

TECHNICAL REFERENCE MANUAL

**Applicable to Apollo510 SoC, Apollo510B SoC,
Apollo510 Lite Series SoCs, Apollo330 Plus Series SoCs**

Ultra-low Power Apollo SoCs

Doc. ID: RM-A5-1p1

Doc. Revision: 1.1, July 2026



IMPORTANT NOTICE:

**This Technical Reference Manual includes content which is accurate
to the extent possible but is subject to change.**

Legal Information and Disclaimers

AMBIQ MICRO BELIEVES THE INFORMATION IN THIS DOCUMENT IS ACCURATE AT THE TIME OF PUBLICATION. HOWEVER, THE INFORMATION MAY CONTAIN TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. AMBIQ MICRO RESERVES THE RIGHT TO MAKE CORRECTIONS, MODIFICATIONS, ENHANCEMENTS, IMPROVEMENTS, OR OTHER CHANGES TO ITS PRODUCTS (AND DOCUMENTATION) AT ANY TIME WITHOUT NOTICE.

INFORMATION IN THIS DOCUMENT IS PROVIDED SOLELY TO ENABLE THE USE OF AMBIQ MICRO PRODUCTS AND IS PROVIDED “AS IS.” NO LICENSE IS GRANTED TO DESIGN OR FABRICATE ANY INTEGRATED CIRCUITS BASED ON THE INFORMATION IN THIS DOCUMENT. THIS DOCUMENT MAY NOT BE USED IN CONNECTION WITH ANY LEGAL ANALYSIS CONCERNING THE AMBIQ MICRO PRODUCTS DESCRIBED HEREIN. AMBIQ MICRO DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NON-INFRINGEMENT.

AMBIQ MICRO PRODUCTS ARE SOLD SUBJECT TO AMBIQ MICRO’S TERMS AND CONDITIONS OF SALE; SOFTWARE IS PROVIDED PURSUANT TO APPLICABLE LICENSE TERMS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, IS GRANTED UNDER ANY INTELLECTUAL PROPERTY RIGHT OF AMBIQ MICRO OR ANY THIRD PARTY BY THIS DOCUMENT.

CUSTOMERS ARE SOLELY RESPONSIBLE FOR THE DESIGN, VALIDATION, TESTING, AND SECURITY OF THEIR PRODUCTS AND APPLICATIONS, INCLUDING THE SPECIFIC MANNER IN WHICH AMBIQ MICRO’S PRODUCTS ARE INCORPORATED. “TYPICAL” PARAMETERS ARE FOR REFERENCE ONLY AND IT IS THE CUSTOMER’S RESPONSIBILITY TO VALIDATE “TYPICAL” SPECIFICATIONS FOR THEIR APPLICATION.

AMBIQ MICRO PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN APPLICATIONS WHERE PRODUCT FAILURE COULD RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE (“HIGH-RISK APPLICATIONS”) WITHOUT EXPRESS WRITTEN APPROVAL. IF PRODUCTS ARE USED IN HIGH-RISK APPLICATIONS WITHOUT SUCH APPROVAL, THE CUSTOMER SHALL INDEMNIFY AND HOLD HARMLESS AMBIQ MICRO AND ITS AFFILIATES FROM ALL RESULTING CLAIMS, DAMAGES, COSTS, AND EXPENSES, INCLUDING REASONABLE ATTORNEYS’ FEES. IN NO EVENT SHALL AMBIQ MICRO BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, OR PUNITIVE DAMAGES, INCLUDING LOST PROFITS OR DATA, ARISING FROM THE USE OF THIS DOCUMENT OR ANY PRODUCT.

“AMBIQ”, “APOLLO”, “SPOT”, AND COMBINATIONS THEREOF ARE TRADEMARKS OF AMBIQ MICRO, INC. OTHER PRODUCT NAMES USED IN THIS PUBLICATION ARE FOR IDENTIFICATION PURPOSES ONLY AND MAY BE TRADEMARKS OF THEIR RESPECTIVE COMPANIES.

Table of Contents

1. Document Revision History	21
2. Introduction	22
3. SoC Architecture	23
3.1 DMA Overview	23
3.1.1 Two-Stage DMA Request Pipeline Feature	24
3.2 Memory Map	25
3.3 Instrumentation Trace Macrocell (ITM)	25
3.4 SoC Control	25
3.5 Memory Controller Programming	25
3.6 Memory/Cache Coherency Management	26
3.6.1 Background	26
3.6.2 Cache Management & Barrier Operations	28
3.6.3 System Operations	30
3.6.4 Cache Management in the AmbiqSuite SDK	31
4. Power Management and Optimization	41
4.1 Power Management	41
4.1.1 Power Control Block	41
4.1.2 SPOT Manager	41
4.2 Power Optimization	43
4.2.1 Low Power Initialization Sequence Requirements	43
4.2.2 Multi-Image Transition Requirements	44
4.3 Temperature Compensation (TempCo)	45
5. Cryptographic Accelerator	46
5.1 Introduction	46
5.2 Functional Description	47
5.2.1 SECURITY Register Interface	47
5.2.2 CryptoCell-312 Hardware Accelerator	47
5.2.3 PUF Hardware Accelerator	48
5.2.4 ERRATUM NOTICE - ERR014: DMA to TCM Restrictions	48
5.2.5 APDMA Issues	49
5.3 Design Considerations	50
5.3.1 SECURITY Usage Considerations	50
5.3.2 PUF Usage Considerations	50
5.3.3 CryptoCell-312 Usage Considerations	51
5.4 Programming Guidelines	52
5.4.1 SECURITY Register Programming	52
5.4.2 4.2 PUF Programming Guidelines	52
5.4.3 CryptoCell-312 Programming Guidelines	53
5.5 Register Summary	54
5.5.1 SECURITY Register Interface	54
5.5.2 5.2 PUF Register Interface	54
5.5.3 CryptoCell-312 Register Interface	54
6. Storage	56
6.1 MRAM	56
6.1.1 MRAM Recovery	56
6.2 One-Time Programmable Memory	56
6.2.1 General Flow	56
6.2.2 Power-On Flow	57
6.2.3 Power-Off Flow	58

7. Clock Generation and Control	59
7.1 HFRC Auto-Adjustment	59
7.2 XTALHS Crystal Calibration on the Apollo510 SoC	60
7.2.1 Determining whether the XTALHS trimming is needed	60
7.2.2 Setting XTALHSCAPTRIM and XTALHSCAP2TRIM register values	61
7.2.3 Trimming the External XTALHS frequency	62
7.3 BLE Crystal Calibration on the Apollo510B SoC	64
7.4 RF XTAL Frequency Trimming on the Apollo510 Lite and Apollo330 Plus SoCs ...	65
7.5 Generating Required RTC Clock	67
7.6 External 32.768 kHz Clock Input on XO	68
7.7 Clock Switching	69
7.8 System PLL Control	70
7.8.1 PLL Control	70
7.8.2 PLL Mode Control	70
7.8.3 Considerations When Programming the PLL Feedback Divider	71
7.9 Clock/Reset Module in Apollo510 Lite and Apollo330 Plus SoCs	72
7.10 HP2 Mode and PLL Handling in Apollo510 Lite and Apollo330 Plus SoCs	73
7.11 Clock Manager	74
7.11.1 Introduction to Clock Manager	74
7.11.2 Clocks Managed by Clock Manager	74
7.11.3 Porting Guide for the BSP	74
7.11.4 Guidelines for XTAL_HS	77
7.11.5 Guidelines for SYSPLL on Apollo510 and Apollo510B	77
7.11.6 Guidelines for PLLVCO and PLLPOSTDIV on Apollo510 Lite and Apollo330 Plus ..	78
7.11.7 Guidelines for EXTREFCLK on Apollo510 and Apollo510B	78
7.11.8 Peripheral Clock Configuration Guidelines	79
8. Real Time Clock (RTC)	83
8.1 Calendar Counters	83
8.2 Calendar Counter Reads	83
8.3 Alarms	83
8.4 Century Control and Leap Year Management	84
8.5 Weekday Function	84
9. Counter/Timer Module (TIMER)	85
9.1 Functional Description	85
9.1.1 Counter Operation	85
9.1.2 Clock Sources and Prescalers	85
9.1.3 Compare and Interrupt Operation	85
9.1.4 Output Control	85
9.1.5 Timer Modes	86
9.1.6 Triggering Functions	95
9.1.7 Clocking Timer/Counters with Other Counter/Timer Outputs	95
9.1.8 Timer Cascade and Synchronization	95
9.1.9 Generating the Sample Rate for the ADC	96
9.1.10 Generating the Sample Rate for the Audio ADC	96
9.1.11 CLR and EN Details	96
9.1.12 Counter Functions	97
9.1.13 Interconnecting Timers	97
9.2 Design Considerations	98
9.2.1 Clock Source Selection	98
9.2.2 PWM Resolution and Frequency	98
9.2.3 Interrupt Latency Impact	98
9.3 Programming Guidelines	99

9.3.1 Configuring Timer Operation	99
9.3.2 PWM Generation Setup	99
9.3.3 Synchronized Multi-Timer Operation	99
9.3.4 Constraints	99
9.4 Register Summary	101
10. System Timer (STIMER)	103
10.1 Design Considerations	103
10.1.1 Power Consumption	103
10.1.2 Timing Resolution vs. Range	103
10.1.3 Real-Time Clock Implementation	103
10.1.4 RTOS Integration	103
10.2 Programming Guidelines	103
10.2.1 Writing to the STIMER Compare Register	103
10.2.2 Initializing the STIMER	104
10.2.3 Scheduling Compare Interrupts	104
10.2.4 Handling Overflow	104
10.2.5 Time Stamping	105
10.3 Register Summary	105
11. Watch Dog Timer (WDT)	106
11.1 Functional Description	106
11.1.1 Watchdog Counter Operation	106
11.1.2 Clock Source Selection	106
11.1.3 Two-Stage Timeout Mechanism	106
11.1.4 Configuration Locking	106
11.1.5 Interrupt Generation and Servicing	107
11.1.6 Reset Generation	107
11.2 Design Considerations	108
11.2.1 Timeout Period Selection	108
11.2.2 Clock Source Stability	108
11.2.3 Configuration Lock Considerations	108
11.2.4 Interrupt vs. Reset Strategy	108
11.2.5 Power Mode Watchdog Behavior	108
11.3 Programming Guidelines	110
11.3.1 Configuring Watchdog Operation	110
11.3.2 Calculating Timeout Values	110
11.3.3 Servicing the Watchdog	110
11.3.4 Interrupt Handling	111
11.3.5 Operating Constraints	111
11.4 Register Summary	112
12. General Purpose Input/Output (GPIO)	113
12.1 General Purpose I/O (GPIO) Functions	113
12.1.1 Configuring the GPIO Functions	113
12.1.2 Reading from a GPIO Pad	113
12.1.3 Writing to a GPIO Pad	113
12.1.4 GPIO Interrupts	114
12.2 Pad Connection Details	115
12.2.1 Output Selection	115
12.2.2 Output Control	115
12.2.3 Input Control	115
12.2.4 Pull-up/Pull-down Control	115
12.2.5 Analog Pad Configuration	115
12.3 Module-specific Pad Configuration	116

12.3.1 Implementing IO Manager Connections	116
12.3.2 Implementing IO Subordinate Connections	117
12.3.3 Implementing Full-Duplex IO Subordinate Connections	118
12.3.4 Implementing Display Controller Connections	119
12.3.5 Implementing Counter/Timer Connections	123
12.3.6 Implementing UART Connections	123
12.3.7 Implementing Audio Connections	130
12.3.8 Implementing Secure Digital IO Connections	133
12.3.9 Implementing GPIO Connections	134
12.3.10 Implementing CLKOUT Connections	134
12.3.11 Implementing 32 kHz CLKOUT Connections	135
12.3.12 Implementing 32 MHz CLKOUT Connections	136
12.3.13 Implementing ADC Connections	136
12.3.14 Implementing Voltage Comparator Connections	137
12.3.15 Implementing the Software Debug Port Connections	139
13. General Purpose ADC and Temperature Sensor Module (GPADC)	141
13.1 Clock Source and Divider	141
13.2 Eleven Selectable Analog Input Channels	141
13.3 Triggering and Trigger Sources	142
13.4 Voltage Reference Source	142
13.5 Eight Automatically Managed Conversion Slots	142
13.6 Automatic Sample Accumulation and Scaling	143
13.7 Sixteen Entry Result FIFO	144
13.8 Window Comparator	147
13.9 Operating Modes and the Mode Controller	148
13.9.1 Single Mode	149
13.9.2 Repeat Mode	150
13.9.3 Low Power Modes	150
13.10 Interrupts	151
13.11 Generating the Sample Rate for the GPADC	152
13.12 Temperature Sensor Operation	152
14. Multi-bit Serial Peripheral Interface Manager Module (MSPI)	153
14.1 Configuration	153
14.2 PIO Operation	154
14.3 DMA Operations	155
14.4 Execute in Place (XIP) Operations	156
14.4.1 XIP Address Mapping and Accesses	156
14.4.2 Optimized XIP Addressing	157
14.4.3 Micron XIP Support	157
14.5 Command Queueing (CQ)	158
14.5.1 Command Queue Data Format	158
14.5.2 CQ Interrupts	159
14.5.3 Pausing CQ Operations	159
14.5.4 Using the CQ Index Registers	160
14.5.5 MSPI and IOM Intercommunication	161
14.6 Data Scrambling	162
14.7 Auto Power Down	162
14.8 Board/Package Considerations for MSPI Pin Timing	163
14.9 MSPI Timing Diagrams with Delay Taps	166
14.9.1 SDR with Non-DQS Mode	166
14.9.2 DDR with DQS mode	166
14.9.3 OCTAL/HEX DDR with DQS Mode: Read Delay Lines with Lower Byte DQS	168

14.9.4 HEX DDR with DQS Mode: Read Delay Lines for Upper Data Byte	168
15. I2C/SPI Manager (IOM)	169
15.1 Programmer's Reference	169
15.2 Interface Clock Generation	170
15.3 Command Operation	171
15.4 FIFO	171
15.5 I2C Interface	172
15.5.1 Bus Not Busy	172
15.5.2 Start Data Transfer	172
15.5.3 Stop Data Transfer	172
15.5.4 Data Valid	172
15.5.5 Acknowledge	173
15.5.6 I2C Subordinate Addressing	173
15.5.7 I2C Offset Address Transmission	174
15.5.8 I2C Write Operation with Address Offset	174
15.5.9 I2C Read Operation with Address Offset	175
15.5.10 I2C Write Operation with No Address Offset	175
15.5.11 I2C Read Operation with No Address Offset	175
15.5.12 Holding the Interface with CONT	176
15.5.13 I2C Multi-Manager Arbitration	176
15.6 SPI Operations	177
15.6.1 SPI Configuration	177
15.6.2 SPI Subordinate Addressing	177
15.6.3 SPI Write with Address Offset	177
15.6.4 SPI Read with Address Offset	178
15.6.5 SPI Write with No Address Offset	178
15.6.6 SPI Read with No Address Offset	178
15.6.7 SPI 3-wire Mode	179
15.6.8 Complex SPI Operations	179
15.6.9 SPI Polarity and Phase	180
15.7 Bit Orientation	180
15.8 SPI Flow Control	181
15.9 Minimizing Power	183
16. I2C/SPI Subordinate (IOS)	184
16.1 Local RAM Allocation	184
16.2 Direct Area Functions	185
16.2.1 Host Burst Write Support	185
16.2.2 Mapping of Direct Area Access Interrupts	186
16.3 FIFO Area Functions	188
16.4 Rearranging the FIFO	190
16.5 Interface Interrupts	191
16.6 Command Completion Interrupts	192
16.7 Host Address Space and Registers	192
16.8 I2C Interface	193
16.8.1 Bus Not Busy	193
16.8.2 Start Data Transfer	193
16.8.3 Stop Data Transfer	193
16.8.4 Data Valid	193
16.8.5 Acknowledge	194
16.8.6 Address Operation	194
16.8.7 Offset Address Transmission	195
16.8.8 Write Operation	195

16.8.9 Read Operation	195
16.8.10 General Address Detection	196
16.9 SPI Interface	197
16.9.1 Write Operation	197
16.9.2 Read Operation	197
16.9.3 Configuring 3-wire vs. 4-wire SPI Mode	198
16.9.4 SPI Polarity and Phase	198
16.10 Bit Orientation	200
16.11 Wakeup Using the I2C/SPI Subordinate	200
16.12 Host Side Address Space and Register	201
16.12.1 Host Address Space and Registers	201
17. Universal Asynchronous Receiver/Transmitter (UART)	206
17.1 Functional Description	206
17.1.1 Baud Rate Generator	206
17.1.2 Transmitter	206
17.1.3 Receiver	206
17.1.4 FIFO Operation	206
17.1.5 DMA Support	207
17.1.6 Flow Control	207
17.2 Design Considerations	208
17.2.1 Enabling and Selecting the UART Clock	208
17.2.2 Requirements when Operating at or above 3 MBaud in DMA Mode	208
17.2.3 Transmit FIFO and Receive FIFO	208
17.2.4 Baud Rate Accuracy	209
17.2.5 Power Consumption	209
17.2.6 Reliable Communication	209
17.2.7 Mode Selection	209
17.3 Programming Guidelines	210
17.3.1 Initializing a UART Module	210
17.3.2 Transmitting Data	210
17.3.3 Receiving Data	210
17.3.4 Error Handling	210
17.4 UART Operating Modes	211
17.4.1 Regular UART Mode	211
17.4.2 Streaming UART Mode	211
17.5 Register Summary	212
18. Universal Serial Bus (USB)	214
18.1 Functional Overview	214
18.2 USB Reset	214
18.3 Soft Connect/Disconnect	215
18.4 Employing DMA	216
18.4.1 DMA Mode 0 - Sharing the DMA Engine	216
18.4.2 DMA Mode 1 - Handling Unknown-Length OUT Transactions	217
18.4.3 Using DMA with IN Endpoints	217
18.4.4 Using DMA with OUT Endpoints	218
18.5 High-speed Mode	219
18.6 USB Interrupt Handling	220
18.7 Index 0 Register Fields	221
18.8 Response to USB Conditions or Host Actions	223
18.8.1 Stall Issued to Control Transfer	223
18.8.2 Zero Length OUT Data Packets in Control Transfers	223
18.9 Suspend/Resume	224

18.9.1 USB Controller Active During Suspend	224
18.9.2 USB Controller Inactive During Suspend	224
18.9.3 Remote Wakeup	224
18.10 Start of Frame Packets	225
18.11 Dynamic FIFO Sizing	226
18.12 Endpoint 0 Handling	227
18.12.1 Zero Data Requests	227
18.12.2 Write Requests	227
18.12.3 Read Requests	228
18.12.4 Endpoint 0 States	229
18.12.5 Endpoint 0 Service Routine	230
18.12.6 Error Handling	236
18.13 IN Endpoint Packet Handling	237
18.13.1 Single Packet Buffering	237
18.13.2 Double Packet Buffering	237
18.13.3 High Bandwidth Isochronous Endpoints	238
18.13.4 Optional Special Handling	239
18.14 OUT Endpoint Packet Handling	241
18.14.1 Single Packet Buffering	241
18.14.2 Double Packet Buffering	241
18.14.3 High Bandwidth Isochronous Endpoints	242
18.14.4 Optional Special Handling	244
18.15 Bulk Transactions	246
18.15.1 Bulk IN Endpoint	246
18.15.2 Bulk OUT Endpoint	247
18.15.3 Interrupt Transactions	249
18.16 Isochronous Transactions	251
18.16.1 Isochronous IN Endpoint	251
18.16.2 Isochronous OUT Endpoint	252
18.16.3 High-speed High-bandwidth OUT Endpoint	255
18.17 Transaction Flows	258
18.17.1 Control Transactions	258
18.17.2 Bulk/Interrupt Transactions	263
18.17.3 Full-speed Low-bandwidth Isochronous Transaction	265
18.17.4 High-bandwidth Isochronous Transactions	267
18.18 Test Modes	269
18.18.1 Test Mode Test_SE0_NAK	269
18.18.2 Test Mode Test_Test_J	269
18.18.3 Test Mode Test_K	269
18.18.4 Test Mode Test_Packet	269
18.19 USBPHY Tuning Guidelines	270
18.19.1 Hardware Requirements	270
18.19.2 Tuning Steps	270
19. Secure Digital Input Output (SDIO)	275
19.1 Functional Description	275
19.1.1 Host Controller Register Set	275
19.1.2 PIO/DMA Controller	275
19.1.3 Block Buffer	276
19.1.4 SD Clock Generation	276
19.1.5 SD Card Detect	276
19.1.6 SD Timeout Control	276
19.1.7 SD Command Control	277
19.1.8 SD Transmit Control	277

19.1.9 SD Receive Control	277
19.1.10 SD Tuning Block	277
19.1.11 SD Intf Control	278
19.2 Clocks	279
19.2.1 SD clocking Architecture	279
19.3 Advanced DMA	281
19.3.1 Block Diagram of ADMA2	281
19.3.2 An Example of ADMA2 Programming	282
19.3.3 Data Address and Data Length Requirements	282
19.3.4 Descriptor Table	283
19.4 Driver flow sequence	284
19.4.1 Non-DMA Transaction	284
19.4.2 DMA Transaction	285
19.4.3 ADMA Transactions	288
19.5 Using the AmbiqSuite SDK to Program and Use the SDIO Module	290
19.5.1 Calibration for 48 MHz SDR or DDR	294
19.6 SD Card I/O Voltages	295
20. Display Controller (DC)	296
20.1 Functional Overview	296
20.2 Software Support	297
20.2.1 Supported Software	297
20.3 Timing Generator	298
20.4 Layer Overlays	299
20.5 Blending Modes	302
20.6 Palette/Gamma Correction	303
20.7 Dithering	304
20.8 Color Modes	305
20.8.1 Input Formats	305
20.8.2 Output Formats	308
20.9 TSc Framebuffer Decompression	310
20.10 Display Formats	311
20.10.1 MIPI DPI-2 (Display Pixel Interface)	311
20.10.2 MIPI DBI-Type B (Display Bus Interface)	314
20.10.3 SPI, 3-/4-wire Serial Peripheral Interface	319
20.10.4 DualSPI and QuadSPI Interface	325
20.10.5 MIP RGB Interface	327
20.10.6 Serial Formats	329
21. Display Serial Interface (DSI)	334
21.1 Features	334
21.2 Functional Overview	336
21.3 Hardware Design Guidelines	338
21.3.1 System Power Sequencing for DSI TX Interface	338
21.4 MIPI DSI Video Mode	338
21.4.1 What DSI Video Mode Is	338
21.4.2 Data Path	338
21.4.3 Initialization Sequence	339
21.4.4 Framebuffer and Layer Setup	340
21.4.5 VSYNC Callback	341
21.4.6 Video Mode Types	341
21.4.7 Command Mode vs Video Mode (in this codebase)	341
21.4.8 Summary	342
22. Graphics and the GFX Library on the Apollo5 Family MCUs	343

22.1 Introduction to Graphics	343
22.2 GPU and Graphics Software Support	345
22.3 GFX Library Architecture	346
22.4 Graphics Pipeline of the GPU	348
22.4.1 Instruction Set Architecture	348
22.4.2 Configuration Register File	355
22.4.3 Command List Processor	355
22.4.4 Vertex Processing	356
22.4.5 Rasterizer	356
22.4.6 Texture Map Unit	356
22.4.7 Fragment Processing Core	356
22.4.8 Render Output Unit	356
22.5 Frame Buffer Compression	358
22.6 GFX Library API Guidelines	361
22.6.1 Command Lists	361
22.6.2 Binding Textures	362
22.6.3 Clipping	363
22.6.4 Blending - Programming the Core	364
22.6.5 Drawing	364
22.7 Color Modes and Binding Textures	367
22.7.1 Color Modes	367
22.7.2 Binding Textures	368
22.8 Geometry Primitives	371
22.9 Blending	372
22.9.1 Blending in the Graphics Core	372
22.9.2 Notations and Conventions	372
22.9.3 Predefined Blending Modes	372
22.9.4 User Defined Modes	375
22.9.5 Additional Operations	376
22.10 Fonts	379
22.11 GFX Library Platform Porting	381
22.11.1 Platform Specific HAL	381
22.11.2 nema_sys_defs.h	381
22.11.3 nema_hal.c	381
22.11.4 Bare Metal Display Control Library	385
22.12 GFX Library Functions	392
22.12.1 nema_blender.h File Reference	392
22.12.2 nema_font.h File Reference	410
22.12.3 nema_graphics.h File Reference	412
22.12.4 nema_hal.h File Reference	421
22.12.5 nema_math.h File Reference	426
22.12.6 nema_matrix3x3.h File Reference	433
22.12.7 nema_matrix4x4.h File Reference	437
22.12.8 nema_programHW.h File Reference	441
23. PDM-to-PCM Converter (PDM)	453
23.1 PDM Clock Configuration	453
23.1.1 PDM Clock Duty Cycle	454
23.1.2 Clock Gating and Data Synchronization	455
23.2 System Gain and Transfer Function	457
23.2.1 Compensation Gain Questions and Examples	457
23.3 Operating Modes	459
23.4 Supported Data Format	460

23.5 Digital Volume Control	461
23.5.1 Soft Mute	461
23.6 Low Pass Filter (LPF)	462
23.7 High Pass Filter (HPF)	462
24. Low Power Analog Audio Interface	464
24.1 Functional Overview	464
24.1.1 Clock Source and Dividers	464
24.1.2 Voltage Reference Source	464
24.1.3 Eight Automatically Managed Conversion Slots	464
24.2 Main Configuration	465
24.2.1 Slot Configuration	465
24.2.2 Sample Data Format	465
24.3 Automatic Sample Accumulation and Scaling	466
24.4 Sixteen Entry Result FIFO	467
24.5 DMA	469
24.6 Operating Modes and the Mode Controller	470
24.6.1 Single Mode	471
24.6.2 Repeat Mode	472
24.6.3 Low Power Modes	472
24.7 Interrupts	473
24.8 Generating the Sample Rate for the Audio ADC	474
25. Inter-IC Sound (I2S)	475
25.1 I2S Clock Management	475
25.2 Transmit and Receive FIFOs	477
25.3 DMA	478
25.4 Interrupts	480
25.5 Data Configurations	481
25.6 Configuration and Control	483
25.6.1 Manager/Subordinate Configurations	483
25.6.2 Module Initialization	483
25.6.3 Data Flow Control	483
25.7 Serial Audio Interface	485
25.7.1 TDM Serial Audio Format	485
25.7.2 I2S Serial Audio Format	487
25.7.3 Left-justified Serial Audio Format	487
25.7.4 Right-justified Serial Audio Format	488
26. BLE Controller	489
26.1 Configuration	489
26.2 Initialization	489
26.3 General Use	490
26.3.1 Pairing	490
26.3.2 Send data over BLE	490
26.3.3 Receive data through BLE	490
26.3.4 Disconnection	490
26.3.5 How to Use uart_ble_bridge to Perform DTM Testing	491
26.3.6 BLE Controller Firmware Update	491
26.3.7 BLE Controller NVDS Configuration	492
26.3.8 Useful APIs of the BLE Controller	492
26.3.9 OTA	492
26.4 Vendor Specific HCI commands	493
27. Ordering Information	496

List of Figures

Figure 1. OTP General Flow	56
Figure 2. OTP Power-On Flow	57
Figure 3. OTP Power-Off Flow	58
Figure 4. Trim Value Determination	61
Figure 5. Frequency Offset Based on XTAL Trim Value	65
Figure 6. Trim Results	66
Figure 7. RFX TAL Configuration	66
Figure 8. Timer Edge Mode (TMRnFN = 0x1) - CMP0 > CMP1	90
Figure 9. Timer Edge Mode (TMRnFN = 0x1) - CMP0 < CMP1	90
Figure 10. Timer Repeated Pulse Up-counter Compare Mode (TMRnFN = 0x2)	91
Figure 11. Timer PWM Mode (TMRnFN = 0x4)	92
Figure 12. Timer Single 32-bit Pattern Output (TMRnFN = 0xC)	93
Figure 13. Timer Single 64-bit Pattern Output (TMRnFN = 0xC)	93
Figure 14. Single 32-bit Pattern Output	94
Figure 15. Single 64-bit Pattern Output	94
Figure 16. Scan Flowchart	149
Figure 17. XIP Block Diagram	156
Figure 18. MSPI TX Interface Timing at SCLK = 48 MHz	163
Figure 19. MSPI RX Interface Timing at SCLK = 48 MHz	164
Figure 20. MSPI RX Interface Timing at SCLK = 96 MHz	165
Figure 21. SDR with Non-DQS Mode Timing	166
Figure 22. DDR with DQS Mode Timing	167
Figure 23. OCTAL/HEX DDR with DQS Mode Timing	168
Figure 24. HEX DDR with DQS Mode Timing	168
Figure 25. I2C/SPI Manager Clock Generation	170
Figure 26. Basic I2C Conditions	172
Figure 27. I2C Acknowledge	173
Figure 28. I2C 7-bit Address Operation	173
Figure 29. I2C 10-bit Address Operation	174
Figure 30. I2C Offset Address Transmission	174
Figure 31. I2C Write Operation with Address Offset	174
Figure 32. I2C Read Operation with Address Offset	175
Figure 33. I2C Write Operation with No Address Offset	175
Figure 34. I2C Read Operation with No Address Offset	176
Figure 35. SPI Normal Write Operation (Single-byte Offset Address)	178
Figure 36. SPI Normal Read Operation	178
Figure 37. SPI Raw Write Operation	178
Figure 38. SPI Raw Read Operation	179
Figure 39. SPI Combined Operation	179
Figure 40. SPI CPOL and CPHA	180
Figure 41. Flow Control at Beginning of a Write Transfer	181
Figure 42. Flow Control at Beginning of a Raw Read Transfer	182
Figure 43. Flow Control in the Middle of a Write Transfer	182
Figure 44. Flow Control in the Middle of a Read Transfer	182
Figure 45. I2C/SPI Subordinate Module LRAM Addressing in Standard Address Mode	184
Figure 46. I2C/SPI Subordinate Module FIFO	189
Figure 47. Basic I2C Conditions	193
Figure 48. I2C Acknowledge	194

Figure 49. I2C 7-bit Address Operation	194
Figure 50. I2C 10-bit Address Operation	195
Figure 51. I2C Offset Address Transmission	195
Figure 52. I2C Write Operation	195
Figure 53. I2C Read Operation	196
Figure 54. SPI Write Operation	197
Figure 55. SPI Read Operation	198
Figure 56. SPI CPOL and CPHA	198
Figure 57. USB Interrupt Service Routine	220
Figure 58. Endpoint 0 States	230
Figure 59. Endpoint 0 Service Routine	232
Figure 60. SETUP Phase of Control Transfer	233
Figure 61. IN Data Phase for Control Transfer	234
Figure 62. OUT Data Phase for Control Transfer	235
Figure 63. High-speed Isochronous IN Endpoint Transmission	239
Figure 64. Packets Sent per Microframe	243
Figure 65. High-Speed High-Bandwidth IN Endpoint	255
Figure 66. High-speed High-bandwidth OUT Endpoint	256
Figure 67. Setup Phase Transaction	258
Figure 68. IN Data Phase Transaction	259
Figure 69. Status Phase Following IN Data Phase Transaction	260
Figure 70. OUT Data Phase Transaction	261
Figure 71. Status Phase Following OUT Data Phase Transaction	262
Figure 72. Bulk/Interrupt IN Transaction	263
Figure 73. Bulk OUT Transaction	264
Figure 74. Full-speed Isochronous IN Transaction	265
Figure 75. Full-speed Isochronous OUT Transaction	266
Figure 76. High-bandwidth Isochronous IN Transaction	267
Figure 77. High-bandwidth Isochronous OUT Transaction	268
Figure 78. Assignable Values of the USBPHY_REG0C_BF62 (ODT) Register Field	270
Figure 79. ODT Settings and Resulting Eye Diagrams for Validation Hardware	271
Figure 80. HS Eye Height Settings and Resulting Eye Diagrams for Validation Hardware ..	272
Figure 81. Pre-emphasis Settings and Resulting Eye Diagrams for Validation Hardware ...	273
Figure 82. Assignable Values of the USBPHY_REG08_BF30 (Squelch Trigger Point) Field	274
Figure 83. SD/SDIO Host Controller Interface Block Diagram	275
Figure 84. SDIO Module Clock Derivation	280
Figure 85. ADMA2 Block Diagram	281
Figure 86. Example Descriptor Table	282
Figure 87. 32-bit Addressing Descriptor Table	283
Figure 88. Descriptor Table Length Field Definitions	283
Figure 89. Non-DMA Transfer Flow Sequence	284
Figure 90. DMA Transfer Flow Sequence	286
Figure 91. ADMA Transaction Flow Sequence	288
Figure 92. Display Controller Timing	298
Figure 93. Display Controller RGBA Background Color	299
Figure 94. Display Controller First Layer	299
Figure 95. Display Controller Second Layer	300
Figure 96. Gamma Correction of RGB Values	303
Figure 97. Dithering on Limited Color Palette	304
Figure 98. FTSC™4 /TSC™6 Framebuffer Compression Module	310

Figure 99. DPI-2 Interface.....	311
Figure 100. DPI-2 Interface Timing Sequence.....	312
Figure 101. DBI-Type B Write Sequence.....	314
Figure 102. DBI-Type B Read Sequence	315
Figure 103. SPI 3- or 4-wire Interface (Dashed Line Used in 4-wire) - Data Out Only	320
Figure 104. SPI 3- or 4-wire Interface (Dashed Line Used in 4-wire) - Bidirectional Data	320
Figure 105. SPI 3-wire Serial Write Transmission	321
Figure 106. SPI 4-wire Serial Write Transmission	321
Figure 107. Single Line Update Mode	321
Figure 108. Multiple Lines Update Mode	322
Figure 109. QuadSPI Interface	325
Figure 110. DualSPI Interface.....	326
Figure 111. Vertical Mode.....	327
Figure 112. Horizontal Mode.....	328
Figure 113. Serial 3-beat RGB Timing.....	329
Figure 114. Serial 4-beat RGB Timing.....	330
Figure 115. Two-Phase Serial 12-bit RGB.....	330
Figure 116. DSI Controller Block Diagram.....	334
Figure 117. Display Serial Interface Bus with DSI Devices	336
Figure 118. Layers in the DSI Data Transfer Model	337
Figure 119. Rendering Flow.....	344
Figure 120. GFX Library Architecture	346
Figure 121. GPU VLIW fragment instruction format	348
Figure 122. TSC™4 /TSC™6/TSC™12 Framebuffer Compression Module.....	358
Figure 123. NEMA® PIX-Presso State after a Conversion.....	359
Figure 124. Original Empty Framebuffer.....	365
Figure 125. Image Background.....	366
Figure 126. Final Output of the Drawing Process	366
Figure 127. Rendered Scene with Two Icons	368
Figure 128. Scene Textures.....	369
Figure 129. Predefined Blending Modes.....	374
Figure 130. Original Framebuffer before Blending.....	374
Figure 131. Scene Textures.....	375
Figure 132. User-defined Blending Modes	376
Figure 133. Source Textures	378
Figure 134. Additional Operations Example	378
Figure 135. Vector and Bitmap Fonts	379
Figure 136. Clock Path and Data Synchronization Diagram.....	456
Figure 137. PDM Converter Core Local Clock Gating	456
Figure 138. Compensation Gain Adjustment.....	458
Figure 139. Scan Flowchart.....	471
Figure 140. I2S Block Diagram.....	475
Figure 141. I2S Serial Clock Generation in Manager Mode	476
Figure 142. DMA Configurations for 16 Bits and 24 Bits	481
Figure 143. DMA Configurations for 8 Bits and 32 Bits	482
Figure 144. Programmable Frame Period and Width	485
Figure 145. Data Delay	485
Figure 146. Dual-phase Frame Example	486
Figure 147. Right Justification.....	486
Figure 148. I2S Formatted Audio Frame	487

Figure 149. Left-justified Audio Frame..... 487

Figure 150. Right-justified Audio Frame 488

List of Tables

Table 1: Document Revision History	21
Table 2: DMA Configuration Fields per Supported Module	24
Table 3: XTALHSCAPTRIM [9:6] Value and Corresponding Shunt Cap Value	61
Table 4: Supported Clock Modes/Rates	75
Table 5: Clock Configuration Pairs Allowing Live Switching	76
Table 6: Board Information	77
Table 7: Clock Configuration	77
Table 8: USB Application Clock Source Selection	80
Table 9: UART Application Clock Source Selection	81
Table 10: Audio Application Clock Source Selection	82
Table 11: Alarm RPT Function	84
Table 12: Timer Modes	87
Table 13: Interrupt Trigger Options	114
Table 14: IO Manager 0 I2C Configuration	116
Table 15: IO Manager 0 4-wire SPI Configuration	116
Table 16: IO Manager 0 3-wire SPI Configuration	117
Table 17: IO Subordinate I2C Configuration	117
Table 18: IO Subordinate 4-wire SPI Configuration	118
Table 19: IO Subordinate 3-wire SPI Configuration	118
Table 20: IO Subordinate 4-wire SPI Configuration	119
Table 21: IO Subordinate 3-wire SPI Configuration	119
Table 22: DISP_SPI Interface Configuration	120
Table 23: DISP_QSPI Interface Configuration	120
Table 24: MIP Display Interface Configuration	121
Table 25: DISP_DPI-2 Interface Configuration	121
Table 26: DISP_DBIB Interface Configuration	123
Table 27: UART0 TX Configuration	124
Table 28: UART0 RX Configuration	124
Table 29: UART0 RTS Configuration	125
Table 30: UART0 CTS Configuration	125
Table 31: UART1 TX Configuration	125
Table 33: UART1 RTS Configuration	126
Table 32: UART1 RX Configuration	126
Table 35: UART2 TX Configuration	127
Table 36: UART2 RX Configuration	127
Table 34: UART1 CTS Configuration	127
Table 37: UART2 RTS Configuration	128
Table 38: UART2 CTS Configuration	128
Table 39: UART3 TX Configuration	129
Table 40: UART3 RX Configuration	129
Table 41: UART3 RTS Configuration	130
Table 42: UART3 CTS Configuration	130
Table 43: PDM CLK Configuration	130
Table 44: PDM DATA Configuration	130
Table 45: I2S CLK Configuration	131
Table 46: I2S WS Configuration	131
Table 48: I2S SDIN Configuration	132
Table 47: I2S DATA Configuration	132

Table 49: I2S SDOOUT Configuration	133
Table 50: SDIO/SDIF Configuration	133
Table 51: CLKOUT Configuration for Apollo510 and Apollo510B	134
Table 52: CLKOUT Configuration for Apollo510 Lite and Apollo330 Plus	134
Table 53: 32 kHz CLKOUT Configuration	135
Table 54: CLKOUT_32M Configuration	136
Table 55: ADC Analog Input Configuration	136
Table 56: ADC Trigger Input Configuration	136
Table 57: Voltage Comparator Reference Configuration	138
Table 58: Voltage Comparator Input Configuration	138
Table 59: Voltage Comparator Output Configuration	138
Table 60: SWO Configuration	139
Table 61: SW Trace Configuration	140
Table 62: SLOT Configuration Register	143
Table 63: Per Slot Sample Accumulator	144
Table 64: Accumulator Scaling	144
Table 65: FIFO Register	145
Table 66: 12-bit FIFO Data Format	145
Table 67: 10-bit FIFO Data Format	145
Table 68: 8-bit FIFO Data Format	146
Table 69: Window Comparator Lower Limit Register	147
Table 70: Window Comparator Upper Limit Register	147
Table 71: GPADC Power Modes	150
Table 72: Command Queue Example	158
Table 73: CQFLAGS	159
Table 74: Direct Area Access Interrupt Mapping and Corresponding REGACCINTSTAT Bits 186	
Table 75: I/O Interface Interrupt Control	191
Table 76: HOST_IER Register	201
Table 77: HOST_IER Register Bits	201
Table 78: HOST_ISR Register	202
Table 79: HOST_ISR Register Bits	202
Table 80: HOST_WCR Register	202
Table 81: HOST_WCR Register Bits	203
Table 82: HOST_WCS Register	203
Table 83: HOST_WCS Register Bits	203
Table 84: FIFOCTRLO Register	204
Table 85: FIFOCTRLO Register Bits	204
Table 86: FIFOCTRUP Register	204
Table 87: FIFOCTRUP Register Bits	204
Table 88: FIFO Register	205
Table 89: FIFO Register Bits	205
Table 90: Index 0 (IDX0) Register Multi-function Fields	221
Table 91: USB Controller Register Map Additions for Dynamic FIFO Sizing	226
Table 92: PID Errors and IncompRx Responses in Peripheral Mode	244
Table 93: IDX0 Register Field Settings for Bulk IN Endpoint	246
Table 94: IDX0 Register Field Settings for Bulk OUT Endpoint	248
Table 95: IDX0 Register Field Settings for Isochronous IN Endpoint	251
Table 96: IDX0 Register Field Settings for Isochronous OUT Endpoint	253
Table 97: PID Error or IncompRx Reporting in Peripheral Mode	257

Table 98: Assignable Values of the USBPHY_REG08_BF64 (HS Eye Height) Field	272
Table 99: Assignable Values of the USBPHY_REG08_BF30 (Squelch Trigger Point) Field ..	273
Table 100: Non-DMA Transfer Flow Sequence Steps	285
Table 101: DMA Transfer Flow Sequence Steps	286
Table 102: ADMA Transaction Flow Sequence Steps	289
Table 103: Layer 0 Example	300
Table 104: Blending Modes	302
Table 105: Pinout of MIP RGB LCD and Connection to the Equivalent DC Pins	327
Table 106: Pinout of MIP LCD and Connection to the Equivalent DC Pins	327
Table 107: Command Mode and Video Mode Comparison	342
Table 108: DestA Operand	349
Table 109: SUBOP0-Instructions	349
Table 110: A0 Operand	349
Table 111: A1 Operand	350
Table 112: A2 Operand	350
Table 113: SUBOP1 Instructions	351
Table 114: DestRGB Operand	351
Table 115: RGB0 Operand	352
Table 116: RGB1 Operand	352
Table 117: RGB2 Operand	353
Table 118: SUBOP2 Instructions	353
Table 119: Pixel Operand	354
Table 120: Addressing Coords Operand	354
Table 121: TEXnBASE Coords Operand	354
Table 122: SUBOP3 Instructions	355
Table 123: LHS, RHS Operands	355
Table 124: Supported Formats	367
Table 125: Shader conventions	368
Table 126: Predefined Blending Modes	373
Table 127: Blend Factors	376
Table 128: ops Arguments	378
Table 129: PDMA_CKO and OSR Settings for Different Sampling Frequencies	454
Table 130: PDM-to-PCM Converter Operating Mode	459
Table 131: 24-bit Unpacked Data	460
Table 132: PGA Gain Control	461
Table 133: SOFTMUTE Register Configuration	461
Table 134: LPF Parameters	462
Table 135: PDM High Pass Filter -3 dB Cutoff Frequency Table	463
Table 136: SLOT Configuration Register	465
Table 137: Per Slot Sample Accumulator	466
Table 138: Accumulator Scaling	467
Table 139: FIFO Register	467
Table 140: 12-bit FIFO Data Format	467
Table 141: 10-bit FIFO Data Format	468
Table 142: 8-bit FIFO Data Format	468
Table 143: Audio ADC Power Modes	472
Table 144: Configurations Supported by the I2S Module	481
Table 145: BLE Disconnection Reasons	491
Table 146: HCI Commands	493
Table 147: API Function Reference	494

Table 148: HCI_VSC_START_TRANSMIT_TEST Parameters 494

Table 149: HCI_VSC_SET_DEV_PUB_ADDR Parameters 495

Table 150: HCI_VSC_SET_LOCAL_SUP_FEAT Parameters 495

Table 151: Ordering Information 496

1. Document Revision History

Table 1: Document Revision History

Revision	Date	Description
1.0	Apr 2026	Document initial release
1.1	July 2026	Cover page: - Updated document title and name CLKGEN: - "RF XTAL Frequency Trimming on the Apollo510 Lite and Apollo330 Plus SoCs" section added DSI: - Video Mode section added

2. Introduction

The purpose of this Technical Reference Manual is to provide developers using an Apollo5 family SoC with additional information to that found in the individual family member datasheets. The information provided in this Technical Reference Manual addresses a more detailed level of how the SoC and its on-chip modules and interfaces operate and how to control them programmatically for their intended operation.

The Apollo510 is the flagship device of the Apollo5 family which is built off of the Apollo4 Plus SoC with the following enhancements¹:

- Upgrades to Cortex-M55 CPU with HP performance increased to 250 MHz
- Increases total SRAM to 3.75 MB (3 MB SSRAM, 256 kB CPU I-TCM / 512 kB CPU D-TCM)
- Increases total MRAM to 4 MB
- Enhanced robustness of secure boot with use of MRAM-OTP/eFuse/ROM improvements and MRAM recovery flow
- Enhanced GPU
 - Increases frequency to 250 MHz
 - Increases cache line buffer and burst length to increase external memory performance
 - Radial color fill support
 - Scalable Vector Graphics (SVG) support
 - TrueType Font (TTF) support
 - TSC12 and TSC12a
- MSPI
 - Frequency increase to 125 MHz DDR for x16 controllers
 - Four MSPI instances total
 - 2x capable of x16 up to 250 MT/s
 - 2x capable of x8 up to 96 MT/s
- MIPI DSI 768 Mbps per lane support (requires update to MIPI DSI controller), 1.5 Gbps total
- Apollo510 BGA package supports 183 GPIO, WLCSP supports 144 GPIO
- Added second SDIO controller
- Low active and Deep Sleep power modes (additional Deeper Sleep Mode or “Ship Mode” in Apollo330 Plus Series SoCs and Apollo510 Lite Series SoCs)

The current SoCs that make up the Apollo5 family include the following:

- Apollo510
- Apollo510B
- Apollo510 Lite
- Apollo510B Lite
- Apollo510D Lite

Unless stated otherwise, references in this guide to “Apollo5 SoC” pertain to all members of the Apollo5 family as well as to newer members of the Apollo3 family, beginning with Apollo330 Plus.

This Technical Reference Manual (TRM) and the relevant datasheet(s) are intended to supplement each other and the user should have ready access to each. This TRM also refers to the AmbiqSuite SDK extensively, as the SDK is the programmers’ primary source of software to use and control the Apollo5 family SoCs’ operation properly. Regarding any reference to registers or bit fields herein, be sure to refer to the register set documentation specific to the SoC of interest which is available in the latest version of the AmbiqSuite SDK. As well, be sure to check the latest errata list document for the SoC of interest to be aware of any issues which might affect operation or features specified in this document.

1. The features and sizes of memory/storage enabled are specific to each SoC. Please see the applicable datasheet(s) for the exact feature set of each device.

3. SoC Architecture

Please refer to the CPU Subsystem feature list and functional overview sections in the datasheet for an overview of the its capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the SoC's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

3.1 DMA Overview

DMA-supported Modules:

- MSPIs (peripheral-to-memory and peripheral-to-peripheral; supports autonomous transfer of data between an external device and internal memory or MRAM)
- IOM (peripheral-to-memory and peripheral-to-peripheral)
- IOS/IOSFDs (half-duplex and, for IOSFD, full duplex modes)
- UARTs (half-duplex mode only, but supports full duplex using two UARTs)
- SDIOs (Advanced DMA version 2 -- ADMA2)
- USB (Concurrent DMA (DMA Mode 1) is supported for both BULK and INTERRUPT endpoint types, but not for Isochronous mode).
- ADC
- AUDADC
- PDM-to-PCM (supports ping-pong DMA jobs)
- I2S (supports ping-pong DMA jobs; enables efficient transfer of data to/from SRAM)
- Display Controller (IO-MMU)
- GPU (Command list-based DMAs)

There exist two kinds of DMA in the system:

- APB-DMA: IOM, MSPI, IOS, Crypto
- HS-DMA (high-speed DMA): eMMC/SDIO, USB, GPU, DC

DMA for each of the HS-DMA peripherals is operationally different. Consult the register set for each peripheral to understand its operational control of DMA.

General limitations for the APB bus use of DMA (APB-DMA) - memory/storage:

- Transferring between memory/NVM (DTCM, SSRAM, MRAM) source and another memory destination is not supported as the APB-DMA does not have the capability to transfer from memory to memory.
- A DMA operation cannot be performed to or from DTCM when the CPU is in deep sleep or the DTCM location is powered down. Access to TCM while the CPU is in deep sleep with memory retained or when the DTCM is powered down returns an incorrect error condition (Success).
- A DMA transfer must not cross a DTCM 4 kB boundary¹.
- A DMA transfer which has a start address near the upper boundary of an SRAM memory block and the transfer extends into the next memory block, a device hang occurs. This condition applies for several boundaries, which are the DTCM - SSRAM0, SSRAM0 - SSRAM1 and SSRAM1 - SSRAM2 boundaries.
- For MRAM, if CPU PMU requester is "OFF" and DMA requesters are "OFF" or in "Sleep", the MRAM power domain is powered OFF.
- For SRAM, each SRAM is powered OFF based either on the SKU memory configuration or whether CPU requester is "OFF" and DMA requesters are "OFF" or in "Sleep" and the SRAM is enabled to power off based on the power control MEMPWDINSLEEP configuration.

1. An SDK HAL function, `am_hal_crc_find_next_boundary()`, which handles boundary checking for the CRC, can be used as a reference to prevent DMA transfers from crossing memory boundaries.

See specific module chapters in this document or the corresponding chapter in the relevant datasheet for more details about that module's support of DMA.

3.1.1 Two-Stage DMA Request Pipeline Feature

Support for a two-stage DMA request pipeline has been added in the I2S, PDM and AUDADC modules to ease software ISR latency requirements for reloading the next DMA request when the current DMA request has been completed.

3.1.1.1 Related Register Fields:

The general register field names related to this feature are:

1. NEXTDMAEN (RW): Set to enable this feature.
2. TARGADDRNEXT (RW): Target address for next DMA job.
3. TOTCNTNEXT (RW): Total count for next DMA job.
4. DMAENNEXT (RWAC): Set to indicate that the next DMA job setting is valid. When the hardware loads the next DMA job to its current job, this bit is cleared.

The mapping of the above fields for each of the supported blocks is as shown in Table 2.

Table 2: DMA Configuration Fields per Supported Module

General Register Field Name	I2S (TX) Register Field (I2S_)	I2S (RX) Register Field (I2S_)	PDM Register Field (PDM_)	AUDADC Register Field (AUDADC_)
NEXTDMAEN	DMACFG_ NEXTDMAEN	DMACFG_ NEXTDMAEN	DMACFG_ NEXTDMAEN	DMACFG_ NEXTDMAEN
TARGADDRNEXT	TXDMAADDRNEXT_ TXTARGADDRNEXT	RXDMAADDRNEXT_ RXTARGADDRNEXT	DMATARGADDRNEXT_ TARGADDRNEXT	DMATARGADDRNEXT_ TARGADDRNEXT
TOTCNTNEXT	TXDMATOTCNTNEXT_ TXTOTCNTNEXT	RXDMATOTCNTNEXT_ RXTOTCNTNEXT	DMATOTCOUNTNEXT_ TOTCOUNTNEXT	DMATOTCOUNTNEXT_ TOTCOUNTNEXT
DMAENNEXT	DMAENNEXTCTRL_ TXDMAENNEXT	DMAENNEXTCTRL_ RXDMAENNEXT	DMAENNEXTCTRL_ DMAENNEXT	DMAENNEXTCTRL_ DMAENNEXT

The software programming sequence is as follows:

1. Program NEXTDMAEN bit to 1.
2. Set up DMA as before but leave the original TARGADDR and TOTCNT fields unprogrammed. These fields will be auto loaded by hardware with TARGADDRNEXT and TOTCNTNEXT fields when the current DMA job is completed.
3. Program TARGADDRNEXT and TOTCNTNEXT.
4. Set DMAENNEXT to 1.
5. Read back the DMAENNEXT bit. For the very first DMA job, this bit will read back 0 as the current DMA job is empty, so the next DMA job is immediately loaded as the current job.
6. If DMAENNEXT reads back 0, software can program another TARGADDRNEXT and TOTCNTNEXT.
7. Wait for the DMACPL interrupt.
8. In the DMACPL interrupt service routine, do not disable DMAEN as the current DMA job is still on-going. Just repeat from step 3.

3.2 Memory Map

SoC-specific Cortex-M55 CPU (full) memory map and the Peripheral Memory Map which breaks out the various memory types and address ranges are located in the datasheet for each device.

Peripheral devices' address spaces within the memory map are allocated on 4 kB boundaries, allowing each device up to 1024 32-bit control and status registers. Unused spaces are marked as "Reserved" in the Peripheral Memory Map. Peripherals return undefined read data when an attempt to access a register which does not exist occurs. Peripherals, whether accessed via the APB or the AHB, always accept any write data sent to their registers without attempting to return an ERROR response. Specifically, a write to a read-only register just becomes a don't-care write.

3.3 Instrumentation Trace Macrocell (ITM)

For system trace the processor integrates an Instrumentation Trace Macrocell (ITM) alongside data watchpoints and a profiling unit. To enable simple and cost-effective profiling of the system events these generate, a Serial Wire Viewer (SWV) can export a stream of software-generated messages, data trace, and profiling information through a single pin.

Please refer to this module's registers in the applicable SoC's register set, which is included in the AmbiqSuite SDK.

3.4 SoC Control

High-level functionality configuration and status of the SoC are offered within an extensive set of SoC Control registers. Among the various device settings and status are the following functions:

- Chip identification and revision information
- Debug control
- Brown-out detection and power-up configuration
- ADC and Audio ADC configuration
- Programmable Gain Amplifier (PGA) configuration
- Crystal (XTAL) clock trimming
- Boot loader and secure boot loader enablement
- MRAM read/write protection
- SRAM write protection from DMA transfers

Please refer to the MCUCTRL register set which is included in the AmbiqSuite SDK.

NOTE

There is a silicon erratum that exists on the Apollo5 family which involves a system hang caused by TPIU register writes when PDDBG is not powered up. For details about this erratum and others that exist on members of the Apollo5 family, please see the errata lists at the Ambiq Content Portal - <https://contentportal.ambiq.com>.

3.5 Memory Controller Programming

The Cortex-M55 supports the ARMv8.1 ISA. Details on the instruction set and overall programming model can be referenced in the "ARM Cortex-M55 Devices Generic User Guide" and the "ARM Cortex-M55 Processor Technical Reference Manual".

Please refer to the CPU register set which is included in the AmbiqSuite SDK.

3.6 Memory/Cache Coherency Management

3.6.1 Background

The Cortex-M55 provides integrated instruction and data caches, but there is no cache coherency hardware. Thus, all of the cache coherency management must be done in software.

- I-cache: invalidates by entire cache or by address
- D-cache: invalidate or clean by entire cache, address or set/way

Data cache operations are further delineated into operations to Point of Unification (PoU) and Point of Coherency (PoC). PoU means that the current processing element's data and instruction caches are guaranteed to see the same copy of a memory location. PoC means that all processing elements in the system are guaranteed to have the same view of a memory location. Operations to PoC are needed when data is to be shared between different processing elements.

There are three sets of attributes that determine the cache behavior:

1. Memory type: normal or device. Only normal memory may be cached.
2. Cacheability: write back or write through. Allocates on read miss, allocates on write miss.
3. Shareability: non-shared, inner-shared, or outer-shared.

ARM has the concept of inner and outer cacheability and sharing. These attributes were created around larger systems that might have a hierarchy of processor, cache, and memory domains. Since the Apollo5 family of devices only have one level of cache on one processor, shareability is limited to only using the inner-shared attribute. The shareability attribute is really more appropriate for Cortex-A systems with cache-coherency hardware, but ARM applies a universal memory architecture model across M, R, and A-class devices. If memory is marked as inner- or outer-shared, then it is also non-cacheable, regardless of the cache attributes. We will consider most memory as non-shared to avoid interactions with the cacheability attributes.

3.6.1.1 Device Memory

Device memory is always non-cacheable, and is treated as shareable. There are three settable memory attributes associated with device memory:

- Gather (G) / no-gather (nG)
- Reorder (R) / no-reorder (nR)
- Early-write-ack (E) / no-early-write-ack (nE)

When accessing the APB interface for a device (or any other memory-mapped register interface), then memory should be configured as nGnRnE. This insures that each write proceeds in the order it was issued.

A device memory attribute can also be assigned to the memory used for buffers to exchange data with devices or DMA. In this case, the GRE settings for the device memory attributes might be acceptable. The drawback of this memory is that it is still non-cacheable, and thus will be more expensive for the processor to access. However, it avoids the need for SW cache management.

3.6.1.2 Normal Memory

Normal memory is the memory space used for most code and data. Normal memory has many possible attributes, but the bulk of normal memory will be:

- non-shared
- write-back cacheable
- write allocate
- read allocate

These memory attributes normally provide the highest performance to the processor.

Note that write-back cacheable means that the write data is held in the cache for an indeterminate period. If the data is to be shared with a device or DMA, the line must be cleaned before sharing it with the that device.

Data memory should be marked as XN, execute-never. Regions containing code must be executable and cacheable. Write-back/write-through distinctions are not meaningful for code, so such a region may have either attribute. It is important that code regions have the “read allocate” attribute.

The transient / non-transient attribute is a cache hint that gives some bias to some cache lines over others for eviction. In particular, if an eviction is required, a transient cache line and a non-transient cache line are possible eviction candidates, and the transient cache line will be the one evicted.

3.6.1.3 Memory Hazards and Barriers

Even in the absence of caches, the ARM AXI bus allows out-of-order operations. Because there are independent read and write channels, there are several types of memory ordering hazards that can occur. In some cases software may be required to insert barriers to avoid the hazard. In other cases, the host must handle the ordering.

Hazards:

- Read-after-write, same location

From a given host, if a read is issued after a write operation to the same location, the read could overtake the write operation and complete before the memory has been updated. The M55 monitors outstanding writes to avoid the read-after-write hazard, so no software intervention is necessary.

- Write-after-write

Ordering of writes to different memory locations is not guaranteed, so writes may be observed out of order. This can manifest in several forms, Examples:

- A. Writing a buffer and then setting a completion flag – the update to the flag may be visible before all updates to the buffer are visible.
- B. Writing to a buffer and then programming a device read or DMA. The registers for a device or the APB DMA are on the Peripheral AHB interface, while the memory buffer may be across the AXI fabric. Register writes could easily overtake writes to buffer memory. It is also possible that reads from the device or DMA engine could overtake the writes to the buffer, thus transferring stale data.

Barriers:

The ARM V8-M defines no less than 6 different types of memory barriers (see ARMv8-M Architecture Reference, section B.7.2.9):

- Instruction Synchronization Barrier (ISB)
- Data Memory Barrier (DMB)
- Data Synchronization Barrier (DSB)
- Consumption of Speculative Data Barrier (CSDB)
- Physical Speculative Store Bypass Barrier (PSSBB)
- Speculative Store Bypass Barrier (SSBB)

For now, this section covers only ISB, DMB and DSB usage. (More research is needed on speculative execution on the M55).

ISB insures that all instructions are executed before the ISB is completed before any instructions after the ISB can be executed. Thus, an ISB should be used during any context-changing operations to insure that

operations in one context have completed before switching to a new context. (ARMv8M ARM, section B7.2.10).

DMB is the weakest of these three barriers. It insures that all memory accesses executed on a PE before the DMB instruction are observable to all PEs before any instruction after the DMB is executed on that PE.

DSB is a superset of the DMB. In addition to DMB functionality, the DSB also insures that all cache and branch predictor maintenance instructions issued before the DSB are complete, and all accesses to the System Control Space issued before the DSB are complete.

3.6.2 Cache Management & Barrier Operations

3.6.2.1 APB Register Accesses

Device registers should be marked as device memory (nGnRnE) such that all reads happen in program order without combining or reordering and all writes happen in program order without combining or reordering. So no barrier is needed between writes and subsequent reads to device registers. Note that this does not take into account any device internal delays between the programming of the registers and when the effects of programming the registers could show up in the programmed device.

3.6.2.2 System Control Space Accesses

With the ARM v8-M architecture, there are two considerations for memory ordering in the System Control Space:

1. Ensure that all the writes to system registers are completed. This can be accomplished by following the writes with a DSB instruction. The DSB instruction will not complete until all previous writes have completed. This ensures that the operations will take effect – not that they have taken full effect.
2. Ensure that all the writes to system registers have taken effect in the system. According to the v8-M architecture manual, the DSB instruction is not sufficient. An ISB instruction (or other context synchronizing event) is required to ensure that these updates have taken full effect in the system. The completion of the ISB instruction ensures that all updates to system control space have taken full effect and are globally visible.

3.6.2.3 Device Buffers

Device buffers are any memory that is shared between the CPU and another device: APB DMA, GPU or any other device host capable of reading or writing memory directly.

Because of the ARM weakly-ordered memory model, and the cache without cache coherency, both memory barriers and cache management may be required for proper operation.

3.6.2.3.1 Device Memory Model

If the memory buffer is defined as device memory (it can be a more weakly-ordered memory than that used to access registers, e.g., GRE instead of nGnRnE), then it is non-cacheable. No cache-management operations are needed, but barriers are required. This management is less complex, but is significantly more expensive for CPU accesses to that memory.

CPU Writing Memory Buffers

1. Write data to memory buffer.
2. Use DSB to ensure that all writes have completed to the point of coherency.
3. Program DMA or Device to consume data.

CPU Reading Memory Buffers

Assuming device treats memory as nGnRnE, then the sequence is the following:

1. CPU programs the device or DMA.
2. Device writes data to memory buffer.
3. Device sends completion interrupt to CPU.
4. CPU may read data.

Note that if the device has the capability to buffer memory accesses, then it is implementation-dependent as to how to determine that all device writes have completed.

3.6.2.3.2 Normal Memory Model

If the memory buffer is defined as Normal memory and cacheable. Then cache-management operations are needed as well as memory barriers. This management is more complex, but it can be significantly more higher performing for CPU accesses to that memory.

CPU Writing Memory Buffers

1. Write data to memory buffer.
2. Clean (or clean and invalidate) the memory region to the point of coherency.
3. Use DSB to ensure that all cache maintenance operations have completed to the point of coherency.
4. Program DMA or device to consume data.

Note that if the memory region is marked as cacheable write-through, then the cleaning step is not required, but the DSB is still required.

CPU Reading Memory Buffers

Assuming device treats memory as nGnRnE, but the CPU treats the buffer as normal memory, then the sequence is the following:

1. CPU Invalidates or clean/invalidates the memory region to insure there is no stale data in the cache. (It may also be necessary to set CCR.DC = 0 to prevent speculative line fills.)
2. Use DSB to ensure that all cache maintenance operations have completed to the point of coherency.
3. CPU programs device or DMA.
4. Device writes data to memory buffer.
5. Device sends completion interrupt to CPU.
6. CPU may read data.

Note that if the device has the capability to buffer memory accesses, then it is implementation-dependent as to how to determine that all device writes have completed.

The following caveats relate to buffer access and alignment in “normal memory”:

1. The CPU must not access the data buffer region during the device/DMA transaction as that may lead to stale data in the cache.
2. Care should be taken to align any DMA buffers on cache-line boundaries. If a buffer is not cache-line aligned, then there may be data in the cache line before the buffer or in the cache line after the end of the buffer. If that head or tail data is accessed during the DMA operation, then that line will be pulled into the cache resulting in stale buffer data.

3.6.3 System Operations

There are a number of system operations that require complex sequences to ensure the desired operation is accomplished completely and has succeeded. They may require a sequence of register settings, barriers, and cache management operations. As noted above, DSB instructions only guarantee that the writes have completed, and an ISB instruction is needed to insure that the operations have taken full effect.

3.6.3.1 Changing SAU Regions

Changing SAU regions can only be done from the secure domain. This sequence insures that all data is coherent after an SAU programming change.

1. Set CCR.DC = 0 and CCR.IC = 0.

The purpose of this step is to prevent speculative line fills.

2. Execute a DSB instruction.

Force any outstanding memory accesses to complete.

3. Clean the affected lines in data or unified caches using the DCC* instruction.
4. Execute a DSB instruction.
5. Set SAU_CTRL.ENABLE = 0.
6. Reprogram SAU region.

(Change the security attribution of address range.)

7. Set SAU_CTRL.ENABLE = 1.
8. Set CCR.DC = 1 and CCR.IC = 1.
9. Execute a DSB instruction.
10. Invalidate the affected lines in all caches using the DCI* and ICI* instructions.
11. Execute a context synchronization event, such as an ISB instruction or an exception return.

3.6.3.2 Changing MPU Regions

When setting up the MPU, and if it has been previously programmed, disable unused regions to prevent any old settings from affecting the latest MPU setup.

1. Execute a DSB instruction to drain out any existing memory transactions.
2. Write to the MPU registers. For a complete list, see the Cortex-M55 TRM section 8.3.1 Memory Protection Unit register summary on page 8-168.
3. Execute a DSB instruction and then an ISB instruction to ensure that all subsequent memory accesses see the updated MPU setup.

3.6.3.3 Cache Maintenance Operations

An ISB is required after cache maintenance operations to insure that the effects of those operations are complete and visible to any instructions executed after the ISB instruction.

3.6.3.4 Initializing the EPU

Both DSB and ISB are needed. See M55 TRM section 5.3 for detailed sequences.

3.6.3.5 Enable or Disable the Instruction and Data Cache

Both DSB and ISB are needed. See the Cortex-M55 TRM section 5.5 for detailed sequences.

3.6.3.6 Enable the Branch Cache

The branch cache is disabled on reset. Both DSB and ISB are needed in the initialization sequence. See the Cortex-M55 TRM Section 5.5 for detailed sequences.

3.6.3.7 Self-modifying Code

Self-modifying code requires a number of DSB instructions followed by an ISB. See the Cortex-M55 TRM V8-M ARM section B7.27 for details.

3.6.4 Cache Management in the AmbiqSuite SDK

Memory/Cache Coherency Management should be performed at the application level as it is better suited to optimize the cache usage and maintenance based on system design and resource allocation. The Ambiqsuite HAL cannot handle complex application scenarios such as a case with multiple-managers when cache is enabled. If a module is used in PIO mode, then cache operations are not needed.

General Guideline for Cache Management

For the cacheable region shared by the CPU and other AXI managers (GPU, DC, APBDMA, etc.):

- Align cacheable memory buffers to 32-bytes to match cache line length if doing cache clean/invalidate by address.
- Clean the cacheable memory range that CPU has written to if it is to be used by other AXI managers.
- Invalidate the cacheable memory range that CPU is expected to read if it has been modified by other AXI managers.
- MRAM writes are done by the MCU's internal MRAM controller and do not go through the cache. The invalidate operation is required before MRAM reads by CPU or debugger and after MRAM writes.

For the non-cacheable region shared by CPU and other AXI managers (GPU, DC, APBDMA, etc., including Command Queue):

- If this region is located in tightly-coupled memory (TCM), cache clean/invalidate operations are not needed because the entire TCM is non-cacheable. The TCM does not support the cache feature.
- If this region is located in SSRAM/PSRAM(XIPMM)/MRAM, the non-cacheable region can be set using the MPU to the memory shared by the CPU and other AXI managers, so cache clean/invalidate operations are not needed. The non-cacheable regions should be 32-byte aligned.

For the cacheable XIP (executable) region in SSRAM/PSRAM:

- If the code in this region was updated by the APBDMA, I-Cache and D-Cache should be invalidated before executing code in this region.
- If the code in this region was updated by the CPU, D-Cache should be cleaned and then the I-Cache should be invalidated before executing code in this region.
- Cache clean is needed for the PSRAM region before disabling or powering off MSPI.

3.6.4.1 MPU Configuration Example Code

The following code provides an example of configuring the MPU.

```
//
// Define MPU attributes (memory type and cache behavior)
//
am_hal_mpu_attr_t sMPUAttr =
{
    .ui8AttrIndex = 0,    // Use attribute index 0
    .bNormalMem = true,   // Cachable normal memory
}
```

```
// L2 cache attribute
.sOuterAttr = {
    .bNonTransient = false,
    .bWriteBack = true,
    .bReadAllocate = false,
    .bWriteAllocate = false
},

// L1 cache attribute
.sInnerAttr = {
    .bNonTransient = false,
    .bWriteBack = true,
    .bReadAllocate = false,
    .bWriteAllocate = false
},
.eDeviceAttr = 0,
};

//
// Define MPU region and configuration (address range and permissions)
//
am_hal_mpu_region_config_t sMPUCfg =
{
    .ui32RegionNumber = 6,                // Region 0-15
    .ui32BaseAddress = (uint32_t)DMATCBBuffer, // Start address
    .eShareable = NON_SHARE,              // Not shareable
    .eAccessPermission = RW_NONPRIV,      // Read/Write, non-privileged
    .bExecuteNever = true,                // No code execution
    .ui32LimitAddress = (uint32_t)DMATCBBuffer + sizeof(DMATCBBuffer) - 1,
    .ui32AttrIndex = 0,                   // Use attribute 0
    .bEnable = true,                      // Enable region
};

//
// Configure attributes
//
am_hal_mpu_attr_configure(&sMPUAttr, 1);

//
```



```
// Clear all regions
//
am_hal_mpu_region_clear();

//
// Configure region
//
am_hal_mpu_region_configure(&sMPUCfg, 1);

//
// Invalidate and clear DCACHE, this is required by CM55 TRF
//
am_hal_cachectrl_dcache_invalidate(NULL, true);

//
// Enable MPU
//
am_hal_mpu_enable(true, true);

//
// Global buffer for DMA
// Need 32-bit alignment for correct cache operation
//
AM_SHARED_RW uint8_t g_TXBuffer[MSPI_BUFFER_SIZE] __attribute__((aligned(32)));
AM_SHARED_RW uint8_t g_RXBuffer[MSPI_BUFFER_SIZE] __attribute__((aligned(32)));

//
// Clean D-Cache after CPU writes TX buffer and before DMA writes
//
am_hal_cachectrl_range_t sRange;

// CPU prepares data in the buffer
for (uint32_t i = 0; i < MSPI_BUFFER_SIZE; i++)
{
    g_TXBuffer[i] = (i * 0xFF);
}
sRange.ui32StartAddr = (uint32_t)g_TXBuffer;
sRange.ui32Size = MSPI_BUFFER_SIZE;
am_hal_cachectrl_dcache_clean(&sRange);
DMA_WRITE_API(g_TXBuffer, g_RXBuffer, MSPI_BUFFER_SIZE, ...);
```

3.6.4.2 Additional Recommendations

- A non-cacheable buffer is recommended when the CPU seldom accesses these regions.
 - Define `SSRAM_NON_CACHABLE`, or
 - Use the MPU to define a non-cacheable region in the application.
- Cacheable buffers might be better when only the CPU needs to access the region frequently.
 - Align cacheable memory buffers to 32-bytes:


```
AM_SHARED_RW uint32_t g_ui32I2S0RxDataBuffer[2 * NUM_OF_SAMPLES] __attribute__((aligned(32)));
AM_SHARED_RW uint32_t g_ui32I2S0TxDataBuffer[2 * NUM_OF_SAMPLES] __attribute__((aligned(32)));
```
 - Cache should be cleaned before DMA/GPU/DC accesses the buffer.
 - Cache should be invalidated after DMA/GPU/DC writes the buffer and before the CPU is to access the buffer.
- Command Queue buffers supplied to the HAL must be in non-cacheable memory.
- Be careful with operations to the entire cache region or to buffers with a wide address range.
 - Cache operations to the entire cache or to an address range may be time-consuming. An entire cache clean takes 160 μ s while a 64 KB size clean takes 140 μ s and a 128 KB clean takes 279 μ s.
 - Cache operation by region should have an address range less than the cache size.

3.6.4.3 MSPI Cache Management

It is required to clean D-Cache before disabling MSPI or PSRAM when PSRAM is used for XIPMM. This applies to different PSRAM use cases including MSPI disable/de-initialization, MSPI power off, XIP disable, PSRAM half-sleep and PSRAM power off.

Also, it is required to set a non-cacheable region to cover the DMA command queue buffer.

In the case where DMA data buffers are placed in a cacheable region, do the following:

- Align DMA data buffers to 32 bytes.
- Clean the TX data buffer after it has been prepared by the CPU and before calling the DMA write API.
- For optimal performance, use the `write_adv` and `read_adv` functions from the device driver so that the user can avoid using the callback function in the device driver and manage cache coherency in the application.

The general rule of thumb is to clean D-Cache before a DMA write and invalidate D-Cache after a DMA read by region.

In the case where DMA data buffers are placed in a non-cacheable region, do the following:

- Put DMA data buffers together with the Command Queue buffer to save MPU regions.
- No other cache management operation is required.

3.6.4.3.1 D-Cache Clean and Invalidate Example Code

The following code provides an example of cleaning and invalidating D-Cache.

```
//
// Global buffer for DMA
// Need 32-bit alignment for correct cache operation
//
AM_SHARED_RW uint8_t g_TXBuffer[MSPI_BUFFER_SIZE] __attribute__((aligned(32)));
AM_SHARED_RW uint8_t g_RXBuffer[MSPI_BUFFER_SIZE] __attribute__((aligned(32)));

//
// Clean D-Cache after CPU writes TX buffer and before DMA writes
```

```

//
am_hal_cachectrl_range_t sRange;

// CPU prepares data in the buffer
for (uint32_t i = 0; i < MSPI_BUFFER_SIZE; i++)
{
    g_TXBuffer[i] = (i * 0xFF);
}
sRange.ui32StartAddr = (uint32_t)g_TXBuffer;
sRange.ui32Size = MSPI_BUFFER_SIZE;
am_hal_cachectrl_dcache_clean(&sRange);
DMA_WRITE_API(g_TXBuffer, g_RXBuffer, MSPI_BUFFER_SIZE, ...);

//
// Invalidate D-Cache after DMA read complete
//
sRange.ui32StartAddr = (uint32_t)g_RXBuffer;
sRange.ui32Size = MSPI_BUFFER_SIZE;
DMA_READ_API(g_TXBuffer, g_RXBuffer, MSPI_BUFFER_SIZE, ...); // Assuming API is blocking
am_hal_cachectrl_dcache_invalidate(&sRange, false);

```

3.6.4.3.2 MSPI write_adv and read_adv APIs

For optimal performance, MSPI write_adv and read_adv functions should be used from a device driver to avoid using the callback function in the driver and manage cache coherency in the application.

3.6.4.3 MSPI Example Code for Using the read_adv API

```

{
    bDMAComplete = false;
    ui32TimerTickBefore = am_hal_timer_read(TIMER_NUM);
    ui32Status = mspi_device_func.mspi_read_adv(pDevHandle,
                                                pTestConfig->pRxBuffer + pTestConfig->ui32ByteOffset + pTestConfig->ui32SectorOffset * ui32Loop,
                                                pTestConfig->ui32SectorAddress + pTestConfig->ui32SectorOffset * ui32Loop,
                                                pTestConfig->ui32NumBytes,
                                                0, 0,
                                                pfnMSPI_Callback, (void *)&bDMAComplete);

    ui32TimerTickAfter = am_hal_timer_read(TIMER_NUM);
    if (AM_ABSTRACT_MSPI_SUCCESS != ui32Status)
    {
        am_util_stdio_printf("Failed to read from target sector. Status = %d\n", ui32Status);
        break;
    }

    for (uint32_t i = 0; i < MSPI_TIMEOUT; i++)
    {
        if (bDMAComplete)
        {
            ui32TimerTickAfter = am_hal_timer_read(TIMER_NUM);
            break;
        }
    }
    if (!bDMAComplete)
    {
        break;
    }

    num_of_tick_read += cal_time(ui32TimerTickBefore, ui32TimerTickAfter);
}
if(ui32Loop == 0)

    num_of_tick_read /= ui32Loop;
}

am_hal_sysctrl_bus_write_flush();

```

3.6.4.4 Recommended Cache Maintenance for IOM Applications

Cache maintenance for IOM is almost the same as for MSPI because IOM and MSPI use the same Command Queue (CQ) logic for DMA transfers. If data and the CQ buffer are located in TCM or a non-cacheable SSRAM region, then there is no need for cache operations. A DMA CQ buffer is required to move data to a non-cacheable region (see “MPU Configuration Example Code” on page 31) If DMA data buffers are in a cacheable region, then cache maintenance is needed in the application code.

If cache clean/invalidate is being done by address, then the buffer needs to be 32-byte aligned.

The TX data buffer must be cleaned after it has been prepared by the CPU and before the DMA write API is called. See example code below and refer to iom_psram example for more details.

```

init_pattern(void)
{
+   am_hal_cachectrl_range_t sRange;
+
    uint32_t i;
    for (i = 0; i < PATTERN_BUF_SIZE; i++)
    {
        gPatternBuf.bytes[i] = i & 0xFF;
    }
+   sRange.ui32StartAddr = (uint32_t)gPatternBuf.bytes;
+   sRange.ui32Size = PATTERN_BUF_SIZE;
+   am_hal_cachectrl_dcache_clean(&sRange);
}

```

3.6.4.5 Recommended Cache Maintenance for Non-CQ Peripheral Applications

Peripherals without CQ support (UART, IOS, I2S, PDM, AUDADC, GPADC) should use the same cache operations as described above if their source/target buffers are in cacheable regions.

If cache clean/invalidate is being done by address, then the cacheable memory buffers need to be 32-byte aligned to match cache line length as shown below.

```
AM_SHARED_RW uint32_t g_ui32I2S0RxDataBuffer[2 * NUM_OF_SAMPLES] __attribute__((aligned(32)));
```

```
AM_SHARED_RW uint32_t g_ui32I2S0TxDataBuffer[2 * NUM_OF_SAMPLES] __attribute__((aligned(32)));
```

The cacheable memory range that the CPU has written to must be invalidated if it is to be used by the non-CQ peripheral. For example, invalidation is done by calling `am_hal_cachectrl_dcache_invalidate(&sRange, false)` after RXDMA completion and before CPU reading.

```

//
// check for DMA completion
//
if (g_ui32I2SDmaCpl[0] && g_ui32I2SDmaCpl[1] && g_ui32I2SDmaCpl[2] && g_ui32I2SDmaCpl[3])
{
    //
    // Get RX and TX buffer addresses.
    //
    uint32_t i2s1_rx_buffer = am_hal_i2s_dma_get_buffer(pI2S1Handle, AM_HAL_I2S_XFER_RX);
    uint32_t i2s0_tx_buffer = am_hal_i2s_dma_get_buffer(pI2S0Handle, AM_HAL_I2S_XFER_TX);
    //
    // Invalidate RX buffer, ensure that what the CPU sees is consistent with the contents of
    // external memory.
    //
    am_hal_cachectrl_range_t sRange = {(uint32_t)i2s1_rx_buffer, sizeof(uint32_t) *
NUM_OF_SAMPLES};

```

```

    am_hal_cachectrl_dcache_invalidate(&sRange, false);
    //
    // CM55 reads and checks data in RX buffer.
    //
    check_i2s_data(NUM_OF_SAMPLES, (uint32_t*)i2s1_rx_buffer, (uint32_t*)i2s0_tx_buffer);
}

```

If the source/target buffers are allocated in a non-cacheable region and located in TCM, there is no need for any cache operations or MPU configuration. If the buffers are located in SSRAM, PSRAM (XIPMM) or MRAM, then that region should be set to non-cacheable using the MPU. Refer to the example code in “MPU Configuration Example Code” on page 31 or to the SDK example `mpu_example`. All memory regions should be configured by using the MPU if there are any non-privileged accesses in the application.

3.6.4.6 Recommended Cache Maintenance for Graphics Applications

Non-cachable graphics buffers are recommended and requires either enabling `SSRAM_NON_CACHABLE` or setting all the graphics buffers to non-cachable by MPU (refer to previous MSPI instructions.)

If cacheable buffers must be used, then the buffers need to be 32-byte aligned. Command list buffers are fully taken care by `nema_hal.c` and no application intervention is needed.

After the CPU writes to texture/frame buffers, `am_hal_cachectrl_dcache_clean (&pRange)` should be called. Then before any CPU reads to texture/frame buffers, `am_hal_cachectrl_dcache_invalidate (&pRange, false)` should be called.

3.6.4.7 Recommended Cache Maintenance for SDIO Applications

The SDIO module is used in DMA mode by default in the HAL. For DMA mode, if DMA data buffers are in a cacheable region, then cache maintenance in application code is needed. If data buffers are located in TCM or in a non-cacheable SSRAM region, then cache operations are not needed.

If cache clean/invalidate is being done by address, then the buffer needs to be 32-byte aligned. The TX data buffer must be cleaned after it has been prepared by the CPU and before calling sync or async write APIs in DMA mode. The RX data buffer must be invalidated after DMA read completes and before other AXI managers access the buffer.

For SDIO async DMA read operations, it is recommended to invalidate the RX data buffer in the callback function. Refer to the below code or to the `emmc_raw_block_read_write` example for more details.

```

void am_hal_card_event_test_cb(am_hal_host_evt_t *pEvt)
{
    am_hal_card_host_t *pHost = (am_hal_card_host_t *)pEvt->pCtx;
+   am_hal_cachectrl_range_t sRange;
+   uint32_t ui32BufLen = 0;

    if (AM_HAL_EVT_XFER_COMPLETE == pEvt->eType &&
        pHost->AsyncCmdData.dir == AM_HAL_DATA_DIR_READ)
    {
        bAsyncReadIsDone = true;
        am_util_stdio_printf("Last Read Xfered block %d\n", pEvt->ui32BlkCnt);
+
+         //
+         // Flush and invalidate cache, if ssram buffer > 64K, clean all is efficient
+         //
+         if ( (uint32_t)(pHost->AsyncCmdData.pui8Buf) >= SSRAM_BASEADDR )
+         {
+             ui32BufLen = pEvt->ui32BlkCnt*512;
+             if ( ui32BufLen > DCACHE_SIZE )
+             {
+                 am_hal_cachectrl_dcache_invalidate(NULL, true);
+             }
+             else
+             {
+                 sRange.ui32StartAddr = (uint32_t)(pHost->AsyncCmdData.pui8Buf);
+                 sRange.ui32Size = ui32BufLen;
+                 am_hal_cachectrl_dcache_invalidate(&sRange, false);
+             }
+         }
+     }
}

```

3.6.4.8 Recommended Cache Maintenance for Security/CRC Engine

If data buffers are located in TCM or in a non-cacheable SSRAM region, cache operations are not needed. Clean D-Cache within the range of the CRC data buffer before calling `am_hal_crc32` APIs, if it is located in cacheable regions like SSRAM or PSRAM.

Please refer to the D-Cache Clean & Invalidate example in “D-Cache Clean and Invalidate Example Code” on page 34.

3.6.4.9 Recommended Cache Maintenance for USB Applications

The USB HAL defaults to PIO mode for which cache maintenance is not required. However, cache maintenance handling needs to be in place to ensure cache coherency when both of the following conditions are true:

- USB HAL is configured to DMA mode via `am_hal_usb_set_xfer_mode( )` API.
- USB Endpoint buffers are allocated from a cacheable memory region.

Example code to configure USB HAL to DMA modes can be found in DCD porting for TinyUSB, in function `dcd_init()`:

```
#if defined (AM_CFG_USB_DMA_MODE_0)
    am_(pUSBHandle, AM_HAL_USB_OUT_DMA_MODE_0);
    am_(pUSBHandle, AM_HAL_USB_IN_DMA_MODE_0);
#elif defined (AM_CFG_USB_DMA_MODE_1)
    am_(pUSBHandle, AM_HAL_USB_OUT_DMA_MODE_1);
    am_(pUSBHandle, AM_HAL_USB_IN_DMA_MODE_1);
#endif
```

For a USB IN transaction, where the CPU is assigning a buffer to be transmitted to a USB Host, D-Cache of the buffer should be cleaned before the `am_hal_usb_ep_xfer()` API is used to initiate the transfer. This ensures that the latest data in cache is cleaned into the memory module from which DMA accesses the data.

For a USB OUT transaction, where the CPU is assigning a buffer to hold upcoming data received from a USB Host, D-Cache of the buffer should be cleaned and invalidated before `am_hal_usb_ep_xfer()` API is used to initiate the transfer. This ensures that new data moved in by DMA is not incorrectly overwritten by stale data from the cache.

An example of the cache handling discussed above is available in DCD porting for TinyUSB, in function `dcd_edpt_xfer()`.

4. Power Management and Optimization

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

4.1 Power Management

Apollo5 family SoCs feature both a hardware Power Control Block and a dedicated software SPOT Manager to achieve best performance while drawing lowest power. Transitions between power modes are largely directed by software and the workload/task requirements, while the actual power mode implementation is achieved by configuring the Power Control Block module.

4.1.1 Power Control Block

The Power Control Block provides control and status for the power state of all the power domains and voltage regulators in the SoC. Software can control these blocks via power control registers within this block.

The Power Control Block controls the power sequence to power up or power down a particular peripheral device and memory power domain. Status of each of these can be monitored in the respective power control status register.

The power controller is also the primary control block for the SIMO Buck converter as well as the LDO regulator when the SIMO Buck is disabled. Similarly, event notification is supported to provide the appropriate handshake to software as needed as well as status register indicators.

This block handles all power sequencing during initial power on and all power mode transitions.

Please refer to this module's registers in the applicable SoC's register set, which is included in the AmbiqSuite SDK.

4.1.2 SPOT Manager

The SPOT Manager is a software module that works in lock-step with SPOT technology in the Apollo5 family of SoCs, along with chip-specific power trim information, to ensure that the SoC is operating with optimal performance for power consumption in various conditions. It behaves like a state machine where various stimuli can trigger a state machine update, which further triggers updates on specific registers that affect the performance and voltage level of the SoC.

The SPOT Manager state machine is controlled by various stimuli in the system:

1. Device Peripheral Status (DEVPWRSTATUS and AUDSSPWRSTATUS) - certain peripheral status updates, such as peripheral power on and peripheral power off, warrant an update to the spot manager state machine.
2. Operating mode of CPU and GPU
 - CPU: LP mode, HP Mode, Normal Sleep or Deep Sleep: Transitions between CPU modes warrant an update to the SPOT Manager state machine
 - GPU: Off, LP mode or HP Mode: Transitions between GPU modes warrant an update to the SPOT Manager state machine
3. Temperature: When the temperature crosses a certain threshold level, an update to the SPOT Manager power state machine should be invoked to guarantee optimal device performance.

It is highly recommended that the user use the SPOTMGR HAL API to enable and disable peripherals, change the CPU and GPU operating modes, and update temperature levels if needed, as the HAL API will handle the SPOT Manager power state update for you. By using the SPOTMGR HAL API, the SPOT

Manager power state update is abstracted from the application, making the application less prone to errors introduced by the update.

For device peripheral enabling and disabling, the `am_hal_pwrctrl_periph_enable()`/`am_hal_pwrctrl_periph_disable()` API is used. The API handles the SPOT Manager power state updates. Note that for the crypto module, the `am_hal_pwrctrl_control()` API should be called instead as there are additional procedures required to properly power up or power down the crypto module. This API will invoke the `am_hal_pwrctrl_peripheral_enable()` / `am_hal_pwrctrl_peripheral_disable()` within it.

For selecting CPU LP/HP operating modes, the `am_hal_pwrctrl_mcu_mode_select()` API is used. The API handles SPOT Manager power state updates.

For CPU sleep entry, the `am_hal_sysctrl_sleep()` API is used to allow the CPU to enter various sleep mode. The API handles SPOT Manager power state updates.

For GPU operating mode enabling and disabling, the `am_hal_pwrctrl_periph_enable()` API is used, which also warrants a SPOT Manager power state update. The `am_hal_pwrctrl_gpu_mode_select()` further controls the performance mode of the GPU, but this does not update the SPOT Manager power state. However it does have an affect on the TON value, which controls the SIMOBUCK switching frequency and timing.

For temperature updates, the `am_hal_pwrctrl_temp_update()` API is used. The API will handle SPOT Manager power state updates.

4.2 Power Optimization

The AmbiqSuite HAL manages the SoC power consumption to always operate in the most optimal conditions, using a number of internal power states. Transitioning from one state to another may require some sequences which are not instantaneous. For some of these transitions, the HAL implements an internal state machine to let normal operations continue to work while the power management transition is happening in the background. To manage this internal state machine, the HAL uses an internal timer, as defined by `AM_HAL_INTERNAL_TIMER_NUM_A`.

Optionally, `AM_HAL_SPOTMANAGER_USE_TIMER_OR_BLOCKING_DELAY` can be used to override the default implementation and instead use a blocking delay. Applications are expected to always keep this interrupt enabled and, at a high priority, ensure the interrupt service latency is kept to a minimum. If guaranteeing low latency for this interrupt is not possible, `AM_HAL_SPOTMANAGER_TIMER_DOUBLEBOOST` should be disabled.

4.2.1 Low Power Initialization Sequence Requirements

To achieve the best power performance and guarantee correct device operation, applications should follow the below sequence for initialization:

1. `am_hal_pwrctrl_low_power_init()`: This API initializes the SPOT Manager and MCU, and configures the power block for low power init.
2. `am_hal_pwrctrl_control(AM_HAL_PWRCTRL_CONTROL_SIMOBUCK_INIT)`: This API enables the SIMOBUCK.
3. `am_hal_pwrctrl_temp_update()`: This API sets the default temperature for SPOT Manager.

A typical call sequence would be as follows:

```
//
// Initialize for low power in the power control block
//
am_hal_pwrctrl_low_power_init();

//
// Enable the I-Cache and D-Cache
//
am_hal_cachectl_icache_enable();
am_hal_cachectl_dcache_enable(true);

#if AM_BSP_ENABLE_SIMOBUCK
//
// Enable SIMOBUCK for this board
//
am_hal_pwrctrl_control(AM_HAL_PWRCTRL_CONTROL_SIMOBUCK_INIT, NULL);
#endif
```

```

#if AM_BSP_SET_ROOM_TEMPS
    //
    // Set default temperature for spotmgr to room temperature
    //
    am_hal_pwrctrl_temp_thresh_t dummy;
    am_hal_pwrctrl_temp_update(25.0f, &dummy);
#endif

```

4.2.2 Multi-Image Transition Requirements

When transitioning between applications, the new application must ensure the system is back in a known state for Spot Manager to ensure stability across operating conditions and peripheral usage. That context can be lost, and a scenario could exist where trim adjustments are applied twice which could result in failures.

For multi-image situations, the application must turn off all peripherals, get back to LP mode (if needed), and call `am_hal_pwrctrl_settings_restore()` (shown below) before exiting one application to transition to another.

```

//
// Power Settings must be restored before switching application images.
//
// This function does not disable peripheral power. If the user has
// enabled any peripherals, these must be shut down separately.
//
am_hal_pwrctrl_settings_restore();

```

This function verifies that it is safe to restore power-related settings by ensuring the MCU is not in high-performance mode and that critical peripherals (GFX, SDIO0, SDIO1) are powered off. The function returns `AM_HAL_STATUS_FAIL` immediately if any check indicates an unsafe state or if any peripheral-status check fails. If all checks pass, it calls `am_hal_spotmgr_default_reset()` to restore SPOT manager defaults and returns that function's result (typically `AM_HAL_STATUS_SUCCESS`).

The call to `am_hal_spotmgr_default_reset()` resets SPOT Manager to its default state by invoking the `g_sSpotMgr.pfnSpotMgrDefaultRst()` callback when that function pointer is not NULL, returning the callback's status. If the callback pointer is NULL, it performs no action and returns `AM_HAL_STATUS_SUCCESS`.

The referenced callback `g_sSpotMgr.pfnSpotMgrDefaultRst` is set to one of the board/part-specific reset handlers. The `am_hal_spotmgr_pcmx_x_default_reset` function applies the saved SPOT Manager power-state settings to hardware restoring LDO trims and the memory LDO configuration from `g_sSpotMgr.INFO1regs` and returns an error if the INFO register indicates invalid data.

4.3 Temperature Compensation (TempCo)

Applications need to have integrated active periodic temperature monitoring for deep sleep regulation for higher temperature operation and for optimal active power consumption. A HAL function, `am_hal_pwrctrl_temp_update()`, exists for this purpose and should be used with the following guidelines:

- Call this API at initialization based on the first temperature reading.
 - This API returns a structure indicating thresholds for when application should make this call again.
- Monitor temperature periodically and call the API when the supplied thresholds are exceeded in either direction.

The application may use Apollo5 ADC-based temperature sensing or have other means to do temperature monitoring. The sampling frequency can be optimized based on usage, as long as it is fast enough to detect a temperature difference of 10°C.

An example application is provided with the SDK using on-chip ADC (`\boards\apollo510b_evb\examples\power\tempco`)"

5. Cryptographic Accelerator

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

Also, please refer to other related and useful security documents in the Ambiq Content Portal:

- Apollo5 Family Security Users Guide
- Apollo5 Family Secure Update Users Guide
- Apollo5 Family OEM Provisioning Tools Users Guide

5.1 Introduction

The Apollo5 family of SoCs provide comprehensive hardware acceleration for data integrity, device identification and cryptographic operations through three independent hardware modules:

- SECURITY register interface
- PUF (Physically Unclonable Function) module
- ARM CryptoCell-312 co-processor

Each module serves distinct security purposes with minimal CPU overhead. The SECURITY register interface provides hardware-accelerated CRC32 calculation for data integrity verification and manages security keys for OTP region locking. The PUF module leverages inherent silicon manufacturing variations to provide device-unique identifiers and cryptographically secure entropy. The CryptoCell-312 is a full-featured cryptographic co-processor offering AES encryption, SHA hashing, RSA public key operations, and true random number generation.

5.2 Functional Description

5.2.1 *SECURITY Register Interface*

CRC32 Calculation

The SECURITY peripheral provides hardware-accelerated CRC32 computation using the standard polynomial (0x04C11DB7) as used in Ethernet, ZIP, and PNG. The engine operates on word-aligned data and automatically processes the specified byte length with a default seed of 0xFFFFFFFF. The hardware implementation significantly outperforms software-based CRC calculations for data integrity verification.

Memory Boundary Handling

The CRC engine may encounter errors when data crosses certain memory boundaries, including 4 kB boundaries within TCM or transitions between major memory regions (DTCM to SSRAM0, SSRAM0 to SSRAM1, SSRAM1 to SSRAM2). The HAL function `am_hal_crc()` automatically detects and splits transfers at these boundaries, chaining the CRC calculations to produce correct results.

Security Key Management

The SECURITY peripheral manages 128-bit keys for controlling access to OTP memory regions. Applications set keys using the LOCKCTRL register to specify lock type (customer OTP program/read, internal OTP access) and KEY0-KEY3 registers for the 128-bit key value. The LOCKSTAT register indicates the current lock/unlock state. These mechanisms protect sensitive OTP data from unauthorized access.

5.2.2 *CryptoCell-312 Hardware Accelerator*

Architecture Overview

The CryptoCell-312 is an independent coprocessor with dedicated processing engines for AES, SHA, RSA, and TRNG operations. It features a multi-stage pipeline design that allows overlapping of data fetch, processing, and result write-back for maximum throughput. Integrated DMA controllers handle data movement between system memory and crypto engines, reducing CPU overhead for large block operations.

AES Operations

Supports AES-128, AES-192, and AES-256 with multiple modes: ECB (Electronic Codebook), CBC (Cipher Block Chaining), CTR (Counter), and GCM (Galois/Counter Mode). GCM mode provides authenticated encryption, combining confidentiality and integrity in a single operation. Hardware key expansion and dedicated round processing engines provide significant performance advantages over software implementations.

SHA Hash Operations

SHA-256 hashing with full hardware message scheduling optimization. The engine supports streaming mode for processing data larger than available memory by maintaining hash context in intermediate state registers. HMAC (Hash-based Message Authentication Code) combines SHA hashing with a secret key for authenticated message digests. Multi-block operations preserve state between invocations for incremental hashing.

RSA Public Key Operations

Supports RSA operations with key sizes up to 4096 bits. The modular arithmetic unit implements Montgomery multiplication for efficient computation of large-number modular exponentiation. Primarily used for firmware signature verification during secure boot, supporting PKCS#1 v1.5 and PSS padding schemes. RSA operations require significantly longer processing time than symmetric operations due to computational complexity.

True Random Number Generation (TRNG)

The CryptoCell TRNG uses physical noise sources to generate cryptographically secure random numbers. Continuous health monitoring with statistical tests (repetition count test, adaptive proportion test) ensures output quality. Failed samples are automatically rejected. The Deterministic Random Bit Generator (DRBG) implements CTR_DRBG mode using AES-256 to produce deterministic sequences from TRNG seeds.

Lifecycle Control State (LCS)

The CryptoCell LCSREG register indicates device security state: CM (Chip Manufacturing), DM (Device Manufacturing), Secure, or RMA (Return Merchandise Authorization). The LCS determines available cryptographic features and secure boot requirements. Applications should query LCS using `am_hal_security_get_info()` to verify feature availability before attempting restricted operations.

5.2.3 PUF Hardware Accelerator

Physical Unclonable Function Principle

The PUF exploits inherent manufacturing variations in silicon that occur during fabrication. These microscopic variations create a unique physical fingerprint for each device that cannot be cloned or predicted. Unlike traditional identifiers stored in non-volatile memory, the PUF-derived identifier is an intrinsic property of the silicon, making it highly resistant to physical attacks and tampering.

Static UID (Unique Identifier)

The PUF provides a 1024-bit device-unique identifier through 32 read-only registers (PUFUID[0] through PUFUID[31]), each containing 32 bits. This static UID remains constant across power cycles, temperature variations, and device lifetime. Applications use the UID for device authentication, firmware binding, license management, anti-counterfeiting, and as seed material for device-specific key derivation. The large UID size (1024 bits) provides sufficient entropy for deriving multiple independent cryptographic keys.

Entropy Generation

The PUF module generates cryptographically secure random numbers by reading the OTP RNG register. Each read produces a 32-bit entropy word derived from PUF measurements. The entropy quality is suitable for cryptographic applications including key generation, initialization vectors, nonces, and seeding software-based random number generators. When entropy is temporarily unavailable, the register returns a sentinel value (0xDEADDEAD) to signal retry required.

Power Management

PUF operations require the OTP power domain to be active. The HAL automatically manages power state transitions, tracking whether OTP was already powered before PUF initialization (`am_hal_puf_entropy_init`). If the HAL enabled OTP, de-initialization (`am_hal_puf_entropy_deinit`) powers it down. If OTP was already active from another peripheral, the PUF HAL preserves the power state to avoid disrupting other operations.

5.2.4 ERRATUM NOTICE - ERR014: DMA to TCM Restrictions

Description

There are three known restrictions on DMAing to TCM. Refer to the Apollo510 Errata document for complete details.

Constraints

- DMA transfers to TCM must not cross 4 kB boundaries. Software must manually split DMA transfers at 4 kB boundaries when targeting TCM.
- TCM must be powered on before any DMA access is attempted.
- DMA access to TCM is not permitted when the CPU is in deep sleep mode with memory retained.

5.2.5 APDMA Issues

Description

CryptoCell-312 DMA Memory Boundary Issues. Refer to the latest version of the Apollo510 Errata document for complete details.

Constraints

- Maximum single-buffer operation size is limited by memory region size (typically 512 kB for DTCM, 1 MB-3 MB for SSRAM regions).
- Performance may be reduced when operating on data in TCM due to alignment requirements.
- Buffer allocation strategies must account for 4 kB alignment requirements in TCM, potentially increasing memory overhead.
- Ensure all input and output buffers for CryptoCell operations are allocated within a single memory region (do not span DTCM/SSRAM boundaries).
- For large data transfers, split operations into smaller chunks that do not cross problematic memory boundaries.
- The SECURITY CRC engine includes automatic boundary detection and splitting in the `am_hal_crc()` function.

Note: This erratum applies specifically to the CryptoCell-312 DMA controller and the SECURITY peripheral CRC engine. The PUF module is not affected as it uses direct register reads without DMA transfers.

5.3 Design Considerations

5.3.1 *SECURITY Usage Considerations*

CRC Performance Trade-offs

Hardware CRC32 significantly outperforms software implementations for data blocks larger than 256 bytes. For very small blocks, software CRC may be more efficient due to setup overhead. The `am_hal_crc()` function with boundary protection adds minimal overhead while ensuring correctness across all memory regions. For performance-critical applications processing data entirely within a single 4kB-aligned region, `am_hal_crc32()` provides slightly faster execution.

Memory Placement for CRC Operations

When possible, allocate buffers requiring CRC verification in contiguous memory regions that do not span major boundaries. SSRAM0, SSRAM1, and SSRAM2 each provide large contiguous regions suitable for most applications. For data in TCM, ensure 4KB alignment or accept the automatic splitting overhead of `am_hal_crc()`.

OTP Lock Management

Security keys for OTP locking should be derived from device-unique sources (e.g., PUF UID) or provisioned during secure manufacturing. Keys must never be stored in plain text in code or non-volatile memory. Once OTP regions are locked, they cannot be unlocked without the correct 128-bit key. Implement robust key backup and recovery procedures before deploying lock mechanisms in production.

5.3.2 *PUF Usage Considerations*

UID Stability and Reliability

The PUF Static UID exhibits excellent stability across temperature ranges (-40°C to +85°C) and device lifetime. However, the UID should be read during system initialization and cached in RAM rather than repeatedly accessing OTP. Frequent OTP reads increase power consumption and are unnecessary given the UID static nature.

Device-Specific Key Derivation

Using the PUF UID for key derivation provides device-binding for cryptographic operations. Apply a Key Derivation Function (KDF) such as HKDF-SHA256 with application-specific context to derive keys from the UID. This ensures derived keys are unique per device and per application. Never use the raw UID directly as an encryption key; always apply a KDF to ensure proper key distribution.

Entropy Quality and Throughput

PUF entropy generation has limited throughput compared to the CryptoCell TRNG. For applications requiring large volumes of random data, use PUF entropy to seed a software DRBG implementation (e.g., CTR_DRBG with AES) rather than reading OTP for every random byte. The 0xDEADDEAD sentinel value indicates temporary entropy shortage; the HAL retries automatically, but applications should implement additional error handling for production deployments.

Power Consumption

OTP power consumption is measurable in battery-operated applications. Minimize OTP power-on duration by batching all PUF operations (UID reads, entropy collection) into a single init/deinit session. Read the complete 1024-bit UID once at boot and cache it in RAM for subsequent use.

5.3.3 *CryptoCell-312 Usage Considerations*

Algorithm Selection

Choose cryptographic algorithms based on security requirements and performance constraints. AES-256 provides stronger security than AES-128 but requires additional processing cycles. SHA-256 is recommended for most applications; SHA-512 is not available in hardware. For authenticated encryption, prefer AES-GCM over separate AES-CBC + HMAC-SHA256 as it provides both operations in a single pass.

Key Storage and Management

The CryptoCell provides secure key storage that is not readable via software once loaded. Keys loaded into hardware key registers cannot be extracted, providing protection against software-based attacks. For session keys, load directly from secure sources. For long-term keys, derive from PUF UID or provision from OTP during secure boot. Never store keys in plaintext in MRAM or SRAM.

DMA Buffer Management

Due to ERR046, care must be taken to manage buffer placement for CryptoCell operations. Allocate crypto buffers within single, contiguous memory regions. Use 16-byte alignment for optimal DMA performance. For TCM-resident buffers, use 4 kB alignment. Applications must manually verify buffer alignment and boundaries before initiating DMA transfers, as the CryptoCell HAL does not include automatic splitting like the SECURITY CRC function.

Power Consumption

Cryptographic operations consume significant power. Verify CryptoCell power state using the PWRCTRL_DEVPWRSTATUS_PWRSTCRYPTO bit before operations. For battery-powered applications, schedule intensive crypto work during periods of external power availability. Gate the CRYPTO peripheral clock when idle to conserve power.

Secure Boot Integration

The CryptoCell is integral to secure boot flows. RSA signature verification validates firmware authenticity. Query LCS state to verify available security features before attempting restricted operations. In production devices (LCS = Secure), debug features may be restricted and secure boot becomes mandatory.

5.4 Programming Guidelines

5.4.1 SECURITY Register Programming

CRC32 Calculation (Basic)

1. Initialize seed: `SECURITY_RESULT = 0xFFFFFFFF` (first operation only).
2. Set source address: `SECURITY_SRCADDR = (uint32_t)data_address`.
3. Set length (must be 4-byte aligned): `SECURITY_LEN = size_in_bytes`.
4. Configure function: `SECURITY_CTRL_FUNCTION = SECURITY_CTRL_FUNCTION_CRC32`.
5. Start operation: `SECURITY_CTRL_ENABLE = 1`.
6. Poll for completion: `while (SECURITY_CTRL_ENABLE != 0) { }`.
7. Check for errors: `if (SECURITY_CTRL_CRCERROR) {handle_error()}`.
8. Read result: `crc32_value = SECURITY_RESULT`.

For chained CRC calculations across multiple buffers, preserve the RESULT value between operations instead of reinitializing to 0xFFFFFFFF. This allows incremental CRC computation.

CRC32 with Boundary Protection (Recommended)

Use the HAL function `am_hal_crc()` for automatic boundary detection and splitting.

Setting OTP Lock Keys

1. Define 128-bit key: `am_hal_security_128bkey_t key = {keys = {key0, key1, key2, key3}}`.
2. Specify lock type: `am_hal_security_locktype_t type = AM_HAL_SECURITY_LOCKTYPE_CUSTOTP_PROG`.
3. Set key: `status = am_hal_security_set_key(type, &key)`.
4. Verify lock status: `bool unlocked: am_hal_security_get_lock_status(type, &unlocked)`.

