_xxx). These should not be initialized, the DMA adapter will load those by itself.
- SCK_STATUS.enable is set to 1 to activate the socket (if so desired)

27.2.4.4 Terminating a socket

Sockets can be terminated at any time. If a socket is active, its activities will be aborted after an unspecified amount of time. The general procedure is as follows

- SCK_STATUS.enable is cleared to 0. The peripheral will continue to perform an unspecified amount of its pending activity.
- It is permissible to write 0 multiple times to SCK_STATUS.enable while the socket is being terminated, but it is illegal to write 1 to it during this time
- SCK_STATUS.enable is read back until its value changes to '0'. No further activity emanates from this socket after SCK_STATUS.enable is observed as cleared.

A socket that is not enabled will not reflect any interrupt status. If interrupt status needs to be read, firmware must do this read before the socket is disabled - use GO_SUSPEND to be able to read the interrupts status and then disable the socket.

27.2.4.5 Modifying a socket

Sockets are normally modified safely only when in suspend state. A socket that is active or stalled must first be suspended. A socket that is suspended will not complete an ongoing transfer, but rather go into suspended state almost immediately. It is possible, though that going into suspend state safely takes a noticeable (but small) number of cycles. The general procedure is as follows:

- SCK_STATUS.GO_SUSPEND is set to '1'
- SCK_STATUS.GO_SUSPEND is read back until its value changes to 1 or SCK_INTR.SUSPEND is used as interrupt source to indicate the same
- Any required changes are made to SCK_xxx or DSCR_xxx registers
- SCK_STATUS.GO_SUSPEND is cleared. If SCK_DSCR.active_dscr was modified the socket will load the new descriptor from memory, otherwise it will resume operation using the current contents of DSCR_xxx register.

Note that GO_SUSPEND will only take effect after the transfer of the current buffer completes. This may take a long time.

When a socket is in the suspended state, it is typical for firmware to clear interrupts. Note that the existing state of a socket will reassert interrupts and/or interrupt status bits if those interrupt conditions are still active. For instance clearing the suspend interrupt while the socket is suspended will result in the socket reasserting the suspend interrupt. Similarly, if the DSCR_NOT_AVL bit is cleared and the descriptor chain is not appended/refreshed, then the socket will reassert the DSCR_NOT_AVL bit. This implies that firmware should first address the conditions indicated by the interrupt before clearing the interrupts. Interrupts will not reassert if the socket is disabled.

Note that it is also possible to modify sockets that are in the stalled state, provided that it is known that no synchronization EVENT(s) will occur. This is normally the case when the socket is waiting for firmware to generate an event, i.e. the socket is not coupled to another socket in another adapter. In this case it is not necessary to first suspend the socket.

27.2.4.6 Inspecting a socket

Sockets can be inspected at any time. When a socket is active values are not guaranteed to be accurate due to activity and clock crossing issues. The general procedure is as follows

- Any value is SCK_xxx or DSCR_xxx registers is read twice in succession
- If the values of two reads match they are accurate. If they are different go back to step 1.

27.2.4.7 Wrapping up a socket

A socket that is in the ACTIVE state, but that is not expected to send/receive any more data can be forcibly wrapped up by asserting SCK_STATUS.WRAPUP. This should never be done for sockets that are not in the active state or that may send/receive data. This option is normally used when the core IP has transitioned into an error or partial completion state and firmware is required to cleanup the remaining, unexecuted, portion of the transfer.

27.2.4.8 Packets, Bursts, Buffers and Transfers

The DMA mechanism can be used in several different ways to realize transfers made up of multiple packets. Differences exist depending on how the core's native packet concept is mapped on DMA buffers and descriptors especially if this packet size is not the same as the buffer size. This mapping does not need to be 1:1. The terms used in this section are used as follows:

- **Buffer**: A buffer is chunk of memory as defined earlier. It has one (or more) DMA descriptors associated with it. More descriptors pointing to the same buffer is used for doing corrections/processing by the CPU as explained later in this section. The buffer size and location needs aligned to 32-byte boundary to increase the throughput. The buffer could be classified as
  - **ZLB**: Zero Length Buffer - a buffer with no data in it
  - **SLB**: Short Length Buffer - a buffer with more space than data
  - **FLB**: Full Length Buffer - a buffer that is entirely filled with data
- **Packet**: A packet is a unit of transmission that is defined in the local context of a specific peripherals. packets may all be of same size or be of variable size. Different peripherals may use different packet sizes. For example, the packet size of USB is typically 512B (HS Bulk) or 1024B (SS Bulk). Packets have a specific meaning in the context of the peripheral that exchanges them (as opposed to bursts, see below). The packet could be classified as
  - **ZLP**: Zero Length Packet - a packet with no data in it
  - **SLP**: Short Length Packet - a packet that is less than max size
  - **FLP**: Full Length Packet - a packet that is of max size
- **Burst**: A burst is a chunk of data being transferred between an IP core and a DMA adapter or on the peripheral interface. Bursts are used only to implement flow control on a higher-than-1 latency interface and carry no semantics associated with the data.
- **Transfer**: A transfer is a large amount of data (up to MB/GB size) that is transferred as a series of packets/bursts/buffers. A transfer could be classified as
  - **ZLT**: Zero Length Transfer - a transfer with no data in it
  - **SLT**: Short Length Transfer - a transfer shorter than expected
  - **FLT**: Full Length Transfer - a transfer of the expected length

27.2.4.9 Connection model

Figure 27-13 depicts the model used to describe how transfers are done. It considers an end-to-end chain of an ingress and egress peripheral and accompanying DMA adapters. It also considers the software drivers for the ingress and egress peripherals (that are mutually independent) and the higher-level software that manages the end-point. An important concept achieved in this model is independence between ingress and egress protocols. This enables coupling between any ingress peripheral to any egress peripheral without overly complex conversions. Basically the representation in memory (in buffers) allows for large transfers in memory between ports using different packet sizes.

In the following sections we will first define the buffer representation of each transfer mode and then consider the interpretation by ingress and egress peripherals of various modes and for packetized vs. continuous streaming peripherals.

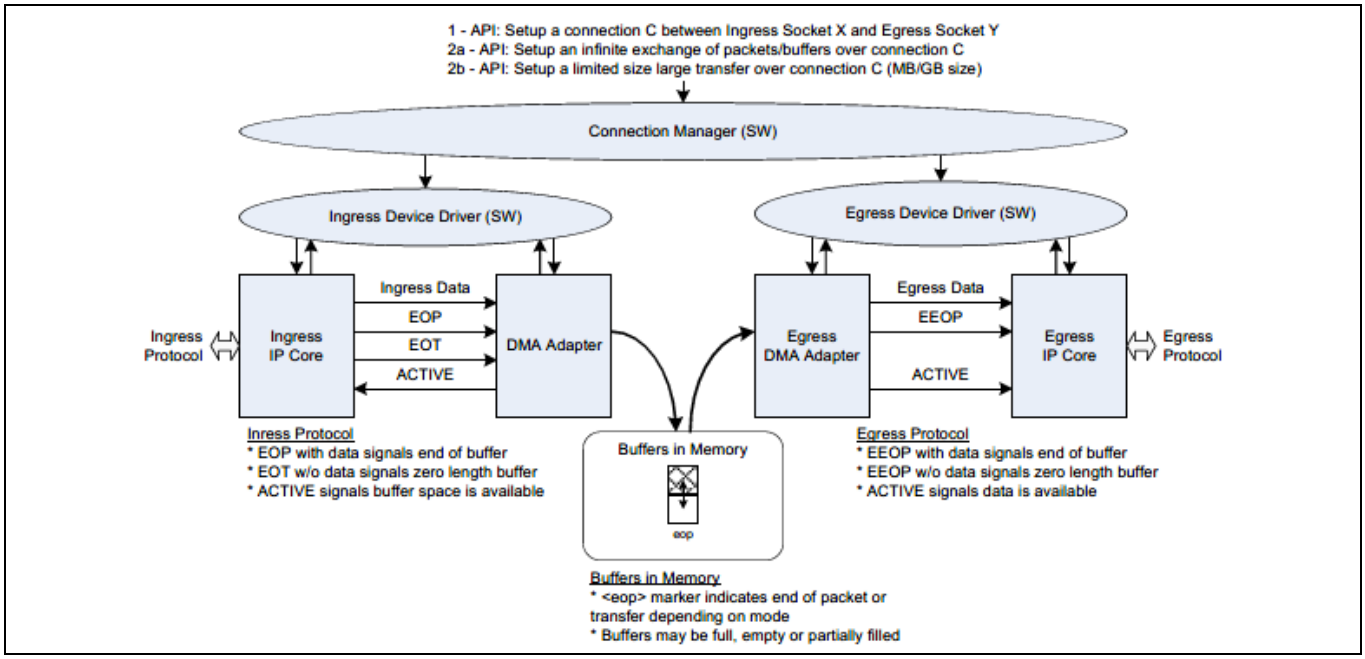


Figure 27-13. Connections

27.2.4.10 Transfer modes

There are four different modes in which data can be represented as buffers in memory:

- **Packet mode**: In this mode each packet is represented as a single buffer in memory. Since packets are often relatively small (smaller than buffers) this is usually not the most efficient mode. It is used for protocols that need to preserve packet sizes between peripherals and/or that require every packet to be forwarded from ingress to egress peripheral immediately (low latency requirement), for example a Modem connection between USB and a processor port.
- **Stream mode**: In this mode an infinite stream of data is transferred between ingress and egress peripherals. ZLP/SLP packet boundaries can still be communicated between ingress and egress peripheral when relevant by transmitting ZLB/SLB; otherwise packets are collected into full buffers. This mode is used for high performance transparent feed throughs, like a high performance USB3 bulk end-point.
- **Large Transfer mode**: In this mode a large amount of data is transferred between two peripherals without any information on how the data was packetized on ingress or how it is to be packetized on egress. It is the responsibility of the ingress/egress peripheral and its driver to ensure proper packetization (including ZLP behavior). This mode is used for large file/block transfers.

27.2.5 DMA adapter

The DMA adapter is the de-centralized DMA controller that is embedded in each block included in this sub-system.

- Each DMA adapter implements one or more threads. Each thread has zero or one executing sockets at any point in time. A thread represents a dedicated set of physical wires over which the peripheral transfers data with its DMA adapter.
- Each thread may be associated with multiple active sockets. Active sockets are sockets that have a valid descriptor and data/space available for transfer.
- A socket must be active before it can execute. If a peripheral attempts to 'execute' (i.e. transfer data into or out of an inactive socket) the DMA adapter will reject the transfer.

27.2.6 DMA register definitions

Every peripheral that is capable of participating in the HBWSS DMA scheme implements a standard set of registers for sockets, descriptors and timeout/event handling as part of the DMA Adapter. These DMA registers follow a standard layout and always start at a fixed offset within the IP config space - to be decided by IPS1. Although the number of sockets implemented by each peripheral may vary, the layout of a single socket is identical across all peripherals.

- **Socket**: There are 4 registers associated with a socket. Typically, these only exist in the IP block as true registers.
- **Descriptor**: There are 4 registers associated with a descriptor. A descriptor has a 16-bit number. A (possibly stale) descriptor copy is always located in memory. The consuming and producing sockets for the descriptor will have local copies in their register space when the descriptor is active on the socket. At appropriate times these copies are written back to memory; this is detailed in **"Socket operation"** on page 402.
- **Event**: There is 1 register associated with an event. In the HBWSS, the event register bit is connected to the hardware trigger matrix output for that socket which allows the direct communication of events between producer and consumer DMA Adapters.
- **Global Registers**: A number of DMA adapter global registers are bit vectors with one bit per socket. These registers are defined here as 256 bit-wide, but in practice are as wide as the respective IP has sockets.

27.3 Operation

The Adapter block is responsible for implementing the decentralized DMA controller. The Adapter is configurable based on the features needed to support the core in terms of:

- Clock domain crossing
- Queue sizing
- Number of outstanding requests

The DMA Adapter implements the DMA messaging protocol. Provides views of the system to the core in the form of socket and direct communication. This interfaces to the AXI interconnect that is responsible for moving data between peripherals. The interface to the core is through a configurable number of threads. Each thread is in turn is individually configurable. A thread contains Sockets, a Descriptor Manager, and a collection of prefetch queues.

A Thread Processor implements the following functions:

- Address Translation: given a socket ID from the core, a system address is generated
- Descriptor Management: Implement the Synchronization Mechanism and Descriptor Management
- Socket Management: context switching between sockets, socket prefetching and pipeline control
- Fetch Queue (FQ): maintain data storage on a per-socket basis for short latency accesses from the core
- Data Alignment and Burst handling.

The thread processor includes the following blocks

- Command Queue (CQ): This block accepts commands from the core
- Response Queue (RQ): This block holds the data read from the memory that is to be delivered back to the core
- Socket Controller (SKT): This block holds the socket status register and implements the socket state machine
- Socket Look-aside Buffer (SLB): The is a memory that holds the socket and descriptor registers for all sockets. The memory also includes the FQ, that holds the prefetched read data for all sockets. Independent pointers are maintained on a per socket basis.
- Transient Socket state machine (TS): This block processes sockets that are in the transient state. This includes:
- Fetching descriptors from memory
- Pre-fetching data from memory
- Generating events

- Executing Socket state machine (ES): This state machine processes the active socket being requested by the core.
- Barrel Shifter (BS): The block is responsible for data alignment and burst handling
- Transaction Queue (TQ): The TQ maintains information needed to process read data coming back from the system

An Adapter can be configured to have multiple threads. Each thread will have a dedicated CQ/RQ/ES/BS structure.

A socket may exist in two kind of states in the Adapter - they are

1. Stable states: A socket may persist in this state for an indefinite amount of time. Each socket can be in any stable state independent of other sockets and therefore requires a dedicated state bit on a per socket basis. The stable states are:
   - Descriptor: This is the default idle state. It indicates that the descriptor is not valid in the socket and should be fetched from memory if possible. A socket that gets terminated by software will return back to this state.
   - Stall: This state indicates that the descriptor is available in the socket but the core is not allowed to transfer data through this socket either because the read data hasn't been pre-fetched into the socket yet or the buffer is not available in memory.
   - Active: This state indicates that the socket is ready for core transfers. The core is the master and can initiate requests as it wills. Read data is available locally in a small prefetch buffer for fast access. Write data can be sent immediately to the buffer in memory.
   - Event: Indicates that all transfers have completed for the current descriptor through the socket. The descriptor needs to be written back to memory, an event sent to the twin socket and ensure that all transient data have been flushed out of the pipe.
2. Transient states: A socket will temporarily enter these state for a short amount of time. They are used to carry out a certain sequence of events needed to enter a new stable state. Since these states are transient, only one socket is allowed to be in a transient state. As a result, a single state machine can be shared among all sockets. Sockets in the Active state do not have any associated transient states. The transient states are:
   - Idle: This designates a stable state boundary. A suspended state will freeze the socket in this state.
   - Cmd: A known sequence of commands will be used to access specific addresses in the memory map. The commands will be issued if the bubble and the Transaction Queue (TQ) are not full. The sequence of commands depends on the socket's current stable state.
     - Descriptor: In this state the adapter fetches all bytes of the descriptor from memory address location specified in the socket
     - Stall: This state is entered for consumer sockets only. Fetch the configured number of bytes to fill the prefetch queue of the socket from the memory location specified in the descriptor.
     - Event: In this state the descriptor is updated back to the memory location specified in the socket. An event is generated to the twin socket.
   - Waiting For Data(WFD): The TQ tracks all outstanding requests. The popped entry marked "Last" indicates that all data words have been fetched.
   - Flush: This state is used to throw away outstanding requests or prefetched data
     - Descriptor: This state is entered when a new Event is received while the descriptor is being fetched from memory. This is a potential race condition that requires discarding the current descriptor and fetching the descriptor again from memory.
     - Stall: This state is entered when the software wants to terminate the socket. All data in the prefetch queue (FQ) of this socket must be thrown out.
     - Event: This state is entered when the software terminated an Active socket or the core decided to terminate the transfer without consuming all the data words in the prefetch queue. All data in the prefetch queue of this socket must be thrown out in preparation for the next descriptor.

27.4 Link layer

The link layer is the bridge between the DMA adapter and the DMA interconnect. This layer interfaces to the DMA adapter push/pop interface and implements the AXI protocol towards the interconnect.

The AXI master interfaces to the push/pop DMA adapter interface and implements the AXI3 protocol. The AXI3 Interconnect runs without any wait states if an AXI Master has a continuous supply of data. An AXI Master will only assert its valid signal if the DMA Adapter signals that data or space is available.

- **Event Signaling**

  In the HBWSS, the event register bit is connected to the hardware trigger matrix output for that socket which allows the direct communication of events between producer and consumer DMA Adapters.

- **Race Conditions**

  As the communication between partner sockets is through hardware triggers, it eliminates many of the race conditions as the trigger timing is deterministic. Message ordering is important to maintaining an accurate and correct state of the system.

- **Event-Data Race**

  The following actions can trigger this scenario:

  – A socket started a read operation to get the next descriptor in the chain
  – The next descriptor was read out of memory and is currently in-flight towards the requesting socket
  – The other socket in the same chain operating on the same descriptor finished its operations on that descriptor and wrote the descriptor back to memory
  – The other socket sent a notification event to the first socket

  RACE: Event 4 got to the first socket ahead of 2. The first socket needs to realize that the data coming back from memory is stale, should be thrown out, and a new read operation initiated to get the new descriptor.

  Solution: A socket receiving an event to a stalled descriptor will reload the descriptor from memory.

- **Write-Read Race**

  The following basic actions can trigger this scenario:

  – A socket finished its operations on a descriptor and wrote the descriptor back to memory. An alternate example is the CPU updating the descriptor list in memory before activating a socket.
  – A notification event was sent to the other socket
  – Having received the event, the other socket initiated a read to memory to get the descriptor

  RACE: The read request 3 got to the memory ahead of the first socket's write 1. Stale data will be returned to the other socket. There is no guarantee that subsequent reads will not reach memory ahead of the write again. This race exists because the write and read actions are carried over two completely independent interconnects.

  Solution: This race cannot occur since the DMA adapter does not issue a trigger until after it gets a grant for the descriptor write. Hence, write data is written to the SRAM before the hardware trigger is issued.

- **Descriptor atomicity**

  If the descriptor is accessed as multiple read/write requests, it is possible that a corrupted descriptor is read by a socket partner. For HBWSS this is an illegal case. The descriptor state is accessed by a single write/read operation.

  Write-Write Race

  The following basic actions can trigger this scenario:

  – The socket is in the process of updating the descriptor in memory. It sent a write operation but the write is still in flight on its way to memory.
  – The CPU sent a Terminate/Suspend signal to the socket and read back the Terminate/Suspend bit from the Socket to ensure the operation completed successfully.
  – Having verified the Terminate command reached the socket correctly, the CPU modified the descriptor in memory

  RACE: The original in-flight descriptor write operation initiated by the socket reaches the memory after the CPU operation and corrupts the CPU update.

  Solution: For the HBWSS, the DMA Adapter does not update the Terminate/Suspend bit until after it has received a grant for the descriptor write back. The guarantees the order of events, DMA Adapter Descriptor write back before the CPU descriptor update.

- **HBWSS Multi-Layer DMA Interconnect**

  The Multi-Layer DMA Interconnect in this sub-system is specified to meet the bandwidth requirements of the USB3.2 and LVDS IPs. The interconnect has the following properties:

  – AXI3 protocol, 64bit data width at 320Mhz (maximum), 4 beat (32 byte) bursts
  – 2 Masters on the bus
  – USB3.2 device controller
  – LVDS controller
  – 1 Slave - SRAM controller
    - One shared address bus for write and read (Address-merge AXI3 bus)
    - All AXI entities are in the same clock domain.
    - All AXI entities use the same interleaved access timing.
    - No bus bandwidth is wasted in wait states
    - Typical data flows
    - USB3.2->SRAM->LVDS
    - LVDS->SRAM->USB3.2

- **Descriptor structure**

  The USB platform defines a 16-byte descriptor and the central DMA defines multiple different descriptor formats with a max length of 32-byte. The descriptor used in this sub-system needs to conform to both the descriptor structure of the distributed DMA controller and the central DMA controller in EZ-USB™ FX10. Descriptors are placed in memory aligned to 32-byte boundaries regardless of the descriptor length. This allows for a simple shift and add to locate a descriptor from a descriptor number.

  To allow the existing Central DMA and HBWSS DMA controllers to use the same descriptors, a new “Dual-Role” descriptor is defined. The mapping between this descriptor and the central/distributed DMA controllers is shown in **Table 27-3**. As 2 fields of the 2D descriptor are used for mapping additional fields in the HBWSS, the central DMA is restricted from using the 2D type when accessing buffers in this sub system.

Table 27-3. Descriptor mapping

EZ-USB™ FX10 - Central DMA controller						USB - Distributed DMA controller		HBWSS		Comments
Size	**16 B**	**24 B**	**32 B**	**20 B**	**16 B**	**Size**	**16B**	**Size**	**32B**	
Type	**Single**	**1D**	**2D**	**Memory copy**	**Scatter**	**Type**	**NA**	**Type**	**NA**	
Register name	**Offset**					**Register name**	**Offset**	**Register name**	**Offset**	
DESCR_CTL	0x00	0x00	0x00	0x00	0x00			DSCR_CTL	0x00	
DESCR_SRC	0x04	0x04	0x04	0x04	0x04	DSCR_BUFFER	0x00	DSCR_BUFFER_W	0x04	Buffer Pointer
DESCR_DST	0x08	0x08	0x08	0x08	---			DSCR_BUFFER_R	0x08	Buffer Pointer
DESCR_X_SIZE	---	0x0c	0x0c	0x0c	0x08			DSCR_SIZE	0x0C	The size field of USB30 DSCR_SIZE is moved to this field in the HBWSS
DESCR_X_INCR	---	0x10	0x10	---	---			Not Used	0x10	
DESCR_Y_SIZE	---	---	0x14	---	---			Not used		
DESCR_Y_INCR	---	---	0x18	---	---			Not Used		
DESCR_NEXT_PTR	0x0c	0x14	0x1c	0x10	0x0c	DSCR_CHAIN	0x08	DSCR_CHAIN	0x14	
						DSCR_SIZE	0x0C	DSCR_MARKER	0x18	In USB30 DSCR_SIZE field has both the size/byte count details and marker details like EOP/ERROR etc
						DSCR_SYNC	0x04	DSCR_SYNC	0x1C	

The buffer pointer is specified by the DSCR_BUFFER register in the DMA adapter and is duplicated in two places:

- For producer sockets, this field is read from the CH_DESCR_SRC field.
- For consumer sockets, this field is read from the CH_DESCR_DST field.

DSCR_SYNC contains information on the partner socket to be used by the DMA adapter to signal events.

The next descriptor in the chain is specified in the DSCR_CHAIN register and is mapped to the same location as the DESCR_NEXT_PTR.

The DSCR_SIZE register contains the size of the buffer for producer sockets and the available data in the buffer for consumer sockets. This is mapped to the same location as DESCR_X_SIZE.

The DSCR_MARKER register contains information about the EOP/Error/Occupied status of the buffer.

- **Descriptor formatter**

  The DMA Adapter from USB block is intended to be re-used with minimal modifications. To map the proposed descriptor structure to the existing DMA adapter format a small glue layer will translate the new descriptor structure to the existing DMA adapter signals. This block snoops DMA adapter descriptor fetches and performs a bit steering function to convert a Dual-Role descriptor to the16 byte descriptor format. This block also performs the bit steering function from the 16 byte descriptor to a Dual-Role descriptor for descriptor write backs.

  The following functions will be implemented in this layer:

  – The buffer address will be translated from DSCR_BUFFER_W or DSCR_BUFFER_R based on the socket direction
  – The next descriptor is derived from the appropriate field position in the DSCR_CHAIN register
  – Event register writes to the socket in DSCR_SYNC register are translated to triggers
- **Trigger multiplexer**

  The HBWSS uses hardware triggers for event signaling. A trigger matrix is used for routing of events. This trigger matrix is programmed by firmware so that it matches the descriptors and transfers setup by firmware.

  The HBWSS DMA adapter has all of the features needed for continuous transfers without a need for firmware involvement. Firmware sets up the transfers between LVDS and USB32 IPs. Then firmware enables the transfers and no further involvement is required by firmware unless there is an error or a timeout event.

  For transfers involving the central DMA controller, the trigger controller snoops descriptor fetches by the Central DMA controller, since the Central DMA controller does not have the ability to update the descriptor state.
- **Memory Array Block**

  This block instantiates the memory array for storing the data transferred through the HBWSS. The features of this block are:

  – 1 MB SRAM
  – Data width - 128
  – Max Frequency - 160 MHz
  – Provides the following ports
    - AXI3 Slave port for USB3.2 Controller
    - AXI3 Slave port for LVDS Controller
    - AHB Slave port for system interconnect
  – AXI3 Slave interface has shim to SRAM
    - Converts 64 bits at 320 MHz to 128 bits at 160 MHz
  – AHB Slave interface is asynchronous to this block
    - FIFO to handle CDC and to minimize AHB wait cycles
  – SRAM controller to enable max bandwidth (up to 80 Gbps)
  – Provides configuration registers for SRAM control
  – Implements a trigger multiplexer for interfacing to Central DMA controller

High-bandwidth DMA subsystem

- **Data flow diagrams**

This section depicts the data flow for different scenarios:

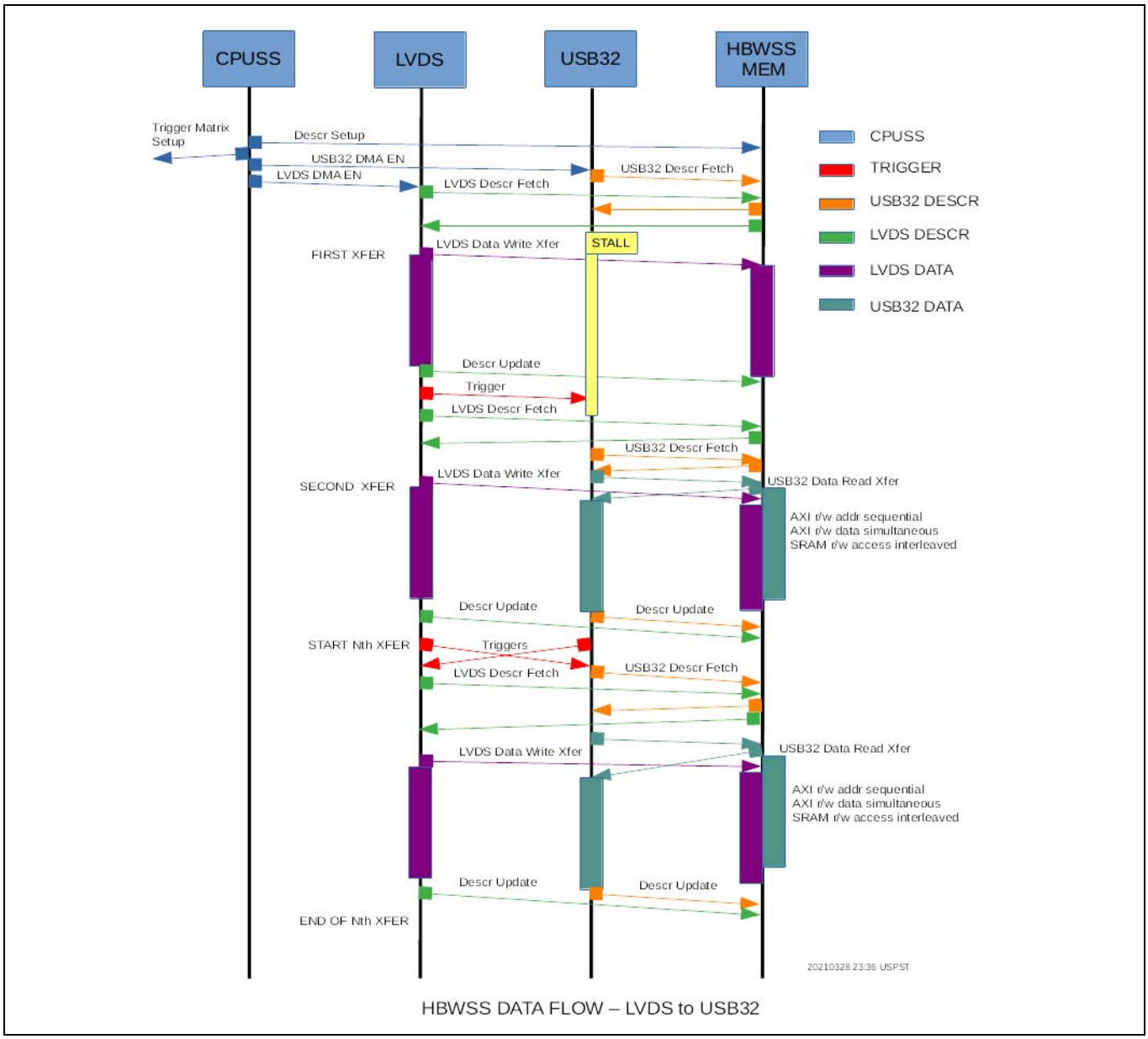


Figure 27-14. HBWSS data flow LVDS to USB32

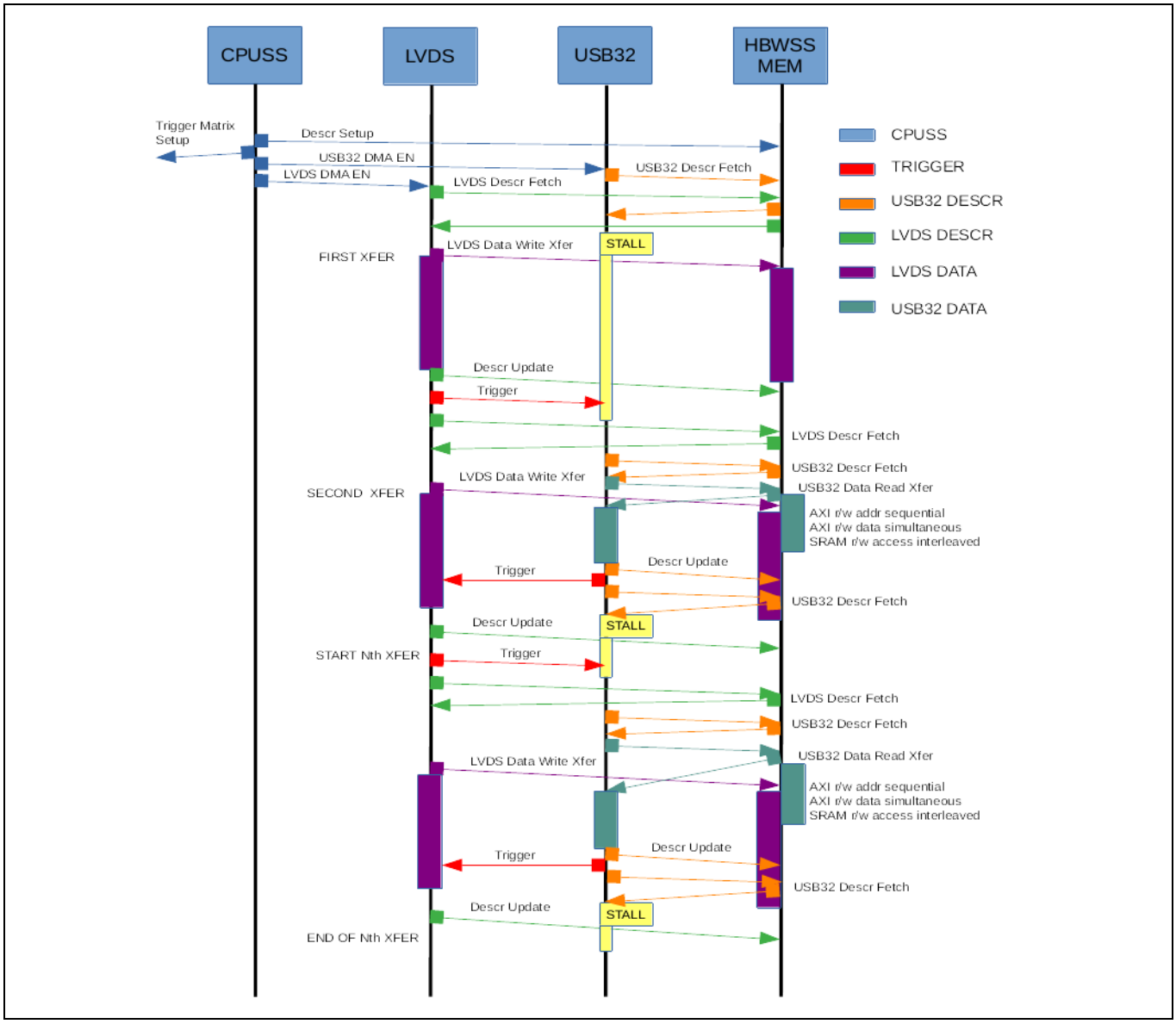


Figure 27-15. HBWSS data flow - Slower LVDS to faster USB32

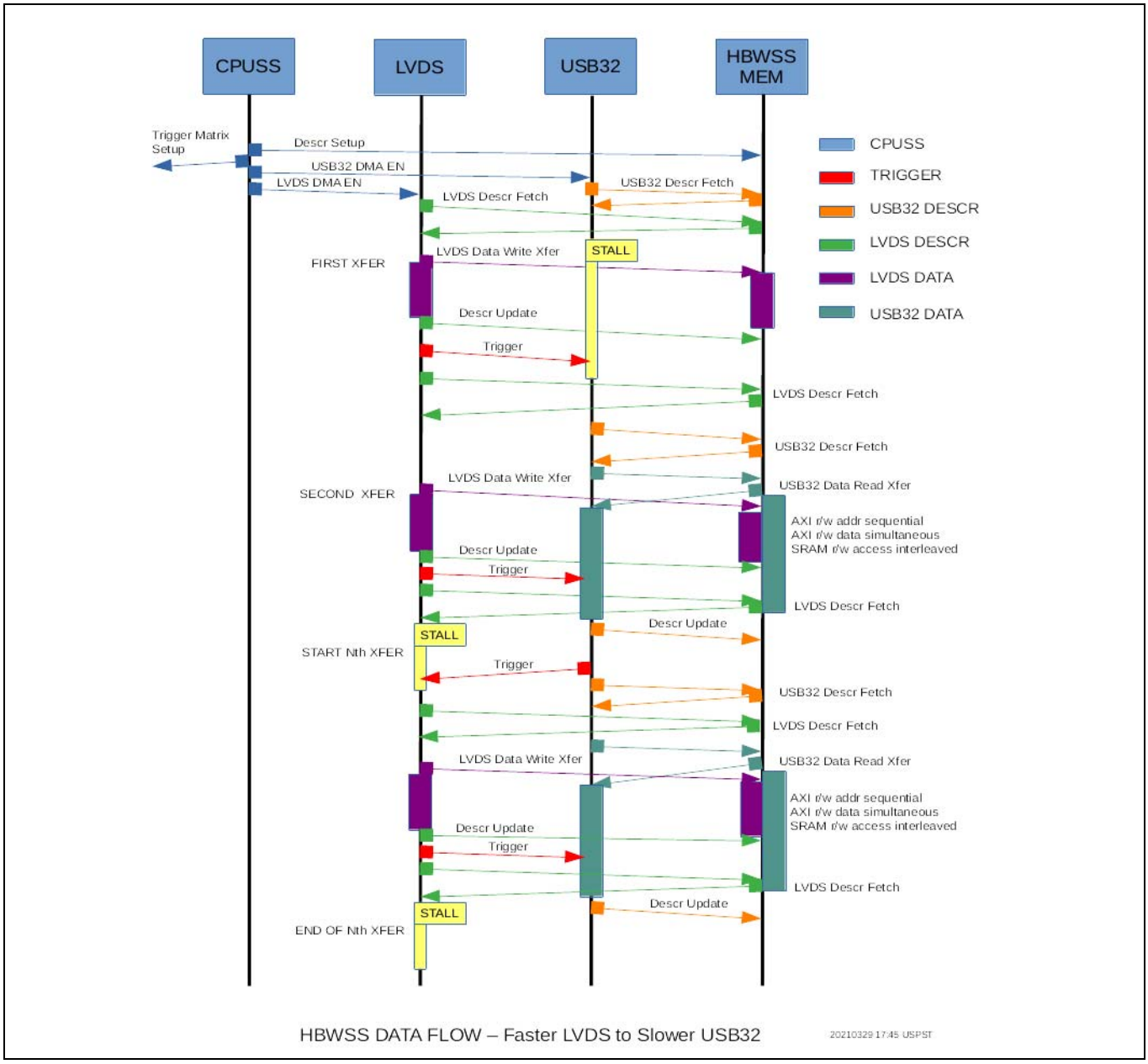


Figure 27-16. HBWSS data flow - Faster LVDS to slower USB32

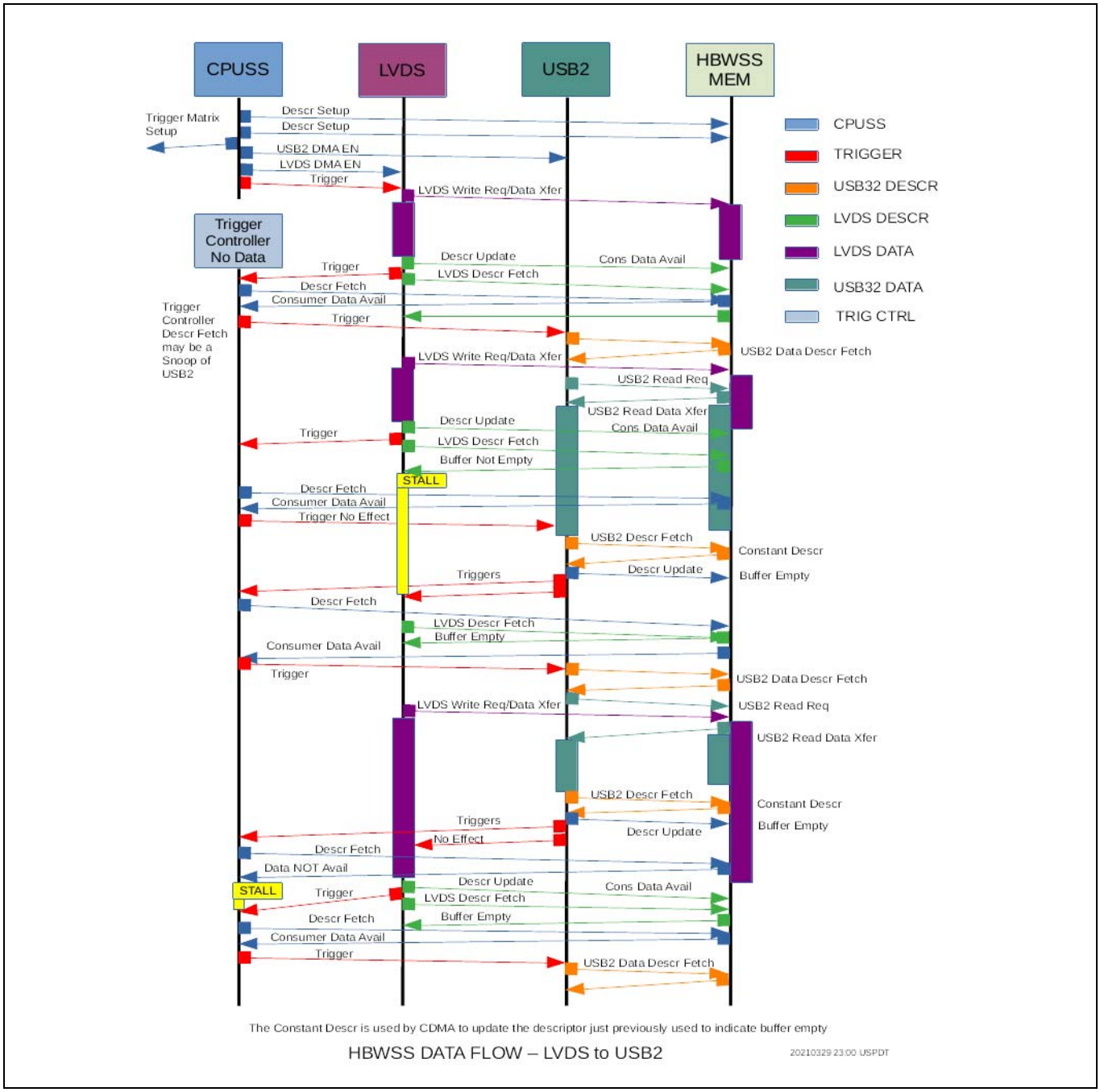


Figure 27-17. HBWSS data flow - LVDS to USB32

27.5 DMA transfer management

This section documents the procedures used to manage the various DMA data transfers by the EZ-USB™ FX10 device firmware. These procedures are primarily implemented at the peripheral driver library (PDL) level and invoked as needed by the application specific modules.

27.5.1 DMA configuration

The DMA data path configuration for a given application is generally done at initialization time and left unmodified thereafter. There can be a few special cases where the configuration is changed as part of switching from one mode of operation to another, but the sequence of one-time data path configuration followed by a large number of transfers remains valid.

The type of USB data connection that will be established (USB 2.0 or USB 3.2) is generally not known at compile time and hence EZ-USB™ FX10 applications will need ability to select the DMA use cases at runtime.

27.5.1.1 DMA channel setup

27.5.1.1.1 HBWSS DMA Channel Creation

While the distributed DMA architecture does not have any notion of a channel, it is useful to think of a DMA channel as a construct that links all the resources required for the transfer of a stream of data.

Figure 27-8 shows the resources that make up an HBWSS DMA channel and their relationships. This includes **Sockets** that either produce (collect from an external interface) or consume (send to an external interface) the data stream, the RAM **buffers** which hold the data packets in transit and the **descriptor structures** which track the buffer status.

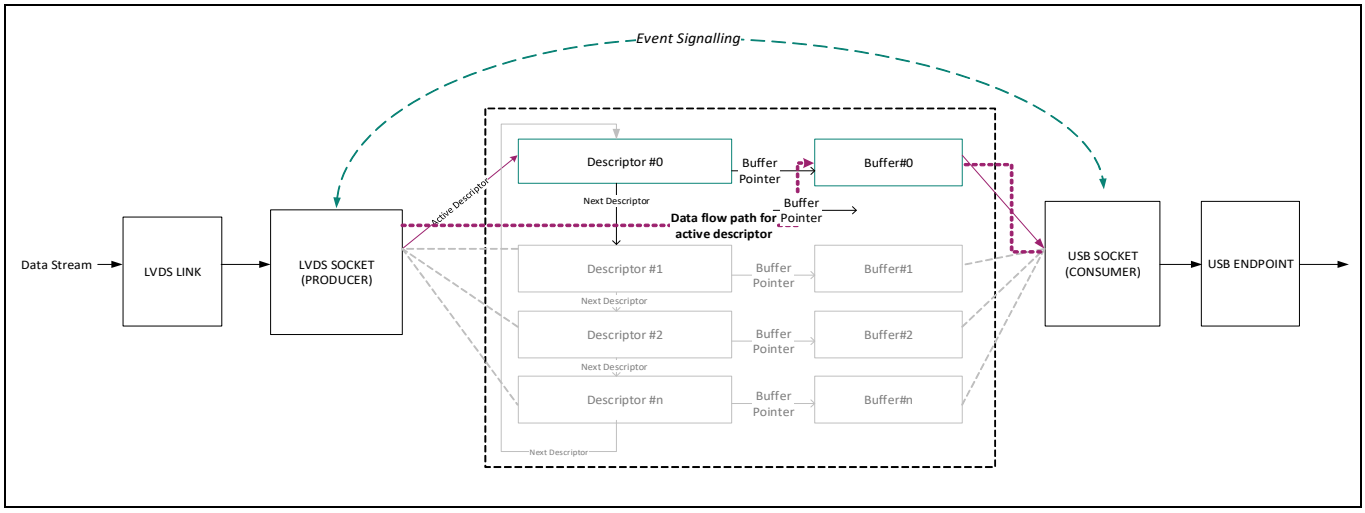


Figure 27-18. HBWSS DMA channel construct

Different types of DMA channels may be created based on the targeted use case:

1. Channel that moves data from one HBWSS DMA socket to another. This is called an **IP to IP DMA** channel.
2. Channel that receives data through one HWBSS DMA socket and forwards to the CPU (firmware) for handling. This is called an **IP to Memory DMA** channel.
3. Channel that takes data generated internally (by CPU or other peripheral) and transmits through a HBWSS DMA socket. This is called a **Memory to IP DMA** channel.

In addition to these basic types, the following variation is also possible:

4. The IP to IP transfer case can be expanded to take data received from two different DMA sockets and forward them (in interleaved fashion) to a single DMA socket. This is called as a Many-to-One DMA channel and is illustrated in **Figure 27-19**.

   The interleaving of data from the two producers is done only at buffer boundaries, with the size of the buffer being user programmable. Since separate "next descriptor" pointers are available in the descriptor structure for the producer and consumer sockets; it is possible to place the same descriptor in two different chains as shown in the **Figure 27-19**.

   This scheme will allow users to continuously receive data on the LVDS interface from an FPGA (or other) data source which is unable to implement a flow control scheme to absorb the time taken by each socket to switch from one DMA buffer to another.

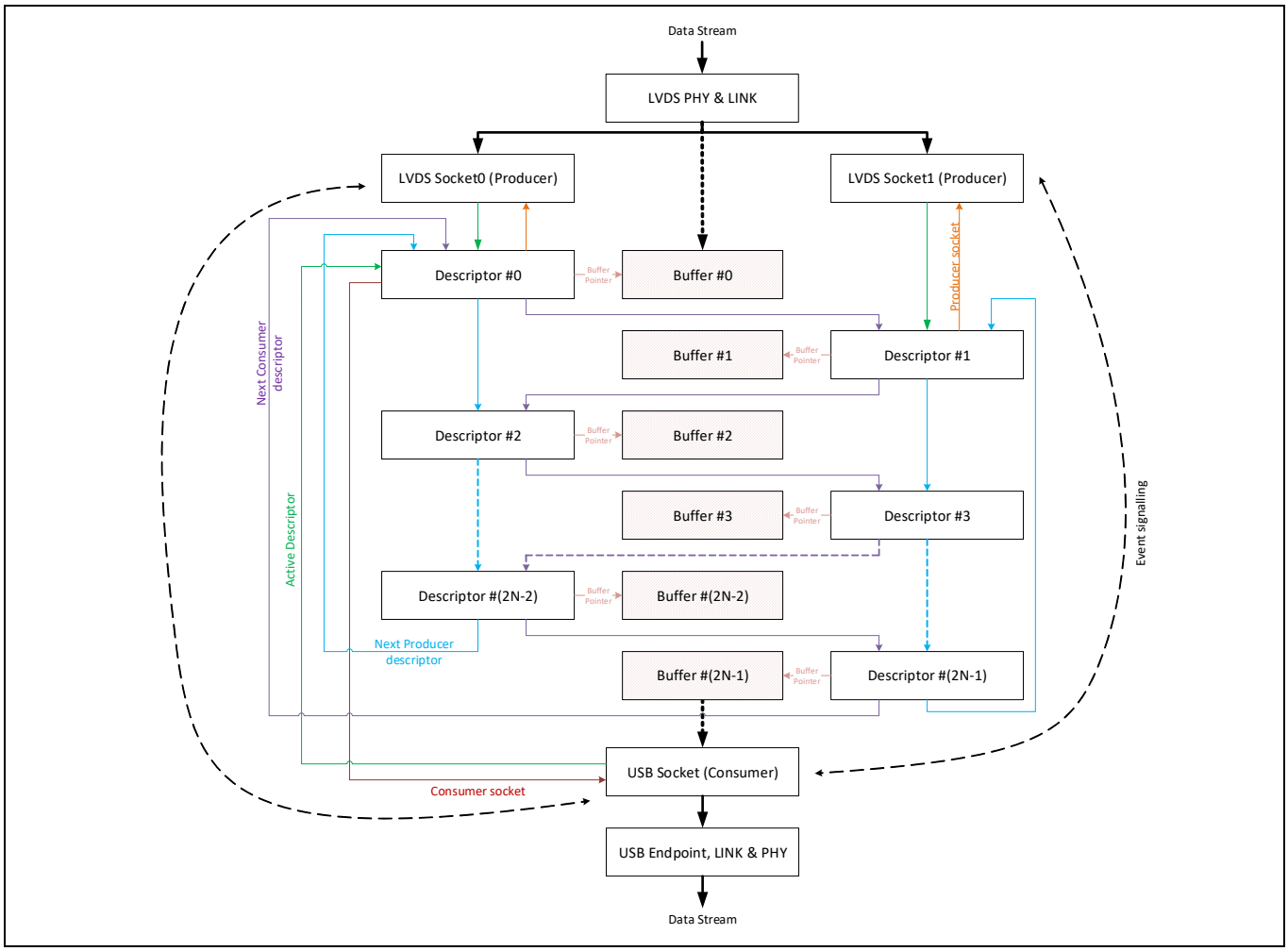


Figure 27-19. IP to IP DMA channel with two producer sockets

27.5.1.1.2 CPUSS DMA channel configuration

In the case of the CPUSS DMA, a channel is a physical construct representing an independent data transfer that is implemented by the DMA controller. The CPUSS DMA controller supports a number of DMA (or DataWire) channels that the firmware can configure as required to move data between peripherals and/or memories.

The API for configuring the CPUSS DMA channels are part of the ModusToolbox™ peripheral driver library (PDL) CAT1.

27.5.1.2 Configuring LVDS to USB DMA channels

27.5.1.2.1 LVDS to USB 3.2 DMA channel creation

In the ideal case, EZ-USB™ FX10 will make a USB 3.2 connection and data transfers will be done from LVDS to USB 3.2 device.

If an USB 3.2 connection is active, the DMA channel is created using an LVDS socket as the producer and an USB socket as the consumer. The DMA event triggers are enabled so that the two sockets can exchange status information directly and the interrupts are disabled as no further CPU involvement is required.

The DMA channel create function maintains a pool of descriptor structures and allocates the required number of descriptors from it to create the channel. This function also makes use of a custom buffer allocation routine to allocate DMA buffers from the available space in the HBWSS SRAM.

27.5.1.2.2 LVDS to USB 2.0 DMA channel creation

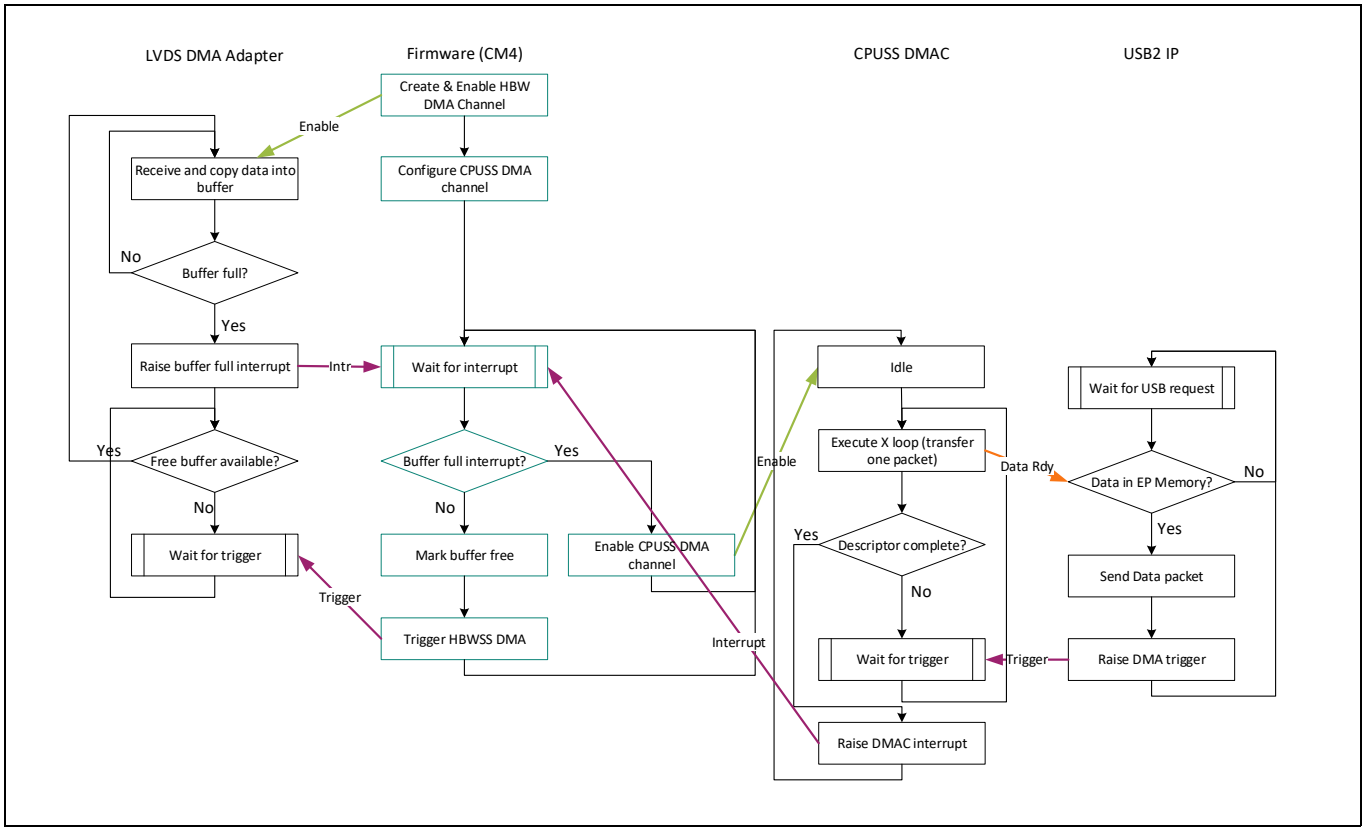


Figure 27-20. LVDS to USB 2.0 flow and interactions

If an USB 2.0 connection is active, the USB socket is not used as the data transfers happen through the Endpoint Memory block connected to the CPUSS DMA. In this case, the DMA configuration involves two steps:

1. Create a DMA channel which will transfer the data received from the LVDS socket into buffers in the HBWSS SRAM. The Cy_HBDma_ChannelCreate function can be used to create a channel of type CY_HBDMA_TYPE_IP_TO_MEM with interrupts enabled at the end of each buffer.
2. A CPUSS DMA channel is created to move the data from the HBWSS SRAM buffers into the Endpoint Memory region of the USB 2.0 IP block. The Cy_DMA_Channel_Init function provided by MTB PDL CAT1 is used to create/configure this DMA channel. 2D transfer mode is enabled in the DMA descriptors, with the X_COUNT set to match the maximum packet size of the USB endpoint and the Y_COUNT set to match the number of packets that are stored in a single data buffer. The trigger signal from the USB 2.0 endpoint memory is selected so that

the DMA controller waits for endpoint ready status before transferring each packet of data. Interrupts are enabled at the completion of each DMA descriptor (one buffer of data has been transferred).

Figure 27-20 shows the operation of the LVDS to USB 2.0 data transfer. These transfers require firmware (Cortex®-M4 CPU) intervention for the transfer of each buffer worth of data. The CPU gets interrupted once one data buffer has been filled by the HBWSS DMA and proceeds to enable the CPUSS DMA channel which moves it into the USB 2.0 Endpoint Memory block. The CPU gets another interrupt once all the data in the buffer has been transferred out by the USB 2.0 IP. The firmware then needs to mark the data buffer empty and send a trigger to HBWSS DMA indicating that free buffer space is available.

27.5.1.3 Configuring USB to LVCMOS DMA Channels

Figure 27-3, shows the USB to LVCMOS DMA configuration is very similar to the LVDS to USB DMA configuration. Only the direction of data transfer is reversed, with data being received through the USB endpoint and transmitted through the LVCMOS interface.

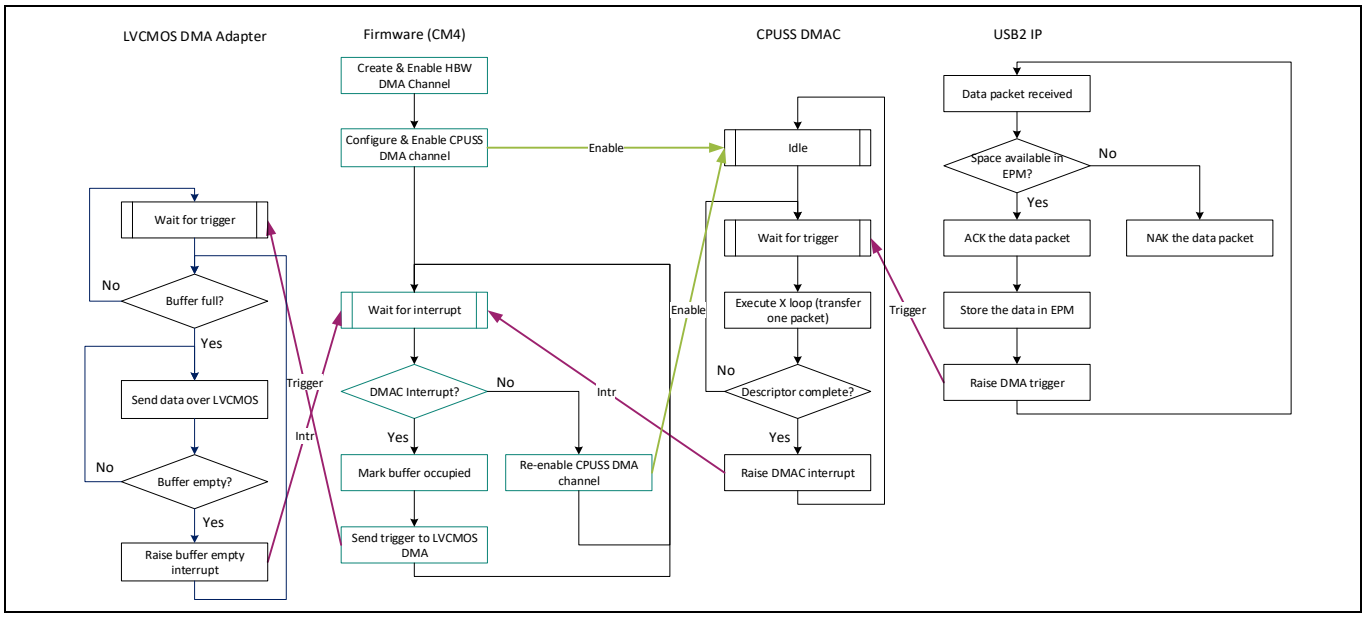


Figure 27-21. USB 2.0 to LVCMOS flow and interactions

In the case of an USB 3.2 connection, an HBWSS DMA channel of type CY_HBDMA_TYPE_IP_TO_IP is created to transfer the data with no further CPU intervention.

In the case of an USB 2.0 connection, two DMA channels are required:

- CPUSS DMA channel which moves the data from the USB 2.0 Endpoint Memory to the HBWSS SRAM
- (Memory to IP) HBWSS DMA channel which moves the data from the SRAM buffers to the LVCMOS socket and over the LVCMOS interface after that

As described in **"LVDS to USB 2.0 DMA channel creation"** on page 419, firmware intervention is required for transfer of each data buffer. The CPUSS DMA shall raise a descriptor complete interrupt when a buffer has been filled with data received from the OUT endpoint. Firmware will then update the HBWSS DMA descriptor to mark the buffer occupied and trigger the HBWSS DMA channel for transfer. If more empty buffers are available, firmware can re-enable the CPUSS DMA channel for transfer into the next buffer.

Once the buffer has been drained, the LVCMOS socket raises a buffer empty interrupt. If the CPUSS DMA channel was previously disabled because it ran out of empty buffers, the channel can be re-enabled for data transfer at this stage.

Figure 27-21 shows the control flow and interactions between the firmware and various hardware blocks involved in the transfer.

27.5.1.4 Configuring RAM to USB DMA channels

RAM to USB DMA channels are used to send data that has been assembled by the CPU (firmware) over an USB endpoint. The data may originate from read-only data stored in the FLASH or may be received from other peripherals and packetized by the firmware.

Separate DMA channels are required to handle these transfers in USB 3.2 and USB 2.0 connections.

- An HBWSS DMA channel of type CY_HBDMA_TYPE_MEM_TO_IP will be created and used for transfers from RAM to USB 3.2 endpoints
- A CPUSS DMA channel will be used to move data from the RAM to USB 2.0 endpoints

In both cases, the data is transferred one buffer at a time under firmware control. Triggers from the USB 2.0 IP are used to manage a multi-packet transfer without frequent CPU intervention. In the case of USB 3.2, the USB endpoint memory fetches data from the DMA buffer as required and splits it into packets of the configured size.

27.5.1.5 Configuring PDM to USB DMA channels

The DMA channels can be used to transfer data from other peripherals such as PDM, I2S, SCB, SMIF to USB endpoints. As in the case of other USB data transfers, the DMA configuration is different for USB 3.2 and USB 2.0 cases.

27.5.1.6 PDM to USB 3.2 DMA channel configuration

Figure 27-5 shows the data flow for transfer from the PDM IP to the USB 3.2 device IP. As shown, a CPUSS DMA channel is used to receive the data from the PDM IP and store it in HBWSS RAM buffers. An HBWSS DMA channel of type Memory to IP is then used to transfer data from these RAM buffers to the USB endpoint.

The interaction between the DMA channels and the CPU are similar to those shown in **Figure 27-21**. The USB 2.0 IP is replaced by the PDM IP as the data source. The LVCMOS DMA adapter is replaced by the USB 3.2 DMA adapter as the data sink.

27.5.1.7 PDM to USB 2.0 DMA channel configuration

Figure 27-9 shows the data flow for transfer from the PDM IP to the USB 2.0 device IP. While direct peripheral to peripheral DMA transfer would be possible, this is not used because of the USB data packetization requirements and need for two trigger signals (data ready on PDM side and space available in USB Endpoint Memory). So, a pair of DMA channels are used to move the data from PDM IP to SRAM and then from SRAM to the USB endpoint.

Figure 27-22 shows the interactions between the various IP blocks and the firmware in this use case. Since two CPUSS DMA channels are used, the operations of the DMA controller with respect to each of these channels is shown separately.

High-bandwidth DMA subsystem

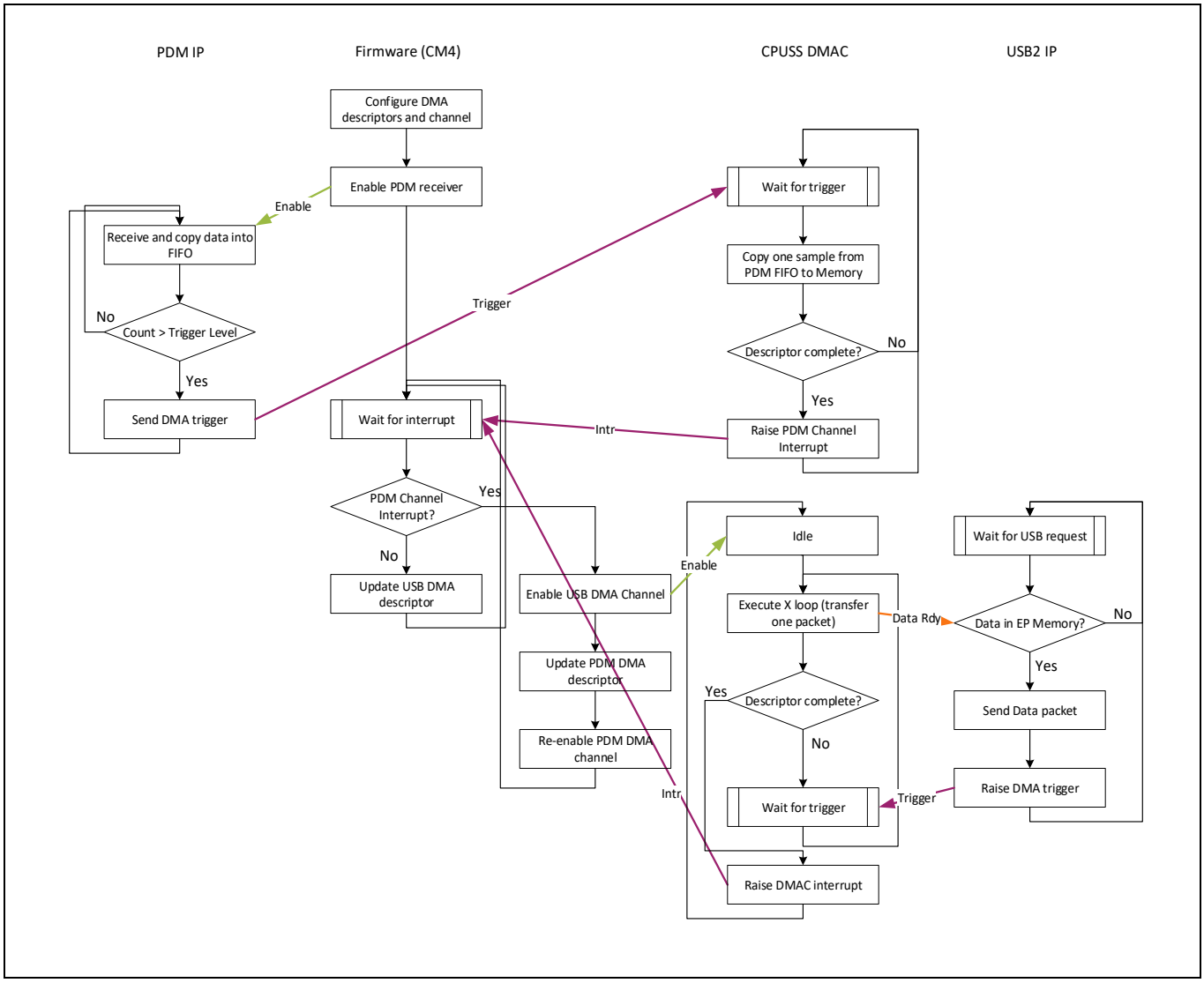


Figure 27-22. PDM to USB 2.0 flow and interactions

27.5.2 DMA operation

This section documents the DMA transfer operation in the HBWSS and CPUSS DMA cases. Some data streams require both CPUSS and HBWSS DMA channels to be used. In such cases, the firmware task is expected to synchronize between the two transfers.

27.5.2.1 HBWSS DMA operation

27.5.2.1.1 IP to IP data transfers

In the high-bandwidth cases like LVDS to USB 3.2 transfers and USB 3.2 to LVCMOS transfers, the firmware responsibility is limited to configuring the DMA channel. The HBWSS DMA automatically takes care of the actual data flow by allowing the producer and consumer sockets to trigger each other.

Figure 27-23 shows the interaction between the two sockets and the firmware in managing these transfers. Firmware is only required to enable the data transfer by configuring the DMA descriptors, mapping the sockets to the first descriptor and then enabling the sockets. The Firmware uses the Cy_HBDma_ChannelCreate function to create and configure the DMA channel and the Cy_HBDma_ChannelEnable function to enable the sockets for data transfer.

Once enabled, the producer socket monitors the descriptor to see whether the corresponding DMA buffer is empty. If so, the socket can receive the data and store it into the buffer. This can be continued until the buffer is full or a signal identifies that the current data transfer has been completed (End Of Transfer (EOT)). At the end of the buffer, the producer socket notifies the consumer socket using a produce event and then starts monitoring the next descriptor for free space.

Similarly, the consumer socket monitors its active descriptor to see whether the corresponding buffer contains any data. While the buffer has more data, the socket reads it out and transmits it out on the external interface. Once the buffer has been drained of data, the descriptor is updated to note that the buffer is empty and the producer socket is notified that a free buffer is available.

This flow is applicable to the LVDS to USB 3.2 and USB 3.2 to LVCMOS DMA transfers which require transfer rates higher than 1 Gbps.

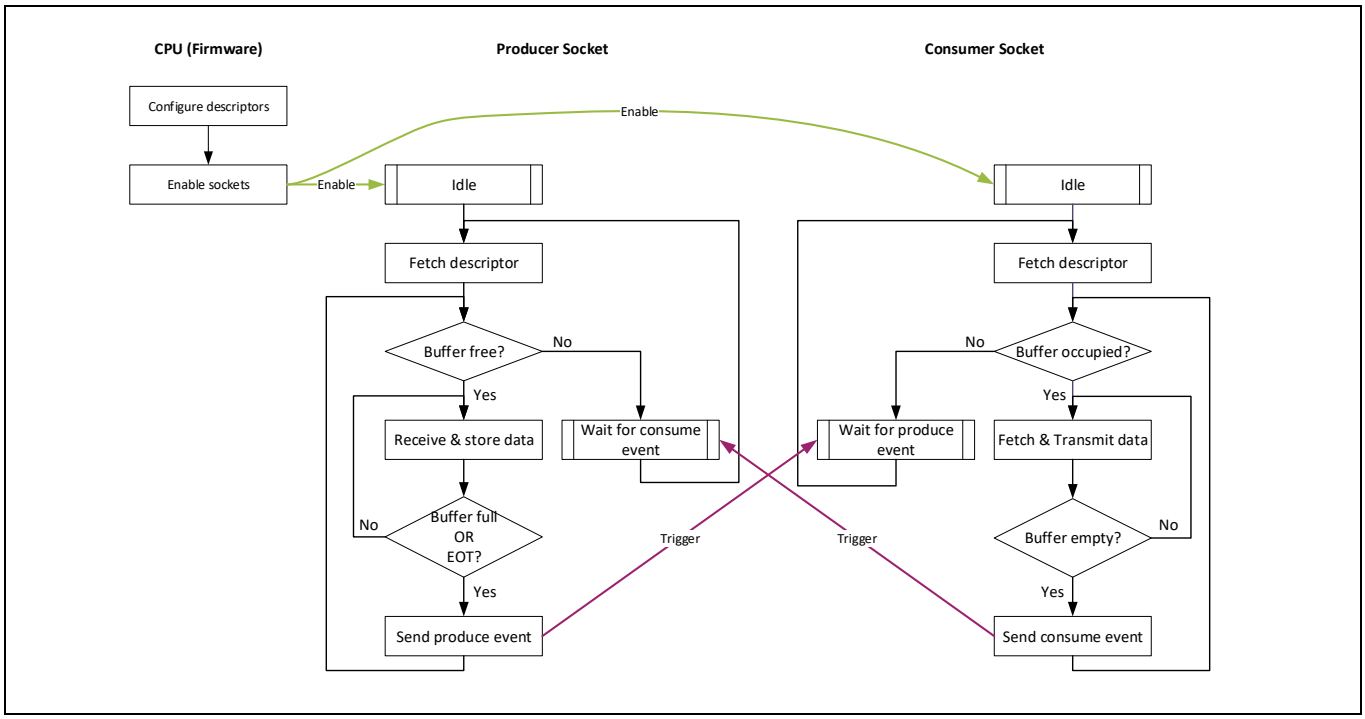


Figure 27-23. HBWSS DMA operation for IP to IP transfer

27.5.2.1.2 Memory to IP transfers

Figure 27-24 shows the flow associated with Memory to IP HBWSS DMA channels (for example, RAM to USB 3.2 device).

Once the firmware task receives the data that needs to be sent on the DMA channel, it calls the Cy_HBDma_Channel_GetBuffer() function to get a pointer to the DMA buffer through which data can be transferred. If the function call succeeds, an empty buffer is available. The firmware task can then fill the buffer with the needed data and call Cy_HBDma_Channel_CommitBuffer() to initiate the transfer of the data to the consumer socket. The API implementation notifies the consumer socket that data is now available, thus triggering the actual data transfer.

Once the consumer socket has read out all the data that was placed in the buffer, it updates the descriptor to show that the buffer is now empty and raises a consume event interrupt to the CPU. If no free buffer was available at the time of Cy_HBDma_Channel_GetBuffer() being called, the firmware task would have remained blocked until the buffer became available. The consume event interrupt is used to unblock the firmware task so that it can continue with the data transfer.

As **Figure 27-24** shows, DMA channels of this type require CPU intervention at the level of each DMA buffer (or descriptor). The size of the DMA buffer should be big enough to avoid very frequent interrupts and interventions from the firmware side.

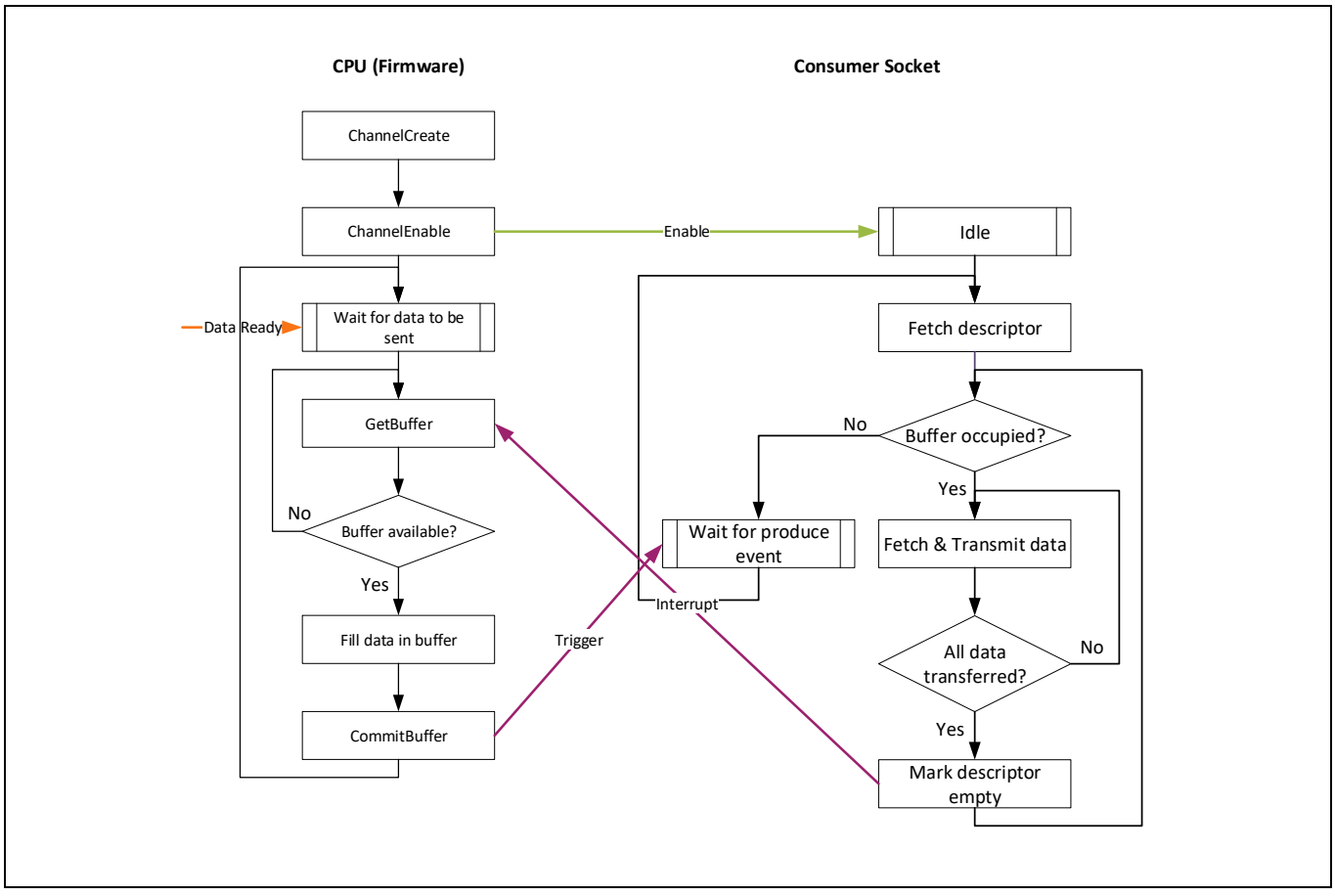


Figure 27-24. Interactions in memory to IP HBWSS DMA channel

27.5.2.1.3 IP to memory transfers

Figure 27-25 shows the interactions between the CPU (firmware task) and the producer socket in the case of IP to memory HBWSS transfers such as USB 3.2 device to RAM use case.

The firmware operation is similar to that in the memory to IP transfer case. Once the firmware has completed processing the data that was received, it needs to call the Cy_HBDma_Channel_DiscardBuffer() API to update the descriptor and notify the producer socket that the CPU is done with the data.

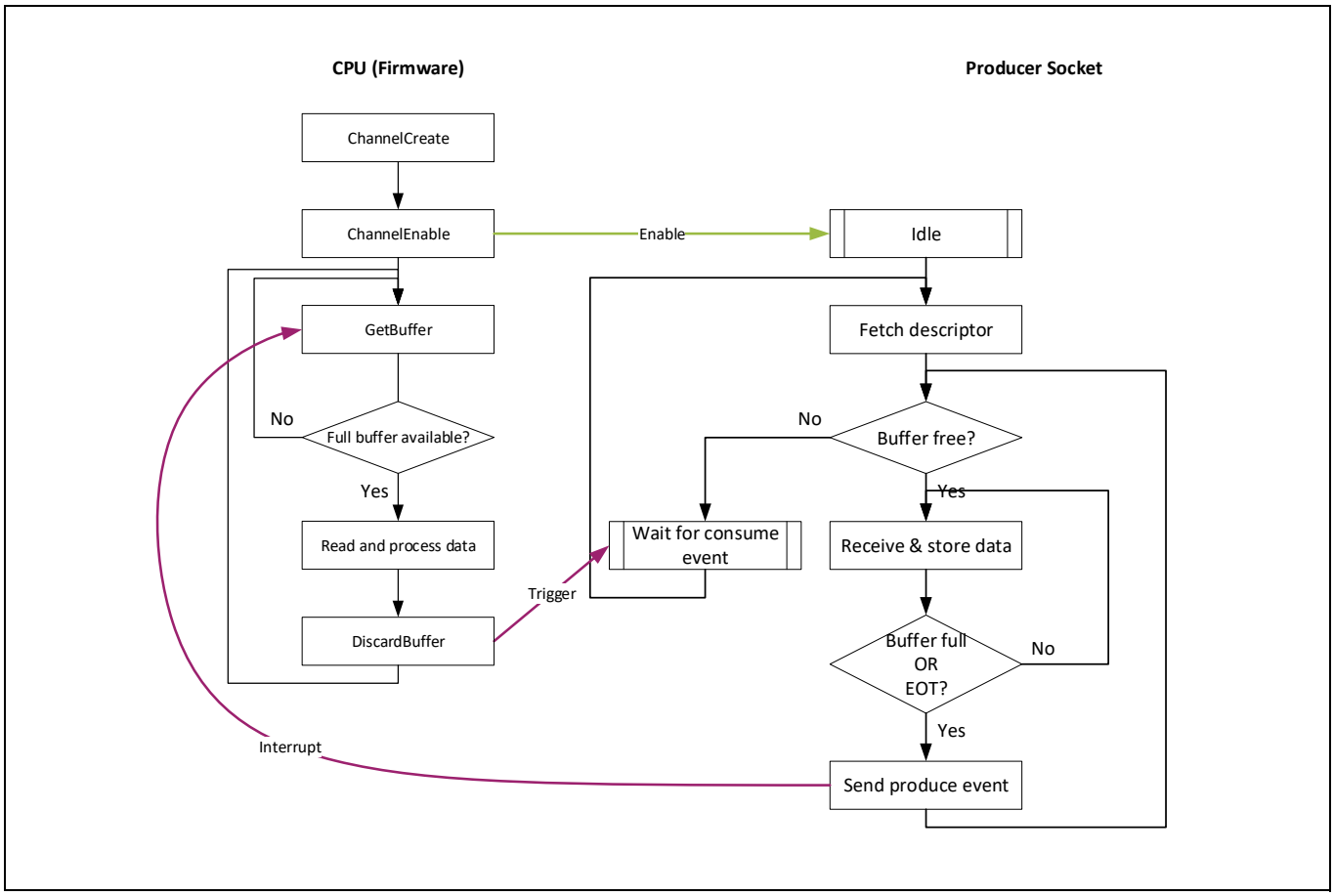


Figure 27-25. Interactions in IP to memory HBWSS DMA channels

27.5.3 CPUSS DMA operation

Unlike the HBWSS DMA, the CPUSS DMA controller is not capable of combining two trigger sources to enable data transfers. Since each DMA channel can take only one trigger input from the peripherals, these transfers require CPU intervention to enable the channel only when the trigger from the other peripheral. For example, in the case of a PDM to USB 2.0 data transfer, firmware will ensure that the DMA channel is only enabled when data is available in the PDM FIFO. The trigger from the USB endpoint would then be used to move the data into the endpoint memory only when space is available.

As in the case of HBWSS DMA, the CPUSS DMA supports IP (peripheral) to IP, Memory to IP and IP to Memory transfers.

27.5.3.1 IP to IP Transfers

IP to IP data transfers using the CPUSS DMA can have two cases:

1. Direct IP to IP transfer using a single channel.
2. A pair of IP to Memory and Memory to IP channels working together.

27.5.3.1.1 Direct IP to IP transfers

A single IP to IP DMA channel can be used in cases where the IP that is generating (producing) the data has sufficient internal buffering to hold one packet of data (for example, USB 2.0 endpoint memory). In this case, the CPU can wait for an interrupt from the IP indicating that data is available in the buffer and then enable the DMA channel to transfer the data.

The Cy_DMA_Channel_Init() API is used to configure the DMA channel as required. 2D transfer mode of the channel is used in all cases from/to USB 2.0 device IP. The X-count is set to match the maximum packet size of the endpoint and the trigger from the USB endpoint memory is selected to trigger one execution of the X-loop (move one USB data packet). The Y-count is set corresponding to the size of the data to be transferred. The channel will stop once it has finished Y-count number of loops and needs CPU intervention to be re-enabled.

The Cy_DMA_Channel_Enable() API is used to enable the DMA channel to complete data transfer as specified by one descriptor.

27.5.3.1.2 IP to IP transfer with two channels

In this case, two DMA channels are configured using the Cy_DMA_Channel_Init() API and enabled using the Cy_DMA_Channel_Enable() API. The first channel moves data from the IP which produces the data into RAM based buffers. The DMA trigger from the IP is used to control the data transfer. 2D transfer mode is used and the X-count is set to match the amount of buffering available within the IP block. Once Y-count number of loops have been completed, the DMA descriptor would be completed, the channel will get disabled and CPU will be interrupted.

On receiving the interrupt from the IP to memory DMA channel, the firmware task will enable the DMA channel which moves the data from the RAM buffer to the consumer IP. The DMA trigger from the consumer IP is used control the data transfer. The X-count will be set to match the amount of buffering available in the IP and Y-count will be set to match the total size of data to be transferred.

Figure 27-22 shows the interactions between the firmware tasks, the DMA controller and the two IPs in this case.

27.5.3.2 Memory to IP transfers

Memory to IP transfers using the CPUSS DMA are managed using the same flow used for the second memory to IP channel in Section **“IP to IP transfer with two channels”** on page 426.

27.5.3.3 IP to memory transfers

IP to memory transfers using the CPUSS DMA are managed using the same flow used for the first IP to memory channel in **"IP to IP transfer with two channels"** on page 426.

27.5.4 Handling short packets on the USB 2.0 device

As the CPUSS DMA controller does not have a notion of packet-based transfers, it will always attempt to transfer the amount of data specified in the DMA descriptor.

In the case of the USB 2.0 Device IP, there is a possibility of short length or zero length data packets to be transferred. When such packets are being transferred, special handling is required by the firmware to program the DMA descriptors to reflect the actual size of data to be transferred.

The following sections describe the procedure for handling short length packets (SLP) on OUT and IN endpoints. zero length packets (ZLP) can be treated as a special case of SLP.

27.5.4.1 Handling SLPs on USB 2.0 OUT endpoints

"Configuring USB to LVCMOS DMA Channels" on page 420 and **Figure 27-21** describe the procedure of handling data transfers on USB 2.0 OUT endpoints. The flow in **Figure 27-21** assumes that the size of the data packets available in the endpoint memory is known and matches the maximum packet size for the Endpoint. If this condition is true, the X-Count value in the descriptor can be set to match the maximum packet size; so that the full packet gets transferred on getting the trigger from the USB IP.

If the USB 2.0 IP receives a SLP (or ZLP) instead of a full-length packet, it does not raise a DMA trigger indicating that data is available. Instead, the IP saves the size of the data packet in the IEPM_ENDPOINT register corresponding to the endpoint and raises an interrupt to notify the CPU that a SLP has been received.

On receiving this interrupt, the DMA driver needs to do the following:

1. Find the current transfer status for the DMA channel using the
   Cy_DMAC_Channel_GetCurrentXloopIndex() and
   Cy_DMAC_Channel_GetCurrentYloopIndex() functions.
2. Compute the RAM buffer address at which the next data packet should be stored using these loop indices.
3. Disable the DMA Channel using Cy_DMAC_Channel_Disable().
4. Configure the DMA Channel with a new descriptor which moves only the available data in the endpoint memory to the RAM address computed in Step 2.
5. Re-enable the DMA channel using Cy_DMAC_Channel_Enable()
6. Trigger the DMA transfer using CPU trigger or by clearing the USB SLP interrupt status.

If the packet received was a ZLP, this procedure is not required and the interrupt can be cleared.

27.5.4.2 Handling SLPs on USB 2.0 IN endpoints

In the case of IN endpoints, the decision to send a SLP is based on the data that is received from another interface such as LVDS, PDM or SCB IP or made when the data to be sent is being generated by the firmware itself.

Once the firmware identifies that a short packet needs to be sent on the IN endpoint, it performs the following steps:

1. Identify the address at which the data to be sent is stored. This may require a procedure similar to that described in section **"Handling SLPs on USB 2.0 OUT endpoints"** on page 427.
2. Disable the DMA Channel using Cy_DMAC_Channel_Disable().
3. Configure the DMA Channel with a new descriptor which moves only the available data in the endpoint memory to the RAM address computed in Step 2.

4. Re-enable the DMA channel using Cy_DMAC_Channel_Enable(). Since the endpoint memory will be in the ready state, a trigger is not required and the transfer will happen automatically.

If the data to be sent on the IN endpoint spans over a few full-packets and a SLP, two DMA descriptors can be used to achieve this. The first descriptor will be programmed with X and Y counts corresponding the packet size and the number of full packets to be sent. It will be chained to the second descriptor which is configured as a 1D descriptor which copies the data corresponding to the short packet that is remaining.

27.6 HBWSS DMA application programming interface

The HBWSS DMA implementation is split between:

1. A set of driver functions that will be implemented as part of the PDL CAT1 library. The PDL functions allow get/set functionality on the DMA descriptor and socket structures.
2. An hbdmac middleware library which implements the DMA channel abstraction and functions used to manage end-to-end data flows.

This section documents the data structures and APIs which are provided as part of the hbdmac middleware library.

27.6.1 HBWSS DMA data structures

27.6.1.1 HBWSS DMA descriptor

Descriptors are data structures maintained in the HBWSS SRAM, which hold information about a data buffer which is used for DMA data transfer. The descriptor structure is defined as follows:

```
typedef struct Cy_HBDma_Descriptor_t
{
    uint8_t    *buffer_p;/* Pointer to buffer used. */
    uint32_t    sync;/* Points to the producer and consumer sockets. */
    uint32_t    chain;/* Next descriptor links. */
    uint32_t    size;/* Buffer size and current data count. */
} Cy_HBDma_Descriptor_t;
```

27.6.1.2 HBWSS DMA socket

A socket represents the interface through which a peripheral such as the USB 3.2 device or LVDS transfers data from/to the AXI bus. Each peripheral block can support multiple sockets so that it can handle multiple data streams, and each socket has the following information associated with it.

```
typedef struct Cy_HBDma_Socket_t
{
    uint32_t dscr_chain;/* Index of active descriptor. */
    uint32_t xfer_size;/* The transfer size requested for this socket. */
    uint32_t xfer_count;/* The completed transfer count for this socket. */
    uint32_t status;/* Socket configuration and status register. */
    uint32_t intr_stat;/* Interrupt status register. */
    uint32_t intr_mask;/* Interrupt mask register. */
    Cy_HBDma_Descriptor_t active_dscr;/* Copy of active descriptor. */
    uint32_t sck_event;/* Register to generate DMA event triggers. */
} Cy_HBDma_Socket_t;
```

27.6.1.3 HBWSS DMA transfer types

Since HBWSS DMA supports peripheral to peripheral, peripheral to memory and memory to peripheral data transfers; three different types of DMA channels are supported by the driver.

```
typedef enum Cy_HBDma_ChannelType_t
{
    CY_HBDMA_TYPE_IP_TO_IP = 0,/* Peripheral to peripheral DMA channel. */
    CY_HBDMA_TYPE_IP_TO_MEM,/* Peripheral to memory DMA channel. */
    CY_HBDMA_TYPE_MEM_TO_IP/* Memory to peripheral DMA channel. */
} Cy_HBDma_ChannelType_t;
```

27.6.1.4 HBWSS DMA channel interrupt notifications

A set of interrupt notifications can be enabled for each HBWSS DMA channel. The available interrupt notifications are listed in the Cy_HBDma_IntrType_t enumeration. Each interrupt notification corresponds to a bit position in a 32-bit interrupt mask.

```
typedef enum Cy_HBDma_IntrType_t
{
    CY_HBDMA_CB_XFER_CPLT,/* Transfer completed. */
    CY_HBDMA_CB_PROD_EVENT,/* Buffer received from producer. */
    CY_HBDMA_CB_CONS_EVENT, /* Buffer drained by the consumer. */
    CY_HBDMA_CB_ERROR,/* DMA transfer error raised by the adapter. */
    CY_HBDMA_CB_SUSPENDED/* DMA transfer has been suspended. */
} Cy_HBDma_IntrType_t;
```

27.6.1.5 HBWSS DMA interrupt notification

The following callback type is used to receive notifications of DMA interrupts from the HBWSS DMA driver.

```
typedef void (*Cy_HBDma_Callback_t) (
        Cy_HBDma_Channel_t *handle,/* Handle to the DMA channel */
        Cy_HBDma_IntrType_t type,/* The type of notification */
        void *user_ctx/* User context for the callback */
        );
```

27.6.1.6 HBWSS DMA channel status

This enumeration lists the various state values possible for an HBWSS DMA channel.

```
typedef enum Cy_HBDma_ChannelState_t
{
    CY_U3P_DMA_NOT_CONFIGURED = 0,/* DMA channel has not been initialized. */
    CY_U3P_DMA_CONFIGURED,/* DMA channel has been created and configured */
    CY_U3P_DMA_ACTIVE,/* DMA Channel has active transaction going on */
    CY_U3P_DMA_ERROR /* Channel has encountered an error. */
} Cy_HBDma_ChannelState_t;
```

27.6.1.7 HBWSS DMA channel configuration structure

To configure a DMA channel as shown in **Figure 27-18**, the create function requires information about the producer and consumer sockets to be used, the number of DMA buffers (also descriptors) to be used and the size of each buffer. The following structure represents the parameters that are required to create a HBWSS DMA channel:

```
typedef struct Cy_HBDma_ChannelConfig_t
{
    uint16_t size;    /* Size of DMA buffers to be used. */
    uint16_t count;   /* Number of DMA buffers to be used. */
    Cy_HBDma_ChannelType_t ch_type;/* Type of DMA channel to be created. */
    uint16_t prodSckCount;/* Number of producer sockets: 0, 1 or 2. */
    uint16_t consSckCount;/* Number of consumer sockets: 0, 1 or 2. */
    Cy_HBDma_SocketID_t prod_sck[2];/* ID of the producer socket(s). */
    Cy_HBDma_SocketID_t cons_sck[2];/* ID of the consumer socket(s). */
    uint32_t event_enable;/* Enable for DMA event triggers. */
    uint32_t intr_enable;/* Enable for DMA interrupts. */
    Cy_HBDma_Callback_t intr_cb;/* Interrupt callback function pointer. */
    void *user_ctx;  /* User specific context to pass to the callback. */
} Cy_HBDma_ChannelConfig_t;
```

27.6.1.8 HBWSS DMA channel structure

The HBWSS DMA channel structure encapsulates all the configuration and status information associated with a DMA channel. A pointer to this structure is passed to all the HBWSS DMA channel API. The structure is defined as follows:

```
typedef struct Cy_HBDma_Channel_t
{
    Cy_HBDma_Context *context_p;/* HBWSS DMA manager context. */
    Cy_HBDma_ChannelType_t type;/* The type of the DMA channel */
    uint16_t prodSckCount;/* Number of producer sockets: 0, 1 or 2. */
    uint16_t consSckCount;/* Number of consumer sockets: 0, 1 or 2. */
    Cy_HBDma_SocketID_t prodSckId[2];/* The producer (ingress) socket IDs */
    Cy_HBDma_SocketID_t consSckId[2];/* The consumer (egress) socket IDs */
    uint16_t size;           /* The buffer size associated with the channel */
    uint16_t count;         /* Number of buffers for the channel */
    uint16_t firstDscrIndex; /* Head of the descriptor chain for this channel */
    uint16_t curProdDscrIndex[2];/* Active descriptor for the producer socket(s). */
    uint16_t curConsDscrIndex[2];/* Active descriptor for the consumer socket(s). */
    uint16_t overrideDscrIndex;/* Descriptor reserved for send/receive transfers */
    uint32_t xferSize;/* Current transfer size */
    uint32_t notification;/* Notifications enabled for this channel */
    Cy_HBDma_Callback_t cbFunc;/* Function to be called on channel events */
    void *cbCtx;     /* User specific context to pass to the callback. */
    Cy_HBDma_ChannelState_t state;/* Current state of the DMA channel. */
} Cy_HBDma_Channel_t;
```

27.6.1.9 DMA buffer status structure

The DMA buffer status structure is used in cases where the CPU/firmware is involved in producing or consuming the data being transferred through a DMA channel. This structure encapsulated all information associated with a particular DMA buffer.

```
typedef struct Cy_HBDma_BufStat_t
{
    uint8_t *buffer_p;/* Pointer to the data buffer. */
    uint16_t size;/* Actual size of the buffer in bytes. */
    uint16_t count;/* Byte count of valid data in buffer. */
} Cy_HBDma_BufStat_t;
```

27.6.1.10 HBWSS DMA context structure

The context structure holds status information for the HBWSS DMA driver.

```
typedef struct Cy_HBDma_Context_t
{
    Cy_HBDma_DscrList_t *dscrPool_p;/* HBWSS DMA descriptor pool. */
    Cy_HBDma_BufMgr_t *bufMgr_p;/* HBWSS DMA buffer manager. */
    uint32_t socketUsed[IP_COUNT];/* Bit-map showing whether sockets are in use. */
} Cy_HBDma_Context_t;
```

27.6.2 HBWSS DMA functions

This section documents the functionality provided by the HBDMAC middleware library.

27.6.2.1 HBWSS DMA descriptor and buffer management

The HBWSS DMA descriptors are data structures maintained in the lower address areas of the HBWSS SRAM. Each descriptor uses a 16-byte area of the SRAM and has the format defined in **"HBWSS DMA descriptor"** on page 428. The descriptors point to another area in HBWSS SRAM which is used to hold the data as it streams through the EZ-USB™ FX10 device.

The programmer shall define the number and size of data buffers to be used to hold the data corresponding to a DMA channel when the channel is being created. Some DMA channels are created/managed by other Infineon owned assets like the FX3USBD middleware stack; and others are created/managed by customer specific code modules such as a USB Video function implementation.

Due to this mix, it is not desirable to statically define all the descriptors and data buffers that will be needed by the application. The limited SRAM space available means that the DMA buffers and descriptors need to be allocated with low management overhead. Also, there is a need to ensure that each DMA data buffer starts on a 32-byte aligned address.

Keeping these factors in consideration, the HBWSS DMA driver provides a set of functions that manages the descriptors and DMA buffers for the application.

27.6.2.1.1 DMA descriptor related functions

The complete list of available HBWSS descriptors are maintained in a free list and allocated/freed from it on demand. The status of each descriptor is tracked using a single bit which identifies whether it is currently in use or not.

- DMA descriptor free list structure

The DMA descriptor free list structure is defined as follows:

```
typedef struct Cy_HBDma_DscrList_t
{
    uint16_t availCount;/* Count of available descriptors.        */
    uint16_t curDscr;/* Position of last allocated descriptor. */
    uint32_t *dscrStatus;/* Bit map representing state of each descriptor. */
} Cy_HBDma_DscrList_t;
```

- DMA descriptor free list creation

The Cy_HBDma_DscrList_Create function is used to initialize the descriptor free list.

```
void
Cy_HBDma_DscrList_Create (
        Cy_HBDma_DscrList_t *list_p,/* Descriptor free list to be initialized. */
        uint16_t maxDscrCount/* Maximum number of descriptors allowed. */
);
```

- DMA descriptor free list deletion

The Cy_HBDma_DscrList_Destroy function is used to de-initialize the descriptor free list. This function is only expected to be called in case the complete DMA driver is being shut down leading to a device reset or control transfer to another application.

```
void
Cy_HBDma_DscrList_Destroy (
        Cy_HBDma_DscrList_t *list_p/* Descriptor free list to be de-initialized. */
);
```

- DMA descriptor allocation

The Cy_HBDma_DscrList_Get function is used to obtain a free descriptor from the free list.

```
Cy_HBDma_Status_t
Cy_HBDma_DscrList_Get (
        Cy_HBDma_DscrList_t *list_p,/* Descriptor free list. */
        uint16_t *dscrIndex_p/* Pointer to be filled with descriptor index. */
);
```

- DMA descriptor de-allocation

The Cy_HBDma_DscrList_Put function is used to free up a previously allocated descriptor.

```
Cy_HBDma_Status_t
Cy_HBDma_DscrList_Put (
        Cy_HBDma_DscrList_t *list_p,/* Descriptor free list. */
        uint16_t dscrIndex/* Index of descriptor to be freed. */
);
```

27.6.2.1.2 DMA buffer related functions

HBWSS DMA buffers have to be allocated from the HBWSS SRAM. The AXI burst mode used for the DMA transfers also requires that each DMA buffer start at an address which is 32-byte aligned. Almost all of the allocations are expected to happen at init time and the total number of buffers allocated is expected to be less than 100. A custom memory allocator implementation which has low memory overhead is to be implemented.

- DMA buffer pool creation

The Cy_HBDma_BufMgr_Create function is used to initialize the DMA buffer allocator.

```
Cy_HBDma_Status_t
Cy_HBDma_BufMgr_Create (
        Cy_HBDma_BufMgr_t *mgr_p,/* Buffer allocator context structure. */
        uint32_t *baseAddr,/* Base address of DMA buffer region. */
        uint32_t regionSize/* Size of the DMA buffer region in bytes. */
);
```

- DMA buffer pool deletion

The Cy_HBDma_BufMgr_Destory function is used to de-initialize the DMA buffer allocator. This is only expected to happen if the DMA driver is being shut down prior to a device reset or control transfer to another application.

```
Cy_HBDma_Status_t
Cy_HBDma_BufMgr_Destroy (
        Cy_HBDma_BufMgr_t *mgr_p,/* Buffer allocator context structure. */
        bool freeAll /* Whether all buffers should be freed. */
);
```

- DMA buffer allocation

The Cy_HBDma_BufMgr_Alloc function is used to allocate a DMA buffer from the buffer manager.

```
void *
Cy_HBDma_BufMgr_Alloc (
        Cy_HBDma_BufMgr_t *mgr_p,/* Buffer allocator context structure. */
        uint16_t bufferSize/* Size of the DMA buffer required in bytes. */
);
```

- DMA buffer de-allocation

The Cy_HBDma_BufMgr_Free function is used to free up a DMA buffer that was previously allocated from the buffer manager.

```
Cy_HBDma_Status_t
Cy_HBDma_BufMgr_Free (
        Cy_HBDma_BufMgr_t *mgr_p,/* Buffer allocator context structure. */
        uint32_t *buffer_p/* Pointer to buffer to be freed. */
);
```

27.6.2.2 HBWSS DMA manager functions

The HBWSS DMA manager needs to be initialized and provided with handles to the descriptor free list as well as the buffer manager, so that it can allocate the descriptors and buffer as needed by the channels.

27.6.2.2.1 DMA manager initialization

The Cy_HBDma_Init() function is used to initialize the HBWSS DMA driver.

```
Cy_HBDma_Status_t
Cy_HBDma_Init (
    Cy_HBDma_Context *context_p,/* HBWSS DMA driver context to be initialized. */
    Cy_HBDma_DscrList_t *dscrPool_p,/* HBWSS DMA descriptor pool. */
    Cy_HBDma_BufMgr_t *bufMgr_p/* HBWSS DMA buffer manager. */
);
```

27.6.2.2.2 DMA manager de-initialization

The Cy_HBDma_DeInit() function is used to de-initialize the HBWSS DMA driver.

```
Cy_HBDma_Status_t
Cy_HBDma_DeInit (
    Cy_HBDma_Context *context_p/* HBWSS DMA driver context to be de-initialized. */
);
```

27.6.2.3 DMA channel related functions

27.6.2.3.1 HBWSS DMA channel creation

The Cy_HBDma_Channel_Create function is used to create and configure the DMA channel.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_Create (
    Cy_HBDma_Context *dmaMgr_p,/* HBWSS DMA manager context pointer. */
    Cy_HBDma_Channel *handle,/* HBWSS DMA channel handle to be initialized. */
    Cy_HBDma_ChannelConfig_t *config/* Channel configuration parameters. */
);
```

The Channel Create API allocates a set of DMA data buffers of the desired size on the HBWSS SRAM. It also allocates the same number of descriptors from the free list and configures them to point to the DMA buffers as well as the producer and consumer sockets associated with the channel. In the case of an IP_TO_IP DMA channel, the event notifications from the producer socket to the consumer socket and vice versa are enabled. The function also unmasks the interrupts specified by the user and registers an ISR (where required) which will relay the interrupt notifications to the user provided callback function.

This function only configures the DMA resources and does not enable the sockets for data transfer.

27.6.2.3.2 HBWSS DMA channel deletion

The Cy_HBDma_Channel_Destroy function is used to de-initialize and free up an HBWSS DMA channel that was previously created.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_Destroy (
    Cy_HBDma_Channel *handle/* HBWSS DMA channel handle to be de-initialized. */
);
```

27.6.2.3.3 Enabling HBWSS DMA channel

Once a DMA channel has been created, it needs to be enabled for data transfer using the Cy_HBDma_Channel_Enable function. This function enables the producer and consumer sockets associated with the channel. The xferSize parameter passed to the function specifies whether the sockets should be configured a finite amount of data. If the xferSize parameter is set to zero, the sockets will be configured to move an infinite amount of data (stream mode of transfer).

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_Enable (
    Cy_HBDma_Channel *handle,/* HBWSS DMA channel handle to be enabled. */
    uint32_t xferSize/* Size of data to be transferred by the channel. */
);
```

27.6.2.3.4 Disabling HBWSS DMA channel

An active DMA channel can be disabled using the Cy_HBDma_Channel_Disable function. This function disables the producer and consumer sockets associated with the channel after ensuring that they are not currently transferring any data. If either of the sockets is actively transferring data, the function configures the socket to wait at the end of the current descriptor and then disables it.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_Disable (
    Cy_HBDma_Channel *handle/* HBWSS DMA channel handle to be disabled. */
);
```

27.6.2.3.5 Resetting HBWSS DMA channel

An active DMA channel can be reset using the Cy_HBDma_Channel_Reset function. This function forcibly disables the producer and consumer sockets and restores all descriptors associated with the channel to their default state (buffers are free). Resetting a DMA channel while it is transferring data can result in a protocol error in the IP which is providing data to or taking data from the channel.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_Reset (
    Cy_HBDma_Channel *handle/* HBWSS DMA channel handle to be reset. */
);
```

27.6.2.3.6 Memory to IP HBWSS DMA channel transfer

Some HBWSS DMA channels are created for Memory to IP data transfers. In this case, it is the responsibility of the CPU (firmware) to provide the data that will be received by the consumer socket. Since there is no IP block which checks the descriptor status and initiates the transfer, API calls are provided to get the data buffer pointer and to then notify the consumer IP that data is ready in the buffer.

The Cy_HBDma_Channel_GetBuffer function shall first be used to get a pointer to the DMA buffer and its size. Once this information is available, the caller can proceed to fill the buffer with the desired data to be transmitted.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_GetBuffer (
    Cy_HBDma_Channel *handle,/* HBWSS DMA channel handle. */
    Cy_HBDma_BufStat_t *bufStat_p/* Pointer to provide the buffer status through. */
);
```

The Cy_HBDma_Channel_CommitBuffer function shall be used by the CPU to notify the consumer socket that data is available in the buffer and that it can proceed to transfer the data.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_CommitBuffer (
    Cy_HBDma_Channel *handle,/* HBWSS DMA channel handle. */
    Cy_HBDma_BufStat_t *bufStat_p/* Updated buffer status including data size. */
);
```

Another option is provided to allow cases where the firmware application wants to transfer data that is already present in the HBWSS SRAM to out through the consumer socket. This operation is only allowed when the DMA channel is not in the enabled state.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_SendData (
    Cy_HBDma_Channel *handle,/* HBWSS DMA channel handle. */
    uint8_t *dataBuf_p,/* Pointer to data buffer. */
    uint16_t dataSize/* Size of data to be sent in bytes. */
);
```

27.6.2.3.7 IP to memory HBWSS DMA channel transfer

Some HBWSS DMA channels are created for IP to Memory data transfers. In this case, it is the responsibility of the CPU (firmware) to consume the data that is provided by the producer socket. Since there is no IP block which checks the descriptor status and initiates the transfer, API calls are provided to get the data buffer pointer to read the data from and to then update the producer socket that the data has been drained.