5.4.2 4.2 PUF Programming Guidelines

Reading Static UID

1. Initialize OTP power: `status = am_hal_puf_entropy_init()`.
2. Allocate buffer: `uint32_t uid_buffer[32]; // 1024 bits`.
3. Read complete UID: `status = am_hal_puf_get_uid(uid_buffer, 0, 32)`.
4. Cache UID in RAM for subsequent use.
5. Deinitialize OTP: `am_hal_puf_entropy_deinit()`.

To read only a subset of the UID, specify a different start word (0-31) and number of words (1-32). For example, to read the first 4 words (128 bits): `am_hal_puf_get_uid(buffer, 0, 4)`.

Collecting Entropy

1. Initialize OTP power: `am_hal_puf_entropy_init()`.
2. Allocate buffer: `uint8_t entropy[32]; // Request 32 bytes`.
3. Collect entropy: `status = am_hal_puf_get_entropy(entropy, 32)`.
4. Check return status: `if (status != AM_HAL_STATUS_SUCCESS) {retry or handle error}`.
5. Use entropy for cryptographic operations (keys, IVs, nonces).
6. De-initialize OTP: `am_hal_puf_entropy_deinit()`.

The HAL function automatically retries up to 5 times when 0xDEADDEAD is encountered. Production applications should implement additional retry logic and error reporting.

Deriving Device-Specific Keys

Example approach for key derivation from PUF UID using a standard Key Derivation Function (KDF) such as HKDF-SHA256.

5.4.3 *CryptoCell-312 Programming Guidelines*

Initialization and Status Check

Before using the CryptoCell, verify power and idle status. Check `PWRCTRL_DEVPWRSTATUS_PWRSTCRYPTO` to confirm power state and `CRYPTO_HOSTCCISIDLE_HOSTCCISIDLE` to verify the co-processor is ready for operations.

For CryptoCell operation, please view the readme in the Ambiqsuite SDK in third-party/crypto as well as view the code in the Github repo: <https://github.com/ARM-software/cryptocell-312-runtime>.

Buffer Alignment for DMA Operations

To avoid ERR046 issues:

- Allocate buffers within a single memory region (DTCM, SSRAM0, SSRAM1, or SSRAM2).
- Verify buffer start and end addresses do not cross memory region boundaries.
- For TCM buffers, ensure 4 kB alignment: $\text{address} \& 0xFFF = 0$.
- For optimal DMA performance, use 16-byte alignment: $\text{address} \& 0xF = 0$.
- If buffers would cross boundaries, split operations manually into multiple transfers.

5.5 Register Summary

5.5.1 SECURITY Register Interface

The SECURITY peripheral provides the following registers:

- CTRL: Control register. Contains ENABLE (start operation), FUNCTION (select CRC32) and CRCERR (error flag) bit fields.
- SRCADDR: Source address register for CRC32 operations. Specifies the memory location of data to be processed. Must be word-aligned.
- LEN: Length register. Specifies the number of bytes to process. Must be a multiple of 4 (word-aligned).
- RESULT: Result register. Contains the computed CRC32 value. Also serves as the seed value for chained operations (write 0xFFFFFFFF for first operation).
- LOCKCTRL: Lock control register. Specifies the lock type for OTP key operations (customer OTP program/read, internal OTP access).
- LOCKSTAT: Lock status register. Indicates the current enable/disable state of OTP regions. Customer-applicable bits are the following:
 - Bit 0: Customer OTP Program (CUSTOTP_PROG)
 - Bit 1: Customer OTP Read (CUSTOTP_READ)
- KEY0: Key register 0. Bits [31:0] of the 128-bit security key.
- KEY1: Key register 1. Bits [63:32] of the 128-bit security key.
- KEY2: Key register 2. Bits [95:64] of the 128-bit security key.
- KEY3: Key register 3. Bits [127:96] of the 128-bit security key.

5.5.2 5.2 PUF Register Interface

The PUF module is accessed through OTP memory-mapped registers:

- **OTP_PUFUID[0..31]**: Static UID register array. Provides 1024-bit device-unique identifier (32 words x 32 bits). Each register is read-only. Base address: OTP base + 0x300. Individual registers at offset 0x300 + (n x 4) where n = 0 to 31. The UID is stable across temperature and device lifetime.
- **OTP_RNG**: Random Number Generator register. Each read returns a 32-bit entropy word derived from PUF measurements. Returns 0xDEADDEAD when entropy is temporarily unavailable (retry required). Read-only.

OTP power domain must be enabled before accessing PUF registers. Use `am_hal_pwrctrl_periph_enable` (AM_HAL_PWRCTRL_PERIPH_OTP) or the PUF HAL initialization function `am_hal_puf_entropy_init()`.

5.5.3 CryptoCell-312 Register Interface

The CryptoCell-312 provides an extensive register set organized into functional groups:

Status and Control Registers

- **CRYPTO_HOSTCCISIDLE**: CryptoCell idle status register. HOSTCCISIDLE bit indicates when the coprocessor is ready for new operations (1 = idle, 0 = busy).
- **CRYPTO_LCSREG**: Lifecycle Control State register. LCSREG field indicates device security state: 0 = CM (Chip Manufacturing), 1 = DM (Device Manufacturing), 5 = Secure, 7 = RMA. Determines available cryptographic features.

Configuration Registers

Control operation modes, algorithm selection (AES, SHA, RSA, TRNG), key sizes (128/192/256-bit), cipher modes (ECB, CBC, CTR, GCM), and enable/disable states. Includes mode control, configuration, and parameter registers specific to each crypto engine.

Data Input/Output Registers

Virtual registers (typically designated R0-R15) for loading input data, encryption keys, initialization vectors (IVs), and reading output results. Memory-mapped for efficient DMA access with separate address spaces for different data types.

Key Storage Registers

Secure storage for cryptographic keys. Keys loaded into these registers are not readable via software, providing protection against key extraction attacks. Supports multiple key slots for AES (128/192/256-bit) and RSA (up to 4096-bit modulus).

Interrupt Registers

Provide operation completion status, error conditions, and interrupt management. Enable asynchronous notification for long-running operations (RSA, large block AES/SHA) to reduce CPU polling overhead.

Detailed register field definitions and bit layouts are provided in the individual Apollo5 family member register set and the ARM CryptoCell-312 Technical Reference Manual.

6. Storage

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

6.1 MRAM

6.1.1 MRAM Recovery

Please see the *Apollo5 MRAM Recovery Users Guide*, document number A-SOCAP5-UGGA01EN, at the Ambiq Content Portal - <https://contentportal.ambiq.com> for information about MRAM recovery.

6.2 One-Time Programmable Memory

6.2.1 General Flow

Figure 1 shows the general flow of operation for the one-time programmable memory (OTP).

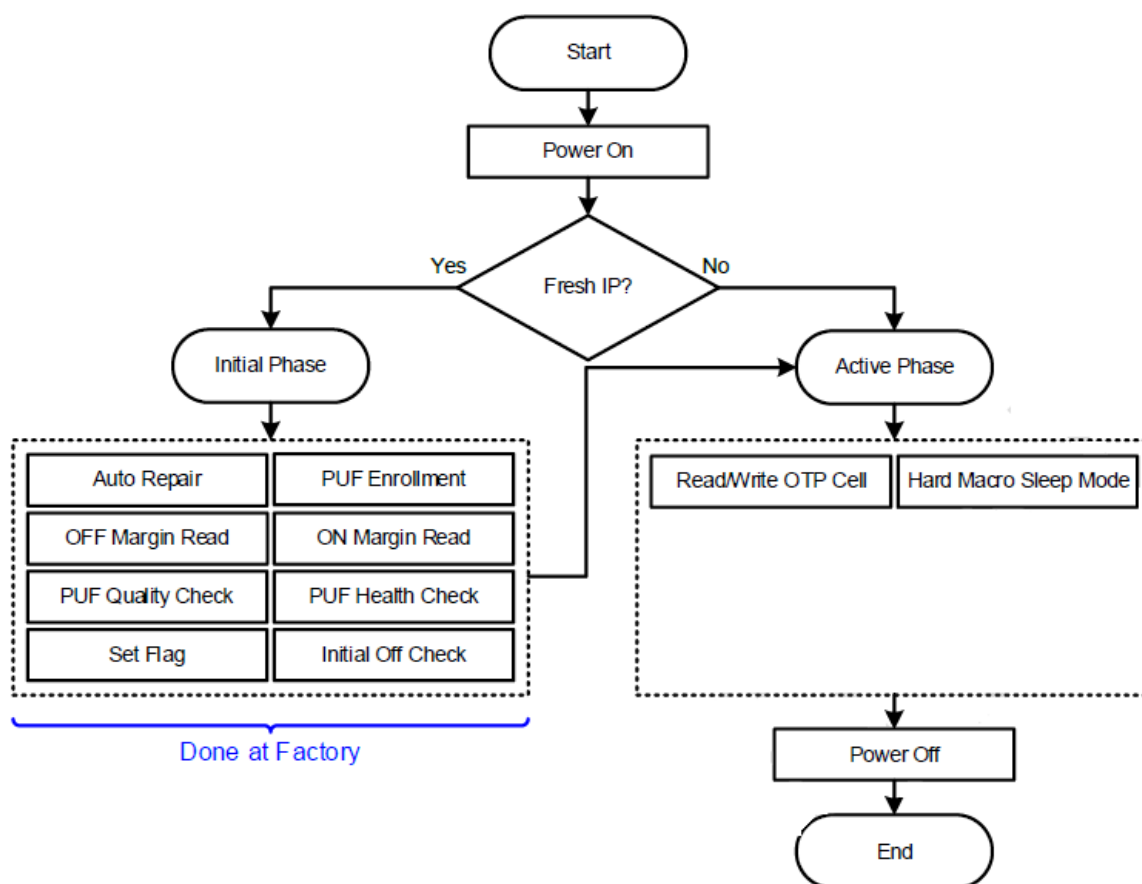


Figure 1. OTP General Flow

The general flow is described in the following sequence of steps:

1. After RST_N transitions from low to high (after the OTP PUFrt is powered on), PUFrt will start its power-on initialization sequence.
2. “Fresh IP” is defined as OTP that has never gone through the Initial Phase.
3. After power on initialization, a fresh IP will need to go through the “Initial Phase” before it is ready to use.
4. The Initial Phase uses four operations to ensure the correct functioning of the OTP:
 - A. Filter out “initial fail” bits in an automatic repair function (not available in this case).
 - B. Initialize PUF to its inborn random values (PUF Enrollment).
 - C. Check health of OTP/PUF cells (PUF Quality Check, PUF Health Check, Initial OFF Check and OFF/ON Margin Read).
 - D. Set options (flag).
5. For most cases, this Initial Phase is done at the factory during the normal testing flow – that is, the user normally does not need to use the Initial Phase operations during regular operations (Active Phase).
6. After a PUFrt has gone through the Initial Phase, it should only be operated in the Active Phase, even after subsequent power on/off cycling.

6.2.2 Power-On Flow

Figure 2 shows the power-on flow of operation for the OTP.

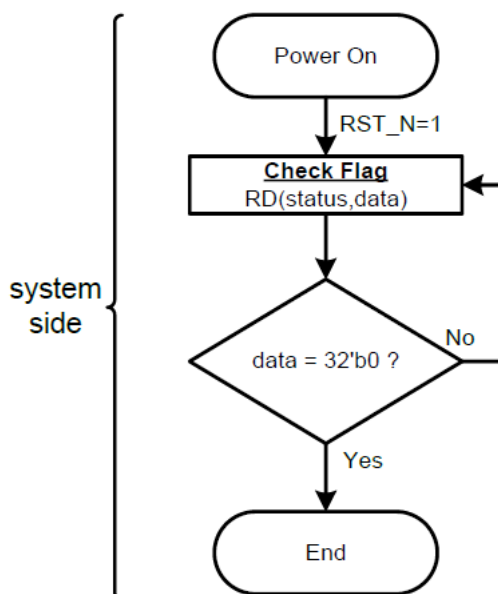


Figure 2. OTP Power-On Flow

The power-on flow is described in the following sequence of steps:

1. After RST_N transitions from low to high, PUFrt will start its power-on initialization sequence.
2. During initialization, PUFrt will check the information block, load the auto-load bits and dump 1kb of PUF code into a dedicated register (for Data Shuffling).
3. Users should wait for the busy flag to go low (thus indicating the end of the power-on sequence) before issuing any commands.

6.2.3 Power-Off Flow

Figure 3 shows the power-off sequence which should be followed to properly power down the OTP block.

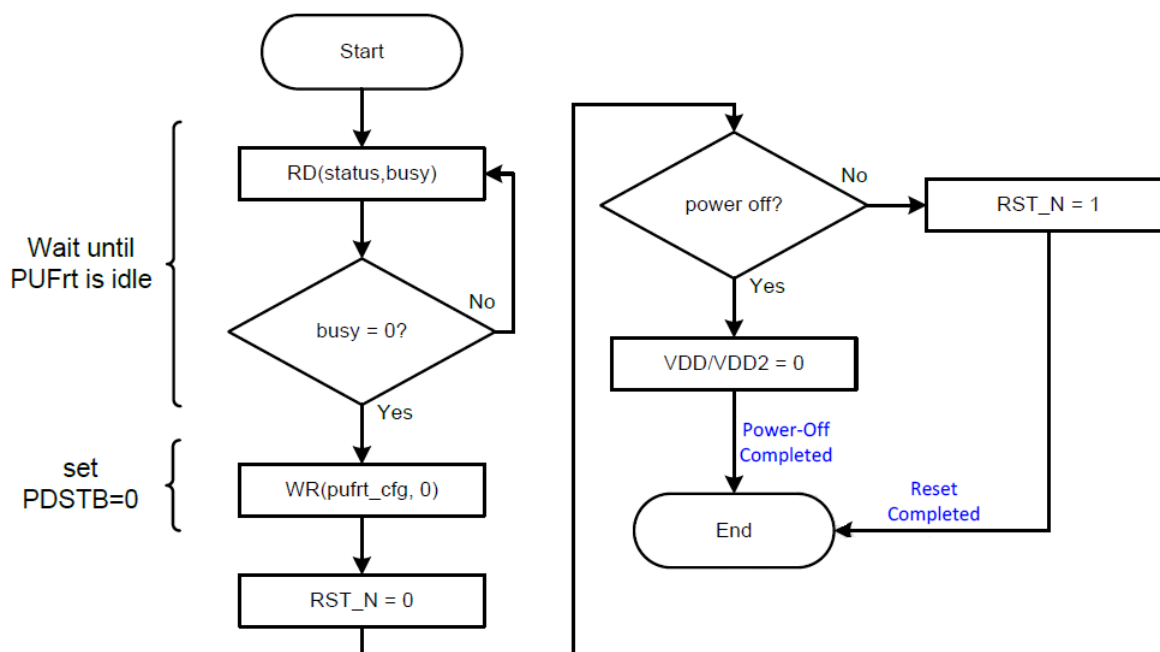


Figure 3. OTP Power-Off Flow

When not in use (at initial boot or when not accessed by crypto operations), the OTP block should be powered off to save power.

7. Clock Generation and Control

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

7.1 HFRC Auto-Adjustment

In some applications it is important that the High Frequency RC Oscillator (HFRC) frequency be more accurate than the $\pm 2\%$ variation typically seen, particularly in cases where the temperature may vary widely. An example of this is in cases where the MCU communicates with another device via the UART. The frequency matching with the other device in the connection is an important factor in the reliability of the connection. In order to support a highly accurate HFRC, a function called Auto-adjustment is provided. Also note that the System PLL is an option for clocking the UART and is used for higher frequency UART communication.

It should be noted that Auto-adjustment is dependent on an accurate clock source such as the crystal. The min/max variation of the HFRC frequency with and without adjustment is different. During auto-adjustment, the number of counted 48 MHz cycles which occur in one 32.768 kHz crystal oscillator (XT) cycle is compared to a target value. If the count is different from the target, an HFRC tuning value is modified to change the HFRC frequency. The target count is held in the REG_CLKGEN_HFADJ_HFXTADJ field, and is initially set to a default value of 0x5B8 (48 MHz / 32.768 kHz).

Auto-adjustment works by periodically enabling the HFRC and the XT, counting the number of cycles for a divided-down HFRC clock which occur in a single XT cycle, subtracting that value from HFXTADJ and adding the resulting difference to the actual HFRC tuning value which is stored in an internal register field. Auto-adjustment is enabled in the REG_CLKGEN_HFADJ Register by loading the repeat frequency value into the HFADJCK field and then setting the HFADJEN bit.

Auto-adjustment cycles will occur continuously if both the XT and the HFRC are currently requested as reference clocks by the modules. If either oscillator is disabled, Auto-adjustment cycles will then occur at intervals determined by the REG_CLKGEN_HFADJ_HFADJCK field, as shown in the register description. Shorter repeat intervals will result in more accurate HFRC frequencies, especially if the temperature is changing rapidly, but will result in higher power consumption. When an Auto-adjustment cycle occurs, if the XT was disabled it is enabled and then a delay occurs to allow the XT to stabilize. This delay is defined by the REG_CLKGEN_HFADJ_HFWARMUP field as defined in the register documentation. Once the HFRC is stable, the HFRC is enabled and several Auto-adjustments occur, each of which results in a refinement of the tuning value. Once those adjustments are complete, the HFRC and XT are powered down unless they are in use by other functions.

7.2 XTALHS Crystal Calibration on the Apollo510 SoC

This section describes how to determine whether the external XTALHS crystal frequency needs to be trimmed, and provides the procedure for performing the trim with the AmbiqSuite SDK version 5.1.0 and later. Refer to examples in the SDK for guidance on how to do the calibration.

This section is applicable to the Apollo510 SoC.

7.2.1 Determining whether the XTALHS trimming is needed

The external XTALHS crystal's accuracy requirement is 40 ppm including initial tolerance, temperature drift, and crystal aging. The crystal must be specified for operation with 6 pF load capacitance and must have an ESR of no more than 60 Ω . The initial tolerance of the crystal is the main frequency tolerance in ppm specified for the crystal and specifies how much the crystal may vary from specified frequency when new, at room temp, and with specified load capacitance applied. The temperature drift will be determined by the crystal specs and is often specified as temperature stability in ppm. The crystal aging should also be specified by the crystal manufacturer in ppm per year.

PLL Reference Clock Frequency Tolerance Requirements:

There is no single universal frequency tolerance requirement for a PLL reference clock. The required frequency tolerance budget is determined entirely by the target application and system-level constraints. If the crystal tolerance as described above meets the system requirement, then a fixed per-design trim may be used. This trim value needs to be determined by measuring the initial frequency on a large population of boards (because the crystals have ± 20 ppm tolerance) and then calculating the trim for the median frequency. This will center the 20 ppm initial tolerance of this particular board design (crystal+board) at the correct frequency.

If a crystal with a capacitive load specification other than the recommended 6 pF is used, then the initial frequency of the crystal will be pulled due to being operated with incorrect load capacitance. In this case, crystal trimming will be required. Only use of 6 pF specified crystals is officially supported with these devices.

- Initial tolerance: ± 40 ppm
- Temperature stability: ± 50 ppm

Recommended Crystal Specification for PLL/USB Applications:

The USB 2.0 HS total frequency tolerance budget is ± 500 ppm (USB 2.0 HS). It includes:

- Initial frequency tolerance
- Temperature drift
- Aging
- Board layout capacitance loading

Recommended Crystal Specification for Audio Applications:

Audio clocks are source-synchronous: the transmitter forwards a clock alongside the data, so absolute frequency tolerance is less critical than jitter.

- Clock tolerance: ± 100 ppm
- Design priority: low jitter and good phase noise rather than tightening absolute frequency tolerance.

General Considerations:

If the design does not meet these qualifications after trimming, then the following steps should be followed:

- Select a crystal with tighter initial tolerance, temperature stability, or aging.
- Perform per-unit calibration trim to reduce the initial tolerance to a much lower frequency tolerance which is limited only by production measurement accuracy used to measure the frequency and the SoC's XTALHS trim step size.

NOTE

In all cases, the user should test and confirm that the initial median frequency and frequency tolerance is correct. The XTALHS frequency median should be less than x ppm where "x" is the initial tolerance in ppm used in the calculation to confirm total variance from the XTALHS frequency is less than 40 ppm.

7.2.2 Setting XTALHSCAPTRIM and XTALHSCAP2TRIM register values

Trimming the external XTALHS is done by tuning two register fields, XTALHSCAPTRIM and XTALHSCAP2TRIM, in the XTALHSTRIMS register in the MCUCTRL register set. Values used for tuning the two trim fields are applied in parallel and therefore are additive to the total capacitance. The code example for how to set these values is shown in the next sub-section.

When the external XTALHS has been checked and it is determined that a change in the trim values is needed, it is best to start with the defaults in the code. The algorithm is defined in the next section.

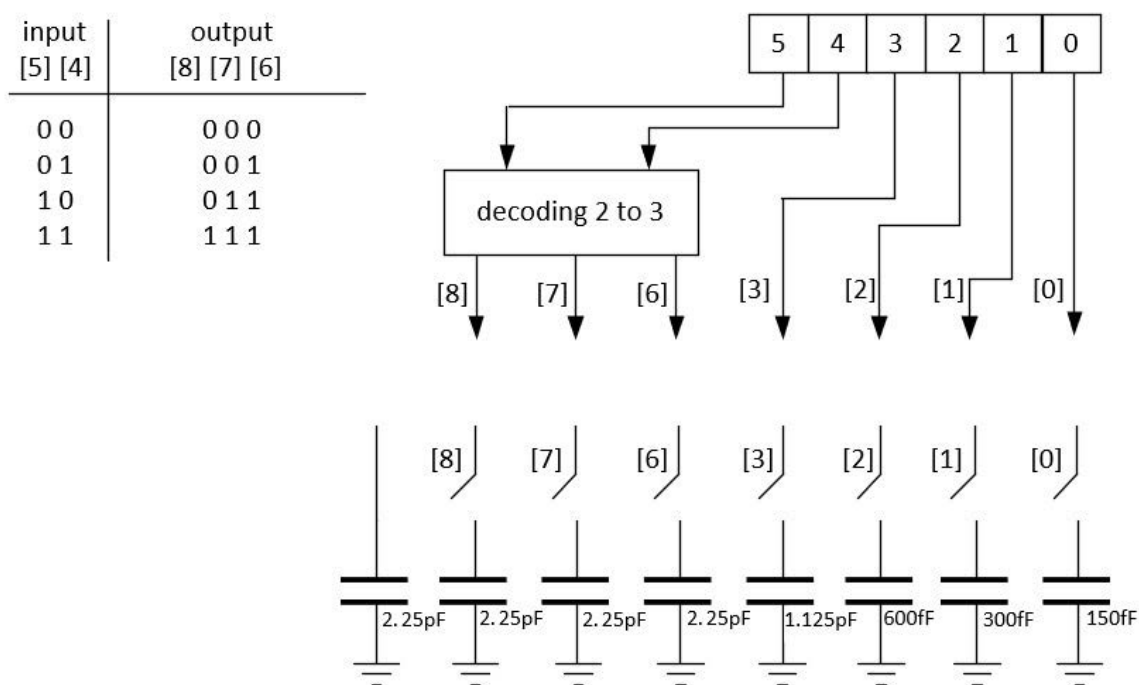


Figure 4. Trim Value Determination

Table 3: XTALHSCAPTRIM [9:6] Value and Corresponding Shunt Cap Value

XTALHSCAPTRIM [9:6]	Shunt Cap Value
0001	0.1 pF
0010	0.2 pF
0100	0.4 pF
1000	0.8 pF

Note that simply increasing the MCUCTRL_XTALHSTRIMS register field values as a whole is going to result in the frequency varying in time with a slight increase in offset.

7.2.3 Trimming the External XTALHS frequency

A summary of the calibration trim is as follows:

1. Download the Apollo510 example project external_hfxtal_trim from the Apollo510 SDK.
2. Change the XTALHS trim:
 - A. Send the command to list the active ports.
 - python hfxtal_trim_control.py --list-ports
 - B. Send the command to set the XTALHS trim.
 - python hfxtal_trim_control.py --port COMx --trim 0x12C
 - C. Start the XTALHS output on pin 46.
 - python hfxtal_trim_control.py --port COMx --kickstart-enable
 - D. Measure the HFXTAL clock signal on GPIO pin 46.
 - E. If desired frequency is not obtained, shut off output by sending the XTALHS disable output command.
 - python hfxtal_trim_control.py --port COMx --hfxtal-disable
 - F. Repeat steps A-E until desired frequency is obtained. This trim setting will not be retained through reset.

Trimming the XTALHS crystal frequency in the SDK is as follows:

```
//*****
//
//!! @brief Set the HFXTAL crystal trim value.
//!
//! @param ui16TrimValue - Crystal trim value (valid range: 0x000 to 0x3FF)
//!                               Default value is 0x12C
//!
//! This function configures the crystal trim capacitance values for the
//! HFXTAL external crystal oscillator. The trim value consists of:
//! - XTALHSCAP2TRIM [5:0] - xtalhs_cap2_trim
//! - XTALHSCAPTRIM  [9:6] - xtalhs_cap_trim
//!
//! @return Standard HAL status code
//!
//! @note Refer to KBA "Apollo510 SoC External HFXTAL Crystal Calibration" for
//! detailed calibration procedures.
//
//*****
static uint32_t
hfxtal_trim_set(uint16_t ui16TrimValue)
{
    if (ui16TrimValue > 0x3FF)
    {
        return AM_HAL_STATUS_OUT_OF_RANGE;
    }

    //
    // XTALHSCAP2TRIM : 6 bits [5:0] - xtalhs_cap2_trim
    // XTALHSCAPTRIM  : 4 bits [9:6] - xtalhs_cap_trim
    //
    g_ui32xtalhscap2trim = (ui16TrimValue & MCUCTRL_XTALHSTRIMS_XTALHSCAP2TRIM_Msk);
    g_ui32xtalhscaptrim  = (ui16TrimValue & MCUCTRL_XTALHSTRIMS_XTALHSCAPTRIM_Msk) >>
MCUCTRL_XTALHSTRIMS_XTALHSCAPTRIM_Pos;
    return AM_HAL_STATUS_SUCCESS;
}
```

7.3 BLE Crystal Calibration on the Apollo510B SoC

The Apollo510B SoC's radio uses an external 48 MHz high-frequency crystal as the reference for the RF synthesizer. Due to component tolerances, board layout, and temperature, the actual oscillation frequency can deviate from the nominal value. This deviation causes carrier frequency error when the device transmits. The Bluetooth Core Specification limits this error (e.g., ± 50 kHz for some tests), and tighter calibration (e.g., ± 5 kHz in this procedure) improves margin and interoperability.

Calibration corrects this by storing a trim value that adjusts the crystal load or tuning so that the effective reference frequency matches the expected value. The radio supports a 6-bit crystal trim value (0x00–0x3F) that is written into the radio's non-volatile configuration memory. This value is applied to the HF crystal oscillator circuitry (mapped to bits 12:7 of the internal register REG_RF_XO_SEQ_DIG).

This section describes how to determine whether the external 48 MHz BLE XTAL crystal frequency needs to be trimmed, and provides the procedure for performing the HFX TAL trim calibration for Apollo510B with the AmbiqSuite SDK version 5.1.0 and later.

The trim algorithm is as follows:

1. Use a BLE compliance tester to monitor the frequency offset in transmission mode.
2. Hardware setup:
 - A. Update Apollo510B BLE Controller firmware to latest version.
 - B. Connect test device to PC.
 - C. Connect test device to the tester.
3. Test method and steps:
 - A. Send the UART command 01 1E 20 03 13 25 02 from the computer to put the DUT into transmission mode.
 - B. Send the '10101010' pattern type to observe carrier frequency offset (which is why the last octet of TX test command is '02').
 - C. Use the tester in manual test mode to check the modulation characteristics at 2440 MHz and observe the Frequency Offset indicator.
 - D. Modify the trim value using the command 01 F1 FC 01 xx (which will trigger calling the API `am_devices_em9305_crystal_trim_set` to update trim value).
 - E. Resend the command 01 1E 20 03 00 25 00 and observe changes in the Frequency Offset indicator.
 - F. Repeat the trim modification via the command 01 F1 FC 01 xx until the Frequency Offset indicator falls within ± 5 kHz.
 - G. The xx in the final try (step 3D) is the optimal trim value.

7.4 RF XTAL Frequency Trimming on the Apollo510 Lite and Apollo330 Plus SoCs

This section is applicable to the Apollo510B Lite, Apollo510D Lite, Apollo330B Plus and Apollo330M Plus SoCs.

Below is the trim procedure for the 48 MHz RF_XTAL_HS crystal.

Prerequisites:

- Spectrum Analyzer or Ellisys Bluetooth Analyzer
- Latest CM4 BTDM/BLE firmware
- Run uart_hci_bridge application

Steps:

1. Call HCI Reset (01 03 0C 00).
 - Wait for Command Complete event.
2. Set XTAL trimming (01 88 FC 01 XX).
 - XX = trim value (1 byte, hex)
3. Enter DTM TX mode (01 1E 20 03 05 10 00).
 - Start TX test and measure frequency offset with spectrum analyzer/Bluetooth analyzer.
4. End DTM (01 1F 20 00).
 - Stop TX test (DTM end).
5. Tune loop.
 - Change XX and repeat steps 2 → 4 until frequency offset is within your target range.
6. Update the final trim value in the project (g_sRfxtalCfg.ui32XOFreqTrim).

Figure 5 shows the resulting frequency offset for sequential crystal trim settings following the above steps. All responses (RX) indicate successful execution of commands (TX).

- When XTAL trim value = 0x60, Freq Offset = +109.4 kHz
- When XTAL trim value = 0xB0, Freq Offset = -31.25 kHz
- When XTAL trim value = 0xA0, Freq Offset = 0 Hz to -15.63 kHz
- XTAL Trim Value step size = 43.8 fF
- XTAL Trim Value Range = 0x00 to 0xFF → 0.00 fF to 11.169 pF

```
[TX] - 01 03 0C 00
[RX] - 04 0E 04 05 03 0C 00
[TX] - 01 88 FC 01 60
[RX] - 04 0E 04 05 88 FC 00
[TX] - 01 1E 20 03 05 10 00
[RX] - 04 0E 04 05 1E 20 00
[TX] - 01 1F 20 00
[RX] - 04 0E 06 05 1F 20 00 00 00
[TX] - 01 88 FC 01 B0
[RX] - 04 0E 04 05 88 FC 00
[TX] - 01 1E 20 03 05 10 00
[RX] - 04 0E 04 05 1E 20 00
[TX] - 01 1F 20 00
[RX] - 04 0E 06 05 1F 20 00 00 00
[TX] - 01 88 FC 01 A0
[RX] - 04 0E 04 05 88 FC 00
[TX] - 01 1E 20 03 05 10 00
[RX] - 04 0E 04 05 1E 20 00
```

Figure 5. Frequency Offset Based on XTAL Trim Value

Figure 6 indicates that the closest center frequency offset was reached with the specified XTAL trim value (in this case 0xA0), and how much deviation might be seen if using different XTAL trim values.

Item	Status	Payload	Initial Center Frequency Off...	Time	Time ...	Communication
Test Traffic (x 49'105, 59.997 s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	+109.4 kHz	94.252 016 125	60.000 9...	Central: Unknown BD_ADDR <
Test Traffic (x 37'746, 60.000 s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	+109.4 kHz	154.252 917 750	60.000 3...	Central: Unknown BD_ADDR <
Test Traffic (x 95'440, 60.000 s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	-15.63 kHz	214.253 281 500	60.000 3...	Central: Unknown BD_ADDR <
Test Traffic (x 95'427, 60.000 s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	-31.25 kHz	274.253 640 125	60.000 3...	Central: Unknown BD_ADDR <
Test Traffic (x 95'422, 60.000 s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	-31.25 kHz	334.253 999 000	60.000 3...	Central: Unknown BD_ADDR <
Test Traffic (x 95'542, 60.000 s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	-31.25 kHz	394.254 358 375	60.000 3...	Central: Unknown BD_ADDR <
Test Traffic (x 78'045, 49.028 s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	-31.25 kHz	454.254 718 250	60.000 3...	Central: Unknown BD_ADDR <
Test Traffic (x 93'989, 1m 00s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	-15.63 kHz	522.717 352 875	68.462 6...	Central: Unknown BD_ADDR <
Test Traffic (x 93'942, 1m 00s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	-15.63 kHz	582.717 883 750	60.000 5...	Central: Unknown BD_ADDR <
Test Traffic (x 94'007, 1m 00s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	+0 Hz	642.718 414 875	60.000 5...	Central: Unknown BD_ADDR <
Test Traffic (x 93'986, 1m 00s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	-15.63 kHz	702.718 946 500	60.000 5...	Central: Unknown BD_ADDR <
Test Traffic (x 94'040, 1m 00s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	+0 Hz	762.719 478 250	60.000 5...	Central: Unknown BD_ADDR <
Test Traffic (x 78'438, 50.007 s)	OK	16 bytes (FF C1 FB E8 4C 90 72 8B E7 B3 51 89 63 AB 23 23)	-15.63 kHz	822.720 010 250	60.000 5...	Central: Unknown BD_ADDR <

Figure 6. Trim Results

Based on the Frequency Offset Reading, update project trim value in boards\{evb type}\bsp\am_bsp.c → g_sRfxTALCfg.

Note: It is recommended to perform the test under RF Chamber for accurate measurement.

```

#ifdef AM_BSP_USE_CUSTOM_RFXTAL_CONFIG
//*****
//
// Custom RFXTAL Configuration
//
//*****
const am_hal_clkmgr_rfxTAL_config_t g_sRfxTALCfg =
{
    .ui32XOFreqTrim = 0xA0,
    .ui32XOConfig = 0x00047c00,
    .ui32RadioTune = 0x00054150
};
#endif // AM_BSP_USE_CUSTOM_RFXTAL_CONFIG

```

Figure 7. RFXTAL Configuration

7.5 Generating Required RTC Clock

The Real Time Clock (RTC) module requires a 100 Hz clock which is provided by the Clock Generator. This clock may come either from the LFRC or the XT oscillators, as determined by the REG_CLKGEN_OCTRL_OSEL bit. Since 100 Hz is not a simple power of two division of either of these oscillators, special functions are used to create it.

For both the XT and LFRC, the 512 Hz signal is divided by 5.12 ($128/25$), which produces a 100 Hz RTC clock with a maximum jitter of less than 4 ms.

7.6 External 32.768 kHz Clock Input on XO

An external clock can be used in lieu of using a 32.768 kHz crystal as the low frequency clock reference. An external signal of approximately 32 kHz with an amplitude of $V_{DDH}/2$ to 2.2 V should be connected between XO and ground. The duty cycle of the input signal, which can be a minimum of 10% and a maximum of 90%, determines the duty cycle of the resulting internal clock. Therefore, if a 50% duty cycle for the internal clock is intended, then the external clock's duty cycle should be 50%.

When using an external clock, fields in the MCUCTRL_XTALCTRL register need to be set as follows:

- XTALPDNB= 'b0
- XTALCOMPPDNB = 'b0
- XTALCOMPBYPASS= 'b1

The HAL function for this is:

```
am_hal_mcuctrl_control(AM_HAL_MCUCTRL_CONTROL_EXTCLK32K_ENABLE, true);
```

If using the Clock Manager, you must do the following as the Clock Manager does not by default monitor the XTAL_LS: Recompile the HAL with the switch AM_HAL_CLKMGR_MANAGE_XTAL_LS and call:

```
am_hal_clkmgr_clock_request(AM_HAL_CLKMGR_CLK_ID_XTAL_LS, AM_HAL_CLKMGR_USER_ID_RESV0).
```

```
//
//! Capability switch on whether Clock Manager should manage XTAL_LS as well.
//! Clock manager will still continue to support clock request and release API
//! for XTAL_LS regardless of whether this switch is defined. However, it will
//! only attempt to enable/disable according to clock request if this switch is
//! defined. This switch is disabled by default for now.
//! Note: calls to am_hal_mcuctrl_control() with following eControl value must
//!       be removed from BSP/Application before enabling this switch, as clock
//!       manager will take over the management of XTAL_LS:
//!       - AM_HAL_MCUCTRL_CONTROL_EXTCLK32K_ENABLE
//!       - AM_HAL_MCUCTRL_CONTROL_EXTCLK32K_DISABLE
//
// #define AM_HAL_CLKMGR_MANAGE_XTAL_LS
```

7.7 Clock Switching

Considering the two levels of clock muxing on PDM, I2S, AUDADC and USB, it is recommended that switching to/from the higher fidelity/low power clocks (at the last level mux) should be done as follows:

- Select the static clock input ('0') from the primary clock select.
- Select alternate PLL or HFRC48 clock from last level mux.

When switching back to primary clock mux, a similar flow is recommended:

- Select the static clock input ('0') from primary clock select.
- Select primary clock at last level mux.
- Power off PLL (optional).

7.8 System PLL Control

7.8.1 PLL Control

All System PLL controls are controlled by software via MCUCTRL registers: PLLCTL0, PLLDIV0, PLLDIV1 and PLLSTAT. There are two VCO output ranges, 60 MHz to 240 MHz and 240 MHz to 960 MHz. The range needs to be selected via the PLLCTL0_VCOSELECT field.

PLL Power ON/OFF

- PLL power rails may be turned on or off in either order:
 - PLLCTL0_SYSPLLVDDFPDNB¹ = 0x1 to enable the System PLL's VDDF power
 - PLLCTL0_SYSPLLVDDHPDNB = 0x1 to enable the System PLL's VDDH power.
- When powering on the PLL, power must be stable by waiting at least 1 μ s after the SYSPLL rails above are powered before the PLL is enabled, which is done by setting the PLL enable register bit:
 - PLLCTL0_SYSPLLPDB = 0x1.

PLL VCO Frequency Adjust

To adjust the PLL Frequency, follow this sequence:

- Disable the PLL: PLLCTL0_SYSPLLPDB = 0x0.
- Change the FBDIV ratio: PLLDIV1_FBDIV = a value of 10 to 960 in integer mode or 10 to 96 in fractional mode. PLL frequency change may also involve changing the PLLDIV0 register if fractional mode is used.
- Enable the PLL: PLLCTL0_SYSPLLPDB = 0x1.
- After the PLL is enable, the HAL will poll the PLLSTAT register to make sure the PLL is locked before marking it safe to use.

7.8.2 PLL Mode Control

There are two modes for setting the PLL - Integer PLL and Fractional PLL. The reference clock when operating in Integer PLL mode has a range of 1 MHz to 800 MHz, and when operating in Fractional PLL mode, the reference clock range is 10 MHz to 800 MHz. The valid range for the PLL feedback clock, reference clock / REFDIV, must be in the range of 1 MHz - 52 MHz in Integer mode and 10 MHz to 52 MHz in Fractional mode. Each mode uses different settings to derive an output frequency, FOUT, based on the reference frequency, FREF.

For Integer PLL mode, FOUT is derived as:

$$FOUT = FREF \times FBDIV / (REFDIV \times POSTDIV1 \times POSTDIV2)$$

For Fractional PLL mode, FOUT is derived as:

$$FOUT = FREF \times (FBDIV + \text{FRAC} / 2^{24}) / (REFDIV \times POSTDIV1 \times POSTDIV2)$$

where:

- FREF = frequency of clock source as specified in MCUCTRL_PLLCTL0_FREFSEL. Clock selections are either the XTALHS clock generated from the internal XTALHS block or the EXTREFCLK input via GPIO15².
- FBDIV = value of MCUCTRL_PLLDIV1_FBDIV field. Valid range for divider in Integer PLL mode is 10 to 960, and for Fractional PLL Mode it is 10 to 96.
- REFDIV = value of MCUCTRL_PLLDIV1_REFDIV field. Valid range of divide values is 1 to 63. Setting of 0 or 1 equates to 1 for calculation purposes.

1. For the Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs, the SYSPLLVDDFPDNB and SYSPLLVDDH-PDNB register fields are without the B suffix. However, the polarity and default settings of these fields are the same.
2. On Apollo510B only EXTREFCLK selection is available since no XTALHS is available on Apollo510B.

- POSTDIV1 = value of MCUCTRL_PLLDIV1_POSTDIV1 field. Valid range of divide values is 1 to 7. Setting of 0 or 1 equates to 1 for calculation purposes.
- POSTDIV2 = value of MCUCTRL_PLLDIV1_POSTDIV2 field. Valid range of divide values is 1 to 7. Setting of 0 or 1 equates to 1 for calculation purposes.
- FRAC = value of MCUCTRL_PLLDIV0_FRAC field. Valid range of fractional divide values is 1 to 16,777,215.

NOTE

In either PLL mode, the value of the first post divider, POSTDIV1, should be maximized before setting/enabling the second post divider, POSTDIV2.

NOTE

Ensure FBDIV is programmed to a valid setting anytime the PLL is enabled to ensure that the PLL will lock.

7.8.3 Considerations When Programming the PLL Feedback Divider

For integer and fractional divide, only static programming is supported. For static-programmed integer and fractional divide values, no special re-timing logic is required. The integer and fractional (if used) divide values should be programmed while the PLL is disabled.

7.9 Clock/Reset Module in Apollo510 Lite and Apollo330 Plus SoCs

The clock generation subsystem in the Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs enables clock selection and configuration for several modules in the Clock/Reset Module (CRM) via the CRM register set. Peripherals supported by the CRM are I2S, PDM, ADC, Display Controller, Display PHY, and UART.

For all CRM registers, the *RSTN field at register bit 2 (if it exists), will reset the logic of the specific peripheral's CRM when asserted (set to 0). The *CRMSTN field at bit 3 will reset the peripheral's CRM logic (not its registers), as well as the logic clocked by the CRM's clock output. A peripheral's *CLKDIV field is zero-based such that 0=DIV1, 1=DIV2, etc. An APB* CRM register is the register clock domain for each peripheral. These are named APB* as the register clocks (PCLK) come from the APBDMA block.

Software can reset the CRM for a specific peripheral at any time by clearing its RSTN bit in the peripheral's CRM register), which will immediately reset the peripheral. In addition, software can perform clock switching and change clock division, including integer divide options, by doing the following:

1. Read the peripheral's CLKSEL register and make sure current and new clock sources are active.
2. Clear the peripheral's CLKEN bit in its CRM register and poll for its read-only CLKACTIVE bit to go 0.
3. After modification of the peripheral's clock select and clock divider settings, software can set the CLKEN bit. No other clock or reset dependency is needed for clock changing.

For peripherals with multiple clock domains, there will be one CRM per clock domain. The general rule of thumb is to setup the APB*CRM (register clock domain) first followed by the functional clocks' CRM. To shut off the module, the APB*CRM will be shut off last.

The hardware handles the manager/subordinate relationship between PWRCTRL and CRM, where the PWRCTRL is the manager control for clock/reset. So, the sequence does not matter. If CRM is enabled, but power is off, there will be no clock downstream.

When turning off the clock, software must wait for the read-only CLKACTIVE bit to become 0. When turning on the clock, it depends on the peripheral as to whether polling for CLKACTIVE to become 1 is needed. Some peripherals request the clock dynamically based on its operation. For such clock domains, polling for CLKACTIVE to become 1 might cause deadlock if the peripheral is turning off the clock itself.

Generally, for APB*CRM, software needs to make sure CLKACTIVE is 1 before accessing the registers of the peripheral, as without a clock, there could be an issue with the access.

RSTN and CLKEN can be set/cleared together in a single register write.

7.10 HP2 Mode and PLL Handling in Apollo510 Lite and Apollo330 Plus SoCs

In Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs, the PLL is used to provide the High Performance 2 Mode (HP2) at approximately 250 MHz while the High Performance 1 Mode (HP1) uses HFRC_192MHz to provide a 192 MHz clock to the Cortex-M55 CPU. Pwrctrl exercises limited control over the PLL with respect to the CPU. Software controls all other usage of the PLL.

The PLL must be configured by software before it can be used. If the application uses audio, the software should program the PLLVCO frequency to 245.76 MHz. If audio is not used, the software may set the PLLVCO frequency to 250 MHz (or lower). Note that if the application needs some other frequency for whatever reasons, the software can still set PLLVCO for that frequency.

When the CPU software requests HP2 mode, hardware (Pwrctrl) will insure that the PLL domain is on and the PLL is set to be the clock source after PLL lock. When the CPU goes into deep sleep and if the CPU is the only user, then the PLL can be gated. If there are other functions using the PLL, it stays on. PLL usage outside of the CPU is indicated by setting the MCUCTRL_PLLCTL0_SYSPLLPDB bit to enabled.

A bit in MCUCTRL_PLLCTL0_SYSPLLWAITHP2 field determines whether the CPU will be held in reset for HP2 mode until the PLL is locked. When the CPU wakes up (and was in HP2 mode), or if a transition to HP2 mode is requested, then if SYSPLLWAITHP2 = 0, the following is the case:

- Pwrctrl will set up the appropriate power rails for (either) HP mode.
- Pwrctrl will clock the CPU at HFRC_192MHz while the PLL is being brought up, and then switch to HP2 when the PLL locks.

This setting ensures that the CPU will execute as soon as possible, minimizing wakeup and transition latencies. However, it means that the CPU will execute at 192 MHz for a number of μ S before switching to approximately 250 MHz.

Otherwise, if (SYSPLLWAITHP2 = 1), then the following will occur:

- Pwrctrl will set up the appropriate power rails for the HP mode.
- Pwrctrl will prevent the CPU from executing while the PLL is being brought up. When the PLL locks, the CPU clock will switch to HP2 and the CPU will be allowed to execute.

This setting ensures that the CPU will only execute once the HP2 frequency is reached, but the CPU will be prevented from executing for an additional 15-30 μ S.

Note that Pwrctrl requesting the PLL does not impact the MCUCTRL_PLLCTL0_SYSPLLPDB bit. This bit is controlled only the software. The Pwrctrl request for PLL is “OR”ed with this bit to enable the PLL.

If the VCO frequency is set to something less than 192 MHz, then the software should not ask for HP2 if it has set VCO to less than 192 MHz. Otherwise, the hardware will switch to that frequency. The HAL PLL wrapper should keep track of the reference frequency and flag an error if the VCO is less than 192 MHz in case the user asks for HP2 mode.

Since the software is responsible for PLL configuration, any other divider frequency outputs are remain unchanged. And since the PLL configuration is in MCUCTRL, the configuration will persistent across PLL power downs. Note that when SYSPLLWAITHP2 = 0, it is possible that the software will request deep sleep before the transition to HP2 mode is complete. Software must check whether this HP2 transition is in progress and not request deep sleep until the transition to HP2 has completed.

7.11 Clock Manager

This section is intended to assist users of the Apollo510 or Apollo510B with porting the new Clock Manager functionality to their hardware platform. Following a brief overview of Clock Manager, specific guidance for the board support package (BSP) configuration is provided. Following that, specific recommendations for the **XTAL_HS**, **EXTREFCLK**, **SYSPLL (Apollo510 SoC and Apollo510B SoC only)** and/or **PLLVC0/PLLPOSTDIV (Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs only)** clock sources are provided. Finally, guidance for clock configuration is provided for the USB, UART, and audio interfaces.

Please consult the AmbiqSuite SDK's Doxygen documentation for the actual API definitions and code for using the Clock Manager.

7.11.1 Introduction to Clock Manager

Clock Manager is a layer in the HAL of the SDK that helps the user manage the clocks in the Apollo510 for the following purposes:

1. Automatically powers down the clocks whenever possible for power saving.
2. Ensure that any required clock will be available when it is requested by peripherals.
3. Checks for clock dependencies and makes sure any requirements for a requested clock are satisfied.
4. Handles proper clock enable and disable sequence, to ensure that the clocks are started and stopped correctly.

With the implementation of Clock Manager, the HAL requests a clock via the **am_hal_clkmgr_clock_request()** API when needed, and releases it via the **am_hal_clkmgr_clock_release()** API when it is no longer needed. Clock Manager maintains a scoreboard of clock usage and ensures that each is powered-up appropriately when it is in use, and powered-down whenever not needed. Hence, manual control for the clocks managed by Clock Manager is no longer required.

7.11.2 Clocks Managed by Clock Manager

The clocks managed by Clock Manager by default are:

- XTAL_HS (Not present on Apollo510B SoC)
- HFRC
- HFRC2 (Apollo510 SoC and Apollo510B SoC only)
- EXTREFCLK
- SYSPLL (Apollo510 SoC and Apollo510B SoC only)
- PLLVC0/PLLPOSTDIV (Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs only)

NOTE

Please consult the datasheet of the Apollo5 family device of interest for its available clock sources. Disregard any references in the sub-sections below to any clocks or clock sources that are unavailable on the device of interest.

7.11.3 Porting Guide for the BSP

There are 2 steps needed for porting the BSP for Clock Manager usage:

Step 1: Board Information

The purpose of this step is to make sure that Clock Manager has the correct information about the clock sources available on the board. Use the **am_hal_clkmgr_board_info_set()** API to set the clock sources.

Step 2: Clock Configuration

Specify the desired clock rates for HFRC, HFRC2 (Apollo510 SoC and Apollo510B SoC only), SYSPLL (Apollo510 SoC and Apollo510B SoC only) and/or PLLVCO/PLLPOSTDIV (Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs only) in the Clock Manager by using the **am_hal_clkmgr_clock_config()** API to set the configuration.

NOTE

In your application, if it is more advantageous for the configuration to be done in application code instead of the BSP, it is advisable to do the configuration as early in the application as possible after **am_bsp_low_power_init()** is executed.

Clock modes/rates supported are as shown in Table 4:

Table 4: Supported Clock Modes/Rates

Clock	Clock Configuration	Description
HFRC	1. AM_HAL_CLKMGR_HFRC_FREQ_FREE_RUN_APPROX_96MHZ	Free running 96 MHz HFRC clock
	2. AM_HAL_CLKMGR_HFRC_FREQ_ADJ_96MHZ	HFRC adjusted to 96 MHz with reference to XTAL_LS clock
HFRC2 (Apollo510 SoC and Apollo510B SoC only)	1. AM_HAL_CLKMGR_HFRC2_FREQ_FREE_RUN_APPROX_250MHZ	Free running 250 MHz HFRC2 clock
	2. AM_HAL_CLKMGR_HFRC2_FREQ_ADJ_250MHZ	HFRC2 adjusted to 250 MHz with reference to XTAL_HS/EXTREFCLK
	3. AM_HAL_CLKMGR_HFRC2_FREQ_ADJ_196P608MHZ	HFRC2 adjusted to 196.608 MHz with reference to XTAL_HS/EXTREFCLK
SYSPLL (Apollo510 SoC and Apollo510B SoC only)	A wide range of frequencies supported	Configure this clock's frequency according to the needs of peripherals using this clock.
PLLVCO / PLLPOSTDIV (Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs only)	A wide range of frequencies supported	Configure this clock's frequency according to the needs of peripherals using this clock.

A typical configuration flow would be as follows:

- Set the board information in the BSP as advised in step 1 above.
- Configure HFRC2 based on the board configuration where HFRC2's source clock would be XTAL_HS or EXTREFCLK (XTAL_HS would be prioritized over EXTREFCLK).
- The Clock Manager HAL will check board information for availability and frequency of XTAL_HS. If available, it will generate the configuration according to the frequency of XTAL_HS for HFRC2 ADJ.
- If XTAL_HS is unavailable, it will then check for EXTREFCLK availability and frequency and configure for this if it is available.
- If both are unavailable, the configuration will return failure.

IMPORTANT NOTES

- Configuration of clocks is not allowed when there is an active user of the clock unless switching only between the configuration pairs shown in Table 5.
- Developers should always ensure that the operation and stability of the application is not affected by the clock configuration change.

Clock pairs allowing live switching are as shown in Table 4:

Table 5: Clock Configuration Pairs Allowing Live Switching

Clock	Type	Configuration 1
HFRC	Configuration 1	AM_HAL_CLKMGR_HFRC_FREQ_FREE_RUN_APPROX_96MHZ
	Configuration 2	AM_HAL_CLKMGR_HFRC_FREQ_ADJ_96MHZ
HFRC2 (Apollo510 SoC and Apollo510B SoC only)	Configuration 1	AM_HAL_CLKMGR_HFRC2_FREQ_FREE_RUN_APPROX_250MHZ
	Configuration 2	AM_HAL_CLKMGR_HFRC2_FREQ_ADJ_250MHZ

7.11.3.1 Example for BSP Implementation

An example for BSP porting can be found in `am_bsp.c`, at `am_bsp_low_power_init()` in the `apollo510_evb` BSP included in the AmbiqSuite SDK.

Both Step 1: Board Information and Step 2: Clock Configuration in the above section “Porting Guide for the BSP” on page 74 are included in this example.

```
//
// Set board related info into clock manager
//
am_hal_clkmgr_board_info_t sClkmgrBoardInfo = {
    .sXtalHs.eXtalHsMode = AM_BSP_XTAL_HS_MODE,
    .sXtalHs.ui32XtalHsFreq = AM_BSP_XTAL_HS_FREQ_HZ,
    .sXtalLs.eXtalLsMode = AM_BSP_XTAL_LS_MODE,
    .sXtalLs.ui32XtalLsFreq = AM_BSP_XTAL_LS_FREQ_HZ,
    .ui32ExtRefClkFreq = AM_BSP_EXTREF_CLK_FREQ_HZ
};
am_hal_clkmgr_board_info_set(&sClkmgrBoardInfo);
// Default config for HFRC and HFRC2 as Free Run clock.
am_hal_clkmgr_clock_config(AM_HAL_CLKMGR_CLK_ID_HFRC,
    AM_HAL_CLKMGR_HFRC_FREQ_FREE_RUN_APPROX_96MHZ, NULL);
am_hal_clkmgr_clock_config(AM_HAL_CLKMGR_CLK_ID_HFRC2,
    AM_HAL_CLKMGR_HFRC2_FREQ_FREE_RUN_APPROX_250MHZ, NULL);
```

where the BSP configuration used above is defined in `am_bsp.h`:

```
#define AM_BSP_XTAL_HS_MODE      AM_HAL_CLKMGR_XTAL_HS_MODE_XTAL
#define AM_BSP_XTAL_HS_FREQ_HZ  (48000000U)
#define AM_BSP_XTAL_LS_MODE      AM_HAL_CLKMGR_XTAL_LS_MODE_XTAL
#define AM_BSP_XTAL_LS_FREQ_HZ  (32768U)
#define AM_BSP_EXTREF_CLK_FREQ_HZ (0U)
```

With the above code snippets, Clock Manager is configured to:

Table 6: Board Information

Clock Source	Configuration
XTAL_HS	48 MHz supplied by crystal oscillator connected to XO32M pin
XTAL_LS	32.768 kHz supplied by crystal oscillator connected to XO pin
EXTREFCLK	External reference clock is not available at GPIO15

Table 7: Clock Configuration

Clock Source	Configuration
HFRC	Free running 96 MHz HFRC is configured
HFRC2	Free running 250 MHz HFRC2 is configured
SYSPLL	System PLL is not configured

7.11.4 Guidelines for XTAL_HS

With implementation of Clock Manager, the starting and stopping of **XTAL_HS** is controlled by Clock Manager automatically when a peripheral's HAL requests or releases the clock. Ensure XTAL_HS information is correctly declared to Clock Manager as stipulated by Figure 7.11.3, step 1.

Note that the Apollo510B SoC does not have a XTAL_HS clock / crystal circuit. A 48 MHz crystal is used to clock the BLE module, and a divide-by-4 of that clock (BLE_CLK) is routed to the internal EXTREFCLK signal which is made available as a reference clock to other modules.

7.11.5 Guidelines for SYSPLL on Apollo510 and Apollo510B

The following are guidelines for **SYSPLL on Apollo510 and Apollo510B**:

1. Ensure at least one of these reference clock sources is available to the chip and is correctly declared to Clock Manager using API `am_hal_clkmgr_board_info_set()`:
 - A. XTAL_HS
 - B. EXTREFCLK
2. Ensure that the SYSPLL clock is configured as stipulated in the *Programming Guide for the BSP*, as SYSPLL needs to be explicitly configured before it can be started.
3. With the implementation of Clock Manager, the configuration for enabling and disabling of SYSPLL clock is done automatically when a peripheral's HAL requests or releases the clock.

Note that XTAL_HS is not available for Apollo510B, hence EXTREFCLK is the only reference clock source for Apollo510B.

7.11.6 Guidelines for PLLVCO and PLLPOSTDIV on Apollo510 Lite and Apollo330 Plus

The following are guidelines for configuring PLLVCO and PLLPOSTDIV on Apollo510 Lite and Apollo330 Plus. Due to PLLVCO and PLLPOSTDIV being interlinked, where PLLPOSTDIV is a 2-stage 1-to-7 divider per stage output of PLLVCO (MCUCTRL_PLLDIV1_POSTDIV1 and _POSTDIV2 with a total maximum divider of 49), there is a relationship that must be maintained.

1. If PLLPOSTDIV is being configured:
 - A. If PLLVCO is already configured:
 - i. PLLVCO is compatible: PLLPOSTDIV configuration is successful (if all other conditions are met).
 - ii. PLLVCO is not compatible: return error.
 - B. If PLLVCO is not configured:
 - i. Lowest power PLLVCO is automatically selected to generate the PLLPOSTDIV. In this case the PLLVCO output is not enabled.
2. If PLLVCO is being configured:
 - A. PLLPOSTDIV configuration will be cleared.

The sequence for the PLLVCO/PLLPOSTDIV configuration is as follows:

1. Ensure at least one of these reference clock sources is available to the chip and is correctly declared in Clock Manager correctly using API `am_hal_clkmgr_board_info_set()`:
 - A. XTAL_HS
 - B. EXTREFCLK
2. Ensure that the PLLVCO and PLLPOSTDIV clocks are configured in the sequence below as stipulated in “Programming Guide for the BSP”, as PLLVCO and PLLPOSTDIV need to be explicitly configured before PLLVCO can be started.
 - If both PLLVCO and PLLPOSTDIV are needed, **configure PLLVCO and then PLLPOSTDIV.**
 - If only PLLVCO is needed, configure PLLVCO only.
 - If only PLLPOSTDIV is needed, configure PLLPOSTDIV only.

If the system needs to switch configuration from PLLVCO only to PLLPOSTDIV, PLLVCO can be configured to a frequency of zero, which will invalidate the PLLVCO configuration and hence allow PLLPOSTDIV to be set directly. When both PLLVCO and PLLPOSTDIV are defined, make sure that PLLPOSTDIV can be derived from PLLVCO through the 2-layer 1-to-7 dividers.

Note that configuring the PLLVCO clock will clear the PLLPOSTDIV configuration. Always configure PLLVCO first if both clocks are needed.

3. With the implementation of Clock Manager, the configuration for enabling and disabling the PLLVCO and PLLPOSTDIV clocks is done automatically when a peripheral's HAL requests or releases the clock.

7.11.7 Guidelines for EXTREFCLK on Apollo510 and Apollo510B

For Apollo510, the configuration of EXTREFCLK on GPIO15 is handled by the Clock Manager in the AmbiqSuite HAL. The user needs to provide a correct clock source to GPIO15 for EXTREFCLK usage.

For Apollo510B, the configuration of EXTREFCLK (output on GPIO15 for diagnostic purposes only) is handled by the Clock Manager in the HAL. The EXTREFCLK clock is internally provided as a divide-by-4 clock of the 48 MHz BLE crystal connected to the BLE_XIN and BLE_XOUT pins. This whole sequence is

handled by the HAL and BSP. The user does not need to directly invoke any API or provide any physical clock source to GPIO15 in order to use this clock.

NOTE

This step is needed only if EXTREFCLK is used for the system.

7.11.8 Peripheral Clock Configuration Guidelines

The guidelines in this section merely reiterate the steps mentioned in “Porting Guide for the BSP” on page 74, “Guidelines for XTAL_HS” on page 77, “Guidelines for SYSPLL on Apollo510 and Apollo510B” on page 77, “Guidelines for PLLVCO and PLLPOSTDIV on Apollo510 Lite and Apollo330 Plus” on page 78 and “Guidelines for EXTREFCLK on Apollo510 and Apollo510B” on page 78, to provide an alternative perspective on the porting process.

7.11.8.1 USB Applications

Confirm that the clock source selected for USB PHY conforms to the guidance provided in Table 8, and that the selection defaults to SYSPLL (Apollo510 SoC and Apollo510B SoC only) and PLLVCO/PLLPOSTDIV (Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs only) for HS mode and HFRC for FS mode unless a specific clock source is selected by the application or USB stack via **am_hal_usb_set_phy_clk_source()** API.

To summarize the USB PHY clock configuration by the Clock Manager, the USB PHY clock can be selected as AM_HAL_USB_PHYCLKSRC_DEFAULT. But when this is selected, the API **am_hal_usb_auto_gen_clk_source()** is invoked, which will select the HS mode clock source based on a priority ranking of XTAL48 > XTAL24 > EXTREFCLK > PLL. The USB PHY clock will then use this clock source. The PLL clock is the default fall-back clock, but it is the lowest priority ranking.

Table 8: USB Application Clock Source Selection

Selection	Procedures
SYSPLL (Apollo510 and Apollo510B only)	- SoC-dependent: - Apollo510: Make sure that XTAL_HS/EXTREFCLK in Board Information is correctly defined as described in step 1 for “<i>Porting Guide for the BSP</i>” on page 74. - Apollo510B: Make sure that EXTREFCLK in Board Information is correctly defined as described in step 1 for “<i>Porting Guide for the BSP</i>” on page 74. - Make sure that SYSPLL clock is configured for 24 MHz as described in step 2 for “<i>Porting Guide for the BSP</i>” on page 74.
PLLPOSTDIV (Apollo510 Lite, Apollo330 Plus only)	- Make sure that XTAL_HS/EXTREFCLK in Board Information is correctly defined as described in step 1 for “<i>Porting Guide for the BSP</i>” on page 74. - Make sure that PLLPOSTDIV clock is configured for 24 MHz as described in step 2 for “<i>Porting Guide for the BSP</i>” on page 74.
XTAL_HS (Not applicable to Apollo510B)	Make sure that XTAL_HS in Board Information is correctly defined as described in step 1 for “<i>Porting Guide for the BSP</i>” on page 74.
EXTREFCLK	Make sure that EXTREFCLK in Board Information is correctly defined as described in step 1 for “<i>Guidelines for EXTREFCLK on Apollo510 and Apollo510B</i>” on page 78.
HFRC	Nothing needs to be done for Clock Manager.

After the above steps are done, the USB HAL will request and release the clock automatically, and Clock Manager will start and stop the clocks according to requirement to achieve the lowest possible power for the device’s state.

7.11.8.2 UART Applications

To prevent potential clock drifting issues which might compromise UART communication at 3 MBaud rate, the ability to specify the use of the SYSPLL (Apollo510 SoC and Apollo510B SoC only) or PLLVCO/PLLPOSTDIV (Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs only) as the UART clock source in the Clock Manager is available. Thus the SYSPLL or PLLVCO/PLLPOSTDIV is utilized by the UART’s HAL when 3 MBaud rate is selected. Note that In cases where the UART HAL utilizes this as its clock source, the following conditions must be met:

- Ensure that the **XTAL_HS/EXTREFCLK** clock in Board Information is correctly defined as described in step 1 for “*Porting Guide for the BSP*” on page 74.
- Ensure that the **SYSPLL or PLLVCO/PLLPOSTDIV** clock is configured correctly as described in step 2 for “*Porting Guide for the BSP*” on page 74 to 49.152 MHz.

For the UART modules on the Apollo510 SoC and Apollo510B SoC, the clocks available at the module input are a 2, 4, 8, 16 or 32 divisor of the HFRC_96MHz clock or the PLL_CLK (SYSPLL).

For the UART modules on the Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs, the clocks available at the module input is HFRC_96MHz or the PLLPOSTDIV clock. Either of these clocks may be subsequently integer divided by 2 to 32.

Table 9: UART Application Clock Source Selection

Selection	Procedures
SYSPLL (Apollo510 and Apollo510B only)	1. SoC-dependent: A. Apollo510: Make sure that XTAL_HS/EXTREFCLK in Board Information is correctly defined as described in step 1 for “<i>Porting Guide for the BSP</i>” on page 74. B. Apollo510B: Make sure that EXTREFCLK in Board Information is correctly defined as described in step 1 for “<i>Porting Guide for the BSP</i>” on page 74. 2. Make sure that SYSPLL is configured correctly as described in step 2 in “<i>Porting Guide for the BSP</i>” on page 74.
PLLPOSTDIV (Apollo510 Lite, Apollo330 Plus only)	1. Configure PLLVCO and PLLPOSTDIV following the sequence in “<i>Guidelines for PLLVCO and PLLPOSTDIV on Apollo510 Lite and Apollo330 Plus</i>” on page 78. Note that the default PLLVCO configured by the BSP is 250 MHz, which is incompatible with the UART clock. So the PLLVCO should be set to a multiple of the PLLPOSTDIV frequency and as close as possible to, but not exceed, 250 MHz.
HFRC	Nothing needs to be done for Clock Manager.

7.11.8.3 Audio Applications (AUDADC/PDM/I²S)

Confirm that clock source selection for the audio block conforms to the guidance provided in Table 10.

Table 10: Audio Application Clock Source Selection

Selection	Procedures
SYSPLL (Apollo510 and Apollo510B only)	- SoC-dependent: - Apollo510: Make sure that XTAL_HS/EXTREFCLK in Board Information is correctly defined as described in step 1 for “Porting Guide for the BSP” on page 74. - Apollo510B: Make sure that EXTREFCLK in Board Information is correctly defined as described in step 1 for “Porting Guide for the BSP” on page 74. - Make sure that SYSPLL is configured correctly as described in step 2 in “Porting Guide for the BSP” on page 74. Unlike other modules, for the I2S modules, the clock available at the module input is the divided SYSPLL clock. Hence, the clock requested to Clock Manager for SYSPLL needs to be divided. AM_HAL_I2S_CLKSEL_PLL_POSTDIV is used to set eClock as follows: - eClock = AM_HAL_I2S_CLKSEL_PLL_FOUT3, request for clock expected at module input multiplied by 6. - eClock = AM_HAL_I2S_CLKSEL_PLL_FOUT4, request for clock expected at module input multiplied by 8.
PLLPOSTDIV (Apollo510 Lite, Apollo330 Plus only)	- Configure PLLVCO and PLLPOSTDIV following the sequence in “Guidelines for PLLVCO and PLLPOSTDIV on Apollo510 Lite and Apollo330 Plus” on page 78. Note that the default PLLVCO configured by the BSP is 250 MHz, which is incompatible with the audio clock. So the PLLVCO should be set to a multiple of the PLLPOSTDIV frequency and as close as possible to, but not exceeding, 250 MHz. Unlike other modules, for the I2S modules, the clock available at the module input is the divided PLLPOSTDIV clock. Hence, the clock requested to the Clock Manager for PLLPOSTDIV needs to be divided. AM_HAL_I2S_CLKSEL_PLL_POSTDIV is used to set eClock as follows: - eClock = AM_HAL_I2S_CLKSEL_PLL_FOUT3, request for clock expected at module input multiplied by 6. - eClock = AM_HAL_I2S_CLKSEL_PLL_FOUT4, request for clock expected at module input multiplied by 8.
XTAL_HS (Not applicable to Apollo510B)	Make sure that XTAL_HS in Board Information is correctly defined as described in step 1 for “Porting Guide for the BSP” on page 74.
EXTREFCLK	Make sure that EXTREFCLK in Board Information is correctly defined as described in step 1 for “Porting Guide for the BSP” on page 74.
HFRC	Nothing needs to be done for Clock Manager.

8. Real Time Clock (RTC)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

8.1 Calendar Counters

The real time is held in a set of eight Calendar Counters, which hold the following counts in BCD format:

- current 1/100th of a second (RTC_CTRL0W_CTRL100)
- current second (RTC_CTRL0W_CTRLSEC)
- current minute (RTC_CTRL0W_CTRLMIN)
- current hour (RTC_CTRL0W_CTRLHR)
- current day of the month (RTC_CTRLUP_CTRLDATE)
- current day of the week (RTC_CTRLUP_CTRLWKDY)
- current month (RTC_CTRLUP_CTRLMO)
- current year (RTC_CTRLUP_CTRLYR)
- current century (RTC_CTRLUP_CCB)

In order to ensure that the RTC starts precisely, the timer chain which generates the 100 Hz clock is reset to 0 whenever any of the Calendar Counter Registers is written. Since unintentional modification of the Calendar Counters is a serious problem, the RTC_RTCCTL_WRTC bit must be set in order to write any of the counters, and should be reset by software after any load of the Calendar Counters.

Software may stop the clock to the Calendar Counters by setting the RTC_RTCCTL_RSTOP bit. This may be used in modes like Stopwatch to precisely start and stop the Calendar Counters.

8.2 Calendar Counter Reads

The RTC includes special logic to help insure that the Calendar Counters may be read reliably, i.e. that no rollover has occurred. Because two 32-bit reads are required to read the complete set of counters, it is possible that a delay occurs between the two reads which causes a rollover to occur. An interrupt is the most likely reason this could occur. Software should read the RTC counters multiple times to insure that the value is correct in case a rollover occurs.

8.3 Alarms

There are seven Alarm Registers which may be used to generate an Alarm interrupt at a specific time. These registers correspond to the 100th of a second (RTC_ALMLOW_ALM100), second (RTC_ALMLOW_ALMSEC), minute (RTC_ALMLOW_ALMMIN), hour (RTC_ALMLOW_ALMHR), day of the month (RTC_ALMUP_ALMDATE), day of the week (RTC_ALMUP_ALMWKDY) and month (RTC_ALMUP_ALMMO) Calendar Counters. The comparison is controlled by the RTC_RTCCTL_RPT field and the RTC_ALMLOW_ALM100 Register as shown in 24-hour mode. In the ALM100 Register, n

indicates any digit 0-9. When all selected Counters match their corresponding Alarm Register, the ALM interrupt flag is set (see the Clock Generator section for the ALM interrupt control).

Table 11: Alarm RPT Function

RPT Value	Interval	Comparison
000	Disabled	None
001	Every year	100 th , second, minute, hour, day, month
010	Every month	100 th , second, minute, hour, day
011	Every week	100 th , second, minute, hour, weekday
100	Every day	100 th , second, minute, hour
101	Every hour	100 th , second, minute
110	Every minute	100 th , second
111	Every second	100 th

All alarm interrupts are asserted on the next 100 Hz clock cycle after the counters match the alarm register, except for 100ths of a second. To get an interrupt that occurs precisely at a certain time, the comparison value in the corresponding alarm register should be set 10 ms (one 100 Hz count) earlier than the desired interrupt time.

For the 100ths of a second interrupt, the first 100 Hz clock sets the comparison with the alarm register and the next clock asserts the interrupt. Therefore, the first 100ths interrupt will be asserted after 20 ms, not 10 ms. This occurs each and every time the 100ths of a second counter with interrupts is enabled if the RTC is stopped. If the RTC is already running when configured, then the first interrupt will occur between 10 and 20 ms after configuration.

8.4 Century Control and Leap Year Management

The RTC_CTRUP_CB bit indicates the current century. A value of 1 indicates the 21st century (2000s) and a value of 0 indicates the 22nd century (2100s).

```
//
```

```
// Handle the century bit is for tracking 2000s - 2100s
```

```
//
```

```
// 2000s -> ui32CenturyBit == 1
```

```
// 2100s -> ui32CenturyBit == 0
```

```
//
```

The century value is used to control the Leap Year functions, which create the correct insertion of February 29 in years which are divisible by 4 and not divisible by 100, and also the year 2000. The CEB bit is no longer supported and should be kept cleared (default) when setting up the RTC in the HAL. There is no automatic toggle of the CB field when the CTRYR field rolls over from 99 to 00."

8.5 Weekday Function

The Weekday Counter is simply a 3-bit counter which counts up to 6 and then resets to 0. It is the responsibility of software to assign particular days of the week to each counter value.

9. Counter/Timer Module (TIMER)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

9.1 Functional Description

9.1.1 Counter Operation

The timer counter implements a 32-bit up-counter that increments at a selected clock rate. Counter operation varies by configured timer function mode. See "Timer Modes" on page 86. In EDGE and UPCOUNT modes, the counter increments continuously from 0 to 0xFFFFFFFF, wrapping on overflow unless stopped by compare match. PWM mode automatically reloads the counter to 0 when it reaches the TMRnCMP0 value ($n = 0$ to 15 throughout), creating periodic operation. The counter data path includes input synchronization for external trigger sources and output comparison logic for dual compare channel functionality.

9.1.2 Clock Sources and Prescalers

The timer peripheral accepts multiple clock sources with programmable prescalers:

- HFRC (96 MHz RC): Divisors of 4, 16, 64, 256, 1024 and 4096 provide 24 MHz down to 23.4 kHz with moderate power consumption. Suitable for general-purpose timing and moderate-speed PWM.
- LFRC (1 kHz RC): Divisors of 1, 2, 32 and 1024 provide 1 kHz down to ~1 Hz. Optimized for low-power long-duration timing with reduced accuracy compared to crystal sources.
- XT/XTAL_LS (32.768 kHz crystal): Divisors of 1, 2, 4, 8, 16, 32 and 128, plus fixed 512 Hz output, provide 32.768 kHz down to 256 Hz for a high-accuracy, low-power clock source for RTC-synchronized timing applications.
- HFRC2¹ (125 MHz): Divisors of 8, 16, 32, 64, 128 and 256 provide 15.625 MHz down to 488 kHz for highest-resolution timing source for precision PWM and fast event counting.
- XTAL_HS (48 MHz crystal): Divisors of 1, 2 and 4 provide 48 MHz down to 12 MHz for a high-accuracy, high-resolution clock source for USB-synchronized or precision timing applications.

9.1.3 Compare and Interrupt Operation

Each timer implements dual independent compare channels. When the counter value equals a compare register (TMRnCMP0 or TMRnCMP1), the timer hardware generates a compare match event. This event can trigger output pin state changes, counter reload (in PWM mode), and interrupt generation to the NVIC. Compare interrupts are individually maskable through the INTEN register, with two interrupt bits allocated per timer (one for each compare channel). The interrupt status register (INTSTAT) provides read-only flags indicating which timers and channels have pending interrupts. Software services interrupts by reading INTSTAT, processing the event, and writing the appropriate bits to INTCLR to acknowledge and clear the interrupt condition.

9.1.4 Output Control

Timer outputs route to GPIO pads through the OUTCFG register array. Each timer generates two output signals: OUT0 and OUT1, corresponding to the two compare channels. The output behavior depends on timer mode: in EDGE mode, outputs toggle on compare match; in PWM mode, OUT0 follows the PWM waveform; in PATTERN mode, outputs reflect the pattern bit sequence. Output polarity is independently invertible via TMRnPOL0 and TMRnPOL1 bits, enabling complementary PWM generation for H-bridge

1. Available on Apollo510 SoC and Apollo510B SoC only.

motor control. The OUTCFG registers use a packed 7-bit encoding to map timer outputs to specific GPIO pads, with software utilizing `am_hal_timer_output_config()` to configure the routing.

9.1.5 Timer Modes

The sixteen 32-bit counters of the Timer Module can be used for a number of generic counting or waveform generation functions as described in Table 12 and the following subsections. Note that any reference to the use of a GPIO for clock or trigger input, or for a timer output, presumes that the GPIO's function selection is set as the CT (Counter/Timer) function. See the GPIO chapter's Section 12.3 on page 116 for more detail.

NOTE

Two timers are reserved by the HAL. One is optional for the RTC (Timer 14, ERR007) - see `am_hal_rtc.h`, and the other is for Spot Manager (Timer 15) - see `am_hal_global.h`.

The timer supports the following operating modes selected via the `TMRnFN` field:

EDGE Mode: Counter increments continuously. Output toggles on `TMRnCMP0` match. Optional second transition on `TMRnCMP1` match enables asymmetric output waveforms. Counter continues incrementing beyond compare matches, suitable for event counting and elapsed time measurement.

UPCOUNT Mode: Simple incrementing counter without compare-driven reload. Counter increments from 0 to `0xFFFFFFFF` and wraps. Compare registers generate interrupts but do not affect counter progression. Suitable for free-running time base or event timestamp applications.

PWM Mode: Counter auto-reloads to 0 when reaching `TMRnCMP0`, creating precise periodic operation. Output asserts at `counter=0` and de-asserts at `counter=TMRnCMP1`, generating PWM waveform with duty cycle = $TMRnCMP1/TMRnCMP0$. Provides glitch-free PWM output suitable for LED dimming, servo control, and power regulation applications.

PATTERN Mode: Generates repeating bit patterns on output pin. Pattern length (1-64 bits) defined by `TMRnLMT` field. Each compare match advances to next pattern bit. Enables serial data transmission or complex waveform generation without CPU intervention.

Table 12: Timer Modes

Mode	Outputs	Description/Uses
EDGE	OUT0 transitions when $TIMERn = CMP0$. OUT1 transitions when $TIMERn = CMP1$ (if $CMP1 < CMP0$).	Edge generation (0x1): $TIMERn$ counts up from 0 to $CMP0$ and stops. A single edge is generated on OUT0 when $TIMERn$ reaches $CMP0$. If $CMP1 < CMP0$, then a single edge also is generated on OUT1 when $TIMERn$ reaches $CMP1$. If $CMP0 < CMP1$, then no edge is generated on OUT1 (because $TIMERn$ never reaches $CMP1$). $TMRnLMT$: Has no effect in this mode. Trigger: Timer does not start until trigger occurs (if enabled). Subsequent triggers ignored.
UPCOUNT	OUT0 pulses for 1 clock cycle when $TIMERn = CMP0$; counter resets to 0; repeats per $TMRnLMT$ setting. OUT1 pulses for 1 clock cycle when $TIMERn = CMP1$.	Repeatable Up-counter (0x2): Counts up from zero to $CMP0$ and stops ($TMRnLMT = 1$), repeats N times ($TMRnLMT = 2 - 255$), or repeats indefinitely until $TMRnEN$ is cleared ($TMRnLMT = 0$). Timer outputs will be a pulse of one source clock period on the output when the $TIMER$ reaches the associated CMP value. If $CMP1 < CMP0$, then a single pulse of one source clock period is also generated on OUT1 when $TIMERn$ reaches $CMP1$. If $CMP0 < CMP1$, then no pulse is generated on OUT1 (because $TIMERn$ never reaches $CMP1$). Trigger: Timer does not start until trigger occurs (if enabled). Subsequent triggers ignored.
PWM	OUT0 transitions when $TIMERn = CMP1$, then transitions again when $TIMERn = CMP0$; repeats per $TMRnLMT$ setting. OUT1 is the complement of OUT0.	Repeatable PWM (0x4): Counts up from zero to $CMP0$ and stops ($TMRnLMT=1$), repeat N times ($TMRnLMT = 2 - 255$), or repeat indefinitely ($TMRnLMT = 0$). $CMP0$ specifies the period and $CMP1$ specifies number of $TIMERn$ counts for the initial phase of the output waveform. Trigger: Timer does not start until trigger occurs (if enabled). Subsequent triggers ignored.

Table 12: Timer Modes

Mode	Outputs	Description/Uses
SINGLEPATTERN	For TMRnLMT < 32 and for TIMERN count = 0 to TMRnLMT: OUT0 = CMP0[TIMERN]. OUT1 = CMP1[TIMERN]. For TMRnLMT between 32 and 63 (maximum 64-bit pattern consisting of CMP1: CMP0) and for TIMERN count = 0 to TMRnLMT: OUT0 = OUT1 = CMP1: CMP0[TIMERN].	Single-run Pattern Generation (0xC): CMP0 and CMP1 are bit-shifted to form the output pattern on OUT0 and OUT1, or concatenation of CMP1: CMP0 is bit shifted to form the output pattern on both OUT0 and OUT1. The TIMER count is an up-counter that indexes into the bits of CMP0 (TIMER values from 1-31) and CMP1 (TIMER values 32-63). TMRnLMT value defines the length of the pattern minus 1 (0 = 1-bit pattern, 63 = 64-bit pattern). OUT1 is the same as OUT0, but can be inverted for motor control applications. INT0 is triggered when all TMRnLMT+1 bits have been streamed out. INT1 is triggered only when TMRnLMT is set to 31 and all 32 bits of CMP0 have been streamed out. Trigger: Timer does not start until trigger occurs (if enabled). After clearing and reconfiguring the timer for this mode, a subsequent trigger restarts the pattern replay.
REPEATPATTERN	For TMRnLMT < 32 and for TIMERN count = 0 to TMRnLMT: OUT0 = CMP0[TIMERN]. OUT1 = CMP1[TIMERN]. For TMRnLMT between 32 and 63 (maximum 64-bit pattern consisting of CMP1: CMP0) and for TIMERN count = 0 to TMRnLMT: OUT0 = OUT1 = CMP1: CMP0[TIMERN]. OUT1 can be configured to be the complement of OUT0.	Repeated Pattern Generation (0xD): CMP0 and CMP1 are bit-shifted to form the output pattern on OUT0 and OUT1 as described in SINGLEPATTERN mode with the exception that the timer repeats the pattern after TMRnLMT+1 bits have been streamed out, resetting TMRnLMT back to 0 after each iteration. Since TMRnLMT is used for the repeat pattern length, there is no option to repeat the pattern N times. The iterations repeat indefinitely until TIMERN is disabled.

NOTE

The TIMER_TIMERN registers are read-only and directly writing to these registers have no effect.

NOTE

Stopping the timer by de-asserting the TIMER_CTRLn_TMRnEN bit after the timer has completed a mode operation will *always* clear the timer count (TIMERN register). The counter should be read before disabling the timer if the count is of interest.

NOTE

For EDGE, UPCOUNT and PWM timer modes, the TMRnINT interrupt ($n = 0$ to 15, $m = 0$ or 1) is triggered, if enabled, when TMRnCMPm value is equal to the TIMERN count value. For SINGLEPATTERN and REPEATPATTERN modes, the TMRn0INT interrupt ($n = 0$ to 15) is triggered, if enabled, when the TMRnLMT value is reached. If TMRnLMT was initially set to 31 then TMRn1INT will also be triggered if enabled.

NOTE

CMP0/CMP1 values should not be changed when the TIMER is running to prevent corrupting the counter.

NOTE

The TMRnLMTVAL register holds the instantaneous value of a counter for TMRnLMT and can be read to get either:

- the number of executed iterations of the mode in EDGE, UPCOUNT or PWM mode, or
- the number of bits already streamed out of the TMRnLMT+1 total bits in SINGLEPATTERN OR REPEATPATTERN mode.

NOTE

Clock synchronization has been implemented between the TIMER module's clock source and apb_clk. However this clock synchronization results in a delay of 2 clock cycles for the following operations:

- TMR_EN
- TRIGGER
- STIMER_COMPARE

For critical timing requirements, the application should account for this delay.

9.1.5.1 Edge-generating Single Run Up-counter (TMRnFN = 0x1)

The Edge-generating Single Run Up-counter, or EDGE, timer mode is selected for TIMERN by setting the CTRLn_TMRnFN field to 0x1 ($n = 0$ to 15).

- When the timer is enabled, the pin outputs, OUT0 and OUT1, are at the level selected by the CTRLn_TMRnPOLm bit ($m = 0$ or 1).
- TIMERN is at zero because CTRLn_TMRnCLR has been asserted previously.
- The timer will not start until a trigger occurs when enabled by the CTRLn_TMRnTMODE and as designated by the MODEn_TMRnTRIGSEL field.
- TIMERN counts up on each clock, the source of which is selected by CTRLn_TMRnCLK field, until TIMERN is equal to the value of the TMRnCMP0 register.

- The state of OUT0 is switched when the TIMERN count hits the value of TMRnCMP0 register, and maintains that level until TIMERN is cleared. TIMERN stops after the successful compare.
- If TMRnCMP1 is set lower than TMRnCMP0 (shown in Figure 8), then the state of OUT1 is switched when the TIMERN count hits the value of TMRnCMP1 register, while TIMERN continues counting. Otherwise, OUT1 does not transition (shown in Figure 9).
- An interrupt, if enabled by setting the INTEN_TMRnmINT bit ($n = 0$ to 15, $m = 0$ or 1), is triggered when TMRnCMPm value is equal to the TIMERN count value. It is cleared by writing a one to the corresponding INTCLR_TMRnmINT bit.
- If the polarity for an output is set to NORMAL (CTRLn_TMRnPOLm = 0x0), then OUTm goes high when TMRnCMPm = TIMERN. When an output is set to INVERTED, then the transitions occur at the same times but are just the opposite in direction.
- After the successful compare and the clearing of the timer (asserting CTRLn_TMRnCLR), subsequent triggers start TIMERN counting again.

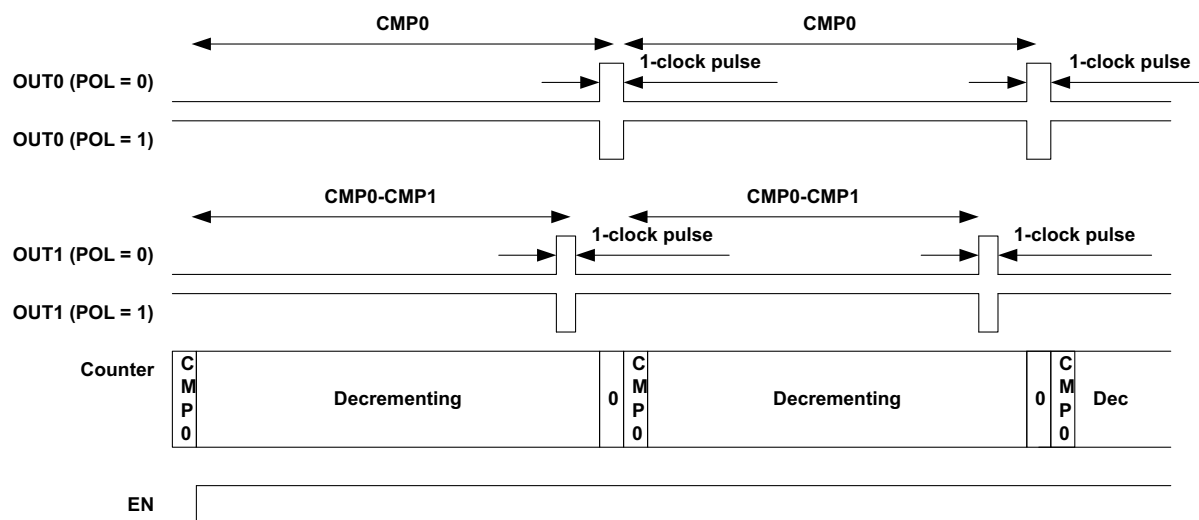


Figure 8. Timer Edge Mode ($TMRnFN = 0x1$) - $CMP0 > CMP1$

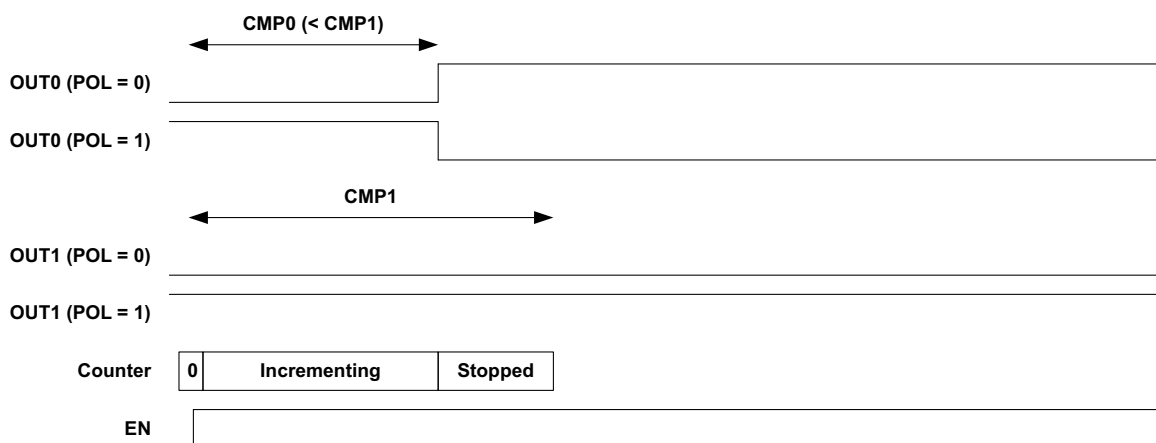


Figure 9. Timer Edge Mode ($TMRnFN = 0x1$) - $CMP0 < CMP1$

9.1.5.2 Repeatable Pulse-generating Up-counter (TMRnFN = 0x2)

The Repeatable Pulse-generating Up-counter, or UPCOUNT, timer mode is selected for TIMERN by setting the CTRLn_TMRnFN field to 0x2 (n = 0 to 15).

- When the timer is enabled, the pin outputs, OUT0 and OUT1, are at the level selected by the CTRLn_TMRnPOLm bit (m = 0 or 1).
- TIMERN is at zero because CTRLn_TMRnCLR has been asserted previously.
- The timer will not start until a trigger occurs when enabled by the CTRLn_TMRnTMODE and as designated by the MODEn_TMRnTRIGSEL field.
- TIMERN counts up on each clock, the source of which is selected by CTRLn_TMRnCLK field, until TIMERN is equal to the value of the TMRnCMP0 register.
- When TIMERN is equal to the value of either of the TMRnCMPm registers, a pulse of polarity opposite of the start state of OUTm is generated, after which OUTm returns to its original state.
 - The pulse is of 1 clock cycle in duration.
 - TMRnCMPm must be at least 1 so that the repeat interval is two clock cycles.
- If the polarity for an output is set to NORMAL (CTRLn_TMRnPOLm = 0), then OUTm starts off in the low state and pulses high when TMRnCMPm = TIMERN. When an output is set to INVERTED (TMRnPOLm = 1), then OUT0/OUT1 starts out in the high state and pulse low for 1 source clock pulse when CMP0/ CMP1 value is reached by TIMERN. The generated pulses occur at the same times but are the opposite in polarity.
- TIMERN continues counting after the successful compare and pulses are generated on OUT0 or OUT1 in the above manner, creating a stream of pulses or interrupts at a fixed interval until TIMERN is stopped.
- When stopped or disabled, TIMERN will stop counting but will not be cleared. Asserting CTRLn_TMRnCLR will reset TIMERN to zero.
- The CTRLn_TMRnLMT field can be set from 0 to 255 to specify the number of timer counter cycles before TIMERN stops counting. If this field is set to 0, then TIMERN counts indefinitely until it is disabled. This operation is as shown in Figure 10.
- An interrupt, if enabled by setting the INTEN_TMRnmINT bit (n = 0 to 15, m = 0 or 1), is triggered when TMRnCMPm value is equal to the TIMERN count value. It is cleared by writing a one to the corresponding INTCLR_TMRnmINT bit.
- After the successful compare and the clearing of the timer (asserting CTRLn_TMRnCLR), subsequent triggers start TIMERN counting again.

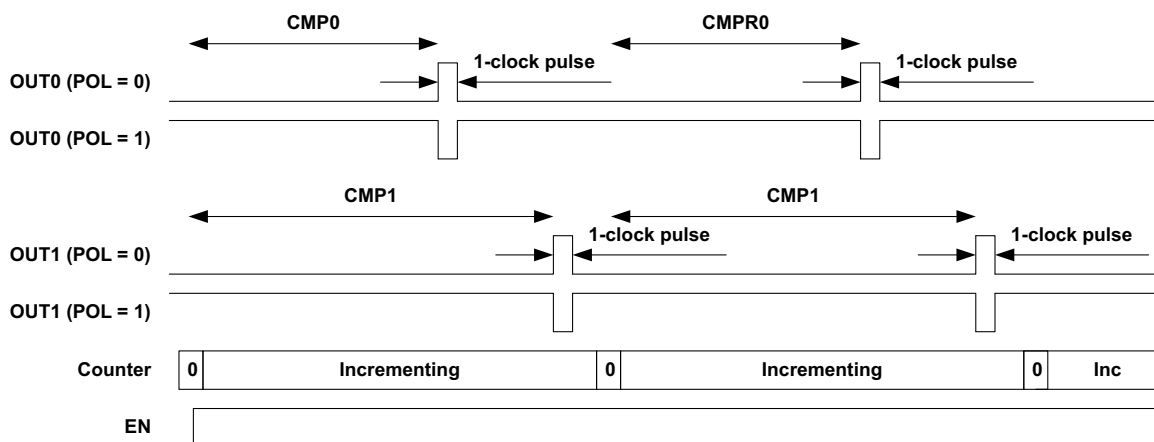


Figure 10. Timer Repeated Pulse Up-counter Compare Mode (TMRnFN = 0x2)

9.1.5.3 Repeatable PWM-generating Up-counter (TMRnFN = 0x4)

The Repeatable PWM-generating Up-counter, or PWM, timer mode is selected for TIMERN by setting the CTRLn_TMRnFN field to 0x4 (n = 0 to 15).

- When the timer is enabled, the pin outputs, OUT0 and OUT1, are at the level selected by the CTRLn_TMRnPOLm bit (m = 0 or 1).
- TIMERN is at zero because CTRLn_TMRnCLR has been asserted previously.
- The CTRLn_TMRnLMT field can be set from 0 to 255 to specify the number of timer counter cycles before TIMERN stops counting. If this field is set to 0, then the timer counts until it is disabled.
- The timer will not start until a trigger occurs when enabled by the CTRLn_TMRnTMODE and as designated by the MODEn_TMRnTRIGSEL field.
- TIMERN counts up on each clock, the source of which is selected by CTRLn_TMRnCLK field, until TIMERN is equal to the value of the TMRnCMP0 register (output period).
- When TIMERN is equal to the value of the TMRnCMP1 register, OUT0 transitions to the opposite polarity (high if CTRLn_TMRnPOLm = 0).
- TIMERN continues counting and when it is equal to the value of the TMRnCMP0 register, OUT0 transitions to the original polarity (low if CTRLn_TMRnPOLm = 0).
 - The complementary signal of OUT0 is output on OUT1.
 - TIMERN clears and repeats the counting-and-compare cycle until the TMRnLMT field value is reached or TIMERN is disabled. This operation is as shown in Figure 11.
- An interrupt, if enabled by setting the INTEN_TMRnmINT bit (n = 0 to 15, m = 0 or 1), is triggered when TMRnCMPm value is equal to the TIMERN count value. It is cleared by writing a one to the corresponding INTCLR_TMRnmINT bit.

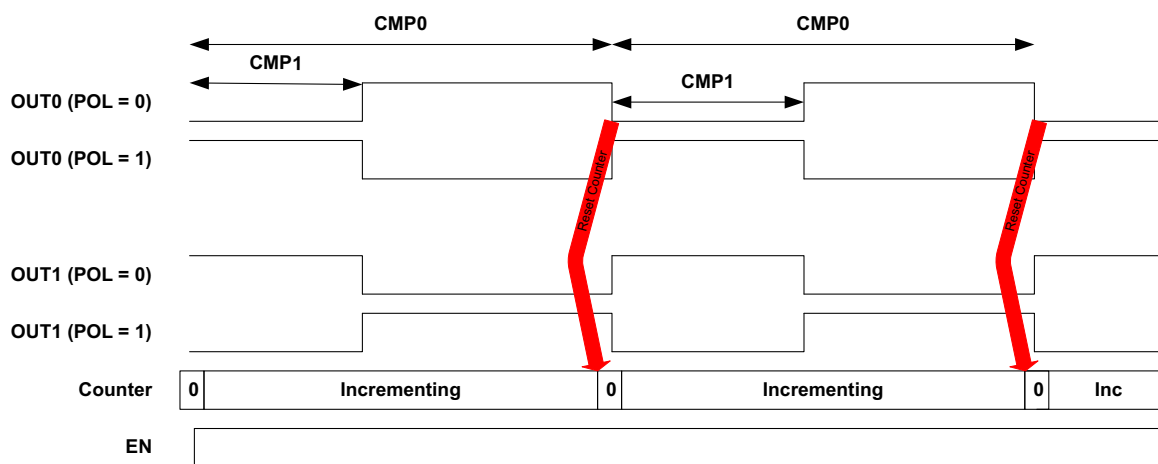


Figure 11. Timer PWM Mode (TMRnFN = 0x4)

9.1.5.4 Single Run Pattern-generating Up-counter (TMRnFN = 0xC)

The Single Run Pattern-generating Up-counter, or SINGLEPATTERN, timer mode is selected for TIMERN by setting the CTRLn_TMRnFN field to 0xC (n = 0 to 15).

There are two general cases for this mode as follows.

- CTRLn_TMRnLMT ≤ 31 (0x1F):
 - OUT0 and OUT1 are independent output patterns derived from (up to) 32 bits stored in TMRnCMP0 and TMRnCMP1, respectively.
 - As TIMERN counts up with value x, OUT0 = TMRnCMP0[x] bit and OUT1 = TMRnCMP1[x] bit for a clock cycle.

- After CTRLn_TMRnLMT cycles, the outputs terminate. See Figure 12.
- CTRLn_TMRnLMT between 32 and 63:
 - OUT0 is a pattern consisting of 33 - 64 bits comprised of the concatenated TMRnCMP1:TMRnCMP0 value.
 - CMP0 holds TIMERN values from 0-31 and CMP1 holds TIMERN values 32-63.
 - As TIMERN counts up with value x, OUT0 = TMRnCMP1:TMRnCMP0[x] bit for a clock cycle and for CTRLn_TMRnLMT total cycles.
 - At the same time OUT1 gets the same values as OUT0, but can be inverted for motor control applications with (CTRLn_TMRnPOL1 = 0x1). See Figure 13.
- When the timer is enabled, the pin outputs, OUT0 and OUT1, are at the zero level because CTRLn_TMRnCLR has been asserted previously.
- The timer will not start until a trigger occurs if enabled by the CTRLn_TMRnTMODE and as designated by the MODEn_TMRnTRIGSEL field.
- TIMERN counts up on each clock, the source of which is selected by CTRLn_TMRnCLK field.
- Each successive TIMERN counter value is the index for reading and outputting the bits of the 32-bit values stored in TMRnCMP0 and TMRnCMP1.
- The INT0 interrupt, if enabled by setting the INTEN_TMRn0INT bit, is triggered when the TIMERN count = TMRnLMT value. It is cleared by writing a one to the corresponding INTCLR_TMRn0INT bit.
- After pattern output and the clearing of the timer by asserting CTRLn_TMRnCLR followed by reconfiguring the timer, a subsequent trigger starts TIMERN counting again.

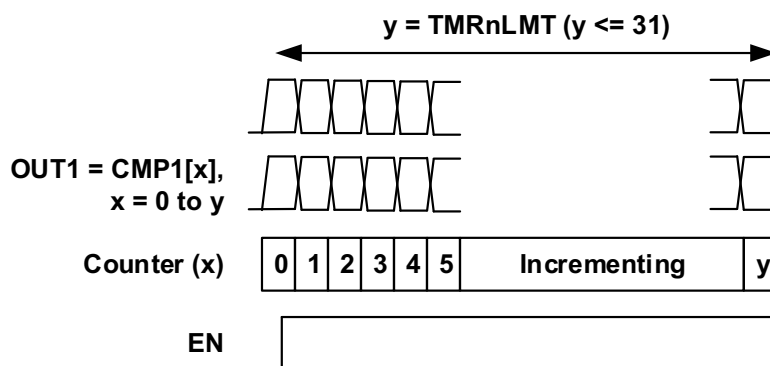


Figure 12. Timer Single 32-bit Pattern Output (TMRnFN = 0xC)

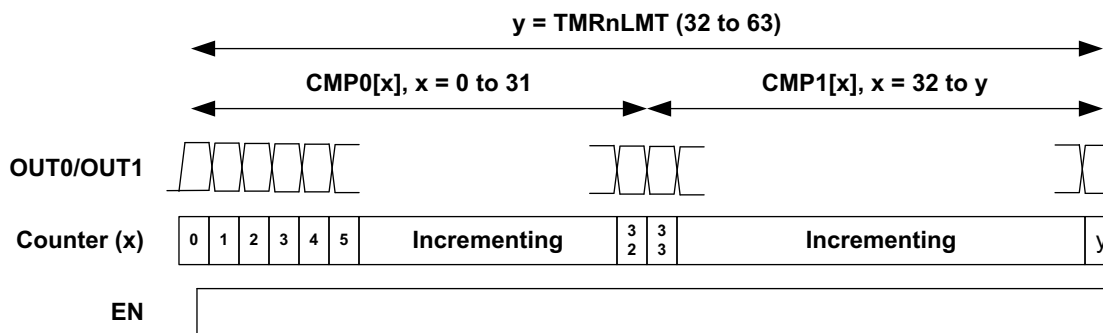


Figure 13. Timer Single 64-bit Pattern Output (TMRnFN = 0xC)

9.1.5.4.1 Single 32-bit Pattern Output Example

- TMRnLMT = 24
- CMP0 is 0x55555555 and CMP1 is 0x33333333
 - CMP0 in binary = 0101 0101 0101 0101 0101 0101 0101 0101
 - CMP1 in binary = 0011 0011 0011 0011 0011 0011 0011 0011
- Since value of TMRnLMT < 32, OUT0 and OUT1 are independent.
 - OUT0 is at the level of each successive bit of CMP0 indexed from LSB to MSB.
 - OUT1 is at the level of each successive bit of CMP1 indexed from LSB to MSB.
 - 25 bits (TMRnLMT+1) are indexed.

The 25 bits for the output waveforms are as shown in Figure 14.

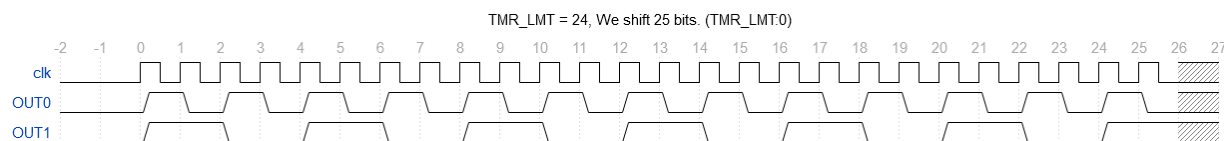


Figure 14. Single 32-bit Pattern Output

9.1.5.4.2 Single 64-bit Pattern Output Example

- TMRnLMT = 45
- CMP0 and CMP1 are both 0x55555555
 - CMP0/CMP1 in binary = 0101 0101 0101 0101 0101 0101 0101 0101
- Since value of TMRnLMT >= 32, OUT0 and OUT1 have the same waveform.
 - OUT0/OUT1 is at the level of each successive bit of CMP0: CMP1 indexed starting with the LSB of CMP0 to the MSB of CMP1.
 - A maximum of 64 bits (TMRnLMT+1) are indexed.

The 45 bits for the output waveforms are as shown in Figure 15.



Figure 15. Single 64-bit Pattern Output

9.1.5.5 Repeated Pattern-generating Up-counter (TMRnFN = 0xD)

The Repeated Pattern-generating Up-counter, or REPEATPATTERN, timer mode is selected for TIMERN by setting the CTRLn_TMRnFN field to 0xD (n = 0 to 15).

The functionality and timing of this mode are exactly the same as Single Pattern Output (TMRnFN = 0xC) except that it repeats the pattern in a continuous manner. Refer to “Single Run Pattern-generating Up-counter” for details.

- Since the CTRLn_TMRnLMT field is used to specify the number of bits used in CMP0 and CMP1 to form the output pattern, it cannot be used to specify the number of times the pattern is repeated (as in other repeatable modes).
- This mode behaves as if the CTRLn_TMRnLMT field is set to 0 and repeats indefinitely until TIMERN is disabled.

9.1.6 Triggering Functions

All timer functions optionally can be enabled with a trigger, which can be a GPIO input or an output of another TIMER. Triggers are enabled by using the CTRLn_TMRnTMODE field, which can be configured to trigger on the rising, falling or either edge of the trigger source. The trigger source is selected in the MODEn_TMRnTRIGSEL field as either output of any TIMER (32 selections) or any of the GPIO inputs. A trigger basically holds the timer at its initial value until the trigger occurs. Once triggered, the timer will execute the count/waveform generation as normal.

9.1.6.1 Initiating a One-shot Operation

If the mode of a TIMER is as a one-shot (EDGE or SINGLEPATTERN), when TMRnEN is asserted one cycle of the operation will be executed. At that point, an edge specified by CTRLn_TMRnTMODE on the trigger signal selected by MODEn_TMRnTRIGSEL will cause the operation to be executed again. This allows the creation of complex operations with a single configuration.

9.1.6.2 Terminating a Repeat Operation

The Repeat Pattern mode and the repeatable modes (UPCOUNT and PWM) are terminated by asserting the TMRnCLR bit. When this is done, all the settings in the timer are cleared and they must be set up again to do a new operation.

9.1.7 Clocking Timer/Counters with Other Counter/Timer Outputs

There are cases where it is useful to use the output of a TIMER as the clock of another TIMER. The TMRnCLK field includes choices which implement this function, in addition to the normal clocks taken from the internal oscillators. If the clock of the timer which produces the TRIG signal were taken from the OUT0 output of the first timer, the CMP0 value used for the trigger generator would be trivially calculated as 2, and would be independent of the actual clock used to generate the OUT0 signal.

9.1.8 Timer Cascade and Synchronization

There are times when it is very important to be able to start multiple Timer/Counters at precisely the same time, particularly in cases where one output is used as the trigger of another. The Timer module supports cascaded operation where one timer's output triggers another timer, enabling extended timing ranges or multi-stage waveform generation. The Counter/Timer Global Enable register (GLOBEN) allows simultaneous start/stop of multiple timers with cycle-accurate synchronization, critical for phased PWM generation or coordinated multi-axis motor control. Cascaded timers can extend the effective counter width beyond 32 bits for applications requiring timing intervals exceeding the base timer range.

The TIMER_GLOBEN register contains one enable bit for each Timer/Counter, which is ANDed with the local CTRLn_TMRnEN bit of the timer. The GLOBEN register normally has all bits set to 1, so that the local CTRLn_TMRnEN bits control the timers. For synchronized enabling, the GLOBEN_ENBn register bits to be synchronized are set to 0, and then the local EN bits of those timers are set to 1. At that point a single write to the GLOBEN register will enable all of the selected timers at once.

NOTE

When utilizing cascaded or synchronized timer configurations, account for erratum ERR045 (4-cycle startup delay) affecting the first timer cycle of each enabled timer. For precision applications, adjust initial TMR0CMP0 values by -4 counts to compensate for the hardware synchronization delay.

In addition, a field in the GLOBEN Register, ENABLEALLINPUTS, is a global enable for all selected *GPIO* inputs to the timers. This global *GPIO* input enable bit affects only the enabling of timer clocks and triggers from a *GPIO*. If GLOBEN_ENABLEALLINPUTS is low, then ALL of the timers' *GPIO* clocks and triggers are disabled. This does not affect any timer which uses a clock other than a *GPIO*, e.g., HFRC, LFRC, XT, RTC, BUCK or output of another timer output. Nor does it affect any timer which uses a trigger other than a *GPIO*, e.g., another timer output. This global setting requires, and does not take the place of, setting the CTRLn_TMRnEN bit (and the CTRLn_TMRnTMODE field if used) for any timer intended to be used.

For the synchronous starting of timers by concurrently setting the GLOBEN_ENBn bits as described above, note that if *GPIO* is selected as the clock for any timers, then the ENABLEALLINPUTS field must be set before (or at the same time of) initiating the synchronous starting of the timers. The ENABLEALLINPUTS field is not an override for the individual ENBn bits, as they serve different functions. It is also important to note that this field does not affect the enabling or use of any *GPIO* set up as a timer output.

The steps to use the GLOBEN_ENABLEALLINPUTS when *GPIO* are used as clock or trigger sources are as follows:

1. Write to the GLOBEN register to clear the ENABLEALLINPUTS bit (cleared by default).
2. Write to the MODEn register to set the TMRnTRIGSEL field and TMRnASYNC field (if used).
3. Write to the CTRLn register to set TMRnCLK, TMRnEN and all other intended settings in this register.
4. Write to the GLOBEN register to set the ENABLEALLINPUTS bit.

9.1.9 Generating the Sample Rate for the ADC

TIMER7 has a special function which allows it to function as the sample trigger generator for the ADC. If the TIMER_GLOBEN_ADCEN bit is set, the output of the timer is sent to the ADC which uses it as a trigger. Typically, TIMER7 is configured in Repeating Up-counter Compare (FN=2) mode. INTEN_TMR70INT may be set to generate an interrupt whenever the trigger occurs, but typically the ADC interrupt will be used for this purpose.

9.1.10 Generating the Sample Rate for the Audio ADC

As with the ADC mentioned above, TIMER6 has a special function which allows it to function as the sample trigger generator for the Audio ADC. If the TIMER_GLOBEN_AUDADCEN bit is set, the output of the timer is sent to the Audio ADC which uses it as a trigger. Typically, TIMER6 is configured in Repeating Up-counter Compare (FN = 2) mode. INTEN_TMR60INT may be set to generate an interrupt whenever the trigger occurs, but typically the Audio ADC interrupt will be used for this purpose.

9.1.11 CLR and EN Details

The overall operation of each TIMER is controlled by two configuration bits, TMRnCLR and TMRnEN, in the CTRLn register. When TMRnEN is set to 1, TIMERn is immediately set to all zeros and will remain there independent of any other configuration. TMRnCLR is typically used (writing a 1 to the bit) to initialize TIMERn to a value specific to the selected mode before use.

TMRnEN is used to enable (when 1) and disable (when 0) the counting function of the TIMER. However, TMRnEN is synchronized to the selected clock, which must be accounted for when used. When TMRnEN is set to 0, the counter will be reset. When TMRnEN is set to 1, the counter will resume counting from 0.

Since the operation of the processor is essentially asynchronous to the selected clock, the synchronization introduces an uncertainty as to when the counter will begin counting. If the frequency of the selected clock is high relative to the processor clock, the impact of the synchronization will be negligible. However, for low frequency clocks, external pin clocks and the buck clocks the effective delay caused by the synchronization may be significant.

9.1.12 Counter Functions

A TIMER operates in counter mode when the TMRnCLK field selects either an external pad input (if 0x100-0x1DF) or a buck pulse input (if 0x1C-0x1F). The different clock selections provide different functions.

9.1.12.1 Counting External Edges

If the TMRnCLK field is 0x100-0x1DF, the TIMER clock input comes from an external pad. This allows the TIMER to monitor pulses or edges on an external signal.

9.1.12.2 Counting Buck Converter Edges

The MCU includes four separate buck converter inputs which provide a clock source from Buck VDDC TON, Buck VDDF TON, Buck VDDS TON, or Buck VDDC_LV TON pulses. Each TIMER may be connected to a pulse stream from any of these sources. One pulse is generated each time the Buck Converter inserts charge into the capacitor, and therefore the number of pulses is a good indication of the amount of energy used by the corresponding power domain in a particular time period.

A possible option to determine energy consumption is as follows. Two counters could be configured for repeated UP-counter mode (FN = 2, TMRnLMT=0) so that they count continuously. One is supplied a Buck Converter pulse stream as its clock, and the other is supplied with a divided version of the LFRC clock to avoid creating extra power consumption due to the power measurement. Once configured such, the two counters should be enabled simultaneously, and after some period of system operation they should be disabled and read. The LFRC count value would now define how much real time has elapsed, and the Buck Converter count value would define how much energy was consumed in that time.

9.1.13 Interconnecting Timers

The OUT0 or OUT1 output of any TIMER may be used as either the trigger or the clock of any other TIMER. The selection of the clock or trigger source is made within each TIMERN configuration in the CTRLn_TMRnCLK field or the MODEn_TMRnTRIGSEL field, respectively.

9.2 Design Considerations

9.2.1 Clock Source Selection

Timer clock source selection directly impacts both timing resolution and power consumption. Low-frequency sources (LFRC, XT at 32 kHz) provide microsecond-level resolution with minimal power draw, suitable for long-duration timing and low-power applications. High-frequency sources (HFRC, HFRC2¹, XTAL_HS) with programmable divisors enable nanosecond to microsecond resolution for precise PWM generation and high-speed event counting. Each clock source includes multiple prescaler options to achieve desired timing granularity without excessive count overflow rates.

9.2.2 PWM Resolution and Frequency

PWM applications face an inherent tradeoff between output frequency and duty cycle resolution. The relationship is defined by: $\text{Resolution} = \text{ClockFreq} / (\text{PWM\_Frequency} \times \text{Period\_Counts})$. For example, generating 1 kHz PWM from a 1 MHz timer clock with a 1000-count period provides 10-bit (0.1%) duty cycle resolution. Increasing PWM frequency to 10 kHz with the same clock reduces resolution to 100 counts (1%). Applications requiring both high frequency and fine resolution must utilize higher clock sources or accept reduced duty cycle granularity.

9.2.3 Interrupt Latency Impact

For timing applications requiring cycle-accurate interrupt response, consider the cumulative latency from interrupt assertion to ISR entry, which includes:

- Interrupt controller pipeline depth and priority arbitration
- CPU pipeline flush and exception entry overhead
- NVIC interrupt tail-chaining behavior for back-to-back interrupts
- Cache miss penalties if ISR code is not cache-resident
- System bus contention from concurrent DMA or peripheral access

Worst-case interrupt latency can reach 50-100 CPU clock cycles. For timing-critical applications, account for this variability or utilize timer output pins with hardware-driven transitions that bypass interrupt latency entirely.

1. Available on Apollo510 SoC and Apollo510B SoC only.

9.3 Programming Guidelines

9.3.1 Configuring Timer Operation

Timer configuration follows this initialization sequence, where “n” is the timer number:

1. Disable the target timer through the CTRLn_TMRnEN bit.
2. Select the operating mode (EDGE, PWM, UPCOUNT, PATTERN) via the CTRLn_TMRnFN field.
3. Configure the clock source and prescaler through the CTRLn_TMR0CLK field.
4. Program the TMRnCMP0 compare register for primary timing/PWM period.
5. Program the TMRnCMP1 compare register for secondary timing/PWM duty cycle.
6. Configure the trigger source and type in the MODEn register if using external triggers.
7. Set the pattern limit in the CTRLn_TMRnLMT field for pattern generator modes.
8. Enable the required interrupts via the INTEN register (CMP0, CMP1 match events).
9. Assert the timer clear bit (CTRLn_TMRnCLR) to reset counter to initial state.
10. Enable timer through CTRLn_TMRnEN bit to commence operation.

9.3.2 PWM Generation Setup

To configure a timer for PWM output generation:

1. Set the function mode to AM_HAL_TIMER_FN_PWM in the configuration structure.
2. Calculate the period count: $\text{Period} = (\text{ClockFreq} / \text{PWM\_Freq})$.
3. Program the TMRnCMP0 with calculated period count value.
4. Calculate the duty cycle count: $\text{Duty} = \text{Period} \times \text{DutyCycle\_Fraction}$.
5. Program the TMRnCMP1 with duty cycle transition point.
6. Configure the output polarity through the CTRLn_TMRnPOL0/TMRnPOL1 bits as needed.
7. Map the timer output to GPIO pin via am_hal_timer_output_config().
8. Configure the GPIO pin for the timer function through the GPIO subsystem.
9. Enable the timer to start PWM output generation.

NOTE

The 4-cycle startup delay caused by and described in erratum ERR045 affects the first PWM period timing. Compensate by reducing TMRnCMP0 by 4 counts for the initial configuration when precise first-edge timing is critical.

9.3.3 Synchronized Multi-Timer Operation

To synchronize multiple timers for simultaneous operation:

1. Configure all required timers independently with desired settings.
2. Ensure all timers are disabled (TMRnEN = 0 for each).
3. Call am_hal_timer_enable_sync() with bitmask of timers to start.
4. Hardware simultaneously enables all specified timers on the same clock edge.
5. For synchronized stop, call am_hal_timer_disable_sync() with the same mask.

Synchronized start eliminates phase offset between multiple PWM channels or coordinated timing events.

9.3.4 Constraints

When utilizing the Counter/Timer peripheral, observe the following constraints:

- Account for 4 timer-clock-cycle startup delay in first-edge timing calculations.
- Subtract 4 from TMRnCMP0 initial value when precise first-event timing is required (for periods > 4 cycles).
- Verify timer periods exceed 4 clock cycles for reliable operation; shorter periods may not generate events.
- Test timer behavior across all utilized clock sources, as the 4-cycle delay is constant in timer clocks.
- For time-critical applications intolerant of startup delay, consider using timer output pins with hardware-timed transitions.

9.4 Register Summary

For detailed register descriptions, refer to the TIMER register section in the Apollo510 SoC register set documentation included with the AmbiqSuite SDK.

Key Registers:

- CTRLn - Timer Control Register n
- TMRnEN - Timer enable/disable control bit
- TMRnCLR - Counter clear/reset control (toggle to reset)
- TMRnPOL0/TMRnPOL1 - Output polarity inversion for compare 0/1
- TMRnFN - Timer function mode selection (EDGE, PWM, UPCOUNT, PATTERN)
- TMRnCLK - Clock source and prescaler selection
- TMRnLMT - Pattern limit value for pattern generator modes
- TMRnTMODE - Trigger mode configuration
- TMRnCMP0 - Timer Compare Register 0
 - Primary compare value for timer operation
 - Defines PWM period in PWM mode
 - Generates compare match interrupt when counter equals this value
 - Automatically reloads counter in repeat modes
- TMRnCMP1 - Timer Compare Register 1
 - Secondary compare value for dual-edge timing
 - Defines PWM duty cycle transition point in PWM mode
 - Generates independent compare match interrupt
 - Must be less than TMRnCMP0 in EDGE mode when enabled
- TIMERN - Timer Counter Register (read-only)
 - Current 32-bit counter value
 - Increments at selected timer clock rate
 - Resets to 0 on compare match (depending on mode)
 - Read using `am_hal_timer_read()` for glitch-free operation
- MODEn - Timer Mode Register n
- TMRnTRIGSEL - Trigger source selection for external triggering
 - Selects GPIO or other timer outputs as trigger inputs
 - Trigger configuration for cascaded timer operation
- GLOBEN - Global Timer Enable Register
 - Bit mask for enabling/disabling each timer instance
 - Provides synchronized start/stop for multiple timers
 - Used with `am_hal_timer_enable_sync()` for phase-locked operation
 - Each bit corresponds to one timer (bit 0 = Timer 0, etc.)
- OUTCFGn - Timer Output Configuration Registers
 - Maps timer outputs to specific GPIO pads
 - Configures which timer output (TMRn, STIMER) drives each pad
 - Each pad occupies 7 bits in packed configuration format
 - Four pads configured per OUTCFG register
- INTEN - Interrupt Enable Register
 - Enable/disable interrupts for Compare 0 match (all timers)
 - Enable/disable interrupts for Compare 1 match (all timers)
 - Two interrupt bits per timer: CMP0 and CMP1 events
 - Bitmask format: bits [1:0] = Timer 0, bits [3:2] = Timer 1, etc.
- INTSTAT - Interrupt Status Register (read-only)
 - Compare 0 match interrupt status flags (all timers)
 - Compare 1 match interrupt status flags (all timers)
 - Raw interrupt status before masking
 - Read by ISR to identify which timer(s) triggered interrupt
- INTCLR - Interrupt Clear Register (write-only)

- Write-one-to-clear interrupt status bits
- Clears corresponding flags in INTSTAT register
- Write 1 to clear specific timer interrupt
- Write 0 has no effect on interrupt status

10. System Timer (STIMER)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

10.1 Design Considerations

10.1.1 Power Consumption

The STIMER is designed to run continuously, even during low-power sleep modes. Select the lowest frequency clock source that meets the timing resolution requirements to minimize power consumption.

10.1.2 Timing Resolution vs. Range

With a 32-bit counter:

- At 32 kHz: ~31 μ s resolution, ~38.5 hour range
- At 1 MHz: ~1 μ s resolution, ~71.6 minute range

Use the overflow interrupt to extend the range as needed for long-term timekeeping.

10.1.3 Real-Time Clock Implementation

To implement a complete real-time clock:

1. Use the STIMER as the hardware timebase.
2. Maintain extended time in software using overflow interrupts.
3. Store epoch offset in NVRAM registers.
4. Implement calendar calculations in firmware.
5. Handle leap seconds, daylight savings time, and other irregularities in software.

10.1.4 RTOS Integration

The STIMER is well-suited for RTOS tick generation. The following implementation is given in the FreeRTOS port in the SDK.

1. Configure one compare register for periodic RTOS tick.
2. In the compare ISR, schedule the next tick by writing the delta.
3. Use remaining compare registers for software timers or other scheduling.

10.2 Programming Guidelines

10.2.1 Writing to the STIMER Compare Register

STIMER logic has been updated on the Apollo510 SoC to better handle SCMPR writes. The hardware takes the register value and adds it to the current value of the STTMR and writes the final value to the SCMPRn register. This register write happens on the STIMER clock.

Because the STIMER clock is not fixed and can be changed by the user by setting the STCFG_CLKSEL bit and because internal to hardware there is synchronization logic, hardware takes some STIMER clocks to update the SCMPR register. Specifically, hardware takes 4 STIMER clock cycles to update the value of the SCMPRn register. See "ERR029: STIMER: Constraints on writing to SCMPRn registers and handling Compare interrupts" in the Apollo510 SoC Errata List.

10.2.1.1 Calculating Write Delay

Do the following calculations to determine the length of this delay:

1. Calculate the period of the STIMER clock (1/CLKSEL value).
2. Multiply the period by four (cycles).

This value in ns/μs is the delay that software must wait after writing to SCMPR before reading the updated value of the SCMPR register.

10.2.1.2 1.2.1.2 Example Calculation

For example, if the STIMER clock is running at 32 kHz:

- Period = $1 / 32,000 \text{ Hz} = 31.25 \text{ } \mu\text{s}$
- Delay = $31.25 \text{ } \mu\text{s} \times 4 = 125 \text{ } \mu\text{s}$

Software must wait at least 125 μs after writing to the SCMPR register before reading back the updated value.

10.2.1.3 Usage Constraints:

1. Back-to-back writes: Back-to-back writes to SCMPRn may not work reliably (i.e., take the last value) unless the application ensures not to write within two STIMER clock cycles of the previous one.
2. Read-after-write: After writing to SCMPRn, the application needs to wait for at least three STIMER clock cycles before reading it back for the new value to be reflected.
3. STCFG write delay: It takes two STIMER clock cycles for the write to STCFG to take effect. This requires 2 extra cycles to be added to the minimum delta.
4. Minimum delta value: The minimum delta value that can be applied to SCMPRn is 4 STIMER clock cycles.

10.2.2 Initializing the STIMER

1. Select the appropriate clock source via STCFG_CLKSEL.
2. Wait for at least two STIMER clock cycles for the configuration to take effect.
3. Initialize the NVRAM registers if maintaining real-time information.
4. Configure any required compare or capture registers.

10.2.3 Scheduling Compare Interrupts

To schedule a compare interrupt, follow the sequence listed below. Instructions 2-4 are taken care of in the HAL if using the function `am_hal_stimer_compare_delta_set`.

1. Calculate the desired delta (offset from current time) in STIMER clock cycles.
2. Ensure the delta is at least 4 STIMER clock cycles.
3. Write the delta value to the desired SCMPRn register.
4. The hardware will add the delta to the current STTMR value.
5. Wait at least 4 STIMER clock cycles before writing to the same register again.

10.2.4 Handling Overflow

The STIMER provides an overflow interrupt that fires when the 32-bit counter wraps from 0xFFFFFFFF to 0x00000000. Firmware should use this interrupt to maintain an extended (> 32-bit) time value for epoch tracking.

10.2.5 Time Stamping

To use the capture registers for sensor synchronization:

1. Configure the GPIO or PWM trigger for the desired capture register.
2. When the trigger event occurs, the current STTMR value is latched into the capture register.
3. Read the capture register to obtain the precise timestamp of the event.

10.3 Register Summary

For detailed register descriptions, refer to the STIMER register section in the Apollo510 SoC register set documentation included with the AmbiqSuite SDK.

Key Registers:

- STCFG - STIMER Configuration Register
 - Clock source selection (CLKSEL)
 - Timer enable/disable
 - Compare and capture configuration
- STTMR - STIMER Counter Register (read-only)
 - Current 32-bit counter value
- SCMPR0-SCMPR7 - STIMER Compare Registers
 - Eight 32-bit compare values
 - Write delta values; hardware computes absolute compare
- SCAPT0-SCAPT3 - STIMER Capture Registers (read-only)
 - Four 32-bit captured timestamp values
- SNVR0-SNVR2 - STIMER NVRAM Registers
 - Three 32-bit registers for epoch offset storage
 - Only reset by POA
- STMINTEN - STIMER Interrupt Enable Register
 - Enable/disable compare interrupts
 - Enable/disable overflow interrupt
 - Enable/disable capture interrupts
- STMINTSTAT - STIMER Interrupt Status Register
 - Compare interrupt status flags
 - Overflow interrupt status flag
 - Capture interrupt status flags
- STMINTCLR - STIMER Interrupt Clear Register
 - Write 1 to clear corresponding interrupt flag

11. Watch Dog Timer (WDT)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

11.1 Functional Description

11.1.1 Watchdog Counter Operation

The watchdog timer implements an up-counting 8-bit counter that increments at the selected clock rate. When enabled via CFG_WDTEN bit, the counter begins incrementing from 0. The counter value is continuously compared against two threshold registers: INTVAL and RESVAL. When the count reaches INTVAL and interrupt generation is enabled, the watchdog asserts an interrupt to the NVIC. If the count reaches RESVAL and reset generation is enabled, the watchdog triggers a system reset through the reset generator peripheral. Software prevents timeout by writing the key value 0xB2 to the RSTRT register, which resets the counter to 0 and restarts the timeout cycle. This architecture provides continuous monitoring of software execution, automatically recovering from firmware hangs or infinite loops.

11.1.2 Clock Source Selection

The watchdog timer accepts multiple clock sources selected via the CFG_CLKSEL field:

- LFRC_DIV8 (128 Hz): Provides timeout range of 0 to 2 seconds with 7.8 ms resolution. Suitable for moderate-speed applications requiring second-scale supervision with low power consumption.
- LFRC_DIV64 (16 Hz): Provides timeout range of 0 to 16 seconds with 62.5 ms resolution. Balanced option for general-purpose applications with reasonable timeout granularity and minimal power impact.
- LFRC_DIV1K (1 Hz): Provides timeout range of 0 to 255 seconds (4.25 minutes) with 1-second resolution. Optimized for long-duration supervision in ultra-low-power or infrequently serviced applications.
- LFRC_DIV16K (0.0625 Hz): Provides timeout range of 0 to 4080 seconds (68 minutes) with 16-second resolution. Maximum timeout duration for applications with very slow execution or extended idle periods.
- XTAL_HS (48 MHz): Provides microsecond-resolution timeouts using external high-speed crystal. Enables precise short-duration supervision but requires crystal hardware and increases power consumption.
- XTAL_HS_DIV2 (24 MHz): Half-rate crystal option balancing precision and timeout range. Extends maximum timeout duration while maintaining crystal accuracy advantages.

11.1.3 Two-Stage Timeout Mechanism

The watchdog implements a graduated fault response architecture with two independent threshold comparators. The first stage (INTVAL) triggers an interrupt when the counter reaches the programmed interrupt threshold, providing early warning of potential timeout. The interrupt service routine can inspect system state, attempt recovery actions, and optionally restart the watchdog to prevent reset escalation. If the ISR successfully resolves the fault and restarts the watchdog, normal operation resumes without system disruption. The second stage (RESVAL) represents the terminal timeout threshold. If the counter reaches RESVAL without intervening restart (indicating either failed recovery or complete firmware lockup preventing ISR execution), the watchdog asserts a hardware reset signal to the reset generator, forcing system-wide reset. This dual-threshold approach maximizes diagnostic capability while maintaining guaranteed fail-safe behavior.

11.1.4 Configuration Locking

The watchdog lock mechanism provides security against unauthorized configuration modification after initial setup. Writing the key value 0x3A to the LOCK register permanently locks the watchdog

configuration, enable state, and lock register itself until the next power-on or external reset event. Once locked, software cannot disable the watchdog, modify timeout values, or change clock sources, ensuring continuous supervision regardless of firmware behavior. Lock protection is essential for safety-critical systems requiring proven-in-use watchdog operation and applications vulnerable to malicious firmware attacks attempting to disable supervision. The lock state persists across warm resets but clears on power-on reset, allowing reconfiguration during system initialization. Applications must verify configuration correctness before engaging lock, as any error becomes uncorrectable until hardware reset.

11.1.5 Interrupt Generation and Servicing

When the watchdog counter reaches the INTVAL threshold and interrupt generation is enabled (both CFG INTEN and WDTIEREN bits set), the watchdog sets the WDTIERSTAT flag and asserts an interrupt request to the NVIC. The interrupt service routine reads WDTIERSTAT to confirm the interrupt source, captures diagnostic context, and determines appropriate action. If recovery is feasible, the ISR calls `am_hal_wdt_restart()` to reset the counter, preventing further escalation. The ISR must then write to WDTIERCLR to acknowledge and clear the interrupt status. If the ISR does not restart the watchdog (either due to explicit decision that recovery is infeasible or because the ISR itself fails to execute due to severe system corruption), the counter continues incrementing toward RESVAL. When RESVAL is reached, the hardware reset path activates regardless of interrupt status, providing ultimate fail-safe behavior independent of software state.

11.1.6 Reset Generation

Watchdog reset generation requires coordination between the WDT peripheral and the reset generator (RSTGEN) module. When the counter reaches RESVAL and reset is enabled (CFG RESEN = 1), the watchdog asserts a reset request signal to the RSTGEN peripheral. The RSTGEN module gates this request through its own WDREN enable bit, which must also be set to 1 for the reset to propagate. This two-level enable architecture provides defense against accidental reset generation and allows system-level control over watchdog reset behavior. When both enables are active and RESVAL expires, RSTGEN generates a system-wide reset equivalent to power-on reset, clearing all registers (except those in always-on domains), reinitializing peripherals, and restarting CPU execution from the reset vector. The HAL `am_hal_wdt_config()` function automatically configures both enable bits based on the `bResetEnable` parameter, ensuring consistent configuration.

11.2 Design Considerations

11.2.1 Timeout Period Selection

Watchdog timeout configuration requires balancing system responsiveness against false-positive reset events. The two-stage timeout architecture provides defensive depth: INTVAL triggers an interrupt allowing software recovery attempts, while RESVAL initiates hard system reset if software fails to respond. Select timeout values accounting for worst-case task execution latency, including interrupt service time, scheduler delays, and any blocking operations. Low-frequency clock sources (LFRC with divisors) provide long timeout ranges (seconds to minutes) suitable for coarse system monitoring with minimal power consumption. High-frequency sources (XTAL_HS) enable fine-grained timeouts (milliseconds) for real-time responsiveness but increase power draw and require more frequent watchdog servicing.

11.2.2 Clock Source Stability

Watchdog reliability depends critically on clock source stability across operational conditions. The internal LFRC oscillator exhibits frequency variation with temperature and voltage (typically $\pm 2\text{-}5\%$), causing actual timeout periods to deviate from nominal calculations. Applications requiring precise timeout accuracy should utilize crystal-based sources (XTAL_HS with divisors) which provide superior stability (± 50 ppm) at the cost of increased power consumption and external component requirements. For LFRC-based configurations, calculate timeout margins accounting for worst-case frequency drift across the specified operating range. In battery-powered applications, verify watchdog behavior during brownout conditions where clock frequency may shift before power supervision circuits trigger.

11.2.3 Configuration Lock Considerations

The watchdog lock mechanism provides irreversible configuration protection until the next system reset, preventing malicious or erroneous software from disabling watchdog supervision. Once locked, the watchdog configuration, enable state, and timeout values become read-only, and the lock itself cannot be cleared without power-on or external reset. Applications implementing watchdog locking must ensure initial configuration is correct and complete before invoking `am_hal_wdt_start()` with `bLock=true`, as any configuration error becomes uncorrectable until reset. Lock protection is essential for safety-critical applications, functional safety compliance (IEC 61508, ISO 26262), and systems requiring tamper resistance against firmware-level attacks.

11.2.4 Interrupt vs. Reset Strategy

The dual-threshold watchdog architecture enables graduated fault response. Configure $\text{INTVAL} < \text{RESVAL}$ to trigger an interrupt before the final reset deadline, providing opportunity for diagnostic logging, safe-state shutdown, or recovery attempts. The interrupt service routine should capture fault context (task state, register dumps, stack trace) and attempt corrective action if feasible. If the ISR successfully resolves the fault condition, it can service the watchdog via `am_hal_wdt_restart()` to prevent escalation to reset. If recovery fails or the ISR does not execute (indicating severe system corruption), the RESVAL threshold expires and forces hardware reset. This strategy maximizes diagnostic visibility while maintaining ultimate fail-safe behavior. Systems prioritizing rapid recovery may set $\text{INTVAL} = \text{RESVAL}$ to bypass interrupt stage and proceed directly to reset.

11.2.5 Power Mode Watchdog Behavior

Watchdog counter progression depends on the selected clock source remaining active across power modes. LFRC-based watchdog configurations continue operating during deep sleep, as LFRC remains enabled for RTC and low-power timer functions. XTAL_HS-based configurations halt during deep sleep if the crystal is disabled, effectively pausing watchdog supervision. Applications requiring watchdog monitoring during sleep states must utilize LFRC clock sources and account for the increased timeout variability. During wakeup transitions, verify watchdog servicing occurs before the accumulated sleep

duration exceeds the configured timeout. For ultra-low-power applications alternating between active bursts and extended sleep, consider dynamically reconfiguring watchdog timeouts before entering sleep or disabling the watchdog entirely during sleep phases if the application architecture permits.

11.3 Programming Guidelines

11.3.1 Configuring Watchdog Operation

Watchdog initialization follows this configuration sequence:

1. Define `am_hal_wdt_config_t` structure with desired settings.
2. Select clock source via `eClockSource` field (LFRC_DIV8/64/1K/16K, XTAL_HS/DIV2).
3. Calculate `ui32InterruptValue` in clock ticks for interrupt threshold.
4. Calculate `ui32ResetValue` in clock ticks for reset threshold (must exceed INTVAL).
5. Set `bInterruptEnable` = true to enable interrupt generation at INTVAL.
6. Set `bResetEnable` = true to enable system reset at RESVAL.
7. Call `am_hal_wdt_config(AM_HAL_WDT_MCU, &config)` to apply settings.
8. Enable watchdog interrupt in NVIC if using interrupt threshold.
9. Call `am_hal_wdt_start(AM_HAL_WDT_MCU, false)` to enable watchdog without lock.
10. Optionally call `am_hal_wdt_start(AM_HAL_WDT_MCU, true)` to enable with lock protection.

NOTE

Note: Configuration must be complete and verified before enabling lock, as locked configuration cannot be modified until system reset.

11.3.2 Calculating Timeout Values

Timeout values are specified in clock ticks and calculated as:

$\text{Timeout\_Ticks} = \text{Desired\_Timeout\_Seconds} \times \text{Clock\_Frequency}$

Example clock frequencies:

- LFRC_DIV8: 1024 Hz / 8 = 128 Hz
- LFRC_DIV64: 1024 Hz / 64 = 16 Hz
- LFRC_DIV1K: 1024 Hz / 1024 = 1 Hz
- LFRC_DIV16K: 1024 Hz / 16384 = 0.0625 Hz (16 second period)
- XTAL_HS: 48 MHz (requires external crystal)
- XTAL_HS_DIV2: 48 MHz / 2 = 24 MHz

For a 10-second timeout using LFRC_DIV64 (16 Hz): $\text{ui32ResetValue} = 10 \times 16 = 160$ ticks. For a 5-second interrupt warning: $\text{ui32InterruptValue} = 5 \times 16 = 80$ ticks.

11.3.3 Servicing the Watchdog

To prevent watchdog timeout, software must periodically restart the timer:

1. Call `am_hal_wdt_restart(AM_HAL_WDT_MCU)` before INTVAL expires.
2. Restart operation writes key value to RSTRT register.
3. Watchdog counter resets to 0, beginning new timeout period.
4. Service frequency must accommodate worst-case execution paths.
5. Common practice: service at 50-75% of timeout period to provide margin.

Example: For 10-second timeout, service watchdog every 5-7 seconds. Place watchdog restart calls in main loop or periodic timer ISR. Avoid servicing within individual interrupt handlers to maintain supervision of main code execution flow.

11.3.4 Interrupt Handling

Watchdog interrupt servicing procedure:

1. Watchdog asserts interrupt when COUNT reaches ui32InterruptValue.
2. ISR reads status via `am_hal_wdt_interrupt_status_get()`.
3. Capture diagnostic information (task state, fault context).
4. Attempt recovery if fault condition is correctable.
5. Call `am_hal_wdt_restart()` to clear counter if recovery successful.
6. Clear interrupt via `am_hal_wdt_interrupt_clear()`.
7. If ISR does not restart watchdog, RESVAL expires and forces reset.

The interrupt stage provides early warning before terminal reset, enabling diagnostic data collection and graceful degradation strategies.

11.3.5 Operating Constraints

When utilizing the Watchdog Timer peripheral, observe the following constraints:

- RESVAL must be greater than INTVAL when both interrupt and reset are enabled.
- Configuration lock is irreversible until system reset; verify settings before locking.
- LFRC-based timeout accuracy varies $\pm 2\text{-}5\%$ with temperature and voltage; design margins accordingly.
- Watchdog servicing must occur before INTVAL expires to prevent interrupt assertion.
- XTAL_HS clock sources may not remain active during deep sleep modes.
- RSTRT register requires key value (0xB2) for restart operation; direct writes are ignored.
- LOCK register requires key value (0x3A) to engage lock protection.
- Watchdog interrupt ISR must complete and restart watchdog before RESVAL expires.

11.4 Register Summary

For detailed register descriptions, refer to the WDT register section in the Apollo510 SoC register set documentation included with the AmbiqSuite SDK.

Key Registers:

- **CFG - Watchdog Configuration Register**
 - WDTEN - Watchdog timer enable/disable control bit
 - INTEN - Interrupt enable at INTVAL threshold
 - RESEN - Reset enable at RESVAL threshold
 - INTVAL - Interrupt threshold value in clock ticks (8-bit field)
 - RESVAL - Reset threshold value in clock ticks (8-bit field)
 - CLKSEL - Clock source selection (LFRC divisors, XTAL_HS options)
- **RSTRT - Watchdog Restart Register (write-only)**
 - Watchdog counter restart/service control
 - Write key value 0xB2 to reset counter to 0
 - Any other value written is ignored (lock-out protection)
 - Resets both interrupt and reset timers simultaneously
- **LOCK - Watchdog Lock Register (write-once)**
 - Configuration lock protection control
 - Write key value 0x3A to permanently lock configuration
 - Locks CFG register, WDTEN bit, and LOCK itself until reset
 - Prevents malicious or erroneous disable of watchdog supervision
- **COUNT - Watchdog Counter Register (read-only)**
 - Current watchdog counter value in clock ticks
 - Increments at selected clock rate from 0 toward threshold
 - Read via `am_hal_wdt_read()` for diagnostic monitoring
 - Resets to 0 on RSTRT write or system reset
- **WDTIEREN - Watchdog Interrupt Enable Register**
 - Enable/disable watchdog interrupt generation
 - Bit 0: Watchdog interrupt enable
 - Set to 1 to enable interrupt at INTVAL threshold
 - Independent of CFG INTEN bit; both must be set for interrupt operation
- **WDTIERSTAT - Watchdog Interrupt Status Register (read-only)**
 - Raw watchdog interrupt status flag
 - Bit 0: Set when COUNT reaches INTVAL
 - Remains asserted until cleared via WDTIERCLR
 - Read by ISR to confirm watchdog interrupt source
- **WDTIERCLR - Watchdog Interrupt Clear Register (write-only)**
 - Write-one-to-clear interrupt status control
 - Write 1 to bit 0 to clear WDTIERSTAT flag
 - Write 0 has no effect on interrupt status
 - Must be written by ISR to acknowledge and clear interrupt
- **WDTIERSET - Watchdog Interrupt Set Register (write-only)**
 - Software interrupt generation for testing
 - Write 1 to bit 0 to manually assert interrupt flag
 - Useful for validating ISR functionality without waiting for timeout
 - Cleared normally via WDTIERCLR
- **RSTGEN_CFG_WDREN - Reset Generator Watchdog Reset Enable**
 - Global watchdog reset enable in reset generator peripheral
 - Located in RSTGEN module, not WDT register space
 - Must be set to 1 for watchdog to generate system reset
 - Configured automatically by `am_hal_wdt_config()` when `bResetEnable = true`

12. General Purpose Input/Output (GPIO)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

12.1 General Purpose I/O (GPIO) Functions

The following sub-sections refer to function selections offered on the various GPIO pads. Please consult the device-specific Pin Mapping table in the GPIO section of the device's datasheet for GPIO available on each SoC package and specific function mapping for each GPIO.

For each GPIO pad, if the GPIO_PINCFGn_FNCSEL0n field (n = GPIO number) is set to 0x3 the pad is connected to the corresponding GPIO signal. This section describes the configuration options specific to GPIO operation.

12.1.1 Configuring the GPIO Functions

Each GPIO must be configured in the GPIO_PINCFGn Register as an input and/or output before using. Note that the PADKEY Register must be set to the value 0x73 in order to write the GPIO_PINCFGn Registers. Each output may be configured as push-pull, open drain, disabled, or tri-stated as selected by the GPIO_PINCFGn_OUTCFGn field. If the output is configured as push-pull, the pad will be driven with the corresponding bit in the GPIO_WTn (n = 0-6) Register. If the output is configured as open drain, the pad will be pulled low if the corresponding bit in the WTn Register is a 0, and will be floating if the corresponding bit in the WTn Register is a 1. If the output is configured as tri-state, the pad will be driven with the corresponding bit in the WTn Register if the corresponding bit in the GPIO_ENn Register is a 1. If the bit in ENn is a 0, the output will be floating.

If the FNCSELn field is set to "NCEn" (n = GPIO number), the pin functions as the chip enable for the communication channel specified by the NCESRCn field in that PINCFGn Register.

12.1.2 Reading from a GPIO Pad

All GPIO inputs are readable at all times provided that the INPENn bit is enabled and the RDZEROn bit is disabled, even if the pad is not configured as a GPIO. The current values of pads are read in the GPIO_RDn Registers as follows:

- Pads 0 to 31 are read in GPIO_RD0
- Pads 32 to 63 are read in GPIO_RD1
- Pads 64 to 95 are read in GPIO_RD2
- Pads 96 to 127 are read in GPIO_RD3
- Pads 128 to 159 are read in GPIO_RD4
- Pads 160 to 191 are read in GPIO_RD5
- Pads 192 to 223 are read in GPIO_RD6

12.1.3 Writing to a GPIO Pad

The GPIO pad outputs are controlled by the GPIO_WT0-WT6 Registers and the GPIO_EN0-EN6 Registers. Each of these registers may be directly written and read. Because each GPIO is often an independent function, the capability also exists to set or clear one or more bits without having to perform a read-modify-write operation. If the GPIO_WTS0-WTS6 Register is written, the corresponding bit in WT0-WT6 will be set if the write data is 1, otherwise the WT0-WT6 bit will not be changed. If the GPIO_WTC0-WTC6 Register is written, the corresponding bit in WT0-WT6 will be cleared if the write data is 1, otherwise the WT0-WT6 bit will not be changed.

If a GPIO pad is configured for tri-state output mode, the EN0-EN6 Register controls the enabling of each bit. These registers may be directly written, and individual bits may be set or cleared by writing the ENS0-ENS6 or ENC0-ENC6 Registers with a 1 in the desired bit position.

12.1.4 GPIO Interrupts

Each GPIO pad can be configured to generate an interrupt on a high-to-low transition, a low-to-high transition, or either as selected by setting the GPIO_PINCFGn_IRPTENn bit. This interrupt will be generated even if the pad is not configured as a GPIO in the Pad Configuration logic.

Table 13 below describes the interrupt trigger options.

Table 13: Interrupt Trigger Options

IRPTENn	Interrupt
00	Disabled
01	High -> low transition
10	Low -> high transition
11	Either low -> high or high -> low transition

Each interrupt is enabled, disabled, cleared or set with a standard set of interrupt registers GPIO_MCUN0INTnEN, GPIO_MCUN0INTnSTAT, GPIO_MCUN0INTnCLR and GPIO_MCUN0INTnSET, where n = 0 to 6 for GPIO pads 0 to 31, 32 to 63, 64 to 95 and 96 to 127, 128 to 159, 160 to 191 and 192 to 223, respectively. The N0 designation in these registers indicates that these are in interrupt register set 0. A duplicate set of registers, with a designation N1, is available and these registers are similarly named GPIO_MCUN1INTnEN, GPIO_MCUN1INTnSTAT, GPIO_MCUN1INTnCLR and GPIO_MCUN1INTnSET.

The purpose of this dual set of registers is to enable the segregation of an interrupt or a small set of interrupts which may have higher importance and thus should be handled at a higher priority level and/or with minimal latency. If only one interrupt is enabled in one of these sets, then there is no need to determine which GPIO caused the interrupt in this bank by reading and processing the Status Register bits. Note that these interrupts get mapped to different IRQs and hence respective interrupts need to be enabled in NVIC and serviced accordingly.

12.2 Pad Connection Details

The following sub-sections describe in more detail GPIO connection options.

12.2.1 Output Selection

There is a multiplexer which selects the module signal to be driven to the output based on the GPIO_PINCFGn_FNCSELn (n = GPIO number) field. This multiplexer implements the selection of a GPIO's function shown in the device-specific Pin Mapping Table for output pads. For all pads, a FNCSELn value of 0x3 selects the value in the corresponding GPIO_WTn Register bit.

Certain functional groups, Timer and MSPI in particular, have additional pre-muxing configuration as noted.

12.2.2 Output Control

The driver for each pad has a data input and an output enable input. Each of these controls is selected from among several alternatives based on the OUTCFG register setting which is set based on the output type.

OUTDATSEL normally selects the data from the output multiplexer, but if the pad is configured as Open Drain the data input is selected to be low. If a pad is configured to be Open Drain, the pad enable is driven with the data from the output multiplexer. If the pad is configured as a GPIO (PADnFNCSEL = 0x3) and the GPIO drive type is tri-state (OUTCFGn = 0x3), the pad enable is driven with the inverse of the corresponding GPIO bit in the GPIO_ENn Register. If the pad is not configured as an output, the pad enable is forced high to turn the driver off.

The drive strength of each pad driver is configured as described in Section 12.1 on page 113.

12.2.3 Input Control

The input circuitry of the pad may be disabled by clearing the PINCFGn_INPENn bit. This configuration should always be set if the pad input is not being used, as it prevents unnecessary current consumption if the pad voltage happens to float to a level between VDD and Ground. If PADnINPEN is 0, the pad will always read as a 0.

If INPENn is set, the pad input then goes to two places. It is driven to the selected module signal as selected in the device-specific Pin Mapping Table. In addition, the pad input can always be read from the GPIO_RDn Register unless the pad is configured as a GPIO (FNCSELn = 0x3) and RDZEROn is high, which will force the GPIO_RD input to be a zero. The ability to always read the pad value is very useful in some diagnostic cases.

The pad input is always sent to the GPIO interrupt logic, and a pad transition in the direction selected by IRPTENn will set the MCUNnINTmSTAT flip-flop when the corresponding MCUNnINTmEN bit is set. Note that this interrupt will be set even if the pad is not configured as a GPIO, which may be useful in detecting functions. As an example, this could be used to generate an interrupt when the I²C/SPI Subordinate nCE signal is driven low by the Interface Host.

12.2.4 Pull-up/Pull-down Control

If PULLCFGn is set to a pull-up or pull-down value, an internal pull-up/pull-down resistor is connected between the pad and VDDHn/VSS.

12.2.5 Analog Pad Configuration

If a pad is configured in analog mode (reference the analog input function selections in the device-specific Pin Mapping Table), the pad is connected directly to the particular analog module signal. In addition, the pad's setting of output (GPIO_PINCFGn_OUTCFGn) or input enable (GPIO_PINCFGn_INPENn) is forced low to disable the pad output, and the input of the pad is disabled independent of the value of INPENn.

12.3 Module-specific Pad Configuration

The following sections describe in detail how to configure the pads for each module function. **Reference the device- and package-specific Pin Mapping Table for pads which are available and can be used for the signals described in the following sub-sections.**

12.3.1 Implementing IO Manager Connections

The eight IO Manager modules must be correctly connected to the appropriate pads in order to operate. Reference the device-specific Pin Mapping Table for pads which can be used for the various signals for each IO Manager instance: MnSCK, MnMOSI and MnMISO for SPI operation, and MnSCL and MnSDAWIR3 for I²C operation, where n = instance number 0 - 7. Any GPIO pad brought out to a pin on the package can be configured as the SPI Chip enable for any of the eight IOM channels. If the FNCSELn field is set to “NCEn” (n = GPIO number), the pin functions as the chip enable for the communication channel specified by the NCESRCn field in that PINCFGn Register.

12.3.1.1 IO Manager I²C Connection

I²C mode of IO Manager 0 uses pad 5 as SCL and pad 6 as SDA. This mode is configured by setting the FNCSELn fields as shown in Table 14. If the internal I²C pullup resistors are to be used, PULLCFG5 and PULLCFG6 should be set to select the desired pullup resistor. If external pullup resistors are used, PULLCFG5 and PULLCFG6 should be cleared.

Table 14: IO Manager 0 I²C Configuration

Field	Value
FNCSEL5	0
FNCSEL6	0

This same setup for I²C mode is done for each of the other seven instances of the IO Manager as needed. Reference the device-specific Pin Mapping Table for pads which are used for the MnSCL (SCL) and MnSDAWIR3 (SDA) signals for each IO Manager instance.

12.3.1.2 IO Manager 4-wire SPI Connection

Four-wire SPI mode of IO Manager 0 uses pad 5 as SCK, pad 6 as MOSI and pad 7 as MISO. This mode is configured by setting the FNCSELn fields as shown in Table 15. PULLCFG5, PULLCFG6 and PULLCFG7 should be cleared. Any GPIO can be configured as the CE for any of the Manager IO instances. This same setup for 4-wire SPI mode is done for each of the other seven instances of the IO Manager as needed. Reference the device-specific Pin Mapping Table for pads which are used for the MnSCK, MnMOSI and MnMISO signals for each IO Manager instance.

It should also be noted that in 4-Wire mode, the MnMOSI pin should be configured with the FOENn bit set to force output enable active on the pin.

Table 15: IO Manager 0 4-wire SPI Configuration

Field	Value
FNCSEL5	1
FNCSEL6	1
FNCSEL7	0

12.3.1.3 IO Manager 0 3-wire SPI Connection

Three-wire SPI mode of IO Manager 0 uses pad 5 as SCK and pad 6 as MOSI/MISO. This mode is configured by setting the FNCSELn fields as shown in Table 16. PULLCFG5 and PULLCFG6 should be cleared. Pad 7 may be used for other functions.

Table 16: IO Manager 0 3-wire SPI Configuration

Field	Value
FNCSEL5	1
FNCSEL6	1

A variety of pads may be used for up to four nCE signals to select up to four separate subordinates. The INPENn and PULLCFGn bits of any pad used for nCE should be cleared.

Any GPIO pad brought out to a pin can be configured as the CE for any of the Manager IO instances.

This same setup for 3-wire SPI mode is done for each of the other seven instances of the IO Manager as needed. Reference the device-specific Pin Mapping Table for pads which are used for the MnSCK (SCK) and MnSDAWIR3 (MOSI/MISO) signals for each IO Manager instance.

12.3.1.4 SPI Flow Control Connections

SPI Flow Control in interrupt mode requires an external pin to be specified as the interrupt pin. This is accomplished by configuring for the desired pin in the IOMxIRQ Register (x = 0 to 7).

12.3.2 Implementing IO Subordinate Connections

The half-duplex IO Subordinate module offered only on the Apollo510 SoC must be correctly connected to the appropriate pads in order to operate.

12.3.2.1 IO Subordinate I²C Connection

I²C mode of the IO Subordinate uses pad 11 as SCL and pad 52 as SDA. This mode is configured by setting the FNCSELn fields as shown in Table 17. The INPEN11 and INPEN52 bits must be set. PULLCFG11 and PULLCFG52 should be cleared.

Table 17: IO Subordinate I²C Configuration

Field	Value
FNCSEL11	9
FNCSEL52	12

12.3.2.2 IO Subordinate 4-wire SPI Connection

Four-wire SPI mode of the IO Subordinate uses pad 11 as SCK, pad 52 as MOSI, pad 83 as MISO and pad 13 as nCE. This mode is configured by setting the FNCSELn fields as shown in Table 18. The

INPEN11, INPEN52 and INPEN13 bits must be set. PULLCFG11, PULLCFG52, PULLCFG83 and PULLCFG13 should be cleared.

Table 18: IO Subordinate 4-wire SPI Configuration

Field	Value
FNCSEL11	10
FNCSEL52	13
FNCSEL83	10
FNCSEL13	9

12.3.2.3 IO Subordinate 3-wire SPI Connection

Three-wire SPI mode of the IO Subordinate uses pad 11 as SCK, pad 52 as MISO/MOSI and pad 13 as nCE. This mode is configured by setting the FNCSELn fields as shown in Table 19. The INPEN11, INPEN52 and INPEN13 bits must be set. PULLCFG11, PULLCFG52 and PULLCFG13 should be cleared. Pad 83 may be used for other functions.

Table 19: IO Subordinate 3-wire SPI Configuration

Field	Value
FNCSEL11	10
FNCSEL52	12
FNCSEL13	9

12.3.2.4 IO Subordinate Interrupt Connection

The IO Subordinate can be configured to generate an interrupt output under a variety of internal conditions. If this function is used, the interrupt will be generated on pad 4. FNCSEL1 must be set to 1, and INPEN4 and PULLCFG4 should be cleared.

12.3.3 Implementing Full-Duplex IO Subordinate Connections

The Full-Duplex IO Subordinate module offered on all SoCs except for Apollo510 SoC must be correctly connected to the appropriate pads in order to operate.

12.3.3.1 Full Duplex IO Subordinate 4-wire SPI Connection

Four-wire SPI mode of the Full-Duplex IO Subordinate uses pad 0 as SCK, pad 1 as MOSI, pad 2 as MISO and pad 3 as nCE for IOSFD0. This mode is configured by setting the FNCSELn fields as shown in Table 18. The INPEN0, INPEN1 and INPEN3 bits must be set. PULLCFG0, PULLCFG1, PULLCFG2 and PULLCFG3 should be cleared.

Table 20: IO Subordinate 4-wire SPI Configuration

Field	IOSFD0 Value
FNCSEL0	1
FNCSEL1	1
FNCSEL2	1
FNCSEL3	1

12.3.3.2 Full Duplex IO Subordinate 3-wire SPI Connection

Three-wire SPI mode of the Full Duplex IO Subordinate uses pad 0 as SCK, pad 1 as MISO/MOSI and pad 3 as nCE. This mode is configured by setting the FNCSELn fields as shown in Table 19. The INPEN0, INPEN1 and INPEN3 bits must be set. PULLCFG0, PULLCFG1 and PULLCFG3 should be cleared. Pad 2 may be used for other functions.

Table 21: IO Subordinate 3-wire SPI Configuration

Field	Value
FNCSEL0	1
FNCSEL1	2
FNCSEL3	1

12.3.3.3 Full Duplex IO Subordinate Interrupt Connection

The Full Duplex IO Subordinate can be configured to generate an interrupt output under a variety of internal conditions. If this function is used, the interrupt will be generated on pad 4. FNCSEL1 must be set to 1, and INPEN4 and PULLCFG4 should be cleared.

12.3.4 Implementing Display Controller Connections

The Display Controller (DC) supports connection to displays via multiple interfaces. One set of pads is available for each of these interfaces, and these pads need to be correctly configured for proper display control. Reference the device-specific Pin Mapping Table for pads which can be used for the DC signals.

The functions of the DC signals are as follows, grouped by interface. INPENn and PULLCFGn should be cleared for output signals. For input and bi-directional signals, INPENn and PULLCFGn should be set.

See Display Controller chapter for description of signals.

SPI

- DISP_SPI_SCK
- DISP_SPI_SD
- DISP_SPI_DCX
- DISP_SPI_SDO
- DISP_SPI_SDI
- DISP_SPI_RST
- DC_SPI_CS_N*

*Selectable from any GPIO pad brought out to a pin to be configured as the CE for the DC SPI or QSPI interface

Table 22: DISP_SPI Interface Configuration

Field	Value	Signal	Direction	Pad
FNCSEL81	9	DISP_SPI_SCK	Output	81
FNCSEL79	9	DISP_SPI_SD	Bi-directional	79
FNCSEL80	9	DISP_SPI_DCX	Output	80
FNCSEL79	10	DISP_SPI_SDO	Output	79
FNCSEL82	9	DISP_SPI_SDI	Input	82
FNCSEL83	9	DISP_SPI_RST	Output	83
FNCSEL100	4	DISP_SPI_SCK	Output	100
FNCSEL101	4	DISP_SPI_SD	Bi-directional	101
FNCSEL102	4	DISP_SPI_DCX	Output	102
FNCSEL100	5	DISP_SPI_SDO	Output	100
FNCSEL103	4	DISP_SPI_SDI	Input	103
FNCSEL104	4	DISP_SPI_RST	Output	104

Quad SPI

- DISP_QSPI_D3 - DISP_QSPI_D0
- DISP_QSPI_SCK
- DC_QSPI_CS_N*

*Selectable from any GPIO pad brought out to a pin to be configured as the CE for the DC SPI or QSPI interface

Table 23: DISP_QSPI Interface Configuration

Field	Value	Signal	Direction	Pad
FNCSEL79	2	DISP_QSPI_D0	Bi-directional	79
FNCSEL80	2	DISP_QSPI_D1	Bi-directional	80
FNCSEL82	2	DISP_QSPI_D2	Bi-directional	82
FNCSEL83	2	DISP_QSPI_D3	Bi-directional	83
FNCSEL81	2	DISP_QSPI_SCK	Output	81

Serial MIP Interface

- DISP_ENB
- DISP_XRST
- DISP_R1
- DISP_R2
- DISP_G1
- DISP_G2
- DISP_B1

- DISP_B2
- DISP_HST
- DISP_VST
- DISP_HCK
- DISP_VCK

Table 24: MIP Display Interface Configuration

Field	Value	Signal	Direction	Pad
FNCSEL209	0	DISP_ENB	Output	209
FNCSEL210	0	DISP_XRST	Output	210
FNCSEL211	0	DISP_R1	Output	211
FNCSEL212	0	DISP_R2	Output	212
FNCSEL213	0	DISP_G1	Output	213
FNCSEL214	0	DISP_G2	Output	214
FNCSEL215	0	DISP_B1	Output	215
FNCSEL216	0	DISP_B2	Output	216
FNCSEL217	0	DISP_HST	Output	217
FNCSEL218	0	DISP_VST	Output	218
FNCSEL219	0	DISP_HCK	Output	219
FNCSEL220	0	DISP_VCK	Output	220

Parallel DPI-2

- DISP_D23 - DISP_D0
- DISP_VS
- DISP_HS
- DISP_DE
- DISP_PCLK
- DISP_SD
- DISP_CM
- DISP_TE

Table 25: DISP_DPI-2 Interface Configuration

Field	Value	Signal	Direction	Pad
FNCSEL220	1	DISP_D0	Bi-directional	220
FNCSEL209	1	DISP_D1	Bi-directional	209
FNCSEL221	1	DISP_D2	Bi-directional	221
FNCSEL219	1	DISP_D3	Bi-directional	219
FNCSEL210	1	DISP_D4	Bi-directional	210
FNCSEL148	1	DISP_D5	Bi-directional	148

Table 25: DISP_DPI-2 Interface Configuration

Field	Value	Signal	Direction	Pad
FNCSEL149	1	DISP_D6	Bi-directional	149
FNCSEL150	1	DISP_D7	Bi-directional	150
FNCSEL151	1	DISP_D8	Bi-directional	151
FNCSEL152	1	DISP_D9	Bi-directional	152
FNCSEL153	1	DISP_D10	Bi-directional	153
FNCSEL154	1	DISP_D11	Bi-directional	154
FNCSEL155	1	DISP_D12	Bi-directional	155
FNCSEL222	1	DISP_D13	Bi-directional	222
FNCSEL223	1	DISP_D14	Bi-directional	223
FNCSEL136	1	DISP_D15	Bi-directional	136
FNCSEL140	1	DISP_D16	Bi-directional	140
FNCSEL141	1	DISP_D17	Bi-directional	141
FNCSEL216	1	DISP_D18	Bi-directional	216
FNCSEL215	1	DISP_D19	Bi-directional	215
FNCSEL214	1	DISP_D20	Bi-directional	214
FNCSEL213	1	DISP_D21	Bi-directional	213
FNCSEL212	1	DISP_D22	Bi-directional	212
FNCSEL211	1	DISP_D23	Bi-directional	211
FNCSEL218	1	DISP_VS	Output	218
FNCSEL217	1	DISP_HS	Output	217
FNCSEL187	1	DISP_DE	Output	187
FNCSEL189	1	DISP_PCLK	Output	189
FNCSEL186	1	DISP_SD	Output	186
FNCSEL188	1	DISP_CM	Output	188
FNCSEL10	9	DISP_TE	Output	10
FNCSEL33	9	DISP_TE	Output	33
FNCSEL50	9	DISP_TE	Output	50
FNCSEL63	9	DISP_TE	Output	63
FNCSEL87	9	DISP_TE	Output	87

NOTE

Use of the DPI-2 interface, which includes pad functions DISP_D0 - DISP_D23, DISP_VS, DISP_HS, DISP_DE, DISP_PCLK, DISP_SD and DISP_CM, is not recommended or supported.

Parallel DBIB

- DBIB_D15 - DBIB_D0
- DBIB_CSX
- DBIB_DCX
- DBIB_WRX
- DBIB_RDX

Table 26: DISP_DBIB Interface Configuration

Field	Value	Signal	Direction	Pad
FNCSEL213	2	DBIB_D0	Bi-directional	213
FNCSEL214	2	DBIB_D1	Bi-directional	214
FNCSEL215	2	DBIB_D2	Bi-directional	215
FNCSEL216	2	DBIB_D3	Bi-directional	216
FNCSEL217	2	DBIB_D4	Bi-directional	217
FNCSEL218	2	DBIB_D5	Bi-directional	218
FNCSEL219	2	DBIB_D6	Bi-directional	219
FNCSEL220	2	DBIB_D7	Bi-directional	220
FNCSEL148	2	DBIB_D8	Bi-directional	148
FNCSEL149	2	DBIB_D9	Bi-directional	149
FNCSEL150	2	DBIB_D10	Bi-directional	150
FNCSEL151	2	DBIB_D11	Bi-directional	151
FNCSEL152	2	DBIB_D12	Bi-directional	152
FNCSEL153	2	DBIB_D13	Bi-directional	153
FNCSEL154	2	DBIB_D14	Bi-directional	154
FNCSEL155	2	DBIB_D15	Bi-directional	155
FNCSEL209	2	DBIB_CSX	Output	209
FNCSEL210	2	DBIB_DCX	Output	210
FNCSEL211	2	DBIB_WRX	Output	211
FNCSEL212	2	DBIB_RDX	Output	212

12.3.5 Implementing Counter/Timer Connections

Each Counter/Timer can optionally count pulses from an input pad, or generate pulses on an output pad. A pad's FNCSELn field is set to CTn (0x6) to connect a Counter/Timer to that pad. If the pad is used as an input, the INPENn bit should be set, otherwise it should be cleared. The PULLCFGn bit may be set if the input signal is open drain.

12.3.6 Implementing UART Connections

The UART signals can be connected to a variety of pads.

12.3.6.1 UART0 TX/RX Connections

The UART0 data signals TX and RX may each be connected to several pads. Note that TX and RX are selected independently. Table 27 shows the connections for TX, which should have the corresponding INPENn and PULLCFGn fields clear. Table 28 shows the connections for RX, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 27: UART0 TX Configuration

Pad	FNCSEL	Apollo510	Apollo510B
0	4	✓	✓
12	4	✓	✓
20	4	✓	✓
30	4	✓	
34	4	✓	✓
41	4	✓	✓
45	4	✓	✓
53	4	✓	
60	4	✓	
66	4	✓	✓
72	4	✓	✓
78	4	✓	

Table 28: UART0 RX Configuration

Pad	FNCSEL	Apollo510	Apollo510B
2	4	✓	✓
21	4	✓	✓
32	4	✓	✓
36	4	✓	✓
43	4	✓	✓
47	4	✓	✓
55	4	✓	
62	4	✓	
68	4	✓	✓
74	4	✓	

12.3.6.2 UART0 RTS/CTS Connections

The UART modem control signals RTS and CTS may each be connected to one of two pads. Note that RTS and CTS are selected independently. Table 29 shows the connections for RTS, which should have the corresponding INPENn and PULLCFGn fields clear. Table 30 shows the connections for CTS, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 29: UART0 RTS Configuration

Pad	FNCSEL	Apollo510	Apollo510B
4	4	✓	✓
24	4	✓	✓
38	4	✓	✓
49	4	✓	✓
57	4	✓	✓
58	4	✓	
64	4	✓	✓
70	4	✓	✓
76	4	✓	

Table 30: UART0 CTS Configuration

Pad	FNCSEL	Apollo510	Apollo510B
6	4	✓	✓
18	4	✓	✓
40	4	✓	✓
51	4	✓	✓
59	4	✓	
65	4	✓	✓
71	4	✓	✓
77	4	✓	

12.3.6.3 UART1 TX/RX Connections

The UART data signals TX and RX may each be connected to several pads. Note that TX and RX are selected independently. Table 31 shows the connections for TX, which should have the corresponding INPENn and PULLCFGn fields clear. Table 32 shows the connections for RX, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 31: UART1 TX Configuration

Pad	FNCSEL	Apollo510	Apollo510B
0	5	✓	✓
12	5	✓	✓
20	5	✓	✓
25	5	✓	✓
41	5	✓	✓
45	5	✓	✓

Table 31: UART1 TX Configuration

Pad	FNCSEL	Apollo510	Apollo510B
53	5	✓	
66	5	✓	✓
72	5	✓	✓
78	5	✓	

Table 32: UART1 RX Configuration

Pad	FNCSEL	Apollo510	Apollo510B
2	5	✓	✓
14	5	✓	✓
21	5	✓	✓
26	5	✓	✓
36	5	✓	✓
43	5	✓	✓
47	5	✓	✓
55	5	✓	
62	5	✓	
68	5	✓	✓

12.3.6.4 UART1 RTS/CTS Connections

The UART modem control signals RTS and CTS may each be connected to one of two pads. Note that RTS and CTS are selected independently. Table 33 shows the connections for RTS, which should have the corresponding INPENn and PULLCFGn fields clear. Table 34 shows the connections for CTS, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 33: UART1 RTS Configuration

Pad	FNCSEL	Apollo510	Apollo510B
4	5	✓	✓
16	5	✓	✓
24	5	✓	✓
49	5	✓	✓
57	5	✓	✓
70	5	✓	✓
76	5	✓	
79	5	✓	

Table 34: UART1 CTS Configuration

Pad	FNCSEL	Apollo510	Apollo510B
6	5	✓	✓
18	5	✓	✓
27	5	✓	✓
29	4	✓	✓
40	5	✓	✓
51	5	✓	✓
59	5	✓	
65	5	✓	
77	5	✓	

12.3.6.5 UART2 TX/RX Connections

The UART data signals TX and RX may each be connected to several pads. Note that TX and RX are selected independently. Table 35 shows the connections for TX, which should have the corresponding INPENn and PULLCFGn fields clear. Table 36 shows the connections for RX, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 35: UART2 TX Configuration

Pad	FNCSEL	Apollo510	Apollo510B
1	4	✓	✓
13	4	✓	✓
22	4	✓	✓
31	4	✓	✓
35	4	✓	✓
42	4	✓	✓
46	4	✓	✓
54	4	✓	
61	4	✓	
67	4	✓	✓
73	4	✓	✓

Table 36: UART2 RX Configuration

Pad	FNCSEL	Apollo510	Apollo510B
3	4	✓	✓
11	4	✓	✓
23	4	✓	✓
33	4	✓	✓

Table 36: UART2 RX Configuration

Pad	FNCSEL	Apollo510	Apollo510B
34	5	✓	✓
37	4	✓	✓
44	4	✓	✓
48	4	✓	✓
56	4	✓	
63	4	✓	
69	4	✓	✓
75	4	✓	

12.3.6.6 UART2 RTS/CTS Connections

The UART modem control signals RTS and CTS may each be connected to one of two pads. Note that RTS and CTS are selected independently. Table 37 shows the connections for RTS, which should have the corresponding INPENn and PULLCFGn fields clear. Table 38 shows the connections for CTS, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 37: UART2 RTS Configuration

Pad	FNCSEL	Apollo510	Apollo510B
5	4	✓	✓
33	5	✓	✓
38	5	✓	✓
39	4	✓	✓
50	4	✓	✓
81	5	✓	

Table 38: UART2 CTS Configuration

Pad	FNCSEL	Apollo510	Apollo510B
7	4	✓	✓
19	4	✓	✓
28	4	✓	✓
31	5	✓	✓
52	4	✓	
64	5	✓	✓
80	5	✓	

12.3.6.7 UART3 TX/RX Connections

The UART data signals TX and RX may each be connected to several pads. Note that TX and RX are selected independently. Table 39 shows the connections for TX, which should have the corresponding

INPENn and PULLCFGn fields clear. Table 40 shows the connections for RX, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 39: UART3 TX Configuration

Pad	FNCSEL	Apollo510	Apollo510B
1	5	✓	✓
13	5	✓	✓
22	5	✓	✓
35	5	✓	✓
42	5	✓	✓
46	5	✓	✓
54	5	✓	
61	5	✓	
67	5	✓	✓
73	5	✓	✓

Table 40: UART3 RX Configuration

Pad	Value	Apollo510	Apollo510B
3	5	✓	✓
11	5	✓	✓
15	5	✓	
23	5	✓	✓
37	5	✓	✓
44	5	✓	✓
48	5	✓	✓
56	5	✓	
63	5	✓	
69	5	✓	✓
75	5	✓	

12.3.6.8 UART3 RTS/CTS Connections

The UART modem control signals RTS and CTS may each be connected to one of two pads. Note that RTS and CTS are selected independently. Table 41 shows the connections for RTS, which should have the corresponding INPENn and PULLCFGn fields clear. Table 42 shows the connections for CTS, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 41: UART3 RTS Configuration

Pad	FNCSEL	Apollo510	Apollo510B
5	5	✓	✓
17	5	✓	✓
39	5	✓	✓
50	5	✓	✓
58	5	✓	
71	5	✓	✓
83	5	✓	✓

Table 42: UART3 CTS Configuration

Pad	FNCSEL	Apollo510	Apollo510B
7	5	✓	✓
19	5	✓	✓
32	5	✓	✓
52	5	✓	
60	5	✓	
74	5	✓	
82	5	✓	

12.3.7 Implementing Audio Connections

The audio signals of the PDM and I2S modules can be connected to a number of pads as described in the sub-sections below.

12.3.7.1 PDM Connections

PDM CLK and DATA signals for the PDM are connected to separate GPIO pads. Note that CLK and DATA are enabled independently. Table 43 shows the connections for PDM CLK, which should have the corresponding INPENn and PULLCFGn fields clear. Table 44 shows the pins for PDM DATA, which must have the corresponding INPENn field set and the corresponding PULLCFGn field clear.

Table 43: PDM CLK Configuration

PDM Instance	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
PDM0	FNCSEL50	0	50	✓	✓

Table 44: PDM DATA Configuration

PDM Instance	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
PDM0	FNCSEL51	0	51	✓	✓

12.3.7.2 I2S Connections

For each available I2S modules, the BCLK, WCLK and DAT signals may each be connected to one of multiple pads. Note that CLK, WS and DATA are selected independently.

- Table 45 shows the connections for I2S CLK, which should have the corresponding PADnINPEN set and the corresponding PULLCFGn field clear.
- Table 46 shows the connections for I2S WS, which should have the corresponding INPENn set and the PULLCFGn field clear.
- Table 47 shows the connections for I2S DATA, which must have the corresponding INPENn and PULLCFGn fields clear.
- Table 48 shows the connections for I2S SDIN, which must have the corresponding INPENn and PULLCFGn fields clear.
- Table 49 shows the connections for I2S SDOUT, which must have the corresponding INPENn and PULLCFGn fields clear.

Table 45: I2S CLK Configuration

I2S Instance	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
I2S0	FNCSEL5	2	5	✓	✓
I2S0	FNCSEL11	2	11	✓	✓
I2S0	FNCSEL31	2	31	✓	✓
I2S0	FNCSEL47	10	47	✓	✓
I2S0	FNCSEL52	10	52	✓	
I2S0	FNCSEL100	9	100	✓	✓
I2S1	FNCSEL8	10	8	✓	✓
I2S1	FNCSEL16	2	16	✓	✓
I2S1	FNCSEL34	10	34	✓	✓
I2S1	FNCSEL55	10	55	✓	
I2S1	FNCSEL61	2	61	✓	
I2S1	FNCSEL130	10	130	✓	

Table 46: I2S WS Configuration

Instance	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
I2S0	FNCSEL7	2	7	✓	✓
I2S0	FNCSEL13	2	13	✓	✓
I2S0	FNCSEL33	2	33	✓	✓
I2S0	FNCSEL48	10	48	✓	✓
I2S0	FNCSEL53	10	53	✓	
I2S0	FNCSEL102	9	102	✓	✓
I2S1	FNCSEL10	10	10	✓	✓
I2S1	FNCSEL18	2	18	✓	✓
I2S1	FNCSEL36	10	36	✓	✓

Table 46: I2S WS Configuration

Instance	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
I2S1	FNCSEL56	10	56	✓	
I2S1	FNCSEL63	2	63	✓	
I2S1	FNCSEL132	10	132	✓	

Table 47: I2S DATA Configuration

I2S Instance	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
I2S0	FNCSEL6	2	6	✓	✓
I2S0	FNCSEL12	2	12	✓	✓
I2S0	FNCSEL32	2	32	✓	✓
I2S0	FNCSEL49	9	49	✓	✓
I2S0	FNCSEL54	9	54	✓	
I2S0	FNCSEL101	9	101	✓	✓
I2S1	FNCSEL9	9	9	✓	✓
I2S1	FNCSEL17	2	17	✓	✓
I2S1	FNCSEL35	10	35	✓	✓
I2S1	FNCSEL57	9	57	✓	✓
I2S1	FNCSEL62	2	62	✓	
I2S1	FNCSEL131	9	131	✓	

Table 48: I2S SDIN Configuration

I2S Instance	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
I2S0	FNCSEL4	9	4	✓	✓
I2S0	FNCSEL14	10	14	✓	✓
I2S0	FNCSEL95	9	95	✓	✓
I2S0	FNCSEL138	9	138	✓	
I2S0	FNCSEL139	10	139	✓	
I2S1	FNCSEL3	10	3	✓	✓
I2S1	FNCSEL19	9	19	✓	✓
I2S1	FNCSEL64	9	64	✓	✓
I2S1	FNCSEL133	10	133	✓	

Table 49: I2S SDOUT Configuration

I2S Instance	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
I2S0	FNCSEL6	9	6	✓	✓
I2S0	FNCSEL12	10	12	✓	✓
I2S0	FNCSEL32	9	32	✓	✓
I2S0	FNCSEL49	10	49	✓	✓
I2S0	FNCSEL54	10	54	✓	
I2S0	FNCSEL101	10	101	✓	✓
I2S1	FNCSEL9	10	9	✓	✓
I2S1	FNCSEL17	9	17	✓	✓
I2S1	FNCSEL35	9	35	✓	✓
I2S1	FNCSEL57	10	57	✓	✓
I2S1	FNCSEL62	9	62	✓	
I2S1	FNCSEL131	10	131	✓	

12.3.8 Implementing Secure Digital IO Connections

The signals for the Secure Digital Input/Output Interface (SDIO) for connection to SD/SDIO/MMC devices can be connected to two sets of pads, SDIF0 and SDIF1, on the device. Table 50 shows the pads used for SDIO data lines, clock and command signals. The pads for the clock (CLKOUT) and command (CMD) signals should have the corresponding INPENn and PULLCFGn fields clear. The pads for the data lines should have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 50: SDIO/SDIF Configuration

Signal	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
SDIF0_DATA0	FNCSEL84	2	84	✓	✓
SDIF0_DATA1	FNCSEL85	2	85	✓	✓
SDIF0_DATA2	FNCSEL86	2	86	✓	✓
SDIF0_DATA3	FNCSEL87	2	87	✓	✓
SDIF0_DATA4	FNCSEL156	0	156	✓	✓
SDIF0_DATA5	FNCSEL157	0	157	✓	✓
SDIF0_DATA6	FNCSEL158	0	158	✓	✓
SDIF0_DATA7	FNCSEL159	0	159	✓	✓
SDIF0_CMD	FNCSEL160	0	160	✓	✓
SDIF0_CLKOUT	FNCSEL88	2	88	✓	✓
SDIF1_DATA0	FNCSEL125	0	125	✓	✓

Signal	Field	FNCSEL Value	Pad	Apollo510	Apollo510B
SDIF1_DATA1	FNCSEL126	0	126	✓	✓
SDIF1_DATA2	FNCSEL127	0	127	✓	✓
SDIF1_DATA3	FNCSEL128	0	128	✓	✓
SDIF1_DATA4	FNCSEL130	0	130	✓	
SDIF1_DATA5	FNCSEL131	0	131	✓	
SDIF1_DATA6	FNCSEL132	0	132	✓	
SDIF1_DATA7	FNCSEL133	0	133	✓	
SDIF1_CMD	FNCSEL134	0	134	✓	✓
SDIF1_CLKOUT	FNCSEL129	0	129	✓	✓

12.3.9 Implementing GPIO Connections

Each pad of the device can be configured as a GPIO port by setting FNCSELn to 3. INPENn and PULLCFGn must be set appropriately depending on the specific GPIO function.

12.3.10 Implementing CLKOUT Connections

The flexible clock output of the Clock Generator module, CLKOUT, may be configured on several pads as shown in Table 51. INPENn and PULLCFGn should be cleared in each case.

Table 51: CLKOUT Configuration for Apollo510 and Apollo510B

Field	FNCSEL Value	Pad	Apollo510	Apollo510B
FNCSEL33	1	33	✓	✓
FNCSEL63	1	63	✓	
FNCSEL66	1	66	✓	✓
FNCSEL67	1	67	✓	✓
FNCSEL71	1	71	✓	✓
FNCSEL72	1	72	✓	✓
FNCSEL80	1	80	✓	
FNCSEL81	1	81	✓	

Table 52: CLKOUT Configuration for Apollo510 Lite and Apollo330 Plus

Field	FNCSEL Value	Pad	Apollo510 Lite	Apollo330 Plus
FNCSEL6	4	6	✓	✓
FNCSEL31	1	31	✓	✓
FNCSEL47	2	47	✓	✓
FNCSEL48	2	48	✓	✓

Table 52: CLKOUT Configuration for Apollo510 Lite and Apollo330 Plus

Field	FNCSEL Value	Pad	Apollo510 Lite	Apollo330 Plus
FNCSEL52	2	52	✓	✓
FNCSEL56	2	56	✓	✓
FNCSEL57	2	57	✓	✓
FNCSEL65	2	65	✓	✓
FNCSEL75	2	75	✓	✓
FNCSEL76	2	76	✓	✓
FNCSEL90	2	90	✓	✓
FNCSEL91	2	91	✓	✓

12.3.11 Implementing 32 kHz CLKOUT Connections

In addition to the CLKOUT mux output, there is also a dedicated 32 kHz clock output, 32KHzXT. This clock is primarily for leveraging the 32 kHz oscillator clock from the device. This clock output may be configured on several pads as shown in Table 53. INPENn and PULLCFGn should be cleared in each case.

Table 53: 32 kHz CLKOUT Configuration

Field	FNCSEL Value	Pad	Apollo510	Apollo510B
FNCSEL4	2	4 ¹	✓	✓
FNCSEL5	5	5 ²	✓	✓
FNCSEL37	2	37	✓	✓
FNCSEL45	2	45	✓	✓
FNCSEL64	1	64	✓	✓
FNCSEL65	1	65	✓	✓
FNCSEL69	1	69	✓	✓
FNCSEL70	1	70	✓	✓
FNCSEL75	1	75	✓	
FNCSEL76	1	76	✓	
FNCSEL82	1	82	✓	
FNCSEL83	1	83	✓	✓
FNCSEL138	1	138	✓	
FNCSEL208	1	208	✓	

1. Apollo510 SoC and Apollo510B SoC only.

2. Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs only.

12.3.12 Implementing 32 MHz CLKOUT Connections

There is also a dedicated 32 MHz¹ clock output, CLKOUT_32M. This clock is primarily for leveraging the high-speed 32 MHz oscillator clock from the device. This clock output may be configured on one pad as shown in Table 54. INPENn and PULLCFGn should be cleared in this case.

Table 54: CLKOUT_32M Configuration

Field	Value	Pad	Apollo510	Apollo510B
FNCSEL46	2	46	✓	

12.3.13 Implementing ADC Connections

Three types of pad connections may be made for the ADC module. Up to eight pads may be selected from and configured as the analog inputs, as shown in Table 55. The ADCREF reference voltage input is supplied on a dedicated input pin.

If an external digital trigger is desired, up to forty selectable pad choices may be selected from and configured, as shown in Table 56. For the trigger inputs, INPENn must be set. For other inputs, INPENn should be cleared. PULLCFGn should be cleared except in the case of an open drain trigger input.

Table 55: ADC Analog Input Configuration

Field	FNCSEL Value	Input	Pad	Apollo510	Apollo510B
FNCSEL19	0	ADCSE0	19	✓	✓
FNCSEL18	0	ADCSE1	18	✓	✓
FNCSEL17	0	ADCSE2	17	✓	✓
FNCSEL16	0	ADCSE3	16	✓	✓
FNCSEL15	0	ADCSE4	15	✓	✓
FNCSEL14	0	ADCSE5	14	✓	✓
FNCSEL13	0	ADCSE6	13	✓	✓
FNCSEL12	0	ADCSE7	12	✓	✓

Table 56: ADC Trigger Input Configuration

Field	FNCSEL Value	Input	Pad	Apollo510	Apollo510B
FNCSEL7	1	TRIG0	7	✓	✓
FNCSEL11	1	TRIG0	11	✓	✓
FNCSEL15	1	TRIG0	15	✓	
FNCSEL27	1	TRIG0	27	✓	✓
FNCSEL29	0	TRIG0	29	✓	✓
FNCSEL36	1	TRIG0	36	✓	✓
FNCSEL41	1	TRIG0	41	✓	✓

1. The output tracks the frequency of XTALHS which may not be 32 MHz.

Table 56: ADC Trigger Input Configuration

Field	FNCSEL Value	Input	Pad	Apollo510	Apollo510B
FNCSEL49	1	TRIG0	49	✓	✓
FNCSEL50	1	TRIG0	50	✓	✓
FNCSEL54	1	TRIG0	54	✓	
FNCSEL59	1	TRIG0	59	✓	
FNCSEL2	2	TRIG1	2	✓	✓
FNCSEL8	1	TRIG1	8	✓	✓
FNCSEL12	1	TRIG1	12	✓	✓
FNCSEL16	1	TRIG1	16	✓	✓
FNCSEL20	1	TRIG1	20	✓	✓
FNCSEL30	0	TRIG1	30	✓	
FNCSEL37	1	TRIG1	37	✓	✓
FNCSEL40	1	TRIG1	40	✓	✓
FNCSEL44	1	TRIG1	44	✓	✓
FNCSEL51	1	TRIG1	51	✓	✓
FNCSEL55	1	TRIG1	55	✓	
FNCSEL60	1	TRIG1	60	✓	
FNCSEL9	1	TRIG2	9	✓	✓
FNCSEL13	1	TRIG2	13	✓	✓
FNCSEL17	1	TRIG2	17	✓	✓
FNCSEL21	1	TRIG2	21	✓	✓
FNCSEL38	1	TRIG2	38	✓	✓
FNCSEL42	1	TRIG2	42	✓	✓
FNCSEL45	1	TRIG2	45	✓	✓
FNCSEL52	1	TRIG2	52	✓	
FNCSEL56	1	TRIG2	56	✓	
FNCSEL10	1	TRIG3	10	✓	✓
FNCSEL14	1	TRIG3	14	✓	✓
FNCSEL24	1	TRIG3	24	✓	✓
FNCSEL39	1	TRIG3	39	✓	✓
FNCSEL43	1	TRIG3	43	✓	✓
FNCSEL46	1	TRIG3	46	✓	✓
FNCSEL53	1	TRIG3	53	✓	
FNCSEL57	1	TRIG3	57	✓	✓

12.3.14 Implementing Voltage Comparator Connections

Two types of pad connections may be used for the Voltage Comparator (VCOMP) module. One of three reference voltages may be used for the comparator negative input, CMPRF, as shown in Table 57. The

voltage to be applied to the comparator positive input, CMPIN, are shown in Table 58. In each case INPENn and PULLCFGn should be cleared. The voltage to be output from the comparator to an output pin are shown in Table 59. In each case INPENn and PULLCFGn should be cleared.

Table 57: Voltage Comparator Reference Configuration

Field	FNCSEL Value	Input	Pad	Apollo510	Apollo510B
FNCSEL9	0	CMPRF0	9	✓	✓
FNCSEL8	0	CMPRF1	8	✓	✓
FNCSEL12	9	CMPRF2	12	✓	✓

Table 58: Voltage Comparator Input Configuration

Field	FNCSEL Value	Input	Pad	Apollo510	Apollo510B
FNCSEL10	0	CMPIN0	10	✓	✓
FNCSEL11	0	CMPIN1	11	✓	✓

Table 59: Voltage Comparator Output Configuration

Field	FNCSEL Value	Output	Pad	Apollo510	Apollo510B
FNCSEL0	9	VCMPO	0	✓	✓
FNCSEL1	9	VCMPO	1	✓	✓
FNCSEL2	9	VCMPO	2	✓	✓
FNCSEL22	9	VCMPO	22	✓	✓
FNCSEL23	9	VCMPO	23	✓	✓
FNCSEL26	9	VCMPO	26	✓	✓
FNCSEL28	1	VCMPO	28	✓	✓
FNCSEL29	1	VCMPO	29	✓	✓
FNCSEL30	1	VCMPO	30	✓	
FNCSEL31	9	VCMPO	31	✓	✓
FNCSEL34	9	VCMPO	34	✓	✓
FNCSEL44	9	VCMPO	44	✓	✓
FNCSEL52	9	VCMPO	52	✓	
FNCSEL56	9	VCMPO	56	✓	
FNCSEL72	9	VCMPO	72	✓	✓
FNCSEL90	9	VCMPO	90	✓	
FNCSEL91	9	VCMPO	91	✓	
FNCSEL92	9	VCMPO	92	✓	

Table 59: Voltage Comparator Output Configuration

Field	FNCSEL Value	Output	Pad	Apollo510	Apollo510B
FNCSEL93	9	VCMPO	93	✓	
FNCSEL94	9	VCMPO	94	✓	

12.3.15 Implementing the Software Debug Port Connections

The software debug clock, SWDCK, and data, SWDIO, must be connected on pads 20 and 21 respectively. FNCSEL20 and FNCSEL21 must be set to 0, INPEN20 and INPEN21 must be set, and PULLCFG20 and PULLCFG21 must be set, which results in a default state of SWDCK low and SWDIO high.

The optional continuous output signal SWO may be configured on a variety of pads as shown in Table 60, and INPENn and PULLCFGn should be cleared for the selected pad.

Table 60: SWO Configuration

Field	FNCSEL Value	Pad	Apollo510	Apollo510B
FNCSEL3	2	3	✓	✓
FNCSEL22	2	22	✓	✓
FNCSEL23	2	23	✓	✓
FNCSEL24	2	24	✓	✓
FNCSEL28	0	28	✓	✓
FNCSEL34	2	34	✓	✓
FNCSEL35	2	35	✓	✓
FNCSEL41	9	41	✓	✓
FNCSEL44	2	44	✓	✓
FNCSEL56	2	56	✓	
FNCSEL57	2	57	✓	✓
FNCSEL64	2	64	✓	✓
FNCSEL65	2	65	✓	✓
FNCSEL66	2	66	✓	✓
FNCSEL67	2	67	✓	✓
FNCSEL68	1	68	✓	✓
FNCSEL69	2	69	✓	✓
FNCSEL79	4	79	✓	

Optional debug trace output signals, SWTRACE0-SWTRACE3, as well as the trace clock (SWTRACECLK), may be configured on a variety of pads as shown in Table 61. INPENn and PULLCFGn should be cleared for the selected pads.

Table 61: SW Trace Configuration

Field	FNCSEL Value	Signal	Pad	Apollo510	Apollo510B
FNCSEL1	0	SWTRACE0	1	✓	✓
FNCSEL39	2	SWTRACE0	39	✓	✓
FNCSEL51	2	SWTRACE0	51	✓	✓
FNCSEL70	2	SWTRACE0	70	✓	✓
FNCSEL80	4	SWTRACE0	80	✓	
FNCSEL143	0	SWTRACE0	143	✓	✓
FNCSEL2	0	SWTRACE1	2	✓	✓
FNCSEL40	2	SWTRACE1	40	✓	✓
FNCSEL52	2	SWTRACE1	52	✓	
FNCSEL71	2	SWTRACE1	71	✓	✓
FNCSEL81	4	SWTRACE1	81	✓	
FNCSEL144	0	SWTRACE1	144	✓	✓
FNCSEL3	0	SWTRACE2	3	✓	✓
FNCSEL41	2	SWTRACE2	41	✓	✓
FNCSEL53	2	SWTRACE2	53	✓	
FNCSEL72	2	SWTRACE2	72	✓	✓
FNCSEL82	4	SWTRACE2	82	✓	
FNCSEL145	0	SWTRACE2	145	✓	✓
FNCSEL4	0	SWTRACE3	4	✓	✓
FNCSEL42	2	SWTRACE3	42	✓	✓
FNCSEL54	2	SWTRACE3	54	✓	
FNCSEL73	2	SWTRACE3	73	✓	✓
FNCSEL83	4	SWTRACE3	83	✓	✓
FNCSEL146	0	SWTRACE3	146	✓	✓
FNCSEL0	0	SWTRACECLK	0	✓	✓
FNCSEL38	2	SWTRACECLK	38	✓	✓
FNCSEL50	2	SWTRACECLK	50	✓	✓
FNCSEL142	0	SWTRACECLK	142	✓	✓

13. General Purpose ADC and Temperature Sensor Module (GPADC)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

13.1 Clock Source and Divider

When the GPADC block is enabled and has an active scan in progress, it requests a clock source. There is an automatic hardware hand shake between the clock generator and the GPADC. If the GPADC is the only block requesting an HFRC based clock, then the HFRC will be automatically started. The GPADC can be configured to completely power down the HFRC between scans if the startup latency is acceptable or it can leave the HFRC powered on between scans if the application requires low latency between successive conversions.

The GPADC on the Apollo510 and Apollo510B offers four options for the reference clock source and frequency via the GPADC's CFG_CLKSEL field:

1. HFRC_48MHz clock (default setting)
2. Inverted HFRC_48MHz clock
3. HFRC_24MHz clock
4. HFRC2_31MHz clock

The Clock/Reset Module (CRM) on the Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs offers two options for the reference clock source via the CRM's ADCCRM_ADCCLKSEL field:

1. HFRC_48MHz clock (default setting)
2. PLLPOSTDIV

An integer divider can be used to further divide down (by 2, 3 or 4) either selected clock source by configuring the ADCCRM_ADCCLKDIV field.

13.2 Eleven Selectable Analog Input Channels

The GPADC block contains an 11 channel analog multiplexer on the input port to the analog to digital converter. Eight (8) of the GPIO pins on the MCU can be selected as analog inputs to the GPADC through a combination of settings in the PAD configuration registers in the GPIO block and settings in the configuration registers described below.

The analog mux channels are connected as follows:

1. ADC_EXT0 external GPIO pin connection.
2. ADC_EXT1 external GPIO pin connection.
3. ADC_EXT2 external GPIO pin connection.
4. ADC_EXT3 external GPIO pin connection.
5. ADC_EXT4 external GPIO pin connection.
6. ADC_EXT5 external GPIO pin connection.
7. ADC_EXT6 external GPIO pin connection.
8. ADC_EXT7 external GPIO pin connection.
9. ADC_TEMP internal temperature sensor.
10. ADC_BATT internal voltage divide-by-3 connection.
11. ADC_VSS internal ground connection.

Refer to the detailed register information below for the exact coding of the channel selection bit field. Also the use of the voltage divider and switchable load resistor are detailed below.

13.3 Triggering and Trigger Sources

The GPADC block can be initially triggered from one of six sources. Once triggered, it can be repetitively triggered from timer 7 in the Timer Module or via the GPADC's internal repeating trigger timer. Four of the GPIO pins on the MCU can be selected as trigger inputs to the GPADC through a combination of settings in the PAD configuration registers in the GPIO block and settings in SLOT configuration registers described below. trigger sources are as follows:

0. ADC_EXT0 (TRIG0) external GPIO pin connection.
1. ADC_EXT1 (TRIG1) external GPIO pin connection.
2. ADC_EXT2 (TRIG2) external GPIO pin connection.
3. ADC_EXT3 (TRIG3) external GPIO pin connection.
4. VCOMP Voltage Comparators trigger.
5. <Reserved>
6. <Reserved>
7. ADC_SWT software trigger.

Refer to the CFG Register in the GPADC register set for the Apollo5 SoC. The initial trigger source is selected in the TRIGSEL field. In addition, one can select a trigger polarity in this register applicable for any of the trigger sources except the software trigger. A number of GPIO pin trigger sources are provided to allow pin configuration flexibility at the system definition and board layout phases of development.

The software trigger is effected by writing 0x37 to the to the SWT field of the Software Trigger Register in the GPADC block. Note that writing 0x37 to the SWT field will initiate a scan regardless of which trigger source is selected. However, a hardware trigger source will not initiate a scan if the software trigger has been selected (in the TRIGSEL field of the CFG Register).

When the GPADC is configured for repeat mode, the initial trigger must be initiated by a software trigger and subsequent scans will be initiated at a repeating rate set by the TIMER7 or the GPADC-internal repeating trigger timer. The discussion of the use of TIMER7 or GPADC-internal timer as a source for repetitive triggering is deferred until later in this chapter.

NOTE

A trigger event applies to all enabled slots as a whole. Individual slots cannot be separately triggered.

13.4 Voltage Reference Source

The GPADC supports one internal reference source to be used for the analog to digital conversion step. The reference voltage is 1.19 V and is not user settable. GPADC input voltages > 1.19 V exceed the GPADC range and return full scale code, but will not damage GPADC inputs.

13.5 Eight Automatically Managed Conversion Slots

The GPADC block contains eight conversion slot control registers, one for each of the eight slots. These can be thought of as time slots in the conversion process. When a slot is enabled, it participates in a conversion cycle. The GPADC's mode controller cycles through up to eight time slots each time it is triggered. For each slot that is enabled, a conversion cycle is performed based on the settings in the slot

configuration register for that slot. A slot is enabled when the SLENn field (n = 0 to 7) of the Slotn Configuration register is set to one. See Table 62.

Table 62: SLOT Configuration Register

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0
Reserved				ADSEL (# Samples to Accum)		TRKCYC						PRMODE	Reserved				CHSEL		Reserved						WCEN	SLEN					

The window comparator enable is discussed in a subsequent section below. See “Window Comparator” on page 147. The number of samples to accumulate is explained in a subsequent section. See “Automatic Sample Accumulation and Scaling” on page 143.

As described above, the Channel Select bit field, CHSELn, specifies which one of the analog multiplexer channels will be used for the conversions requested for an individual slot. See “Eleven Selectable Analog Input Channels” on page 141.

The Precision Mode field, PRMODE, enables the selection of the slot’s precision - 8-bit, 10-bit or 12-bit.

An additional number of ADC clock cycles may be added to the default value of 5 cycles to specify the input signal sampling/tracking time by setting the TRKCYC field. Five cycles are added to the value set in this field to determine the total number of sampling/tracking cycles.

Each of the eight conversion slots can independently specify:

- Analog Multiplexer Channel Selection
- Participation in Window Comparisons
- Automatic Sample Accumulation

13.6 Automatic Sample Accumulation and Scaling

The GPADC block offers a facility for the automatic accumulation of samples without requiring core involvement. Thus up to 128 samples per slot can be accumulated without waking the core. This facilitates averaging algorithms to smooth out the data samples. Each slot can request from 1 to 128 samples to be accumulated before producing a result in the FIFO.

NOTE

Each slot can independently specify how many samples to accumulate so results can enter the FIFO from different slots at different rates.

All slots write their accumulated results to the FIFO in exactly the same format regardless of how many samples were accumulated to produce the results. The precision mode for each determines the format for the FIFO data. 12-bit, 10-bit and 8-bit precision modes correspond to 12.6, 10.6 and 8.6 formats, respectively.

NOTE

If the accumulation control for a slot is set for two samples with 8-bit precision, then the 8-bit average integer value will be placed in bits 6 through 13, the 1 bit fractional number is placed in bit 5 and the lower 5 fractional bits are zero'd.

Each slot contains a 21-bit accumulator as shown in Table 63 on page 144. When the GPADC is triggered for the last sample of an accumulation, the accumulator is cleared and the FIFO will be written with the final average value. When each active slot obtains a sample from the GPADC, it is added to the value in its accumulator.

Table 63: Per Slot Sample Accumulator

2	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Accumulator																				

Table 64 shows the maximum possible 10-bit accumulated values. Note that for the 128 sample accumulation case, the LSB of the accumulator is discarded when the results are written to the FIFO.

Table 64: Accumulator Scaling

# Samples	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0			
128	10.6																		0	
64	X	10.6																		
32	X	X	10.5																	
16	X	X	X	10.4																
8	X	X	X	X	10.3															
4	X	X	X	X	X	10.2														
2	X	X	X	X	X	X	10.1													
1	X	X	X	X	X	X	X	10												

13.7 Sixteen Entry Result FIFO

All results written to the FIFO have exactly the same format as shown in Table 65. The properly scaled accumulation results are written the lower half word in the aforementioned format. Since each slot can

R S V	3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0			
	Slot Number.			FIFO Count								FIFO DATA																							

# Samples	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0	
128	0	0	12.6																		
64	0	0	12.6																		
32	0	0	12.5																		X
16	0	0	12.4																	X	X
8	0	0	12.3															X	X	X	
4	0	0	12.2														X	X	X	X	
2	0	0	12.1													X	X	X	X	X	
1	0	0	12												X	X	X	X	X	X	

# Samples	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
128	0	0	0	0	10										6					
64	0	0	0	0	10										6					
32	0	0	0	0	10										5				X	
16	0	0	0	0	10										4			X	X	
8	0	0	0	0	10										3		X	X	X	
4	0	0	0	0	10										2		X	X	X	X
2	0	0	0	0	10										1	X	X	X	X	X
1	0	0	0	0	10										X	X	X	X	X	X

Table 68: 8-bit FIFO Data Format

# Samples	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
128	0	0	0	0	0	0	8.6													
64	0	0	0	0	0	0	8.6													
32	0	0	0	0	0	0	8.5												X	
16	0	0	0	0	0	0	8.4											X	X	
8	0	0	0	0	0	0	8.3										X	X	X	X
4	0	0	0	0	0	0	8.2										X	X	X	X
2	0	0	0	0	0	0	8.1								X	X	X	X	X	X
1	0	0	0	0	0	0	8								X	X	X	X	X	X

Software accesses the contents of the FIFO through the ADCFIFO register. This register will be written by the GPADC digital controller simultaneous with the conversion complete interrupt (if enabled) after accumulating the number of samples to average configured for the slot. The ADCFIFO register contains the earliest written data, the number of valid entries within the FIFO and the slot number associated with the FIFO data. Thus the interrupt handler servicing GPADC interrupts can easily distribute results to different RTOS tasks by simply looking up the target task using the slot number from the FIFO register.

NOTE

After a conversion complete ISR (CNVCMP) is asserted, a minimum time delay of 30 μ s is required before reading the FIFO, otherwise the FIFO data count may be incorrect.

Three other features greatly simplify the task faced by firmware developers of interrupt service routines for the GPADC block:

1. The FIFO count bit field is not really stored in the FIFO. Instead it is a live count of the number of valid entries currently residing in the FIFO. If the interrupt service routine was entered because of a conversion then this value will be at least one. When the interrupt routine is entered it can pull successive sample values from the FIFO until this bit field goes to zero. Thus avoiding wasteful re-entry of the interrupt service routine. Note that no further I/O bus read is required to determine the FIFO depth.
2. This FIFO has no read side effects. This is important to firmware for a number of reasons. One important result is that the FIFO register can be freely read repetitively by a debugger without affecting the state of the FIFO. In order to pop this FIFO and look at the next result, if any, one simply writes any value to this register. Any time the FIFO is read, then the compiler has gone to the trouble of generating an address for the read. To pop the FIFO, one simply writes to that same address with any value. This give firmware a positive handshake mechanism to control exactly when the FIFO pops.
3. When a conversion completes resulting in hardware populating the 12th valid FIFO entry, the FIFOVR1 (FIFO 75% full) interrupt status bit will be set. When a conversion completes resulting in hardware populating the 16th valid FIFO entry, the FIFOVR2 interrupt status bit will be set. In a FIFO full condition with 16 valid entries, the GPADC will not overwrite existing valid FIFO contents. Before subsequent conversions will populate the FIFO with conversion data, software must free an open FIFO entry by writing to the FIFO Register or by resetting the GPADC by disabling and enabling the GPADC using the ADC_CFG register.

13.8 Window Comparator

A window comparator is provided which can generate an interrupt whenever a sample is determined to be inside the window limits or outside the window limits. These are two separate interrupts with separate interrupt enables. Thus one can request an interrupt any time a specified slot makes an excursion outside the window comparator limits.

The window comparison function has an option for comparing the contents of the limits registers directly with the FIFO data (default) or for scaling the limits register depending on the precision mode selected for the slots.

Firmware has to participate in the determination of whether an actual excursion occurred. The window comparator interrupts set their corresponding interrupt status bits continuously whenever the inside or outside condition is true. Thus if one enables and receives an “*excursion*” interrupt then the status bit can’t be usefully cleared while the GPADC slot is sampling values outside the limits. That is, if one receives an excursion interrupt and clears the status bit, it will immediately set again if the next GPADC sample is still outside the limits. Thus firmware should reconfigure the interrupt enables upon receiving an excursion interrupt so that the next interrupt will occur when an GPADC sample ultimately goes back inside the window limits. Firmware may also want to change the windows comparator limit at that time to utilize a little hysteresis in these window comparator decisions.

Table 69: Window Comparator Lower Limit Register

1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Lower Limit																			

Table 70: Window Comparator Upper Limit Register

1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Upper Limit																			

The determination of whether a sample is *inside* or *outside* of the window limits is made by comparing the data format of the slot result written to the FIFO with the 20 bit window limits. An GPADC sample is inside if the following relation is true:

$$14.6 \text{ Lower Limit} \leq \text{GPADC SAMPLE} \leq 14.6 \text{ Upper Limit}$$

Thus setting both limits to the same value, say 700.0 (0x2BC<<6 = 0xAF00), will only produce an inside interrupt when the GPADC sample is exactly 700.0 (0xAF00). Furthermore, note that if the lower limit is set to zero (0x00000) and the upper limit is set to 0xFFFFF then all accumulated results from the GPADC will be inside the window limits and no excursion interrupts can ever be generated. In fact, in this case, the incursion interrupt status bit will be set for every sample from any active slot with its window comparator bit enabled. If the incursion interrupt is enabled then an interrupt will be generated for every such sample written to the FIFO.

The window comparator limits are a shared resource and apply to all active slots which have their window comparator bits enabled. If window limits are enabled for multiple enabled slots with different precision modes, the window comparison function can be configured to automatically scale the 14.6 upper and lower limits value to match the corresponding precision mode format for the enabled slots through the ADCSCWLIM register.

13.9 Operating Modes and the Mode Controller

The mode controller is a sophisticated state machine that manages not only the time slot conversions but also the power state of the GPADC analog components and the hand shake with the clock generator to start the HFRC clock source if required. Thus once the various control registers are initialized, the core can go to sleep and only wake up when there are valid samples in the FIFO for the interrupt service routine to distribute. Firmware does not have to keep track of which block is using the HFRC clock source since the devices in conjunction with the clock generator manage this automatically. The GPADC block's mode controller participates in this clock management protocol.

From a firmware perspective, the GPADC mode controller is controlled from bit fields in the GPADC configuration register and from the various bit fields in the eight slot configuration registers.

The most overriding control is the GPADC enable bit in the PWRCTRL_DEVPWREN register of the power control block. This bit must be set to '1' to enable power to the GPADC subsystem. Furthermore, the PWRENADC bit in the GPADC configuration register is a global functional enable bit for general GPADC operation. Setting this bit to zero has many of the effects of a software reset, such as resetting the FIFO pointers. Setting this bit to one enables the mode controller to examine its inputs and proceed to autonomously handle analog to digital conversions.

A GPADC scan is the process of sampling the analog voltages at each input of the GPADC following a trigger event. If the GPADC is enabled and one or more slots are enabled, a scan is initiated after the GPADC receives a trigger through one of the configured trigger sources. The scan flowchart diagram can be found in Figure 16

A GPADC conversion is the process of averaging measurements following one or more scans for each slot that is enabled.

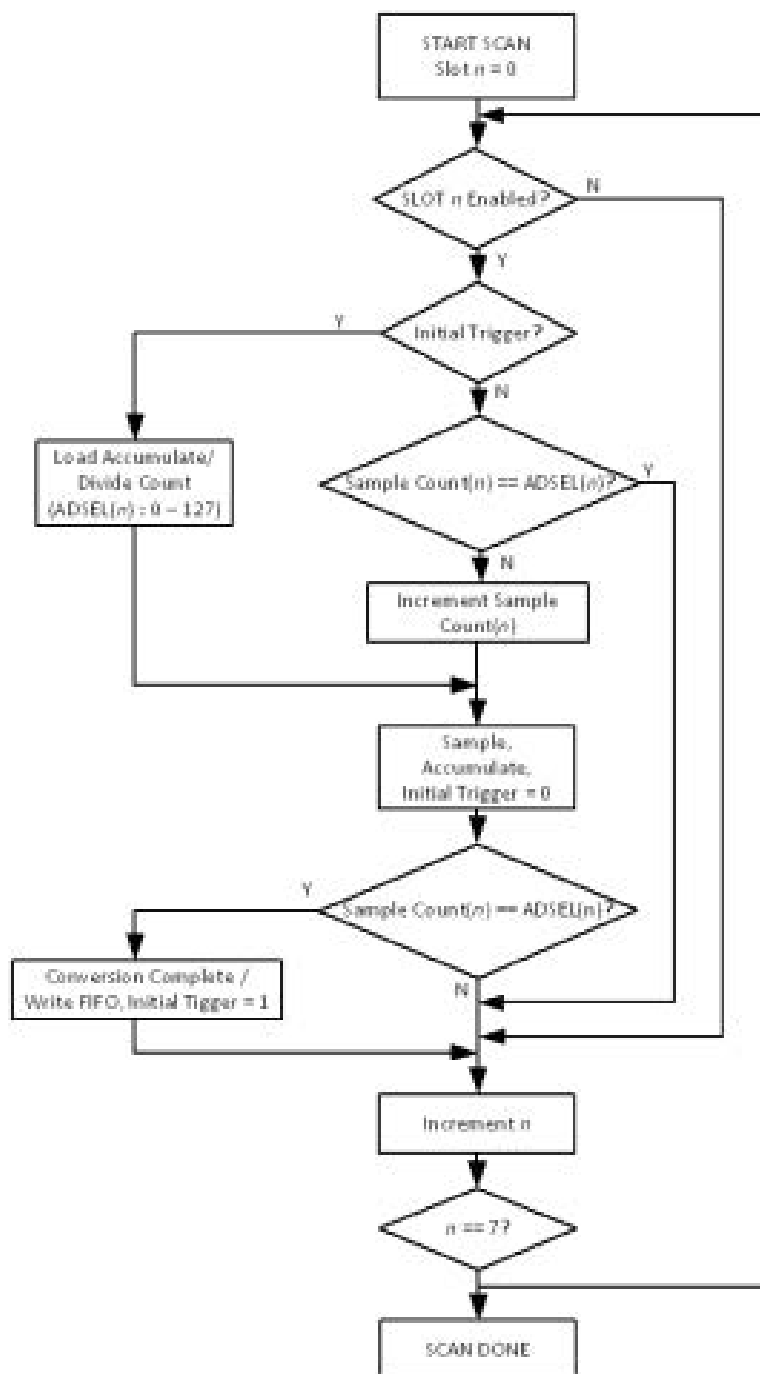


Figure 16. Scan Flowchart

13.9.1 Single Mode

In single mode, one trigger event produces one scan of all enabled slots. Depending on the settings of the accumulate and scale bit field for the active slots, this may or may not result in writing a result to the FIFO. When the trigger source is an external pin then one external pin transition of the proper polarity will result

in one complete scan of all enabled slots. If the external pin is connected to a repetitive pulse source then repeating scans of all enabled slots are run at the input trigger rate.

13.9.2 Repeat Mode

Timing for repeat triggers in Repeat Mode can be selected between the GPADC-internal repeat trigger timer and Timer 7 of the Timer Module. Selection is made via the RPTTRIGSEL bit of the GPADC's CFG register. If the GPADC-internal timer is selected for the repeating trigger timing, then the GPADC's INTTRIGTIMER register is used to enable this timer (TIMEREN), select the timer's clock divider (CLKDIV), and set the trigger count (TIMERMAX).

To use The Timer Module's Timer7 for repeat trigger timing, there is a bit in the Timer module's Global Register, ADCEN, which allows Counter/Timer 7 to be a source of repetitive triggers for the GPADC. If counter/timer 7 is initialized for this purpose then one only needs to turn on the RPTEN bit in the GPADC CFG register to enable this mode in the GPADC.

NOTE

The mode controller does **not** process these repetitive triggers from the counter/timer until a first triggering event occurs from the normal trigger sources. Thus one can select software triggering in the TRIGSEL field and set up all of the other GPADC registers for the desired sample acquisitions. Then one can write to the software trigger register and the mode controller will enter REPEAT mode. In repeat mode, the mode controller waits only for each successive counter/timer 7 input to launch a scan of all enabled slots.

13.9.3 Low Power Modes

An application may use the GPADC in one of three power modes. Each mode has different implications from overall energy perspective relative to the startup latency from trigger-to-data as well as the standby power consumed. The table below is intended to provide guidance on which mode may be more effective based on latency tolerance. This table should only be used as a reference.

Table 71: GPADC Power Modes

LPMODE	Definition	Entry Latency
0	GPADC is kept active continuously (used in continuous sampling scenarios)	0
1	GPADC is mostly powered off between samples, HFRC is duty cycled between samples.	<70μs (shorter for lower resolution)
2	GPADC is completely powered off between samples, HFRC is duty cycled between samples.	<660μs

13.9.3.1 Low Power Mode 0

Low Power Mode 0 (LPMODE0) enables the lowest latency from trigger to conversion data available. This mode leaves the reference buffer powered on between scans to bypass any startup latency between triggers¹.