The Cy_HBDma_Channel_GetBuffer function defined above is used to fetch details about the active data buffer. If the count in the bufStat_p structure indicates the presence of data, the caller can read and process the data from the buffer.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_GetBuffer (
    Cy_HBDma_Channel *handle,/* HBWSS DMA channel handle. */
    Cy_HBDma_BufStat_t *bufStat_p/* Pointer to provide the buffer status through. */
);
```

The Cy_HBDma_Channel_DiscardBuffer function shall be used by the CPU to notify the producer socket that the data buffer previously generated has been drained and that the corresponding memory can be over-written.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_DiscardBuffer (
    Cy_HBDma_Channel *handle,/* HBWSS DMA channel handle. */
    Cy_HBDma_BufStat_t *bufStat_p/* Pointer to the buffer being freed up. */
);
```

Another option is provided to allow cases where the firmware application wants to receive a specified amount of data into a specific HBWSS SRAM buffer. This operation is only allowed when the DMA channel is not in the enabled state.

```
Cy_HBDma_Status_t
Cy_HBDma_Channel_ReceiveData (
    Cy_HBDma_Channel *handle,/* HBWSS DMA channel handle. */
    uint8_t *dataBuf_p,/* Pointer to data buffer. */
    uint16_t dataSize/* Size of data to be received in bytes. */
);
```

27.7 CPUSS DMA functions

The functions that are relevant to the DMA data flow management on EZ-USB™ FX10 are shown in **Table 27-4**.

Table 27-4. CPUSS DMA driver APIs

API	Category	Description
Cy_DMA_Enable	DataWire (DW)	Enable the DataWire DMA block
Cy_DMA_Disable	DW	Disable the DataWire DMA block
Cy_DMA_Channel_Init	DW	Initialize a DataWire DMA channel with the descriptor and optionally enables it
Cy_DMA_Channel_DeInit	DW	De-initialize a DataWire DMA channel
Cy_DMA_Channel_Enable	DW	Enable a DataWire DMA channel
Cy_DMA_Channel_Disable	DW	Disable a DataWire DMA channel
Cy_DMA_Descriptor_Init	DW	Configure a DataWire DMA descriptor with source, destination and size parameters
Cy_DMA_Descriptor_DeInit	DW	Clears a DataWire DMA descriptor structure
Cy_DMAC_Enable	DMA controller (DMAC)	Enables a DMA controller block
Cy_DMAC_Disable	DMAC	Disables a DMA controller block
Cy_DMAC_Channel_Init	DMAC	Initialize a DMAC channel with the descriptor and optionally enables it
Cy_DMAC_Channel_DeInit	DMAC	De-initialize a DMAC channel
Cy_DMAC_Channel_Enable	DMAC	Enable a DMAC channel
Cy_DMAC_Channel_Disable	DMAC	Disable a DMAC channel
Cy_DMAC_Descriptor_Init	DMAC	Configure a DMAC descriptor with source, destination and size parameters
Cy_DMAC_Descriptor_DeInit	DMAC	Clears a DMAC descriptor structure

Section F: Digital subsystem

This section encompasses the following chapters:

- **"Serial communications block (SCB)"** on page 439
- **"Serial memory interface (SMIF)"** on page 506
- **"CAN FD controller"** on page 525
- **"Timer, counter, and PWM (TCPWM)"** on page 600
- **"Inter-IC sound bus"** on page 639
- **"PDM-PCM converter"** on page 651

Top level architecture

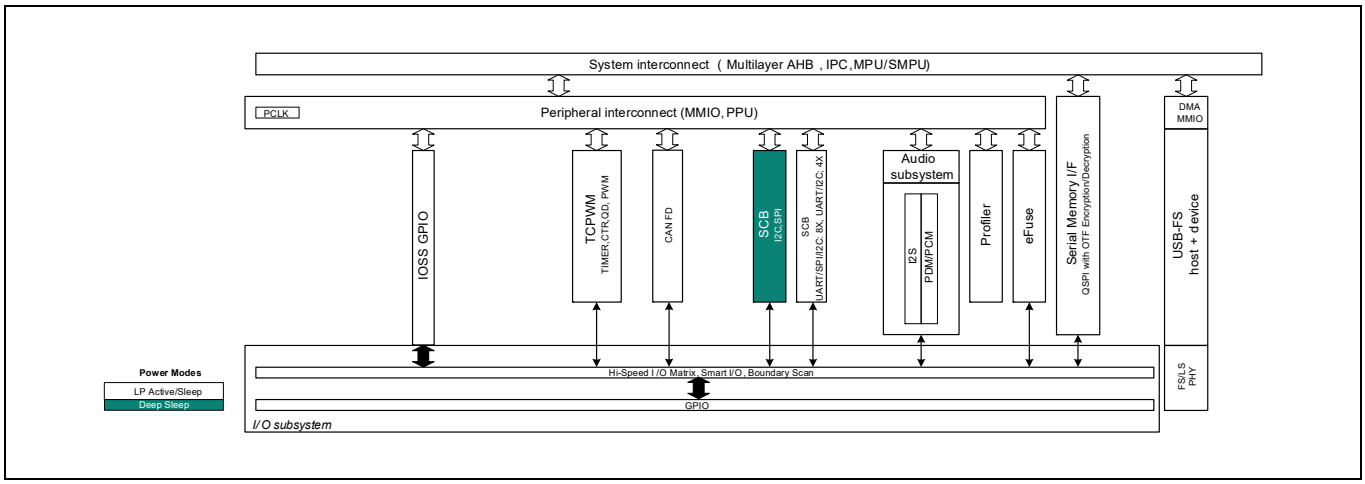


Figure F-1. Digital system block diagram

28 Serial communications block (SCB)

The serial communications block (SCB) supports three serial communication protocols: Serial Peripheral Interface (SPI), Universal Asynchronous Receiver Transmitter (UART), and Inter Integrated Circuit (I^2C or IIC). Only one of the protocols is supported by an SCB at any given time. The number of SCBs in a EZ-USB™ FX10 MCU varies by part number; refer the device datasheet to determine number of SCBs and the SCB pin locations. Not all SCBs support all three modes (SPI, UART, and I2C); refer the device datasheet to determine which modes are supported by which SCBs. Not all SCBs operate in Deep Sleep, refer the device datasheet to determine which SCBs operate in Deep Sleep.

28.1 Features

The SCB supports the following features:

- Standard SPI master and slave functionality with Motorola, Texas Instruments, and National Semiconductor protocols
- Standard UART functionality with SmartCard reader, Local Interconnect Network (LIN), and IrDA protocols
  - Standard LIN slave functionality with LIN v1.3 and LIN v2.1/2.2 specification compliance
- Standard I^2C master and slave functionality
- Trigger outputs for connection to DMA
- Multiple interrupt sources to indicate status of FIFOs and transfers
- Features available only on Deep Sleep-capable SCB:
  - EZ mode for SPI and I^2C slaves; allows for operation without CPU intervention
  - CMD_RESP mode for SPI and I^2C slaves; allows for operation without CPU intervention
  - Low-power (Deep Sleep) mode of operation for SPI and I^2C slaves (using external clocking)
  - Deep Sleep wakeup on I^2C slave address match or SPI slave selection

28.2 Architecture

The operation modes supported by SCB are described in the following sections.

28.2.1 Buffer modes

Each SCB has 256 bytes of dedicated RAM for transmit and receive operation. This RAM can be configured in three different modes (FIFO, EZ, or CMD_RESP). The following sections give a high-level overview of each mode. The sections on each protocol will provide more details.

- Masters can only use FIFO mode
- I^2C and SPI slaves can use all three modes

Note: EZ Mode and CMD Response Mode are available only on the Deep Sleep-capable SCB.

- UART only uses FIFO mode

28.2.2 FIFO mode

In this mode the RAM is split into two 128-byte FIFOs, one for transmit (TX) and one for receive (RX). The FIFOs can be configured to be 8 bits x 128 elements or 16 bits × 64 elements; this is done by setting the BYTE_MODE bit in the SCB control register.

FIFO mode of operation is available only in Active and Sleep power modes. However, the I^2C address or SPI slave select can be used to wake the device from Deep Sleep on the Deep Sleep-capable SCB.

Statuses are provided for both the RX and TX FIFOs. There are multiple interrupt sources available, which indicate the status of the FIFOs, such as full or empty; see **"SCB interrupts"** on page 496.

28.2.2.1 EZ mode

In easy (EZ) mode the RAM is used as a single 256-byte buffer. The external master sets a base address and reads and writes start from that base address.

EZ Mode is available only for SPI slave and I^2C slave. It is available only on the Deep Sleep capable SCB.

EZ mode is available in Active, Sleep, and Deep Sleep power modes.

Note: *This document discusses hardware implementation of the EZ mode; for the firmware implementation, see the PDL.*

28.2.2.2 CMD_RESP mode

Command Response (CMD_RESP) mode is similar to EZ mode except that the base address is provided by the CPU not the external master.

CMD_RESP mode is available only for SPI slave and I^2C slave. It is available only on the Deep Sleep-capable SCB.

CMD_RESP mode operation is available in Active, Sleep, and Deep Sleep power modes.

28.2.3 Clocking modes

The SCB can be clocked either by an internal clock provided by the peripheral clock dividers (referred to as clk_scb in this document), or it can be clocked by the external master.

- UART, SPI master, and I^2C master modes must use clk_scb
- Only SPI slave and I^2C slave can use the clock from and external master, and only the Deep Sleep capable SCB supports this

Internally- and externally-clocked slave functionality is determined by two register fields of the SCB CTRL register:

- EC_AM_MODE indicates whether SPI slave selection or I^2C address matching is internally ('0') or externally ('1') clocked
- EC_OP_MODE indicates whether the rest of the protocol operation (besides SPI slave selection and I^2C address matching) is internally ('0') or externally ('1') clocked

Note:

- FIFO mode supports an internally- or externally-clocked address match (EC_AM_MODE is '0' or '1'); however, data transfer must be done with internal clocking. (EC_OP_MODE is '1')
- EZ and CMD_RESP modes are supported with externally clocked operation (EC_OP_MODE is '1')

Table 28-1 provides an overview of the clocking and buffer modes supported for each communication mode.

Table 28-1. Clock mode compatibility

Mode	Internally clocked (IC)			Externally clocked (EC) (Deep Sleep SCB only)		
	FIFO	EZ	CMD_RESP	FIFO	EZ	CMD_RESP
I^2C master	Yes	No	No	No	No	No
I^2C slave	Yes	Yes	No	Yes[a)]	Yes	Yes
I^2C 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
UART transmitter	Yes	No	No	No	No	No
UART receiver	Yes	No	No	No	No	No

a) In Deep Sleep mode the external-clocked logic can handle slave address matching, it then triggers an interrupt to wake up the CPU. The slave can be programmed to stretch the clock, or NACK until internal logic takes over. This applies only to the Deep Sleep-capable SCB.

b) In Deep Sleep mode the external-clocked logic can handle slave selection detection, it then triggers an interrupt to wake up the CPU. Writes will be ignored and reads will return 0xFF until internal logic takes over. This applies only to the Deep Sleep-capable SCB.

Table 28-2. Clock configuration and mode support

Mode	EC_AM_MODE is '0'; EC_OP_MODE is '0'	'EC_AM_MODE is '1'; EC_OP_MODE is '0'	'EC_AM_MODE is '1'; EC_OP_MODE is '1'
FIFO mode	Yes	Yes	No
EZ mode	Yes	Yes	Yes
CMD_RESP mode	No	No	Yes

28.3 Serial peripheral interface (SPI)

The SPI protocol is a synchronous serial interface protocol. Devices operate in either master or slave mode. The master initiates the data transfer. The SCB supports single-master-multiple-slaves topology for SPI. Multiple slaves are supported with individual slave select lines.

28.3.1 Features

- Supports master and slave functionality
- Supports three types of SPI protocols:
  - Motorola SPI – modes 0, 1, 2, and 3
  - Texas Instruments SPI, with coinciding and preceding data frame indicator – mode 1 only
  - National Semiconductor (MicroWire) SPI – mode 0 only
- Master supports up to four slave select lines
  - Each slave select has configurable active polarity (high or low)
  - Slave select can be programmed to stay active for a whole transfer, or just for each byte
- Master supports late sampling for better timing margin
- Master supports continuous SPI clock
- Data frame size programmable from 4 bits to 16 bits
- Programmable oversampling
- MSb or LSb first
- Median filter available for inputs
- Supports FIFO mode
- Supports EZ mode (slave only) and CMD_RESP mode (slave only) on the Deep Sleep-capable SCB

28.3.2 General description

Figure 28-1 illustrates an example of SPI master with four slaves.

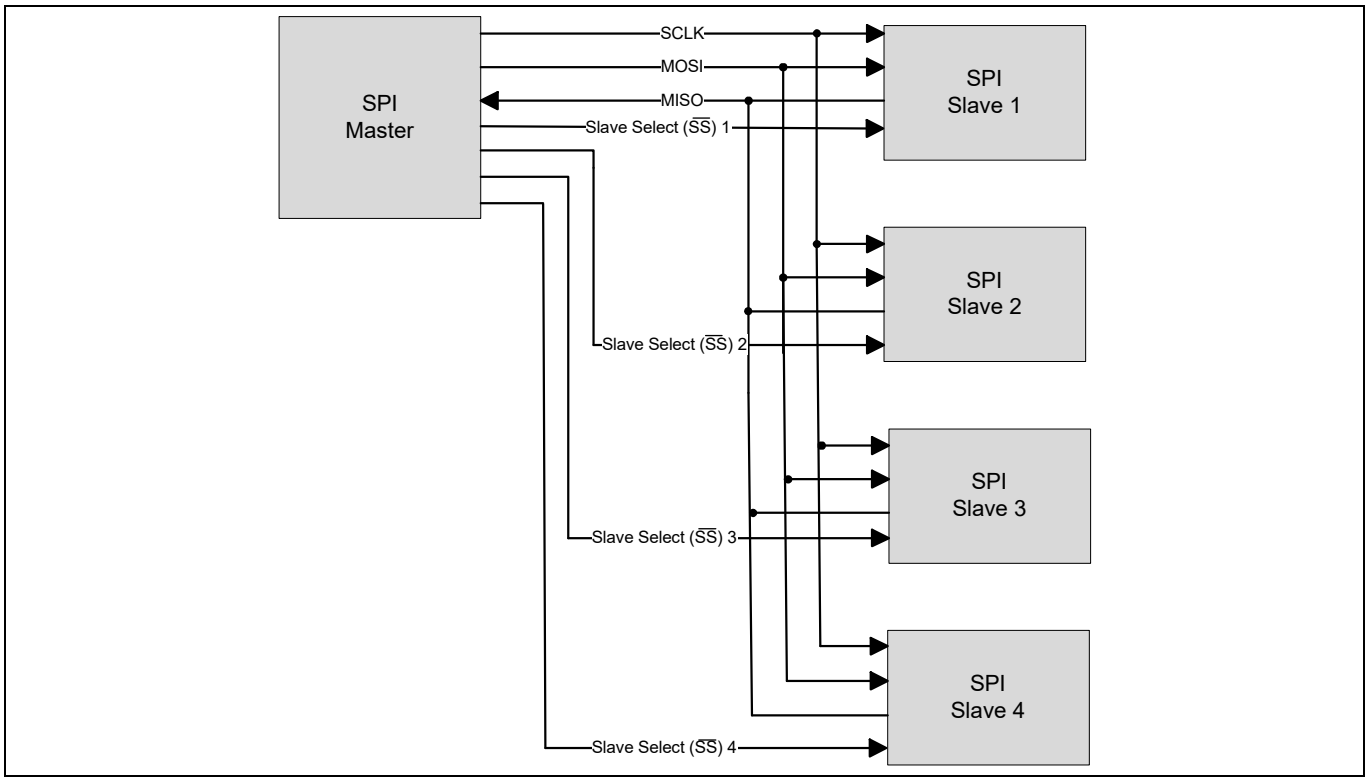


Figure 28-1. SPI example

A standard SPI interface consists of four signals as follows.

- SCLK: Serial clock (clock output from the master, input to the slave)
- MOSI: Master-out-slave-in (data output from the master, input to the slave)
- MISO: Master-in-slave-out (data input to the master, output from the slave)
- Slave Select ($\overline{SS}$): Typically an active low signal (output from the master, input to the slave)

A simple SPI data transfer involves the following: the master selects a slave by driving its $\overline{SS}$ line, then it drives data on the MOSI line and a clock on the SCLK line. The slave uses either of the edges of SCLK depending on the configuration to capture the data on the MOSI line; it also drives data on the MISO line, which is captured by the master.

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 4 to 16 bits. The serial data can be transmitted either most significant bit (MSb) first or least significant bit (LSb) first.

Three different variants of the SPI protocol are supported by the SCB:

- Motorola SPI: This is the original SPI protocol
- Texas Instruments SPI: A variation of the original SPI protocol, in which data frames are identified by a pulse on the $\overline{SS}$ line
- National Semiconductors SPI: A half-duplex variation of the original SPI protocol

28.3.3 SPI modes of operation

28.3.3.1 Motorola SPI

The original SPI protocol was defined by Motorola. It is a full duplex protocol. Multiple data transfers may happen with the $\overline{SS}$ line held at '0'. When not transmitting data, the $\overline{SS}$ line is held at '1'.

Clock modes of Motorola SPI

The Motorola SPI protocol has four different clock modes based on how data is driven and captured on the MOSI and MISO lines. These modes are determined by clock polarity (CPOL) and clock phase (CPHA).

Clock polarity determines the value of the SCLK line when not transmitting data. CPOL = '0' indicates that SCLK is '0' when not transmitting data. CPOL = '1' indicates that SCLK is '1' when not transmitting data.

Clock phase determines when data is driven and captured. CPHA = 0 means sample (capture data) on the leading (first) clock edge, while CPHA = 1 means sample on the trailing (second) clock edge, regardless of whether that clock edge is rising or falling. With CPHA = 0, the data must be stable for setup time before the first clock cycle.

- Mode 0: CPOL is '0', CPHA is '0': Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK
- Mode 1; CPOL is '0', CPHA is '1': Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK
- Mode 2: CPOL is '1', CPHA is '0': Data is driven on a rising edge of SCLK. Data is captured on a falling edge of SCLK
- Mode 3: CPOL is '1', CPHA is '1': Data is driven on a falling edge of SCLK. Data is captured on a rising edge of SCLK

Figure 28-2 illustrates driving and capturing of MOSI/MISO data as a function of CPOL and CPHA.

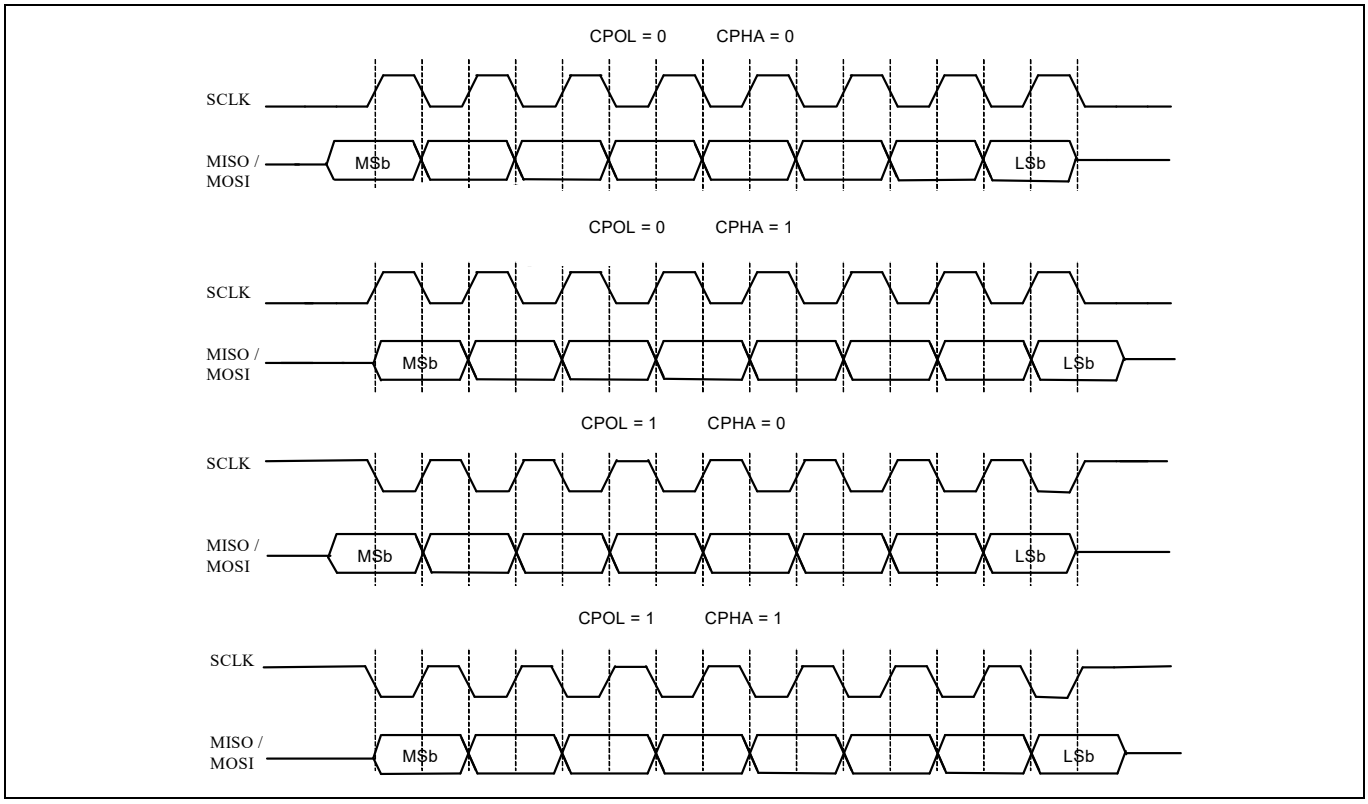


Figure 28-2. SPI Motorola, 4 modes

Figure 28-3 illustrates a single 8-bit data transfer and two successive 8-bit data transfers in mode 0 (CPOL is '0', CPHA is '0').

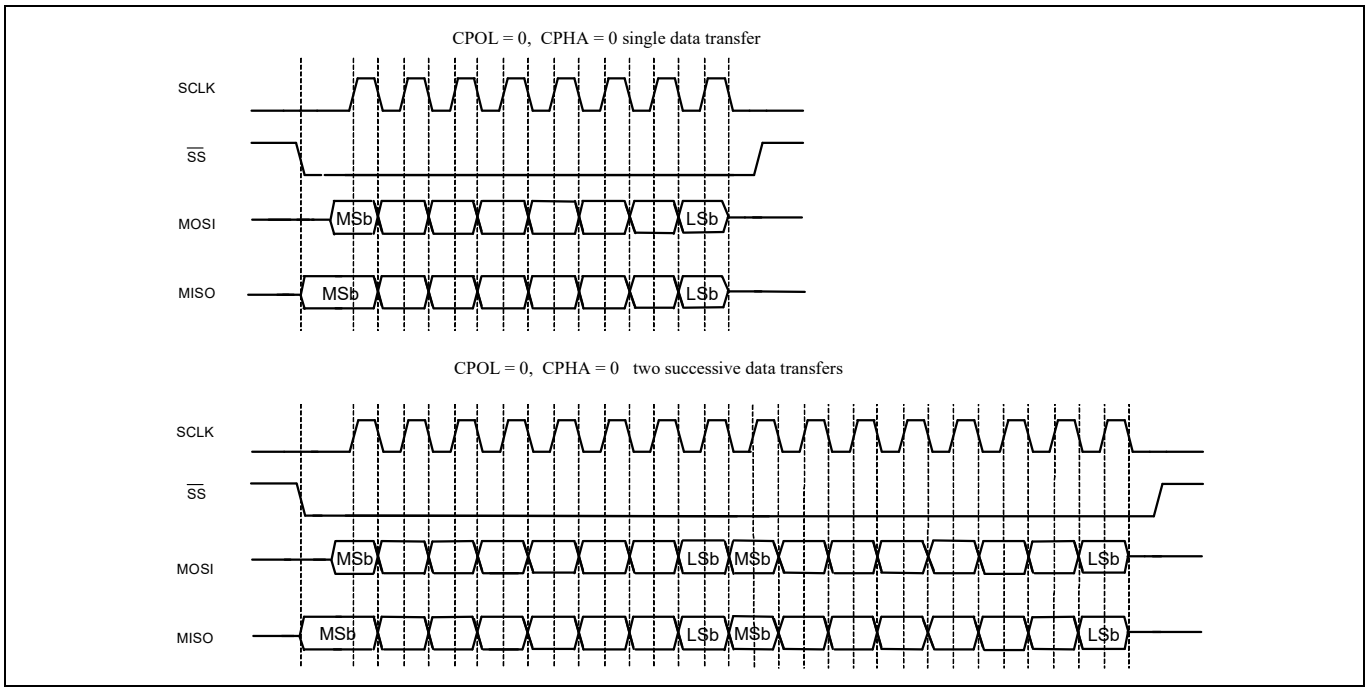


Figure 28-3. SPI Motorola data transfer example

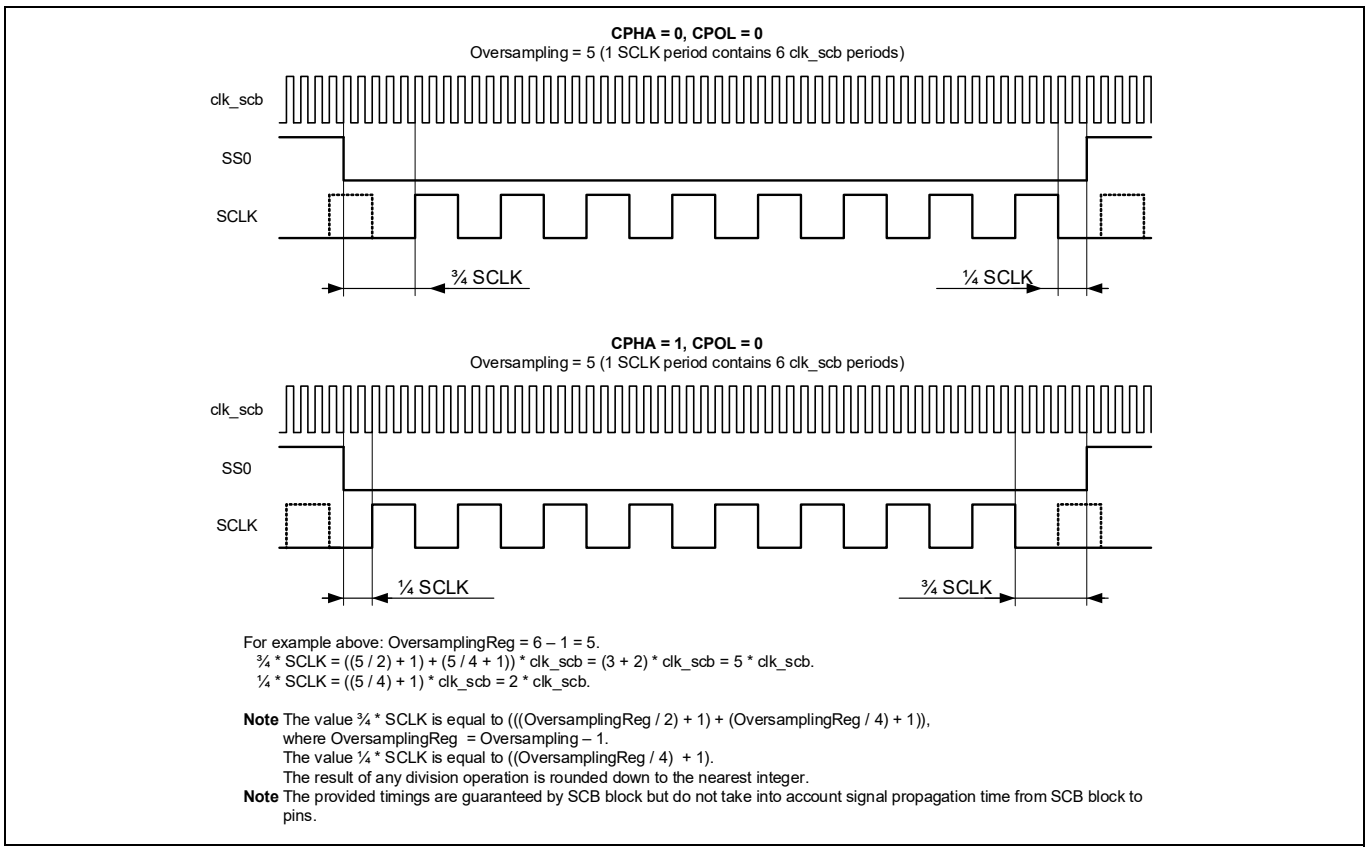


Figure 28-4. SELECT and SCLK timing correlation

Configuring SCB for SPI Motorola mode

To configure the SCB for SPI Motorola mode, set various register bits in the following order:

1. Select SPI by writing ‘01’ to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Select SPI Motorola mode by writing ‘00’ to the MODE (bits [25:24]) of the SCB_SPI_CTRL register.
3. Select the clock mode in Motorola by writing to the CPHA and CPOL fields (bits 2 and 3 respectively) of the SCB_SPI_CTRL register.
4. Follow steps 2 to 4 mentioned in **“Enabling and initializing SPI”** on page 458.

For more information on these registers, see the registers reference manual.

28.3.3.2 Texas Instruments SPI

The Texas Instruments’ SPI protocol redefines the use of the $\overline{SS}$ signal. It uses the signal to indicate the start of a data transfer, rather than a low active slave select signal, as in the case of Motorola SPI. The start of a transfer is indicated by a high active pulse of a single bit transfer period. This pulse may occur one cycle before the transmission of the first data bit, or may coincide with the transmission of the first data bit. The TI SPI protocol supports only mode 1 (CPOL is ‘0’ and CPHA is ‘1’): data is driven on a rising edge of SCLK and data is captured on a falling edge of SCLK.

Figure 28-5 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SELECT pulse precedes the first data bit. Note how the SELECT pulse of the second data transfer coincides with the last data bit of the first data transfer.

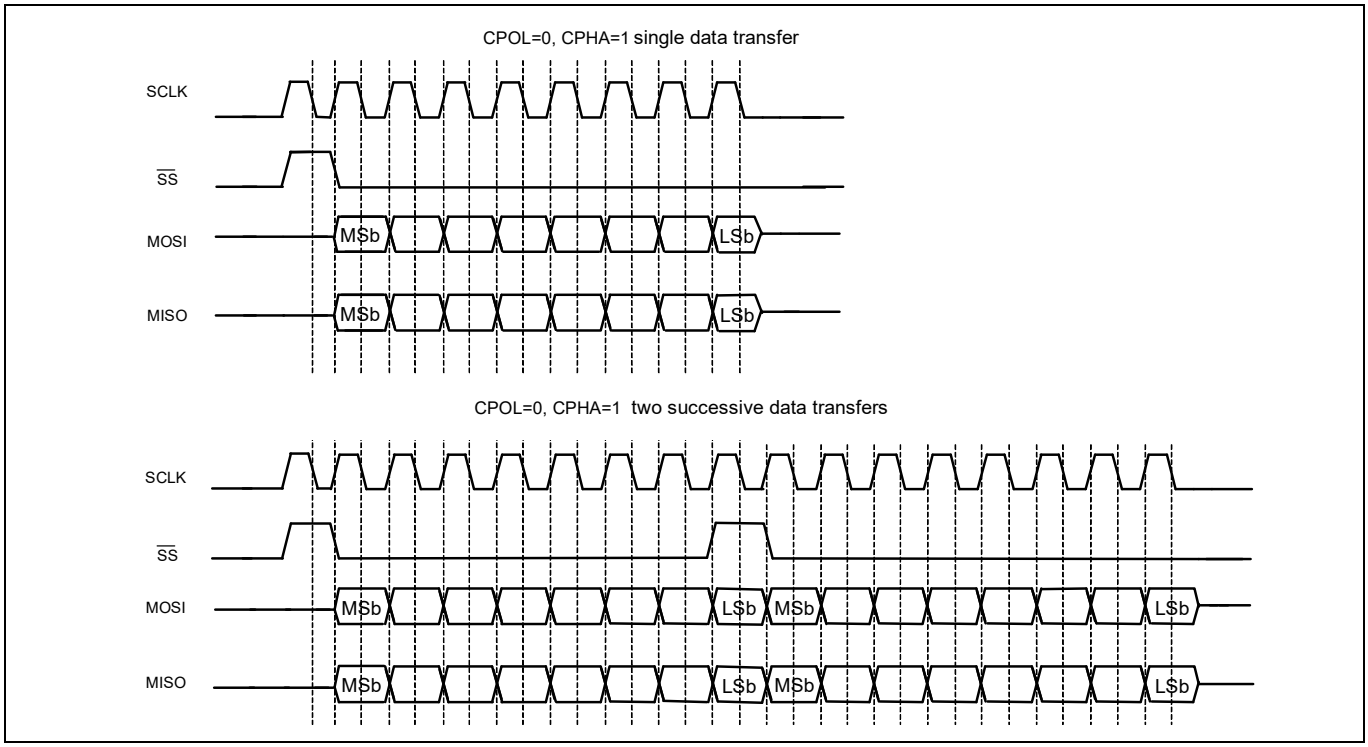


Figure 28-5. SPI TI data transfer example

Figure 28-6 illustrates a single 8-bit data transfer and two successive 8-bit data transfers. The SELECT pulse coincides with the first data bit of a frame.

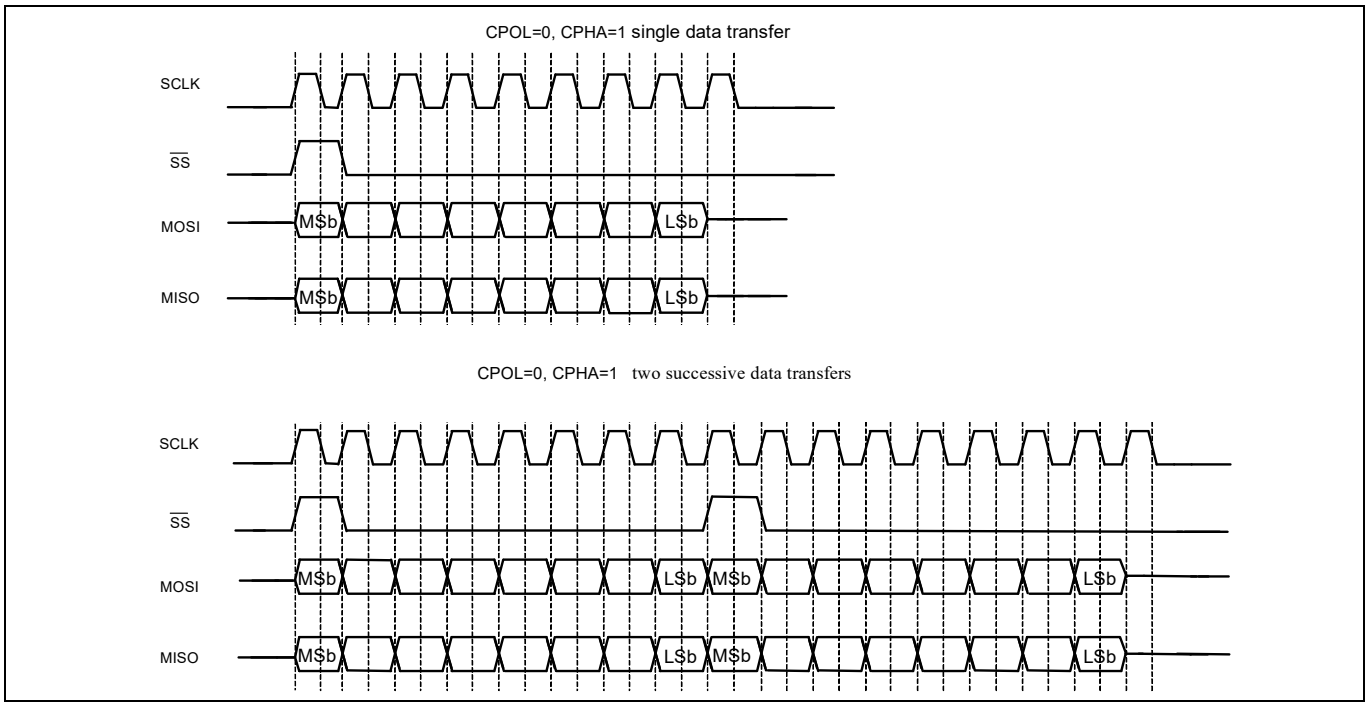


Figure 28-6. SPI TI data transfer example

Configuring SCB for SPI TI Mode

To configure the SCB for SPI TI mode, set various register bits in the following order:

1. Select SPI by writing ‘01’ to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Select SPI TI mode by writing ‘01’ to the MODE (bits [25:24]) of the SCB_SPI_CTRL register.
3. Select the mode of operation in TI by writing to the SELECT_PRECEDE field (bit 1) of the SCB_SPI_CTRL register (‘1’ configures the SELECT pulse to precede the first bit of next frame and ‘0’ otherwise).
4. Set the CPHA bit of the SCB_SPI_CONTROL register to ‘0’, and the CPOL bit of the same register to ‘1’.
5. Follow steps 2 to 4 mentioned in **“Enabling and initializing SPI”** on page 458.

For more information on these registers, see the registers reference manual.

28.3.3.3 National Semiconductors SPI

The National Semiconductors' SPI protocol is a half-duplex protocol. Rather than transmission and reception occurring at the same time, they take turns. The transmission and reception data sizes may differ. A single 'idle' bit transfer period separates transfers from reception. However, successive data transfers are not separated by an idle bit transfer period.

The National Semiconductors SPI protocol supports only mode 0.

Figure 28-7 illustrates a single data transfer and two successive data transfers. In both cases, the transmission data transfer size is eight bits and the reception data transfer size is four bits.

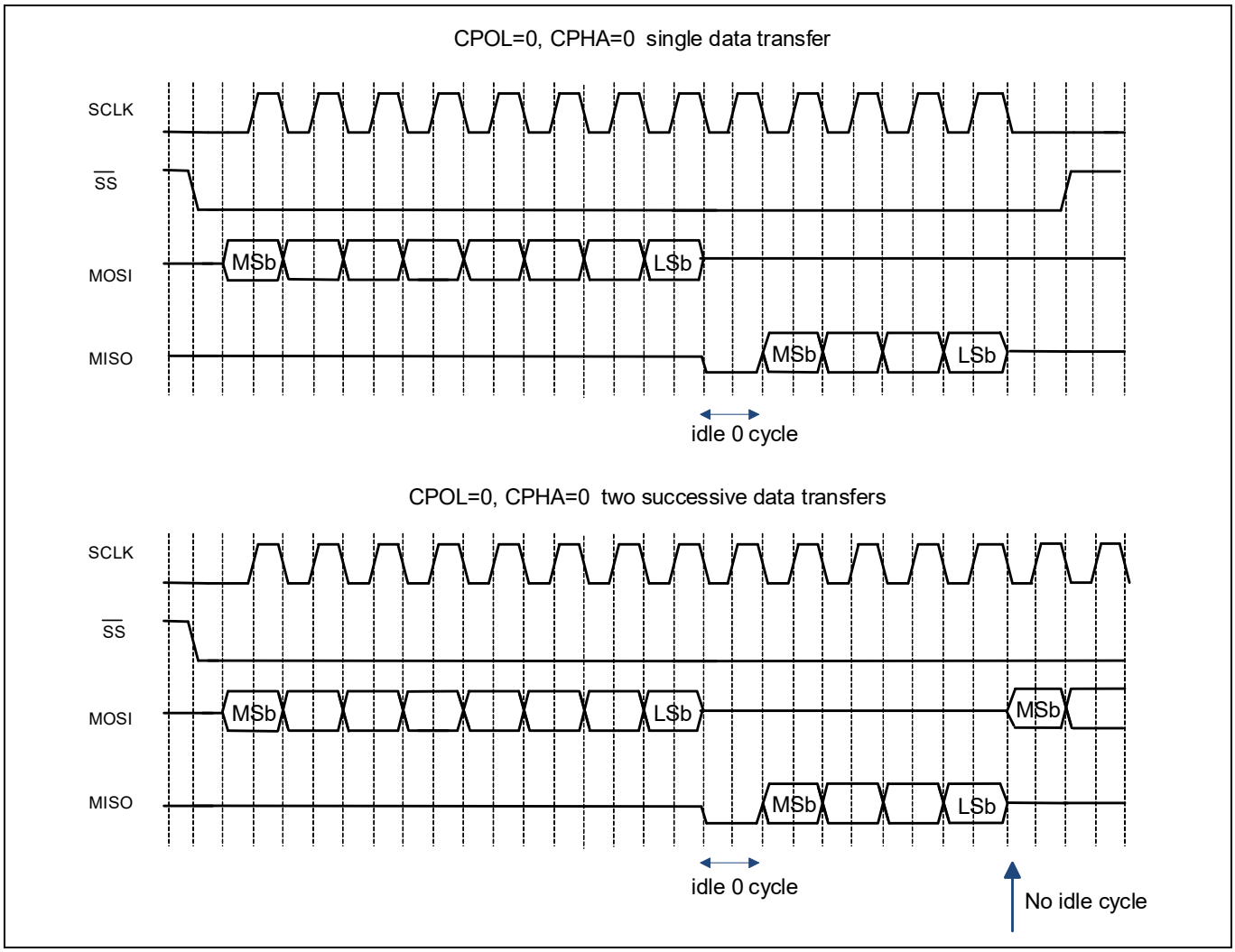


Figure 28-7. SPI NS data transfer example

Configuring SCB for SPI NS Mode

To configure the SCB for SPI NS mode, set various register bits in the following order:

1. Select SPI by writing '01' to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Select SPI NS mode by writing '10' to the MODE (bits [25:24]) of the SCB_SPI_CTRL register.
3. Set the CPOL and CHPA bits of the SCB_SPI_CTRL register to '0'.
4. Follow steps 2 to 4 mentioned in **"Enabling and initializing SPI"** on page 458.

For more information on these registers, see the registers reference manual.

28.3.4 SPI Buffer modes

SPI can operate in three different buffer modes – FIFO, EZ, and CMD_RESP modes. The buffer is used in different ways in each of these modes. The following subsections explain each of these buffer modes in detail.

28.3.4.1 FIFO mode

The FIFO mode has a TX FIFO for the data being transmitted and an RX FIFO for the data received. Each FIFO is constructed out of the SRAM buffer. The FIFOs are either 64 elements deep with 16-bit data elements or 128 elements deep with 8-bit data elements. The width of a FIFO is configured using the BYTE_MODE bitfield of the SCB.CTRL register.

FIFO mode of operation is available only in Active and Sleep power modes, and not in the Deep Sleep mode.

Transmit and receive FIFOs allow write and read accesses. A write access to the transmit FIFO uses the TX_FIFO_WR register. A read access from the receive FIFO uses the RX_FIFO_RD register. For SPI master mode, data transfers are started when data is written into the TX FIFO. Note that when a master is transmitting and the FIFO becomes empty the slave is de-selected.

Transmit and receive FIFO status information is available through status registers, TX_FIFO_STATUS and RX_FIFO_STATUS, and through the INTR_TX and INTR_RX registers.

Each FIFO has a trigger output. This trigger output can be routed through the trigger mux to various other peripheral on the device such as DMA or TCPWMs. The trigger output of the SCB is controlled through the TRIGGER_LEVEL field in the RX_CTRL and TX_CTRL registers.

- For a TX FIFO a trigger is generated when the number of entries in the transmit FIFO is less than TX_FIFO_CTRL.TRIGGER_LEVEL
- For the RX FIFO a trigger is generated when the number of entries in the FIFO is greater than the RX_FIFO_CTRL.TRIGGER_LEVEL

Note that the DMA has a trigger deactivation setting. For the SCB this should be set to 16.

Active to Deep Sleep transition

Before going to Deep Sleep ensure that all active communication is complete. For a master this can easily be done by checking the SPI_DONE bit in the INTR_M register, and ensuring the TX FIFO is empty.

For a slave this can be achieved by checking the BUS_BUSY bit in the SPI Status register. Also the RX FIFO should be empty before going to Deep Sleep. Any data in the FIFO will be lost during Deep Sleep.

Also before going to Deep Sleep the clock to the SCB needs to be disabled. This can be done by setting the SDA_IN_FILT_TRIM[1] bit in the I2C_CFG register to "0"

Lastly, when the device goes to Deep Sleep the SCB stops driving the GPIO lines. This leads to floating pins and can lead to undesirable current during Seep Sleep power modes. To avoid this condition before entering Deep Sleep mode change the HSIOM settings of the SCB pins to GPIO driven, then change the drive mode and drive state to the appropriate state to avoid floating pins. Refer the **“I/O system”** on page 332 for more information on pin drive modes.

Deep Sleep to Active transition

EC_AM = 1, EC_OP = 0, FIFO Mode.

When the SPI Slave Select line is asserted the device will be awoken by an interrupt. After the device is awoken change the SPI pin drive modes and HSIOM settings back to what they were before Deep Sleep. When clk_hf[0] is at the desired frequency set SDA_IN_FILT_TRIM[1] to ‘1’ to enable the clock to the SCB. Then write data into the TX FIFO. At this point the master can read valid data from the slave. Before that any data read by the master will be invalid.

28.3.4.2 EZSPI mode

The Easy SPI (EZSPI) protocol only works in the Motorola mode, with any of the clock modes. It allows communication between master and slave without the need for CPU intervention. In the EZ-USB™ FX10 MCU, only the Deep Sleep-capable SCB supports EZSPI mode.

The EZSPI protocol defines a single memory buffer with an 8-bit EZ address that indexes the buffer (256-entry array of eight bit per entry) located on the slave device. The EZ address is used to address these 256 locations. All EZSPI data transfers have 8-bit data frames.

The CPU writes and reads to the memory buffer through the SCB_EZ_DATA registers. These accesses are word accesses, but only the least significant byte of the word is used.

EZSPI has three types of transfers: a write of the EZ address from the master to the slave, a write of data from the master to an addressed slave memory location, and a read by the master from an addressed slave memory location.

Note: When multiple bytes are read or written the master must keep SSEL low during the entire transfer.

EZ Address Write

A write of the EZ address starts with a command byte (0x00) on the MOSI line indicating the master's intent to write the EZ address. The slave then drives a reply byte on the MISO line to indicate that the command is acknowledged (0xFE) or not (0xFF). The second byte on the MOSI line is the EZ address.

Memory Array Write

A write to a memory array index starts with a command byte (0x01) on the MOSI line indicating the master's intent to write to the memory array. The slave then drives a reply byte on the MISO line to indicate that the command was registered (0xFE) or not (0xFF). Any additional bytes on the MOSI line are written to the memory array at locations indicated by the communicated EZ address. The EZ address is automatically incremented by the slave as bytes are written into the memory array. When the EZ address exceeds the maximum number of memory entries (256), it remains there and does not wrap around to 0. The EZ base address is reset to the address written in the EZ Address Write phase on each slave selection.

Memory Array Read

A read from a memory array index starts with a command byte (0x02) on the MOSI line indicating the master's intent to read from the memory array. The slave then drives a reply byte on the MISO line to indicate that the command was registered (0xFE) or not (0xFF). Any additional read data bytes on the MISO line are read from the memory array at locations indicated by the communicated EZ address. The EZ address is automatically incremented by the slave as bytes are read from the memory array. When the EZ address exceeds the maximum number of memory entries (256), it remains there and does not wrap around to 0. The EZ base address is reset to the address written in the EZ Address Write phase on each slave selection.

Figure 28-8 illustrates the write of EZ address, write to a memory array and read from a memory array operations in the EZSPI protocol.

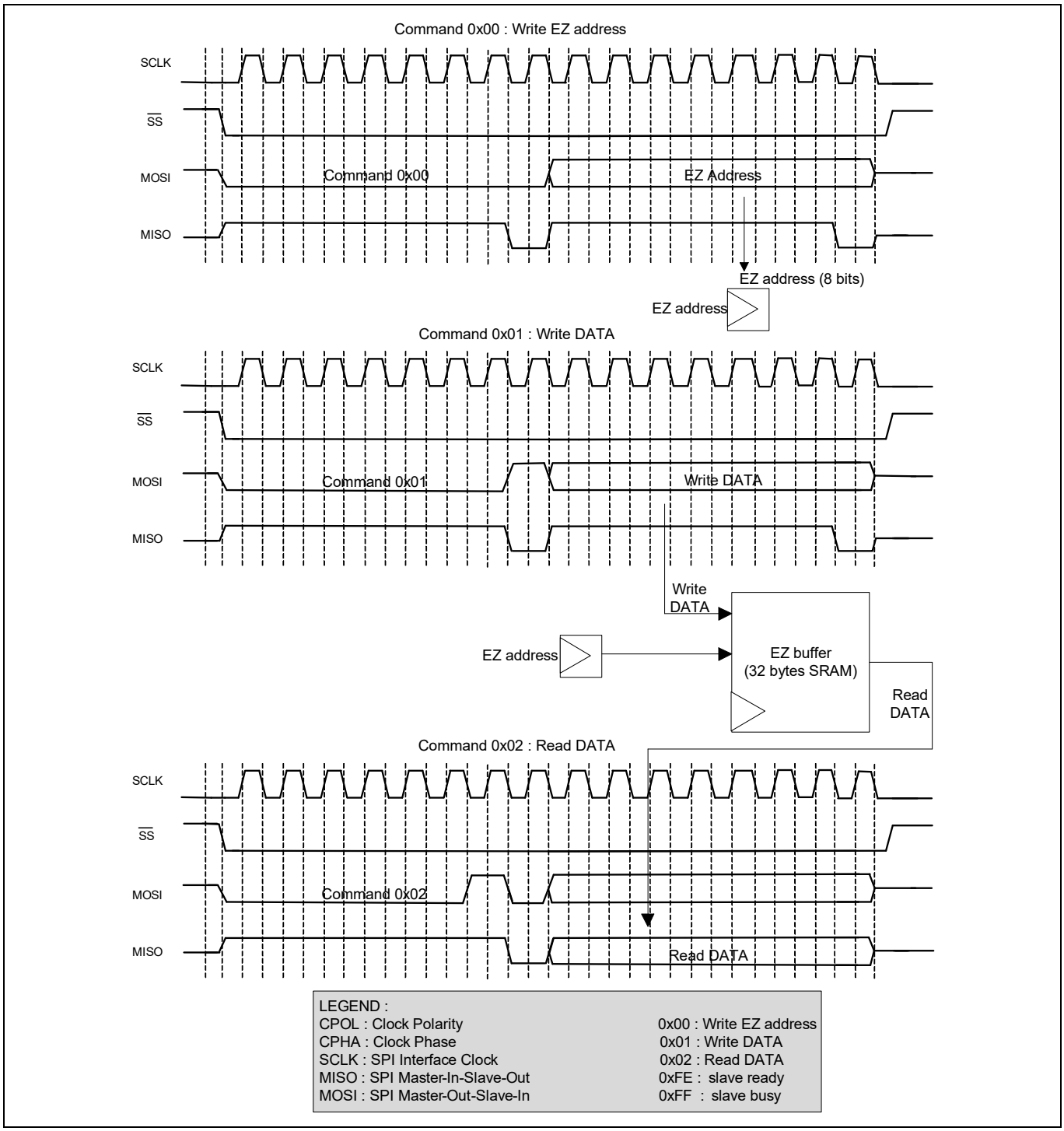


Figure 28-8. EZSPI example

Configuring SCB for EZSPI mode

By default, the SCB is configured for non-EZ mode of operation. To configure the SCB for EZSPI mode, set the register bits in the following order:

1. Select EZ mode by writing '1' to the EZ_MODE bit (bit 10) of the SCB_CTRL register.
2. Set the EC_AM and EC_OP modes in the SCB_CTRL register as appropriate.
3. Set the BYTE_MODE bit of the SCB_CTRL register to '1'.
4. Follow the steps in **"Configuring SCB for SPI Motorola mode"** on page 446.
5. Follow steps 2 to 4 mentioned in **"Enabling and initializing SPI"** on page 458.

For more information on these registers, see the registers reference manual.

Active to Deep Sleep transition

Before going to Deep Sleep ensure the master is not currently transmitting to the slave. This can be done by checking the BUS_BUSY bit in the SPI_STATUS register.

If the bus is not busy, disable the clock to the SCB by setting the SDA_IN_FILT_TRIM[1] bit to '0' in the I2C_CFG register.

Deep Sleep to Active transition

- **EC_AM = 1, EC_OP = 0, EZ Mode:** MISO transmits 0xFF until the internal clock is enabled. Data on MOSI is ignored until the internal clock is enabled. Do not enable the internal clock until clk_hf[0] is at the desired frequency. After clk_hf[0] is at the desired frequency set the SDA_IN_FILT_TRIM[1] bit to '1' to enable the clock. The external master needs to be aware that when it reads 0xFF on MISO the device is not ready yet.
- **EC_AM = 1, EC_OP = 1, EZ Mode:** Do not enable the internal clock until clk_hf[0] is at the desired frequency. After clk_hf[0] is at the desired frequency set the SDA_IN_FILT_TRIM[1] bit to '1' to enable the clock.

28.3.4.3 Command-Response mode

The command-response mode is defined only for an SPI slave. In the EZ-USB™ FX10 MCU, only the Deep Sleep-capable SCB supports this mode. This mode has a single memory buffer, a base read address, a current read address, a base write address, and a current write address that are used to index the memory buffer. The base addresses are provided by the CPU. The current addresses are used by the slave to index the memory buffer for sequential accesses of the memory buffer. The memory buffer holds 256 8-bit data elements. The base and current addresses are in the range [0, 255]. This mode is only supported by the Motorola mode of operation.

The CPU writes and reads to the memory buffer through the SCB_EZ_DATA registers. These accesses are word accesses, but only the least significant byte of the word is used.

The slave interface accesses the memory buffer using the current addresses. At the start of a write transfer (SPI slave selection), the base write address is copied to the current write address. A data element write is to the current write address location. After the write access, the current address is incremented by '1'. At the start of a read transfer, the base read address is copied to the current read address. A data element read is to the current read address location. After the read data element is transmitted, the current read address is incremented by '1'.

If the current addresses equal the last memory buffer address (address equals 255), the current addresses are not incremented. Subsequent write accesses will overwrite any previously written value at the last buffer address. Subsequent read accesses will continue to provide the (same) read value at the last buffer address. The bus master should be aware of the memory buffer capacity in command-response mode.

The base addresses are provided through CMD_RESP_CTRL. The current addresses are provided through CMD_RESP_STATUS. At the end of a transfer (SPI slave de-selection), the difference between a base and current address indicates how many read/write accesses were performed. The block provides interrupt cause fields to identify the end of a transfer. Command-response mode operation is available in Active, Sleep, and Deep Sleep power modes.

The command-response mode has two phases of operation:

- **Write phase** – The write phase begins with a selection byte, which has its last bit set to '0' indicating a write. The master writes 8-bit data elements to the slave's memory buffer following the selection byte. The slave's current write address is set to the slave's base write address. Received data elements are written to the current write address memory location. After each memory write, the current write address is incremented.
- **Read phase** – The read phase begins with a selection byte, which has its last bit set to '1' indicating a read. The master reads 8-bit data elements from the slave's memory buffer. The slave's current read address is set to the slave's base read address. Transmitted data elements are read from the current address memory location. After each read data element is transferred, the current read address is incremented.

During the reception of the first byte, the slave (MISO) transmits either 0x62 (ready) or a value different from 0x62 (busy). When disabled or reset, the slave transmits 0xFF (busy). The byte value can be used by the master to determine whether the slave is ready to accept the SPI request.

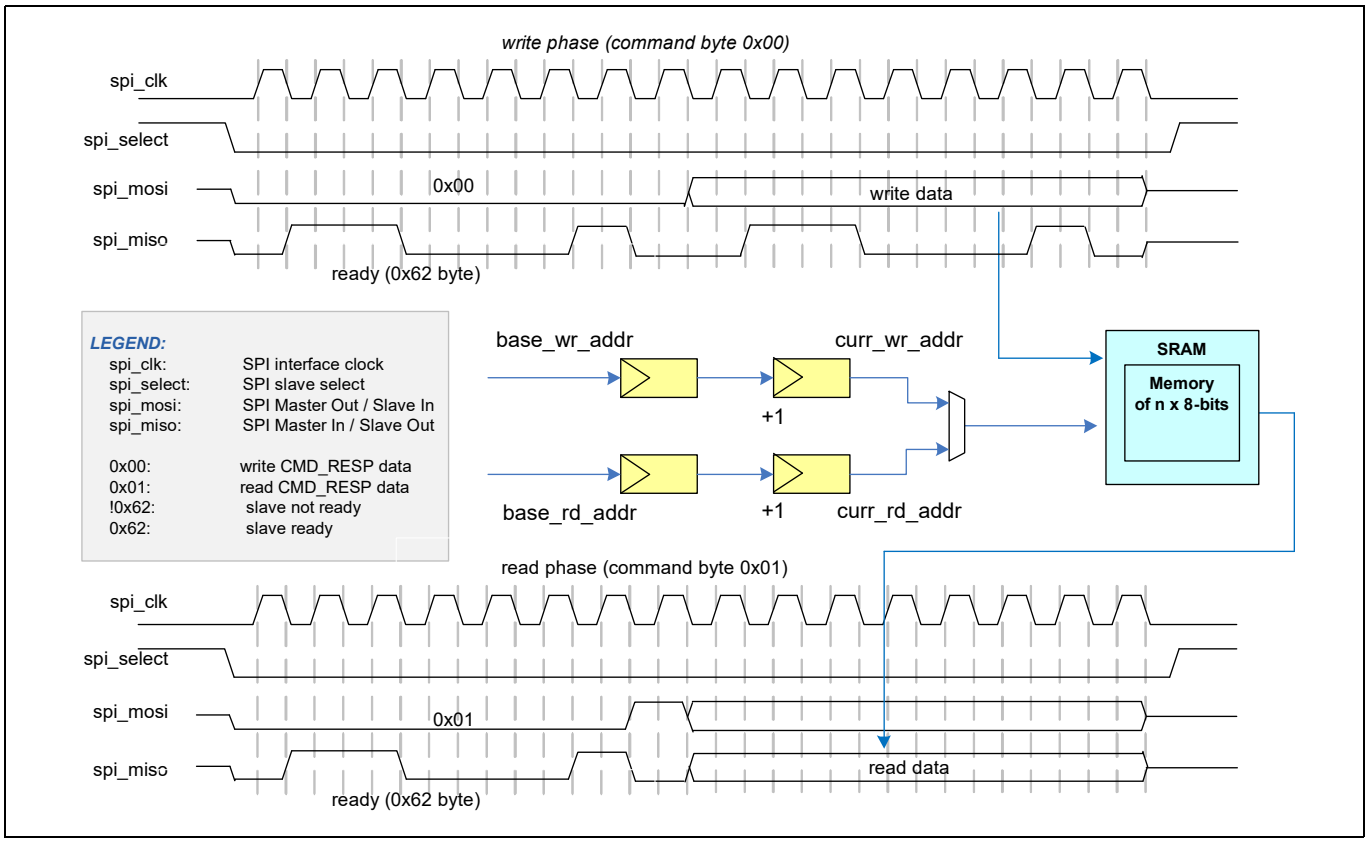


Figure 28-9. Command-Response mode example

Note that a slave's base addresses are updated by the CPU and not by the master.

Active to Deep Sleep transition

Before going to Deep Sleep ensure the master is not currently transmitting to the slave. This can be done by checking the BUS_BUSY bit in the SPI_STATUS register.

If the bus is not busy, disable the clock to the SCB by setting the SDA_IN_FILT_TRIM[1] bit to 0 in the I2C_CFG register.

Deep Sleep to Active transition

EC_AM = 1, EC_OP = 1, CMD_RESP Mode.

Do not enable the internal clock until clk_hf[0] is at the desired frequency. When clk_hf[0] is at the desired frequency, set the SDA_IN_FILT_TRIM[1] bit to '1' to enable the clock.

Configuring SCB for CMD_RESP mode

By default, the SCB is configured for non-CMD_RESP mode of operation. To configure the SCB for CMD_RESP mode, set the register bits in the following order:

1. Select the CMD_RESP mode by writing '1' to the CMD_RESP_MODE bit (bit 12) of the SCB_CTRL register.
2. Set the EC_AM and EC_OP modes to '1' in the SCB_CTRL register.
3. Set the BYTE_MODE bit in the SCB_CTRL register.
4. Follow the steps in **"Configuring SCB for SPI Motorola mode"** on page 446.
5. Follow steps 2 to 4 mentioned in**"Enabling and initializing SPI"** on page 458.

For more information on these registers, see the registers reference manual.

28.3.5 Clocking and oversampling

28.3.5.1 Clock modes

The SCB SPI supports both internally and externally clocked operation modes. Two bitfields (EC_AM_MODE and EC_OP MODE) in the SCB_CTRL register determine the SCB clock mode. EC_AM_MODE indicates whether SPI slave selection is internally (0) or externally (1) clocked. EC_OP_MODE indicates whether the rest of the protocol operation (besides SPI slave selection) is internally (0) or externally (1) clocked.

An externally-clocked operation uses a clock provided by the external master (SPI SCLK).

Note: In the EZ-USB™ FX10 MCU only the Deep Sleep-capable SCB supports externally-clocked mode of operation and only for SPI slave mode.

An internally-clocked operation uses the programmable clock dividers. For SPI, an integer clock divider must be used for both master and slave. For more information on system clocking, see the **"Clocking system"** on page 290.

The SCB_CTRL bitfields EC_AM_MODE and EC_OP_MODE can be configured in the following ways.

- EC_AM_MODE is '0' and EC_OP_MODE is '0': Use this configuration when only Active mode functionality is required.
  - FIFO mode: Supported
  - EZ mode: Supported
  - Command-response mode: Not supported. The slave (MISO) transmits a value different from a ready (0x62) byte during the reception of the first byte if command-response mode is attempted in this configuration.
- EC_AM_MODE is '1' and EC_OP_MODE is '0': Use this configuration when both Active and Deep Sleep functionality are required. This configuration relies on the externally-clocked functionality to detect the slave selection and relies on the internally-clocked functionality to access the memory buffer.

  The "hand over" from external to internal functionality relies on a busy/ready byte scheme. This scheme relies on the master to retry the current transfer when it receives a busy byte and requires the master to support busy/ready byte interpretation. When the slave is selected, INTR_SPI_EC.WAKE_UP is set to '1'. The associated Deep Sleep functionality interrupt brings the system into Active power mode.

- FIFO mode: Supported. The slave (MISO) transmits 0xFF until the CPU is awoken and the TX FIFO is populated. Any data on the MOSI line will be dropped until clk_scb is enabled see **“Deep Sleep to Active transition”** on page 454 for more details.
- EZ mode: Supported. In Deep Sleep power mode, the slave (MISO) transmits a busy (0xFF) byte during the reception of the command byte. In Active power mode, the slave (MISO) transmits a ready (0xFE) byte during the reception of the command byte.
- CMD_RESP mode: Not supported. The slave transmits (MISO) a value different from a ready (0x62) byte during the reception of the first byte.
- EC_AM_MODE is ‘1’ and EC_OP_MODE is ‘1’. Use this mode when both Active and Deep Sleep functionality are required. When the slave is selected, INTR_SPI_EC.WAKE_UP is set to ‘1’. The associated Deep Sleep functionality interrupt brings the system into Active power mode. When the slave is deselected, INTR_SPI_EC.EZ_STOP and/or
  INTR_SPI_EC.EZ_WRITE_STOP are set to ‘1’.
- FIFO mode: Not supported
- EZ mode: Supported
- CMD_RESP mode: Supported

If EC_OP_MODE is ‘1’, the external interface logic accesses the memory buffer on the external interface clock (SPI SCLK). This allows for EZ and CMD_RESP mode functionality in Active and Deep Sleep power modes.

In Active system power mode, the memory buffer requires arbitration between external interface logic (on SPI SCLK) and the CPU interface logic (on system peripheral clock). This arbitration always gives the highest priority to the external interface logic (host accesses). The external interface logic takes two serial interface clock/bit periods for SPI. During this period, the internal logic is denied service to the memory buffer. The EZ-USB™ FX10 MCU provides two programmable options to address this “denial of service”:

- If the BLOCK bitfield of SCB_CTRL is ‘1’: An internal logic access to the memory buffer is blocked until the memory buffer is granted and the external interface logic has completed access. This option provides normal SCB register functionality, but the blocking time introduces additional internal bus wait states.
- If the BLOCK bitfield of SCB_CTRL is ‘0’: An internal logic access to the memory buffer is not blocked, but fails when it conflicts with an external interface logic access. A read access returns the value 0xFFFF:FFFF and a write access is ignored. This option does not introduce additional internal bus wait states, but an access to the memory buffer may not take effect. In this case, the following failures are detected:
  - Read Failure: A read failure is easily detected because the returned value is 0xFFFF:FFFF. This value is unique as non-failing memory buffer read accesses return an unsigned byte value in the range 0x0000:0000-0x0000:00ff.
  - Write Failure: A write failure is detected by reading back the written memory buffer location, and confirming that the read value is the same as the written value

For both options, a conflicting internal logic access to the memory buffer sets INTR_TX.BLOCKED field to ‘1’ (for write accesses) and INTR_RX.BLOCKED field to ‘1’ (for read accesses). These fields can be used as either status fields or as interrupt cause fields (when their associated mask fields are enabled).

If a series of read or write accesses is performed and CTRL.BLOCKED is ‘0’, a failure is detected by comparing the “logical-or” of all read values to 0xFFFF:FFFF and checking the INTR_TX.BLOCKED and INTR_RX.BLOCKED fields to determine whether a failure occurred for a series of write or read operations.

Table 28-3. SPI modes compatibility

	Internally clocked (IC)			Externally clocked (EC) (Deep Sleep SCB only)		
	FIFO	**EZ**	**CMD_RESP**	**FIFO**	**EZ**	**CMD_RESP**
SPI master	Yes	No	No	No	No	No
SPI slave	Yes	Yes	No	Yes[a)]	Yes	Yes

a) In SPI slave FIFO mode, the external-clocked logic does selection detection, then triggers an interrupt to wake up the CPU. Writes will be ignored and reads will return 0xFF until the CPU is ready and the FIFO is populated.

28.3.5.2 Using SPI Master to Clock Slave

In a normal SPI Master mode transmission, the SCLK is generated only when the SCB is enabled and data is being transmitted. This can be changed to always generate a clock on the SCLK line while the SCB is enabled. This is used when the slave uses the SCLK for functional operations other than just the SPI functionality. To enable this, write '1' to the SCLK_CONTINUOUS (bit 5) of the SCB_SPI_CTRL register.

28.3.5.3 Oversampling and bit rate

- **SPI Master mode**

The SPI master does not support externally clocked mode. In internally clocked mode, the logic operates under internal clock. The internal clock has higher frequency than the interface clock (SCLK), such that the master can oversample its input signals (MISO).

The OVS (bits [3:0]) of the SCB_CTRL register specify the oversampling. The oversampling rate is calculated as the value in OVS register + 1. In SPI master mode, the valid range for oversampling is 4 to 16, when MISO is used; if MISO is not used then the valid range is 2 to 16. The bit rate is calculated as follows.

$$Bit\ Rate = clk\_scb/OVS \tag{28.1}$$

Hence, with clk_scb at 100 MHz, the maximum bit rate is 25 Mbps with MISO, or 50 Mbps without MISO.

The numbers above indicate how fast the SCB hardware can run SCLK. It does not indicate that the master will be able to correctly receive data from a slave at those speeds. To determine that, the path delay of MISO must be calculated. It can be calculated using the following equation:

$$\frac{1}{2}t_{SCLK} \geq t_{SCLK\_PCB\_D} + t_{DSO} + t_{SCLK\_PCB\_D} + t_{DSI} \tag{28.2}$$

Where:

t_{SCLK} is the period of the SPI clock

$t_{SCLK\_PCB\_D}$ is the SCLK PCB delay from master to slave

t_{DSO} is the total internal slave delay, time from SCLK edge at slave pin to MISO edge at slave pin

$t_{SCLK\_PCB\_D}$ is the MISO PCB delay from slave to master

t_{DSI} is the master setup time

Most slave datasheets will list tDSO, It may have a different name; look for MISO output valid after SCLK edge. Most master datasheets will also list tDSI, or master setup time. tSCLK_PCB_D and tSCLK_PCB_D must be calculated based on specific PCB geometries.

If after doing these calculations the desired speed cannot be achieved, then consider using the MISO late sample feature of the SCB. This can be done by setting the SPI_CTRL.LATE_MISO_SAMPLE register. MISO late sample tells the SCB to sample the incoming MISO signal on the next edge of SCLK, thus allowing for ½ SCLK cycle more timing margin, see **Figure 28-10**.

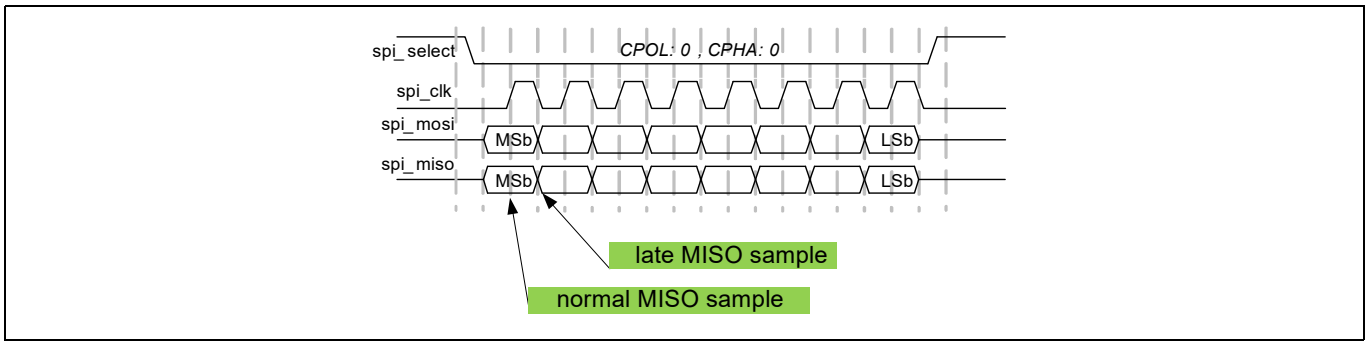


Figure 28-10. MISO sampling timing

This changes the equation to:

$$_{SCLK} \geq t_{SCLK\_PCB\_D} + t_{DSO} + t_{SCLK\_PCB\_D} + t_{DSI} \quad (28.3)$$

Because late sample allows for better timing, leave it enabled all the time. The t_{DSI} specification in the EZ-USB™ FX10 MCU datasheet assumes late sample is enabled.

Note: The SPI_CTRL.LATE_MISO_SAMPLE is set to '0' by default.

- **SPI Slave mode**

In SPI slave mode, the OVS field (bits [3:0]) of SCB_CTRL register is not used. The data rate is determined by **Equation 28.2** and **Equation 28.3**. Late MISO sample is determined by the external master in this case, not by SPI_CTRL.LATE_MISO_SAMPLE.

For EZ-USB™ FX10 MCUs, t_{DSO} is given in the device datasheet. For internally-clocked mode, it is proportional to the frequency of the internal clock. For example it may be 20 ns + 3 *t_{CLK_SCB}. Assuming 0 ns PCB delays, and a 0 ns external master t_{DSI} Equation 26-1 can be re-arranged to $t_{CLK\_SCB} \leq ((t_{SCLK}) - 40\ ns)/6$.

28.3.6 Enabling and initializing SPI

The SPI must be programmed in the following order:

1. Program protocol specific information using the SCB_SPI_CTRL register. This includes selecting the sub-modes of the protocol and selecting master-slave functionality. EZSPI and CMD_RESP can be used with slave mode only.
2. Program the OVS field and configure clk_scb as appropriate. See the **"Clocking system"** on page 290 for more information on how to program clocks and connect it to the SCB.
3. Configure SPI GPIO by setting appropriate drive modes and HSIOM settings.
4. Select the desired Slave Select line and polarity in the SCB_SPI_CTRL register.
5. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers:
   a) Specify the data frame width. This should always be 8 for EZSPI and CMD_RESP.
   b) Specify whether MSb or LSb is the first bit to be transmitted/received. This should always be MSb first for EZSPI and CMD_RESP.
6. Program the transmitter and receiver FIFOs using the SCB_TX_FIFO_CTRL and SCB_RX_FIFO_CTRL registers respectively, as shown in SCB_TX_FIFO_CTRL/SCB_RX_FIFO_CTRL registers. Only for FIFO mode.
   a) Set the trigger level.
   b) Clear the transmitter and receiver FIFO and Shift registers.
7. Enable the block (write a '1' to the ENABLED bit of the SCB_CTRL register). After the block is enabled, control bits should not be changed. Changes should be made after disabling the block; for example, to modify the operation mode (from Motorola mode to TI mode) or to go from externally clocked to internally clocked operation. The change takes effect only after the block is re-enabled.

Note: *Re-enabling the block causes re-initialization and the associated state is lost (for example, FIFO content).*

28.3.7 I/O pad connection

28.3.7.1 SPI master

Figure 28-11 and **Table 28-4** list the use of the I/O pads for SPI Master.

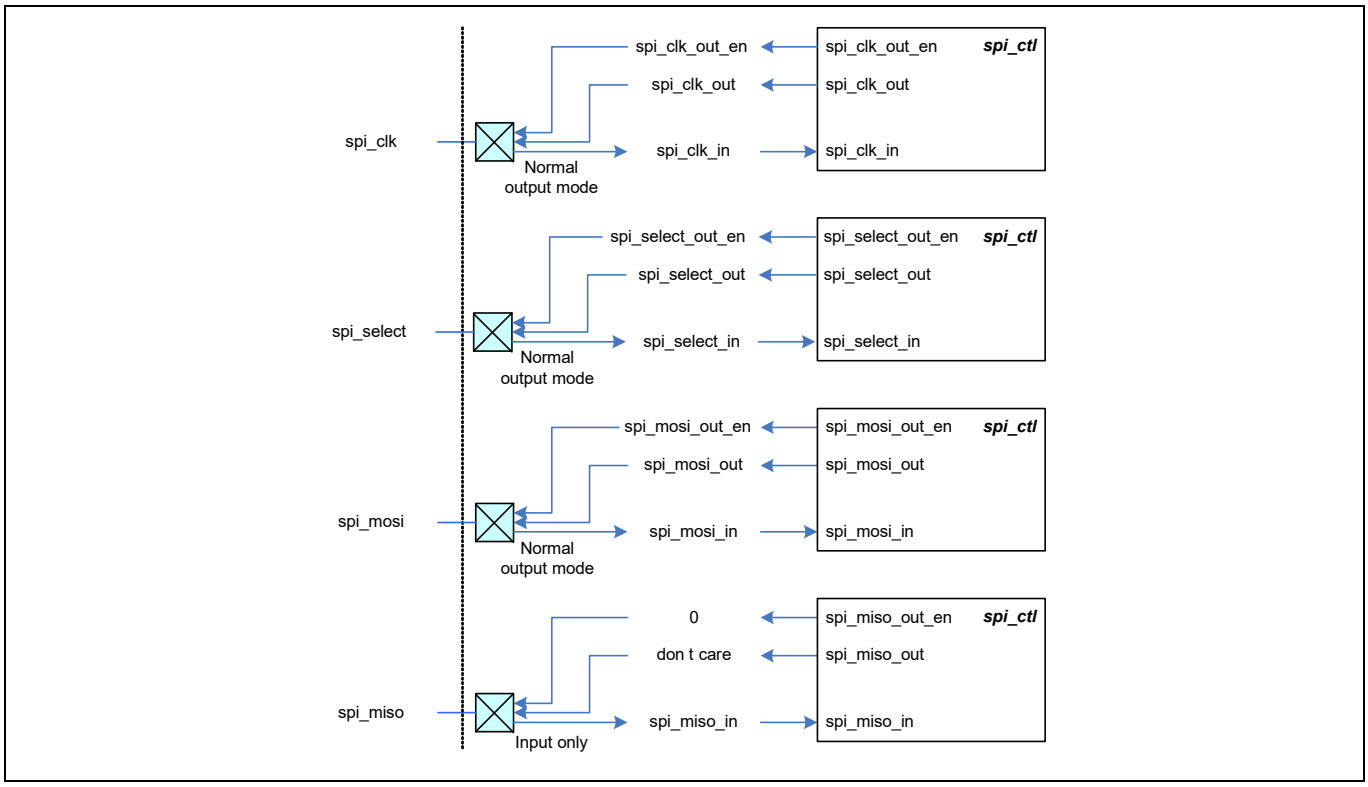


Figure 28-11. SPI Master I/O pad connections

Table 28-4. SPI Master I/O pad connection usage

I/O pads	Drive mode	On-chip I/O signals	Usage
spi_clk	Normal output mode	spi_clk_out_en spi_clk_out	Transmit a clock signal
spi_select	Normal output mode	spi_select_out_en spi_select_out	Transmit a select signal
spi_mosi	Normal output mode	spi_mosi_out_en spi_mosi_out	Transmit a data element
spi_miso	Input only	spi_miso_in	Receive a data element

28.3.7.2 SPI Slave

Figure 28-12 and Table 28-5 list the use of I/O pads for SPI Slave.

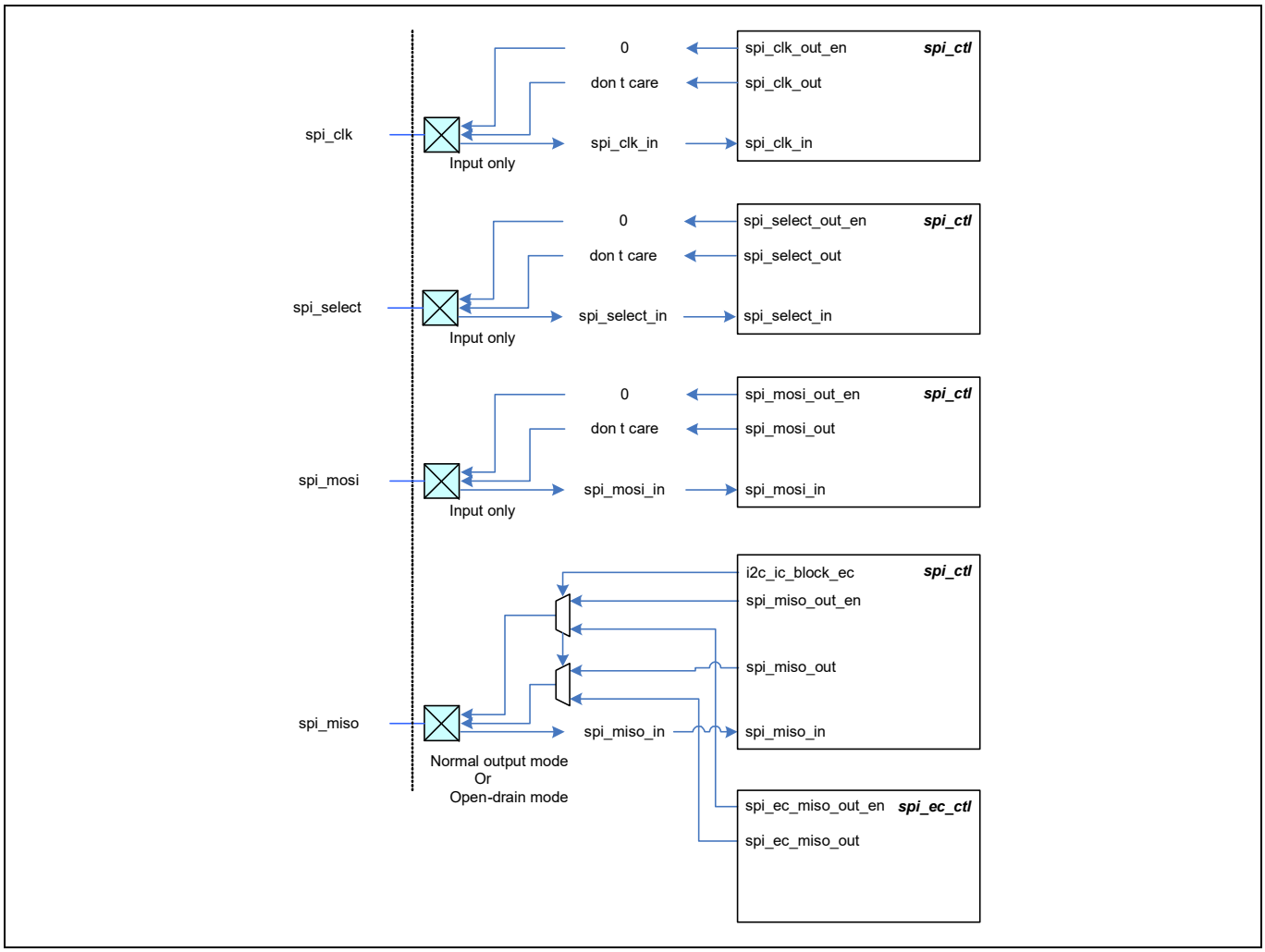


Figure 28-12. SPI Slave I/O pad connections

Open_Drain is set in the TX_CTRL register. In this mode the SPI MISO pin is actively driven low, and then high-z for driving high. This means an external pull-up is required for the line to go high. This mode is useful when there are multiple slaves on the same line. This helps to avoid bus contention issues.

Table 28-5. SPI Slave I/O signal description

I/O pads	Drive mode	On-chip I/O signals	Usage
spi_clk	Input mode	spi_clk_in	Receive a clock signal
spi_select	Input mode	spi_select_in	Receive a select signal
spi_mosi	Input mode	spi_mosi_in	Receive a data element
spi_miso	Normal output mode	spi_miso_out_en spi_miso_out	Transmit a data element

28.3.7.3 Glitch avoidance at System Reset

The SPI outputs are in high-impedance digital state when the device is coming out of system reset. This can cause glitches on the outputs. This is important if you are concerned with SPI master SS0 – SS3 or SCLK output pins activity at either device startup. External pull-up or pull-down resistor can be connected to the output pin to keep it in the inactive state.

28.3.8 SPI registers

The SPI interface is controlled using a set of 32-bit control and status registers listed in **Table 28-6**. Some of these registers are used specifically with the Dee p Sleep SCB; for more information on these registers, see the registers reference manual.

Table 28-6. SPI registers

Register name	Operation
SCB_CTRL	Enables the SCB, selects the type of serial interface (SPI, UART, I^2C), and selects internally and externally clocked operation, and EZ and non-EZ modes of operation.
SCB_STATUS (Deep Sleep SCB only)	In EZ mode, this register indicates whether the externally clocked logic is potentially using the EZ memory.
SCB_SPI_CTRL	Configures the SPI as either a master or a slave, selects SPI protocols (Motorola, TI, National) and clock-based submodes in Motorola SPI (modes 0,1,2,3), selects the type of $\overline{SS}$ signal in TI SPI.
SCB_SPI_STATUS	Indicates whether the SPI bus is busy and sets the SPI slave EZ address in the internally clocked mode.
SCB_TX_CTRL	Specifies the data frame width and specifies whether MSb or LSb is the first bit in transmission.
SCB_RX_CTRL	Performs the same function as that of the SCB_TX_CTRL register, but for the receiver. Also decides whether a median filter is to be used on the input interface lines.
SCB_TX_FIFO_CTRL	Specifies the trigger level, clears the transmitter FIFO and shift registers, and performs the FREEZE operation of the transmitter FIFO.
SCB_RX_FIFO_CTRL	Performs the same function as that of the SCB_TX_FIFO_CTRL register, but for the receiver.
SCB_TX_FIFO_WR	Holds the data frame written into the transmitter FIFO. Behavior is similar to that of a PUSH operation.
SCB_RX_FIFO_RD	Holds the data frame read from the receiver FIFO. Reading a data frame removes the data frame from the FIFO - behavior is similar to that of a POP operation. This register has a side effect when read by software: a data frame is removed from the FIFO.
SCB_RX_FIFO_RD_SILENT	Holds the data frame read from the receiver FIFO. Reading a data frame does not remove the data frame from the FIFO; behavior is similar to that of a PEEK operation.
SCB_TX_FIFO_STATUS	Indicates the number of bytes stored in the transmitter FIFO, the location from which a data frame is read by the hardware (read pointer), the location from which a new data frame is written (write pointer), and decides whether the transmitter FIFO holds the valid data.
SCB_RX_FIFO_STATUS	Performs the same function as that of the SCB_TX_FIFO_STATUS register, but for the receiver.
SCB_EZ_DATA (Deep Sleep SCB only)	Holds the data in EZ memory location

28.4 UART

The Universal Asynchronous Receiver/Transmitter (UART) protocol is an asynchronous serial interface protocol. UART communication is typically point-to-point. The UART interface consists of two signals:

- TX: Transmitter output
- RX: Receiver input

Additionally, two side-band signals are used to implement flow control in UART. Note that the flow control applies only to TX functionality.

- Clear to Send (CTS): This is an input signal to the transmitter. When active, the receiver signals to the transmitter that it is ready to receive.
- Ready to Send (RTS): This is an output signal from the receiver. When active, it indicates that the receiver is ready to receive data.

Not all SCBs support UART mode; see the device datasheet for details.

28.4.1 Features

- Supports UART protocol
  - Standard UART
  - Multi-processor mode
- SmartCard (ISO7816) reader
- IrDA
- Supports Local Interconnect Network (LIN)
  - Break detection
  - Baud rate detection
  - Collision detection (ability to detect that a driven bit value is not reflected on the bus, indicating that another component is driving the same bus)
- Data frame size programmable from 4 to 16 bits
- Programmable number of STOP bits, which can be set in terms of half bit periods between 1 and 4
- Parity support (odd and even parity)
- Median filter on RX input
- Programmable oversampling
- Start skipping
- Hardware flow control

28.4.2 General description

Figure 28-13 illustrates a standard UART TX and RX.

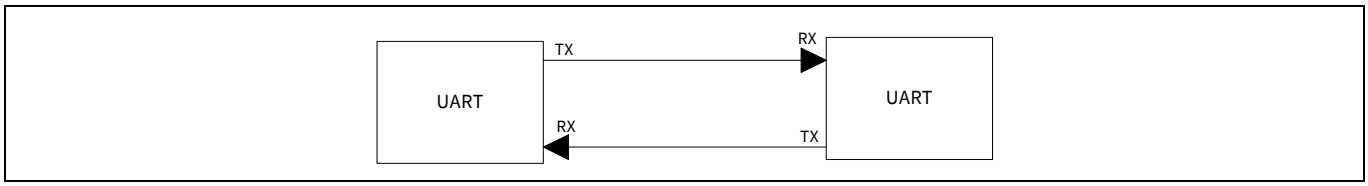


Figure 28-13. UART example

A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit and finally completed by one or more stop bits. The start and stop bits indicate the start and end of data transmission. The parity bit is sent by the transmitter and is used by the receiver to detect single bit errors. Because the interface does not have a clock (asynchronous), the transmitter and receiver use their own clocks; thus, the transmitter and receiver need to agree on the baud rate.

By default, UART supports a data frame width of eight bits. However, this can be configured to any value in the range of 4 to 9. This does not include start, stop, and parity bits. The number of stop bits can be in the range of 1 to 4. The parity bit can be either enabled or disabled. If enabled, the type of parity can be set to either even parity or odd parity. The option of using the parity bit is available only in the Standard UART and SmartCard UART modes. For IrDA UART mode, the parity bit is automatically disabled.

Note: UART interface does not support external clocking operation. Hence, UART operates only in the Active and Sleep system power modes. UART also supports only the FIFO buffer mode.

28.4.3 UART modes of operation

28.4.3.1 Standard protocol

A typical UART transfer consists of a start bit followed by multiple data bits, optionally followed by a parity bit and finally completed by one or more stop bits. The start bit value is always ‘0’, the data bits values are dependent on the data transferred, the parity bit value is set to a value guaranteeing an even or odd parity over the data bits, and the stop bit value is ‘1’. The parity bit is generated by the transmitter and can be used by the receiver to detect single bit transmission errors. When not transmitting data, the TX line is ‘1’ – the same value as the stop bits.

Because the interface does not have a clock, the transmitter and receiver must agree upon the baud rate. The transmitter and receiver have their own internal clocks. The receiver clock runs at a higher frequency than the bit transfer frequency, such that the receiver may oversample the incoming signal.

The transition of a stop bit to a start bit is represented by a change from ‘1’ to ‘0’ on the TX line. This transition can be used by the receiver to synchronize with the transmitter clock. Synchronization at the start of each data transfer allows error-free transmission even in the presence of frequency drift between transmitter and receiver clocks. The required clock accuracy is dependent on the data transfer size.

The stop period or the amount of stop bits between successive data transfers is typically agreed upon between transmitter and receiver, and is typically in the range of 1 to 3-bit transfer periods. **Figure 28-14** illustrates the UART protocol.

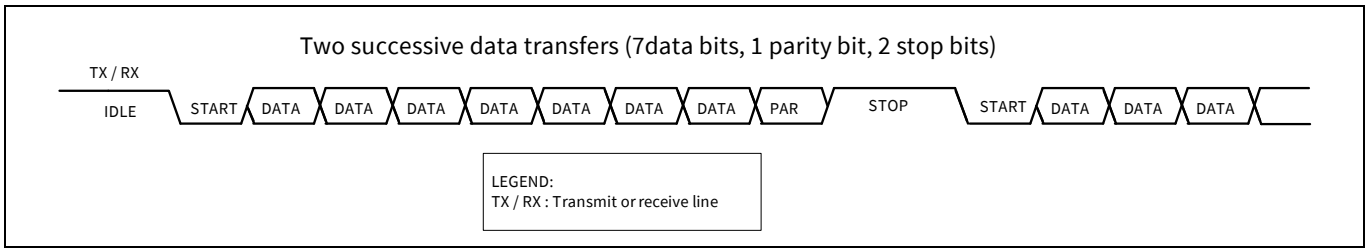


Figure 28-14. UART, standard protocol example

The receiver oversamples the incoming signal; the value of the sample point in the middle of the bit transfer period (on the receiver's clock) is used. **Figure 28-15** illustrates this.

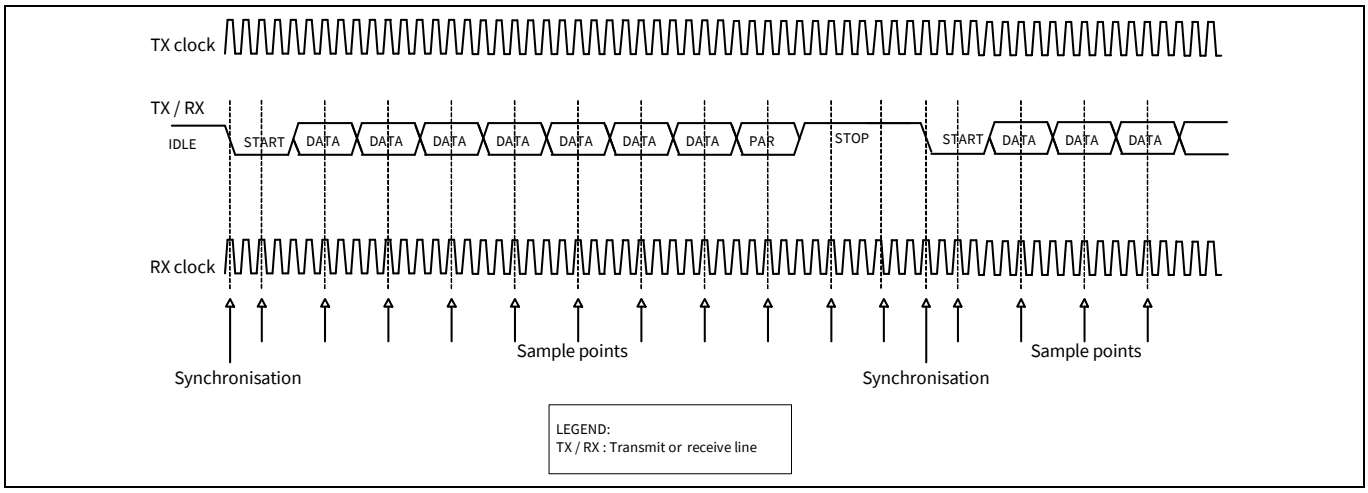


Figure 28-15. UART, standard protocol example (single sample)

Alternatively, three samples around the middle of the bit transfer period (on the receiver's clock) are used for a majority vote to increase accuracy; this is enabled by enabling the MEDIAN filter in the SCB_RX_CTRL register. **Figure 28-16** illustrates this.

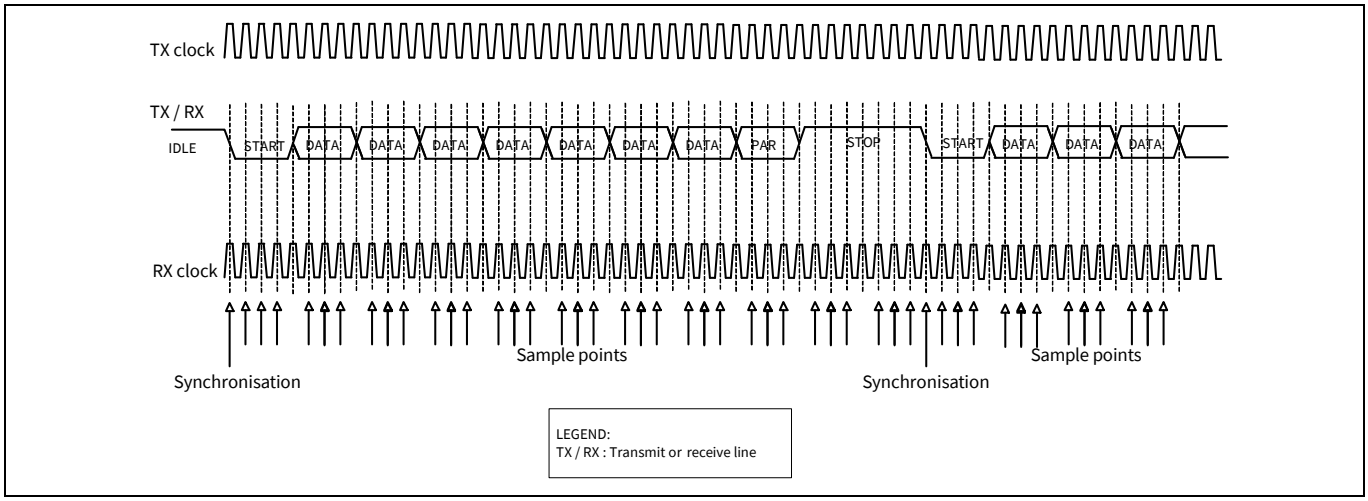


Figure 28-16. UART, standard protocol (multiple samples)

- **Parity**

  This functionality adds a parity bit to the data frame and is used to identify single-bit data frame errors. The parity bit is always directly after the data frame bits.

  The transmitter calculates the parity bit (when UART_TX_CTRL.PARITY_ENABLED is 1) from the data frame bits, such that data frame bits and parity bit have an even (UART_TX_CTRL.PARITY is 0) or odd (UART_TX_CTRL.PARITY is 1) parity. The receiver checks the parity bit (when UART_RX_CTRL.PARITY_ENABLED is 1) from the received data frame bits, such that data frame bits and parity bit have an even (UART_RX_CTRL.PARITY is 0) or odd (UART_RX_CTRL.PARITY is 1) parity.

  Parity applies to both TX and RX functionality and dedicated control fields are available.

  – Transmit functionality: UART_TX_CTRL.PARITY and UART_TX_CTRL.PARITY_ENABLED.
  – Receive functionality: UART_RX_CTRL.PARITY and UART_RX_CTRL.PARITY_ENABLED.

  When a receiver detects a parity error, the data frame is either put in RX FIFO (UART_RX_CTRL.DROP_ON_PARITY_ERROR is 0) or dropped (UART_RX_CTRL.DROP_ON_PARITY_ERROR is 1).

  The following figures illustrate the parity functionality (8-bit data frame).

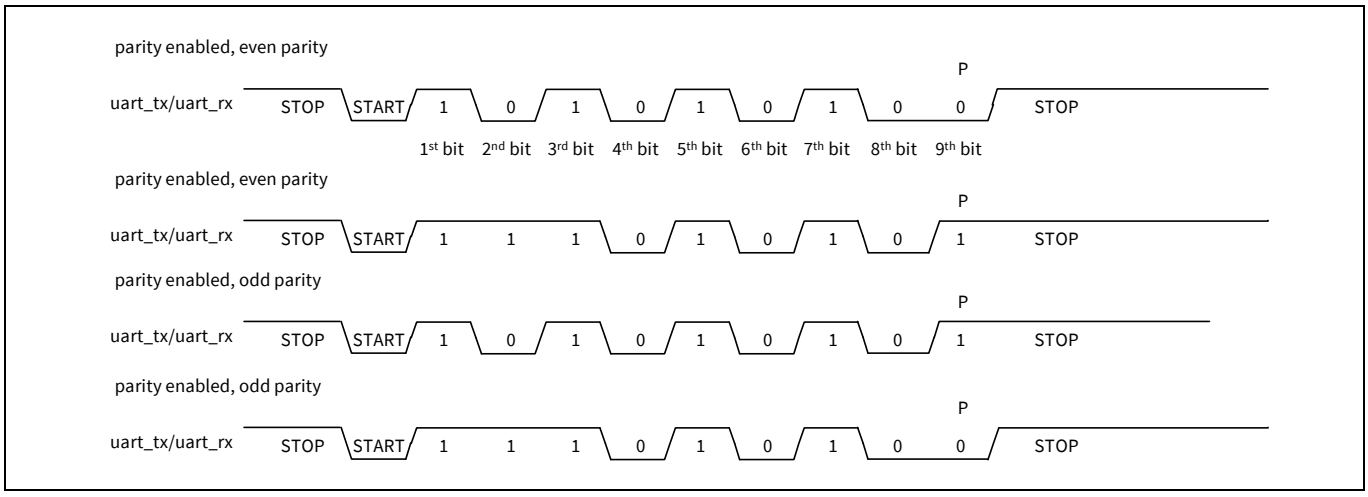


Figure 28-17. UART parity examples

- **Start skipping**

  Start skipping applies only to receive functionality. The standard UART mode supports “start skipping”. Regular receive operation synchronizes on the START bit period (a 1 to 0 transition on the UART RX line), start skipping receive operation synchronizes on the first received data frame bit, which must be a ‘1’ (a 0 to 1 transition on UART RX).

  Start skipping is used to allow for wake up from system Deep Sleep mode using UART. The process is described as follows:

  – Before entering Deep Sleep power mode, UART receive functionality is disabled and the GPIO is programmed to set an interrupt cause to ‘1’ when UART RX line has a ‘1’ to ‘0’ transition (START bit).
  – While in Deep Sleep mode, the UART receive functionality is not functional.
  – The GPIO interrupt is activated on the START bit and the system transitions from Deep Sleep to Active power mode.
  – The CPU enables UART receive functionality, with UART_RX_CTRL.SKIP_START bitfield set to ‘1’.
  – The UART receiver synchronizes data frame receipt on the next ‘0’ to ‘1’ transition. If the UART receive functionality is enabled in time, this is the transition from the START bit to the first received data frame bit.

– The UART receiver proceeds with normal operation; that is, synchronization of successive data frames is on the START bit period.

Figure 28-18 illustrates the process.

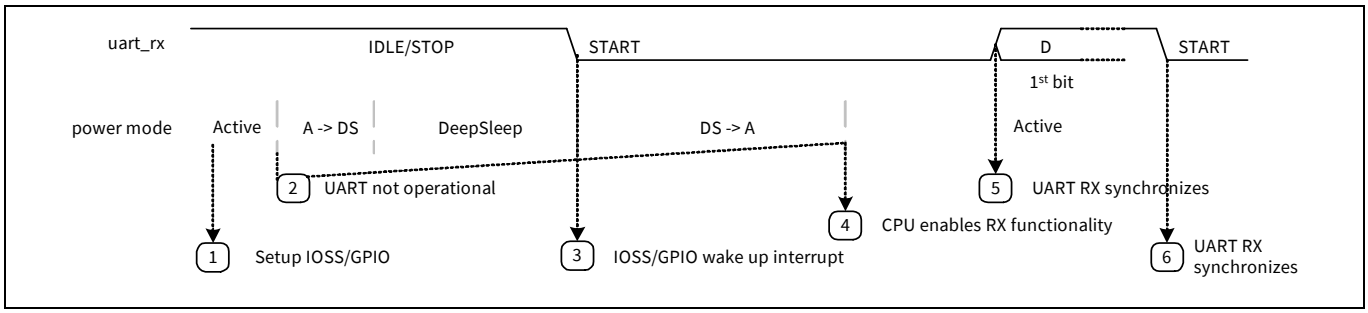


Figure 28-18. UART Start Skip and Wakeup from Deep Sleep

Note that the above process works only for lower baud rates. The Deep Sleep to Active power mode transition and CPU enabling the UART receive functionality should take less than 1-bit period to ensure that the UART receiver is active in time to detect the '0' to '1' transition.

In step 4 of the above process, the firmware takes some time to finish the wakeup interrupt routine and enable the UART receive functionality before the block can detect the input rising edge on the UART RX line.

If the above steps cannot be completed in less than 1 bit time, first send a "dummy" byte to the device to wake it up before sending real UART data. In this case, the SKIP_START bit can be left as 0. For more information on how to perform this in firmware, visit the UART section of the PDL.

- **Break detection**

  Break detection is supported in the standard UART mode. This functionality detects when UART RX line is low (0) for more than UART_RX_CTRL.BREAK_WIDTH bit periods. The break width should be larger than the maximum number of low (0) bit periods in a regular data transfer, plus an additional 1-bit period. The additional 1-bit period is a minimum requirement and preferably should be larger. The additional bit periods account for clock inaccuracies between transmitter and receiver.

  For example, for an 8-bit data frame with parity support, the maximum number of low (0) bit periods is 10 (START bit, 8 '0' data frame bits, and one '0' parity bit). Therefore, the break width should be larger than 10 + 1 = 11 (UART_RX_CTRL.BREAK_WIDTH can be set to 11).

  Note that the break detection applies only to receive functionality. A UART transmitter can generate a break by temporarily increasing TX_CTRL.DATA_WIDTH and transmitting an all zeroes data frame. A break is used by the transmitter to signal a special condition to the receiver. This condition may result in a reset, shut down, or initialization sequence at the receiver.

  Break detection is part of the LIN protocol. When a break is detected, the INTR_RX.BREAK_DETECT interrupt cause is set to '1'. **Figure 28-19** illustrates a regular data frame and break frame (8-bit data frame, parity support, and a break width of 12-bit periods).

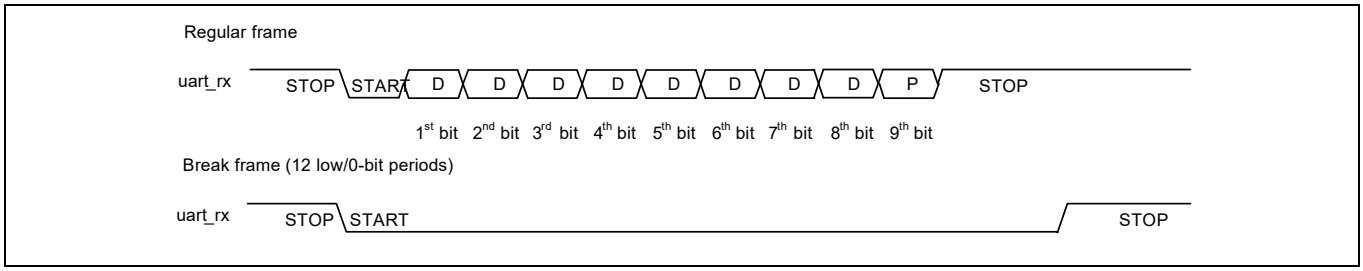


Figure 28-19. UART – Regular frame and break frame

- **Flow control**

  The standard UART mode supports flow control. Modem flow control controls the pace at which the transmitter transfers data to the receiver. Modem flow control is enabled through the UART_FLOW_CTRL.CTS_ENABLED register field. When this field is '0', the transmitter transfers data when its TX FIFO is not empty. When '1', the transmitter transfers data when UART CTS line is active and its TX FIFO is not empty.

  Note that the flow control applies only to TX functionality. Two UART side-band signal are used to implement flow control:

  – UART RTS (uart_rts_out): This is an output signal from the receiver. When active, it indicates that the receiver is ready to receive data (RTS: Ready to Send).
  – UART CTS (uart_cts_in): This is an input signal to the transmitter. When active, it indicates that the transmitter can trans-fer data (CTS: Clear to Send).