1. The reference buffer will not be powered on when the GPADC is configured for external reference

13.9.3.2 Low Power Mode 1

Low power mode 1 (LPMODE1) is a power mode whereby the GPADC Digital Controller will automatically power off the GPADC clocks, analog GPADC and reference buffer between scans. This mode may operate autonomously without CPU interaction, even while the CPU is in sleep or deep sleep mode for repeat mode triggers or hardware triggers. While operating in this mode, the GPADC Digital Controller may be used to burst through multiple scans enabling max sample rate data collection if the triggers are running at a rate at least 2x the maximum sample rate until the final scan has completed. When a scan completes without a pending trigger latched, the GPADC subsystem will enter a low power state until the next trigger event.

13.9.3.3 Low Power Mode 2

If desirable, for applications requiring infrequent conversions, software may choose to operate the GPADC in LPMODE2, whereby the full GPADC Analog and Digital subsystem remains completely powered off between samples. In this use case, the software configures the power control GPADC enable register followed by configuring the GPADC slots and the GPADC configuration register between conversion data collections, followed by disabling the GPADC in the power control GPADC enable register. Although this mode provides extremely low power operation, using the GPADC in this mode will result in a cold start latency including reference buffer stabilization delay. In this mode, the GPADC must be reconfigured prior to any subsequent GPADC operation.

13.10 Interrupts

The GPADC has 8 interrupt status bits with corresponding interrupt enable bits, as follows:

1. Conversion Complete Interrupt
2. Scan Complete Interrupt
3. FIFO Overflow Level 1
4. FIFO Overflow Level 2
5. Window Comparator Excursion Interrupt (a.k.a. outside interrupt)
6. Window Comparator Incursion Interrupt (a.k.a. inside interrupt)
7. DMA Complete (DCMP)
8. DMA Error (DERR)

The window comparator interrupts are discussed above. See “Window Comparator” on page 147.

There are two interrupts based on the *fullness* of the FIFO. When the respective interrupts are enabled, Overflow 1 fires when the FIFO reaches 75% full, viz. 6 entries. Overflow 2 fires when the FIFO is completely full.

When enabled, the conversion complete interrupt fires when a single slot completes its conversion and the resulting conversion data is pushed into the FIFO.

When enabled, the scan complete interrupt indicates that all enabled slots have sampled their respective channels following a trigger event.

When a single slot is enabled and programmed to average over exactly one measurement and the scan complete and conversion complete interrupts are enabled, a trigger event will result in the conversion complete and scan complete interrupts firing simultaneously upon completion of the GPADC scan. Again, if both respective interrupts are enabled and a single slot is enabled and programmed to average over 128 measurements, 128 trigger events result in 128 scan complete interrupts and exactly one conversion complete interrupt following the 128 GPADC scans. When multiple slots are enabled with different settings for the number of measurements to average, the conversion complete interrupt signifies that one or more of the conversions have completed and the FIFO contains valid data for one or more of the slot conversions.

NOTE

After a conversion complete ISR (CNVCMP) is asserted, a minimum time delay of 30 μ s is required before reading the FIFO, otherwise the FIFO data count may be incorrect.

13.11 Generating the Sample Rate for the GPADC

TIMER7 of the Timer Module has a special function which allows it to function as the sample trigger generator for the GPADC. If the TIMER_GLOBEN_ADCEN bit is set, the output of the timer is sent to the GPADC which uses it as a trigger. Typically, TIMER7 is configured in Repeatably Up-counter Compare (UPCOUNT - FN =2) mode. INTEN_TMR70INT may be set to generate an interrupt whenever the trigger occurs, but typically the GPADC interrupt will be used for this purpose.

13.12 Temperature Sensor Operation

Since the temperature sensor's voltage vs temperature graph is approximately linear between -273.15°C (0°K) and 85°C (358.15°K) across PVT, a constant slope is used to compute temperature.

The slope for the temperature sensor generally exhibits only very slight deviations when characterized over PVT. Given that fact, temperature computations can be simplified to the following equation:

$$T = m \times V + (T1 - (m \times V1)) \text{ where}$$

m = Constant slope value specified in °K/V

$V = V_{\text{meas}} - V_{\text{off}}$ where

V_{meas} = Measured voltage (based on the ADC code)

V_{off} = Calibration offset (stored in volts)

$V1$ = Calibration voltage (stored in volts)

$T1$ = Calibration temperature (stored in °K)

The equation can be expanded and rearranged as:

$$T = T1 + m(V_{\text{meas}} - V1 - V_{\text{off}}) \text{ or}$$

$$T = m \times V_{\text{meas}} + T1 - m(V1 + V_{\text{off}})$$

Note that the second and third terms of the second form can be computed one time and saved such that for each subsequent sample, only two computations are needed - $m \times V_{\text{meas}}$ and then add the saved term.

14. Multi-bit Serial Peripheral Interface Manager Module (MSPI)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

NOTE

Only two MSPI instances are pinned out on the Apollo510B, as MSPI1 is used for internal communication between the Apollo510B MCU and the BLE Controller module. Consult the datasheet for possible operating limitations of the individual MSPI instances.

NOTE

Check the datasheet of the applicable device for MSPI clocking and functional restrictions.

14.1 Configuration

Each MSPIn module should be configured to match the transfer characteristics of the external device(s) on the bus. Generally, the configuration sequence would proceed as follows:

- Configure MSPIn clock divider (CLKDIV0 field in the DEV0CFG register). The MSPI's core clock frequency is 125 MHz or 96 MHz depending on the MSPI instance, so the resulting clock frequency is $F_{\text{Core\_CLK}}/\text{CLKDIV}$ value as set in the MSPIn_DEV0CFG_CLKDIV0 field.
- Configure the MSPIn PADOUTEN register to enable the desired bits on the MSPIn bus (clock plus relevant data bits).
- Configure other MSPIn transfer characteristics in the DEV0CFG register.

NOTE

Enabling unused data lines in the MSPIn_PADOUTEN_OUTEN field will impact the voltage levels present on those pads even if the GPIO function select (GPIO_PINCFGn_FNCSELn) is not set to MSPI.

- Program the external device to the appropriate mode, either dual, quad, octal or hex (if supported).
- Update DEV0CFG register to new settings (in cases of a transfer mode or addressing change).
- Program the DEV0XIP register to set read/write instructions and transfer characteristics for DMA/XIP operations (and optionally enable XIP mode).

Each MSPIn's DEV0CFG register contains the controller's settings when communicating with any given device and it is expected that these values will be static after initial configuration of the external memory devices. The MSPIn_DEV0CFG_DEVCFG0 field specifies both the transfer mode (serial, dual, quad, etc) as well as which chip select is used to access the device. The MSPIn_DEV0CFG_ISIZE0 and MSPIn_DEV0CFG_ASIZE0 fields indicate the number of bytes transmitted for the instruction and address phases, but individual operations can select whether to transmit these or not. The TURNAROUND0 field indicates the number of cycles between the TX of instruction/address and reception of the first RX byte (the flash device must be programmed to use the same count). The same register also includes a WRITELATENCY0 field which can be used to insert wait states between write address and write data in a manner similar to the TURNAROUND for read operations.

14.2 PIO Operation

Software can issue general PIO (Programmed Input/Output) operations to devices on the MSPI bus using the INSTR, ADDR, and CTRL registers. Software should first write the instruction to be sent to the INSTR register and the address to be sent to the ADDR register (if required) followed by a write to the CTRL register to start the transfer. The CTRL_TXRX bit indicates whether data should flow to or from the device and CTRL_XFERBYTES indicates the number of bytes to transfer. CTRL_SENDA and CTRL_SENDI can be used to enable or disable the instruction or address phases and the CTRL_ENTURN is used to enable the turnaround phase. The transfer will only commence if the CTRL_START bit is set. Software may read the BUSY and STATUS fields to check on transaction status, otherwise the CMDCPL interrupt (INTEN_CMDCPL) can be used to indicate completion.

```
AM_REG(MSPIn, INSTR) = instr;
AM_REG(MSPIn, CTRL) = AM_REG_MSPIn_CTRL_XFERBYTES(bytes) |
                      AM_REG_MSPIn_CTRL_SENDA_M |
                      AM_REG_MSPIn_CTRL_TXRX(1) |
                      AM_REG_MSPIn_CTRL_START_M;
```

Write latency, the time between the address and first data byte, can be set and controlled by setting the CTRL_ENWLAT bit to enable the Write Latency Counter, which sets the latency from 0 to 63 counter clocks with the setting in the DEV0CFG_Writelatency0 field.

For write (TX) operations, data should be written to the TXFIFO after the transaction has been started. Software should read TXENTRIES before writing to ensure that space is available in the FIFO before writing new TX data. For read operations, software should read the RXENTRIES to determine the number of words available and then read the data from the RXFIFO register.

```
// Example TX data write loop
for (i = 0; i < count; ) {
    temp1 = AM_REG(MSPI, TXENTRIES);
    for (; (temp1 < 16) && (i < count); temp1++, i++) {
        AM_REG(MSPIn, TXFIFO) = data[i];
    }
}
```

14.3 DMA Operations

Each MSPI controller tightly integrates the DMA controller with the transfer interface and automatically handles sequencing of instructions and address to serial flash device and the subsequent transfer of data to/from system memory. Before starting DMA operations, software should have already configured the DEV0CFG register (to specify device configuration) and the DEV0XIP register (to specify the template used for DMA operations). Software should first set up the static DMA parameters which specify the DMA burst parameters:

```
MSPIn(mspiModule)->DMATHRESH=8;           // Issue new DMA at FIFO half empty/full condition
MSPIn(mspiModule)->DMABCOUNT=32;          // burst count=32 bytes (8 words)
```

The MSPI implements a single FIFO for both TX and RX transfers as well as separate threshold values for RX/TX operations. In most cases, the DMATHRESH_DMATXTHRESH field should be set at 8 to indicate that a TX DMA (read from SRAM) will be triggered when the FIFO drops below eight entries and will trigger an RX DMA (write to SRAM) when the FIFO level reaches eight entries. The DMABCOUNT_BCOUNT field indicates the number of words that will be transferred each time that DMA is triggered. The DMA will also trigger automatically to flush or fill the FIFOs at the end of transfer if the total count is not a multiple of 32 bytes.

To initiate a DMA transfer, software should issue the following register operations:

```
MSPIn(mspiModule)->DMADEVADDR=(uint32_t) addr;           // set device address
MSPIn(mspiModule)->DMATARGADDR=(uint32_t) data;          // set address in system memory
MSPIn(mspiModule)->DMATOTCOUNT=(count<<2);             // set total number of bytes
MSPIn(mspiModule)->DMACFG=                                // enable DMA peripheral to memory
AM_REG_MSPI_DMACFG_DMAEN | AM_REG_MSPI_DMACFG_DMADIR_P2M;
```

When complete, the MSPIn will issue the DMACPL interrupt or software can monitor the status by reading the DMASTAT_DMATIP bit. For writes to peripherals, software should also check the INTSTAT_CMDCMP bit to ensure the transaction has finished. Transfers to the flash device are initiated by setting the DMACFG_DMADIR field to M2P (Memory to Peripheral).

Each controller will use the template in the DEV0XIP register to determine whether to send the instruction and address phases (XIPSENDI0, XIPSENDA0) and whether to insert turnaround cycles (DEV0XIP_XIPENTURNn). Instruction and address lengths are determined by the settings in the DEV0CFG register and the address and transfer count are set by the DMADEVADDR and DMATOTCOUNT registers. The instruction sent for read (RX) operations is specified in the READINSTR field of the DEV0INSTR register and likewise the WRITEINSTR field is used when transmitting data to the flash device.

If the AUTO DMA cannot be used because the device's characteristics don't fit into the template, software can issue PIO operations to initiate a more complex transfer setup and then enable DMA for just the bulk DMA portions using the DMAEN_EN instead of DMAEN_AUTO.

Optionally, the MSPI can turn off its power domain at the end of a DMA transfer if the DMAPWROFF bit is set in the DMACFG register. The domain will only power off once the entire DMA is complete (i.e. writes have been committed to system memory or have completed to the external flash device).

14.4 Execute in Place (XIP) Operations

The XIP mode of operation allows devices on the MSPI interface to be mapped into the flash cache's address space and appear as an extension to the internal flash array(s). Once enabled by the XIPEN bit in the DEV0XIP register, the flash/cache module will decode the address region and forward operations to the MSPI interface for completion. XIP mode uses the same configuration information as DMA mode and will automatically execute a cache line read fetch from the attached device and return it to the cache controller.

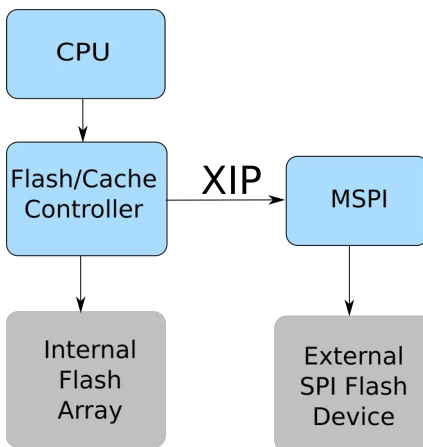


Figure 17. XIP Block Diagram

XIP and DMA/PIO operations can all be interleaved since the MSPI controller will allow the current operation to complete before performing the XIP operation. XIP may be interleaved within an ongoing DMA transfer as configured by either the DEV0BOUNDARY_DMATIMELIMIT0 transaction time or the DEV0BOUNDARY_DMABOUND0 address boundary.

Generally DMA read operations can safely be interleaved with XIP, however, XIP mode may have to be disabled during flash programming operations since the flash array within the device may not be available during program or erase operations and thus would return invalid data.

14.4.1 XIP Address Mapping and Accesses

The MSPI supports a memory-mapped XIP mode (XIPMM) that enables full read/write mapping of an MSPI device such as a PSRAM to the CPU's peripheral address map at offset 0x60000000-0x6FFFFFFF. The MSPI device can be accessed by both XIP and XIPMM accesses (the regions overlap).

XIPMM seamlessly supports word, halfword, and byte read and write accesses, however there are a few restrictions and caveats:

- Since the MSPI address regions are write-back by default in the M55 address space, the D-cache must be "cleaned" to force the data to be written back to the MSPI memory device.
- For scrambled regions, XIP can only be written safely by writing words (byte and halfword writes will corrupt the scrambled data at that location). Byte, halfword, and word reads may all be performed to scrambled regions.
- Read/Write performance to the XIP region will be significantly slower than accesses to internal SRAM since there are multiple cycles of command, addressing, and data transfer overhead. For this reason, internal SRAM should be used for frequently accessed data and XIPMM should be used for infrequently used data.

14.4.2 Optimized XIP Addressing

Some SPI flash devices support an optimized XIP mode that minimizes the number of instruction/address cycles that must be transmitted in order to reduce overall fetch latency. To activate this mode, software should program the flash device's registers to enter the device XIP mode, and then update the DEV0CFG_DEVCFG0 field to the specified number of address bytes and then disable the DEV0XIP_XIPSENDI0 field (assuming that no instruction needs to be sent). To exit the device's XIP mode, software should reconfigure the MSPI interface in order to send the required XIP exit sequence to the device.

14.4.3 Micron XIP Support

Micron flash devices support an XIP mode that does not require the instruction byte to be transmitted, which minimizes the access time to the device. In order to transition in and out of this mode, the MSPI controller must issue an acknowledgment of XIP mode during the first turnaround cycle for each XIP access. When transitioning into and out of XIP mode, software must set the DEV0XIP_XIPACK0 field appropriately.

Under normal operation, the XIPACK0 should be set to NOACK (0x0), indicating that no acknowledgment should be sent. To transition into XIP mode, software should perform the following actions:

1. Activate XIP in the Micron device by writing the Volatile Configuration register
2. Set the DEV0XIP_XIPACK0 bit field to ACK (0x2)
3. Perform a memory read from the Micron device (instruction must be sent). This access will allow the MSPI controller to acknowledge switching into the XIP mode
4. Set the DEV0XIP_XIPSENDI0 bit field to 0 to indicate that the instruction byte no longer needs to be sent.

The MSPI will now transmit just the address to the Micron device and drive a 0 onto the data lines on the first turnaround cycle to remain in XIP mode. It is important that software ONLY perform read operations to the flash device until XIP mode has been exited.

To terminate XIP mode, software should perform the following sequence:

1. Set the DEV0XIP_XIPACK0 bit field to TERMINATE (0x3)
2. Issue a memory read to the Micron device. This will allow the MSPI controller to signal termination of XIP mode by driving the data lines high during the first turnaround cycle.
3. Set the DEV0XIP_XIPACK0 bit field to NOACK (0x0) and the DEV0XIP_XIPSENDI0 bit field to 1.

After this sequence has completed, software can erase, program, or send any other instructions to the Micron flash again.

14.5 Command Queueing (CQ)

The MSPI's command queueing (CQ) interface is similar to command queueing implementation in the IOM and BLE modules. To utilize the command queue, software basically constructs a series of register operations that would be issued to the MSPI device, but instead places them in an array in system SRAM (or internal flash). The start of this buffer is then written to the CQADDR register and the commands can be issued by enabling the CQEN bit in the CQCFG register. The CQ logic then reads the address/data pairs via DMA operations and will continue executing them until the end of the command queue, which is denoted as a write to the STOP bit in the CQPAUSE register. As the CQ logic issues register operations, it will automatically pause fetching new operations while the transfer module is busy or can be paused to wait for external events based on the status of the CQPAUSE and CQFLAGS registers.

The primary limitation of CQ operations is that all addresses must reside within the MSPI module since the operations are executed internally by the MSPI module (i.e. it cannot write register in other modules, etc).

14.5.1 Command Queue Data Format

As the command queue resides in system memory, the general format is pairs of words that form the register address to write as well as the data to write. Assuming the CQ base address is 0x10000, system SRAM might look like the following table:

Table 72: Command Queue Example

Address	Data	Description
0x10000	0x50014258	DMATARGADDR register address
0x10004	0x00002800	Data to write to DMATARGADDR (i.e. 0x2800 is the target buffer)
0x10008	0x5001425C	DMADEVADDR register address
0x1000C	0x00304000	Address within flash device
0x10010	0x50014260	DMATOTCOUNT register address
0x10014	0x00000100	Transfer 256 bytes of data
0x10018	0x50014250	DMACFG register address
0x1001C	0x00000003	AUTO DMA enable on peripheral to memory transfer
0x10020	0x50014288	CQPAUSE register address
0x10024	0x00008000	End of Command Queue (write to STOP bit)

The AM_REG macros can be used to construct the CQ table in a manner similar to below:

```
uint32_t *cqptr = 0x10000;
*cqptr++ = AM_REG_ADDR(MSPI,DMADEVADDR);
*cqptr++ = devaddr; // set device address (for encryption)
*cqptr++ = AM_REG_ADDR(MSPI,DMATARGADDR);
*cqptr++ = data_buffer; // set source address in memory
*cqptr++ = AM_REG_ADDR(MSPI,DMATOTCOUNT);
*cqptr++ = 4*num_words; // set total number of bytes
*cqptr++ = AM_REG_ADDR(MSPI,DMACFG);
*cqptr++ = AM_REG_MSPI_DMACFG_DMAEN_AUTO |
            AM_REG_MSPI_DMACFG_DMADIR_M2P; // enable DMA write
*cqptr++ = AM_REG_ADDR(MSPI,CQPAUSE);
*cqptr++ = AM_REG_MSPI_CQFLAGS_STOP_M;
```

14.5.2 CQ Interrupts

The MSPI CQ module provides several interrupts via the INTEN register to provide feedback to software as the MSPI works through its command queue.

- **CQERR:** Indicates that the command queue encountered an error when fetching the command queue instructions. This can be caused by an invalid CQ pointer that points to an invalid flash or SRAM address (SRAM powered down, etc).
- **CQPAUSED:** Indicates that the command queue has encountered a pause condition. This can be triggered by an index match or when the CQ is waiting on a software or hardware flag.
- **CQCPL:** Indicates that the command queue has completed operations. This is typically used when the command queue is executing a single-shot set of commands which end with the CQ writing the STOP bit in the CQPAUSE register.
- **CQTIP:** indicate that a CQ transfer is active and this will remain active even when paused waiting for external event.

Software can generate a CQTIP interrupt at any point during command queue operation by setting bit[0] of the register address of the command to a 1 (basically OR 0x1 with the address portion of a CQ entry. This can be useful when software would like intermediate interrupts as operations complete such as after each CQ index is updated.

14.5.3 Pausing CQ Operations

While the basic operation of the CQ functionality is pretty straightforward, constructing more complex scenarios such as queuing of multiple operations requires additional logic to accommodate handshaking with the software managing the queue and other modules within the chip. The MSPI accomplishes both of these by providing the ability to pause the CQ processing using a pause mask (CQPAUSE register) and software and hardware pause flags.

After the MSPI executes a CQ write operation, it will check all bits specified in the CQPAUSE register against their CQFLAGS status, and will pause operation if all of the associated CQFLAGS bits are set. Since all registers are available to be written by both CPU software and CQ commands, there are numerous ways these can be used, but two common scenarios are

- Software can initially set a mask in CQPAUSE and CQ operation will continue until the matching CQFLAGS condition is encountered.
- The CQ command stream can set the CQPAUSE register during execution and pause until the status in FLAGS changes to indicate that it should restart.

The CQFLAGS register contains 8 soft flags (register bits that can be controlled by either the CPU or the QC operation) and an additional 812 hard flags, which are hardware status flags tied to logic in the MSPI module or other modules in the chip. The lowest two soft flags are also exported to the IOM SPI modules and the other two MSPI modules to facilitate communication between an IOM and the MSPI to enable management of common MSPI/IOM buffers via the command queues. Table 73 lists the flags available in the MSPI.

Table 73: CQFLAGS

Bit	Type	Mnemonic	Description/Use
15	Hard	STOP	CQ Stop Flag. When set to 1, CQ processing will terminate and the CQCPL interrupt will be generated.
14	Hard	CQIDX	CQ Index Pointer Match. Will be set to 1 when the CURIDX and ENDIDX pointers match. Generally used by software when forming a request queue.
13	Hard	BUF1XOREN	Buffer 1 Ready Status (from selected IOM/MSPI). This status is the result of XOR'ing the IOM1START with the incoming status from the IOM. When high, MSPI can transfer the buffer.

Table 73: CQFLAGS

Bit	Type	Mnemonic	Description/Use
12	Hard	BUF0XOREN	Buffer 0 Ready Status (from selected IOM/MSPI). This status is the result of XOR'ing the IOM0START with the incoming status from the IOM. When high, MSPI can transfer the buffer.
11	Hard	DMACPL	DMA Complete Status (hardwired DMACPL bit in DMASTAT)
10	Hard	CMDCPL	PIO Operation completed (STATUS bit in CTRL register)
9	Hard	IOM1READY/ BUF1XNOREN	Buffer 1 Ready Status. IOM Buffer 1 Ready Status (from selected IOM). This status is the result of XNOR'ing the IOM0START with the incoming status from the IOM. When high, MSPI can send to the buffer.
8	Hard	IOM0READY/ BUF0XNOREN	Buffer 0 Ready Status. IOM Buffer 0 Ready Status (from selected IOM). This status is the result of XNOR'ing the IOM0START with the incoming status from the IOM. When high, MSPI can send to the buffer.
7	Soft	SWFLAG7	Software flag
6	Soft	SWFLAG6	Software flag
5	Soft	SWFLAG5	Software flag
4	Soft	SWFLAG4	Software flag
3	Soft	SWFLAG3	Software flag
2	Soft	SWFLAG2	Software flag
1	Soft	IOM1START	Flag wired to IOM devices as a hard flag for intercommunication. Typically indicates that buffer 1 has been filled by MSPI and can be emptied by the IOM.
0	Soft	IOM0START	Flag wired to IOM devices as a hard flag for intercommunication. Typically indicates that buffer 0 has been filled by MSPI and can be emptied by the IOM.

The soft flags can be set/cleared/toggled via writes to the CQSETCLEAR register and their status can be read by software by reading the CQFLAGS register directly. The CQPAUSE mask bits are enumerated in the same manner.

In order to minimize the need to pause for individual operations, the CQ will automatically pause any time the MSPI's transfer block is active (for PIO, DMA, or XIP operations). Thus, whenever the CQ enables a DMA operation, there is an implicit pause until the operation completes, and then the CQ will resume fetching additional commands. To terminate the CQ processing, the CQ or software should set the most significant (unreserved) CQPAUSE register bit (STOP), which will cause the MSPI to terminate processing of the command queue and issue a CQCPL interrupt.

14.5.4 Using the CQ Index Registers

The MSPI command queuing implementation also includes a pair of registers that allow software to manage a list of outstanding operations: CQCURIDX and CQENDIDX. When initializing the command queue software can set both of these registers to the same value, which indicate an index or reference into the position of the command queue. The CQPAUSE can then be set to CQIDX and the command queue enabled. Since the CQCURIDX equals the CQENDIDX, the command queue will immediately pause and wait for them to be not equal again triggering resumption of CQ processing.

For each group of commands in the command queue, software can place a write to the CQCURIDX after each DMA operation in the command queue and then directly write the CQENDIDX register with the index of the last operation in the queue. Since the CQENDIDX now mismatches the CQCURIDX, the command queue will begin processing commands and start working its way through the queue. After completing the first operation, the command queue will include a write to the CQCURIDX to indicate that the operation has

completed, and the CQ logic will check to see if the CQCURIDX equals the CQENDIDX and either pause or continue processing until the two are equal again.

This mechanism allows software to asynchronously post additional operations to the command queue by simply writing the new commands to memory and then updating the CQENDIDX to the index of the last operation. Because the MSPI CQ hardware simply looks for a match between the registers, software may roll over from 0xFF to 0x00 or use the indices in any manner they see fit as long as the end index value is not found elsewhere in the command queue.

Software can monitor the progress of the MSPI's CQ processing by enabling the DMACPL interrupt, which will generate an interrupt after each DMA completion. The interrupt routine can read the CQCURIDX register to determine which operations have completed in order to return the proper status to the application.

14.5.5 MSPI and IOM Intercommunication

The MSPI modules and IOM modules can be linked through the command queue flags to allow a simple form of handshaking to facilitate data flow between the two modules. The MSPI only has a single pair of hardware flags dedicated to IOM communication so software must write the IOMSEL field in the MSPICFG register to select which IOM or MSPI device is paired with the MSPI.

Since MSPI is memory mapped, IOM can directly perform DMA reads/writes to an MSPI device instead of using the internal SRAM buffers. However, using an internal buffer allows for longer DMA bursts, which may be more efficient than direct addressing via DMA. Also MSPI devices can be set up to DMA or handshake with other MSPI devices in a manner similar to the IOM handshaking.

With the introduction of three MSPI modules on the MCU, the MSPI/IOM handshaking has been extended to support MSPI-to-MSPI handshaking as well. The target MSPI is selected in the same manner as an IOM in the IOMSEL field. The CQPAUSE flags on the MSPI have also been updated to include a BUFnXNOR function, which is required for an MSPI that is a consumer of data.

A typical use model for this feature is for transmitting blocks of data stored in external flash to a device (such as a display) on the IOM interface. In this scenario, software would allocate two buffers in SRAM which would be filled by the MSPI and emptied by the IOM/MSPI device. At the beginning of the operation, software would clear the IOM0START and IOM1START flags and initialize the MSPI command queue with two read operations to load data into buffer 0 and buffer 1. Software would also initialize the corresponding flags in the other device and set up the command queue to point to begin reading at buffer 0, but pause until it sees the buffer0 status is ready.

When the MSPI command queue is enabled, it will check the IOM0READY flag (which will be zero since the incoming bit is zero and the IOM0START flag is zero) and begin processing the operation which would DMA data from the external flash to fill buffer 0. At the end of the operation, the CQ would write the CQPAUSE register with the mask for IOM1READY. The status of IOM1READY will also be zero, so it will continue processing to fill buffer 1. At the end of this operation, the CQ will write the CQPAUSE register to IOM0READY again, but this time it will likely pause because the IOM is still reading data out of buffer 0. Once the IOM finishes its reads from buffer 0, its CQ will set the flag for buffer 0, which will in turn cause the IOM0READY hardware flag to become zero and allow the MSPI to continue processing (which would fill buffer 0 again). In this manner, software would only need to continue adding commands to the MSPI command queue in order to continuously feed data frames to the IOM device.

14.6 Data Scrambling

In order to protect customer data stored on external flash devices, the MSPI module supports a data scrambling algorithm to obfuscate data on the MSPI bus. Scrambling can be enabled by programming the DEV0SCRAMBLING_SCRSTART0 and DEV0SCRAMBLING_SCREND0 fields to correspond to the address range to be encrypted and setting the SCRENABLEn bit in the DEV0SCRAMBLING register. Scrambling is enabled for all DMA and XIP operations that fall within the scrambling window.

Accesses to the scrambling region must always be to an aligned, four-byte boundary (i.e. device address must always end in 0x0, 0x4, 0x8, 0xC). Accesses through the XIP region are always aligned to cache lines, but software must ensure that DMA operations are properly aligned. In the case of a mis-aligned DMA access, the MSPI will issue the SCRERR interrupt (SCRambling ERRor).

14.7 Auto Power Down

The MSPI module has the ability to power itself down at the end of a DMA or CQ operation. This would usually be done while the system is going into deep sleep but desires the MSPI to transfer data to or from a flash device during the beginning of the sleep period. To enable auto-power down, software should enable the DMA with the DMACFG_DMAPWROFF bit set or command queuing with the CQCFG_CQPWROFF bit set.

14.8 Board/Package Considerations for MSPI Pin Timing

The MSPI logic contains controls to adjust I/O timings to accommodate differences in board or device timings through the RXCAP0, RXNEG0, and TXNEG0 bits in the DEV0CFG register. The discussion below assumes SPI mode 0 (CPHA=0, CPOL=0) and that in dual/quad/octal modes that MOSI refers to all pins in transmission mode and MISO refers to all pins when in receive mode.

In an ideal model where there are no delays present, data is launched on the negative edge of the clock and captured on the positive edge of the clock at both the manager (MSPI) and subordinate (flash) device. However, the presence of delays in the system complicates the timing, and the timing diagram shown in Figure 18 demonstrates how transmission (TX) delays are accommodated in the MSPI interface design with a SCLK = 48 MHz or lower.

CLK (int) refers to the internal 96 MHz clock used by the MSPI, and SCLK (int) and MOSI (int) are the internal chip timings for the outgoing clock and MOSI lines. Likewise, the @ Dev signals indicate the SCLK and MOSI timing at the target device's pins. (Delays shown are just representative and may not reflect actual device timings.)

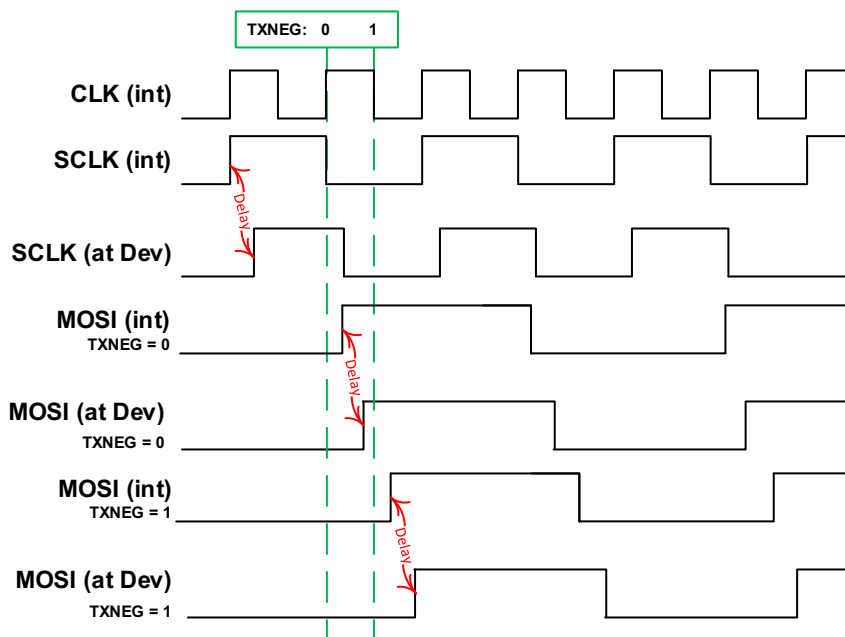


Figure 18. MSPI TX Interface Timing at SCLK = 48 MHz

Bit transmission from the MSPI to the target is fairly straightforward since both the SCLK and MOSI are delayed by similar amounts (red arrows). Depending on which pins are used, there may be some skew between the SCLK and MOSI, however, it should be relatively small compared to the half-cycle of setup time. If required, setting the TXNEG bit to 1 launches MOSI a half cycle (~5 ns) later, which is indicated by the “TXNEG = 1” dashed green line.

NOTE

The option to set TXNEG to 0 does not apply when SCLK = 96 MHz. Clocking at this SCLK rate requires TXNEG to be set to 1.

Subordinate to manager (MISO) timings on the MSPI interface are a bit more difficult to handle because of the cumulative round trip delay that consists of the clock delay from manager to subordinate, the access time at the target itself, and the return delay MISO path (red arrows). For this reason, read timings often dictate the frequency of the SPI bus.

When using a SCLK of 48 MHz or lower, the RXCAP and RXNEG bits may be used together to determine the incoming RX data capture point. In an ideal world (zero delays), the MSPI would capture data at the rising edge of the internal SCLK, which would correspond to the setting of RXCAP = 0 / RXNEG = 0 (the first vertical green dashed bar in Figure 19). It is useful, however, to push out the RX capture point to accommodate the late arrival of MISO. A setting of RXCAP = 0 / RXNEG = 1 delays the capture point by about 5 ns (one half period of the internal 96 MHz clock) as indicated by the second dashed green line.

The MSPI also supports RXCAP = 1 / RXNEG = 0 and RXCAP = 1 / RXNEG = 1 combinations, which delay capture of data by 10 ns and 15 ns, respectively.

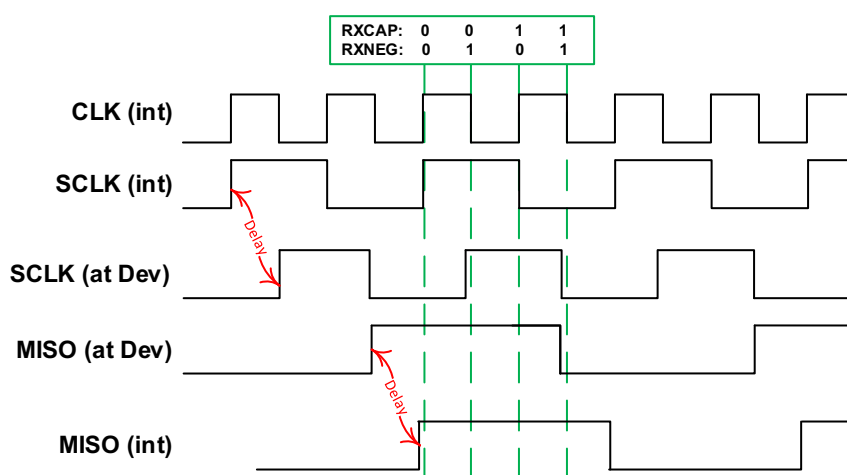


Figure 19. MSPI RX Interface Timing at SCLK = 48 MHz

With SCLK = 96 MHz (for an MSPI instance that supports that rate), the RXCAP delay is not available and the RXNEG bit is the single setting to lengthen the sampling delay time. The MSPI captures data at the rising edge of the internal SCLK with a setting of RXNEG = 0 (the first vertical green dashed line in Figure 20). To push out the RX capture point to accommodate a later arrival of MISO, a setting of RXNEG = 1 is used to delay the capture point by about 5 ns (one half period of the internal 96 MHz clock) as indicated by the second dashed green line.

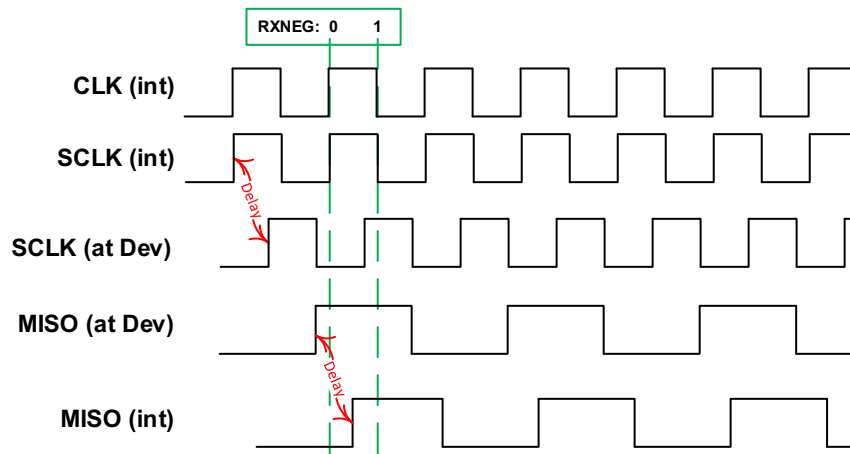


Figure 20. MSPI RX Interface Timing at SCLK = 96 MHz

14.9 MSPI Timing Diagrams with Delay Taps

The Delay steps described in this section each represent a delay of 80 to 200 ps / LSB as specified in MSPI section in the Electricals chapter of all Apollo5 family datasheets. The timing parameters specified in the datasheet are based on 0-tap delay on both TX and RX delay lines, regardless of the actual number of taps being programmed.

14.9.1 SDR with Non-DQS Mode

TX delay taps delay the SCLK as illustrated by the red dotted line in Figure . With more TX delay taps, $T_{\text{valid(out)}}$ is faster with respect to the delayed SCLK, but $T_{\text{hold(out)}}$ is faster as well.

An RX delay tap delays the internal sampling clock (not shown in the timing diagram). More RX delay taps allow a longer $T_{\text{setup(in)}}$ for an external device, but a shorter $T_{\text{hold(in)}}$.

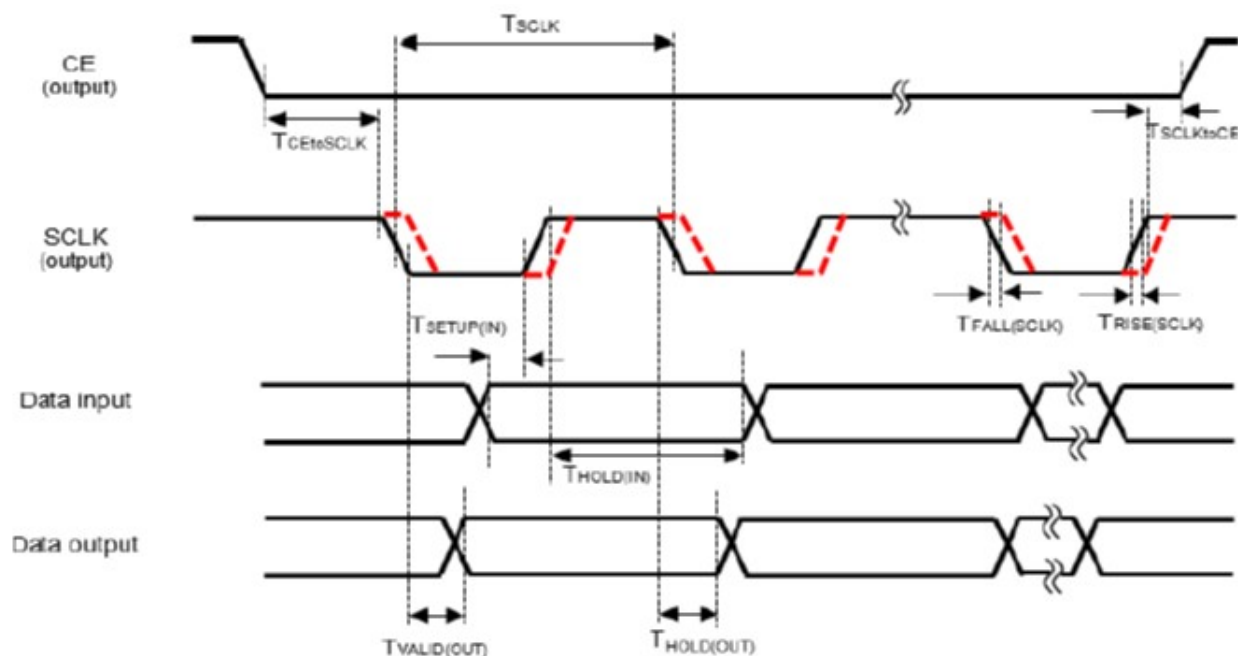


Figure 21. SDR with Non-DQS Mode Timing

14.9.2 DDR with DQS mode

TX delay taps (TXDQSDELAY0) delay the SCLK, as illustrated by the red dotted line in Figure 22. With more TX delay taps, $T_{\text{valid(out)}}$ is faster with respect to the delayed SCLK, but $T_{\text{hold(out)}}$ is faster as well.

An RX delay tap (RXDQSDELAY*0) delays the DQS input in the chip, illustrated by the red dotted line below. More RX delay taps allow a longer $T_{\text{setup(in)}}$ for an external device, but a shorter $T_{\text{hold(in)}}$.

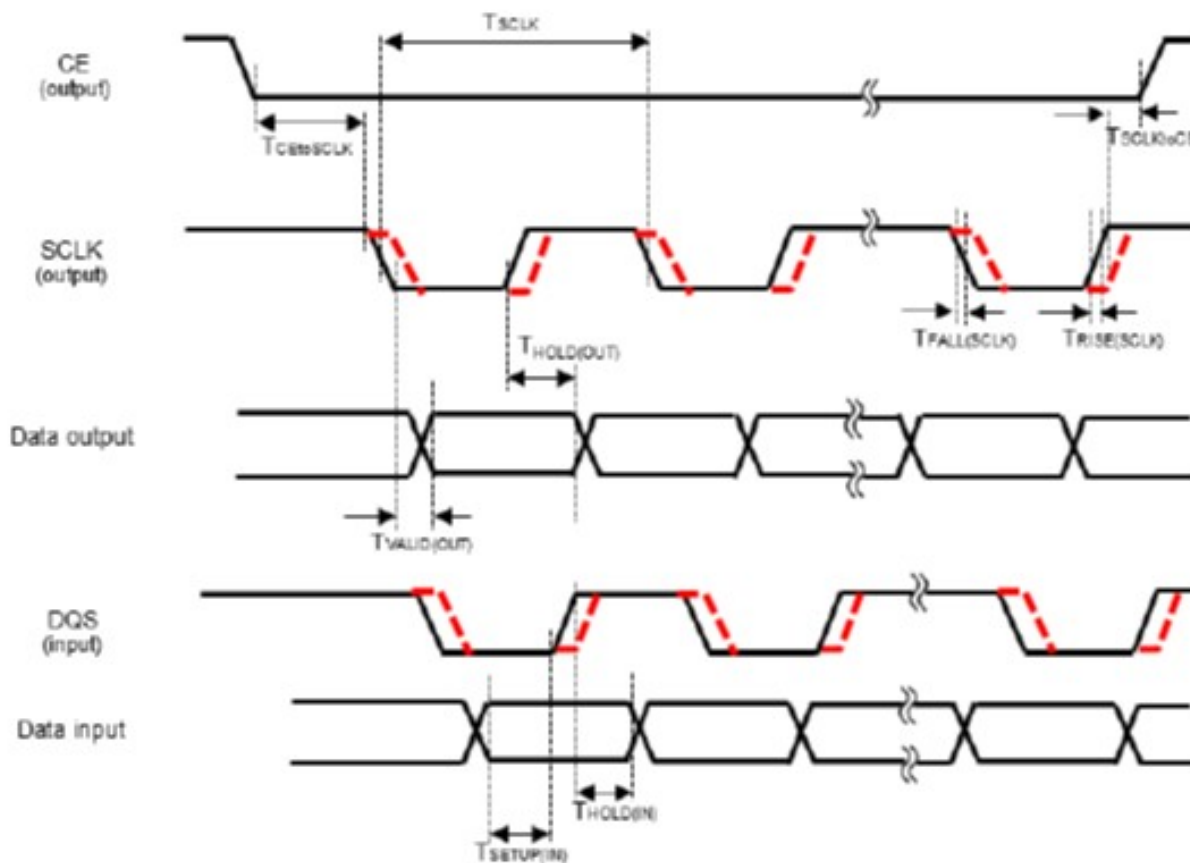


Figure 22. DDR with DQS Mode Timing

14.9.3 OCTAL/HEX DDR with DQS Mode: Read Delay Lines with Lower Byte DQS

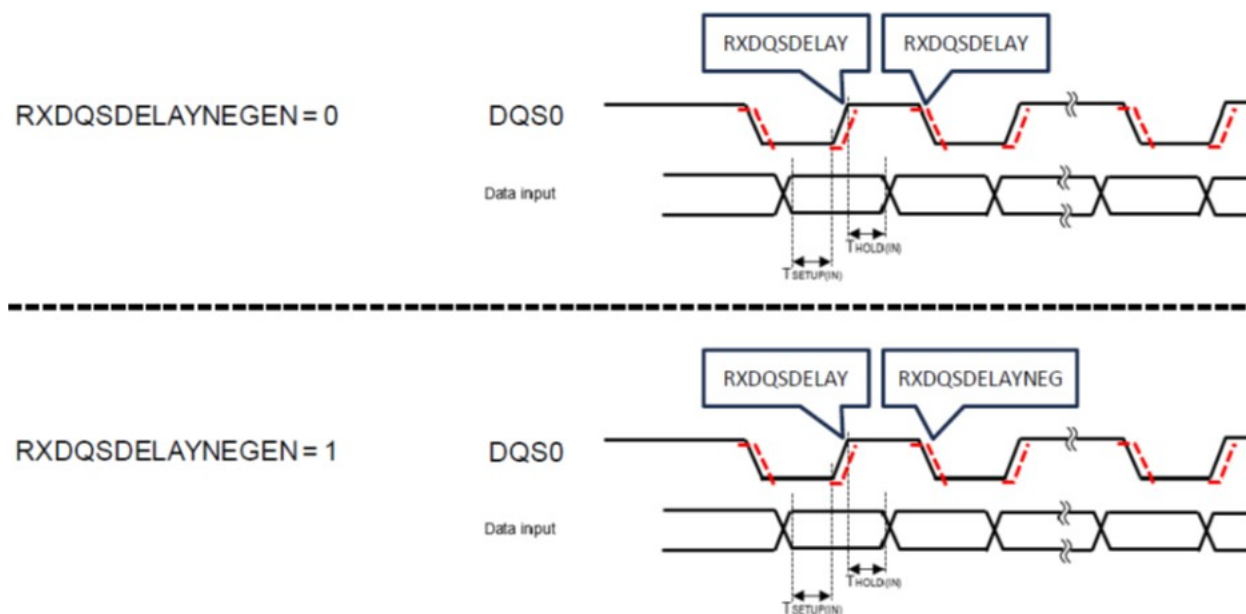


Figure 23. OCTAL/HEX DDR with DQS Mode Timing

14.9.4 HEX DDR with DQS Mode: Read Delay Lines for Upper Data Byte

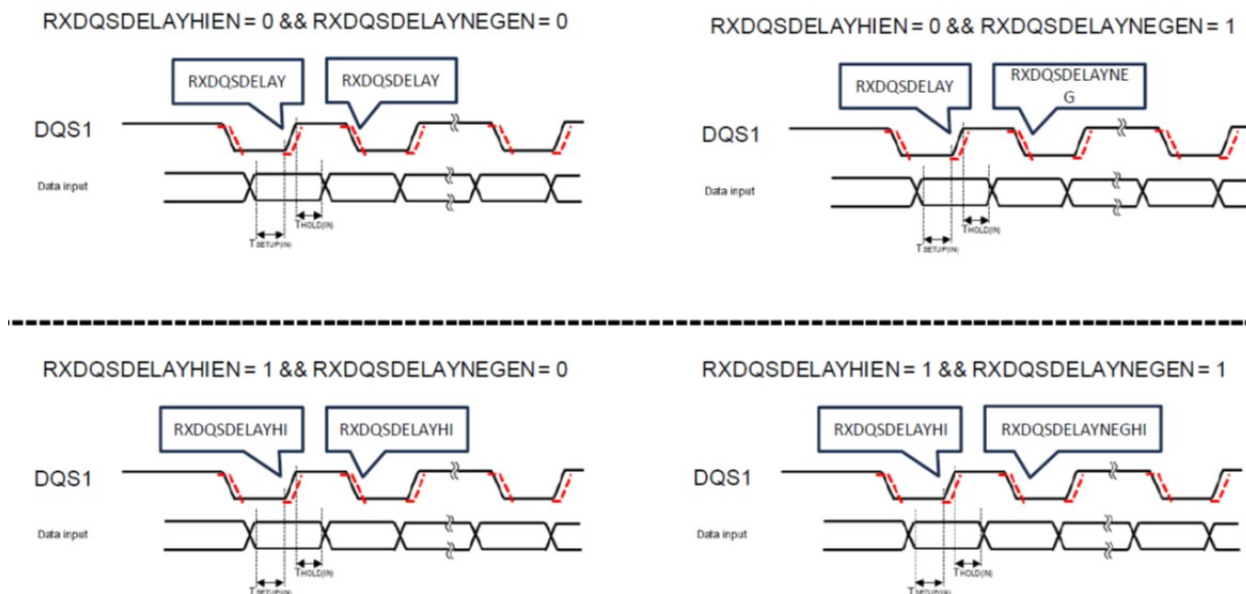


Figure 24. HEX DDR with DQS Mode Timing

15. I²C/SPI Manager (IOM)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

15.1 Programmer's Reference

An example register sequence to initiate an operation is shown below (note this does not show the data portion of the operation, only the command):

SPI SAMPLE OPERATION:

```
// Enable clock for 24MHz SPI operation
AM_REG(IOM,CLKFG) = ( 0 << AM_REG_IOM_CLKCFG_LOWPER_S ) |
    ( 0 << AM_REG_IOM_CLKCFG_TOTPER_S ) |
    ( 0 << AM_REG_IOM_CLKCFG_DIVEN_S ) |
    ( 1 << AM_REG_IOM_CLKCFG_DIV3_S ) |
    ( 1 << AM_REG_IOM_CLKCFG_FSEL_S ) |
    ( 1 << AM_REG_IOM_CLKCFG_IOCLKEN_S );

// Setup the SPI configuration register. MSB first, no flow control, not full duplex, mode 0
AM_REG(IOM, SPICFG) = ((0 << AM_REG_IOM_MSPICFG_MSPIRST_S) & AM_REG_IOM_MSPICFG_MSPIRST_M) |
    ((0 << AM_REG_IOM_MSPICFG_DOUTDLY_S) & AM_REG_IOM_MSPICFG_DOUTDLY_M) |
    ((0 << AM_REG_IOM_MSPICFG_DINDLY_S) & AM_REG_IOM_MSPICFG_DINDLY_M) |
    ((0 << AM_REG_IOM_MSPICFG_SPILSB_S) & AM_REG_IOM_MSPICFG_SPILSB_M) |
    ((0 << AM_REG_IOM_MSPICFG_RDFCPOL_S) & AM_REG_IOM_MSPICFG_RDFCPOL_M) |
    ((0 << AM_REG_IOM_MSPICFG_WTFCPOL_S) & AM_REG_IOM_MSPICFG_WTFCPOL_M) |
    ((0 << AM_REG_IOM_MSPICFG_WTFCIRQ_S) & AM_REG_IOM_MSPICFG_WTFCIRQ_M) |
    ((0 << AM_REG_IOM_MSPICFG_MOSIINV_S) & AM_REG_IOM_MSPICFG_MOSIINV_M) |
    ((0 << AM_REG_IOM_MSPICFG_RDFC_S) & AM_REG_IOM_MSPICFG_RDFC_M) |
    ((0 << AM_REG_IOM_MSPICFG_WTFC_S) & AM_REG_IOM_MSPICFG_WTFC_M) |
    ((0 << AM_REG_IOM_MSPICFG_FULLDUP_S) & AM_REG_IOM_MSPICFG_FULLDUP_M) |
    ((0 << AM_REG_IOM_MSPICFG_SPHA_S) & AM_REG_IOM_MSPICFG_SPHA_M) |
    ((0 << AM_REG_IOM_MSPICFG_SPOL_S) & AM_REG_IOM_MSPICFG_SPOL_M);

// Send a read command (2) of size 0x20 using 1 byte offset of 0x32 to device on CEN
AM_REG(IOM, CMD) = (( 2 << AM_REG_IOM_CMD_CMD_S) & AM_REG_IOM_CMD_CMD_M) | // READ COMMAND
    (( 0 << AM_REG_IOM_CMD_CMDSEL_S) & AM_REG_IOM_CMD_CMDSEL_M) |
    (( 0x20 << AM_REG_IOM_CMD_TSIZE_S) & AM_REG_IOM_CMD_TSIZE_M) |
    (( 0 << AM_REG_IOM_CMD_CONT_S) & AM_REG_IOM_CMD_CONT_S) |
    (( 1 << AM_REG_IOM_CMD_OFFSETCNT_S) & AM_REG_IOM_CMD_OFFSETCNT_M) |
    (( 0x32 << AM_REG_IOM_CMD_OFFSETLO_S) & AM_REG_IOM_CMD_OFFSETLO_M);
```

15.2 Interface Clock Generation

The I²C/SPI Manager (IOM) can generate a wide range of I/O interface clocks, as shown in Figure 25. The source clock is a scaled version of the HFRC 96 MHz clock, selected by IOMn_CLKCFG_FSEL. A divide-by-3 circuit may be selected by IOMn_CLKCFG_DIV3, which is particularly important in creating a useful SPI frequency of 32 MHz. The output of the divide-by-3 circuit may then be divided by an 8-bit value, IOMn_CLKCFG_TOTPER + 1, to produce the interface clock. This structure allows very precise specification of the interface frequency, and produces a minimum available interface frequency of 1.2 kHz.

If TOTPER division is enabled by IOMn_CLKCFG_DIVEN, the length of the low period of the clock is specified by IOMn_CLKCFG_LOWPER + 1. Otherwise, the clock will have a 50% duty cycle.

NOTE

Selection of HFRC96MHZ for IOMn_CLKCFG_FSEL is not supported.

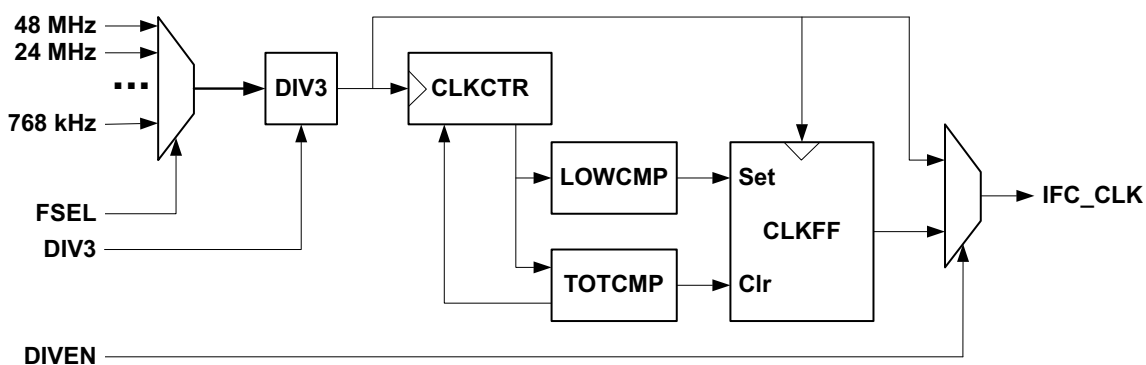


Figure 25. I²C/SPI Manager Clock Generation

15.3 Command Operation

In order to minimize the amount of time the CPU must be awake during I²C/SPI Manager operations, the architecture of the I²C/SPI Manager is organized around processing commands which transfer data to and from an internal 64-byte FIFO.

The IOMn_CMD register is used for command operations for both the SPI and I2C communication channels.

For writes to the interface, software writes data to the FIFO (IOMn_FIFO) and then sends a single command to the IOMn_CMD Register. Unless the TSIZE field of the CMD is zero, at least one word (4 bytes) of data must be written into the FIFO prior to writing the CMD Register or an ICMD interrupt will be generated and the operation will be terminated. The Command includes either the I²C Subordinate address or the SPI channel select, the desired address offset and the length of the transfer. At that point the I²C/SPI Manager executes the entire transfer, so the CPU can go to sleep. If more than 64 bytes are to be transferred, the manager will generate a THR interrupt when the FIFOSIZ value, IOMn_FIFOPTR_FIFOSIZ, drops below the write threshold IOMn_FIFOTHR_FIFOWTHR so the CPU can wake up and refill the FIFO. The I²C/SPI Manager will generate the CMDCMP interrupt when the command is complete. In each case, the total number of bytes transferred in each operation is specified in the LENGTH field of the CMD Register. If software executes a write to the FIFO when it is full (FIFOSIZ is greater than 124) the FOVFL interrupt will be generated and the transfer will be terminated.

For reads, the CMD Register is first written with the command and the CPU can go to sleep. The manager initiates the read and transfers read data to the FIFO. If the FIFOSZ value exceeds the read threshold IOMn_FIFOTHR_FIFOTHR, a THR interrupt is generated so the CPU can wake up and empty the FIFO. A CMDCMP interrupt is also generated when the Command completes. If software executes a read from the FIFO when it has less than a word of data the FUNDFL interrupt will be generated and the transfer will be terminated. FUNDFL will not be generated if the read transfer has already completed, so that software can read the last FIFO word even if it is incomplete.

If the FIFO empties on a write or fills on a read, the I²C/SPI Manager will simply pause the interface clock until the CPU has read or written a byte from the FIFO. This avoids the requirement that the thresholds be set conservatively so that the processor can wake up fewer times on long transfers without a risk of an underflow or overflow aborting a transfer in progress.

If software initiates an incorrect operation, such as attempting to read the FIFO on a write operation or when it is empty, or write the FIFO on a read operation or when it is full, the manager will generate an IACC error interrupt. If software attempts to write the Command Register when another Command is underway or write the CMD register with a write command when the FIFO is empty (unless the LENGTH field in the CMD is zero), the manager will generate an ICMD error interrupt.

15.4 FIFO

The I²C/SPI Manager includes a 64-byte local RAM (LRAM) for data transfers. The LRAM functions as a FIFO. Only 32-bit word accesses are supported to the FIFO from the CPU. When a write operation is underway, a word written to the FIFO will increment the IOMn_FIFOPTR_FIFOnSIZ register by 4 and decrement the IOMn_FIFOPTR_FIFOnREM register by 4. Reading a byte from the FIFO via the I/O interface decrements FIFOnSIZ by 1 and increments FIFOnREM by 1. When a read operation is underway, a word read from the FIFO decrements FIFOnSIZ by 4 and increments FIFOnREM by 4. A byte read from the I/O interface into the FIFO increments FIFOnSIZ by 1 and decrements FIFOnREM by 1. If FIFOnSIZ becomes one during a write operation or 0x40 on a read operation and there is more data to be transferred, the clock of the I/O interface is paused until software accesses the FIFO.

Two threshold registers, FIFOTHR and FIFOWTHR indicate when a THR interrupt should be generated to signal the processor that data should be transferred.

15.5 I²C Interface

The I²C/SPI Manager supports a flexible set of Commands to implement a variety of standard I²C operations. The I²C interface consists of two lines: one bi-directional data line (SDA) and one clock line (SCL). Both the SDA and the SCL lines must be connected to a positive supply voltage via a pull-up resistor. By definition, a device that sends a message is called the “transmitter”, and the device that accepts the message is called the “receiver”. The device that controls the message transfer by driving SCL is called “manager”. The devices that are controlled by the manager are called “subordinates”. The I²C Manager is always a manager device.

The following protocol has been defined:

- Data transfer may be initiated only when the bus is not busy.
- During data transfer, the data line must remain stable whenever the clock line is high.
- Changes in the data line while the clock line is high will be interpreted as control signals.

A number of bus conditions have been defined (see Figure 26) and are described in the following sections.

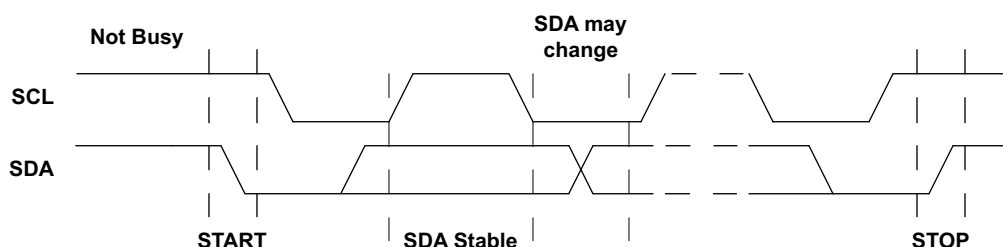


Figure 26. Basic I²C Conditions

I²C operations may transfer up to 512 bytes in a single transfer.

15.5.1 Bus Not Busy

Both SDA and SCL remain high.

15.5.2 Start Data Transfer

A change in the state of SDA from high to low, while SCL is high, defines the START condition. A START condition which occurs after a previous START, but before a STOP, is called a RESTART condition, and functions exactly like a normal STOP followed by a normal START.

15.5.3 Stop Data Transfer

A change in the state of SDA from low to high, while SCL is high, defines the STOP condition.

15.5.4 Data Valid

After a START condition, SDA is stable for the duration of the high period of SCL. The data on SDA may be changed during the low period of SCL. There is one clock pulse per bit of data. Each data transfer is initiated with a START condition and terminated with a STOP condition. The number of data bytes transferred between the START and STOP conditions is not limited. The information is transmitted byte-wide and each receiver acknowledges with a ninth bit.

15.5.5 Acknowledge

Each byte of eight bits is followed by one acknowledge (ACK) bit as shown in Figure 27. This acknowledge bit is a low level driven onto SDA by the receiver, whereas the manager generates an extra acknowledge related SCL pulse. A subordinate receiver which is addressed is obliged to generate an acknowledge after the reception of each byte. Also, on a read transfer, a manager receiver must generate an acknowledge after the reception of each byte that has been clocked out of the subordinate transmitter. The device that acknowledges must pull down the SDA line during the acknowledge clock pulse in such a way that the SDA line is a stable low during the high period of the acknowledge related SCL pulse. A manager receiver must signal an end-of-data to the subordinate transmitter by not generating an acknowledge (a NAK) on the last byte that has been clocked out of the subordinate. In this case, the transmitter must leave the data line high to enable the manager to generate the STOP condition. If I/O Host attempts an I²C operation but no subordinate device generates an ACK, or if a subordinate fails to generate an ACK on a data byte before the transfer is complete, a NAK interrupt will be generated.

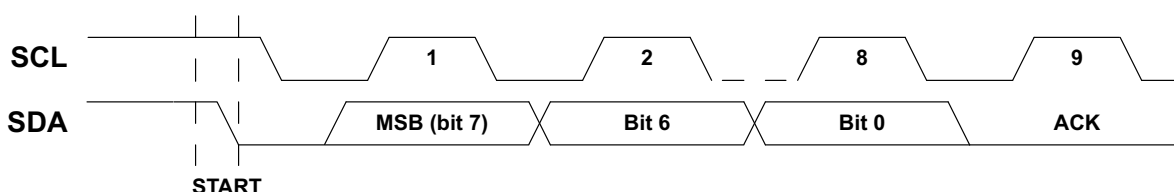


Figure 27. I²C Acknowledge

15.5.6 I²C Subordinate Addressing

For normal I²C reads and writes, the Command specifies the address to be sent on the interface. Both 7-bit and 10-bit addressing are supported, as selected by 10BIT in the Command. The address is specified in the ADDRESS field.

Figure 28 shows the operation in 7-bit mode in which the manager addresses the subordinate with a 7-bit address configured as 0xD0 in the lower 7 bits of the ADDRESS field. After the START condition, the 7-bit address is transmitted MSB first. If this address matches the lower 7 bits of an attached subordinate device, the eighth bit indicates a write (RW = 0) or a read (RW = 1) operation and the subordinate supplies the ACK. If no subordinate acknowledges the address, the transfer is terminated and a NAK error interrupt is generated.

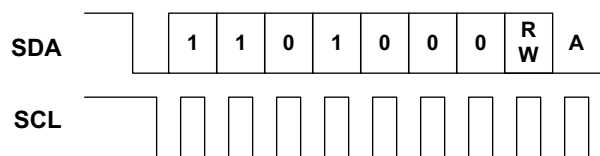


Figure 28. I²C 7-bit Address Operation

Figure 29 shows the operation with which the manager addresses the subordinate with a 10-bit address configured at 0x536. After the START condition, the 10-bit preamble 0b11110 is transmitted first, followed by the upper two bits of the ADDRESS field and the eighth bit indicating a write (RW = 0) or a read (RW = 1) operation. If the upper two bits match the address of an attached subordinate device, it supplies the ACK. The next transfer includes the lower 8 bits of the ADDRESS field, and if these bits also match

I2CADDR the subordinate again supplies the ACK. If no subordinate acknowledges either address byte, the transfer is terminated and a NAK error interrupt is generated.

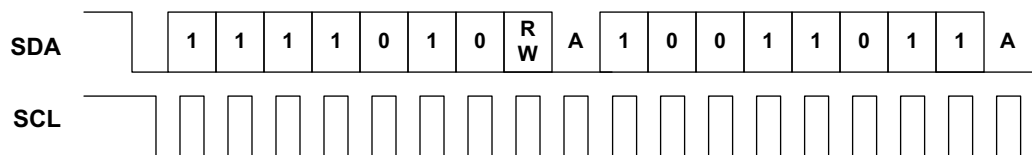


Figure 29. I²C 10-bit Address Operation

15.5.7 I²C Offset Address Transmission

If the OFFSETCNT field of the CMD register specifies that there is at least one byte of address offset for either a read or write command, then the I²C/SPI Manager will first send one or more 8-bit Offset Address bytes, where the offset is specified in the OFFSETLO field of the CMD register, and the OFFSETHI field of the OFFSETHI register if multiple offset bytes have been specified.

This transfer is shown in Figure 30. The Offset Address is loaded into the Address Pointer of the subordinate.

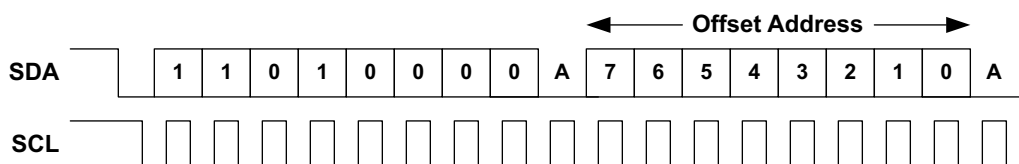


Figure 30. I²C Offset Address Transmission

15.5.8 I²C Write Operation with Address Offset

In a write operation the I²C/SPI Manager transmits to a subordinate receiver. The Address Operation has a RW value of 0, and the second byte contains the Offset Address, as in Figure 30. The next byte is written to the subordinate register selected by the Address Pointer (which was loaded with the Offset Address) and the Address Pointer is incremented. Subsequent transfers write bytes into successive registers until a STOP condition is received, as shown in Figure 31.

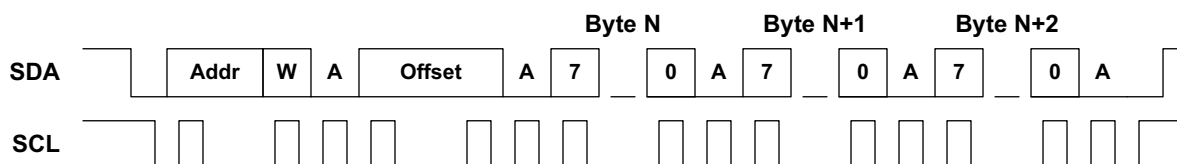


Figure 31. I²C Write Operation with Address Offset

15.5.9 I²C Read Operation with Address Offset

If a read operation with address offset is selected, the I²C/SPI Manager first executes an Offset Address Transmission to load the Address Pointer of the subordinate with the desired Offset Address.

A subsequent operation will again issue the address of the subordinate but with the RW bit as a 1 indicating a read operation. As shown in Figure 32, this transaction begins with a RESTART condition so that the interface will be held in a multi-manager environment. After the address operation, the subordinate becomes the transmitter and sends the register value from the location pointed to by the Address Pointer, and the Address Pointer is incremented. Subsequent transactions produce successive register values, until the I²C/SPI Manager receiver responds with a NAK and a STOP to complete the operation.

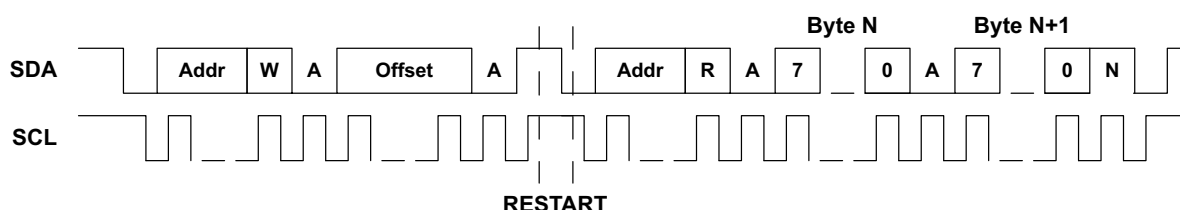


Figure 32. I²C Read Operation with Address Offset

15.5.10 I²C Write Operation with No Address Offset

If a write with no address offset is selected in the CMD and OFFSETCNT fields of the CMD register, the I²C/SPI Manager does not execute the Offset Address Transmission, but simply begins transferring bytes as shown in Figure 33. This provides support for subordinate devices which do not implement the standard offset address architecture. The OFFSETHI and OFFSETLO fields are not used in this case.

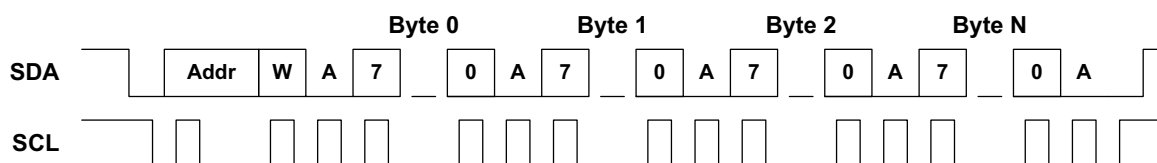


Figure 33. I²C Write Operation with No Address Offset

15.5.11 I²C Read Operation with No Address Offset

If a read with no address offset is selected in the CMD and OFFSETCNT fields of the CMD register, the I²C/SPI Manager does not execute the Offset Address Transmission, but simply begins transferring bytes with a read as shown in Figure 34. This is important for subordinate devices which do not support an Address Pointer architecture. For devices which do include an Address Pointer, multiple reads with no address offset may be executed after a read with address offset to access subsequent registers as the Address Pointer increments, without having to execute the Offset Address Transmission for each access.

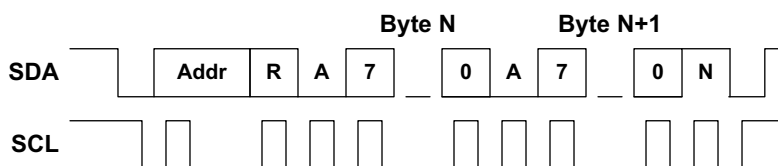


Figure 34. I²C Read Operation with No Address Offset

15.5.12 Holding the Interface with *CONT*

In all of the previously described transactions, the I²C/SPI Manager terminates the I²C operation with a STOP sequence. In environments where there are other managers connected to the I²C interface, it may be necessary for the MCU to hold the interface between Commands to insure that another manager does not inadvertently access the same subordinate that the MCU is accessing. In order to implement this functionality, the CONT bit should be set in the CMD Register. This will cause the I²C/SPI Manager to keep SDA high at the end of the transfer so that a STOP does not occur, and the next transaction begins with a RESTART instead of a START. Note that for a Normal Read the interface is held between the Offset Address Transmission and the actual read independent of the state of CONT, but if CONT is set the read transaction will not terminate with a STOP.

15.5.13 I²C Multi-Manager Arbitration

The I²C/SPI Manager supports multi-manager arbitration in I²C mode. There are two cases which must be handled.

The first is the case where another manager initiates an I²C operation when the MCU manager is inactive. In this case the I²C/SPI Manager will detect an I²C START operation on the interface and the START interrupt will be asserted, which tells the software not to generate any IO operations (which will not be executed in any case). Software then waits for the STOP interrupt, which reenables operation.

The second case is where another manager initiates an operation at the same time as the MCU. In this case there will be a point where one manager detects that it is not driving SDA low but the bus signal is low, and that manager loses the arbitration to the other manager. If the MCU I²C/SPI Manager detects that it has lost arbitration, it will assert the ARB interrupt and immediately terminate its operation. Software must then wait for the STOP interrupt and re-execute the current Command.

15.6 SPI Operations

15.6.1 SPI Configuration

The I²C/SPI Manager supports all combinations of the polarity (CPOL) and phase (CPHA) modes of SPI using the IOMn_MSPICFG_SPOL and IOMn_MSPICFG_SPHA bits. It also may be configured in either 3-wire or 4-wire mode.

In 4-wire mode, the MOSI and MISO interface signals use separate IO pins. In this mode the GPIO used for the MOSI signal should be configured with the GPIO_PINCFGn_FOENn bit set to force output enable active.

In 3-wire mode, MOSI and MISO are multiplexed on a single IO pin for more efficient pin utilization. The 3/4 wire configuration is selected in the mapping function of the PINCFG module.

SPI operations may transfer up to 4095 bytes in a single transfer, as the TSIZE field in the CMD register provides a 12-bit length specification.

15.6.2 SPI Subordinate Addressing

In SPI mode, the Command specifies the subordinate channel to be used in the CMDSEL field. The I²C/SPI Manager supports up to four subordinates, each of which has its own nCE signal which can be configured on an IO pin. Additional subordinates may be supported using GPIO pins and external decoding.

15.6.3 SPI Write with Address Offset

Figure 35 shows the case of a SPI Write with a one-byte address offset operation, whereby a write operation is selected in the CMD field. The operation is initiated when the I²C/SPI Manager pulls one of the four nCE signals low. At that point the I²C/SPI Manager begins generating the clock on SCK and the offset address is transmitted from the manager on the MOSI line, with the upper R/W bit of the offset field indicating read (if 0) or write (if 1). In this example the R/W bit is a one selecting a write operation. The entire multi-byte offset, the length of which is specified by the OFFSETCNT field, is taken from the OFFSETLO field of the CMD and, depending on the value in OFFSETCNT, the OFFSETHI field in the OFFSETHI register. The MSB of the entire OFFSET should be set to 1 if the subordinate expects a RW bit. If the subordinate does not expect a RW bit, this allows the first byte of a write to be completely specified in the OFFSET field, and a single byte write in that case can be executed without requiring any data to be loaded in to the FIFO.

Each subsequent byte is read from the FIFO and transmitted. The operation is terminated when the I²C/SPI Manager brings the nCE signal high. Note that the MISO line is not used in a write operation and is held in the high impedance state by the I²C/SPI Manager.

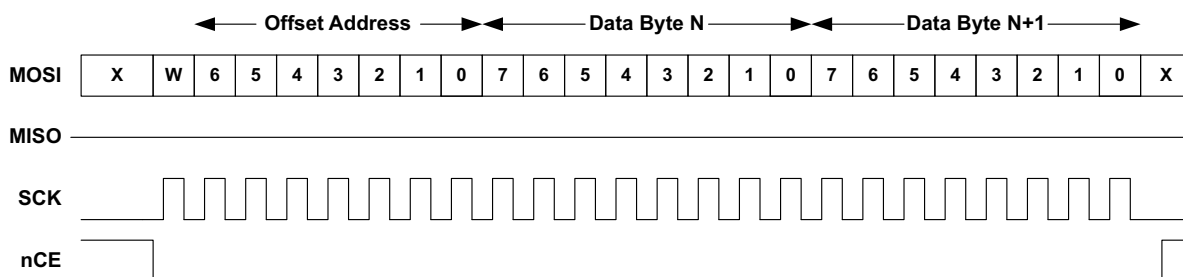


Figure 35. SPI Normal Write Operation (Single-byte Offset Address)

15.6.4 SPI Read with Address Offset

Figure 36 shows the case of a Read with a one-byte address offset operation, whereby a read operation is selected in the CMD field. The operation is initiated when the I²C/SPI Manager pulls one of the four nCE signals low. At that point the I²C/SPI Manager begins driving the clock onto SCK and the address is transferred from the manager to the subordinate just as it is in a write operation, but in this case the R/W bit is a 0 indicating a read. After the transfer of the last address bit (bit 0), the I²C/SPI Manager stops driving the MOSI line and begins loading the FIFO with the data on the MISO line. The transfer continues until the I²C/SPI Manager brings the nCE line high.

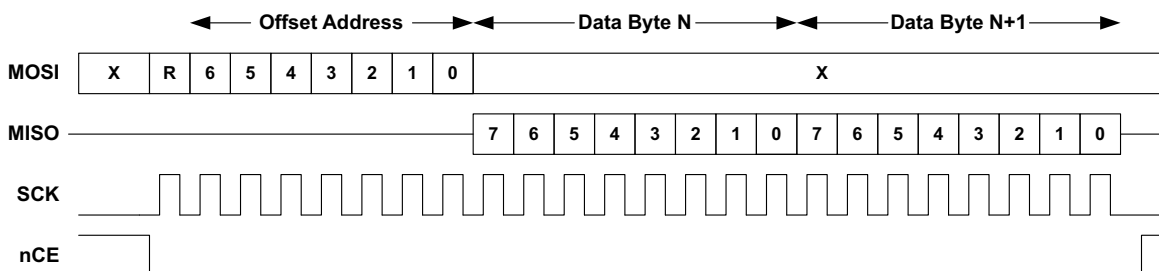


Figure 36. SPI Normal Read Operation

As with a Normal Write, the Offset Address byte including the R/W bit is taken from the offset field(s) of CMD. If the subordinate expects a R/W bit, the msb of the offset must be set accordingly. This allows reads from devices which have different formats for the address byte.

15.6.5 SPI Write with No Address Offset

If a write with no address offset is selected in the CMD field, the Offset Address byte is not sent and all data comes directly from the FIFO as shown in Figure 37. The OFFSET field is not used in this case.

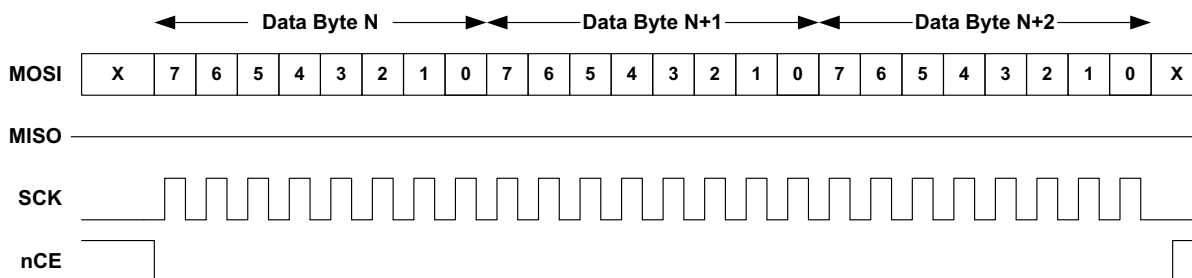


Figure 37. SPI Raw Write Operation

15.6.6 SPI Read with No Address Offset

If a read with no address offset is selected in the CMD field, data goes directly to the FIFO as shown in Figure 38. The OFFSET field is not used in this case.

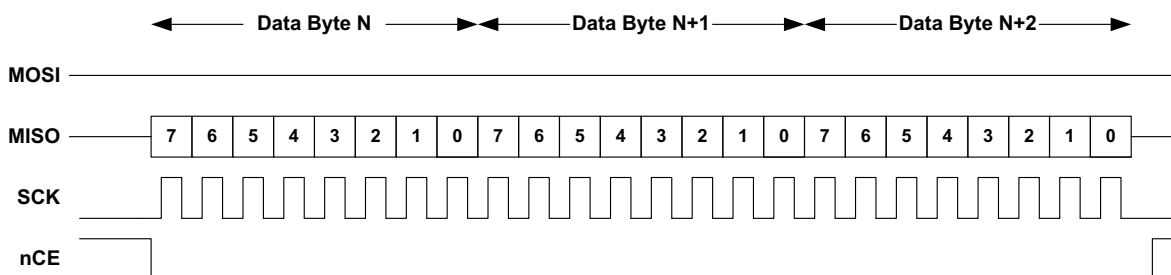


Figure 38. SPI Raw Read Operation

15.6.7 SPI 3-wire Mode

In 3-wire mode, the MOSI and MISO lines are shared on a single pin. As described in the previous sections, the MISO and MOSI lines are not driven at the same time, so 3-wire mode is equivalent to simply tying them together external to the MCU. 3-wire mode is configured by selecting the MxWIR3 alternative (x = 0 to 5 selecting the I2C/SPI Manager) in the GPIO Pad Multiplexor rather than the MxMOSI and MxMISO alternatives. Detailed configuration information is supplied in the GPIO and Pad Configuration Module chapter.

15.6.8 Complex SPI Operations

In some cases peripheral devices require more complex transaction sequences than those supported by a single Command. In order to support these transactions, the CONT bit may be set in the Command. In this case, the nCE pin selected by the Channel will remain asserted low at the end of the transaction, so that the next SPI operation will be seen as part of the same transaction. For example, there are peripheral devices which require both a Function and an Address Offset to be transmitted at the beginning of a read. Implementing this can be done in several ways. One example as shown in Figure 39 is:

1. Execute a Raw SPI write of length 2, with the data bytes being the Function and Offset. Set the CONT bit in this Command so nCE remains asserted low.
2. Execute a Raw SPI Read of the desired transfer length. The data will then be read into the FIFO. The CONT bit is not set in this Command.

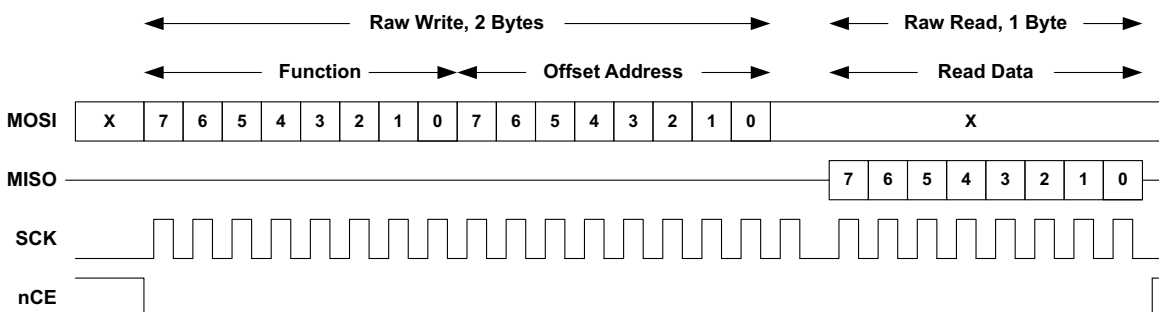


Figure 39. SPI Combined Operation

15.6.9 SPI Polarity and Phase

The MCU supports all combinations of CPOL (clock polarity) and CPHA (data phase) in SPI mode, as defined by the SPOL and SPHA bits. Figure 40 shows how these two bits affect the interface signal behavior.

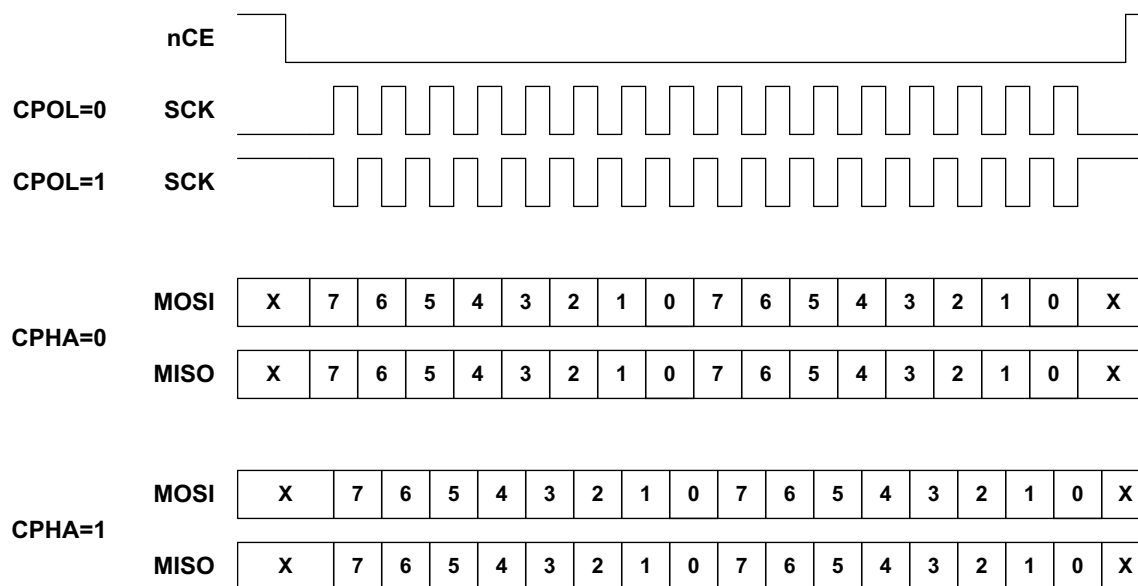


Figure 40. SPI CPOL and CPHA

If CPOL is 0, the clock SCK is normally low and positive pulses are generated during transfers. If CPOL is 1, SCK is normally high and negative pulses are generated during transfers.

If CPHA is 0, the data on the MOSI and MISO lines is sampled on the edge corresponding to the first SCK edge after nCE goes low (i.e. the rising edge if CPOL is 0 and the falling edge if CPOL is 1). Data on MISO and MOSI is driven on the opposite edge of SCK.

If CPHA is 1, the data on the MOSI and MISO lines is sampled on the edge corresponding to the second SCK edge after nCE goes low (i.e. the falling edge if CPOL is 0 and the rising edge if CPOL is 1). Data on MISO and MOSI is driven on the opposite edge of SCK.

The SPOL and SPHA bits may be changed between Commands if different subordinate devices have different requirements. In this case the SUBMODCTRL_SMODnEN bit should be set to 0 either before or at the same time as SPHA and SPOL are changed, and then set back to 1 before CMD is written.

15.7 Bit Orientation

In both I²C and SPI modes, the I²C/SPI Manager supports data transmission either LSB first or MSB first as configured by the LSB bit in the Command. If LSB is 0, data is transmitted and received MSB first. If LSB is 1, data is transmitted and received LSB first.

15.8 SPI Flow Control

The I²C/SPI Manager supports flow control from the subordinate, which is controlled by several configuration bits. Either read or write (or both) flow control may be implemented. Read flow control is enabled by setting the IOMn_MSPICFG_RDFC bit, in which case the I²C/SPI Manager will check the state of the Flow Control IRQ pin, and if it is inactive the SPI clock will stop at the completion of the current byte transfer until it becomes active. The Flow Control IRQ can be any of the 50 pins as selected by the GPIO_IOMnIRQ register corresponding to the particular I²C/SPI Manager. The polarity of the active state of the Flow Control IRQ is selected by the IOMn_MSPICFGn_RDFCPOL.

Write flow control is enabled by setting the IOMn_MSPICFGn_WTFC bit, but in this case either the Flow Control IRQ or the state of the MISO line may be used for flow control, as selected by the IOMn_MSPICFGn_WTFCIRQ bit. If IRQ is selected by setting a one, the clock control is identical to that described for reads above and the IRQ polarity is set by the IOMn_MSPICFGn_WTFCPOL bit. If MISO is selected by setting a zero in WTFCIRQ, the clock will be stopped if the MISO line is at the inactive polarity, which is set by the WTFCPOL bit.

Subordinate devices supporting flow control typically require specific states of the MOSI line prior to the start of a transfer. This state is controlled by the IOMn_MSPICFGn_MOSIINV bit. If this bit is zero, MOSI will be driven to a 1 at the start of a write transaction and to a 0 at the start of a read transaction – this is the normal operation of devices with flow control support. If MOSIINV is set to one, these polarities will be inverted.

Flow control may be asserted either prior to the first byte transfer, which will delay the start of SCK, or within each byte transferred, which will pause SCK at the end of that byte. The examples below assume that WTFCPOL or RDFCPOL are set to 0.

Figure 41 shows the operation of flow control at the beginning of a write transfer or a normal read transfer which begins with an offset byte write. Either MISO or IRQ (selected by WTFCIRQ) must be deasserted low within $\frac{1}{2}$ of the SCK period after nCE is asserted low in order to delay the clock. SCK will continue in its inactive state until MISO or IRQ is changed to the active state, and then will begin normal operation.

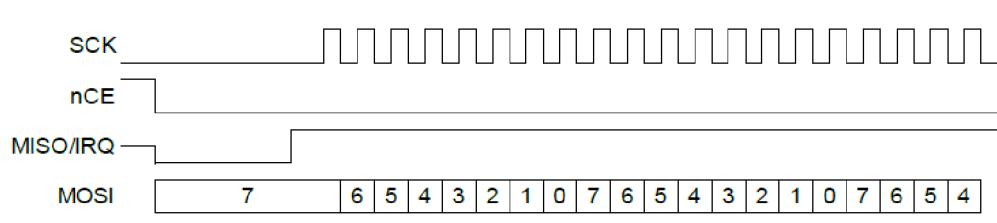


Figure 41. Flow Control at Beginning of a Write Transfer

Figure 42 shows the operation of flow control at the beginning of a raw read transfer. IRQ must be deasserted low within $\frac{1}{2}$ of the SCK period after nCE is asserted low in order to delay the clock. SCK will continue in its inactive state until IRQ is changed to the active state, and then will begin normal operation.

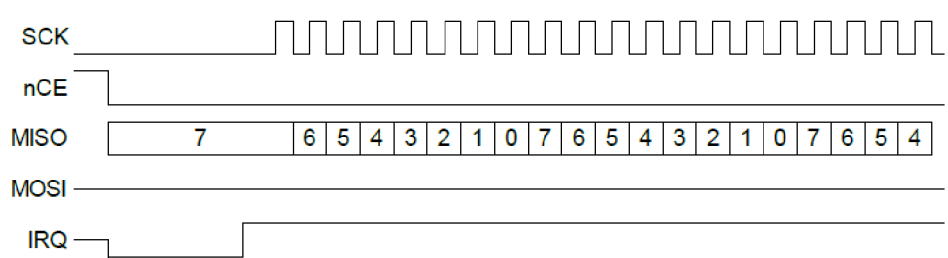


Figure 42. Flow Control at Beginning of a Raw Read Transfer

Figure 43 shows the operation of flow control in the middle of a write transfer. MISO or IRQ must be deasserted after the leading edge of SCK on the first bit of the byte (labeled 7) and before the falling edge of the 7th bit of the byte (labeled 1) in order to insure that SCK stops at the end of the byte. De-asserting MISO or IRQ outside of that window can produce unpredictable results. SCK will resume at some point after the assertion of MISO or IRQ.

Figure 43. Flow Control in the Middle of a Write Transfer

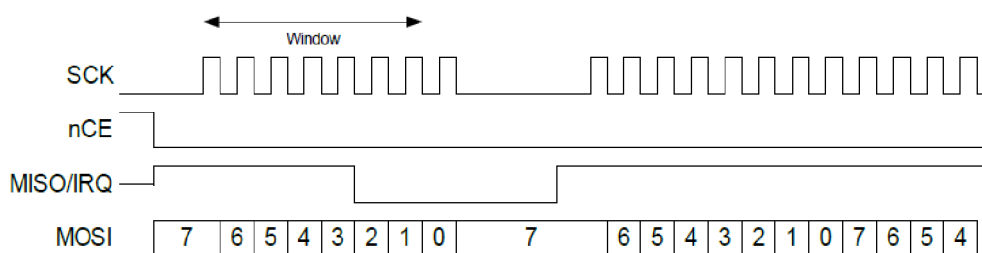


Figure 44 shows the operation of flow control in the middle of a read transfer. IRQ must be deasserted after the leading edge of SCK on the first bit of the byte (labeled 7) and before the falling edge of the 7th bit of the byte (labeled 1) in order to insure that SCK stops at the end of the byte. De-asserting IRQ outside of that window can produce unpredictable results. SCK will resume at some point after the assertion of IRQ.

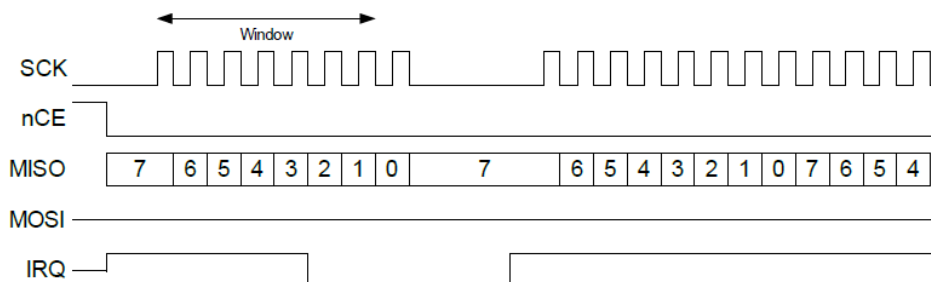


Figure 44. Flow Control in the Middle of a Read Transfer

15.9 Minimizing Power

Each I²C/SPI Manager submodule has an interface enable bit IOMn_SUBMODCTRL_SMODnEN. This bit should be kept at 0 (along with PWRCTRL_DEVPWREN_PWRIOMn fields) whenever the interface is not being used in order to minimize power consumption.

16. I²C/SPI Subordinate (IOS)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

16.1 Local RAM Allocation

The I²C/SPI Subordinate (IOS) is built around a 256-byte local RAM (LRAM), through which all data flows between the CPU AHB and the IO interface. The I²C/SPI Subordinate supports a 128-byte offset space when accessed from the I/O interface.

The LRAM is divided into three separate areas on 8-byte boundaries. These areas are:

1. A Direct Area for direct communication between the host and the SoC, which is mapped between the AHB address space and the I/O address space. This area is from LRAM address 0x00 to the address calculated from the 5-bit FIFOBASE field in the FIFO configuration register (FIFOCFG), minus 1. This 5-bit field (IOSUBORDINATE_FIFOCFG_FIFOBASE) should contain a value that represents the start of the FIFO Area and, in so doing, defines the size of the Direct Area in 8-byte segments. Part of this area can be defined as IO Subordinate Read-only starting at any 8-byte segment defined by IOSUBORDINATE_FIFOCFG_ROBASE and extending through the end of the Direct Area at FIFOBASE*8-1.
2. A FIFO Area which is used to stream data from the MCU. This memory is directly addressed from the AHB, but accessed from the I/O Interface using a single I/O address 0x7F as a streaming port. The FIFO area is from the LRAM address calculated from the value in the FIFOBASE field, FIFOBASE*8, to the LRAM address calculated from the value in the FIFOMAX field of the FIFOCFG register, IOSUBORDINATE_FIFOCFG_FIFOMAX. The upper FIFO Area address is FIFOMAX*8-1. The maximum value for FIFOMAX is 0x20, which would result in an upper FIFO Area address of 0xFF.
3. A RAM Area which is accessible only from the AHB Subordinate. The RAM area is from the LRAM address calculated from the value in the FIFOMAX field of the FIFOCFG register, IOSUBORDINATE_FIFOCFG_FIFOMAX, to address 0xFF. Setting FIFOMAX to 0x20 would result in a RAM area of zero size.

The data in the LRAM is maintained in Deep Sleep Mode.

Figure 45 below shows the LRAM address mapping between the I/O interface and the AHB.

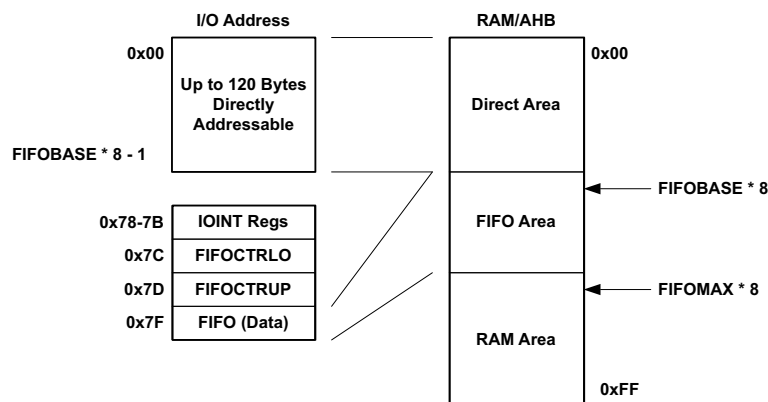


Figure 45. I²C/SPI Subordinate Module LRAM Addressing in Standard Address Mode

16.2 Direct Area Functions

The Direct Area is used for direct communications between the interface Host and the MCU. The Host may write a register in this Register Access space, called REGACC, and read it back without requiring the CPU to wake up, so that very low-power interactions are supported. In some cases, however, accesses require interaction with the CPU.

REGACC interrupts are mapped in the Direct Area and operate as follows. Each REGACC interrupt status bit will be set whenever there is a read or write over the I2C or SPI interface in the Direct Area with an offset address which corresponds to a particular REGACC interrupt. Table 74 in Section 16.2.2 lists the offsets to memory locations within the Direct Area and corresponding interrupt bit settings in the REGACCINTSTAT register.

I/O writes to locations 0x0-0xF will set a corresponding interrupt flag in the REGACCINTSTAT register. These locations are typically used for specific commands to the MCU. Note that not all flags need generate an actual interrupt, so small multi-byte commands may be transmitted in this area. For example, a write to location 0x0 will set bit 31 of the REGACCINTSTAT register, a write to location 0x1 will set bit 30 of REGACCINTSTAT, and a write to location 0xF will set bit 16 of the REGACCINTSTAT register.

The upper 16 REGACC interrupts are each generated on an access to a multi-byte range. In this standard mode, each upper 16 REGACC interrupt covers a range which is identified as the last byte of a 32-bit word, starting at 0x10. I/O writes to locations 0x10 to 0x4F will set a corresponding interrupt flag in the REGACCINTSTAT register if the I/O address modulo 4 is 3 (i.e., addresses 0x13, 0x17, 0x1B, etc.). This allows transfers to be sent in a burst with a trigger being generated on the last write, and it also allows specifying a data buffer of any whole word size to have an interrupt generated on access to the last byte of the buffer.

For example, a write to location 0x13 will set bit 15 of the REGACCINTSTAT register, a write to location 0x17 will set bit 14 of REGACCINTSTAT, and a write to location 0x4F will set bit 0 of the REGACCINTSTAT register. Note that in this mode there are no interrupts which would be set for word accesses in the transaction beyond the first 80 bytes (0x0 to 0x4F address range).

16.2.1 Host Burst Write Support

An unlimited-length host burst write mode, enabled by setting the CFG_WRAPPTR bit, is offered to extend a host write transaction length to any arbitrary size and with ability to generate interrupts on longer (8-byte and 16-byte) address ranges.

This mode re-purposes the FIFO and controls (from purely host read functionality) to be used for host burst write buffering. The size of the FIFO can be set to receive a section, or all, of a host burst transaction. The arbitrarily long transaction size is achieved by the hardware automatically wrapping around the address pointer to the beginning of the FIFO, FIFOBASE*8, after it hits the end of the FIFO at FIFOMAX*8-1, essentially creating a hardware ring buffer.

The FIFO base address, as set in the IOSUBORDINATE_FIFOCFG_FIFOBASE field, can establish a FIFO start address (and wrap-to address) on any 8-byte segment starting at 0x0 through 0xF8. Additionally, the address pointer automatically skips Direct Area locations 0x78 to 0x7F (if the FIFO Area encompasses these locations) to avoid writing to the Host Registers during a host burst write. The wrap-to address (=FIFOBASE) must not be defined as a location within the Host Register space (address 0x78 when IOSUBORDINATE_FIFOCFG_FIFOBASE = 0xF), as undefined results will occur.

For burst writes > 248 bytes, the end address of the FIFO, as defined by IOSUBORDINATE_FIFOCFG_FIFOMAX, should be set to maximize each iteration before wraparound occurs. This value is 248 (0xF8) bytes because the address pointer skips addresses 0x78-0x7F. Therefore FIFOMAX should be set to 0x100 to get the largest (0xF8) transfer before wraparound occurs.

A “mapped address” is created in LRAM whose offset is the same as the address pointer when (address pointer < 78), and is (address pointer - 8) when the address pointer is larger than 0x7F. In this mapping to

the LRAM, transfers to 0x80, for example, are mapped to LRAM address 0x78, and so on, to prevent a "hole" in the LRAM which would cause overhead in the software to work around this. The highest LRAM address which can be written in this mode is 0xF7.

With respect to REGACC interrupts, multi-byte ranges for each of the 16 higher address interrupts which occur upon Direct Area accesses are different sizes in this Host Burst Write Mode than for the standard Host Read Mode. This is as such to be able to cover the entire iteration of 0x0 to 0xF7 (248) bytes, with the 32 possible interrupts. After the first 16 single-byte locations, 0x0-0xF, for which each sets a corresponding interrupt flag (INT31-INT16) in the REGACCINTSTAT register, the multi-byte ranges are either 8 bytes or 16 bytes long. In Table 74 they are identified with the last byte of an 8-byte or 16-byte range, starting at Direct Area address 0x10. I/O writes to bytes 0x10 to 0xF7 in a 248-byte iteration of a transaction sets a corresponding interrupt flag in the REGACCINTSTAT register when the I/O address corresponds to the last byte location for any of the ranges. (i.e., addresses 0x17, 0x1F, 0x2F, etc.). In this mode, a write to location 0x17 will set bit 15 of the REGACCINTSTAT register, a write to location 0x1F will set bit 14 of REGACCINTSTAT, and a write to location 0xF7 will set bit 0 of the REGACCINTSTAT register.

Note that the application software must process the data loaded in the FIFO in this mode before a wraparound when new data will overwrite any existing data. As well, the application must service any enabled REGACC interrupts and re-arm interrupts on time and as required by the application before new data is loaded.

NOTE

IO Subordinate Read-Only address mapping, as specified by the IOSUBORDINATE_FIFOCFG_ROBASE setting, is not supported in Host Burst Write Mode (IOSUBORDINATE_CFG_WRAPPTR = 1).

16.2.2 Mapping of Direct Area Access Interrupts

Table 74 shows the memory locations within the Direct Area for each of the Direct Area access interrupt bits in the REGACCINTSTAT register. The lower-order register bits corresponding to multi-byte access locations are at different offsets for the Host Read Mode and the Host Burst Write Mode. This is to enable interrupts across the entire section of the longer burst write transactions.

Table 74: Direct Area Access Interrupt Mapping and Corresponding REGACCINTSTAT Bits

REGACCINTSTAT Bit	Direct Area Offset Address - Host Read Mode (CFG_WRAPPTR = 0)	Direct Area Offset Address - Host Burst Write Mode (CFG_WRAPPTR = 1)
31	0x0	0x0
30	0x1	0x1
29	0x2	0x2
28	0x3	0x3
27	0x4	0x4
26	0x5	0x5
25	0x6	0x6
24	0x7	0x7
23	0x8	0x8
22	0x9	0x9
21	0xA	0xA

Table 74: Direct Area Access Interrupt Mapping and Corresponding REGACCINTSTAT Bits

REGACCINTSTAT Bit	Direct Area Offset Address - Host Read Mode (CFG_WRAPPTR = 0)	Direct Area Offset Address - Host Burst Write Mode (CFG_WRAPPTR = 1)
20	0xB	0xB
19	0xC	0xC
18	0xD	0xD
17	0xE	0xE
16	0xF	0xF
15	0x13	0x17
14	0x17	0x1F
13	0x1B	0x2F
12	0x1F	0x3F
11	0x23	0x4F
10	0x27	0x5F
9	0x2B	0x6F
8	0x2F	0x7F
7	0x33	0x8F
6	0x37	0x9F
5	0x3B	0xAF
4	0x3F	0xBF
3	0x43	0xCF
2	0x47	0xDF
1	0x4B	0xEF
0	0x4F	0xF7

The REGACCINTSTAT register provides status of the 32 individual write interrupts. If an interrupt is enabled and set, it shows as a high bit in this register. The highest priority REGACC bit is bit 31 (set on access to address 0x00), and the lowest priority is bit 0 (set on access to address 0x4F in Standard Address Mode or 0xF7 in Address Pointer Wrap Mode). The 5-bit IOSUBORDINATE_PRENC register provides an encoded value of the highest priority of these interrupts to speed software decoding, and is therefore very useful for quickly servicing the highest priority REGACC interrupt (i.e., the one at the lowest offset address). The encoding works such that if interrupt 31 is set, PRENC will be 0. If interrupt 31 is not set and bit 30 is set, PRENC will be 1, and so on to the point where if bits 31-1 are not set and bit 0 is set PRENC will be 31. If no interrupts are set the value in PRENC is indeterminate.

The final special memory space within the Direct Area is a read-only area for the I/O Host, which is from I/O address (FIFOCFG_ROBASE * 8) to (FIFOCFG_ROBASE * 8 – 1). I/O writes to this address space will not change the LRAM, which allows the space to be used for returning status to the I/O Host. ROBASE should have a minimum value of 0x0A, representing a start address of 0x50 to allow space for special commands and burst writes in lower Direct Area space.

16.3 FIFO Area Functions

The FIFO is used to provide very efficient flow of data from the MCU to the I/O Host processor with minimal CPU interaction. A FIFO of up to 1023 bytes can be easily maintained by software, with the oldest bytes residing in the LRAM FIFO Area and the newer data being held in system SRAM and transferred to the I²C/SPI Subordinate on demand. Several hardware features support this operation.

Figure 46 shows the basic FIFO operation. The main FIFO is held in a buffer in SRAM, and the oldest data in that FIFO has been transferred to the FIFO Area of the I/O Subordinate. The IOSUBORDINATE_FIFOPTR_FIFOPTR register points to the next byte to be read on the I/O interface. IOSUBORDINATE_FIFOPTR_FIFOSIZ holds the current number of valid bytes in the FIFO on the I²C/SPI Subordinate, and FIFOCR holds the total number of bytes in the FIFO. The value in IOSUBORDINATE_FIFOCR may be read indirectly at any time by the Host processor via the FIFOCRUP_FIFOCRLO registers to determine if there is FIFO data available (and how much is currently in the FIFO). I/O Host access to the FIFO counter is at offset 0x7C/0x7D.

WARNING

The host read of the FIFOCR value via FIFOCRUP_FIFOCRLO is not synchronized to the write clock. So if the host read happens during a FIFOCR update (either through a read-modify-write of FIFOCR register or an automatic update because of a write to the FIFOINC register by the subordinate CPU), it is possible for the count value to be out of sync, impacting the value read in either or both the upper (FIFOCRUP) and lower (FIFOCRLO) bytes. This is a very rare case, but proper code would have the host read the two registers for the FIFOCR value multiple times until consecutive reads are the same.

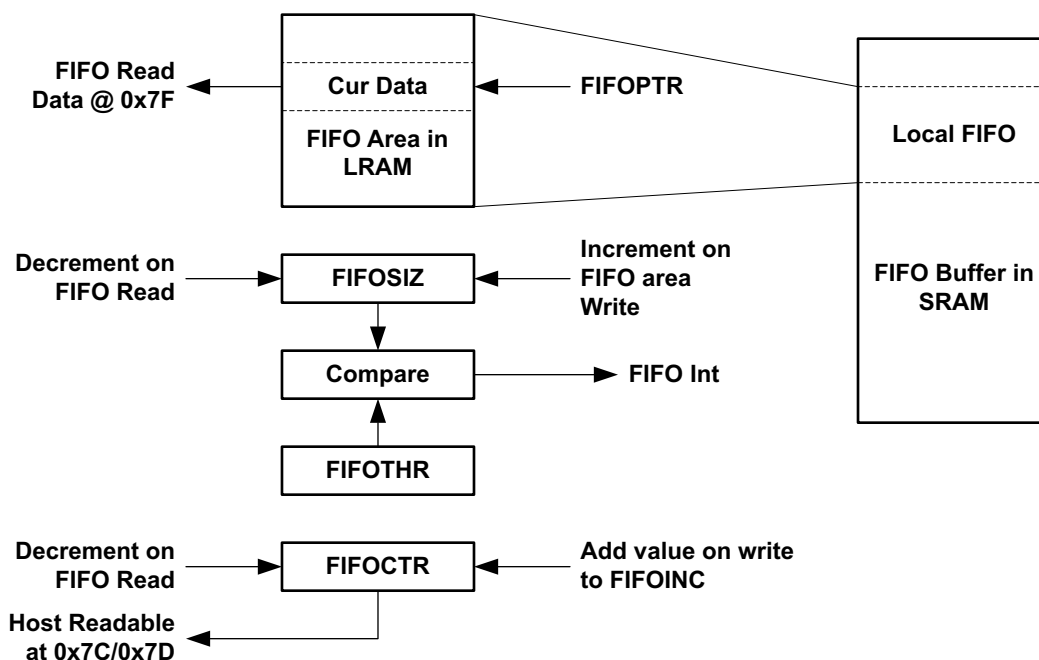


Figure 46. I²C/SPI Subordinate Module FIFO

When the host reads a byte from the FIFO, the data retrieved is pointed to by FIFOPTR, FIFOPTR is incremented and wraps around in the FIFO Area if it reaches FIFOMAX. FIFOSIZ and FIFOCTR are each decremented by one. The Host can read FIFOCTR and then read that many bytes without further checking. Note that this process can continue without requiring a CPU wakeup. If the Host attempts to read the FIFO when FIFOSIZ is 0, the FUNDFL interrupt flag is set in both the I²C Subordinate interrupt block and in the Host interrupt block.

When FIFOSIZ drops below the configured threshold IOSUBORDINATE_FIFOTHR the FSIZE interrupt flag is set and if enabled an interrupt is sent to the CPU which will wake it up. At that point, the CPU can move as much data from the SRAM FIFO to the I²C/SPI Subordinate FIFO as possible in a single operation and then go back to sleep. Since the FIFO Area can be quite large, CPU wake-ups will be very infrequent. If a write to the FIFOCTR which would increment the value beyond 1023 occurs, the FOVFL interrupt flag is set.

When some other process, such as a sensor read, produces new data for the FIFO, the CPU will add that data to the FIFO in SRAM, wrapping around as necessary. The IOSUBORDINATE_FIFOINC register is then written with the number of bytes added to the FIFO, which is added to the FIFOCTR register in an atomic fashion. In this way the Host processor can always determine how much read data is available.

The FIFO interface offset 0x7F is treated uniquely by the I²C/SPI Subordinate, in that an access to this address does not increment the Address Pointer. This allows the Host to initiate a burst read from address 0x7F of any length, and each read will supply the next byte in the FIFO.

16.4 Rearranging the FIFO

In normal operation the Host reads the oldest data from the FIFO, and the CPU writes new data onto the FIFO. In some cases it is desirable to modify this process, in particular for the FIFO to provide the newest data. The MCU supports such operation using a special control function.

If software is intended to write the current sample to the front of the FIFO, it first checks the IOSUBORDINATE_FUPD_IOREAD status bit to ensure that there is not a Host read operation from the FIFO underway. Once IOREAD is clear, software sets the IOSUBORDINATE_FUPD_FIFOUPT bit, writes the new sample data to the front of the FIFO and modifies the FIFOPTR to point to the new data. At that point the FIFOUPT bit is cleared.

If the Host attempts a FIFO read operation while the FIFOUPT is set, a RDERR interrupt will be generated to the Host and the FRDERR interrupt flag will be set. The Host must either poll the RDERR interrupt bit at the end of each operation or configure a hardware interrupt. Note that if the software does not support alternate FIFO ordering, the Host does not have to check the RDERR function.

16.5 Interface Interrupts

The CPU may also signal the Host via the IOINT interrupt, which may be connected to an MCU pin and driven to the Host. Eight interrupts are available to be combined into the IOINT interrupt, and the Host can enable, read, clear and set these interrupts via the I/O interface. Software on the CPU can set 6 of the interrupts (SWINT0 through SWINT5) to communicate a variety of situations to the Host, and the other two interrupts indicate errors such as an attempt by the Host to read the FIFO when it is empty. A CPU interrupt is generated whenever the Host writes any IOINT registers (for example, to clear an interrupt) so the CPU can manage the interrupt interaction.

The I2C/SPI Subordinate includes a mechanism to allow the Host CPU and the MCU to each interrupt the other via a set of eight interrupts. The Host CPU accesses these interrupts via interface locations 0x78 - 0x7B, and the Apollo accesses these interrupts in the IOINTCTL Register.

The Host CPU may enable or disable any of the eight interrupts by writing the corresponding bit in the IOINTEN field of the IOINTCTL Register, which is accessed by the Host at interface location 0x78. The Host CPU may then clear or set any of the interrupts by writing a 1 to the corresponding bit of the clear (at location 0x7A) or set (at location 0x7B) registers. The current state of all eight interrupts may be read in the IOINT field at location 0x79. Note that this structure is identical to the standard MCU interrupts in all modules. The SoC can read the value of the eight interrupt enables in the IOINTEN field of IOINTCTL, and can read the values of the eight interrupt status bits in the IOINT field of the IOINTCTL register. These two fields are read only. Table 75 summarizes these I/O interface interrupts and how they can be controlled and read.

Table 75: I/O Interface Interrupt Control

RAM Location	IOINT Register ¹	Function	SoC Register_Field	Description
0x78	IOINTEN	I/O Interrupt Enable	IOINTCTL_IOINTEN (R/O)	Each interrupt can be individually enabled by I/O Host, but can only be read by the SoC.
0x79	IOINT	I/O Interrupt State	IOINTCTL_IOINT (R/O)	State of each interrupt, set or cleared, can be read by either the I/O Host or by the SoC.
0x7A	IOINTCLR	I/O Interrupt Clear	IOINTCTL_IOINTCLR (W/O)	Each interrupt can be individually cleared by the I/O Host, but the SoC can (only) clear all of them at once.
0x7B	IOINTSET	I/O Interrupt Set	IOINTCTL_IOINTSET (W/O)	Each interrupt can be individually set by either the I/O Host or the SoC.

1. Readable by the I/O Host

The MCU software may set any of the eight interrupt status register bits by writing a 1 to the corresponding bit of the IOINTSET field of the IOINTCTL Register, and may clear all of the interrupts by writing a 1 to the IOINTCLR bit of the IOINTCTL register. This allows the SoC to generate a software interrupt to the Host device. In addition, a FIFO underflow interrupt FUNDFL in the I2C/SPI Subordinate will set interrupt bit 7, and a FIFO read error interrupt FRDERR will set interrupt bit 6 of the IO interrupt status register IOINT. Note that the SoC software cannot write the IOINTEN register, so that IO interrupts are controlled completely by the Host processor.

If any of the IOINT interrupt bits are set and the corresponding bit in IOINTEN is set, an IOINT interrupt will be generated. If the GPIO configuration registers have configured PAD4 as IOINT, that interrupt will be driven directly onto PAD_IO[4]. This pin should be connected to an interrupt input pin of the Host interface device so that it can receive the interrupt and service it.

If the Host device writes to any of the interrupt register access locations (any location in 0x78 - 0x7B) the IOINTW interrupt will be set in the I2C/SPI INTSTAT Register. This allows MCU software to receive a

software interrupt from the Host device. Note that this interrupt will occur for all writes by the Host, including a write to clear an interrupt.

16.6 Command Completion Interrupts

Four interrupts in the I²C/SPI Subordinate module are generated when the Host interface device completes a transfer. This allows the SoC to be easily awakened for any transfer from the Host while maximizing the time the SoC is in sleep mode. The XCMPWR interrupt is generated at the completion of a Host write transfer which includes addresses in the currently configured Direct Register space, and the XCMPRR interrupt is generated on the completion of a Host read transfer to that space. The XCMPWF interrupt is generated at the completion of a Host write transfer which includes the FIFO address 0x7F (although that is an invalid access), and the XCMPRF interrupt is generated at the completion of a Host read transfer which includes the FIFO address 0x7F.

NOTE

A write to 0x7F, which is the FIFO address, uses the address 0xFF since this includes the R/W bit in the upper (first) bit followed by the 7-bit Direct Register address (offset). The prescribed usage of IOS FIFO is only for READ from the host, and hence writing to the FIFO is generally an invalid operation. So, even though XCMPWF flag/interrupt is defined, it is likely never going to be used.

NOTE

A burst transfer which begins in the Direct Register address space and is long enough to cause the Address Pointer to be 0x7F can set both the Direct Register and FIFO interrupts, although that would in general be an invalid operation.

16.7 Host Address Space and Registers

The Host of the I/O interface can access 128 bytes in the I²C/SPI Subordinate in either I²C or SPI mode. Offsets 0x00 to 0x77 may be directly mapped to the Direct RAM Area. The remaining eight offset locations access hardware functions within the I²C/SPI Subordinate. The R/W indicator is referring to accesses from the Host.

16.8 I²C Interface

The MCU I²C Subordinate interface operates as a standard subordinate. The device is accessed at an address configured in the `IOSUBORDINATE_IOSCFG_I2CADDR` field, and supports Fast Mode Plus (up to 1 MHz). Both 7-bit and 10-bit address modes are supported, as selected by `IOSUBORDINATE_IOSCFG_10BIT`. The I²C interface consists of two lines: one bi-directional data line (SDA) and one clock line (SCL). Both the SDA and the SCL lines must be connected to a positive supply voltage via a pull-up resistor. By definition, a device that sends a message is called the “transmitter”, and the device that accepts the message is called the “receiver”. The device that controls the message transfer by driving SCL is called “manager”. The devices that are controlled by the manager are called “subordinates”. The SoC’s I²C Subordinate is always a subordinate device.

The following protocol has been defined:

- Data transfer may be initiated only when the bus is not busy.
- During data transfer, the data line must remain stable whenever the clock line is high.
- Changes in the data line while the clock line is high will be interpreted as control signals.

A number of bus conditions have been defined (see Figure 47) and are described in the following sections.

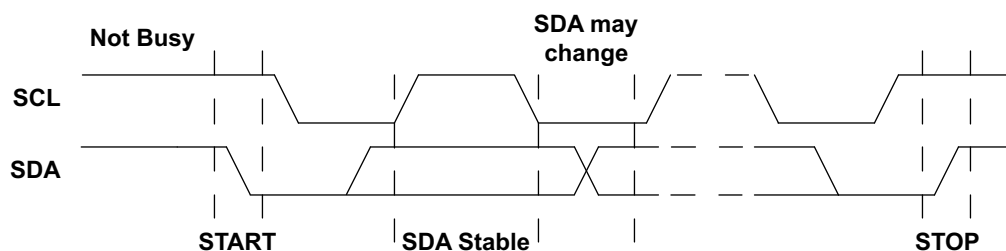


Figure 47. Basic I²C Conditions

16.8.1 Bus Not Busy

Both SDA and SCL remain high.

16.8.2 Start Data Transfer

A change in the state of SDA from high to low, while SCL is high, defines the START condition. A START condition which occurs after a previous START but before a STOP is called a RESTART condition, and functions exactly like a normal STOP followed by a normal START.

16.8.3 Stop Data Transfer

A change in the state of SDA from low to high, while SCL is high, defines the STOP condition.

16.8.4 Data Valid

After a START condition, SDA is stable for the duration of the high period of SCL. The data on SDA may be changed during the low period of SCL. There is one clock pulse per bit of data. Each data transfer is initiated with a START condition and terminated with a STOP condition. The number of data bytes transferred between the START and STOP conditions is not limited. The information is transmitted byte-wide and each receiver acknowledges with a ninth bit.

16.8.5 Acknowledge

Each byte of eight bits is followed by one Acknowledge (ACK) bit as shown in Figure 48. This Acknowledge bit is a low level driven onto SDA by the receiver, whereas the manager generates an extra ACK related SCL pulse. A subordinate receiver which is addressed is obliged to generate an Acknowledge after the reception of each byte. Also, on a read transfer a manager receiver must generate an Acknowledge after the reception of each byte that has been clocked out of the subordinate transmitter. The device that acknowledges must pull down the SDA line during the Acknowledge clock pulse in such a way that the SDA line is a stable low during the high period of the Acknowledge related SCL pulse. A manager receiver must signal an end-of-data to the subordinate transmitter by not generating an Acknowledge (a NAK) on the last byte that has been clocked out of the subordinate. In this case, the transmitter must leave the data line high to enable the manager to generate the STOP condition.

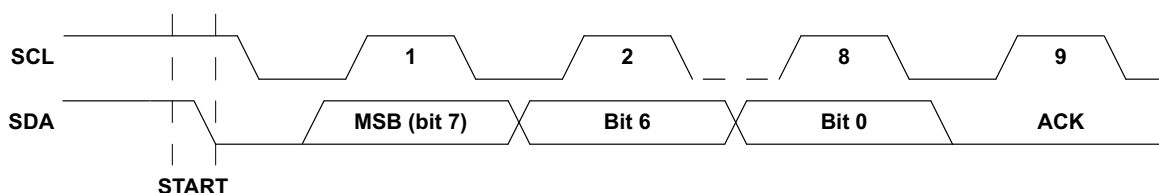


Figure 48. I²C Acknowledge

16.8.6 Address Operation

In I²C mode, the I²C/SPI Subordinate supports either 7-bit or 10-bit addressing, selected by the 10BIT bit in the IOSCFG Register. Figure 49 shows the operation in 7-bit mode in which the manager addresses the MCU with a 7-bit address configured as 0xD2 in the CFG_I2CADDR field. After the START condition, the 7-bit address is transmitted MSB first. If this address matches the lower 7 bits of the CFG_I2CADDR field, the SoC is selected, the eighth bit indicate a write (RW = 0) or a read (RW = 1) operation and the SoC supplies the ACK. The SoC ignores all other address values and does not respond with an ACK.

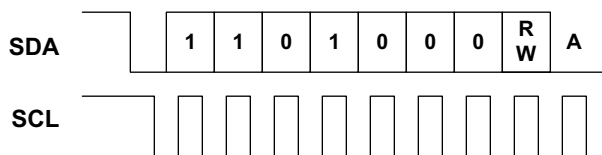


Figure 49. I²C 7-bit Address Operation

Figure 50 shows the operation with which the manager addresses the SoC with a 10-bit address configured at 0x536. After the START condition, the 10-bit preamble 0b11110 is transmitted first, followed by the first two address bits and the eighth bit indicating a write (RW = 0) or a read (RW = 1) operation. If the upper two bits match the I2CADDR value, the I²C/SPI Subordinate supplies the ACK. The next transfer includes the lower 8 bits of the address, and if these bits also match I2CADDR the SoC again supplies the ACK. The I²C/SPI Subordinate ignores all other address values and does not respond with an ACK.

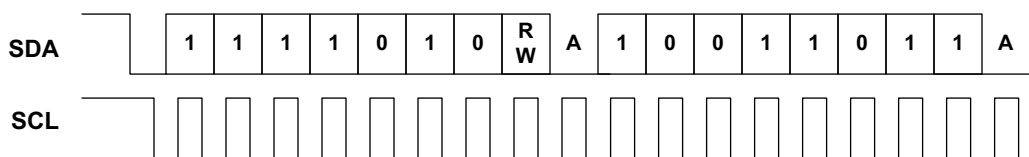


Figure 50. I²C 10-bit Address Operation

16.8.7 Offset Address Transmission

If the RW bit of the Address Operation indicates a write, the next byte transmitted from the manager is the Offset Address as shown in Figure 51. This value is loaded into the Address Pointer of the I²C/SPI Subordinate.

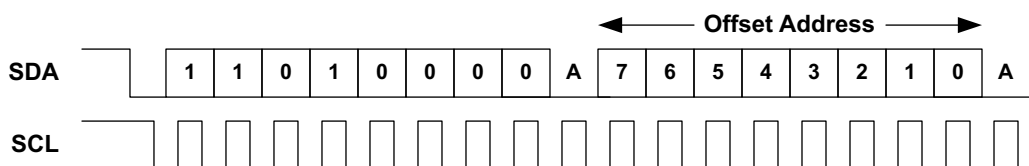


Figure 51. I²C Offset Address Transmission

16.8.8 Write Operation

In a write operation the manager transmitter transmits to the MCU subordinate receiver. The Address Operation has a RW value of 0, and the second byte contains the Offset Address as in Figure 51. The next byte is written to the register selected by the Address Pointer (which was loaded with the Offset Address) and the Address Pointer is incremented. Subsequent transfers write bytes into successive registers until a STOP condition is received, as shown in Figure 52. Note that if the Address Pointer is at 0x7F, it will not increment on the write.

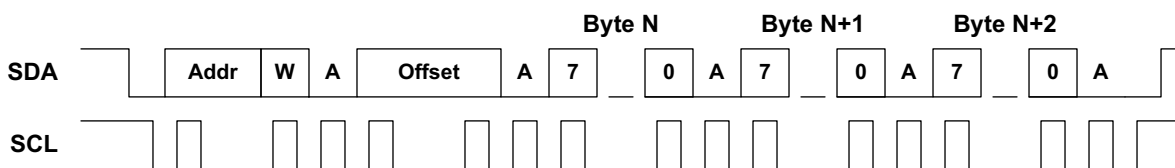


Figure 52. I²C Write Operation

16.8.9 Read Operation

In a read operation, the manager first executes an Offset Address Transmission to load the Address Pointer with the desired Offset Address. A subsequent operation will again issue the address of the SoC but with the RW bit as a 1 indicating a read operation. Figure 53 shows this transaction beginning with a RESTART condition, although a STOP followed by a START may also be used. After the address operation, the subordinate becomes the transmitter and sends the register value from the location pointed to by the Address Pointer, and the Address Pointer is incremented. Subsequent transactions produce

successive register values, until the manager receiver responds with a NAK and a STOP to complete the operation. Because the Address Pointer holds a valid register address, the manager may initiate another read sequence at this point without performing another Offset Address operation. Note that if the Address Pointer is at 0x7F, it will not increment on the read.

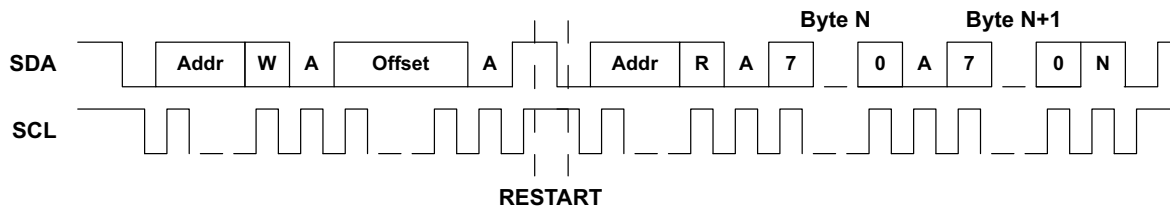


Figure 53. I²C Read Operation

16.8.10 General Address Detection

The I²C/SPI Subordinate may be configured to detect an I²C General Address (0x00) write. If this address is detected, the first data byte written is stored in the IOSUBORDINATE_GADATA Register and the GENAD interrupt flag is set. This allows software to create the appropriate response, which is typically to reset the I²C/SPI Subordinate.

16.9 SPI Interface

The I²C/SPI Subordinate includes a standard 3-wire or 4-wire SPI interface. The serial peripheral interface (SPI) bus is intended for synchronous communication between different ICs. 4-wire SPI consists of four signal lines: serial data input (MOSI), serial data output (MISO), serial clock (SCL) and an active low chip enable (nCE). The I²C/SPI Subordinate may be connected to a manager with a 3-wire SPI interface by configuring 3-wire mode in the pin configuration block of the GPIO module, which will tie MOSI and MISO together. By definition, a device that sends a message is called the “transmitter”, and the device that accepts the message is called the “receiver”. The device that controls the message transfer by driving SCL is called “manager”. The devices that are controlled by the manager are called “subordinates”. The I²C/SPI Subordinate SPI subordinate is always a subordinate device.

The nCE input is used to initiate and terminate a data transfer. The SCL input is used to synchronize data transfer between the manager and the subordinate devices via the MOSI (manager to subordinate) and MISO (subordinate to manager) lines. The SCL input, which is generated by the manager, is active only during address and data transfer to any device on the SPI bus.

The I²C/SPI Subordinate supports all SPI configurations of CPOL and CPHA using the SPOL configuration bit. There is one clock for each bit transferred. Address and data bits are transferred in groups of eight bits.

16.9.1 Write Operation

Figure 54 shows a SPI write operation. The operation is initiated when the nCE signal to the SoC goes low. At that point an 8-bit Address byte is transmitted from the manager on the MOSI line, with the upper RW bit indicating read (if 0) or write (if 1). In this example the RW bit is a one selecting a write operation, and the lower 7 bits of the Address byte contain the Offset Address, which is loaded into the Address Pointer of the I²C/SPI Subordinate.

Each subsequent byte is loaded into the register selected by the Address Pointer, and the Address Pointer is incremented. The operation is terminated by the manager by bringing the nCE signal high. Note that the MISO line is not used in a write operation and is held in the high impedance state by the I²C/SPI Subordinate. Note also that if the Address Pointer is 0x7F, it does not increment on the read.

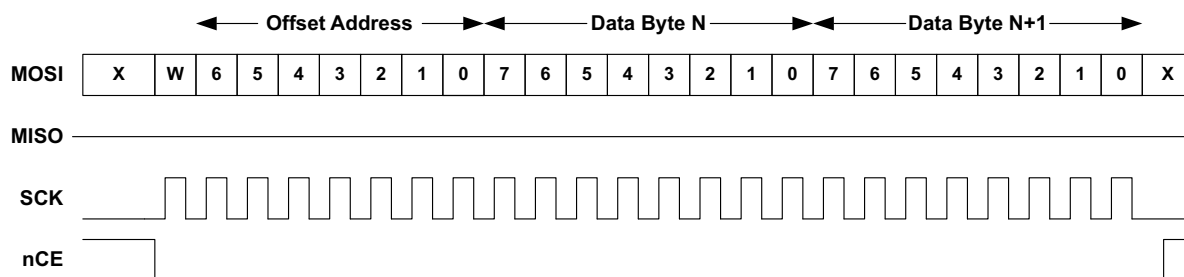


Figure 54. SPI Write Operation

16.9.2 Read Operation

Figure 55 shows a read operation. The address is transferred from the manager to the subordinate just as it is in a write operation, but in this case the RW bit is a 0 indicating a read. After the transfer of the last address bit (bit 0), the I²C/SPI Subordinate begins driving data from the register selected by the Address Pointer onto the MISO line, bit 7 first, and the Address Pointer is incremented. The transfer continues until the manager brings the nCE line high. Note that if the Address Pointer is 0x7F, it does not increment on the read.

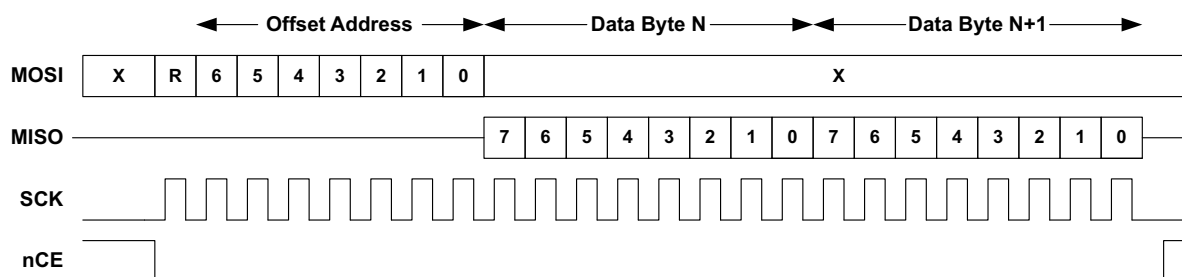


Figure 55. SPI Read Operation

16.9.3 Configuring 3-wire vs. 4-wire SPI Mode

The I²C/SPI Subordinate can operate in either 4-wire SPI mode, where the MISO and MOSI signals are on separate wires, or in 3-wire SPI mode where MISO and MOSI share a wire. This configuration is performed in the Pin Configuration module, and no configuration is necessary in the I²C/SPI Subordinate itself.

16.9.4 SPI Polarity and Phase

The I²C/SPI Subordinate supports all combinations of CPOL (clock polarity) and CPHA (data phase) in SPI mode. Figure 56 shows how these two bits affect the interface signal behavior.

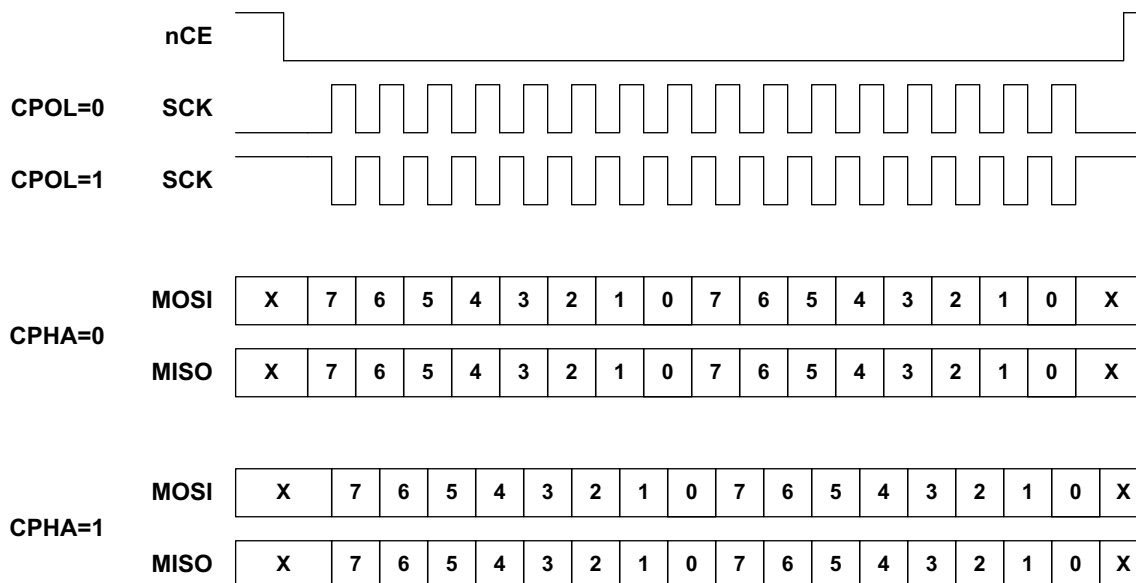


Figure 56. SPI CPOL and CPHA

If CPOL is 0, the clock SCK is normally low and positive pulses are generated during transfers. If CPOL is 1, SCK is normally high and negative pulses are generated during transfers.

If CPHA is 0, the data on the MOSI and MISO lines is sampled on the edge corresponding to the first SCK edge after nCE goes low (i.e. the rising edge if CPOL is 0 and the falling edge if CPOL is 1). Data on MISO and MOSI is driven on the opposite edge of SCK.

If CPHA is 1, the data on the MOSI and MISO lines is sampled on the edge corresponding to the second SCK edge after nCE goes low (i.e. the falling edge if CPOL is 0 and the rising edge if CPOL is 1). Data on MISO and MOSI is driven on the opposite edge of SCK.

The I²C/SPI Subordinate has only a single SPOL bit to control the polarity. If CPOL = CPHA, IOSUBORDINATE_IOSCFG_SPOL must be set to 0. If CPOL ≠ CPHA, SPOL must be set to 1.

16.10 Bit Orientation

In both I²C and SPI modes, the I²C/SPI Subordinate supports data transmission either LSB first or MSB first as configured by the IOSUBORDINATE_IOSCFG_LSB bit. If LSB is 0, data is transmitted and received MSB first. If LSB is 1, data is transmitted and received LSB first.

16.11 Wakeup Using the I²C/SPI Subordinate

The I²C/SPI Subordinate can continue to operate even if the SoC's CPU is in Sleep or Deep Sleep mode. The hardware will enable and disable the I²C/SPI Subordinate clock and oscillators as necessary. The only consideration in this environment is when the SoC is in a deep sleep mode, such that the HFRC Oscillator is powered down, and a manager attempts to access the I²C/SPI Subordinate. In this case the HFRC Oscillator must be powered up before anything is transferred to or from the internal RAM. This process takes roughly 5-10 μ s, and is initiated by nCE going low in SPI mode or by the detection of a START in I²C mode.

NOTE

If the host reads or writes the register space from offset 0x78 through 0x7D (IOINT is 0x78 through 0x7B), there is no dependence on the HFRC running because the LRAM is not being accessed, so these transactions occur even if in Deep Sleep. If the host accesses the direct memory space on a read or write (all addresses not in the range 0x78 to 0x7F), or accesses the LRAM through the FIFO read port at offset 0x7F, the LRAM is being accessed and the HFRC must be running.

For I²C applications, the time delay is typically not relevant. At the fastest system clock of 1 MHz, the manager must transfer 9 bits of address plus 9 bits of offset before any FIFO access can occur, and that is a minimum of 18 μ s. The clocks will have started prior to that point in every case.

For SPI applications with fast interface clocks (faster than 1 MHz), the manager must be programmed to pull nCE low at least 10 μ s prior to sending the first clock. If a manager is unable to control the timing of nCE in this way, then there are other options for ensuring that manager commands/data are received reliably by the subordinate are as follows:

1. A GPIO interrupt can be configured to wake the SoC from Deep Sleep mode prior to initiating any SPI transfers.
2. An IOINT interrupt from the host can be configured to wake up the subordinate by writing to the IOINT register over the interface.
3. The HFRC clock can be forced to remain active during Deep Sleep mode by setting the CLKGEN_MISC_FRCHFRC bit.

Regarding option 3, forcing HFRC to stay active is done by setting the FRCHFRC bit of the CLKGEN's MISC Register. If this bit is set, the HFRC will continue to be active even if the SoC's CPU is in deep sleep mode, so that the subordinate can immediately begin transferring data independent of the transfer rate. This will result in higher power because the HFRC remains active, so the FRCHFRC bit should only be set if it is known that a transfer is likely to begin prior to another interrupt.

There is no delay restriction if the SoC is in normal Sleep mode. In that case the HFRC is not powered down and the I²C/SPI Subordinate clock will start immediately when nCE goes low.

16.12 Host Side Address Space and Register

16.12.1 Host Address Space and Registers

The Host of the I/O interface can access 128 bytes in the I²C/SPI Subordinate in either I²C or SPI mode. Offsets 0x00 to 0x77 may be directly mapped to the Direct RAM Area. The remaining eight offset locations access hardware functions within the I²C/SPI Subordinate. The R/W indicator refers to accesses from the Host.

16.12.1.1 HOST_IER Register

Host Interrupt Enable

OFFSET: 0x78

This register enables the FIFO read interrupts.

Table 76: HOST_IER Register

0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
FUNDFLEN	RDERREN	SWINT5EN	SWINT4EN	SWINT3EN	SWINT2EN	SWINT1EN	SWINT0EN

Table 77: HOST_IER Register Bits

Bit	Name	Reset	RW	Description
7	FUNDFLEN	0x0	RW	If 1, enable an interrupt that triggers when the FIFO underflows
6	RDERREN	0x0	RW	If 1, enable the interrupt which occurs when the Host attempts to access the FIFO when read access is locked
5	SWINT5EN	0x0	RW	If 1, enable software interrupt 5
4	SWINT4EN	0x0	RW	If 1, enable software interrupt 4
3	SWINT3EN	0x0	RW	If 1, enable software interrupt 3
2	SWINT2EN	0x0	RW	If 1, enable software interrupt 2
1	SWINT1EN	0x0	RW	If 1, enable software interrupt 1
0	SWINT0EN	0x0	RW	If 1, enable software interrupt 0

16.12.1.2 HOST_ISR Register

Host Interrupt Status Register

OFFSET: 0x79

The host uses this register to read interrupt status.

Table 78: HOST_ISR Register

0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
FUNDFLSTAT	RDERRSTAT	SWINT5STAT	SWINT4STAT	SWINT3STAT	SWINT2STAT	SWINT1STAT	SWINT0STAT

Table 79: HOST_ISR Register Bits

Bit	Name	Reset	RW	Description
7	FUNDFLSTAT	0x0	RO	This bit is set by writing a 1 to bit 31 of the IOINTCTL Register, or if the Host attempts a FIFO read when FIFOCTR is 0.
6	RDERRSTAT	0x0	RO	This bit is set by writing a 1 to bit 30 of the IOINTCTL Register, or if the Host attempts a FIFO read when the FIFOUdp bit is a 1.
5	SWINT5STAT	0x0	RO	This bit is set by writing a 1 to bit 29 of the IOINTCTL Register.
4	SWINT4STAT	0x0	RO	This bit is set by writing a 1 to bit 28 of the IOINTCTL Register.
3	SWINT3STAT	0x0	RO	This bit is set by writing a 1 to bit 27 of the IOINTCTL Register.
2	SWINT2STAT	0x0	RO	This bit is set by writing a 1 to bit 26 of the IOINTCTL Register.
1	SWINT1STAT	0x0	RO	This bit is set by writing a 1 to bit 25 of the IOINTCTL Register.
0	SWINT0STAT	0x0	RO	This bit is set by writing a 1 to bit 24 of the IOINTCTL Register.

NOTE

All bits are cleared by a write to the IOINTCLR bit of the IOINTCTL Register.

16.12.1.3 HOST_WCR Register

Host Interrupt Write-to-Clear Register

OFFSET: 0x7A

Write a 1 to a bit in this register to clear a pending interrupt.

Table 80: HOST_WCR Register

0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
FUNDFLWC	RDERRWC	SWINT5WC	SWINT4WC	SWINT3WC	SWINT2WC	SWINT1WC	SWINT0WC

Table 81: HOST_WCR Register Bits

Bit	Name	Reset	RW	Description
7	FUNDFLWC	0x0	WO	Writing a 1 to this bit will clear the pending interrupt status bit FUNDFLSTAT
6	RDERRWC	0x0	WO	Writing a 1 to this bit will clear the pending interrupt status bit RDERRSTAT
5	SWINT5WC	0x0	WO	Writing a 1 to this bit will clear the pending interrupt status bit SWINT5STAT
4	SWINT4WC	0x0	WO	Writing a 1 to this bit will clear the pending interrupt status bit SWINT4STAT
3	SWINT3WC	0x0	WO	Writing a 1 to this bit will clear the pending interrupt status bit SWINT3STAT
2	SWINT2WC	0x0	WO	Writing a 1 to this bit will clear the pending interrupt status bit SWINT2STAT
1	SWINT1WC	0x0	WO	Writing a 1 to this bit will clear the pending interrupt status bit SWINT1STAT
0	SWINT0WC	0x0	WO	Writing a 1 to this bit will clear the pending interrupt status bit SWINT0STAT

16.12.1.4 HOST_WCS Register**Host Interrupt Write-to-Set Register****OFFSET:** 0x7B

Write a 1 to a bit in this register to set the status bit of a pending interrupt.

Table 82: HOST_WCS Register

0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
FUNDFLWS	RDERRWS	SWINT5WS	SWINT4WS	SWINT3WS	SWINT2WS	SWINT1WS	SWINT0WS

Table 83: HOST_WCS Register Bits

Bit	Name	Reset	RW	Description
7	FUNDFLWS	0x0	WO	Writing a 1 to this bit will set the pending interrupt status bit FUNDFLSTAT
6	RDERRWS	0x0	WO	Writing a 1 to this bit will set the pending interrupt status bit RDERRSTAT
5	SWINT5WS	0x0	WO	Writing a 1 to this bit will set the pending interrupt status bit SWINT5STAT
4	SWINT4WS	0x0	WO	Writing a 1 to this bit will set the pending interrupt status bit SWINT4STAT
3	SWINT3WS	0x0	WO	Writing a 1 to this bit will set the pending interrupt status bit SWINT3STAT
2	SWINT2WS	0x0	WO	Writing a 1 to this bit will set the pending interrupt status bit SWINT2STAT
1	SWINT1WS	0x0	WO	Writing a 1 to this bit will set the pending interrupt status bit SWINT1STAT
0	SWINT0WS	0x0	WO	Writing a 1 to this bit will set the pending interrupt status bit SWINT0STAT

16.12.1.5 FIFOCTRLO Register

FIFOCTR Low Byte

OFFSET: 0x7C

This register allows the host to read the lower eight bits of the FIFOCTR register.

Table 84: FIFOCTRLO Register

0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
FIFOCTRLO							

Table 85: FIFOCTRLO Register Bits

Bit	Name	Reset	RW	Description
7:0	FIFOCTRLO	0x0	RO	Reads the lower eight bits of FIFOCTR

16.12.1.6 FIFOCTRUP Register

FIFOCTR Upper Byte

OFFSET: 0x7D

This register allows the host to read the upper two bits of the FIFOCTR register.

Table 86: FIFOCTRUP Register

0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
RSVD						FIFOCTRUP	

Table 87: FIFOCTRUP Register Bits

Bit	Name	Reset	RW	Description
1:0	FIFOCTRUP	0x0	RO	Reads the upper two bits of FIFOCTR

16.12.1.7 FIFO Register

FIFO Read Data

OFFSET: 0x7F

Read this register for FIFO data.

Table 88: FIFO Register

0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
FIFO							

Table 89: FIFO Register Bits

Bit	Name	Reset	RW	Description
7:0	FIFO	0x0	RO	Reads the top byte of the FIFO

17. Universal Asynchronous Receiver/Transmitter (UART)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

17.1 Functional Description

17.1.1 Baud Rate Generator

The UART register block is used to configure the UART Module. The data width, number of stop bits, and parity may all be configured using the UART_LCRH register.

The baud rate generator derives the serial bit clock from the peripheral clock input through a two-stage divider implementing integer and fractional division. The integer divisor (UART_IBRD) provides coarse frequency division, while the fractional divisor (UART_FBRD) implements fine-grained adjustment through a 6-bit fractional multiplier (representing 1/64 increments). This architecture enables precise baud rate synthesis across a wide range of peripheral clock frequencies.

The baud rate calculation follows the formula: $BR = \text{ClockRate} / (16 \times (\text{IBRD} + \text{FBRD}/64))$, where ClockRate represents the UART peripheral clock frequency, IBRD is the integer divisor value, and FBRD is the fractional divisor value. The factor of 16 is derived from the internal 16× oversampling used for robust bit detection and center sampling.

The UART Module supports independent CTS and RTS hardware flow control. All flow control configuration may be set using the UART_CR register.

17.1.2 Transmitter

The transmitter subsystem serializes parallel data words into serial bit streams conforming to the configured frame format. Data written to the UART_DR transmit register transfers into a 16-entry hardware FIFO. The transmit shift register extracts data from the FIFO and serializes it with proper start bit, data bits, optional parity bit, and stop bit(s) according to the UART_LCRH field configurations. The FIFO provides elasticity buffering to accommodate software latency in servicing transmit interrupts.

Hardware flow control, when enabled via UART_CR, monitors the CTS (Clear To Send) input signal. The transmitter suspends character transmission when CTS is deasserted, preventing transmit buffer overflow at the remote receiver. Transmission resumes automatically when CTS reasserts.

17.1.3 Receiver

The receiver subsystem implements 16× oversampling of the RX input line for robust bit detection in the presence of clock frequency variation and noise. Start bit detection triggers frame reception, with subsequent bits sampled at their midpoint for maximum noise margin. The receiver performs frame format validation (stop bit detection) and optional parity verification. Validated data characters transfer into a 16-entry hardware FIFO for retrieval by software or DMA.

Hardware flow control, when enabled, automatically asserts the RTS (Request To Send) output signal when the receive FIFO has sufficient space, and de-asserts RTS when the FIFO approaches full condition. This provides automatic back-pressure to the remote transmitter without software intervention.

17.1.4 FIFO Operation

Independent 16-entry FIFOs buffer transmit and receive data paths. Software configures interrupt threshold levels through UART_IFLS for both directions independently. Available threshold settings include 1/8, 1/4, 1/2, 3/4, and 7/8 full conditions, allowing tuning of interrupt frequency versus latency trade-offs.

The receive FIFO includes timeout detection that generates an interrupt when characters reside in the FIFO but the threshold has not been reached and no new characters have arrived within the timeout period (typically 32-bit periods).

17.1.5 DMA Support

The UART peripheral implements standard AMBA DMA request signaling for both transmit and receive directions. DMA requests assert based on FIFO threshold levels, allowing the DMA controller to autonomously transfer data between memory and UART FIFOs without CPU intervention. The streaming UART HAL implements sophisticated DMA descriptor chaining for continuous operation, including circular buffer management for transmit and double-buffering (ping-pong) for receive to achieve zero-gap continuous data streaming.

17.1.6 Flow Control

Hardware flow control implements the RTS/CTS handshaking protocol in hardware logic, eliminating software latency from flow control response. The receiver automatically manages RTS output assertion based on receive FIFO occupancy, providing advance warning to the remote transmitter before FIFO overflow. The transmitter monitors CTS input and suspends transmission when CTS is deasserted, resuming automatically upon CTS reassertion. This mechanism prevents data loss due to buffer overflow in both directions without software involvement.

For applications not requiring hardware flow control, the RTS and CTS signals may remain unconnected and flow control disabled in UART_CR. In this configuration, software must ensure adequate buffering and processing throughput to prevent transmit underrun or receive overflow conditions.

17.2 Design Considerations

17.2.1 Enabling and Selecting the UART Clock

The UART module receives two clocks - UART_clk which is used to derive the UART serial clock and UART_hclk, which is the bus interface clock of the UART module. Unlike other modules, the UART requires a bus clock whenever it is transmitting or receiving, so special controls are required when the UART is to transfer data while the MCU is in a sleep mode and its normal bus clocks are not operating.

UART_clk is selected in the UARTx_CR_CLKSEL field, with values from 3 MHz to 48 MHz plus a disabled value NOCLK, and is enabled by the UARTx_CR_CLKEN bit. If the UART is inactive, CLKSEL should be set to the NOCLK value (0) to minimize power, and the CLKEN bit should be 0. When the UART is active, the serial clock is created by the baud rate generator based on UART_clk.

The System PLL is another option for clocking the UART and is used for higher frequency UART communication. A higher UART_clk frequency can produce more precise and higher serial clock frequencies but will cause the UART to use more power. It is thus recommended that UART_clk be set to the minimum frequency which produces acceptable serial clocks.

17.2.2 Requirements when Operating at or above 3 MBaud in DMA Mode

To be able to receive data reliably at or above 3 Mbps baud rate on UART1, UART2 or UART3, the UART clock source must be set to SYSPLL by setting the UARTn_CR_CLKSEL = PLL_CLK. Also, the PCLK for the specific UART in use must be set to always on by setting the MCUCTRL_D2ASPARE_UARTnALWAYSON bit. When bit 22 to 25 of this register is set, it will force the PCLK clock enable for UART0 to UART3, respectively, to be always on. UART0 is not affected by this limitation.

To receive at baud rates lower than 3 MBaud, it is not necessary to set this bit or to use the SYSPLL as the UART's clock source.

17.2.2.1 UART DMA Reception in Full Duplex Mode at or above 3 MBaud

Since UART DMA is only supported in half-duplex mode, two UART instances are required (one for TX and one for RX) to support full-duplex DMA. At or above a baud rate of 3 MBaud, reliable DMA-based data reception while in full duplex mode requires specific transmitter configuration. The rapid character arrival rate at 3 MBaud or higher creates timing constraints for the DMA controller to service receive FIFO requests without data loss.

In this full duplex mode, the TX instance should be configured to use 2 stop bits. The receiving UART (running DMA mode) should be configured for 1 stop bit. This asymmetric configuration provides sufficient inter-character gap for the DMA controller to service the receive FIFO before subsequent character arrival. The external UART TX needs to use 2 stop bits setting.

Applications requiring symmetric UART configuration (both TX and RX with identical stop bit count) should limit maximum baud rate to 1.5 MBaud when using DMA reception to ensure data integrity. Polled and interrupt-driven reception modes are not affected by this constraint at any supported baud rate.

17.2.3 Transmit FIFO and Receive FIFO

The transmit and receive FIFOs may both be accessed via the same 8-bit word in the UART_DR register. The transmit FIFO stores up to 32 8-bit words and can be written using writes to UART_DR. The receive FIFO stores up to 32 12-bit words and can be read using reads to UART_DR. Note that each 12-bit receive FIFO word includes an 8-bit data word and a 4-bit error status word.

17.2.4 Baud Rate Accuracy

Precise baud rate generation is essential for maintaining communication integrity in UART applications. The cumulative baud rate error between transmitter and receiver should remain under 2% to ensure reliable bit sampling and frame synchronization. The UART peripheral includes fractional divider support to achieve accurate baud rates that cannot be precisely generated through integer division of the source clock alone. When selecting clock sources, verify that the desired baud rate can be generated within acceptable error limits across the full operating temperature and voltage range.

17.2.5 Power Consumption

Several strategies can significantly reduce UART peripheral power consumption while maintaining functionality:

- Select the minimum baud rate sufficient for application throughput requirements.
- Power down unused UART instances through peripheral power control registers.
- Leverage DMA-based transfers to permit CPU entry into low-power sleep states.
- Tune FIFO interrupt thresholds to balance latency against interrupt service frequency.
- Implement hardware flow control (RTS/CTS) to prevent receive buffer overflow conditions.
- Gate peripheral clock inputs when UART communication is idle.

17.2.6 Reliable Communication

Implementing robust UART communication requires attention to error detection, protocol design, and system testing:

- Configure and monitor error detection mechanisms including framing, parity, and overrun interrupts.
- Structure protocols with explicit frame delimiters and end-to-end integrity verification (CRC/checksum).
- Enable parity generation and checking where protocol overhead allows.
- Utilize receive timeout interrupts to identify packet boundaries in variable-length protocols.
- Allocate receive buffers with sufficient margin for burst traffic scenarios.
- Deploy hardware flow control for sustained high-throughput data streams.
- Validate baud rate accuracy across operating corner conditions (temperature, voltage extremes).

17.2.7 Mode Selection

Regular UART mode is appropriate for command-response protocols, moderate throughput applications (≤ 115.2 kbaud typically), and implementations requiring straightforward blocking/non-blocking I/O semantics. Streaming UART mode is optimized for high-throughput continuous data applications (>115.2 kbaud), scenarios requiring minimal CPU intervention through advanced DMA buffer management, and systems demanding efficient handling of sustained bidirectional data flows.

17.3 Programming Guidelines

17.3.1 Initializing a UART Module

UART peripheral initialization follows this sequence:

1. Assert peripheral clock enable in the clock control subsystem.
2. Configure GPIO pin multiplexing for UART function assignments (TX, RX, optional RTS/CTS).
3. Disable transmitter and receiver during configuration to prevent spurious operation.
4. Program integer and fractional baud rate divisor registers based on clock source frequency.
5. Define serial frame format parameters (data bits, parity mode, stop bit count).
6. Establish FIFO watermark thresholds for transmit and receive directions.
7. Unmask required interrupt sources in the interrupt enable register.
8. Configure DMA descriptors if utilizing DMA transfer mode.
9. Assert enable bits for transmitter and receiver to commence operation.

17.3.2 Transmitting Data

The UART peripheral supports three distinct data transmission methodologies. Polled transmission queries FIFO status flags and directly writes data when space is available, suitable for low-throughput scenarios. Interrupt-driven transmission enables TX FIFO threshold interrupts and services the FIFO within the interrupt handler, balancing throughput with CPU efficiency. DMA-based transmission configures a DMA channel descriptor to autonomously transfer data from memory buffers to the UART TX FIFO, maximizing throughput while minimizing CPU intervention.

17.3.3 Receiving Data

Data reception employs parallel methodologies to transmission. Polled reception continuously monitors FIFO status and extracts available data through register reads. Interrupt-driven reception enables both RX FIFO threshold and timeout interrupts, allowing efficient data extraction when the FIFO reaches the configured threshold or when inter-character gaps exceed the timeout period. DMA-based reception configures a DMA channel to autonomously transfer received data from the UART RX FIFO to memory buffers, with completion callbacks triggered upon buffer fill or timeout conditions.

17.3.4 Error Handling

The UART peripheral implements hardware detection for multiple error conditions:

- Framing Error: Absence of expected stop bit indicates baud rate mismatch or noise corruption.
- Parity Error: Parity bit verification failure signals bit-level data corruption during transmission.
- Overrun Error: New data reception while FIFO is full results in data loss (dropped characters).
- Break Condition: RX line asserts low for a duration exceeding one character frame period.

Error status bits reside in the flag status register and can generate dedicated interrupts for immediate software notification. Firmware must implement periodic error status polling or interrupt-driven error handling to detect and recover from communication faults. Error flags require explicit software clearing through write-one-to-clear semantics.

17.4 UART Operating Modes

The Apollo510 UART peripherals provide two distinct operating modes implemented through separate HAL (Hardware Abstraction Layer) software interfaces. Both modes access identical underlying hardware registers but expose different programming models optimized for specific use cases. Selection between modes occurs at peripheral initialization and remains fixed for the operational lifetime of that UART instance.

17.4.1 Regular UART Mode

Regular mode implements a general-purpose UART interface supporting both blocking and non-blocking I/O operations suitable for command-response protocols and moderate data rates.

Operating characteristics:

- Synchronous blocking and asynchronous non-blocking read/write functions
- Direct FIFO access with optional software-managed transmit and receive queues
- Basic DMA support for single-shot transfer operations
- Appropriate for interactive protocols and moderate throughput applications (<115.2 kbaud typical)

Initialization:

Invoke `am_hal_uart_initialize()` to configure the UART instance for regular mode operation.

17.4.2 Streaming UART Mode

Streaming mode is architected for high-throughput continuous data transfer with sophisticated DMA buffer management and minimal CPU overhead, supporting sustained data rates up to 3 Mbps.

Operating characteristics:

- Optimized for sustained high-speed bidirectional data streaming
- Advanced DMA descriptor chaining: single-buffer (circular) and double-buffer (ping-pong) modes
- Automatic DMA buffer switching with completion callbacks for zero-gap continuous reception
- Software-configurable receive timeout for packet boundary detection in variable-length protocols
- Designed for applications requiring sustained throughput exceeding 115.2 kbaud

Initialization:

Invoke `am_hal_uart_stream_initialize()` to configure the UART instance for streaming mode operation.

NOTE

Each UART instance must exclusively use either Regular Mode or Streaming Mode APIs throughout its operational lifetime. Intermixing API calls from both modes on a single UART instance produces undefined behavior and is explicitly unsupported.

17.5 Register Summary

For detailed register descriptions, refer to the UART register section in the Apollo510 SoC register set documentation included with the AmbiqSuite SDK.

Key Registers:

- UARTDR - Data Register
 - Combined transmit and receive data register
 - Write: Transmit data (8-bit) written to TX FIFO
 - Read: Receive data (8-bit) extracted from RX FIFO
 - Includes error status bits (framing, parity, break, overrun) on read
- UARTCR - Control Register
- UART enable/disable control
 - Transmit enable (TXE) and receive enable (RXE) bits
 - Hardware flow control enable (CTSEN/RTSEN)
 - Loopback mode enable for diagnostic testing
 - Receive interrupt enable
- UARTLCR - Line Control Register
 - Serial frame format configuration
 - Word length selection (5, 6, 7, or 8 data bits)
 - Parity enable and parity type selection (even/odd)
 - Stop bit count (1 or 2 stop bits)
 - FIFO enable/disable control
- UARTIBRD/UARTFBRD - Baud Rate Divisor Registers
- UARTIBRD: 16-bit integer baud rate divisor
- UARTFBRD: 6-bit fractional baud rate divisor
 - Divides peripheral clock to generate serial bit clock
 - Supports fractional division for precise baud rate generation
- UARTFR - Flag Register (read-only)
 - TX FIFO full status flag
 - RX FIFO empty status flag
 - TX FIFO empty status flag
 - RX FIFO full status flag
 - UART busy flag (transmission in progress)
 - Clear To Send (CTS) modem status
- UARTIFLS - Interrupt FIFO Level Select Register
 - TX FIFO interrupt threshold level (1/8, 1/4, 1/2, 3/4, 7/8 full)
 - RX FIFO interrupt threshold level (1/8, 1/4, 1/2, 3/4, 7/8 full)
 - Configures when FIFO interrupts are triggered
 - Allows tuning of interrupt frequency versus latency
- UARTIMSC - Interrupt Mask Set/Clear Register
 - Enable/disable individual interrupt sources
 - Receive interrupt mask
 - Transmit interrupt mask
 - Receive timeout interrupt mask
 - Error interrupt masks (framing, parity, break, overrun)
 - Modem status interrupt masks (CTS)
- UARTRIS - Raw Interrupt Status Register (read-only)
 - Raw interrupt status before masking
 - Shows all active interrupt conditions
 - Useful for diagnostics and debugging
- UARTMIS - Masked Interrupt Status Register (read-only)
 - Interrupt status after applying mask (UARTIMSC)
 - Shows only enabled interrupts that are active

- Read by interrupt handler to determine service requirements
- UARTICR - Interrupt Clear Register (write-only)
 - Write-one-to-clear interrupt status bits
 - Clears corresponding bits in UARTRIS and UARTMIS
 - Write 1 to clear specific interrupt flags
 - Write 0 has no effect
- UARTDMACR - DMA Control Register
 - DMA enable for transmit direction (TXDMAE)
 - DMA enable for receive direction (RXDMAE)
 - DMA error handling control
 - Enables DMA request signaling to DMA controller

18. Universal Serial Bus (USB)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

18.1 Functional Overview

The USB subsystem provides support for USB full speed (12 Mbps) and high speed (480 Mbps) interface. This interface is primarily used for bulk data transfer, firmware updates and charging detect.

The USB controller supports up to 5 IN / 5 OUT endpoints plus 1 control. The FIFO sizing for each endpoint is dynamically configurable up to 4 kB, with the restriction that IN and OUT port sizing must be up to half the total FIFO size. The controller also supports DMA transfers to/from system memory.

The SoC has an integrated USB 2.0 PHY with support for suspend mode operation.

Battery charger detection is supported within the PHY to enable battery charge algorithm execution and control of the external battery charge / power management IC. The charger detection supports Battery Charging Specification 1.2 (BC1.2) and also supports other non-BC1.2 standards. See individual Apollo5 family SoC datasheets for details on battery charger detection functionality.

The CPU interface to the USB Controller provides access to the control/status registers and the FIFOs for each endpoint. It also generates interrupts to the CPU when packets are successfully transmitted or received, and when the core enters suspend mode or resumes from suspend mode. The USB controller interfaces to the CPU over the APB.

The RAM controller uses RAM to buffer packets between the CPU and USB. It takes the FIFO pointers from the endpoint controllers, converts them to address pointers within the RAM block and generates the RAM access control signals.

18.2 USB Reset

When a reset condition is detected on the USB, the USB Controller performs the following actions:

- Sets CFG0_FuncAddr to 0.
- Sets CFG3_ENDPOINT (index) to 0.
- Flushes all endpoint FIFOs.
- Clears all control/status registers.
- Enables all endpoint interrupts.
- Generates a Reset interrupt.

If the HSEnab bit in the CFG0 register was set, the USB Controller also tries to negotiate for high-speed operation. Whether high-speed operation is selected is indicated by the CFG0_HSMODE bit.

When the software receives a Reset interrupt, it should close any open pipes and wait for bus enumeration to begin.

18.3 Soft Connect/Disconnect

As mentioned earlier, the USB Controller can be configured to allow the connection of the controller to the USB PHY to be controlled by software, where the PHY can be switched between normal mode and non-driving mode by setting/clearing the CFG0_AMSPECIFIC bit. When the AMSPECIFIC bit is set to 1, the PHY is placed in its normal mode and the D+/D- lines of the USB bus are enabled. When the AMSPECIFIC bit is zero, the PHY is put into non-driving mode and D+ and D- are tri-stated. The USB Controller then appears to the CPU as if it has been disconnected.

After a hardware reset ($nRST = 0$), AMSPECIFIC is cleared to 0. The USB Controller will therefore appear disconnected until the software has set AMSPECIFIC to 1. The application software can then choose when to set the PHY to its normal mode. Systems with a lengthy initialization procedure may use this to ensure that initialization is complete and the system is ready to perform enumeration before connecting to the USB.

Once the AMSPECIFIC bit has been set to 1, the software can also simulate a disconnect by clearing this bit.

18.4 Employing DMA

The advantage of employing DMA is that it improves bus and processor utilization when loading or unloading the endpoint FIFOs, and is particularly useful when large blocks of data are transferred through Bulk endpoints. The USB controller provides two DMA operating modes to support this:

- **DMA Mode 0 (DMA0)** — a general-purpose DMA mode suitable for **all endpoint types**, including Bulk, Interrupt, and Isochronous. It provides a simple and robust way to service endpoint FIFOs using DMA across a wide range of transfer patterns, where one USB packet is transferred with each DMA request.
- **DMA Mode 1 (DMA1)** — a high-throughput DMA mode intended to **maximize Bulk transfer efficiency** by reducing software servicing overhead during sustained, max-packet traffic. This mode is best suited to large data moves where throughput and low CPU involvement are the primary goals, where multiple USB packets are transferred with each DMA request.

NOTE

Cache Coherency: When DMA is used, software must perform the required cache maintenance (clean/flush before DMA reads memory; invalidate after DMA writes memory). Failure to do so may cause stale data to be transmitted to the host or stale data to be read by the USB stack after reception.

NOTE

Isochronous+DMA: If Isochronous endpoints are used with DMA, DMA Mode 0 should always be selected.

18.4.1 DMA Mode 0 - Sharing the DMA Engine

DMA Mode 0 is a single-instance DMA implementation, where only one endpoint-direction transaction can be armed at a time. However, multiple endpoint-direction transactions may be requested by the USB stack. Therefore, the USB device driver must provide a central arbitration/queue to:

- Decide which endpoint + direction gets the DMA engine next
- Prevent one endpoint's DMA process from blocking other endpoints that become ready

18.4.1.1 IN Endpoint Requests

For IN endpoints, the TX DMA should be enqueued when the endpoint FIFO is able to accept the next data packet. This is how the queue should be handled for TX transactions:

1. The USB stack requests a TX transaction, which in turn enqueues a TX work-item when the IN endpoint is idle.
2. The software arbitrator picks up the work-item and arms the DMA0 engine accordingly.
3. When the DMA completes, the transfer completion event is propagated upward to the USB stack.
4. The software arbitrator picks up next work-item and arms the DMA0 engine accordingly, if any.

18.4.1.2 OUT Endpoint Requests

For OUT endpoints, the RX DMA should be enqueued when both conditions are met: endpoint buffer is assigned, and data is available in the OUT FIFO for unloading. This is how the queue should be handled for RX transactions:

1. The USB stack requests an RX transaction, the transfer details are stored.
2. A USB packet is received from the USB host.
3. When both (1) and (2) above are fulfilled (in any order), the RX work-item is enqueued.
4. The software arbitrator picks up the work-item and arms the DMA0 engine accordingly.
5. When DMA completed, the transfer completion event is propagated upward to the USB stack.
6. The software arbitrator picks up next work-item and arms the DMA0 engine accordingly, if any.

18.4.2 DMA Mode 1 - Handling Unknown-Length OUT Transactions

DMA Mode 1 detects the end of transfer when any of the following conditions occur:

- A short packet received
- A zero-length packet (ZLP) received
- An amount of data equivalent to the programmed DMA trigger size is received

DMA Mode 1 is incapable of detecting end-of-transfer when the host transfers a data stream whose size is:

- A multiple of MAXPACKET, and
- Smaller than the programmed DMA trigger size, and
- Does not provide a terminating ZLP.

This scenario is commonly encountered with Communication Device Class (CDC) devices.

If DMA Mode 1 is to be used when unknown-length OUT transactions are expected, a polling mechanism should be implemented to monitor the DMA Mode 1 registers for the "transferred-and-paused" OUT transaction condition, and finalize the DMA transaction upon detection.

18.4.3 Using DMA with IN Endpoints

18.4.3.1 DMA Mode 0

To use DMA0 to send a USB packet to an IN endpoint, the DMA controller should be programmed with:

- TOTCOUNT = DMA transfer size, up to the MPS value of the endpoint
- DMAReqMode = 0 and DMAReqEnb = 1 (DMA Mode 0)

With this configuration, the DMA engine will load up to 1 packet of data into the endpoint FIFO. Upon completion, a DMA complete interrupt will be triggered. Subsequently, the InPktRdy flag for the endpoint should be set to signal endpoint to start transmitting the data to the USB host. An endpoint interrupt will be generated when InPktRdy is cleared by the USB controller, indicating that the data has been successfully transferred to the USB host.

Since there is only one instance of the DMA Mode 0 register set, it is recommended to implement a software arbiter to control access to the DMA0 engine and share it among all endpoint-direction transactions, as described in Section 17.16.3.1.

18.4.3.2 DMA Mode 1

To use DMA1 to send multiple USB packets to an IN endpoint, the DMA controller should be programmed with:

- ADMATOTCOUNT = DMA transfer size as requested
- ADMAREQSIZE = MPS value of the endpoint.
- DMAReqMode = 1 and DMAReqEnb = 1 (DMA Mode 1)
- AutoSet = 1

With this configuration, the DMA will load data from endpoint buffer into the endpoint FIFO automatically once FIFO space is made available after a successful transmission of the packet to the USB host.

Furthermore, with the AutoSet feature enabled, the InPktRdy flag is automatically set once an MPS-sized amount of data is moved into the FIFO by the DMA. This allows the data transfer to keep going without requiring intervention by the CPU until all data is loaded into the FIFO. Once all data is loaded into the endpoint FIFO, the ADMA complete interrupt will be triggered. Subsequently, if the last packet loaded is less than the MPS, the InPktRdy flag will not be set automatically and will therefore need to be set manually by the CPU to allow it to be sent to the USB host.

18.4.4 Using DMA with OUT Endpoints

18.4.4.1 DMA Mode 0

To use DMA0 to receive a USB packet from an OUT endpoint, the DMA controller should be programmed with the following settings when the OutPktRdy flag of the endpoint is set:

- TOTCOUNT = DMA transfer size, up to the MPS value of the endpoint
- DMAReqMode = 0 and DMAReqEnb = 1 (DMA Mode 0)

With this configuration, the DMA engine will unload up to one MPS-sized amount of data from the endpoint FIFO into the endpoint buffer. Upon completion, a DMA complete interrupt will be triggered, indicating that the data has been successfully unloaded from the endpoint FIFO.

Since there is only one instance of the DMA Mode 0 register set, it is recommended to implement a software arbiter to control access to the DMA0 engine and share it among all endpoint-direction transactions, as described in 18.4.1.

18.4.4.2 DMA Mode 1

To use DMA1 to receive multiple USB packets from an OUT endpoint, the DMA controller should be programmed with:

- ADMATOTCOUNT = DMA transfer size as requested
- ADMAREQSIZE = MPS value of the endpoint
- DMAReqMode = 1 and DMAReqEnb = 1 (DMA Mode 1)
- AutoClear = 1

With this configuration, the DMA will unload data from the endpoint FIFO into the endpoint buffer automatically whenever an MPS-sized amount of data is available in the endpoint FIFO. Furthermore, with the AutoClear feature enabled, the OutPktRdy flag is automatically cleared once an MPS-sized amount of data is unloaded from the endpoint FIFO. This allows the data transfer to keep going without requiring intervention by the CPU until any of the stop conditions below occur:

- A short packet received (endpoint interrupt will be triggered)
- A zero-length packet (ZLP) received (endpoint interrupt will be triggered)
- An amount of data equivalent to the programmed DMA trigger size is received (ADMA complete interrupt will be triggered)

In the case where a short packet is received, the remaining data must be unloaded manually by the CPU, followed by clearing the OutPktRdy flag.

Note that, if DMA Mode 1 is used to handle unknown-length OUT transactions, the "transferred-and-paused" detection as described in Section 17.16.3.2 should be implemented, to ensure that data from the USB host is not unexpectedly withheld in the endpoint buffer.

18.5 High-speed Mode

If the HSENAB bit of the CFG0 register is set (it is set by default at power-up), then when the USB Controller is reset by a USB reset signal from the hub, it will negotiate for High-speed mode. If the USB host, and all hubs between the USB Controller and the host, support High-speed operation then the HSMODE bit of the CFG0 register will be set and the USB Controller will operate in High-speed mode. If the High-speed negotiation fails, the HSMODE bit will not be set and the USB Controller will operate in Full-speed mode only.

18.6 USB Interrupt Handling

When the CPU is interrupted with a USB interrupt, it needs to read the interrupt status register to determine which endpoint(s) have caused the interrupt and jump to the appropriate routine. If multiple endpoints have caused the interrupt, Endpoint 0 should be serviced first, followed by the other endpoints. The Suspend interrupt should be serviced last.

A flowchart for the USB Interrupt Service Routine is shown in Figure 57.

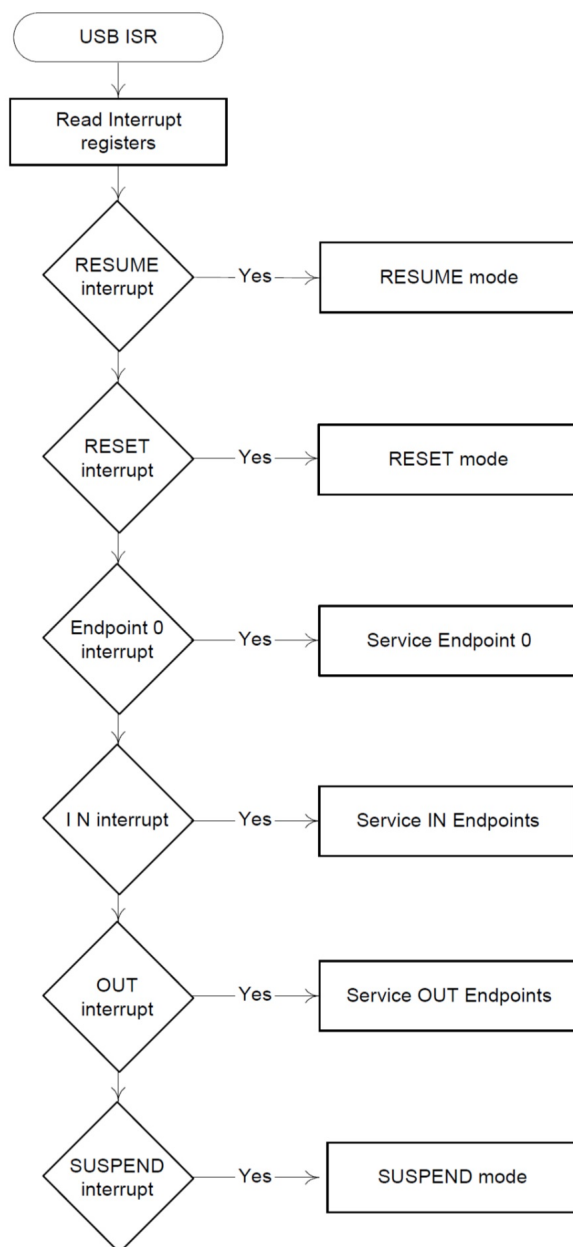


Figure 57. USB Interrupt Service Routine

18.7 Index 0 Register Fields

NOTE

This section is included here to describe how certain fields in the USB Controller's IDX0 register are muxed and named to reflect that muxing. These fields are referenced throughout this chapter and, in the interest of clarity and brevity, references to these multi-function fields will use the field name appropriate to the endpoint that is being discussed.

The Index 0 Register (IDX0) has fields whose functions are different depending on whether the Endpoint index has been set in CFG3_ENDPOINT to Endpoint 0 or to a non-zero endpoint, e.g., IN Endpoint1 - IN Endpoint5 or OUT Endpoint1 - OUT Endpoint5.

Table 90 below lists the functionality for these fields. See the field descriptions in the IDX0 register for more information.

Table 90: Index 0 (IDX0) Register Multi-function Fields

Field Name in Register Set	ENDPOINT0 Field Name	ENDPOINT0 Use	All Other Endpoint Field Name	All Other Endpoint Use
IncompTxServiceSetupEnd	ServiceSetupEnd	CPU writes a 1 to this bit to clear the SetupEnd bit. Cleared automatically.	IncompTx	Indicates where a large packet has been split into 2 or 3 packets for transmission but insufficient I tokens have been received to send all the parts.
ClrDataTogServicedOutPktRdy	ServicedOutPktRdy	CPU writes a 1 to this bit to clear the OutPktRdy bit. Cleared automatically.	ClrDataTog	CPU writes a 1 to this bit to reset the endpoint IN data toggle to 0.
SentStallSendStall	SendStall	CPU writes a 1 to this bit to terminate the current transaction. The STALL handshake will be transmitted and then this bit will be cleared automatically.	SentStall	Set when a STALL handshake is transmitted. The FIFO is flushed and the InPktRdy bit is cleared.
SendStallSetupEnd	SetupEnd	Set when a control transaction ends before the DataEnd bit has been set. An interrupt will be generated and the FIFO flushed. Cleared by CPU writing a 1 to the ServicedSetupEnd bit.	SendStall	CPU writes a 1 to this bit to issue a STALL handshake to an IN token. CPU clears this bit to terminate the stall condition.

Table 90: Index 0 (IDX0) Register Multi-function Fields

Field Name in Register Set	ENDPOINT0 Field Name	ENDPOINT0 Use	All Other Endpoint Field Name	All Other Endpoint Use
FlushFIFODataEnd	DataEnd	CPU sets this bit: 1. When setting InPktRdy for the last data packet. 2. When clearing OutPktRdy after unloading the last data packet. 3. When setting InPktRdy for a zero length data packet. Cleared automatically.	FlushFIFO	CPU writes a 1 to this bit to flush the next packet to be transmitted from the endpoint IN FIFO. The FIFO pointer is reset and the InPktRdy bit is cleared. May be set simultaneously with InPktRdy to abort the packet that has just been loaded into the FIFO.
UnderRunSentStall	SentStall	Set when a STALL handshake is transmitted. The CPU should clear this bit.	UnderRun	In ISO mode, this bit is set when a zero length data packet is sent after receiving an IN token with the InPktRdy bit not set. In Bulk/Interrupt mode, this bit is set when a NAK is returned in response to an IN token. The CPU should clear this bit.
FIFONotEmptyInPktRdy	InPktRdy	CPU sets this bit after loading a data packet into the FIFO. Cleared automatically when the data packet has been transmitted. An interrupt is generated when this bit is cleared (if enabled).	FIFONotEmpty	Set when there is at least 1 packet in the IN FIFO.
InPktRdyOutPktRdy	OutPktRdy	Set when a data packet has been received. An interrupt is generated when this bit is set (if enabled). CPU clears this bit by setting the Serviced-OutPktRdy bit.	InPktRdy	CPU sets this bit after loading a data packet into the FIFO. Cleared automatically when a data packet has been transmitted. If FIFO is double-buffered, it is also automatically cleared when there is space for a second packet in the FIFO. An interrupt is generated (if enabled) when the bit is cleared.

18.8 Response to USB Conditions or Host Actions

The USB Controller core responds automatically to certain conditions on the USB bus or actions by the host. The details are given in the following sections.

18.8.1 Stall Issued to Control Transfer

The USB Controller core will automatically issue a STALL handshake to a Control transfer under the following conditions:

1. The host sends more data during an OUT Data phase of a Control transfer than was specified in the device request during the SETUP phase. This condition is detected by the USB Controller when the host sends an OUT token (instead of an IN token) after the CPU has unloaded the last OUT packet and set `IDX0_DATAEND`.
2. The host requests more data during an IN data phase of a Control transfer than was specified in the device request during the SETUP phase. This condition is detected by the USB Controller when the host sends an IN token (instead of an OUT token) after the CPU has cleared `IDX0_InPktRdy` and set `IDX0_DATAEND` in response to the ACK issued by the host to what should have been the last packet.
3. The Host sends more than `IDX0_MAXPAYLOAD` data with an OUT data token.
4. The Host sends more than a zero length data packet for the OUT Status phase.

18.8.2 Zero Length OUT Data Packets in Control Transfers

A zero-length OUT data packet is used to indicate the end of a Control transfer. In normal operation, such packets should only be received after the entire length of the device request has been transferred (i.e., after the CPU has set `IDX0_DATAEND`). If, however, the host sends a zero-length OUT data packet before the entire length of device request has been transferred, this signals the premature end of the transfer. In this case, the USB Controller will automatically flush any IN token loaded by CPU ready for the Data phase from the FIFO and set `IDX0_SETUPEND`.

18.9 Suspend/Resume

When the USB Controller sees no activity on the USB for 3 ms, it will go into Suspend mode. If the Suspend interrupt has been enabled, a Suspend interrupt will also be generated. At this point, the USB Controller can then be left active or the application may arrange to disable the USB Controller by stopping its clock. When in Suspend mode, the CFG0_SUSPEN bit will go low (if enabled).

The USB may exit Suspend mode by sending Resume signaling on the bus. If the Resume interrupt is enabled, an interrupt will be generated.

No Resume interrupt is generated when Suspend mode is exited by the CPU. When the CFG0_RESUME bit is set, then the USB Controller will exit Suspend mode. When this bit is set, the USB Controller will exit Suspend mode and drive Resume signaling onto the bus. The CPU should clear this bit after 10 ms (a maximum of 15 ms) to end Resume signaling.

Alternatively, software may perform a “Remote Wakeup”. How the USB Controller responds depends on whether it has been left active or inactive during the suspend, which is discussed below.

18.9.1 USB Controller Active During Suspend

When the USB Controller goes into Suspend mode, the transceiver will also be put into Suspend mode by the SUSPENDM line if the CFG0_Enabl bit is set. When the USB Controller remains active, however, it can detect when Resume signaling occurs on the USB. It will then bring the transceiver out of Suspend mode and generate a Resume interrupt.

18.9.2 USB Controller Inactive During Suspend

When the Suspend interrupt described above is received, the software may disable the USB Controller by stopping its clock. However, the USB Controller will not then be able to detect Resume signaling on the USB.

18.9.3 Remote Wakeup

If the USB Controller is in Suspend mode and the software wants to initiate a remote wakeup, it should set the CFG0_Resume bit to 1. If the clock to the USB Controller has been stopped, it will need to be restarted before this write can occur (by clearing CLKCTRL_PHYAPBCLKDIS bit).

The software should leave the CFG0_Resume bit set for approximately 10 ms (minimum of 2 ms, a maximum of 15 ms) then reset it to 0. By this time the hub should have taken over driving Resume signaling on the USB.

NOTE

No Resume interrupt will be generated when the software initiates a remote wakeup.

18.10 Start of Frame Packets

The USB Controller should receive a Start-Of-Frame packet from the host once every millisecond when in Full-speed mode, or every 125 microseconds when in High-speed mode. When the SOF packet is received, the 11-bit frame number contained in the packet is written into the CFG3_FRMNUM field. In normal running, the core will also generate a SOF interrupt (CFG2_SOF bit), if enabled by setting the CFG2_SOF bit, to indicate that the core's internal frame timer is starting a new frame.

Following a reset or a resume, there is a period of synchronization during which a SOF_PULSE and a SOF interrupt will not necessarily be generated following each received SOF packet. After synchronization has been achieved, the core expects to receive a SOF packet every millisecond (or every 125 microseconds). If no SOF packet is received after 1.00358 ms (or 125.125 μ s), it is assumed that the packet has been lost and a SOF_PULSE (together with a SOF interrupt, if enabled) will still be generated though the Frame register will not be updated. The USB Controller will continue to generate a SOF_PULSE every millisecond (or 125 microseconds) and will resynchronize these pulses to the received SOF packets when these packets are successfully received again.

18.11 Dynamic FIFO Sizing

The USB Controller is configured to have a single overall FIFO size of 4 kB, areas of which may then be allocated to the different endpoints, excluding endpoint 0, when the USB Controller is initialized.

The allocation of FIFO space to the different endpoints requires the specification for each IN and OUT endpoint of:

- The start address of the FIFO within the RAM block
- The maximum size of packet to be supported
- Whether double-buffering is required

(These last two together define the amount of space that needs to be allocated to the FIFO.)

These details may be specified through the four register fields shown in Table 91, which are in the Indexed area of the USB Controller register map. The `IDX2_INFIFOSZ` and `IDX2_OUTFIFOSZ` fields are each 5-bit fields. The most significant bit of each is set if double-packet buffering is supported. The four lower bits of each field is set to the maximum packet size to be allowed, where $2^{(D3:D0 + 3)}$ is the maximum packet size. E.g., if `D3:D0 = 7`, then the maximum packet size is 1024 bytes. Since the total FIFO size is limited to 4 kB, then the maximum value of either of these 4-bit size fields is 9. If double-packet buffering is not selected, then the FIFO will be the same size as the maximum packet size, otherwise it will be twice the size.

Table 91: USB Controller Register Map Additions for Dynamic FIFO Sizing

Address	Register_Field	Description
1A	<code>IDX2_INFIFOSZ</code>	IN Endpoint FIFO size
1B	<code>IDX2_OUTFIFOSZ</code>	OUT Endpoint FIFO size
1C, 1D	<code>FIFOADD_INFIFOADD</code>	IN Endpoint FIFO address
1E, 1F	<code>FIFOADD_OUTFIFOADD</code>	OUT Endpoint FIFO address

NOTE

The option of setting FIFO sizes dynamically only applies to Endpoints 1 to 5. The Endpoint 0 FIFO is fixed to 64 bytes and starts at address 0. It is the responsibility of the firmware (and the system designer) to ensure that all the IN and OUT endpoints that are active in the current USB configuration have a block of RAM assigned exclusively to them that is at least as large as the maximum packet size set for the endpoint.

The `FIFOADD_INFIFOADD` and `FIFOADD_OUTFIFOADD` fields are each 13-bits in length and are programmed to specify the start address of an endpoint FIFO, where the field value $\times 8$ is the start address. Since the size of the FIFO for Endpoint 0 is fixed at 64 bytes, the start address of the FIFO for any other Endpoint is no lower than address 0x40, which means that its address field value is at least 8.

18.12 Endpoint 0 Handling

Endpoint 0 is the main control endpoint of the core. As such, the routines required to service Endpoint 0 are more complicated than those required to service other endpoints.

The software is required to handle all the Standard Device Requests that may be received via Endpoint 0. These are described in Universal Serial Bus Specification, Revision 2.0, Chapter 9. The protocol for these device requests involves different numbers and types of transaction per transfer. To accommodate this, the CPU needs to take a state machine approach to command decoding and handling.

The Standard Device Requests can be divided into three categories: Zero Data Requests (in which all the information is included in the command), Write Requests (in which the command will be followed by additional data), and Read Requests (in which the device is required to send data back to the host).

This section examines the sequence of events that the software must perform to process the different types of device request.

NOTE

The Setup packet associated with any Standard Device Request should include an 8-byte command. Any Setup packet containing a command field of anything other than 8 bytes will be automatically rejected by the USB Controller core.

18.12.1 Zero Data Requests

Zero data requests have all their information included in the 8-byte command and require no additional data to be transferred. Examples of zero data Standard Device Requests are: SET_FEATURE, CLEAR_FEATURE, SET_ADDRESS, SET_CONFIGURATION, SET_INTERFACE.

The sequence of events will begin, as with all requests, when the software receives an Endpoint 0 interrupt. The `IDX0_OutPktRdy` bit will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO, decoded and the appropriate action taken. For example if the command is SET_ADDRESS, the 7-bit address value contained in the command should be written to the `CFG0_FuncAddr` field.

Both the `IDX0_ServicedOutPktRdy` bit (indicating that the command has been read from the FIFO) and the `IDX0_DataEnd` bit (indicating that no further data is expected for this request) should be set.

When the host moves to the status stage of the request, a second Endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software: the second interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the `IDX0_ServicedOutPktRdy` bit and the `IDX0_SendStall` bit should be set. When the host moves to the status stage of the request, the USB Controller will send a STALL to tell the host that the request was not executed. A second Endpoint 0 interrupt will be generated and the `IDX0_SentStall` bit will be set.

If the host sends more data after the `DataEnd` bit has been set, then the USB Controller will send a STALL. An Endpoint 0 interrupt will be generated and the `SentStall` bit will be set.

18.12.2 Write Requests

Write requests involve an additional packet (or packets) of data being sent from the host after the 8-byte command. An example of a write Standard Device Request is: SET_DESCRIPTOR.

The sequence of events will begin, as with all requests, when the software receives an Endpoint 0 interrupt. The `IDX0_OutPktRdy` bit will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO and decoded.

As with a zero data request, the `IDX0_ServicedOutPktRdy` bit (indicating that the command has been read from the FIFO) should be set, but in this case the `IDX0_DataEnd` bit should not be set (indicating that more data is expected).

When a second Endpoint 0 interrupt is received, the `IDX0` register should be read to check the endpoint status. The `IDX0_OutPktRdy` bit should be set to indicate that a data packet has been received. The `IDX2_ENDPTOUTCOUNT` field should then be read to determine the size of this data packet. The data packet can then be read from the Endpoint 0 FIFO.

If the length of the data associated with the request (indicated by the `wLength` field in the command) is greater than the maximum packet size for Endpoint 0, further data packets will be sent. In this case, the `IDX0_ServicedOutPktRdy` bit should be set, but the `IDX0_DataEnd` bit should not be set.

When all the expected data packets have been received, the `IDX0_ServicedOutPktRdy` bit and the `DataEnd` bit (indicating that no more data is expected) should be set.

When the host moves to the status stage of the request, another Endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software, the interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the `IDX0_ServicedOutPktRdy` bit and the `IDX0_SendStall` bit should be set. When the host sends more data, the USB Controller will send a STALL to tell the host that the request was not executed. An Endpoint 0 interrupt will be generated and the `IDX0_SentStall` bit will be set.

If the host sends more data after the `DataEnd` has been set, then the USB Controller will send a STALL. An Endpoint 0 interrupt will be generated and the `SentStall` bit will be set.

18.12.3 Read Requests

Read requests have a packet (or packets) of data sent from the function to the host after the 8-byte command. Examples of read Standard Device Requests are: `GET_CONFIGURATION`, `GET_INTERFACE`, `GET_DESCRIPTOR`, `GET_STATUS`, `SYNCH_FRAME`.

The sequence of events will begin, as with all requests, when the software receives an Endpoint 0 interrupt. The `IDX0_OutPktRdy` bit will also have been set. The 8-byte command should then be read from the Endpoint 0 FIFO and decoded. The `IDX0_ServicedOutPktRdy` bit (indicating that the command was read from the FIFO) should then be set.

The data to be sent to the host should then be written to the Endpoint 0 FIFO. If the data to be sent is greater than the maximum packet size for Endpoint 0, only the maximum packet size should be written to the FIFO. The `IDX0_InPktRdy` bit (indicating that there is a packet in the FIFO to be sent) should be set. When the packet has been sent to the host, another Endpoint 0 interrupt will be generated and the next data packet can be written to the FIFO.

When the last data packet has been written to the FIFO, the `IDX0_InPktRdy` bit and the `IDX0_DataEnd` bit (indicating that there is no more data after this packet) should be set.

When the host moves to the status stage of the request, another Endpoint 0 interrupt will be generated to indicate that the request has completed. No further action is required from the software: the interrupt is just a confirmation that the request completed successfully.

If the command is an unrecognized command, or for some other reason cannot be executed, then when it has been decoded, the `IDX0_ServicedOutPktRdy` bit and the `IDX0_SendStall` bit should be set. When the host requests data, the USB Controller will send a STALL to tell the host that the request was not executed. An Endpoint 0 interrupt will be generated and the `IDX0_SentStall` bit will be set.

If the host requests more data after the `DataEnd` has been set, then the USB Controller will send a STALL. An Endpoint 0 interrupt will be generated and the `SentStall` bit will be set.

18.12.4 Endpoint 0 States

As illustrated in Figure 58, the Endpoint 0 control needs three modes – IDLE, TX and RX – corresponding to the different phases of the control transfer and the states Endpoint 0 enters for the different phases of the transfer. The default mode on power-up or reset should be IDLE.

IDX0_OutPktRdy becoming set when Endpoint 0 is in IDLE state indicates a new device request. Once the device request is unloaded from the FIFO, the USB Controller decodes the descriptor to find whether there is a Data phase and, if so, the direction of the Data phase for the control transfer (in order to set the FIFO direction).

Depending on the direction of the Data phase, Endpoint 0 goes into either TX state or RX state. If there is no Data phase, Endpoint 0 remains in IDLE state to accept the next device request.

The actions that the CPU needs to take at the different phases of the possible transfers (e.g. Loading the FIFO, Setting InPktRdy) are indicated in the diagram on the following page.

NOTE The USB Controller changes the FIFO direction depending on the direction of the Data phase independently of the CPU.

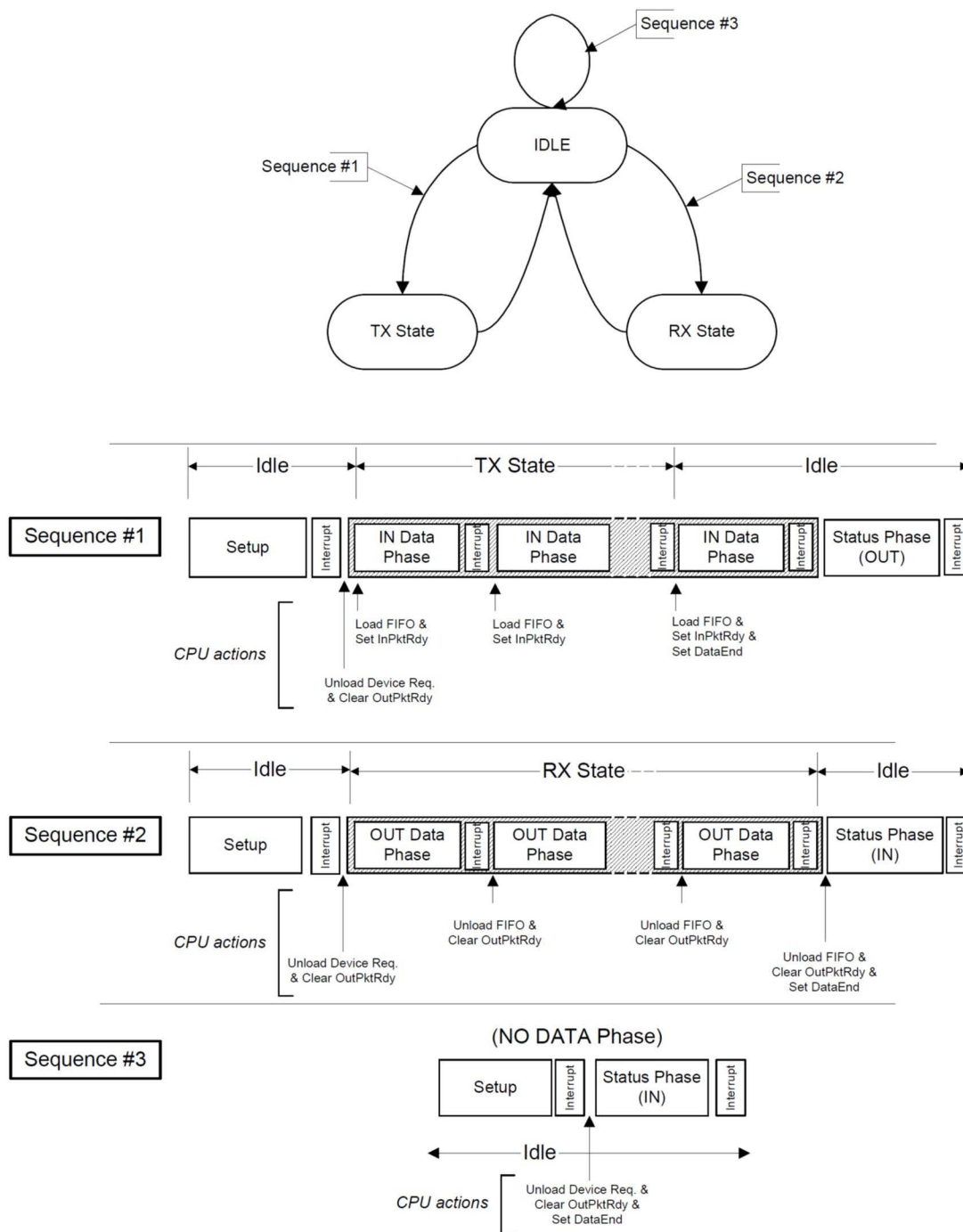


Figure 58. Endpoint 0 States

18.12.5 Endpoint 0 Service Routine

An Endpoint 0 interrupt is generated:

- When the core sets the `IDX0_OutPktRdy` bit after a valid token has been received and data has been written to the FIFO.
- When the core clears the `IDX0_InPktRdy` bit after the packet of data in the FIFO has been successfully transmitted to the host.
- When the core sets the `IDX0_SentStall` bit after a control transaction is ended due to a protocol violation.
- When the core sets the `IDX0_SetupEnd` bit because a control transfer has ended before `IDX0_DataEnd` bit is set.

Whenever the Endpoint 0 service routine is entered, the firmware must first check to see if the current control transfer has been ended due to either a STALL condition or a premature end of control transfer. If the control transfer ends due to a STALL condition, the `IDX0_SentStall` bit would be set. If the control transfer ends due to a premature end of control transfer, the `SetupEnd` bit would be set. In either case, the firmware should abort processing the current control transfer and set the state to IDLE.

Once the firmware has determined that the interrupt was not generated by an illegal bus state, the next action taken depends on the Endpoint state as shown in Figure 59.

If Endpoint 0 is in IDLE state, the only valid reason an interrupt can be generated is as a result of the core receiving data from the USB bus. The service routine must check for this by testing the `OutPktRdy` bit. If this bit is set, then the core has received a SETUP packet. This must be unloaded from the FIFO and decoded to determine the action the core must take. Depending on the command contained within the SETUP packet, Endpoint 0 will enter one of three states:

- If the command is a single packet transaction (`SET_ADDRESS`, `SET_INTERFACE`, etc.) without any data phase, the endpoint will remain in IDLE state.
- If the command has an OUT data phase (`SET_DESCRIPTOR`, etc.) the endpoint will enter RX state.
- If the command has an IN data phase (`GET_DESCRIPTOR`, etc.) the endpoint will enter TX state.

If the endpoint is in TX state, the interrupt indicates that the core has received an IN token and data from the FIFO has been sent.

The firmware must respond to this either by placing more data in the FIFO if the host is still expecting more data2 or by setting the `DataEnd` bit to indicate that the data phase is complete. Once the data phase of the transaction has been completed,

Endpoint 0 should be returned to IDLE state to await the next control transaction.

If the endpoint is in RX state, the interrupt indicates that a data packet has been received. The firmware must respond by unloading the received data from the FIFO. The firmware must then determine whether it has received all of the expected data2. If it has, the firmware should set the `DataEnd` bit and return Endpoint 0 to IDLE state. If more data is expected, the firmware should set the `IDX0_ServicedOutPktRdy` bit to indicate that it has read the data in the FIFO and leave the endpoint in RX state.

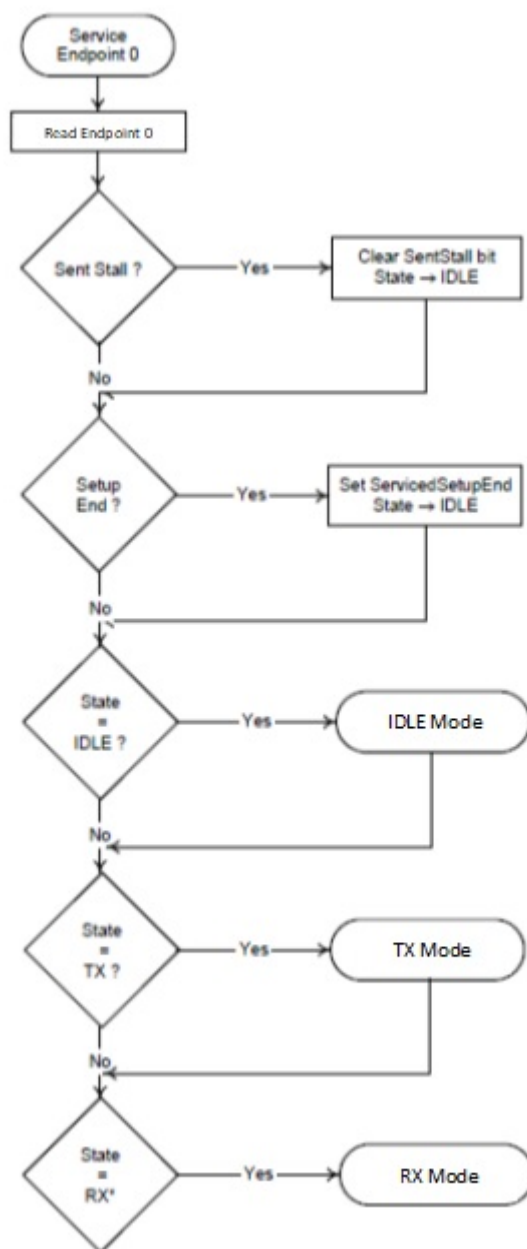


Figure 59. Endpoint 0 Service Routine

18.12.5.1 Idle Mode

IDLE mode is the mode the Endpoint 0 control needs to select at power-on or reset and is the mode to which the Endpoint 0 control should return when the RX and TX modes are terminated.

It is also the mode in which the SETUP phase of control transfer is handled as outlined in Figure 60 below.

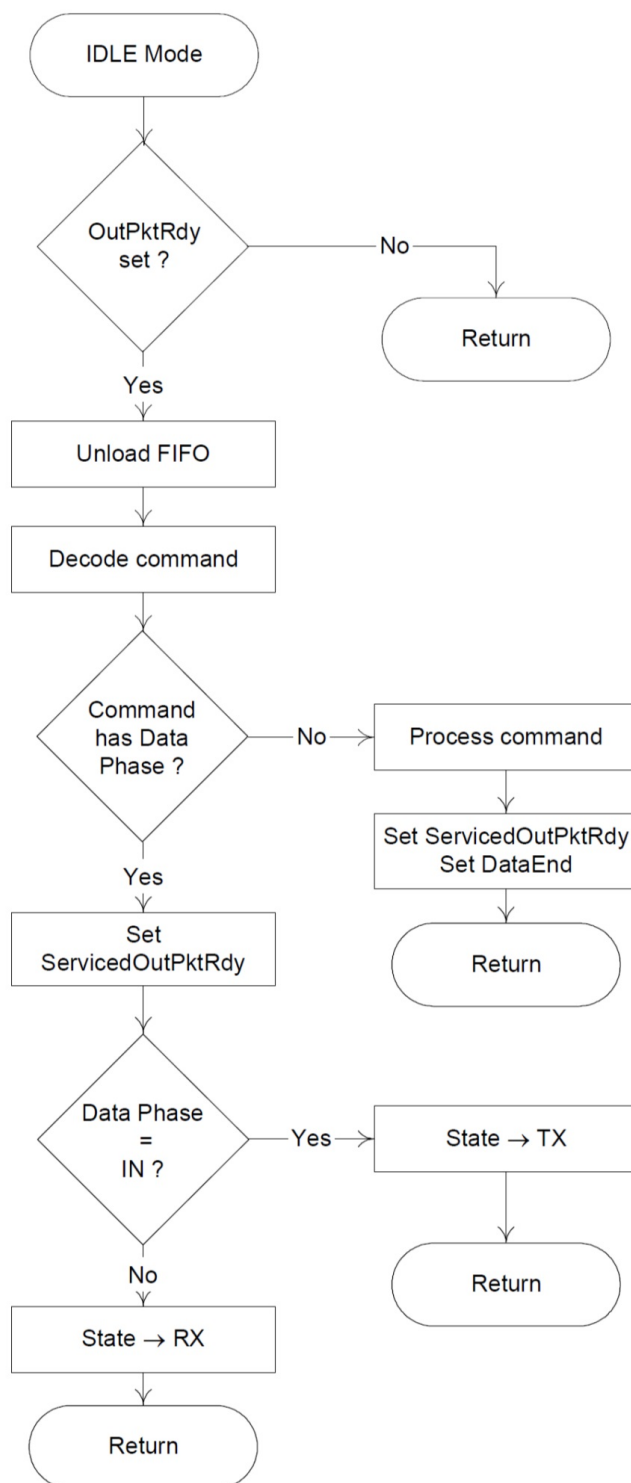


Figure 60. SETUP Phase of Control Transfer

18.12.5.2 TX Mode

When the endpoint is in TX state, all arriving IN tokens need to be treated as part of a Data phase until the required amount of data has been sent to the host. If either a SETUP or an OUT token is received whilst the endpoint is in the TX state, this will cause a Setup End condition to occur as the core expects only IN tokens.

Three events can cause TX mode to be terminated before the expected amount of data has been sent:

- The host sends an invalid token causing a `IDX0_SetupEnd` condition.
- The firmware sends a packet containing less than the maximum packet size for Endpoint 0 (`IDX0_MAXPAYLOAD`).
- The firmware sends an empty data packet.

Until the transaction is terminated, the firmware simply needs to load the FIFO when it receives an interrupt which indicates that a packet has been sent from the FIFO. (An interrupt is generated when `InPktRdy` is cleared.)

When the firmware forces the termination of a transfer (by sending a short or empty data packet), it should set the `DataEnd` bit to indicate to the core that the Data phase is complete and that the core should next receive an acknowledge packet.

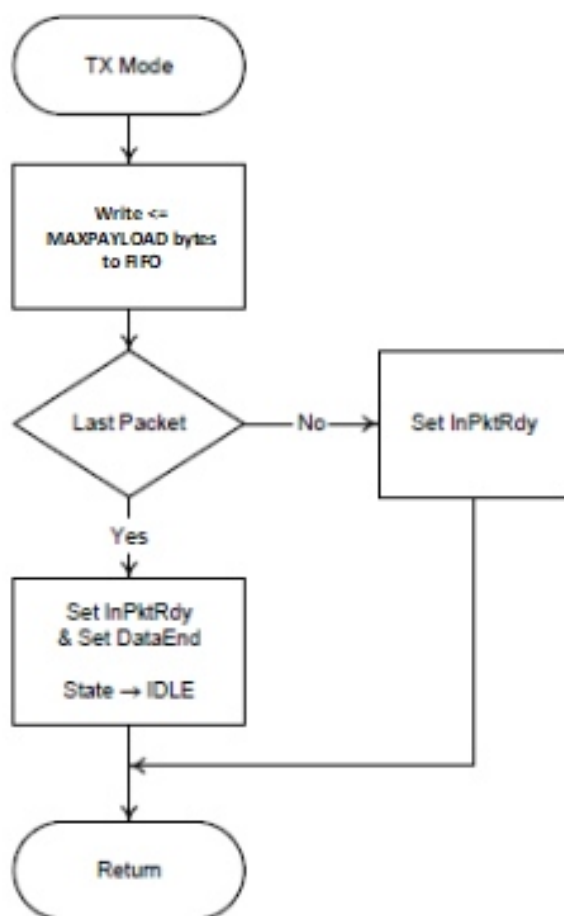


Figure 61. IN Data Phase for Control Transfer

18.12.5.3 RX Mode

In RX mode, all arriving data should be treated as part of a Data phase until the expected amount of data has been received. If either a SETUP or an IN token is received while the endpoint is in RX state, this will cause a Setup End condition to occur as the core expects only OUT tokens.

Three events can cause RX mode to be terminated before the expected amount of data has been received:

- The host sends an invalid token causing an `IDX0_SetupEnd` condition
- The host sends a packet which contains less than the maximum packet size for Endpoint 0
- The host sends an empty data packet

Until the transaction is terminated, the firmware simply needs to unload the FIFO when it receives an interrupt which indicates that new data has arrived (`IDX1_OutPktRdy` set) and to clear `OutPktRdy` by setting the `IDX0_ServicedOutPktRdy` bit.

When the firmware detects the termination of a transfer (by receiving either the expected amount of data or an empty data packet), it should set the `DataEnd` bit to indicate to the core that the Data phase is complete and that the core should receive an acknowledge packet next.

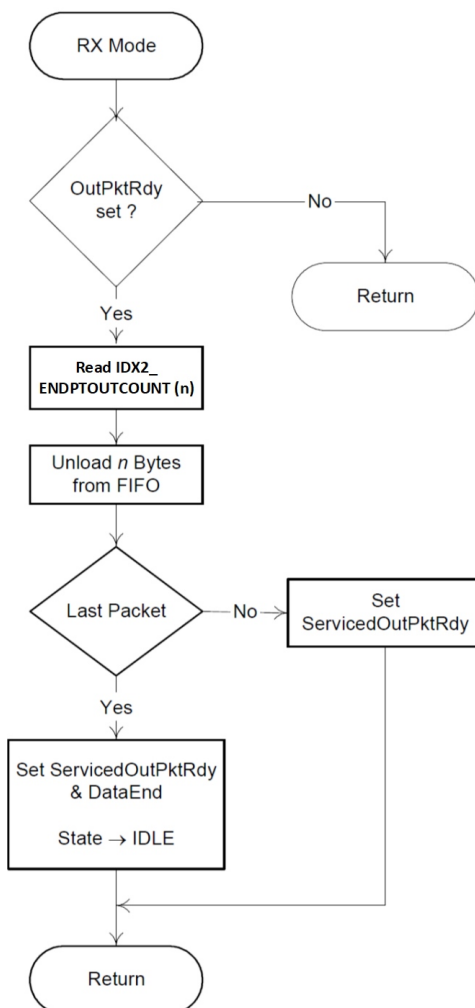


Figure 62. OUT Data Phase for Control Transfer

18.12.6 Error Handling

A control transfer may be aborted due to a protocol error on the USB, the host prematurely ending the transfer, or if the function controller software wishes to abort the transfer (e.g., because it cannot process the command).

The USB Controller will automatically detect protocol errors and send a STALL packet to the host under the following conditions:

- The host sends more data during the OUT Data phase of a write request than was specified in the command. This condition is detected when the host sends an OUT token after the `IDX0_DataEnd` bit has been set.
- The host request more data during the IN Data phase of a read request than was specified in the command. This condition is detected when the host sends an IN token after the `DataEnd` bit has been set.
- The host sends more than `IDX0_MAXPAYLOAD` data bytes in an OUT data packet.
- The host sends a non-zero length `DATA1` packet during the `STATUS` phase of a read request.

When the USB Controller has sent the STALL packet, it sets the `IDX0_SentStall` bit and generates an interrupt. When the software receives an Endpoint 0 interrupt with the `SentStall` bit set, it should abort the current transfer, clear the `SentStall` bit, and return to the IDLE state.

If the host prematurely ends a transfer by entering the `STATUS` phase before all the data for the request has been transferred, or by sending a new `SETUP` packet before completing the current transfer, then the `SetupEnd` bit will be set and an Endpoint 0 interrupt generated. When the software receives an Endpoint 0 interrupt with the `SetupEnd` bit set, it should abort the current transfer, set the `IDX0_ServicedSetupEnd` bit, and return to the IDLE state. If the `OutPktRdy` bit is set this indicates that the host has sent another `SETUP` packet and the software should then process this command.

If the software wants to abort the current transfer, because it cannot process the command or has some other internal error, then it should set the `SendStall` bit. The USB Controller will then send a STALL packet to the host, set the `SentStall` bit and generate an Endpoint 0 interrupt.

18.13 IN Endpoint Packet Handling

The sizes of the IN FIFOs for Endpoints 1 to 5 are selected through the INFIFOSZ field of the IDX2 register. The maximum size of data packets that may be placed in an IN endpoint's FIFO for transmission is programmable and is determined by the values written to the INDX0_MAXPAYLOAD field.

The use of single or double packet buffering is part of the specification for the endpoint FIFO – see “Dynamic FIFO Sizing” on page 226.) When double packet buffering is enabled, two data packets can be buffered in the FIFO. When single packet buffering is enabled, only one packet can be buffered even if the packet is less than half the FIFO size.