  The receiver's uart_rts_out signal is connected to the transmitter's uart_cts_in signal. The receiver's uart_rts_out signal is de-rived by comparing the number of used receive FIFO entries with the UART_FLOW_CTRL.TRIGGER_LEVEL field. If the number of used receive FIFO entries are less than UART_FLOW_CTRL.TRIGGER_LEVEL, uart_rts_out is activated.

  Typically, the UART side-band signals are active low. However, sometimes active high signaling is used. Therefore, the polarity of the side-band signals can be controlled using bitfields UART_FLOW_CTRL.RTS_POLARITY and UART_FLOW_CTRL.CTS_POLARITY.

  Figure 28-20 gives an overview of the flow control functionality.

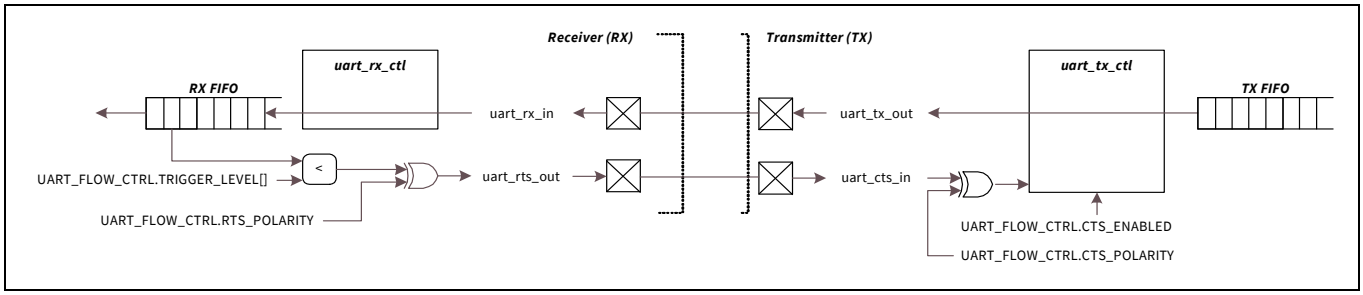


Figure 28-20. UART flow control connection

- **UART Multi-Processor mode**

  The UART_MP (multi-processor) mode is defined with single-master-multi-slave topology, as Figure 28-21 shows. This mode is also known as UART 9-bit protocol because the data field is nine bits wide. UART_MP is part of Standard UART mode.

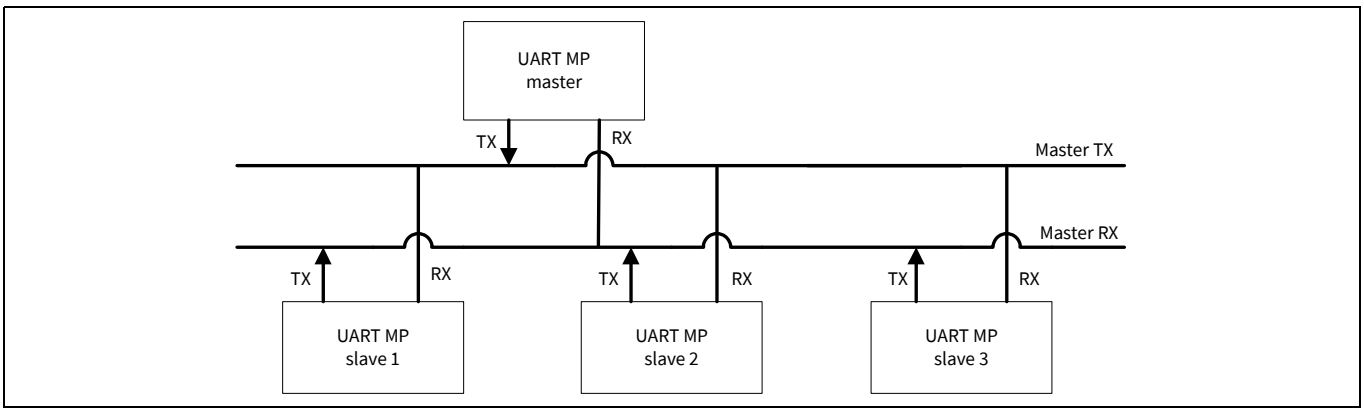


Figure 28-21. UART MP mode bus connections

The main properties of UART_MP mode are:

- Single master with multiple slave concept (multi-drop network)
- Each slave is identified by a unique address
- Using 9-bit data field, with the ninth bit as address/data flag (MP bit). When set high, it indicates an address byte; when set low it indicates a data byte. A data frame is illustrated in **Figure 28-22**
- Parity bit is disabled

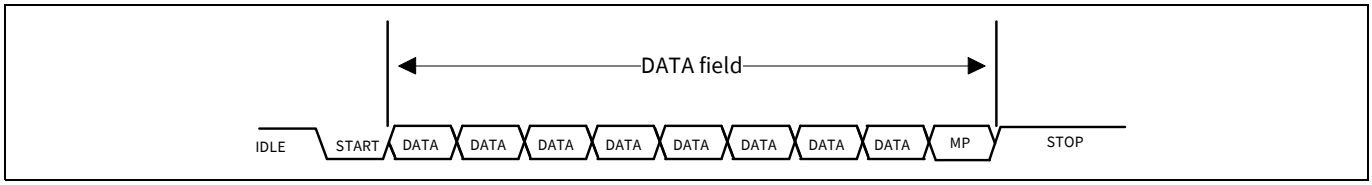


Figure 28-22. UART MP address and data frame

The SCB can be used as either master or slave device in UART_MP mode. Both SCB_TX_CTRL and SCB_RX_CTRL registers should be set to 9-bit data frame size. When the SCB works as UART_MP master device, the firmware changes the MP flag for every address or data frame. When it works as UART_MP slave device, the MP_MODE field of the SCB_UART_RX_CTRL register should be set to '1'. The SCB_RX_MATCH register should be set for the slave address and address mask. The matched address is written in the RX_FIFO when ADDR_ACCEPT field of the SCB_CTRL register is set to '1'. If the received address does not match its own address, then the interface ignores the following data, until next address is received for compare.

- **Configuring the SCB as Standard UART interface**

  To configure the SCB as a standard UART interface, set various register bits in the following order:

  - Configure the SCB as UART interface by writing '10b' to the MODE field (bits [25:24]) of the SCB_CTRL register.
  - Configure the UART interface to operate as a Standard protocol by writing '00' to the MODE field (bits [25:24]) of the SCB_UART_CTRL register.
  - To enable the UART MP Mode or UART LIN Mode, write '1' to the MP_MODE (bit 10) or LIN_MODE (bit 12) respectively of the SCB_UART_RX_CTRL register.
  - Follow steps 2 to 4 described in **"Enabling and initializing the UART"** on page 475.

  For more information on these registers, see the registers reference manual.

28.4.3.2 UART Local Interconnect Network (LIN) mode

The LIN protocol is supported by the SCB as part of the standard UART. LIN is designed with single-master-multi-slave topology. There is one master node and multiple slave nodes on the LIN bus. The SCB UART supports both LIN master and slave functionality. The LIN specification defines both physical layer (layer 1) and data link layer (layer 2). **Figure 28-23** illustrates the UART_LIN and LIN transceiver.

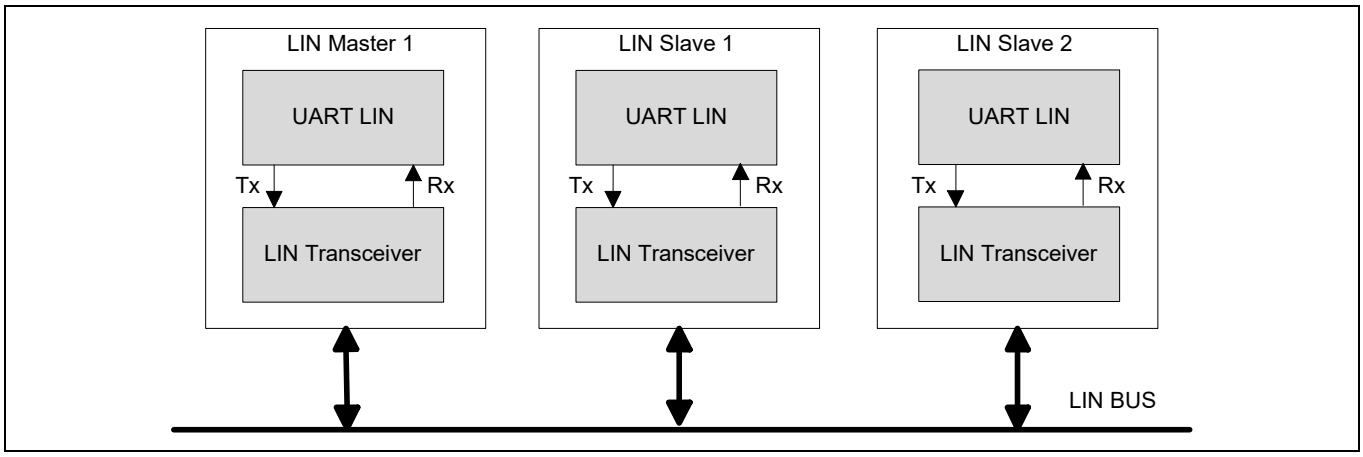


Figure 28-23. UART_LIN and LIN transceiver

LIN protocol defines two tasks:

- Master task: This task involves sending a header packet to initiate a LIN transfer.
- Slave task: This task involves transmitting or receiving a response.

The master node supports master task and slave task; the slave node supports only slave task, as shown in **Figure 28-24**.

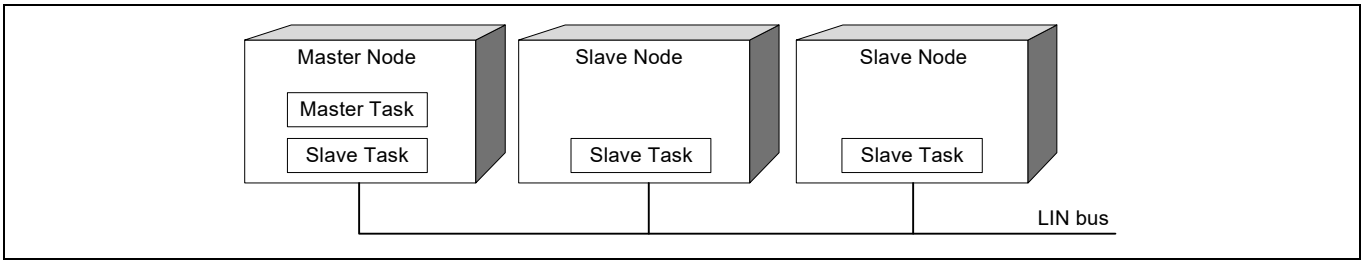


Figure 28-24. LIN bus nodes and tasks

- **LIN frame structure**

  LIN is based on the transmission of frames at pre-determined moments of time. A frame is divided into header and response fields, as shown in **Figure 28-25**.

  – The header field consists of:
    - Break field (at least 13 bit periods with the value ‘0’).
    - Sync field (a 0x55 byte frame). A sync field can be used to synchronize the clock of the slave task with that of the master task.
    - Identifier field (a frame specifying a specific slave).
  – The response field consists of data and checksum.

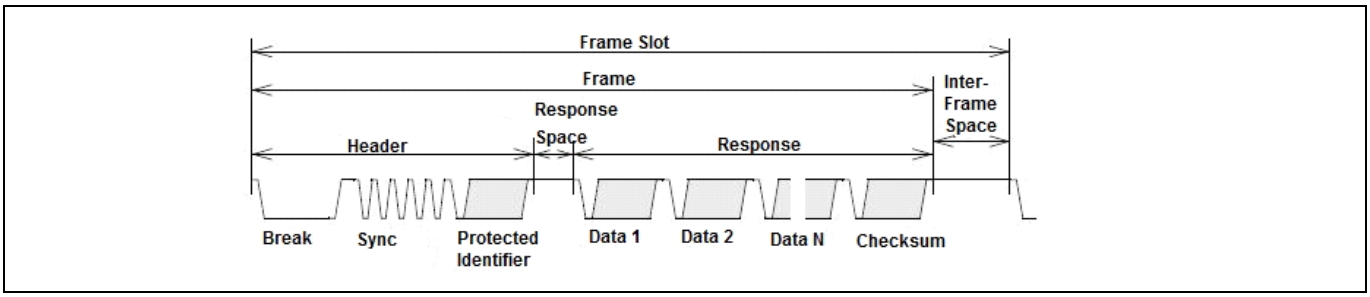


Figure 28-25. LIN frame structure

In LIN protocol communication, the least significant bit (LSb) of the data is sent first and the most significant bit (MSb) last. The start bit is encoded as zero and the stop bit is encoded as one. The following sections describe all the byte fields in the LIN frame.

- **Break field**

  Every new frame starts with a break field, which is always generated by the master. The break field has logical zero with a minimum of 13 bit times and followed by a break delimiter. The break field structure is as shown in **Figure 28-26**.

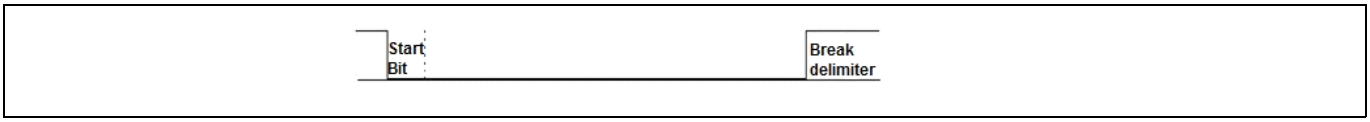


Figure 28-26. LIN Break field

- **Sync field**

  This is the second field transmitted by the master in the header field; its value is 0x55. A sync field can be used to synchronize the clock of the slave task with that of the master task for automatic baud rate detection. **Figure 28-27** shows the LIN sync field structure.

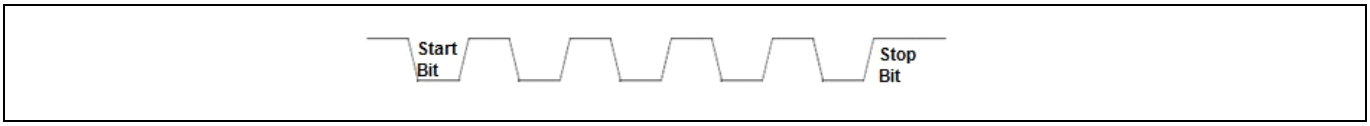


Figure 28-27. LIN Sync field

- **Protected Identifier (PID) field**

  A protected identifier field consists of two sub-fields: the frame identifier (bits 0-5) and the parity (bit 6 and bit 7). The PID field structure is shown in **Figure 28-28**.

  – Frame identifier: The frame identifiers are divided into three categories
    - Values 0 to 59 (0x3B) are used for signal carrying frames
    - 60 (0x3C) and 61 (0x3D) are used to carry diagnostic and configuration data
    - 62 (0x3E) and 63 (0x3F) are reserved for future protocol enhancements
  – Parity: Frame identifier bits are used to calculate the parity

  Figure 28-28 shows the PID field structure.

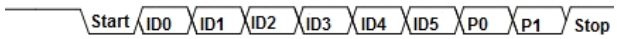


Figure 28-28. PID field

- **Data**

  In LIN, every frame can carry a minimum of one byte and maximum of eight bytes of data. Here, the LSb of the data byte is sent first and the MSb of the data byte is sent last.

- **Checksum**

  The checksum is the last byte field in the LIN frame. It is calculated by inverting the 8-bit sum along with carryover of all data bytes only or the 8-bit sum with the carryover of all data bytes and the PID field. There are two types of checksums in LIN frames. They are:

  – Classic checksum: the checksum calculated over all the data bytes only (used in LIN 1.x slaves)
  – Enhanced checksum: the checksum calculated over all the data bytes along with the protected identifier (used in LIN 2.x slaves)

- **LIN frame types**

  The type of frame refers to the conditions that need to be valid to transmit the frame. According to the LIN specification, there are five different types of LIN frames. A node or cluster does not have to support all frame types.

- **Unconditional frame**

  These frames carry the signals and their frame identifiers (of 0x00 to 0x3B range). The subscriber will receive the frames and make it available to the application; the publisher of the frame will provide the response to the header.

- **Event-triggered frame**

  The purpose of an event-triggered frame is to increase the responsiveness of the LIN cluster without assigning too much of the bus bandwidth to polling of multiple slave nodes with seldom occurring events. Event-triggered frames carry the response of one or more unconditional frames. The unconditional frames associated with an event triggered frame should:

  – Have equal length
  – Use the same checksum model (either classic or enhanced)
  – Reserve the first data field to its protected identifier
  – Be published by different slave nodes
  – Not be included directly in the same schedule table as the event-triggered frame

- **Sporadic frame**

  The purpose of the sporadic frames is to merge some dynamic behavior into the schedule table without affecting the rest of the schedule table. These frames have a group of unconditional frames that share the frame slot. When the sporadic frame is due for transmission, the unconditional frames are checked whether they have any updated signals. If no signals are updated, no frame will be transmitted and the frame slot will be empty.

- **Diagnostic frames**

  Diagnostic frames always carry transport layer, and contains eight data bytes.

  The frame identifier for diagnostic frame is:

  – Master request frame (0x3C), or
  – Slave response frame (0x3D)

  Before transmitting a master request frame, the master task queries its diagnostic module to see whether it will be transmitted or whether the bus will be silent. A slave response frame header will be sent unconditionally. The slave tasks publish and subscribe to the response according to their diagnostic modules.

- **Reserved frames**

  These frames are reserved for future use; their frame identifiers are 0x3E and 0x3F.

- **LIN Go-To-Sleep and Wake-Up**

  The LIN protocol has the feature of keeping the LIN bus in Sleep mode, if the master sends the go-to-sleep command. The go-to-sleep command is a master request frame (ID = 0x3C) with the first byte field is equal to 0x00 and rest set to 0xFF. The slave node application may still be active after the go-to-sleep command is received. This behavior is application specific. The LIN slave nodes automatically enter Sleep mode if the LIN bus inactivity is more than four seconds.

  Wake-up can be initiated by any node connected to the LIN bus – either LIN master or any of the LIN slaves by forcing the bus to be dominant for 250 µs to 5 ms. Each slave should detect the wakeup request and be ready to process headers within 100 ms. The master should also detect the wakeup request and start sending headers when the slave nodes are active.

  To support LIN, a dedicated (off-chip) line driver/receiver is required. Supply voltage range on the LIN bus is 7 V to 18 V. Typically, LIN line drivers will drive the LIN line with the value provided on the SCB TX line and present the value on the LIN line to the SCB RX line. By comparing TX and RX lines in the SCB, bus collisions can be detected (indicated by the SCB_UART_ARB_LOST field of the SCB_INTR_TX register).

28.4.3.3 SmartCard (ISO7816)

ISO7816 is asynchronous serial interface, defined with single-master-single slave topology. ISO7816 defines both Reader (master) and Card (slave) functionality. For more information, refer to the **ISO7816 specification**. Only the master (reader) function is supported by the SCB. This block provides the basic physical layer support with asynchronous character transmission. The UART_TX line is connected to SmartCard I/O line by internally multiplexing between UART_TX and UART_RX control modules.

The SmartCard transfer is similar to a UART transfer, with the addition of a negative acknowledgement (NACK) that may be sent from the receiver to the transmitter. A NACK is always '0'. Both master and slave may drive the same line, although never at the same time.

A SmartCard transfer has the transmitter drive the start bit and data bits (and optionally a parity bit). After these bits, it enters its stop period by releasing the bus. Releasing results in the line being '1' (the value of a stop bit). After one bit transfer period into the stop period, the receiver may drive a NACK on the line (a value of '0') for one bit transfer period. This NACK is observed by the transmitter, which reacts by extending its stop period by one bit transfer period. For this protocol to work, the stop period should be longer than one bit transfer period. Note that a data transfer with a NACK takes one bit transfer period longer, than a data transfer without a NACK. Typically, implementations use a tristate driver with a pull-up resistor, such that when the line is not transmitting data or transmitting the Stop bit, its value is '1'. **Figure 28-29** illustrates the SmartCard protocol.

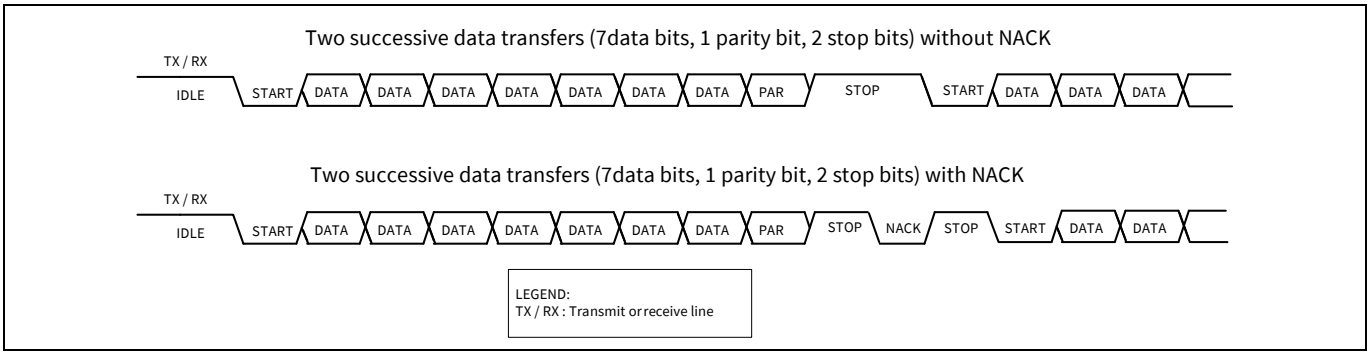


Figure 28-29. SmartCard example

The communication Baud rate while using SmartCard is given as:

Baud rate = Fscbclk/Oversample

28.4.3.3.1 Configuring SCB as UART SmartCard interface

To configure the SCB as a UART SmartCard interface, set various register bits in the following order; note that ModusToolbox does all this automatically with the help of GUIs. For more information on these registers, see the registers reference manual.

1. Configure the SCB as UART interface by writing '10b' to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Configure the UART interface to operate as a SmartCard protocol by writing '01' to the MODE (bits [25:24]) of the SCB_UART_CTRL register.
3. Follow steps 2 to 4 described in **"Enabling and initializing the UART"** on page 475.

28.4.3.4 Infrared Data Association (IrDA)

The SCB supports the IrDA protocol for data rates of up to 115.2 kbps using the UART interface. It supports only the basic physical layer of IrDA protocol with rates less than 115.2 kbps. Hence, the system instantiating this block must consider how to implement a complete IrDA communication system with other available system resources.

The IrDA protocol adds a modulation scheme to the UART signaling. At the transmitter, bits are modulated. At the receiver, bits are demodulated. The modulation scheme uses a Return-to-Zero-Inverted (RZI) format. A bit value of '0' is signaled by a short '1' pulse on the line and a bit value of '1' is signaled by holding the line to '0'. For these data rates (<=115.2 kbps), the RZI modulation scheme is used and the pulse duration is 3/16 of the bit period. The sampling clock frequency should be set 16 times the selected baud rate, by configuring the SCB_OVS field of the SCB_CTRL register. In addition, the EZ-USB™ FX10 MCU SCB supports a low-power IrDA receiver mode, which allows it to detect pulses with a minimum width of 1.41 µs.

Different communication speeds under 115.2 kbps can be achieved by configuring clk_scb frequency. Additional allowable rates are 2.4 kbps, 9.6 kbps, 19.2 kbps, 38.4 kbps, and 57.6 kbps. **Figure 28-30** shows how a UART transfer is IrDA modulated.

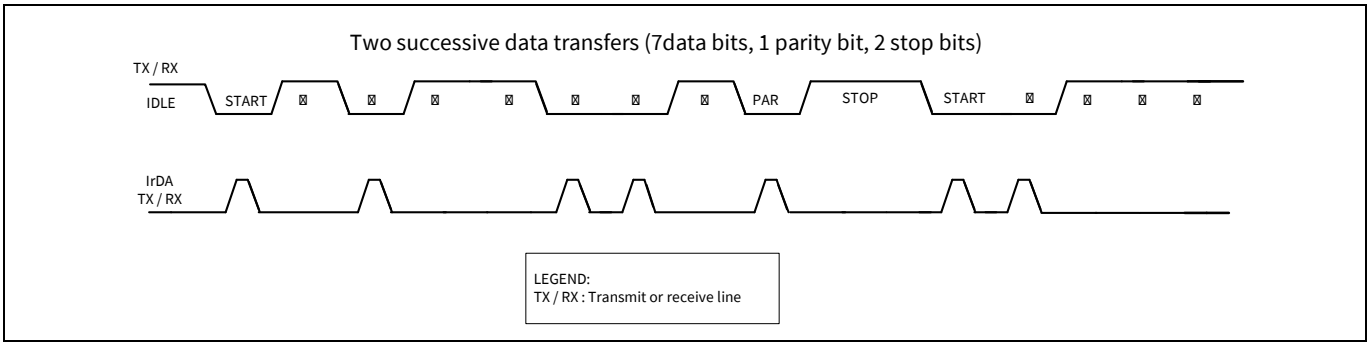


Figure 28-30. IrDA example

28.4.3.4.1 Configuring the SCB as a UART IrDA interface

To configure the SCB as a UART IrDA interface, set various register bits in the following order; note that ModusToolbox does all this automatically with the help of GUIs. For more information on these registers, see the registers reference manual.

1. Configure the SCB as a UART interface by writing '10b' to the MODE (bits [25:24]) of the SCB_CTRL register.
2. Configure the UART interface to operate as IrDA protocol by writing '10' to the MODE (bits [25:24]) of the SCB_UART_CTRL register.
3. Enable the Median filter on the input interface line by writing '1' to MEDIAN (bit 9) of the SCB_RX_CTRL register.
4. Configure the SCB as described in **"Enabling and initializing the UART"** on page 475.

28.4.4 Clocking and oversampling

The UART protocol is implemented using clk_scb as an oversampled multiple of the baud rate. For example, to implement a 100-kHz UART, clk_scb could be set to 1 MHz and the oversample factor set to '10'. The oversampling is set using the SCB_CTRL.OVS register field. The oversampling value is SCB_CTRL.OVS + 1. In the UART standard sub-mode (including LIN) and the SmartCard sub-mode, the valid range for the OVS field is [7, 15].

In UART transmit IrDA sub-mode, this field indirectly specifies the oversampling. Oversampling determines the interface clock per bit cycle and the width of the pulse. This sub-mode has only one valid OVS value–16 (which is a value of 0 in the OVS field of the SCB_CTRL register); the pulse width is roughly 3/16 of the bit period (for all bit rates).

In UART receive IrDA sub-mode (1.2, 2.4, 9.6, 19.2, 38.4, 57.6, and 115.2 kbps), this field indirectly specifies the oversampling. In normal transmission mode, this pulse is approximately 3/16 of the bit period (for all bit rates). In low-power transmission mode, this pulse is potentially smaller (down to 1.62 µs typical and 1.41 µs minimal) than 3/16 of the bit period (for less than 115.2 kbps bit rates).

Pulse widths greater than or equal to two SCB input clock cycles are guaranteed to be detected by the receiver. Pulse widths less than two clock cycles and greater than or equal to one SCB input clock cycle may be detected by the receiver. Pulse widths less than one SCB input clock cycle will not be detected by the receiver. Note that the SCB_RX_CTRL.MEDIAN should be set to '1' for IrDA receiver functionality.

The SCB input clock and the oversampling together determine the IrDA bit rate. See the registers reference manual for more details on the OVS values for different baud rates.

28.4.5 Enabling and initializing the 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. This includes selecting the submodes of the protocol, transmitter-receiver functionality, and so on.
2. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers
   a) Specify the data frame width
   b) Specify whether MSb or LSb is the first bit to be transmitted or received
3. Program the transmitter and receiver FIFOs using the SCB_TX_FIFO_CTRL and SCB_RX_FIFO_CTRL registers, respectively.
   a) Set the trigger level
   b) Clear the transmitter and receiver FIFO and Shift registers
4. Enable the block (write a '1' to the ENABLE bit of the SCB_CTRL register). After the block is enabled, control bits should not be changed. Changes should be made after disabling the block; for example, to modify the operation mode (from SmartCard to IrDA). The change takes effect only after the block is re-enabled. Note that re-enabling the block causes re-initialization and the associated state is lost (for example FIFO content).

28.4.6 I/O pad connection

28.4.6.1 Standard UART mode

Figure 28-31 and **Table 28-7** list the use of the I/O pads for the Standard UART mode.

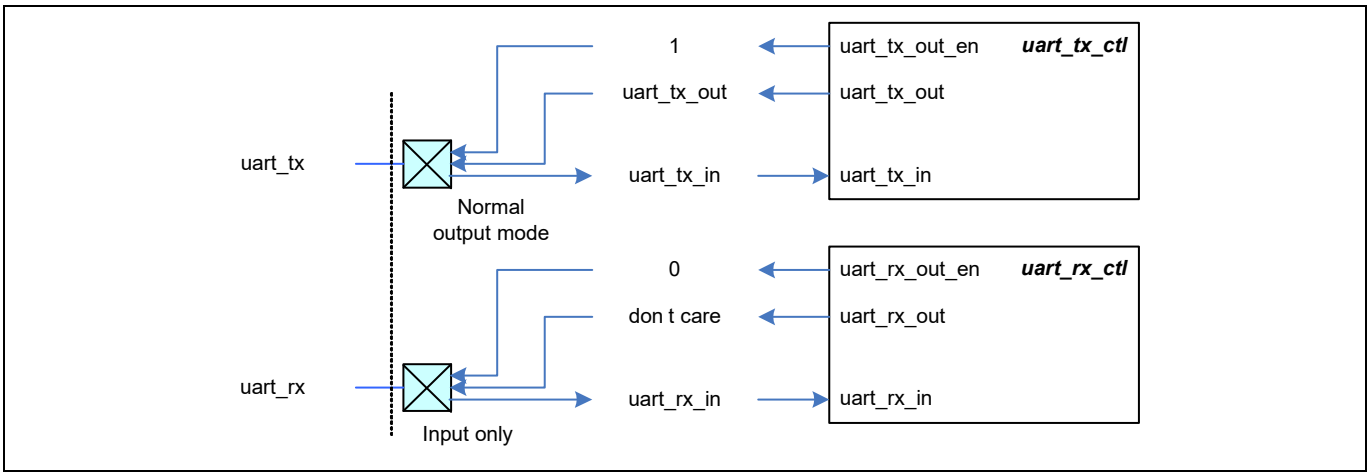


Figure 28-31. Standard UART mode I/O pad connections

Table 28-7. 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

28.4.6.2 SmartCard mode

Figure 28-32 and **Table 28-8** list the use of the I/O pads for the SmartCard mode.

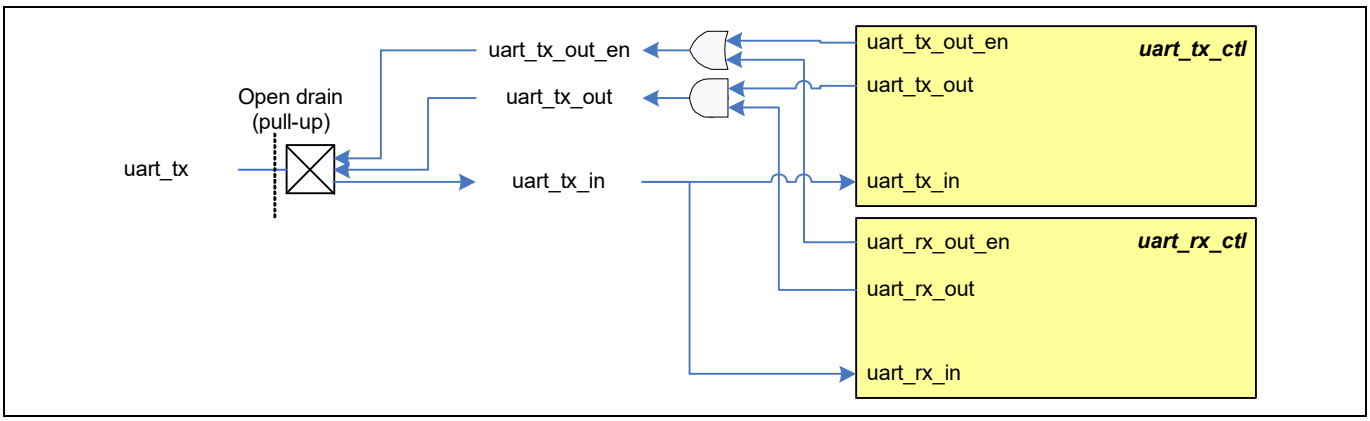


Figure 28-32. SmartCard mode I/O pad connections

Table 28-8. SmartCard mode I/O pad connections

I/O pads	Drive mode	On-chip I/O signals	Usage
uart_tx	Open drain with pull-up	uart_tx_in	Used to receive a data element Receive a negative acknowledgement of a transmitted data element
		uart_tx_out_en uart_tx_out	Transmit a data element Transmit a negative acknowledgement to a received data element

28.4.6.3 LIN mode

Figure 28-33 and **Table 28-9** list the use of the I/O pads for LIN mode.

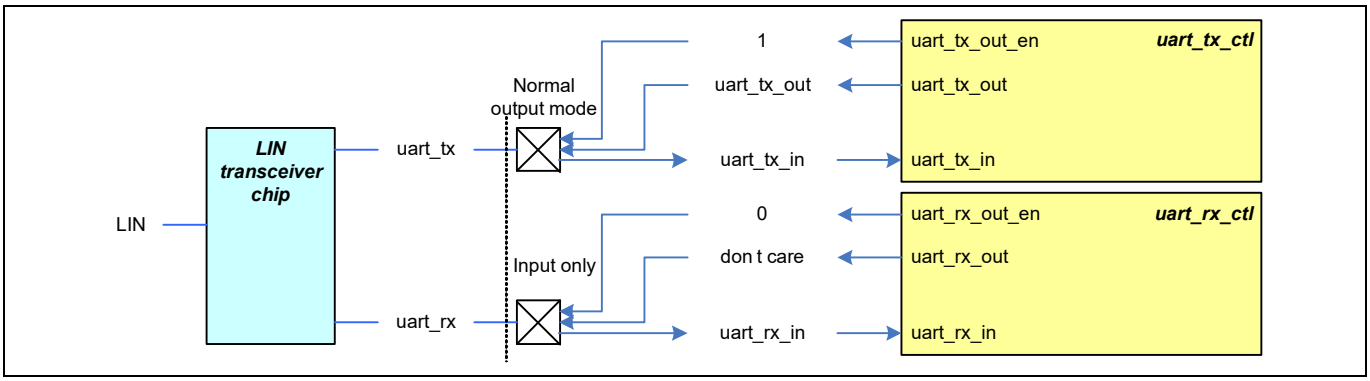


Figure 28-33. LIN mode I/O pad connections

Table 28-9. 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

28.4.6.4 IrDA mode

Figure 28-34 and Table 28-10 list the use of the I/O pads for IrDA mode.

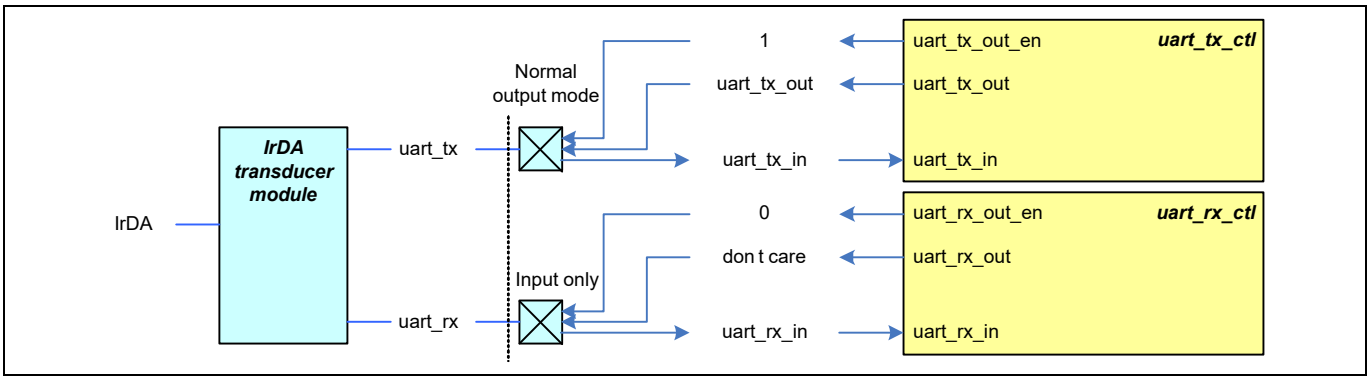


Figure 28-34. IrDA mode I/O pad connections

Table 28-10. 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

28.4.7 UART registers

The UART interface is controlled using a set of 32-bit registers listed in Table 28-11. For more information on these registers, see the registers reference manual.

Table 28-11. UART registers

Register name	Operation
SCB_CTRL	Enables the SCB; selects the type of serial interface (SPI, UART, I^2C)
SCB_UART_CTRL	Used to select the sub-modes of UART (standard UART, SmartCard, IrDA), also used for local loop back control.
SCB_UART_RX_STATUS	Used to specify the BR_COUNTER value that determines the bit period. This is used to set the accuracy of the SCB clock. This value provides more granularity than the OVS bit in SCB_CTRL register.
SCB_UART_TX_CTRL	Used to specify the number of stop bits, enable parity, select the type of parity, and enable retransmission on NACK.
SCB_UART_RX_CTRL	Performs same function as SCB_UART_TX_CTRL but is also used for enabling multi processor mode, LIN mode drop on parity error, and drop on frame error.
SCB_TX_CTRL	Used to specify the data frame width and to specify whether MSb or LSb is the first bit in transmission.
SCB_RX_CTRL	Performs the same function as that of the SCB_TX_CTRL register, but for the receiver. Also decides whether a median filter is to be used on the input interface lines.
SCB_UART_FLOW_CONTROL	Configures flow control for UART transmitter.

28.5 Inter integrated circuit (I^2C)

This section explains the I^2C implementation in the EZ-USB™ FX10 MCU. For more information on the I^2C protocol specification, refer to the I^2C-bus specification available on the **NXP website**.

28.5.1 Features

This block supports the following features:

- Master, slave, and master/slave mode
- Standard-mode (100 kbps), fast-mode (400 kbps), and fast-mode plus (1000 kbps) data-rates
- 7-bit slave addressing
- Clock stretching
- Collision detection
- Programmable oversampling of I^2C clock signal (SCL)
- Auto ACK when RX FIFO not full, including address
- General address detection
- FIFO Mode
- EZ and CMD_RESP modes

28.5.2 General description

Figure 28-35 illustrates an example of an I^2C communication network.

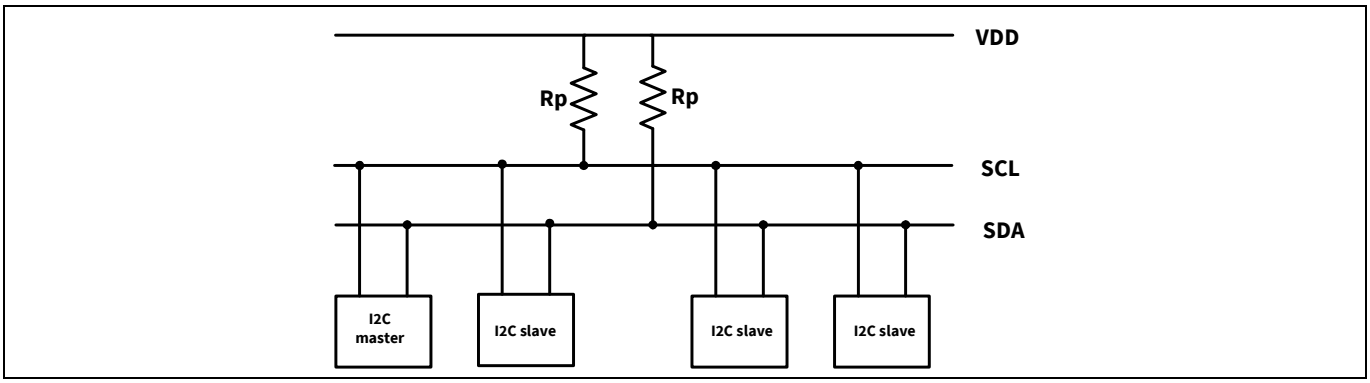


Figure 28-35. I^2C interface block diagram

The standard I^2C bus is a two wire interface with the following lines:

- Serial data (SDA)
- Serial clock (SCL)

I^2C devices are connected to these lines using open collector or open-drain output stages, with pull-up resistors (Rp). A simple master/slave relationship exists between devices. Masters and slaves can operate as either transmitter or receiver. Each slave device connected to the bus is software addressable by a unique 7-bit address.

28.5.3 External electrical connections

As shown in **Figure 28-36**, the I²C bus requires external pull-up resistors. The pull-up resistors (R_P) are primarily determined by the supply voltage, bus speed, and bus capacitance. For detailed information on how to calculate the optimum pull-up resistor value for your design Cypress recommends using the UM10204 I²C-bus specification and user manual Rev. 6, available from the NXP website at **www.nxp.com.**

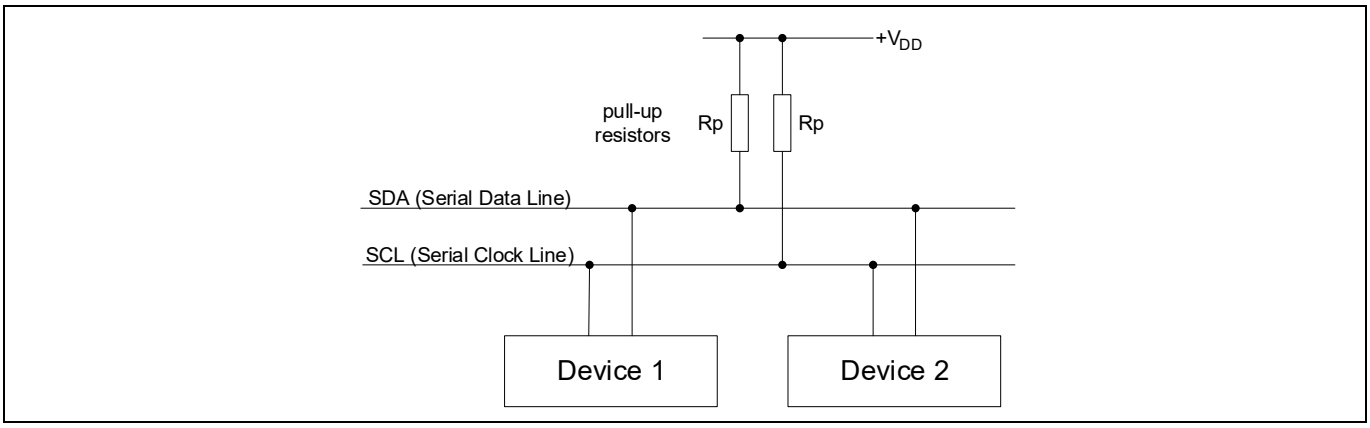


Figure 28-36. Connection of devices to the I2C bus

For most designs, the default values shown in **Table 28-12** provide excellent performance without any calculations. The default values were chosen to use standard resistor values between the minimum and maximum limits.

Table 28-12. Recommended default pull-up resistor values

Standard Mode (0 – 100 kbps)	Fast Mode (0 – 400 kbps)	Fast Mode Plus (0 – 1000 kbps)	Unit
4.7 k, 5%	1.74 k, 1%	620, 5%	W

These values work for designs with 1.8 V to 5.0 V V_{DD}, less than 200 pF bus capacitance (CB), up to 25 µA of total input leakage (I_{IL}), up to 0.4 V output voltage level (V_{OL}), and a max V_{IH} of 0.7 * V_{DD}. Calculation of custom pull-up resistor values is required if your design does not meet the default assumptions, you use series resistors (R_S) to limit injected noise, or you want to maximize the resistor value for low power consumption. Calculation of the ideal pull-up resistor value involves finding a value between the limits set by three equations detailed in the NXP I²C specification. These equations are:

$$R_{PMIN} = (V_{DD}(max) - V_{OL}(max)) / I_{OL}(min) \quad (28.4)$$

$$R_{PMAX} = T_R(max) / 0.8473 \times C_B(max) \quad (28.5)$$

$$R_{PMAX} = V_{DD}(min) - (VIH(min) + VNH(min)) / IIH(max) \quad (28.6)$$

Equation parameters:

- V_{DD} = Nominal supply voltage for I^2C bus
- V_{OL} = Maximum output low voltage of bus devices
- I_{OL}= Low-level output current from I^2C specification
- T_R = Rise time of bus from I^2C specification
- C_B = Capacitance of each bus line including pins and PCB traces
- V_{IH} = Minimum high-level input voltage of all bus devices
- V_{NH} = Minimum high-level input noise margin from I^2C specification
- I_{IH} = Total input leakage current of all devices on the bus

The supply voltage (V_{DD}) limits the minimum pull-up resistor value due to bus devices maximum low output voltage (V_{OL}) specifications. Lower pull-up resistance increases current through the pins and can therefore exceed the spec conditions of V_{OH}. **Equation 28.4** is derived using Ohm's law to determine the minimum resistance that will still meet the V_{OL} specification at 3 mA for standard and fast modes, and 20 mA for fast mode plus at the given V_{DD}.

Equation 28.5 determines the maximum pull-up resistance due to bus capacitance. Total bus capacitance is comprised of all pin, wire, and trace capacitance on the bus. The higher the bus capacitance the lower the pull-up resistance required to meet the specified bus speeds rise time due to RC delays. Choosing a pull-up resistance higher than allowed can result in failing timing requirements resulting in communication errors. Most designs with five of fewer I^2C devices and up to 20 centimeters of bus trace length have less than 100 pF of bus capacitance.

A secondary effect that limits the maximum pull-up resistor value is total bus leakage calculated in **Equation 28.5**. The primary source of leakage is I/O pins connected to the bus. If leakage is too high, the pull-ups will have difficulty maintaining an acceptable V_{IH} level causing communication errors. Most designs with five or fewer I^2C devices on the bus have less than 10 µA of total leakage current.

28.5.4 Terms and definitions

Table 28-13 explains the commonly used terms in an I^2C communication network.

Table 28-13. Definition of I^2C 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
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

28.5.4.1 Clock stretching

When a slave device is not yet ready to process data, it may drive a '0' on the SCL line to hold it down. Due to the implementation of the I/O signal interface, the SCL line value will be '0', independent of the values that any other master or slave may be driving on the SCL line. This is known as clock stretching and is the only situation in which a slave drives the SCL line. The master device monitors the SCL line and detects it when it cannot generate a positive clock pulse ('1') on the SCL line. It then reacts by delaying the generation of a positive edge on the SCL line, effectively synchronizing with the slave device that is stretching the clock. The SCB on the EZ-USB™ FX10 MCU can and will stretch the clock.

28.5.4.2 Bus arbitration

The I^2C protocol is a multi-master, multi-slave interface. Bus arbitration is implemented on master devices by monitoring the SDA line. Bus collisions are detected when the master observes an SDA line value that is not the same as the value it is driving on the SDA line. For example, when master 1 is driving the value '1' on the SDA line and master 2 is driving the value '0' on the SDA line, the actual line value will be '0' due to the implementation of the I/O signal interface. Master 1 detects the inconsistency and loses control of the bus. Master 2 does not detect any inconsistency and keeps control of the bus.

28.5.5 I^2C modes of operation

I^2C is a synchronous single master, multi-master, multi-slave serial interface. Devices operate in either master mode, slave mode, or master/slave mode. In master/slave mode, the device switches from master to slave mode when it is addressed. Only a single master may be active during a data transfer. The active master is responsible for driving the clock on the SCL line. **Table 28-14** illustrates the I^2C modes of operation.

Table 28-14. I^2C modes

Mode	Description
Slave	Slave only operation (default)
Master	Master only operation
Multi-master	Supports more than one master on the bus

Table 28-15 lists some common bus events that are part of an I^2C data transfer. The **"Write transfer"** on page 483 and **"Read transfer"** on page 483 sections explain the I^2C bus bit format during data transfer.

Table 28-15. I^2C bus events terminology

Bus event	Description
START	A HIGH to LOW transition on the SDA line while SCL is HIGH
STOP	A LOW to HIGH transition on the SDA line while SCL is HIGH
ACK	The receiver pulls the SDA line LOW and it remains LOW during the HIGH period of the clock pulse, after the transmitter transmits each byte. This indicates to the transmitter that the receiver received the byte properly.
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 did not receive the byte properly.
Repeated START	START condition generated by master at the end of a transfer instead of a STOP condition
DATA	SDA status change while SCL is LOW (data changing), and no change while SCL is HIGH (data valid).

With all of these modes, there are two types of transfer - read and write. In write transfer, the master sends data to slave; in read transfer, the master receives data from slave.

28.5.5.1 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 and a write indicator (‘0’) after the START condition. The addressed slave transmits an acknowledgment byte by pulling the data line low during the ninth bit time.
- If the slave address does not match any of the slave devices or if the addressed device does not want to acknowledge the request, it transmits a no acknowledgment (NACK) by not pulling the SDA line low. The absence of an acknowledgement, results in an SDA line value of ‘1’ due to the pull-up resistor implementation.
- If no acknowledgment is transmitted by the slave, the master may end the write transfer with a STOP event. The master can also generate a repeated START condition for a retry attempt.
- The master may transmit data to the bus if it receives an acknowledgment. The addressed slave transmits an acknowledgment to confirm the receipt of every byte of data written. Upon receipt of this acknowledgment, the master may transmit another data byte.
- When the transfer is complete, the master generates a STOP condition

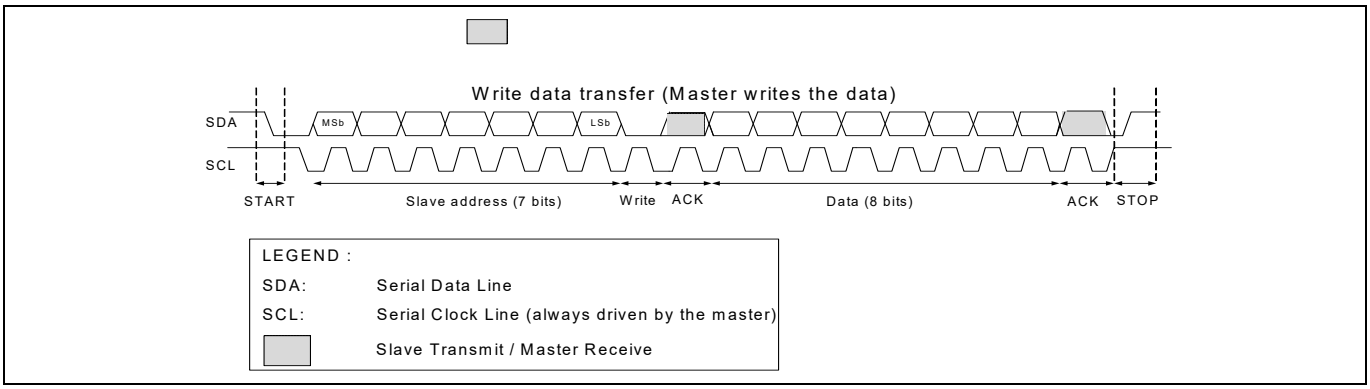


Figure 28-37. Master write data transfer

28.5.5.2 Read transfer

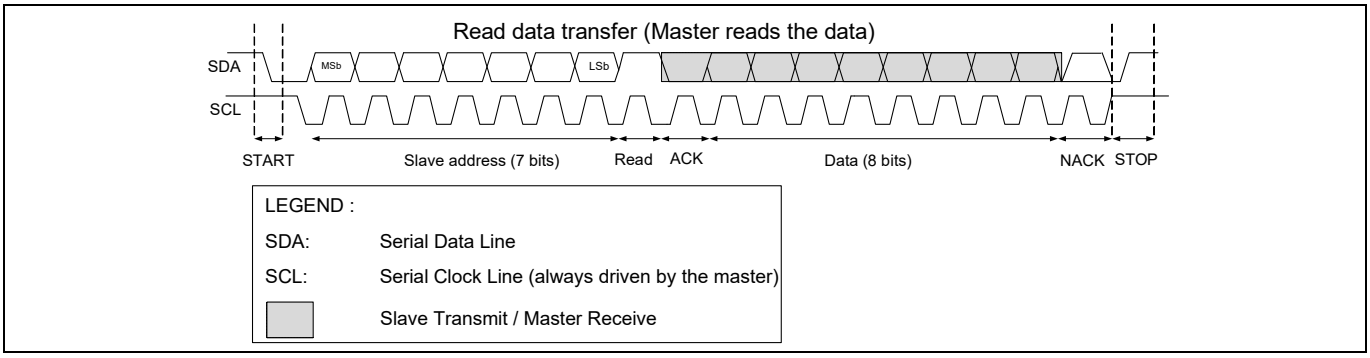


Figure 28-38. Master read data transfer

- 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 and a read indicator (‘1’) after the START condition. The addressed slave transmits an acknowledgment by pulling the data line low during the ninth bit time.
- If the slave address does not match with that of the connected slave device or if the addressed device does not want to acknowledge the request, a no acknowledgment (NACK) is transmitted by not pulling the SDA line low. The absence of an acknowledgment, results in an SDA line value of ‘1’ due to the pull-up resistor implementation.
- If no acknowledgment is transmitted by the slave, the master may end the read transfer with a STOP event. The master can also generate a repeated START condition for a retry attempt.
- If the slave acknowledges the address, it starts transmitting data after the acknowledgment signal. The master transmits an acknowledgment to confirm the receipt of each data byte sent by the slave. Upon receipt of this acknowledgment, the addressed slave may transmit another data byte.
- The master can send a NACK signal to the slave to stop the slave from sending data bytes. This completes the read transfer.
- When the transfer is complete, the master generates a STOP condition

28.5.6 I²C Buffer modes

I²C can operate in three different buffered modes – FIFO, EZ, and CMD_RESP modes. The buffer is used in different ways in each of the modes. The following subsections explain each of these buffered modes in detail.

FIFO mode

The FIFO mode has a TX FIFO for the data being transmitted and an RX FIFO for the data being received. Each FIFO is constructed out of the SRAM buffer. The FIFOs are either 64 elements deep with 16-bit data elements or 128 elements deep with 8-bit data elements. The width of the data elements are configured using the CTRL.BYTE_MODE bitfield of the SCB. For I²C, put the FIFO in BYTE mode because all transactions are a byte wide.

The FIFO mode operation is available only in Active and Sleep power modes, not in the Deep Sleep power mode. However, on the Deep Sleep-capable SCB the slave address can be used to wake the device from sleep.

A write access to the transmit FIFO uses register TX_FIFO_WR. A read access from the receive FIFO uses register RX_FIFO_RD.

Transmit and receive FIFO status information is available through status registers TX_FIFO_STATUS and RX_FIFO_STATUS. When in Debug mode, a read from this register behaves as a read from the SCB_RX_FIFO_RD_SILENT register; that is, data will not be removed from the FIFO.

Each FIFO has a trigger output. This trigger output can be routed through the trigger mux to various other peripheral on the device such as DMA or TCPWMs. The trigger output of the SCB is controlled through the TRIGGER_LEVEL field in the RX_CTRL and TX_CTRL registers.

- For a TX FIFO a trigger is generated when the number of entries in the transmit FIFO is less than TX_FIFO_CTRL.TRIGGER_LEVEL
- For the RX FIFO a trigger is generated when the number of entries in the FIFO is greater than the RX_FIFO_CTRL.TRIGGER_LEVEL

Note that the DMA has a trigger deactivation setting. For the SCB this should be set to 16.

- **Active to Deep Sleep transition**

  Before going to Deep Sleep ensure that all active communication is complete. This can be done by checking the BUS_BUSY bit in the I2C_Status register.

  Ensure that the TX and RX FIFOs are empty as any data will be lost during Deep Sleep.

  Before going to Deep Sleep the clock to the SCB needs to be disabled. This can be done by setting the SDA_IN_FILT_TRIM[1] bit in the I2C_CFG register to '0'.

- **Deep Sleep to Active transition**

  EC_AM = 1, EC_OP = 0, FIFO Mode.

  The following descriptions only apply to slave mode.

- **Master Write**
  - I2C_CTRL.S_NOT_READY_ADDR_NACK = 0, I2C_CTRL.S_READY_ADDR_ACK = 1. The clock is stretched until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the clock stretch will be released
  - I2C_CTRL.S_NOT_READY_ADDR_NACK = 0, I2C_CTRL.S_READY_ADDR_ACK = 0. The clock is stretched until SDA_IN_FILT_TRIM[1] is set to '1', and S_ACK or S_NACK is set in the I2C_S_CMD register.
  - I2C_CTRL.S_NOT_READY_ADDR_NACK = 1, I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is NACK'd until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the slave will respond with an ACK to a master address.
- **Master Read**
  - I2C_CTRL.S_NOT_READY_ADDR_NACK = 0, I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is stretched until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the clock stretch will be released.
  - I2C_CTRL.S_NOT_READY_ADDR_NACK = 1, I2C_CTRL.S_READY_ADDR_ACK = x. The incoming address is NACK'd until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the slave will ACK.

Note: *When doing a repeated start after a write, wait until the UNDERFLOW interrupt status is asserted before setting the I2C_M_CMD.START bit and writing the new address into the TX_FIFO. Otherwise, the address in the FIFO will be sent as data and not as an address.*

28.5.6.1 EZI2C mode

The Easy I²C (EZI2C) protocol is a unique communication scheme built on top of the I²C protocol by Cypress. It uses a meta protocol around the standard I²C protocol to communicate to an I²C slave using indexed memory transfers. This removes the need for CPU intervention.

The EZI2C protocol defines a single memory buffer with an 8-bit address that indexes the buffer (256-entry array of 8-bit per entry is supported) located on the slave device. The EZ address is used to address these 256 locations. The CPU writes and reads to the memory buffer through the EZ_DATA registers. These accesses are word accesses, but only the least significant byte of the word is used.

The slave interface accesses the memory buffer using the current address. At the start of a transfer (I²C START/RESTART), the base address is copied to the current address. A data element write or read operation is to the current address location. After the access, the current address is incremented by '1'.

If the current address equals the last memory buffer address (255), the current address is not incremented. Subsequent write accesses will overwrite any previously written value at the last buffer address. Subsequent read accesses will continue to provide the (same) read value at the last buffer address. The bus master should be aware of the memory buffer capacity in EZ mode.

The I²C base and current addresses are provided through I2C_STATUS. At the end of a transfer, the difference between the base and current addresses indicates how many read or write accesses were performed. The block provides interrupt cause fields to identify the end of a transfer. EZI2C can be implemented through firmware or hardware. All SCBs can implement EZI2C through a firmware implementation in both Active and Sleep power modes. The Deep Sleep SCB can implement a hardware-and firmware-based EZI2C with a Deep Sleep power mode. This document focuses on hardware-implemented EZI2C; for more information on software implementation, see the PDL.

EZI2C distinguishes three operation phases:

- Address phase: The master transmits an 8-bit address to the slave. This address is used as the slave base and current address.
- Write phase: The master writes 8-bit data element(s) to the slave's memory buffer. The slave's current address is set to the slave's base address. Received data elements are written to the current address memory location. After each memory write, the current address is incremented.
- Read phase: The master reads 8-bit data elements from the slave's memory buffer. The slave's current address is set to the slave's base address. Transmitted data elements are read from the current address memory location. After each memory read, the current address is incremented.

Note that a slave's base address is updated by the master and not by the CPU.

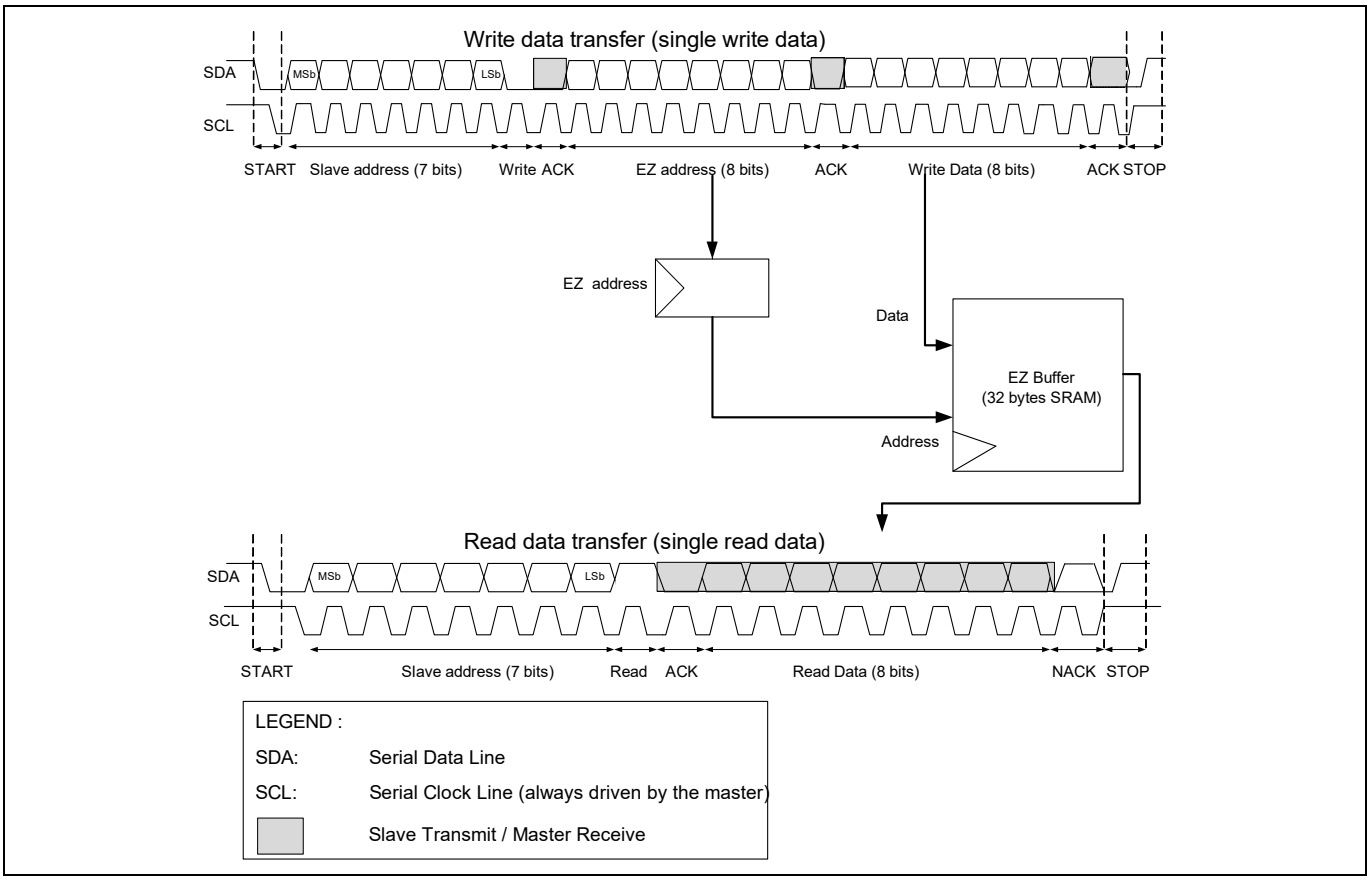


Figure 28-39. EZI2C Write and Read Data transfer

- **Active to Deep Sleep transition**

  Before going to Deep Sleep ensure that all active communication is complete. This can be done by checking the BUS_BUSY bit in the I2C_Status register.

  Ensure that the TX and RX FIFOs are empty as any data will be lost during Deep Sleep.

  Before going to Deep Sleep the clock to the SCB needs to be disabled. This can be done by setting the SDA_IN_FILT_TRIM[1] bit in the I2C_CFG register to '0'.

- **Deep Sleep to Active transition**

  EC_AM = 1, EC_OP = 0, EZ Mode.

  – S_NOT_READY_ADDR_NACK = 0, S_READY_ADDR_ACK = 1. The clock is stretched until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the clock stretch will be released.
  – S_NOT_READY_ADDR_NACK = 1, S_READY_ADDR_ACK = x. The incoming address is NACK'd until SDA_IN_FILT_TRIM[1] is set to '1'. After that bit is set to 1, the slave will ACK.

28.5.6.2 Command-Response mode

This mode has a single memory buffer, a base read address, a current read address, a base write address, and a current write address that are used to index the memory buffer. The base addresses are provided by the CPU. The current addresses are used by the slave to index the memory buffer for sequential accesses of the memory buffer. The memory buffer holds 256 8-bit data elements. The base and current addresses are in the range [0 to 255].

The CPU writes and reads to the memory buffer through the SCB_EZ_DATA registers. These are word accesses, but only the least significant byte of the word is used.

The slave interface accesses the memory buffer using the current addresses. At the start of a write transfer (I^2C START/RESTART), the base write address is copied to the current write address. A data element write is to the current write address location. After the write access, the current address is incremented by '1'. At the start of a read transfer, the base read address is copied to the current read address. A data element read is to the current read address location. After the read data element is transmitted, the current read address is incremented by '1'.

If the current addresses equal the last memory buffer address (255), the current addresses are not incremented. Subsequent write accesses will overwrite any previously written value at the last buffer address. Subsequent read accesses will continue to provide the (same) read value at the last buffer address. The bus master should be aware of the memory buffer capacity in command-response mode.

The base addresses are provided through CMD_RESP_CTRL. The current addresses can be viewed in CMD_RESP_STATUS. At the end of a transfer (I^2C stop), the difference between a base and current address indicates how many read/write accesses were performed. This block provides interrupts to identify the end of a transfer, which can be found in SCB8_INTR_I2C_EC and SCB8_INTR_SPI_EC register sections. Command-response mode operation is available in Active, Sleep, and Deep Sleep power modes. The command-response mode has two phases of operation:

- Write phase - The write phase begins with a START/RESTART followed by the slave address with read/write bit set to '0' indicating a write. The slave's current write address is set to the slave's base write address. Received data elements are written to the current write address memory location. After each memory write, the current write address is incremented.
- Read phase - The read phase begins with a START/RESTART followed by the slave address with read/write bit set to '1' indicating a read. The slave's current read address is set to the slave's base read address. Transmitted data elements are read from the current address memory location. After each read data element is transferred, the current read address is incremented.

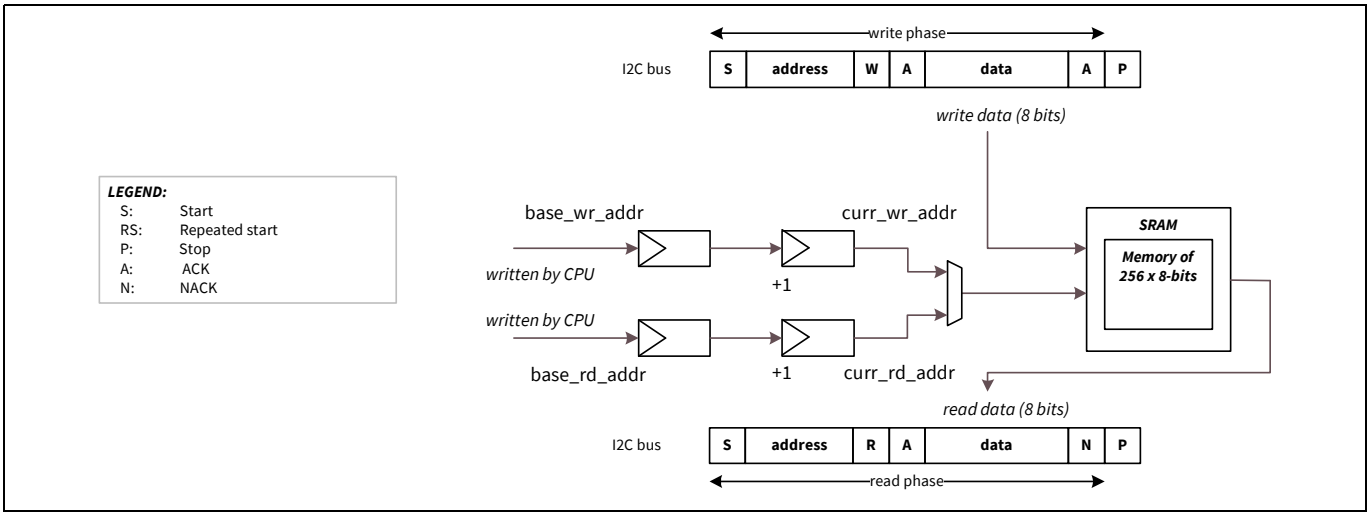


Figure 28-40. I^2C Command-Response mode

Note: A slave's base addresses are updated by the CPU and not by the master.

28.5.7 Clocking and oversampling

The SCB I²C supports both internally and externally clocked operation modes. Two bitfields (EC_AM_MODE and EC_OP MODE) in the SCB_CTRL register determine the SCB clock mode. EC_AM_MODE indicates whether I²C address matching is internally (0) or externally (1) clocked. I²C address matching comprises the first part of the I²C protocol. EC_OP_MODE indicates whether the rest of the protocol operation (besides I²C address matching) is internally (0) or externally (1) clocked. The externally clocked mode of operation is supported only in the I²C slave mode.

An internally-clocked operation uses the programmable clock dividers. For I²C, an integer clock divider must be used for both master and slave. For more information on system clocking, see the **"Clocking system"** on page 290. The internally-clocked mode does not support the command-response mode.

The SCB_CTRL bitfields EC_AM_MODE and EC_OP_MODE can be configured in the following ways.

- EC_AM_MODE is '0' and EC_OP_MODE is '0': Use this configuration when only Active mode functionality is required.
  - FIFO mode: Supported
  - EZ mode: Supported
  - Command-response mode: Not supported. The slave NACKs every slave address.
- EC_AM_MODE is '1' and EC_OP_MODE is '0': Use this configuration when both Active and Deep Sleep functionality are required. This configuration relies on the externally clocked functionality for the I²C address matching and relies on the internally clocked functionality to access the memory buffer. The "hand over" from external to internal functionality relies either on an ACK/NACK or clock stretching scheme. The former may result in termination of the current transfer and relies on a master retry. The latter stretches the current transfer after a matching address is received. This mode requires the master to support either NACK generation (and retry) or clock stretching. When the I²C address is matched, INTR_I2C_EC.WAKE_UP is set to '1'. The associated Deep Sleep functionality interrupt brings the system into Active power mode.
  - FIFO mode: See **"Deep Sleep to Active transition"** on page 485
  - EZ mode: See **"Deep Sleep to Active transition"** on page 487.
  - CMD_RESP mode: Not supported. The slave NACKs every slave address.
- EC_AM_MODE is '1' and EC_OP_MODE is '1'. Use this mode when both Active and Deep Sleep functionality are required. When the slave is selected, INTR_I2C_EC.WAKE_UP is set to '1'. The associated Deep Sleep functionality interrupt brings the system into Active power mode. When the slave is deselected, INTR_I2C_EC.EZ_STOP and/or INTR_I2C_EC.EZ_WRITE_STOP are set to '1'.
  - FIFO mode: Not supported
  - EZ mode: Supported
  - CMD_RESP mode: Supported

An externally-clocked operation uses a clock provided by the serial interface. The externally clocked mode does not support FIFO mode. If EC_OP_MODE is '1', the external interface logic accesses the memory buffer on the external interface clock (I²C SCL). This allows for EZ and CMD_RESP mode functionality in Active and Deep Sleep power modes.

In Active system power mode, the memory buffer requires arbitration between external interface logic (on I²C SCL) and the CPU interface logic (on system peripheral clock). This arbitration always gives the highest priority to the external interface logic (host accesses). The external interface logic takes one serial interface clock/bit periods for the I²C. During this period, the internal logic is denied service to the memory buffer. The EZ-USB™ FX10 MCU provides two programmable options to address this "denial of service":

- If the BLOCK bitfield of SCB_CTRL is '1': An internal logic access to the memory buffer is blocked until the memory buffer is granted and the external interface logic has completed access. For a 100-kHz I²C interface, the maximum blocking period of one serial interface bit period measures 10 µs (approximately 208 clock cycles on a 48 MHz SCB input clock). This option provides normal SCB register functionality, but the blocking time introduces additional internal bus wait states.
- If the BLOCK bitfield of SCB_CTRL is '0': An internal logic access to the memory buffer is not blocked, but fails when it conflicts with an external interface logic access. A read access returns the value 0xFFFF:FFFF and a write access is ignored. This option does not introduce additional internal bus wait states, but an access to the memory buffer may not take effect. In this case, following failures are detected:
  - Read Failure: A read failure is easily detected, as the returned value is 0xFFFF:FFFF. This value is unique as non-failing memory buffer read accesses return an unsigned byte value in the range 0x0000:0000-0x0000:00FF.
  - Write Failure: A write failure is detected by reading back the written memory buffer location, and confirming that the read value is the same as the written value.

For both options, a conflicting internal logic access to the memory buffer sets INTR_TX.BLOCKED field to '1' (for write access-es) and INTR_RX.BLOCKED field to '1' (for read accesses). These fields can be used as either status fields or as interrupt cause fields (when their associated mask fields are enabled).

If a series of read or write accesses is performed and CTRL.BLOCKED is '0', a failure is detected by comparing the logical OR of all read values to 0xFFFF:FFFF and checking the INTR_TX.BLOCKED and INTR_RX.BLOCKED fields to determine whether a failure occurred for a (series of) write or read operation(s).

28.5.7.1 Glitch filtering

The EZ-USB™ FX10 MCU SCB I²C has analog and digital glitch filters. Analog glitch filters are applied on the i2c_scl_in and i2c_sda_in input signals (AF_in) to filter glitches of up to 50 ns. An analog glitch filter is also applied on the i2c_sda_out output signal (AF_out). Analog glitch filters are enabled and disabled in the SCB.I2C_CFG register. Do not change the _TRIM bitfields; only change the _SEL bitfields in this register.

Digital glitch filters are applied on the i2c_scl_in and i2c_sda_in input signals (DF_in). The digital glitch filter is enabled in the SCB.RX_CTRL register via the MEDIAN bitfield.

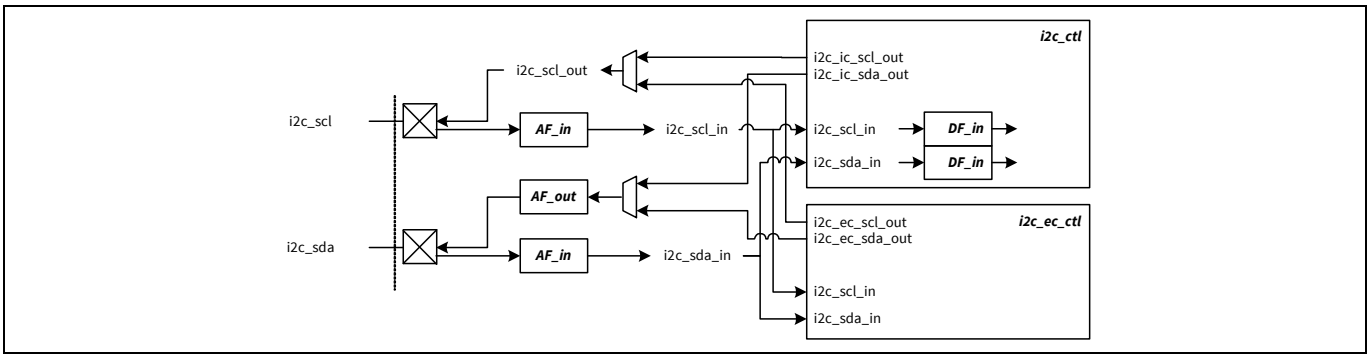


Figure 28-41. I²C glitch filtering connection

Table 28-16 lists the useful combinations of glitch filters.

Table 28-16. Glitch filter combinations

AF_in	AF_out	DF_in	Comments
0	0	1	Used when operating in internally-clocked mode and in master in Fast-Mode Plus (1-MHz speed mode)
1	0	0	Used when operating in internally-clocked mode (EC_OP_MODE is '0')
1	1	0	Used when operating in externally-clocked mode (EC_OP_MODE is '1'). Only slave mode.

When operating in EC_OP_MODE = 1, the 100-kHz, 400-kHz, and 1000-kHz modes require the following settings for AF_out:

AF_in	AF_out	DF_in	
1	1	0	100-kHz mode: I2C_CFG.SDA_OUT_FILT_SEL = 3 400-kHz mode: I2C_CFG.SDA_OUT_FILT_SEL = 3 1000-kHz mode: I2C_CFG.SDA_OUT_FILT_SEL = 1

28.5.7.2 Oversampling and bit rate

- **Internally-clocked master**

  The EZ-USB™ FX10 MCU implements the I²C clock as an oversampled multiple of the SCB input clock. In master mode, the block determines the I²C frequency. Routing delays on the PCB, on the chip, and the SCB (including analog and digital glitch filters) all contribute to the signal interface timing. In master mode, the block operates off clk_scb and uses programmable oversampling factors for the SCL high (1) and low (0) times. For high and low phase oversampling, see I2C_CTRL.LOW_PHASE_OVS and I2C_CTRL.HIGH_PHASE_OVS registers. For simple manipulation of the oversampling factor, see the SCB_CTRL.OVS register.

Table 28-17. I²C frequency and oversampling requirements in I²C Master mode

AF_in	AF_out	DF_in	Mode	Supported frequency	LOW_PHASE_OVS	HIGH_PHASE_OVS	clk_scb frequency
0	0	1	100 kHz	[62, 100] kHz	[9, 15]	[9, 15]	[1.98-3.2] MHz
			400 kHz	[264, 400] kHz	[13, 5]	[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 28-17 assumes worst-case conditions on the I²C bus. The following equations can be used to determine the settings for your own system. This will involve measuring the rise and fall times on SCL and SDA lines in your system.

$t_{CLK\_SCB(Min)} = (t_{LOW} + t_F)/LOW\_PHASE\_OVS$

If clk_scb is any faster than this, the t_{LOW} of the I²C specification will be violated. t_F needs to be measured in your system.

$t_{CLK\_SCB(Max)} = (t_{VD} - t_{RF} - 100\ ns)/3$ (When analog filter is enabled and digital disabled)

$t_{CLK\_SCB(Max)} = (t_{VD} - t_{RF})/4$ (When analog filter is disabled and analog filter is enabled)

t_{RF} is the maximum of either the rise or fall time. If clk_scb is slower than this frequency, t_{VD} will be violated.

- **I²C Master Clock synchronization**

  The HIGH_PHASE_OVS counter does not start counting until the SCB detects that the SCL line is high. This is not the same as when the SCB sets the SCL high. The differences are explained by three delays:

  – Delay from SCB to I/O pin
  – I²C bus t_R
  – Input delay (filters and synchronization)

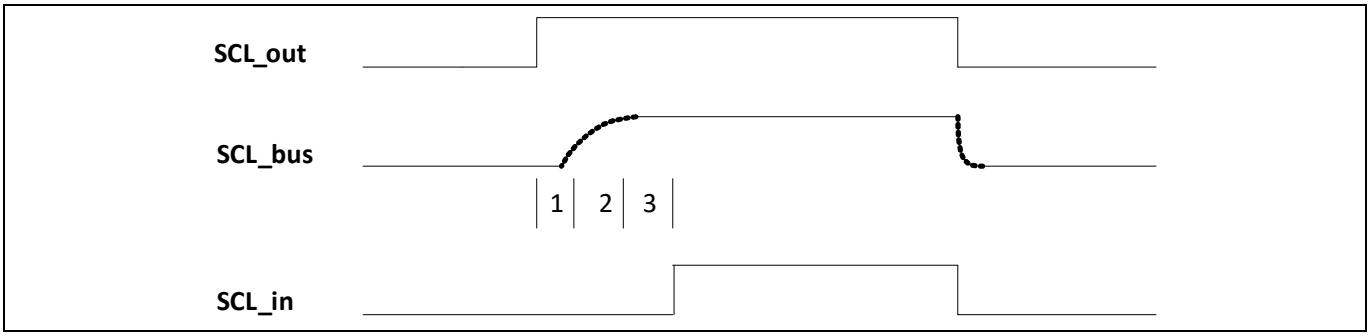


Figure 28-42. I²C SCL turnaround path

If the above three delays combined are greater than one clk_scb cycle, then the high phase of the SCL will be extended. This may cause the actual data rate on the I²C bus to be slower than expected. This can be avoided by:

– Decreasing the pull-up resistor, or decreasing the bus capacitance to reduce t_R
– Reducing the I2C_CTRL.HIGH_PHASE_OVS value

- **Internally-clocked Slave**

  In slave mode, the I²C frequency is determined by the incoming I²C SCL signal. To ensure proper operation, clk_scb must be significantly higher than the I²C bus frequency. Unlike master mode, this mode does not use programmable oversampling factors.

Table 28-18. 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

$t_{CLK\_SCB(Max)} = (t_{VD} - t_{RF} - 100\ ns) / 3$ (When analog filter is enabled and digital disabled)

$t_{CLK\_SCB(Max)} = (t_{VD} - t_{RF}) / 4$ (When analog filter is disabled and analog filter is enabled)

t_{RF} is the maximum of either the rise or fall time. If clk_scb is slower than this frequency, t_{VD} will be violated.

The minimum period of clk_scb is determined by one of the following equations:

$t_{CLK\_SCB(MIN)} = (t_{SU;DAT(min)} + t_{RF}) /16$

or

$t_{CLK\_SCB(Min)} = (0.6 * t_F - 50\ ns)/2$ (When analog filter is enabled and digital disabled)

$t_{CLK\_SCB(Min)} = (0.6 * t_F)/3$ (When analog filter is disabled and digital enabled)

The result that yields the largest period from the two sets of equations above should be used to set the minimum period of clk_scb.

- **Master-Slave**

  In this mode, when the SCB is acting as a master device, the block determines the I²C frequency. When the SCB is acting as a slave device, the block does not determine the I²C frequency. Instead, the incoming I²C SCL signal does.

  To guarantee operation in both master and slave modes, choose clock frequencies that work for both master and slave using the tables above.

28.5.8 Enabling and initializing the I²C

The following section describes the method to configure the I²C block for standard (non-EZ) mode and EZI2C mode.

28.5.8.1 Configuring for I²C FIFO mode

The I²C interface must be programmed in the following order.

1. Program protocol specific information using the SCB_I2C_CTRL register. This includes selecting master - slave functionality.
1. Program the generic transmitter and receiver information using the SCB_TX_CTRL and SCB_RX_CTRL registers.
2. Set the SCB_CTRL.BYTE_MODE to '1' to enable the byte mode.
3. Program the SCB_CTRL register to enable the I²C block and select the I²C mode. For a complete description of the I²C registers, see the registers reference manual.

28.5.8.2 Configuring for EZ and CMD_RESP modes

To configure the I²C block for EZ and CMD_RESP modes, set the following I²C register bits

1a. Select the EZI2C mode by writing '1' to the EZ_MODE bit (bit 10) of the SCB_CTRL register.

1b. Select CMD_RESP mode by writing a 1 to the CMD_RESP bit (bit 12) of the SCB_CTRL register.

2. Set the S_READY_ADDR_ACK (bit 12) and S_READY_DATA_ACK (bit 13) bits of the SCB_I2C_CTRL register.

Note: For all modes clk_scb must also be configured. For information on configuring a peripheral clock and connecting it to the SCB consult the ***"Clocking system"*** *on page 290.*

The GPIO must also be connected to the SCB; see the following section for more details.

28.5.9 I/O pad connections

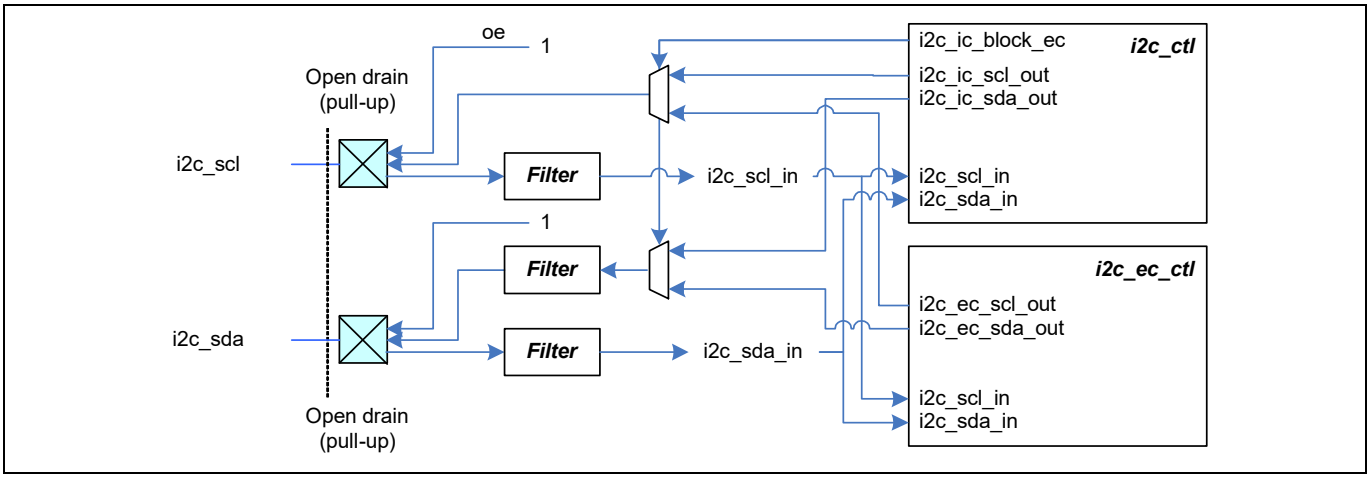


Figure 28-43. I²C I/O pad connections

Table 28-19. I²C I/O pad descriptions

I/O pads	Drive mode	On-chip I/O signals	Usage
i2c_scl	Open drain with external pull-up	i2c_scl_in	Receive a clock
		i2c_scl_out	Transmit a clock
i2c_sda	Open drain with external pull-up	i2c_sda_in	Receive data
		i2c_sda_out	Transmit data

When configuring the I²C SDA/SCL lines, the following sequence must be followed. If this sequence is not followed, the I²C lines may initially have overshoot and undershoot.

1. Set SCB_CTRL_MODE to '0'.
2. Configure HSIOM for SCL and SDA to connect to the SCB.
3. Set TX_CTRL.OPEN_DRAIN to '1'.
4. Configure I²C pins for high-impedance drive mode.
5. Configure SCB for I²C
6. Enable SCB
7. Configure I²C pins for Open Drain Drives Low.

28.5.10 I²C registers

The I²C interface is controlled by reading and writing a set of configuration, control, and status registers, as listed in Table 28-20.

Table 28-20. I²C registers

Register	Function
SCB_CTRL	Enables the SCB block and selects the type of serial interface (SPI, UART, I²C). Also used to select internally and externally clocked operation and EZ and non-EZ modes of operation.
SCB_I2C_CTRL	Selects the mode (master, slave) and sends an ACK or NACK signal based on receiver FIFO status.
SCB_I2C_STATUS	Indicates bus busy status detection, read/write transfer status of the slave/master, and stores the EZ slave address.
SCB_I2C_M_CMD	Enables the master to generate START, STOP, and ACK/NACK signals.
SCB_I2C_S_CMD	Enables the slave to generate ACK/NACK signals.
SCB_STATUS	Indicates whether the externally clocked logic is using the EZ memory. This bit can be used by software to determine whether it is safe to issue a software access to the EZ memory.
SCB_I2C_CFG	Configures filters, which remove glitches from the SDA and SCL lines.
SCB_TX_CTRL	Specifies the data frame width; also used to specify whether MSb or LSb is the first bit in transmission.
SCB_TX_FIFO_CTRL	Specifies the trigger level, clearing of the transmitter FIFO and shift registers, and FREEZE operation of the transmitter FIFO.
SCB_TX_FIFO_STATUS	Indicates the number of bytes stored in the transmitter FIFO, the location from which a data frame is read by the hardware (read pointer), the location from which a new data frame is written (write pointer), and decides if the transmitter FIFO holds the valid data.
SCB_TX_FIFO_WR	Holds the data frame written into the transmitter FIFO. Behavior is similar to that of a PUSH operation.
SCB_RX_CTRL	Performs the same function as that of the SCB_TX_CTRL register, but for the receiver. Also decides whether a median filter is to be used on the input interface lines.
SCB_RX_FIFO_CTRL	Performs the same function as that of the SCB_TX_FIFO_CTRL register, but for the receiver.
SCB_RX_FIFO_STATUS	Performs the same function as that of the SCB_TX_FIFO_STATUS register, but for the receiver.
SCB_RX_FIFO_RD	Holds the data read from the receiver FIFO. Reading a data frame removes the data frame from the FIFO; behavior is similar to that of a POP operation. This register has a side effect when read by software: a data frame is removed from the FIFO.
SCB_RX_FIFO_RD_SILENT	Holds the data read from the receiver FIFO. Reading a data frame does not remove the data frame from the FIFO; behavior is similar to that of a PEEK operation.
SCB_RX_MATCH	Stores slave device address and is also used as slave device address MASK.
SCB_EZ_DATA	Holds the data in an EZ memory location.

Note: Detailed descriptions of the I²C register bits are available in the registers reference manual.

28.6 SCB interrupts

SCB supports interrupt generation on various events. The interrupts generated by the SCB block vary depending on the mode of operation.

Table 28-21. SCB interrupts

Interrupt	Functionality	Active/ Deep Sleep	Registers
interrupt_master	I^2C master and SPI master functionality	Active	INTR_M, INTR_M_SET, INTR_M_MASK, INTR_M_MASKED
interrupt_slave	I^2C slave and SPI slave functionality	Active	INTR_S, INTR_S_SET, INTR_S_MASK, INTR_S_MASKED
interrupt_tx	UART transmitter and TX FIFO functionality	Active	INTR_TX, INTR_TX_SET, INTR_TX_MASK, INTR_TX_MASKED
interrupt_rx	UART receiver and RX FIFO functionality	Active	INTR_RX, INTR_RX_SET, INTR_RX_MASK, INTR_RX_MASKED
interrupt_i2c_ec	Externally clocked I^2C slave functionality	Deep Sleep	INTR_I2C_EC, INTR_I2C_EC_MASK, INTR_I2C_EC_MASKED

Note: To avoid being triggered by events from previous transactions, whenever the firmware enables an interrupt mask register bit, it should clear the interrupt request register in advance.

Note: If the DMA is used to read data out of RX FIFO, the NOT_EMPTY interrupt may never trigger. This can occur when clk_peri (clocking DMA) is running much faster than the clock to the SCB. As a workaround to this issue, set the RX_FIFO_CTRL.TRIGGER_LEVEL to '1' (not 0); this will allow the interrupt to fire.

The following register definitions correspond to the SCB interrupts:

- **INTR_M:** This register provides the instantaneous status of the interrupt sources. A write of '1' to a bit will clear the interrupt.
- **INTR_M_SET:** A write of '1' into this register will set the interrupt
- **INTR_M_MASK:** The bit in this register masks the interrupt sources. Only the interrupt sources with their masks enabled can trigger the interrupt.
- **INTR_M_MASKED:** This register provides the instantaneous value of the interrupts after they are masked. It provides logical and corresponding request and mask bits. This is used to understand which interrupt triggered the event.

Note: While registers corresponding to INTR_M are used here, these definitions can be used for INTR_S, INTR_TX, INTR_RX, INTR_I2C_EC, and INTR_SPI_EC.

Figure 28-44 shows the physical interrupt lines. All the interrupts are OR'd together to make one interrupt source that is the OR of all six individual interrupts. All the externally-clocked interrupts make one interrupt line called interrupt_ec, which is the OR'd signal of interrupt_i2C_ec and interrupt_spi_ec. All the internally-clocked interrupts make one interrupt line called interrupt_ic, which is the OR'd signal of interrupt_master, interrupt_slave, interrupt_tx, and interrupt_rx. The Active functionality interrupts are generated synchronously to clk_peri while the Deep Sleep functionality interrupts are generated asynchronously to clk_peri.

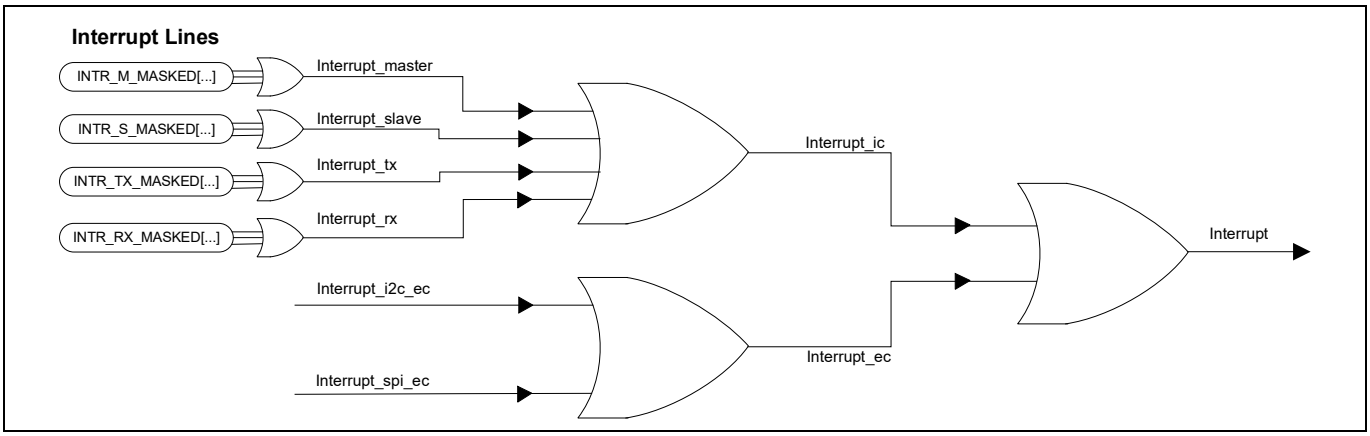


Figure 28-44. Interrupt lines

28.6.1 SPI interrupts

The SPI interrupts can be classified as master interrupts, slave interrupts, TX interrupts, RX interrupts, and externally clocked (EC) mode interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources are true. The SCB also provides an interrupt cause register (SCB_ INTR_CAUSE) that can be used to determine interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bitfield. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the registers reference manual. The SPI supports interrupts on the following events:

- SPI Master interrupts
  - SPI master transfer done – All data from the TX FIFO are sent. This interrupt source triggers later than TX_FIFO_EMPTY by the amount of time it takes to transmit a single data element. TX_FIFO_EMPTY triggers when the last data element from the TX FIFO goes to the shifter register. However, SPI Done triggers after this data element is transmitted. This means SPI Done will be asserted one SCLK clock cycle earlier than the completion of data element reception.
- SPI Slave interrupts
  - SPI Bus Error – Slave deselected at an unexpected time in the SPI transfer. The firmware may decide to clear the TX and RX FIFOs for this error.
  - SPI slave deselected after any EZSPI transfer occurred
  - SPI slave deselected after a write EZSPI transfer occurred

- SPI TX
  - TX FIFO has less entries than the value specified by TRIGGER_LEVEL in SCB_TX_FIFO_CTRL
  - TX FIFO not full – At least one data element can be written into the TX FIFO
  - TX FIFO empty – The TX FIFO is empty
  - TX FIFO overflow – Firmware attempts to write to a full TX FIFO
  - TX FIFO underflow – Hardware attempts to read from an empty TX FIFO. This happens when the SCB is ready to transfer data and EMPTY is ‘1’.
  - TX FIFO trigger – Less entries in the TX FIFO than the value specified by TX_FIFO_CTRL.TRIGGER_LEVEL
- SPI RX
  - RX FIFO has more entries than the value specified by TRIGGER_LEVEL in SCB_RX_FIFO_CTRL
  - RX FIFO full - RX FIFO is full
  - RX FIFO not empty - RX FIFO is not empty. At least one data element is available in the RX FIFO to be read
  - RX FIFO overflow - Hardware attempt to write to a full RX FIFO
  - RX FIFO underflow - Firmware attempts to read from and empty RX FIFO
  - RX FIFO trigger - More entries in the RX FIFO than the value specified by RX_FIFO_CTRL.TRIGGER_LEVEL.
- SPI externally clocked
  - Wake up request on slave select – Active on incoming slave request (with address match). Only set when EC_AM is ‘1’.
  - SPI STOP detection at the end of each transfer – Activated at the end of every transfer (I2C STOP). Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is ‘1’.
  - SPI STOP detection at the end of a write transfer – Activated at the end of a write transfer (I2C STOP). This event is an indication that a buffer memory location has been written to. For EZ mode, a transfer that only writes the base address does not activate this event. Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is ‘1’.
  - SPI STOP detection at the end of a read transfer – Activated at the end of a read transfer (I2C STOP). This event is an indication that a buffer memory location has been read from. Only set for a slave request with an address match, in EZ and CMD_RESP modes when EC_OP is ‘1’.

Figure 28-45 and **Figure 28-46** show how each of the interrupts are triggered. **Figure 28-45** shows the TX buffer and the corresponding interrupts while **Figure 28-46** shows all the corresponding interrupts for the RX buffer. The FIFO has 256 split into 128 bytes for TX and 128 bytes for RX instead of the 8 bytes shown in the figures. For more information on how to implement and clear interrupts, see the SPI (SCB_SPI_PDL) datasheet and the PDL.

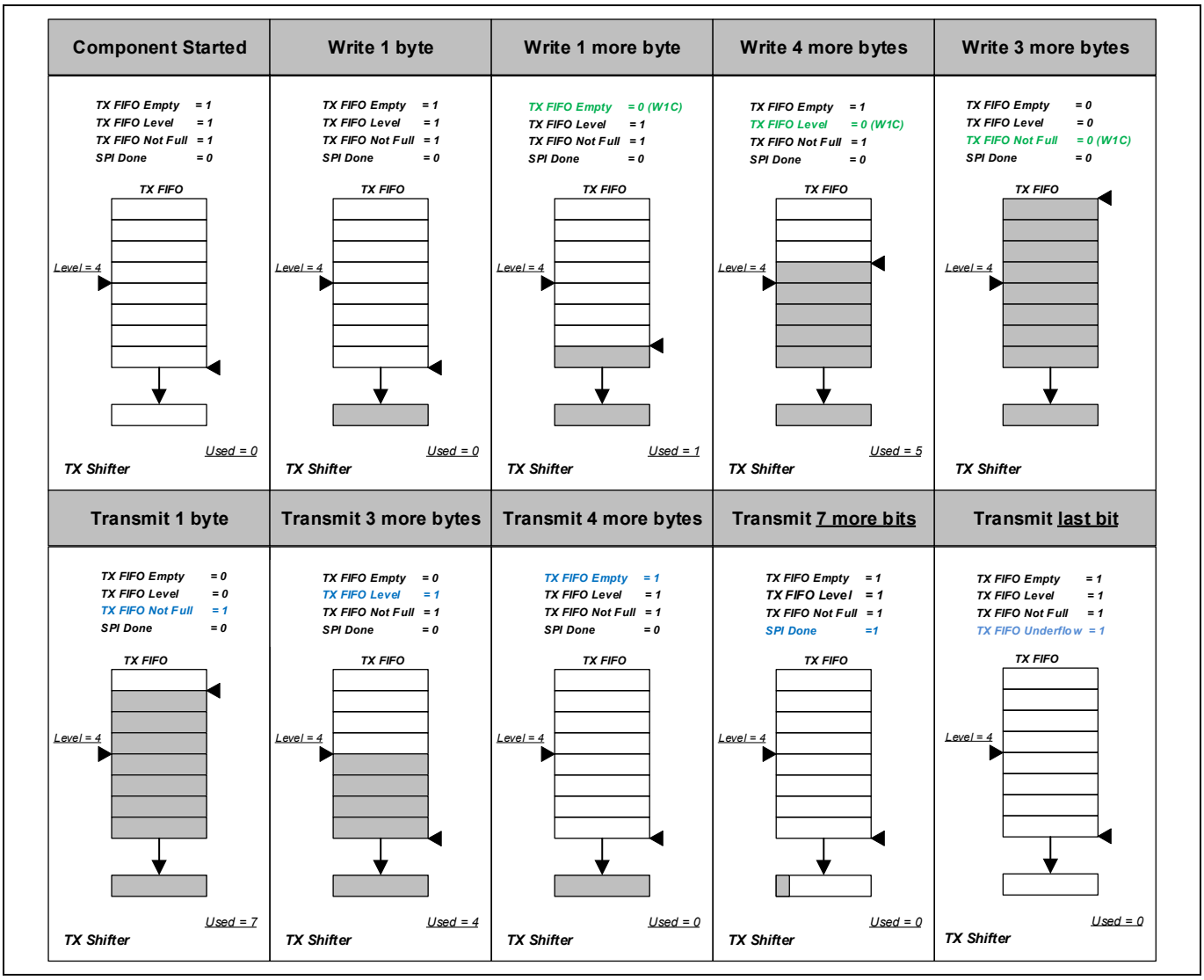


Figure 28-45. TX interrupt source operation

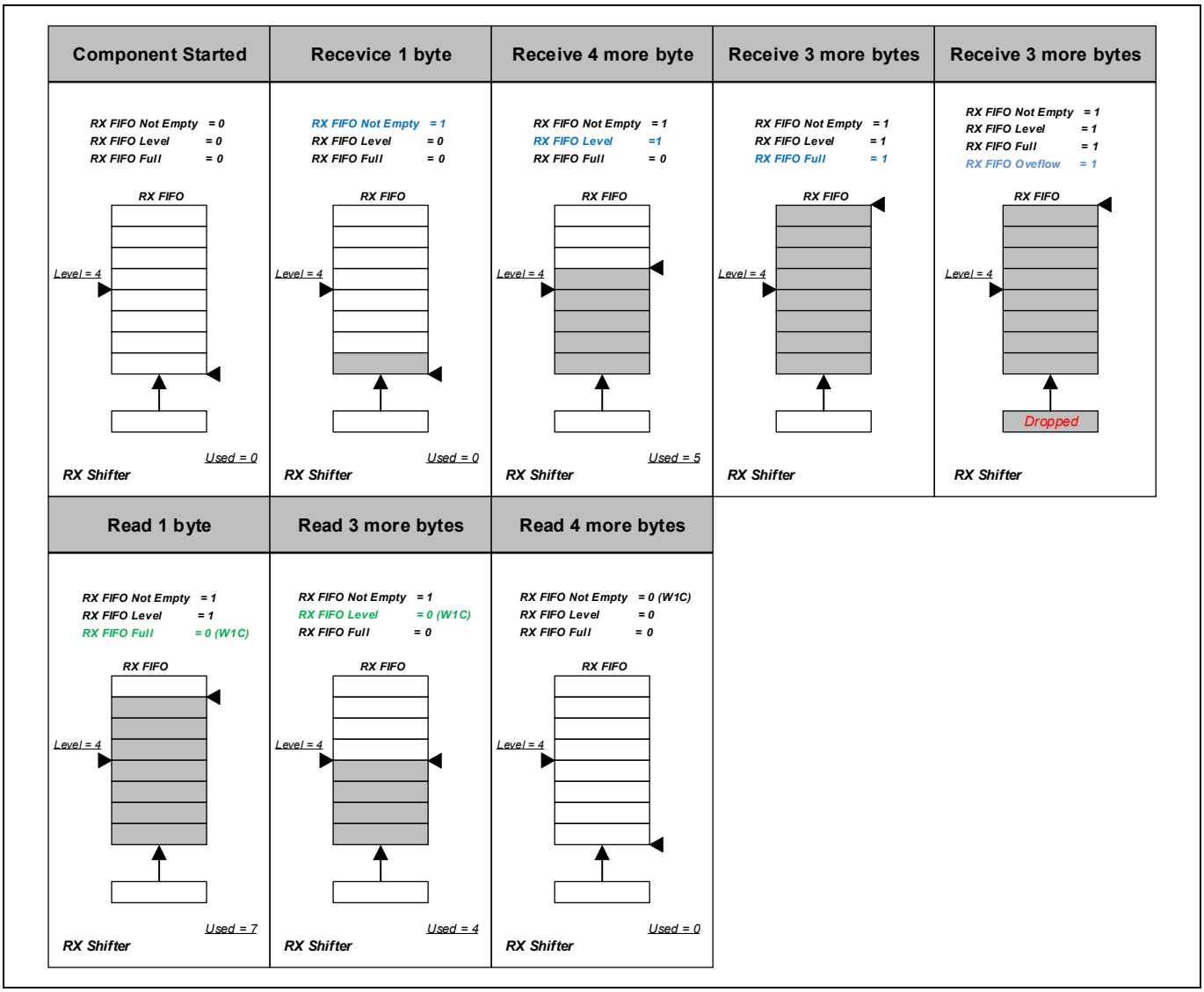


Figure 28-46. RX interrupt source operation

28.6.2 UART interrupts

The UART interrupts can be classified as TX interrupts and RX interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources are true. The SCB also provides an interrupt cause register (SCB_ INTR_CAUSE) that can be used to determine interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bitfield. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the registers reference manual. The UART block generates interrupts on the following events:

- UART TX
  - TX FIFO has fewer entries than the value specified by TRIGGER_LEVEL in SCB_TX_FIFO_CTRL
  - TX FIFO not full – TX FIFO is not full. At least one data element can be written into the TX FIFO
  - TX FIFO empty – The TX FIFO is empty
  - TX FIFO overflow – Firmware attempts to write to a full TX FIFO
  - TX FIFO underflow – Hardware attempts to read from an empty TX FIFO. This happens when the SCB is ready to transfer data and EMPTY is '1'
  - TX NACK – UART transmitter receives a negative acknowledgment in SmartCard mode.
  - TX done – This happens when the UART completes transferring all data in the TX FIFO and the last stop field is transmitted (both TX FIFO and transmit shifter register are empty)
  - TX lost arbitration – The value driven on the TX line is not the same as the value observed on the RX line. This condition event is useful when transmitter and receiver share a TX/RX line. This is the case in LIN or SmartCard modes.
- UART RX
  - RX FIFO has more entries than the value specified by TRIGGER_LEVEL in SCB_RX_FIFO_CTRL
  - RX FIFO full – RX FIFO is full. Note that received data frames are lost when the RX FIFO is full
  - RX FIFO not empty – RX FIFO is not empty
  - RX FIFO overflow – Hardware attempts to write to a full RX FIFO
  - RX FIFO underflow – Firmware attempts to read from an empty RX FIFO
  - Frame error in received data frame – UART frame error in received data frame. This can be either a start of stop bit error:
    Start bit error: After the beginning of a start bit period is detected (RX line changes from 1 to 0), the middle of the start bit period is sampled erroneously (RX line is '1').
    Stop bit error: *The RX line is sampled as '0', but a '1' was expected. A stop bit error may result in failure to receive successive data frames.*

Note: A start bit error is detected before a data frame is received.