NOTE

The maximum packet size set for any endpoint must not exceed the FIFO size. Also note that unexpected results may occur if the MAXPAYLOAD field is written to while data is in the FIFO.

18.13.1 Single Packet Buffering

If the size of the IN endpoint FIFO is less than twice the maximum packet size for this endpoint (as set in the IDX2_INFIFOSZ field), only one packet can be buffered in the FIFO and single packet buffering is enabled.

As each packet to be sent is loaded into the IN FIFO, the InPktRdy bit in the IDX0 register needs to be set. If the IDX0_AUTOSET field is set, the IDX0_InPktRdy bit will be automatically set when a maximum-sized packet is loaded into the FIFO. For packet sizes less than the maximum and where IDX0_AUTOSET may not be used (high bandwidth Isochronous/Interrupt transactions), IDX0_InPktRdy will always have to be set manually (i.e., by the CPU).

When the IDX1_InPktRdy bit is set, either manually or automatically, the packet is deemed ready to be sent. The FIFONotEmpty bit in IDX0 is also set.

When the packet has been successfully sent, both InPktRdy and FIFONotEmpty are cleared and the appropriate IN endpoint interrupt generated (if enabled). The next packet can then be loaded into the FIFO.

18.13.2 Double Packet Buffering

NOTE

Double packet buffering is *disabled* if DPKTBUFDIS field of the IDX0 register is set. The default setting for this bit is enabled. However, for double packet buffering to be enabled, bit 4 of the 5-bit IDX2_INFIFOSZ field must be set.

The following conditions must exist to enable buffering of two data packets (double packet buffering):

- The size of the IN endpoint FIFO (as set in the IDX2_INFIFOSZ field) is at least twice the maximum packet size for the endpoint (as set in the IDX1_MAXPAYLOAD field).
- Bit 4 of the IDX2_INFIFOSZ field must be set.
- DPKTBUFDIS field of the IDX0 register is cleared.

When the first packet to be received is loaded into the OUT FIFO, the IDX1_OutPktRdy field is set and the appropriate OUT endpoint interrupt is generated (if enabled) to signal that a packet can now be unloaded from the FIFO.

As each packet to be sent is loaded into the IN FIFO, the `IDX1_InPktRdy` bit needs to be set. If the `IDX0_AUTOSET` field is set, the `InPktRdy` bit will be automatically set when a maximum-sized packet is loaded into the FIFO. For packet sizes less than the maximum, `InPktRdy` will always have to be set manually (i.e., by the CPU).

When the `InPktRdy` bit is set, either manually or automatically, the packet is deemed ready to be sent. The `FIFONotEmpty` bit in `IDX0` is also set.

After the first packet is loaded, `InPktRdy` will then immediately be cleared and an interrupt is generated. A second packet can now be loaded into the IN FIFO and `InPktRdy` set again (either manually or automatically if the packet is the maximum size). Both packets are now ready to be sent.

When the first packet has been successfully sent, `InPktRdy` is cleared and the appropriate IN endpoint interrupt generated (if enabled) to signal that another packet can now be loaded into the IN FIFO. The state of the `FIFONotEmpty` bit at this point indicates how many packets may be loaded. If the `FIFONotEmpty` bit is set then there is another packet in the FIFO and only one more packet can be loaded. If the `FIFONotEmpty` bit is clear then there are no packets in the FIFO and two more packets can be loaded.

18.13.3 High Bandwidth Isochronous Endpoints

In High-speed mode, Isochronous IN endpoints can transmit up to three 'USB' packets in any microframe, with a payload of up to 1024 bytes in each packet, corresponding to a data transfer rate of up to 3072 bytes per microframe.

The USB Controller supports this by allowing the user to load data packets of up to 3072 bytes (i.e., 3×1024 bytes) into the associated FIFO in a single transaction. From the point of view of the software in the CPU, the operation is then exactly as described above for Single Packet Buffering or Double Packet Buffering (as appropriate) except that `IDX0_InPktRdy` will always need to be set manually (i.e., by the CPU) as `IDX0_AUTOSET` does not operate with high-bandwidth Isochronous transfers.

Any data packet loaded into the FIFO that is larger than the maximum payload is automatically split into 'USB' packets of the maximum payload, or smaller, for transmission over the USB. The number of USB packets transmitted per microframe and the maximum payload in each packet is defined through the `INDX0` register. The `INDX0_MAXPAYLOAD` determine the maximum payload in any USB packet while the `PKTSPLITOPTION` field determines the maximum number of such packets that can be sent in one microframe (2 or 3). Together, these set the maximum size of packet that can be loaded into the FIFO.

At least one USB packet will always be sent: the number of further USB packets sent in the same microframe will depend on the amount of data loaded into the FIFO. The `InPktRdy` bit will be cleared and an interrupt generated only when all the packets have been sent.

Each USB packet is sent in response to an IN token. If, at the end of a microframe, the USB Controller has not received enough IN tokens to send all the USB packets (e.g. because one of the IN tokens received was corrupted), the remainder of the data packet will be flushed from the FIFO. The `IDX0_InPktRdy` bit will then be cleared and the `IncompTx` bit in the `IDX0` register set to indicate that not all of the data loaded into the FIFO was sent.

NOTE

Any second data packet in the FIFO will not be flushed.

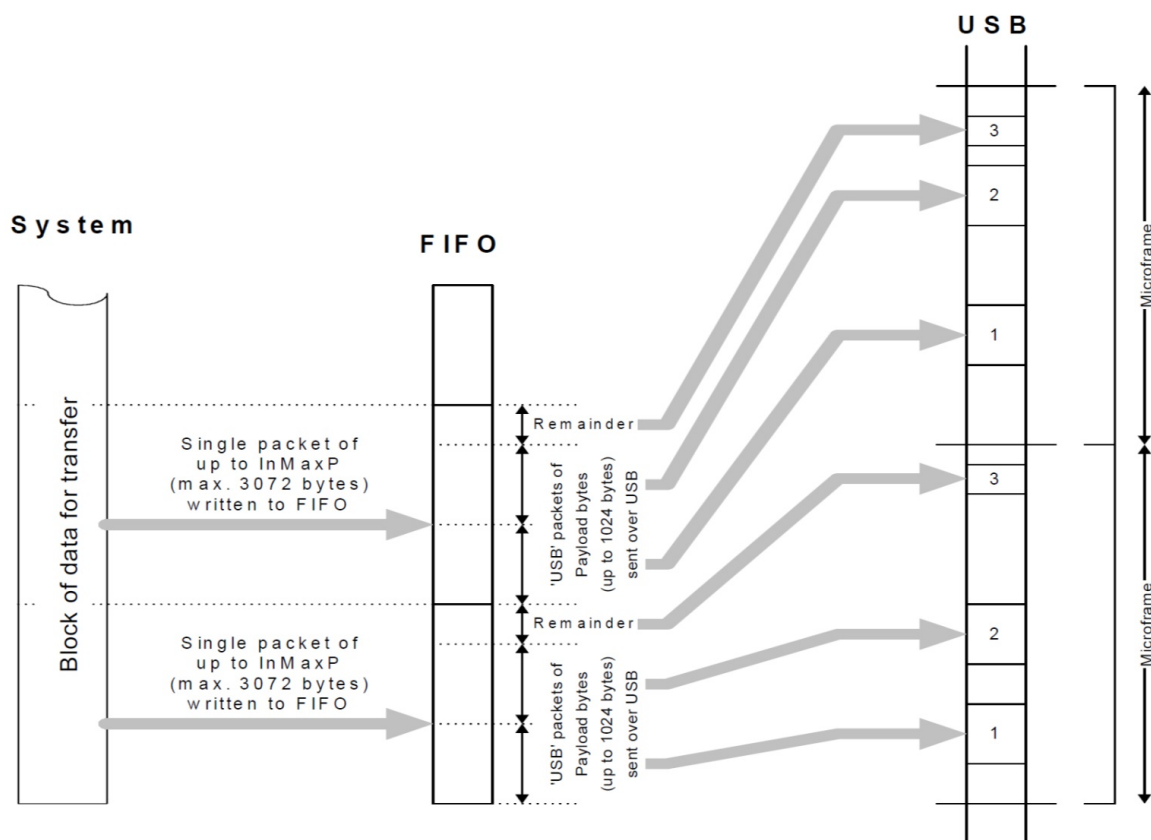


Figure 63. High-speed Isochronous IN Endpoint Transmission

18.13.4 Optional Special Handling

The packets transferred in Bulk operations are defined by the USB Specification to be either 8, 16, 32, 64 or 512 bytes in size, with the 512 byte option only applying to High Speed transfers. For some system designs, however, it may be more convenient for the application software to write larger amounts of data to an endpoint in a single operation than can be transferred in a single USB operation. A particular case in point is where the same endpoint is used for high-speed transfers of 512 bytes under certain circumstances but for full-speed transfers under other circumstances. When operating at full-speed, the maximum amount of data transferred in a single operation is then just 64 bytes.

To cater to such circumstances, the USB Controller includes a configuration option which, if selected, allows larger data packets to be written to Bulk IN endpoints which are then split into packets of an appropriate (specified) size for transfer across the USB bus. (A similar option exists for reading from Bulk OUT endpoints in larger volumes than individual USB packets.)

Under this option, the IDX0 register for the endpoint is increased to 16 bits and the MAXPAYLOAD field of the register defines the payload for each individual transfer, while the PKTSPLITOPTION field defines a multiplier. The application software can then write data packets of size multiplier × payload to the FIFO which the USB Controller will then split into individual packets of the stated payload for transmission over

the USB. From the application software's point of view, the resulting operation will not differ from the transmission of a single USB packet except in the size of the packet written.

NOTE

This feature is only for use with Bulk endpoints and, in accordance with the USB Specification, the payload must be either 8, 16, 32, 64 or 512 bytes with the 512-byte option only applicable for High-Speed transfers. The payload recorded in the `INDX0_MAXPAYLOAD` field must also match the `wMaxPacketSize` field of the Standard Endpoint Descriptor for the endpoint. The associated FIFO must also be large enough to accommodate the data packet prior to being split.

18.14 OUT Endpoint Packet Handling

The sizes of the OUT FIFOs for Endpoints 1 to 5 are determined through the `IDX2_OUTFIFOSZ` field. The maximum amount of data received by an OUT endpoint in any frame or microframe (in High-speed mode) is programmable and is determined by the value written to the `IDX1_MAXPAYLOAD` field for that endpoint. (maximum payload × number of transactions/microframe (where applicable)).

The use of single or double packet buffering is part of the specification for the endpoint FIFO. When double packet buffering is enabled, two data packets can be buffered in the FIFO: when single packet buffering is enabled, only one packet can be buffered even if the packet is less than half the FIFO size.

NOTE

The maximum packet size must not exceed the FIFO size.

18.14.1 Single Packet Buffering

If the size of the OUT endpoint FIFO is less than twice the maximum packet size for this endpoint (as set in the `IDX2_OUTFIFOSZ` field), only one data packet can be buffered in the FIFO and single packet buffering is enabled.

When a packet is received and placed in the OUT FIFO, the `IDX1_OutPktRdy` bit and the `FIFOFULL` bit in `IDX1` are set and the appropriate OUT endpoint is generated (if enabled) to signal that a packet can now be unloaded from the FIFO.

After the packet has been unloaded, the `IDX1_OutPktRdy` field needs to be cleared in order to allow further packets to be received. If the `AUTOCLEAR` bit in `IDX1` is set and a maximum-sized packet is unloaded from the FIFO, the `IDX1_OutPktRdy` field is cleared automatically. The `FIFOFULL` bit is also cleared. For packet sizes less than the maximum, `OutPktRdy` will always have to be cleared manually (i.e., by the CPU).

18.14.2 Double Packet Buffering

NOTE

Double packet buffering is *disabled* if `DPKTBUFDIS` field of the `IDX0` register is set. The default setting for this bit is enabled. However, for double packet buffering to be enabled, bit 4 of the 5-bit `IDX2_OUTFIFOSZ` field must be set.

The following conditions must exist to enable buffering of two data packets (double packet buffering):

- The size of the OUT endpoint FIFO (as set in the `IDX2_OUTFIFOSZ` field) is at least twice the maximum packet size for the endpoint (as set in the `IDX1_MAXPAYLOAD` field).
- Bit 4 of the `IDX2_OUTFIFOSZ` field must be set.
- `DPKTBUFDIS` field of the `IDX1` register is cleared.

When the first packet to be received is loaded into the OUT FIFO, the `IDX1_OutPktRdy` field is set and the appropriate OUT endpoint interrupt is generated (if enabled) to signal that a packet can now be unloaded from the FIFO.

NOTE

The `FIFOFULL` bit in `IDX1` is not set at this point: it is only set if a second packet is received and loaded into the OUT FIFO.

After each packet has been unloaded, `OutPktRdy` needs to be cleared in order to allow further packets to be received. If the `AUTOCLEAR` bit in `IDX1` is set and a maximum-sized packet is unloaded from the FIFO,

the OutPktRdy field will be cleared automatically. For packet sizes less than the maximum, OutPktRdy will always have to be cleared manually (i.e., by the CPU).

If the FIFOFULL bit was set to 1 when OutPktRdy is cleared, the USB Controller will first clear the FIFOFULL bit. It will then set OutPktRdy again to indicate that there is another packet waiting in the FIFO to be unloaded.

18.14.3 High Bandwidth Isochronous Endpoints

In High-speed mode, Isochronous OUT endpoints can receive up to three 'USB' packets in any microframe, with a payload of up to 1024 bytes in each packet, corresponding to a data transfer rate of up to 3072 bytes per microframe.

The USB Controller supports this by automatically combining all the USB packets received during a microframe into a single packet of up to 3072 bytes (i.e., 3×1024 bytes) within the OUT FIFO. From the point of view of the software in the CPU, the operation is then exactly as described above for Single Packet Buffering or Double Packet Buffering (as appropriate) except that OutPktRdy will always need to be cleared manually (i.e., by the CPU) as AUTOCLEAR does not operate with high-bandwidth Isochronous transfers.

The maximum number of USB packets that may be received in any microframe and the maximum payload of these packets are defined through the IDX1 register. The IDX1_MAXPAYLOAD field determines the maximum payload in any USB packet while the IDX1_PKTSPLOPTION determines the maximum number of these packets that may be received in a microframe (2 or 3).

The number of USB packets sent in any microframe will depend on the amount of data to be transferred, and is indicated through the PIDs used for the individual packets. If the indicated number of packets have not been received by the end of a microframe, the IncompRx bit in the IDX1 register will be set to indicate that the data in the FIFO is incomplete. Equally, if a packet of the wrong data type is received, then the PID Error bit in the IDX1 register will be set. In each case an interrupt will, however, still be generated to allow the data that has been received to be read from the FIFO.

NOTE

The circumstances in which a PID Error or IncompRx is reported depends on the precise sequence of packets received. When the core is operating in Peripheral mode, the details are as follows.

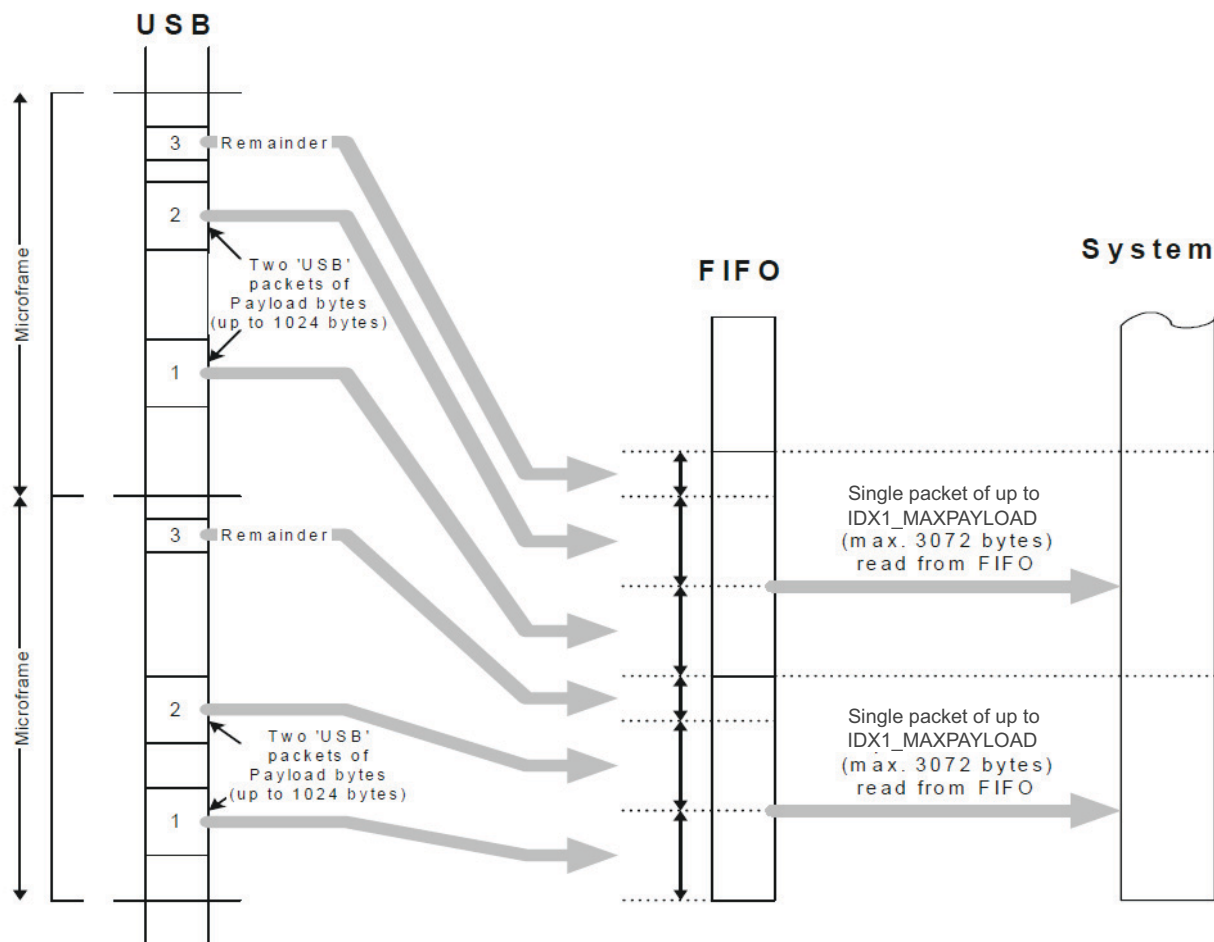


Figure 64. Packets Sent per Microframe

Table 92: PID Errors and IncompRx Responses in Peripheral Mode

Number of Packets Expected	Data Packet(s) Received	Response	Number of Packets Expected	Data Packet(s) Received	Response
1	DATA0 ('D0')	OK	3	D0	OK
	DATA1 ('D1')	PID Error set		D1	IncompRx set
	DATA2 ('D2')	PID Error set		D2	IncompRx set
	MDATA ('DM')	PID Error set		DM	IncompRx set
2	D0	OK		DM D0	PID Error set
	D1	IncompRx set		DM D1	OK
	D2	IncompRx set + PID Error set		DM D2	IncompRx set
	DM	IncompRx set		DM DM	IncompRx set
	DM D0	PID Error set		DM DM D0	PID Error set
	DM D1	OK		DM DM D1	PID Error set
	DM D2	PID Error set		DM DM D2	OK
	DM DM	PID Error set		DM DM DM	PID Error set

18.14.4 Optional Special Handling

The packets transferred in Bulk operations are defined by the USB Specification to be either 8, 16, 32, 64 or 512 bytes in size, with the 512 byte option only applying to High Speed transfers. For some system designs, however, it may be more convenient for the application software to read larger amounts of data from an endpoint in a single operation than can be transferred in a single USB operation. A particular case in point is where the same endpoint is used for high-speed transfers of 512 bytes under certain circumstances but for full-speed transfers under other circumstances. When operating at full-speed, the maximum amount of data transferred in a single operation is then just 64 bytes.

To cater for such circumstances, the USB Controller includes a configuration option which, if selected, causes the USB Controller to combine the packets received across the USB bus into larger data packets prior to being read by the application software. (A similar option exists for writing to Bulk IN endpoints in larger volumes than individual USB packets – see Section 9.4.)

The IDX1 register for the endpoint has the bottom 11 bits of the register defined as the payload, MAXPAYLOAD field, for each individual transfer, while the PKTSPLITOPTION field defines a multiplier. The USB Controller will then combine the appropriate number of the USB packets it receives into a single data packet of size multiplier × payload within the FIFO before asserting OutPktRdy to alert the application software to the presence of a packet to read in the FIFO. From the application software's point of view, the resulting operation will not differ from the receipt of a single USB packet except in the size of the packet read.

This facility is offered as a configuration option rather than as a standard feature because it increases the gate count.

NOTE

This feature is only for use with Bulk endpoints and, in accordance with the USB Specification, the payload must be either 8, 16, 32, 64 or 512 bytes with the 512-byte option only applicable for High-Speed transfers. The payload recorded in the `IDX1_MAXPAYLOAD` field must also match the `wMaxPacketSize` field of the Standard Endpoint Descriptor for the endpoint. The associated FIFO must also be large enough to accommodate the amalgamated data packet.

NOTE

`OutPktRdy` is only set when either the specified number of packets have been received or a “short” USB packet is received (i.e., a packet of less than the specified payload for the endpoint). If a protocol is being used whereby the endpoint receives bulk transfers that are a multiple of the recorded payload size with no short packet to terminate it, the `IDX1_MAXPAYLOAD` field should not be programmed to expect more packets than there are in the transfer (otherwise the software will not be interrupted at the end of the transfer).

18.15 Bulk Transactions

18.15.1 Bulk IN Endpoint

A Bulk IN endpoint is used to transfer non-periodic data from the function controller to the host. The following optional features are available for use with a Bulk IN endpoint:

- Double packet buffering

The use of single or double packet buffering is part of the specification for the endpoint FIFO. When enabled, up to two packets can be stored in the FIFO awaiting transmission to the host.

- AutoSet

When the AutoSet feature is enabled, the `IDX0_InPktRdy` bit will be automatically set when a packet of `MAXPAYLOAD` bytes has been loaded into the FIFO.

- Automatic Packet Splitting

For some system designs, it may be convenient for the application software to write larger amounts of data to an endpoint in a single operation than can be transferred in a single USB operation. A particular case in point is where the same endpoint is used for high-speed transfers of 512 bytes under certain circumstances but for full-speed transfers under other circumstances. When operating at full-speed, the maximum amount of data transferred in a single operation is then just 64 bytes. To support such circumstances, the USB Controller includes a configuration option which, if selected, allows larger data packets to be written to Bulk endpoints which are then split into packets of an appropriate (specified) size for transfer across the USB bus. The necessary packet size information is set via the `IDX0_MAXPAYLOAD` field.

18.15.1.1 Setup

Before using a Bulk IN endpoint the `INDX0_MAXPAYLOAD` field must be written with the maximum packet size (in bytes) for the endpoint. This value should be the same as the `wMaxPacketSize` field of the Standard Endpoint Descriptor for the endpoint. In addition, the relevant interrupt enable bit in the `CFG2` register should be set to 1 (if an interrupt is required for this endpoint), and the relevant fields in the `IDX0` register should be set as shown in Table 93.

Table 93: IDX0 Register Field Settings for Bulk IN Endpoint

Field	Setting	Description
AutoSet	0/1	Set to 1 if the AutoSet feature is required.
ISO	0	Set to 0 to enable Bulk protocol.
Mode	1	Set to 1 to ensure the FIFO is enabled (only necessary if the FIFO is shared with an OUT endpoint).
FrcDataTog	0	Set to 0 to allow normal data toggle operation.
DPktBufDis	0	Set to 0 to enable Double Packet Buffering. Set to 1 to disable double packet buffering.

When a Bulk IN endpoint is first configured, following a `SET_CONFIGURATION` or `SET_INTERFACE` command on Endpoint 0, then `IDX0_ClrDataTog` bit should be set. This will ensure that the data toggle (which is handled automatically by the USB Controller) starts in the correct state. Also if there are any data packets in the FIFO (indicated by the `IDX0_FIFONotEmpty` bit being set), they should be flushed by setting

the `IDX0_FlushFIFO` bit. It may be necessary to set this bit twice in succession if double buffering is enabled.

18.15.1.2 Operation

When data is to be transferred over a Bulk IN pipe, a data packet is loaded into the FIFO and the `IDX0` register written to set the `InPktRdy` bit. When the packet has been sent, the `InPktRdy` bit will be cleared by the USB Controller and an interrupt generated so that the next packet can be loaded into the FIFO. If double packet buffering is enabled (i.e., if the size of the FIFO is at least twice the maximum packet size set in the `IDX0` register), then after the first packet has been loaded and the `InPktRdy` bit set, `InPktRdy` will be immediately cleared by the USB Controller and an interrupt generated so that a second packet can be loaded into the FIFO. This means the software can operate the same way, loading a packet when it receives an interrupt, regardless of whether double packet buffering is enabled or not.

In general, the packet size must not exceed the payload specified in the `IDX0_MAXPAYLOAD` field. This defines the maximum packet size for a single transfer over the USB and, for bulk transfers, is required by the USB Specification to be either 8, 16, 32, 64 (Full-Speed or High-Speed) or 512 bytes (High-Speed only). If more than this amount of data is to be transferred, this needs to be sent as multiple USB packets which should all carry the full payload, except for the last packet which holds the residue.

The exception to this rule applies where the automatic Bulk packet splitting option has been selected when the core was configured. Where this option has been selected, packets up to 32 times `IDX0_MAXPAYLOAD` can be written to the FIFO (assuming that the FIFO is big enough to accept these larger packets) which are then split by the core into packets of the appropriate size for transfer over the USB. The size of the packets written to the FIFO is given by $m \times \text{payload}$ where `IDX0_PKTSPLOPTION` = $m - 1$. All the application software needs to do to take advantage of this feature is to set the appropriate values in the `IDX0_MAXPAYLOAD` field (and ensure that the value written to it matches the value given in the `wMaxPacketSize` field of the Standard Endpoint Descriptor for the associated endpoint). As far as the application software is concerned, the process of transferring these larger packets is no different from that used to transfer a standard-sized Bulk packet.

The host may determine that all the data for a transfer has been sent by knowing the total size of the data block. Alternatively, it may infer that all the data have been sent when it receives a packet which is less than the `MAXPAYLOAD` in size. In the latter case, if the total size of the data block is an exact multiple of the payload, it will be necessary for the function to send a null packet after all the data has been sent. This is done by setting `InPktRdy` when the next interrupt is received, without loading any data into the FIFO.

18.15.1.3 Error Handling

If the software wants to shut down the Bulk IN pipe, it should set the `IDX0_SendStall` bit. When the USB Controller receives the next IN token, it will send a STALL to the host, set the `IDX0_SentStall` bit and generate an interrupt. When the software receives an interrupt with the `SentStall` bit set, it should clear the `SentStall` bit. It should leave the `SendStall` bit set until it is ready to re-enable the Bulk IN pipe.

NOTE

If the host failed to receive the STALL packet for some reason, it will send another IN token, so it is advisable to leave the `SendStall` bit set until the software is ready to re-enable the Bulk IN pipe. When a pipe is re-enabled, the data toggle sequence should be restarted by setting the `IDX0_ClrDataTog` bit.

18.15.2 Bulk OUT Endpoint

A Bulk OUT endpoint is used to transfer non-periodic data from the host to the function controller.

The following optional features are available for use with a Bulk OUT endpoint:

- Double packet buffering

The use of single or double packet buffering is part of the specification for the endpoint FIFO). When enabled, up to two packets can be stored in the FIFO.

- AutoClear

When the AutoClear feature is enabled, the `IDX1_OutPktRdy` bit will be automatically cleared when a packet of `MAXPAYLOAD` bytes has been unloaded from the FIFO.

- Automatic Packet Combining

For some system designs, it may be convenient for the application software to read larger amounts of data from an endpoint in a single operation than can be transferred in a single USB operation. A particular case in point is where the same endpoint is used for high-speed transfers of 512 bytes under certain circumstances but for full-speed transfers under other circumstances. When operating at full-speed, the maximum amount of data transferred in a single operation is then just 64 bytes. To support such circumstances, the USB Controller includes a configuration option which, if selected, causes the USB Controller to combine the packets received across the USB bus into larger data packets prior to being read by the application software. The necessary packet size information is set via the `PKTSPLITOPTION` and `MAXPAYLOAD` fields.

18.15.2.1 Setup

Before using a Bulk OUT endpoint, the `IDX0_MAXPAYLOAD` field must be written with the maximum packet size (in bytes) for the endpoint. This value should be the same as the `wMaxPacketSize` field of the Standard Endpoint Descriptor for the endpoint. In addition, the relevant interrupt enable bit in the `IntrOutE` register should be set to 1 (if an interrupt is required for this endpoint) and `CFG2` register should be set as shown in Table 94. Bit D0 is unused/Read-only.

Table 94: IDX0 Register Field Settings for Bulk OUT Endpoint

Field	Setting	Description
AutoClear	0/1	Set to 1 if the AutoClear feature is required.
ISO	0	Set to 0 to enable Bulk protocol.
DisNye	0	Set to 0 to allow normal PING flow control.
DPktBufDis	0	Set to 0 to enable double packet buffering. Set to 1 to disable double packet buffering.

When a Bulk OUT endpoint is first configured, following a `SET_CONFIGURATION` or `SET_INTERFACE` command on Endpoint 0, the `IDX1` register should be written to set the `ClrDataTog` bit. This will ensure that the data toggle (which is handled automatically by the USB Controller) starts in the correct state. Also if there are any data packets in the FIFO (indicated by the `OutPktRdy` bit being set), they should be flushed by setting the `FlushFIFO` bit. It may be necessary to set this bit twice in succession if double buffering is enabled.

18.15.2.2 Operation

When a data packet is received by a Bulk OUT endpoint, the `IDX1_OutPktRdy` bit is set and an interrupt is generated. The software should read the `IDX2_ENDPTOUTCOUNT` field for the endpoint to determine the size of the data packet. The data packet should be read from the FIFO, then the `OutPktRdy` bit should be cleared.

The packets received should not exceed the size specified in the `IDX1_MAXPAYLOAD` field (because this should match the value set in the `wMaxPacketSize` field of the endpoint descriptor sent to the host). When a block of data larger than `wMaxPacketSize` needs to be sent to the function, it will be sent as multiple packets. All the packets will be `wMaxPacketSize` in size, except the last packet which will contain the remainder. The software may use an application specific method of determining the total size of the block and hence when the last packet has been received. Alternatively, it may infer that the entire block has been received when it receives a packet which is less than `wMaxPacketSize` in size. (If the total size of the data block is a multiple of `wMaxPacketSize`, a null data packet will be sent after the data to signify that the transfer is complete.)

1. In general, the application software will need to read each packet from the FIFO individually. The exception to this rule applies where the option for automatic combining of Bulk packets has been selected when the core was configured. Where this option has been selected, the core can receive up to 32 packets at a time and combine them into a single packet within the FIFO (assuming that the FIFO is big enough to accept these larger packets). The size of the packets written to the FIFO is given by $m \times wMaxPacketSize$ where `IDX1_PKTSPLOPTION = m - 1`. All the application software needs to do to take advantage of this feature is set the appropriate values in the `IDX1_MAXPAYLOAD` field (and ensure that the value written to it matches the value given in the `wMaxPacketSize` field of the endpoint descriptor). As far as the application software is concerned, the process of transferring these larger packets is no different from that used to transfer a standard-sized Bulk packet.

18.15.2.3 Error Handling

If the software wants to shut down the Bulk OUT pipe, it should set the `SendStall` bit. When the USB Controller receives the next packet it will send a STALL to the host, set the `SentStall` bit and generate an interrupt.

When the software receives an interrupt with the `SentStall` bit set, it should clear the `SentStall` bit. It should leave the `SendStall` bit set until it is ready to re-enable the Bulk OUT pipe.

NOTE

If the host failed to receive the STALL packet for some reason, it will send another packet, so it is advisable to leave the `SendStall` bit set until the software is ready to re-enable the Bulk OUT pipe. When a Bulk OUT pipe is re-enabled, the data toggle sequence should be restarted by setting the `IDX1_ClrDataTog` bit.

18.15.3 Interrupt Transactions

18.15.3.1 Interrupt IN Endpoint

An Interrupt IN endpoint is used to transfer periodic data from the function controller to the host.

An Interrupt IN endpoint uses the same protocol as a Bulk IN endpoint and can be used the same way. Interrupt IN endpoints also support one feature that Bulk IN endpoints do not, in that they support continuous toggle of the data toggle bit. This feature is enabled by setting the `FrcDataTog` bit in the `IDX0` register. When this bit is set to 1, the USB Controller will consider the packet as having been successfully sent and toggle the data bit for the endpoint, regardless of whether an ACK was received from the host.

18.15.3.2 Interrupt OUT Endpoint

An Interrupt OUT endpoint is used to transfer periodic data from the host to a function controller.

An Interrupt OUT endpoint uses almost the same protocol as a Bulk OUT endpoint and can be used the same way. The one difference is that Interrupt endpoints do not support PING flow control. This means that the USB Controller should never respond with a NYET handshake, only ACK/NAK/STALL. To ensure this,

the DisNye bit in the IDX1 register should be set to 1 to disable the transmission of NYET handshakes in High-speed mode.

18.16 Isochronous Transactions

18.16.1 Isochronous IN Endpoint

An Isochronous IN endpoint is used to transfer periodic data from the function controller to the host. This section describes the use of full-speed Isochronous IN endpoints and low bandwidth (1 packet per microframe) high-speed Isochronous IN endpoints. High bandwidth high-speed (> 8 Mbps) endpoints are described in a later section.

The following optional features are available for use with an Isochronous IN endpoint:

- Double packet buffering

The use of single or double packet buffering is part of the specification for the endpoint FIFO.) When enabled, up to two packets can be stored in the FIFO awaiting transmission to the host. Double packet buffering is generally advisable for Isochronous IN endpoints in order to avoid data underrun (see 'Operation' below).

- AutoSet

When the AutoSet feature is enabled, the InPktRdy bit will be automatically set when a packet of MAXPAYLOAD bytes has been loaded into the FIFO. However, this feature is not particularly useful with Isochronous endpoints because the packets transferred often are not maximum packet size and the InCSRL register needs to be accessed following every packet to check for Underrun errors.

18.16.1.1 Setup

Before using an Isochronous IN endpoint, the MAXPAYLOAD field must be written with the maximum packet size (in bytes) for the endpoint. This value should be the same as the wMaxPacketSize field of the Standard Endpoint Descriptor for the endpoint. In addition, the relevant interrupt enable bit in the CFG2 register should be set to 1 (if an interrupt is required for this endpoint) and the high byte of the IDX0 register should be set as shown in Table 95. (Bit D0 is unused):

Table 95: IDX0 Register Field Settings for Isochronous IN Endpoint

Field	Setting	Description
AutoSet	0/1	Set to 1 if the AutoSet feature is required.
ISO	1	Set to 1 to enable Isochronous protocol.
Mode	1	Set to 1 to ensure the FIFO is enabled (only necessary if the FIFO is shared with an OUT endpoint).
FrcDataTog	0	Ignored in Isochronous mode.
DPktBufDis	0	Set to 0 to enable double packet buffering. Set to 1 to disable double packet buffering.

18.16.1.2 Operation

An Isochronous endpoint does not support data retries, so if data underrun is to be avoided, the data to be sent to the host must be loaded into the FIFO before the IN token is received. The host will send one IN token per frame (or microframe in Highspeed mode), however the timing within the frame (or microframe) can vary. If an IN token is received near the end of one frame and then at the start of the next frame, there will be little time to reload the FIFO. For this reason, double buffering is usually required for an Isochronous IN endpoint.

The AutoSet feature can be used with an Isochronous IN endpoint, in the same way as for a Bulk IN endpoint. However, unless the data arrives from the source at an absolutely consistent rate, synchronized to the host's frame clock, the size of the packets sent to the host will have to increase or decrease from frame to frame (or from microframe to microframe) to match the source data rate. This means that the actual packet sizes will not always be MAXPAYLOAD in size, rendering the AutoSet feature useless.

An interrupt is generated whenever a packet is sent to the host and the software may use this interrupt to load the next packet into the FIFO and set the InPktRdy bit in the IDX0 register in the same way as for a Bulk IN endpoint. As the interrupt could occur almost any time within a frame(/microframe), depending on when the host has scheduled the transaction, this may result in irregular timing of FIFO load requests. If the data source for the endpoint is coming from some external hardware, it may be more convenient to wait until the end of each frame (or microframe) before loading the FIFO as this will minimize the requirement for additional buffering. This can be done by using the SOF interrupt to trigger the loading of the next data packet. The CFG2_SOF bit is set once per frame(/microframe) when a SOF packet is received. The interrupts may still be used to set the InPktRdy bit in IDX0 and to check for data overruns/underruns (see 'Error Handling' below).

Starting up a double-buffered Isochronous IN pipe can be a source of problems. Double buffering requires that a data packet is not transmitted until the frame(/microframe) after it is loaded. There is no problem if the function loads the first data packet at least one frame(/microframe) before the host sets up the pipe (and therefore starts sending IN tokens). But if the host has already started sending IN tokens by the time the first packet is loaded, the packet may be transmitted in the same frame(/microframe) as it is loaded, depending on whether it is loaded before, or after, the IN token is received. This potential problem can be avoided by setting the ISO Update bit in the CFG2 register. When this bit is set to 1, any data packet loaded into an Isochronous IN endpoint FIFO will not be transmitted until after the next SOF packet has been received, thereby ensuring that the data packet is not sent too early.

18.16.1.3 Error Handling

If the endpoint has no data in its FIFO when an IN token is received, it will send a null data packet to the host and set the UnderRun bit in the IDX0 register. This is an indication that the software is not supplying data fast enough for the host. It is up to the application to determine how this error condition is handled.

If the software is loading one packet per frame(/microframe) and it finds that the InPktRdy bit in the IDX0 register is set when it wants to load the next packet, this indicates that a data packet has not been sent (perhaps because an IN token from the host was corrupted). It is up to the application how it handles this condition: it may choose to flush the unsent packet by setting the FlushFIFO bit in the IDX0 register, or it may choose to skip the current packet.

18.16.2 Isochronous OUT Endpoint

An Isochronous OUT endpoint is used to transfer periodic data from the host to the function controller. This section describes the use of Full-speed Isochronous OUT endpoints and low bandwidth (1 packet per microframe) High-speed Isochronous OUT endpoints. High bandwidth High-speed (> 8 Mbps) endpoints are described in a later section.

The following optional features are available for use with an Isochronous OUT endpoint:

- Double packet buffering

The use of single or double packet buffering is part of the specification for the endpoint FIFO.) When enabled, up to two packets can be stored in the FIFO awaiting transmission to the host.

NOTE

Double packet buffering is generally advisable for Isochronous OUT endpoints in order to avoid data overrun (see 'Operation' below).

- AutoClear

When the AutoClear feature is enabled, the IDX1_OutPktRdy bit will be automatically cleared when a packet of MAXPAYLOAD bytes has been unloaded from the FIFO. However, this feature is not particularly useful with Isochronous endpoints because the packets transferred often are not maximum packet size and the IDX1 register needs to be accessed following every packet to check for Overrun or CRC errors.

18.16.2.1 Setup

Before using an Isochronous OUT endpoint, the MAXPAYLOAD field must be written with the maximum packet size (in bytes) for the endpoint. This value should be the same as the wMaxPacketSize field of the Standard Endpoint Descriptor for the endpoint.

In addition, the relevant interrupt enable bit in the CFG2 register should be set to 1 (if an interrupt is required for this endpoint) and IDX1 register should be set as shown in Table 96.

Table 96: IDX0 Register Field Settings for Isochronous OUT Endpoint

Field	Setting	Description
AutoClear	0/1	Set to 1 if the AutoClear feature is required.
ISO	0	Set to 1 to enable Isochronous protocol.
DisNye	0	Ignored in Isochronous mode.
DPktBufDis	0	Set to 0 to enable double packet buffering. Set to 1 to disable double packet buffering.

18.16.2.2 Operation

An Isochronous endpoint does not support data retries so if a data overrun is to be avoided, there must be space in the FIFO to accept a packet when it is received. The host will send one packet per frame (or microframe in High-speed mode), however the time within the frame can vary. If a packet is received near the end of one frame(/microframe) and another arrives at the start of the next frame, there will be little time to unload the FIFO. For this reason, double buffering is usually required for an Isochronous OUT endpoint.

The AutoClear feature can be used with an Isochronous OUT endpoint, in the same way as for a Bulk OUT endpoint. However, unless the data sink receives data at an absolutely consistent rate and is synchronized to the host's frame clock, the size of the packets sent from the host will have to increase or decrease from frame to frame (or from microframe to microframe) to match the required data rate. This means that the actual packet sizes will not always be MAXPAYLOAD in size, rendering the AutoClear feature useless.

An interrupt is generated whenever a packet is received from the host and the software may use this interrupt to unload the packet from the FIFO and clear the OutPktRdy bit in the IDX1 register in the same way as for a Bulk OUT endpoint.

As the interrupt could occur almost any time within a frame(/microframe), depending on when the host has scheduled the transaction, the timing of FIFO unload requests will probably be irregular. If the data sink for the endpoint is going to some external hardware, it may be better to minimize the requirement for additional buffering by waiting until the end of each frame(/microframe) before unloading the FIFO. This can be done by using the SOF interrupt to trigger the unloading of the data packet. Other interrupts may still be used to clear the OutPktRdy bit in the IDX1 register and to check for data overruns/underruns (see 'Error Handling' below).

18.16.2.3 Error Handling

If there is no space in the FIFO to store a packet when it is received from the host, the OverRun bit in the IDX1 register will be set. This is an indication that the software is not unloading data fast enough for the host. It is up to the application to determine how this error condition is handled.

If the USB Controller finds that a received packet has a CRC error, it will still store the packet in the FIFO and set the IDX1_OutPktRdy bit and the IDX1_DataError bit. It is left up to the application how this error condition is handled.

18.16.2.4 High-speed High-bandwidth IN Endpoint

Full-speed Isochronous endpoints and low bandwidth High-speed Isochronous endpoints transfer a single packet per frame (/microframe). In High-speed mode, this allows a maximum data transfer rate of 8 MB/sec (64 Mbps), as the maximum payload for any packet on the USB is 1024 bytes. However, if the endpoint is defined to be high bandwidth, it can support data transfer rates up to 24 MB/sec (192 Mbps) by performing up to three such transactions per microframe.

High bandwidth Isochronous endpoints use PID sequencing to ensure that the host and function know how many packets are being transferred and to detect lost packets. The USB Controller handles the PID sequencing and the splitting of data into 'USB' packets for transmission over the USB automatically. To the software, it should seem as though a single packet of up to 3072 bytes (3×1024 bytes) is being transferred.

When setting up a high bandwidth Isochronous IN endpoint, the IDX0_MAXPAYLOAD field should be written with the same value that was used in the wMaxPacketSize field of the endpoint descriptor. The lower 11 bits specify the maximum payload of a single transaction over the USB, while bits 11 and 12 specify the maximum number of such transactions per microframe.

The data to be sent in a microframe is loaded into the FIFO as a single packet and the IDX0_InPktRdy bit is set in the same way as for a low bandwidth Isochronous IN endpoint. If the amount of data loaded exceeds the maximum data payload specified in the lower 11 bits of MAXPAYLOAD, the USB Controller will automatically split the data into two or three 'USB' packets for transmission.

The only difference the software should see between a high bandwidth and a low bandwidth Isochronous IN endpoint is an additional error condition that can occur with a high bandwidth endpoint. If the USB Controller has split the data to be sent into two or three 'USB' packets but has not received sufficient IN tokens from the host to send all the packets, the remainder of the data packet will be flushed from the FIFO at the end of the microframe and the InPktRdy bit will be cleared. In addition, the IDX0_IncompTx bit will be set to indicate to the software that not all the data was sent. Any second data packet in the FIFO will not be flushed.

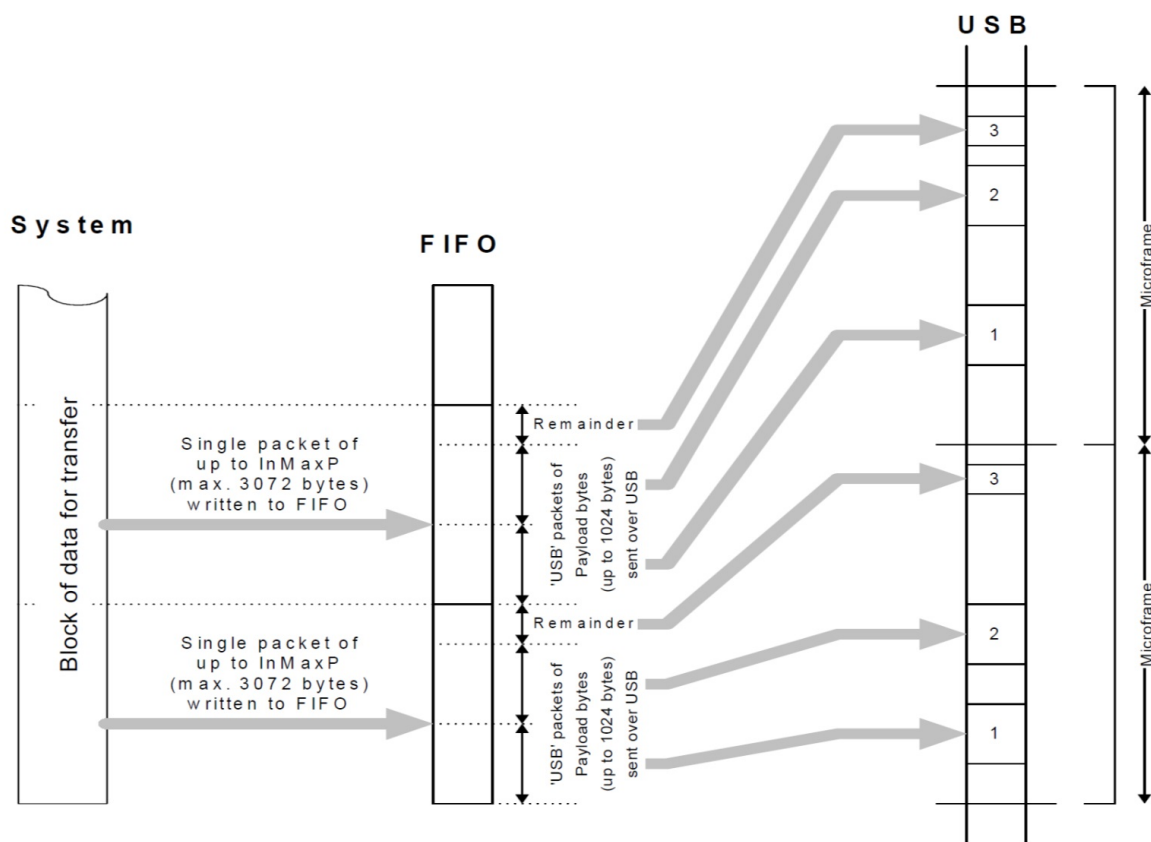


Figure 65. High-Speed High-Bandwidth IN Endpoint

18.16.3 High-speed High-bandwidth OUT Endpoint

Full-speed Isochronous endpoints and low bandwidth High-speed Isochronous endpoints transfer a single packet per frame (/microframe). In High-speed mode, this allows a maximum data transfer rate of 8 MB/sec (64 Mbps), as the maximum payload for any packet on the USB is 1024 bytes. However, if the endpoint is defined to be high bandwidth, it can support data transfer rates up to 24 MB/sec (192 Mbps) by performing up to three such transactions per microframe.

High bandwidth Isochronous endpoints use PID sequencing to ensure that the host and function know how many packets are being transferred and to detect lost packets. The USB Controller handles the PID sequencing and the combining of data from two or three USB packets automatically. To the software, it should seem as though a single packet, of up to 3072 bytes (3×1024 bytes), has been received.

When setting up a high bandwidth Isochronous OUT endpoint, the IDX1 register should be written with the same value that was used in the wMaxPacketSize field of the endpoint descriptor. The lower 11 bits specify the maximum payload of a single transaction over the USB, while bits 11 and 12 specify the maximum number of such transactions per microframe.

The USB Controller will combine the data from each of the packets received during the microframe and place the combined data packet into the FIFO. The IDX1_OutPktRdy bit will be set when all the data packets for that microframe have been received. The software can then unload the data from the FIFO in the same way as for a low bandwidth Isochronous OUT endpoint.

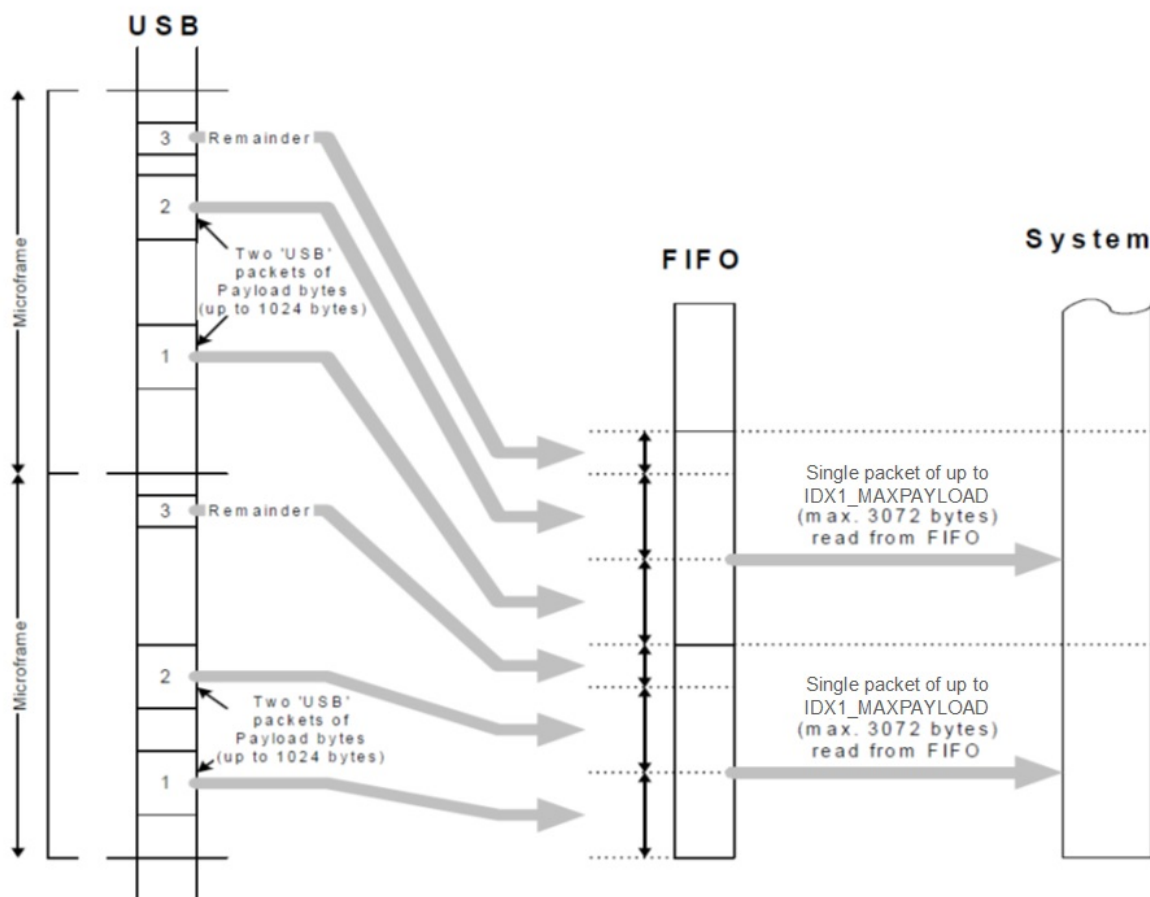


Figure 66. High-speed High-bandwidth OUT Endpoint

The number of USB packets sent in any microframe depends on the amount of data to be transferred, and is indicated through the PIDs used for the individual packets. If the indicated number of packets have not been received by the end of a microframe, the `IDX1_IncompRx` bit gets set to indicate that the data in the FIFO is incomplete. Equally, if a packet of the wrong data type is received, then the `IDX1_DisNye` bit gets set. However, in each case an interrupt is still generated to allow the data that has been received to be read from the FIFO.

The circumstances in which a PID Error (indicated in `IDX1_DisNye` field) or `IncompRx` is reported depends on the precise sequence of packets received. When the core is operating in Peripheral mode, the details are as follows.

Table 97: PID Error or IncompRx Reporting in Peripheral Mode

No. pkts expected	Data pkt(s) received	Response	No. pkts expected	Data pkt(s) received	Response
1	DATA0 ('D0')	OK	3	D0	OK
	DATA1 ('D1')	PID Error set		D1	IncompRx set
	DATA2 ('D2')	PID Error set		D2	IncompRx set
	MDATA ('DM')	PID Error set		DM	IncompRx set
2	D0	OK		DM D0	PID Error set
	D1	IncompRx set		DM D1	OK
	D2	IncompRx set + PID Error set		DM D2	IncompRx set
	DM	IncompRx set		DM DM	IncompRx set
	DM D0	PID Error set		DM DM D0	PID Error set
	DM D1	OK		DM DM D1	PID Error set
	DM D2	PID Error set		DM DM D2	OK
	DM DM	PID Error set		DM DM DM	PID Error set

18.17 Transaction Flows

In the following transaction flows, host actions are shown against a white background, and the USB Controller actions are shown shaded.

18.17.1 Control Transactions

18.17.1.1 Setup Phase

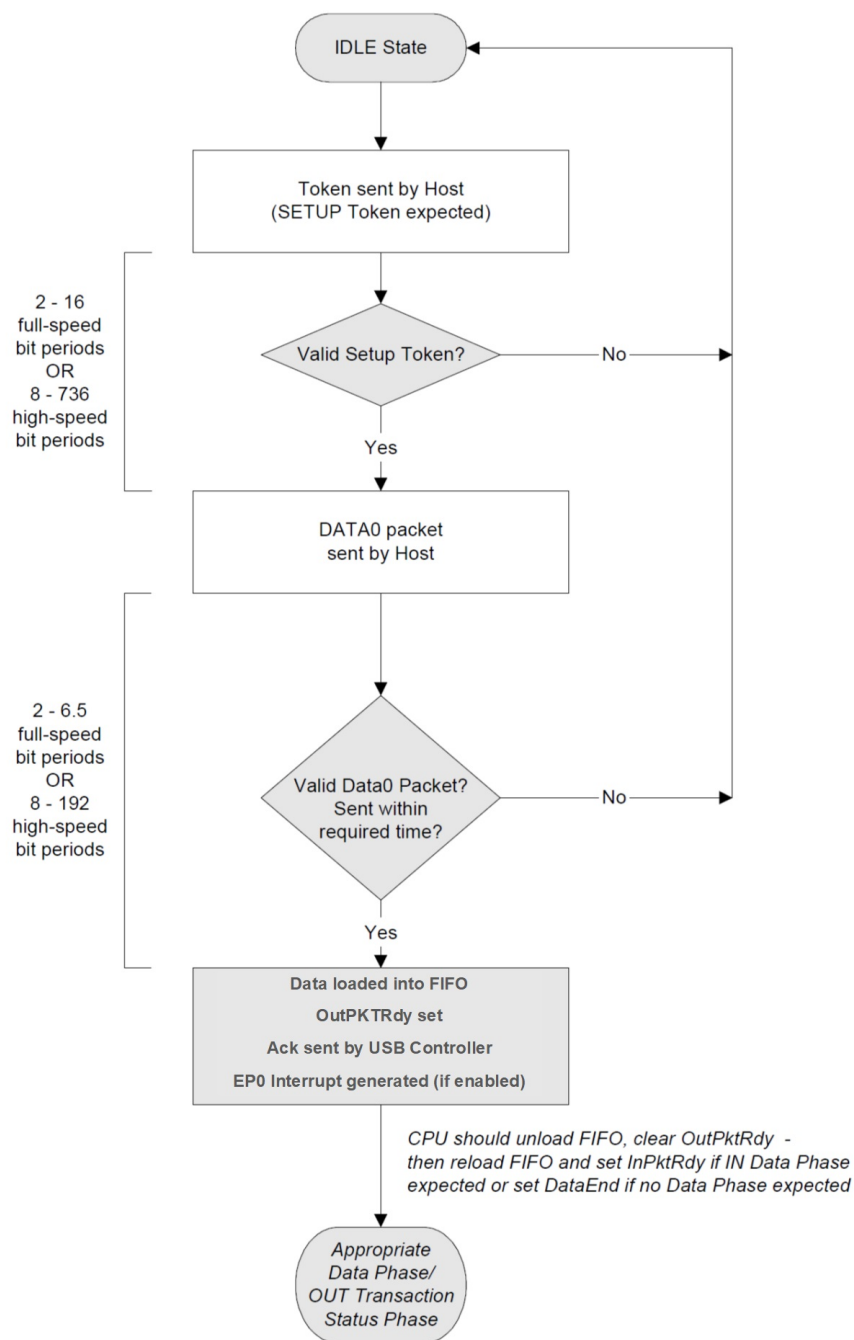


Figure 67. Setup Phase Transaction

18.17.1.2 IN Data Phase

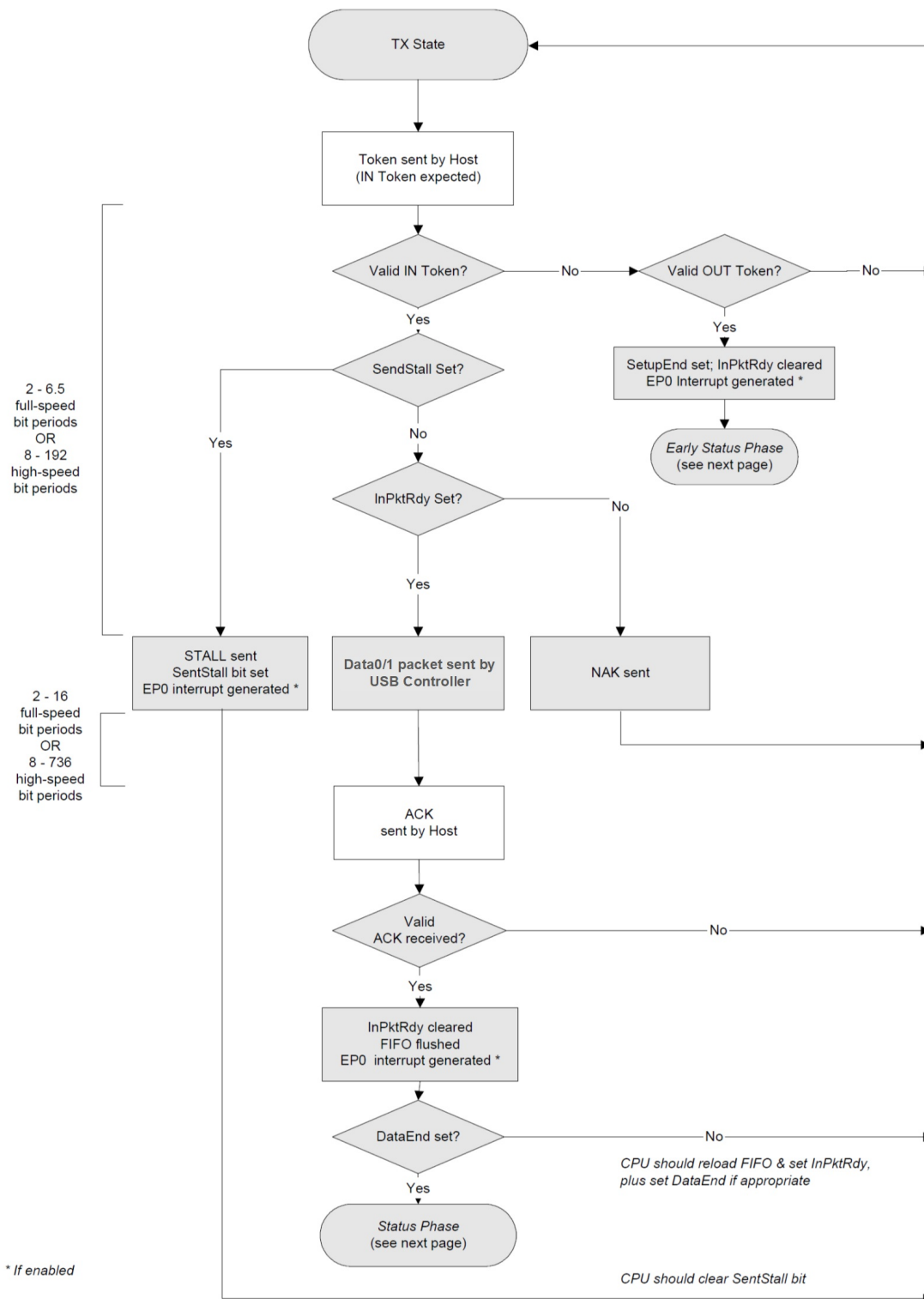


Figure 68. IN Data Phase Transaction

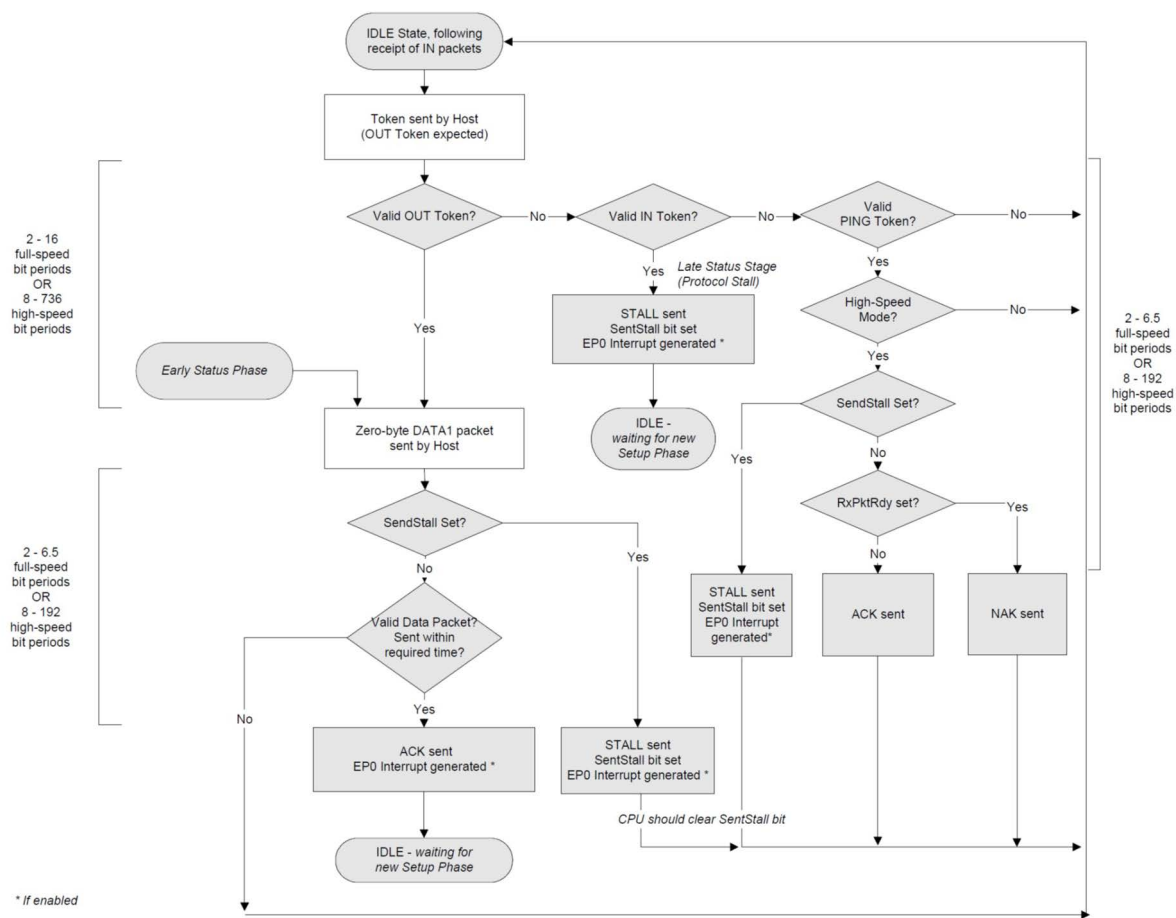


Figure 69. Status Phase Following IN Data Phase Transaction

18.17.1.3 OUT Data Phase

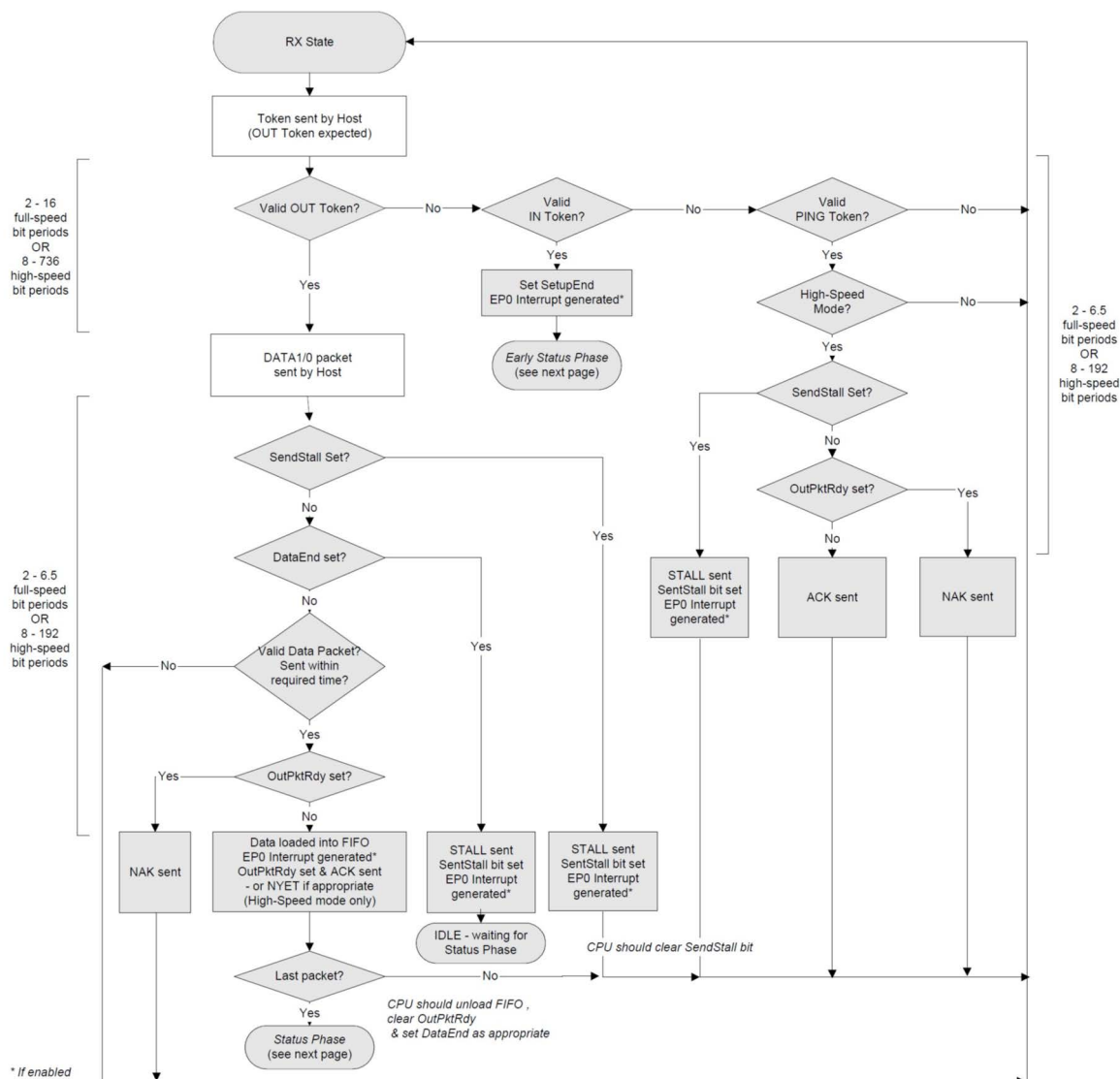


Figure 70. OUT Data Phase Transaction

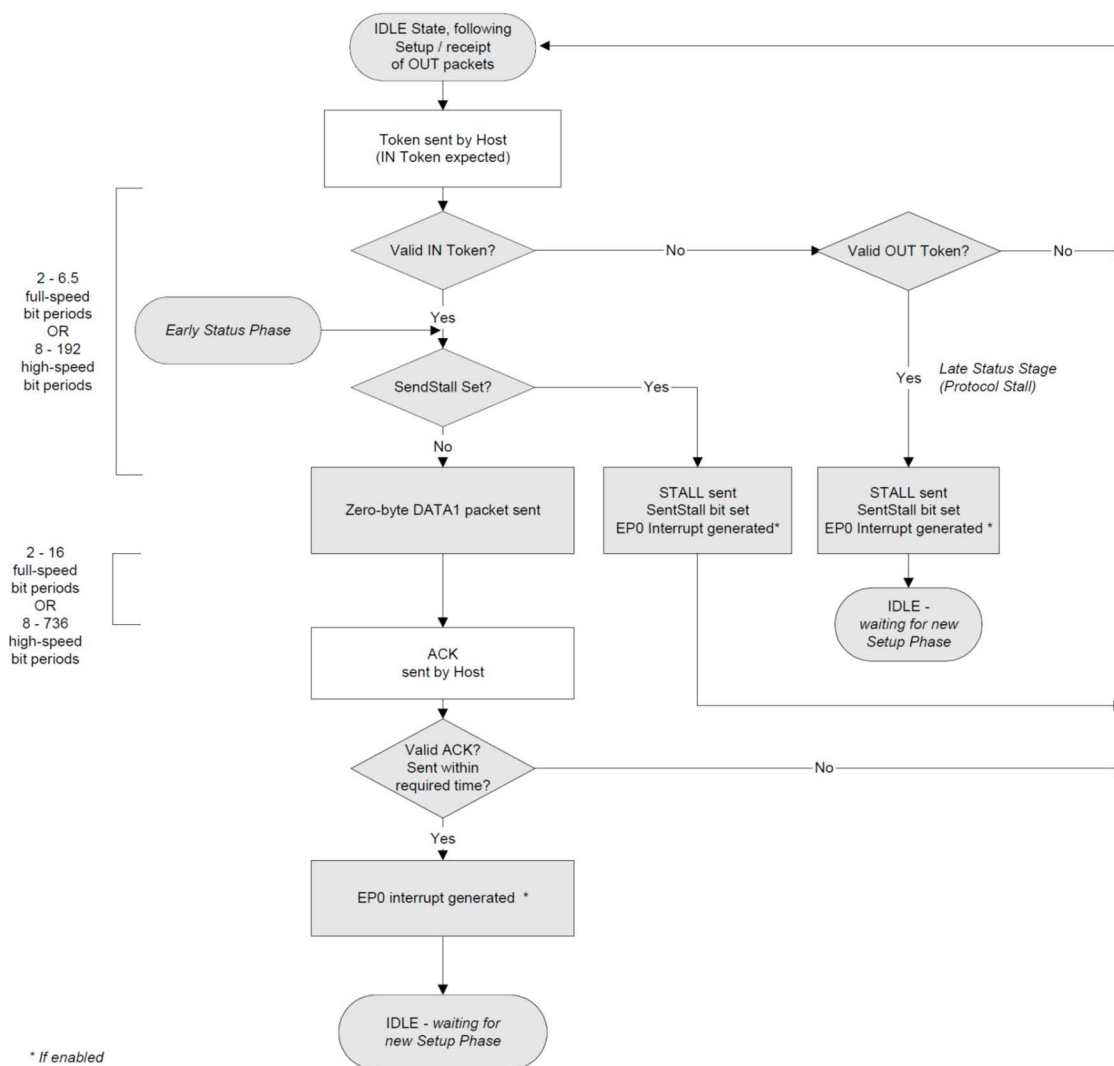


Figure 71. Status Phase Following OUT Data Phase Transaction

18.17.2 Bulk/Interrupt Transactions

18.17.2.1 IN Transactions

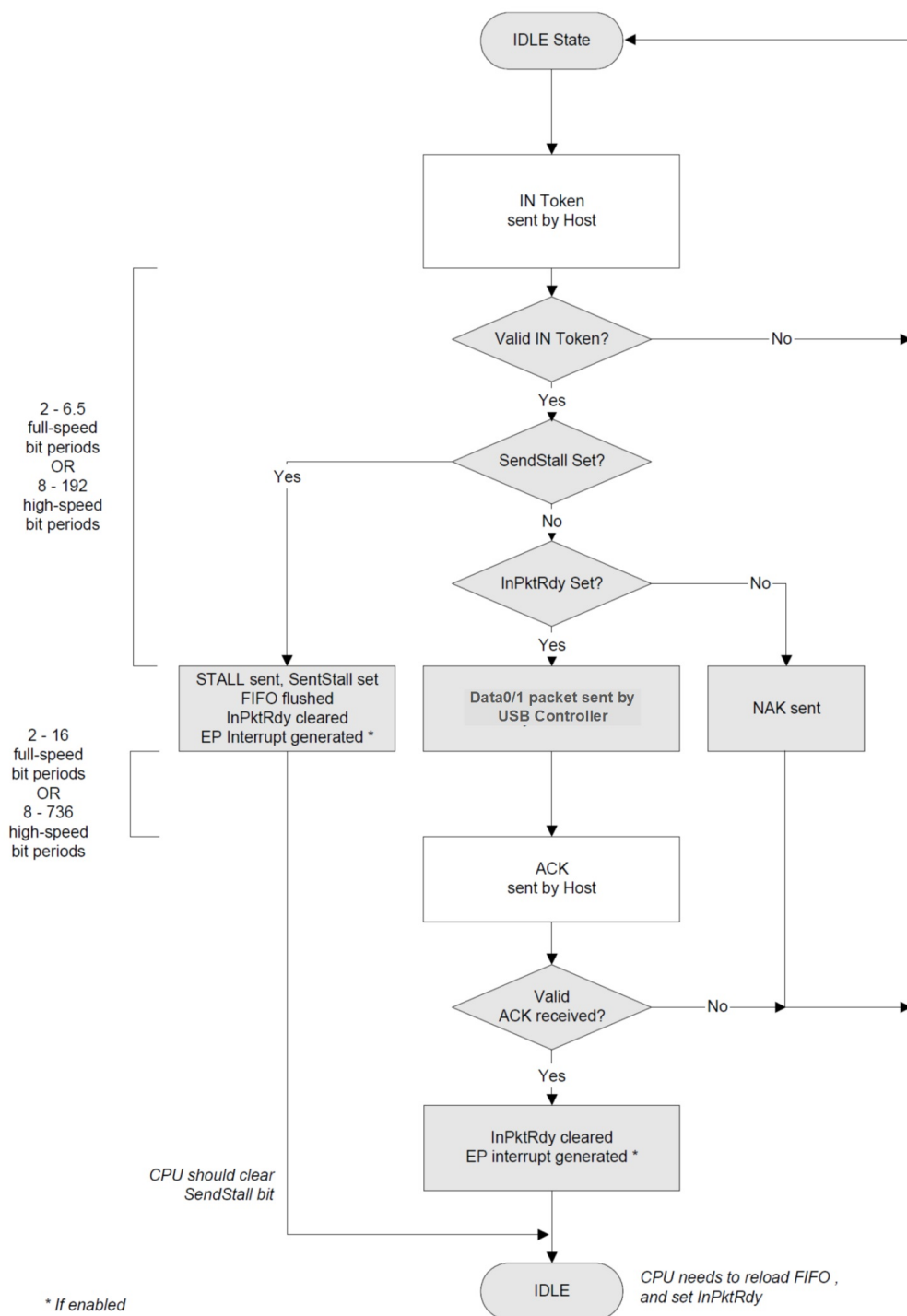


Figure 72. Bulk/Interrupt IN Transaction

18.17.2.2 OUT Transaction

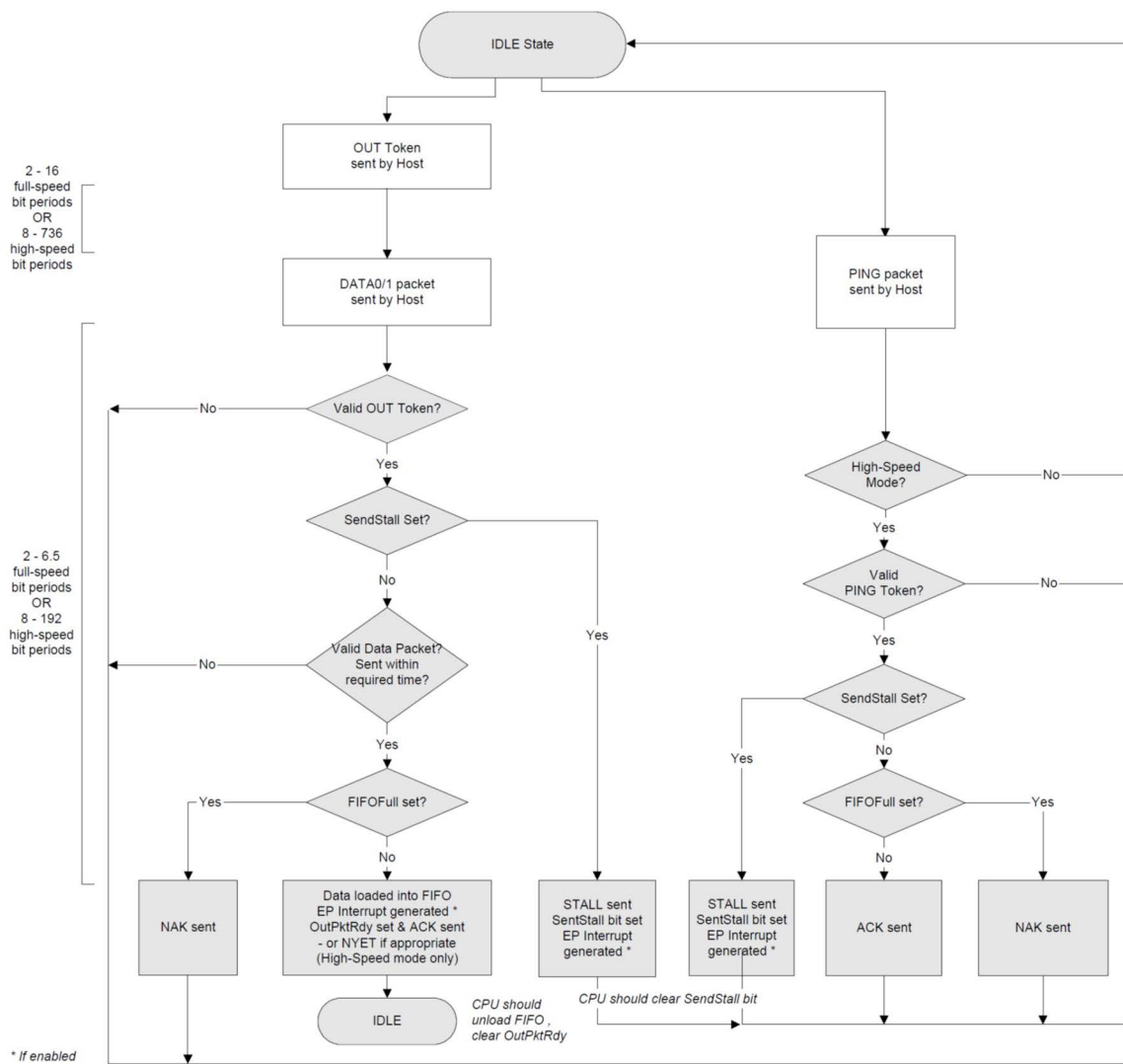
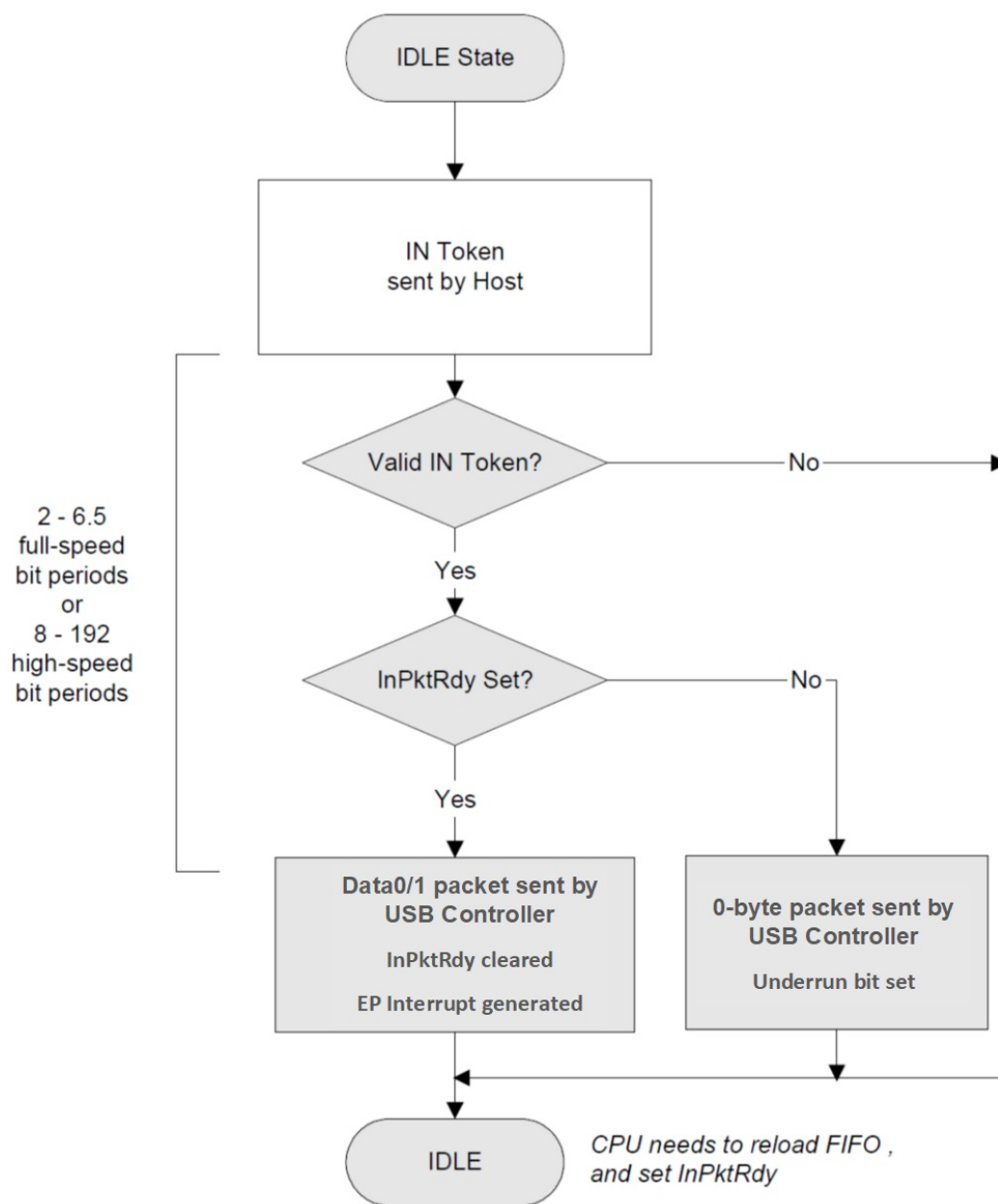


Figure 73. Bulk OUT Transaction

18.17.3 Full-speed Low-bandwidth Isochronous Transaction

18.17.3.1 IN Transaction



* If enabled

Figure 74. Full-speed Isochronous IN Transaction

18.17.3.2 OUT Transaction

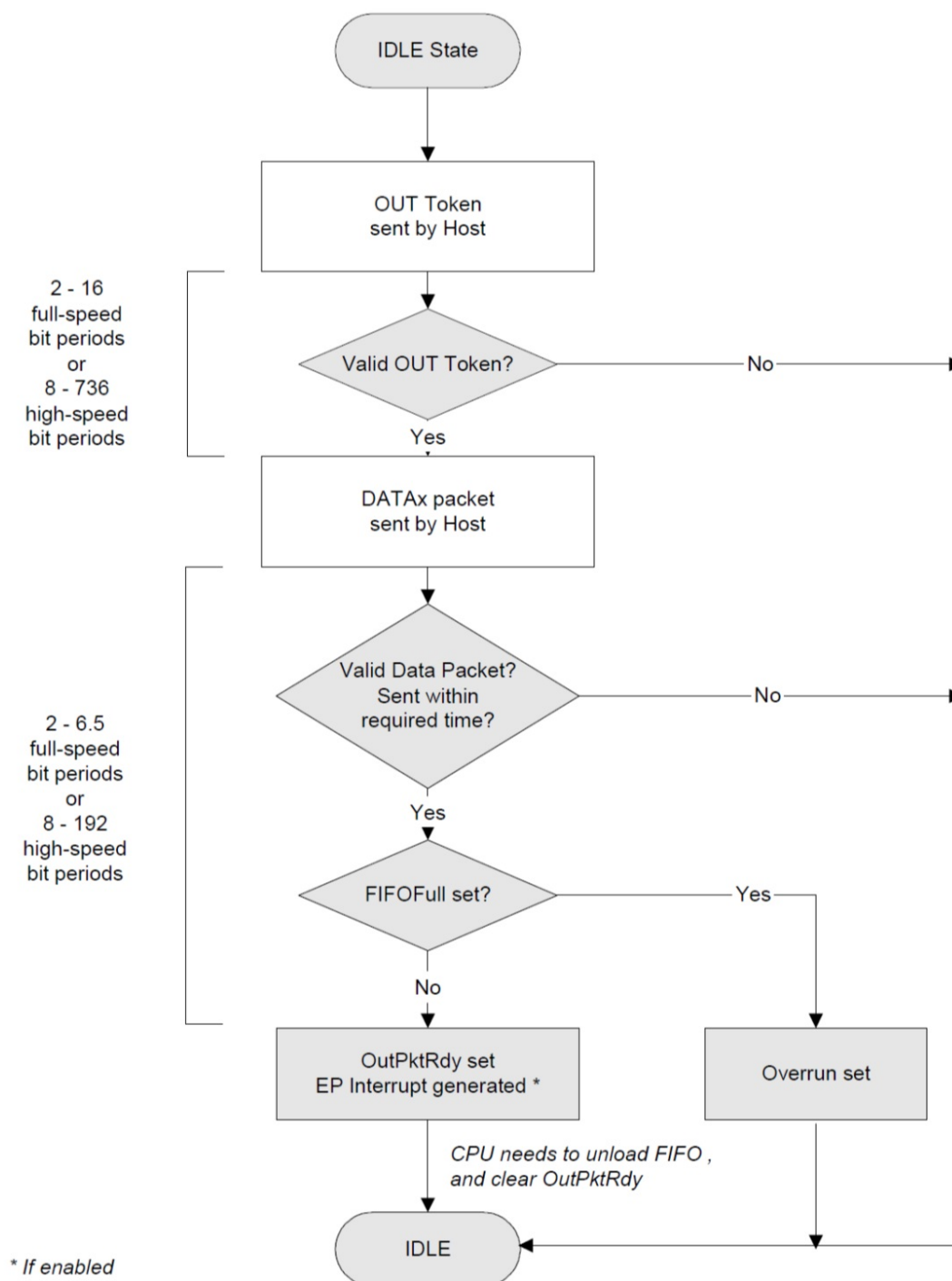


Figure 75. Full-speed Isochronous OUT Transaction

18.17.4 High-bandwidth Isochronous Transactions

18.17.4.1 IN Transaction

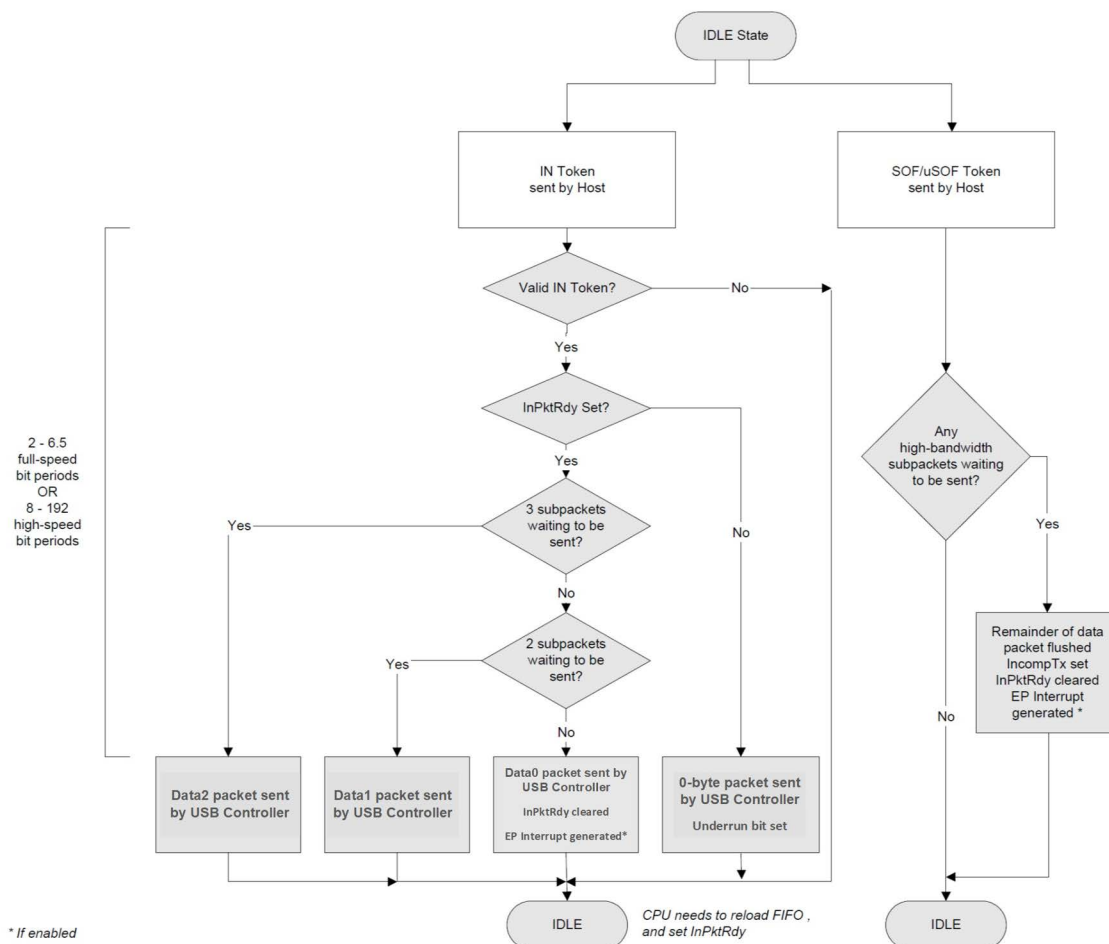


Figure 76. High-bandwidth Isochronous IN Transaction

18.17.4.2 OUT Transaction

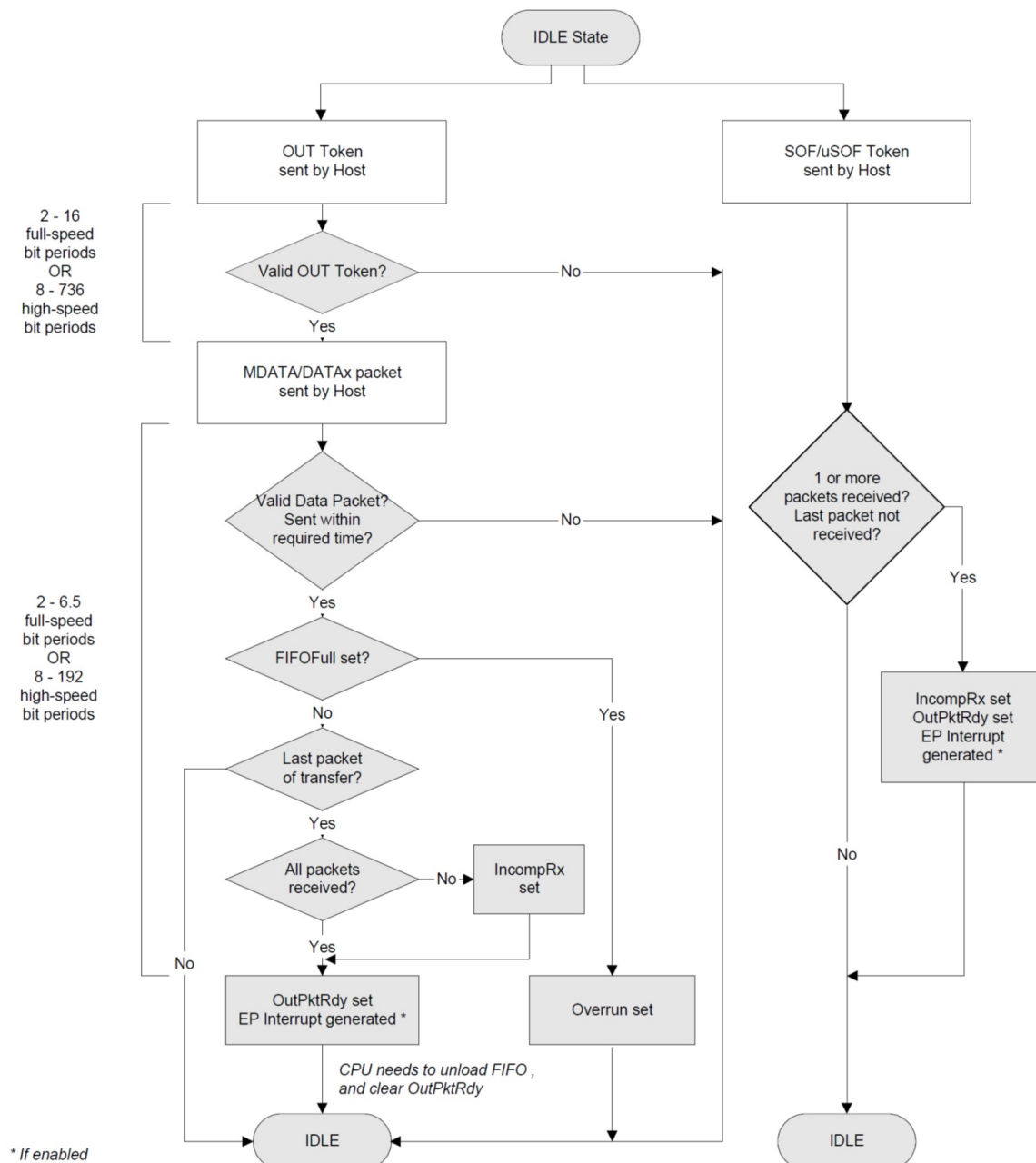


Figure 77. High-bandwidth Isochronous OUT Transaction

18.18 Test Modes

The USB Controller supports the four USB 2.0 test modes defined for High-speed functions. The test modes are entered by writing to the TestMode register (address 0Fh). A test mode is usually requested by the host sending a SET_FEATURE request to Endpoint 0. When the software receives the request, it should wait until the Endpoint 0 transfer has completed (when it receives the Endpoint 0 interrupt indicating that the status phase has completed) then write to the TestMode register.

NOTE

These test modes have no purpose in normal operation.

18.18.1 Test Mode Test_SE0_NAK

To enter the Test_SE0_NAK test mode, the software should set the Test_SE0_NAK bit by writing 6'h01 to the TestMode register. The USB Controller will then go into a mode in which it responds to any valid IN token with a NAK.

18.18.2 Test Mode Test_Test_J

To enter the Test_J test mode, the software should set the Test_J bit by writing 6'h02 to the TestMode register. The USB Controller will then go into a mode in which it transmits a continuous J on the bus.

18.18.3 Test Mode Test_K

To enter the Test_K test mode, the software should set the Test_K bit by writing 6'h04 to the TestMode register. The USB Controller will then go into a mode in which it transmits a continuous K on the bus.

18.18.4 Test Mode Test_Packet

To execute the Test_Packet test, the software should first write the standard test packet (shown below) to the Endpoint 0 FIFO and set the InPktRdy bit in the CSR0 register (D1). It should then write 6'h08 to the TestMode register to enter Test_Packet test mode.

The 53 byte test packet to load is as follows (all bytes in hex). The test packet only has to be loaded once, the USB Controller will keep re-sending the test packet without any further intervention from the software.

```
00 00 00 00 00 00 00 00
00 AA AA AA AA AA AA AA
AA EE EE EE EE EE EE EE
EE FE FF FF FF FF FF FF
FF FF FF FF FF 7F BF DF
EF F7 FB FD FC 7E BF DF
EF F7 FB FD 7E
```

This data sequence is defined in Universal Serial Bus Specification Revision 2.0, Section 7.1.20. The USB Controller will add the DATA0 PID to the head of the data sequence and the CRC to the end.

18.19 USBPHY Tuning Guidelines

Ambiq's USB transceivers are certified with the USB-IF to meet the USB specifications for silicon components. However, the end-user device must also pass within the same limits.

To enable maximum reliability and flexibility in the end application, the USB PHY offers several software controlled tuning registers, such as On-Die-Termination (ODT), High-Speed Eye Height, Pre-emphasis, and Squelch Trigger Point. This section provides a brief overview of these settings within the AmbiqSuite SDK.

18.19.1 Hardware Requirements

A successful USB design requires attention to the USB data signal routing. The data pair traces must be routed according to the following guidelines:

- Impedance controlled to 90-ohm differential.
- Length matched within 0.025" (0.635mm) of each other.
- Minimized total trace length.

Common system elements, such as common mode EMI chokes, ESD suppression, external USB switches, and connectors, can all serve to negatively impact the electrical performance of the USB circuit.

18.19.2 Tuning Steps

Users should adjust the USB PHY settings for their design in the following order:

1. On-Die-Termination (ODT)
2. High-Speed Eye Height
3. Pre-emphasis
4. Squelch Trigger Point

18.19.2.1 ODT Configuration

ODT Configuration is done by setting the USBPHY_REG0C_BF62 register field (45 ohm HS ODT value tuning and FS/LS drive strength tuning). A value of 11111 for this 5-bit field is the smallest HS ODT value, the largest FS/LS drive strength and the fastest FS/LS slew rate. A value of 10000 is the largest HS ODT value, smallest FS/LS driver strength and the slowest FS/LS slew rate.

```
typedef enum
{
    AM_HAL_USB_PHY_R_ODT_VAL_15 = 0b11111, // Smallest HS ODT value,
                                              // Highest FS driver strength,
                                              // Fastest FS slew rate.
    AM_HAL_USB_PHY_R_ODT_VAL_14 = 0b11110,
    AM_HAL_USB_PHY_R_ODT_VAL_13 = 0b11101,
    AM_HAL_USB_PHY_R_ODT_VAL_12 = 0b11100,
    AM_HAL_USB_PHY_R_ODT_VAL_11 = 0b11011,
    AM_HAL_USB_PHY_R_ODT_VAL_10 = 0b11010,
    AM_HAL_USB_PHY_R_ODT_VAL_9  = 0b11001,
    AM_HAL_USB_PHY_R_ODT_VAL_8  = 0b11000,
    AM_HAL_USB_PHY_R_ODT_VAL_7  = 0b10111,
    AM_HAL_USB_PHY_R_ODT_VAL_6  = 0b10110,
    AM_HAL_USB_PHY_R_ODT_VAL_5  = 0b10101,
    AM_HAL_USB_PHY_R_ODT_VAL_4  = 0b10100,
    AM_HAL_USB_PHY_R_ODT_VAL_3  = 0b10011,
    AM_HAL_USB_PHY_R_ODT_VAL_2  = 0b10010,
    AM_HAL_USB_PHY_R_ODT_VAL_1  = 0b10001,
    AM_HAL_USB_PHY_R_ODT_VAL_0  = 0b10000, // Biggest HS ODT value,
                                              // Lowest FS driver strength,
                                              // Slowest FS slew rate.
}
am_hal_usb_phy_rodt_tuning_val_e;
```

Figure 78. Assignable Values of the USBPHY_REG0C_BF62 (ODT) Register Field

- Set to AM_HAL_USB_PHY_R_ODT_TUNE_VALUE at USB power-up in am_hal_usb_power_control().
- The SDK sets this to 0b10101 by default. The optimal setting is design-dependent and should be tuned to the impedance of the system.
- A method for tuning this value is as follows:
 - Start by placing the PHY in the HS_SE0_NAK state. This removes any voltage or current flow on D+ and D- without tri-stating the pins.
 - Configure the ODT value to 10001 and measure the resistance. This will provide the resistance R1 from D+ down through the termination resistance to ground, then back up through the D- termination resistance to the D- pin.
 - This resistance includes the resistance Runit/17 along with the parasitic resistances of the implementation Rx.
 - Generate a second resistance point R2 by configuring the ODT value to 10101 and measuring the resistance again. Note that the parasitic values will be the same for both measurements.
 - Compute the optimal setting for ODT with the following formula:

$$N = \frac{119 * (R1 - R2)}{60}$$

- The value for N is then rounded, converted into binary, and saved as the ODT value AM_HAL_USB_PHY_R_ODT_TUNE_VALUE.

An example of the impact of the ODT setting on the eye diagram for Ambiq validation hardware is seen in Figure 79.

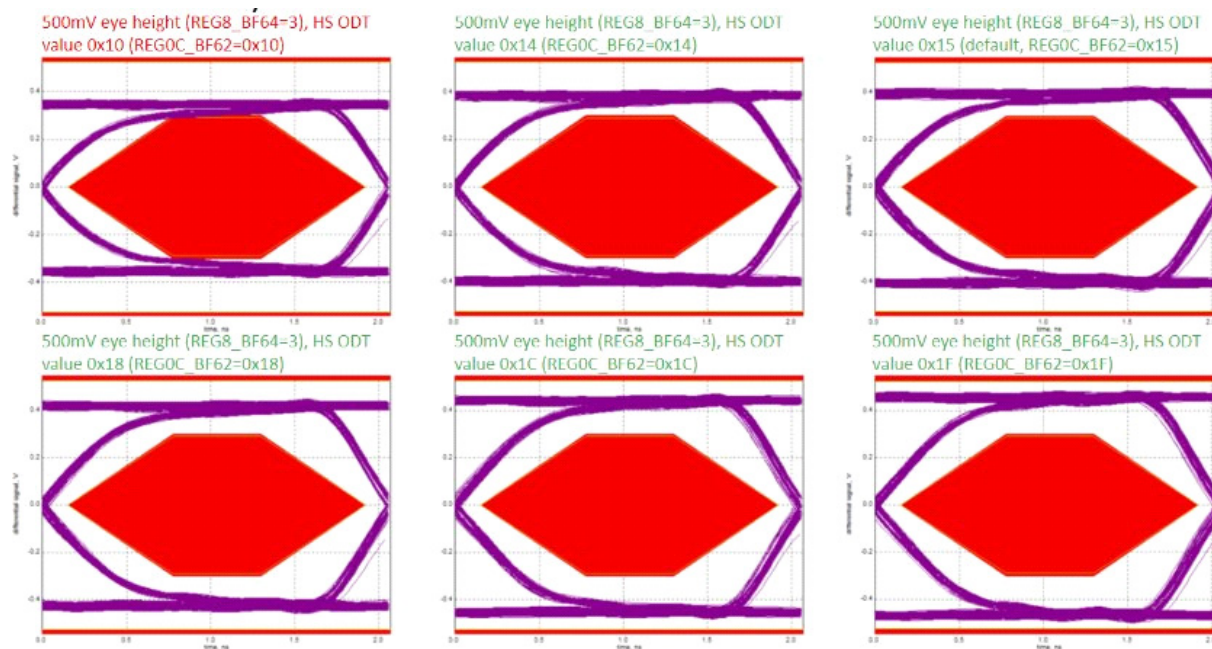


Figure 79. ODT Settings and Resulting Eye Diagrams for Validation Hardware

18.19.2.2 High-Speed Eye Height

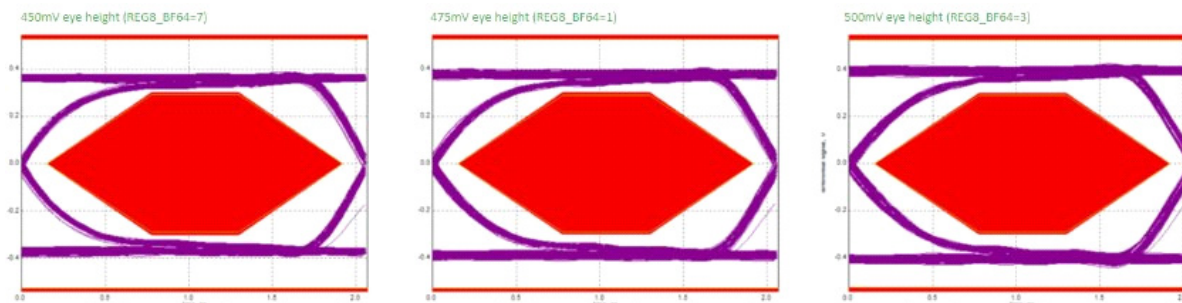
High-speed eye height tuning is done by setting the USBPHY_REG08_BF64 register field. Values for this 3-bit field result in the HS eye heights shown in Table 98.

Table 98: Assignable Values of the USBPHY_REG08_BF64 (HS Eye Height) Field

Field Value	Eye Height (mv)
000	400
001	475
010	350
011	500
100	412.5
101	425
110	437.5
111	450

Set this field value to `AM_HAL_USB_PHY_HS_EYE_HEIGHT_TUNE_VALUE` at USB power-up in `am_hal_usb_power_control()`. The SDK sets this field to 011 (500 mV). Adjustments may be needed for devices failing EL_4 or EL_5 (whichever test is applicable).

An example of the impact of the HS Eye Height setting on the eye diagram for Ambiq validation hardware is seen in Figure 80.