Note: A stop bit error is detected after a data frame is received.

  - Parity error in received data frame – If UART_RX_CTL.DROP_ON_PARITY_ERROR is '1', the received frame is dropped. If UART_RX_CTL.DROP_ON_PARITY_ERROR is '0', the received frame is sent to the RX FIFO. In SmartCard sub mode, negatively acknowledged data frames generate a parity error. Note that firmware can only identify the erroneous data frame in the RX FIFO if it is fast enough to read the data frame before the hardware writes a next data frame into the RX FIFO.
  - LIN baud rate detection is completed – The receiver software uses the UART_RX_STATUS.BR_COUNTER value to set the clk_scb to guarantee successful receipt of the first LIN data frame (Protected Identifier Field) after the synchronization byte.

– LIN break detection is successful – The line is ‘0’ for UART_RX_CTRL.BREAK_WIDTH + 1 bit period. Can occur at any time to address unanticipated break fields; that is, “break-in-data” is supported. This feature is supported for the UART standard and LIN submodes. For the UART standard submodes, ongoing receipt of data frames is not affected; firmware is expected to take proper action. For the LIN submode, possible ongoing receipt of a data frame is stopped and the (partially) received data frame is dropped and baud rate detection is started. Set to ‘1’, when event is detected. Write with '1' to clear bit.

Figure 28-47 and **Figure 28-48** show how each of the interrupts are triggered. **Figure 28-47** shows the TX buffer and the corresponding interrupts while **Figure 28-48** shows all the corresponding interrupts for the RX buffer. The FIFO has 256 split into 128 bytes for TX and 128 bytes for RX instead of the 8 bytes shown in the figures. For more information on how to implement and clear interrupts see the UART (SCB_UART_PDL) datasheet and the PDL.

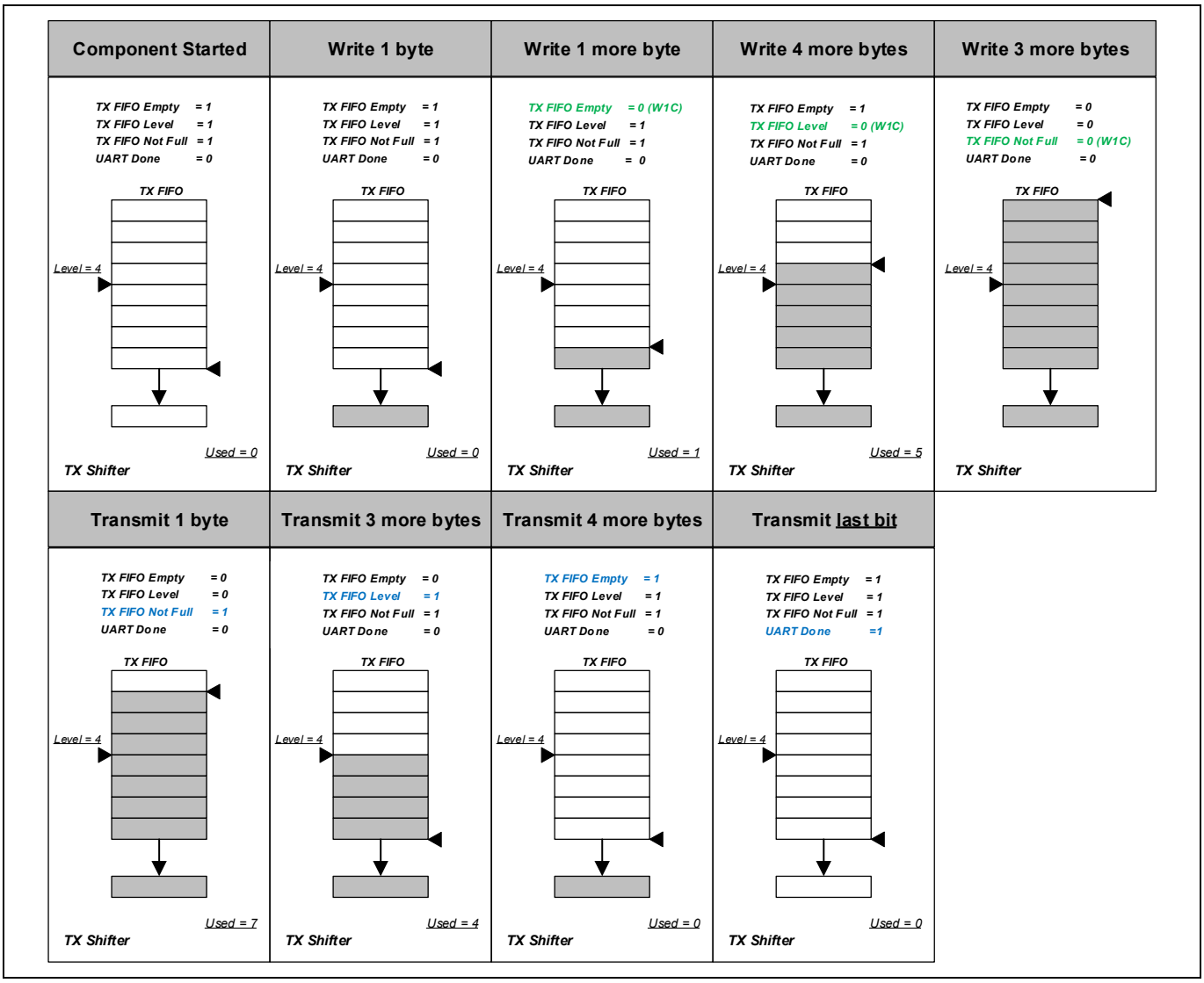


Figure 28-47. TX interrupt source operation

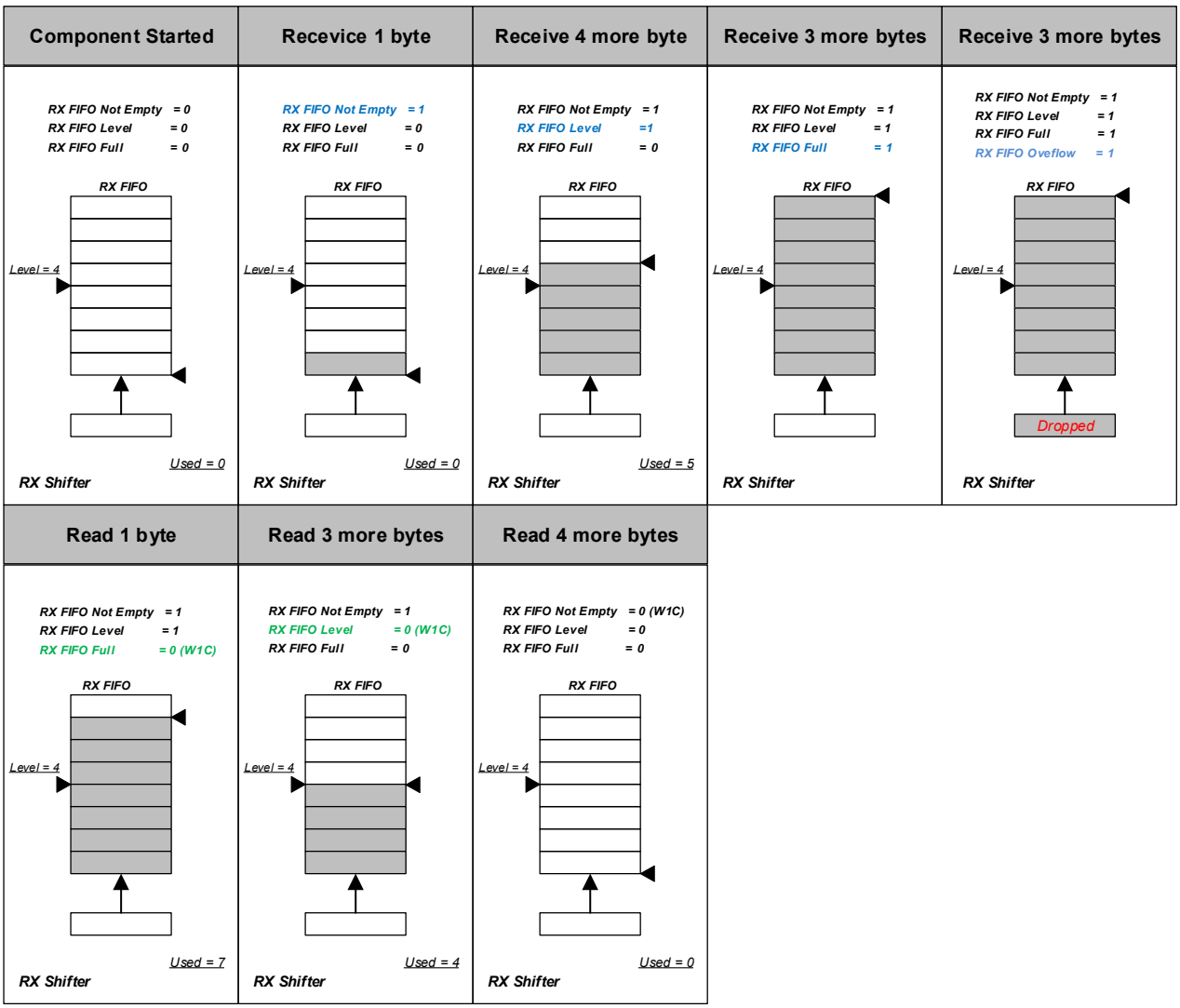


Figure 28-48. RX interrupt source operation

28.6.3 I²C interrupts

I²C interrupts can be classified as master interrupts, slave interrupts, TX interrupts, RX interrupts, and externally clocked (EC) mode interrupts. Each interrupt output is the logical OR of the group of all possible interrupt sources classified under the section. For example, the TX interrupt output is the logical OR of the group of all possible TX interrupt sources. This signal goes high when any of the enabled TX interrupt sources are true. The SCB also provides an interrupt cause register (SCB_ INTR_CAUSE) that can be used to determine interrupt source. The interrupt registers are cleared by writing '1' to the corresponding bitfield. Note that certain interrupt sources are triggered again as long as the condition is met even if the interrupt source was cleared. For example, the TX_FIFO_EMPTY is set as long as the transmit FIFO is empty even if the interrupt source is cleared. For more information on interrupt registers, see the registers reference manual. The I²C block generates interrupts for the following conditions.

- I²C Master
  - I²C master lost arbitration – The value driven by the master on the SDA line is not the same as the value observed on the SDA line
  - I²C master received NACK – When the master receives a NACK (typically after the master transmitted the slave address or TX data)
  - I²C master received ACK – When the master receives an ACK (typically after the master transmitted the slave address or TX data)
  - I²C master sent STOP – When the master has transmitted a STOP
  - I²C bus error – Unexpected stop/start condition is detected
- I²C Slave
  - I²C slave lost arbitration – The value driven on the SDA line is not the same as the value observed on the SDA line (while the SCL line is '1'). This should not occur; it represents erroneous I²C bus behavior. In case of lost arbitration, the I²C slave state machine aborts the ongoing transfer. Software may decide to clear the TX and RX FIFOs in case of this error.
  - I²C slave received NACK – When the slave receives a NACK (typically after the slave transmitted TX data)
  - I²C slave received ACK – When the slave receives an ACK (typically after the slave transmitted TX data)
  - I²C slave received STOP – I²C STOP event for I²C (read or write) transfer intended for this slave (address matching is performed). When STOP or REPEATED START event is detected. The REPEATED START event is included in this interrupt cause such that the I²C transfers separated by a REPEATED START can be distinguished and potentially treated separately by the firmware. Note that the second I²C transfer (after a REPEATED START) may be to a different slave address.
    The event is detected on any I²C transfer intended for this slave. Note that an I²C address intended for the slave (address matches) will result in an I2C_STOP event independent of whether the I²C address is ACK'd or NACK'd.
  - I²C slave received START – When START or REPEATED START event is detected. In the case of an externally-clocked address matching (CTRL.EC_AM_MODE is '1') and clock stretching is performed (until the internally-clocked logic takes over) (I2C_CTRL.S_NOT_READY_ADDR_NACK is '0'), this field is not set. Firmware should use INTR_S_EC.WAKE_UP, INTR_S.I2C_ADDR_MATCH, and INTR_S.I2C_GENERAL.
  - I²C slave address matched – I²C slave matching address received. If CTRL.ADDR_ACCEPT, the received address (including the R/W bit) is available in the RX FIFO. In the case of externally-clocked address matching (CTRL.EC_AM_MODE is '1') and internally-clocked operation (CTRL.EC_OP_MODE is '0'), this field is set when the event is detected.
  - I²C bus error – Unexpected STOP/START condition is detected

- I^2C TX
  - TX trigger – TX FIFO has fewer entries than the value specified by TRIGGER_LEVEL in SCB_TX_FIFO_CTRL
  - TX FIFO not full – At least one data element can be written into the TX FIFO
  - TX FIFO empty – The TX FIFO is empty
  - TX FIFO overflow – Firmware attempts to write to a full TX FIFO
  - TX FIFO underflow – Hardware attempts to read from an empty TX FIFO
- I^2C RX
  - RX FIFO has more entries than the value specified by TRIGGER_LEVEL in SCB_RX_FIFO_CTRL
  - RX FIFO is full – The RX FIFO is full
  - RX FIFO is not empty – At least one data element is available in the RX FIFO to be read
  - RX FIFO overflow – Hardware attempts to write to a full RX FIFO
  - RX FIFO underflow – Firmware attempts to read from an empty RX FIFO
- I^2C externally clocked
  - Wake up request on address match – Active on incoming slave request (with address match). Only set when EC_AM is '1'
  - I^2C STOP detection at the end of each transfer – Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is '1'
  - I^2C STOP detection at the end of a write transfer – Activated at the end of a write transfer (I^2C STOP). This event is an indication that a buffer memory location has been written to. For EZ mode, a transfer that only writes the base address does not activate this event. Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is '1'.
  - I^2C STOP detection at the end of a read transfer – Activated at the end of a read transfer (I^2C STOP). This event is an indication that a buffer memory location has been read from. Only set for a slave request with an address match, in EZ and CMD_RESP modes, when EC_OP is '1'.

For more information on how to implement and clear interrupts see the I^2C (SCB_I2C_PDL) datasheet and the PDL.

29 Serial memory interface (SMIF)

The SMIF block implements a single-SPI, dual-SPI, quad-SPI, or octal-SPI communication to interface with external memory chips. The SMIF block's primary use case is to set up the external memory and have it mapped to the EZ-USB™ FX10 MCU memory space using the hardware. This mode of operation, called the XIP mode, allows the bus masters in the EZ-USB™ FX10 MCU to directly interact with the SMIF for memory access to an external memory location.

29.1 Features

The SMIF block provides a master interface to serial memory devices that supports the following functionality:

- Interfacing up to four memory devices (slaves) at a time
- SPI protocol
  - SPI mode 0: clock polarity (CPOL) and clock phase (CPHA) are both '0'
  - Support for single, dual, quad, and octal SPI protocols
  - Support for dual-quad SPI mode: the use of two quad SPI memory devices to increase data bandwidth for SPI read and write transfers
  - Support for configurable MISO sampling time and programmable receiver clock
- Support for device capacities in the range of 64 KB to 128 MB
- A memory-mapped mode of operation, eXecute-In-Place (XIP), which enables mapping the external memory into an internal memory address
- A command mode (MMIO mode), which enables using the SMIF block as a simple communication hardware
- Supports a 4-KB read cache in memory mapped (XIP) mode
- Supports on-the-fly 128-bit encryption and decryption

29.2 Architecture

Figure 29-1 shows a high-level block diagram of the SMIF hardware in EZ-USB™ FX10 MCUs. Notice that the block is divided into multiple clock domains. This enables multiple domains to access the SMIF and still enable maintaining an asynchronous clock for the communication interface.

The SMIF block can also generate DMA triggers and interrupt signals. This allows events in the SMIF block to trigger actions in other parts of the system.

The SMIF interface is implemented using eight data lines, four slave select lines, and a clock line.

The access to the SMIF block can be by two modes: MMIO mode or XIP mode. The MMIO mode gives access to the SMIF's peripheral registers and the internal FIFOs. This mode is used when the user code is responsible for constructing the command structure for the external memory. Typically, this mode is used when the SMIF writes to an external flash memory. The MMIO interface is also used to configure the SMIF hardware block, including configuring the device registers that set up the XIP operation of the SMIF block.

The XIP mode of operation maps the external memory space to a range of addresses in the EZ-USB™ FX10 MCU's address space. Refer to the registers reference manual for details. When this address range is accessed, the hardware automatically generates the commands required to initiate the associated transfer from the external memory. The typical use case for the XIP mode is to execute code placed in external memory. Here, the memory will be mapped into the internal address space of the EZ-USB™ FX10 MCU using the XIP mode. Thus code is execution from external memory is seamless.

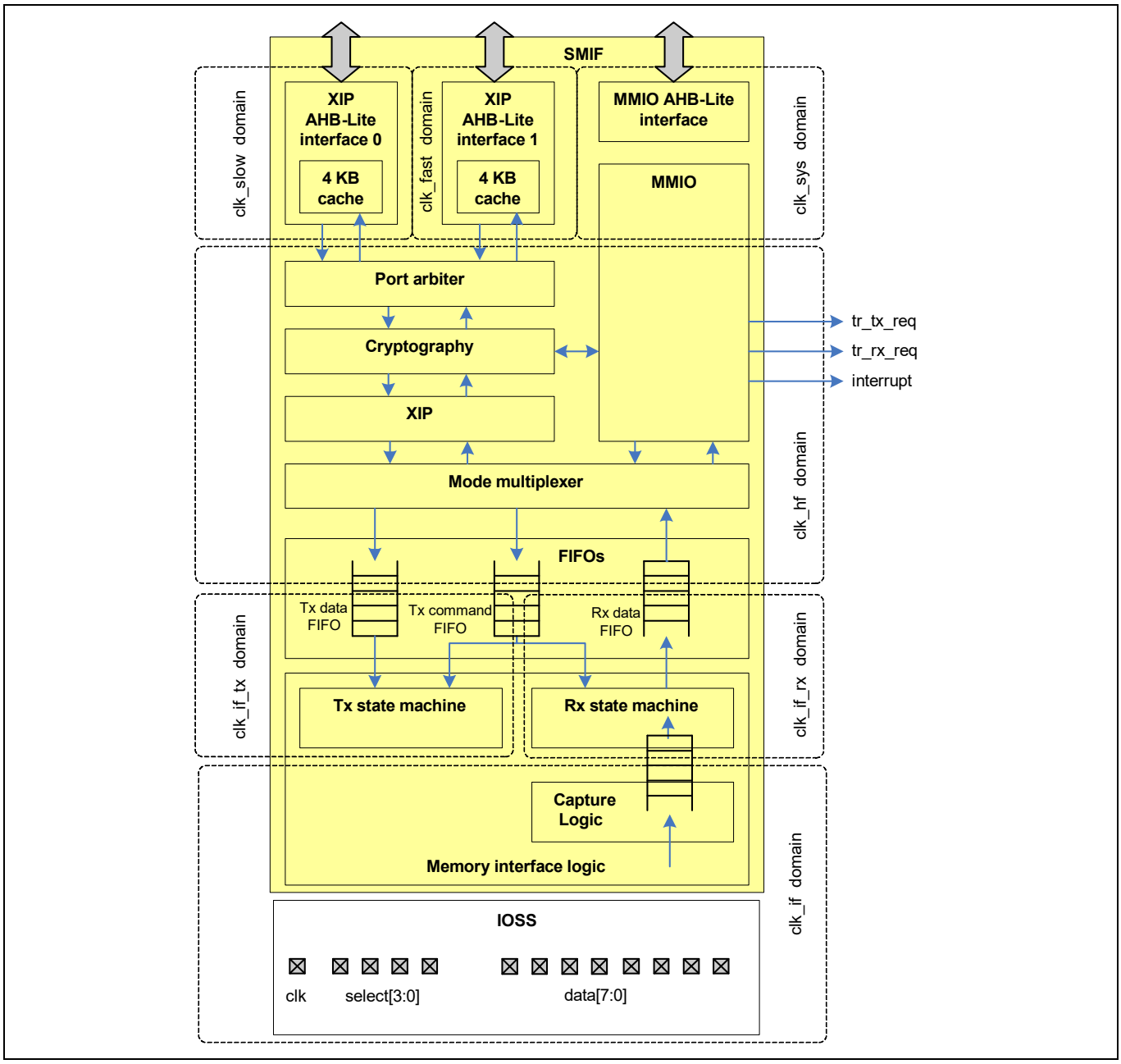


Figure 29-1. SMIF hardware block diagram

The SMIF block has three AHB-Lite interfaces:

- An AHB-Lite interface to access the SMIF's MMIO registers
- Two AHB-Lite interfaces to support eXecute-In-Place (XIP)

All interfaces provide access to external memory devices. At any time, either the MMIO AHB-Lite interface or the two XIP AHB-Lite interfaces have access to the memory interface logic and external memory devices. The operation mode is specified by SMIFn_CTL.XIP_MODE. The operation mode should not be modified when the SMIF is busy (STATUS.BUSY is '1').

In the MMIO AHB-Lite interface, access is supported through software writes to transmit (Tx) FIFOs and software reads from receive (Rx) FIFOs. The FIFOs are mapped on SMIF registers. This interface provides the flexibility to implement any memory device transfer. For example, memory device transfers to setup, program, or erase the external memory devices.

In an XIP AHB-Lite interface, access is supported through XIP: AHB-Lite read and write transfers are automatically (by the hardware) translated in memory device read and write transfers. This interface provides efficient implementation of memory device read and write transfers, but does NOT support other types of memory device transfers. To improve XIP performance, the XIP AHB-Lite interface has a 4-KB read cache.

As mentioned, MMIO mode and XIP mode are mutually exclusive. The operation modes share Tx and Rx FIFOs and memory interface logic. In MMIO mode, the Tx and Rx FIFOs are accessed through the SMIF registers and under software control. In XIP mode, the Tx and Rx FIFOs are under hardware control. The memory interface logic is controlled through the Tx and Rx FIFOs and is agnostic of the operation mode.

29.2.1 TX and RX FIFOs

The SMIF block has two TX FIFOs and one RX FIFO. These FIFOs provide an asynchronous clock domain transfer between clk_hf logic and clk_if_tx/clk_if_rx memory interface logic. The memory interface logic is completely 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

29.2.1.1 TX command FIFO

The TX command FIFO consists of four 20-bit entries. Each entry holds a command. A memory transfer consists of a series of commands. In other words, a command specifies a phase of a memory transfer. Four different types of commands are supported:

- **TX command**. A memory transfer must start with a TX command. The TX command includes a byte that is to be transmitted over the memory interface. The TX command specifies the width of the data transfer (single, dual, quad, or octal data transfer). The TX command specifies whether the command is for the last phase of the memory transfer (explicit “last command” indication). The TX command specifies which of the four external devices are selected (multiple devices can be selected simultaneously); that is, the device selection as encoded by the TX command is used for the complete memory transfer. The TX command asserts the corresponding slave select lines. This is the reason every memory transfer should 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 relies on the TX data FIFO to provide the bytes that are to be transmitted over the memory interface. The TX_COUNT command specifies the width of the data transfer and always constitutes the last phase of the memory transfer (implicit “last command” indication - de-asserts the slave select). Note that the TX_COUNT command does not assert the slave select lines. This must be done by a TX command preceding it.
- **RX_COUNT command**. The RX_COUNT command specifies the number of bytes to be received from the RX data FIFO. This command relies on the RX data FIFO to accept the bytes that are received over the memory interface. The RX_COUNT command specifies the width of the data transfer and always constitutes the last phase of the memory transfer (implicit “last command” indication - de-asserts the slave select). Note that the RX_COUNT command does not assert the slave select lines. This must be done by a TX command preceding it.
- **DUMMY_COUNT command**. The DUMMY_COUNT command specifies a number of dummy cycles. Dummy cycles are used to implement a turn-around (TAR) time in which the memory master changes from a transmitter driving the data lines to a receiver receiving on the same data lines. The DUMMY_COUNT command never constitutes the last phase of the memory transfer (implicit NOT “last command” indication - de-asserts the slave select); that is, it must be followed by another command. Note that the DUMMY COUNT command does not assert the slave select lines. This must be done by a TX command preceding it.

Together, the four command types can be used to construct any SPI transfer. The TX command FIFO is used by both the memory interface transmit and receive logic. This ensures lockstep operation. The TX command is a representation of a queue of commands that are to be processed.

The software will write the sequence of commands into the TX command FIFO to generate a sequence responsible for the communication with 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.

The software can write to the TX command FIFO through 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 '0', an AHB-Lite bus error is generated
- If CTL.BLOCK is '1', the AHB-Lite write transfer is extended until an entry is available. This increases latency.

29.2.1.2 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; that is, specifies the number of TX data FIFO entries used. The TX data FIFO is used 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 through the TX_DATA_FIFO_WR1, TX_DATA_FIFO_WR2, and TX_DATA_FIFO_WR4 registers:

- The TX_DATA_FIFO_WR1 register supports a write of a single byte to the FIFO
- The TX_DATA_FIFO_WR2 register supports a write of two bytes to the FIFO
- The TX_DATA_FIFO_WR4 register supports a write of four bytes to the FIFO. If software attempts to write more bytes than available entries in the TX data FIFO, the MMIO CTL.BLOCK field specifies the behavior:
- If CTL.BLOCK is '0', an AHB-Lite bus error is generated.
- If CTL.BLOCK is '1', the AHB-Lite write transfer is extended until the required entries are available

29.2.1.3 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; that is, specifies the number of RX data FIFO entries used. The memory interface transmit logic will stop generating the SPI clock when the RX data FIFO is full. This is how flow control is achieved.

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 through the MMIO RX_DATA_FIFO_RD1, RX_DATA_FIFO_RD2, and RX_DATA_FIFO_RD4 registers:

- The RX_DATA_FIFO_RD1 register supports a read of a single byte from the FIFO
- The RX_DATA_FIFO_RD2 register supports a read of two bytes from the FIFO
- The RX_DATA_FIFO_RD4 register supports a read of four bytes from the FIFO. If software attempts to read more bytes than available in the Rx data FIFO, the MMIO CTL.BLOCK field specifies the behavior:
- If BLOCK is '0', an AHB-Lite bus error is generated and hard fault occurs.

If BLOCK is '1', the AHB-Lite read transfer is extended until the bytes are available.

Software can also read the first byte of the RX data FIFO without changing the status of the FIFO through the RX_DATA_FIFO_RD1_SILENT register.

29.2.2 MMIO mode

If CTL.XIP_MODE is '0', the SMIF is in MMIO mode. Software generates SPI transfers by accessing the TX FIFOs and RX FIFO. Software writes to the TX FIFOs and reads from the RX FIFO. The TX command FIFO has formatted commands (TX, TX_COUNT, RX_COUNT, and DUMMY_COUNT) that are described in the registers reference manual.

Software should ensure that it generates correct memory transfers and accesses the FIFOs correctly. For example, if a memory transfer is generated to read four bytes from a memory device, software should read the four bytes from the RX data FIFO. Similarly, if a memory transfer is generated to write four bytes to a memory device, software should write the four bytes to the TX command FIFO or TX data FIFO.

Incorrect software behavior can lock up the memory interface. For example, a memory transfer to read 32 bytes from a memory device, without software reading the RX data FIFO will lock up the memory transfer as the memory interface cannot provide more than eight bytes to the RX data FIFO (the RX data FIFO has eight entries). This will prevent any successive memory transfers from taking place. Hence, the software should make sure that it read the FIFOs to avoid congestion. Note that a locked up memory transfer due to TX or RX FIFO states is still compliant to the memory bus protocol (but undesirable): the SPI protocol allows shutting down the interface clock in the middle of a memory transfer.

29.2.3 XIP mode

If CTL.XIP_MODE is '1', the SMIF is in XIP mode. Hardware automatically (without software intervention) generates memory transfers by accessing the TX FIFOs and RX FIFO. Hardware supports only memory read and write transfers. Other functionality such as status reads are not supported. This means operations such as writing into a flash device may not be supported by XIP mode. This is because the writing operation into a flash memory involves not only a write command transfer, but also a status check to verify the status of the operation.

- Hardware generates a memory read transfer for an AHB-Lite read transfer (to be precise: only 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) should be initialized. The device configuration register sets up the following parameters for memory:

- Write enable (WR_EN): Used to disable writes in XIP mode
- Crypto enable (CRYPTO_EN): When enabled, all read access to the memory is decrypted and write access encrypted automatically
- Data Select (DATA_SEL): Selects the data lines to be used (**"Connecting SPI memory devices"** on page 514)
- Base address and size: Sets up the mapped memory space. Any access in this space is converted to the access to the external memory automatically.
- Read and write commands: Used to communicate with the external memory device. These commands are defined by multiple settings.

As different memory devices support different types of memory read and write commands, you must provide the hardware with device specifics, such that it can perform the automatic translations. To this end, each memory device has a set of MMIO registers that specify its memory read and write transfers. This specification includes:

- 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 produces 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 is not in a 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') or 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

29.2.4 Cache

To improve XIP performance, the XIP AHB-Lite interface has a cache. The cache is defined as follows:

- 4 KB capacity
- Read-only cache. Write transfers bypass the cache. A write to an address, which is prefetched in the cache, invalidates the associated cache subsector. If there is a write to a memory in MMIO mode then you must invalidate the cache while 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, enabled by default. Any XIP access can be cached if a cache is enabled. There are separate cache registers for the slow cache (in the clk_slow domain) and 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 "hit" are processed by the cache. Read transfers that "miss" result in a XIP memory read transfer.

If CA_CTL.PREF_EN is '1', prefetching is enabled and if CA_CTL.PREF_EN is '0', prefetching is disabled. If prefetch is enabled, a cache miss results in a 16 B (subsector) refill for the missing data AND a 16 B prefetch for the next sequential data (independent of whether this data is already in the cache or not).

Cache coherency is not supported by the hardware. For example, an XIP interface 0 write to an address in the XIP interface 0 cache invalidates (clears) the associated cache subsector in the XIP interface 0 cache, but not in the XIP interface 1 cache. This means XIP interface 1 cache now has outdated data. The user code can manually invalidate cache by using the SLOW_CA_CMD[INV] or FAST_CA_CMD[INV] register.

Caches should also be invalidated upon mode transitions. For example, in MMIO mode, a write to an address in the cache interface will cause the data in the cache interface to be outdated. The cache should be invalidated when transitioning to XIP mode to ensure that only valid data is used.

29.2.5 Arbitration

The SMIF provides two AHB-Lite slave interfaces to CPUSS (one fast interface and one slow interface). Both interfaces have a cache (as described in **"Cache"** on page 511) and can generate XIP requests to the external memory devices.

An arbitration component (as shown in **Figure 29-1**) arbitrates between the two interfaces. Arbitration is based on the master identifier of the AHB-Lite transfer. The arbitration priority is specified by a system wide priority. Each master identifier has a 2-bit priority level ("0" is the highest priority level and "3" is the lowest priority level). Master identifiers with the same priority level are within the same priority group. Within a priority group, round-robin arbitration is performed.

29.2.6 Deselect Delay

The SMIF supports configuration of deselect delay between transfers. The SMIFn_CTL.DESELECT_DELAY field controls the minimum number of interface cycles to hold the chip select line inactive.

29.2.7 Cryptography

In XIP mode, a cryptography component supports on-the-fly encryption for write data and on-the-fly decryption for read data. The use of on-the-fly cryptography is determined 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 usage scenario for cryptography is: data is encrypted in the external memory devices. Therefore, memory read and write data transfers require decryption and encryption functionality respectively. By storing data encrypted in the external memory devices (nonvolatile devices), leakage of sensitive data is avoided.

Encryption and decryption are based on the AES-128 forward block cipher: advanced encryption standard block cipher with a 128-bit key. KEY[127:0] is a secret (private) key programmed into the CRYPTO_KEY3, …, CRYPTO_KEY0 registers. These registers are software write-only: a software read returns "0". In the SMIF hardware, by applying AES-128 with KEY[127:0] on a plaintext PT[127:0], we get 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 not software accessible. The XIP address is extended with the CRYPTO_INPUT3, …, CRYPTO_INPUT0 registers.

In MMIO mode, the MMIO CRYPTO_INPUT3, …, CRYPTO_INPUT0 registers provide the plaintext PT[]. The resulting ciphertext CT[] is provided through the MMIO CRYPTO_OUTPUT3, …, CRYPTO_OUTPUT0 registers.

Figure 29-2 illustrates the functionality in XIP and MMIO modes.

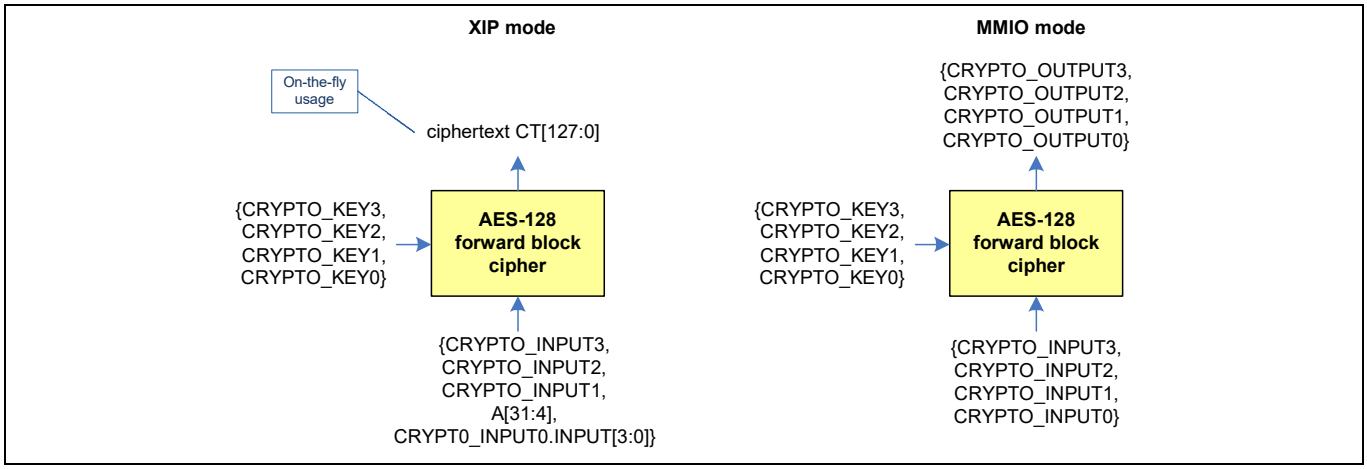


Figure 29-2. Cryptography in XIP and MMIO Modes

In XIP mode, the resulting ciphertext CT[] (of the encrypted address) is XOR'd with the memory transfer's read data or write data. Note that the AES-128 block cipher is on the address of the data and not on the data itself. For memory read transfers, this means that as long as the latency of the memory transfer's read data is longer than the AES-128 block cipher latency, the on-the-fly decryption does not add any delay. **Figure 29-3** illustrates the complete XIP mode functionality.

The XIP mode only encrypts the address and XORs with the data; to implement the same in MMIO, you must provide the address as the PT[] into the crypto_INPUTx registers.

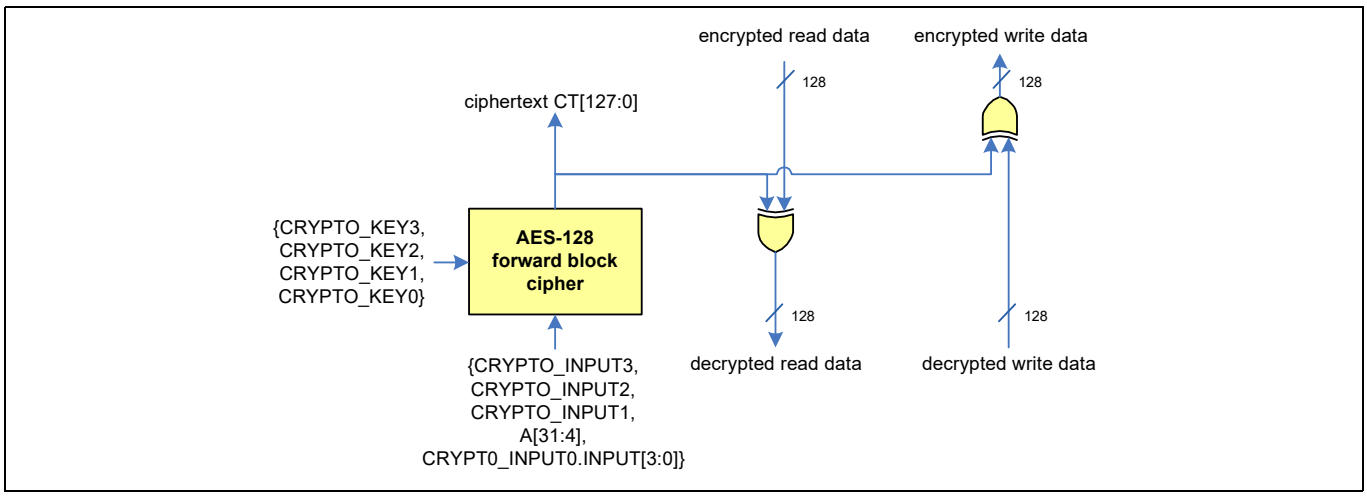


Figure 29-3. XIP Mode Functionality

29.3 Memory device signal interface

The SMIF acts 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': driving of data is on the falling edge of the clock; capturing of data is specified by CTL.CLOCK_IF_RX_SEL

The above configuration is also known as SPI configuration 0 and is supported by SPI memory devices.

29.3.1 Specifying memory devices

The SMIF requires that the memory devices are defined for their operation in XIP mode. The SMIF supports up to four memory devices. Each memory device is defined by a set of registers. The memory device specific register structure includes:

- The device base address and capacity. The ADDR register specifies the memory device's base address in the EZ-USB™ FX10 MCU address space and the MASK register specifies the memory device's size/capacity. If a memory device is not present or is disabled, the ADDR and MASK registers specify a memory device with 0 B capacity. Typically, the devices' address regions in the EZ-USB™ FX10 MCU address space are non-overlapping (except for dual-quad SPI mode) to ensure that the activation of select signals is mutually exclusive.
- The device data signal connections (as described in **"Connecting SPI memory devices"** on page 514)
- The definition of a read transfer to support XIP mode
- The definition of a write transfer to support XIP mode

Each memory device uses a dedicated device select signal: memory device 0 uses select[0], memory device 1 uses select[1], and so on. In other words, there is a fixed, one-to-one connection between memory device, register set, and select signal connection.

In XIP mode, the XIP AHB-Lite bus transfer address is compared with the device region. If the address is within the device region, the device select signal is activated. If an XIP AHB-Lite bus transfer address is within multiple regions (this is possible if the device regions overlap), all associated device select signals are activated. This overlap enables XIP in dual-quad SPI mode: the command, address, and mode bytes can be 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 '1', the transfer address is divided by 2 and the divided by 2 address is provided to the memory devices.

In dual quad SPI mode, each memory device contributes a 4-bit nibble for each 8-bit byte. However, both memory devices are quad SPI memories with a byte interface. Therefore, 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'):

- The transfer address is not a multiple of 2. In this case the divided by 2 address for the memory devices is incorrect.
- The transfer size is not a multiple of 2. In this case, the memory devices contribute only a nibble of a byte. This is not supported as the memory devices have a byte interface.

29.3.2 Connecting SPI memory devices

Memory device I/O signals (SCK, $\overline{CS}$, 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 29-1. Memory device I/O signals

Memory device	IO signals
Single SPI memory	SCK, $\overline{CS}$, SI, SO. This memory device has two data signals (SI and SO).
Dual SPI memory	SCK, $\overline{CS}$, IO0, IO1. This memory device has two data signals (IO0, IO1).
Quad SPI memory	SCK, $\overline{CS}$, IO0, IO1, IO2, IO3. This memory device has four data signals (IO0, IO1, IO2, IO3).
Octal SPI memory	SCK, $\overline{CS}$, IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7. This memory device has eight data signals (IO0, IO1, IO2, IO3, IO4, IO5, IO6, IO7).

Table 29-1 illustrates that each memory has a single clock signal SCK, a single (low active) select signal ($\overline{CS}$), and multiple data signals (IO0, IO1, …).

Each memory device has a fixed select signal connection (to select[3:0]).

Each memory device has programmable data signal connections (to data[7:0]): the CTL.DATA_SEL[1:0] field specifies how a device's data signals are connected. The CTL.DATA_SEL[1:0] is responsible for configuring the selection of data lines to be used by a slave. This is not to be confused with the select lines that are used for addressing the four slaves of the SMIF master. This information is used by the SMIF interface to drive out data on the correct spi_data[] outputs and 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 legal/supported. Supported connections are dependent on the type of memory device.

Table 29-2. Data signal connections

DATA_SEL[1:0]	Single SPI device	Dual SPI device	Quad SPI device	Octal SPI device
0	spi_data[0] = SI spi_data[1] = SO	spi_data[0] = IO0 spi_data[1] = IO1	spi_data[0] = IO0 … spi_data[3] = IO3	spi_data[0] = IO0 … spi_data[7] = IO7
1	spi_data[2] = SI spi_data[3] = SO	spi_data[2] = IO0 spi_data[3] = IO1	Illegal	Illegal
2	spi_data[4] = SI spi_data[5] = SO	spi_data[4] = IO0 spi_data[5] = IO1	spi_data[4] = IO0 … spi_data[7] = IO3	Illegal
3	spi_data[6] = SI spi_data[7] = SO	spi_data[6] = IO0 spi_data[7] = IO1	Illegal	Illegal

Memory devices can:

- Use shared data signal connections
- Use dedicated data signal connections. This reduces the load on the data lines allowing faster signal level changes, which in turn allows for a faster I/O interface.

Note that dual-quad SPI mode requires dedicated data signals to enable read and/or write data transfer from and to two quad SPI devices simultaneously.

Figure 29-4 illustrates memory device 0, which is a single SPI memory with data signals connections to spi_data[1:0].

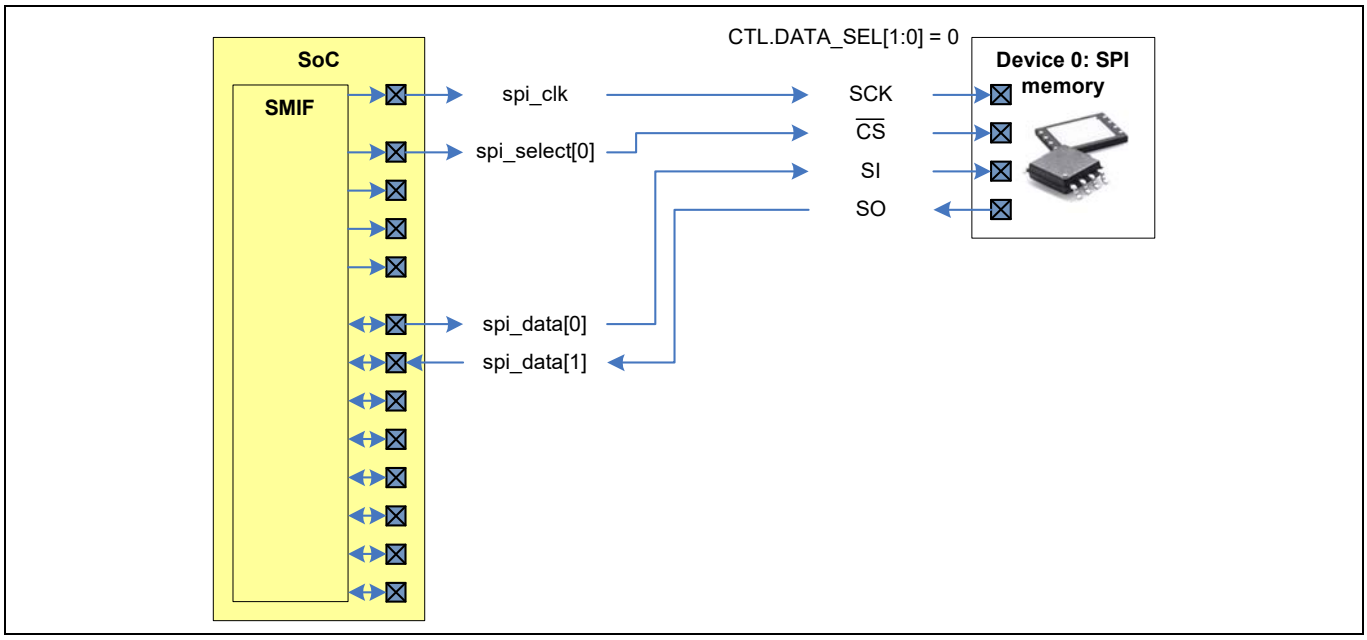


Figure 29-4. Single SPI memory device 0 connected to spi_data[1:0]

Because of the pin layout, you might want to connect a memory device to specific data lines. **Figure 29-5** illustrates memory device 0, which is a single SPI memory with data signals connections to spi_data[7:6].

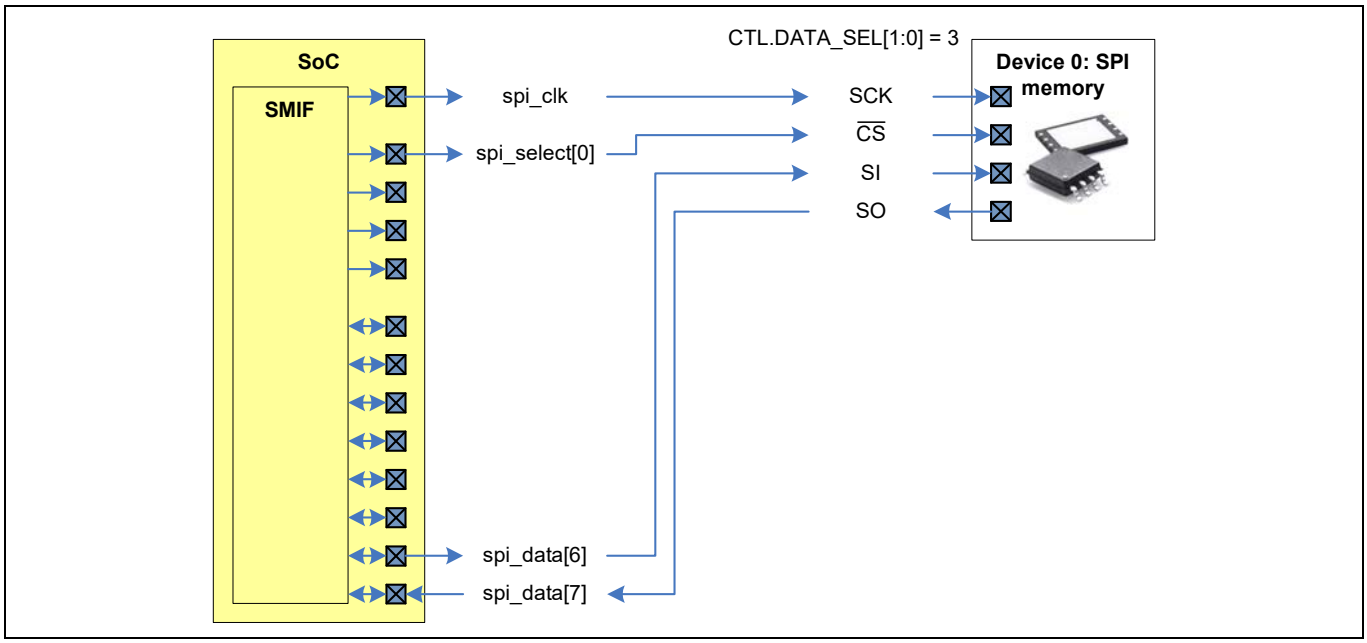


Figure 29-5. Single SPI Memory Device 0 Connected to spi_data[7:6]

Figure 29-6 illustrates memory devices 0 and 1, both of which are single SPI memories. Each device uses dedicated data signal connections. The device address regions in the EZ-USB™ FX10 MCU address space must be non-overlapping to ensure that the activation of select[0] and select[1] are mutually exclusive.

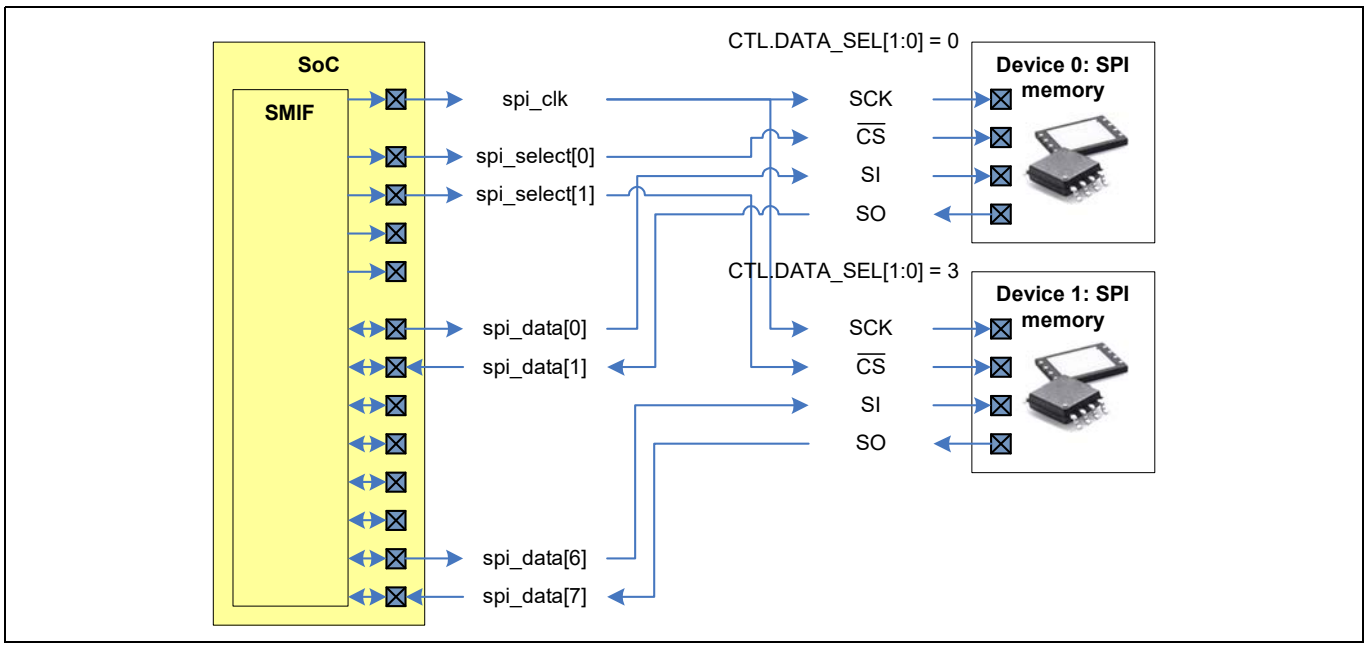


Figure 29-6. Single SPI Memory Devices 0 and 1 - Dedicated data signal

Figure 29-7 illustrates memory devices 0 and 1, both of which are single SPI memories. Both devices use shared data signal connections. The devices' address regions in the EZ-USB™ FX10 MCU address space must be non-overlapping to ensure that the activation of select[0] and select[1] are mutually exclusive. Note that this solution increases the load on the data lines, which may result in a slower I/O interface.

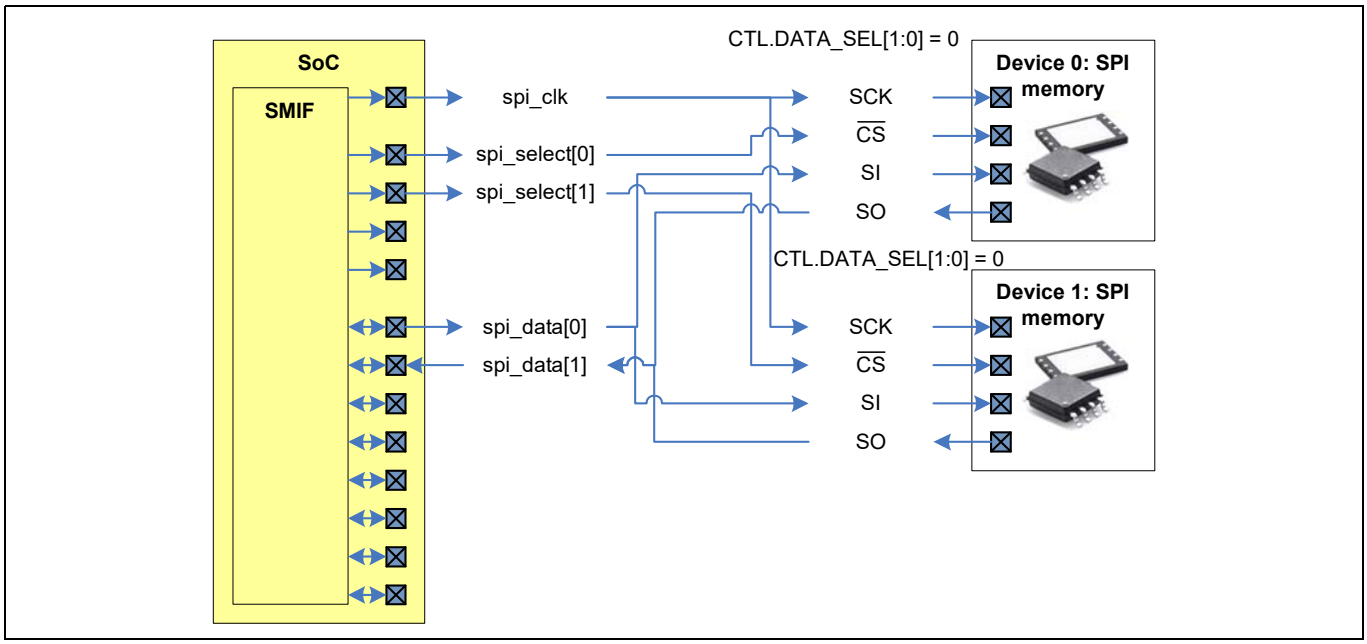


Figure 29-7. Single SPI Memory Devices 0 and 1 - Shared data signal

Figure 29-8 illustrates memory device 0, which is a quad SPI memory with data signals connections to spi_data[7:4].

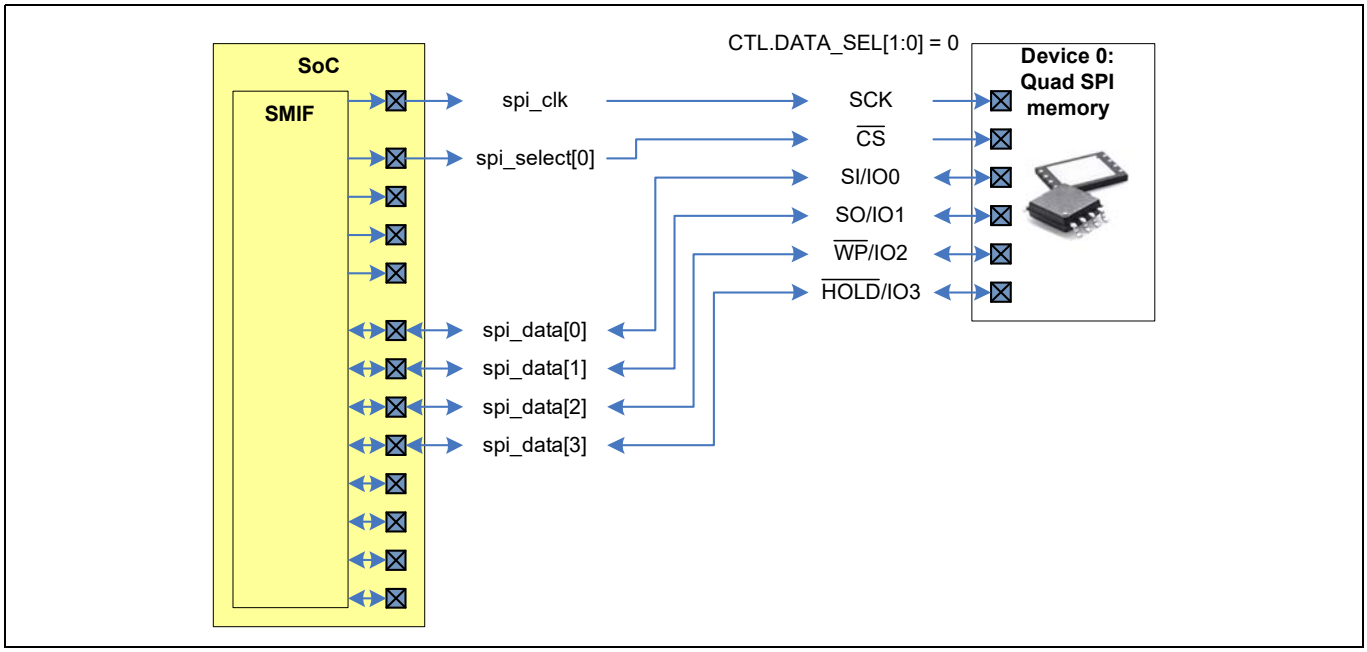


Figure 29-8. Quad SPI Memory Device 0

Serial memory interface (SMIF)

Figure 29-9 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 EZ-USB™ FX10 MCU address space must be non-overlapping to ensure that the activation of select[0] and select[1] are mutually exclusive.

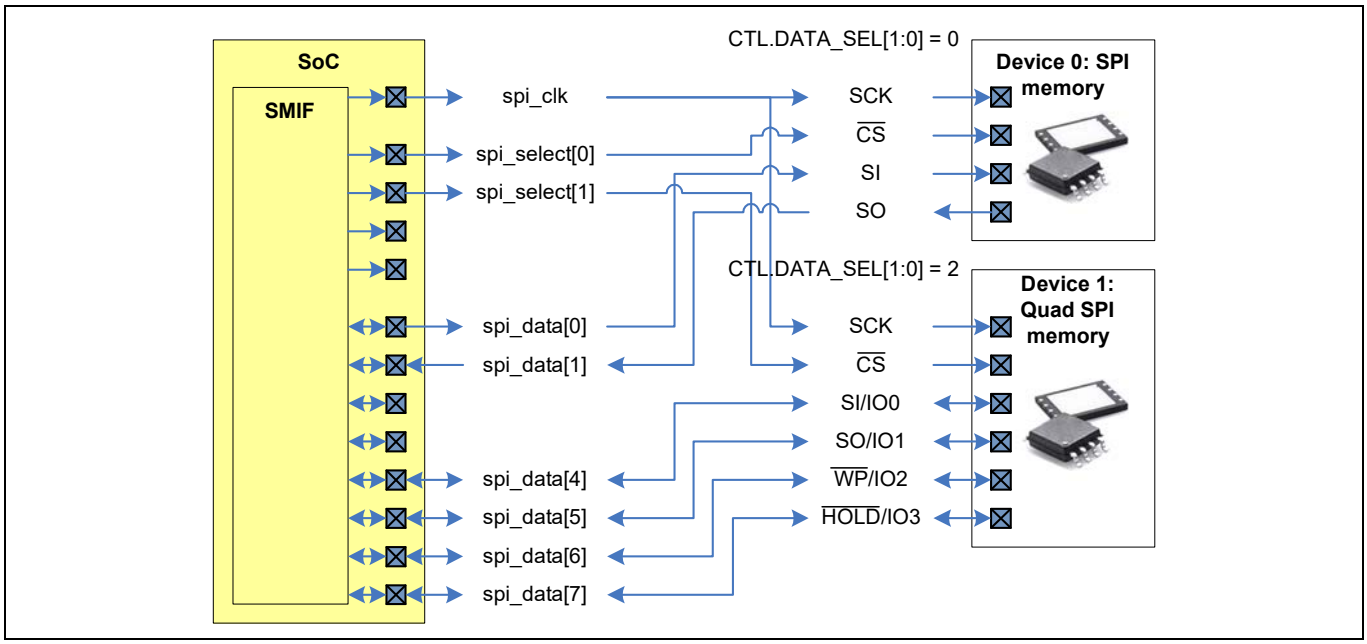


Figure 29-9. Single SPI Memory 0 and Quad SPI Memory 1 - Dedicated data signal

Figure 29-10 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 EZ-USB™ FX10 MCU address space must be non-overlapping to ensure that the activation of select[0] and select[1] are mutually exclusive.

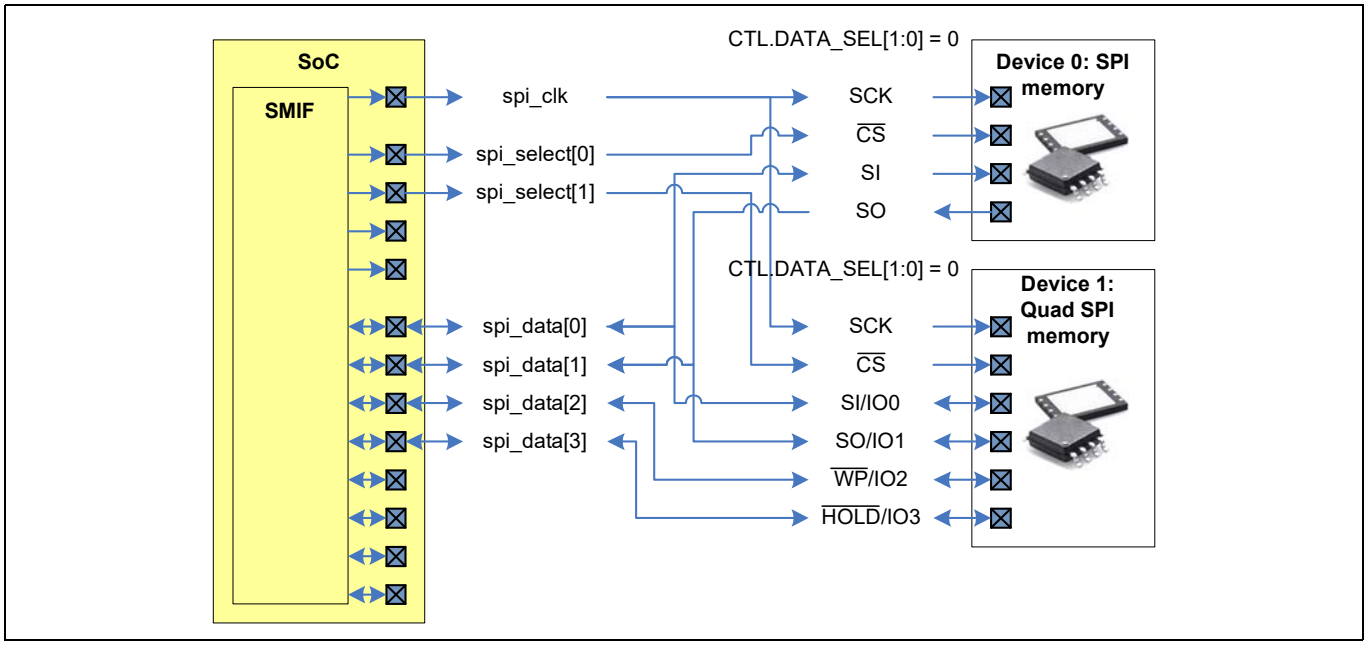


Figure 29-10. Single SPI Memory Device 0 and Quad SPI Memory Device 1 - Shared data signal

Figure 29-11 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 EZ-USB™ FX10 MCU address space are the same to ensure that the activation of select[0] and select[1] are the same (in XIP mode). This is known as a dual-quad configuration: during SPI read and write transfers, each device provides a nibble of a byte.

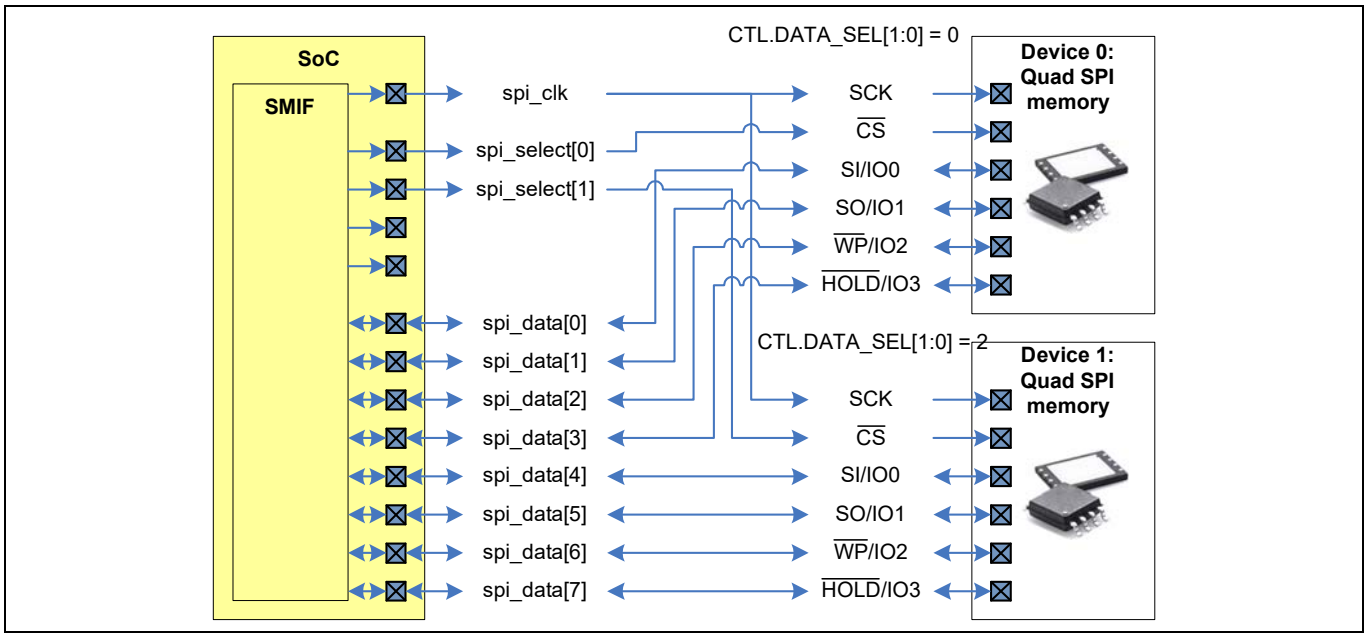


Figure 29-11. Quad SPI Memory Devices 0 and 1

Figure 29-12 illustrates memory device 0, which is a octal SPI memory with data signals connections to spi_data[7:0].

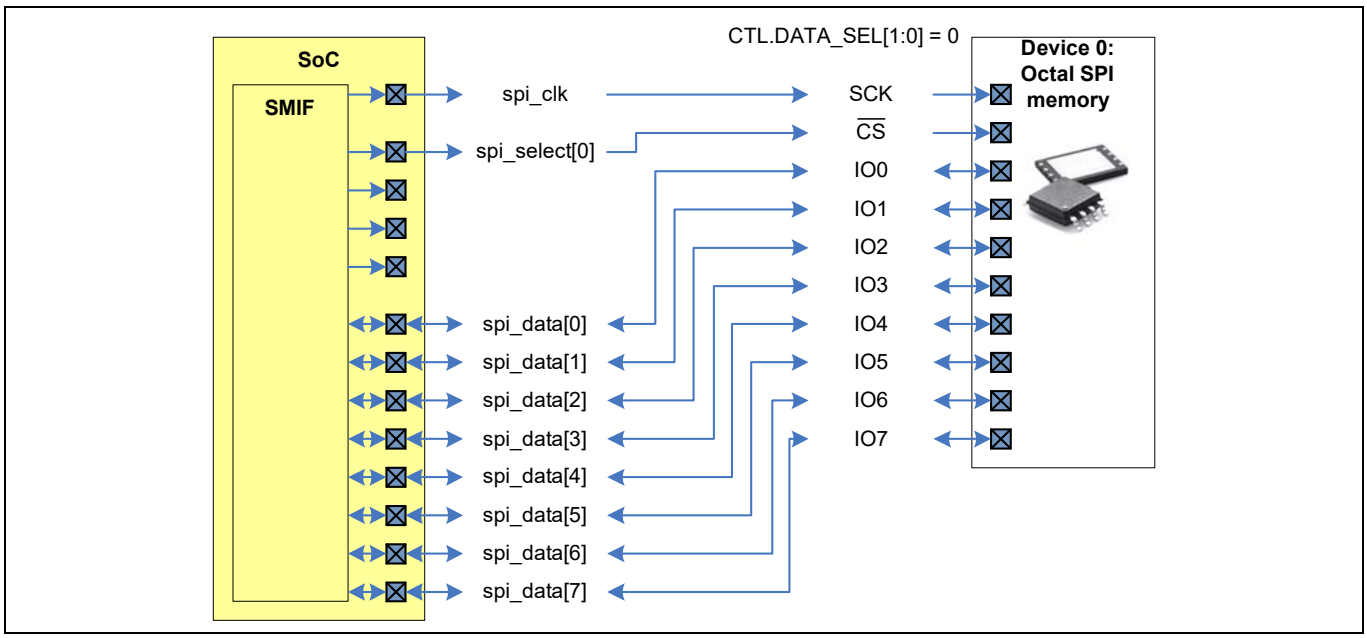


Figure 29-12. Octal SPI Memory Device 0

29.3.3 SPI Data Transfer

SPI data transfer uses most-significant-bit (MSb) for the first data transfer. This means that for a byte B, consisting of bits b7, b6, …, b0, bit b7 is transferred first, followed by bit b6, and so on. For dual, quad, dual quad, and octal SPI transfers, multiple bits are transferred per cycle. For a single SPI device and device data signal connections to spi_data[1:0] (DATA_SEL is “0”), **Table 29-3** summarizes the transfer of byte B.

Table 29-3. Single data transfer

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.

Note that in single SPI data transfer, the data signals are uni-directional: in the table, data[0] is exclusively used for write data connected to the device SI input signal and data[1] is exclusively used for read data connected to the device SO output signal.

For a dual SPI device and device data signal connections to data[1:0] (DATA_SEL is “0”), **Table 29-4** summarizes the transfer of byte B.

Table 29-4. 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 and device data signal connections to data[3:0] (DATA_SEL is “0”), **Table 29-5** summarizes the transfer of byte B.

Table 29-5. 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 a octal SPI device and device data signal connections to data[7:0] (DATA_SEL is “0”), **Table 29-6** summarizes the transfer of byte B.

Table 29-6. 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) should have device data signal connections to data[3:0] (DATA_SEL is 0)
- The second device (the device with the higher device structure index) should have device data signal connection to data[7:4] (DATA_SEL is 2)

The command and data phases of the SPI transfer use different width data transfers:

- The command, address, and mode bytes use quad SPI data transfer.
- The read data and write data use octal data transfer. Each device provides a nibble of each data byte: the first device provides the lower nibble and the second device provides the higher nibble.

Table 29-7 summarizes the transfer of a read data and write data byte B.

Table 29-7. 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.

29.3.4 Example of Setting up SMIF

Devices 0 and 1 are used to implement the dual-quad SPI mode. Both devices are 1 MB / 8 Mb; the address requires 3 bytes. Device 0 has device data signal connections to data[3:0] and device 1 has device data signal connections to data[7:4].

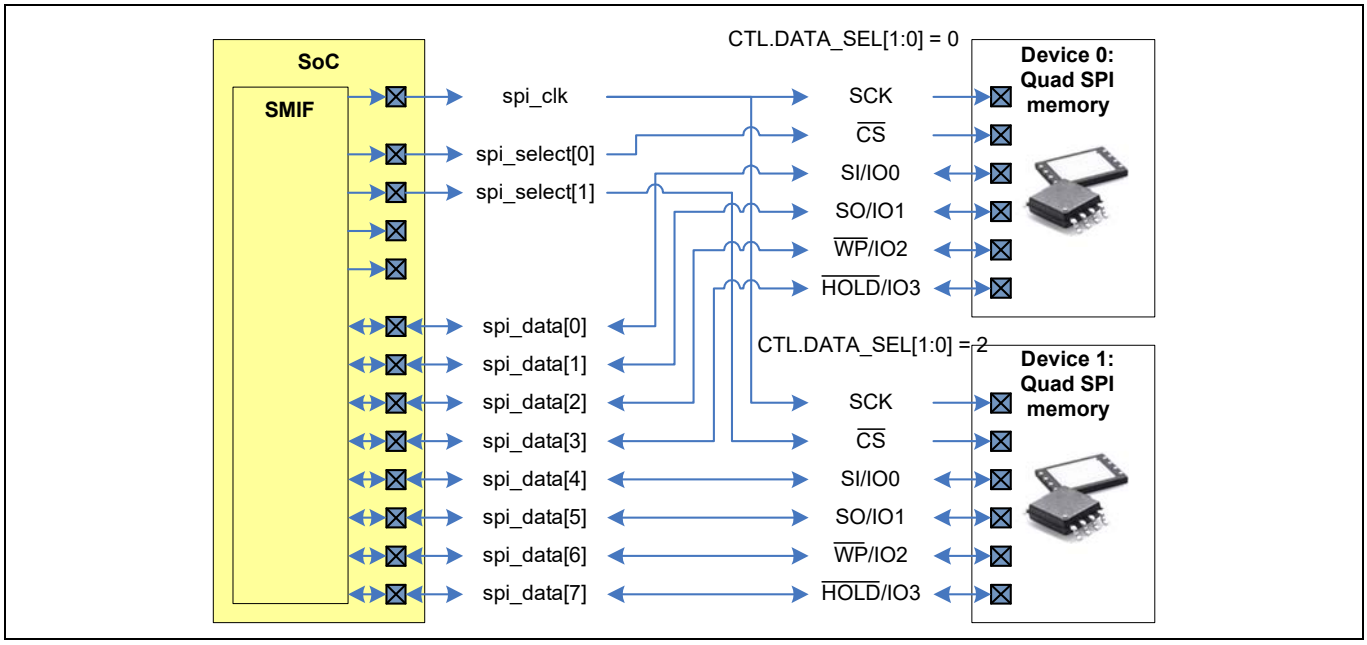


Figure 29-13. Setting up SMIF

For dual quad SPI mode, the AHB-Lite bus transfer address is divided by two. Cryptography and write functionality are disabled in the following example.

```
#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
                | (0 << SMIF_DEVICE_CTL_WR_EN_Pos));       // WR_EN
DEV0_ADDR_CTL =  (1 << SMIF_DEVICE_ADDR_CTL_DIV2_Pos)         // DIV2: enabled
                | ((3-1) << SMIF_DEVICE_ADDR_CTL_SIZE2_Pos));   // SIZE: 3 B 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
                | (1 << SMIF_DEVICE_CTL_WR_EN_Pos));       // WR_EN
DEV1_ADDR_CTL =   (1 << SMIF_DEVICE_ADDR_CTL_DIV2_Pos)         // DIV2: enabled
                | ((3-1) << SMIF_DEVICE_ADDR_CTL_SIZE2_Pos));   // SIZE: 3 B address
```

For XIP read transfers, the 0xEB command/instruction is used (**Figure 29-14** illustrates a two-byte transfer from devices 0 and 1 in dual quad SPI mode).

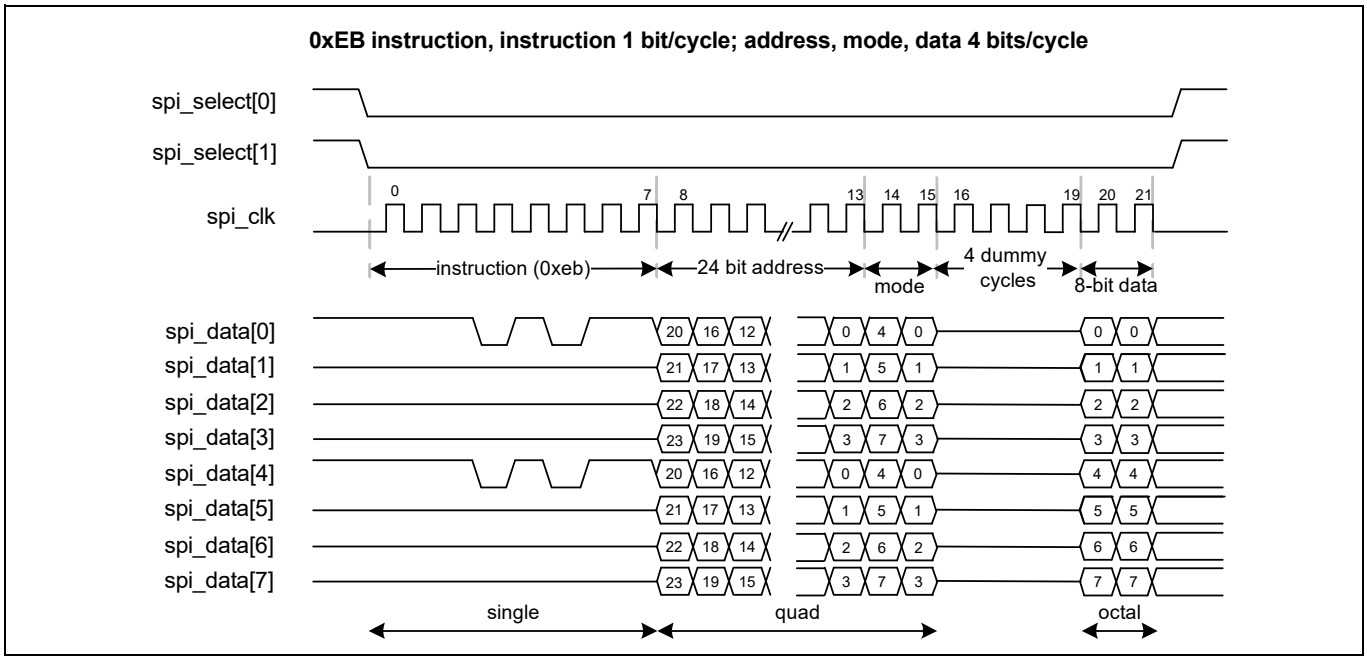


Figure 29-14. Two-bye transfer in Dual Quad SPI mode

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
```

Note that the command uses single data transfer, the address and mode byte use quad data transfer, and the read data byte uses octal data transfer.

29.4 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

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 the XIP operation mode is selected (CTL.XIP_MODE is '1'), the triggers functionality is disabled. The trigger functionality is defined as follows:

- The MMIO TX_DATA_FIFO_CTL.TRIGGER_LEVEL field specifies a number of FIFO entries. The tr_tx_req trigger is active when the number of used Tx data FIFO entries is smaller or equal than the specified number; that is, TX_DATA_FIFO_STATUS.USED ≤ TRIGGER_LEVEL.
- The MMIO RX_DATA_FIFO_CTL.TRIGGER_LEVEL field specifies a number of FIFO entries. The tr_rx_req trigger is active when the number of used Rx data FIFO entries is greater than the specified number; that is, RX_DATA_FIFO_STATUS.USED > TRIGGER_LEVEL.

29.5 Interrupts

The SMIF has a single interrupt output with six interrupt causes:

- INTR.TR_TX_REQ. This interrupt cause is activated in MMIO mode when the tr_tx_req trigger is activated.
- INTR.TR_RX_REQ. This interrupt cause is activated in MMIO mode when the tr_rx_req trigger is activated.
- INTR.XIP_ALIGNMENT_ERROR. This interrupt cause is activated in XIP mode when the selected device's ADDR_CTL.DIV2 field is '1' and the AHB-Lite bus address is not a multiple of 2, or the requested transfer size is not a multiple of 2. This interrupt cause identifies erroneous behavior in dual-quad SPI mode (the selected device ADDR_CTL.DIV2 field is set to '1').
- INTR.TX_CMD_FIFO_OVERFLOW. This interrupt cause is activated in MMIO mode, on an AHB-Lite write transfer to the Tx command FIFO (TX_CMD_FIFO_WR) with insufficient free entries.
- INTR.TX_DATA_FIFO_OVERFLOW. This interrupt cause is activated in MMIO mode, on an AHB-Lite write transfer to the Tx data FIFO (TX_DATA_FIFO_WR1, TX_DATA_FIFO_WR2, and TX_DATA_FIFO_WR4) with insufficient free entries.
- INTR.RX_DATA_FIFO_OVERFLOW. This interrupt cause is activated in MMIO mode, on an AHB-Lite read transfer from the Rx data FIFO (RX_DATA_FIFO_RD1, RX_DATA_FIFO_RD2, and RX_DATA_FIFO_RD4) with insufficient free entries.

29.6 Sleep operation

The SMIF hardware is operational in the Sleep and Active power modes. In the Sleep power mode, only DMA driven transactions can be performed because the CPU acting as the SPI master is not active.

29.7 Performance

Accesses to the external memory will have some latency, which is dependent upon the mode of SMIF operation, the amount of data being transferred, caching, and cryptography. In MMIO mode, the number of interface clock cycles per transfer is determined by the equation:

$$\underset{\textit{(in interface clock cycles)}}{\text{Time to Transfer N bytes}} = \left(\left(\frac{\text{opcode size}}{\text{opcode width}}\right) + \left(\frac{\text{address size}}{\text{address width}}\right) + \left(\frac{\text{mode size}}{\text{mode width}}\right) + \text{dummycycles} + \left(\frac{\text{data size}}{\text{data width}}\right)\right)$$

For example, in **Figure 29-14**, the equation to calculate the number of cycles would be (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, the performance is affected by the cache. For data reads that hit in the cache, the read will not incur any interface cycles. Read operations that miss in the cache will occur as a normal SMIF read operation and, if prefetching is enabled, will result in two 16 B cache sub-sector refills. Writes to external memory in XIP mode will occur as a normal SMIF write operation.

Enabling cryptography may impact SMIF performance. The AES-128 block cipher has a typical latency of 13 clk_hf cycles. This means that for transfers that take more than 13 cycles, the on-the-fly decryption does not add any delay. If the transfer is less than 13 cycles, the transfer latency will be 13 cycles. When the cache is enabled, the 13-cycle latency for encryption is incurred only once for every 16 B fetched for the cache.

30 CAN FD controller

30.1 Overview

The CAN FD controller complies with the ISO11898-1 (CAN specification Rev. 2.0 parts A and B). In addition, it supports the Time-Triggered CAN (TTCAN) protocol defined in ISO 11898-4.

All message handling functions are implemented by the RX and TX handlers. The RX handler manages message acceptance filtering, transfer of received messages from the CAN core to a message RAM, and receive message status information. The TX handler transfers transmit messages from the message RAM to the CAN core and provides transmit status information.

Two separate clocks are provided to the CAN FD controller: CAN clock (clk_can) for CAN operation and system clock (clk_sys) for internal block operation. Acceptance filtering is implemented by a combination of up to 192 filter elements, where each can be configured as a range, as a bit mask, or as a dedicated ID filter.

The CAN FD controller functions only in Active and Sleep power modes. In DeepSleep mode, it is not functional but is fully retained except the Shared Time Stamp (TS) counter.

30.1.1 Features

The CAN FD controller has the following features:

- Flexible data-rate (FD) (ISO 11898-1: 2015)
  - Up to 64 data bytes per message
  - Maximum 8 Mbps supported
- Time-Triggered (TT) communication on CAN (ISO 11898-4: 2004)
  - TTCAN protocol level 1 and level 2 completely in hardware
- AUTOSAR support
- Acceptance filtering
- Two 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 support
- Shared message RAM (MRAM)
- Receive FIFO top pointer logic
  - Enables DMA access on the FIFO
- DMA for debug message and received FIFOs
- Shared time stamp counter

Note: Refer to the device datasheet to identify which device supports M_TTCAN and the total size of the related MRAM.

30.1.2 Features not supported

- Asynchronous serial communication (ASC)
- Interrupt of Bit Error Corrected (IR.BEC) in M_TTCAN
  - This bit is fixed at ‘0’

30.2 Configuration

30.2.1 Block diagram

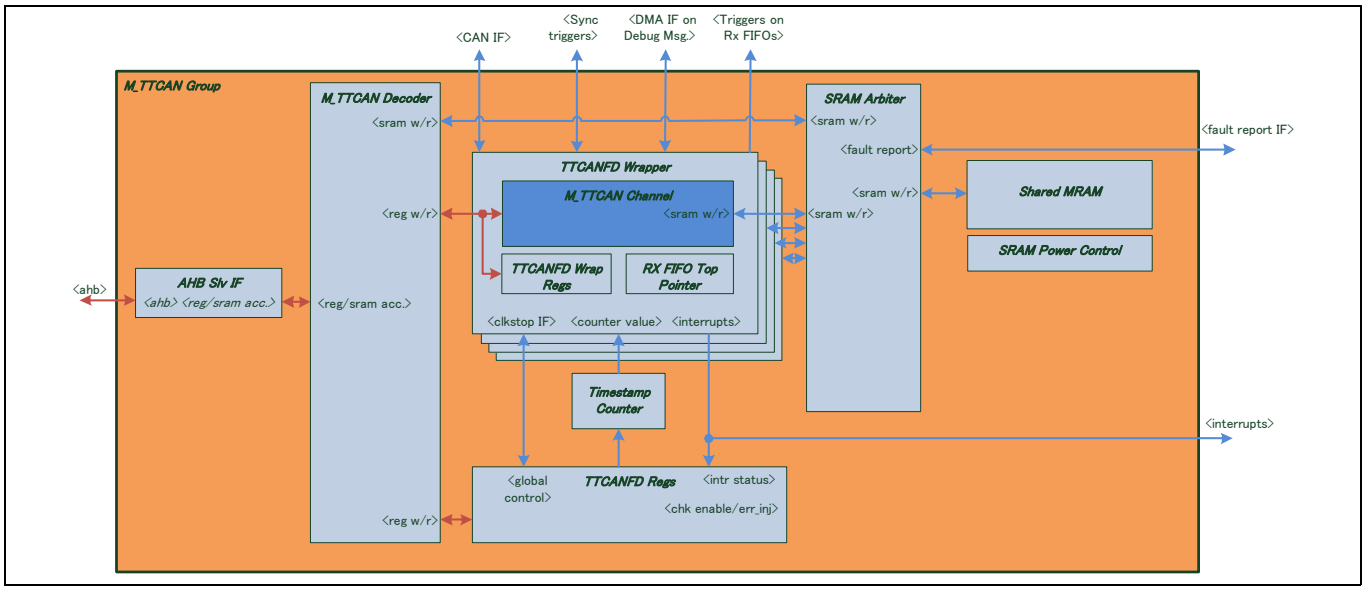


Figure 30-1. M_TTCAN block diagram

30.2.2 Dual clock sources

The M_TTCAN channel has two clock inputs: clk_can and clk_sys. The clk_can, derived from the system peripheral clock dividers, is used for the CAN (or CAN FD) operation. The clk_sys is used for everything except CAN operation; for example, register accesses, SRAM accesses and so on. For the CAN FD operation, it is recommended to use 20 MHz, 40 MHz, or 80 MHz; for clk_can. clk_sys must run equal to or faster than clk_can, and it is allowed to be derived from a spread-spectrum PLL to minimize EMC unlike clk_can.

See the **"Clocking system"** on page 290 for more details about clock configuration.

30.2.3 Interrupt lines

The M_TTCAN channel provides two interrupt lines: interrupt0 and interrupt1. Interrupts from any source within the M_TTCAN channel can be routed either to interrupt0 or interrupt1 by using the ILS and TTILS registers. By default, all interrupts are routed to interrupt0. By programming Enable Interrupt Line 0 (ILE.EINT0) and Enable Interrupt Line 1 (ILE.EINT1), the interrupt lines can be enabled or disabled separately for each interrupt source.

30.3 Functional description

30.3.1 Operation modes

30.3.1.1 Software initialization

This refers to setting or resetting the initialization bit (CCCR.INIT). The CCCR.INIT bit is set

- either by software or hardware reset
- when an uncorrected bit error is detected in MRAM
- by going Bus Off

While the CCCR.INIT is set:

- message transfer from and to the CAN bus is stopped
- the status of the CAN bus output canfd[0].ttcan_tx[0] is recessive (high)
- the protocol error counters are unchanged

Setting CCCR.INIT does not change any configuration register.

Resetting CCCR.INIT finishes the software initialization. The CAN FD controller then synchronizes itself to the data transfer on the CAN bus by waiting for the occurrence of a sequence of 11 consecutive recessive bits (Bus Idle) before it can take part in bus activities and start the message transfer.

30.3.1.1.1 Access/Set/Reset Properties of Registers affected by Configuration Change Enable (CCCR.CCE)

Access to the configuration registers is only enabled when both bits CCCR.INIT and CCCR.CCE are set (write-protected). CCCR.CCE can only be set/reset while CCCR.INIT is 1. CCCR.CCE is automatically reset when CCCR.INIT is reset.

The following registers are reset when CCCR.CCE is set

- HPMS - High Priority Message Status
- RXF0S - RX FIFO 0 Status
- RXF1S - RX FIFO 1 Status
- TXFQS - TX FIFO/Queue Status
- TXBRP - TX Buffer Request Pending
- TXBTO - TX Buffer Transmission Occurred
- TXBCF - TX Buffer Cancellation Finished
- TXEFS - TX Event FIFO Status
- TTOST - TT Operation Status
- TTLGT - TT Local and Global Time, only Global Time TTLGT.GT is reset
- TTCTC - TT Cycle Time and Count
- TTCSM - TT Cycle Sync Mark

In addition

- Timeout Counter value (TOCV.TOC[15:0]) is preset to the value configured by the Timeout Period (TOCC.TOP[15:0]) when CCCR.CCE is set
- State machines of TX and RX handlers are held in idle state while CCCR.CCE is 1

The following registers are only writeable while CCCR.CCE is 0.

- TXBAR - TX Buffer Add Request
- TXBCR - TX Buffer Cancellation Request

Test Mode Enable (CCCR.TEST) and Bus Monitoring mode (CCCR.MON) can only be set by the CPU while CCCR.INIT is 1 and CCCR.CCE is 1. Both bits may be reset at any time.

Disable Automatic Retransmission (CCCR.DAR) can only be set/reset while CCCR.INIT is 1 and CCCR.CCE is 1.

30.3.1.1.2 Message RAM Initialization

Each MRAM word should be reset by writing 0x00000000 before configuration of the CAN FD controller. This prevents MRAM bit errors when reading uninitialized words, and also avoids unexpected filter element configurations in MRAM.

30.3.1.2 Normal operation

The M_TTCAN's default operating mode after hardware reset is event-driven CAN communication without time triggers (TTOCF.OM = 00). Both CCCR.INIT and CCCR.CCE must be set before the TT operation mode is changed.

When M_TTCAN is initialized and CCCR.INIT is reset to zero, M_TTCAN synchronizes itself to the CAN bus and is ready for communication.

After passing the acceptance filtering, received messages including Message ID and DLC are stored into a dedicated RX buffer or into RX FIFO 0 or RX FIFO 1.

For messages to be transmitted, dedicated TX buffers and a TX FIFO/TX queue can be initialized or updated. Automated transmission on reception of remote frames is not implemented.

30.3.1.3 CAN FD operation

The two variants in CAN FD frame transmission are:

- CAN FD frame without bit rate switching
- CAN FD frame where the control, data, and CRC fields are transmitted with a higher bit rate than the beginning and end of the frame

The previously reserved bit in CAN frames with 11-bit identifiers and 29-bit identifiers will now be decoded as FDF bit.

- FDF = Recessive signifies a CAN FD frame
- FDF = Dominant signifies a classic CAN frame

In a CAN FD frame, the two bits following FDF, reserved bits (res), and bit rate switch (BRS) decide whether the bit rate inside the CAN FD frame is switched. A CAN FD bit rate switch signified by res is dominant and BRS is recessive. The coding of res as recessive is reserved for future expansion of the protocol. If the M_TTCAN receives a frame with FDF and res as recessive, it will signal a Protocol Exception Event by setting the PSR.PXE bit. When Protocol Exception Handling is enabled (CCCR.PXHD = 0), it causes the operation state to change from Receiver (PSR.ACT = 10) to Integrating (PSR.ACT = 00) at the next sample point. If Protocol Exception Handling is disabled (CCCR.PXHD = 1), the M_TTCAN will treat a recessive res bit as a form error and respond with an error frame.

CAN FD operation is enabled by programming CCCR.FDOE. If CCCR.FDOE is '1', transmission and reception of CAN FD frames is enabled. Transmission and reception of classic CAN frames is always possible. Whether a CAN FD frame or a classic CAN frame is transmitted can be configured via the FDF bit in the respective TX buffer element. With CCCR.FDOE as '0', received frames are interpreted as classic CAN frames, which leads to the transmission of an error frame when receiving a CAN FD frame. When CAN FD operation is disabled, no CAN FD frames are transmitted even if the FDF bit of a TX buffer element is set. CCCR.FDOE and CCCR.BRSE can only be changed while CCCR.INIT and CCCR.CCE are both set.

With CCCR.FDOE as '0', the setting of FDF and BRS is ignored and frames are transmitted in classic CAN format. When CCCR.FDOE = 1 and CCCR.BRSE = 0, only FDF of a TX buffer element is evaluated. When CCCR.FDOE = 1 and CCCR.BRSE = 1, transmission of CAN FD frames with bit rate switching is enabled. All TX buffer elements with FDF and BRS bits set are transmitted in CAN FD format with bit rate switching.

A mode change during CAN operation is only recommended under the following conditions:

- The failure rate in the CAN FD data phase is significantly higher than in the CAN FD arbitration phase. In this case disable the CAN FD bit rate switching option for transmissions
- During system startup, all nodes transmit classic CAN messages until it is verified that they can communicate in CAN FD format. If this is true, all nodes switch to CAN FD operation
- Wake-up messages in CAN partial networking must be transmitted in classic CAN format
- End-of-line programming occurs in case all nodes are not CAN FD capable. Non-CAN FD nodes are held in Silent mode until programming is completed. Then all nodes switch back to classic CAN communication.

In the CAN FD format, the coding of the DLC differs from the standard CAN format. The DLC codes 0 to 8 have the same coding as in standard CAN, codes 9 to 15, which in standard CAN have a data field of 8 bytes, are coded according to **Table 30-1**.

Table 30-1. 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 will be switched inside the frame after the (BRS) bit, if this bit is recessive. Before the BRS bit, in the CAN FD arbitration phase, the nominal CAN bit timing is used as defined by the Nominal Bit Timing and Prescaler Register (NBTP). In the following CAN FD data phase, the data phase bit timing is used as defined by the Data Bit Timing and Prescaler Register (DBTP). The bit timing is switched back from the data phase timing at the CRC delimiter or when an error is detected, whichever occurs first.

The maximum configurable bit rate in the CAN FD data phase depends on the CAN clock frequency (clk_can). For example, with a CAN clock frequency of 20 MHz and the shortest configurable bit time of 4 tq, the bit rate in the data phase is 5 Mbps.

In both data frame formats, CAN FD and CAN FD with bit rate switching, the value of the bit ESI (Error Status Indicator) is determined by the transmitter's error state at the start of the transmission. If the transmitter is error passive, ESI is transmitted recessive, else it is transmitted dominant.

30.3.1.4 Transmitter delay compensation

During the data phase of a CAN FD transmission only one node is a transmitter; all others are receivers. The length of the bus line has no impact. When transmitting via pin canfd[0].ttcan_tx[0], the M_TTCAN receives the transmitted data from its local CAN transceiver via pin canfd[0].ttcan_rx[0]. The received data is delayed by the transmitter delay. In case this delay is greater than TSEG1 (time segment before sample point), a bit error is detected. To enable a data phase bit time that is even shorter than the transmitter delay, the delay compensation is introduced. Without transmitter delay compensation, the bit rate in the data phase of a CAN FD frame is limited by the transmitter delay.

30.3.1.4.1 Description

The M_TTCAN’s protocol unit has implemented a delay compensation mechanism to compensate the transmitter delay. This enables transmission with higher bit rates during the CAN FD data phase, independent of the delay of a specific CAN transceiver.

To check for bit errors during the data phase of transmitting nodes, the delayed transmit data is compared against the received data at the Secondary Sample Point (SSP). If a bit error is detected, the transmitter will react on this bit error at the next following regular sample point. During the arbitration phase the delay compensation is always disabled.

The transmitter delay compensation enables configurations where the data bit time is shorter than the transmitter delay, it is described in detail in the new ISO 11898-1:2015. It is enabled by setting bit DBTP.TDC.

The received bit is compared against the transmitted bit at the SSP. The SSP position is defined as the sum of the measured delay from the M_TTCAN's transmit output canfd[0].ttcan_tx[0] through the transceiver to the receive input rx plus the transmitter delay compensation offset as configured by TDCR.TDCO. The transmitter delay compensation offset is used to adjust the position of the SSP inside the received bit (for example, half of the bit time in the data phase). The position of the secondary sample point is rounded down to the next integer number of mtq (clk_can period).

PSR.TDCV shows the actual transmitter delay compensation value. PSR.TDCV is cleared when CCCR.INIT is set and is updated at each transmission of an FD frame while 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 canfd[0].ttcan_tx[0] to canfd[0].ttcan_rx[0] and the configured transmitter delay compensation offset TDCR.TDCO must be less than 6 bit times in the data phase.
- The sum of the measured delay from canfd[0].ttcan_tx[0] to canfd[0].ttcan_rx[0] and the configured transmitter delay compensation offset TDCR.TDCO should be less than or equal 127 mtq. In case this sum exceeds 127 mtq, the maximum value of 127 mtq is used for transmitter delay compensation.
- The data phase ends at the sample point of the CRC delimiter that stops checking of receive bits at the SSPs.

30.3.1.4.2 Transmitter Delay Compensation Measurement

If transmitter delay compensation is enabled by programming DBTP.TDC = 1, the measurement is started within each transmitted CAN FD frame at the falling edge of bit FDF to bit res. The measurement is stopped when this edge is seen at the receive input canfd[0].ttcan_rx[0] of the transmitter. The resolution of this measurement is one mtq (minimum time quanta).

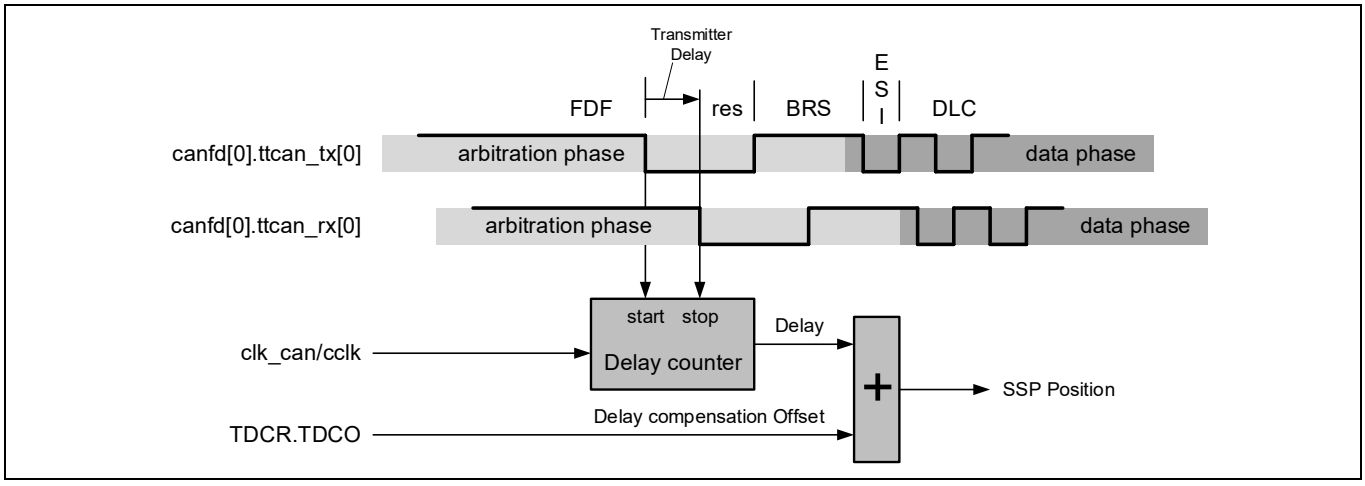


Figure 30-2. Transmitter delay measurement

A dominant glitch inside the received FDF bit ends the delay compensation measurement before the falling edge of the received res bit, resulting in an early SSP position. The use of a transmitter delay compensation filter window can be enabled by programming TDCR.TDCF. This defines a minimum value for the SSP position. Dominant edges on canfd[0].ttcan_rx[0], that results in an earlier SSP position are ignored for transmitter delay measurement. The measurement is stopped when the SSP position is at least TDCR.TDCF and canfd[0].ttcan_rx[0] is low.

30.3.1.5 Restricted Operation mode

In Restricted Operation mode, the node is able to receive data and remote frames and acknowledge valid frames, but it does not send data frames, remote frames, active error frames, or overload frames. In case of an error or overload condition, it does not send dominant bits; instead it waits for the occurrence of a bus idle condition to resynchronize itself to the CAN communication. The error counters (ECR.REC and ECR.TEC) are frozen while Error Logging (ECR.CEL) is active.

The CPU can set the CAN FD controller into Restricted Operation mode by setting the Restricted Operation mode bit (CCCR.ASM). CCCR.ASM can only be set by the CPU when both CCCR.CCE and CCCR.INIT are set to '1'. CCCR.ASM can be reset by the CPU at any time.

The CAN FD controller enters Restricted Operation mode automatically when the TX handler is not able to read data from the MRAM in time. To leave this mode, the CPU should reset CCCR.ASM.

The Restricted Operation mode can be used in applications that adapt themselves to different CAN bit rates. In this case, the application tests different bit rates and leaves the mode after it has received a valid frame.

Note: The Restricted Operation mode must not be combined with the Loop Back mode (internal or external).

30.3.1.6 Bus Monitoring mode

The M_TTCAN is set in Bus Monitoring mode by programming CCCR.MON to '1' or when error level S3 (TTOST.EL = 11) is entered. In Bus Monitoring mode, the M_TTCAN is able to receive valid data frames and valid remote frames, but cannot start a transmission. In this mode, it sends only recessive bits on the CAN bus, if the M_TTCAN is required to send a dominant bit (ACK bit, overload flag, or active error flag), the bit is rerouted internally so that the M_TTCAN monitors this dominant bit, although the CAN bus may remain in recessive state. In Bus Monitoring mode, the TXBRP register is held in reset state.

The Bus Monitoring mode can be used to analyze the traffic on a CAN bus without affecting it by the transmission of dominant bits. **Figure 30-3** shows the connection of signals canfd[0].ttcan_tx[0] and canfd[0].ttcan_rx[0] to the M_TTCAN in Bus Monitoring mode.

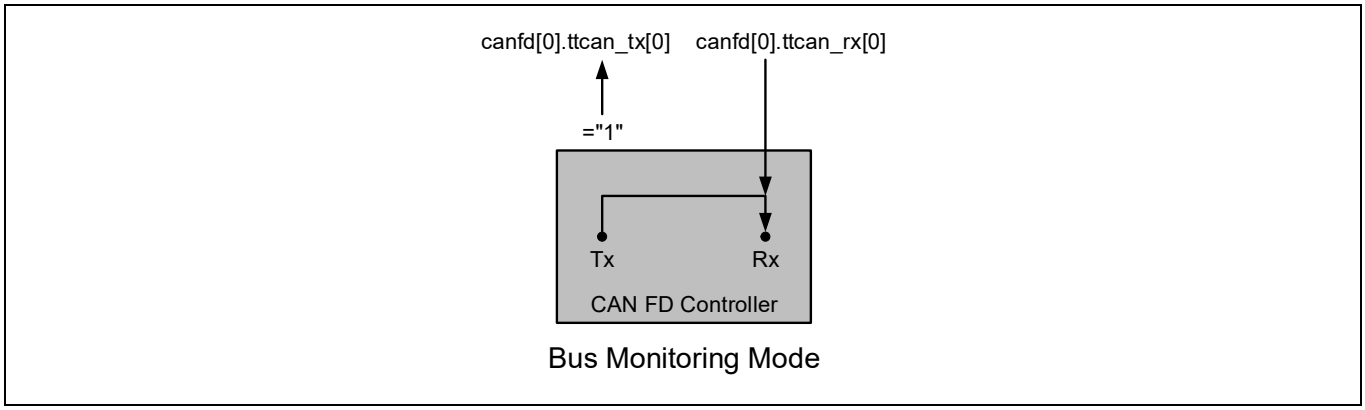


Figure 30-3. Pin control in Bus Monitoring mode

30.3.1.7 Disable automatic Retransmission

M_TTCAN supports automatic retransmission of frames that have lost arbitration or that 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), the automatic retransmission may be disabled via CCCR.DAR.

In DAR mode, all transmissions are automatically canceled after they are started on the CAN bus. The TX Request Pending bit TXBRP.TRPx is reset after successful transmission, when a transmission has not yet started at the point of cancellation, is aborted due to lost arbitration, or when an error occurred during frame transmission.

- Successful transmission:
  Corresponding TX Buffer Transmission Occurred bit TXBTO.TOx set
  Corresponding TX Buffer Cancellation Finished bit TXBCF.CFx not set
- Successful transmission in spite of cancellation:
  Corresponding TX Buffer Transmission Occurred bit TXBTO.TOx set
  Corresponding TX Buffer Cancellation Finished bit TXBCF.CFx set
- Arbitration lost or frame transmission disturbed:
  Corresponding TX Buffer Transmission Occurred bit TXBTO.TOx not set
  Corresponding TX Buffer Cancellation Finished bit TXBCF.CFx 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).

30.3.1.8 Power down (Sleep mode)

The M_TTCAN channel can be set into power down mode via Clock Stop Request (CTL.STOP_REQ). As long as clock stop request is active, STOP_REQ bit is read as one.

When all pending transmission requests have completed, the M_TTCAN waits until bus idle state is detected. Then the M_TTCAN sets CCCR.INIT to one to prevent any further CAN transfers. Now the M_TTCAN acknowledges that it is ready for power down by setting Clock Stop Acknowledge (STATUS.STOP_ACK). Upon receiving ack from channel, hardware automatically switches off the clock to the respective channel.

To leave power down mode, the application must reset CTL.STOP_REQ. The M_TTCAN will acknowledge this by resetting STATUS.STOP_ACK. Afterwards, the application can restart CAN communication by resetting bit CCCR.INIT.

When the clock stop request is triggered through CTL.STOP_REQ, it must not be cleared before STATUS.STOP_ACK bit is set.

Note: *Do not use the TTCAN CCCR.CSR register for the power down control, instead use CTL.STOP_REQ. Similarly, use of CCCR.CSA should be avoided, instead use STATUS.STOP_ACK.*

30.3.1.9 Test mode

To enable write access to TEST register, Test Mode Enable bit (CCCR.TEST) must be set to one. 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] 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] pin can be read from TEST.RX. Both functions can be used to check the CAN bus physical layer.

Due to the synchronization mechanism between CAN clock and host clock domain, there may be a delay of several host clock periods between writing to TEST.TX until the new configuration is visible at output pin canfd[0].ttcan_tx[0]. This applies also when reading input pin canfd[0].ttcan_rx[0] via TEST.RX.

Note: Test modes should be used for production tests or self-test only. The software control for pin canfd[0].ttcan_tx[0] interferes with all CAN protocol functions. It is not recommended to use test modes for application.

30.3.1.9.1 External Loop Back Mode

The M_TTCAN can be set in External Loop Back mode by programming TEST.LBCK to one. In Loop Back 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 30-4** shows the connection of signals canfd[0].ttcan_tx[0] and canfd[0].ttcan_rx[0] to the M_TTCAN in External Loop Back mode.

This mode is provided for hardware self-test. To be independent from external stimulation, the M_TTCAN ignores acknowledge errors (recessive bit sampled in the acknowledge slot of a data/remote frame) in Loop Back mode. In this mode the M_TTCAN performs an internal feedback from its TX output to its RX input. The actual value of the canfd[0].ttcan_rx[0] input pin is disregarded by the M_TTCAN. The transmitted messages can be monitored at the canfd[0].ttcan_tx[0] pin.

30.3.1.9.2 Internal Loop Back Mode

Internal Loop Back mode is entered by programming the TEST.LBCK and CCCR.MON bits to one. This mode can be used for a “Hot Selftest”, meaning the M_TTCAN can be tested without affecting a running CAN system connected to the canfd[0].ttcan_tx[0] and canfd[0].ttcan_rx[0] pins. In this mode, canfd[0].ttcan_rx[0] pin is disconnected from the M_TTCAN and canfd[0].ttcan_tx[0] pin is held recessive. **Figure 30-4** shows the connection of canfd[0].ttcan_tx[0] and canfd[0].ttcan_rx[0] to the M_TTCAN in Internal Loop Back mode.

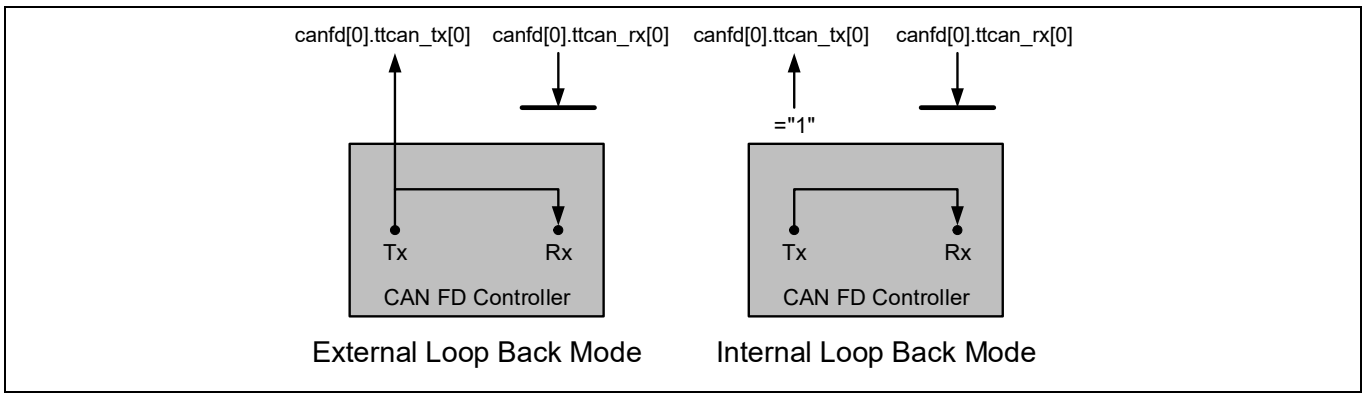


Figure 30-4. Pin control in Loop Back modes

30.3.1.10 Application Watchdog

The application watchdog is served by reading the TTOST register. When the application watchdog is not served in time, TTOST.AWE bit is set, all TTCAN communication is stopped, and the M_TTCAN is set into Bus Monitoring mode.

The TT application watchdog can be disabled by programming the Application Watchdog Limit TTOCF.AWL to 0x00. The TT application watchdog should not be disabled in a TTCAN application program.

30.3.2 Timestamp generation

The M_TTCAN channel uses a 16-bit counter to record when messages are sent or received. This allows the application software to know the order in which events occured.

The timestamp counter is configured through the TS_CTL register. The TS_CTL.ENABLED bit will enable the counter. Upon enabling, it will start incrementing according to the TS_CTL.PRESCALE [15:0]. The application can read the counter value through the TS_CNT register. Write access to the TS_CNT register will clear the TS_CNT.

When the timestamp counter is enabled, internal counter for prescaler counts with every cycle of clk_sys; when the counter value reaches the prescaler value, the timestamp counter increments by one and internal prescaler counter is cleared. When TS_CTL.PRESCALE changes, TS_CNT should be written to reset them. This can make the internal prescaler counter follow a new value of TS_CTL.PRESCALE immediately.

The shared timestamp counter is a wrap-around counter. When the counter wraps around, IR.TSW for the M_TTCAN channel will be raised.

On start of frame reception/transmission, the timestamp counter value is captured and stored into the timestamp section of an RX buffer/RX FIFO (RXTS [15:0]) or TX Event FIFO (TXTS [15:0]) element. **Note:** The counter value TS_CNT is not retained in DeepSleep mode whereas the TS_CTL is retained.

30.3.3 Timeout counter

To signal timeout conditions for RX FIFO 0, RX FIFO 1, and the TX Event FIFO, the M_TTCAN supplies a 16-bit Timeout Counter. It operates as 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…16). The timeout counter is configured via register TOCC. The actual counter value can be read from TOCV.TOC.

The timeout counter can only be started while CCCR.INIT = 0. It is stopped when CCCR.INIT = 1; for example, when the M_TTCAN enters Bus_Off state.

The operation mode is selected by TOCC.TOS. When operating in Continuous mode, the counter starts when CCCR.INIT is reset. A write to TOCV presets the counter to the value configured by TOCC.TOP and continues down-counting.

When the timeout counter is controlled by one of the FIFOs, an empty FIFO presets the counter to the value configured by TOCC.TOP. Down-counting is started when the first FIFO element is stored. Writing to TOCV has no effect.

When the counter reaches zero, 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 time the Timeout Counter is decremented may vary due to the synchronization/resynchronization mechanism of the CAN Core. If the bit rate switch feature in CAN FD is used, the timeout counter is clocked differently in arbitration and data field.

30.3.4 RX handling

The RX handler controls acceptance filtering, transfer of received messages to the RX buffers or to one of the two RX FIFOs, as well as RX FIFO's Put and Get Indices.

30.3.4.1 Acceptance filtering

The M_TTCAN offers the possibility to configure two sets of acceptance filters, one for standard identifiers and one for extended identifiers. These filters can be assigned to an RX buffer or to RX FIFO 0,1. For acceptance filtering each list of filters is executed from element #0 until the first matching element. Acceptance filtering stops at the first matching element. The following filter elements are not evaluated for this message.

The main features are:

- Each filter element can be configured as
  - Range filter (from - to)
  - Filter for one or two dedicated IDs
  - Classic bit mask filter
- Each filter element is configurable for acceptance or rejection filtering
- Each filter element can be enabled/disabled individually
- Filters are checked sequentially; execution stops with the first matching filter element

Related configuration registers are:

- Global Filter Configuration (GFC)
- Standard ID Filter Configuration (SIDFC)
- Extended ID Filter Configuration (XIDFC)
- Extended ID AND Mask (XIDAM)

Depending on the configuration of the filter element (SFEC/EFEC) a match triggers one of the following actions:

- Store received frame in FIFO 0 or FIFO 1
- Store received frame in RX buffer
- Store received frame in RX buffer and generate pulse at filter event pin
- Reject received frame
- Set High-Priority Message interrupt flag IR.HPM
- Set High-Priority Message interrupt flag IR.HPM and store received frame in FIFO 0 or FIFO 1

Acceptance filtering is started after the complete identifier is received. After acceptance filtering has completed, if a matching RX buffer or RX FIFO is found, the message handler starts writing the received message data in portions of 32 bits to the matching RX buffer or RX FIFO. If the CAN protocol controller has detected an error condition (such as CRC error), this message is discarded with the following impact on the affected RX buffer or RX FIFO:

- RX Buffer

New data flag of the matching RX buffer is not set, but RX buffer is (partly) overwritten with received data. For error type, see PSR.LEC and PSR.DLEC, respectively.

- RX FIFO

Put index of matching RX FIFO is not updated, but related RX FIFO element is (partly) overwritten with received data. For error type, see PSR.LEC and PSR.DLEC, respectively. If the matching RX FIFO is operated in overwrite mode, the boundary conditions described in **“RX FIFO Overwrite mode”** on page 539 should be considered.

Note: When an accepted message is written to one of the two RX FIFOs, or into an RX buffer, the unmodified received identifier is stored independent of the filter(s) used. The result of the acceptance filter process depends on the sequence of configured filter elements.

30.3.4.1.1 Range Filter

The filter matches for all received frames with Message IDs in the range defined by SF1ID/SF2ID resp. EF1ID/EF2ID.

The two possibilities when range filtering is used with extended frames are:

- EFT = 00: The Message ID of received frames is ANDed with the XIDAM before the range filter is applied
- EFT = 11: The XIDAM is not used for range filtering

30.3.4.1.2 Filter for specific IDs

A filter element can be configured to filter one or two specific Message IDs. To filter a specific Message ID, the filter element should be configured with SF1ID = SF2ID and EF1ID = EF2ID, respectively.

30.3.4.1.3 Classic Bit Mask Filter

Classic bit mask filtering is intended to filter groups of Message IDs by masking single bits of a received Message ID. With classic bit mask filtering, SF1ID/EF1ID is used as Message ID filter, while SF2ID/EF2ID is used as filter mask.

A zero bit at the filter mask will mask the corresponding bit position of the configured ID filter; for example, the value of the received Message ID at that bit position is not relevant for acceptance filtering. Only those bits of the received Message ID where the corresponding mask bits are one are relevant for acceptance filtering.

In case all mask bits are one, a match occurs only when the received Message ID and the Message ID filter are identical. If all mask bits are zero, all Message IDs match.

30.3.4.1.4 Standard Message ID Filtering

Figure 30-5 shows the flow for standard Message ID (11-bit Identifier) filtering. The Standard Message ID Filter element is described in **"Standard Message ID Filter Element"** on page 559.

Controlled by the GFC and SIDFC Message IDs, the Remote Transmission Request bit (RTR) and the Identifier Extension bit (IDE) of received frames are compared against the list of configured filter elements.

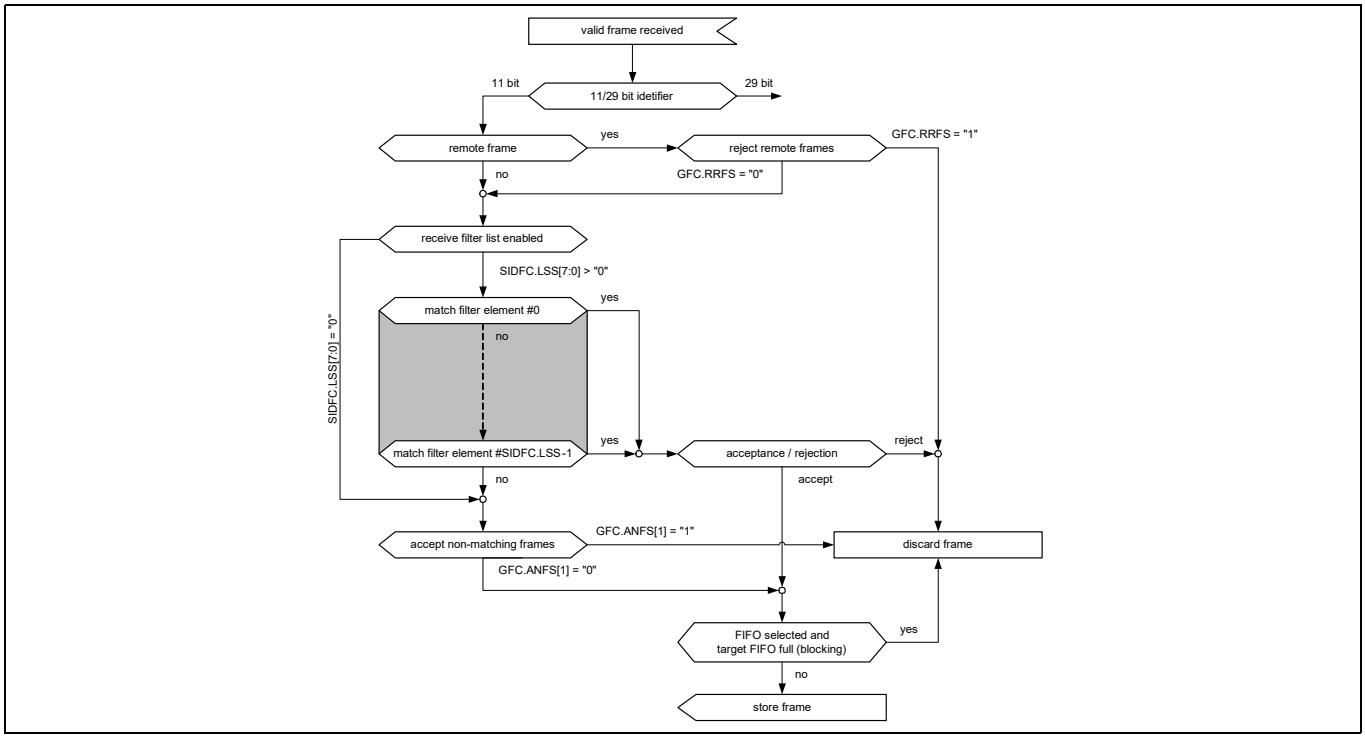


Figure 30-5. Standard Message ID filter

30.3.4.1.5 Extended Message ID Filtering

Figure 30-6 shows the flow for extended Message ID (29-bit Identifier) filtering. The Extended Message ID Filter element is described in **"Extended Message ID filter element"** on page 561.

Controlled by the GFC and XIDFC Message IDs, the RTR bit, and IDE bit of received frames are compared against the list of configured filter elements.

The Extended ID AND Mask XIDAM[28:0] is ANDed with the received identifier before the filter list is executed.

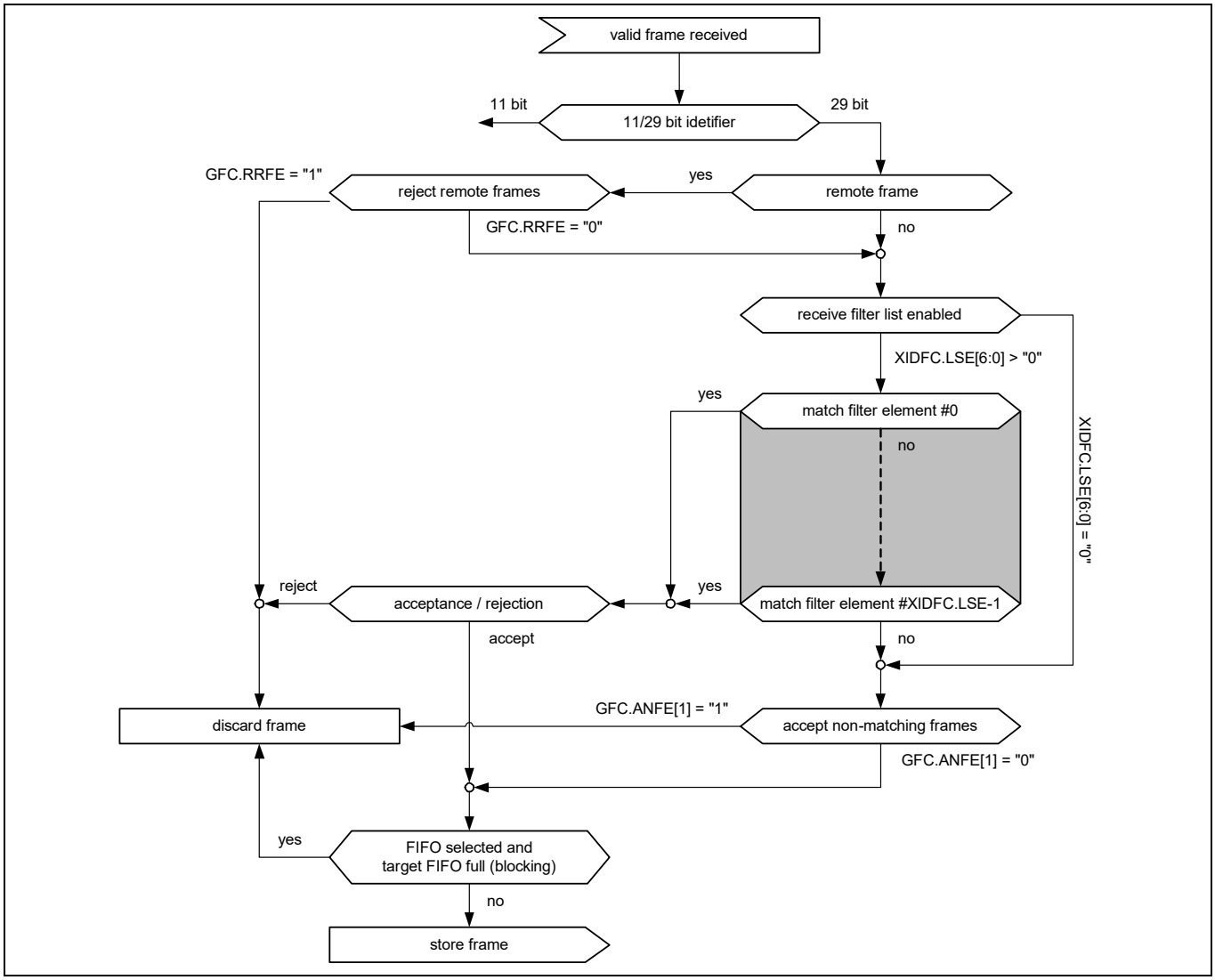


Figure 30-6. Extended Message ID filter path

30.3.4.2 RX FIFOs

RX FIFO 0 and RX FIFO 1 can be configured to hold up to 64 elements each. The two RX FIFOs are configured via the RXF0C and RXF1C registers.

Received messages that pass acceptance filtering are transferred to the RX FIFO as configured by the matching filter element. For a description of the filter mechanisms available for RX FIFO 0 and RX FIFO 1, see **"Acceptance filtering"** on page 535. The RX FIFO element is described in **"RX buffer and FIFO element"** on page 551.

To avoid an RX FIFO overflow, the RX FIFO watermark can be used. When the RX FIFO fill level reaches the RX FIFO watermark configured by RXFnC.FnWM, interrupt flag IR.RFnW is set. When the RX FIFO Put Index reaches the RX FIFO Get Index, an RX FIFO Full condition is signaled by RXFnS.FnF. In addition, interrupt flag IR.RFnF is set. The FIFO watermark interrupt flags can be used to trigger the DMA. DMA request for FIFO will remain set until the respective trigger is cleared by software. Software can clear the trigger by clearing the watermark flag.

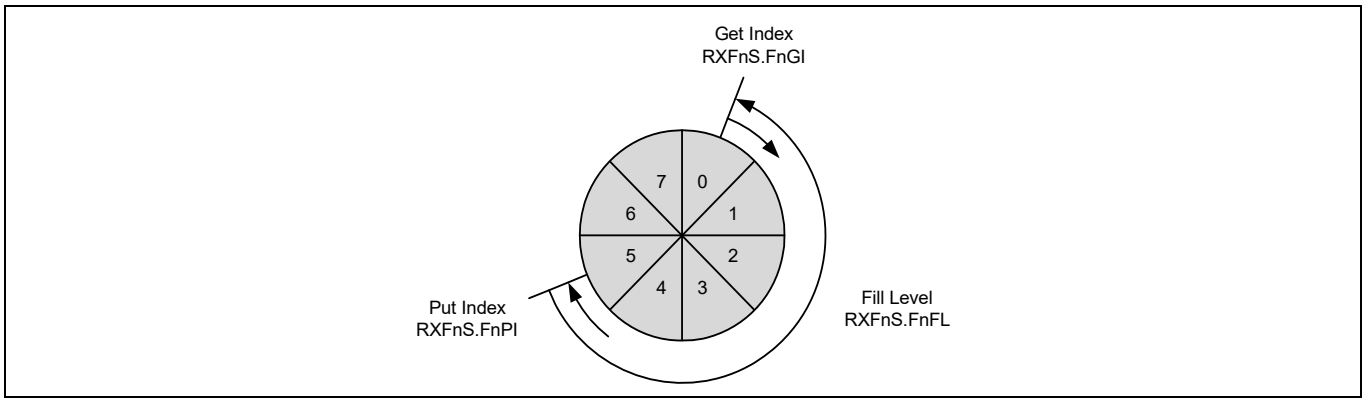


Figure 30-7. RX FIFO status

When reading from an RX FIFO, RX FIFO Get Index RXFnS.FnGI x FIFO Element Size has to be added to the corresponding RX FIFO start address RXFnC.FnSA. RX FIFO Top pointer logic is added to the CAN FD controller to make reading faster. See **"RX FIFO top pointer"** on page 540.

Table 30-2. 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

30.3.4.2.1 RX FIFO Blocking mode

The RX FIFO blocking mode is configured by RXFnC.FnOM = 0. This is the default operation mode for RX FIFOs.

When an RX FIFO full condition is reached (RXFnS.FnPI = RXFnS.FnGI), no further messages are written to the corresponding RX FIFO until at least one message is read and the RX FIFO Get Index is incremented. An RX FIFO full condition is signaled by RXFnS.FnF = 1. In addition, the interrupt flag IR.RFnF is set.

If a message is received while the corresponding RX FIFO is full, this message is discarded and the message lost condition is signaled by RXFnS.RFnL = 1. In addition, the interrupt flag IR.RFnL is set.

30.3.4.2.2 RX FIFO Overwrite mode

The RX FIFO overwrite mode is configured by RXFnC.FnOM = 1.

When an RX FIFO full condition (RXFnS.FnPI = RXFnS.FnGI) is signaled by RXFnS.FnF = 1, the next message accepted for the FIFO will overwrite the oldest FIFO message. Put and Get indices are both incremented by one.

When an RX FIFO is operated in overwrite mode and an RX FIFO full condition is signaled, reading of the RX FIFO elements should start at least at Get Index + 1. This is because a received message may be written to the MRAM (Put Index) while the CPU is reading from the MRAM (Get Index). In this case, inconsistent data may be read from the respective RX FIFO element. Adding an offset to the Get Index when reading from the RX FIFO avoids this problem. The offset depends on how fast the CPU accesses the RX FIFO. **Figure 30-8** shows an offset of two with respect to the Get Index when reading the RX FIFO. In this case, the two messages stored in element 1 and 2 are lost.

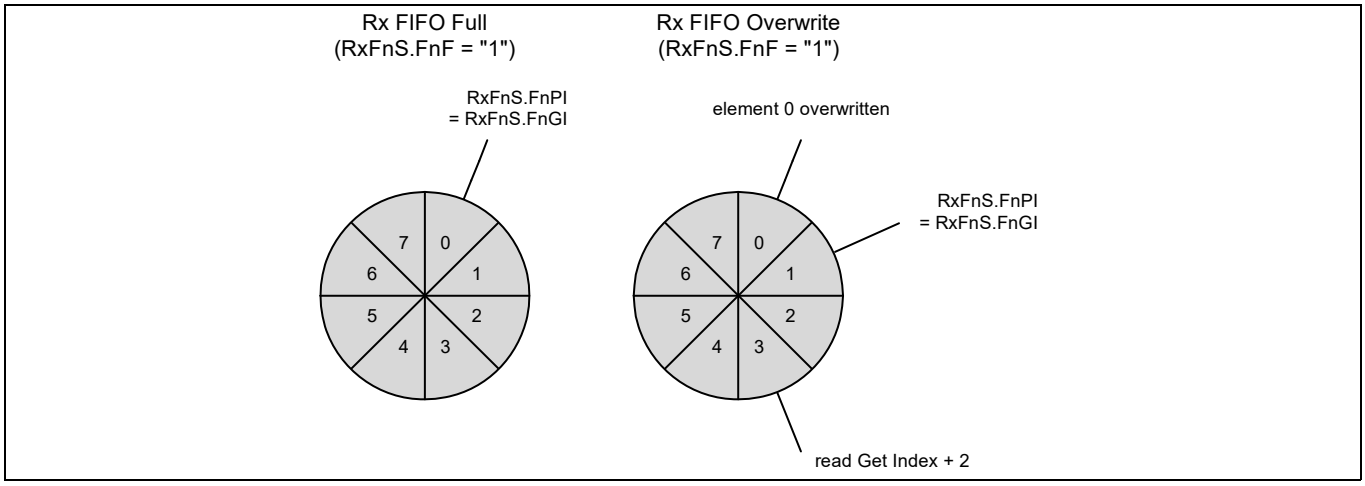


Figure 30-8. RX FIFO overflow handling

After reading from the RX FIFO, the number of the last element read must be written to the RX FIFO Acknowledge Index RXFnA.FnA. This increments the Get Index to that element number. If the Put Index is not incremented to this RX FIFO element, the RX FIFO full condition is reset (RXFnS.FnF = 0).

30.3.4.2.3 RX FIFO top pointer

M_TTCAN supports two receive FIFOs. Reading from these FIFOs requires application to go through following steps:

- Retrieve read pointer
- Calculate correct MRAM address
- Read the data from MRAM
- Update the read pointer

To avoid all these steps, RX FIFO Top Pointer logic has been integrated in the CAN FD controller. It provides a single MMIO location (RXFTOPn_DATA; n = 0,1) to read the data from. Using such hardware logic has the following benefits:

- Higher performance data access
- Less bus traffic
- Reduced CPU load
- Reduced power
- Enables DMA access to FIFO

This logic is enabled when RXFTOP_CTL.FnTPE is set. Setting this bit enables the logic to set the FIFO top address (FnTA) and internal message word counter. Receive FIFO in the top status register (RXFTOPn_STAT) shows the respective FIFO top address and RXFTOPn_DATA provides the data located at the top address. Refer to register definitions for more details on both registers.

If RXFTOPn_DATA is read, the top pointer logic also updates the RX FIFO Acknowledge Index (RXFnA.FnA) in TTCAN channel.

Note: *Top pointer logic is disabled when the channel is being configured (CCCR.CCE = 1). Reading RXFTOPn_DATA while the logic is disabled will return the invalid data.*

Note: *When RX FIFO size is set to 1, do not use the RX FIFO top pointer function. Instead, read the received messages from MRAM directly. When RX FIFO size is set to 2 or more, there are no limitations.*

30.3.4.3 Dedicated RX Buffers

The M_TTCAN supports up to 64 dedicated RX buffers. The start address of the dedicated RX buffer section is configured via RXBC.RBSA.

For each RX buffer, a Standard or Extended Message ID Filter Element with SFEC/EFEC = 111 and SFID2/EFID2[10:9] = 00 must be configured (see **“Standard Message ID Filter Element”** on page 559 and **“Extended Message ID filter element”** on page 561).

After a received message is accepted by a filter element, the message is stored into the RX buffer in the message RAM referenced by the filter element. The format is the same as for an RX FIFO element. In addition, the flag IR.DRX (message stored in a dedicated RX buffer) in the interrupt register is set.

Table 30-3. 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 NDAT1 and 2 registers is set. As long as the New Data flag is set, the respective RX buffer is locked against updates from received matching frames. The New Data flags should be reset by the host by writing a '1' to the respective bit position.

While an RX buffer's New Data flag is set, a Message ID Filter Element referencing the specific RX buffer will not match, causing the acceptance filtering to continue. The following Message ID Filter Elements may cause the received message to be stored into another RX buffer, or into an RX FIFO, or the message may be rejected, depending on filter configuration.

- Reset interrupt flag IR.DRX
- Read New Data registers
- Read messages from message RAM
- Reset New Data flags of processed messages

30.3.4.4 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 is the same for RX buffer and RX FIFO elements.

To filter debug messages Standard/Extended Filter Elements with SFEC/EFEC = 111 should be set up. Messages that match these filter elements are stored into the RX buffers addressed by SFID2/EFID2[5:0].

After message C is stored, the DMA request is activated and the three messages can be read from the message RAM under DMA control. The RAM words holding the debug messages will not be changed by the M_TTCAN while DMA request is activated. The behavior is similar to that of an RX buffer with its New Data flag set.

After the DMA transfer is completed, an acknowledge from DMA resets the DMA request. Now the M_TTCAN is prepared to receive the next set of debug messages.

30.3.4.4.1 Filtering debug messages

Debug messages are filtered by configuring one Standard/Extended Message ID filter element for each of the three debug messages. To enable a filter element to filter debug messages, SFEC/EFEC should be programmed to “111”. In this case the SFID1/SFID2 and EFID1/EFID2 fields have a different meaning (see **“Standard Message ID Filter Element”** on page 559 and **“Extended Message ID filter element”** on page 561). While SFID2/EFID2[10:9] controls the debug message handling state machine, SFID2/EFID2[5:0] controls the storage location of a received debug message.

When a debug message is stored, neither the respective New Data flag nor IR.DRX are set. The reception of debug messages can be monitored via RXF1S.DMS.

Table 30-4. 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

30.3.4.4.2 Debug message handling

The debug message handling state machine assures that debug messages are stored to three consecutive RX buffers in the correct order. If there are missing messages, the process is restarted. The DMA request is activated only when all three debug messages A, B, and C are received in correct order.

The status of the debug message handling state machine is signaled via RXF1S.DMS.

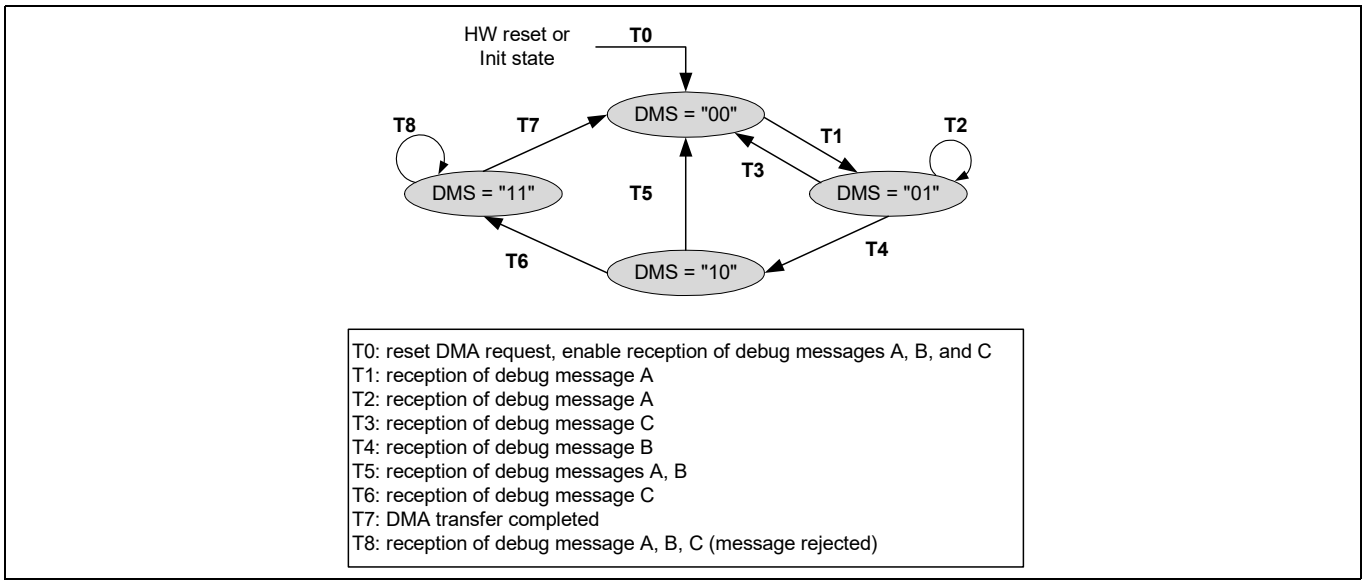


Figure 30-9. Debug message handling state machine

30.3.5 TX handling

The TX handler handles transmission requests for the dedicated TX buffers, TX FIFO, and TX Queue. It controls the transfer of transmit messages to the CAN Core, the Put and Get Indices, and the TX Event FIFO. Up to 32 TX buffers can be set up for message transmission. The CAN mode for transmission (Classic CAN or CAN FD) can be configured separately for each TX buffer element. The TX buffer element is described in **"TX buffer element"** on page 554. **Table 30-5** describes the possible configurations for frame transmission.

Table 30-5. 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
1	1	1	0	FD without bit rate switching
1	1	1	1	FD with bit rate switching

The TX handler starts a TX scan to check for the highest priority pending TX request (TX buffer with lowest Message ID) when the TX Buffer Request Pending register (TXBRP) is updated, or when a transmission is started.

30.3.5.1 Transmit pause

The transmit pause feature is intended for use in CAN systems where the CAN message identifiers are (permanently) assigned to specific values and cannot be changed easily. These message identifiers may have a higher CAN arbitration priority than other defined messages, while in a specific application their relative arbitration priority should be inverse. This may lead to a case where one Electronic Control Unit (ECU) sends a burst of CAN messages that cause another ECU's CAN messages to be delayed because the other messages have a lower CAN arbitration priority.

For example, if CAN ECU-1 has the transmit pause feature enabled and is requested by the application software to transmit four messages, it will, after the first successful message transmission, wait for two nominal bit times of bus idle before it is allowed to start the next requested message. If there are other ECUs with pending messages, those messages are started in the idle time, they will not need to arbitrate with the next message of ECU-1. After having received a message, ECU-1 is allowed to start its next transmission as soon as the received message releases the CAN bus.

The transmit pause feature is controlled by the Transmit Pause bit (CCCR.TXP). If the bit is set, the M_TTCAN controller will, each time it has successfully transmitted a message, pause for two nominal bit times before starting the next transmission. This enables other CAN nodes in the network to transmit messages even if their messages have lower prior identifiers. Default is transmit pause disabled (CCCR.TXP = 0).

This feature loses burst transmissions coming from a single node and protects against “babbling idiot” scenarios where the application program erroneously requests too many transmissions.

30.3.5.2 Dedicated TX buffers

Dedicated TX buffers are intended for message transmission under complete control of the CPU. Each dedicated TX buffer is configured with a specific Message ID. If multiple TX buffers are configured with the same Message ID, the TX buffer with the lowest buffer number is transmitted first.

If the data section is updated, a transmission is requested by an Add Request via TXBAR.ARn. The requested messages arbitrate internally with messages from an optional TX FIFO or TX Queue and externally with messages on the CAN bus, and are sent out according to their Message ID.

30.3.5.2.1 Addressing Dedicated TX Buffers

A dedicated TX buffer allocates an Element Size of 32-bit words in the message RAM as shown in the **Table 30-6**. Therefore, the start address of a dedicated TX buffer in the message RAM is calculated by:

(transmit buffer index (0 to 31) × Element Size) + TX Buffers Start Address (TXBC.TBSA[15:2]).

Table 30-6. 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

30.3.5.3 TX FIFO

TX FIFO operation is configured by programming TXBC.TFQM to '0'. Messages stored in the TX FIFO are transmitted starting with the message referenced by the Get Index TXFQS.TFGI. After each transmission, the Get Index is incremented cyclically until the TX FIFO is empty. The TX FIFO enables transmission of messages with the same Message ID from different TX buffers in the order these messages are written to the TX FIFO. The M_TTCAN calculates the TX FIFO Free Level TXFQS.TFFL as difference between Get and Put Index. It indicates the number of available (free) TX FIFO elements.

New transmit messages must be written to the TX FIFO starting with the TX buffer referenced by the Put Index TXFQS.TFQPI. An Add Request increments the Put Index to the next free TX FIFO element. When the Put Index reaches the Get Index, TX FIFO Full (TXFQS.TFQF = 1) is signaled. In this case no further messages should be written to the TX FIFO until the next message is transmitted and the Get Index is incremented.

When a single message is added to the TX FIFO, the transmission is requested by writing a '1' to the TXBAR bit related to the TX buffer referenced by the TX FIFO's Put Index.

When multiple (n) messages are added to the TX FIFO, they are written to n consecutive TX buffers starting with the Put Index. The transmissions are then requested via TXBAR. The Put Index is then cyclically incremented by n. The number of requested TX buffers should not exceed the number of free TX buffers as indicated by the TX FIFO Free Level.

When a transmission request for the TX buffer referenced by the Get Index is canceled, the Get Index is incremented to the next TX buffer with pending transmission request and the TX FIFO Free Level is recalculated. When transmission cancellation is applied to any other TX buffer, the Get Index and the FIFO Free Level remain unchanged.

A TX FIFO element allocates Element Size 32-bit words in the message RAM as shown in **Table 30-6**. Therefore, the start address of the next available (free) TX FIFO buffer is calculated by adding TX FIFO/Queue Put Index TXFQS.TFQPI (0…31) × Element Size to the TX buffer Start Address TXBC.TBSA.

30.3.5.4 TX queue

TX Queue operation is configured by programming TXBC.TFQM to '1'. Messages stored in the TX Queue are transmitted starting with the message with the lowest Message ID (highest priority). If multiple queue buffers are configured with the same Message ID, the queue buffer with the lowest buffer number is transmitted first.

New messages must be written to the TX buffer referenced by the Put Index TXFQS.TFQPI. An Add Request cyclically increments the Put Index to the next free TX buffer. If the TX Queue is full (TXFQS.TFQF = 1), the Put Index is not valid and no further message should be written to the TX Queue until at least one of the requested messages is sent out or a pending transmission request is canceled.

The application may use the TXBRP register instead of the Put Index and may place messages to any TX buffer without pending transmission request.

A TX Queue buffer allocates element size of 32-bit words in the message RAM as shown in **Table 30-6**. Therefore, the start address of the next available (free) TX Queue buffer is calculated by adding TX FIFO/Queue Put Index TXFQS.TFQPI (0…31) × Element Size to the TX buffer Start Address TXBC.TBSA.

30.3.5.5 Mixed dedicated TX buffers/TX FIFO

In this case, the TX Buffers section in the MRAM is subdivided into a set of dedicated TX buffers and a TX FIFO. The number of dedicated TX buffers is configured by TXBC.NDTB. The number of TX buffers assigned to the TX FIFO is configured by TXBC.TFQS. If TXBC.TFQS is programmed to zero, only the dedicated TX buffers are used.

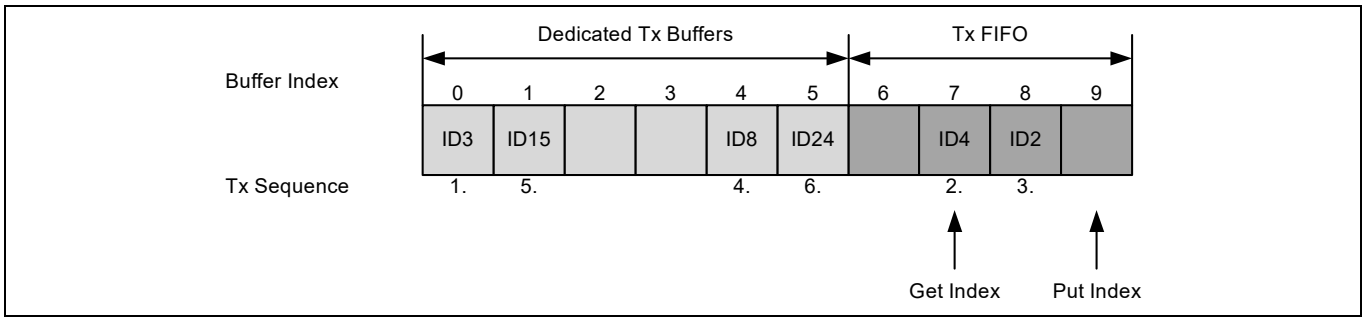


Figure 30-10. Example of Mixed Configuration Dedicated TX Buffers/TX FIFO

TX prioritization:

- Scan dedicated TX buffers and oldest pending TX FIFO buffer (referenced by TXFS.TFGI)
- Buffer with the lowest Message ID gets highest priority and is transmitted next

30.3.5.6 Mixed dedicated TX buffers/TX queue

In this case, the TX Buffers section in the MRAM is subdivided into a set of dedicated TX buffers and a TX Queue. The number of dedicated TX buffers is configured by TXBC.NDTB. The number of TX Queue buffers is configured by TXBC.TFQS. In case TXBC.TFQS is programmed to zero, only dedicated TX buffers are used.

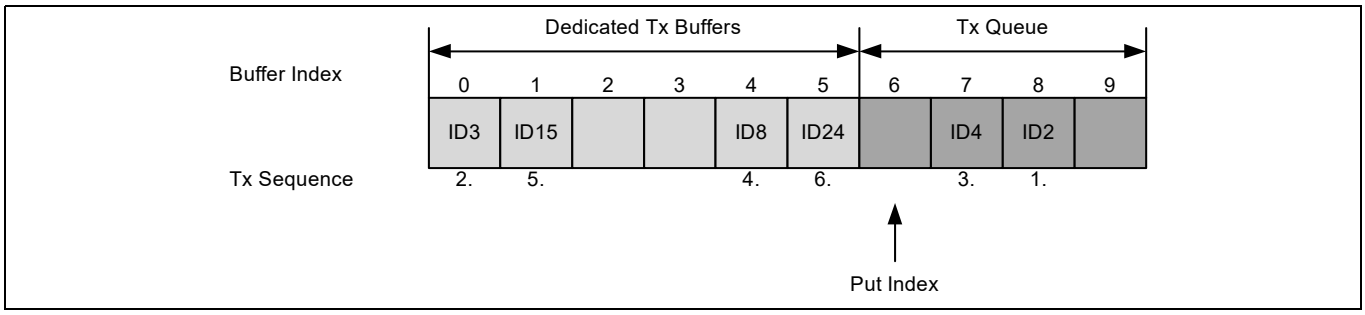


Figure 30-11. Example of mixed configuration dedicated TX buffers/TX queue

TX prioritization:

- Scan all TX buffers with activated transmission request
- TX buffer with the lowest Message ID gets highest priority and is transmitted next

30.3.5.7 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) of 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 case a transmit cancellation is requested while a 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, there follows a short time window where no transmission is started even if another message is also pending in this node. This may enable another node to transmit a message, which may have a lower priority than the second message in this node.*

30.3.5.8 TX event handling

To support TX event handling, the M_TTCAN has implemented a TX Event FIFO. After the M_TTCAN has transmitted 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 a maximum of 32 elements. The TX Event FIFO element is described in **"TX event FIFO element"** on page 557.

The purpose of the TX Event FIFO is to decouple handling transmit status information from transmit message handling; that is, a TX buffer holds only the message to be transmitted, while the transmit status is stored separately in the TX Event FIFO. This has the advantage, especially when operating a dynamically managed transmit queue, that a TX buffer can be used for a new message immediately after successful transmission. There is no need to save transmit status information from a TX buffer before overwriting that TX buffer.

When a TX Event FIFO full condition is signaled by IR.TEFF, no further elements are written to the TX Event FIFO until at least one element is read out and the TX Event FIFO Get Index is incremented. In case a TX event occurs while the TX Event FIFO is full, this event is discarded and interrupt flag IR.TEFL is set.

To avoid a TX Event FIFO overflow, the TX Event FIFO watermark can be used. When the TX Event FIFO fill level reaches the TX Event FIFO watermark configured by TXEFC.EFWM, interrupt flag IR.TEFW is set.

When reading from the TX Event FIFO, the TX Event FIFO Get Index TXEFS.EFGI must be added twice to the TX Event FIFO start address TXEFC.EFSA.

30.3.6 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 updates the FIFO Fill Level.

When the application does not use 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 (value is the index of the last element read), to update the FIFO's Get Index

Because the CPU has free access to the M_TTCAN's message RAM, take care when reading FIFO elements in an arbitrary order (Get Index not considered). 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 a wrong position and alter the FIFO's Fill Level. Some older FIFO elements are lost.

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.

30.3.7 Configuring the CAN bit timing

Each node in the CAN network has its own clock generator (usually a quartz oscillator). The time parameter of the bit time can be configured individually for each CAN node. Even if each CAN node's oscillator has a different period, a common bit rate can be generated.

The oscillator frequencies vary slightly because of changes in temperature or voltage, or deterioration of components. As long as the frequencies vary only within the tolerance range of the oscillators, the CAN nodes can compensate for the different bit rates by resynchronizing to the bit stream.

30.3.7.1 CAN bit timing

The CAN FD operation defines two bit times – nominal bit time and data bit time. The nominal bit time is for the arbitration phase. The data bit time has an equal or shorter length and can be used to accelerate the data phase (see **“CAN FD operation”** on page 528).

The basic construction of a bit time is shared with both the nominal and data bit times. The bit time can be divided into four segments according to the CAN specifications (see **Figure 30-12**, the synchronization segment (Sync_Seg), the propagation time segment (Prop_Seg), the phase buffer segment 1 (Phase_Seg1), and the phase buffer segment 2 (Phase_Seg2). The sample point at which the bus level is read and interpreted as the value of that respective bit, is located at the end of Phase_Seg1.

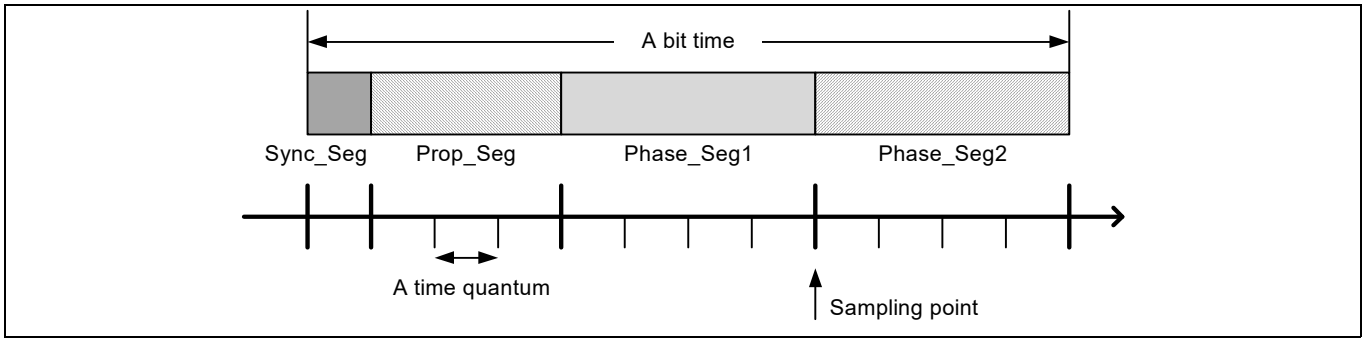


Figure 30-12. Bit time construction

Each segment consists of a programmable number of time quanta, which is a multiple of the time quantum that is defined by clk_can and a prescaler. The values and prescalers used to define these parameters differ for the nominal and data bit times, and are configured by NBTP (Nominal Bit Timing and Prescaler Register) and DBTP (Data Bit Timing and Prescaler Register) as shown in **Table 30-7**.

Table 30-7. 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
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

These relations result in the following equations for the nominal and data bit times:

Nominal Bit time

= [Sync_Seg + Prop_Seg + Phase_Seg1 + Phase_Seg2] × tq

= [1 + (NBTP.NTSEG1[7:0] + 1) + (NBTP.NTSEG2[6:0] + 1)] × [(NBTP.NBRP[8:0] +1) × clk_can period]

Data Bit time

= [1 + (DBTP.DTSEG1[4:0] + 1) + (DBTP.DTSEG2[3:0] + 1)] × [(DBTP.DBRP[4:0] +1) × clk_can period]

Note: The Information Processing Time (IPT) of the CAN FD controller is zero; this means that the data for the next bit is available at the first CAN clock edge after the sample point. Therefore, the IPT does not have to be accounted for when configuring Phase_Seg2, which is the maximum of Phase_Seg1 and the IPT.

30.3.7.2 CAN bit rates

The bit rate is the inverse of bit time; therefore, the nominal bit rate is

1 / [{1 + (NBTP.NTSEG1[7:0] + 1) + (NBTP.NTSEG2[6:0] + 1)} × {(NBTP.NBRP[8:0] +1) × clk_can period}]

and the data bit rate is

1 / [{1 + (DBTP.DTSEG1[4:0] + 1) + (DBTP.DTSEG2[3:0] + 1)} × {(DBTP.DBRP[4:0] +1) × clk_can period}]

From these formulas, we can see that the bit rates of the CAN FD controller depends on the CAN clock (clk_can) period, and the range each parameter can be configured to. The following tables list examples of the configurable bit rates at varying CAN clock frequencies. Empty boxes indicate that the desired bit rate cannot be configured at the specified input CAN clock frequency.

CAN clock frequency	8MHz		10MHz		16MHz		20MHz		32MHz		40MHz	
configuration / nominal bit rate	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1	Number of tqs per bit time	NBTP.NBRP + 1
125Kbps	64tq 32tq 16tq 8tq	1 2 4 8	80tq 40tq 20tq 10tq	1 2 4 8	128tq 64tq 32tq 16tq 8tq	1 2 4 8 16	160tq 80tq 40tq 20tq 10tq	1 2 4 8 16	256tq 128tq 64tq 32tq 16tq 8tq	1 2 4 8 16 32	320tq 160tq 80tq 40tq 20tq 10tq	1 2 4 8 16 32
250Kbps	32tq 16tq 8tq	1 2 4	40tq 20tq 10tq	1 2 4	64tq 32tq 16tq 8tq	1 2 4 8	80tq 40tq 20tq 10tq	1 2 4 8	128tq 64tq 32tq 16tq 8tq	1 2 4 8 16	160tq 80tq 40tq 20tq 10tq	1 2 4 8 16
500Kbps	16tq 8tq	1 2	20tq 10tq	1 2	32tq 16tq 8tq	1 2 4	40tq 20tq 10tq	1 2 4	64tq 32tq 16tq 8tq	1 2 4 8	80tq 40tq 20tq 10tq	1 2 4 8
1Mbps	8tq	1	10tq	1	16tq 8tq	1 2	20tq 10tq	1 2	32tq 16tq 8tq	1 2 4	40tq 20tq 10tq	1 2 4

Figure 30-13. 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 30-14. Example configuration for data bit rates

Note: The user must configure the CAN bit timings to comply with the corresponding CAN standards to ensure proper communication on the CAN bus.

30.4 Message RAM

The MRAM stores RX/TX messages and filter configurations.

Note:

- The MRAM should be made zero before configuration of the CAN FD controller to prevent bit errors when reading uninitialized words, and to avoid unexpected filter element configurations in the MRAM
- Unused MRAM cannot be used for general purposes

30.4.1 Message RAM configuration

The message RAM has a width of 32 bits. The CAN FD controller can be configured to allocate up to 4480 words in the MRAM (note that the number of words that can be used will be limited by the size of the actual MRAM). It is not necessary to configure each of the sections listed in **Figure 30-15**, nor is there any restriction with respect to the sequence of the sections.

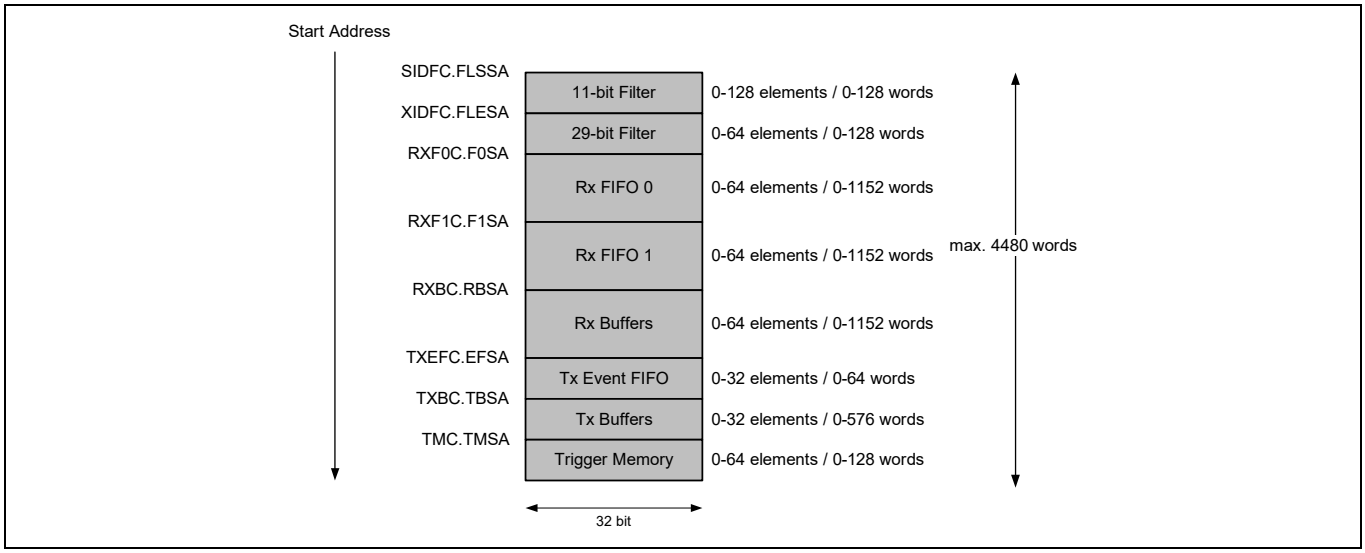


Figure 30-15. Message RAM configuration

The CAN FD controller addresses the message RAM in 32-bit words, not single bytes. The configurable start addresses are 32-bit word addresses – only bits 15 to 2 are evaluated, the two least significant bits are ignored.

Note:

- The CAN FD controller does not check for erroneous configuration of the message RAM. The configuration of the start addresses of different sections and the number of elements of each section should be done carefully to avoid falsification or loss of data.
- Message RAM is accessible by both M_TTCAN and CPU. Dynamic round-robin scheme is implemented to allocate access

30.4.2 RX buffer and FIFO element

An RX buffer and FIFO element is a block of 32-bit words, which holds the data and status of a received frame that was stored in the message RAM.

Up to 64 RX buffers and two RX FIFOs 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 30-16**. The element size can be configured to store CAN FD messages with up to 64 bytes data field via register RXESC (RX buffer/FIFO element Size Configuration).

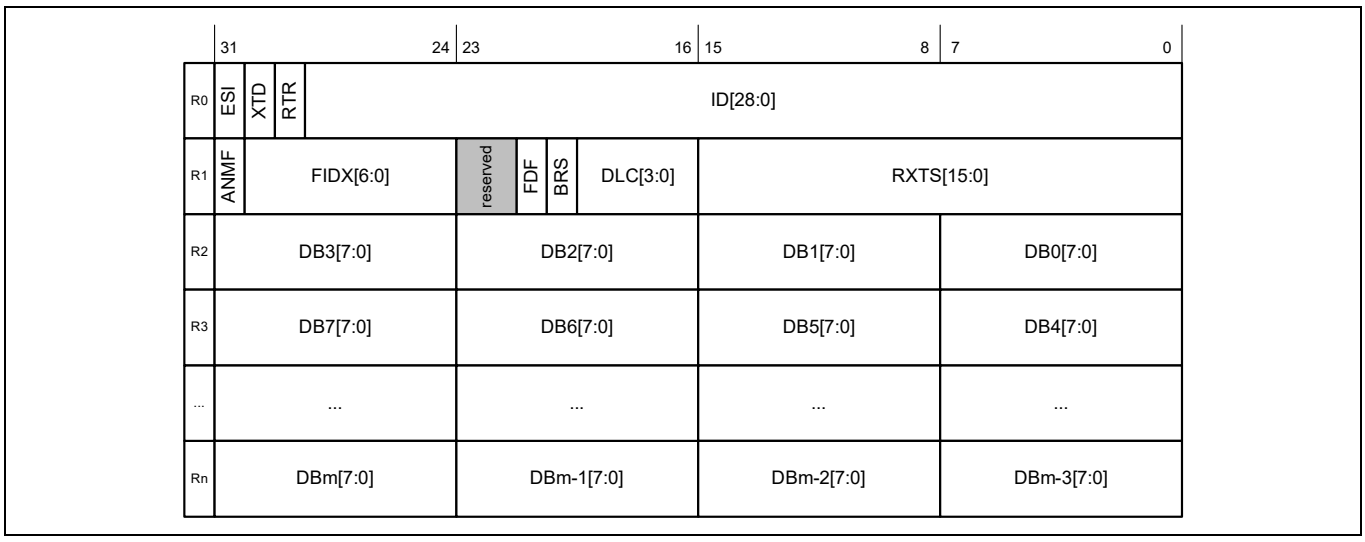


Figure 30-16. RX buffer and FIFO

R0 [bit31] ESI: Error State Indicator

Bit	Description
0	Transmitting node is error active.
1	Transmitting node is error passive.

R0 [bit30] XTD: Extended Identifier

Signals to the CPU whether the received frame has a standard or extended identifier.

Bit	Description
0	11-bit standard identifier.
1	29-bit extended identifier.

R0 [bit29] RTR: Remote Transmission Request

Signals to the CPU whether the received frame is a data frame or a remote frame.

Bit	Description
0	Received frame is a data frame.
1	Received frame is a remote frame.

Note: There are no remote frames in CAN FD format. In CAN FD frames (FDF = 1), the dominant RRS (Remote Request Substitution) bit replaces bit RTR (Remote Transmission Request).

R0 [bit28:0] ID[28:0]: Identifier

Standard or extended identifier depending on bit XTD. A standard identifier is stored into ID[28:18].

R1 [bit31] ANMF: Accepted Non-matching Frame.

Acceptance of non-matching frames may be enabled via GFC.ANFS[1:0] (Accept Non-matching Frames Standard) and GFC.ANFE[1:0] (Accept Non-matching Frames Extended).

Bit	Description
0	Received frame matching filter index FIDX.
1	Received frame did not match any RX filter element.

R1 [bit30:24] FIDX[6:0]: Filter Index

FIDX[6:0]	Description
0-127	Index of matching RX acceptance filter element (invalid if ANMF = 1). Range is 0 to List Size Standard/Extended minus 1 (SIDFC.LSS - 1 resp. XIDFC.LSE - 1).

R1 [bit23:22] Reserved: Reserved Bits

When writing, always write ‘0’. The read value is undefined.

R1 [bit21] FDF: Extended Data Length

Bit	Description
0	Classic CAN frame format.
1	CAN FD frame format.

R1 [bit20] BRS: Bit Rate Switch

Bit	Description
0	Frame received without bit rate switching.
1	Frame received with bit rate switching.

R1 [bit19:16] DLC[3:0]: Data Length Code

DLC[3:0]	Description
0-8	Classic CAN + CAN FD: received frame has 0-8 data bytes.
9-15	Classic CAN: received frame has 8 data bytes. CAN FD: received frame has 12/16/20/24/32/48/64 data bytes. See **Table 30-1** for details.

R1 [bit15:0] RXTS[15:0]: RX Timestamp

Timestamp Counter value captured on start of frame reception. Resolution depending on configuration of the Shared Timestamp Counter Pre-scaler TS_CTL.PRESCALE[15:0].

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)), Rn will vary from n = 3 to 17
- m is a function of n, m = (n - 1) × 4 - 1
- The number of valid data bytes are defined by the Data Length Code

30.4.3 TX buffer element

A TX buffer element is a block of 32-bit words stored in the message RAM that holds data and control information of a frame to be transmitted by the CAN FD controller.

The TX Buffers section can be configured to hold dedicated TX buffers and a TX FIFO/TX Queue. If the TX Buffers section is shared by dedicated TX buffers and a TX FIFO/TX Queue, the dedicated TX buffers start at the beginning of the TX Buffers section followed by the buffers assigned to the TX FIFO or TX Queue. The TX handler distinguishes between dedicated TX buffers and TX FIFO/TX Queue by evaluating the TX buffer configuration TXBC.TFQS[5:0] (Transmit FIFO/Queue Size) and TXBC.NDTB[5:0] (Number of Dedicated Transmit Buffers). The element size can be configured to store CAN FD messages with up to 64 bytes data field via register TXESC (TX Buffer Element Size Configuration).

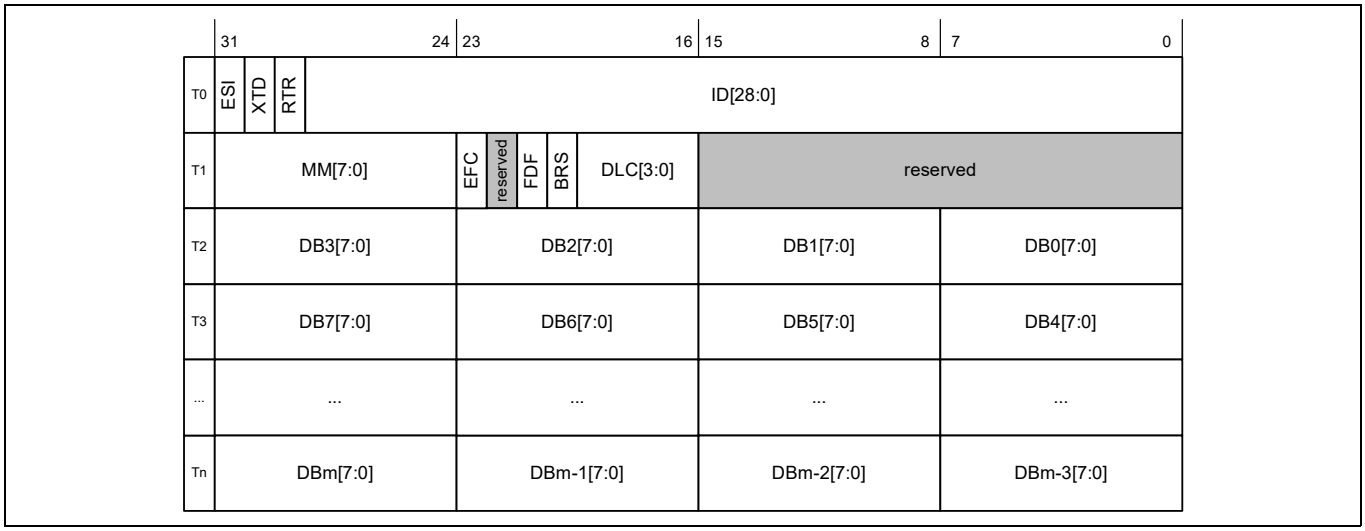


Figure 30-17. TX buffer element

T0 [bit31] ESI: Error State Indicator

Bit	Description
0	ESI bit in CAN FD format depends only on error passive flag.
1	ESI bit in CAN FD format transmitted recessive.