**Figure 80. HS Eye Height Settings and Resulting Eye Diagrams for Validation Hardware**

18.19.2.3 Pre-emphasis

High-speed pre-emphasis strength is set via the `USBPHY_REG14_BF42` register field. Values for this 3-bit field result in a range of strengths from the weakest (000) to the strongest (111).

This field is set to `AM_HAL_USB_PHY_PREEMP_ENABLE_STATE` at USB power-up in `am_hal_usb_power_control()`. The SDK sets this field to 001.

Enabling pre-emphasis increases the slew rate on DP/DM and should be used to compensate for excessive capacitive loading. The designer must take care not to violate the `TEST_J` or `TEST_K` tests by applying a pre-emphasis that is too high in these states. A setting of 111 should be used as a last resort.

It is possible that pre-emphasis can lead to signal integrity degradation when the transmission lines do not exhibit significant high-frequency attenuation or have characteristics that do not respond well to the higher slew rate, such as short transmission lines, low-loss transmission lines, well-matched impedance lines, or lines with limited bandwidth.

An example of the impact of the pre-emphasis setting on the eye diagram for Ambiq validation hardware with a 7" cable is seen in Figure 81.

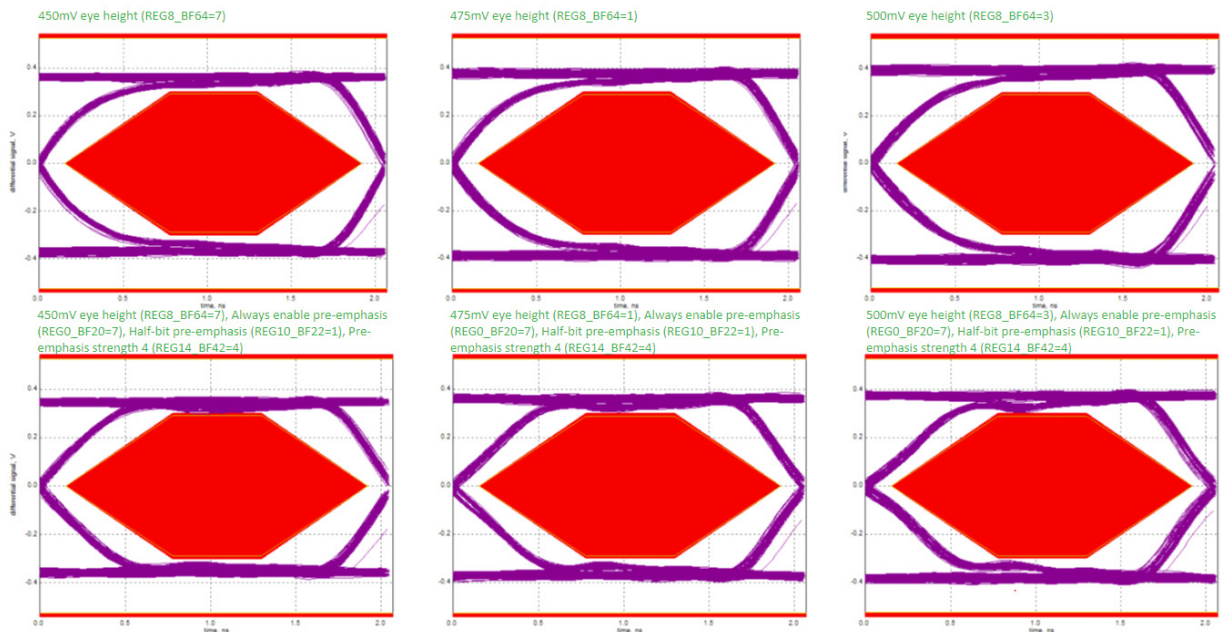


Figure 81. Pre-emphasis Settings and Resulting Eye Diagrams for Validation Hardware

18.19.2.4 Squelch Trigger Point

Tuning of the RX squelch trigger point sets the bias current to compensate for package and board level parasitic, and is done by setting the USBPHY_REG08_BF30 register field. Values for this 4-bit field result in squelch trigger points shown in Table 99.

Table 99: Assignable Values of the USBPHY_REG08_BF30 (Squelch Trigger Point) Field

Field Value	Squelch Trigger Point (mv)
0000	112.5
0001	150
0010	87.5
0011	162.5
0100	100
0101	137.5
0110	75
0111	150
1000	125
1001	162.5
1010	100

Table 99: Assignable Values of the USBPHY_REG08_BF30 (Squelch Trigger Point) Field

Field Value	Squelch Trigger Point (mv)
1011	175
1100	150 (default)
1101	187.5
1110	125
1111	200

This field is set to AM_HAL_USB_PHY_HS_SQUELCH_TUNE_VALUE at USB power-up in `am_hal_usb_power_control()`. The SDK sets this field to 150 mv (1100).

```
typedef enum
{
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1125V = 0,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1500V_1 = 1,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P0875V = 2,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1625V = 3,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1000V = 4,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1375V = 5,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P0750V = 6,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1500V_2 = 7,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1250V = 8,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1625V_2 = 9,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1000V_2 = 10,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1750V = 11,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1500V = 12,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1875V = 13,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P1250V_2 = 14,
    AM_HAL_USB_PHY_HS_SQUELCH_TUNE_0P2000V = 15,
}
am_hal_usb_phy_hs_squelch_tuning_e;
```

Figure 82. Assignable Values of the USBPHY_REG08_BF30 (Squelch Trigger Point) Field

The programmable squelch trigger point gives the product designer the ability to lower the threshold of the squelch detector. This can be helpful for devices failing to pass the EL_17 test. Note that setting the trigger point value too low can make the device overly sensitive, thus failing the high-speed receiver sensitivity test, EL_16.

19. Secure Digital Input Output (SDIO)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

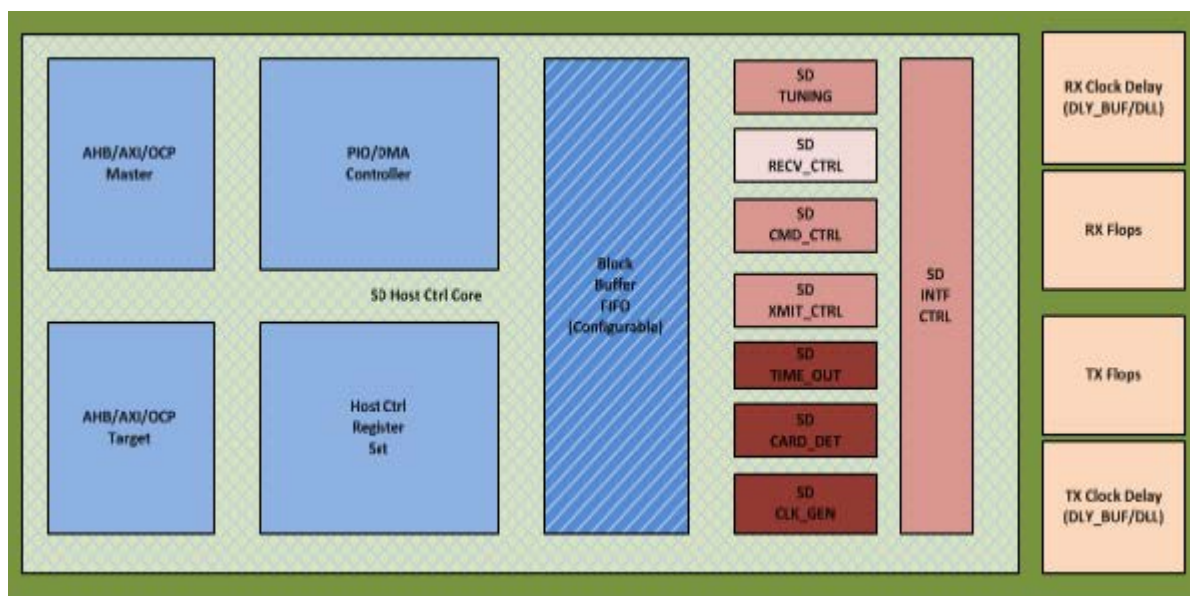


Figure 83. SD/SDIO Host Controller Interface Block Diagram

19.1 Functional Description

The SD/SDIO Host Controller, or referred here as just SD Host Controller, interfaces to the system bus and has the components and functionality described in the following sub-sections.

19.1.1 Host Controller Register Set

The Host Controller register set implements the registers defined by the SD Host Controller Specification (Version 3.00). The registers are byte/DWORD accessible from the Target Interface.

The Host Controller register set also implements the Data Port Registers for the PIO Mode transfers. The register set provides the control signals to rest of the blocks and monitors the status signals from the blocks to set interrupt status bits and eventually generate an interrupt signal to the host bus. The Host Controller register set acts as the bridge between host CPU and Host Controller. The SD/SDIO controller registers are programmed by the host processor through the host (AHB) target interface. Interrupts are generated to the host processor based on the values set in the interrupt status register and interrupt enable registers.

19.1.2 PIO/DMA Controller

The PIO/DMA Controller module implements the SDMA and ADMA2 engines as defined in the SD Host Controller Specification and maintains the block transfer counts for PIO operation. It interacts with the registers set and starts the DMA engine when a command with data transfer is involved. The DMA Controller interfaces to the host (AHB) manager module to generate transfers. On the other side it interfaces with the block buffer to store/fetch block data. The DMA Controller implements a separate DMA

for SDMA operation and separate DMA for the ADMA2 operation. In addition it implements the Host Transaction Generator that generates controls for the Host Manager Interface module.

19.1.3 Block Buffer

The Host Controller uses a dual-port block buffer (read/write on both ports) that is used to store the block data during SD transfers. The size of the block buffer is 2 kB. This allows support for a 512-byte block size for SD/eMMC and up to a 1 kB block size for SDIO implementations. The block buffer uses circular buffer architecture. One side of the block buffer is interfaced with the DMA controller and operates on the host clock. The other side of the block buffer interfaces with SD control logic and operates on the SD clock. During a write transaction (data transferred from the SoC to the SD3.0 / SDIO3.0 / eMMC4.51 card), the data is fetched from the system memory and is stored in the block buffer. When a block of data is available, the SD control logic will transfer it onto the SD interface. The DMA controller continues to fetch additional blocks of data when the block buffer has space. During a read transaction (data transferred from SD3.0 / SDIO3.0 / eMMC4.51 card to the host controller), the data from the SD3.0 / SDIO3.0 / eMMC4.51 card will be written to the block buffer and at the end, when the CRC of the block is valid, the data is committed. When a block of data is available, then the DMA controller transfers this data to the system memory. The SD control logic meanwhile receives the next block of data provided there is space in the block buffer. If the Host Controller cannot accept any data from the SD3.0 / SDIO3.0 / eMMC4.51 card, then it will issue read wait (if the card supports a read wait mechanism) to stop the data transfer from the card or by stopping the clock.

NOTE

FIFO depth can be varied using parameter passed to the core using the 'dot parameter instantiation'.

NOTE

When the Block Buffer size is twice that of the block size, the block buffer behaves like a ping-pong buffer.

19.1.4 SD Clock Generation

The SD Clock Generator module generates the SD clock from the reference clock (xin_clk), based on the controls programmed in the Clock Control Register. These include the Clock Divide Value, SD Clock Enable etc. The outputs from this module are the SD_CLK and the SD_CARD Clock. The SD_CLK is used by the SD control logic and the SD_CARD clock connected to the "CLK" pin on the SD interface. This module also generates system resets to various clock domains.

19.1.5 SD Card Detect

The SD Card detect logic monitors the SD_CD# pin for card insertion/removal events. It implements de-bouncing logic to filter the false transitions on the SD_CD# pin. Card insertion and removal events are reported in the SD host register set from which the interrupt is eventually generated.

19.1.6 SD Timeout Control

The SD timeout control logic implements a timeout check between block transfers. It uses the contents of the Timeout Control Register to implement a timeout between blocks. This module operates under the control of the Transmit Control and Receive Control modules (based on direction). When a timeout is detected, the event is reported to Transmit Control or Receive Control module.

19.1.7 SD Command Control

The SD Command Control module generates a command sequence on the CMD line of the SD interface for every new command programmed by the software. The Command Control module also implements the response reception and checks the validity of the response. It uses the response type field to determine the length of the response and the presence of the CRC7 field. The response is received on the receive clock, which is either the looped back clock or the tuned clock. Once the response is received, the contents of the response (start bit, command Index, CRC7 and end bit) are verified and the response status is captured in various status bits in the module's register set. It also implements a timeout check upon response reception to make sure that the response is received within the specified time (5 or 64 clocks based on command type). The received response is then stored in appropriate fields in the Response Register. The SD Command Control module generates controls in the SD Transmit Control module and SD Receive Control module based on transfer direction. The SD Command Control module also generates an auto command (AutoCMD12 or AutoCMD23) when enabled.

19.1.8 SD Transmit Control

The SD Transmit Control module is used for write transfers when transferring data to the card. Once the command is issued, this module waits for the block of data to be available in the block buffer and transfers this on to the SD's DATn line(s). Based on the configuration of the data lines (1-bit, 4-bits or 8-bits), the data from the block buffer is appropriately routed. The CRC16 is individually calculated on a per-lane basis and is attached at the end of the block transfer before the END bit. In the case of DDR operation, it implements separate CRC16 for each edge of the clock. At the end of block transfer, it waits for the CRC response on the DAT0 line and the result of the CRC check is stored in the CRC status register field.

This module also checks for a Write Busy indication (DAT0 Line) before transferring the next block of data. A timeout check is implemented to make sure that the Write Busy is asserted before the specified time limit.

19.1.9 SD Receive Control

The SD Receive Control module is used for read transfers when receiving data from the card. Once the command is issued, this module waits for the block of data to be received from the card. Based on the configuration of the data lines (1-bit, 4-bits or 8-bits), the data from the SD interface is assembled into byte, and eventually in 32-bit word, format before it is written to the block buffer. The CRC16 is individually calculated on a per-lane basis and is compared to the received CRC16 at the end of the block transfer before the END Bit. In the case of DDR operation, it implements separate CRC16 checks for each edge of the clock.

The data is received on the receive clock. This receive clock is either the looped-back clock (SDCARD_CLK from the IO_BUF) or the tuned clock using DLL or DLY elements. A timeout check is performed to make sure that the gap between received blocks does not exceed the specified time limit.

19.1.10 SD Tuning Block

The SD tuning block is used for SDR104 or SDR50 (optionally when enabled) and eMMC Hs200 modes to tune the receive clock. The tuning block generates the delay controls in the external Delay Controller module. The tuning modules receive the 64-byte tuning block (SD mode) or 128-byte tuning block (eMMC mode) and maintains a tuning vector to determine the optimal delay. The tuning block can be configured with a number of delay taps (maximum of 32). Using this the tuning block performs the tuning and selects the optimal tap point for the receive clock.

19.1.11 SD Intf Control

The SD interface control block maps the internal signals to the external SD Interface and vice versa. Based on the bus width (1-, 4- or 8-bit) the internal signals are driven out appropriately. In the case of DDR operation, the outputs are driven on the negative edge of the SD_CLK.

The inputs from RxFlops module are latched on the RX_CLK (looped-back or tuned clock) and is output to the Receive Control module for further processing.

19.2 Clocks

The Host Controller has the following clocks:

xin_clk (external input) - This is the main clock used on the SD interface side of the Host Controller. This clock input is used to generate the SD clock based on the divisor value programmed by the host driver. To support SDR50, this clock needs to be about 100 MHz (use 96 MHz HFRC clock).

ahb_clk (external input, commonly called host_clk) - host bus clock. Based on the selected host interface, the appropriate clock is used. The host bus interface modules (AHB manager/target modules), the SD register set and the DMA controller operate on this clock.

sd_clk (internally generated) - This clock is derived from xin_clk based on the clock control register value programmed by the driver. This clock is used by most of the SD control logic (SD command control, SD transmit control, SD tuning module and the other side of the block buffer).

sdcard_clk - same as sd_clk, except that this is available only when SD clock enable is set by the driver. This is the clock supplied to the SD card. The Host Controller supports both full speed and high speed cards. For high speed cards, the Host Controller should clock out the data on the rising edge of clk_sd and for full speed cards, the Host Controller should clock out the data on the falling edge of clk_sd. During read transactions, when the read FIFOs are full and there is no space to accept one block of data from the card, the host controller will halt the transaction. This will stop the clock to the card so an overrun condition will not occur on the SD side.

sdcard_clk_dly - This is the delayed version of the sd_clk. This clock is used to flop the SD outputs to provide hold time on the output pins.

rxclk_in - This is the looped-back clock from the sdcard_clk. This will account for the delay on the clock being driven onto the SD Interface through the chip and the IO pad.

rx_clk - This is selected from the rxclk_in (after manual tap delay) or the tuned clock (auto tap delay). The response receive logic in the command control and the SD receive controller operates on this clock. The received data is assembled into bytes and is synchronized to sd_clk before writing into the block buffer.

19.2.1 SD clocking Architecture

Figure 84 shows the architecture of clocks used in the interface part of the SD Controller. The Primary Clock input **xin_clk** is the main clock used to generate various derived clocks.

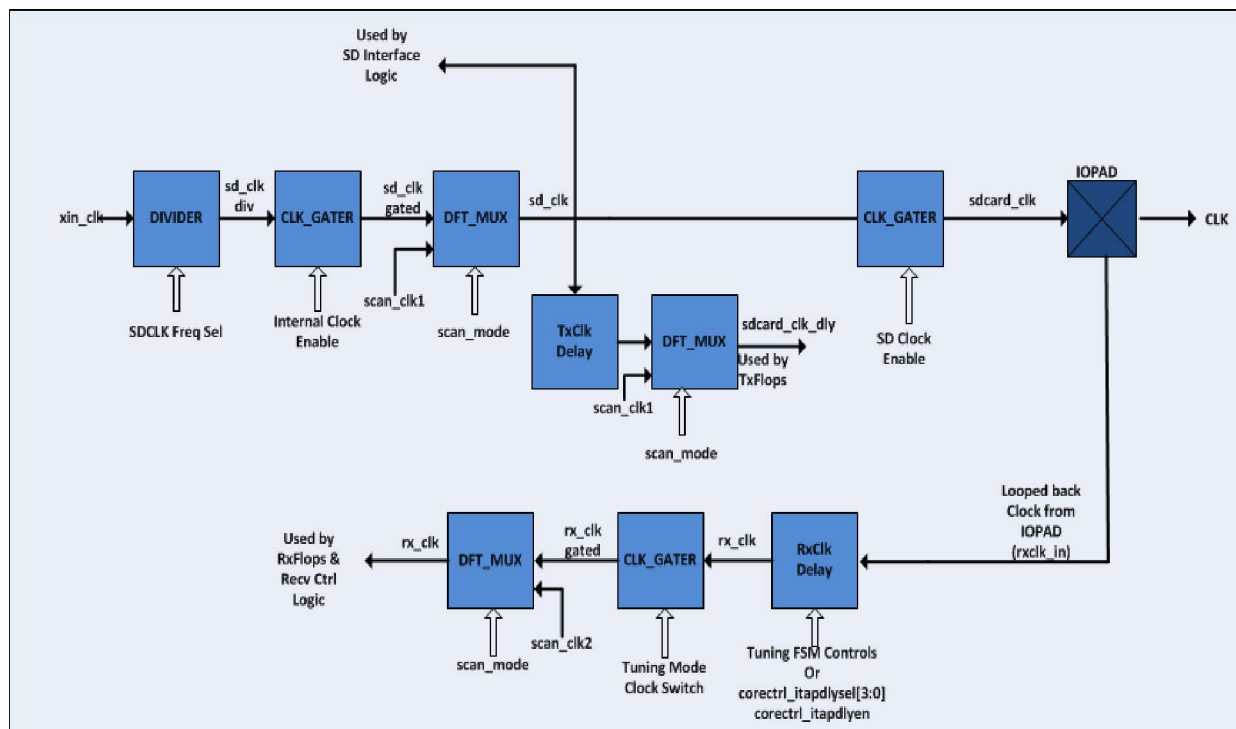


Figure 84. SDIO Module Clock Derivation

descriptor line. ADMA2 includes blocks for SDMA, state machine and registers. ADMA2 does not use 32-bit SDMA System Address Register (offset 0) but uses the 64-bit Advanced DMA System Address register (offset 058h) for descriptor pointer. Writing to the Command Register triggers off an ADMA2 transfer. ADMA2 fetches one descriptor line and executes it. This procedure is repeated until the end of the descriptor is reached (End=1 in attribute).

19.3.2 An Example of ADMA2 Programming

The CPU forms the Descriptor Table when each slice is placed somewhere in contiguous system memory. It describes each descriptor with a set of address, length and attributes fields. Each sliced data is transferred in turn as programmed in the descriptor, as shown in Figure 86.

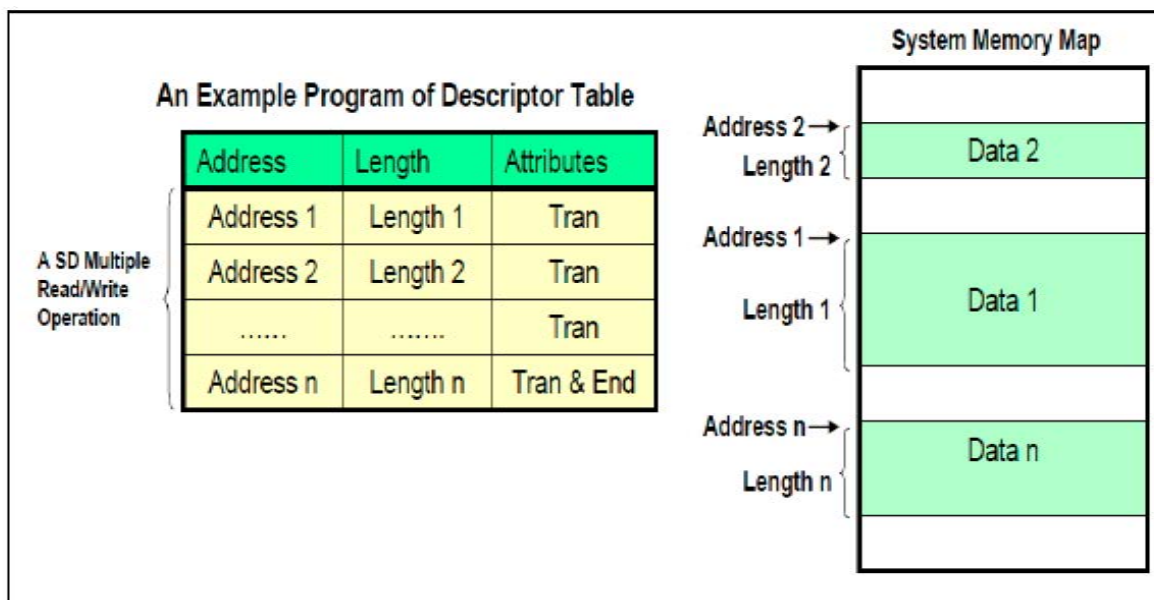


Figure 86. Example Descriptor Table

19.3.3 Data Address and Data Length Requirements

There are 3 requirements when programming the descriptor.

1. The minimum unit of address is 4 bytes.
2. The maximum data length of each descriptor line is less than 64KB.
3. Total Length = Length 1 + Length 2 + Length 3 + ... + Length n = multiple of Block Size.

If the total length of a descriptor is not a multiple of block size, an ADMA2 transfer might not be terminated. In this case, the transfer should be aborted by a data timeout.

The Block Count register limits the maximum blocks transferred to 65535 blocks. If an ADMA2 operation is less than or equal to 65535 blocks transferred, then the Block Count register can be used. In this case, total length of the Descriptor Table is equal to the block size multiplied by the block count. If an ADMA2 operation is more than 65535 blocks transferred, then the Block Count Register is disabled by setting Block Count Enable to 0 in the Transfer Mode Register. In this case, the length of the data transfer is not designated by block count but by the Descriptor Table. Therefore, the timing of detecting the last block on the SD bus may be different and it affects the control of Read Transfer Active, Write Transfer Active and DAT line Active in the Present State register. In case of a read operation, several blocks may be read more

than required. The CPU should ignore an out of range error if the read operation is for the last block of memory area.

19.3.4 Descriptor Table

Figure 87 shows a 32-bit addressing Descriptor Table.

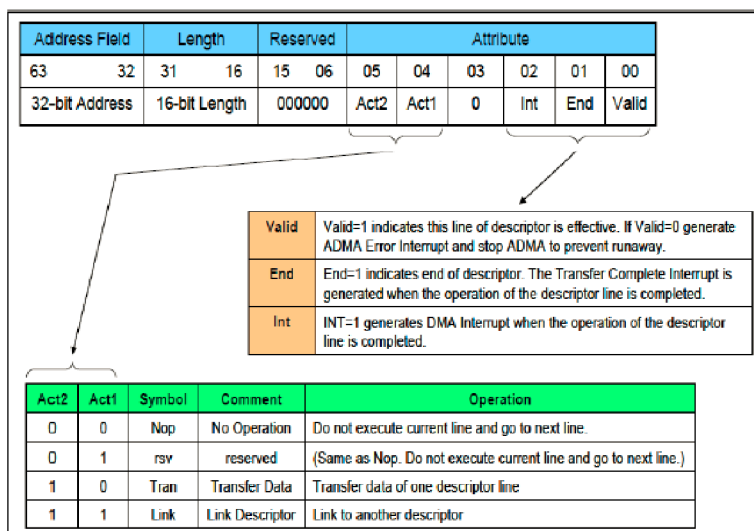


Figure 87. 32-bit Addressing Descriptor Table

One descriptor line consumes a 64-bit (8-byte) memory space. Attributes are used to specify an action in the control descriptor. Three separate actions are specified. A "Nop" operation skips the current descriptor line and fetches the next one. A "Tran" operation transfers data designated by the address and length fields. A "Link" operation is used to connect two separate descriptors. The address field of the link operation points to the next Descriptor Table. The combination of Act2=0 and Act1=1 is reserved and performs the same operation as Nop. A future version of the controller may use this field and define a new operation. A 32-bit address is stored in the lower 32 bits of a 64-bit address register. The Address field should be set on a 32-bit boundary (lower 2 bits are always set to 0) for a 32-bit address descriptor table.

Figure 88 below shows the definition of the length field in the Descriptor Table.

Length Field	Value Of Length
0000h	65536bytes
0001h	1byte
0002h	2bytes
-----	-----
FFFFh	65535bytes

Figure 88. Descriptor Table Length Field Definitions

19.4 Driver flow sequence

The following sub-sections describe the flow sequence for three types of SDIO transactions:

1. Non DMA data transaction
2. DMA data transaction
3. ADMA data transaction

19.4.1 Non-DMA Transaction

The sequence for transactions when not using DMA (non-DMA) is shown in Figure 89 and the steps are listed in Table 100.

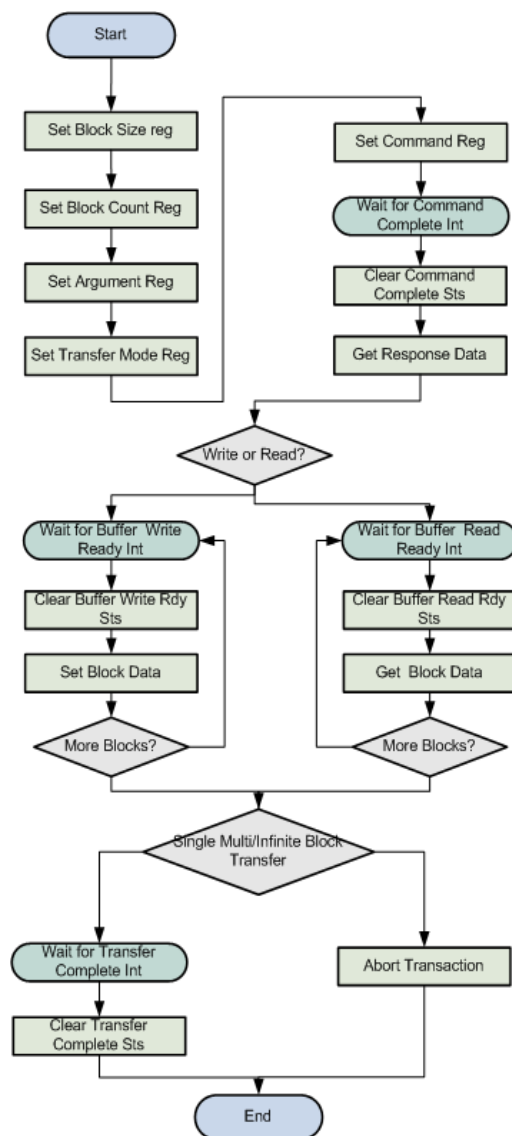


Figure 89. Non-DMA Transfer Flow Sequence

Table 100: Non-DMA Transfer Flow Sequence Steps

Step	Description
1	Set the value corresponding to the executed data byte length of one block to Block Size register.
2	Set the value corresponding to the executed data block count to Block Count register.
3	Set the value corresponding to the issued command to Argument register.
4	Set the value to Multi / Single Block Select and Block Count Enable. Set the value corresponding to the issued command to Data Transfer Direction, Auto CMD12 Enable and DMA Enable.
5	Set the value corresponding to the issued command to Command register. When writing the upper byte of Command register, SD command is issued.
6	Wait for the Command Complete Interrupt
7	Write 1 to the Command Complete in the Normal Interrupt Status register for clearing this bit.
8	Read Response register and get necessary information in accordance with the issued command.
9	In the Case Where this sequence is for write to a card, go to step (10-W). In case of read from a card, go to step (10-R).
10-W	Wait for Buffer Write Ready Interrupt. Non DMA Write Transfer On receiving the Buffer Write Ready interrupt the ARM processor will act as a master and start transferring the data via Buffer data port register (fifo_1). Transmitter starts sending the data in SD bus when a block of data is ready in fifo_1. While transmitting the data in sd bus the buffer write ready interrupt is sent to the ARM Processor for the second block of data. The ARM processor will act as a master and start sending the second block of data via Buffer data port register to fifo_2. Buffer write ready interrupt will be asserted only when a fifo is empty to receive a block of data.
11-W	Write 1 to the Buffer Write Ready in the Normal Interrupt Status register for clearing this bit.
12-W	Write block data (in according to the number of bytes specified at the step (1)) to Buffer Data Port register.
13-W	Repeat until all blocks are sent and then go to step (14).
10-R	Non DMA Read Transfer Buffer Read Ready interrupt is asserted whenever a block of data is ready in one of the fifo's. On receiving the Buffer Read Ready interrupt the ARM processor will act as a master and start reading the data via Buffer data port register (fifo_1). Receiver start reading the data from SD bus only when a fifo is empty to receive a block of data. When both the fifo's are full the host controller will stop the data coming from the card through read wait mechanism (if card supports read wait) or through clock stopping. Wait for Buffer Read Ready Interrupt
11-R	Write 1 to the Buffer Read Ready in the Normal Interrupt Status register for clearing this bit.
12-R	Read block data (in according to the number of bytes specified at the step (1)) from the Buffer Data Port register.
13-R	Repeat until all blocks are received and then go to step (14).
14	If this sequence is for Single or Multiple Block Transfer, go to step (15). In case of Infinite Block Transfer, go to step (17).
15	Wait for Transfer Complete Interrupt.
16	Write 1 to the Transfer Complete in the Normal Interrupt Status register for clearing this bit.
17	Perform the sequence for Abort Transaction.

Note: Step (1) and (2) can be executed at same time. Step (4) and(5) can be executed at same time.

Note: Step (1) and (2) can be executed at same time. Step (4) and(5) can be executed at same time.

19.4.2 DMA Transaction

The burst types like 8-beat incrementing burst or 4-beat incrementing burst or single transfer is used to transfer or receive the data from system memory mainly to avoid the hold of the Host/System bus by the manager for a longer time.

The sequence for using DMA is shown in Figure 90 and the flow sequence steps are listed in Table 101.

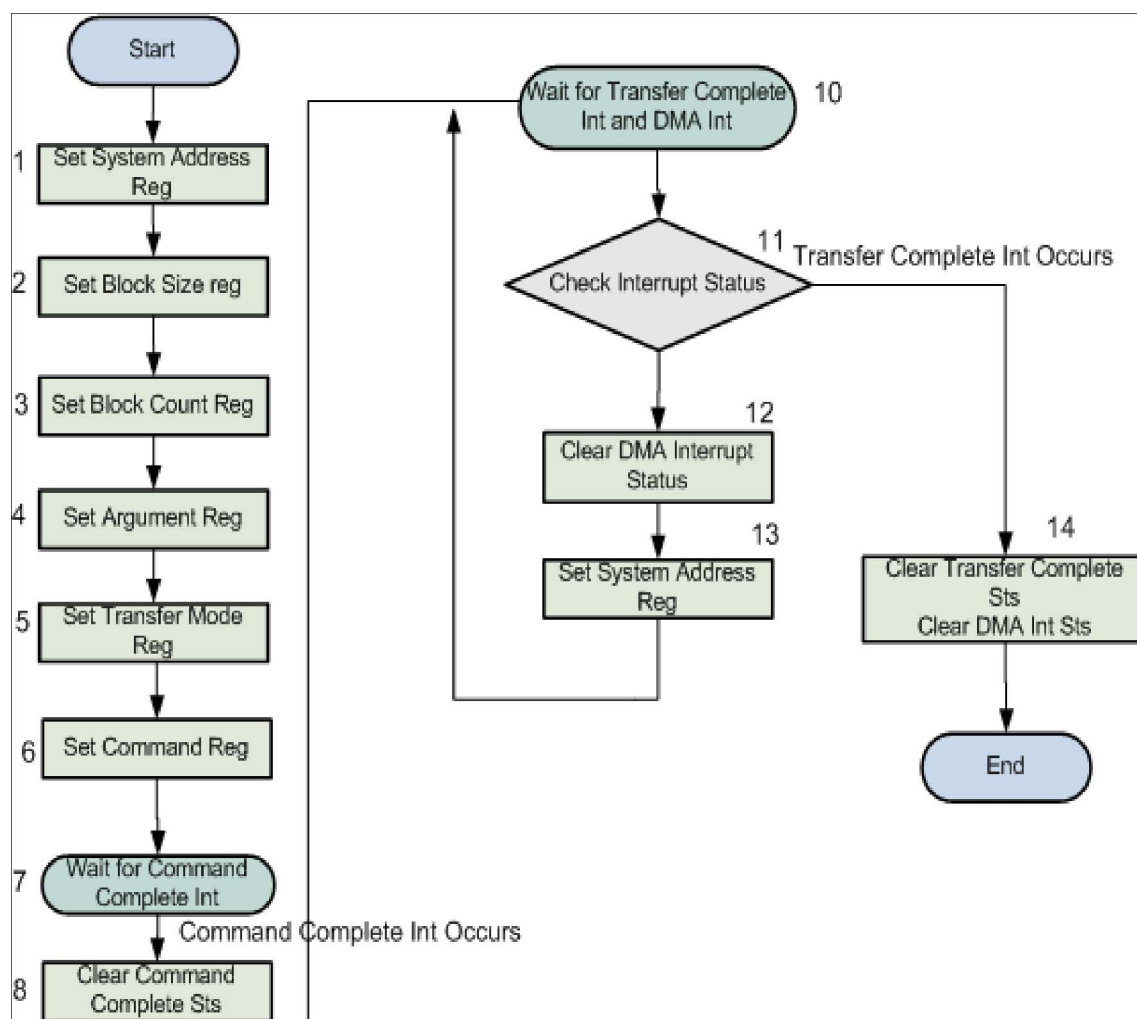


Figure 90. DMA Transfer Flow Sequence

Table 101: DMA Transfer Flow Sequence Steps

Step	Description
1	Set the system address for DMA in the System Address register.
2	Set the value corresponding to the executed data byte length of one block in the Block Size register.
3	Set the value corresponding to the executed data block count in the Block Count register.
4	Set the value corresponding to the issued command to Argument register.
5	Set the value to Multi / Single Block Select and Block Count Enable. Set the value corresponding to the issued command to Data Transfer Direction, Auto CMD12 Enable and DMA Enable.

Step	Description
6	Set the value corresponding to the issued command to Command register. When writing the upper byte of Command register, SD command is issued.
7	Wait for the Command Complete Interrupt
8	Write 1 to the Command Complete in the Normal Interrupt Status register for clearing this bit.
9	Read Response register and get necessary information in accordance with the issued command. DMA read transfer On receiving the response end bit from the card for the write command (data flowing from Host to Card) the SD Host controller will act as the master and request the System/Host bus. After receiving the grant the host controller will start reading a block of data from the system memory and fills the first fifo. Whenever a block of data is ready the transmitter will start sending the data in SD bus. While transmitting the data in SD bus the host controller requests the bus to fill the second block in second fifo. Ping Pong fifo's are used to increase the throughput. Similarly the host controller reads a block of data from the system memory whenever a fifo is empty. This will continue till all the blocks are read from the System memory. Transfer complete Interrupt will be set only after transferring all the blocks of data to the card. DMA write transfer The block of data received from the Card (data flowing from Card to Host) is stored in first half of the fifo. Whenever a block of data is ready the SD Host controller will act as the master and request the System/Host bus. After receiving the grant the host controller will start writing a block of data into the system memory from the first fifo. While transmitting the data into System memory the host controller will receive the second block of data and store in second fifo. Similarly the host controller write a block of data into the system memory whenever data is ready. This will continue till all the blocks are transferred to the System memory. Transfer complete Interrupt will be set only after transferring all the blocks of data to the System memory. Note: Host controller will receive a block of data from the card only when it has room to store a block of data in fifo. When both the fifo's are full the host controller will stop the data coming from the card through read wait mechanism (if card supports read wait) or through clock stopping.
10	Wait for the Transfer Complete Interrupt and DMA Interrupt.
11	If Transfer Complete is set 1, go to Step(4) else if DMA Interrupt is set to 1, go to Step(12). Transfer Complete is higher priority than DMA Interrupt.
12	Write 1 to the DMA Interrupt in the Normal Interrupt Status register to clear this bit.
13	Set the next system address of the next data position to the System Address register and go to Step (10).
14	Write 1 to the Transfer Complete and DMA Interrupt in the Normal Interrupt Status register to clear this bit.

Note: Step (2) and Step (3) can be executed at same time. Step (5) and Step (6) can also be executed at same time.

As an example, if the host wants to transfer 4 kB of data to the card, assuming a maximum block size of 512 bytes, then the host driver will program the Block Size Register as 512 and the Block Count Register with the value 8. The manager and transmitter residing inside the Host Controller will get the information (how much data to transfer) from these registers. Using the above information, the manager will initiate a data read transaction (to read a block of data - 512 bytes from system memory). Whenever a block of data is ready in the FIFO, the transmitter will start transmitting the block of data (512) on the SD bus. After transmitting the entire block of data to the card, the transmitter will wait for a status response from the card. The transmitter will send the next block of data only after it receives a good status response from the card for the previous block of data, otherwise the transaction will be aborted and the host will go for a fresh transaction.

19.4.3 ADMA Transactions

The sequence for using DMA is shown in Figure 90 and the flow sequence steps are listed in Table 101.

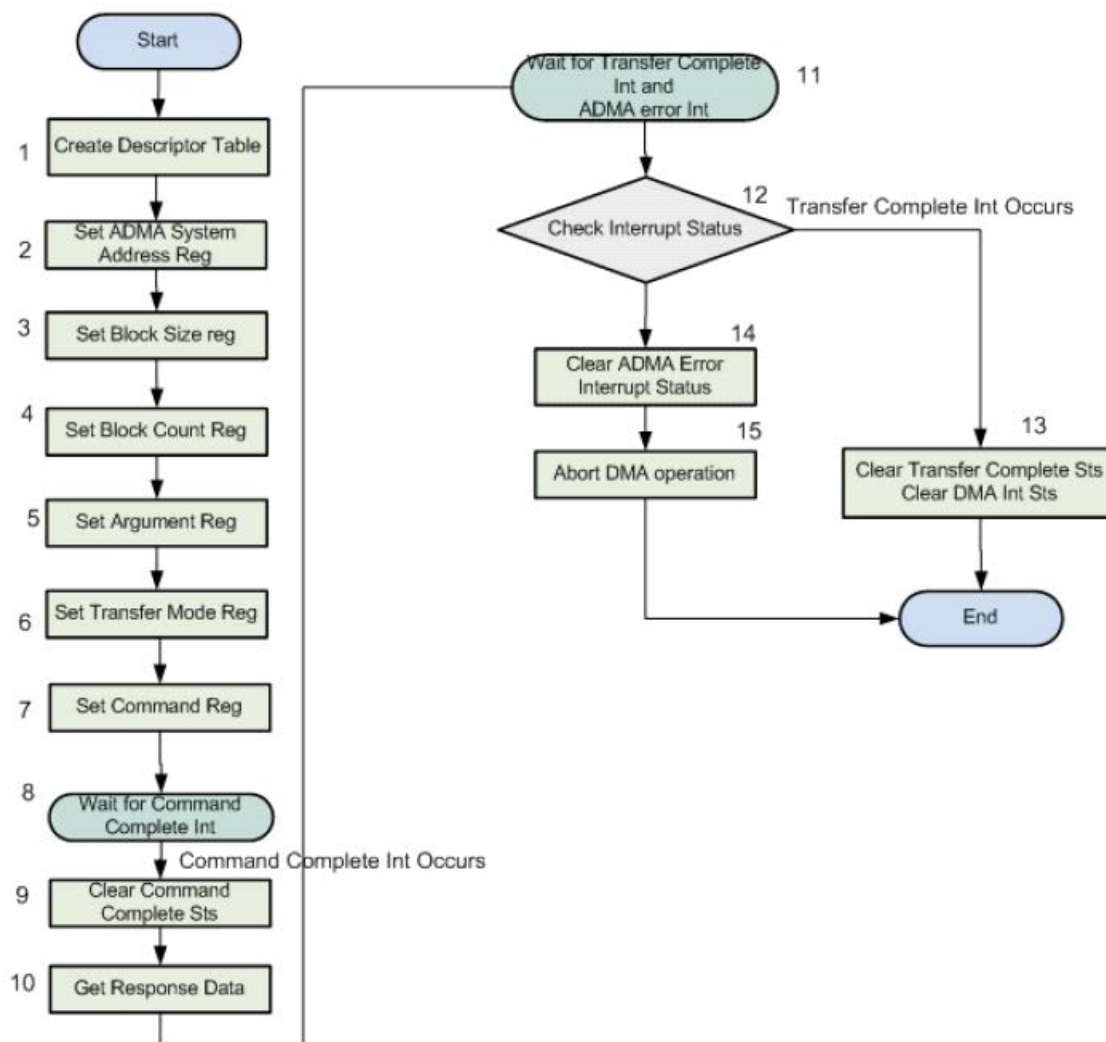


Figure 91. ADMA Transaction Flow Sequence

Table 102: ADMA Transaction Flow Sequence Steps

Step	Description
1	Create Descriptor table for ADMA in the system memory
2	Set the Descriptor address for ADMA in the ADMA System Address register.
3	Set the value corresponding to the executed data byte length of one block in the Block Size.
4	Set the value corresponding to the executed data block count in the Block Count register. If the Block Count Enable in the Transfer Mode register is set to 1, total data length can be designated by the Block Count register and the Descriptor Table. These two parameters shall indicate same data length. However, transfer length is limited by the 16-bit Block Count register. If the Block Count Enable in the Transfer Mode register is set to 0, total data length is designated by not Block Count register but the Descriptor Table. In this case, ADMA reads more data than length programmed in descriptor from SD card. Too much read operation is aborted asynchronously and extra read data is discarded when the ADMA is completed
5	Set the argument value to the Argument register.
6	Set the value to the Transfer Mode register. The host driver determines Multi / Single Block Select, Block Count Enable, Data Transfer Direction, Auto CMD12 Enable and DMA Enable.
7	Set the value to the Command register. Note: When writing to the upper byte [3] of the Command register, the SD command is issued and DMA is started.
8	Wait for the Command Complete Interrupt.
9	Write 1 to the Command Complete in the Normal Interrupt Status register to clear this bit.
10	Read Response register and get necessary information of the issued command.
11	Wait for the Transfer Complete Interrupt and ADMA Error Interrupt.
12	If Transfer Complete is set 1, go to Step (13) else if ADMA Error Interrupt is set to 1, go to Step
13	Write 1 to the Transfer Complete Status in the Normal Interrupt Status register to clear this bit.
14	Write 1 to the ADMA Error Interrupt Status in the Error Interrupt Status register to clear this bit.
15	Abort ADMA operation. SD card operation should be stopped by issuing abort command. If necessary, the host driver checks ADMA Error Status register to detect why ADMA error is generated.

Note: Step (3) and Step (4) can be executed simultaneously. Step (6) and Step (7) can also be executed simultaneously.

19.5 Using the AmbiqSuite SDK to Program and Use the SDIO Module

In order to access the eMMC/SDIO/SD card, follow the steps listed below. Currently the AmbiqSuite SDK supports 3 types of cards.

- **Define a global card host pointer**

```
am_hal_card_host_t *pSdhcCardHost = NULL;
```

- **Configure the SDIO pins**

1-bit, 4-bit and 8-bit bus widths are supported. Set the bus width appropriately for the hardware design.

```
//
// Configure SDIO PINs.
//
am_bsp_sdio_pins_enable(SDIO_INSTANCE, AM_HAL_HOST_BUS_WIDTH_8);
```

- **Initialize the SDHC card host handle with current SDIO instance number**

Currently two card hosts are supported. The SDIO host instance is listed in `am_hal_host_inst_index_e`.

```
//
// Get the underlying SDHC card host instance
//
pSdhcCardHost = am_hal_get_card_host(eIndex, true);

if (pSdhcCardHost == NULL)
{
    AM_HAL_CARD_DEBUG("No such card host and stop\n", 0);
    return AM_HAL_STATUS_FAIL;
}
```

- **Define a card and check if it is present**

We can define 1 of 3 card types here and use one of them in `am_hal_card_host_find_card()`.

```
am_hal_card_t Card
//
// check if card is present
//
if (am_hal_card_host_find_card(pSdhcCardHost, &Card) != AM_HAL_STATUS_SUCCESS)
```

```

{
    AM_HAL_CARD_DEBUG("No card is present now\n", 0);
    return AM_HAL_STATUS_FAIL;
}

```

▪ Initialize a card

The card initialization supports different card types:

- AM_HAL_CARD_TYPE_EMMC
- AM_HAL_CARD_TYPE_SDHC
- AM_HAL_CARD_TYPE_SDIO

It also supports auto-detecting a card type. When the card type is unknown, the user can provide a card power control callback function during card initialization, as well as establish a card “power off” policy.

There are 3 SDIO power policies (eCardPwrCtrlPolicy):

- AM_HAL_CARD_PWR_CTRL_SDHC_OFF
- AM_HAL_CARD_PWR_CTRL_SDHC_OFF_AND_CARD_SLEEP
- AM_HAL_CARD_PWR_CTRL_SDHC_AND_CARD_OFF

```

uint32_t am_hal_card_init(am_hal_card_t *pCard, am_hal_card_type_e eType,
                          am_hal_card_pwr_ctrl_func pCardPwrCtrlFunc,
                          am_hal_card_pwr_ctrl_policy_e eCardPwrCtrlPolicy);

```

```

if (am_hal_card_init(&Card, AM_HAL_CARD_TYPE_EMMC, NULL, AM_HAL_CARD_PWR_CTRL_NONE) !=
    AM_HAL_STATUS_SUCCESS)
{
    AM_HAL_CARD_DEBUG("card initialization failed\n", 0);
    return AM_HAL_STATUS_FAIL;
}

```

After initialization, the card is running with the default setting - 1-bit bus width and lowest clock speed.

▪ Configure the card

Configure the width, mode and voltage for card and SDIO bus. for eMMC card, usually set as 8-bit DDR mode:

```

//
// 48MHz, 8-bit DDR mode for read and write
//
while (am_hal_card_cfg_set(&Card, AM_HAL_CARD_TYPE_SDHC,
                          AM_HAL_HOST_BUS_WIDTH_8, 48000000, AM_HAL_HOST_BUS_VOLTAGE_1_8,
                          AM_HAL_HOST_UHS_DDR50) != AM_HAL_STATUS_SUCCESS)
{

```

```

    am_util_delay_ms(1000);
    am_util_stdio_printf("setting DDR50 failed\n");
}

```

An SDIO or SD card is usually set to 48 MHz, 4-bit, and SDR mode. The code is as follow:

```

//
// 48 MHz, 4-bit SDR mode for read and write
//
while (am_hal_card_cfg_set(&Card, AM_HAL_CARD_TYPE_EMMC,
    AM_HAL_HOST_BUS_WIDTH_4, 48000000, AM_HAL_HOST_BUS_VOLTAGE_1_8,
    AM_HAL_HOST_UHS_SDR50) != AM_HAL_STATUS_SUCCESS)
{
    am_util_delay_ms(1000);
    am_util_stdio_printf("setting SDR50 failed\n");
}

```

▪ Perform synchronous block write or read

SD card:

```

ui32Status = am_hal_sd_card_block_write_sync
    (&Card, START_BLK, ui32BlockCnt, (uint8_t *)ui8WrBuf);

ui32Status = am_hal_sd_card_block_read_sync
    (&Card, START_BLK, ui32BlockCnt, (uint8_t *)ui8RdBuf);

```

eMMC card:

```

ui32Status = am_hal_card_block_write_sync(&Card, START_BLK, ui32BlockCnt,
    (uint8_t *)ui8WrBuf);

if((ui32Stat & 0xffff) != 0)
{
    am_util_stdio_printf("Failed to write card.\n");
    return false;
}

ui32Status = am_hal_card_block_read_sync(&Card, START_BLK, ui32BlockCnt,
    (uint8_t *)ui8RdBuf);

if((ui32Stat & 0xffff) != 0 )
{
    am_util_stdio_printf("Failed to read card.\n");
    return false;
}

```

SDIO card:

The SDIO card should be set to a block size before block write or read. The block size is usually set as 256 bytes as follow:

```
//
// Set SDIO card function 1 block size
//
ui32BlkSize = 256;
ui32Status = am_hal_sdio_card_set_block_size (&Card, ui32FuncNum, ui32BlkSize);
```

The start address is the address of the FIFO or the RAM address in the SDIO Card. The function is usually selected as 1 (used for Wi-Fi card function) with the block write or read code as follow:

```
ui32Status = am_hal_sdio_card_multi_bytes_write_sync(
    &Card, ui32Func, Addr, (uint8_t *)tx_data, no_of_blocks, ui32BlkSize, true);
ui32Status = am_hal_sdio_card_multi_bytes_read_sync(
    &Card, ui32Func, Addr, (uint8_t *)read_buff, no_of_blocks, ui32BlkSize, true);
```

- **Perform asynchronous block write or block read**

An asynchronous block write or block read needs to define an SDIO interrupt service routine and a register callback function.

```
void am_sdio_isr(void)
{
    uint32_t ui32IntStatus;
    am_hal_sdhc_intr_status_get(pSdhcCardHost->pHandle, &ui32IntStatus, true);
    am_hal_sdhc_intr_status_clear(pSdhcCardHost->pHandle, &ui32IntStatus);
    am_hal_sdhc_interrupt_service(pSdhcCardHost->pHandle, &ui32IntStatus);
}
```

```
am_hal_card_register_evt_callback(&Card, am_hal_card_event_test_cb);
```

An eMMC asynchronous block write or read function is as follow:

```
ui32Status = am_hal_card_block_write_async(&Card, START_BLK, ui32BlockCnt,
                                           (uint8_t *)ui8WrBuf);

if((ui32Stat & 0xffff) != 0 )
{
    am_util_stdio_printf("Failed to write card.\n");
    return false;
}
```

```

ui32Status = am_hal_card_block_read_async(&Card, START_BLK, ui32BlockCnt,
                                         (uint8_t *)ui8RdBuf);

if((ui32Stat & 0xffff) != 0 )
{
    am_util_stdio_printf("Failed to read card.\n");
    return false;
}

```

An SD card asynchronous block write or read function is as follows:

```

bAsyncWriteIsDone = false;
ui32Status am_hal_sd_card_block_write_async
    (&Card, ui32StartBlock, ui32BlockCount, pui8WrBuf);

bAsyncReadIsDone = false;
ui32Status am_hal_sd_card_block_write_async
    (&Card, ui32StartBlock, ui32BlockCount, pui8WrBuf);

```

SDIO card asynchronous block write or read function is as follows:

```

bAsyncWriteIsDone = false;
am_hal_sdio_card_multi_bytes_write_async
    (&Card, 1, ui32MemAddr, (uint8_t *)ui8WrBuf, no_of_blocks, ui32BlkSize, true);

bAsyncReadIsDone = false;
am_hal_sdio_card_multi_bytes_read_async
    (&Card, 1, ui32MemAddr, (uint8_t *)ui8RdBuf, no_of_blocks, ui32BlkSize, true);

```

19.5.1 Calibration for 48 MHz SDR or DDR

Due to board layout and other factors, the highest specified data rate may require a timing calibration to assure robust communication. A calibration function is available which allows the user to determine the proper timing settings for the SDIO interface. Performing this calibration is recommended before using the block-oriented read or write functions.

In order to get more accurate Tx/Rx settings, a much larger number of sectors may be used for the calibration. Note that doing this may require more time for the calibration to determine optimum Tx/Rx timing settings.

There are 3 calibration APIs:

- am_hal_card_emmc_calibrate
- am_hal_sd_card_calibrate
- am_hal_sdio_card_calibrate

eMMC Card Calibration:

```
am_hal_card_emmc_calibrate
    (SDIO_TEST_MODULE, AM_HAL_HOST_UHS_DDR50, 48000000, AM_HAL_HOST_BUS_WIDTH_8,
    (uint8_t *)ui8WrBuf, START_BLK, 2, ui8TxRxDelays);
```

SD Card Calibration:

```
am_hal_sd_card_calibrate
    (SDIO_TEST_MODULE, AM_HAL_HOST_UHS_SDR50, 48000000, AM_HAL_HOST_BUS_WIDTH_4,
    (uint8_t *)ui8WrBuf, START_BLK, 10, ui8TxRxDelays, AM_HAL_CARD_PWR_CYCLE);
```

SDIO Card Calibration:

```
am_hal_sdio_card_calibrate
    (SDIO_TEST_MODULE, AM_HAL_HOST_UHS_SDR50, 48000000, AM_HAL_HOST_BUS_WIDTH_4,
    (uint8_t *)ui8WrBuf, START_BLK, 10, AM_HAL_BUS_VOLTAGE_1_8, ui8TxRxDelays,
    AM_HAL_CARD_PWR_CYCLE);
```

After calling the function, the Tx/Rx settings will take effective immediately. If the Tx/Rx settings are already known, then they can be installed directly by calling the below function.

```
uint8_t ui8TxRxDelays[2] = {4,5};
am_hal_card_host_set_txrx_delay(pSdhcCardHost, ui8TxRxDelays);
```

19.6 SD Card I/O Voltages

The Apollo5 has no way of interfacing directly with most SD Cards on the SDIO0 interface. Interfacing with SD Cards on the SDIO0 interface must be made through a level shifter. Furthermore, while the SDIO1 interface is able to support 3.3 V, there is no mechanism internal to the Apollo5 that is capable of switching the interface to 1.8 V.

20. Display Controller (DC)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

20.1 Functional Overview

The Display Controller (DC) is a dedicated hardware block allowing offload of display interface and control functions. The display controller is configured for 4-layer blending. The configuration register file controls the operation of the display controller.

Timing parameters are programmable and data are fetched for each layer by a dedicated DMA engine. Additional functions include:

- Gamma adjustment
- Dithering application

The DC can be configured to continuously refresh the display unit from the display buffer while the GPU accesses the memory. The refresh rate, resolution and color depth of the display determine the parameters of the DC.

The DC supports up to four layers sourced from different memory regions. Each layer can have separate color modes, alpha blending, and filtering attributes. The main control registers for each layer is the LAYERnMODE (n = 0-3) registers. Optionally, the layer can support scaling.

During the blending process, a translucent foreground color (current layer) with a background color (previous layer) are combined and a new blended color is produced. Foreground color translucency may range from completely transparent to completely opaque. If the foreground color is completely transparent, the blended color will be the background color and if the foreground color is completely opaque, the blended color will be the foreground color. When the translucency is somewhere in between, the blended color is computed as a weighted average of the foreground and background colors.

The DC supports palette color mapping and gamma correction per layer.

Dithering is the process of degrading the color image with a method that tries to produce better results than information truncation. Dithering is used to create the illusion of "color depth" in images with a limited color palette. In a dithered image, colors that are not available in the palette are approximated by a diffusion of colored pixels from within the available palette. The human eye perceives the diffusion as a mixture of the colors within it.

20.2 Software Support

The Display Controller is supported in both FreeRTOS and BareMetal (no operating system). A library of primitive graphics functions is available in pure ANSI C with no dependencies and is supported for FreeRTOS. The software package comes with a bit-accurate emulator and the NEMA®| PIX-Presso, a utility software for converting images to/from formats suitable for low power embedded devices.

20.2.1 Supported Software

- OS support
 - FreeRTOS support
 - No OS (BareMetal)
- Graphics API support
 - BareMetal Library in portable ANSI C
- Software Emulators and suites
 - DC API
 - NEMA®| PIX-Presso

NOTE

To ensure optimal performance, it is recommended to allocate framebuffers in internal SSRAM. If the framebuffer must be placed in external memory, such as PSRAM, the application must ensure that the available external memory bandwidth exceeds the display rendering throughput. Bandwidth limitations, whether caused by insufficient interface frequency or contention from concurrent memory access activities such as XIP, may result in display image corruption. As a mitigation, we have added support to manually control the DBIB_CSX signal. This allows the application to avoid the risk of display corruption, but comes at the cost of an increased CPU load.

20.3 Timing Generator

The DC continually refreshes the display unit from the display buffer while the GPU accesses the memory. The refresh rate, resolution and color depth of the display determine the parameters of the DC.

The Timing Generator is designed to be easily programmed using timing information in the same format as X11 Modeline definitions. Figure 92 shows how the parameters are defined.

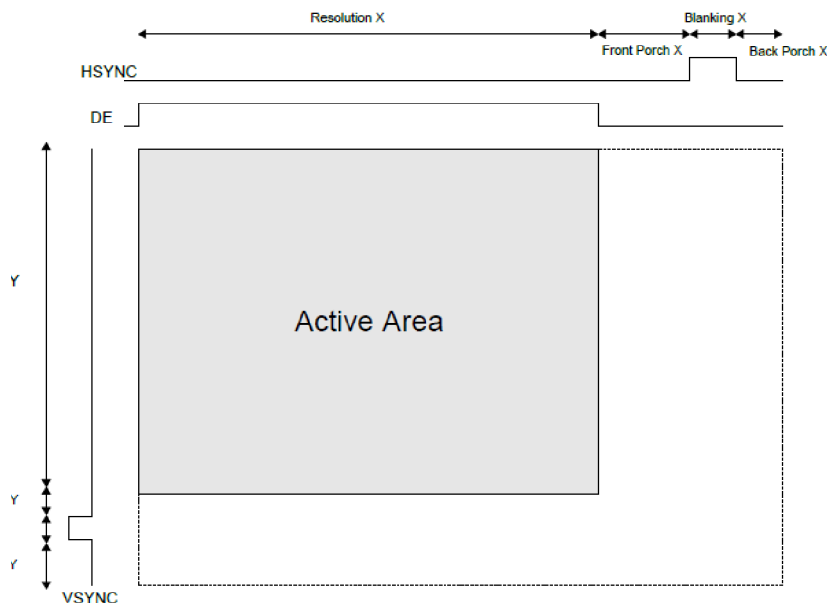


Figure 92. Display Controller Timing

The Timing Generator requires timing parameters as specified in the equations below for the vertical and horizontal sections, where the horizontal timing is pixel clock based while the vertical timing is line based.

The following equations determine the X11 Modeline horizontal in pixel clock cycles:

1. Front Porch (FRONTPORCHXY_FPCLKCYCLES field)

$$FRONTPORCHXY\_FPCLKCYCLES = ResolutionX + FrontPorchX$$

2. Blanking Period (BLANKINGXY_HSYNCPULSE field)

$$BLANKINGXY\_HSYNCPULSE = ResolutionX + FrontPorchX + BlankingX$$

3. Back Porch (BACKPORCHXY_BPCLKCYCLES field)

$$BACKPORCHXY\_BPCLKCYCLES = ResolutionX + FrontPorchX + BlankingX + BackPorchX$$

The following equations determine the X11 Modeline vertical in number of lines:

1. Front Porch (FRONTPORCHXY_FLINES field)

$$FRONTPORCHXY\_FLINES = ResolutionY + FrontPorchY$$

2. Blanking Period (BLANKINGXY_VSYNCLINES field)

$$BLANK\_XY\_VSYNCLINES = ResolutionY + FrontPorchY + BlankingY$$

3. Back Porch (BACKPORCHXY_BLINES field)

$$BACKPORCHXY\_BLINES = ResolutionY + FrontPorchY + BlankingY + BackPorchY$$

20.4 Layer Overlays

The Display Controller supports four layers¹ ($n = 0, 1, 2, 3$) sourced from different memory regions. Each layer can have separate color modes, alpha blending, and filtering attributes. The main control registers for each layer is the LAYERnMODE ($n = 0-3$) registers.

Layering starts with a 32-bit background color (RGBA), as shown in Figure 93, which is applied on the entire screen. If it is not needed the background register BGCOLOR field is set to 0.

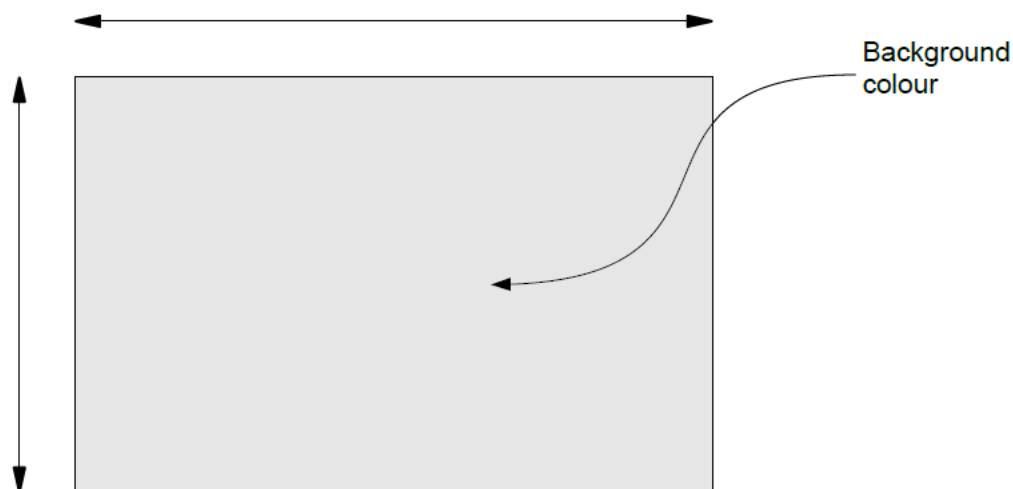


Figure 93. Display Controller RGBA Background Color

Layer 0 is applied on top of that with a requested blending method as shown in Figure 94. The background color can be used for blending and global values, such as alpha blending and palettes. If no layer is enabled (LAYERnMODE_LAYERnEN = 0), only the background color will be displayed.

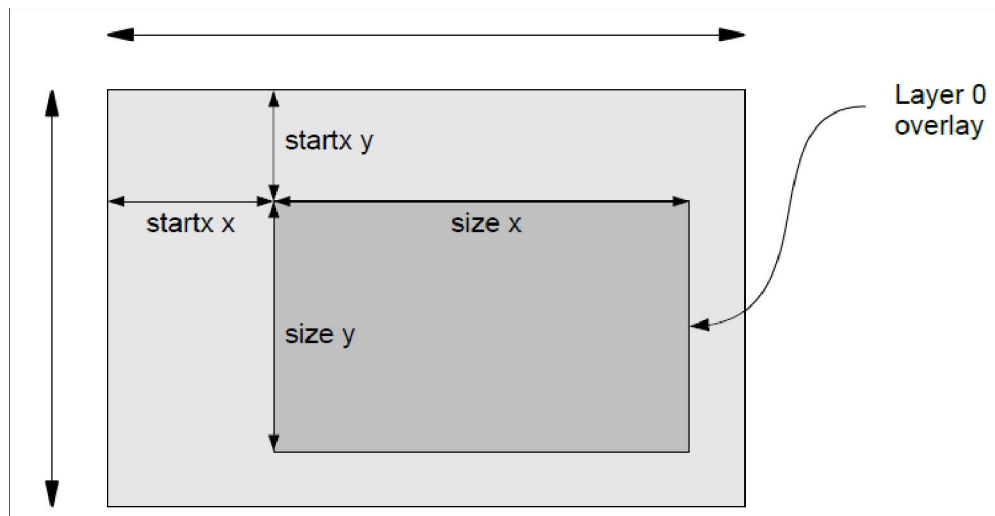


Figure 94. Display Controller First Layer

1. Apollo510 Lite Series SoCs supports one layer only.

The base frame address is set in the LAYERnADDR registers. The number of bytes from one row of pixels to the next row of pixels in memory is called stride. These bytes affect how the image is stored in memory and the LAYERnSTRIDE_LAYERnSTRIDEDIST bits are used to add the space between the frame lines.

For each layer, the start location (LAYERnSTARTXY), the visible size (LAYERnSIZEXY) and the resolution (LAYERnRESXY) are needed. For example, if the resolution of an image is 800x600 and the size of the required image for layer 0 is 400x300 at location (68, 32) then the value of the registers are shown in Table 103:

Parameter	Register	Register Field	Value
Resolution X	RESXY	XRES	800
Resolution Y	RESXY	YRES	600
Start coordinate X	LAYER0_STARTXY	LAYER0XOFF	68
Start coordinate Y	LAYER0_STARTXY	LAYER0YOFF	32
Pixel size in X coordinate	LAYER0_SIZEXY	LAYER0PIXSZEX	400
Pixel size in y coordinate	LAYER0_SIZEXY	LAYER0PIXSZEY	300

Table 103: Layer 0 Example

Each layer supports Alpha Blending (LAYERnMODE_LAYERnDBLEND). If no background is desired, the start x,y coordinates can be placed on (0,0) and the size (LAYERnSIZEXY) can be equal to resolution (registers LAYERnRESXY).

The layers can have different resolutions depending on the layer needs. In addition, each layer has a choice of color output modes that are set by the LAYERnMODE_LAYERnCOLMODE field. So, each layer can have different color formats. Figure 95 shows how the second layer is overlaid.

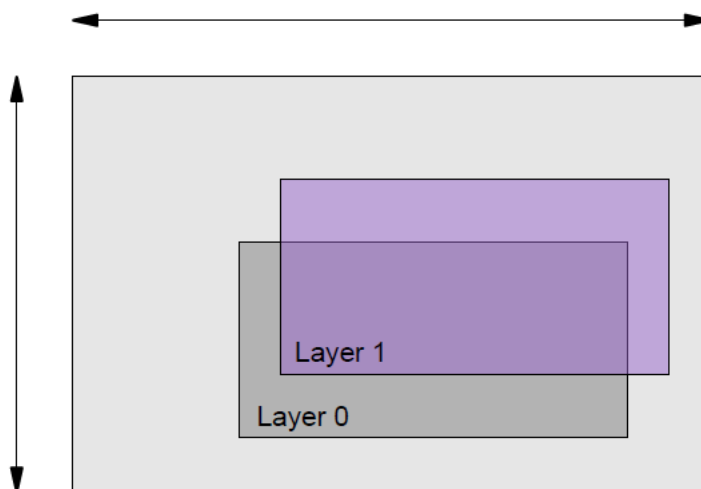


Figure 95. Display Controller Second Layer

NOTE

MIP screens generally require and utilize a function for flipping the screen 180 degrees. When setting both `flipx_en` and `flipy_en` to 1, the 180-degree flip does not work as expected for Multiple Partial Update mode of the MiP interface due to non-functionality of FlipY.

- For full-frame updates, `flipx_en` and `flipy_en` (and no flip) work for single or multiple layers.
- For single partial updates, `flipx_en` and no flip work for single and multiple layers. `Flipy_en` requires a software workaround.
- For multiple (up to 16) partial updates, `flipx_en` and no flip work for both single and multiple layers. `Flipy_en` does not work and there is no workaround.

20.5 Blending Modes

During blending process, a translucent foreground color (current layer) with a background color (previous layer) are combined and a new blended color is produced. Foreground color's translucency may range from completely transparent to completely opaque. If the foreground color is completely transparent, the blended color will be the background color and if the foreground color is completely opaque, the blended color will be the foreground color. When the translucency ranges in between, the blended color is computed as a weighted average of the foreground and background colors. Each layer is blended on top of the previous generated blended layer based on the following equation:

$$c = c_s * F_s + c_d * F_d$$

Table 104 shows the blending modes the Display Controller supports:

LAYERnDBLEND Field Value	Blending Mode	F_s	F_d
0000	blend black	0	0
0001	blend white	1	1
0010	blend alpha source	a_s	a_s
0011	blend alpha global	a_g	a_g
0100	blend alpha source and alpha global	$a_s * a_g$	$a_s * a_g$
0101	blend inverted source	$1 - a_s$	$1 - a_s$
0110	blend inverted global	$1 - a_g$	$1 - a_g$
0111	blend inverted source and inverted global	$1 - (a_s * a_g)$	$1 - (a_s * a_g)$
1010	blend alpha destination	a_d	a_d
1101	blend inverted destination	$1 - a_d$	$1 - a_d$

Table 104: Blending Modes

Blending is enabled when the LAYERnMODE_LAYERnDBLEND field is set with the respective values of Table 104. The destination blending function refers to the current layer while the source blending function refers to the previous layer.

20.6 Palette/Gamma Correction

Palette/Gamma correction is enabled when the MODE_GAMARAMPEN register bit is set. For Palette/Gamma correction, the Color Look Up Table (LnLUT) memory must be programmed. The Global Palette/Gamma register memory (GLLUT) is a 3x256x8 memory array that holds the RGB value for each of the 256 colors in the palette.

The same memories can be used to map RGB values to new RGB values to perform gamma correction. In this mode, the memory area containing the color data to the display will contain the LUT indexes instead of the actual color data. Then, the LUT maps these indexes to the color values contained in the palette registers before being sent to the LCD display. There are also optional LUT on a per layer basis.

Figure 96 shows that each RGB value is independently mapped to a new RGB value based on the content of the lookup tables. This also allows a 256-color palette as each gray value is mapped to a distinct color.

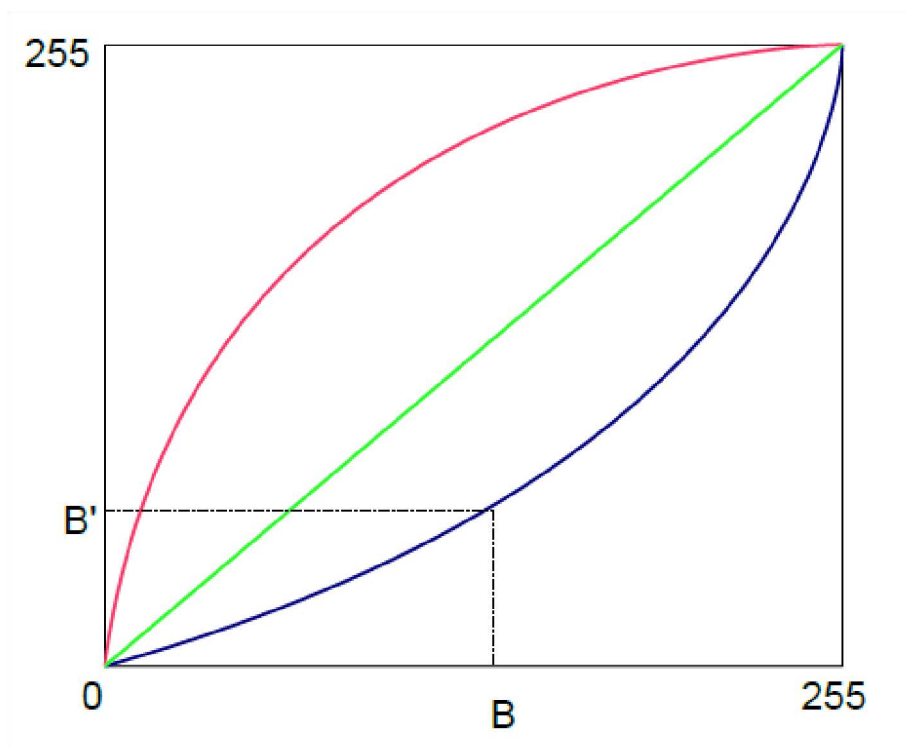


Figure 96. Gamma Correction of RGB Values

20.7 Dithering

Dithering is the process of degrading the color image with a method that tries to produce better results than information truncation.

Dithering is used to create the illusion of "color depth" in images with a limited color palette. In a dithered image, colors that are not available in the palette are approximated by a diffusion of colored pixels from within the available palette. The human eye perceives the diffusion as a mixture of the colors within it. Dithering is enabled by setting the `MODE_DITHEREN` bit.



a) RGBA8888

b) RGBA4444

c) RGBA4444 with dithering

Figure 97. Dithering on Limited Color Palette

20.8 Color Modes

The DC supports multiple color formats. The supported color formats are described in the following sections.

20.8.1 Input Formats

20.8.1.1 Binary BW 1-bit

Values range from 0 (white) to 1 (black)

BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW	BW

20.8.1.2 Grayscale 8-bit

Values range from 0 (black) to 255 (white)

L7	L6	L5	L4	L3	L2	L1	L0	L7	L6	L5	L4	L3	L2	L1	L0	L7	L6	L5	L4	L3	L2	L1	L0	L7	L6	L5	L4	L3	L2	L1	L0

20.8.1.3 Palette 256 8-bit (LUT8)

Values range from 0 to 255. These are looked-up from the Palette Color Table.

Pixel 0							
7	6	5	4	3	2	1	0
I7	I6	I5	I4	I3	I2	I1	I0
Byte 0							

20.8.1.4 RGBA 8888 32-bits

R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0	A7	A6	A5	A4	A3	A2	A1	A0

20.8.1.5 ARGB 8888 32-bit

A7	A6	A5	A4	A3	A2	A1	A0	R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0

20.8.1.6 ABGR 8888 32-bit

(Apollo510/Apollo510B only)

A7	A6	A5	A4	A3	A2	A1	A0	B7	B6	B5	B4	B3	B2	B1	B0	G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0

20.8.1.7 BGRA 8888 32-bit

(Apollo510/Apollo510B only)

B7	B6	B5	B4	B3	B2	B1	B0	G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	A7	A6	A5	A4	A3	A2	A1	A0

20.8.1.8 RGB 888 24-bit

R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0	R7	R6	R5	R4	R3	R2	R1	R0
G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0	R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0
B7	B6	B5	B4	B3	B2	B1	B0	R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0

20.8.1.9 RGBA 5551 16-bit

R4	R3	R2	R1	R0	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	A0	R4	R3	R2	R1	R0	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	A0

20.8.1.10 RGB 565 16-bit

R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0

20.8.1.11 RGB 332 8-bit

R2	R1	R0	G2	G1	G0	B1	B0	R2	R1	R0	G2	G1	G0	B1	B0	R2	R1	R0	G2	G1	G0	B1	B0	R2	R1	R0	G2	G1	G0	B1	B0

20.8.1.12 YUYV 32-bit 2-pixels

(Apollo510/Apollo510B only)

Y07	Y06	Y05	Y04	Y03	Y02	Y01	Y00	U7	U6	U5	U4	U3	U2	U1	U0	Y17	Y16	Y15	Y14	Y13	Y12	Y11	Y10	V7	V6	V5	V4	V3	V2	V1	V0

20.8.1.13 YUY2 32-bit 2-pixels

(Apollo510/Apollo510B only)

U7	U6	U5	U4	U3	U2	U1	U0	Y07	Y06	Y05	Y04	Y03	Y02	Y01	Y00	V7	V6	V5	V4	V3	V2	V1	V0	Y17	Y16	Y15	Y14	Y13	Y12	Y11	Y10

20.8.1.14 TSc4 16-pixels / 64-bits

TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4
TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4
TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4
TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4

20.8.1.15 TSc6 16-pixel / 96-bits

TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6

20.8.1.16 TSc6A 16-pixels with Alpha / 96-bits

TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A

20.8.1.17 TSc12 4-pixels / 96-bits

Valid Input and Output color format.

20.8.1.18 TSc12A 4-pixels with Alpha / 96-bits

Valid Input and Output color format

20.8.1.19 Custom formats

Additional color formats can be easily supported upon our client's requests. Please do not hesitate to contact us for more information.

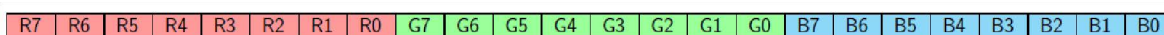
20.8.1.20 Color expansion

The internal color format on the DC is always on 8-bit format. Therefore lower order color formats are expanded to 8-bits. This is achieved by high-order bit replication. For example, a format with just 5 bits on a color component is replicated as:

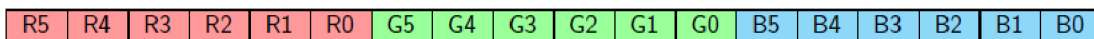
$$C[7:0] = \{C[4:0];C[4:2]\}$$

20.8.2 Output Formats

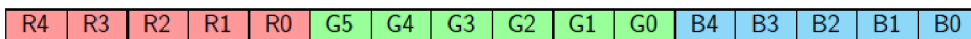
20.8.2.1 RGB 888 24-bit



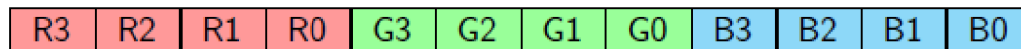
20.8.2.2 RGB 666 18-bit



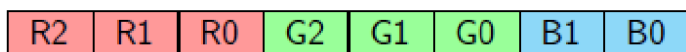
20.8.2.3 RGB 565 16-bit



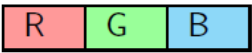
20.8.2.4 RGB 444 12-bit



20.8.2.5 RGB 332 8-bit



20.8.2.6 RGB 111 3-bit



20.8.2.7 Binary

Values range from 0 (black or white) to 1 (black or white)



20.9 TSc Framebuffer Decompression

TSc Framebuffer compression operates in screen blocks (4x4 pixel blocks) and, depending on the configuration, achieves TSC™4, TSC™6 and TSC™6a lossy, fixed-ratio compression.

- TSC™4 is a 6:1 compression (4bpp)
- TSC™6 is a 4:1 compression (6bpp)
- TSC™6a is a 5.33:1 compression (6bpp) with alpha channel
- TSC™12 is a 2:1 compression (12bpp)
- TSC™12a is a 2.67:1 compression (12bpp) with alpha channel

Compression is performed at run time using minimal hardware. Pixel data can be stored in the framebuffer in compressed form and decompressed in the DC. Figure 98 shows the TS compression operation. The output of the TSC™4 compression is 64 bits per 4x4 block of pixels, the output of the TSC™6 compression is 96 bits per 4x4 block of pixels and the output of TSC™12 compression is 48 bits per 2x2 block of pixels.

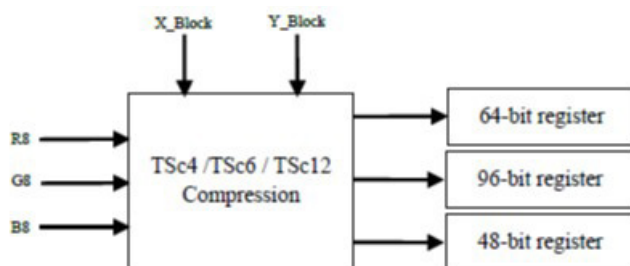


Figure 98. FTSC™4 / TSC™6 Framebuffer Compression Module

20.10 Display Formats

The Display Controller is enabled, and display data format and interface configuration are selected by setting fields in the MODE register. The DISPFMT field in this register sets one of several data formats. The DBITYPEBEN field enables the DBI Type-B interface used for the Serial/Dual/Quad SPI, and the DC400ACT field enables the Display Controller.

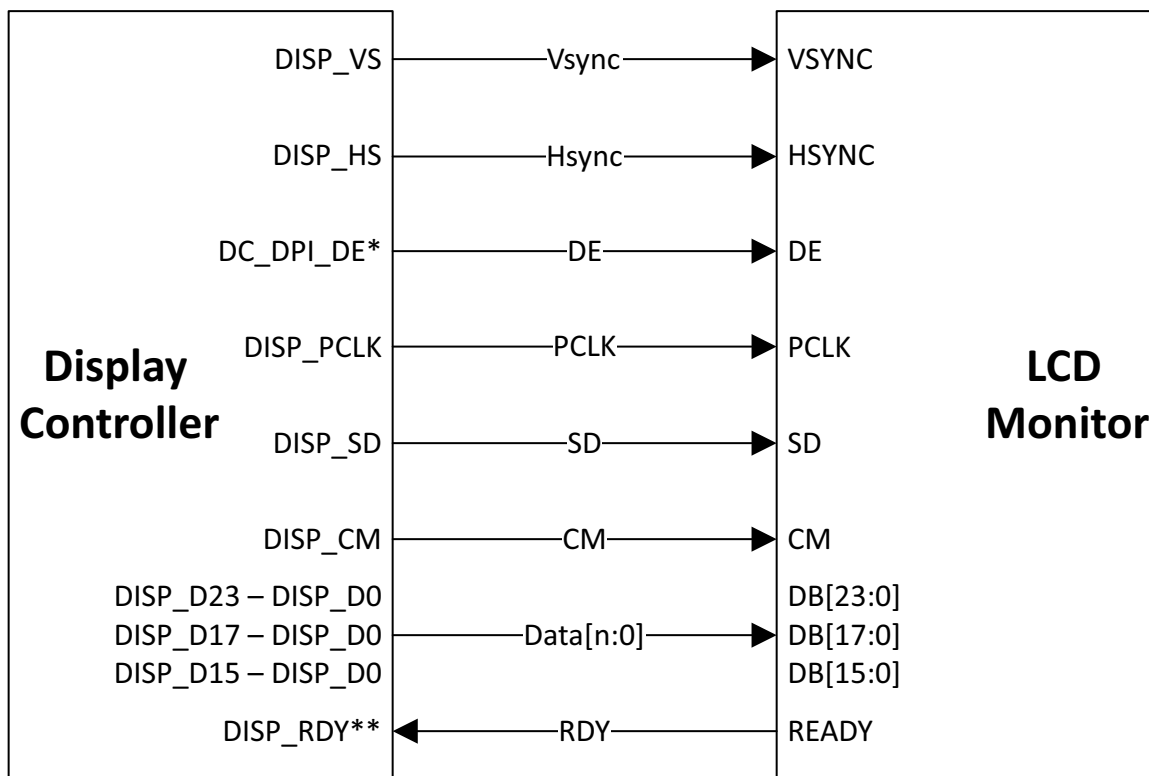
When the DBITYPEBEN field is set to enable DBI Type-B interface, the DBICFG register is used to activate the interface, specify 3- or 4-wire SPI, DualSPI or QuadSPI, and set data width, data word order and color format.

The following sections describe the display formats the DC supports.

To see an image from a frame buffer on a given TFT display at least one layer will need to be defined and set up using one or more LAYERn registers.

20.10.1 MIPI DPI-2 (Display Pixel Interface)

The DC supports MIPI DPI-2 protocol. Figure 99 shows how the DC is connected to an LCD monitor via the DPI-2 interface. The data interface can be 16 (DISP_D15-DISP_D0), 18 (DISP_D17-DISP_D0) or 24 (DISP_D23-DISP_D0) bits wide.



*Selected from NCESRC options available on all GPIO pins

**Configured as a standard GPIO input

Figure 99. DPI-2 Interface

The MODE register can be used to change the polarity of the Display Controller's timing output lines. The clocks speed are generated from the CLKCTRL register. The DC continuously reads data from the display

buffer and outputs it to the display with the display clock, vertical and horizontal synchronization signals, and enable signal configured to the specifications of the display.

Timing of the synchronization signals, polarity of the signals, and the required frame rate of the display are determined from the display specifications and translated to values to be programmed into the registers of the display controller.

Hsync is asserted on every scan line and Vsync is asserted on every frame. Pixel Clock can be defined as either positive polarity or negative polarity. The length of each pulse is programmable. A typical timing sequence of DPI-2 interface is depicted in Figure 100. The DC transfers data as 16-bit, 18-bit, or 24-bit words. The DC supports also shutdown (SD) and color mode (CM) signals. These signals can be configured via the GPIO register.

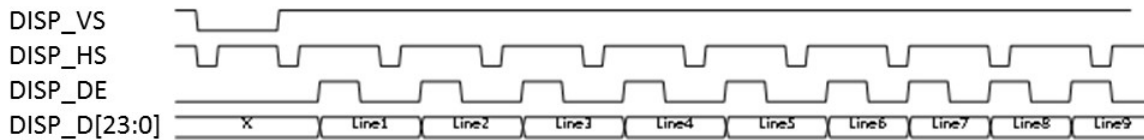


Figure 100. DPI-2 Interface Timing Sequence

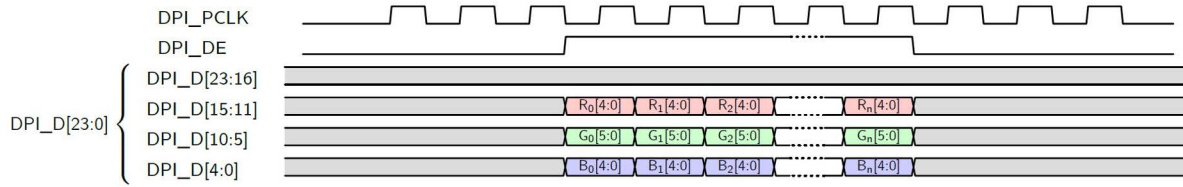
20.10.1.1 DPI-2 Output Formats

The following figures show the color coding DPI-2 Interface supports.

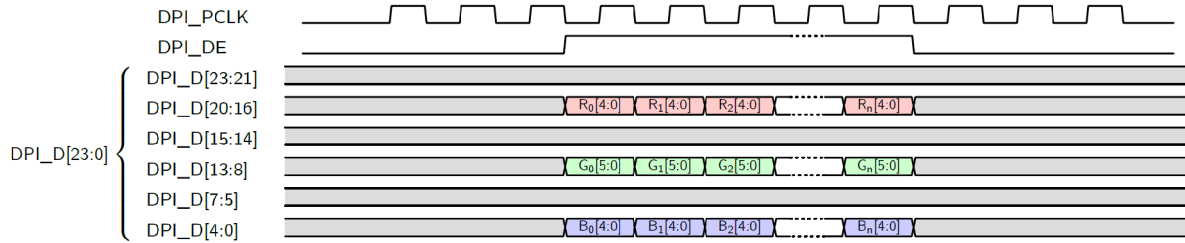
20.10.1.1.1 RGB 565 16-bit Configurations

For the RGB565 16-bit format, for the Red color, the MSB is R4 and the LSB is R0. For the Green primary color the MSB is G5 and the LSB is G0. For the Blue primary color, the MSB is B4 and the LSB is B0.

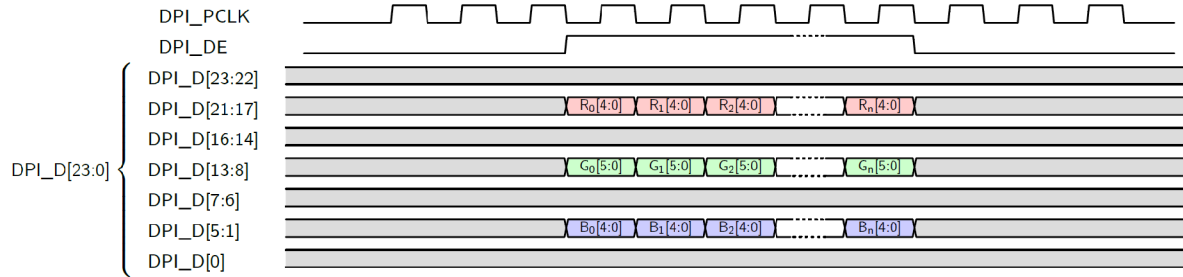
Configuration 1



Configuration 2



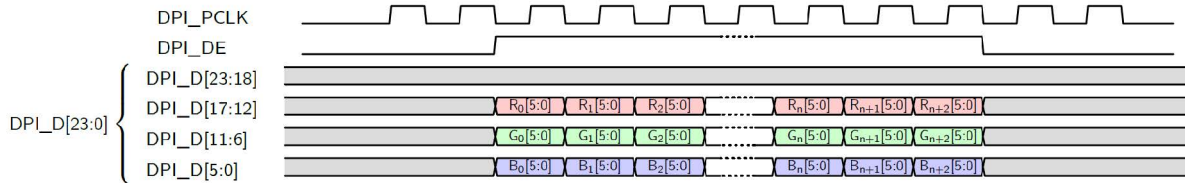
Configuration 3



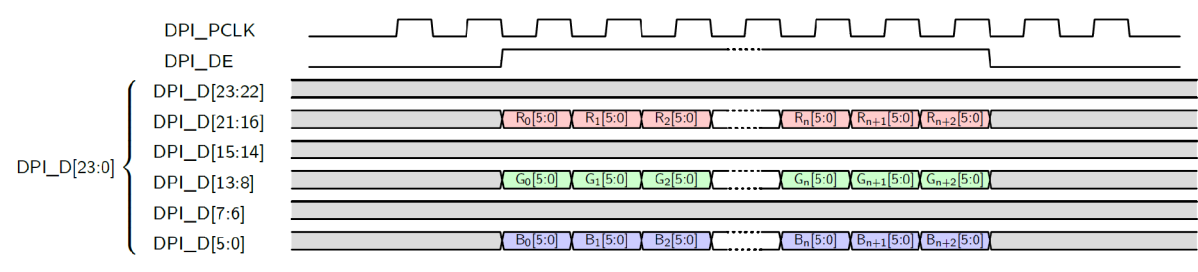
20.10.1.1.2 RGB 666 18-bit Configurations

For the RGB666 18-bit format, for the Red color, the MSB is R5 and the LSB is R0. For the Green primary color the MSB is G5 and the LSB is G0. For the Blue primary color, the MSB is B5 and the LSB is B0.

Configuration 1

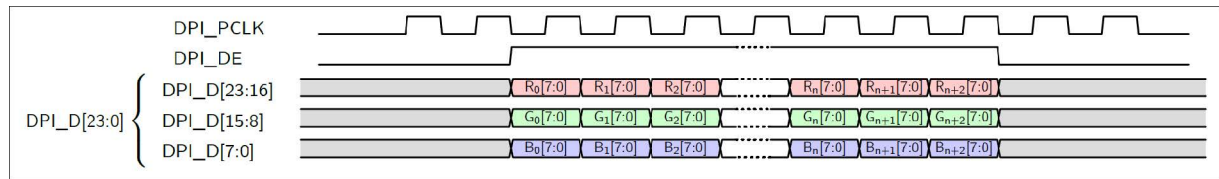


Configuration 2



20.10.1.1.3RGB 888 24-bit

For the RGB888 24-bit format, for the Red color, the MSB is R7 and the LSB is R0. For the Green primary color the MSB is G7 and the LSB is G0. For the Blue primary color, the MSB is B7 and the LSB is B0.



20.10.2 MIPI DBI-Type B (Display Bus Interface)

Internally the DC uses a MIPI DBI-Type B interface to the Display Serial Interface (DSI) module. The data width can be 8, 9 or 16 bits wide.

During a write cycle, the DC sends data to the DSI. WRX is driven from high to low then pulled back to high during the write cycle. The DC provides information during the write cycle while the DSI reads the information on the rising edge of WRX. DCX is driven low while command information is on the interface and is pulled high when data is present.

Figure 101 depicts the write sequence timing waveforms. Note that blue line indicates the high impedance state.

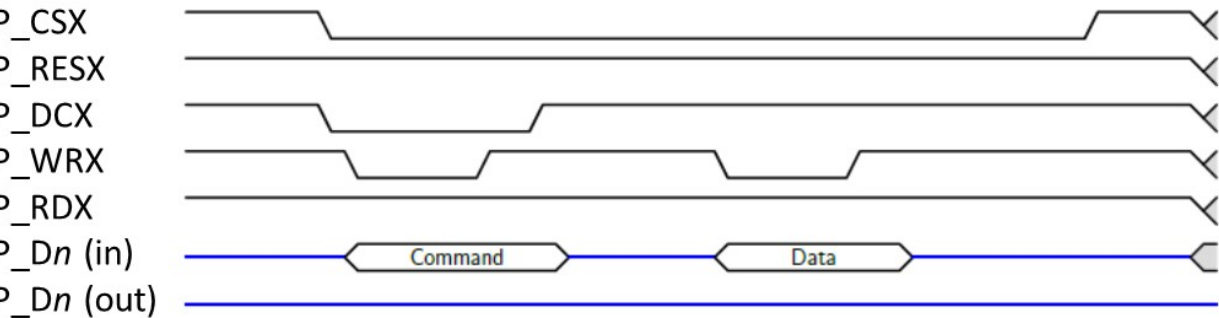


Figure 101. DBI-Type B Write Sequence

During a read cycle, the DC reads data from the DSI. RDX is driven from high to low then pulled back to high during the read cycle. The DSI provides information to the DC during the read cycle which is valid on the rising edge of RDX. DCX is driven high during the read cycle. Figure 102 depicts the read sequence timing waveforms. Note that blue line indicates the high impedance state.

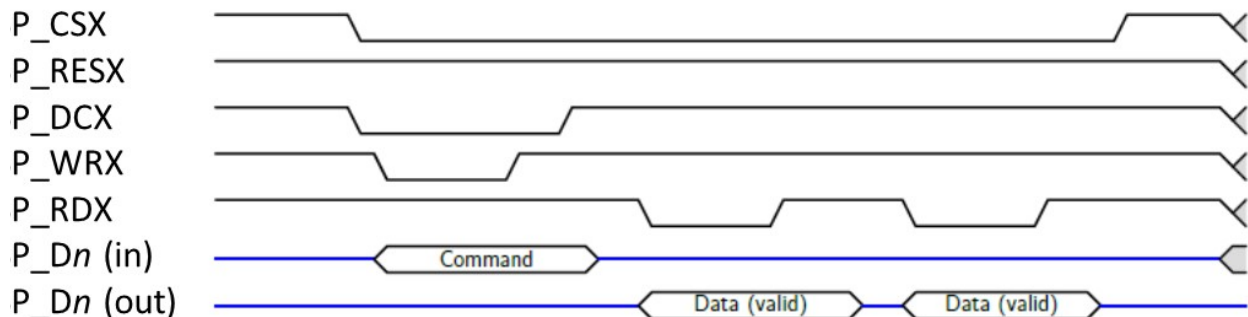


Figure 102. DBI-Type B Read Sequence

DBI-Type B interface is enabled through the MODE_DBTYPEBEN register bit. After the DBICFG_CSXCFG bit is set, then the DBICFG_CSXSET bit controls the setting of the CSX signal. Setting the DBICFG_RESXLOW bit sets the RESX signal. The output data width for this interface is controlled by the TYPEBWIDTH field, and the color format by the DBICOLORFMT field. The CLKCTRL register is set accordingly to activate the DBIB_CLK. The DBI_CMD register controls the read/write commands from/to the DBI-Type B interface, and register DBI_RDAT stores the value of data during read mode.

20.10.2.1 DBI-Type B Output Formats - 8bit Interface

The following figures show the color coding DBI-Type B Interface support when Data Bus (DBIB_DB) is 8 bits wide.

20.10.2.1.1 RGB332, 8bits/pixel, 256 colors

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit2	R1, Bit1	R1 Bit0	G1, Bit2	G1, Bit1	G1, Bit0	B1, Bit1	B1, Bit0
Pixel n+1	R2, Bit2	R2, Bit1	R2 Bit0	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit1	B2, Bit0
Pixel n+2	R3, Bit2	R3, Bit1	R3 Bit0	G3, Bit2	G3, Bit1	G3, Bit0	B3, Bit1	B3, Bit0

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit2, LSB = Bit0 for Red and Green data, MSB = Bit1, LSB = Bit0 for Blue data.

20.10.2.1.2 RGB444, 12bits/pixel 4,096 colors

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0
Pixel n / Pixel n+1	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0
Pixel n+1	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit3	B2, Bit2	B2, Bit1	B1, Bit0

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit3, LSB = Bit0 for Red, Green, and Blue data.

20.10.2.1.3 RGB565, 16bits/pixel 65,536 colors

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0	G1, Bit5	G1, Bit4	G1, Bit3
Pixel n	G1, Bit2	G1, Bit1	G1, Bit0	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0
Pixel n+1	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0	G2, Bit5	G2, Bit4	G2, Bit3
Pixel n+1	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B1, Bit0

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit5, LSB = Bit0 for Green data, MSB = Bit4, LSB = Bit0 for Red and Blue data.

20.10.2.1.4 RGB666, 18bits/pixel 262,144 colors

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0		
Pixel n	G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0		
Pixel n	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0		
Pixel n+1	R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0		
Pixel n+1	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0		
Pixel n+1	B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0		

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit5, LSB = Bit0 for Red, Green, and Blue data.

20.10.2.1.5 RGB888, 24bits/pixel 16,777,216 colors

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit7	R1, Bit6	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0
Pixel n	G1, Bit7	G1, Bit6	G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0
Pixel n	B1, Bit7	B1, Bit6	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0
Pixel n+1	R2, Bit7	R2, Bit6	R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0
Pixel n+1	G2, Bit7	G2, Bit6	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0
Pixel n+1	B2, Bit7	B2, Bit6	B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit7, LSB = Bit0 for Red, Green, and Blue data.

20.10.2.2 DBI-Type B Output Formats - 9-bit Interface

The following figures show the color coding DBI-Type B Interface supports when Data Bus (DBIB_DB) is 9 bits wide:

20.10.2.2.1 RGB666, 18bits/pixel 262,144 colors

Data	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0	G1, Bit5	G1, Bit4	G1, Bit3
Pixel n	G1, Bit2	G1, Bit1	G1, Bit0	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0
Pixel n+1	R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0	G2, Bit5	G2, Bit4	G2, Bit3
Pixel n+1	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0

The Data order is as follows, MSB = D8, LSB = D0. Picture Data is MSB = Bit5, LSB = Bit0 for Red, Green, and Blue data.

20.10.2.3 DBI-Type B Output Formats - 16bit Interface

The following figures show the color coding DBI-Type B Interface supports when Data Bus (DBIB_DB) is 16 bits wide: RGB332, 8bits/pixel, 256 colors

Data	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n / Pixel n+1	R1, Bit2	R1, Bit1	R1, Bit0	G1, Bit2	G1, Bit1	G1, Bit0	B1, Bit1	B1, Bit0	R2, Bit2	R2, Bit1	R2, Bit0	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit1	B2, Bit0
Pixel n+2 / Pixel n+3	R3, Bit2	R3, Bit1	R3, Bit0	G3, Bit2	G3, Bit1	G3, Bit0	B3, Bit1	B3, Bit0	R4, Bit2	R4, Bit1	R4, Bit0	G4, Bit2	G4, Bit1	G4, Bit0	B4, Bit1	B4, Bit0
Pixel n+4 / Pixel n+5	R5, Bit2	R5, Bit1	R5, Bit0	G5, Bit2	G5, Bit1	G5, Bit0	B5, Bit1	B5, Bit0	R6, Bit2	R6, Bit1	R6, Bit0	G6, Bit2	G6, Bit1	G6, Bit0	B6, Bit1	B6, Bit0

The Data order is as follows: MSB = D15, LSB = D0. Picture data is MSB = Bit2, LSB = Bit0 for Red and Green data, and MSB = Bit1, LSB = Bit0 for Blue data.

20.10.2.3.1 RGB444, 16bits/pixel 65,536 colors

Data	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n					R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0
Pixel n+1					R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0
Pixel n+2					R3, Bit3	R3, Bit2	R3, Bit1	R3, Bit0	G3, Bit3	G3, Bit2	G3, Bit1	G3, Bit0	B3, Bit3	B3, Bit2	B3, Bit1	B3, Bit0

The Data order is as follows, MSB = D15, LSB = D0. Pixel data is MSB = Bit3, LSB = Bit0 for Red, Green, and Blue data.

20.10.2.3.2 RGB565, 16bits/pixel 65,536 colors

Data	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit 4	R1, Bit 3	R1, Bit 2	R1, Bit 1	R1, Bit 0	G1, Bit 5	G1, Bit 4	G1, Bit 3	G1, Bit 2	G1, Bit 1	G1, Bit 0	B1, Bit 4	B1, Bit 3	B1, Bit 2	B1, Bit 1	B1, Bit 0
Pixel n+1	R2, Bit 4	R2, Bit 3	R2, Bit 2	R2, Bit 1	R2, Bit 0	G2, Bit 5	G2, Bit 4	G2, Bit 3	G2, Bit 2	G2, Bit 1	G2, Bit 0	B2, Bit 4	B2, Bit 3	B2, Bit 2	B2, Bit 1	B2, Bit 0
Pixel n+2	R3, Bit 4	R3, Bit 3	R3, Bit 2	R3, Bit 1	R3, Bit 0	G3, Bit 5	G3, Bit 4	G3, Bit 3	G3, Bit 2	G3, Bit 1	G3, Bit 0	B3, Bit 4	B3, Bit 3	B3, Bit 2	B3, Bit 1	B3, Bit 0

The Data order is as follows, MSB = D15, LSB = D0. Pixel data is MSB = Bit5, LSB = Bit0 for Green data and MSB = Bit4, LSB = Bit0 for Red and Blue data.

20.10.2.3.3 RGB666, 18bits/pixel 262,144 colors - Option 1

Data	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0			G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0		
Pixel n / Pixel n+1	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0			R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0		
Pixel n+1	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0			B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0		

The Data order is as follows, MSB = D15, LSB = D0. Pixel data is MSB = Bit5, LSB = Bit0 for Red, Green, and Blue data.

20.10.2.3.4 RGB666, 18bits/pixel 262,144 colors - Option 2

Data	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n									R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0		
Pixel n	G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0			B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0		
Pixel n+1									R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0		
Pixel n+1	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0			B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0		

The Data order is as follows, MSB = D15, LSB = D0. Pixel data is MSB = Bit5, LSB = Bit0 for Red, Green, and Blue data.

20.10.2.3.5 RGB888, 24bits/pixel 16,777,216 colors - Option 1

Data	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit7	R1, Bit6	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0	G1, Bit7	G1, Bit6	G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0
Pixel n / Pixel n+1	B1, Bit7	B1, Bit6	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0	R2, Bit7	R2, Bit6	R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0
Pixel n+1	G2, Bit7	G2, Bit6	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit7	B2, Bit6	B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0

The Data order is as follows, MSB = D15, LSB = D0. Pixel data is MSB = Bit7, LSB = Bit0 for Red, Green, and Blue data.

20.10.2.3.6 RGB888, 24bits/pixel 16,777,216 colors - Option 2

Data	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n									R1, Bit7	R1, Bit6	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0
Pixel n	G1, Bit7	G1, Bit6	G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0	B1, Bit7	B1, Bit6	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0
Pixel n+1									R2, Bit7	R2, Bit6	R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0
Pixel n+1	G2, Bit7	G2, Bit6	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit7	B2, Bit6	B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0

The Data order is as follows, MSB = D15, LSB = D0. Pixel data is MSB = Bit7, LSB = Bit0 for Red, Green, and Blue data.

20.10.3 SPI, 3-/4-wire Serial Peripheral Interface

The Display Controller supports SPI serial interface with three or four distinct signals. The 3-wire (9-bit data word) serial interface uses the DC_SPI_CS_N (chip select), DISP_SPI_SCK (serial clock), and DISP_SPI_SD (bidirectional serial data) or DISP_SPI_SDO (serial data out only). The 9-bit data word consists of 1-bit data/command indicator and 8-bit data word.

The 4-wire (8-bit data word) serial interface uses DC_SPI_CS_N (chip select), DISP_SPI_SCK (serial clock), DISP_SPI_SD (bidirectional serial data) or DISP_SPI_SDO (serial data out only), and one additional signal DISP_SPI_DCX (data/command indicator). The DISP_SPI_SD or DISP_SPI_SDO signal is regarded as a command when DISP_SPI_DCX is low and as data when DISP_SPI_DCX is high.

The chip select can be assigned to any GPIO0-GPIO104 via the PINCFGn_NCESRCn field. See the “Implementing Display Controller Connections” section in the GPIO chapter for pad configuration information for these SPI signals.

The following figures shows how the DC is connected to an LCD monitor using the aforementioned methods.