Note: The ESI bit of the transmit buffer is ORed with the error passive flag to decide the value of the ESI bit in the transmitted FD frame. As required by the CAN FD protocol specification, an error active node may optionally transmit the ESI bit recessive, but an error passive node will always transmit the ESI bit recessive.

T0 [bit30] XTD: Extended Identifier

Bit	Description
0	11-bit standard identifier.
1	29-bit extended identifier.

T0 [bit29] RTR: Remote Transmission Request

Bit	Description
0	Transmit data frame.
1	Transmit remote frame.

Note: When RTR = 1, the CAN FD controller transmits a remote frame according to ISO11898-1, even if FD Operation Enable (CCCR.FDOE) enables the transmission in CAN FD format.

T0 [bit28:0] ID[28:0]: Identifier

Standard or extended identifier depending on bit XTD. A standard identifier has to be written to ID[28:18].

T1 [bit31:24] MM[7:0]: Message Marker

Written by CPU during TX buffer configuration. Copied into TX Event FIFO element for identification of TX message status.

T1 [bit23] EFC: Event FIFO ControlT1 [bit22] Reserved: Reserved Bit

Bit	Description
0	Don't store TX events.
1	Store TX events.

When writing, always write ‘0’. The read value is undefined.

T1 [bit21] FDF: FD Format

Bit	Description
0	Frame transmitted in Classic CAN format.
1	Frame transmitted in CAN FD format.

T1 [bit20] BRS: Bit Rate Switching

Bit	Description
0	CAN FD frames transmitted without bit rate switching.
1	CAN FD frames transmitted with bit rate switching.

*Note: Bits ESI, FDF, and BRS are only evaluated when CAN FD operation is enabled CCCR.FDOE = 1. Bit BRS is only evaluated when in addition CCCR.BRSE = 1. See **Table 30-5** for details of bits FDF and BRS.*

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 has 12/16/20/24/32/48/64 data bytes.

T1 [bit15:0] Reserved: Reserved Bits

When writing, always write ‘0’. The read value is undefined.

T2 [bit31:24]	DB3[7:0] :	Data Byte 3
T2 [bit23:16]	DB2[7:0] :	Data Byte 2
T2 [bit15:8]	DB1[7:0] :	Data Byte 1
T2 [bit7:0]	DB0[7:0] :	Data Byte 0
T3 [bit31:24]	DB7[7:0] :	Data Byte 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 17.
- m is a function of n: m = (n – 1) × 4 – 1.

30.4.4 TX event FIFO element

Each TX Event FIFO Element stores information about transmitted messages. By reading the TX Event FIFO, the CPU gets this information in the order the messages were transmitted. Status information about the TX Event FIFO can be obtained from register TXEFS (TX Event FIFO Status).

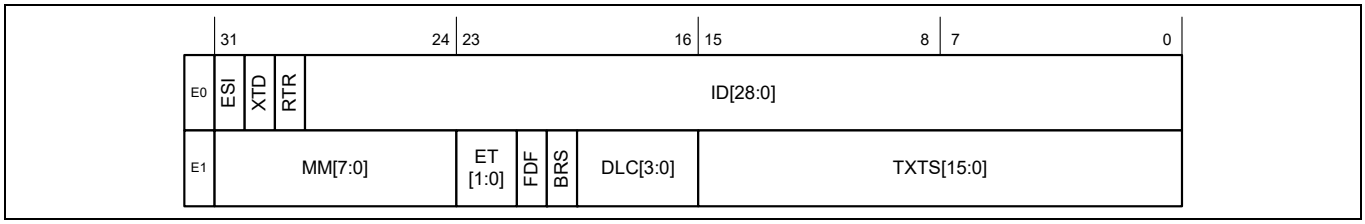


Figure 30-18. TX event FIFO element

E0 [bit31] ESI: Error State Indicator

Bit	Description
0	Transmitting node is error active.
1	Transmitting node is error passive.

E0 [bit30] XTD: Extended Identifier

Bit	Description
0	11-bit standard identifier.
1	29-bit extended identifier.

E0 [bit29] RTR: Remote Transmission Request

Bit	Description
0	Data frame transmitted.
1	Remote frame transmitted.

E0 [bit28:0] ID[28:0]: Identifier

Standard or extended identifier depending on bit XTD. A standard identifier is stored into ID[28:18].

E1 [bit31:24] MM[7:0]: Message Marker

Copied from TX buffer into TX Event FIFO element for identification of TX message status.

E1 [bit23:22] ET[1:0]: Event Type

ET[1:0]	Description
00	Reserved.
01	TX event.
10	Transmission in spite of cancellation. Always set for transmissions in DAR mode (Disable Automatic Retransmission mode).
11	Reserved.

E1 [bit21] FDF: FD Format

Bit	Description
0	Classic CAN frame format.
1	CAN FD frame format (new DLC-coding and CRC).

E1 [bit20] BRS: Bit Rate Switching

Bit	Description
0	Frame transmitted without bit rate switching.
1	Frame transmitted with bit rate switching.

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 12/16/20/24/32/48/64 data bytes transmitted. See **Table 30-1** for details.

E1 [bit15:0] TXTS[15:0]: TX Timestamp

Timestamp Counter value captured on start-of-frame transmission. Resolution depending on configuration of the Shared Timestamp Counter Pre-scaler TS_CTL.PRESCALE[15:0].

30.4.5 Standard Message ID Filter Element

A Standard Message ID Filter Element consists of a single 32-bit word, and can be configured as a range filter, dual filter, classic bit mask filter, or filter for a single dedicated ID, for messages with 11-bit standard IDs.

Up to 128 filter elements can be configured for 11-bit standard IDs. When accessing a Standard Message ID Filter element, its address is

Filter List Standard Start Address (SIDFC.FLSSA[15:2]) + index of the filter element (0 to 127).

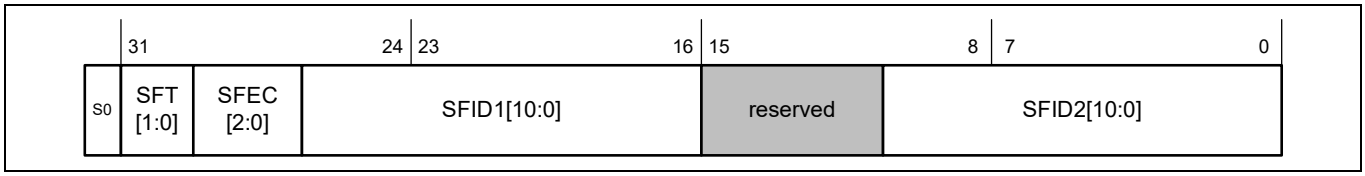


Figure 30-19. Standard Message ID filter

S0 [bit31:30] SFT[1:0]: Standard Filter Type

SFT[1:0]	Description
00	Range filter from SFID1[10:0] to SFID2[10:0] (SFID2[10:0] ≥ received ID ≥ SFID1[10:0]).
01	Dual ID filter for SFID1[10:0] or SFID2[10:0].
10	Classic filter: SFID1[10:0] = filter, SFID2[10:0] = mask. Only those bits of SFID1[10:0] where the corresponding SFID2[10:0] bits are 1 are relevant.
11	Filter element disabled.

Note: With SFT = 11, the filter element is disabled and the acceptance filtering continues (same behavior as with SFEC = 000).

S0 [bit29:27] SFEC[2:0]: Standard Filter Element Configuration

All enabled filter elements are used for acceptance filtering of standard frames. Acceptance filtering stops at the first matching enabled filter element or when the end of the filter list is reached.

If SFEC[2:0] = 100, 101, or 110 a match sets interrupt flag IR.HPM (High Priority Message) and, if enabled, an interrupt is generated. In this case register HPMS (High Priority Message Status) is updated with the status of the priority match.

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 [bit26:16] SFID1[10:0]: Standard Filter ID 1

This bit field has a different meaning depending on the configuration of SFEC[2:0]:

- SFEC[2:0] = 001 to 110

Set SFID1[10:0] according to the SFT[1:0] setting.

- SFEC[2:0] = 111

SFID1[10:0] defines the ID of a standard dedicated RX buffer or debug message to be stored. The received identifiers must match, no masking mechanism is used.

S0 [bit15:11] Reserved: Reserved Bits

When writing, always write ‘0’. The read value is undefined.

S0 [bit10:0] SFID2[10:0]: Standard Filter ID 2

This bit field has a different meaning depending on the configuration of SFEC[2:0]:

- SFEC[2:0] = 001 to 110

Set SFID2[10:0] according to the SFT[1:0] setting

- SFEC[2:0] = 111

Filter for dedicated RX buffers or for debug messages

SFID2[10:9] decides whether the received message is stored into a dedicated RX buffer or treated as message A, B, or C of the debug message sequence.

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] to store a matching message.

Note: Debug message is used to debug on CAN feature.

30.4.6 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, dual filter, classic bit mask filter, or filter for a single dedicated ID, for messages with 29-bit extended IDs.

Up to 64 filter elements can be configured for 29-bit extended IDs. When accessing an Extended Message ID Filter element, its address is

Filter List Extended Start Address (XIDFC.FLESA[15:2]) + 2 × index of the filter element (0 to 63).

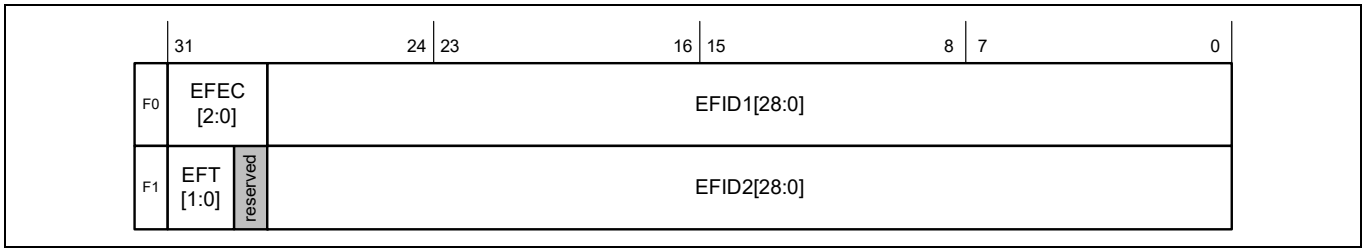


Figure 30-20. Extended Message ID filter

F0 [bit31:29] EFEC[2:0]: Extended Filter Element Configuration

All enabled filter elements are used for acceptance filtering of extended frames. Acceptance filtering stops at the first matching enabled filter element or when the end of the filter list is reached.

If EFEC[2:0] = 100, 101, or 110 a match sets interrupt flag IR.HPM (High Priority Message) and, if enabled, an interrupt is generated. In this case register HPMS (High Priority Message Status) is updated with the status of the priority match.

EFEC[2:0]	Description
000	Disable filter element.
001	Store in RX FIFO 0 if filter matches.
010	Store in RX FIFO 1 if filter matches.
011	Reject ID if filter matches.
100	Set priority if filter matches.
101	Set priority and store in RX FIFO 0 if filter matches.
110	Set priority and store in RX FIFO 1 if filter matches.
111	Store into dedicated RX buffer or as debug message, configuration of EFT[1:0] ignored.

F0 [bit28:0] EFID1[28:0]: Extended Filter ID 1

This bit field has a different meaning depending on the configuration of EFEC[2:0].

- EFEC[2:0] = 001 to 110

Set EFID1[28:0] according to the EFT[1:0] setting.

- EFEC[2:0] = 11

EFID1[28:0] defines the ID of an extended dedicated RX buffer or debug message to be stored. The received identifiers must match, only XIDAM masking mechanism is used.

F1 [bit31:30] EFT[1:0]: Extended Filter Type

F1 [bit29] Reserved: Reserved Bit

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.
10	Classic filter: EFID1[28:0] = filter, EFID2[28:0] = mask. Only those bits of EFID1[28:0] where the corresponding EFID2[28:0] bits are 1 are relevant. Matches when the received ID ANDed with XIDAM is equal to EFID1[28:0] masked by EFID2[28:0].
11	Range filter from EFID1[28:0] to EFID2[28:0] (EFID2[28:0] ≥ EFID1[28:0]), XIDAM mask not applied.

When writing, always write ‘0’. The read value is undefined.

F1 [bit28:0] EFID2[28:0]: Extended Filter ID 2

This bit field has a different meaning depending on the configuration of EFEC[2:0]:

- EFEC[2:0] = 001 to 110

Set EFID2[28:0] according to the EFT[1:0] setting

- EFEC[2:0] = 111

EFID2[28:0] is used to configure this filter for dedicated RX buffers or for debug messages

EFID2[28:11] are reserved bits. When writing, always write ‘0’. The read value is undefined.

EFID2[10:9] decides whether the received message is stored into a dedicated RX buffer or treated as message A, B, or C of the debug message sequence.

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: Debug message is used to debug on CAN feature.

30.4.7 Trigger memory element

Up to 64 trigger memory elements can be configured. When accessing a trigger memory element, its address is the Trigger Memory Start Address TTTMC.TMSA plus the index of the trigger memory element (0…63).

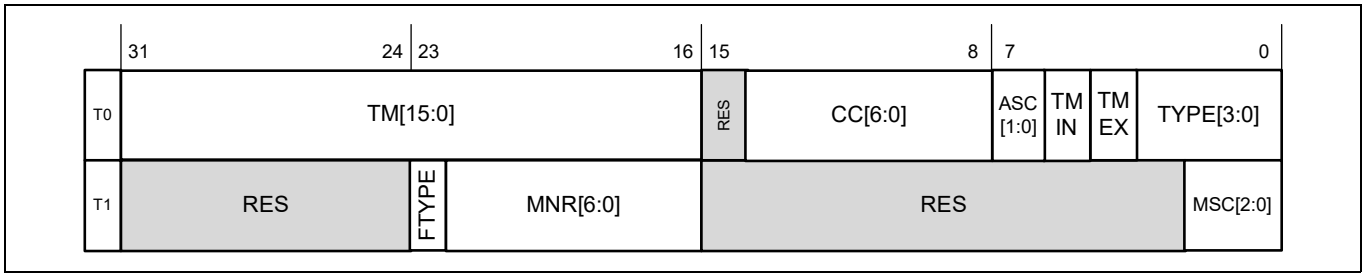


Figure 30-21. Trigger memory element

T0 Bit 31:16 TM[15:0]: Time Mark

Cycle time for which the trigger becomes active.

T0 Bit 14:8 CC[6:0]: Cycle Code

Cycle count for which the trigger is valid. Ignored for trigger types Tx_Ref_Trigger, Tx_Ref_Trigger_Gap, Watch_Trigger, Watch_Trigger_Gap, and End_of_List.

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.

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.

T0 Bit 5 TMIN: Time Mark Event Internal

TMIN	Description
0	No action.
1	TTIR.TTMI is set when trigger memory element becomes active.

T0 Bit 4 TMEX: Time Mark Event External

TMEX	Description
0	No action.
1	Pulse at output of Trigger Time Mark with the length of one clk_can period is generated when the time mark of the trigger memory element becomes active and TTOCN.TTMIE = 1.

T0 Bit 3:0 TYPE[3:0]: Trigger Type

Watch_Trigger_Gap - valid when in gap.

Note:

- For ASC operation (ASC = 10, 11) only trigger types Rx_Trigger and Time_Base_Trigger should be used
- ASC operation is not supported in this device

T1 Bit 23 FTYPE: Filter Type

FTYPE	Description
0	11-bit standard message ID.
1	29-bit extended message ID.

T1 Bit 22:16 MNR[6:0]: Message Number

Transmission: Trigger is valid for configured TX buffer number. Valid values are 0 to 31.

Reception: Trigger is valid for standard/extended message ID filter element number. Valid values are 0 to 63 and 0 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 and in event-driven CAN communication (ISO 11898-1:2015).

Note:

- The trigger memory elements should be written when the M_TTCAN is in INIT state. Write access to the trigger memory elements outside INIT state is not allowed.
- There is an exception for TMIN and TMEX when they are defined as part of a trigger memory element of TYPE Tx_Ref_Trigger. In this case they become active at the time mark modified by the actual Reference Trigger Offset (TTOST.RTO).

MRAM OFF

Message RAM can be turned off to save power by setting CTL.MRAM_OFF bit. Default value of this bit is '0' and MRAM is retained in this configuration during DeepSleep power mode.

The M_TTCAN channel must be powered down before setting MRAM_OFF bit. See **“Power down (Sleep mode)”** on page 532 to power down the M_TTCAN channel. When MRAM is OFF, any access to MRAM may raise Address Error (MRAM_SIZE = 0).

After switching the MRAM on again, software needs to allow a certain power-up time before MRAM can be used, that is, before STOP_REQ can be de-asserted. Check the RAM_PWR_DELAY_CTL register to see the required time for MRAM power up process. Refer to the registers reference manual for more details.

30.4.8 RAM watchdog (RWD)

The RAM Watchdog monitors the READY output of the Message RAM. A Message RAM access starts the Message RAM watchdog counter with the value configured by RWD.WDC. The counter is reloaded with RWD.WDC when the Message RAM signals successful completion by activating its READY output. In case there is no response from the Message RAM until the counter has counted down to zero, the counter stops and interrupt flag IR.WDI is set. The RAM watchdog counter is clocked by the host clock (clk_sys). Refer to the registers reference manual for information about the RWD register.

30.5 TTCAN operation

30.5.1 Reference message

A reference message is a data frame characterized by a specific CAN identifier. It is received and accepted by all nodes except the time master (sender of the reference message).

For Level 1, the data length must be at least one. For Level 0 and Level 2, the data length must be at least four; otherwise, the message is not accepted as a reference message. The reference message may be extended by other data up to the sum of eight CAN data bytes. All bits of the identifier except the three LSBs characterize the message as a reference message. The last three bits specify the priorities of up to eight potential time masters. Reserved bits are transmitted as logical 0 and are ignored by the receivers. The reference message is configured using the TTRMC register.

The time master transmits the reference message. If the reference message is disturbed by an error, it is retransmitted immediately. In a retransmission, the transmitted Master_Ref_Mark is updated. The reference message is sent periodically, but it is allowed to stop the periodic transmission (Next_is_Gap bit). It can initiate event-synchronized transmission at the start of the next basic cycle by the current time master or by one of the other potential time masters.

The node transmitting the reference message is the current time master. The time master is allowed to transmit other messages. If the current time master fails, its function is replicated by the potential time master with the highest priority. Nodes that are neither time master nor potential time master are time-receiving nodes.

30.5.1.1 Level 1

Level 1 operation is configured via TTOCF.OM = 01 and TTOCF.GEN. External clock synchronization is not available in Level 1.

The information related to the reference message is stored in the first data byte as shown in **Table 30-8**. Cycle_Count is optional.

Table 30-8. First byte of level 1 reference message

Bits	7	6	5	4	3	2	1	0
First Byte	Next_is_Gap	Reserved	Cycle_Count [5:0]					

30.5.1.2 Level 2

Level 2 operation is configured via TTOCF.OM = 10 and TTOCF.GEN.

The information related to the reference message is stored in the first four data bytes as shown in **Table 30-9**. Cycle_Count and the lower four bits of NTU_Res are optional. The M_TTCAN does not evaluate NTU_Res[3:0] from received reference messages, it always transmits these bits as zero.

Table 30-9. First four bytes 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]							

30.5.1.3 Level 0

Level 0 operation is configured via TTOCF.OM = 11. External event-synchronized time-triggered operation is not available in Level 0.

The information related to the reference message is stored in the first four data bytes as shown in **Table 30-10**. In Level 0, Next_is_Gap is always zero. Cycle_Count and the lower four bits of NTU_Res are optional. The M_TTCAN does not evaluate NTU_Res[3:0] from received reference messages; it always transmits these bits as zero.

Table 30-10. 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]							

30.5.2 TTCAN configuration

30.5.2.1 TTCAN timing

The Network Time Unit (NTU) is the unit in which all times are measured. The NTU is a constant of the whole network and is defined by the network system designer. In TTCAN Level 1 the NTU is the nominal CAN bit time. In TTCAN Level 0 and Level 2 the NTU is a fraction of the physical second.

The NTU is the time base for the local time. The integer part of the local time (16-bit value) is incremented once for each NTU. Cycle time and global time are both derived from local time. The fractional part (3-bit value) of local time, cycle time, and global time is not readable.

In TTCAN Level 0 and Level 2, the length of the NTU is defined by the Time Unit Ratio (TUR). The TUR is a non-integer number given by the formula TUR = TURNA.NAV/TURCF.DC. The length of the NTU is given by the formula NTU = CAN Clock Period × TUR.

The TUR Numerator Configuration NC is an 18-bit number, TURCF.NCL[15:0] can be programmed in the range 0x0000-0xFFFF. TURCF.NCH[17:16] is hard-wired to 0b01. When the number 0xnnnn is written to TURCF.NCL[15:0], TURNA.NAV starts with the value 0x10000 + 0x0nnnn = 0x1nnnn. The TUR Denominator Configuration TURCF.DC is a 14-bit number. TURCF.DC may be programmed in the range 0x0001 - 0x3FFF; 0x0000 is an illegal value.

In Level 1, NC must be ≥ 4 × TURCF.DC. In Level 0 and Level 2 NC must be ≥ 8 × TURCF.DC to allow the 3-bit resolution for the internal fractional part of the NTU.

A hardware reset presets TURCF.DC to 0x1000 and TURCF.NCL to 0x10000, resulting in an NTU consisting of 16 CAN clock periods. Local time and application watchdog are not started before either the CCCR.INIT is reset or TURCF.ELT is set. TURCF.ELT may not be set before the NTU is configured. Setting TURCF.ELT to '1' also locks the write access to register TURCF.

At startup TURNA.NAV is updated from NC (= TURCF.NCL + 0x10000) when TURCF.ELT is set. In TTCAN Level 1 there is no drift compensation. TURNA.NAV does not change during operation, it always equals NC.

In TTCAN Level 0 and Level 2, there are two possibilities for TURNA.NAV to change. When operating as time slave or backup time master, and when TTOCF.ECC is set, TURNA.NAV is updated automatically to the value calculated from the monitored global time speed, as long as the M_TTCAN is in synchronization states In_Schedule or In_Gap. When it loses synchronization, it returns to NC. When operating as the actual time master, and when TTOCF.EECS is set, the host may update TURCF.NCL. When the host sets TTOCN.ECS, TURNA.NAV will be updated from the new value of NC at the next reference message. The status flag TTOST.WECS is set when TTOCN.ECS is set and is cleared when TURNA.NAV is updated. TURCF.NCL is write-locked while TTOST.WECS is set.

In TTCAN Level 0 and Level 2, the clock calibration process adapts TURNA.NAV in the range of the synchronization deviation limit (SDL) of NC ± 2 (TTOCF.LDSDL+5). TURCF.NCL should be programmed to the largest applicable numerical value to achieve the best accuracy in the calculation of TURNA.NAV.

The synchronization deviation (SD) is the difference between NC and TURNA.NAV (SD = |NC - TURNA.NAV|). It is limited by the SDL, which is configured by its dual logarithm TTOCF.LDSDL (SDL = 2 (TTOCF.LDSDL+5)) and should not exceed the clock tolerance given by the CAN bit timing configuration. SD is calculated at each new basic cycle. When the calculated TURNA.NAV deviates by more than SDL from NC, or if the Disc_Bit in the reference message is set, the drift compensation is suspended, TTIR.GTE is set, and TTOSC.QCS is reset; if Disc_Bit = 1, TTIR.GTD is set. TUR configuration examples are shown in **Table 30-11**.

Table 30-11. TUR configuration examples

TUR	8	10	24	50	510	125000	32.5	100/12	529/17
NC	0x1FFF8	0x1FFFE	0x1FFF8	0x1FFEA	0x1FFFE	0x1FFE0	0x1FFE0	0x19000	0x10880
TURCF.DC	0x3FFF	0x3333	0x1555	0x0A3D	0x0101	0x0001	0x0FC0	0x3000	0x0880

TTOCN.ECS schedules NC for activation by the next reference message. TTOCN.SGT schedules TTGTP.TP for activation by the next reference message. Setting of TTOCN.ECS and TTOCN.SGT requires TTOCF.EECS to be set (external clock synchronization enabled) while the M_TTCAN is actual time master.

The M_TTCAN module provides an application watchdog to verity the function of the application program. The host has to serve this watchdog regularly, else all CAN bus activity is stopped. The Application Watchdog Limit TTOCF.AWL specifies the number of NTUs the watchdog has to be served. The maximum number of NTUs is 256. The Application Watchdog is served by reading register TTOST. TTOST.AWE indicates whether the watchdog is served in time. In case the application failed to serve the application watchdog, interrupt flag TTIR.AW is set. For software development, the application watchdog may be disabled by programming TTOCF.AWL to 0x00 (see **“Application Watchdog”** on page 534).

30.5.2.2 Message scheduling

TTOCF.TM controls whether the M_TTCAN operates as a potential time master or as a time slave. If it is a potential time master, the three LSBs of the reference message identifier TTRMC.RID define the master priority, 0 being the highest and 7 the lowest priority. Two nodes in the network may not useg the same master priority. TTRMC.RID is used for recognition of reference messages. TTRMC.RMPS is not relevant for time slaves.

The Initial Reference Trigger Offset TTOCF.IRTO is a 7-bit-value that defines (in NTUs) how long a backup time master waits before it starts the transmission of a reference message, when a reference message is expected but the bus remains idle. The recommended value for TTOCF.IRTO is the master priority multiplied with a factor depending on the expected clock drift between the potential time masters in the network. The sequential order of the backup time masters, when one of them starts the reference message if the current time master fails, should correspond to their master priority, even with maximum clock drift.

TTOCF.OM decides whether the node operates in TTCAN Level 0, Level 1, or Level 2. In one network, all potential time masters should operate on the same level. Time slaves may operate on Level 1 in a Level 2 network, but not vice versa. The configuration of the TTCAN operation mode via TTOCF.OM is the last step in the setup. When TTOCF.OM = 00 (event-driven CAN communication), the M_TTCAN operates according to ISO 11898-1:2015, without time triggers. When TTOCF.OM = 01 (Level 1), the M_TTCAN operates according to ISO 11898-4, but without the possibility to synchronize the basic cycles to external events, the Next_is_Gap bit in the reference message is ignored. When TTOCF.OM = 10 (Level 2), the M_TTCAN operates according to ISO 11898-4, including the event-synchronized start of a basic cycle. When TTOCF.OM = 11 (Level 0), the M_TTCAN operates as event-driven CAN but maintains a calibrated global time base similar to Level 2.

TTOCF.EECS enables the external clock synchronization, allowing the application program of the current time master to update the TUR configuration during time-triggered operation, to adapt the clock speed and (in Level 0,2 only) the global clock phase to an external reference.

TTMLM.ENTT in the TT Matrix Limits register specifies the number of expected Tx_Triggers in the system matrix. This is the sum of Tx_Triggers for exclusive single arbitrating and merged arbitrating windows, excluding the Tx_Ref_Triggers. Note that this is usually not the number of Tx_Trigger memory elements; the number of basic cycles in the system matrix and the trigger's repeat factors must be taken into account. An inaccurate configuration of TTMLM.ENTT will result in either a TX Count Underflow (TTIR.TXU = 1 and TTOST.EL = 01, severity 1) or in a TX Count Overflow (TTIR.TXO = 1 and TTOST.EL = 10, severity 2).

Note: *In case the first reference message seen by a node does not have Cycle_Count zero, this node may finish its first matrix cycle with its TX count resulting in a TX Count Underflow condition. As long as a node is in state, synchronizing its Tx_Triggers will not lead to transmissions.*

TTMLM.CCM specifies the number of the last basic cycle in the system matrix. The counting of basic cycles starts at 0. In a system matrix consisting of eight basic cycles TTMLM.CCM would be 7. TTMLM.CCM is ignored by time slaves, a receiver of a reference message considers the received cycle count as the valid cycle count for the actual basic cycle.

TTMLM.TXEW specifies the length of the TX enable window in NTUs. The TX enable window is the period at the beginning of a time window where a transmission may be started. If the sample point of the first bit of a transmit message is not inside the TX enable window, the transmission cannot be started in that time window at all. An example is because of an overlap from the previous time window's message. TTMLM.TXEW should be chosen based on the network's synchronization quality and the relation between the length of the time windows and the length of messages.

30.5.2.3 Trigger memory

The trigger memory is part of the message RAM. It stores up to 64 trigger elements. A trigger memory element consists of Time Mark TM, Cycle Code CC, Trigger Type TYPE, Filter Type FTYPE, Message Number MNR, Message Status Count MSC, Time Mark Event Internal TMIN, Time Mark Event External TMEX, and Asynchronous Serial Communication ASC (see **“Trigger memory element”** on page 563).

The time mark defines at which cycle time a trigger becomes active. The trigger elements in the trigger memory must be sorted by their time marks. The trigger element with the lowest time mark is written to the first trigger memory word. Message number and cycle code are ignored for triggers of type Tx_Ref_Trigger, Tx_Ref_Trigger_Gap, Watch_Trigger, Watch_Trigger_Gap, and End_of_List.

When the cycle time reaches the time mark of the actual trigger, the FSE switches to the next trigger and starts to read it from the trigger memory. For a transmit trigger, the TX handler starts to read the message from the message RAM as soon as the FSE switches to its trigger. The RAM access speed defines the minimum time step between a transmit trigger and its preceding trigger, the TX handler should be able to prepare the transmission before the transmit trigger's time mark is reached. The RAM access speed also limits the number of non-matching (with regard to their cycle code) triggers between two matching triggers, the next matching trigger must be read before its time mark is reached. If the reference message is n NTU long, a trigger with a time mark less than n will never become active and will be treated as a configuration error.

The starting point of cycle time is the sample point of the reference message's start-of-frame bit. The next reference message is requested when cycle time reaches the Tx_Ref_Trigger's time mark. The M_TTCAN reacts to the transmission request at the next sample point. A new Sync_Mark is captured at the start-of-frame bit, but the cycle time is incremented until the reference message is successfully transmitted (or received) and the Sync_Mark is taken as the new Ref_Mark. At that point, cycle time is restarted. As a consequence, cycle time can never (with the exception of initialization) be seen at a value less than n, with n being the length of the reference message measured in NTU.

Length of a basic cycle: Tx_Ref_Trigger time mark + 1 NTU + 1 CAN bit time

The trigger list will be different for all nodes in the TTCAN network. Each node knows only the Tx_Triggers for its own transmit messages, the Rx_Triggers for the receive messages that are processed by this node, and the triggers concerning the reference messages.

30.5.2.3.1 Trigger types

Tx_Ref_Trigger (TYPE = 0000) and Tx_Ref_Trigger_Gap (TYPE = 0001) cause the transmission of a reference message by a time master. A configuration error (TTOST.EL = 11, severity 3) is detected when a time slave encounters a Tx_Ref_Trigger(_Gap) in its trigger memory. Tx_Ref_Trigger_Gap is only used in external event-synchronized time-triggered operation mode. In that mode, Tx_Ref_Trigger is ignored when the M_TTCAN synchronization state is In_Gap (TTOST.SYS = 10).

Tx_Trigger_Single (TYPE = 0010), Tx_Continous (TYPE = 0011), Tx_Trigger_Arbitration (TYPE = 0100), and Tx_Trigger_Merged (TYPE = 0101) cause the start of a transmission. They define the start of a time window.

Tx_Trigger_Single starts a single transmission in an exclusive time window when the message buffer's Transmission Request Pending bit is set. After successful transmission, the Transmission Request Pending bit is reset.

Tx_Trigger_Continuous starts a transmission in an exclusive time window when the message buffer's transmission Request Pending bit is set. After successful transmission, the Transmission Request Pending bit remains set, and the message buffer is transmitted again in the next matching time window.

Tx_Trigger_Arbitration starts an arbitrating time window, Tx_Trigger_Merged a merged arbitrating time window. The last Tx_Trigger of a merged arbitrating time window must be of type Tx_Trigger_Arbitration. A Configuration Error (TTOST.EL = 11, severity 3) is detected when a trigger of type Tx_Trigger_Merged is followed by any other Tx_Trigger than one of type Tx_Trigger_Merged or Tx_Trigger_Arbitration. Several Tx_Triggers may be defined for the same TX message buffer. Depending on their cycle code, they may be ignored in some basic cycles. The cycle code should be considered when the expected number of Tx_Triggers (TTMLM.ENTT) is calculated.

Watch_Trigger (TYPE = 0110) and Watch_Trigger_Gap (TYPE = 0111) check for missing reference messages. They are used by both time masters and time slaves. Watch_Trigger_Gap is only used in external event-synchronized time-triggered operation mode. In that mode, a Watch_Trigger is ignored when the M_TTCAN synchronization state is In_Gap (TTOST.SYS = 10).

Rx_Trigger (TYPE = 1000) is used to check for the reception of periodic messages in exclusive time windows. Rx_Triggers are not active until state In_Schedule or In_Gap is reached. The time mark of an Rx_Trigger should be placed after the end of that message transmission, independent of time window boundaries. Depending on their cycle code, Rx_Triggers may be ignored in some basic cycles. At the Rx_Trigger time mark, it is checked whether the last received message before this time mark and after start of cycle or previous Rx_Trigger matches the acceptance filter element referenced by MNR. Accepted messages are stored in one of two receive FIFOs, according to the acceptance filtering, independent of the Rx_Trigger. Acceptance filter elements that are referenced by Rx_Triggers should be placed at the beginning of the filter list to ensure that the filtering is finished before the Rx_Trigger time mark is reached.

Time_Base_Trigger (TYPE = 1001) is used to generate internal/external events depending on the configuration of ASC, TMIN, and TMEX.

End_of_List (TYPE = 1010…1111) is an illegal trigger type, a configuration error (TTOST.EL = 11, severity 3) is detected when an End_of_List trigger is encountered in the trigger memory before the Watch_Trigger or Watch_Trigger_Gap.

30.5.2.3.2 Restrictions for the Node's Trigger List

Two triggers may not be active at the same cycle time and cycle count, but triggers that are active in different basic cycles (different cycle code) may share the same time mark.

Rx_Triggers and Time_Base_Triggers may not be placed inside the TX enable windows of Tx_Trigger_Single/Continuous/Arbitration, but they may be placed after Tx_Trigger_Merged.

Triggers that are placed after the Watch_Trigger (or the Watch_Trigger_Gap when TTOST.SYS = 10) will never become active. The watch triggers themselves will not become active when the reference messages are transmitted on time.

All unused trigger memory words (after the Watch_Trigger or after the Watch_Trigger_Gap when TTOST.SYS = 10) must be set to trigger type End_of_List.

A typical trigger list for a potential time master will begin with a number of Tx_Triggers and Rx_Triggers followed by the Tx_Ref_Trigger and Watch_Trigger. For networks with external event-synchronized time-triggered communication, this is followed by the Tx_Ref_Trigger_Gap and the Watch_Trigger_Gap. The trigger list for a time slave will be the same but without the Tx_Ref_Trigger and the Tx_Ref_Trigger_Gap.

At the beginning of each basic cycle, that is at each reception or transmission of a reference message, the trigger list is processed starting with the first trigger memory element. The FSE looks for the first trigger with a cycle code that matches the current cycle count. The FSE waits until cycle time reaches the trigger's time mark and activates the trigger. Later, the FSE looks for the next trigger in the list with a cycle code that matches the current cycle count.

Special consideration is needed for the time around Tx_Ref_Trigger and Tx_Ref_Trigger_Gap. In a time master competing for master ship, the effective time mark of a Tx_Ref_Trigger may be decremented to be the first node to start a reference message. In backup time masters the effective time mark of a Tx_Ref_Trigger or Tx_Ref_Trigger_Gap is the sum of its configured time mark and the Reference Trigger Offset TTOCF.IRTO. If error level 2 is reached (TTOST.EL = 10), the effective time mark is the sum of its time mark and 0x127. No other trigger elements should be placed in this range; otherwise, the time marks may appear out of order and are flagged as a configuration error. Trigger elements that are coming after Tx_Ref_Trigger may never become active as long as the reference messages come in 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

30.5.2.3.3 Examples of trigger handling

The following example shows how the trigger list is derived from a node's system matrix. Assume that node A is a first time master; a section of the system matrix shown in **Table 30-12**.

Table 30-12. 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 with 0 – 0, 1, 3, 7, 15, 31, 63 (the number of basic cycles in the system matrix is 1, 2, 4, 8, 16, 32, 64). The maximum cycle count is configured by TTMLM.CCM. The Cycle Code (CC) is composed of repeat factor (value of most significant '1') and the number of the first basic cycle in the system matrix (bit field after most significant '1').

Example: When CC is 0b0010011 (repeat factor: 16, first basic cycle: 3) and maximum cycle count of TTMLM.CCM = 0x3F, matches occur at cycle counts 3, 19, 35, 51.

A trigger element consists of Time Mark (TM), Cycle Code (CC), Trigger Type (TYPE), and Message Number (MNR). For transmission, MNR references the TX buffer number (0..31). For reception, MNR references the number of the filter element (0..127) that matched during acceptance filtering. Depending on the configuration of the Filter Type FTYPE, the 11-bit or 29-bit message ID filter list is referenced.

In addition, a trigger element can be configured for Asynchronous Serial Communication (ASC), generation of Time Mark Event Internal (TMIN), and Time Mark Event External (TMEX). The Message Status Count (MSC) holds the counter value (0..7) for scheduling errors for periodic messages in exclusive time windows when the time mark of the trigger element becomes active.

Table 30-13. Trigger list node A

Trigger	Time Mark TM[15:0]	Cycle Code CC[6:0]	Trigger Type TYPE[3:0]	Mess. No. MNR[6:0]
0	Mark1	0000100	Tx_Trigger_Single	7
1	Mark1	1000000	Rx_Trigger	3
2	Mark1	1000011	Tx_Trigger_Single	7
3	Mark3	1000001	Tx_Trigger_Merged	2
4	Mark3	1000011	Rx_Trigger	5
5	Mark4	1000001	Tx_Trigger_Arbitration	4
6	Mark4	1000100	Rx_Trigger	6
7	Mark6	n.a.	Tx_Ref_Trigger	0 (Ref)
8	Mark7	n.a.	Watch_Trigger	n.a.
9	n.a.	n.a.	End_of_List	n.a.

Tx_Trigger_Single, Tx_Trigger_Continous, Tx_Trigger_Merged, Tx_Trigger_Arbitration, Rx_Trigger, and Time_Base_Trigger are only valid for the specified cycle code. For all other trigger types the cycle code is ignored.

The FSE starts the basic cycle by scanning the trigger list starting from zero until a trigger with time mark that is greater than the cycle time is reached, CC matches the actual cycle count, or a trigger of type Tx_Ref_Trigger, Tx_Ref_Trigger_Gap, Watch_Trigger, or Watch_Trigger_Gap is encountered.

When the cycle time reaches TM, the action defined by TYPE and MNR is started. There is an error in the configuration when it reaches End_of_List.

At Mark6, the reference message (always TxRef) is transmitted. After transmission, the FSE returns to the beginning of the trigger list. When it reaches Watch Trigger at Mark7, the node is unable to transmit the reference message; error treatment is then started.

30.5.2.3.4 Detection of configuration errors

A configuration error is signaled via TTOST.EL = 11 (severity 3) when:

- The FSE comes to a trigger in the list with a cycle code that matches the current cycle count but with a time mark that is less than the cycle time
- The previous active trigger was a Tx_Trigger_Merged and the FSE comes to a trigger in the list with a cycle code that matches the current cycle count but that is neither a Tx_Trigger_Merged nor a Tx_Trigger_Arbitration nor a Time_Base_Trigger nor an Rx_Trigger
- The FSE of a node with TTOCF.TM = 0 (time slave) encounters a Tx_Ref_Trigger or a Tx_Ref_Trigger_Gap
- Any time mark placed inside the TX enable window (defined by TTMLM.TXEW) of a Tx_Trigger with a matching cycle code
- A time mark is placed near the time mark of a Tx_Ref_Trigger and the Reference Trigger Offset TTOST.RTO causes a reversal of their sequential order measured in cycle time

30.5.2.4 TTCAN schedule initialization

The synchronization to the M_TTCAN message schedule starts when CCCR.INIT is reset. The M_TTCAN can operate time-triggered (TTOCF.GEN = 0) or external event-synchronized time-triggered (TTOCF.GEN = 1). All nodes start with cycle time zero at the beginning of their trigger list with TTOST.SYS = 00 (out of synchronization); no transmission is enabled with the exception of the reference message. Nodes in external event-synchronized time-triggered operation mode will ignore Tx_Ref_Trigger and Watch_Trigger and use Tx_Ref_Trigger_Gap and Watch_Trigger_Gap instead until the first reference message decides whether a gap is active.

30.5.2.4.1 Time slaves

After configuration, a time slave will ignore its Watch_Trigger and Watch_Trigger_Gap when it does not receive any message before reaching the Watch_Triggers. When it reaches Init_Watch_Trigger, interrupt flag TTIR.IWT is set, the FSE is frozen, and the cycle time will become invalid. However, the node will still be able to take part in CAN bus communication (to give acknowledge or to send error flags). The first received reference message will restart the FSE and the cycle time.

Note: Init_Watch_Trigger is not part of the trigger list. It is implemented as an internal counter that counts up to 0xFFFF = maximum cycle time.

When a time slave receives any message but the reference message before reaching the Watch_Triggers, it will assume a fatal error (TTOST.EL = 11, severity 3), set interrupt flag TTIR.WT, switch off its CAN bus output, and enter the bus monitoring mode (CCCR.MON set to '1'). In the bus monitoring mode, it is still able to receive messages, but cannot send any dominant bits and therefore, cannot acknowledge.

Note: To leave the fatal error state, the host must set CCCR.INIT = 1. After reset of CCCR.INIT, the node restarts TTCAN communication.

When no error is encountered during synchronization, the first reference message sets TTOST.SYS = 01 (synchronizing), the second sets the TTCAN synchronization state (depending on its Next_is_Gap bit) to TTOST.SYS = 11 (In_Schedule) or TTOST.SYS = 10 (In_Gap), enabling all Tx_Triggers and Rx_Triggers.

30.5.2.4.2 Potential time master

After configuration, a potential time master will start the transmission of a reference message when it reaches its Tx_Ref_Trigger (or its Tx_Ref_Trigger_Gap when in external event-synchronized time-triggered operation). It will ignore its Watch_Trigger and Watch_Trigger_Gap when it does not receive any message or transmit the reference message successfully before reaching the Watch_Triggers (the reason assumed is that all other nodes still in reset or configuration and does not acknowledge). When it reaches Init_Watch_Trigger, the attempted transmission is aborted, interrupt flag TTIR.IWT is set, the FSE is frozen, and the cycle time will become invalid, but the node will still be able to take part in CAN bus communication (to acknowledge or send error flags). Resetting TTIR.IWT will re-enable the transmission of reference messages until the next time Init_Watch_Trigger condition is met, or another CAN message is received. The FSE will be restarted by the reception of a reference message.

When a potential time master reaches the Watch_Triggers after it has received any message but the reference message, it will assume a fatal error (TTOST.EL = 11, severity 3), set interrupt flag TTIR.WT, switch off its CAN bus output, and enter the bus monitoring mode (CCCR.MON set to ‘1’). In bus monitoring mode, it is still able to receive messages, but it cannot send any dominant bits and therefore, cannot acknowledge.

When no error is detected during initialization, the first reference message sets TTOST.SYS = 01 (synchronizing), the second sets the TTCAN synchronization state (depending on its Next_is_Gap bit) to TTOST.SYS = 11 (In_Schedule) or TTOST.SYS = 10 (In_Gap), enabling all Tx_Triggers and Rx_Triggers.

A potential time master is current time master (TTOST.MS = 11) when it is the transmitter of the last reference message; otherwise, it is the backup time master (TTOST.MS = 10).

When all potential time masters have finished configuration, the node with the highest time master priority in the network will become the current time master.

30.5.3 TTCAN gap control

All functions related to gap control apply only when the M_TTCAN is operated in external event-synchronized time-triggered mode (TTOCF.GEN = 1). In this operation mode the TTCAN message schedule may be interrupted by inserting gaps between the basic cycles of the system matrix. All nodes connected to the CAN network should be configured for external event-synchronized time-triggered operation.

During a gap, all transmissions are stopped and the CAN bus remains idle. A gap is finished when the next reference message starts a new basic cycle. The gap starts at the end of a basic cycle that was started by a reference message with bit Next_is_Gap = 1; for example, gaps are initiated by the current time master.

The current time master has two options to initiate a gap. A gap can be initiated under software control when the application program writes TTOCN.NIG = 1. The Next_is_Gap bit will be transmitted as '1' with the next reference message. A gap can also be initiated under hardware control when the application program writes TTOCN.GCS = 1. When a reference message is started and TTOCN.GCS is set, Next_is_Gap = 1 will be set.

As soon as that reference message is completed, the TTOST.WFE bit will announce the gap to the time master and slaves. The current basic cycle will continue until its last time window. The time after the last time window is the gap time.

For the actual time master and the potential time masters, TTOST.GSI will be set when the last basic cycle has finished and the gap time starts. In nodes that are time slaves, the TTOST.GSI bit will remain at '0'.

When a potential time master is in synchronization state In_Gap (TTOST.SYS = 10), it has four options to intentionally finish a gap:

- Under software control by writing TTOCN.FGP = 1
- Under hardware control (TTOCN.GCS = 1), TTOCN.FGP will automatically be set when an edge from HIGH to LOW at the internal event trigger input pin is detected and restarts the schedule.
- The third option is a time-triggered restart. When TTOCN.TMG = 1, the next register time mark interrupt (TTIR.RTMI = 1) will set TTOCN.FGP and start the reference message.
- Any potential time master will finish a gap when it reaches its Tx_Ref_Trigger_Gap, assuming that the event to synchronize to did not occur on time

None of these options can cause a basic cycle to be interrupted with a reference message.

Setting TTOCN.FGP after the gap time has started will start the transmission of a reference message immediately and will thereby synchronize the message schedule. When TTOCN.FGP is set before the gap time has started (while the basic cycle is still in progress), the next reference message is started at the end of the basic cycle, at the Tx_Ref_Trigger – there will be no gap time in the message schedule.

In time-triggered operation, bit Next_is_Gap = 1 in the reference message will be ignored, and the bits TTOCN.NIG, TTOCN.FGP, and TTOCN.TMG will be considered.

30.5.4 Stop watch

The stop watch function enables capturing of M_TTCAN internal time values (local time, cycle time, or global time) triggered by an external event.

To enable the stop watch function, the application program must first define local time, cycle time, or global time as stop watch source via TTOCN.SWS. When TTOCN.SWS is not equal to '00' and TT Interrupt Register flag TTIR.SWE is '0', the actual value of the time selected by TTOCN.SWS will be copied into TTCPT.SWV on the next rising/falling edge (as configured via TTOCN.SWP) on Stop Watch Trigger. This will set interrupt flag TTIR.SWE. After the application program has read TTCPT.SWV, it may enable the next stop watch event by resetting TTIR.SWE to '0'.

30.5.5 Local time, cycle time, global time, and external clock synchronization

There are two possible levels in time-triggered CAN: Level 1 and Level 2. Level 1 provides only time-triggered operation using cycle time. Level 2 additionally provides increased synchronization quality, global time, and external clock synchronization. In both levels, all timing features are based on a local time base – the local time.

The local time is a 16-bit cyclic counter, it is incremented once each NTU. Internally the NTU is represented by a 3-bit counter, which can be regarded as a fractional part (three binary digits) of the local time. Generally, the 3-bit NTU counter is incremented eight times each NTU. If the length of the NTU is shorter than eight CAN clock periods (as may be configured in Level 1, or as a result of clock calibration in Level 2), the length of the NTU fraction is adapted, and the NTU counter is incremented only four times each NTU.

Figure 30-22 describes the synchronization of the cycle time and global time, performed in the same manner by all TTCAN nodes, including the time master. Any message received or transmitted invokes a capture of the local time taken at the message's frame synchronization event. This frame synchronization event occurs at the sample point of each Start-of-Frame (SoF) bit and causes the local time to be stored as Sync_Mark. Sync_Marks and Ref_Marks are captured including the 3-bit fractional part.

Whenever a valid reference message is transmitted or received, the internal Ref_Mark is updated from the Sync_Mark. The difference between Ref_Mark and Sync_Mark is the Cycle Sync Mark (Cycle Sync Mark = Sync_Mark – Ref_Mark) stored in register TTCSM. The most significant 16 bits of the difference between Ref_Mark and the actual value of the local time is the cycle time (Cycle Time = Local Time – Ref_Mark).

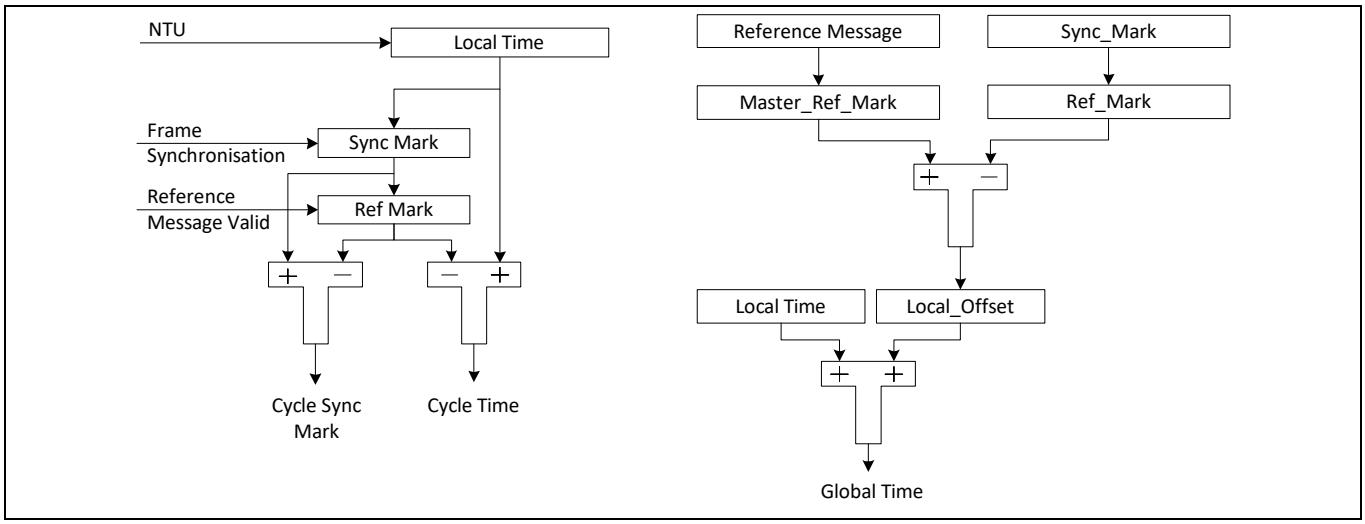


Figure 30-22. Cycle time and global time synchronization

The cycle time that can be read from TTCTC.CT is the difference of the node's local time and Ref_Mark, both synchronized into the host clock domain and truncated to 16 bits.

The global time exists for TTCAN Level 0 and Level 2 only, in Level 1 it is invalid. The node's view of the global time is the local image of the global time in (local) NTUs. After configuration, a potential time master will use its own local time as global time. This is done by transmitting its own Ref_Marks as Master_Ref_Marks in the reference message (bytes 3 and 4). The global time that can be read from TTLGT.GT is the sum of the node's local time and its local offset, both synchronized into the host clock domain and truncated to 16 bit. The fractional part is used for clock synchronization only.

A node that receives a reference message calculates its local offset to the global time by comparing its local Ref_Mark with the received Master_Ref_Mark (see **Figure 30-22**). The node's view of the global time is local time + local offset. In a potential time master that has never received another time master's reference message, Local_Offset will be zero. When a node becomes the current time master after having received other reference messages first, Local_Offset will be frozen at its last value. In the time receiving nodes, Local_Offset may be subject to small adjustments, due to clock drift, when another node becomes time master, or when there is a global time discontinuity, signaled by Disc_Bit in the reference message. With the exception of global time discontinuity, the global time provided to the application program by register TTLGT is smoothed by a low-pass filtering to have a continuous monotonic value.

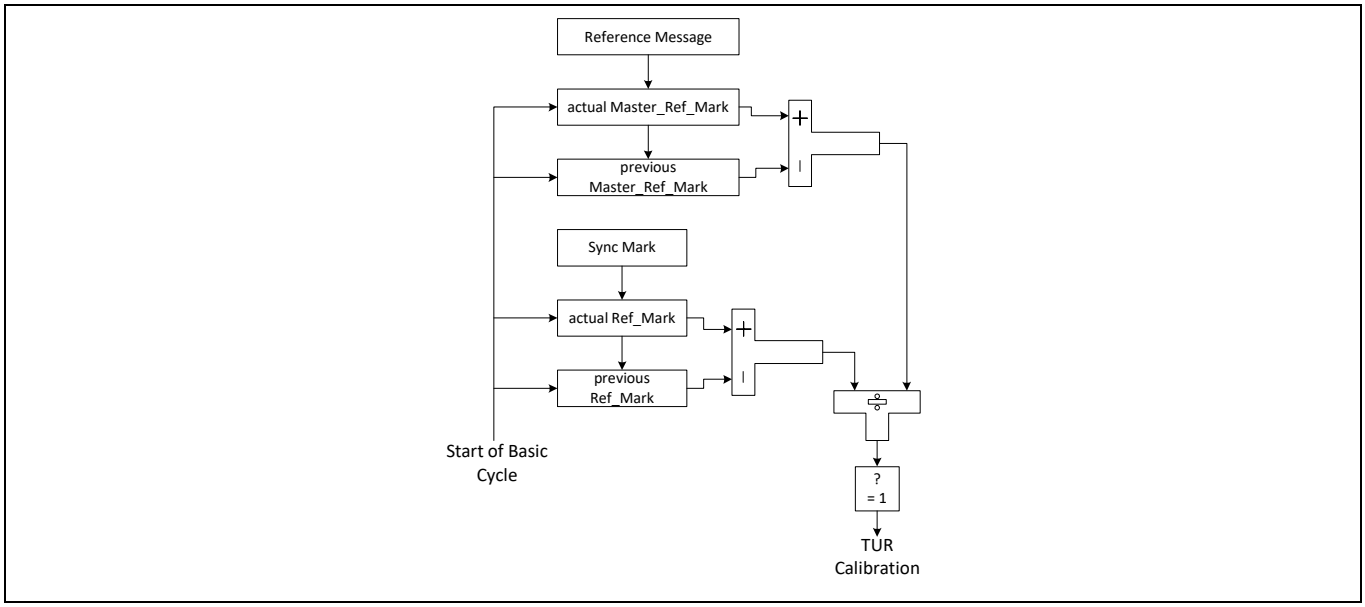


Figure 30-23. TTCAN Level 0 and Level 2 drift compensation

Figure 30-23 illustrates how in TTCAN Level 0 and Level 2 the receiving node compensates the drift between its own local clock and the time master's clock by comparing the length of a basic cycle in local time and in global time. If there is a difference between the two values, and the Disc_Bit in the reference message is not set, a new value for TURNA.NAV is calculated. If the synchronization deviation (SD) = |NC – TURNA.NAV| ≤ SDL, the new value for TURNA.NAV takes effect. Otherwise, the automatic drift compensation is suspended.

In TTCAN Level 0 and Level 2, TTOST.QCS indicates whether the automatic drift compensation is active or suspended. In TTCAN Level 1, TTOST.QCS is always '1'.

The current time master may synchronize its local clock speed and the global time phase to an external clock source. This is enabled by bit TTOCF.EECS.

The stop watch function (see **"Stop watch"** on page 575) may be used to measure the difference in clock speed between the local clock and the external clock. The local clock speed is adjusted by first writing the newly calculated Numerator Configuration Low to TURCF.NCL (TURCF.DC cannot be updated during operation). The new value takes effect by writing TTOCN.ECS to '1'.

The global time phase is adjusted by first writing the phase offset into the TT Global Time Preset register (TTGTP). The new value takes effect by writing TTOCN.SGT to '1'. The first reference message transmitted after the global time phase adjustment will have the Disc_Bit set to '1'.

TTOST.QGTP shows whether the node's global time is in phase with the time master's global time. TTOST.QGTP is permanently '0' in TTCAN Level 1 and when the SDL is exceeded in TTCAN Level 0,2 (TTOST.QCS = 0). It is temporarily '0' while the global time is low-pass filtered to supply the application with a continuous monotonic value. There is no low-pass filtering when the last reference message contains a Disc_Bit = 1 or when TTOST.QCS = 0.

30.5.6 TTCAN error level

The ISO 11898-4 specifies four levels of error severity:

- 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 several errors are detected at the same time, the highest severity prevails. When an error is detected, the application is notified by TTIR.ELC. The error level is monitored by TTOST.EL.

The M_TTCAN signals the following error conditions as required by ISO 11898-4:

- **Config_Error (S3)**

  Sets error level TTOST.EL to ‘11’ when a merged arbitrating time window is not properly closed or when there is a Tx_Trigger with a time mark beyond the Tx_Ref_Trigger.
- **Watch_Trigger_Reached (S3)**

  Sets error level TTOST.EL to ‘11’ when a watch trigger is reached because the reference message is missing.
- **Application_Watchdog (S3)**

  Sets error level TTOST.EL to ‘11’ when the application fails to serve the application watchdog. The application watchdog is configured via TTOCF.AWL. It is served by reading the TTOST register. When the watchdog is not served in time, bit TTOST.AWE and interrupt flag TTIR.AW are set, all TTCAN communication is stopped, and the M_TTCAN is set into bus monitoring mode (CCCR.MON set to '1').
- **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.

30.5.7 TTCAN message handling

30.5.7.1 Referencemessage

For potential time masters, the identifier of the reference message is configured via TTRMC.RID. No dedicated TX buffer is required for transmission of the reference message. When a reference message is transmitted, the first data byte (TTCAN Level 1) and the first four data bytes (TTCAN Level 0 and Level 2) will be provided by the FSE.

If the Payload Select reference message TTRMC.RMPS is set, the rest of the reference message's payload (Level 1: bytes 2-8, Level 0 and Level 2: bytes 5-6) is taken from TX Buffer 0. In this case, the data length DLC code from message buffer 0 is used.

Table 30-14. Number of data bytes transmitted with a reference messages

TTRMC.RMPS	TXBRP.TRP0	Level 0	Level 1	Level 2
0	0	4	1	4
0	1	4	1	4
1	0	4	1	4
1	1	4 + MB0	1 + MB0	4 + MB0

To send additional payload with the reference message in Level 1, a DLC > 1 should be configured. For Level 0 and Level 2 a DLC > 4 is required. In addition, the transmission request pending bit TXBRP.TRP0 of message buffer 0 must be set (see **Table 30-14**). If TXBRP.TRP0 is not set when a reference message is started, the reference message is transmitted with the data bytes supplied by the FSE only.

For acceptance filtering of reference messages the Reference Identifier TTRMC.RID is used.

30.5.7.2 Message reception

Message reception is done via the two RX FIFOs in the same way as for event-driven CAN communication.

The MSC is part of the corresponding trigger memory element and must be initialized to zero during configuration. It is updated while the M_TTCAN is in synchronization states In_Gap or In_Schedule. The update happens at the message's Rx_Trigger. At this point, it is checked at which acceptance filter element the latest message received in this basic cycle is matched. The matching filter number is stored as the acceptance filter result. If this is the same as the filter number defined in this trigger memory element, the MSC is decremented by one. If the acceptance filter result is not the same filter number as defined for this filter element, or if the acceptance filter result is cleared, the MSC is incremented by one. At each Rx_Trigger and at each start of cycle, the last acceptance filter result is cleared.

The time mark of an Rx_Trigger should be set to a value that ensures reception and acceptance filtering for the targeted message is completed. This should consider the RAM access time and the order of the filter list. It is recommended, that filters used for Rx_Triggers are placed at the beginning of the filter list. It is not recommended to use an Rx_Trigger for the reference message.

30.5.7.3 Message transmission

For time-triggered message transmission, the M_TTCAN supplies 32 dedicated TX buffers (see **"TTCAN configuration"** on page 566). A TX FIFO or TX queue is not available when the M_TTCAN is configured for time-triggered operation (TTOCF.OM = 01 or 10).

Each Tx_Trigger in the trigger memory points to a particular TX buffer containing a specific message. There may be more than one Tx_Trigger for a given TX buffer if that TX buffer contains a message that is to be transmitted more than once in a basic cycle or matrix cycle.

The application program must update the data regularly and on time, synchronized to the cycle time. The host CPU should ensure that no partially updated messages are transmitted. To assure this the host should proceed in the following way:

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 and/or payload
- Issue an Add Request to set the TX Buffer Request Pending bit

Tx_Trigger_Continous:

- Issue a Cancellation Request to reset the TX Buffer Request Pending bit
- Check whether the cancellation has finished by reading TXBCF
- Update TX buffer configuration and/or payload
- Issue an Add Request to set the TX Buffer Request Pending bit

The message MSC stored with the corresponding Tx_Trigger provides information on the success of the transmission.

The MSC is incremented by one when the transmission cannot be started because the CAN bus was not idle within the corresponding transmit enable window or when the message was started but could not be completed successfully. The MSC is decremented by one when the message is transmitted successfully or when the message could have been started within its transmit enable window but was not started because transmission was disabled (M_TTCAN in Error Level S2 or host has disabled this particular message).

The TX buffers may be managed dynamically – several messages with different identifiers may share the same TX buffer element. In this case the host must ensure that no transmission request is pending for the TX buffer element to be reconfigured by checking TXBRP.

If a TX buffer with pending transmission request should be updated, the host must first issue a cancellation request and check whether the cancellation has completed by reading TXBCF before it starts updating.

The TX handler will transfer a message from the message RAM to its intermediate output buffer at the trigger element, which becomes active immediately before the Tx_Trigger element that defines the beginning of the transmit window. During and after transfer time, the transmit message may not be updated and its TXBRP bit may not be changed. To control this transfer time, an additional trigger element may be placed before the Tx_Trigger. An example is a Time_Base_Trigger, which does not cause any other action. The difference in time marks between the Tx_Trigger and the preceding trigger should be large enough to guarantee that the TX handler can read four words from the message RAM even at high RAM access load from other modules.

30.5.7.3.1 Transmission in Exclusive Time Windows

A transmission is started time-triggered when the cycle time reaches the time mark of a Tx_Trigger_Single or Tx_Trigger_Continous. There is no arbitration on the bus with messages from other nodes. The MSC is updated according the result of the transmission attempt. After successful transmission started by a Tx_Trigger_Single, the respective TX Buffer Request Pending bit is reset. After successful transmission started by a Tx_Trigger_Continous the respective TX Buffer Request Pending bit remains set. When the transmission is not successful due to disturbances, it will be repeated the next time one of its Tx_Triggers becomes active.

30.5.7.3.2 Transmission in Arbitrating Time Windows

A transmission is started time-triggered when the cycle time reaches the time mark of a Tx_Trigger_Arbitration. Several nodes may start to transmit at the same time. In this case the message has to arbitrate with the messages from other nodes. The MSC is not updated. When the transmission is not successful (lost arbitration or disturbance), it will be repeated the next time one of its Tx_Triggers becomes active.

30.5.7.3.3 Transmission in Merged Arbitrating Time Windows

The purpose of a merged arbitrating time window is to enable multiple nodes to send a limited number of frames, which are transmitted in immediate sequence, the order given by CAN arbitration. It is not intended for burst transmission by a single node. Because the node does not have exclusive access within this time window, all requested transmissions may not be successful.

Messages that have lost arbitration or were disturbed by an error, may be retransmitted inside the same merged arbitrating time window. The retransmission will not be started if the corresponding Transmission Request Pending flag was reset by a successful TX cancellation.

In single transmit windows, the TX handler transmits the message indicated by the message number of the trigger element. In merged arbitrating time windows, it can handle up to three message numbers from the trigger list. Their transmissions will be attempted in the sequence defined by the trigger list. If the time mark of a fourth message is read before the first is transmitted (or canceled by the host), the fourth request will be ignored.

The transmission inside a merged arbitrating time window is not time-triggered. The transmission of a message may start before its time mark, or after the time mark if the bus was not idle.

The messages transmitted by a specific node inside a merged arbitrating time window will be started in the order of their Tx_Triggers. Therefore, a message with low CAN priority may prevent the successful transmission of a following message with higher priority, if there is competing bus traffic. This should be considered for the configuration of the trigger list. Time_Base_Triggers may be placed between consecutive Tx_Triggers to define the time until the data of the corresponding TX buffer needs to be updated.

30.5.8 TTCAN interrupt and error handling

The TT Interrupt Register TTIR consists of four segments. Each interrupt can be enabled separately by the corresponding bit in the TT Interrupt Enable register TTIE. The flags remain set until the host clears them. A flag is cleared by writing a ‘1’ to the corresponding bit position.

The first segment consists of flags CER, AW, WT, and IWT. Each flag indicates a fatal error condition where the CAN communication is stopped. With the exception of IWT, these error conditions require a reconfiguration of the M_TTCAN module before the communication can be restarted.

The second segment consists of flags ELC, SE1, SE2, TXO, TXU, and GTE. Each flag indicates an error condition where the CAN communication is disturbed. If they are caused by a transient failure, such as by disturbances on the CAN bus, they will be handled by the TTCAN protocol's failure handling and do not require intervention by the application program.

The third segment consists of flags GTD, GTW, SWE, TTMI, and RTMI. The first two flags are controlled by global time events (Level 0 and Level 2 only) that require a reaction by the application program. With a Stop Watch Event, internal time values are captured. The Trigger Time Mark Interrupt notifies the application that a specific Time_Base_Trigger is reached. The Register Time Mark Interrupt signals that the time referenced by TTOCN.TMC (cycle, local, or global) equals time mark TTTMK.TM. It can also be used to finish a gap.

The fourth segment consists of flags SOG, CSM, SMC, and SBC. These flags provide a means to synchronize the application program to the communication schedule.

30.5.9 Level 0

TTCAN Level 0 is not part of ISO11898-4. This operation mode makes the hardware, that in TTCAN Level 2 maintains the calibrated global time base, also available for event-driven CAN according to ISO 11898-1:2015.

Level 0 operation is configured via TTOCF.OM = 11. In this mode, M_TTCAN operates in event-driven CAN communication; there is no fixed schedule, the configuration of TTOCF.GEN is ignored. External event-synchronized operation is not available in Level 0. A synchronized time base is maintained by transmission of reference messages.

In Level 0 the trigger memory is not active and need not be configured. The time mark interrupt flag (TTIR.TTMI) is set when the cycle time has reached TTOCF.IRTO × 0x200. It reminds the host to set a transmission request for message buffer 0. The Watch_Trigger interrupt flag (TTIR.WT) is set when the cycle time has reached 0xFF00. These values were chosen to have enough margin for a stable clock calibration. There are no further TT-error-checks.

Register time mark interrupts (TTIR.RTMI) are also possible.

The reference message is configured as for Level 2 operation. Received reference messages are recognized by the identifier configured in register TTRMC. For the transmission of reference messages only message buffer 0 may be used. The node transmits reference messages any time the host sets a transmission request for message buffer 0; there is no reference trigger offset.

Level 0 operation is configured via:

- TTRMC
- TTOCF except EVTP, AWL, GEN
- TTMLM except ENTT, TXEW
- TURCF

Level 0 operation is controlled via:

- TTOCN except NIG, TMG, FGP, GCS, TTMIE
- TTGTP
- TTTMK
- TTIR excluding bits CER, AW, IWT SE2, SE1, TXO, TXU, SOG (no function)
- TTIR – the following bits have changed function:
  - TTMI not defined by trigger memory - activated at cycle time TTOCF.IRTO × 0x200
  - WT not defined by trigger memory - activated at cycle time 0xFF00

Level 0 operation is signaled via:

- TTOST excluding bits AWE, WFE, GSI, GFI, RTO (no function)

30.5.9.1 Synchronizing

Figure 30-24 describes the states and state transitions in TTCAN Level 0 operation. Level 0 has no In_Gap state.

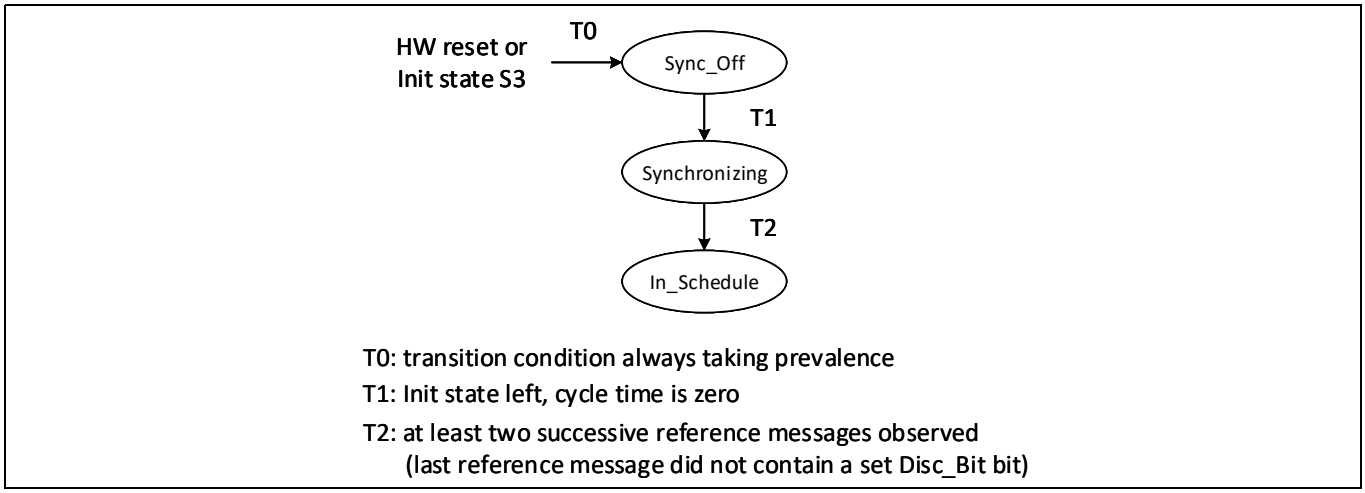


Figure 30-24. Level 0 schedule synchronization state machine

30.5.9.2 Handling error levels

During Level 0 operation only the following error conditions may occur:

- Watch_Trigger_Reached (S3), reached cycle time 0xFF00
- CAN_Bus_Off (S3)

Because S1 and S2 errors are not possible, the error level can only switch between S0 (No Error) and S3 (Severe Error). In TTCAN Level 0 an S3 error is handled differently. When S3 error is reached, both TTOST.SYS and TTOST.MS are reset, and interrupt flags TTIR.GTE and TTIR.GTD are set.

When S3 (TTOST.EL = 11) is entered, bus monitoring mode is, contrary to TTCAN Level 1 and Level 2, not entered. S3 error level is left automatically after transmission (time master) or reception (time slave) of the next reference message.

30.5.9.3 Master slave relation

Figure 30-25 describes the master slave relation in TTCAN Level 0. In case of an S3 error, the M_TTCAN returns to state Master_Off.

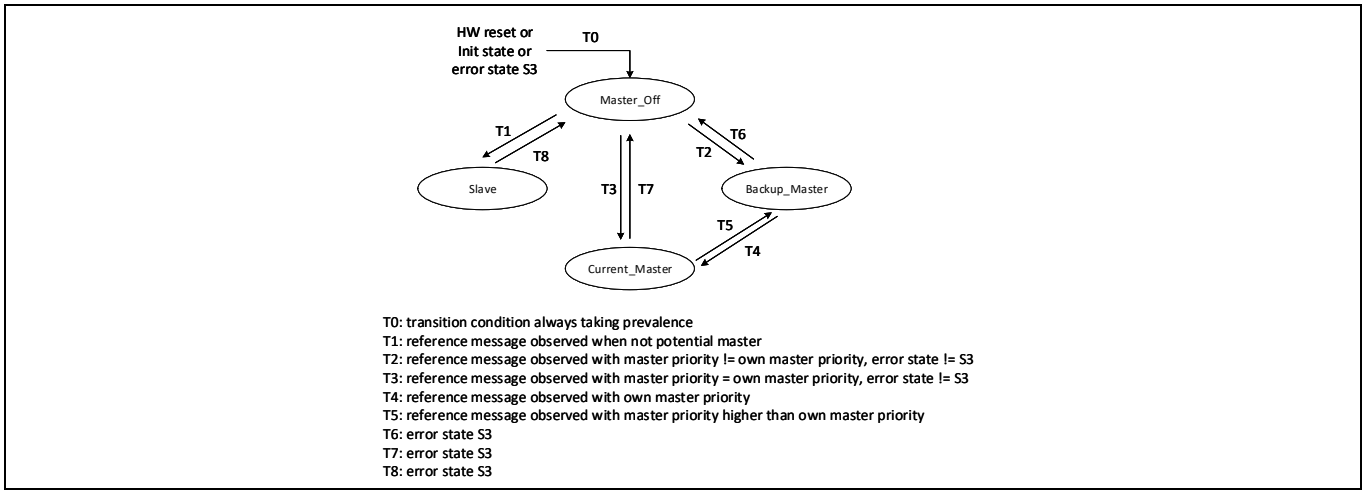


Figure 30-25. Level 0 master to slave relation

30.5.10 Synchronization to external time schedule

This feature can be used to synchronize the phase of the M_TTCAN's schedule to an external schedule (for example, that of a second TTCAN network). It is applicable only when the M_TTCAN is current time master (TTOST.MS = 11).

External synchronization is controlled by the TTOCN.ESCN bit. If TTOCN.ESCN is set, at rising edge of the internal event trigger pin, the M_TTCAN compares its actual cycle time with the target phase value configured by TTGTP.CTP.

Before setting TTOCN.ESCN, the host should adapt the phases of the two time schedules, for example, by using the TTCAN gap control (see **“TTCAN gap control”** on page 574). When the host sets TTOCN.ESCN, TTOST.SPL is set.

If the difference between the cycle time and target phase value TTGTP.CTP at the trigger is greater than 9 NTU, the phase lock bit TTOST.SPL is reset, and interrupt flag TTIR.CSM is set. TTOST.SPL is also reset (and TTIR.CSM is set), when another node becomes time master.

If both TTOST.SPL and TTOCN.ESCN are set, and if the difference between the cycle time and the target phase value TTGTP.CTP is less or equal 9 NTU, the phase lock bit TTOST.SPL remains set, and the measured difference is used as reference trigger offset value to adjust the phase at the next transmitted reference message.

Note: *The rising edge detection at the internal pin is enabled at the start of each basic cycle. The first rising edge triggers the compare of the actual cycle time with TTGTP.CTP. All further edges until the beginning of the next basic cycle are ignored.*

30.6 Setup procedures

This section provides example procedures for configurations of the M_TTCAN channel.

30.6.1 General program flow

This is a general flow to configure the M_TTCAN module.

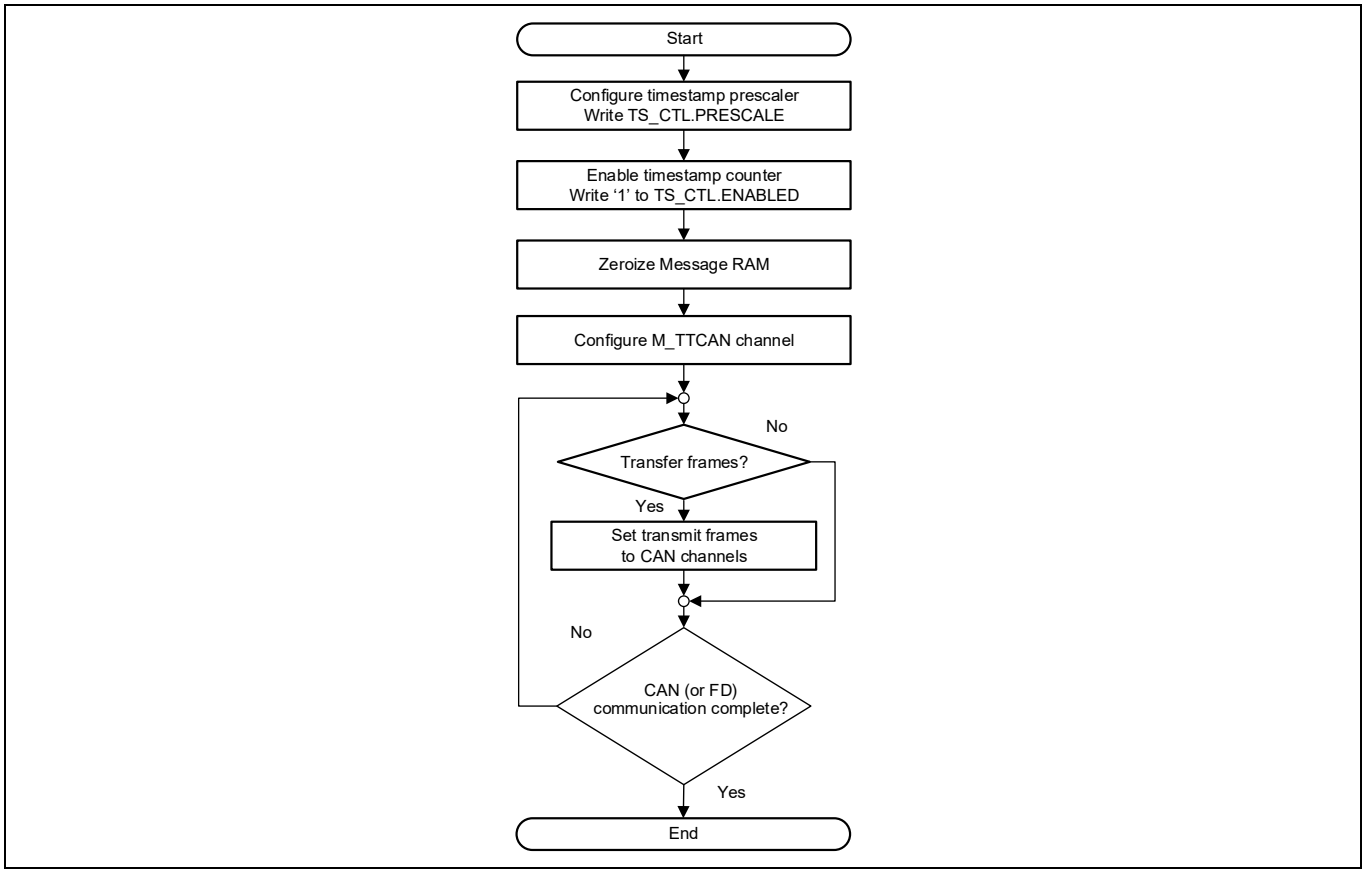


Figure 30-26. General program flow

30.6.2 Clock stop request

To save power, the application can stop providing clock to the unused M_TTCAN channel by following these steps.

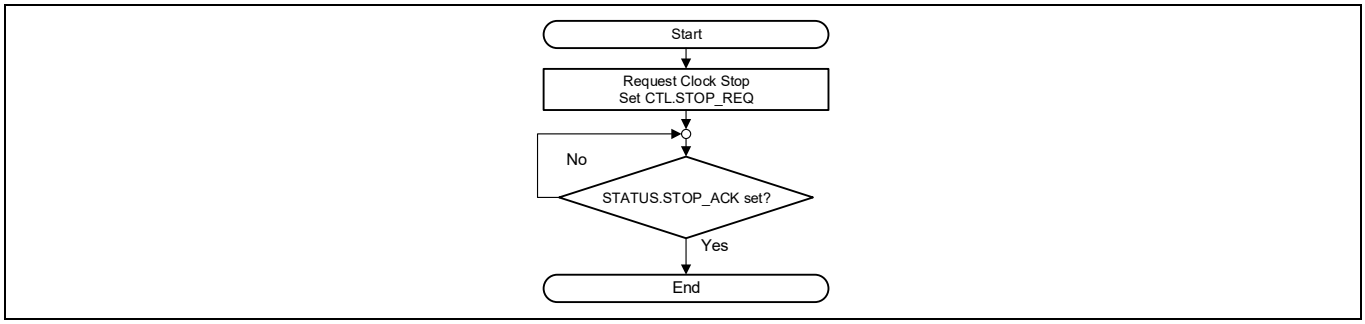


Figure 30-27. Clock stop request procedure

To resume providing clock, the CTL.STOP_REQ bit should be reset.

30.6.3 MRAM OFF operation

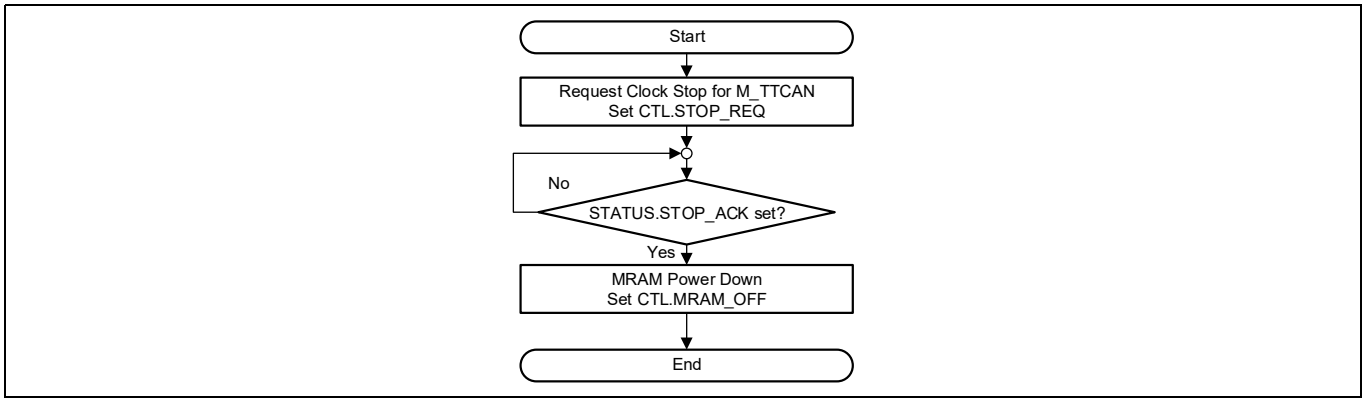


Figure 30-28. Message RAM OFF operation

30.6.4 MRAM ON operation

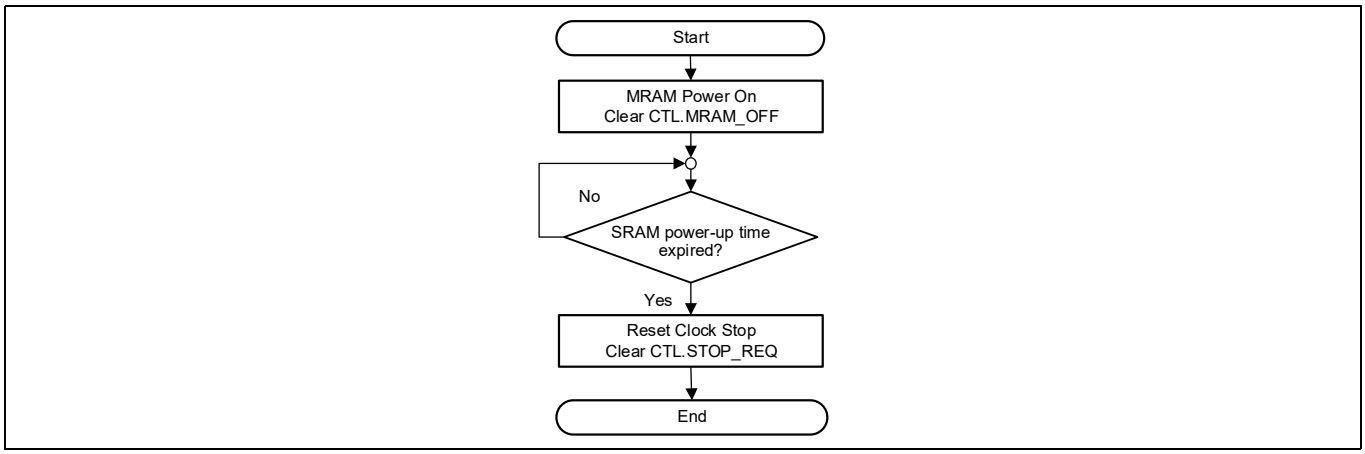


Figure 30-29. Message RAM on procedure

After switching MRAM ON again, software needs to allow a certain power-up time before MRAM can be used; that is, before STOP_REQ can be de-asserted. The power-up time is equivalent to the system SRAM power-up time specified in the CPUSS.RAM_PWR_DELAY_CTL register.

30.6.5 Procedures specific to M_TTCAN channel

This section describes sample procedure for the M_TTCAN channel. **Figure 30-30** shows the general program flow.

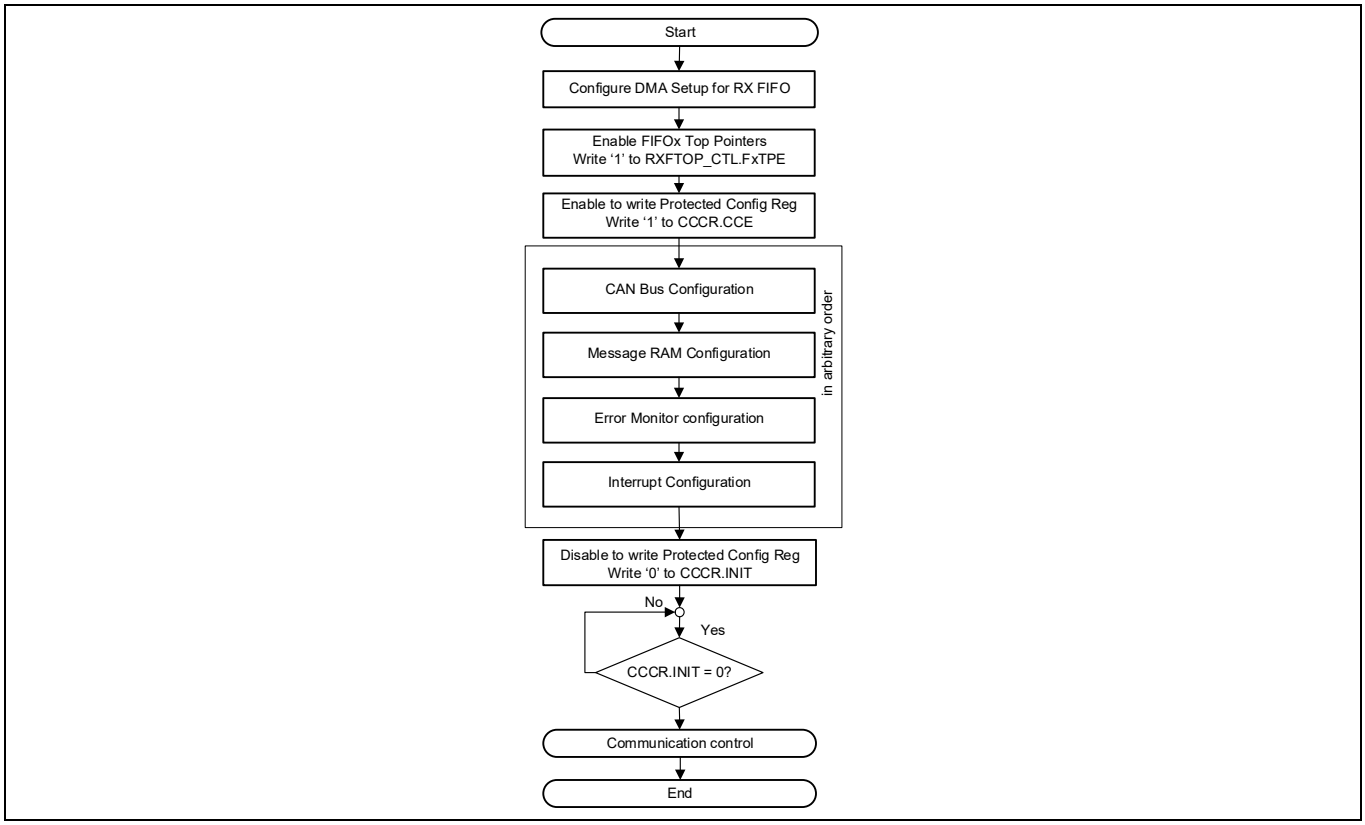


Figure 30-30. Configuration sequence specific to channel

30.6.5.1 CAN bus configuration

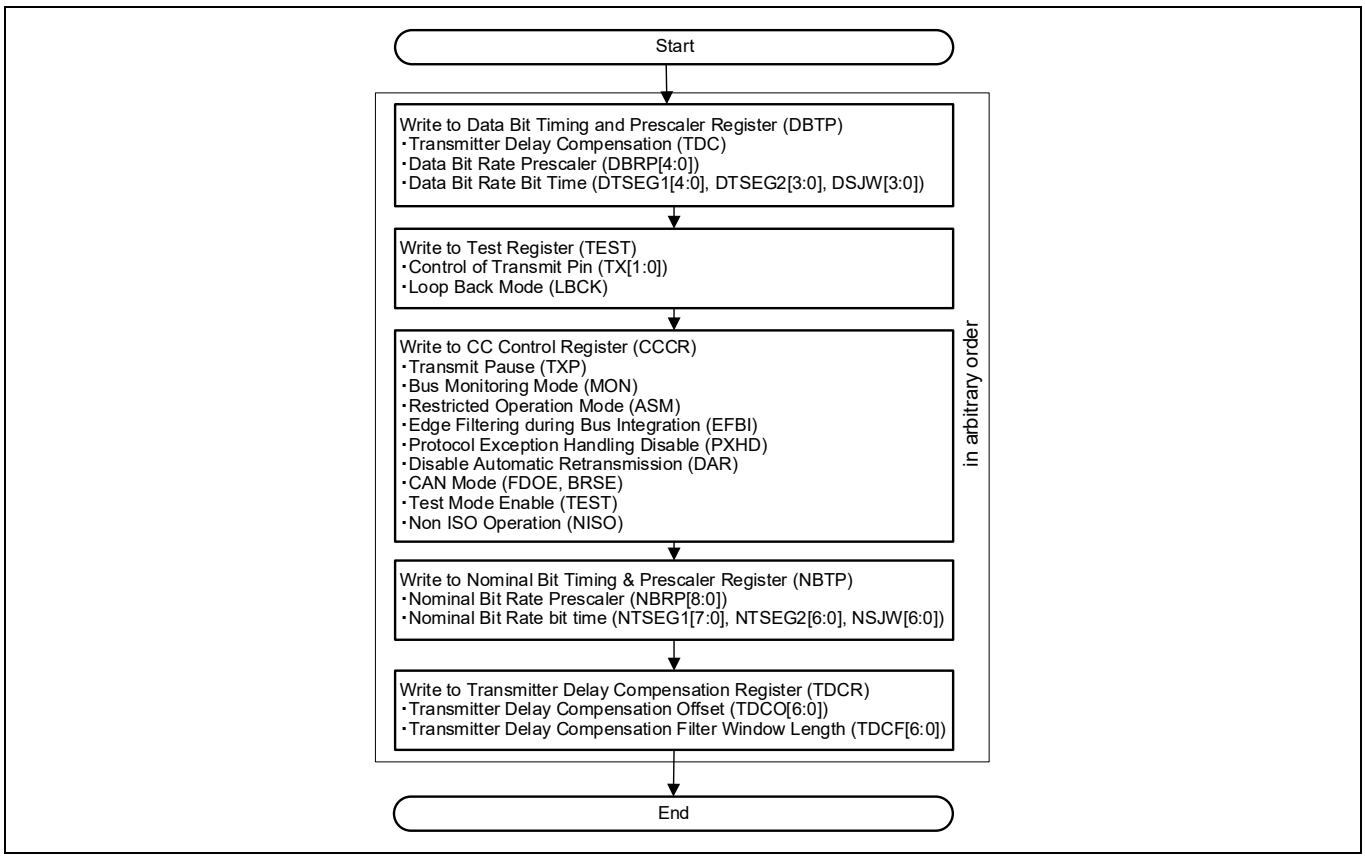


Figure 30-31. Configuration required for CAN bus

30.6.5.2 Message RAM configuration

Figure 30-32 shows an overview of the message RAM configuration.

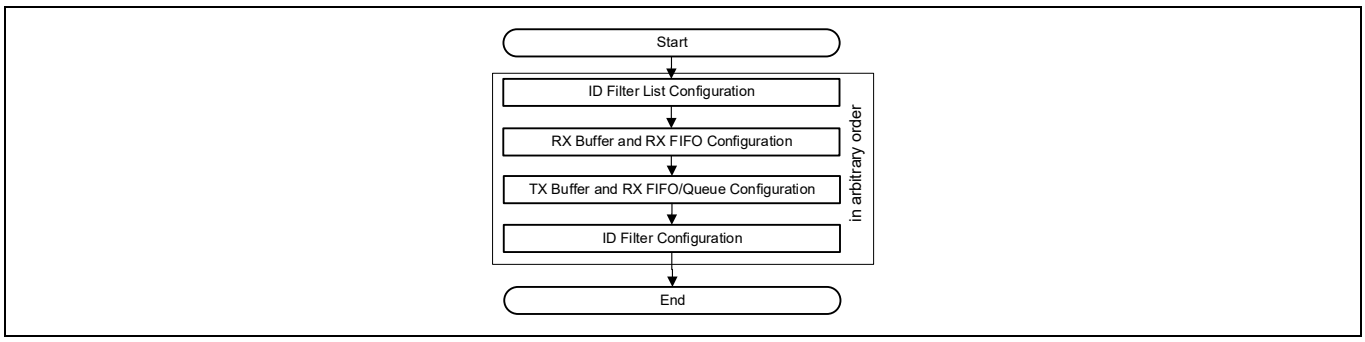


Figure 30-32. Message RAM configuration overview

CAN FD controller

Each configuration mentioned in the overview is detailed in **Figure 30-33** through **Figure 30-36**.

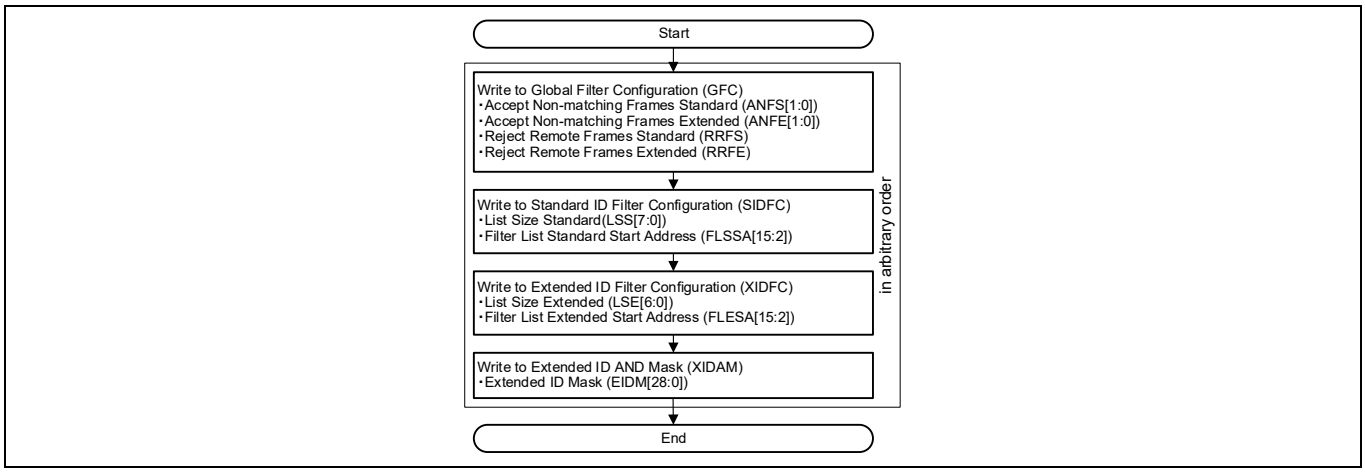


Figure 30-33. ID filter list configuration

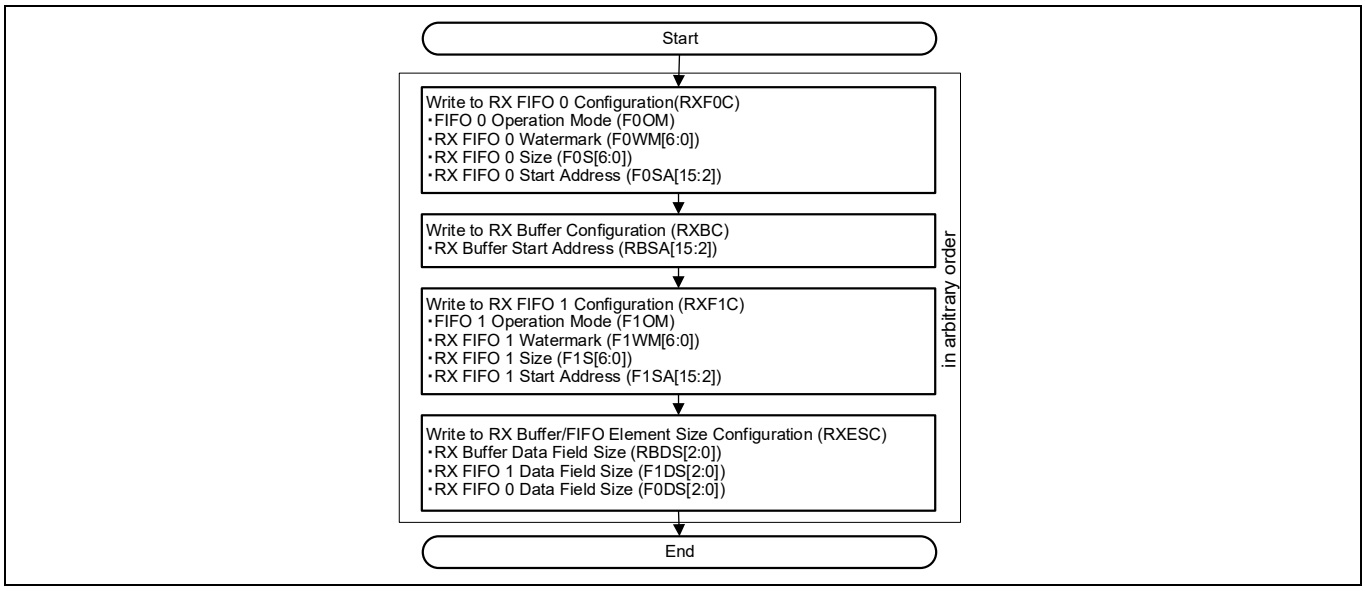


Figure 30-34. RX FIFO and RX buffer configuration

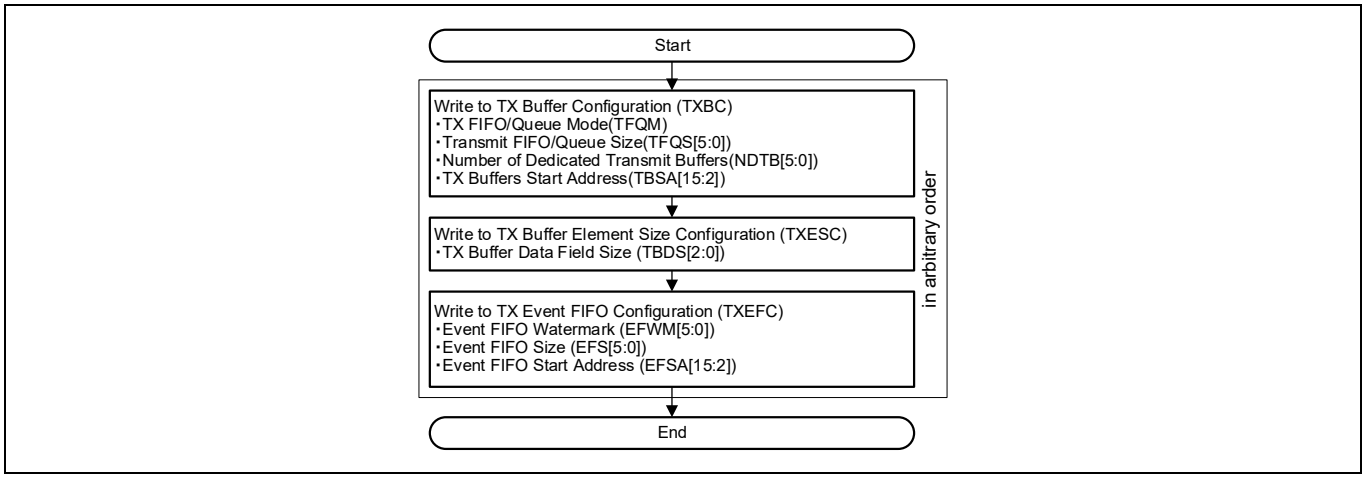


Figure 30-35. TX buffer and TX FIFO/queue configuration

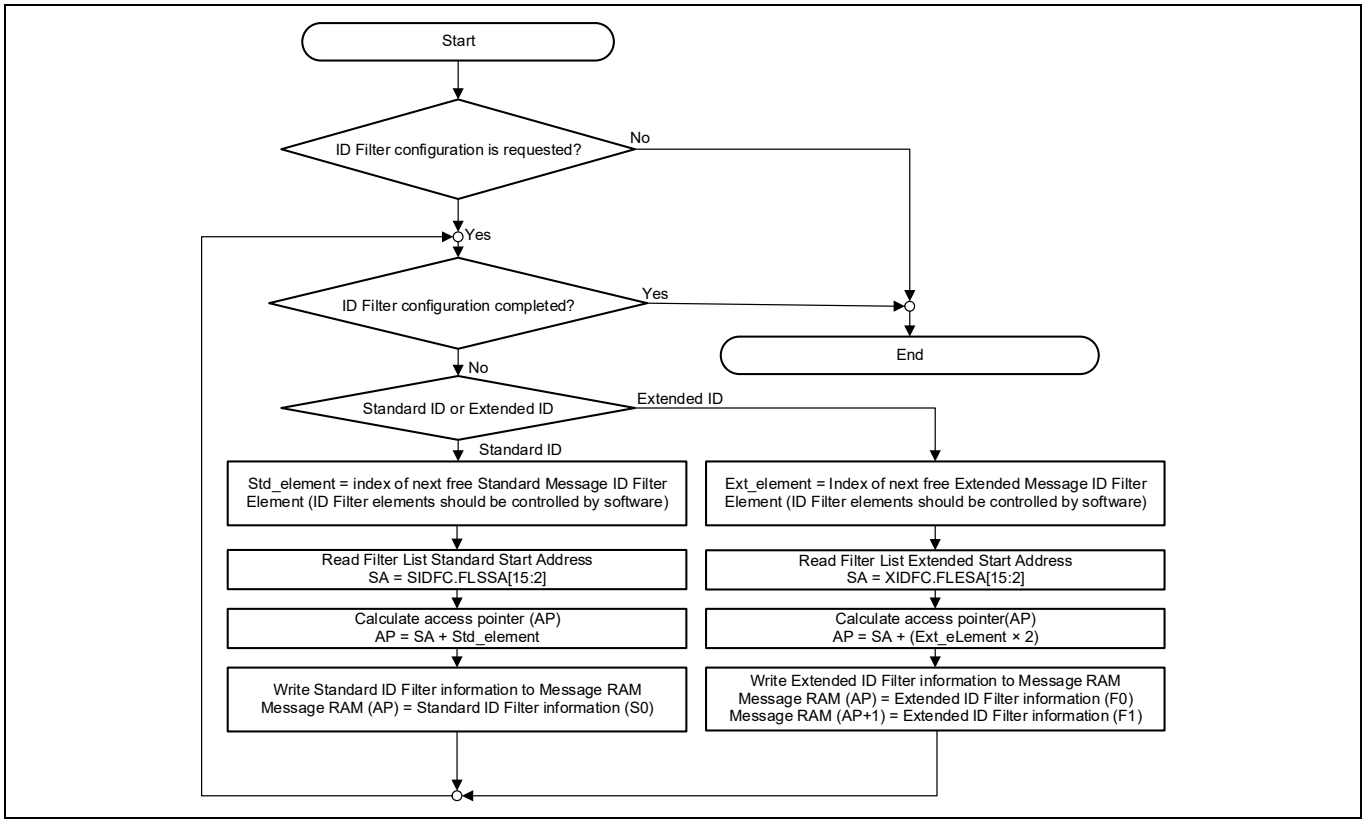


Figure 30-36. ID filter configuration

30.6.5.3 Interrupt Configuration

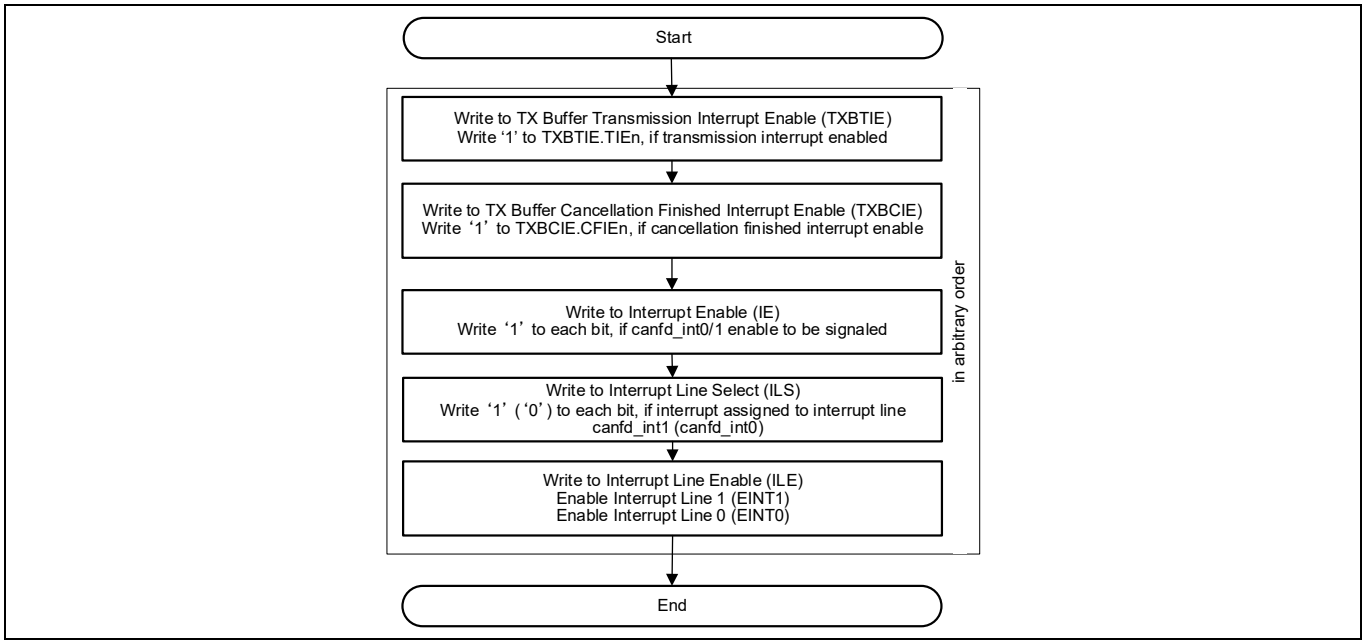


Figure 30-37. Interrupt configuration

30.6.5.4 Transmit frame configuration

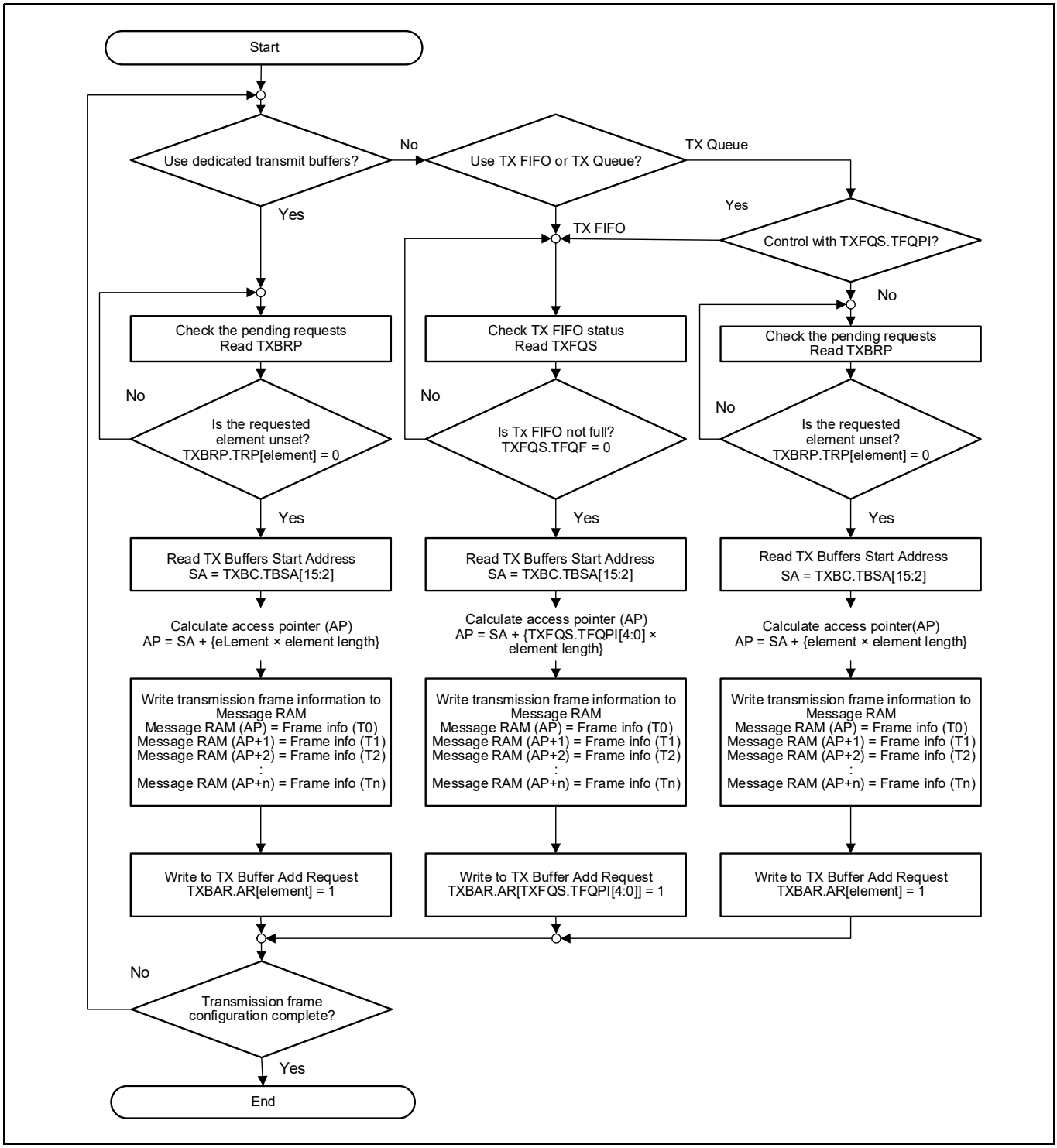


Figure 30-38. Transmit frame configuration

30.6.5.5 Interrupt handling

Figure 30-39 shows a general interrupt handling flow chart.

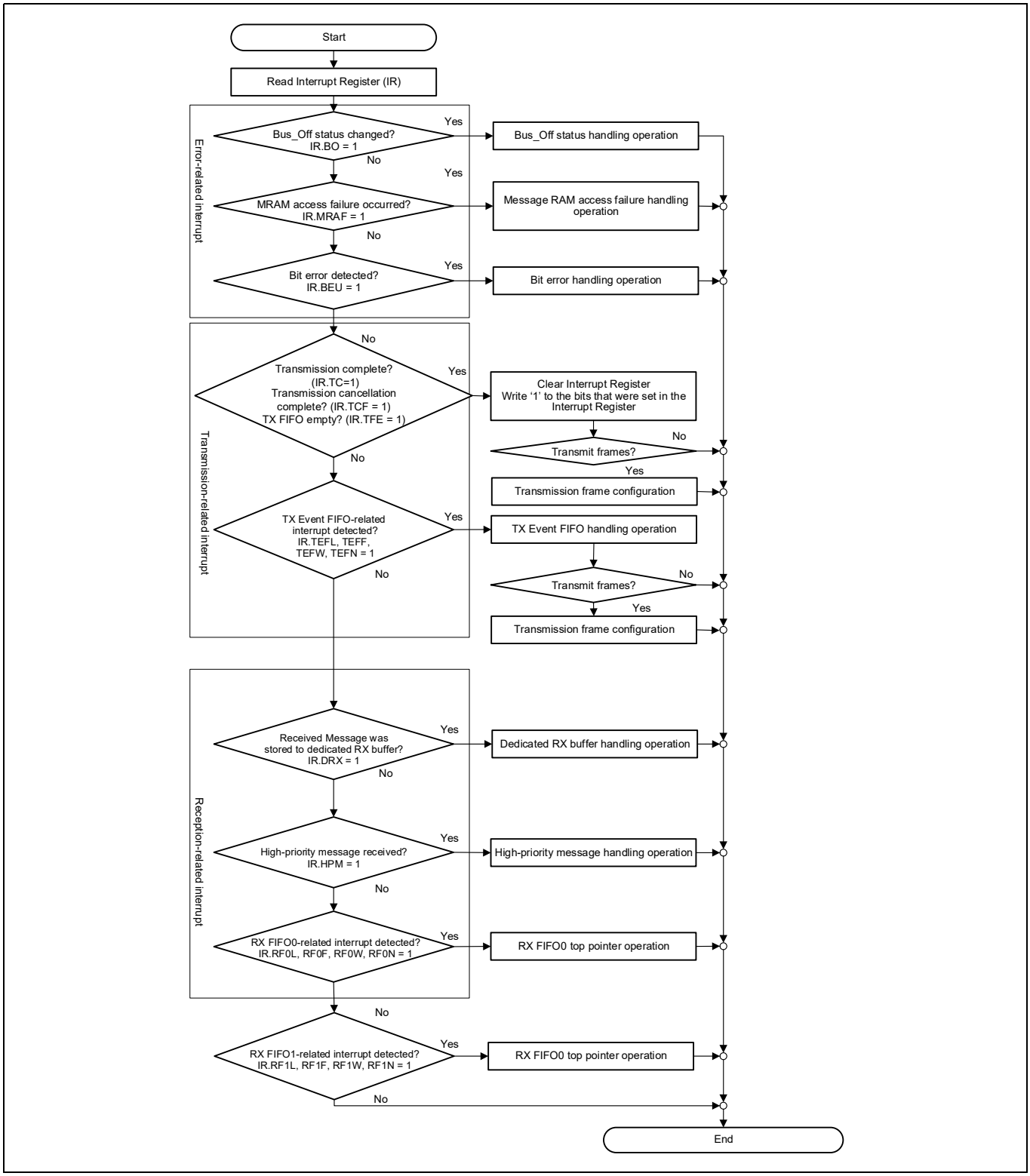


Figure 30-39. Interrupt handling

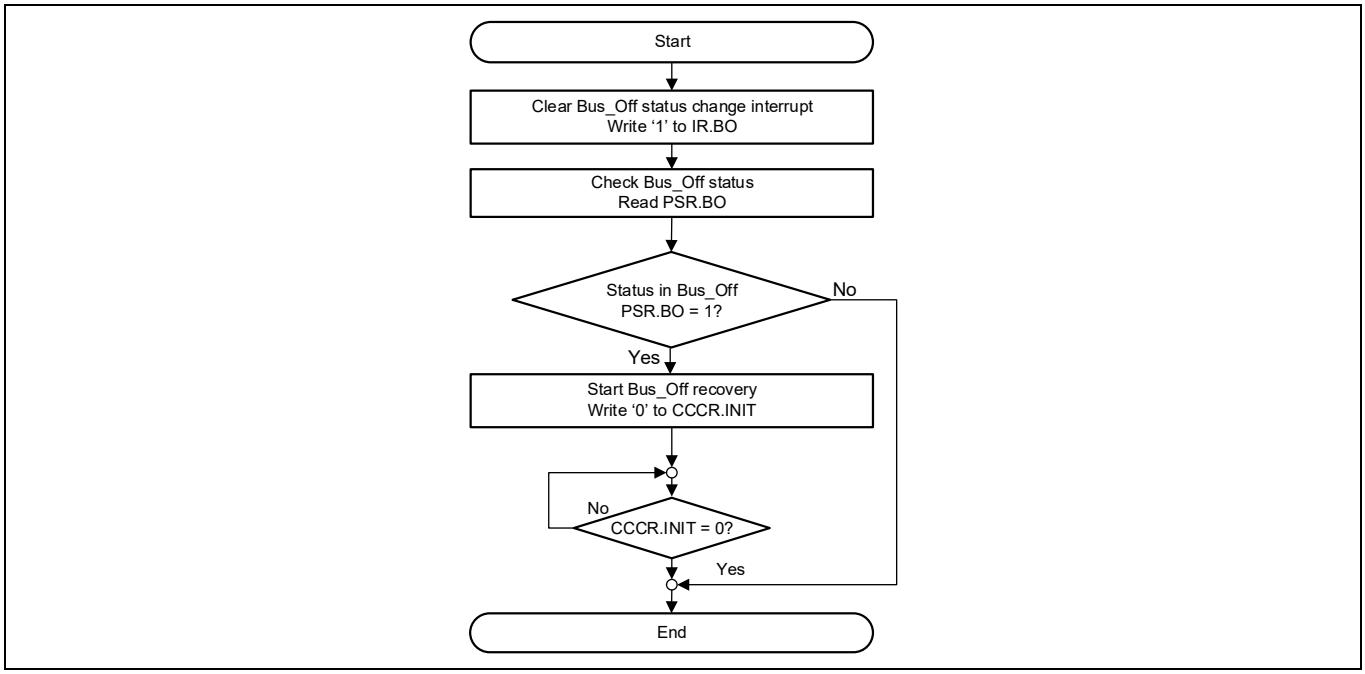


Figure 30-40. Bus OFF error handling

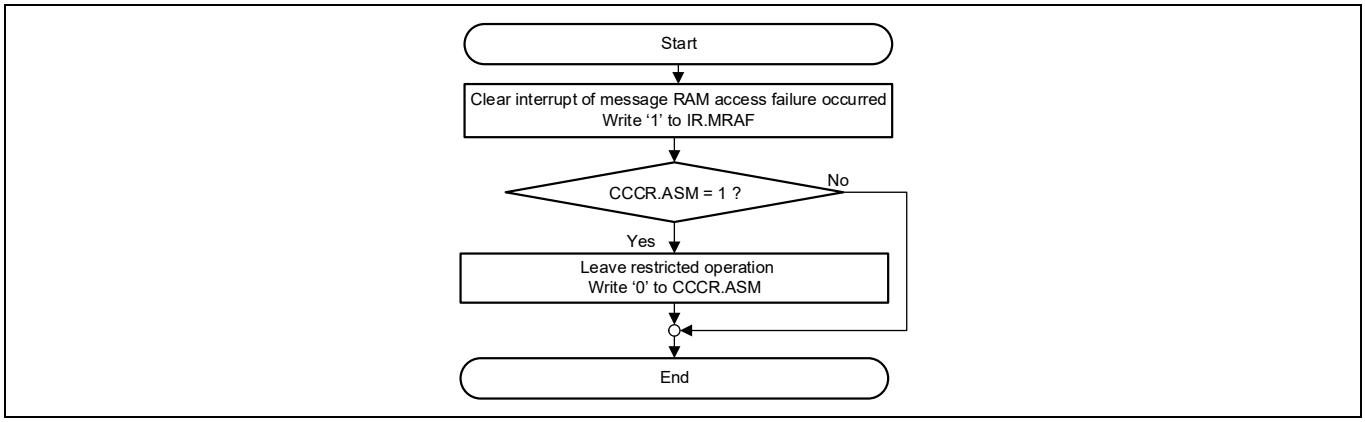


Figure 30-41. Message RAM access failure handling

CAN FD controller

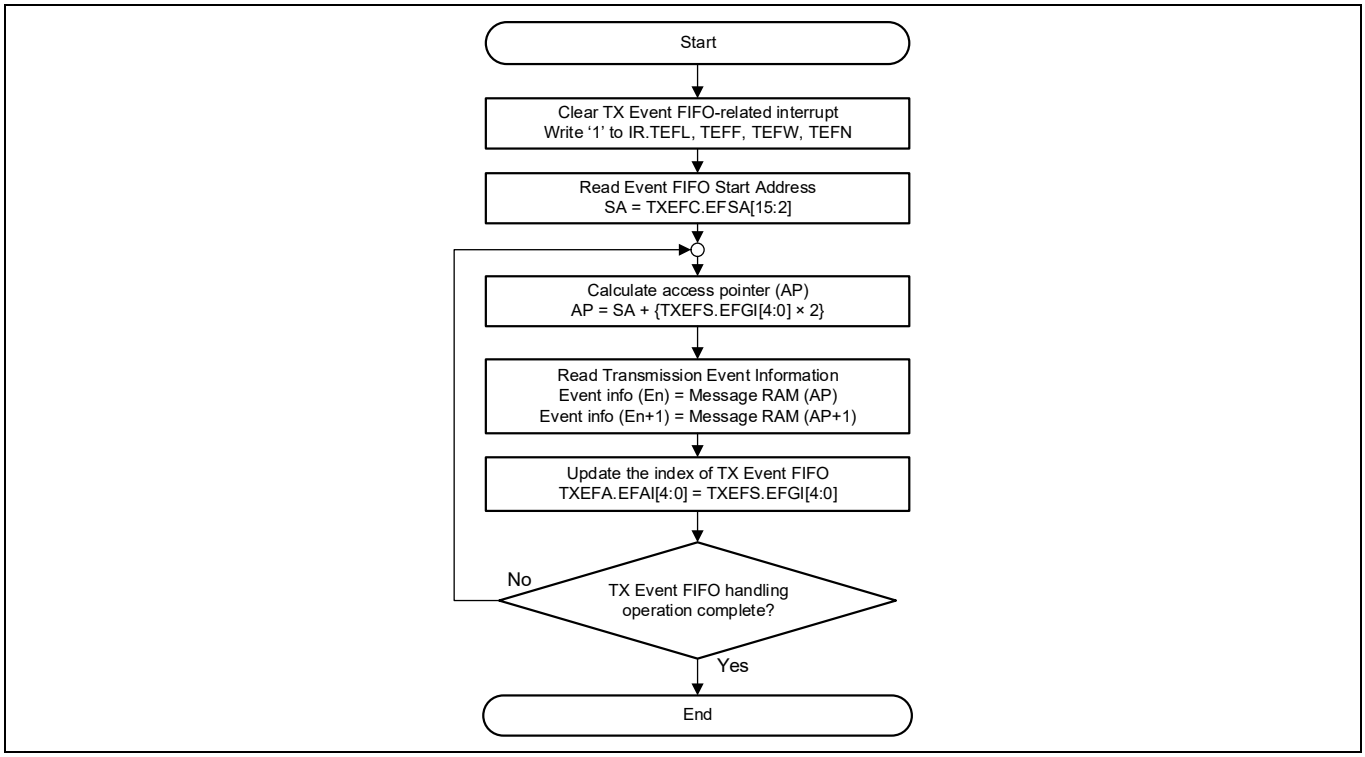


Figure 30-42. TX event FIFO handling

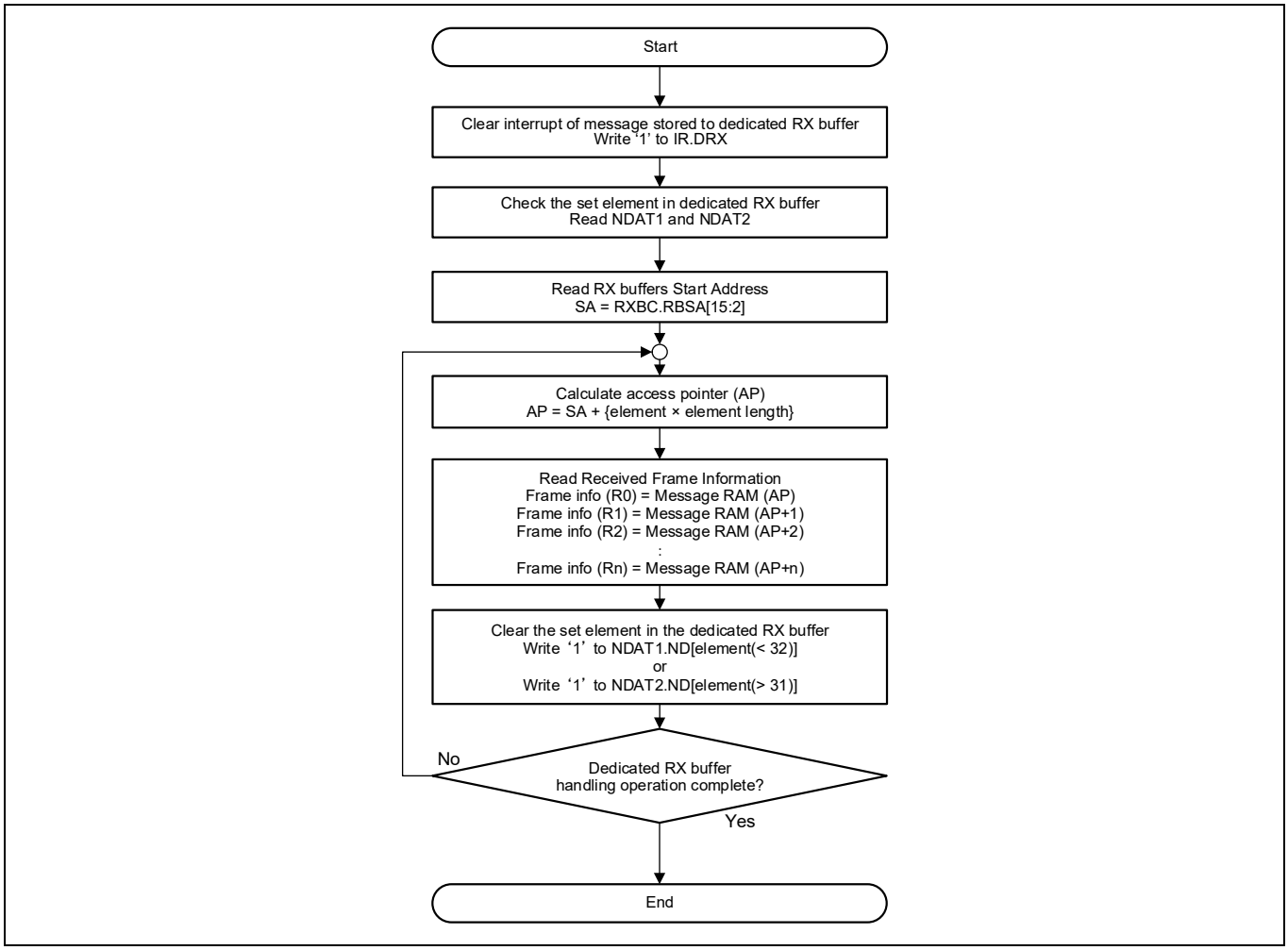


Figure 30-43. Dedicated RX buffer handling

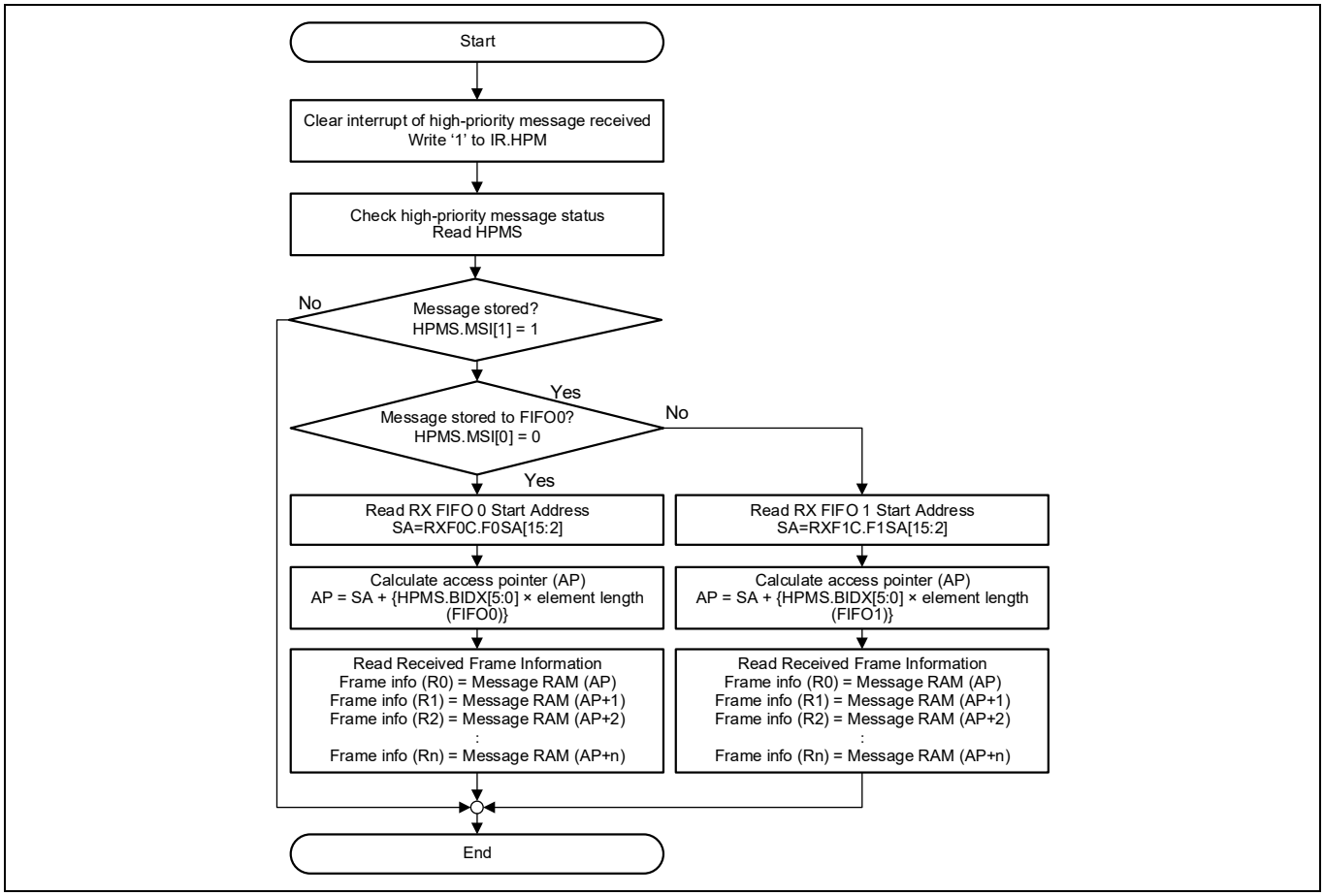


Figure 30-44. High priority message handling

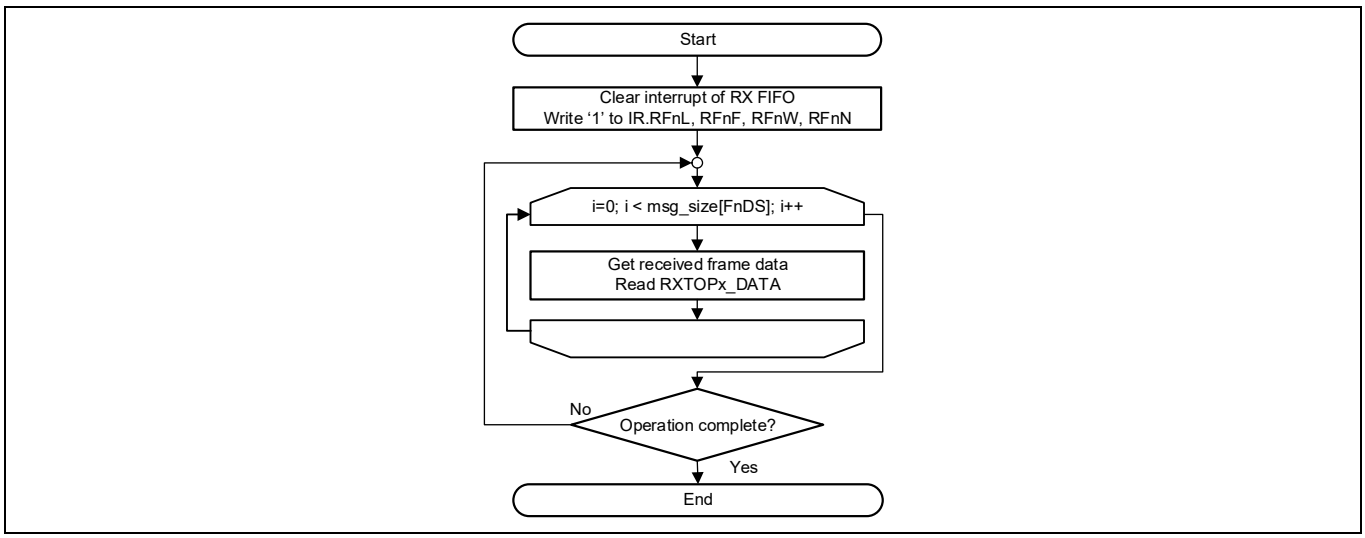


Figure 30-45. RX FIFO top pointer handling

30.7 Registers

Table 30-15. Registers

Register	Name	Description
CANFD_CH_CREL	Core Release Register	Displays the revision of the CAN FD controller.
CANFD_CH_ENDN	Endian Register	Checks the endianness of the CAN FD controller when accessed by the CPU.
CANFD_CH_DBTP	Data Bit Timing and Prescaler Register	Configures the data bit time and enables Transmitter Delay Compensation.
CANFD_CH_TEST	Test Register	Monitors the canfd[0].ttcan_rx[0]/canfd[0].ttcan_tx[0] pins. It is also used to enable the Loop Back modes.
CANFD_CH_RWD	RAM Watchdog	Monitors the message RAM to see if it is ready to be accessed.
CANFD_CH_CCCR	CC Control Register	Configures various operating modes of the CAN FD controller.
CANFD_CH_NBTP	Nominal Bit Timing and Prescaler Register	Configures the nominal bit time of the CAN FD controller.
CANFD_CH_TSCC	Timestamp Counter Configuration	Holds the settings for Timestamp Generation.
CANFD_CH_TSCV	Timestamp Counter Value	Hold the value of the Timestamp Counter.
CANFD_CH_TOCC	Timeout Counter Configuration	Holds the settings for the Timeout Counter.
CANFD_CH_TOCV	Timeout Counter Value	Holds the value of the Timeout Counter.
CANFD_CH_ECR	Error Counter Register	Holds the values of the Error Counters
CANFD_CH_PSR	Protocol Status Register	Displays the CAN protocol status of the CAN FD controller.
CANFD_CH_TDCR	Transmitter Delay Compensation Register	Configures the offset value and the filter window length for Transmitter Delay Compensation
CANFD_CH_IR	Interrupt Register	Holds the flags that are set when one of the listed conditions is detected (edge-sensitive).
CANFD_CH_IE	Interrupt Enable	The settings in this register determine which status changes in the Interrupt Register (IR) will be signaled on an interrupt line
CANFD_CH_ILS	Interrupt Line Select	Assigns an interrupt generated by a specific interrupt flag from the Interrupt Register (IR) to one of the two CAN FD controller interrupt lines (CANFD_int0/1).
CANFD_CH_ILE	Interrupt Line Enable	This register can separately enable/disable each of the two interrupt lines to the CPU.
CANFD_CH_GFC	Global Filter Configuration	Global settings for Message ID filtering.
CANFD_CH_SIDFC	Standard ID Filter Configuration	Settings for 11-bit standard Message ID filtering.
CANFD_CH_XIDFC	Extended ID Filter Configuration	Settings for 29-bit extended Message ID filtering.

Table 30-15. Registers (continued)

Register	Name	Description
CANFD_CH_XIDAM	Extended ID AND Mask	Defines the valid bits of a 29-bit ID for acceptance filtering.
CANFD_CH_HPMS	High Priority Message Status	This register is updated every time a Message ID filter element configured to generate a priority event matches.
CANFD_CH_NDAT1	New Data 1	Holds flags that are set when the respective dedicated RX buffer receives a frame.
CANFD_CH_NDAT2	New Data 2	Holds flags that are set when the respective dedicated RX buffer receives a frame.
CANFD_CH_RXF0C	RX FIFO 0 Configuration	Settings for the RX FIFO 0.
CANFD_CH_RXF0S	RX FIFO 0 Status	Status of the RX FIFO 0.
CANFD_CH_RXF0A	RX FIFO 0 Acknowledge	Acknowledges that the CPU has read a message or a sequence of messages from the RX FIFO 0 to indicate to the CAN FD controller that the corresponding message RAM area may be released.
CANFD_CH_RXBC	RX Buffer Configuration	Defines the start address of the RX Buffer section in the message RAM.
CANFD_CH_RXF1C	RX FIFO 1 Configuration	Settings for the RX FIFO 1.
CANFD_CH_RXF1S	RX FIFO 1 Status	Status of the RX FIFO 1.
CANFD_CH_RXF1A	RX FIFO 1 Acknowledge	Acknowledges that the CPU has read a message or a sequence of messages from the RX FIFO 1 to indicate to the CAN FD controller that the corresponding message RAM area may be released.
CANFD_CH_RXESC	RX Buffer/FIFO Element Size Configuration	Configures the number of data bytes belonging to an RX buffer and FIFO element.
CANFD_CH_TXBC	TX Buffer Configuration	Settings for TX buffers stored in the message RAM
CANFD_CH_TXFQS	TX FIFO/Queue Status	Related to the pending TX requests listed in the TX Buffer Request Pending register (TXBRP).
CANFD_CH_TXESC	TX Buffer Element Size Configuration	Configures the number of data bytes belonging to a TX buffer element
CANFD_CH_TXBRP	TX Buffer Request Pending	Holds the status of the transmission requests of each corresponding TX Buffer
CANFD_CH_TXBAR	TX Buffer Add Request	Requests the transmission of each corresponding TX buffer.
CANFD_CH_TXBCR	TX Buffer Cancellation Request	Cancels transmission requests of each corresponding TX buffer.
CANFD_CH_TXBTO	TX Buffer Transmission Occurred	Displays whether the corresponding TX buffer is transmitted.
CANFD_CH_TXBCF	TX Buffer Cancellation Finished	Signals whether the cancellation request of the corresponding TX buffer is successful.
CANFD_CH_TXBTIE	TX Buffer Transmission Interrupt Enable	The settings in this register determine which TX buffer will assert an interrupt upon transmission.

Table 30-15. Registers (continued)

Register	Name	Description
CANFD_CH_TXBCIE	TX Buffer Cancellation Finished Interrupt Enable	The settings in this register determine which TX buffer will assert an interrupt upon completion of a transmission cancellation request.
CANFD_CH_TXEFC	TX Event FIFO Configuration	Settings for the TX Event FIFO.
CANFD_CH_TXEFS	TX Event FIFO Status	Status of the TX Event FIFO.
CANFD_CH_TXEFA	TX Event FIFO Acknowledge	Acknowledges that the CPU has read an event from the TX Event FIFO to indicate to the CAN FD controller that the corresponding message RAM area may be released.
CANFD_CH_TTTMC	TT Trigger Memory Configuration	Configures memory element and memory start address.
CANFD_CH_TTRMC	TT Reference Message Configuration	Configures the reference message such as reference identifier, reference payload type, and type of identifier.
CANFD_CH_TTOCF	TT Operation Configuration	Configures fundamentals for time-triggered operations such as TTCAN operation level, time master, and clock calibration.
CANFD_CH_TTMLM	TT Matrix Limits	Configures cycle counts and synchronization for the clock start, and enables TX Window.
CANFD_CH_TURCF	TUR Configuration	Configures numerator and denominator for time unit configuration.
CANFD_CH_TTOCN	TT Operation Control	Controls main TTCAN operation.
CANFD_CH_TTGTP	TT Global Time Preset	Sets preset value and defines target of cycle time when a rising edge of ttcan event is expected.
CANFD_CH_TTTMK	TT Time Mark	Configures number of cycles in which time mark will be valid.
CANFD_CH_TTIR	TT Interrupt Register	Flags in the TTIR is set when particular conditions are met.
CANFD_CH_TTIE	TT Interrupt Enable	Provides possibility to enable interrupt for several status changes.
CANFD_CH_TTILS	TT Interrupt Line Select	User can select dedicated Interrupt0 or Interrupt1 line for specific interrupt source.
CANFD_CH_TTOST	TT Operation Status	Status register for TT operation.
CANFD_CH_TURNA	TUR Numerator Actual	Shows actual numerator value for time unit configuration.
CANFD_CH_TTLGT	TT Local and Global Time	Shows non-fractional part of the global and local time.
CANFD_CH_TTCTC	TT Cycle Time and Count	Read-only register that shows current cycle count and non-fraction part of the cycle time.
CANFD_CH_TTCPT	TT Capture Time	Read-only register that shows current cycle count and stop watch value.
CANFD_CH_TTCSM	TT Cycle Sync Mark	Read-only register that shows cycle sync mark in terms of cycle time.

Table 30-15. Registers (continued)

Register	Name	Description
CANFD_CH_RXFTOP_CTL	Receive FIFO Top control	Enables Receive FIFO Top Control logic for both FIFOs.
CANFD_CH_RXFTOP0_STAT	Receive FIFO 0 Top Status	This is a pointer to the next word in the message buffer defined by FIFO start address.
CANFD_CH_RXFTOP0_DATA	Receive FIFO 0 Top Data	Data placed at the address by RXFTOP0_STAT.
CANFD_CH_RXFTOP1_STAT	Receive FIFO 1 Top Status	This is a pointer to the next word in the message buffer defined by FIFO start address.
CANFD_CH_RXFTOP1_DATA	Receive FIFO 1 Top Data	Data placed at the address by RXFTOP1_STAT.
CANFD_CTL	Global CAN Control Register	Provides clock control to the TTCAN channel.
CANFD_STATUS	Global CAN Status Register	Read-only register that shows the acknowledge from the respective TTCAN channel for the clock stop request.
CANFD_INTR0_CAUSE	Consolidated Int0 Cause Register	Shows pending interrupt0 for the TTCAN channel.
CANFD_INTR1_CAUSE	Consolidated Int1 Cause Register	Shows pending interrupt1 for the TTCAN channel.
CANFD_TS_CTL	Time Stamp Control Register	Configuration for the Timestamp prescaler and counter enable is done in this register.
CANFD_TS_CNT	Time Stamp Count Register	Shows timestamp counter value.

31 Timer, counter, and PWM (TCPWM)

The Timer, counter, and pulse-width modulator (TCPWM) block in the EZ-USB™ FX10 MCU implements a 16-bit or 32-bit timer, counter, pulse width modulator (PWM), or quadrature decoder functionality. 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.

31.1 Features

- The TCPWM block supports the following operational modes:
  - Timer-counter with compare
  - Timer-counter with capture
  - Quadrature decoding
  - Pulse width modulation
  - Pseudo-random PWM
  - PWM with dead time
- Up, Down, and Up/Down counting modes
- Clock prescaling (division by 1, 2, 4, ... 64, 128)
- Double buffering of compare/capture and period values
- Underflow, overflow, and capture/compare output signals
- Supports interrupt on:
  - Terminal count – Depends on the mode; typically occurs on overflow or underflow
  - Capture/compare – The count is captured to the capture register or the counter value equals the value in the compare register
- Complementary output for PWMs
- Selectable start, reload, stop, count, and capture event signals for each TCPWM with rising edge, falling edge, both edges, and level trigger options

31.2 Architecture

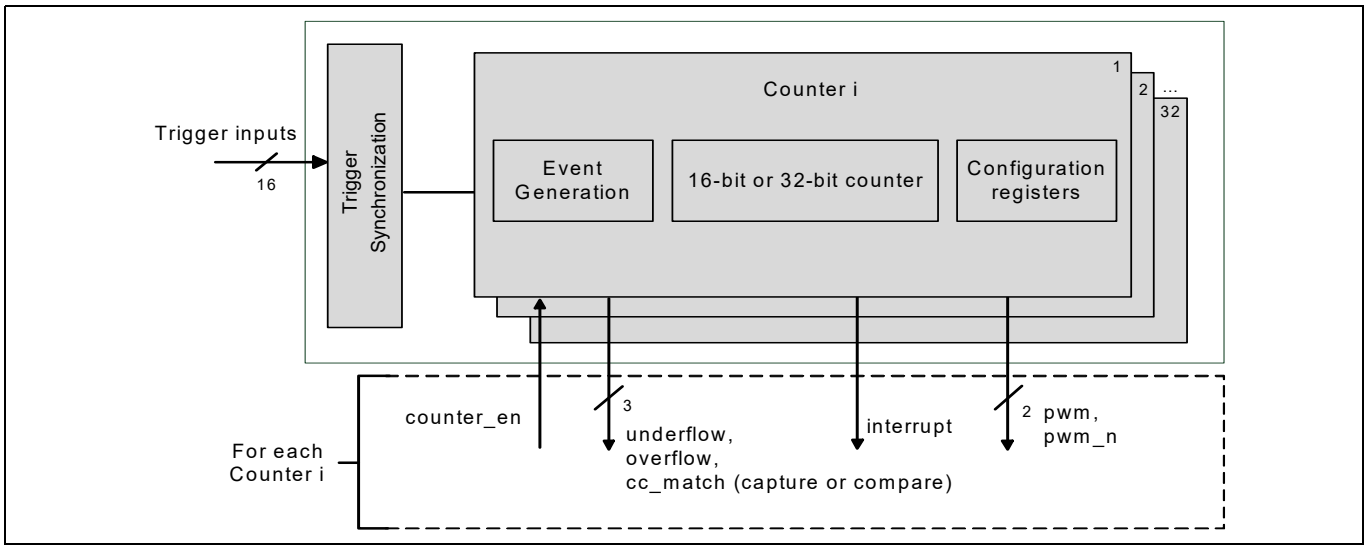


Figure 31-1. TCPWM block diagram

The TCPWM block in a EZ-USB™ FX10 MCU can contain up to 32 counters. Each counter can be 16- or 32-bit wide. The number of 16- and 32-bit counters are device specific; see the device datasheet for details.

In this chapter, a TCPWM refers to the entire block and all the counters inside. A counter refers to the individual counter inside the TCPWM.

TCPWM has these interfaces:

- I/O signal interface: Consists of input triggers (such as reload, start, stop, count, and capture) and output signals (such as pwm, pwm_n, overflow (OV), underflow (UN), and capture/compare (CC)).
- Interrupts: Provides interrupt request signals from each counter, based on TC or CC conditions.

The TCPWM block can be configured by writing to the TCPWM registers. See **"TCPWM registers"** on page 638 for more information on all registers required for this block.

31.2.1 Enabling and disabling counters in TCPWM block

A counter can be enabled by setting the corresponding bit of the TCPWM_CTRL_SET register. It can be disabled by setting the bit in the TCPWM_CTRL_CLR register. These registers are used to avoid race-conditions on read-modify-write attempts to the TCPWM_CTRL register, which controls the enable/disable fields of the counters.

Note: The counter must be configured before enabling it. Disabling the counter retains the values in the registers.

31.2.2 Clocking

Each TCPWM counter can have its own clock source. The only source for the clock is from the configurable peripheral clock dividers generated by the clocking system; see the **"Clocking system"** on page 290 for details. To select a clock divider for a particular counter inside a TCPWM, use the CLOCK_CTL register from the PERI register space. In this section the clock to the counter will be called clk_counter. Event generation is performed on the clk_counter. Another clock, clk_sys is used for the pulse width of the output triggers. clk_sys is synchronous to clk_peri (see CLK_PERI in EZ-USB™ FX10 register reference manual), but can be divided using CLOCK_CTL from the PERI_GROUP_STRUCT register space.

31.2.2.1 Clock prescaling

clk_counter can be further divided inside each counter, with values of 1, 2, 4, 8…64, 128. This division is called prescaling. The prescaling is set in the GENERIC field of the TCPWM_CNT_CTLR register.

Note: Clock prescaling is not available in quadrature mode and pulse width modulation mode with dead time.

31.2.2.2 Count event

The counter functionality is performed on an "active count" prescaled clock, which is gated by a "count event" signal as shown in **Figure 31-2**. For example, a counter increments or decrements by '1' every clk_counter cycle in which a count event is detected. Count event generation depends on edge detection modes on count inputs.

When the count input is configured as level, the count value is changed on each prescaled clk_counter edge in which the count input is high.

When the count input is configured as rising/falling the count value is changed on each prescaled clk_counter edge in which an edge is detected on the count input.

The next section contains additional details on edge detection modes configuration.

Note: Count events are not supported in quadrature and pulse-width modulation pseudo-random modes; the clk_counter is used in these cases instead of the active count prescaled clock.

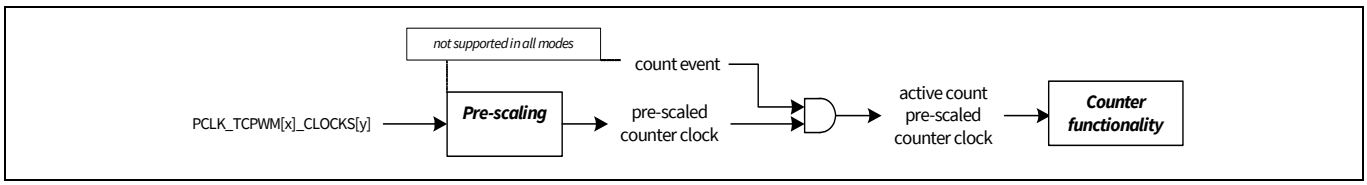


Figure 31-2. Counter clock Ggeneration

31.2.3 Trigger inputs

Each TCPWM block has 14 Trigger_In signals, which come from other on-chip resources such as other TCPWMs, SCBs, or DMA. The Trigger_In signals are shared with all counters inside of one TCPWM block. Use the Trigger Mux registers to configure which signals get routed to the Trigger_In for each TCPWM block. See the **"Trigger multiplexer block"** on page 344 for more details. Two constant trigger inputs '0' are '1' are available in addition to the 14 Trigger_In. 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 to be the source for any of the following events:

- Reload
- Start
- Stop/Kill
- Count
- Capture/swap

The TCPWM_CMD_RELOAD, TCPWM_CMD_STOP, TCPWM_CMD_START, and TCPWM_CMD_CAPTURE registers can be used to trigger the reload, stop, start, and capture respectively from software.

The sections describing each TCPWM mode will describe the function of each input event in detail.

Typical operation uses the reload event once to initialize and start the counter and the stop event to stop the counter. When the counter is stopped, the start event can be used to start the counter with its counter value unmodified from when it was stopped.

If stop, reload, and start events coincide, the following precedence relationship holds:

- A stop event has higher priority than a reload event
- A reload event has higher priority that a start event

As a result, when a reload or start event coincides with a stop event, the reload or start event has no effect.

Before going to the counter each Trigger_IN can pass through a positive edge detector, negative edge detector, both edge detector, or pass straight through to the counter. This is controlled using TCPWM_CNT_TR_CTRL1. In the quadrature mode, edge detection is done using clk_counter. For all other modes, edge detection is done using the clk_peri.

Multiple detected events are treated as follows:

- In the rising edge and falling edge modes, multiple events are effectively reduced to a single event. As a result, events may be lost.
- In the rising/falling edge mode, an even number of events are not detected and an odd number of events are reduced to a single event. This is because the rising/falling edge mode is typically used for capture events to determine the width of a pulse. The current functionality will ensure that the alternating pattern of rising and falling is maintained.

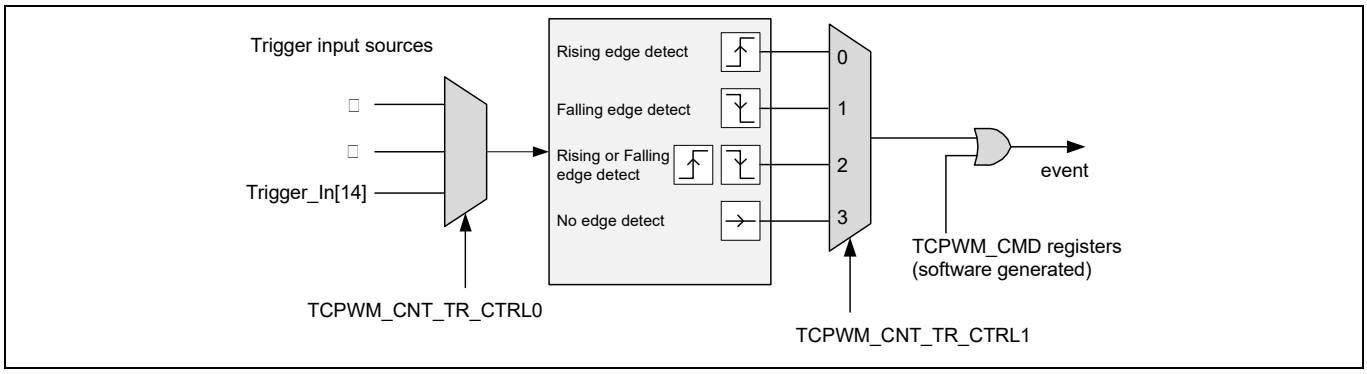


Figure 31-3. TCPWM input events

Note:

- All trigger inputs are synchronized to clk_peri
- When more than one event occurs in the same clk_counter period, one or more events may be missed. This can happen for high-frequency events (frequencies close to the counter frequency) and a timer configuration in which a pre-scaled (divided) clk_counter is used.

31.2.4 Trigger outputs

Each counter can generate three trigger output events. These trigger output events can be routed through the trigger mux to other peripherals on the device. The three trigger outputs are:

- Overflow (OV): An overflow event indicates that in up-counting mode, COUNTER equals the PERIOD register, and is changed to a different value
- Underflow (UN): An underflow event indicates that in a down-counting mode, COUNTER equals 0, and is changed to a different value
- Compare/Capture (CC): This event is generated when the counter is running and one of the following conditions occur
  - Counter equals the compare value. This event is either generated when the match is about to occur (COUNTER does not equal CC and is changed to CC) or when the match is not about to occur (COUNTER equals CC and is changed to a different value)
  - A capture event has occurred and the CC/CC_BUFF registers are updated

Note: These signals remain high only for two cycles of clk_sys.

31.2.5 Interrupts

The TCPWM block provides a dedicated interrupt output for each counter. This interrupt can be generated for a TC or CC event. A TC is the logical OR of the OV and UN events.

Four registers are used to handle interrupts in this block, as shown in **Table 31-1**.

Table 31-1. Interrupt register

Interrupt registers	Bits	Name	Description
TCPWM_CNT_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	CC_MATCH	This bit is set to ‘1’ when the counter value matches capture/compare register value. Write '1' to clear this bit.
TCPWM_CNT_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.
	1	CC_MATCH	Write '1' to set the corresponding bit in the interrupt request register. When read, this register reflects the interrupt request register status.
TCPWM_CNT_INTR_MASK (Interrupt mask register)	0	TC	Mask bit for the corresponding TC bit in the interrupt request register.
	1	CC_MATCH	Mask bit for the corresponding CC_MATCH bit in the interrupt request register.
TCPWM_CNT_INTR_MASKED (Interrupt masked request register)	0	TC	Logical AND of the corresponding TC request and mask bits.
	1	CC_MATCH	Logical AND of the corresponding CC_MATCH request and mask bits.

31.2.6 PWM outputs

Each counter two outputs, pwm and pwm_n (complementary of pwm). Note that the OV, UN, and CC conditions are used to drive pwm and pwm_n, by configuring the TCPWM_CNT_TR_CTRL2 register (**Table 31-2**).

Table 31-2. Configuring output for OV, UN, and CC conditions

Field	Bit	Value	Event	Description
CC_MATCH_MODE Default Value = 3	1:0	0	Set pwm to '1	Configures output line on a compare match (CC) event
		1	Clear pwm to '0	
		2	Invert pwm	
		3	No change	
OVERFLOW_MODE Default Value = 3	3:2	0	Set pwm to '1	Configures output line on a overflow (OV) event
		1	Clear pwm to '0	
		2	Invert pwm	
		3	No change	
UNDERFLOW_MODE Default Value = 3	5:4	0	Set pwm to '1	Configures output line on a underflow (UN) event
		1	Clear pwm to '0	
		2	Invert pwm	
		3	No change	

31.2.7 Power modes

The TCPWM block works in Active and Sleep modes. The TCPWM block is powered from V_{CCD}. The configuration registers and other logic are powered in Deep Sleep mode to keep the states of configuration registers. See **Table 31-3** for details.

Table 31-3. Power modes in TCPWM block

Power mode	Block status
Active	This block is fully operational in this mode with clock running and power switched on.
Sleep	The CPU is in sleep but the block is still functional in this mode. All counter clocks are on.
Deep Sleep	Both power and clocks to the block are turned off, but configuration registers retain their states.

31.3 Operation modes

The counter block can function in six operational modes, as shown in **Table 31-4**. The MODE [26:24] field of the counter control register (TCPWM_CNTx_CTRL) configures the counter in the specific operational mode.

Table 31-4. Operational mode xonfiguration

Mode	MODE field [26:24]	Description
Timer	000	The counter increments or decrements by '1' at every clk_counter cycle in which a count event is detected. The Compare/Capture register is used to compare the count.
Capture	010	The counter increments or decrements by '1' at every clk_counter cycle in which a count event is detected. A capture event copies the counter value into the capture register.
Quadrature	011	Quadrature decoding. The counter is decremented or incremented based on two phase inputs according to an X1, X2, or X4 decoding scheme.
PWM	100	Pulse width modulation.
PWM_DT	101	Pulse width modulation with dead time insertion.
PWM_PR	110	Pseudo-random PWM using a 16- or 32-bit linear feedback shift register (LFSR) to generate pseudo-random noise.

The counter can be configured to count up, down, and up/down by setting the UP_DOWN_MODE[17:16] field in the TCPWM_CNT_CTRL register, as shown in **Table 31-5**.

Table 31-5. 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) and overflow (OV) 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 and underflow (UN) 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. TC and UN conditions are 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. OV and UN conditions are generated similar to how they are generated in UP and DOWN counting modes respectively.

31.3.1 Timer mode

The timer mode is commonly used to measure the time of occurrence of an event or to measure the time difference between two events. The timer functionality increments/decrements a counter between 0 and the value stored in the PERIOD register. When the counter is running, the count value stored in the COUNTER register is compared with the compare/capture register (CC). When the counter changes from a state in which COUNTER equals CC, the cc_match event is generated.

Timer functionality is typically used for one of the following:

- Timing a specific delay – the count event is a constant '1'.
- Counting the occurrence of a specific event – the event should be connected as an input trigger and selected for the count event.

Table 31-6. Timer mode trigger input description

Trigger inputs	Usage
Reload	Sets the counter value 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
Capture	Not used

Incrementing and decrementing the counter is controlled by the count event and the counter clock clk_counter. Typical operation will use a constant ‘1’ count event and clk_counter without pre-scaling. Advanced operations are also possible; for example, the counter event configuration can decide to count the rising edges of a synchronized input trigger.

Table 31-7. Timer mode supported features

Supported features	Description
Clock pre-scaling	Pre-scales the counter clock clk_counter
One-shot	Counter is stopped by hardware, after a single period of the counter: • COUNT_UP: on an overflow event. • COUNT_DOWN, COUNT_UPDN1/2: on an underflow event.
Auto reload CC	CC and CC_BUFF are exchanged on a cc_match event (when specified by CTRL.AUTO_RELOAD_CC)
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 31-8 lists the trigger outputs and the conditions when they are triggered.

Table 31-8. Timer mode trigger outputs

Trigger outputs	Description
cc_match (CC)	Counter changes from a state in which COUNTER equals CC.
Underflow (UN)	Counter is decrementing and changes from a state in which COUNTER equals “0”.
Overflow (OV)	Counter is incrementing and changes from a state in which COUNTER equals PERIOD.

*Note: Each output is only two clk_sys wide and is represented by an arrow in the timing diagrams in this chapter, for example see **Figure 31-5**.*

Table 31-9. Timer mode interrupt outputs

Interrupt outputs	Description
tc	Specified by UP_DOWN_MODE: • COUNT_UP: The tc event is the same as the overflow event. • COUNT_DOWN: The tc event is the same as the underflow event. • COUNT_UPDN1: The tc event is the same as the underflow event. • COUNT_UPDN2: The tc event is the same as the logical OR of the overflow and underflow events.
cc_match (CC)	Counter changes from a state in which COUNTER equals CC.

Table 31-10. Timer mode PWM outputs

PWM outputs	Description
pwm	Not used.
pwm_n	Not used.

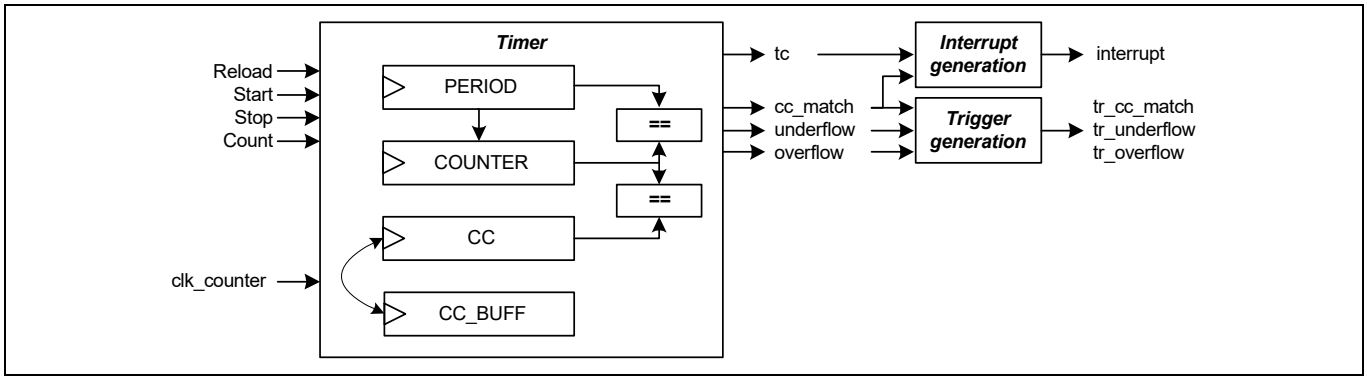


Figure 31-4. Timer functionality

Note:

- The timer functionality uses only PERIOD (and not PERIOD_BUFF)
- Do not write to COUNTER when the counter is running

Figure 31-5 illustrates a timer in up-counting mode. The counter is initialized (to 0) and started with a software-based reload event.

Note:

- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods. The CC register is 2, and sets the condition for a cc_match event.
- When the counter changes from a state in which COUNTER is 4, overflow and tc events are generated
- When the counter changes from a state in which COUNTER is 2, a cc_match event is generated
- A constant count event of '1' and clk_counter without prescaling is used in the following scenarios. If the count event is '0' and a reload event is triggered, the reload will be registered only on the first clock edge when the count event is '1'.

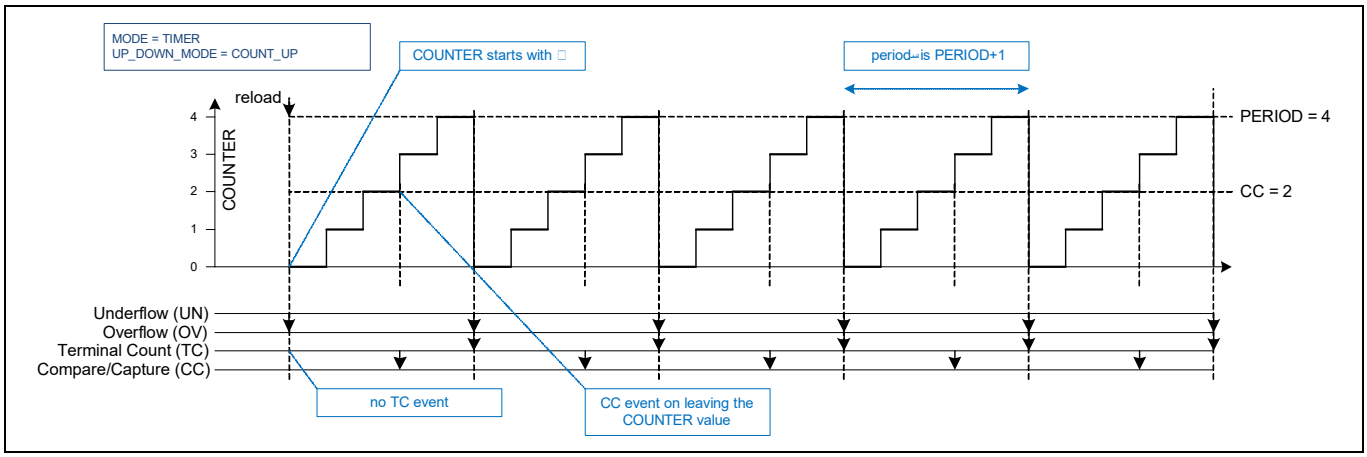


Figure 31-5. Timer in Up-counting mode

Figure 31-6 illustrates a timer in “one-shot” operation mode. Note that the counter is stopped on a tc event.

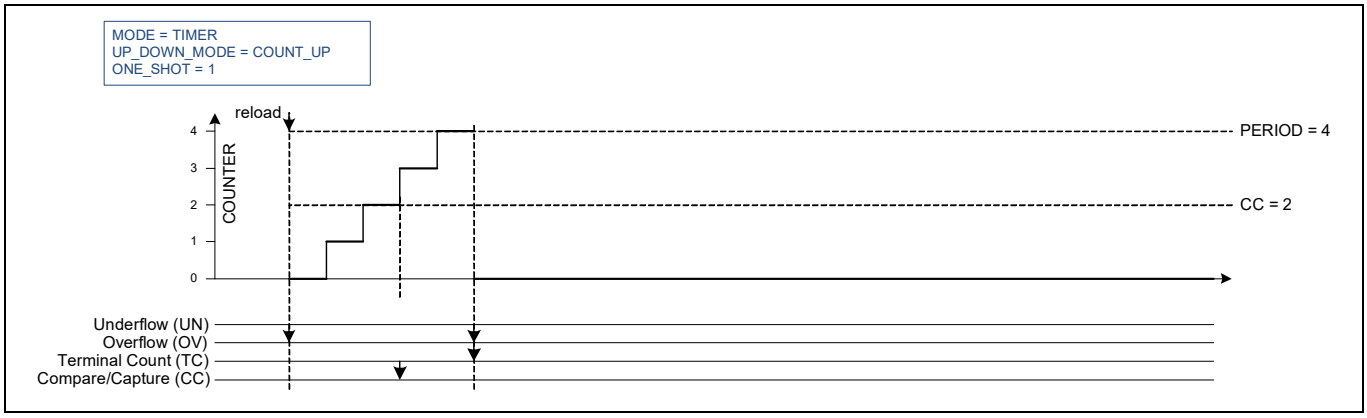


Figure 31-6. Timer in One-shot mode

Figure 31-7 illustrates clock pre-scaling. Note that the counter is only incremented every other counter cycle.

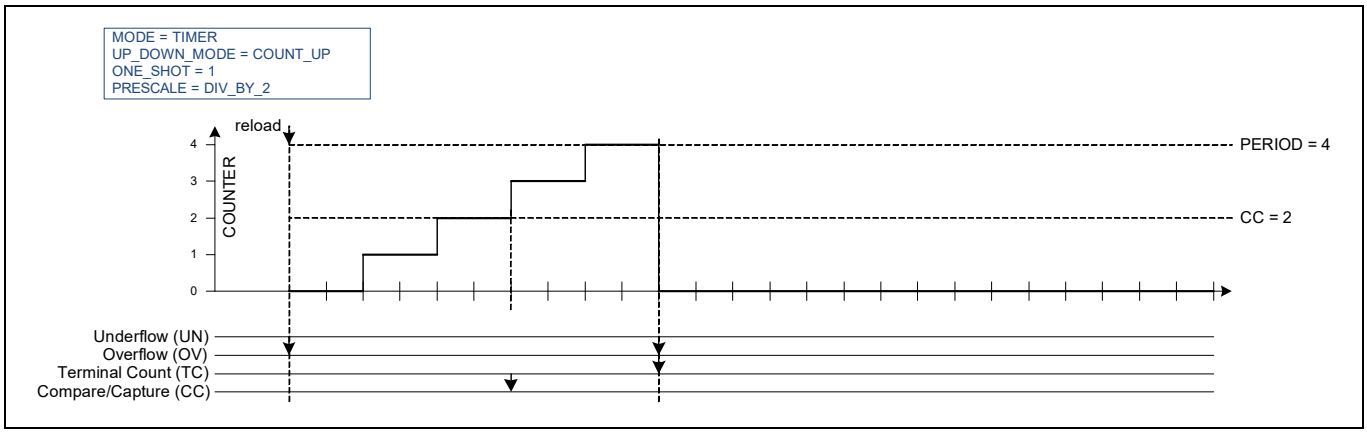


Figure 31-7. Timer clock pre-scaling

Figure 31-8 illustrates a counter that is initialized and started (reload event), stopped (stop event), and continued/started (start event). Note that the counter does not change value when it is not running (STATUS.RUNNING).

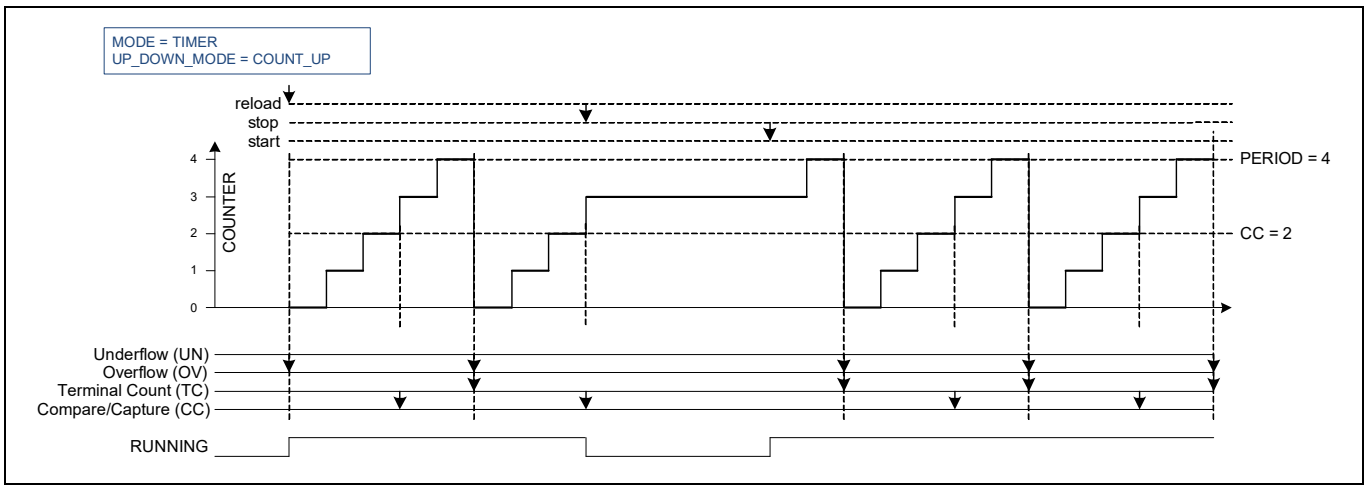


Figure 31-8. Counter Start/Stopped/Continued

Figure 31-9 illustrates a timer that uses both CC and CC_BUFF registers. Note that CC and CC_BUFF are exchanged on a cc_match event.

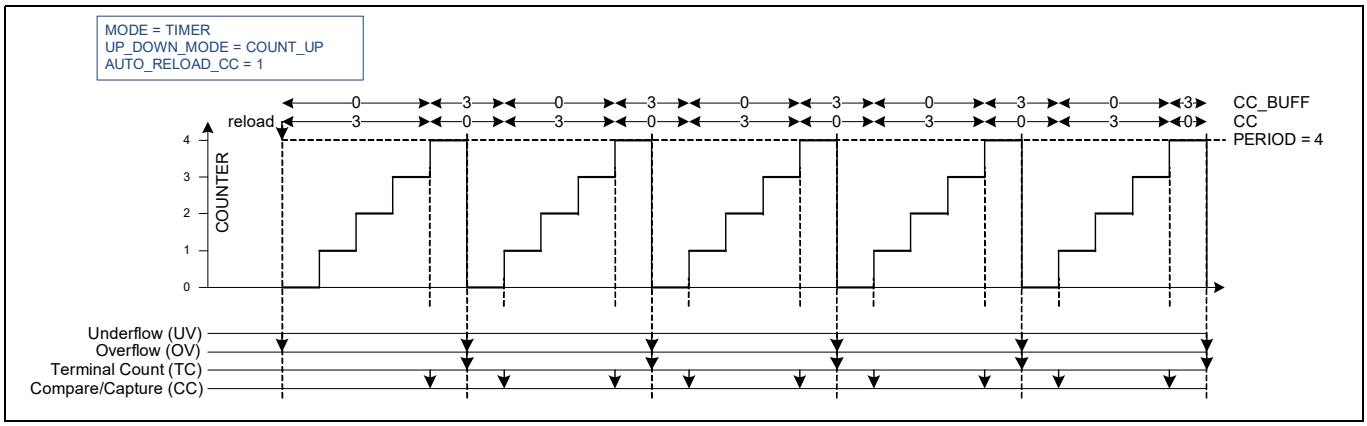


Figure 31-9. Use of CC and CC_BUFF register bits

Figure 31-10 illustrates a timer in down counting mode. The counter is initialized (to PERIOD) and started with a software-based reload event.

Note:

- When the counter changes from a state in which COUNTER is 0, a UN and TC events are generated
- When the counter changes from a state in which COUNTER is 2, a cc_match event is generated
- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods

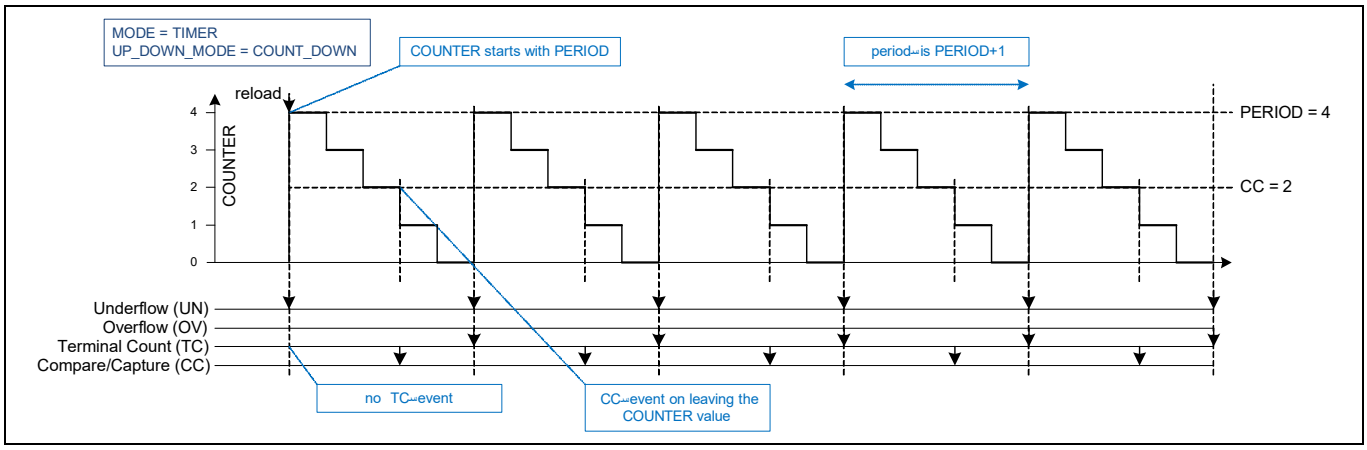


Figure 31-10. Timer in Down-counting mode

Figure 31-11 illustrates a timer in up/down counting mode 1. The counter is initialized (to 1) and started with a software-based reload event.

Note:

- When the counter changes from a state in which COUNTER is 4, an overflow is generated
- When the counter changes from a state in which COUNTER is 0, an underflow and tc event are generated
- When the counter changes from a state in which COUNTER is 2, a cc_match event is generated
- PERIOD is 4, resulting in an effective repeating counter pattern of 2*4 = 8 counter clock periods

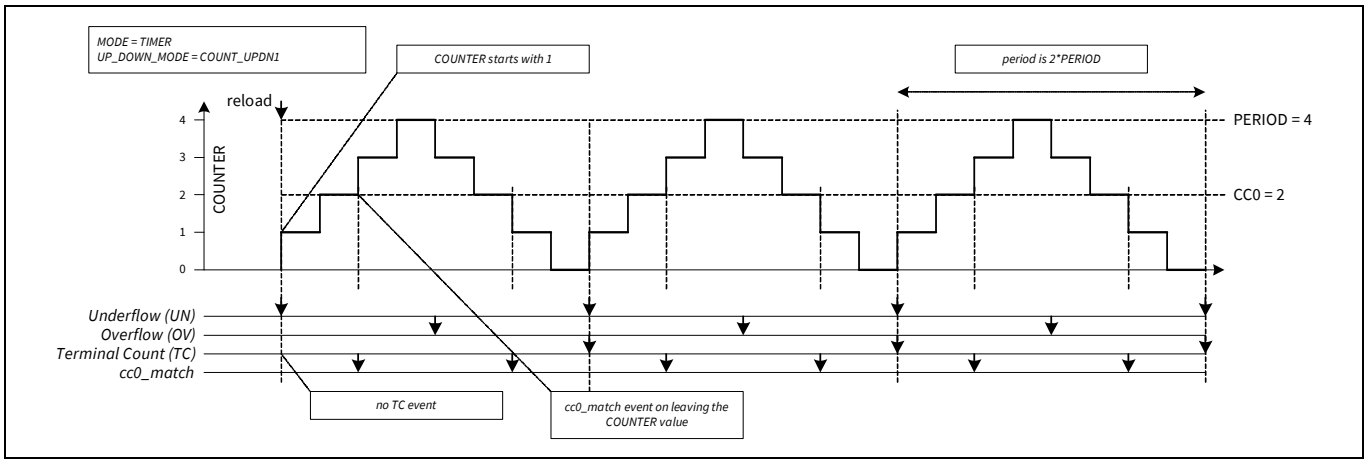


Figure 31-11. Timer in Up/Down Counting Mode 1

Figure 31-12 illustrates a timer in up/down counting mode 1, with different CC values.

Note:

- When CC is 0, the cc_match event is generated at the start of the period (when the counter changes from a state in which COUNTER is 0)
- When CC is PERIOD, the cc_match event is generated at the middle of the period (when the counter changes from a state in which COUNTER is PERIOD)

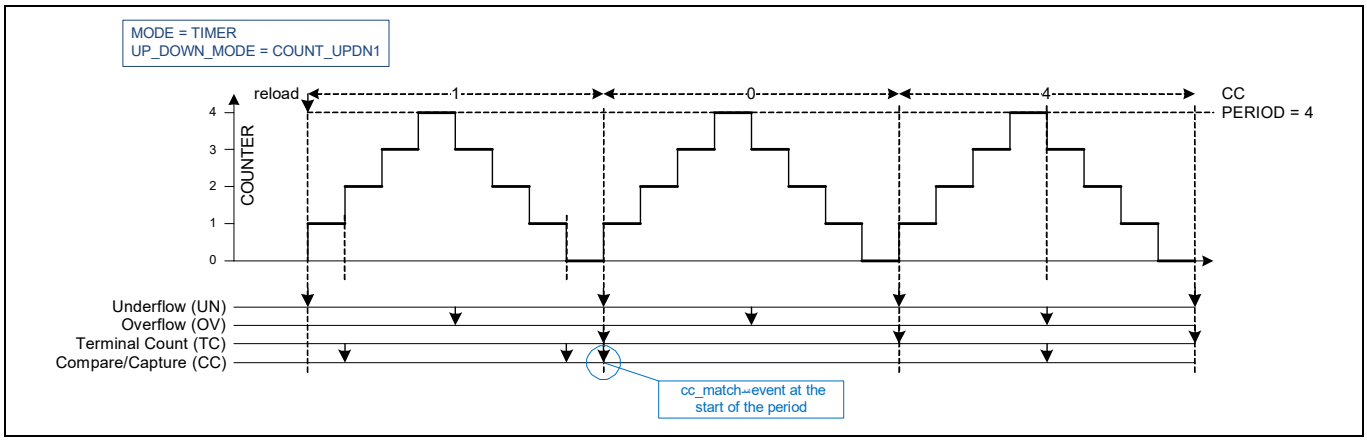


Figure 31-12. Up/Down Counting Mode with different CC values

Figure 31-13 illustrates a timer in up/down counting mode 2. This mode is same as up/down counting mode 1, except for the TC event, which is generated when either underflow or overflow event occurs.

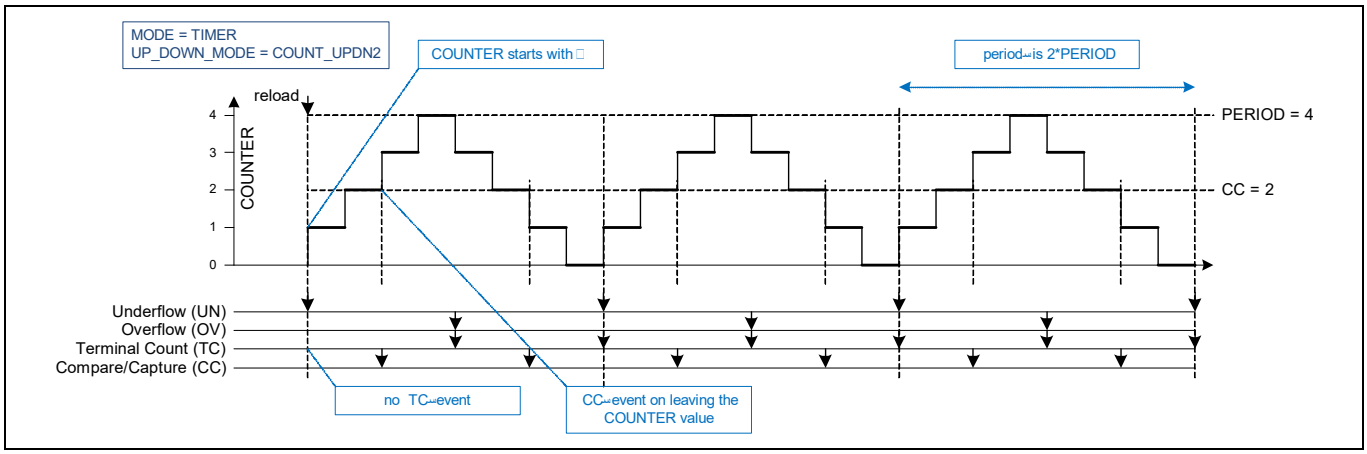


Figure 31-13. Up/Down Counting Mode 2

31.3.1.1 Configuring counter for Timer mode

The steps to configure the counter for Timer mode of operation and the affected register bits are as follows.

1. Disable the counter by writing '1' to the TCPWM_CTRL_CLR register.
2. Select Timer mode by writing '000' to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required 16- or 32-bit period in the TCPWM_CNT_PERIOD register.
4. Set the 16- or 32-bit compare value in the TCPWM_CNT_CC register and the buffer compare value in the TCPWM_CNT_CC_BUFF register.
5. Set AUTO_RELOAD_CC field of the TCPWM_CNT_CTRL register, if required to swap values at every CC condition.
6. Set clock prescaling by writing to the GENERIC[15:8] field of the TCPWM_CNT_CTRL register.
7. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register.
8. The timer can be configured to run either in continuous mode or one-shot mode by writing 0 or 1, respectively to the ONE_SHOT[18] field of the TCPWM_CNT_CTRL register.
9. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, stop, capture, and count).

10. Set the TCPWM_CNT_TR_CTRL1 register to select the edge of the trigger that causes the event (reload, start, stop, capture, and count).
11. If required, set the interrupt upon TC or CC condition.
12. Enable the counter by writing '1' to the TCPWM_CTRL_SET register. A start trigger must be provided through firmware (TCPWM_CMD_START register) to start the counter if the hardware start signal is not enabled.

31.3.2 Capture mode

The capture functionality increments and decrements a counter between 0 and PERIOD. When the capture event is activated the counter value COUNTER is copied to CC (and CC is copied to CC_BUFF).

The capture functionality can be used to measure the width of a pulse (connected as one of the input triggers and used as capture event).

The capture event can be triggered through the capture trigger input or through a firmware write to command register (TCPWM_CMD_CAPTURE).

Table 31-11. Capture mode trigger input description

Trigger inputs	Usage
reload	Sets the counter value 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 only when the counter is 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: • 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. Can be used only when the counter is not running.
stop	Stops the counter
count	Count event increments/decrements the counter
capture	Copies the counter value to CC and copies CC to CC_BUFF.

Table 31-12. Capture mode supported features

Supported features	Description
Clock pre-scaling	Pre-scales the counter clock clk_counter
One-shot	Counter is stopped by hardware, after a single period of the counter: • COUNT_UP: on an overflow event. • COUNT_DOWN, COUNT_UPDN1/2: on an underflow event.
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 31-13. Capture mode trigger output description

Trigger outputs	Description
cc_match (CC)	CC is copied to CC_BUFF and counter value is copied to CC (cc_match equals capture event).
Underflow (UN)	Counter is decrementing and changes from a state in which COUNTER equals “0”.
Overflow (OV)	Counter is incrementing and changes from a state in which COUNTER equals PERIOD.

Table 31-14. Capture mode interrupt outputs

Interrupt outputs	Description
tc	Specified by UP_DOWN_MODE: • COUNT_UP: tc event is the same as the overflow event. • COUNT_DOWN: tc event is the same as the underflow event. • COUNT_UPDN1: tc event is the same as the underflow event. • COUNT_UPDN2: tc event is the same as the logical OR of the overflow and underflow events.
cc_match (CC)	CC is copied to CC_BUFF and counter value is copied to CC (cc_match equals capture event).

Table 31-15. Capture mode PWM outputs

PWM outputs	Description
pwm	Not used
pwm_n	Not used

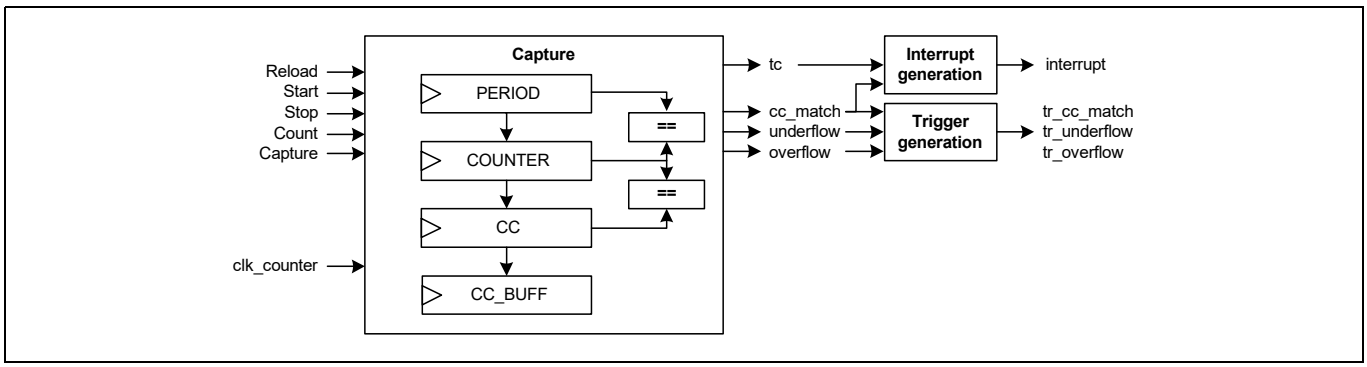


Figure 31-14. Capture functionality

Figure 31-15 illustrates capture behavior in the up counting mode.

Note:

- The capture event detection uses rising edge detection. As a result, the capture event is remembered until the next “active count” pre-scaled counter clock.
- When a capture event occurs, COUNTER is copied into CC. CC is copied to CC_BUFF
- A cc_match event is generated when the counter value is captured

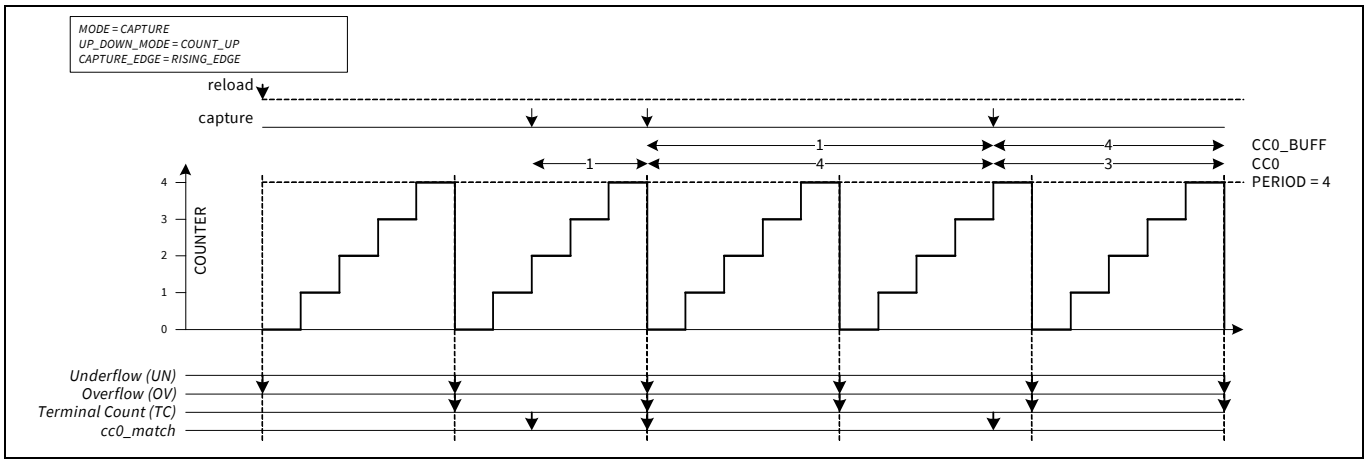


Figure 31-15. Capture in Up Counting mode

When multiple capture events are detected before the next “active count” pre-scaled counter clock, capture events are treated as follows:

- In the rising edge and falling edge modes, multiple events are effectively reduced to a single event
- In the rising/falling edge mode, an even number of events is not detected and an odd number of events is reduced to a single event

This behavior is illustrated by **Figure 31-16**, in which a pre-scaler by a factor of 4 is used.

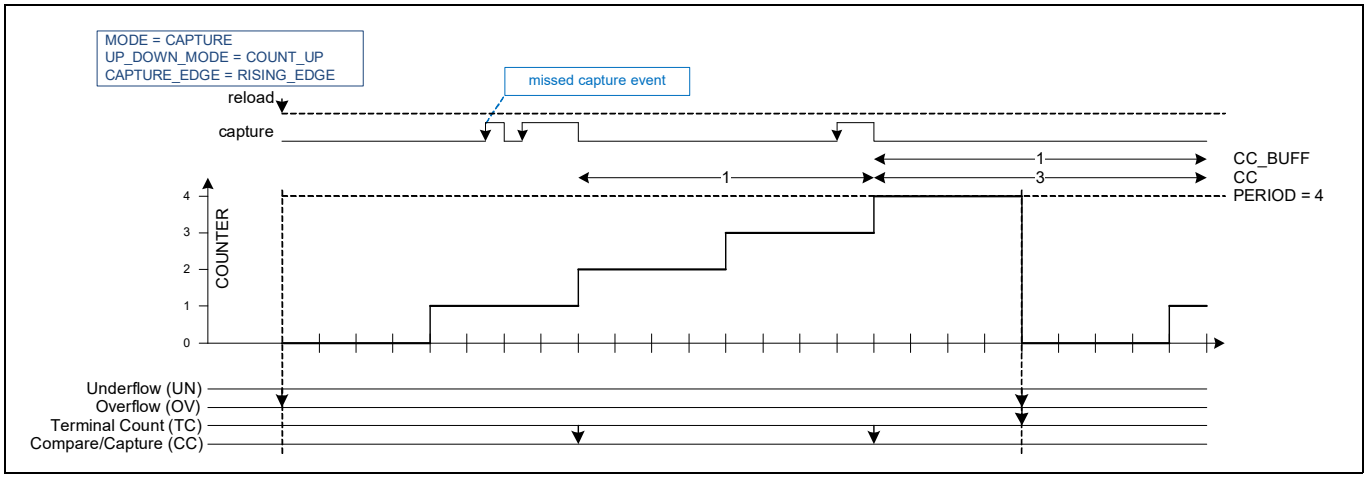


Figure 31-16. Multiple events detected before Active-Count

31.3.2.1 Configuring counter for Capture mode

The steps to configure the counter for Capture mode operation and the affected register bits are as follows.

1. Disable the counter by writing '1' to the TCPWM_CTRL_CLR register.
2. Select Capture mode by writing '010' to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required 16-bit period in the TCPWM_CNT_PERIOD register.
4. Set clock prescaling by writing to the GENERIC[15:8] field of the TCPWM_CNT_CTRL register.
5. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register.
6. Counter can be configured to run either in continuous mode or one-shot mode by writing 0 or 1, respectively to the ONE_SHOT[18] field of the TCPWM_CNT_CTRL register.
7. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, stop, capture, and count).
8. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (reload, start, stop, capture, and count).
9. If required, set the interrupt upon TC or CC condition.
10. Enable the counter by writing '1' to the TCPWM_CTRL_SET register. A start trigger must be provided through firmware (TCPWM_CMD_START register) to start the counter if the hardware start signal is not enabled.

31.3.3 Quadrature Decoder mode

Quadrature functionality increments and decrements a counter between 0 and 0xFFFF or 0xFFFFFFFF (32-bit mode). Counter updates are under control of quadrature signal inputs: index, phiA, and phiB. The index input is used to indicate an absolute position. The phiA and phiB inputs are used to determine a change in position (the rate of change in position can be used to derive speed). The quadrature inputs are mapped onto triggers (as described in **Table 31-16**).

Table 31-16. Quadrature mode trigger input description

Trigger input	Usage
reload/index	This event acts as a quadrature index input. It initializes the counter to the counter midpoint 0x8000 (16-bit) or 0x8000000 (32-bit mode) and starts the quadrature functionality. Rising edge event detection or falling edge detection mode should be used.
start/phiB	This event acts as a quadrature phiB input. Pass through (no edge detection) event detection mode should be used.
stop	Stops the quadrature functionality.
count/phiA	This event acts as a quadrature phiA input. Pass through (no edge detection) event detection mode should be used.
capture	Not used

Table 31-17. Quadrature mode supported features

Supported features	Description
Quadrature encoding	Three encoding schemes for the phiA and phiB inputs are supported (as specified by CTRL.QUADRATURE_MODE): X1 encoding. X2 encoding. X4 encoding.

Note: Clock pre-scaling is not supported and the count event is used as a quadrature input phiA. As a result, the quadrature functionality operates on the counter clock (clk_counter), rather than on an "active count" prescaled counter clock.

Table 31-18. Quadrature mode trigger output description

Trigger outputs	Description
cc_match (CC)	Counter value COUNTER equals 0 or 0xFFFF or 0xFFFFFFFF (32-bit mode) or a reload/index event
Underflow (UN)	Not used
Overflow (OV)	Not used

Table 31-19. Quadrature mode interrupt outputs

Interrupt outputs	Description
cc_match (CC)	Counter value COUNTER equals 0 or 0xFFFF or 0xFFFFFFFF (32-bit mode) or a reload/index event
tc	Reload/index event

Table 31-20. Quadrature mode PWM outputs

PWM outputs	Description
pwm	Not used
pwm_n	Not used

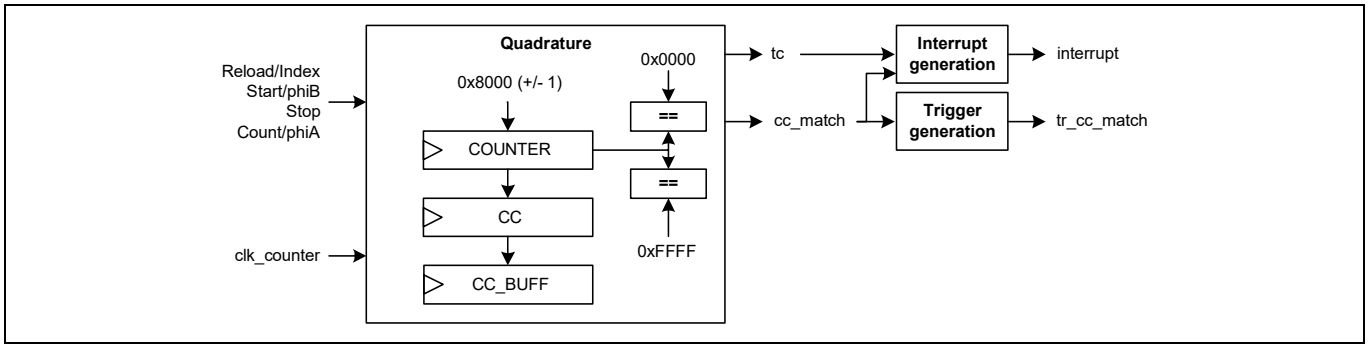


Figure 31-17. Quadrature Functionality (16-bit example)

Quadrature functionality is described as follows:

- A software-generated reload event starts quadrature operation. As a result, COUNTER is set to 0x8000 or 0x80000000, which is the counter midpoint (the COUNTER is set to 0x7FFF if the reload event coincides with a decrement event; the COUNTER is set to 0x8001 if the reload event coincides with an increment event). Note that a software-generated reload event is generated only once, when the counter is not running. All other reload/index events are hardware-generated reload events as a result of the quadrature index signal.

- During quadrature operation:
  - The counter value COUNTER is incremented or decremented based on the specified quadrature encoding scheme
  - On a reload/index event, CC is copied to CC_BUFF, COUNTER is copied to CC, and COUNTER is set to 0x8000. In addition, the tc and cc_match events are generated
  - When the counter value COUNTER is 0x0000, CC is copied to CC_BUFF, COUNTER (0x0000) is copied to CC, and COUNTER is set to 0x8000. In addition, the cc_match event is generated
  - When the counter value COUNTER is 0xFFFF, CC is copied to CC_BUFF, COUNTER (0xFFFF) is copied to CC, and COUNTER is set to 0x8000. In addition, the cc_match event is generated

The software interrupt handler uses the tc and cc_match interrupt cause fields to distinguish between a reload/index event and a situation in which a minimum/maximum counter value was reached (about to wrap around). The CC and CC_BUFF registers are used to determine when the interrupt causing event occurred.

Note that a counter increment/decrement can coincide with a reload/index/tc event or with a situation cc_match event. Under these circumstances, the counter value set to either 0x8000+1 (increment) or 0x8000-1 (decrement).

Counter increments (incr1 event) and decrements (decr1 event) are determined by the quadrature encoding scheme as illustrated by **Figure 31-18**.

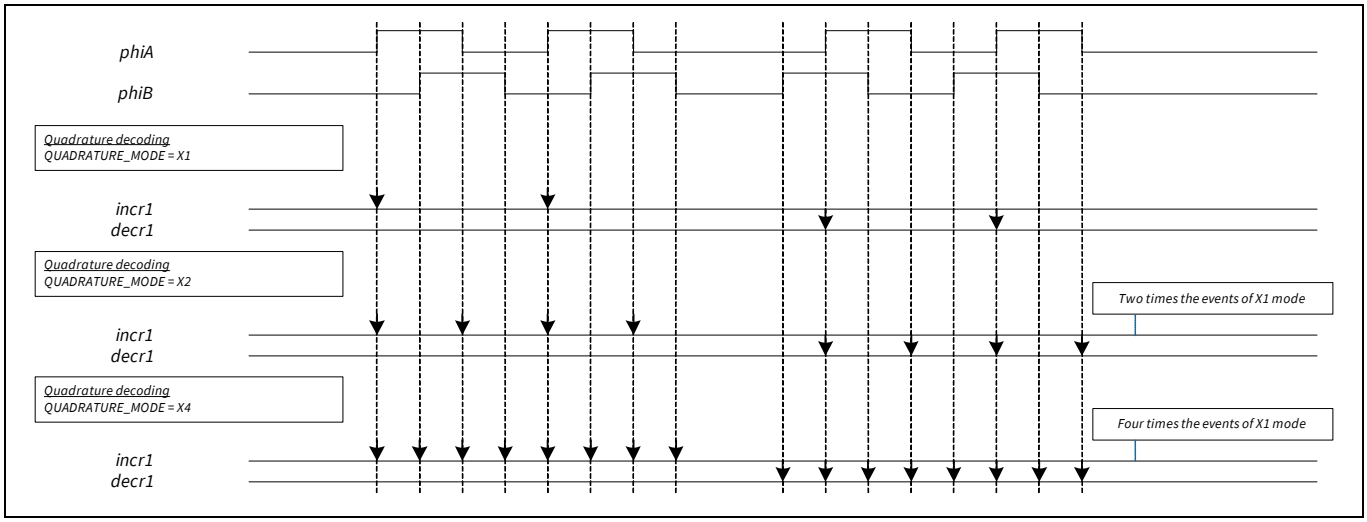


Figure 31-18. Quadrature mode waveforms

Figure 31-19 illustrates quadrature functionality as a function of the reload/index, incr1, and decr1 events. Note that the first reload/index event copies the counter value COUNTER to CC.

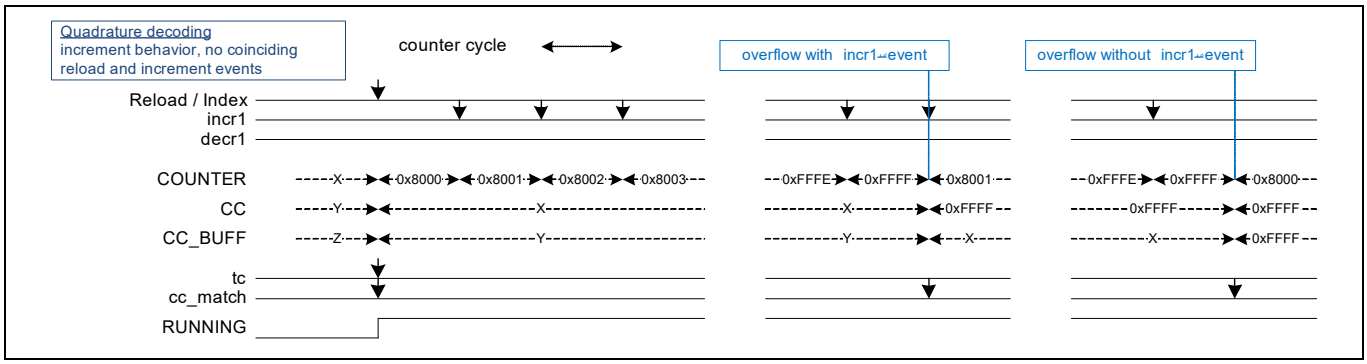


Figure 31-19. Quadrature mode reload/index timing

Figure 31-20 illustrate quadrature functionality for different event scenarios (including scenarios with coinciding events). In all scenarios, the first reload/index event is generated by software when the counter is not yet running.

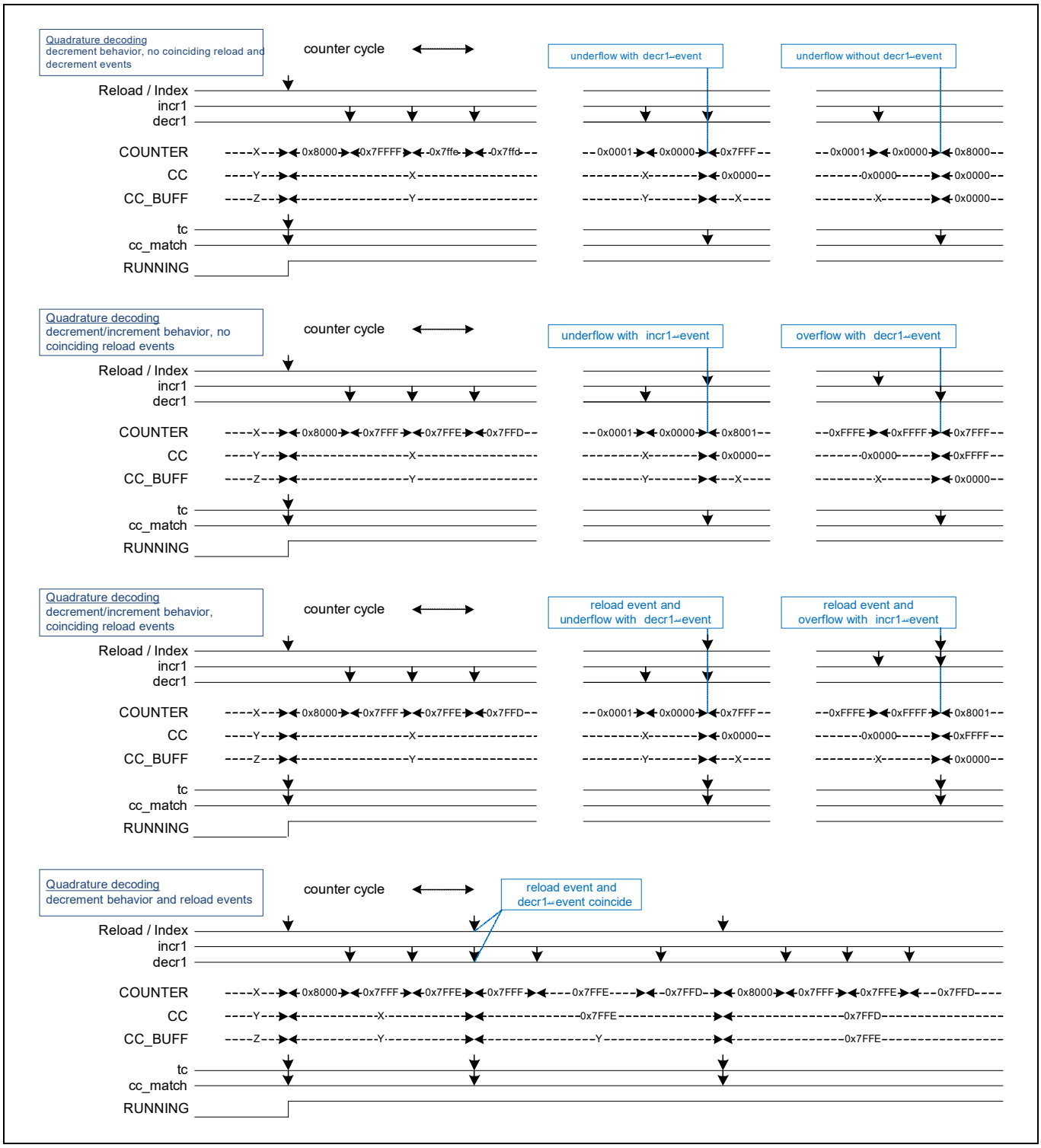


Figure 31-20. Quadrature mode timing cases

31.3.3.1 Configuring counter for Quadrature mode

The steps to configure the counter for quadrature mode of operation and the affected register bits are as follows.

1. Disable the counter by writing '1' to the TCPWM_CTRL_CLR register.
2. Select Quadrature mode by writing '011' to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required encoding mode by writing to the QUADRATURE_MODE[21:20] field of the TCPWM_CNT_CTRL register.
4. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (Index and Stop).
5. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (Index and Stop).
6. If required, set the interrupt upon TC or CC condition.
7. Enable the counter by writing '1' to the TCPWM_CTRL_SET register. A start trigger must be provided through firmware (TCPWM_CMD_START register) to start the counter if the hardware start signal is not enabled.

31.3.4 Pulse-width modulation mode

PWM functionality increments/decrements a counter between 0 and PERIOD. When the counter is running, the counter value COUNTER is compared with CC. When COUNTER equals CC, the cc_match event is generated. Additionally, on a counter overflow and counter underflow, the overflow and underflow events are generated. Combined, the cc_match, overflow and underflow events are used to generate a pulse-width modulated signal on the PWM "pwm" and "pwm_n" output signals. Left-aligned, right-aligned, and center-aligned PWM signals can be generated. Asymmetric PWM signals can be generated using the COUNT_UPDN2 mode.

Table 31-21. PWM mode trigger input description

Trigger inputs	Usage
reload	Sets the counter value 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 only when the counter is 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: • 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 set to "up". Can be used only when the counter is not running.
stop/kill	Stops the counter or suppresses the PWM output, depending on PWM_STOP_ON_KILL and PWM_SYNC_KILL.
count	Count event increments/decrements the counter.
capture/swap	This event acts as a swap event. When this event is active, the CC/CC_BUFF and PERIOD/PERIOD_BUFF registers are exchanged on a tc event (when specified by CTRL.AUTO_RELOAD_CC and CTRL.AUTO_RELOAD_PERIOD). A swap event requires rising, falling, or rising/falling edge event detection mode. Pass-through mode is not supported, unless the selected event is a constant '0' or '1'. *Note: When COUNT_UPDN2 mode exchanges PERIOD and PERIOD_BUFF at a tc event that coincides with an OV event, software should ensure that the PERIOD and PERIOD_BUFF values are the same.* When a swap event is detected and the counter is running, the event is kept pending until the next tc event. When a swap event is detected and the counter is not running, the event is cleared by hardware.

Table 31-22. PWM mode supported features

Supported features	Description
Clock pre-scaling	Pre-scales the counter clock “clk_counter”.
One-shot	Counter is stopped by hardware, after a single period of the counter: • COUNT_UP: on an overflow event. • COUNT_DOWN and COUNT_UPDN1/2: on an underflow event.
Compare Swap	CC and CC_BUFF are exchanged on a swap event and tc event (when specified by CTRL.AUTO_RELOAD_CC).
Period Swap	PERIOD and PERIOD_BUFF are exchanged on a swap event and tc event (when specified by CTRL.AUTO_RELOAD_PERIOD). *Note: When COUNT_UPDN2/Asymmetric mode exchanges PERIOD and PERIOD_BUFF at a tc event that coincides with an overflow event, software should ensure that the PERIOD and PERIOD_BUFF values are the same.*
Alignment (Up/Down modes)	Specified by UP_DOWN_MODE: • COUNT_UP: The counter counts from 0 to PERIOD. Generates a left-aligned PWM output. • COUNT_DOWN: The counter counts from PERIOD to 0. Generates a right-aligned PWM output. • COUNT_UPDN1/2: The counter counts from 1 to PERIOD and back to 0. Generates a center-aligned/asymmetric PWM output.
Kill modes	Specified by PWM_STOP_ON_KILL and PWM_SYNC_KILL: • PWM_STOP_ON_KILL = ‘1’ (PWM_SYNC_KILL = don’t care): Stop on Kill mode. This mode stops the counter on a stop/kill event. Reload or start event is required to start count running. • PWM_STOP_ON_KILL = ‘0’ and PWM_SYNC_KILL = ‘0’: Asynchronous kill mode. This mode keeps the counter running, but suppresses the PWM output signals and continues to do so for the duration of the stop/kill event. • PWM_STOP_ON_KILL = ‘0’ and PWM_SYNC_KILL = ‘1’: Synchronous kill mode. This mode keeps the counter running, but suppresses the PWM output signals and continues to do so until the next tc event without a stop/kill event.

Note that the PWM mode does not support dead time insertion. This functionality is supported by the separate PWM_DT mode.

Table 31-23. PWM mode trigger output description

Trigger output	Description
cc_match (CC)	Specified by UP_DOWN_MODE: • COUNT_UP and COUNT_DOWN: The counter changes to a state in which COUNTER equals CC. • COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC.
Underflow (UN)	Counter is decrementing and changes from a state in which COUNTER equals ‘0’.
Overflow (OV)	Counter is incrementing and changes from a state in which COUNTER equals PERIOD.

Table 31-24. PWM mode interrupt output description

Interrupt outputs	Description
tc	Specified by UP_DOWN_MODE: • COUNT_UP: tc event is the same as the overflow event. • COUNT_DOWN: tc event is the same as the underflow event. • COUNT_UPDN1: tc event is the same as the underflow event. • COUNT_UPDN2: tc event is the same as the logical OR of the overflow and underflow events.
cc_match (CC)	Specified by UP_DOWN_MODE: • COUNT_UP and COUNT_DOWN: The counter changes to a state in which COUNTER equals CC. • COUNT_UPDN1/2: counter changes from a state in which COUNTER equals CC.

Table 31-25. PWM mode PWM outputs

PWM outputs	Description
pwm	PWM output.
pwm_n	Complementary PWM output.

Note that the cc_match event generation in COUNT_UP and COUNT_DOWN modes are different from the generation in other functional modes or counting modes. This is to ensure that 0 percent and 100 percent duty cycles can be generated.

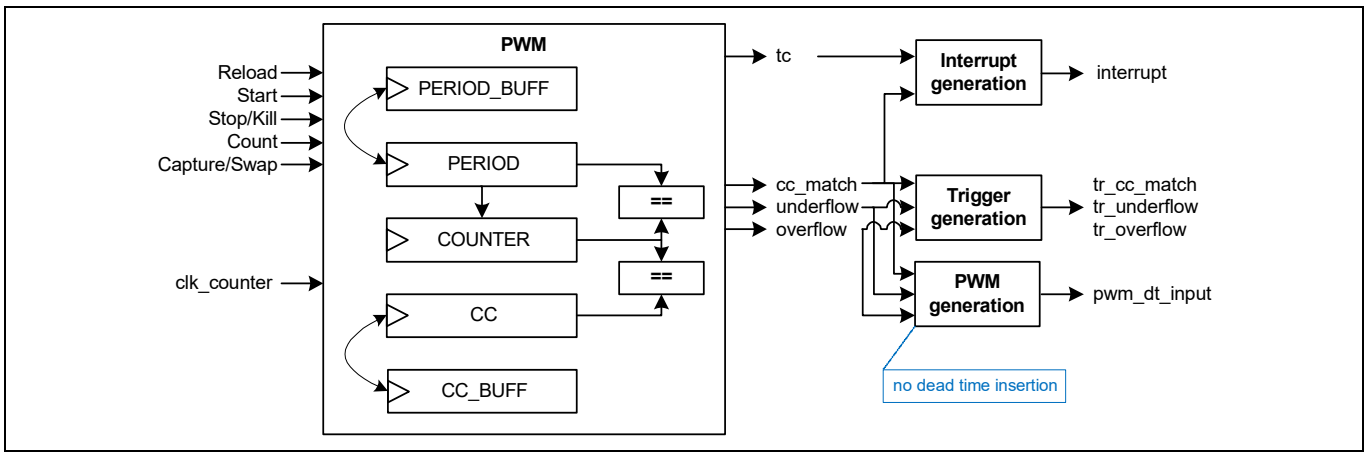


Figure 31-21. PWM mode functionality

The generation of the PWM output signals is a multi-step process and is illustrated in **Figure 31-22**.

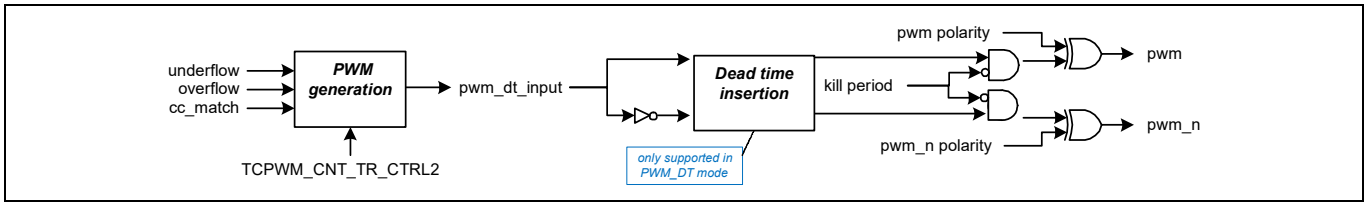


Figure 31-22. PWM output generation

PWM behavior depends on the PERIOD and CC registers. The software can update the PERIOD_BUFF and CC_BUFF registers, without affecting the PWM behavior. This is the main rationale for double buffering these registers.

Figure 31-23 illustrates a PWM in up counting mode. The counter is initialized (to 0) and started with a software-based reload event.

Note:

- When the counter changes from a state in which COUNTER is 4, an overflow and tc event are generated
- When the counter changes to a state in which COUNTER is 2, a cc_match event is generated
- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods

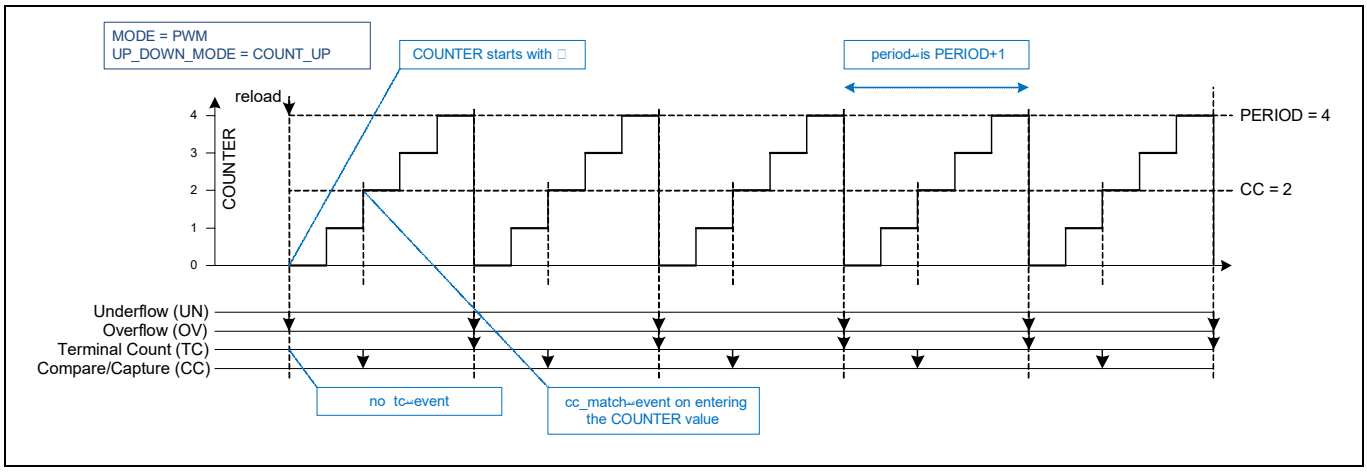


Figure 31-23. PWM in Up Counting mode

Figure 31-24 illustrates a PWM in up counting mode generating a left-aligned PWM. The figure also illustrates how a right-aligned PWM can be created using the PWM in up counting mode by inverting the OVERFLOW_MODE and CC_MATCH_MODE and using a CC value that is complementary (PERIOD+1 - pulse width) to the one used for left-aligned PWM. Note that CC is changed (to CC_BUFF, which is not depicted) on a tc event. The “pulse width” used as CC is the PWM output’s duty cycle x (PERIOD+1).

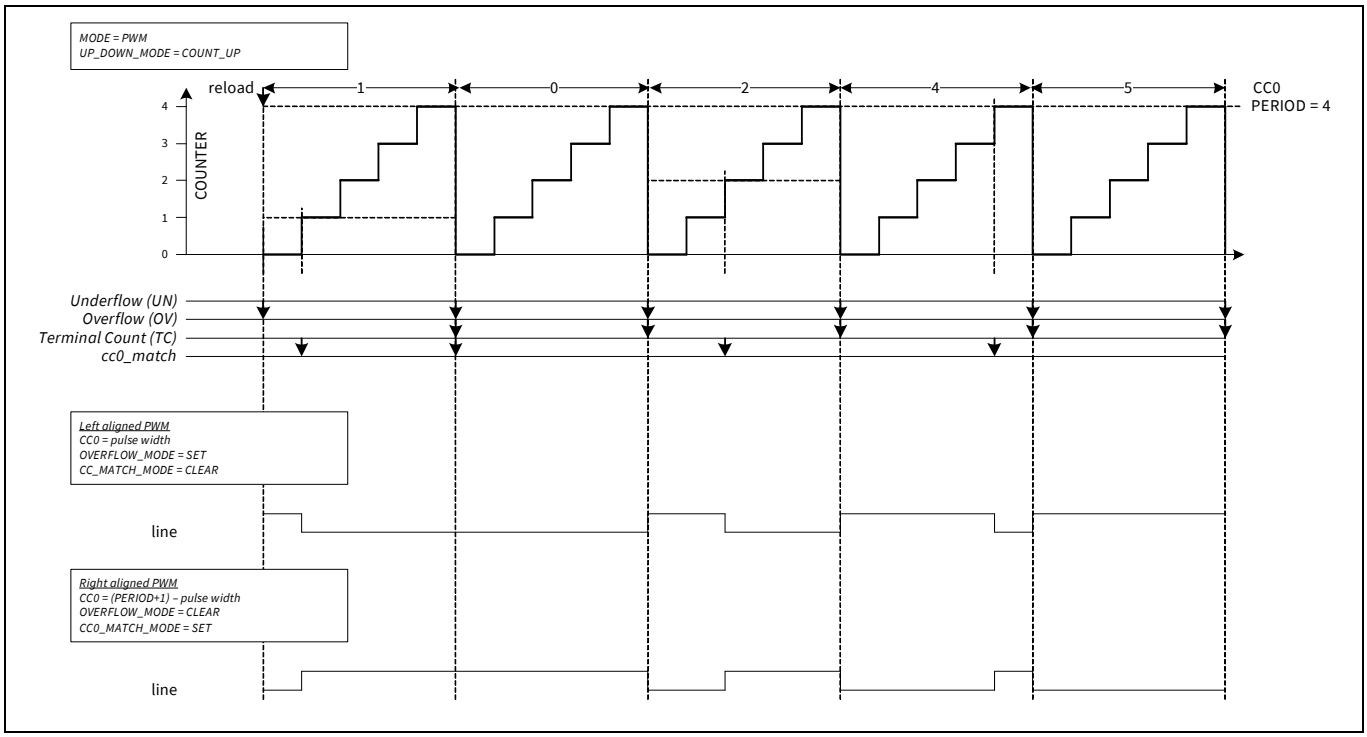


Figure 31-24. PWM left- and right-aligned outputs

Figure 31-25 illustrates a PWM in down counting mode. The counter is initialized (to PERIOD) and started with a software-based reload event.

Note:

- When the counter changes from a state in which COUNTER is 0, an underflow and tc event are generated
- When the counter changes to a state in which COUNTER is 2, cc_match event is generated
- PERIOD is 4, resulting in an effective repeating counter pattern of 4+1 = 5 counter clock periods

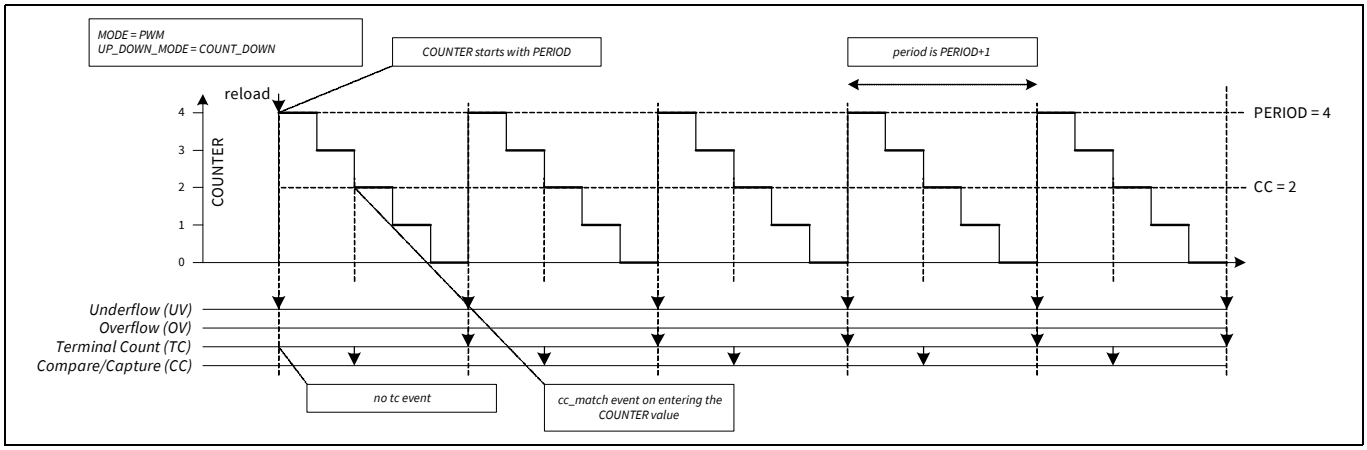


Figure 31-25. PWM in Down Counting mode

Figure 31-26 illustrates a PWM in down counting mode with different CC values. The figure also illustrates how a right-aligned PWM can be creating using the PWM in down counting mode. Note that the CC is changed (to CC_BUFF, which is not depicted) on a tc event.

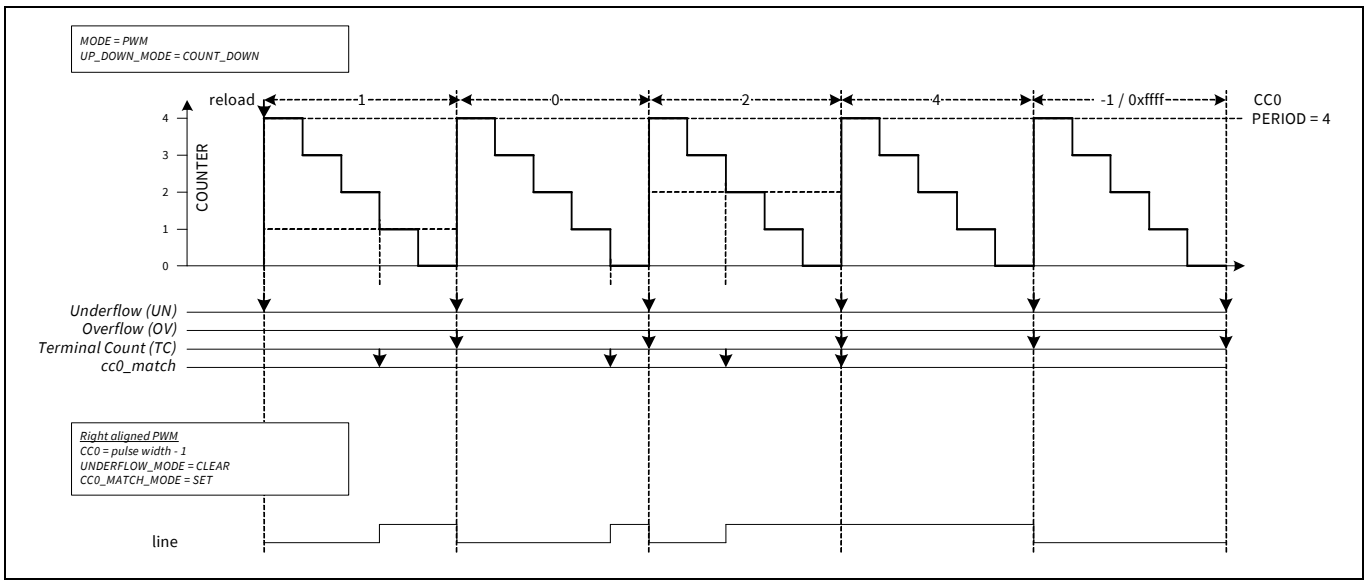


Figure 31-26. Right-aligned down counting PWM

Figure 31-27 illustrates a PWM in up/down counting mode. The counter is initialized (to 1) and started with a software-based reload event.

Note:

- When the counter changes from a state in which COUNTER is 4, an overflow is generated
- When the counter changes from a state in which COUNTER is 0, an underflow and tc event are generated
- When the counter changes from a state in which COUNTER is 2, a cc_match event is generated. Note that the actual counter value COUNTER from before the reload event is NOT used, instead the counter value before the reload event is considered to be 0.
- PERIOD is 4, resulting in an effective repeating counter pattern of 2*4 = 8 counter clock periods

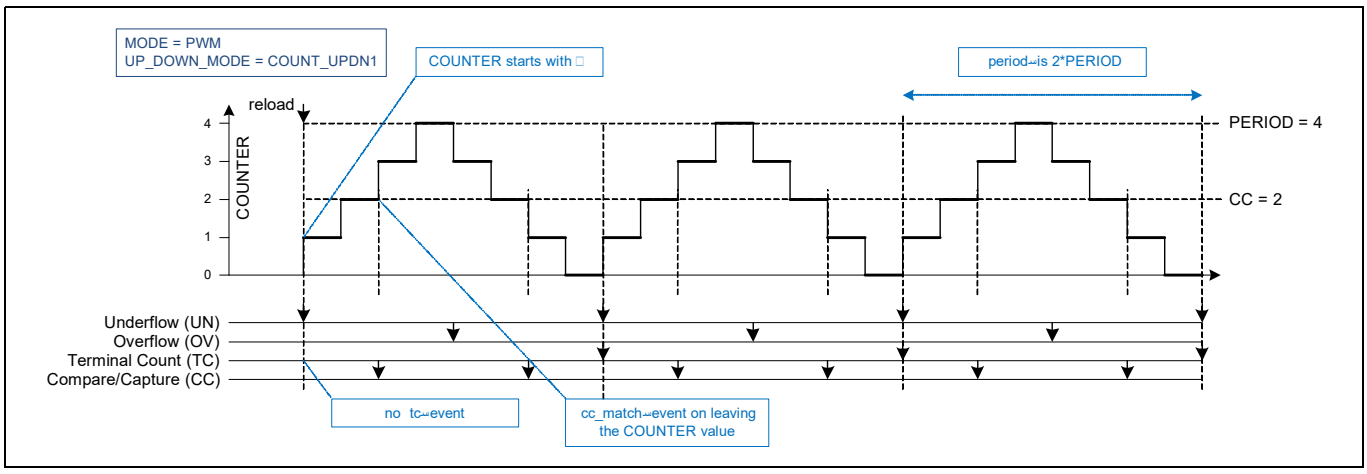


Figure 31-27. Up/down counting PWM

Figure 31-28 illustrates a PWM in up/down counting mode with different CC values. The figure also illustrates how a center-aligned PWM can be creating using the PWM in up/down counting mode.

Note:

- The actual counter value COUNTER from 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 CC value at the reload event is 0, a cc_match event is generated.
- CC is changed (to CC_BUFF, which is not depicted) on a tc event

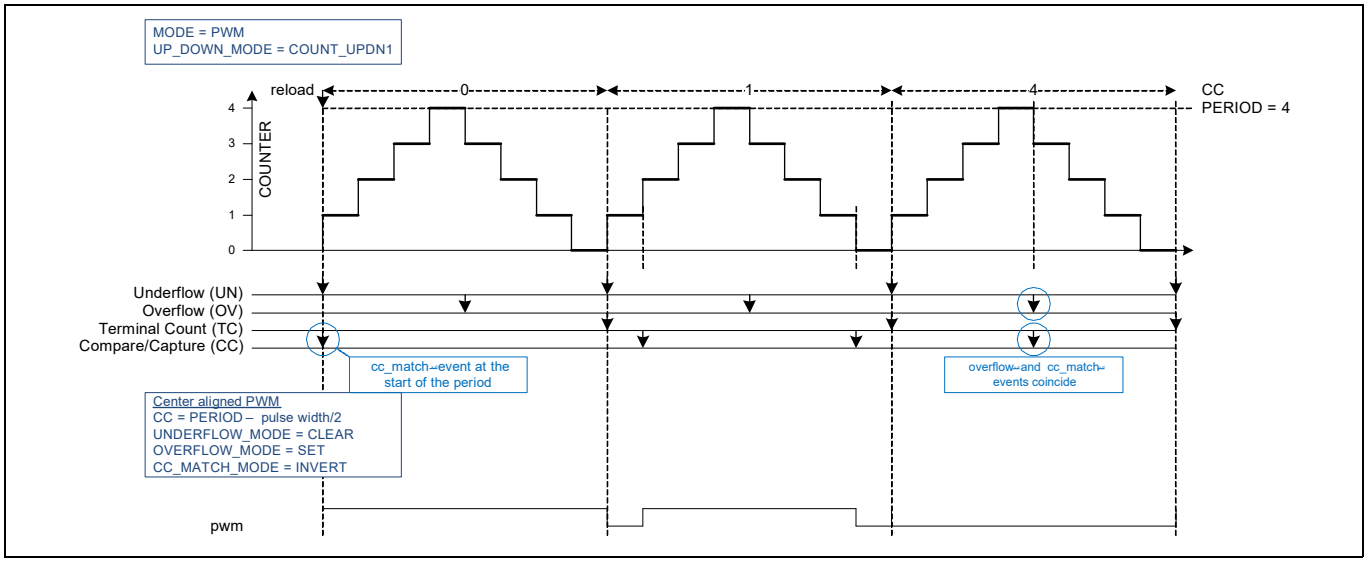


Figure 31-28. Up/down counting center-aligned PWM

Different stop/kill modes exist. The mode is specified by PWM_STOP_ON_KILL and PWM_SYNC_KILL.

The following three modes are supported:

- PWM_STOP_ON_KILL is '1' (PWM_SYNC_KILL is don't care): Stop on Kill mode. This mode stops the counter on a stop/kill event. Reload or start event is required to run the counter. Both software and external trigger input can be selected as stop kill. Edge detection mode is required.
- PWM_STOP_ON_KILL is '0' and PWM_SYNC_KILL is '0': Asynchronous Kill mode. This mode keeps the counter running, but suppresses the PWM output signals synchronously with the next count clock ("active count" pre-scaled clk_counter) and continues to do so for the duration of the stop/kill event. Only the external trigger input can be selected as asynchronous kill. Pass through detection mode is required.
- PWM_STOP_ON_KILL is '0' and PWM_SYNC_KILL is '1': Synchronous Kill mode. This mode keeps the counter running, but suppresses the PWM output signals synchronously with the next count clock ("active count" pre-scaled clk_counter) and continues to do so until the next tc event without a stop/kill event. Only the external trigger input can be selected as synchronous kill. Rising edge detection mode is required.

Figure 31-29, **Figure 31-30**. and **Figure 31-31** illustrate the above three modes.

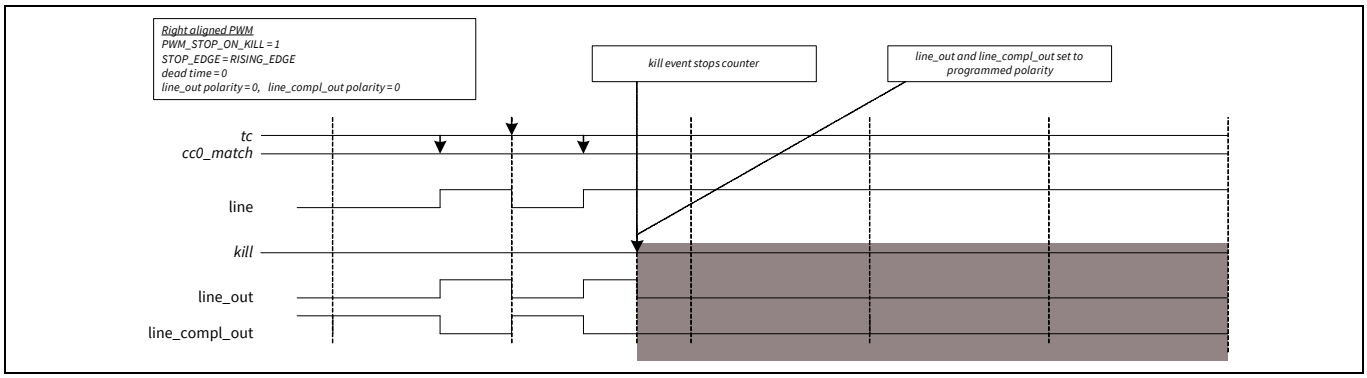


Figure 31-29. PWM Stop on Kill

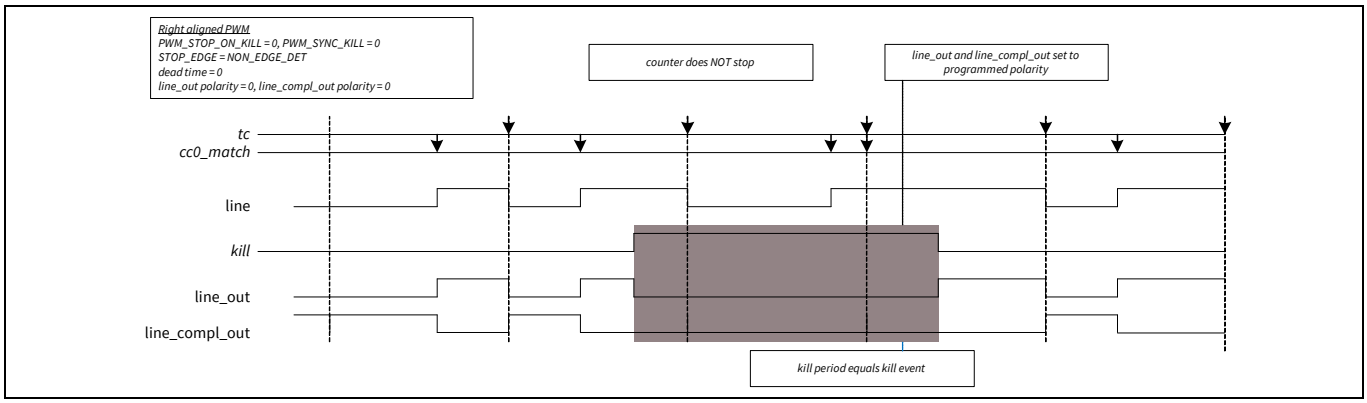


Figure 31-30. PWM Async Kill

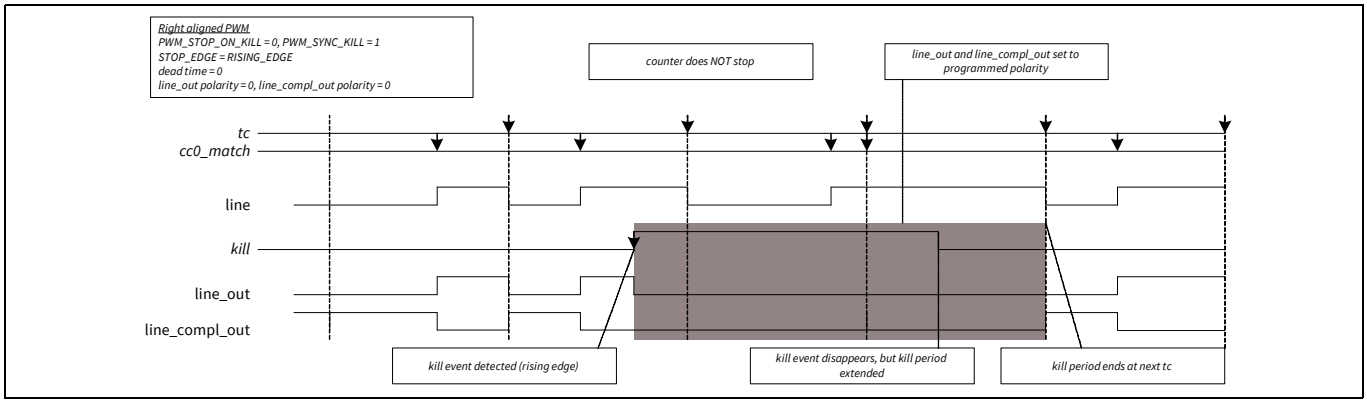


Figure 31-31. PWM Sync Kill

Figure 31-32 illustrates center-aligned PWM with PERIOD/PERIOD_BUFF and CC/CC_BUFF registers (up/down counting mode 1). At the TC condition, the PERIOD and CC registers are automatically exchanged with the PERIOD_BUFF and CC_BUFF registers. The swap event is generated by hardware trigger 1, which is a constant '1' and therefore always active at the TC condition. After the hardware exchange, the software handler on the tc interrupt cause updates PERIOD_BUFF and CC_BUFF.

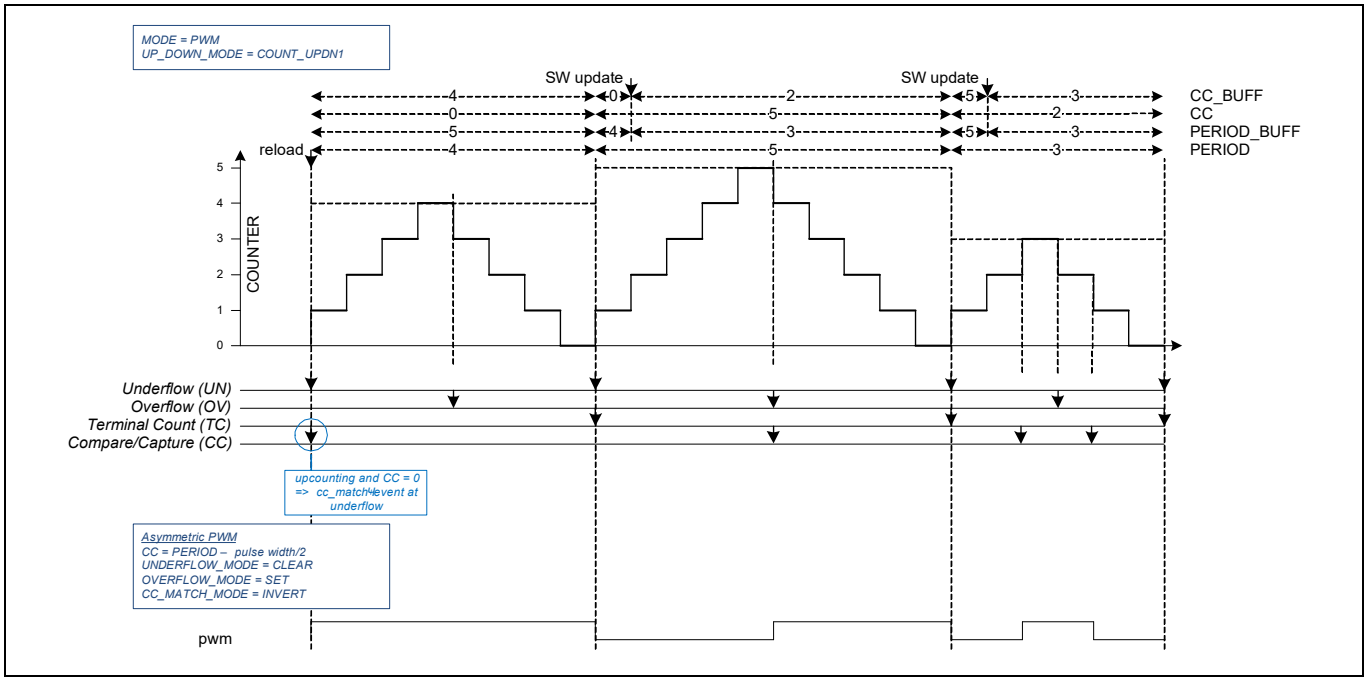


Figure 31-32. PWM mode CC Swap event

A potential problem arises when software updates are not completed before the next tc event with an active pending swap event. For example, if software updates PERIOD_BUFF before the tc event and CC_BUFF after the tc event, swapping does not reflect the CC_BUFF register update. To prevent this from happening, the swap event should be generated by software through a register write after both the PERIOD_BUFF and CC_BUFF registers are updated. The swap event is kept pending by the hardware until the next tc event occurs.

The previous section addressed synchronized updates of the CC/CC_BUFF and PERIOD/PERIOD_BUFF registers of a single PWM using a software-generated swap event. During motor control, three PWMs work in unison and updates to all period and compare register pairs should be synchronized. All three PWMs have synchronized periods and as a result have synchronized tc events. The swap event for all three PWMs is generated by software through a single register write. The software should generate the swap events after the PERIOD_BUFF and CC_BUFF registers of all three PWMs are updated.

Figure 31-22 uses CTRL.QUADRATURE_MODE[0] for "pwm" polarity and CTRL.QUADRATURE_MODE[1] for "pwm_n" polarity. **Figure 31-33** illustrates how the polarity settings control the PWM output signals "pwm" and "pwm_n".

Note: When the counter is not enabled or not running ((temporarily) stopped or killed), the PWM output signals values are determined by their respective polarity settings.

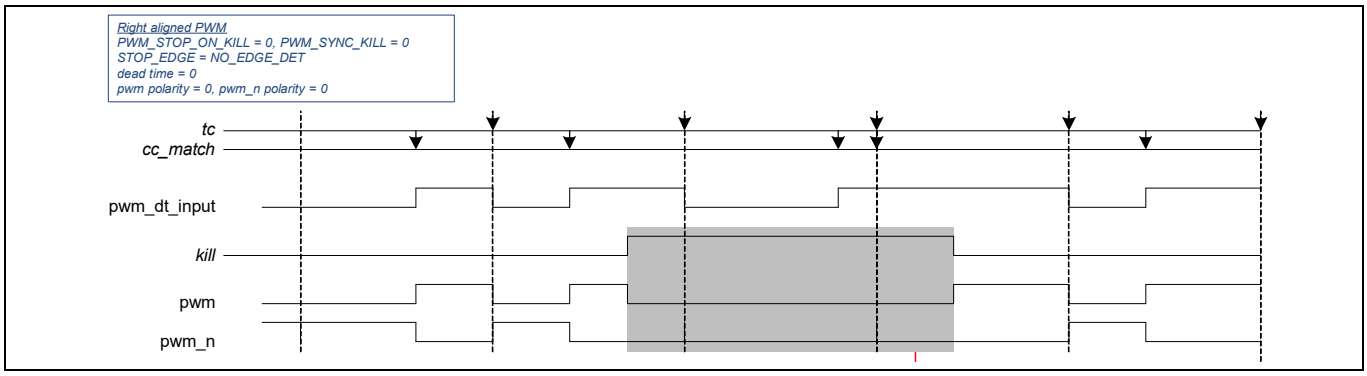


Figure 31-33. PWM outputs when killed

31.3.4.1 Asymmetric PWM

The PWM mode supports the generation of an asymmetric PWM. For an asymmetric PWM, the pwm_dt_input pulse is not necessarily centered in the middle of the period. This functionality is realized by having a different CC value when counting up and when counting down. The CC and CC_BUFF values are exchanged on an overflow event. Note that this restricts the asymmetry of the generated pwm_dt_input pulse.

The COUNT_UPDN2 mode should use the same period value when counting up and counting down. When PERIOD and PERIOD_BUFF are swapped on a tc event (overflow or underflow event), care should be taken to ensure that:

- Within a PWM period (tc event coincides with an overflow event), the period values are the same (an overflow swap of PERIOD and PERIOD_BUFF should not change the period value; that is, PERIOD_BUFF should be PERIOD)
- Between PWM periods (tc event coincides with an underflow event), the period value can change (an underflow swap of PERIOD and PERIOD_BUFF may change the period value; that is, PERIOD_BUFF may be different from PERIOD).

Figure 31-34 illustrates how the COUNT_UPDN2 mode is used to generate an asymmetric PWM.

Note:

- When up counting, when CC value at the underflow event is 0, cc_match event is generated
- When down counting, when CC value at the overflow event is PERIOD, a cc_match event is generated
- A tc event is generated for both an underflow and overflow event. The tc event is used to exchange the CC and CC_BUFF values.

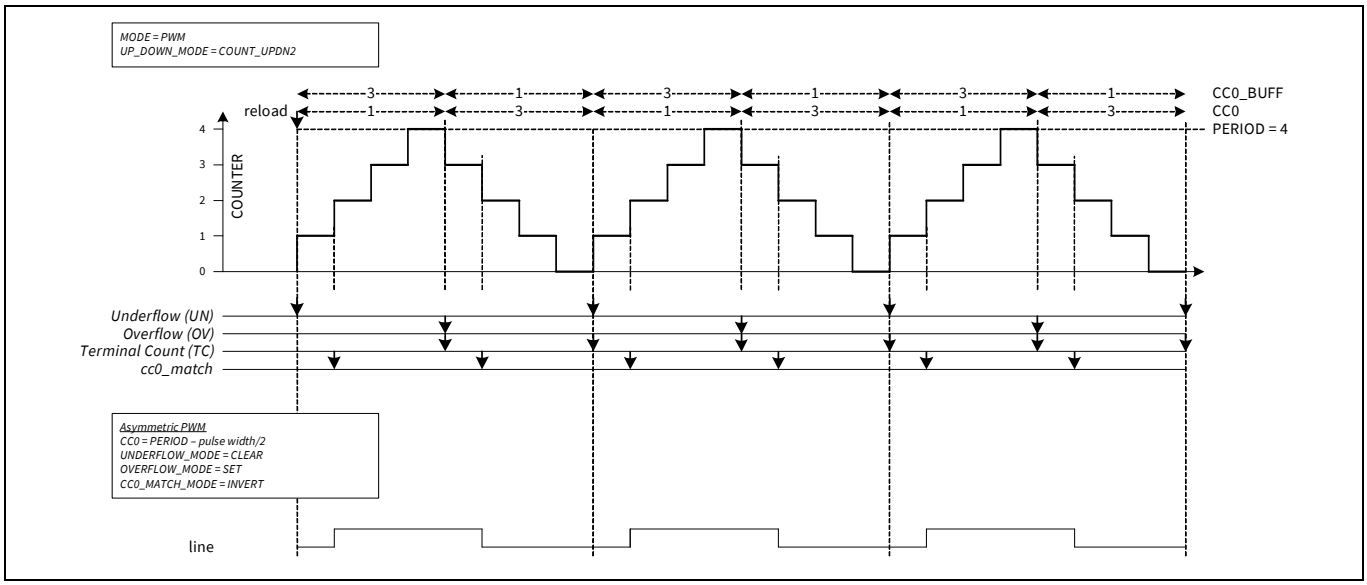


Figure 31-34. Asymmetric PWM

The previous waveform illustrated functionality when the CC values are neither “0” not PERIOD. Corner case conditions in which the CC values equal “0” or PERIOD are illustrated as follows.

Figure 31-35 illustrates how the COUNT_UPDN2 mode is used to generate an asymmetric PWM.

Note:

- When up counting, when CC value at the underflow event is 0, a cc_match event is generated
- When down counting, when CC value at the overflow event is PERIOD, a cc_match event is generated
- A tc event is generated for both an underflow and overflow event. The tc event is used to exchange the CC and CC_BUFF values.
- Software updates CC_BUFF and PERIOD_BUFF in an interrupt handler on the tc event (and overwrites the hardware updated values from the CC/CC_BUFF and PERIOD/PERIOD_BUFF exchanges)

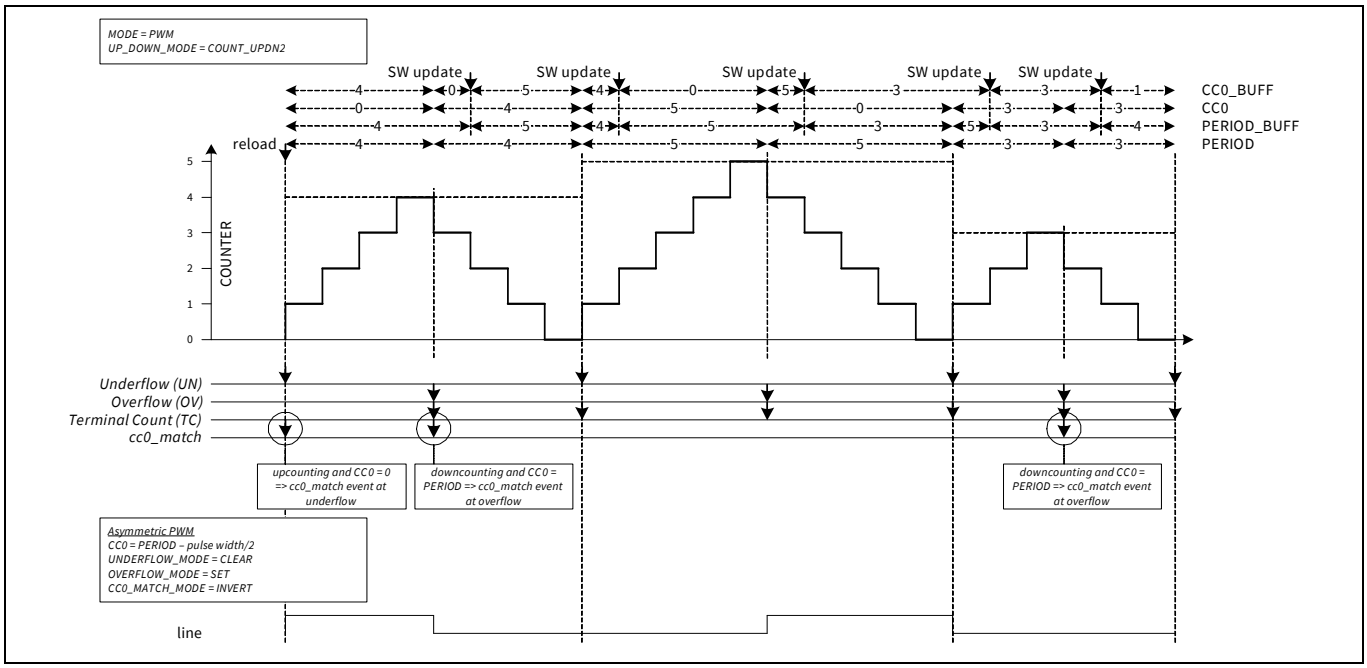


Figure 31-35. Asymmetric PWM when Compare = 0 or Period

31.3.4.2 Configuring counter for PWM mode

The steps to configure the counter for the PWM mode of operation and the affected register bits are as follows.

1. Disable the counter by writing '1' to the TCPWM_CTRL_CLR register.
2. Select PWM mode by writing '100b' to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set clock prescaling by writing to the GENERIC[15:8] field of the TCPWM_CNT_CTRL register.
4. Set the required 16-bit period in the TCPWM_CNT_PERIOD register and the buffer period value in the TCPWM_CNT_PERIOD_BUFF register to swap values, if required.
5. Set the 16-bit compare value in the TCPWM_CNT_CC register and buffer compare value in the TCPWM_CNT_CC_BUFF register to swap values, if required.
6. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register to configure left-aligned, right-aligned, or center-aligned PWM.
7. Set the PWM_STOP_ON_KILL and PWM_SYNC_KILL fields of the TCPWM_CNT_CTRL register as required.
8. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, kill, swap, and count).
9. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (reload, start, kill, swap, and count).
10. pwm and pwm_n can be controlled by the TCPWM_CNT_TR_CTRL2 register to set, reset, or invert upon CC, OV, and UN conditions.
11. If required, set the interrupt upon TC or CC condition.
12. Enable the counter by writing '1' to the TCPWM_CTRL_SET register. A start trigger must be provided through firmware (TCPWM_CMD_START register) to start the counter if the hardware start signal is not enabled.

31.3.5 Pulse width modulation with Dead Time mode

The PWM-DT functionality is the same as PWM functionality, except for the following differences:

- PWM_DT supports dead time insertion; PWM does not support dead time insertion
- PWM_DT does not support clock pre-scaling; PWM supports clock pre-scaling

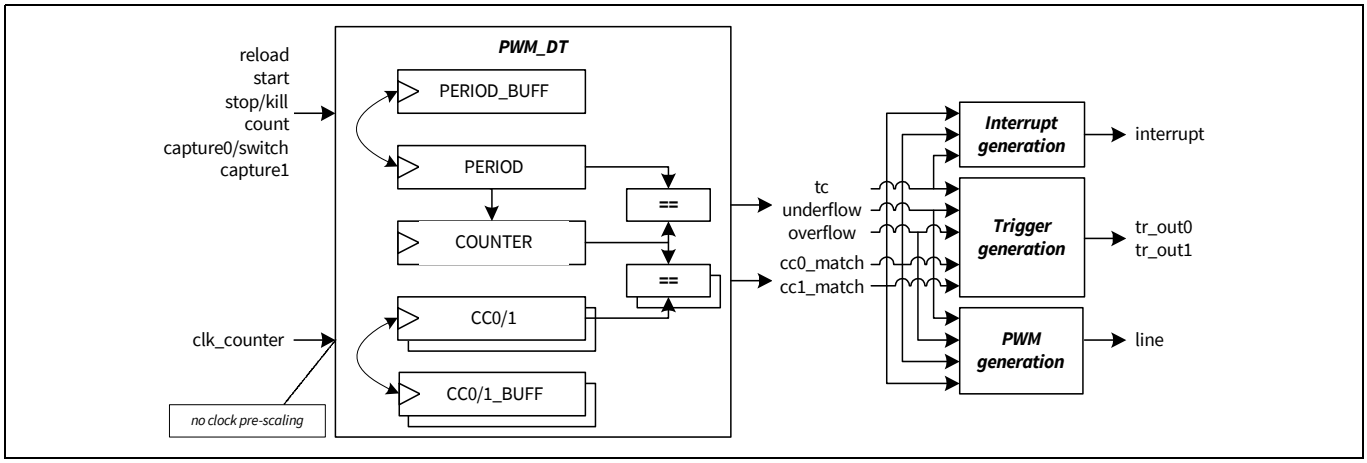


Figure 31-36. PWM with dead time functionality

Dead time insertion is a step that operates on a preliminary PWM output signal pwm_dt_input, as illustrated in **Figure 31-36**.

Figure 31-37 illustrates dead time insertion for different dead times and different output signal polarity settings.

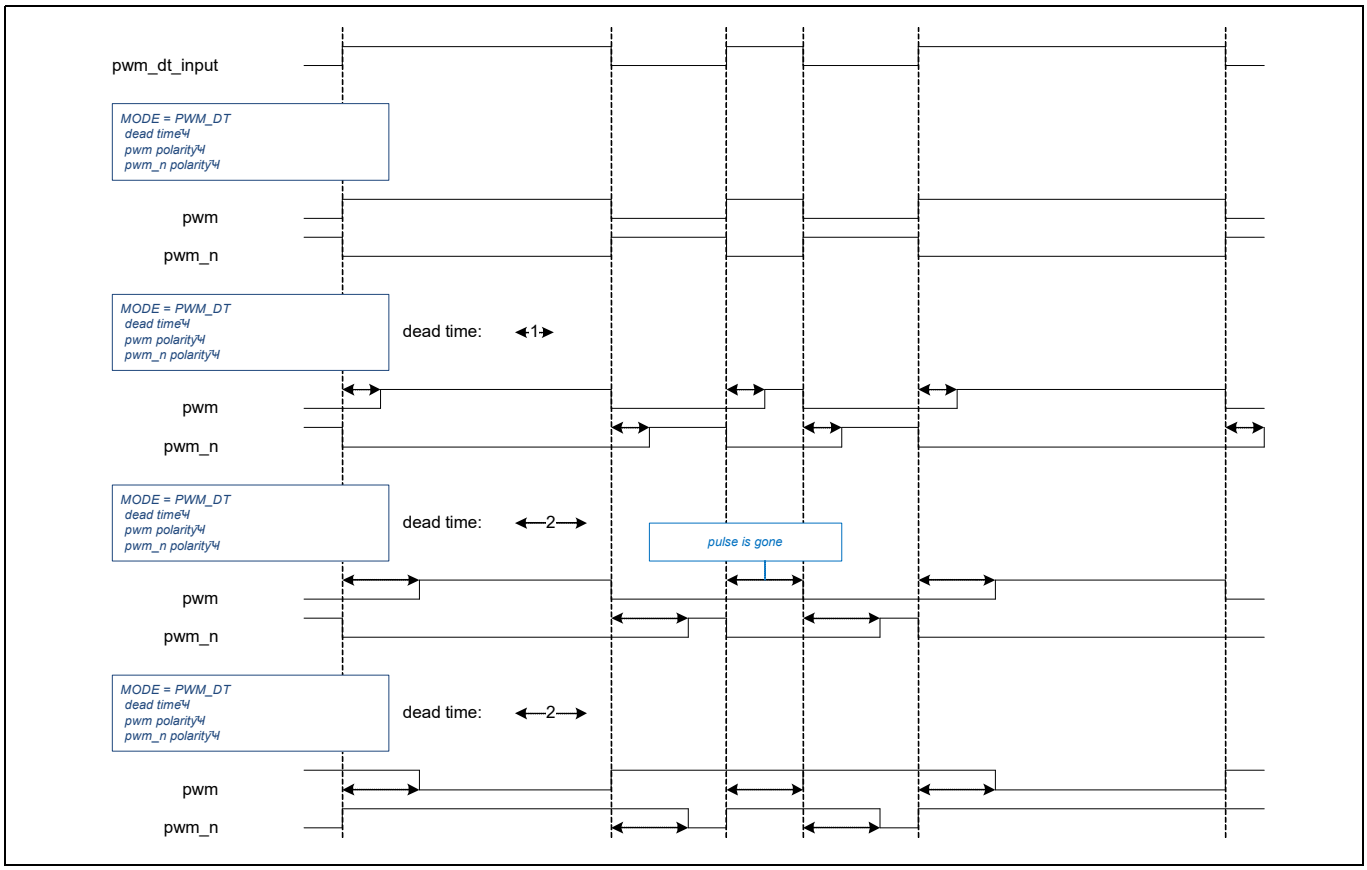


Figure 31-37. Dead-time timing

Figure 31-38 illustrates how the polarity settings and stop/kill functionality combined control the PWM output signals “pwm” and “pwm_n”.

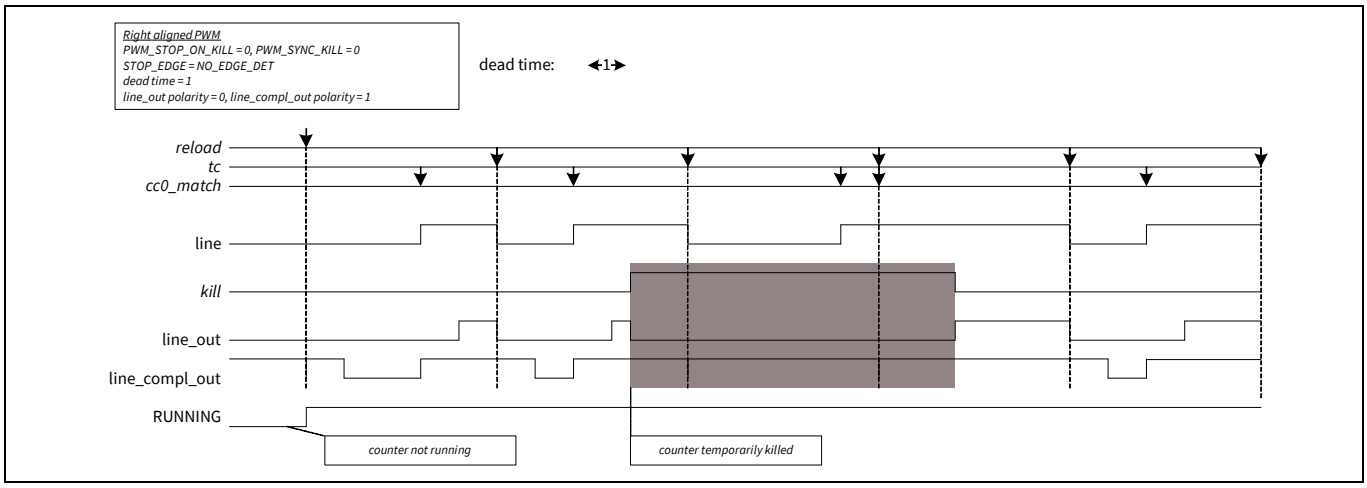


Figure 31-38. Dead Time and Kill

31.3.5.1 Configuring counter for PWM with Dead Time mode

The steps to configure the counter for PWM with Dead Time mode of operation and the affected register bits are as follows:

1. Disable the counter by writing ‘1’ to the TCPWM_CTRL_CLR register.
2. Select PWM with Dead Time mode by writing ‘101’ to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required dead time by writing to the GENERIC[15:8] field of the TCPWM_CNT_CTRL register.
4. Set the required 16-bit period in the TCPWM_CNT_PERIOD register and the buffer period value in the TCPWM_CNT_PERIOD_BUFF register to swap values, if required.
5. Set the 16-bit compare value in the TCPWM_CNT_CC register and the buffer compare value in the TCPWM_CNT_CC_BUFF register to swap values, if required.
6. Set the direction of counting by writing to the UP_DOWN_MODE[17:16] field of the TCPWM_CNT_CTRL register to configure left-aligned, right-aligned, or center-aligned PWM.
7. Set the PWM_STOP_ON_KILL and PWM_SYNC_KILL fields of the TCPWM_CNT_CTRL register as required.
8. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, kill, swap, and count).
9. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (reload, start, kill, swap, and count).
10. pwm and pwm_n can be controlled by the TCPWM_CNT_TR_CTRL2 register to set, reset, or invert upon CC, OV, and UN conditions.
11. If required, set the interrupt upon TC or CC condition.
12. Enable the counter by writing ‘1’ to the TCPWM_CTRL_SET register. A start trigger must be provided through firmware (TCPWM_CMD_START register) to start the counter if the hardware start signal is not enabled.

31.3.6 Pulse width modulation Pseudo-Random mode (PWM_PR)

The PWM_PR functionality changes the counter value using the linear feedback shift register (LFSR). This results in a pseudo random number sequence. A signal similar to PWM signal is created by comparing the counter value COUNTER with CC. The generated signal has different frequency/noise characteristics than a regular PWM signal.

Table 31-26. PWM_PR mode trigger inputs

Trigger inputs	Usage
reload	Same behavior as start event. Can be used only when the counter is not running.
start	Starts the counter. The counter is not initialized by hardware. The current counter value is used. Behavior is independent on UP_DOWN_MODE. Can be used only when the counter is not running.
stop/kill	Stops the counter. Different stop/kill modes exist.
count	Not used.
capture	This event acts as a swap event. When this event is active, the CC/CC_BUFF and PERIOD/PERIOD_BUFF registers are exchanged on a tc event (when specified by CTRL.AUTO_RELOAD_CC and CTRL.AUTO_RELOAD_PERIOD). A swap event requires rising, falling, or rising/falling edge event detection mode. Pass-through mode is not supported, unless the selected event is a constant '0' or '1'. When a swap event is detected and the counter is running, the event is kept pending until the next tc event. When a swap event is detected and the counter is not running, the event is cleared by hardware.

Note: Event detection is on the peripheral clock, clk_peri.

Table 31-27. PWM_PR supported features

Supported features	Description
Clock pre-scaling	Pre-scales the counter clock, clk_counter.
One-shot	Counter is stopped by hardware, after a single period of the counter (counter value equals period value PERIOD).
Auto reload CC	CC and CC_BUFF are exchanged on a swap event AND tc event (when specified by CTRL.AUTO_RELOAD_CC).
Auto reload PERIOD	PERIOD and PERIOD_BUFF are exchanged on a swap event and tc event (when specified by CTRL.AUTO_RELOAD_PERIOD).
Kill modes	Specified by PWM_STOP_ON_KILL. See memory map for further details.

Note: The count event is not used. As a result, the PWM_PR functionality operates on the pre-scaled counter clock (clk_counter), rather than on an "active count" pre-scaled counter clock.

Table 31-28. PWM_PR Trigger outputs

Trigger outputs	Description
cc_match (CC)	Counter changes from a state in which COUNTER equals CC.
Underflow (UN)	Not used.
Overflow (OV)	Not used.

Table 31-29. PWM_PR Interrupt outputs

Interrupt outputs	Description
cc_match (CC)	Counter changes from a state in which COUNTER equals CC.
tc	Counter changes from a state in which COUNTER equals PERIOD.

Table 31-30. PWM_PR PWM outputs

PWM outputs	Description
pwm	PWM output.
pwm_n	Complementary PWM output.

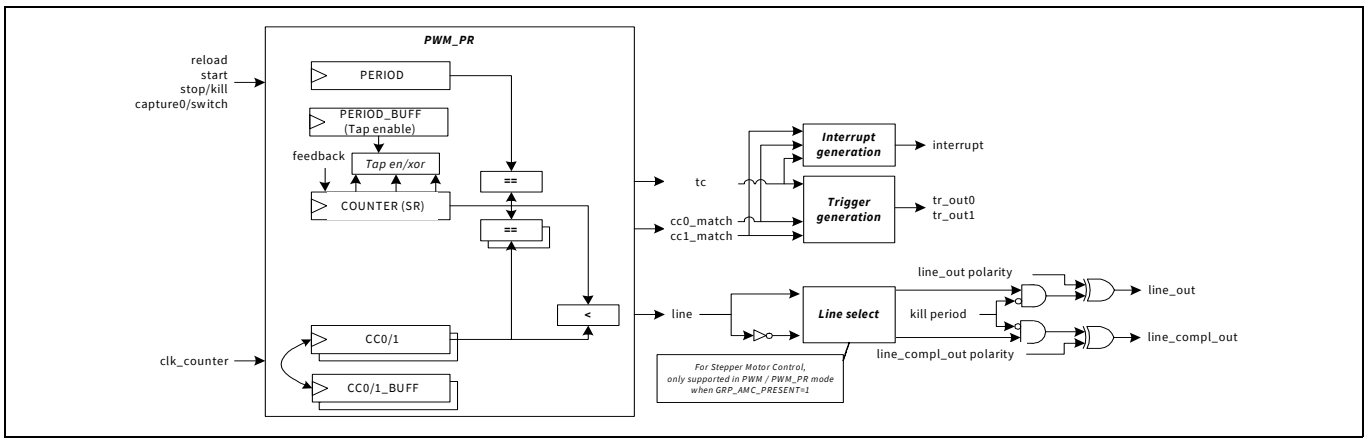


Figure 31-39. PWM_PR functionality

The PWM_PR functionality is described as follows:

- The counter value COUNTER is initialized by software (to a value different from 0).
- A reload or start event starts PWM_PR operation.
- During PWM_PR operation:
  - The counter value COUNTER is changed based on the LFSR polynomial: $x^{16} + x^{14} + x^{13} + x^{11} + 1$)

    temp = COUNTER [16-16] ^ COUNTER [16-14] ^ COUNTER [16-13] ^ COUNTER [16-11] or

    temp = COUNTER [0] ^ COUNTER [2] ^ COUNTER [3] ^ COUNTER [5];

    COUNTER = (temp << 15) | (COUNTER >> 1)

    This will result in a pseudo random number sequence for COUNTER. For example, when COUNTER is initialized to 0xACE1, the number sequence is: 0xACE1, 0x5670, 0xAB38, 0x559C, 0x2ACE, 0x1567, 0x8AB3…

    This sequence will repeat itself after $2^{16} - 1$ or 65535 counter clock cycles.
  - A 32-bit counter uses the LFSR polynomial: x^32 + x^30 + x^26 + x^25 + 1
  - When the counter value COUNTER equals CC, a cc_match event is generated
  - When the counter value COUNTER equals PERIOD, a tc event is generated
  - On a tc event, the CC/CC_BUFF and PERIOD/PERIOD_BUFF can be conditionally exchanged under control of the capture/swap event and the CTRL.AUTO_RELOAD_CC and CTRL.AUTO_RELOAD_PERIOD field (see PWM functionality).
  - The output pwm_dt_input reflects: COUNTER[14:0] < CC[15:0]. Note that only the lower 15 bits of COUNTER are used for comparison, while the COUNTER itself can run up to 16- or 32-bit values. As a result, for CC greater or equal to 0x8000, pwm_dt_input is always 1. The pwm polarity can be inverted (as specified by CTRL.QUADRATURE_MODE[0]).

As mentioned, different stop/kill modes exist. The mode is specified by PWM_STOP_ON_KILL (PWM_SYNC_KILL should be '0' – asynchronous kill mode). The memory map describes the modes and the desired settings for the stop/kill event. The following two modes are supported:

- PWM_STOP_ON_KILL is '1'. This mode stops the counter on a stop/kill event
- PWM_STOP_ON_KILL is '0'. This mode keeps the counter running, but suppresses the PWM output signals immediately and continues to do so for the duration of the stop/kill event

Note that the LFSR produces a deterministic number sequence (given a specific counter initialization value). Therefore, it is possible to calculate the COUNTER value after a certain number of LFSR iterations, n. This calculated COUNTER value can be used as PERIOD value, and the tc event will be generated after precisely n counter clocks. **Figure 31-40** illustrates PWM_PR functionality.

Note:

- The grey shaded areas represent the counter region in which the pwm_dt_input value is '1', for a CC value of 0x4000. There are two areas, because only the lower 15 bits of the counter value are used
- When CC is set to 0x4000, roughly one-half of the counter clocks will result in a pwm_dt_input value of '1'

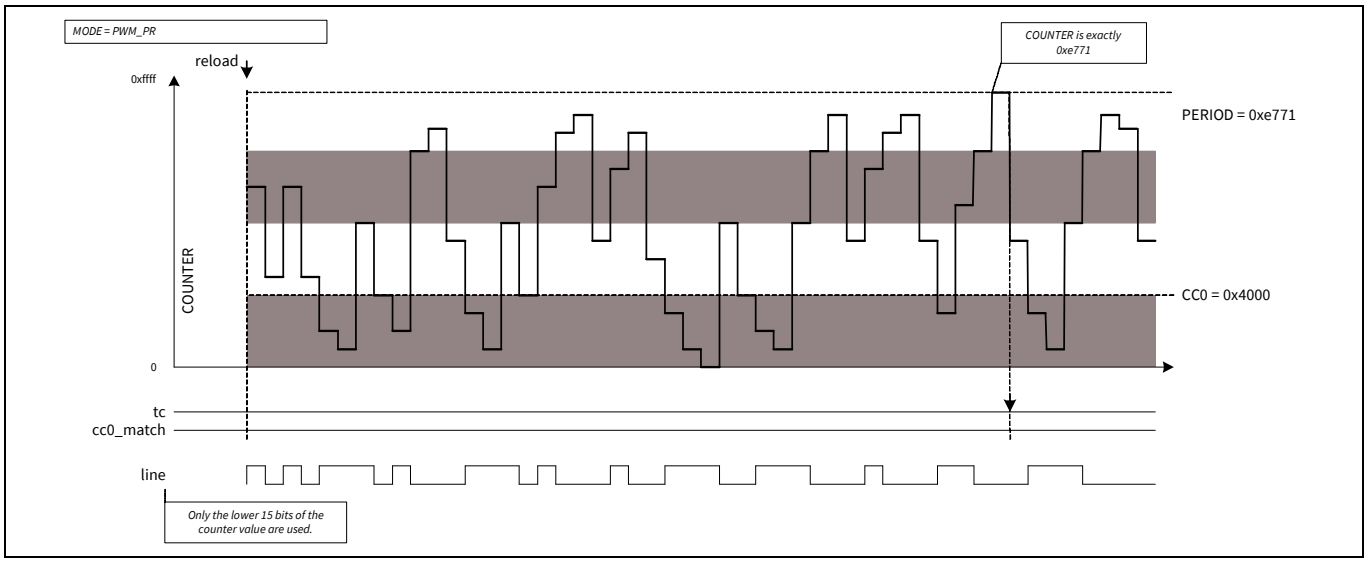


Figure 31-40. PWM_PR output

31.3.6.1 Configuring counter for Pseudo-Random PWM mode

The steps to configure the counter for pseudo-random PWM mode of operation and the affected register bits are as follows.

1. Disable the counter by writing ‘1’ to the TCPWM_CTRL_CLR register.
2. Select pseudo-random PWM mode by writing ‘110’ to the MODE[26:24] field of the TCPWM_CNT_CTRL register.
3. Set the required period (16 bit) in the TCPWM_CNT_PERIOD register and buffer period value in the TCPWM_CNT_PERIOD_BUFF register to swap values, if required.
4. Set the 16-bit compare value in the TCPWM_CNT_CC register and the buffer compare value in the TCPWM_CNT_CC_BUFF register to swap values.
5. Set the PWM_STOP_ON_KILL and PWM_SYNC_KILL fields of the TCPWM_CNT_CTRL register as required.
6. Set the TCPWM_CNT_TR_CTRL0 register to select the trigger that causes the event (reload, start, kill, and swap).
7. Set the TCPWM_CNT_TR_CTRL1 register to select the edge that causes the event (reload, start, kill, and swap).
8. pwm and pwm_n can be controlled by the TCPWM_CNT_TR_CTRL2 register to set, reset, or invert upon CC, OV, and UN conditions.
9. If required, set the interrupt upon TC or CC condition.
10. Enable the counter by writing ‘1’ to the TCPWM_CTRL_SET register. A start trigger must be provided through firmware (TCPWM_CMD_START register) to start the counter if the hardware start signal is not enabled.

31.4 TCPWM registers

Table 31-31. List of TCPWM registers

Register	Comment	Features
TCPWM_CTRL	TCPWM control register	Enables the counter block
TCPWM_CTRL_CLR	TCPWM control clear register	Used to avoid race-conditions on read-modify-write attempt to the CTRL register
TCPWM_CTRL_SET	TCPWM control set register	Used to avoid race-conditions on read-modify-write attempt to the CTRL register
TCPWM_CMD_CAPTURE	TCPWM capture command register	Generate a capture trigger input from software
TCPWM_CMD_RELOAD	TCPWM reload command register	Generate a reload trigger input from software
TCPWM_CMD_STOP	TCPWM stop command register	Generate a stop trigger input from software
TCPWM_CMD_START	TCPWM start command register	Generate a start trigger input from software
TCPWM_INTR_CAUSE	TCPWM counter interrupt cause register	Determines the source of the combined interrupt signal
TCPWM_CNT_CTRL	Counter control register	Configures counter mode, encoding modes, one-shot mode, swap mode, kill mode, dead time, clock pre-scaling, and counting direction
TCPWM_CNT_STATUS	Counter status register	Reads the direction of counting, dead time duration, and clock pre-scaling; checks whether the counter is running
TCPWM_CNT_COUNTER	Count register	Contains the 16- or 32-bit counter value
TCPWM_CNT_CC	Counter compare/capture register	Captures the counter value or compares the value with counter value
TCPWM_CNT_CC_BUFF	Counter buffered compare/capture register	Buffer register for counter CC register; swaps period value
TCPWM_CNT_PERIOD	Counter period register	Contains upper value of the counter
TCPWM_CNT_PERIOD_BUFF	Counter buffered period register	Buffer register for counter period register; swaps compare value
TCPWM_CNT_TR_CTRL0	Counter trigger control register 0	Selects trigger for specific counter events
TCPWM_CNT_TR_CTRL1	Counter trigger control register 1	Determine edge detection for specific counter input signals
TCPWM_CNT_TR_CTRL2	Counter trigger control register 2	Controls counter output lines upon CC, OV, and UN conditions
TCPWM_CNT_INTR	Interrupt request register	Sets the register bit when TC or CC condition is detected
TCPWM_CNT_INTR_SET	Interrupt set request register	Sets the corresponding bits in interrupt request register
TCPWM_CNT_INTR_MASK	Interrupt mask register	Mask for interrupt request register
TCPWM_CNT_INTR_MASKED	Interrupt masked request register	Bitwise AND of interrupt request and mask registers

32 Inter-IC sound bus

The Inter-IC sound bus (I^2S) is a serial bus interface standard used to connect digital audio devices together. The specification is from Philip Semiconductor (I^2S bus specification: February 1986, revised June 5, 1996). In addition to the standard I^2S format, the I^2S block also supports the Left Justified (LJ) format and the Time Division Multiplexed (TDM) format.

32.1 Features

- Supports standard Philips I^2S, LJ, and eight-channel TDM digital audio interface formats
- Supports both master and slave mode operation in all the digital audio formats
- Supports independent operation of Receive (RX) and Transmit (TX) directions
- Supports operating from an external master clock provided through an external IC such as audio codec
- Provides configurable clock divider registers to generate the required sample rates
- Supports data word length of 8-bit, 16-bit, 18-bit, 20-bit, 24-bit, and 32-bit per channel
- Supports channel length of 8-bit, 16-bit, 18-bit, 20-bit, 24-bit, and 32-bit per channel (channel length fixed at 32-bit in TDM format)644
- Provides two hardware FIFO buffers, one each for the TX block and RX block, respectively
- Supports both DMA- and CPU-based data transfers

32.2 Architecture

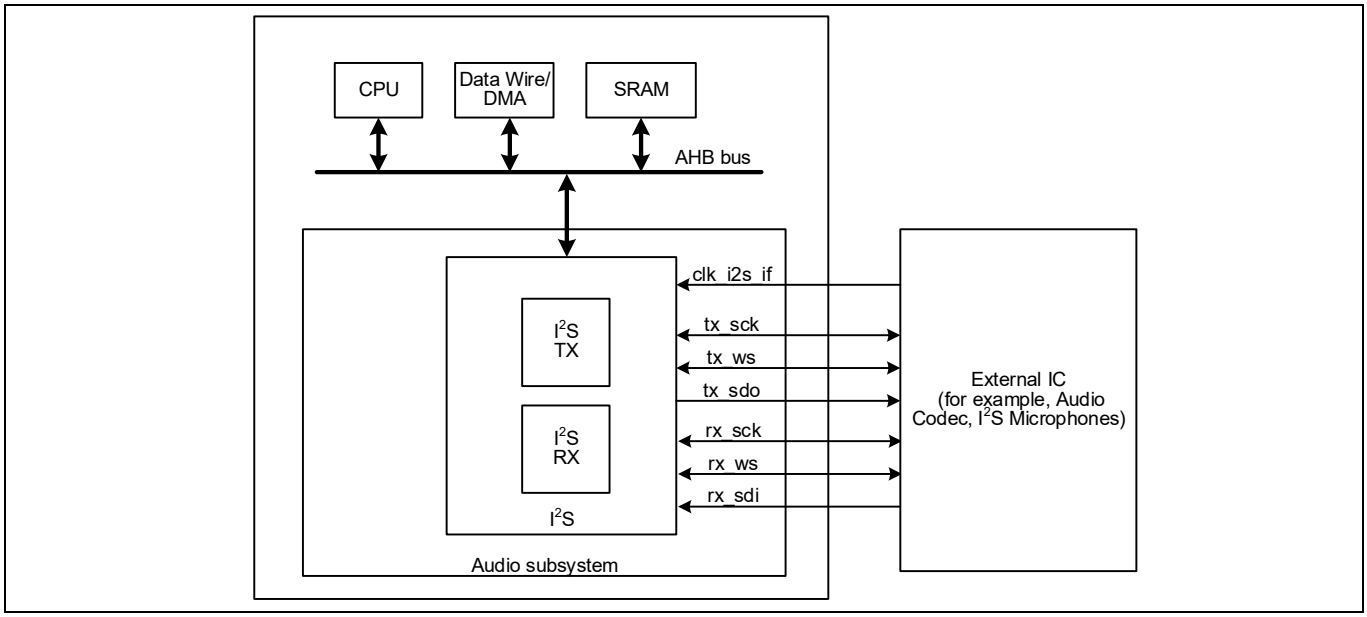


Figure 32-1. I^2S block diagram

Figure 32-1 shows the high-level block diagram of the I^2S block, which consists of two sub-blocks – I^2S Transmitter (TX) and I^2S Receiver (RX). The digital audio interface format and master/slave mode configuration can be done independently for the TX and RX blocks. In the master mode, the word select (ws) and serial data clock (sck) are generated by the I^2S block in the EZ-USB™ FX10 MCU. In the slave mode, the ws and sck signals are inputs signals to the EZ-USB™ FX10 MCU, and are generated by the external master device. The I^2S block configuration, control, and status registers, along with the FIFO data buffers are accessible through the AHB bus. AHB bus masters such as CPU and DMA can access the I^2S registers through the AHB interface. See the device datasheet for information on port pin assignments of the I^2S block signals and AC/DC electrical specifications.

32.3 Digital audio interface formats

The I²S block supports the following digital audio interface formats.

- Standard I²S format
- Left Justified 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 I2S_TX_CTL and I2S_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 I2S_TX_CTL and I2S_RX_CTL registers is used to configure the blocks in master or slave mode.

32.3.1 Standard I²S format

Figure 32-2 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 signal (ws) 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/right channel data. All the serial data (sd), ws signal transitions on the falling edge of the sck signal, and the read operations on the ws and sd lines are usually done on the rising edge of sck. Therefore, the I²S TX block writes to the serial data (tx_sdo) line on the falling edge of tx_sck, and the I²S RX block reads the 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/sck signals are either generated by the block (master mode) or 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 I2S_TX_CTL and I2S_RX_CTL registers, respectively. The channel length for the TX and RX blocks can be configured using the CH_LEN bits in the I2S_TX_CTL and I2S_RX_CTL registers respectively. The channel length configuration should always be greater than or equal to the word length configuration. Ensure that when the I²S RX block is operated in slave mode, the master TX device ensures that its channel length configuration aligns with the I²S RX block channel length setting. If there is channel length mismatch, the EZ-USB™ FX10 I²S RX block in slave mode will not operate correctly.

In the TX block, when the channel length is greater than the word length, the unused bits can be transmitted either as ‘0’ or ‘1’. This selection is made using the OVHDATA bit in the I2S_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 sign bit extended. This selection is made using the BIT_EXTENSION bit in the I2S_RX_CTL register.

Inter-IC sound bus
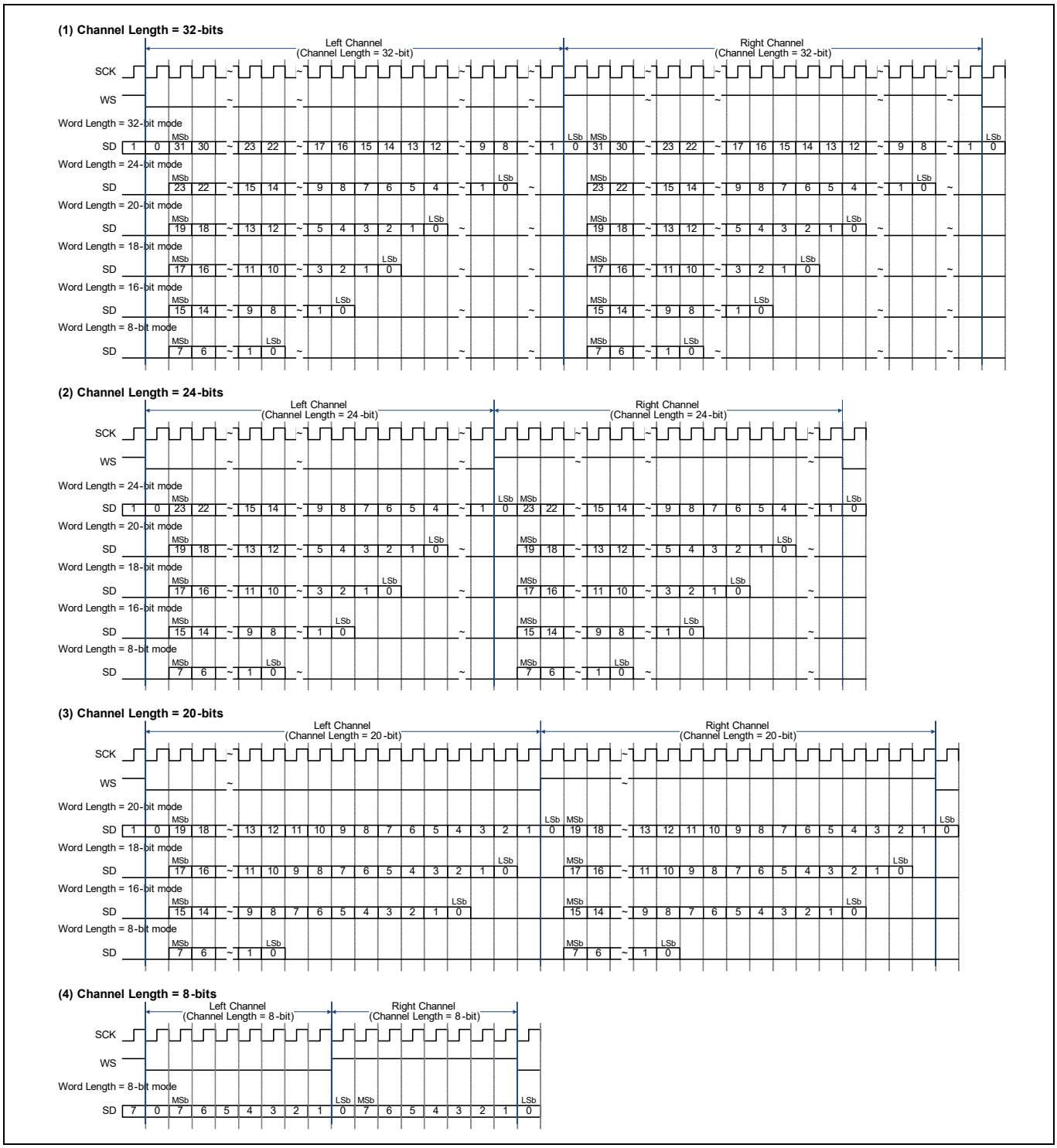


Figure 32-2. Standard I²S format (word length and channel length combination timing diagrams)

Table 32-1 lists the supported word length and channel length combinations.

Table 32-1. Word length and channel length combinations

		Word length					
		8-bit	**16-bit**	**18-bit**	**20-bit**	**24-bit**	**32-bit**
Channel length	32-bit	Valid	Valid	Valid	Valid	Valid	Valid
	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

32.3.2 Left Justified (LJ) format

Figure 32-3 shows the timing diagrams for the Left Justified 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:

- In the standard I²S format, WS signal is low for left channel data and high for right channel data. In the LJ format, WS signal is high for left channel data and low for right channel data.
- In the standard I²S format, WS signal transitions one bit-clock (sck) early relative to the start of the channel data (coincides with 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.

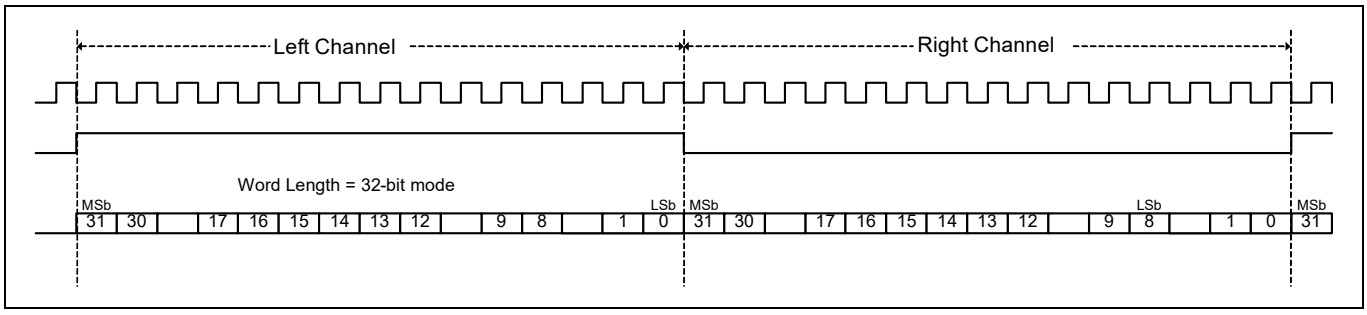


Figure 32-3. Left Justified digital audio format

32.3.3 Time division multiplexed (TDM) format

Figure 32-4 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/LJ formats and the TDM format are as follows:

- Standard I²S/LJ formats support only two channels (left/right) per frame, while TDM format supports up to eight channels per frame
- In the TDM format, channel length for all eight channels is fixed at 32 bits. In the standard I²S/LJ formats, the channel length is configurable. The word length per channel is configurable similar to the standard I²S and the data is also transmitted most significant bit first. Similar to I²S, when the word length per channel is less than the 32-bit channel length for TX block, the OVHDATA bit in the I2S_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, and thus the frame width is fixed at 256 bits. You have the option to configure the number of active channels in a frame by configuring the CH_NR bits in the I2S_TX_CTL and I2S_RX_CTL registers. In the standard I²S/LJ format, the CH_NR should always be configured for two channels. The number of active channels in the TDM format can be less than or equal to eight channels. The unused (inactive) channels always follow the active channels in a frame. As an example, if CH_NR is set for four channels, CH0 to CH3 are the active channels and CH4 to CH7 are the unused channels. The OVHDATA bit in the I2S_TX_CTL register is used to fill the unused channels with either 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 selection is made using the WS_PULSE bit in the I2S_TX_CTL and I2S_RX_CTL registers. The pulse width is fixed to one channel width in the I²S/LJ format.
- Two types of TDM formats are supported. In TDM mode A, the WS rising edge signal to signify the start of frame coincides with the start of CH0 data. In TDM mode B, the WS rising edge signal to signify the start of frame is one bit clock (sck) early, relative to the start of CH0 data (coincides with the last bit of the previous frame). The selection between the two TDM formats is made using the I2S_MODE bits in the I2S_TX_CTL and I2S_RX_CTL registers.

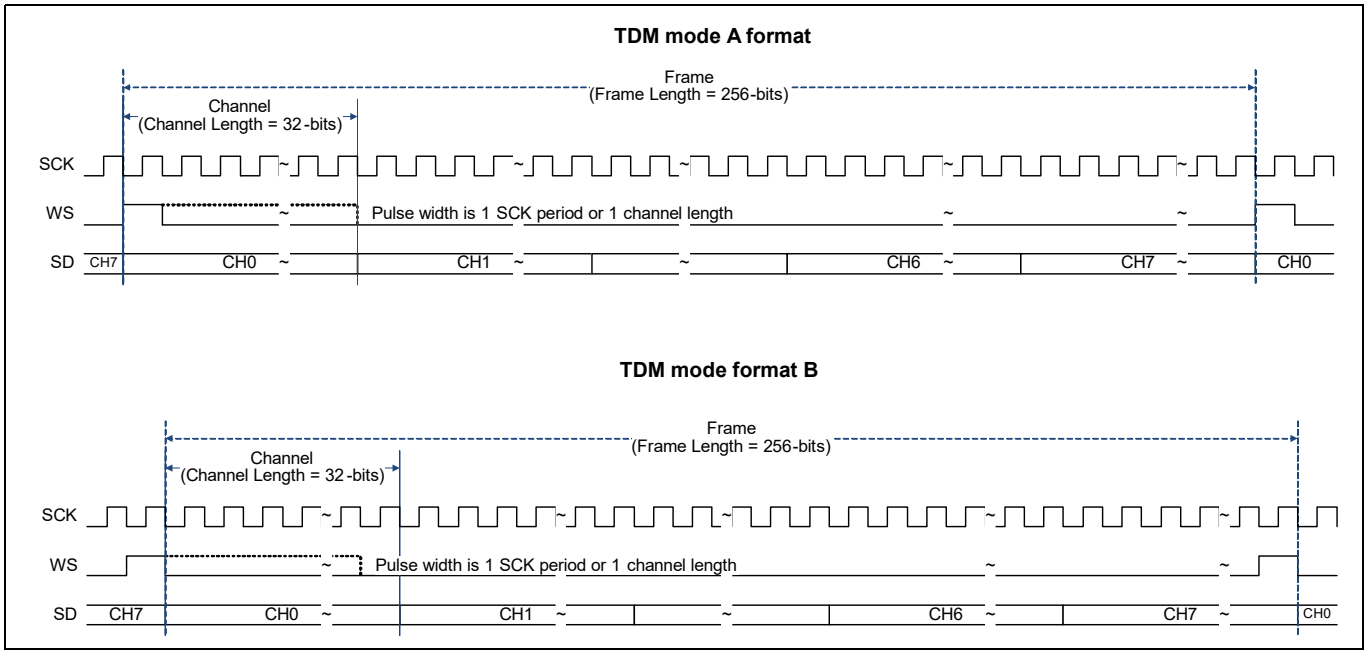


Figure 32-4. TDM Digital Audio Interface format

32.4 Clocking polarity and delay options

The I²S block supports configurable clock polarity and delay options to alleviate any timing issues in the system involving PCB signal propagation delays, and delays associated with internal device signal routing.

When the I²S TX block operates in the slave mode, the tx_sck and tx_ws signals are input signals to the EZ-USB™ FX10 MCU, and the tx_sdo output signal is transmitted off the tx_sck falling edge. The tx_sdo signal is sampled by the external master device RX block on the subsequent tx_sck rising edge. Timing issues arise if the tx_sdo signal reaching the master side Rx block does not meet the setup and hold time requirements for input data on the master side. The I²S TX block in the EZ-USB™ FX10 MCU has an option to advance the serial data transmission by 0.5 SCK cycles when the B_CLOCK_INV bit in the I2S_TX_CTL register is set. This feature can be used if there are timing issues while operating the I²S Tx block in slave mode.

Similarly, when the I²S RX block operates in the master mode, the rx_sck and rx_ws signals are output signals from the EZ-USB™ FX10 MCU, and the rx_sdi signal is transmitted by the external master device on the falling edge of rx_sck. The EZ-USB™ FX10 I2S Rx block samples the rx_sdi signal on the subsequent rx_sck rising edge. Timing issues arise if the rx_sdi signal reaching the EZ-USB™ FX10 block does not meet the setup and hold time requirements for input data. The I²S Rx block has an option to delay the serial data capture by 0.5 SCK cycles when the B_CLOCK_INV bit in the I2S_RX_CTL register is set. This feature can be used if there are timing issues while operating the I²S RX block in master mode.

In addition to these clock delay options, there is also an option to invert the outgoing bit clock (sck) in master mode by setting the SCKO_POL bit in the I2S_TX_CTL and I2S_RX_CTL registers. Similarly, in the slave mode, there is an option to invert the incoming bit clock (sck) by setting the SCKI_POL bit in the I2S_TX_CTL and I2S_RX_CTL registers.

Refer to the registers reference manual for detailed description of the B_CLOCK_INV, SCKI_POL, and SCKO_POL register configurations.

32.5 Interfacing with Audio Codecs

The I²S block in the EZ-USB™ FX10 MCU interfaces with an audio codec device based on the choice of codec device and the end application requirements. Some scenarios and the 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 EZ-USB™ FX10 I²S block and the codec device will be as shown in **Figure 32-1** where the EZ-USB™ FX10 I²S TX signals (tx_sck, tx_ws, tx_sdo) connect to the codec Rx signals, and the EZ-USB™ FX10 I²S RX signals (rx_sck, rx_ws, rx_sdi) connect to the codec TX signals. The direction of sck (tx_sck, rx_sck) and ws (tx_ws, rx_ws) signals depends on which device is the master and which device is the slave.
- Codecs with common ws and sck signals for both RX and TX directions: There are two possible configurations to interface these codecs with the EZ-USB™ FX10 MCU as shown in **Figure 32-5**. In both configurations, the sck signals (tx_sck, rx_sck, codec_sck) are shorted externally. The same goes for the ws signal connections as well (tx_ws, rx_ws, codec_ws). Ensure that only one block is driving the sck and ws lines. So when the codec acts as the slave device, the EZ-USB™ FX10 I²S RX block should be in the master mode, and the EZ-USB™ FX10 I²S TX block should be in the slave mode (or EZ-USB™ FX10 I²S RX as slave and EZ-USB™ FX10 I²S TX as master). When the codec acts as the master device, both the EZ-USB™ FX10 I²S RX and EZ-USB™ FX10 I²S TX blocks should be in slave mode.

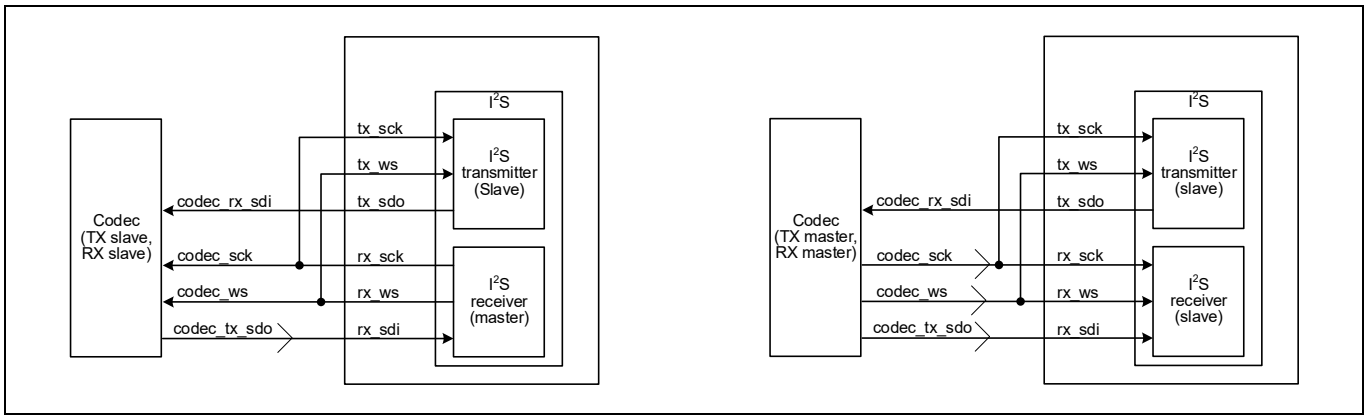


Figure 32-5. Interfacing with codecs having common ws and sck signals

32.6 Clocking features

The I²S unit has three clock inputs.

Table 32-2. Clock inputs

Signal	Description
clk_sys_i2s	System clock. This clock is used for the AHB slave Interface, control, status, and interrupt registers, and also clocks the DMA trigger control logic.
clk_audio_i2s	I²S internal clock. This clock is used for I²S transmitter (TX)/receiver (RX) blocks; it is asynchronous with the clk_sys_i2s. This clock is connected to the CLK_HF[1] high-frequency clock in the device. Refer to the **"Clocking system"** on page 290 for more details on high frequency clocks.
clk_i2s_if	I²S external clock. This clock is provided from an external I²S bus host through a port pin. It is used in place of the clk_audio_i2s clock to synchronize I²S data to the clock used by the external I²S bus host.

Figure 32-6 shows the clocking divider structure in the I²S block. In the master mode, the sck and ws signals are generated either using the clk_audio_i2s internal clock or the clk_i2s_if external clock. See the device datasheet for the port pin assignment of clk_i2s_if clock. The CLOCK_SEL bit in the I2S_CLOCK_CTL register controls the selection between internal and external clocks.

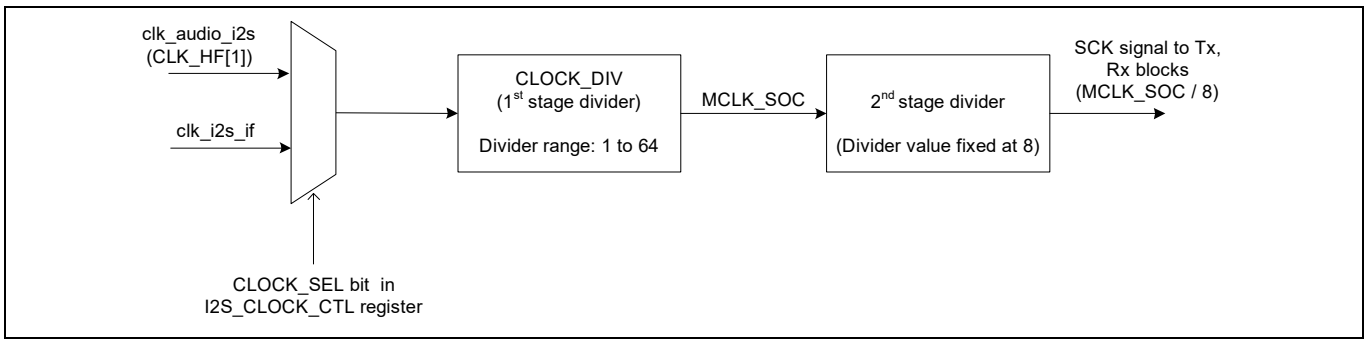


Figure 32-6. Clocking divider structure

Inter-IC sound bus

There are two stages of clock dividers in the I²S block as follows.

- The first stage clock divider is used to generate the internal I²S master clock (MCLK_SOC). The input clock to the first stage divider is either clk_audio_i2s or clk_i2s_if. The first stage clock divider is configured using the CLOCK_DIV bits in I2S_CLOCK_CTL register. Divider values from 1 to 64 are supported.
- The second stage clock divider is used to generate the sck signals. The input clock is the output from 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 in slave mode, the internal clock (MCLK_SOC) frequency should still be eight times the frequency of the input serial clock. You must choose the appropriate clock source and the CLOCK_DIV divider value to guarantee this condition is met in the slave mode of operation. Usually, when the EZ-USB™ FX10 I²S block operates in the slave mode, the host sends a master clock which 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 can then be adjusted to ensure that the MCLK_SOC is eight times the input SCK frequency.

Table 32-3 gives an example of the clock divider settings for operating the I²S block at the standard sampling rates in the standard I²S format. Note that the first stage divider values in the table are 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 maximum values of SCK frequency, and the output sampling rates.

Table 32-3. 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_HF3 (or clk_i2s_if) (MHz)	(CLK_HF[3])/ 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		48	5	
32	32	2.048		24	2	
48	32	3.072		16	1	
44.1	32	2.8224	45.1584	32	3	

32.7 FIFO buffer and DMA support

The I²S block has two FIFO buffers - one each for the Tx block and Rx block, respectively. The ordering format of the channel data in both Tx and Rx FIFOs depends on the configured digital audio format. This ordering format should be considered when writing to the Tx FIFO or reading from the Rx FIFO. In the standard I²S and LJ digital audio formats, the ordering of the data is (L, R, L, R, L, ...) where L refers to the left channel data and R refers to the right channel data. In the TDM format with the number of active channels set to four, the data order will be (CH0, CH1, CH2, CH3, CH0, CH1, CH2, CH3, CH0, ...). If the number of active channels is set to eight, the cycle will repeat after CH0–CH7 data.

I²S Tx FIFO: The I²S Tx block has a hardware FIFO of depth 256 elements where each element is 32-bit wide. In addition to this 256-element FIFO, the I²S block has an internal transmit buffer that can store four 32-bit data to be transmitted. This four-element buffer is used as an intermediary to hold data to be transferred on the I²S bus, and is not exposed to the AHB BUS interface.

The TX FIFO can be paused by setting the TX_PAUSE bit in I2S_CMD. When the TX_PAUSE bit is set, the data sent over I²S is "0", instead of TX FIFO data. To resume normal operation, the TX_PAUSE bit must be cleared.

The I2S_TX_FIFO_CTL register is used for FIFO control operations. The TRIGGER_LEVEL bits in the I2S_TX_FIFO_CTL register can be used to generate a transmit trigger event when the Tx FIFO has less entries than the value configured in the TRIGGER_LEVEL bits.

The FIFO freeze operation can be enabled by setting the FREEZE bit in the I2S_TX_FIFO_CTL register. When the FREEZE bit is set and the Tx block is operational (TX_START bit in I2S_CMD is set), hardware reads from the Tx FIFO do not remove the FIFO entries. Also, the Tx FIFO read pointer will not be advanced. Any writes to the I2S_TX_FIFO register will increment the Tx FIFO write pointer; when the Tx FIFO becomes full, the internal write pointer stops incrementing. The freeze operation may be used for firmware debug purposes. This operation is not intended for normal operation. To return to normal operation after using the freeze operation, the I^2S must be reset by clearing the TX_ENABLED bit in the I2S_CTL register, and then setting the bit again.

The CLEAR bit in the I2S_TX_FIFO_CTL register is used to clear the Tx FIFO by resetting the read/write pointers associated with the FIFO. Write accesses to the Tx FIFO using the I2S_TX_FIFO_WR or I2S_TX_FIFO_WR_SILENT registers are not allowed while the CLEAR bit is set.

The I2S_TX_FIFO_STATUS register provides FIFO status information. This includes number of used entries in the Tx FIFO and the current values of the Tx FIFO read/write pointers. This register can be used for debug purposes. The I^2S Tx FIFO read pointer is updated whenever the data is transferred from the Tx FIFO to the internal transmit buffer. Tx FIFO write pointer is updated whenever the data is written to the I2S_TX_FIFO_WR register, either through the CPU or the DMA controller.

For Tx FIFO data writes using the CPU, the hardware can be used to trigger an interrupt event for any of the 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 I2S_TX_FIFO_WR register every time the TX_TRIGGER interrupt event is triggered. In addition, interrupt events can be generated for FIFO overflow/underflow conditions.

For DMA-based Tx data transfers, the I^2S Tx DMA trigger signal (tr_i2s_tx_req) can be enabled by writing '1' to the TX_REQ_EN bit in I2S_TR_CTL register. The trigger signal output will become high whenever the Tx FIFO has less entries than that configured in the TRIGGER_LEVEL field. The DMA channel can be configured to transfer up to (256 - TRIGGER_LEVEL) words from the applicable source address (such as Flash and SRAM regions). The destination address of the DMA should always be the I2S_TX_FIFO_WR register address, with the destination address increment feature disabled in the DMA channel configuration. This FIFO address increment logic is handled internally to adjust the write pointer, and the DMA should not increment the destination address. For more details on DMA channel configuration, refer to the **"Data wire controller (DW)"** on page 141.

The data in the I2S_TX_FIFO is always right-aligned. The I2S_TX_FIFO_WR format for different word length configurations is provided in **Figure 32-7**.

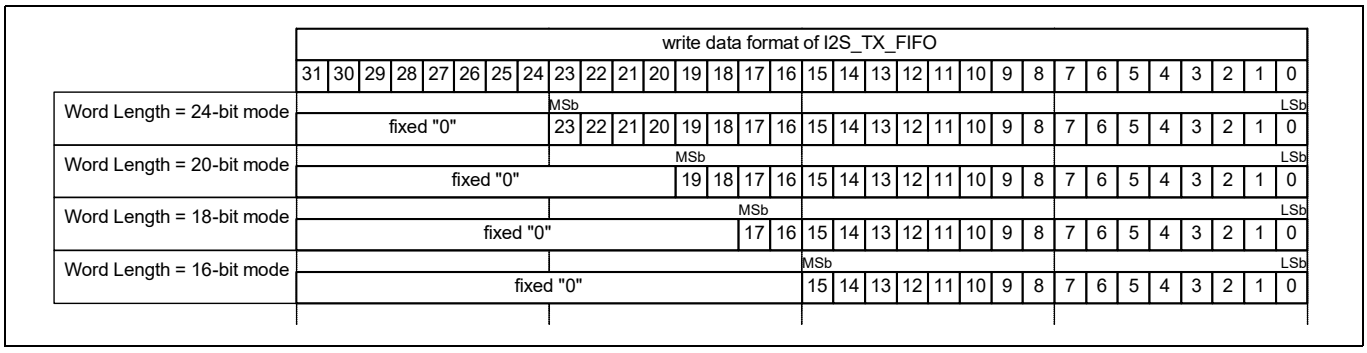


Figure 32-7. I2S_TX_FIFO_WR Register Format for Different Word Lengths

Inter-IC sound bus

I²S Rx FIFO: The I²S Rx block has a hardware FIFO of depth 256 elements where each element is 32-bit wide. In addition to this 256-element FIFO, the I²S block has an internal receive buffer that can store four 32-bit data to be received. This four-element buffer is used as an intermediary to hold data received on the I²S bus, and is not exposed to the AHB BUS interface.

The I2S_RX_FIFO_CTL register is used for FIFO control operations. The TRIGGER_LEVEL bits in the I2S_RX_FIFO_CTL register is 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/LJ format, the TRIGGER_LEVEL bits can be configured up to the allowed maximum value of 253. In the TDM format, the maximum value of 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 I2S_RX_FIFO_CTL register. When the FREEZE bit is set and the Rx block is operational (RX_START bit in the I2S_CMD register is set), hardware will not write to the Rx FIFO. Also, the Rx FIFO write pointer will not be advanced. Any reads from the I2S_RX_FIFO register will increment the Rx FIFO read pointer; when the Rx FIFO becomes empty, the internal read pointer stops incrementing. The freeze operation may be used for firmware debug purposes. This operation is not intended for normal operation. To return to normal operation after using the freeze operation, the I²S must be reset by clearing the RX_ENABLED bit in the I2S_CTL register and then setting the bit again.

The CLEAR bit in I2S_RX_FIFO_CTL register is used to clear the Rx FIFO by resetting the read/write pointers associated with the FIFO. Read accesses from the Rx FIFO using the I2S_RX_FIFO_RD or I2S_RX_FIFO_RD_SILENT registers are not allowed while the CLEAR bit is set.

The I2S_RX_FIFO_STATUS register provides FIFO status information. This includes number of used entries in the Rx FIFO and the current values of the Rx FIFO read/write pointers. This register can be used for debug purposes. The I²S Rx FIFO write pointer is updated whenever the data is transferred to the Rx FIFO from the internal receive buffer. Rx FIFO read pointer is updated whenever the data is read from the I2S_RX_FIFO_RD register, either through the CPU or the DMA controller. For debug purposes, the I2S_RX_FIFO_RD_SILENT register is available, which always returns the top element of the Rx FIFO without updating the read pointer.

For Rx FIFO data reads using the CPU, the hardware can be used to trigger an interrupt event for any of the FIFO conditions such as RX_TRIGGER, RX_NOT_EMPTY, and RX_FULL. As part of the interrupt handler, the CPU can read from the I2S_RX_FIFO_RD register. The recommended method is to read (TRIGGER_LEVEL + 1) words from the I2S_RX_FIFO_RD register every time the RX_TRIGGER interrupt event is triggered. In addition, interrupt events can be generated for FIFO overflow/underflow conditions.

For DMA-based Rx data transfers, the I²S Rx DMA trigger signal (tr_i2s_rx_req) can be enabled by writing '1' to the RX_REQ_EN bit in the I2S_TR_CTL register. The trigger signal output will become high whenever the Rx FIFO has more entries than that configured in the TRIGGER_LEVEL field. The DMA channel can be configured to transfer up to (TRIGGER_LEVEL + 1) words to the applicable destination address (such as SRAM regions). The source address of the DMA should always be the I2S_RX_FIFO_RD register address, with the source address increment feature disabled in the DMA channel configuration. This FIFO address increment logic is handled internally to adjust the read pointer, and the DMA should not increment the source address. For more details on DMA channel configuration, refer to the **"Data wire controller (DW)"** on page 141.

The data in the I2S_RX_FIFO is always right aligned. The I2S_RX_FIFO_RD format for different word length configurations is provided in **Figure 32-8**. Note that the unused most significant bits are either set as '0' or sign-bit extended depending on the BIT_EXTENSION bit in the I2S_RX_CTL register.

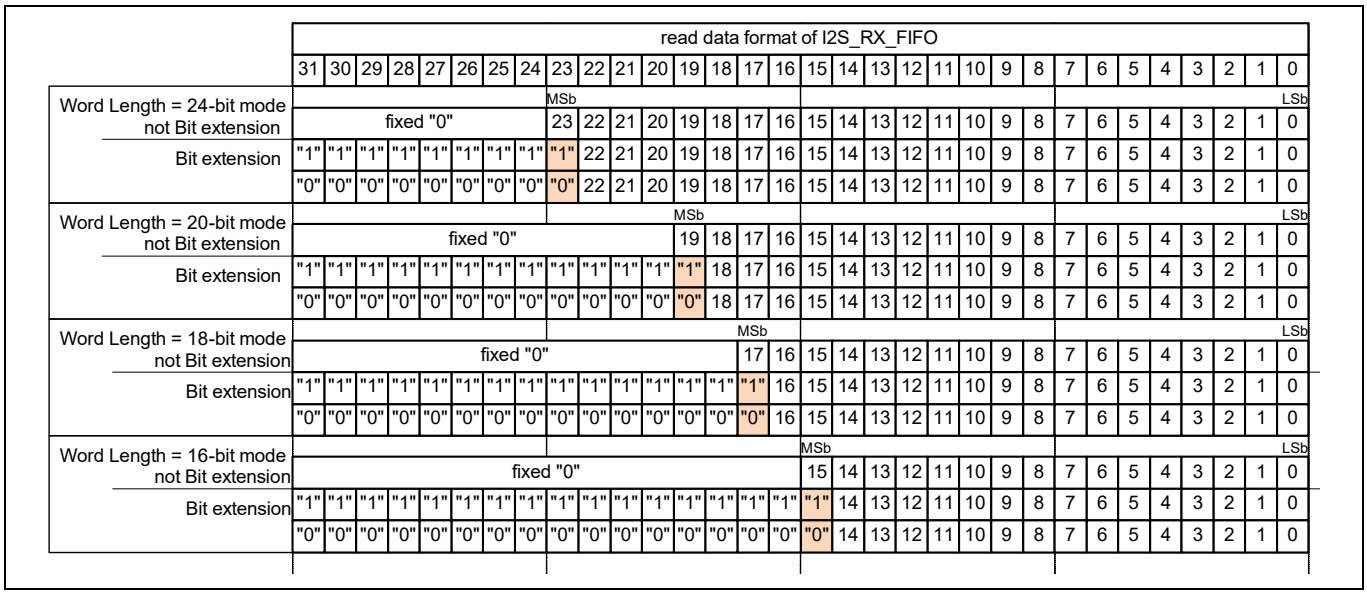


Figure 32-8. I2S_RX_FIFO_RD Register Format for Different Word Lengths

32.8 Interrupt support

The I²S block has one interrupt output signal that goes to the interrupt controller in the CPU. Refer to the **"Interrupts"** on page 89 for details on the vector number of the I²S interrupt and the procedure to configure the interrupt priority, vector address, and enabling/disabling. The I²S interrupt can be triggered under 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, or RX_WD. Each of the interrupt events can be individually enabled/disabled to generate an interrupt condition. The I2S_INTR_MASK register is used to enable the required events by writing '1' to the corresponding bit. Irrespective of the INTR_MASK settings, if any of the events occur, the corresponding event status bit will be set by the hardware in the I2S_INTR register. The I2S_INTR_MASKED register is the bitwise AND of the I2S_INTR_MASK and I2S_INTR registers. The final I²S interrupt signal is the logical OR of all the bits in the I2S_INTR_MASKED register. So only those events that are enabled in the I2S_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 I2S_INTR_SET register. **Figure 32-9** illustrates the interrupt signal generation.

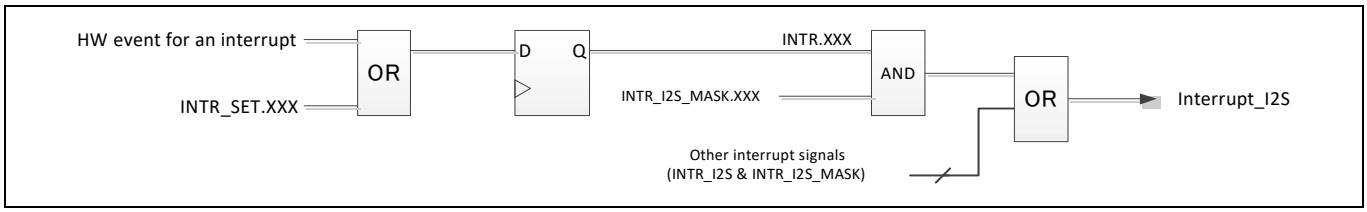


Figure 32-9. Interrupt Signal Generation

In the interrupt service routine (ISR), the I2S_INTR_MASKED register should be read to know the events that triggered the interrupt event. Multiple events can trigger the interrupt because the final interrupt signal is the logical OR output of the events. The ISR should do the tasks corresponding to each interrupt event that was triggered. At the end of the ISR, the value read in the I2S_INTR_MASKED register earlier should be written to the I2S_INTR register to clear the bits whose interrupt events were processed in the ISR. Unless the bits are not cleared by writing '1' to the I2S_INTR register, the interrupt signal will always be high. A dummy read of the I2S_INTR register should be done for the earlier register write to take effect.

32.9 Watchdog timer

The TX and RX blocks have independent watchdog timers, which can be used to generate an interrupt event if the Word Select (WS) input is idle for more than the configured time period. This feature is available only in the slave mode of operation where the external master drives the WS input lines (tx_ws, rx_ws). This feature can be used to detect any signal transmission issues, master device issues, 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, then only one of the watchdog timers can be enabled to cause the interrupt event. Although the following explanation covers Tx watchdog, the same explanation applies to Rx watchdog as well.

To enable the Tx watchdog timer feature, WD_EN bit in the I2S_TX_CTL register should be set. The watchdog timer reload value (32-bit timer) is configured by writing to the I2S_TX_WATCHDOG register. A value of zero written to the I2S_TX_WATCHDOG register will also disable the watchdog timer. **Figure 32-10** illustrates the watchdog behavior when the timer is enabled. The timer runs off the CLK_PERI system clock. Refer to the **"Clocking system"** on page 290 for details on generation of CLK_PERI. The timer starts running when WD_EN and TX_START bits are set. The timer reload happens either on a rising edge event on tx_ws input signal, or when the timer values reaches zero. When the timer value reaches zero, the TX_WD interrupt event is generated. The TX_WD bit in the I2S_INTR_MASK register should be set to enable interrupt generation by the watchdog timer interrupt event. The interrupt event can be cleared by writing '1' to the TX_WD bit in the I2S_INTR register.

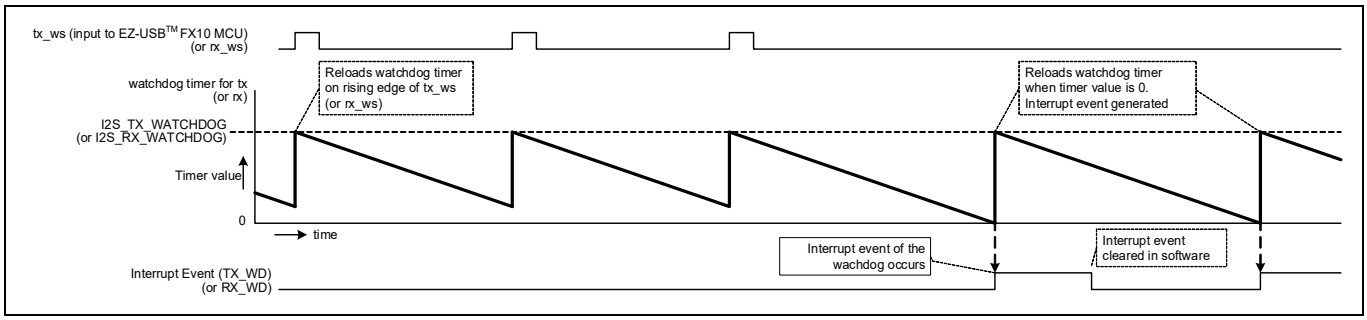


Figure 32-10. Watchdog timer working

33 PDM-PCM converter

The PDM-PCM unit accepts a stereo or mono serial data stream (pulse modulated 1-bit stream) coming from external digital PDM microphones. The PDM-PCM converter consists of a fifth order cascaded integrator comb (CIC) filter followed by a decimator, and a final stage high-pass filter. This block simplifies the conversion process by exposing the different configuration settings as registers, which you can program to meet the application needs. The entire PDM-PCM conversion process is handled in hardware; the PCM output data streaming can be done using the DMA controller thus freeing up the CPU bandwidth from performing periodic audio streaming activities.

33.1 Features

- Supports Mono/Stereo mode pulse density modulation (PDM) to pulse code modulation (PCM) conversion
- Accepts 1-bit PDM input and can generate 16-, 18-, 20-, or 24-bit PCM digital data output
- Configurable PDM microphone clock frequency
- Ability to generate standard audio sampling rates by adjusting the decimation rate and clock divider values
- Digital volume control: Programmable gain amplifier (PGA) control from –12 dB to +10.5 dB in 1.5-dB steps
- Smooth PGA and soft-mute control
- Hardware receive buffer: 24-bit wide, 255-element FIFO with support for DMA controller-based data transfer
- Optional high-pass filter to remove DC and low-frequency noise

33.2 Architecture

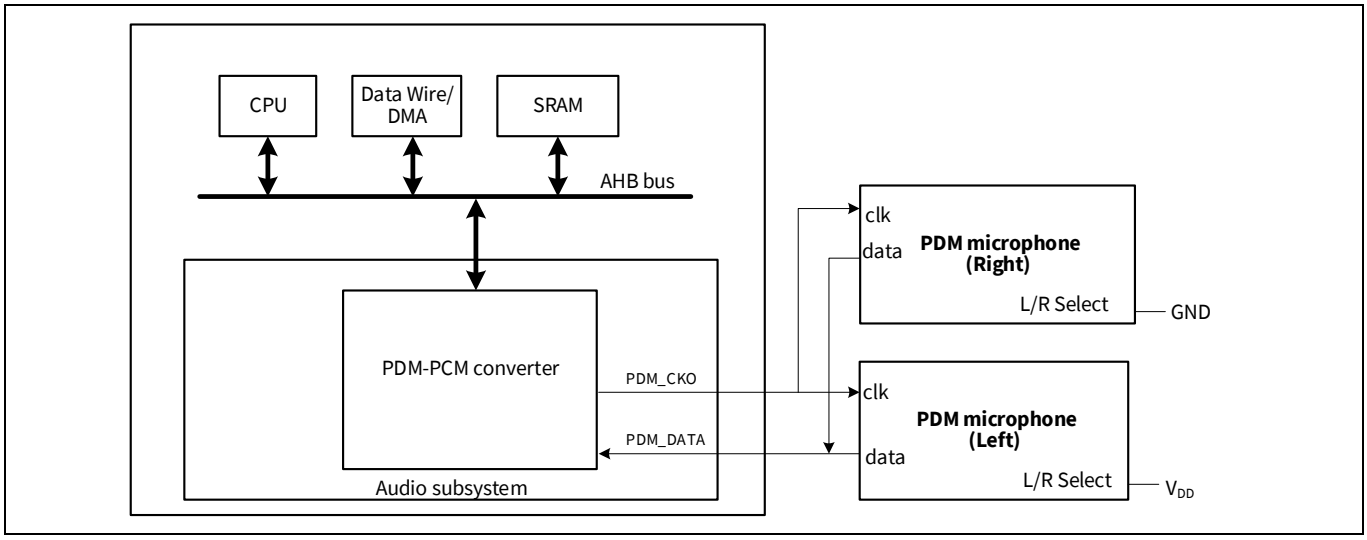


Figure 33-1. Block diagram

Figure 33-1 shows the block diagram of the PDM-PCM converter. See the device datasheet for information on port pin assignments of the PDM block signals, electrical specifications, and the interrupt vector number.

33.2.1 Enable/disable converter

The PDM-PCM converter can be powered ON or OFF by using the ENABLED bit in the PDM_CTL register. The block can be turned OFF when not used to save power. When the block is powered off by writing '0' to the ENABLED bit, the non-retention registers lose their current values. Refer to the registers reference manual to know which registers are retention and non-retention type. When the block is enabled again, the non-retention registers are reset to their default values. User firmware should ensure that all non-retention registers are configured as required after the ENABLED bit is set, and before the PDM-PCM conversion is started by setting the STREAM_EN bit in the PDM_CMD register. See **"Operating procedure"** on page 659 for the recommended procedure to configure this PDM-PCM converter before starting the PDM-PCM conversion.

33.2.2 Clocking features

The block uses the CLK_HF[1] root high-frequency clock for performing the PDM-PCM conversion process. Refer to the **"Clocking system"** on page 290 for details on selecting the clock source for CLK_HF[1], and configuring the divider registers to generate CLK_HF[1].

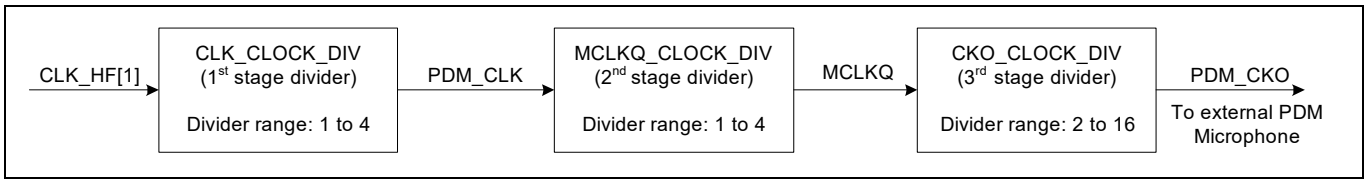


Figure 33-2. PDM-PCM clocking dividers

Figure 33-2 shows the clock divider structure in the block. The block has three stages of clock dividers to generate the clock (PDM_CKO), which goes to the external PDM microphone clock input.

1. The first stage clock divider (CLK_CLOCK_DIV field in the PDM_CLOCK_CTL register) is used to generate the actual clock signal (PDM_CLK) that goes to the PDM-PCM converter. The input is CLK_HF[1]; the CLK_CLOCK_DIV can be a value between 0 and 3 (divider value between 1 and 4).

   $f_{PDM\_CLK} = f_{CLK\text{-}HF[1]} / (CLK\_CLOCK\_DIV + 1)$
2. The second stage clock divider (MCLKQ_CLOCK_DIV field in the PDM_CLOCK_CTL register) is used to generate the internal master clock and can take a value between 0 and 3 (divider value between 1 and 4). The input clock is pdm_clk and the output of the divider is MCLKQ.

   $f_{MCLKQ} = f_{PDM\_CLK} / (MCLKQ\_CLOCK\_DIV + 1)$
3. The third stage clock divider (CKO_CLOCK_DIV field in the PDM_CLOCK_CTL register) is used to generate the clock that goes to the PDM microphone clock (PDM_CKO) through the output pin of the EZ-USB™ FX10 MCU. The input clock is MCLKQ. The divider register can be between 1 and 15 (divider value between 2 and 16).

   $f_{PDM\_CKO} = f_{MCLKQ} / (CKO\_CLOCK\_DIV + 1)$, *if* $CKO\_CLOCK\_DIV \geq 1$

   $f_{PDM\_CKO} = f_{MCLKQ} / 2$, *if* $CKO\_CLOCK\_DIV = 0$

33.2.3 Over-sampling ratio

Over-sampling ratio (OSR) is the ratio between the frequency of the PDM microphone clock (f_{PDM_CKO}) and the output PCM sampling rate frequency (f_S). The OSR is determined by the SINC_RATE bits in the PDM_CLOCK_CTL register. The relation is as follows.

$OSR = f_{PDM\_CKO} / f_S = 2 \times SINC\_RATE$

Table 33-1 gives an example of the PDM clock divider and SINC_RATE register configurations to generate the PCM output at standard sampling rates. Note that the PDM divider values in the table are 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 maximum values of PDM_CKO frequency, PDM_CLK frequency, and the output sampling rates.

Table 33-1. PDM Clock Divider Values for Standard Audio Sampling Rates

Sampling rate (SR) (kHz)	SINC_RATE (= OSR/2)	PDM_CKO (=2*SINC_RATE*SR) (MHz)	CLK_HF3 (MHz)	(CLK_HF[3])/(PDM_CKO) (total divider ratio)	CLK_CLOCK_DIV (first divider)	MCLKQ_CLOCK_ DIV (second divider)	CKO_CLOCK_ DIV (third divider)
8	32	0.512	49.152	96	1	3	11
16	32	1.024		48	1	3	5
32	32	2.048		24	0	3	5
48	32	3.072		16	0	1	7
44.1	32	2.8224	45.1584	32	0	1	7

There are various methods to generate the CLK_HF[3] frequencies listed in the table depending on the clocking options available on the device. Refer to the **"Clocking system"** on page 290 for details on the clocking options available in the device, including the clock sources and PLL/FLL circuitry. For example, an external crystal oscillator (ECO) can be used in conjunction with a phase-locked loop (PLL) to generate the CLK_HF[3] at the desired frequency of 49.152 MHz or 45.1584 MHz.

One possible combination of PLL divider values to generate the 49.152 MHz frequency from a 17.2032 MHz ECO are: REFERENCE_DIV = 7, FEEDBACK_DIV = 100, OUTPUT_DIV = 5. One possible combination of PLL divider values to generate the 45.1584 MHz frequency from a 17.2032 MHz ECO are: REFERENCE_DIV = 8, FEEDBACK_DIV = 105, OUTPUT_DIV = 5.

33.2.4 Mono/stereo microphone support

The PDM-PCM converter supports mono-left, mono-right, stereo, and swapped stereo modes of operation. The operation mode is controlled by the PDM_CH_SET and SWAP_LR bits in the PDM_MODE_CTL register. The register settings for the different operation modes are given in **Table 33-2**. The table also lists the invalid register settings, which you must not use in the firmware.

Table 33-2. Operation mode register settings

Register setting	Operation mode
PCM_CH_SET = 0, SWAP_LR = 0 or 1	Recording OFF
PCM_CH_SET = 1, SWAP_LR = 0	Mono-Left recording mode. Only the left microphone channel is sampled on the rising edge of PDM_CKO. FIFO buffer contains only left channel data.
PCM_CH_SET = 2, SWAP_LR = 1	Mono-Right recording mode. Only the right microphone channel is sampled on the falling edge of PDM_CKO. FIFO buffer contains only right channel data.
PCM_CH_SET = 3, SWAP_LR = 0	Stereo recording mode. The right microphone channel is sampled on the falling edge of PDM_CKO and left channel on rising edge. FIFO buffer contains data in L/R format (left channel followed by right channel)
PCM_CH_SET = 3, SWAP_LR = 1	Swapped Stereo recording mode. The right microphone channel is sampled on the rising edge of PDM_CKO and left channel on falling edge. FIFO buffer contains data in L/R format (left channel followed by right channel)
PCM_CH_SET = 1, SWAP_LR = 1 or PCM_CH_SET = 2, SWAP_LR = 0	Invalid setting (not supported). Do not operate the PDM-PCM converter in these configuration settings.

Figure 33-3 shows the timing diagrams for the different operating modes.

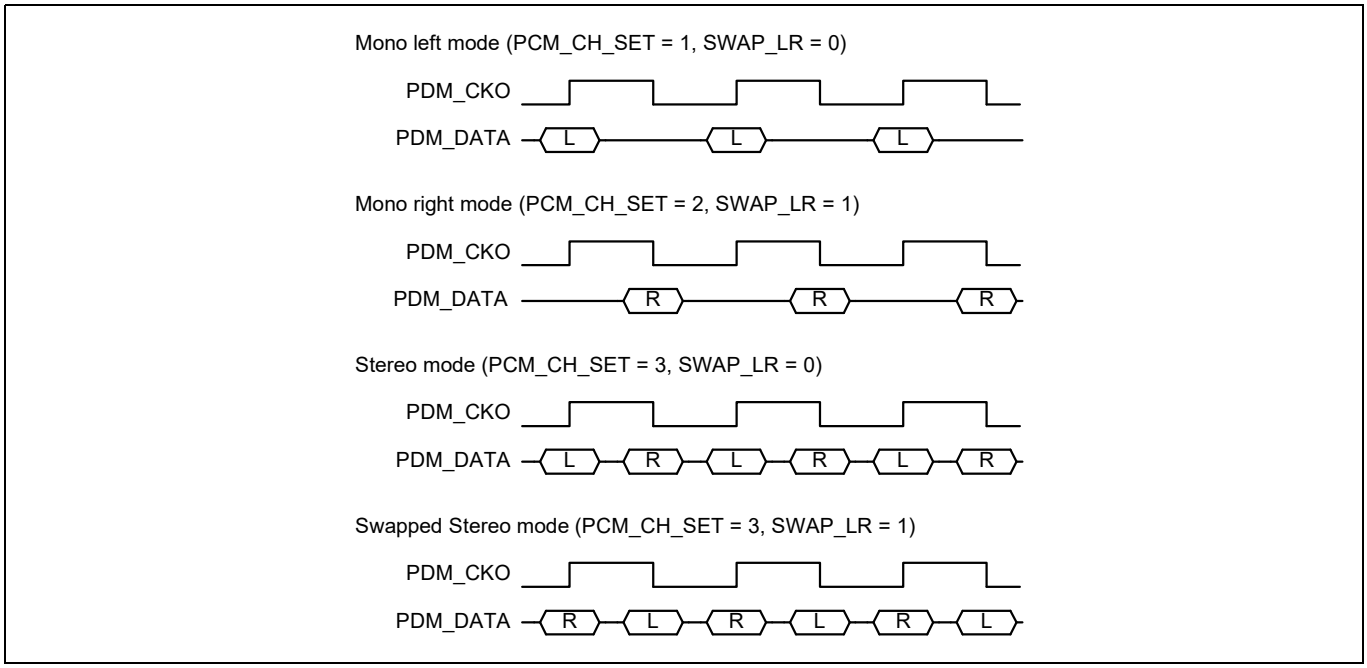


Figure 33-3. PDM mono/stereo timing

To alleviate uncertain board delay impact on PDM_IN setup and hold timing constraints, the PDM-PCM provides CKO_DELAY bits in the PDM_MODE_CTL register to add extra delay for the PDM_CKO path to internal sampler. CKO_DELAY can be a value between 0 and 7. A value of '0' implies that internal sampling of PDM data is advanced by three PDM_CLK clock cycles for the PDM_CKO transition. A value of '7' implies that internal sampling of PDM data is delayed by four PDM_CLK clock cycles for the PDM_CKO transition. Refer to the registers reference manual for details on the meaning of different CKO_DELAY values.

Note: Variations have been observed in the recommendation for left/right sampling logic among the different PDM microphone manufacturers. The SWAP_LR bit in the PDM-PCM converter ensures that you can adjust the sampling logic according to the microphone datasheet recommendations. Refer to the PDM microphone manufacturer datasheet for the exact timing details. Also, in stereo mode, use the same manufacturer for both the left/right PDM microphones to ensure the timing behavior is uniform for both channels.

33.2.5 Hardware FIFO buffers and DMA controller support

The PDM-PCM converter has a hardware FIFO depth of 255 elements where each element is 24-bit wide.

The PDM_RX_FIFO_CTL register is used for FIFO control operations. Refer to the register description in the registers reference manual for more details. The TRIGGER_LEVEL field in the PDM_RX_FIFO_CTL register is used to generate a receive trigger event (interrupt event, DMA trigger signal) when the Rx FIFO has more entries than the value configured in the TRIGGER_LEVEL field. The TRIGGER_LEVEL field can be configured up to 254 in the mono microphone recording mode and up to 253 in the stereo microphone recording mode.

The FIFO freeze operation can be enabled by setting the FREEZE bit in the PDM_RX_FIFO_CTL register. When the FREEZE bit is set and the Rx block is operational (STREAM_EN bit in the PDM_CMD register is set), hardware will not write to the Rx FIFO. Also, the Rx FIFO write pointer will not be advanced. Any reads from the PDM_RX_FIFO_RD register will increment the Rx FIFO read pointer; when the Rx FIFO becomes empty, the internal read pointer stops incrementing. The freeze operation may be used for firmware debug purposes. This operation is not intended for normal operation. To return to normal operation after using the freeze operation, the PDM-PCM must be reset by clearing the ENABLED bit in PDM_CTL register, and then setting the bit again.

The CLEAR bit in the PDM_RX_FIFO_CTL register is used to clear the Rx FIFO by resetting the read/write pointers associated with the FIFO. Read accesses from the Rx FIFO using PDM_RX_FIFO_RD or PDM_RX_FIFO_RD_SILENT registers are not allowed while the CLEAR bit is set.

The PDM_RX_FIFO_STATUS register provides FIFO status information. This includes number of used entries in the Rx FIFO and the current values of the Rx FIFO read/write pointers. This register can be used for debug purposes. The Rx FIFO write pointer is updated whenever the data is transferred to the Rx FIFO from the internal receive buffer. Rx FIFO read pointer is updated whenever the data is read from the PDM_RX_FIFO_RD register, either through the CPU or the DMA controller. For debug purposes, the PDM_RX_FIFO_RD_SILENT register is available, which always returns the top element of the Rx FIFO without updating the read pointer.

For Rx FIFO data reads using the CPU, the hardware can be used to trigger an interrupt event for any of the FIFO conditions such as RX_TRIGGER and RX_NOT_EMPTY. As part of the interrupt handler, the CPU can read from the PDM_RX_FIFO_RD register. The recommended method is to read (TRIGGER_LEVEL + 1) words from the PDM_RX_FIFO_RD register every time the RX_TRIGGER interrupt event is triggered. In addition, interrupt events can be generated for FIFO overflow and underflow conditions.

For DMA-based data transfers, the DMA trigger signal (tr_pdm_rx_req) can be enabled by writing '1' to the RX_REQ_EN bit in the PDM_TR_CTL register. The trigger signal output will become high whenever the Rx FIFO has more entries than that configured in the TRIGGER_LEVEL field. Refer to the **"Trigger multiplexer block"** on page 344 for details on how to connect the DMA trigger signal to a particular DMA channel. The DMA channel can be configured to transfer up to (TRIGGER_LEVEL + 1) words to the applicable destination address (such as SRAM regions). The source address of the DMA should always be the PDM_RX_FIFO_RD register address, with the source address increment feature disabled in the DMA channel configuration. This FIFO address increment logic is handled internally to adjust the read pointer, and the DMA should not increment the source address. For more details on DMA channel configuration, refer to the **"Data wire controller (DW)"** on page 141.

The successive data read from the PDM_RX_FIFO_RD follows the Left 1/Right 1/Left 2/Right 2/… format in stereo and swapped stereo modes of operation. For mono left and mono right recording modes, the data read from FIFO contains either the left channel data (mono left mode) or the right channel data (mono right mode).

The data in the PDM_RX_FIFO_RD is always right-aligned. The PDM_TX_FIFO_RD format for different word length configurations is provided in **Figure 33-4**. Note that the unused most significant bits are either set as '0' or sign-bit extended depending on the BIT_EXTENSION field setting in the PDM_DATA_CTL register.

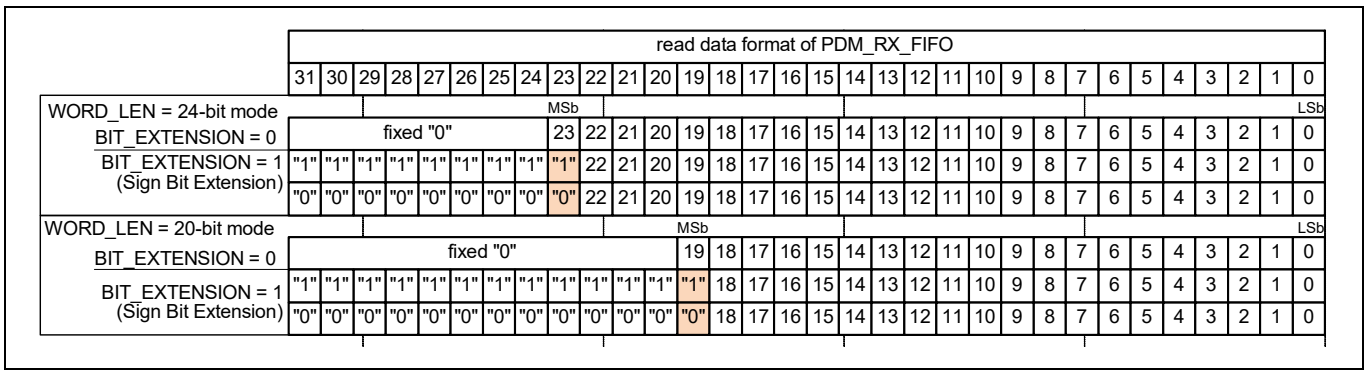


Figure 33-4. FIFO register structure

33.2.6 Interrupt support

The block has one output signal (interrupt_pdm) that goes to the interrupt controller in the CPU. See the device datasheet for details on the vector number of the PDM-PCM interrupt. Refer to the **"Interrupts"** on page 89 for the procedure to configure the interrupt priority, vector address, and enabling/disabling.

The PDM interrupt can be triggered under any of the following events – RX_TRIGGER, RX_NOT_EMPTY, RX_OVERFLOW, or RX_UNDERFLOW. **Table 33-3** lists the trigger conditions and details of these events.

Table 33-3. Interrupt event trigger condition

Interrupt event	Trigger conditions
RX_TRIGGER	The PDM Rx FIFO has more entries than the value specified in the TRIGGER_LEVEL field in the PDM_RX_FIFO_CTL register. At least (TRIGGER_LEVEL + 1) words can be read from the PDM_RX_FIFO_RD register in the interrupt service routine.
RX_NOT_EMPTY	The PDM Rx FIFO has at least one word that can be read from the PDM_RX_FIFO_RD register.
RX_OVERFLOW	The PDM Rx FIFO content is overwritten by the PDM-PCM converter due to the number of unread words exceeding the 256 word buffer capacity. The output PCM data after the overflow condition is discarded and not transferred to the FIFO. This event can be used to detect bandwidth constraints in the end application due to DMA or the interrupt used for data transfer not getting the required priority for executing the data transfers.
RX_UNDERFLOW	Attempt to read from an empty PDM Rx FIFO. This can happen due to incorrect configuration of the DMA or the interrupt service routine code used to do the data transfer.

Each of the interrupt events can be individually enabled or disabled to generate the interrupt condition. The PDM_INTR_MASK register is used to enable the required events by writing '1' to the corresponding bit.

Irrespective of the INTR_MASK settings, if any event occurs, the corresponding event status bit will be set by the hardware in the PDM_INTR register. The PDM_INTR_MASKED register is the bitwise AND of the PDM_INTR_MASK and PDM_INTR registers. The final PDM interrupt signal is the logical OR of all the bits in the PDM_INTR_MASKED register. So only those events that are enabled in the PDM_INTR_MASK register are propagated as interrupt events to the interrupt controller.

Interrupt events can also be triggered in software by writing to the corresponding bits in PDM_INTR_SET register.

PDM-PCM converter

Figure 33-5 illustrates the interrupt signal generation from the PDM-PCM block as explained above. Only the RX_TRIGGER interrupt generation is highlighted in the figure; the remaining interrupt events also follow the same generation logic.

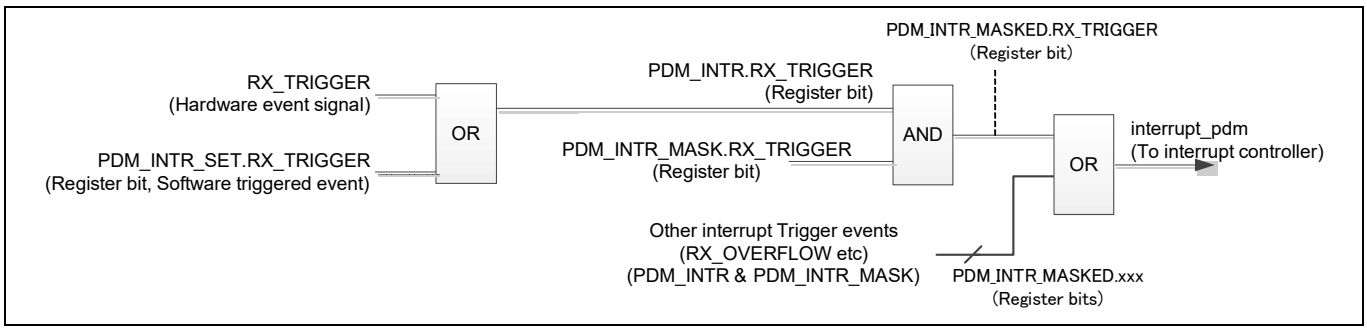


Figure 33-5. PDM interrupt signal generation

In the interrupt service routine (ISR) corresponding to the interrupt vector number of interrupt_pdm, the PDM_INTR_MASKED register should be read to know the events that triggered the interrupt event. Multiple events can trigger the interrupt because the final interrupt signal is the logical OR output of the events. The ISR should do the tasks corresponding to each interrupt event that was triggered. At the end of the ISR, the value read in the PDM_INTR_MASKED register earlier should be written to the PDM_INTR register to clear the bits whose interrupt events were processed in the ISR. A dummy read of the PDM_INTR register should be done for the earlier register write to PDM_INTR to take effect.

All of the interrupt event bits in PDM_INTR register will continue to indicate the event condition regardless of the true state until that bit is cleared (for example, when set, the RX_OVERFLOW bit will continue to indicate an OVERFLOW state until the RX_OVERFLOW bit is cleared regardless of the true state of the FIFO. A mere FIFO read of the FIFO will not clear the RX_OVERFLOW bit.). Unless the PDM_INTR bits that are used to generate the interrupt are not cleared by writing '1' to the PDM_INTR bits, the interrupt signal will always be high.

33.2.7 Digital volume gain

The PDM-PCM converter supports independent digital volume control on the left/right channels with a range from –12.5 dB to +10.5 dB in steps of 1.5 dB. It is programmed by configuring the PGA_R and PGA_L bits in the PDM_CTL register. PGA gain may be changed on the fly during normal operation, or as a one-time setting before starting the PDM-PCM conversion process.

33.2.8 Smooth gain transition

To reduce zipper or clip noise during on-the-fly gain transition or during soft mute operation, a built-in volume smoother is implemented with fine gain steps and fine time steps that enable soft ramp up or ramp down of the volume levels. Two fine gain options of 0.13 dB and 0.26 dB step sizes are available. The fine gain is set by the STEP_SEL bit in the PDM_CTL register. In addition to the fine gain steps, a time step is available for the fine gain change in terms of the number of sample cycles. The time step can be configured to a value between 64 sample periods and 512 sample periods using the S_CYCLES bits in the PDM_MODE_CTL register. So the STEP_SEL and S_CYCLES bit settings together determine the rate at which PGA gain or the soft mute transitions take effect.

33.2.9 Soft mute

The PDM-PCM contains a built-in software-controlled mute function that digitally attenuates signals to imperceptible levels or zero. When mute function is enabled by setting the SOFT_MUTE bit in the PDM_CTL register, the corresponding PCM output is decreased from current level to mute state through predefined granular gain step per time constant transition. The STEP_SEL bit setting determines the gain step and the S_CYCLES bits determine the time constant. During soft-mute, the block is still ON and the PCM data streaming is operational; the DMA or CPU-based data transfer also happens as usual. Only the PCM output level is muted. When mute function is disabled by setting SOFT_MUTE = 0, the mute function is OFF and the PDM-PCM returns to normal operation where output signal level goes up to normal with current PGA gain.

33.2.10 Word length and sign bit extension

The PCM output word length can be configured for either 16-bits, 18-bits, 20-bits, or 24-bits using WORD_LEN bits in the PDM_DATA_CTL register. Irrespective of the word length setting, the PCM output is always read from the FIFO data buffer register (PDM_RX_FIFO_RD) as a 32-bit value. The unused most significant bits in the 32-bit value can either be sign extended or extended by '0' by using the BIT_EXTENSION bit in the PDM_DATA_CTL register.

33.2.11 High-pass filter

The PDM-PCM converter has a final stage high-pass filter (HPF) that blocks DC offset and low-frequency noise in signal band. The HPF is enabled when the HPF_EN_N bit in the PDM_MODE_CTL register is zero, and disabled when the HPF_EN_N bit is 1.

The filter response for HPF is characterized as:

$$H_{(z)} = \frac{1 - z^{-1}}{1 - (1 - 2^{-HPF\_GAIN})z^{-1}}$$

The HPF operates at the final PCM output sampling rate. HPF_GAIN is a 4-bit gain configuration setting in the PDM_MODE_CTL register. In default mode, HPF_GAIN = 0xB, so the HPF can be formulated by polynomial:

$$H_{(z)} = \frac{1 - z^{-1}}{1 - 0.99951z^{-1}}$$

The HPF_GAIN setting can be tuned to adjust HPF cutoff corner frequency for better system configuration.

33.2.12 Enable/disable streaming

The PDM-PCM conversion process can be dynamically enabled/disabled by using the STREAM_EN bit in the PDM_CMD register.

33.2.13 Power modes

The PDM-PCM can operate in Active and Sleep CPU modes while in LP system power modes. It is not operational in system or Deep Sleep power modes. When the device transitions from Deep Sleep power mode to the LP power mode, the non-retention registers lose their previous configuration values. So the non-retention registers must be appropriately configured before enabling the PDM-PCM again for LP mode operation. One option is to store the non-retention register values in SRAM before entering Deep Sleep modes. When returning to the LP mode, the SRAM values can be copied to the registers after enabling the PDM-PCM by setting the ENABLED bit in the PDM_CTL register. Refer to the registers reference manual to identify the non-retention registers for the PDM.

33.3 Operating procedure

33.3.1 Initial configuration

The sequence of steps for initial configuration of the PDM-PCM converter before starting the conversion process is as follows:

1. Configure the clock dividers and decimation rate in the PDM_CLOCK_CTL register. This register configuration should be done before enabling the PDM-PCM converter. If the ENABLED bit in the PDM_CTL register is set, it should be cleared before changing the clock configuration.
2. Enable the block; set the PGA gain and fine gain step setting as required by writing to the PDM_CTL register.
3. Configure the PDM_MODE_CTL and PDM_DATA_CTL registers as required.
4. Configure the Rx FIFO trigger level setting by writing to the TRIGGER_LEVEL bits in the PDM_RX_FIFO_CTL register. The CLEAR and FREEZE bits in PDM_RX_FIFO_CTL are not set for normal operation.
5. Configure the events that must generate the interrupt by setting the corresponding bits in the PDM_INTR_MASK register, and clearing the remaining bits.
6. Configure the interrupt PDM interrupt vector and enable the interrupt vector. See the **“Interrupts”** on page 89 for details.
7. If a DMA-based data transfer is required, connect the PDM DMA trigger signal (tr_pdm_rx_req) to the trigger input of the required DMA channel. See the **“Trigger multiplexer block”** on page 344 for details on how to connect to the DMA channel trigger input. Configure the DMA channel as required - the source address of the DMA descriptor is PDM_RX_FIFO_RD register with the source address increment feature disabled and the source data length is word type (32-bits). The DMA channel can be used to transfer (TRIGGER_LEVEL + 1) words from the PDM_RX_FIFO_RD register whenever the trigger signal becomes high. The destination address configuration depends on the application requirements. See the **“Data wire controller (DW)”** on page 141 for details on DMA channel configuration.
8. Enable the DMA trigger signal generation by setting the RX_REQ_EN bit in the PDM_TR_CTL register.

33.3.2 Interrupt service routine (ISR) configuration

The code for the PDM interrupt service routine should have the following flow:

1. The events that triggered the interrupt can be found by reading the PDM_INTR_MASKED register in the ISR. All the bits that are set causes the interrupt event. The register value should also be in a variable “var”.
2. For each of the event bits that are set in PDM_INTR_MASKED, appropriate application level tasks can be executed. For example, the RX_TRIGGER event can be used for CPU-based data transfers if a DMA-based data transfer is not used. DMA transfers should use the tr_pdm_rx_req trigger signal (by setting the RX_REQ_EN bit in the PDM_TR_CTL register). The DMA trigger should not use the RX_TRIGGER interrupt event to reduce CPU usage for data transfer. The RX_OVERFLOW event can be used to take appropriate counter measures such as giving higher priority to PDM-PCM DMA channel. The RX_UNDERFLOW event typically indicates wrong data transfer logic in the application – either in the CPU-based data transfer code or in the DMA channel configuration used to transfer data.
3. After the event conditions have been processed, the “var” value read from PDM_INTR_MASKED should be written to the PDM_INTR register to clear the events that are set in the register. Due to the buffered write logic, the PDM_INTR register should also be read after the write process to ensure the write process is completed in the slower peripheral clock domain.

33.3.3 Enabling / disabling streaming

The PDM-PCM conversion process starts after the STREAM_EN bit is set in the PDM_CMD register. Depending on the application needs, the streaming can be dynamically started and stopped using the STREAM_EN bit. Clear the Rx FIFO before starting the streaming process to reset the read/write pointers and FIFO state. The procedure to clear the FIFO is to write a ‘1’ to the CLEAR bit in PDM_RX_FIFO_CTL followed by writing a ‘0’ to the CLEAR bit. When the CLEAR bit is set, all the data entries in the Rx FIFO are cleared by resetting the internal read/write pointers. Read accesses to the PDM_RX_FIFO_RD and PDM_RX_FIFO_RD_SILENT registers are prohibited when the CLEAR bit is 1. Therefore, the CLEAR bit should be cleared before starting the streaming operation.

Revision history

Document revision	Date	Description of changes
**	2025-03-17	Initial release

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2025-03-17
Published by

Infineon Technologies AG
81726 Munich, Germany

© 2025 Infineon Technologies AG. All Rights Reserved.

Do you have a question about this document?
Email:
erratum@infineon.com

Document reference
002-41180 Rev. **

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.