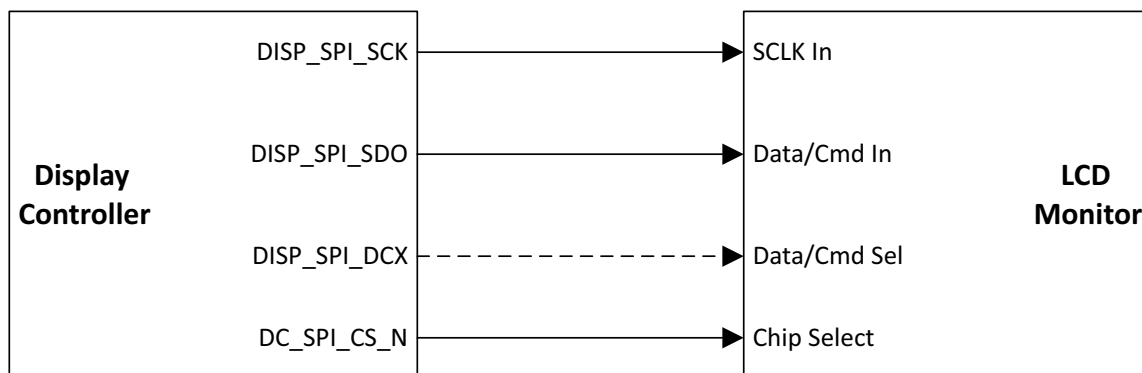


Figure 103. SPI 3- or 4-wire Interface (Dashed Line Used in 4-wire) - Data Out Only

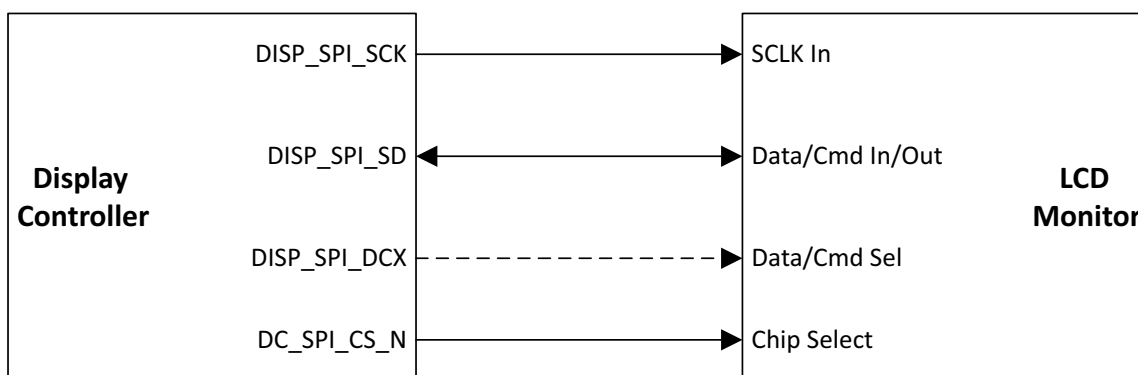


Figure 104. SPI 3- or 4-wire Interface (Dashed Line Used in 4-wire) - Bidirectional Data

NOTE

Because of pad function limitation, each of the MSPI2 and the Display SPI signals is available on only one pad, and since signals for each of these two interfaces share the same pads, only one of MSPI2 or Display SPI can be used at a time.

To enable the DC to use one of these interfaces, the DC_MODE_DBITYPEBEN bit must be set. Then the selection of SPI3 or SPI4 is done by setting the DC_DBICFG_SPI3 or DC_DBICFG_SPI4 bit.

After the LCD is initialized and during the write mode, the Display Controller sends commands and data to the LCD. The 3-wire serial data packet (DISP_SPI_SD) contains a control bit and a transmission byte, 9 bits in total. In 4-wire serial interface, data packet (DISP_SPI_SD) contains only the transmission byte (8 bits) and the control bit is transferred by the DISP_SPI_DCX signal separately. The MSB is transmitted first.

The CS signal, the GPIO configured as GPIO_PINCFGn_NCESRCn = DC_SPI_CS_N, is configurable. It can be set high or low to indicate the start of the data transmission. Data can be sampled either by the falling or the rising edge of the DISP_SPI_SCK depending on the configuration. Also, the clock polarity is configurable (whether the clock starts at high or low edge). If the CS pin remains high/low (depending on the configuration) after the last bit of DISP_SPI_SD signal, the serial interface expects the DISP_SPI_DCX

bit (3-wire serial interface) or D7 (4-wire serial interface) of the next byte at the next rising/falling edge of DISP_SPI_SCK. The serial clock (DISP_SPI_SCK) is idle when no communication is taking place.

Figure 105 and Figure 106 depict the corresponding timing waveforms of the 3-wire and 4-wire serial interface during write mode respectively. In these examples the falling edge of SPI_CS indicates the start of data transmission and data are sampled at the rising edge of the serial clock. The SPI_DC signal in Figure 106 when low, indicates a command and when high, indicates data.

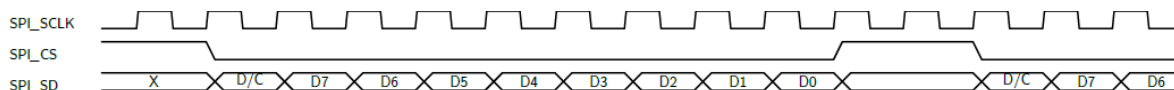


Figure 105. SPI 3-wire Serial Write Transmission

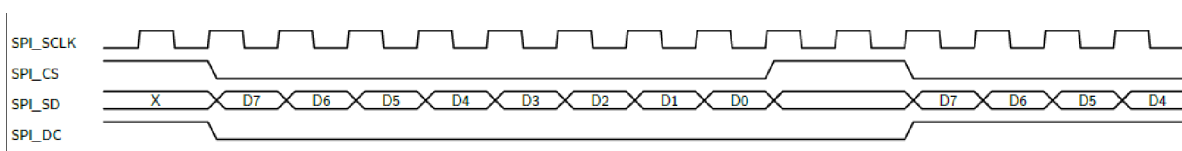


Figure 106. SPI 4-wire Serial Write Transmission

In case of SPI LCD displays, command and data are bind. The addresses are generated and the address bits can be reversed depending on the type of the LCD display. These displays use the 4-wire serial interface with DC signal not being used. Figure 107 shows one line transmission data.

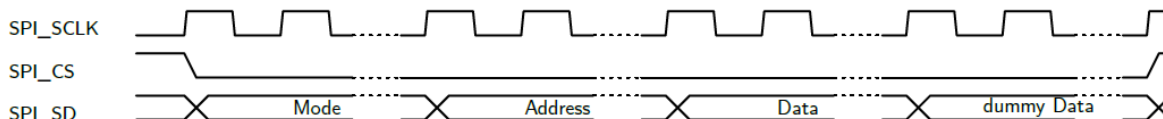


Figure 107. Single Line Update Mode

During the mode select period, 6 bits are transmitted. During the address select period, 10 bits of the line address are transmitted. The data are transited soon after and the transmission duration depends on the RGB format. After the end of the line, 16 clock cycles are needed. Dummy data are sent during that period and can be either High or Low. Figure 108 shows multiple lines transmission data.

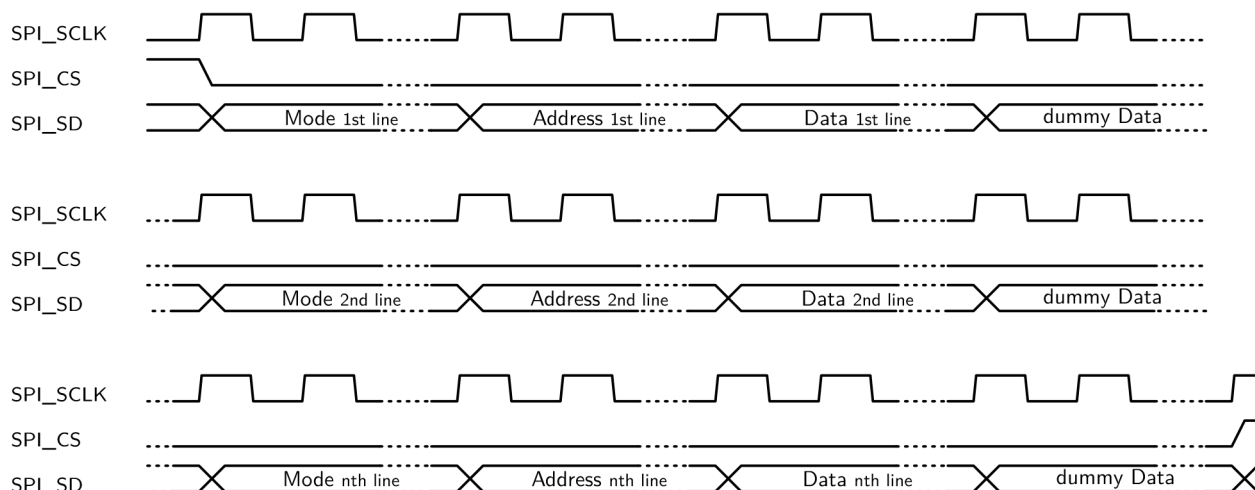


Figure 108. Multiple Lines Update Mode

As with the transmission of only one line, the mode select period needs 6 clock cycles and the address select period needs 10 clock cycles. The duration of the transmitted data depends on the RGB format and the number of lines of each frame from the screen resolution. The dummy data transmission, needed after the end of each line, takes 6 clock cycles. Dummy data can be either High or Low. At the end of the frame (nth line) the duration of the dummy data transmission is 16 clock cycles.

20.10.3.1 SPI Output formats

Through the SPI interface, the DC supports the following output color formats:

- Binary (Black & White) 1 bit/pixel
- RGB111 3 bits/pixel
- RGB332 8 bits/pixel
- RGB565 16 bits/pixel
- RGB666 18 bits/pixel
- RGB888 24 bits/pixel

20.10.3.1.1 Configuration 0xC1 (RGB111 - Option 0)

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n / Pixel n+1			R1	G1	B1	R2	G2	B2

The Data order is as follows, MSB = D7, LSB = D0.

20.10.3.1.2 Configuration 0xC9 (RGB111 - Option 1)

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n / Pixel n+1		R1	G1	B1		R2	G2	B2

The Data order is as follows, MSB = D7, LSB = D0.

20.10.3.1.3 Configuration 0xD1 (RGB111 - Option 2)

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n / Pixel n+1	R1	G1	B1		R2	G2	B2	

The Data order is as follows, MSB = D7, LSB = D0.

20.10.3.1.4 Configuration 0xE1 (RGB111 - Option 4)

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n / Pixel n+1 / Pixel n+2	R1	G1	B1	R2	G2	B2	R3	G3

The Data order is as follows, MSB = D7, LSB = D0.

20.10.3.1.5 Configuration 0xC2 (RGB332 - Option 0)

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit2	R1, Bit1	R1 Bit0	G1, Bit2	G1, Bit1	G1, Bit0	B1, Bit1	B1, Bit0
Pixel n+1	R2, Bit2	R2, Bit1	R2 Bit0	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit1	B2, Bit0
Pixel n+2	R3, Bit2	R3, Bit1	R3 Bit0	G3, Bit2	G3, Bit1	G3, Bit0	B3, Bit1	B3, Bit0

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit2, LSB = Bit0 for Red and Green data, MSB = Bit1, LSB = Bit0 for Blue data.

20.10.3.1.6 Configuration 0xC3 (RGB444 - Option 0)

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0
Pixel n / Pixel n+1	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0
Pixel n+1	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit3	B2, Bit2	B2, Bit1	B1, Bit0

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit3, LSB = Bit0 for Red, Green, and Blue data.

20.10.3.1.7 Configuration 0xC5 (RGB565 - Option 0)

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0	G1, Bit5	G1, Bit4	G1, Bit3
Pixel n	G1, Bit2	G1, Bit1	G1, Bit0	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0
Pixel n+1	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0	G2, Bit5	G2, Bit4	G2, Bit3
Pixel n+1	G2, Bit2	G2, Bit1	G2, Bit0	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B1, Bit0

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit5, LSB = Bit0 for Green data, MSB = Bit4, LSB = Bit0 for Red and Blue data.

20.10.3.1.8 RGB666 (0xC6 - Option 0)

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0		
Pixel n	G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0		
Pixel n	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0		
Pixel n+1	R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0		
Pixel n+1	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0		
Pixel n+1	B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0		

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit5, LSB = Bit0 for Red, Green, and Blue data.

20.10.3.1.9 RGB888 (0xC7 - Option 0)

Data	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	R1, Bit7	R1, Bit6	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2	R1, Bit1	R1, Bit0
Pixel n	G1, Bit7	G1, Bit6	G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2	G1, Bit1	G1, Bit0
Pixel n	B1, Bit7	B1, Bit6	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2	B1, Bit1	B1, Bit0
Pixel n+1	R2, Bit7	R2, Bit6	R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2	R2, Bit1	R2, Bit0
Pixel n+1	G2, Bit7	G2, Bit6	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2	G2, Bit1	G2, Bit0
Pixel n+1	B2, Bit7	B2, Bit6	B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2	B2, Bit1	B2, Bit0

The Data order is as follows, MSB = D7, LSB = D0. Picture Data is MSB = Bit7, LSB = Bit0 for Red, Green, and Blue data.

20.10.4 DualSPI and QuadSPI Interface

The Display Controller supports DualSPI and QuadSPI interface using DC_QSPI_CS_N (chip select), DISP_QSPI_SCK (serial clock), and either two or four bidirectional serial data lines. For QuadSPI, the DISP_QSPI_D3 - DISP_QSPI_D0 lines are used, and for DualSPI, the DISP_SPI_DCX and DISP_SPI_SD lines are used. For this section, all other information is applicable in either DualSPI or QuadSPI mode.

The chip select can be assigned to any GPIO0-GPIO104 via the PINCFGn_NCESRCn field. See the “Implementing Display Controller Connections” section in the GPIO chapter for pad configuration information for these SPI signals. Figure 109 and Figure 110 show how the DC is connected to an LCD monitor in either QuadSPI or DualSPI mode using the aforementioned signals.

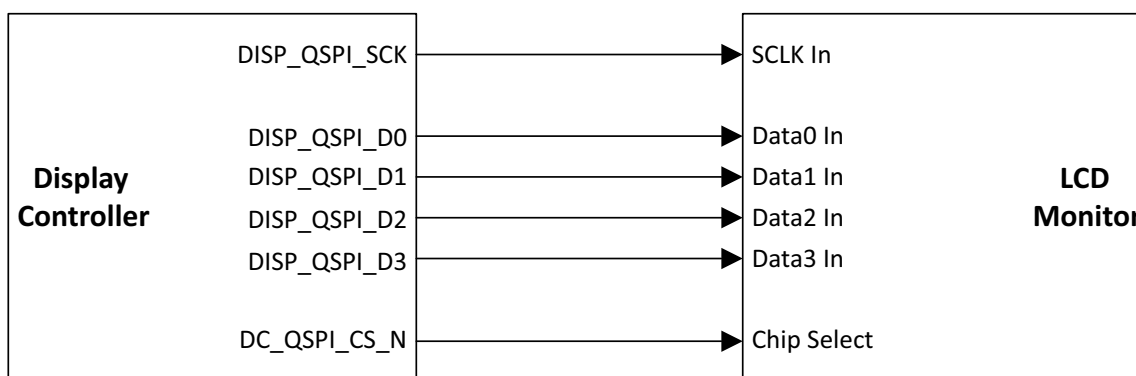


Figure 109. QuadSPI Interface

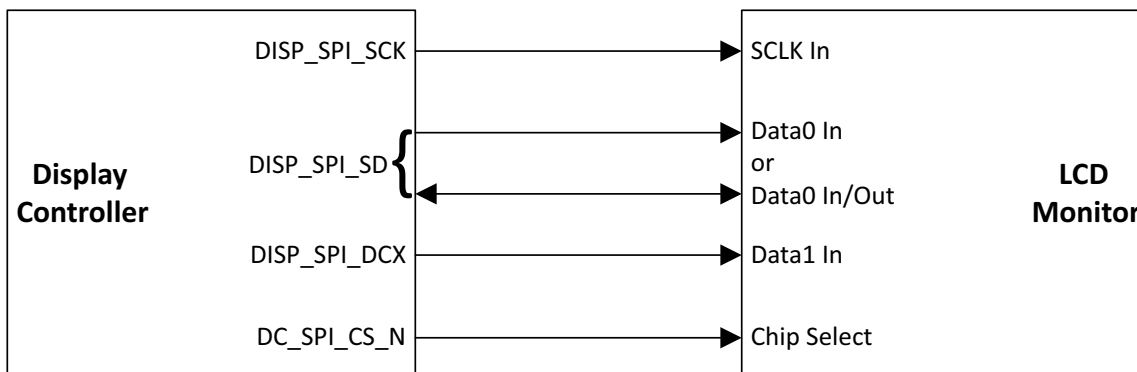


Figure 110. DualSPI Interface

To enable the DC to use one of these interfaces, the `DC_MODE_DBITYPEBEN` bit must be set. After the LCD is initialized and during the write mode, the Display Controller sends commands and data to the LCD. The data packet transmitted on the data lines contains a control bit and a transmission byte, 9 bits in total. The MSB is transmitted first.

The CS signal, the GPIO configured as `GPIO_PINCFGn_NCESRCn = DC_QSPI_CS_N` for QuadSPI or `DC_SPI_CS_N` for DualSPI, is configurable. It can be set high or low to indicate the start of the data transmission. Data can be sampled either by the falling or the rising edge of the clock depending on the configuration. Also, the clock polarity is configurable (whether the clock starts at high or low edge). If the CS pin remains high/low (depending on the configuration) after the last bit of the data signal, the serial interface expects D7 of the next byte at the next rising/falling edge of the clock. The serial clock is idle when no communication is occurring.

The relative timing of the clock, data and chip select signals generally conform to the 3-wire SPI interface timing as shown and described in “SPI, 3-/4-wire Serial Peripheral Interface” on page 319.

20.10.5 MIP RGB Interface

For a Memory-In-Pixel (MIP) RGB display, a pixel has 2-bit static random-access memory (SRAM) to store the image data for each sub-pixel and it is capable of displaying 64 colors.

Table 106 shows the interface for a MIP RGB display, signal description and the equivalent pins that are connected to the DC when this module is enabled:

Table 105: Pinout of MIP RGB LCD and Connection to the Equivalent DC Pins

Display Signal	Direction	Description	Display Controller Pin	Direction
VST	in	Vertical start signal (JDI)	DISP_VST	out
VCK	in	Vertical shift clock (JDI)	DISP_VCK	out
ENB	in	Write enable signal for the pixel memory (JDI)	DISP_ENB	out
HST	in	Horizontal start signal (JDI)	DISP_HST	out
HCK	in	Horizontal shift clock (JDI)	DISP_HCK	out
R1	in	Red signal for even pixels	DISP_R1	out
R2	in	Even red data (JDI)	DISP_R2	out
G1	in	Green signal for even pixels	DISP_G1	out
G2	in	Even green data (JDI)	DISP_G2	out
B1	in	Blue signal for even pixels	DISP_B1	out
B2	in	Even blue data (JDI)	DISP_B2	out
RST	In	Reset signal for horizontal and vertical drivers (JDI)	DISP_RST	out

Table 106: Pinout of MIP LCD and Connection to the Equivalent DC Pins

Figure 111 depicts the timing waveforms for Vertical Update mode. LPB is the Large Pixel Bit whereas SPB is the Small Pixel Bit. Figure 112 depicts the corresponding timing waveforms for Horizontal Update mode. The DC first sends the MSB and then the LSB for this type of displays. Scan doubling must be enabled.

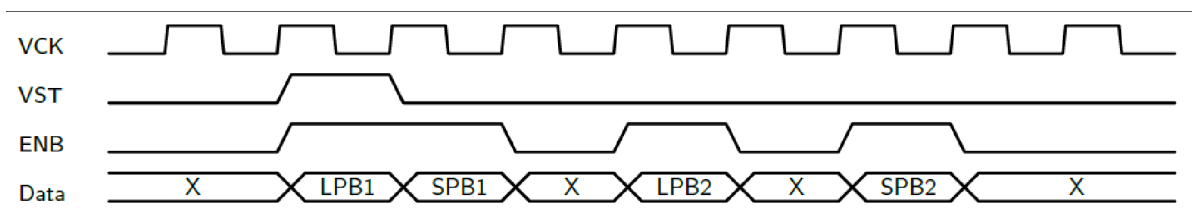


Figure 111. Vertical Mode

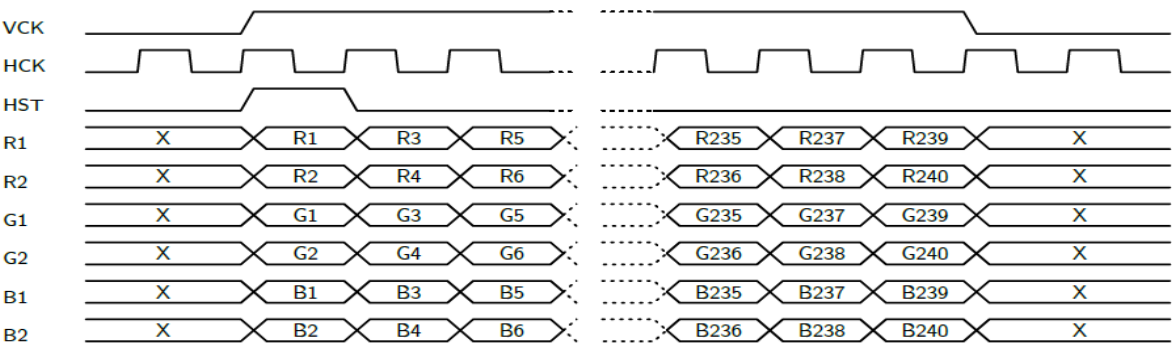


Figure 112. Horizontal Mode

20.10.6 Serial Formats

The Display Controller supports the following serial formats:

- Serial 3-beat RGB
- Serial 4-beat RGBA
- Two phase serial 12-bit RGB
- BT.656
- MIP

The serial formats are enabled through the MODE register's YUYVEN and DISPFMT fields, which configure the display format. These serial modes have specific timing requirements and the output is driven on only certain pins. The used pins are DISP_PCLK for clock, DISP_DE for data enable and DISP_D[23:0] for data. Certain pins of DISP_D[23:0] are used depending on the serial interface.

20.10.6.1 Serial 3-beat RGB

For the Serial 3-beat RGB format, the DISP_D[15:14] and DISP_D[7:2] are used where pins DISP_D[15:14] contain the two LSBs of each color and pin DISP_D[7] contains the MSB of each color. Pins DISP_D[23:16], DISP_D[13:8] and DISP_D[1:0] are unused.

Data	D23	D22	D21	D20	D19	D18	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n									R1, Bit1	R1, Bit0							R1, Bit7	R1, Bit6	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2		
									G1, Bit1	G1, Bit0							G1, Bit7	G1, Bit6	G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2		
									B1, Bit1	B1, Bit0							B1, Bit7	B1, Bit6	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2		
Pixel n+1									R2, Bit1	R2, Bit0							R2, Bit7	R2, Bit6	R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2		
									G2, Bit1	G2, Bit0							G2, Bit7	G2, Bit6	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2		
									B2, Bit1	B2, Bit0							B2, Bit7	B2, Bit6	B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2		

One pixel needs three clock cycles to be transmitted as shown in Figure 113. R1, G1 and B1 are the red, green and blue colors of pixel 1, R2, G2 and B2 are the red, green and blue colors of pixel 2 and so on.

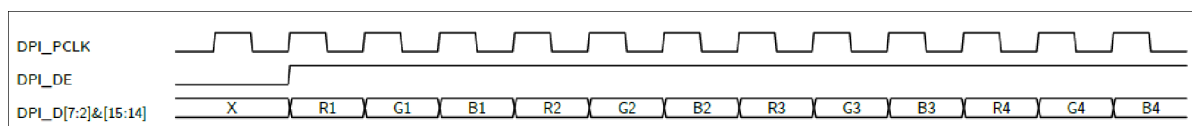
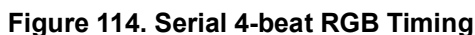


Figure 113. Serial 3-beat RGB Timing

20.10.6.2 Serial 4-beat RGB

For the Serial 4-beat RGB format, the DISP_D[15:14] and DISP_D[7:2] are used. Pins DISP_D[15:14] contain the two LSBs of each color and the two LSBs of the alpha value. Pin DISP_D[7] contains the MSB of each color and the MSB of the alpha value. Pins DISP_D[23:16], DISP_D[13:8] and DISP_D[1:0] are unused.

Data	D23	D22	D21	D20	D19	D18	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n									R1, Bit1	R1, Bit0							R1, Bit7	R1, Bit6	R1, Bit5	R1, Bit4	R1, Bit3	R1, Bit2		
									G1, Bit1	G1, Bit0							G1, Bit7	G1, Bit6	G1, Bit5	G1, Bit4	G1, Bit3	G1, Bit2		
									B1, Bit1	B1, Bit0							B1, Bit7	B1, Bit6	B1, Bit5	B1, Bit4	B1, Bit3	B1, Bit2		
									A1, Bit1	A1, Bit0							A1, Bit7	A1, Bit6	A1, Bit5	A1, Bit4	A1, Bit3	A1, Bit2		
Pixel n+1									R2, Bit1	R2, Bit0							R2, Bit7	R2, Bit6	R2, Bit5	R2, Bit4	R2, Bit3	R2, Bit2		
									G2, Bit1	G2, Bit0							G2, Bit7	G2, Bit6	G2, Bit5	G2, Bit4	G2, Bit3	G2, Bit2		
									B2, Bit1	B2, Bit0							B2, Bit7	B2, Bit6	B2, Bit5	B2, Bit4	B2, Bit3	B2, Bit2		
									A2, Bit1	A2, Bit0							A2, Bit7	A2, Bit6	A2, Bit5	A2, Bit4	A2, Bit3	A2, Bit2		



For the Two phase serial 12-bit RGB format, the DISP_D[23:20], DISP_D[15:12] and DISP_D[7:4] are used. Pins DISP_D[19:16], DISP_D[11:8] and DISP_D[3:0] are unused. In this serial mode, one pixel needs two clock cycles (two-phase) to be transmitted.

	Data	D23	D22	D21	D20	D19	D18	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Pixel n	phase 1	R1, Bsr7	R1, Bsr6	R1, Bsr5	R1, Bsr4					R1, Bsr3	R1, Bsr2	R1, Bsr1	R1, Bsr0					G1, Bsr7	G1, Bsr6	G1, Bsr5	G1, Bsr4				
	phase 2	G1, Bsr3	G1, Bsr2	G1, Bsr1	G1, Bsr0					R1, Bsr3	R1, Bsr2	R1, Bsr1	R1, Bsr0					G1, Bsr7	G1, Bsr6	G1, Bsr5	G1, Bsr4				
Pixel n+1	phase 1	R2, Bsr7	R2, Bsr6	R2, Bsr5	R2, Bsr4					R2, Bsr3	R2, Bsr2	R2, Bsr1	R2, Bsr0					G2, Bsr7	G2, Bsr6	G2, Bsr5	G2, Bsr4				
	phase 2	G2, Bsr3	G2, Bsr2	G2, Bsr1	G2, Bsr0					R2, Bsr3	R2, Bsr2	R2, Bsr1	R2, Bsr0					G2, Bsr7	G2, Bsr6	G2, Bsr5	G2, Bsr4				

Figure 115. Two-Phase Serial 12-bit RGB

[illegible]

20.10.6.5 Grayscale 8-bit

Values range from 0 (black) to 255 (white)

L7	L6	L5	L4	L3	L2	L1	L0	L7	L6	L5	L4	L3	L2	L1	L0	L7	L6	L5	L4	L3	L2	L1	L0

20.10.6.6 Palette 256 8-bit (LUT8)

Values range from 0 to 255. These are looked-up from the Palette Color Table.

Pixel 0							
7	6	5	4	3	2	1	0
I7	I6	I5	I4	I3	I2	I1	I0
Byte 0							

20.10.6.7 RGBA 8888 32-bits

R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0	A7	A6	A5	A4	A3	A2	A1	A0

20.10.6.8 ARGB 8888 32-bit

A7	A6	A5	A4	A3	A2	A1	A0	R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0

20.10.6.9 ABGR 8888 32-bit

A7	A6	A5	A4	A3	A2	A1	A0	B7	B6	B5	B4	B3	B2	B1	B0	G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0

20.10.6.10 BGRA 8888 32-bit

B7	B6	B5	B4	B3	B2	B1	B0	G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	A7	A6	A5	A4	A3	A2	A1	A0

20.10.6.11 RGB 888 24-bit

R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0	R7	R6	R5	R4	R3	R2	R1	R0
G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0	R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0
B7	B6	B5	B4	B3	B2	B1	B0	R7	R6	R5	R4	R3	R2	R1	R0	G7	G6	G5	G4	G3	G2	G1	G0	B7	B6	B5	B4	B3	B2	B1	B0

20.10.6.12 RGBA 5551 16-bit

R4	R3	R2	R1	R0	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	A0	R4	R3	R2	R1	R0	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	A0

20.10.6.13 RGB 565 16-bit

R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0	R4	R3	R2	R1	R0	G5	G4	G3	G2	G1	G0	B4	B3	B2	B1	B0

20.10.6.14 RGB 332 8-bit

R2	R1	R0	G2	G1	G0	B1	B0	R2	R1	R0	G2	G1	G0	B1	B0	R2	R1	R0	G2	G1	G0	B1	B0	R2	R1	R0	G2	G1	G0	B1	B0

20.10.6.15 YUYV 32-bit 2-pixels

Y07	Y06	Y05	Y04	Y03	Y02	Y01	Y00	U7	U6	U5	U4	U3	U2	U1	U0	Y17	Y16	Y15	Y14	Y13	Y12	Y11	Y10	V7	V6	V5	V4	V3	V2	V1	V0

20.10.6.16 YUV2 32-bit 2-pixels

U7	U6	U5	U4	U3	U2	U1	U0	Y07	Y06	Y05	Y04	Y03	Y02	Y01	Y00	V7	V6	V5	V4	V3	V2	V1	V0	Y17	Y16	Y15	Y14	Y13	Y12	Y11	Y10

20.10.6.17 YUV Packed

Y37	Y36	Y35	Y34	Y33	Y32	Y31	Y30	Y27	Y26	Y25	Y24	Y23	Y22	Y21	Y20	Y17	Y16	Y15	Y14	Y13	Y12	Y11	Y10	Y07	Y06	Y05	Y04	Y03	Y02	Y01	Y00
Y77	Y76	Y75	Y74	Y73	Y72	Y71	Y70	Y67	Y66	Y65	Y64	Y63	Y62	Y61	Y60	Y57	Y56	Y55	Y54	Y53	Y52	Y51	Y50	Y47	Y46	Y45	Y44	Y43	Y42	Y41	Y40
U17	U16	U15	U14	U13	U12	U11	U10	U07	U06	U05	U04	U03	U02	U01	U00	V17	V16	V15	V14	V13	V12	V11	V10	V07	V06	V05	V04	V03	V02	V01	V00
U37	U36	U35	U34	U33	U32	U31	U30	U27	U26	U25	U24	U23	U22	U21	U20	V37	V36	V35	V34	V33	V32	V31	V30	V27	V26	V25	V24	V23	V22	V21	V20

20.10.6.18 TSc4 16-pixels / 64-bits

TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4
TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4
TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4
TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4	TSc4

20.10.6.19 TSc6 16-pixel / 96-bits

TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6
TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6	TSc6

20.10.6.20 TSc6A 16-pixels with Alpha / 96-bits

TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A
TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A	TSc6A

20.10.6.21 TSc12 4-pixels / 96-bits

Valid Input and Output color format.

20.10.6.22 TSc12A 4-pixels with Alpha / 96-bits

Valid Input and Output color format

21. Display Serial Interface (DSI)

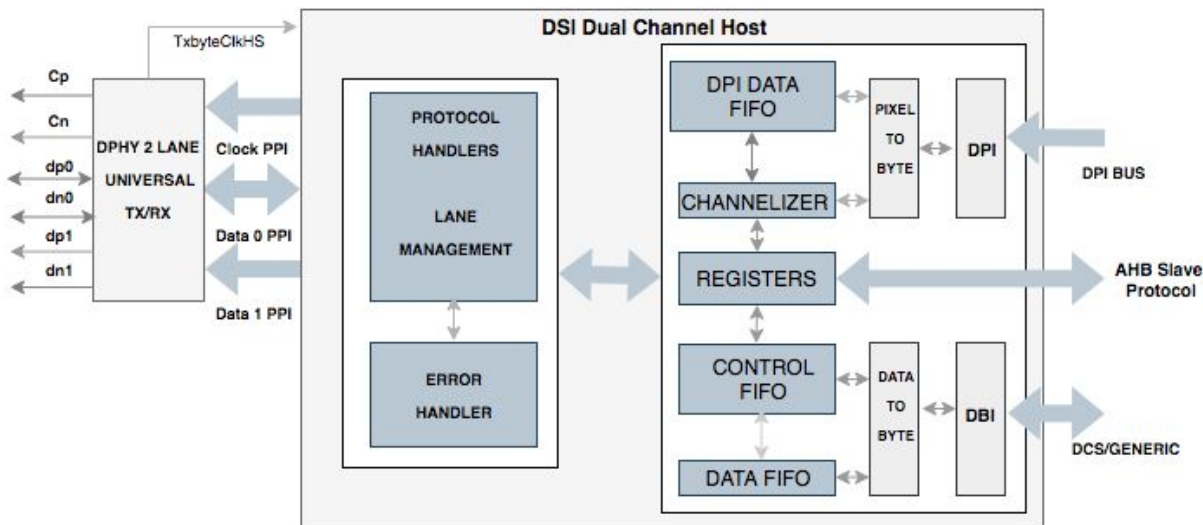


Figure 116. DSI Controller Block Diagram

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

21.1 Features

The DSI on the Apollo5 SoC supports the features listed below.

- Standard D-PHY transceiver compliant to MIPI Specification
- Type1 display architecture in command mode
- Can be programmed to support command mode in single channel mode
- Generic read and write commands
- Low power data transfer for both DBI/generic
- Pixel formats of types:
 - 16bpp [RGB565]
 - 18bpp [RGB666] and [Loosely packed RGB666]
 - 24bpp [RGB888]
- Recovery from contention
- Timers and recovery schemes to come out of mode fault errors
- Watchdog timers to monitor D-PHY activity:
 - in low power mode
 - in high speed mode
 - during turn-around
- Interrupts to report protocol errors and expiry of timers
- Programmable device initialization timers
- Programmable maximum return packet size command
- One or two PHY data lanes
- DBI interface for DCS commands and data transfer
- Data lane switching to low power mode during idle time
- Signals tearing effect

- Ultra low power mode switching
- Bus turn-around
- EOT disabling capacity to suit backward compatible displays
- Clock stop enabling feature during idle time
- Added support for deskew calibration
- Supported display resolutions:
 - QCIF
 - QVGA
 - CIF
 - VGA

The DSI is compliant with the following standards:

- DSI MIPI specification for Display Serial Interface (Version1.3.1)
- D-PHY standard MIPI specification (Version1.2)
- MIPI Alliance Standard for Display Bus Interface version2.0 0
- MIPI Alliance Standard for Display Command Set Version 1.02

NOTE

The DSI supports Type 1 display architecture in Command Mode, and Video Mode.

21.2 Functional Overview

The Display Serial Interface bus (DSI) on the Apollo5 SoC is a type of serial bus that enables transfer of data between a transmitter device and a receiver device. The DSI device has a point-to-point connection with DSI devices via D-PHYs as shown in Figure 117.

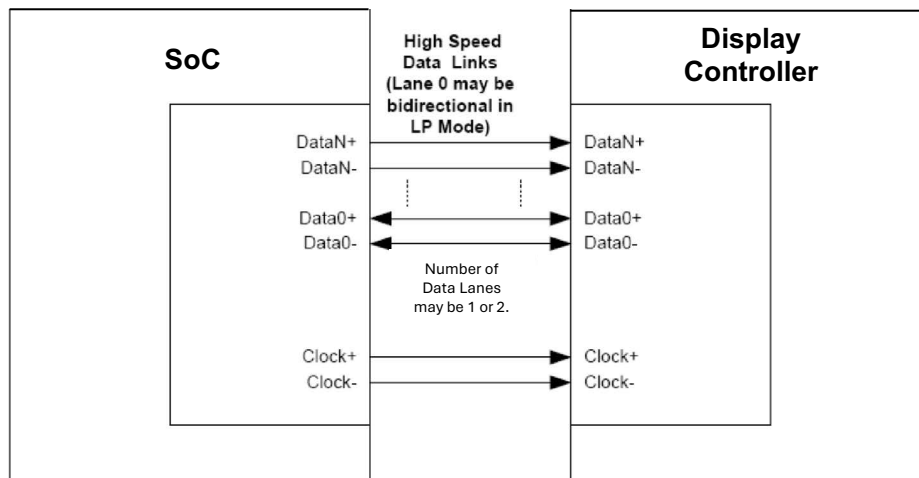


Figure 117. Display Serial Interface Bus with DSI Devices

The DSI module is configured to specify the interface and provide a connect between the MCU and a peripheral such as a display module. It is built on existing MIPI Alliance standards by adopting pixel formats, controlling pins and a command set specified in DBI-2 and DCS standards.

The D-PHY's data lane signals are transferred point-to-point as differential signals using two signal lanes and a clock lane. There are two signaling modes: high speed mode that operates at a rate of 768 Mbps/lane and a low power mode (LP) that operates at a lower transfer rate of 10 Mbps. The mode is set to a low power mode and a stop state at start up / power up. Depending on the desired data transfer type, the lanes switch between high and low power modes. High speed data transfer is unidirectional and data transfer at low speed can be unidirectional or bidirectional.

DSI devices operate in a layered fashion. There are 4 layers identified both at receiver and transmitter ends. Figure 118 shows the layers in the DSI data transfer model.

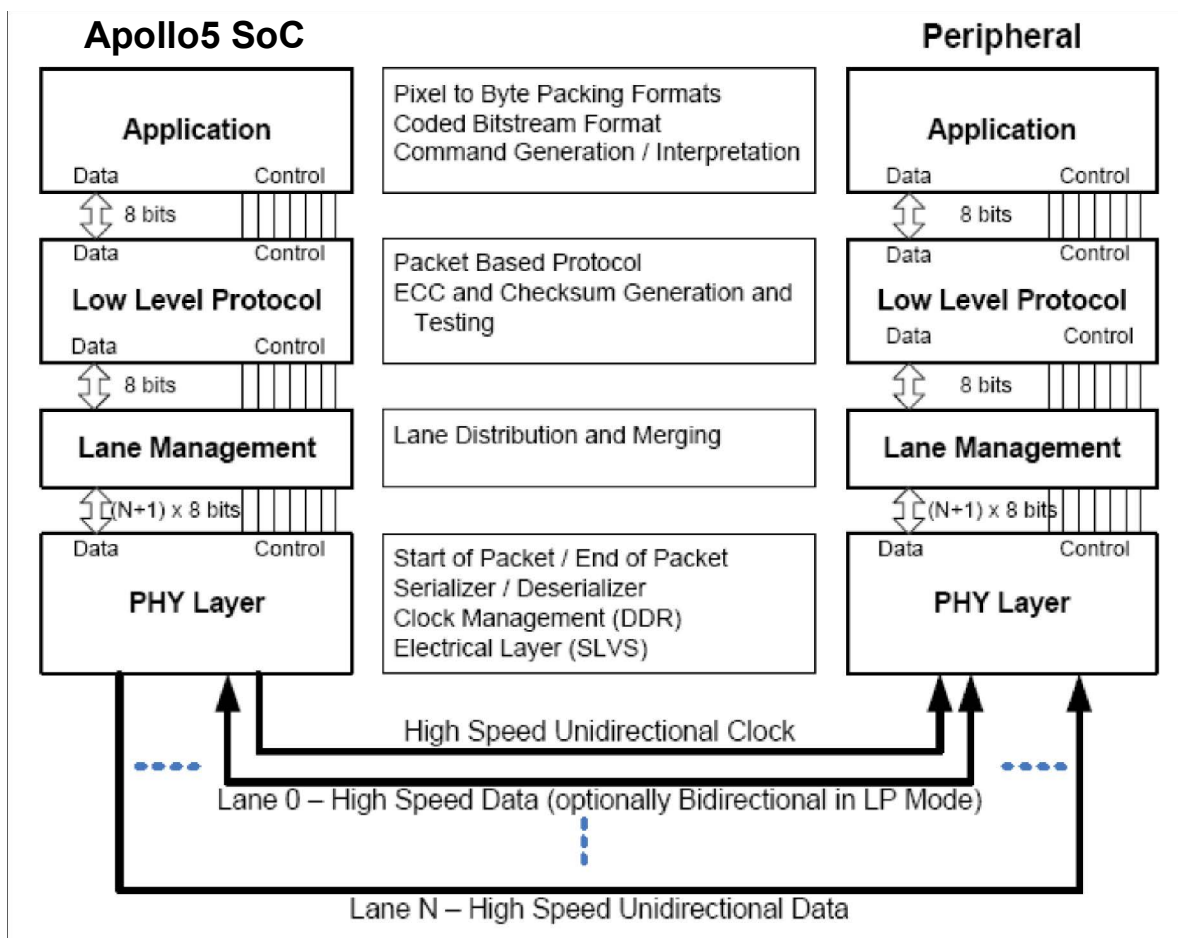


Figure 118. Layers in the DSI Data Transfer Model

PHY Layer: An embedded electrical layer that sends and detects start-of-packet and end-of-packet signaling on the data lanes. It has a serializer and de-serializer unit to dialogue with the PPI / lane management unit. It also has clock divider unit to source and receive clock during different modes of operation.

PPI / Lane Management Unit: This layer does the lane buffering and distributes the data in the lanes as programmed in a round robin manner and also merges them to stream line to the LLP/ PLI unit.

PLI / Low Level Protocol Unit: This layer packetizes as well as de-packetizes the data with respect to channels, frames, colors and line formats. There is an ECC generator and corrector unit to recover the data free from errors in the packet headers. It has a CRC checker or CRC generator unit to pack the payload data with CRC checksum bits for payload data protections.

Application: This layer describes higher-level encoding and interpretation of data contained in the data stream. Depending on the display subsystem architecture, it may consist of pixels having a prescribed format, or of commands that are interpreted by the display controller inside a display module.

21.3 Hardware Design Guidelines

The following sub-sections provide design guidelines for the use of the Apollo5 SoCs' DSI PHY. Also, please consult the DSI section of the Electricals for voltage, power and timing requirements of the PHY.

21.3.1 System Power Sequencing for DSI TX Interface

The power sequence for the DSI TX interface is contingent on the PHY(s) used in the system. The sections below address the proper sequences which include powering VDD18 only or with VDDUSB33 (3.3 V) and VDDUSB0P9 (0.9 V) supplies, and cover the cases for DSI TX Interface (D-PHY) only, and both USB PHY and D-PHY in the system.

Refer to the DSI PHY section in the Electricals for supply voltage specifications.

21.3.1.0.1 DSI TX Initialization Sequence

The sequence below assumes that the SoC is powered on. During booting to the application, the following actions should be performed.

1. Enable power to DSI TX and D-PHY (VDDF_DSIPHY_SW).
2. Holding D-PHY in reset, enable power to the external VDD18 rail.
3. Configure DSI TX to desired mode.

For power saving put DSI TX to ULPS state during periods of inactivity, as D-PHY in LP STOP mode consumes high power from the VDD18 rail (~2.5mA). DPHY's bias, LDOs and PLL blocks can be disabled through trim registers upon ULPS PHY entry:

- DSI->AFETRIM3 |= 0x00038000; //
- DSI->AFETRIM2 |= 0x0000001C; //

21.3.1.0.2 DSI TX Shutdown Sequence

On terminating DSI TX / D-PHY operation, the SoC should follow the below shut-down sequence:

1. Disable power to the external VDD18 rail.
2. Disable power to D-PHY (VDDF_DSIPHY_SW).

21.4 MIPI DSI Video Mode

21.4.1 What DSI Video Mode Is

In Command Mode, the MCU sends pixels frame by frame with MIPI DCS commands (e.g., write memory). The panel waits for each transfer.

In Video Mode, the SoC streams pixels continuously — like a traditional RGB/DPI interface. The panel passively receives a timed pixel stream over DSI. No per-frame write memory is needed after setup.

Video Mode bring-up uses a two-phase approach:

1. Command path — initialize the panel over MIPI
2. Video path — NemaDC streams pixels through an on-chip DPI-to-DSI bridge

21.4.2 Data Path

The data path is as follows:

Framebuffer in SRAM → NemaDC Layer 0 → Internal DPI(PCLK, HSYNC, VSYNC, DE, RGB) → DSI Host(Video Mode) → Display panel

After initialization, NemaDC scans the framebuffer at a fixed refresh rate. The DSI Host packs that parallel stream into MIPI DSI high-speed packets.

21.4.3 Initialization Sequence

The initialization sequence is the following:

1. System and DSI bring-up
 - Enable GFX and DISP peripherals
 - am_hal_dsi_init() — DSI clocks and PHY power
 - nema_init() / nemadc_init() — graphics and display controller

2. Panel timing and DSI configuration

Panel timing (resolution, porches) comes from the display driver:

```
am_devices_dc_dsi_forcelead_get_parameter(&sDCCConfig);
sDCCConfig.bTEEnable = false;
```

DSI is configured for video streaming:

```
am_hal_dsi_config_t sDSIConfig = {0};
sDSIConfig.ui8LanesNum = 2;
sDSIConfig.eDBIBusWidth = (am_hal_dsi_dbi_width_e)g_sDispCfg.eDbiWidth;
sDSIConfig.bSendUlpPattern = false;
sDSIConfig.eFormat = DSI_FORMAT_RGB888;
sDSIConfig.eMode = VIDEO_MODE_BURST;
```

am_hal_dsi_config() programs the DSI Host with:

- Lane count and pixel format (RGB565 / RGB666 / RGB888)
- Video mode type (Burst, Non-Burst Pulse, or Non-Burst Events)
- Horizontal/vertical timing (HACT, HSA, HBP, HFP, VACT, etc.)
- PCLK frequency and derived DSI HS clock

3. Phase A — DBIDSI: panel initialization

NemaDC is first set to DBIDSI so MIPI commands can reach the panel:

- sDCCConfig.eInterface = DISP_INTERFACE_DBIDSI;
- sDCCConfig.ui32PixelFormat = MIPICFG_16RGB888_OPT0;
- nemadc_configure(&sDCCConfig);

Then `am_hal_dsi_config()` is called, the panel is reset, and `am_devices_dc_dsi_forcelead_init()` sends the full MIPI init sequence (power, gamma, lane config, etc.) via `nemadc_mipi_cmd_write()`, ending with exit sleep and display on.

At this point the panel is ready, but no video stream has started.

4. Phase B — DPI: start continuous streaming

The interface switches to DPI:

```
sDCConfig.eInterface = DISP_INTERFACE_DPI;
sDCConfig.ui32PixelFormat = NEMADC_P_RGB3;
// ... setup Layer 0 framebuffer ...
nemadc_configure(&sDCConfig);
```

`nemadc_configure(DPI)` tells NemaDC to:

- Output parallel RGB with timing signals (PCLK, HSYNC, VSYNC, DE)
- Apply porch timing and pixel clock division
- Set color format and sync polarities

The DSI Host's internal DPI input receives this stream and forwards it as MIPI Video Mode traffic. DPI signals do not need to go to external GPIOs — the connection is on-chip.

5. Start display

```
nemadc_set_layer(0, &sLayer0);
nemadc_set_vsync_interrupt_callback(next_frame_callback, NULL);
nemadc_reg_write(NEMADC_REG_INTERRUPT, 1 << 4);
nemadc_set_mode(NEMADC_ENABLE | nemadc_reg_read(NEMADC_REG_MODE));
```

- Layer 0 points at the framebuffer (here, a static `butterfly_rgb24` image)
- `NEMADC_ENABLE` starts continuous scan-out
- `VSYNC` interrupt is enabled for per-frame callbacks

21.4.4 Framebuffer and Layer Setup

```
sLayer0.resx      = sDCConfig.ui16ResX;
sLayer0.resy      = sDCConfig.ui16ResY;
sLayer0.format     = NEMADC_RGB24;
sLayer0.stride     = nemadc_stride_size(sLayer0.format, sLayer0.resx);
sLayer0.baseaddr_virt = tsi_malloc(sLayer0.resy*sLayer0.stride);
sLayer0.baseaddr_phys = (unsigned)(sLayer0.baseaddr_virt);
memcpy(sLayer0.baseaddr_virt, butterfly_rgb24, sizeof(butterfly_rgb24));
```

NemaDC reads this buffer every frame. For animation or double buffering, update the buffer (or swap baseaddr) in the VSYNC callback.

21.4.5 VSYNC Callback

Following is the VSYNC callback code. This is demo logic only — it toggles RGB/BGR every 10 frames. Video Mode does not require software to trigger each frame; hardware keeps scanning. The callback is useful for tear-free buffer swaps or frame-rate stats.

```
void next_frame_callback(void* pHandle, uint32_t status)
{
    if (g_ui32FrameIndex++ % 10 == 0)
    {
        (sLayer0.format == NEMADC_RGB24) ? (sLayer0.format = NEMADC_BGR24) :
                                           (sLayer0.format = NEMADC_RGB24);

        nemadc_set_layer(0, &sLayer0);
    }
    nemadc_reg_write(NEMADC_REG_INTERRUPT, 1 << 4);
}
```

21.4.6 Video Mode Types

Burst Mode (VIDEO_MODE_BURST) sends active pixels in one HS burst per line, which is efficient on bandwidth.

Alternatives (configured via sDSIConfig.eMode):

- Non-Burst with Sync Pulses — separate sync packets
- Non-Burst with Sync Events — sync embedded as events

PCLK limits depend on mode, lane count, and pixel format.

21.4.7 Command Mode vs Video Mode (in this codebase)

Table 107 contrasts differences in parameters and functional areas between command mode and video mode.

Table 107: Command Mode and Video Mode Comparison

Parameter/Function	Command Mode	Video Mode
NemaDC interface	DISP_INTERFACE_DBIDS	DISP_INTERFACE_DPI (after init)
Pixel delivery	Per-frame MIPI commands	Continuous hardware stream
TE (tearing effect)	Often required	Not used (bTEEnable = false)
CPU per frame	Must kick off each transfer	Only optional VSYNC work
Init	MIPI command sequence	Same init, then switch to DPI

21.4.8 Summary

Video Mode operation proceeds as follows:

1. Initialize the panel over the DBIDS command path.
2. Configure the DSI Host for Video Mode (format, lanes, timing, clock).
3. Switch NemaDC to DPI and point Layer 0 at a framebuffer.
4. Enable NemaDC — it continuously scans the buffer; the on-chip DPI→DSI bridge streams pixels to the panel.

The essential idea: command path for setup, DPI path for display. Once running, the display is driven by hardware scan-out, not by software sending MIPI write commands every frame.

22. Graphics and the GFX Library on the Apollo5 Family MCUs

This chapter is a programming guide for graphics and the use of the GFX library and GPU on the MCU.

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

22.1 Introduction to Graphics

Computer graphics is the science of communicating visually via a display and its interaction devices. It is a cross-disciplinary field in which physics, mathematics, human perception, human-computer interaction and engineering blend, towards creating artificial images with the help of programming. It heavily involves computations, creation and manipulation of data and is based on a set of well-defined principles. There are several structural elements that computer graphics are built upon, the most significant of which can be found in the following list.

Pixel in digital imaging is the smallest addressable element in an all points addressable display device. A pixel is generally considered as the smallest single component of a digital image and is often used as a measurement unit.

Raster image (or bitmapped image) is a matrix data structure representing the actual image content. Raster graphics are resolution-bound therefore unable to scale up without apparent loss of quality.

Vector graphics is a technique of using polygons, plane figures bound by a finite chain of straight-line segments closing a loop, to represent images. Vector graphics have inherent scale up abilities, only depending on the rendering device capability.

Rasterization is the process of converting an image described in a vector graphics format to a raster image consists of pixels for output on a video display or for storage in a bitmap format.

Texture is the digital representation of an object's surface. In addition to two-dimensional qualities such as color and transparency, a texture also incorporates three-dimensional ones such as reflectiveness. Well-defined textures are very important for realistic three-dimensional image representation.

Texture mapping is the process of wrapping a pre-defined texture around any two or three-dimensional object. Through this process, digital images and objects obtain a high level of detail.

Texel is the fundamental unit of texture space. Textures are represented by arrays of texels in the same way that pictures are represented by arrays of pixels.

Vertex is a data structure that describes the location of an object by properly define its corners as positions of points in two or three-dimensional space.

Primitives in computer graphics are the simplest geometric objects a system can handle. Common sets of two-dimensional primitives include lines, points, triangles and polygons while all other geometric elements are built up from these primitives. In three-dimensions, properly positioned triangles or polygons can be used as primitives to model more complex forms.

Blending is the process in which two or more images are combined per-pixel and weights to create new pictures.

Fragment is the data necessary to generate a single-pixel primitive. This data is possible to include raster position, color or texture coordinates.

Interpolation in computer graphics is the process of generating intermediate values between two known reference points to give the appearance of continuity and smooth transition. Several distinct interpolation techniques are used in both computer graphics and animation, such as linear, bilinear, spline and polynomial interpolation.

Graphics Pipeline is an abstract sequence that incorporates the basic operations of generic rasterizer implementations, in particular:

- (Vertex) Per-vertex transformation to screen space
- (Rasterize) Per-triangle iteration over pixels with perspective-correct interpolation
- (Pixel) Per-pixel shading
- (Output Merge) Merging the output of shading with the current color and depth buffers

The term "pipeline" is used due to the sequential steps that are used for the actual transformation from mathematical model to pixels; the results of the one stage are pushed on to the next stage so that the first stage can begin processing the next element immediately.

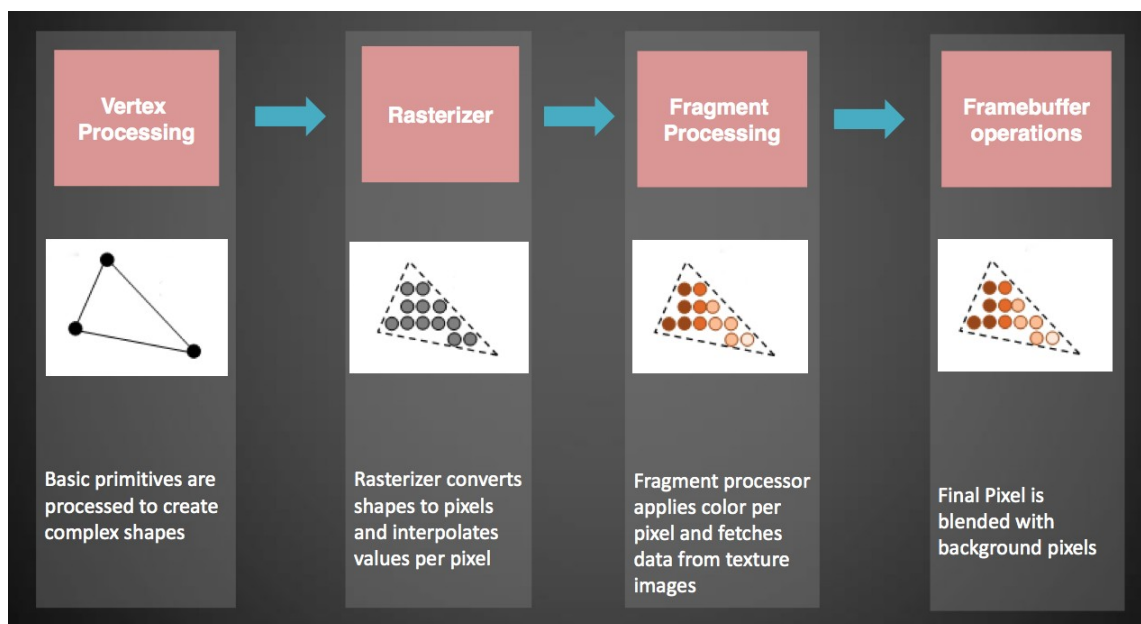


Figure 119. Rendering Flow

22.2 GPU and Graphics Software Support

The following is supported in software for graphics development:

- FreeRTOS support
 - Bare Metal (no-OS)
- Graphics API support
 - GFX API library in pure C
- Available software emulators and suites
 - NEMA®| PIX-Presso

The GFX library is available in pure ANSI C with no dependencies and is ported to FreeRTOS with the following features:

- Enables high quality 2.5D graphics on RTOS and OS-less systems.
- Proprietary low level library that interfaces directly with the GPU and provides a software abstraction layer to organize and employ drawing commands with ease and efficiency.
- Can be used as a back-end to existing APIs and as a standalone Graphics API.

The software package includes the NEMA®| PIX-Presso, a utility software for converting images to/from formats suitable for low power embedded devices.

22.3 GFX Library Architecture

The GFX Library is a low level library that interfaces directly with the GPU and provides a software abstraction layer to organize and employ drawing commands with ease and efficiency. The target of the GFX library is to be able to be used as a back-end to existing APIs (such as OpenGL®, DirectFB or any proprietary one) but also to expose higher level drawing functions, so as to be used as a standalone Graphics API. Its small footprint, efficient design and lack of any external dependencies, makes it ideal for use in embedded applications. By leveraging the sophisticated architecture, it allows great performance with minimum CPU/SoC usage and power consumption.

The GFX library includes a set of higher level calls, forming a complete standalone Graphics API for applications in systems where no other APIs are needed. This API is able to carry out draw operations from as simple as lines, triangles and quadrilaterals to more complex ones like blitting and perspective correct texture mapping.

The GFX library is built on a modular architecture. An implementor may use only the lower layers of the architecture that provides communication to the GPU hardware, synchronization and basic primitives drawing. The very thin Hardware Abstraction Layer allows for fast integration to the underlying hardware. The upper low level drawing API acts as a back-end interface for accelerating any higher 3rd party Graphics API.

The GFX library modules are generally stacked one over another, forming a layered scheme. This gives the implementor the freedom to tailor the software stack according to ones needs.

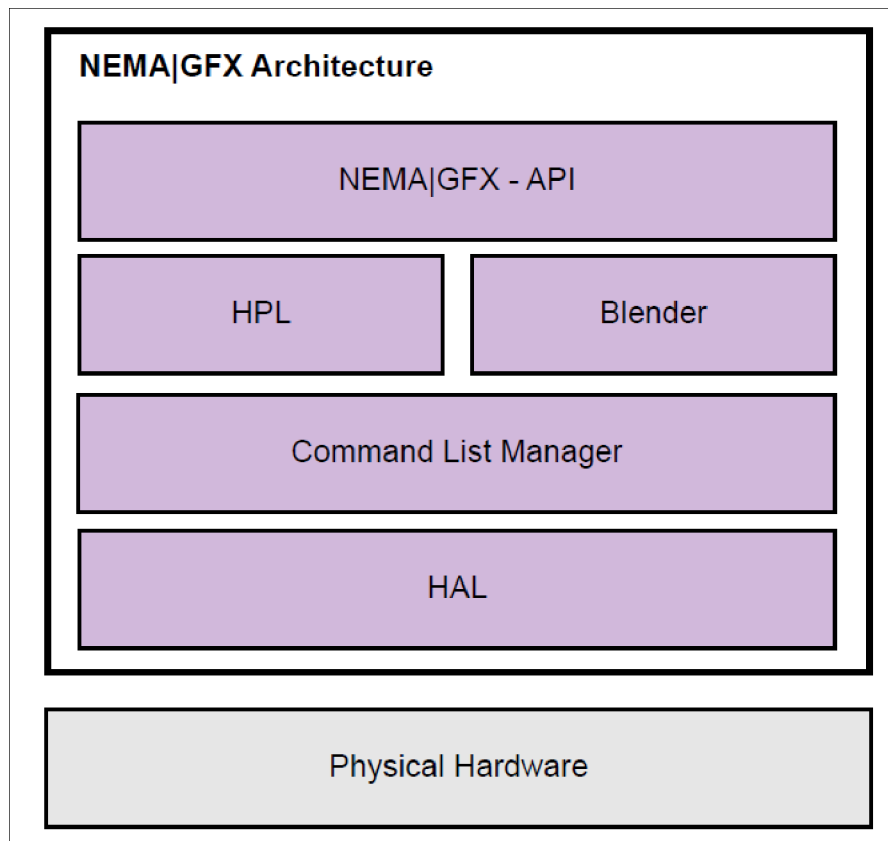


Figure 120. GFX Library Architecture

The lowest layer is a thin Hardware Abstraction Layer (HAL). It includes some hooks for basic interfacing with the hardware such as register accessing, interrupt handling etc.

The layer above is the Command List Manager. It provides the appropriate API for creating, organizing and issuing Command Lists. This topic is discussed in detail in Section 22.6.1 on page 361.

Above the Command List Manager lies the Hardware Programming Layer (HPL). This is a set of helper functions that assemble commands for programming the GPU. These commands actually write to the GPU's Configuration Register File, which is used to program the submodules of the GPU.

Alongside the HPL resides the Blender module. This module programs the GPU's Programmable Processing Core. It creates binary executables for the Core. These executables correspond to the various blending modes that are supported by the GFX library.

On top of the GFX library stack lies the graphics API. This API offers function calls to draw geometry primitives (lines, triangles, quadrilaterals etc), blit images, render text, transform geometry objects, perform perspective correct texture mapping etc. When using the GFX library as a back-end for a third party Graphics API, much of the GFX library Graphics API may be disabled.

22.4 Graphics Pipeline of the GPU

The GPU has been designed for graphics efficiency in ultra-compact silicon area. Its fixed-point data path and instruction set architecture (ISA) are tailored to GUIs acceleration and small display applications leading to substantial improvements in power consumption and silicon area. The GPU microarchitecture combines hardware-level support for multi-threading, VLIW and low-level vector processing in the most power efficient way. The GPU is connected to the CPU core via a 32/64-bit AXI4 bus.

The GPU is a modular architecture which includes a separate programmable Fragment Processing Core, a Z Buffer Unit, a Texture Map Unit and Render Output Unit. It utilizes a fixed-point data path and custom VLIW Instruction Set Architecture (ISA) in a heavily multi-threaded execution pipeline which is optimized for extracting the maximum performance in 3D acceleration in a power and silicon area constrained system.

22.4.1 Instruction Set Architecture

The microarchitecture of the GPU for fragment processing is a four-issue VLIW core. The compact GPU VLIW Instruction Set Architecture (ISA) has been carefully built to support a variety of graphic operations in a few instructions. Each VLIW instruction is 64 bits long and consists of four sub-instructions (corresponding to sub-operations), as shown in Figure 121. In accordance with Figure 121, the sub-operations for the GPU are: i) instructions for Alpha channel arithmetic operations (SUBOP0), ii) instructions for RGB vector arithmetic operations (SUBOP1), iii) memory addressing instructions for reading and writing to memory (SUBOP2), and iv) branching instructions based on comparisons between RGB or Alpha values (SUBOP3).

Each VLIW bundle is designed to perform six arithmetic operations per cycle (simultaneously). If more than one VLIW bundle is enabled, the addressing sub-operations (SUBOP2) are issued once for all the enabled bundles while the remaining sub-operations (SUBOP0, SUBOP1, and SUBOP3) are issued once for each VLIW bundle.

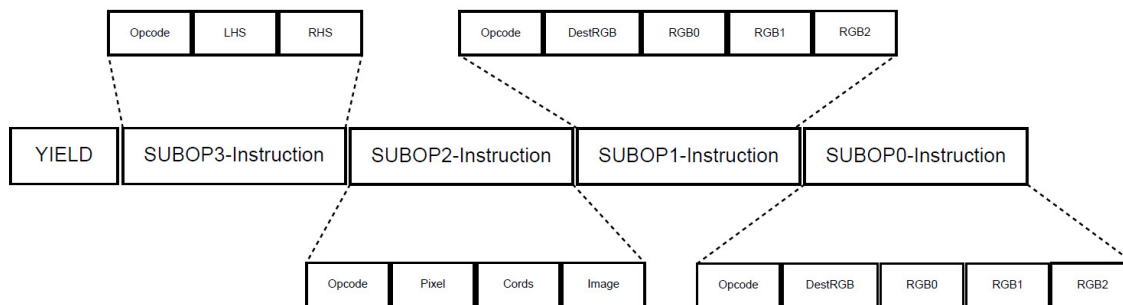


Figure 121. GPU VLIW fragment instruction format

A Yield bit is responsible to stop the execution of a VLIW bundle when it is activated. Consider the register file organization, four simultaneous, read/write operations to general purpose registers (four general purpose registers in total; R0 to R3) and four read-only operations to constant registers (four constant registers in total; C0 to C3) are allowed at each cycle. Consider the alpha channel sub-operations (SUBOP0), more complex arithmetic operations such as "1-alpha" are natively supported (i.e., they can be done without extra instructions). For example, see Table 111 in which the sub-operand A1 is set to 0x4 value which reads the value 1-alpha of R1 register.

22.4.1.1 SUBOP0 format set

The SUBOP0 instructions is a subset of the GPU ISA that contains instructions for alpha operations. These instructions have one destination operand that can be written, see Table 108.

Table 108: DestA Operand

Value	Register	Bits	Summary for DestA
0x0	R0	7:0	Writing R0's Alpha value.
0x1	R1	7:0	Writing R1's Alpha value.
0x2	R2	7:0	Writing R2's Alpha value.
0x3	R3	7:0	Writing R3's Alpha value.

The Alpha operations that can be executed are described in Table 109.

Table 109: SUBOP0-Instructions

Opcode	Instruction	Dest	A0	A1	A2	Summary
0x0	NOP					No operation
0x1	AMOV	DestA		A1		Move A1 to DestA register
0x2	AMADD	DestA	A0	A1	A2	The result of multiplication A0 with A1 is added with A2 and saved on Dest Register
0x3	AMUL	DestA		A1		The result of reciprocal with A1 is saved on DestA register

The source operands are A0, A1 and A2. The configuration values are described in Table 110, Table 111 and Table 112, respectively.

Table 110: A0 Operand

Value	Register	Bits	Summary for Operand A0
0x0	R0	7:0	Read Alpha value from R0
0x1	R1	7:0	Read Alpha value from R1
0x2	R2	7:0	Read Alpha value from R2
0x3	R3	7:0	Read Alpha value from R3

Table 111: A1 Operand

Value	Register	Bits	Summary for Operand A1
0x0	R0	7:0	Read Alpha value from R0
0x1	R1	7:0	Read Alpha value from R1
0x2	R2	7:0	Read Alpha value from R2
0x3	R3	7:0	Read Alpha value from R3
0x4	R0	7:0	Read (1 - Alpha) value from R0
0x5	R1	7:0	Read (1 - Alpha) value from R1
0x6	R2	7:0	Read (1 - Alpha) value from R2
0x7	R3	7:0	Read (1 - Alpha) value from R3
0x8	R0	15:8	Read Blue value from R0
0x9	R0	23:16	Read Green value from R0
0xa	R0	31:24	Read Red value from R0
0xb	C0	7:0	Read Alpha value from constant C0
0xc	C1	7:0	Read Alpha value from constant C1
0xd	C2	7:0	Read Alpha value from constant C2
0xe	Immediate	7:0	Read 0xff immediate value
0xf	Immediate	7:0	Read 0x0 immediate value

Table 112: A2 Operand

Value	Register	Bits	Summary for Operand A2
0x0	R0	7:0	Read Alpha value from R0
0x1	R1	7:0	Read Alpha value from R1
0x2	R2	7:0	Read Alpha value from R2
0x3	R3	7:0	Read Alpha value from R3
0x4	C0	7:0	Read Alpha value from constant C0
0x5	C1	7:0	Read Alpha value from constant C1
0x6	C2	7:0	Read Alpha value from constant C2
0x7	C3	7:0	Read Alpha value from constant C3

22.4.1.2 SUBOP1 format set

The SUBOP1 instructions is a subset of the GPU ISA that contains instructions for the RGB operations. Operands are vectors of 3 elements. In case that an operand reads alpha value, the 8bit alpha value Becomes a vector of 3 elements with the same value. The RGB operations that can be executed are described in Table 113.

Opcode	Instruction	Dest	RGB0	RGB1	RGB2	Summary
0x0	NOP					No operation.
0x1	RMADD	DestRGB	RGB0	RGB1	RGB2	The result of multiplication RGB0 with RGB1 is added to RGB2 and saved on Dest Register.
0x2	RMOV	DestRGB		RGB1		Move RGB1 to Dest register.
0x3	RADD	DestRGB		RGB1	RGB2	Add RGB1 with RGB2 and save it to Dest register.
0x5	RMSB	DestRGB	RGB0	RGB1	RGB2	RGB2 is subed from the result of multiplication between RGB0 with RGB1 and saved on Dest Register.
0x6	RMAX	DestRGB		RGB1	RGB2	If RGB2 is greater than RGB1 then RGB0 is written to DST else RGB1.
0x7	RMIN	DestRGB		RGB1	RGB2	If RGB2 is less than RGB1 then RGB0 is written to DST else RGB1.
0x8	ADDXY	DestRGB		RGB1	RGB2	R2 XY values are added with R1 XY values
0x9	ANDXY	DestRGB		RGB1	RGB2	R2 XY values are ANDed with R1 XY values

Table 113: SUBOP1 Instructions

These instructions have one destination operand that can be written, see Table 114.

Value	Register	Bits	Summary for Operand DestRGB
0x0	R0	31:8	Writing R0's RGB values
0x1	R1	31:8	Writing R1's RGB values
0x2	R2	31:8	Writing R2's RGB values
0x3	R3	31:8	Writing R3's RGB values

Table 114: DestRGB Operand

The source operands are RGB0, RGB1 and RGB2. The accepted values are described in Table 115, Table 116 and Table 117, respectively.

Value	Register	Bits	Summary for RGB0
0x0	R0	7:0	Read RGB value from R0
0x1	R1	7:0	Read RGB value from R1
0x2	R2	7:0	Read RGB value from R2
0x3	R3	7:0	Read RGB value from R3

Table 115: RGB0 Operand

Value	Register	Bits	Summary for RGB1
0x0	R0	31:8	Read RGB values from R0
0x1	R1	31:8	Read RGB values from R1
0x2	R2	31:8	Read RGB values from R2
0x3	R3	31:8	Read RGB values from R3
0x4	R0	7:0	Read Alpha value from R0
0x5	R1	7:0	Read Alpha value from R1
0x6	R2	7:0	Read Alpha value from R2
0x7	R3	7:0	Read Alpha value from R3
0x8	R0	31:8	Read $(1 - R)(1 - G)(1 - B)$ values from R0
0x9	R1	31:8	Read $(1 - R)(1 - G)(1 - B)$ values from R1
0xa	R2	31:8	Read $(1 - R)(1 - G)(1 - B)$ values from R2
0xb	R3	31:8	Read $(1 - R)(1 - G)(1 - B)$ values from R3
0xc	R0	7:0	Read $(1 - \text{Alpha})$ value from R0
0xd	R1	7:0	Read $(1 - \text{Alpha})$ value from R1
0xe	R2	7:0	Read $(1 - \text{Alpha})$ value from R2
0xf	R3	7:0	Read $(1 - \text{Alpha})$ value from R3
0x10	C0	31:8	Read RGB values from constant C0
0x11	C1	31:8	Read RGB values from constant C1

Table 116: RGB1 Operand

Value	Register	Bits	Summary for RGB2
0x0	R0	31:8	Read RGB values from R0
0x1	R1	31:8	Read RGB values from R1
0x2	R2	31:8	Read RGB values from R2
0x3	R3	31:8	Read RGB values from R3
0x4	C0	31:8	Read RGB values from constant C0
0x5	C1	31:8	Read RGB values from constant C1
0x6	C2	31:8	Read RGB values from constant C2
0x7	C3	31:8	Read RGB values from constant C3

Table 117: RGB2 Operand

22.4.1.3 SUBOP2 format set

The SUBOP2 instructions is a subset of the GPU ISA that contains instructions for addressing, reading and writing to memory. The Addressing operations that can be executed are described in Table 118.

Opcode	Instruction	Dest	Coords	Image	Summary
0x0	NOP				No operation.
0x1	CREADTEX	PIXEL	Coords	IMG	Conditionally Read Texture
0x3	READTEX	PIXEL	Coords	IMG	Read from Texture
0x5	PIXOUT	PIXEL	Coords	IMG	Pixout

Table 118: SUBOP2 Instructions

Table 119 provides a summary for Pixel Operand while Table 120 and Table 121 describe the Coords Operand.

Value	Register	Bits	Summary for PIXEL Operand
0x0	R0	7:0	Writing R0's value.
0x1	R1	7:0	Writing R1's value.
0x2	R2	7:0	Writing R2's value.
0x3	R3	7:0	Writing R3's value.

Table 119: Pixel Operand

Value	Register	Bits	Summary for Coords Operand
0x0	iY, iX		Read iY, iX values for addressing.
0x1	i_R10	31:0	Read <i>TY</i> 4, <i>TX</i> 4 values for addressing.

Table 120: Addressing Coords Operand

Value	Register	Summary for Coords Operand
0x0	TEX0BASE	Read Image from IMAGE0.
0x1	TEX1BASE	Read Image from IMAGE1.
0x2	TEX2BASE	Read Image from IMAGE2.
0x3	TEX3BASE	Read Image from IMAGE3.

Table 121: TEXnBASE Coords Operand

22.4.1.4 SUBOP3 format set

The SUBOP3-instructions is a subset of the GPU ISA that contains compare instructions between the RGB values of the two operands LHS and RHS. The Compare operations that can be executed are described in Table 122.

Opcode	Instruction	LHS	RHS	Summary
0x0	NOP	LHS	RHS	No operation
0x1	EQ	LHS	RHS	Compare if LHS is equal to RHS
0x2	NEQ	LHS	RHS	Compare if LHS is not equal to RHS
0x3	LESS	LHS	RHS	Compare if LHS is less than RHS

Table 122: SUBOP3 Instructions

The source operands are LHS and RHS. The accepted values are described on Table 123.

Value	Register	Bits	Summary for LHS and RHS Operands
0x0	R0	31:8	Read RGB values from R0
0x1	R1	31:8	Read RGB values from R1
0x2	R2	31:8	Read RGB values from R2
0x3	R3	31:8	Read RGB values from R3
0x4	C0	31:8	Read RGB values from constant C0
0x5	C1	31:8	Read RGB values from constant C1
0x6	C2	31:8	Read RGB values from constant C2
0x7	C3	31:8	Read RGB values from constant C3

Table 123: LHS, RHS Operands

22.4.2 Configuration Register File

The GPU is programmed through a set of registers. This register set is called the Configuration Register File (CRF) and each sub-module of the GPU is programmed through a subset of the CRF. The CRF can be memory mapped to the CPU address space, thus making it directly accessible. Writing the CRF directly is considered inefficient since it consumes a large volume of the CPU resources and ties the CPU execution to the GPU. For this reason, it can also be accessed indirectly through the Command List Processor (CLP).

22.4.3 Command List Processor

In order to decouple CPU and GPU execution and achieve both better performance and lower power consumption, the GPU incorporates an advanced Command List Processor (CLP), capable of reading entire list of commands from the main memory and relay them to the Configuration Register File.

The CPU pre-assembles Command Lists (CL) prior to submitting them to the Command List Processor for execution, while a single Command List can be submitted multiple times. This approach alleviates the CPU from recalculating drawing operations for repetitive tasks, resulting in more efficient resource utilization.

The steps for writing commands to the Configuration Registers through the Command List Processor are the following:

1. The CPU assembles a Command List, through the GFX library.
2. The CPU submits the Command List for execution. The Command List Processor is informed of a pending Command List.
3. The Command List Processor reads the Command List from the System Memory.
4. The Command List Processor relays the commands to the Configuration Register File.

22.4.4 Vertex Processing

The Vertex Processing unit in the rendering pipeline handles the processing of individual vertices. The Vertex Processing unit utilizes 64-bit VLIW (very long instruction word) instructions and performs computations on vertices which sends them to the Rasterizer unit through the Configuration Registers. The Vertex Processing unit is programmable through binary executables called Vertex Shaders. It transforms each vertex's 3D position in object space to the 2D coordinate at which it will appear on the screen. It also calculates the depth value for the Z-buffer and manipulates properties such as position, depth, color and texture coordinates.

22.4.5 Rasterizer

The GPU is capable of drawing a multitude of geometrical shapes called Geometric Primitives, such as lines, rectangles, triangles and quadrilaterals. The Rasterizer Unit reads the coordinates of the primitives' vertices and feeds the rest of the graphics pipeline with the fragments contained in the geometry. A fragment contains information concerning a single pixel. This information includes raster position (coordinates), texture coordinates, interpolated color alpha and depth values.

The Rasterizer of the GPU gives each core (1-4) a specific pixel to draw as part of the image. Each core can draw independently from the others, but a shape should be fully drawn, before the next one starts.

22.4.6 Texture Map Unit

The Texture Map Unit produces texels that sends to the Fragment Processing Core. It is fed with texture's attributes (base address, dimensions, color format) and the required coordinates. The Texture Map Unit performs some internal processing and outputs the corresponding texel. Generating a texture element requires a series of operations like wrapping (clamp, mirror, repeat etc), reading corresponding color values from memory, converting the color values to RGBA8888 format and performing filtering if necessary. If texture compression technique is used, then on-the-fly decompression is performed.

22.4.7 Fragment Processing Core

The Fragment Processing Core is the main processing unit of the GPU's architecture. It is a 64-bit VLIW processor which performs computations on the fragments coming from the Rasterizer Unit and on the texels coming from the Texture Map Unit and calculates the final color and depth to a fragment. The Core is programmable through binary executables called Fragment Shaders.

22.4.8 Render Output Unit

The Render Output Unit (ROP) is the last stage of the Graphics Pipeline. The Fragment Processing Core feeds the Render Output Unit with the pixel's coordinates and color value. Before the color value is written to the memory, the color is converted to the Frame Buffer's format. If texture compression technique is used, then decompression is performed while reading from the Frame Buffer and compression is performed while writing to the Frame Buffer.

With the H/W Blender, the Render Output Unit reads pixels from the Fragment Processing Core (source) and pixels from the Frame Buffer (destination) to perform blending. Blending requires a series of

calculations between the source (foreground) and destination (background) color fragments to produce the final color, which is written back to memory. The following equations are used for the final color:

$$Fc = Sc \cdot Sf + Dc \cdot Df \quad Fa = Sa \cdot Sf + Da \cdot Df$$

The Color and Alpha values range from 0 to 1, therefore each calculation result is also clamped to the same range. The available Blend Factors and the resulting RGBA values are listed in Table 127. Figure 129 shows the effect of the blending modes.

22.5 Frame Buffer Compression

Framebuffer compression operates in 4x4 pixel screen blocks for TSC™4, TSC™6 and TSC™6a or in 2x2 pixel screen blocks for TSC™12 and TSC™12a lossy fixed-ratio compression, depending on the configuration.

- TSC™4 is a 6:1 compression (4 bpp)
- TSC™6 is a 4:1 compression (6 bpp)
- TSC™6a is a 5.33:1 compression (6 bpp) with alpha channel
- TSC™12 is a 2:1 compression (12 bpp)
- TSC™12a is a 2.67:1 compression (12 bpp) with alpha channel

Compression is performed at run time using minimal hardware. Pixel data can be stored in the framebuffer in compressed form and decompressed in the NEMA®|dc. Figure 122 shows the TS compression operation. The output of TSC™4 compression is 64 bits per 4x4 block of pixels, the output of TSC™6 compression is 96 bits per 4x4 block of pixels and the output of TSC™12 compression is 48 bits per 2x2 block of pixels.

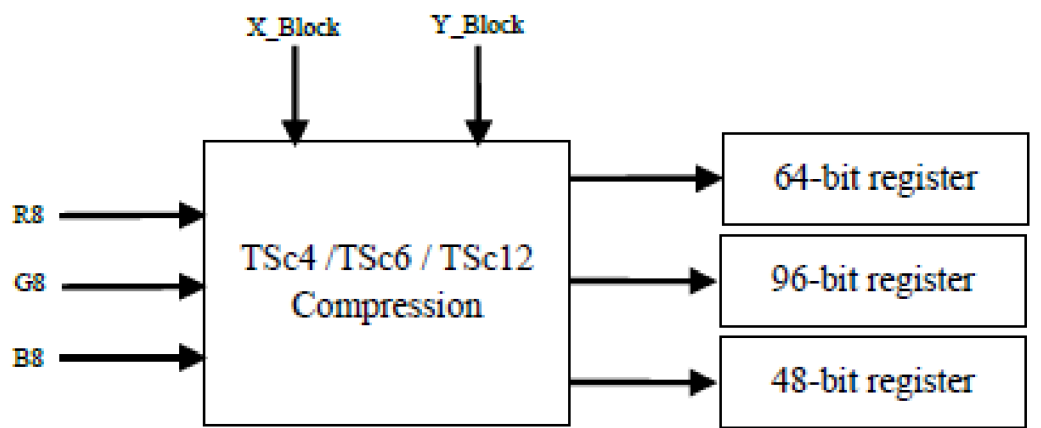


Figure 122. TSC™4 /TSC™6/TSC™12 Framebuffer Compression Module

Compression is performed using the tsc binary file located in NemaTS/Hardware/bin/tsc directory. The syntax is:

```
./ tsc [ options ] <source > <target >
```

where <source> is the source image and <target> is the compressed image.

The available [options] are:

-c -mode <mode>	Compress source image in png format using TS compression (default option)
-d <image_width>x<image_height> -mode <mode>	Decompress source image of TS format to target image in png format.
-e	Use diff command to compare the source image (in png format) with the target image (in png format). The result is rms error.

The available modes <mode> are:

- [0] TSfc4 Use TS framebuffer (de)compression 4bpp (default)
- [1] TSfc6 Use TS framebuffer (de)compression 6bpp

- [2] TSfc6A Use TS framebuffer (de)compression 6bpp with alpha channel support
- [3] TStc4 Use TS texture (de)compression 4bpp
- [4] TStc6 Use TS texture (de)compression 6bpp
- [5] TStc6A Use TS texture (de)compression 6bpp with alpha channel support

Examples:

If a source image input.png needs compression in TSC™4 format, the command is:

```
./tsc -c -mode TSfc4 input .png output .tsfc4
```

or:

```
./tsc -c -mode [0] input .png output .tsfc4
```

If a source image in TSC™4 format input.tsfc4 needs decompression in 256x256 png format, the command is:

```
./tsc -d 256 x256 -mode TSfc4 input .tsfc4 output .png
```

or:

```
./tsc -d 256 x256 -mode [0] input .tsfc4 output .png
```

To compare two images in png format (input1.png and input2.png), the command is:

```
./tsc -e input1 .png input2 .png
```

The NEMA®| PIX-Presso can be used to aid the development of applications that use TSC formats. NEMA®| PIX-Presso is a utility for converting images to formats suitable for low power embedded devices and can produce TSC™4 and TSC™6 compressed textures which the developer can then load to memory.

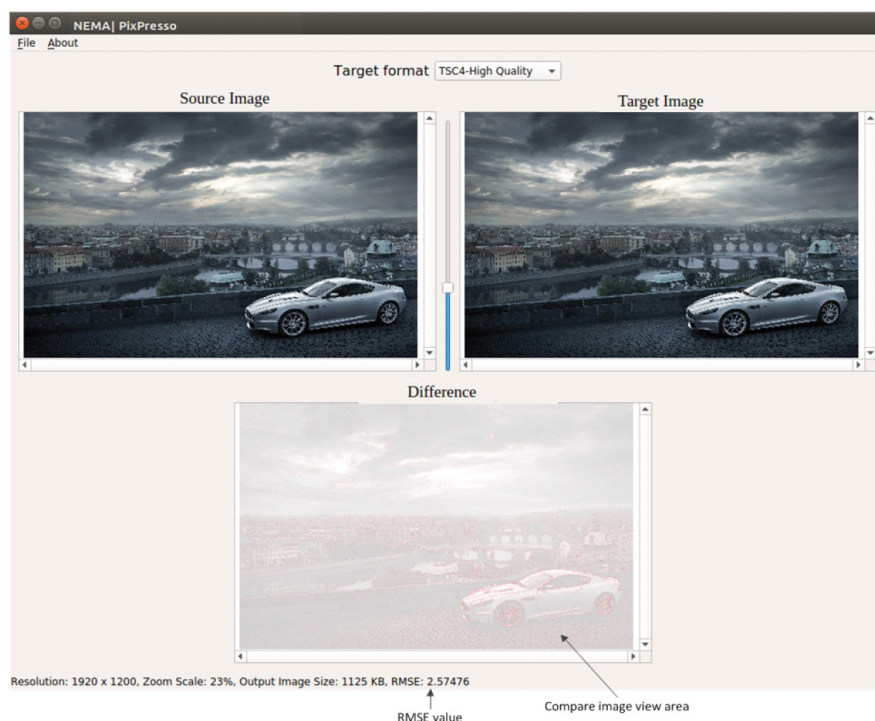


Figure 123. NEMA®| PIX-Presso State after a Conversion

NEMA®| PIX-Presso emulates the h/w compression and decompression algorithms, so the developer can check the resulting quality of the image and decide if TSC compression can be used for the specific application, or which TSc algorithm (TSC™4 or TSC™6) is more appropriate.

22.6 GFX Library API Guidelines

This section provides an overview of the GFX Library API, including some simple guidelines on how to use the library for development purposes. The information included in this section can be found elsewhere in this chapter but spans across several paragraphs, which makes it daunting to get started. This section introduces some simple code examples using the primitives of the GFX library to draw a graphics image in a timely manner.

22.6.1 Command Lists

A Command List (CL) is considered to be one of the most important features of the GPU. CL usage facilitates GPU and CPU decoupling, while its inherent re-usability greatly contributes to the decrease of the computational effort of the CPU. This approach renders the overall architecture capable of drawing complicated scenes while keeping the CPU workload to the very minimum.

The design principles of CLs allow developers to extend the features of their application while optimizing its functionality at the same time. For instance, a CL is capable of jumping to another CL, thus forming a chain of seamlessly interconnected commands. In addition, a CL is able to branch to another CL and once the branch execution is concluded, resume its functionality after the branching point.

The GFX library helps developers to easily take advantage of all these features through certain basic function calls that trigger the whole spectrum of CL capabilities. A short presentation of the most fundamental subset of them is listed in the following sections.

22.6.1.1 Create

The most straightforward command for initiating a simple coding example is the "Create" command which is listed below.

```
nema_cmdlist_t nema_cl_create(void)
```

This fundamental command allocates and initializes a new Command List for later use.

22.6.1.2 Bind

```
nema_cl_bind_cmdlist(nema_cmdlist_t *cl)
```

This command sets the referred Command List as active. From that point on, each subsequent drawing call will incrementally be incorporated in the active Command List. At any time, all drawing operations should be called when there is a bound Command List.

22.6.1.3 Unbind

```
nema_cl_unbind_cmdlist(void)
```

Unbind the currently bound Command List.

22.6.1.4 Submit

```
nema_cl_submit_cmdlist(nema_cmdlist_t *cl)
```

Submit the referred Command List for execution. If this CL is currently the one that is bound, this call unbinds it. When a CL is submitted for execution, it should never be altered until it finishes execution. Writing in such a CL results in undefined behavior.

A typical routine for drawing would be the following:

```
nema_cmdlist_t cl = nema_cl_create();    // Create a new CL
nema_cl_bind_cmdlist(&cl);              // Bind it

/* Drawing Operations */                // Draw scene

nema_cl_unbind_cmdlist();               // Unbind CL (optionally)
nema_cl_submit_cmdlist(&cl);            // Submit CL for execution
```

22.6.2 Binding Textures

Every drawing operation should have an effect on a given destination texture. The texture must reside in some memory space which is visible to the GPU.

```
void nema_bind_dst_tex(uint32_t baseaddr_phys,
                      uint32_t width, uint32_t height,
                      nema_tex_format_t format, int32_t stride)
```

The above function binds a texture to serve as destination. The texture's attributes (GPU address, width, height, format and stride) are written inside the bound CL. Each subsequent drawing operation will have an effect on this destination texture.

Most common graphics operations include some kind of image blitting (copying), like drawing a background image, GUI icons or even font rendering. The following command binds a texture to be used as foreground:

```
void nema_bind_src_tex(uint32_t baseaddr_phys,
                      uint32_t width, uint32_t height,
                      nema_tex_format_t format, int32_t stride, nema_tex_mode_t
                      mode);
```

This function call has a very similar functionality to the `NemaGFX_bind_dst_tex`. It has one extra argument, `NEMA_tex_mode_t mode`, that determines how to read a texture (point/bilinear sampling, wrapping mode etc).

The above example can now be extended as follows:

```
nema_cmdlist_t cl = nema_cl_create();    // Create a new CL
nema_cl_bind_cmdlist(&cl);              // Bind it

// Bind Destination Texture:
nema_bind_dst_tex(DST_IMAGE,           // Destination address
                 320, 240,              // width, height
```

```

        NEMA_RGBA8888,      // Image format (32bit, rgba)
        320*4);           // Stride in bytes (width*4 bytes per pixel)

// Bind Foreground Texture:
nema_bind_src_tex(SRC_IMAGE, // Source address
                 320, 240,    // width, height
                 NEMA_RGBA8888, // Image format (32bit, rgba)
                 320*4,      // Stride in bytes (width*4 bytes per pixel)
                 NEMA_FILTER_PS); // Do point sampling (default option)

/* Drawing Operations */      // Draw scene

nema_cl_unbind_cmdlist();     // Unbind CL (optionally)
nema_cl_submit_cmdlist(&cl);  // Submit CL for execution

```

22.6.3 Clipping

When drawing a scene, it is often necessary to be able to define a rectangular area that the GPU is allowed to draw. This way, if some parts of a primitive (e.g. a triangle) falls outside the clipping area, that part is not going to be drawn at all, assuring correctness, better performance and improved power efficiency. The Clipping Rectangle can be defined as follows:

```
void nema_set_clip(int32_t x, int32_t y, int32_t w, int32_t h)
```

This function defines a Clipping Rectangle whose upper left vertex coordinates are (x, y) and its dimensions are w·h.

The default Clipping Rectangle usually is the entire canvas. In the above examples, we used textures with dimensions of 320x240. So, adding Clipping would result the following:

```

nema_cmdlist_t cl = nema_cl_create(); // Create a new CL
nema_cl_bind_cmdlist(&cl);           // Bind it

// Bind Destination Texture:
nema_bind_dst_tex(DST_IMAGE, // Destination address
                 320, 240,    // width, height
                 NEMA_RGBA8888, // Image format (32bit, rgba)
                 320*4);      // Stride in bytes (width*4 bytes per pixel)

// Bind Foreground Texture:
nema_bind_src_tex (SRC_IMAGE, // Source address
                 320, 240,    // width, height
                 NEMA_RGBA8888, // Image format (32bit, rgba)
                 320*4,      // Stride in bytes (width*4 bytes per pixel)
                 NEMA_FILTER_PS); // Do point sampling (default option)

nema_set_clip(0, 0, 320, 240); // Define a 320x240 Clipping Rectangle

/* Drawing Operations */      // Draw scene

nema_cl_unbind_cmdlist();     // Unbind CL (optionally)
nema_cl_submit_cmdlist(&cl);  // Submit CL for execution

```

22.6.4 Blending - Programming the Core

When building a graphical interface, the developer has to define what would be the result of drawing a pixel on the canvas. Since the canvas already contains the previous drawn scene, there must be a consistent way to determine how the source or foreground color (the one that is going to be drawn) will blend with the destination or background color that is already drawn. The source pixel can be fully opaque, thus will be drawn over the destination one, or it can be translucent and the result would be a blend of both the source and destination colors.

For example, blitting a background image of a GUI would require the Source Texture to cover entirely whatever is already drawn on the canvas. Afterwards, blitting an icon would require the background to be partially visible on the translucent areas of the icon. In order to make this possible, the GFX library incorporates a powerful set of predefined blending modes that allow the developer to build functional and eye catching applications:

```
void nema_set_blend_fill(nema_blend_mode_t blending_mode)
void nema_set_blend_blit(nema_blend_mode_t blending_mode)
```

These two functions refer to blending when filling a primitive (e.g. triangle) with a color or when blitting a texture respectively.

The previous example, after setting the correct blending mode for blitting a background texture, would evolve to the following:

```
nema_cmdlist_t cl = nema_cl_create();    // Create a new CL
nema_cl_bind_cmdlist(&cl);              // Bind it

// Bind Destination Texture:
nema_bind_dst_tex(DST_IMAGE,            // Destination address
                 320, 240,              // width, height
                 NEMA_RGBA8888,         // Image format (32bit, rgba)
                 320*4);               // Stride in bytes (width*4 bytes per pixel)

// Bind Foreground Texture:
nema_bind_src_tex (SRC_IMAGE,           // Source address
                 320, 240,              // width, height
                 NEMA_RGBA8888,         // Image format (32bit, rgba)
                 320*4,                // Stride in bytes (width*4 bytes per pixel)
                 NEMA_FILTER_PS);       // Do point sampling (default option)

nema_set_clip(0, 0, 320, 240);          // Define a 320x240 Clipping Rectangle

nema_set_blend_blit(NEMA_BL_SRC);       // Program the Core to draw the source color
// without blending it with the destination
// texture

/* Drawing Operations */                // Draw scene

nema_cl_unbind_cmdlist();               // Unbind CL (optionally)
nema_cl_submit_cmdlist(&cl);            // Submit CL for execution
```

22.6.5 Drawing

Finally, after setting up the above, the CL contains all the information needed to blit an image or fill a Geometric Primitive with color. The GFX library has a rich set of functions to do that. For the example

above, let's assume that we need to draw a background 320x240 image, starting at screen coordinate (0,0) (the upper left corner of the canvas), and then draw a red rectangle that starts at point (20, 30) with dimensions 100x200:

```
nema_cmdlist_t cl = nema_cl_create();    // Create a new CL
nema_cl_bind_cmdlist(&cl);              // Bind it

// Bind Destination Texture:
nema_bind_dst_tex(DST_IMAGE,            // Destination address
                  320, 240,              // width, height
                  NEMA_RGBA8888,         // Image format (32bit, rgba)
                  320*4);                // Stride in bytes (width*4 bytes per pixel)

// Bind Foreground Texture:
nema_bind_src_tex (SRC_IMAGE,            // Source address
                  320, 240,              // width, height
                  NEMA_RGBA8888,         // Image format (32bit, rgba)
                  320*4,                 // Stride in bytes (width*4 bytes per pixel)
                  NEMA_FILTER_PS);       // Do point sampling (default option)

nema_set_clip(0, 0, 320, 240);           // Define a 320x240 Clipping Rectangle

nema_set_blend_blit(NEMA_BL_SRC);        // Program the Core to draw the source
// texture without blending it with the
// destination texture
nema_blit(0, 0);                         // Blit the bound Source Texture to
// Destination Texture

nema_set_blend_fill(NEMA_BL_SRC);        // Program the Core to fill the Geometric
// Primitive without blending it with the
// destination texture
nema_fill_rect(20, 30, 100, 200, RED);   // Fill a rectangular area with red color

nema_cl_unbind_cmdlist();                // Unbind CL (optionally)
nema_cl_submit_cmdlist(&cl);             // Submit CL for execution
```

The overall process described in the previous paragraphs, produces the output presented in the following figures.



Figure 124. Original Empty Framebuffer

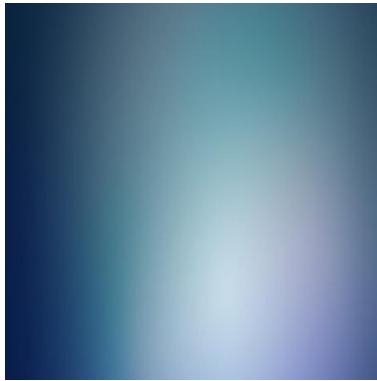


Figure 125. Image Background

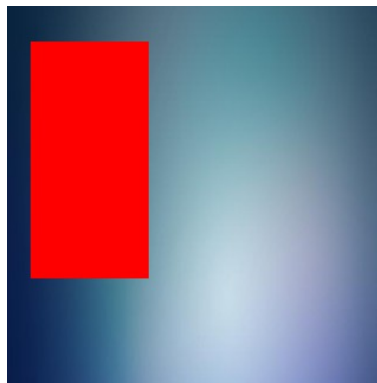


Figure 126. Final Output of the Drawing Process

22.7 Color Modes and Binding Textures

22.7.1 Color Modes

The GPU natively support a large set of texture formats, therefore they are capable of performing fast read and write operations by executing on the fly color conversion/decompression. Native formats expand from full 32-bit RGBA to 1-bit black and white colors, together with an optional proprietary compressed 4-bit-per-pixel lossy format and can all be used as source or destination textures. The list of all supported formats is presented in Table 124 for quick reference purposes.

<u>Color Mode</u>	<u>Description</u>
RGBX8888	32-bit color with no transparency
RGBA8888	32-bit color with transparency
XRGB8888	32-bit color with no transparency
ARGB8888	32-bit color with transparency
BGRA8888	32-bit color with transparency
BGRX8888	32-bit color with transparency
RGBA5650	16-bit color with no transparency
RGBA5551	16-bit color with 1-bit transparency
RGBA4444	16-bit color with transparency
RGBA3320	8-bit color with no transparency
L8	8-bit gray scale (luminance) color
A8	8-bit translucent color
L2	2-bit grayscale (luminance) color
L4	4-bit grayscale (luminance) color
BW1	1-bit color (black or white)
UYVY	UYVY color
TSC™ 4, TSC 6/6A, TSC 12/12A	4-bit / 6-bit / 12-bit proprietary compressed
YUV	YUV
Z24_8	32-bit (24+4) depth and stencil
Z16	16-bit depth

Table 124: Supported Formats

NOTE

Use of RGBA4444 or ARGB4444 input color modes are supported, but the DC does not support any of RGB4444 interfaces, including NEMADC_RGBA4444 and NEMADC_ARGB4444. NEMADC_RGBA4444 interface is supported.

22.7.2 Binding Textures

The color modes discussed in 22.7.1 can be used for both source and destination textures. The GPU incorporate 4 texture slots allowing 4 textures to be bound simultaneously. This means that the hardware allows a single Shader to read from and/or write to 4 different textures. These textures have to be bound before the Shader is submitted for execution. The GFX library uses pre-assembled Shaders to perform blending operations. These Shaders are built upon the conventions of Table 125.

<u>Texture Slot</u>	<u>Texture Usage</u>
NEMA_TEX0	Destination/Background Texture
NEMA_TEX1	Foreground Texture
NEMA_TEX2	Background Texture
NEMA_TEX3	DepthBuffer

Table 125: Shader conventions

For further clarifying the aforementioned conventions, let's assume the following example: We need to draw the scene shown in Figure 127, consisted of a background image and two icons. The scene requires the 3 source textures shown in Figure 128, and a Framebuffer, i.e., the destination texture.

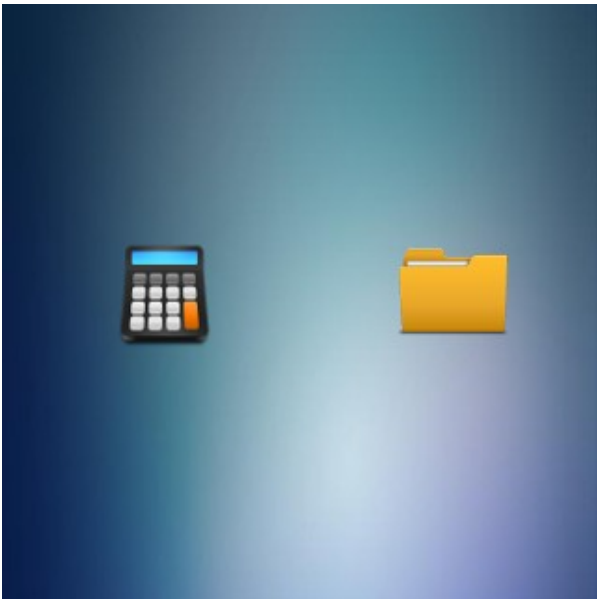


Figure 127. Rendered Scene with Two Icons



Figure 128. Scene Textures

Scenario 1

The scene will be drawn in 3 passes: first draw the background, then draw Icon_0 and last draw Icon_1.

1. First Pass: Draw Background
 - A. Bind the Framebuffer to NEMA_TEX0 slot
 - B. Bind the Background texture to NEMA_TEX1 slot
 - C. Set corresponding blending mode (NEMA_BL_SRC)
 - D. Blit NEMA_TEX1 slot to NEMA_TEX0 slot
2. Second Pass: Draw Icon_0
 - A. Bind the Framebuffer to NEMA_TEX0 slot
 - B. Bind Icon0 to NEMA_TEX1 slot
 - C. Set corresponding blending mode (e.g. NEMA_BL_SRC_OVER)
 - D. Blit NEMA_TEX1 slot to NEMA_TEX0 slot
3. Third Pass: Draw Icon_1
 - A. Bind the Framebuffer to NEMA_TEX0 slot
 - B. Bind Icon1 to NEMA_TEX1 slot
 - C. Set corresponding blending mode (e.g. NEMA_BL_SRC_OVER)
 - D. Blit NEMA_TEX1 to NEMA_TEX0 slot

If we build a single Command List for the above operations, we need to bind the Framebuffer only once, in the beginning of the draw process. This sequence will result in 3 blitting operations. However, there is a more efficient approach, described in Scenario 2.

Scenario 2

The scene will be drawn in 2 passes: first draw Icon_0 on top of the background and then draw Icon_1.

1. First Pass: Draw Icon_0 on top of the Background

- A. Bind the Framebuffer to NEMA_TEX0 slot
 - B. Bind Icon0 to NEMA_TEX1 slot
 - C. Bind the Background texture to NEMA_TEX2 slot
 - D. Set corresponding blending mode (e.g. NEMA_BL_SRC_OVER)
 - E. Blit NEMA_TEX1 on top of NEMA_TEX2 slot to NEMA_TEX0 slot
2. Second Pass: Draw Icon_1
- A. Bind the Framebuffer to NEMA_TEX0 slot
 - B. Bind Icon1 to NEMA_TEX1 slot
 - C. Set corresponding blending mode (e.g. NEMA_BL_SRC_OVER)
 - D. Blit NEMA_TEX1 slot to NEMA_TEX0 slot

22.7.2.1 Texture Binding Functions

Use this function to bind the destination texture. It implies binding to NEMA_TEX0 slot:

```
void nema_bind_dst_tex(uint32_t baseaddr_phys,
                      uint32_t width,
                      uint32_t height,
                      nema_tex_format_t format,
                      int32_t stride)
```

Use the following function to bind the foreground (source) texture. This is needed only for Blit operations. Fill operations don't have a source texture. This function implies binding to NEMA_TEX1 slot:

```
void nema_bind_src_tex(uint32_t baseaddr_phys,
                      uint32_t width,
                      uint32_t height,
                      nema_tex_format_t format,
                      int32_t stride)
```

The following function binds a background texture to NEMA_TEX2 slot. This is needed when the Blending Mode to be used does not use the destination texture (NEMA_TEX0) as background texture at the blending operation.

```
void nema_bind_src2_tex(uint32_t baseaddr_phys,
                      uint32_t width,
                      uint32_t height,
                      nema_tex_format_t format,
                      int32_t stride,
                      nema_tex_mode_t mode)
```

22.8 Geometry Primitives

In computer graphics, geometry primitives are the basic geometric shapes that a system can draw. In the GPU architecture, these primitives are generated by the Rasterizer module. The Rasterizer generates the fragments contained inside the primitive and feeds them to the Programmable Core for processing. The GPU can draw the following Geometry Primitives:

- Points
- Lines
- Filled Triangles
- Filled Rectangles
- Filled Quadrilaterals

All the aforementioned primitives can be processed by the Programmable Core to do simple operations (e.g. filling with a constant color or gradient, blitting etc) or more advanced ones (e.g. blurring, edge detection etc). The GPU functionality can be extended through software to draw:

- Triangles
- Rectangles
- Polygons
- Filled Polygons
- Triangle Fans
- Triangle Strips
- Circles
- Filled Circles
- Arcs
- Rounded Rectangles

22.9 Blending

Alpha blending is a basic process in computer graphics. It refers to a convex combination of two colors, a translucent source (foreground) and a destination (background) one, allowing transparency effects. The basic blending algorithms described by Porter and Duff in [[XREF](#)>2], define a set of mathematical operations for the Color channels (RGB) and the Alpha (transparency) channel of a fragment. Blending process is essential for rendering fonts and/or creating GUIs. In the graphics pipeline, blending is carried-out in the Graphics Core.

22.9.1 Blending in the Graphics Core

The Graphics Core is a programmable VLIW processor, which allows rapid calculations between colors. Normally, such per-fragment calculations are an overwhelming computational burden for a CPU or SoC. The Graphics Core is programmed through instructions in binary form, called Shaders. However, in embedded applications, running a compiler for creating these kinds of Shaders is not a realistic scenario. Therefore, the GFX library provides a lightweight and (user-friendly) easy to use interface that employs pre-assembled commands to create a powerful set of blending algorithms.

The Graphics Core can be programmed through the following functions:

```
void nema_set_blend_fill(nema_blend_mode_t blending_mode)
void nema_set_blend_blit(nema_blend_mode_t blending_mode)
```

These functions are defined in NemaGFX_blender.h file. They should be used on fill and blit operations respectively. The `blending_mode` argument is possible to be a predefined blending mode or a more refined User Defined Mode.

22.9.2 Notations and Conventions

Blending requires a series of calculations between the source (foreground) and destination (background) color fragments for producing the final color, which will be written in memory. The Color and Alpha channels are noted as follows:

- *Sc*: Source Color
- *Sa*: Source Alpha
- *Sf*: Source Blend Factor (multiplier)
- *Dc*: Destination Color
- *Da*: Destination Alpha
- *Df*: Destination Blend Factor (multiplier)
- *Fc*: Final Color
- *Fa*: Final Alpha
- *Cc*: Constant Color
- *Ca*: Constant Alpha

The Color and Alpha values range from 0 to 1, therefore each calculation result is also clamped to the same range. For consistency reasons Color and Alpha calculations are always described separately, as in some cases these calculations are not identical. When a constant color is used (noted as *Cc* and *Ca*), it can be set using the following function:

```
void nema_set_const_color(uint32_t rgba)
```

22.9.3 Predefined Blending Modes

Predefined Blending Modes is a set of commonly used modes, each implying different calculations between the source and destination colors for Color (RGB) channel and Alpha channel respectively. Table

1304 presents the entire list of the available Predefined Blending Modes along with the corresponding calculations that produce the final fragment color. Table 126 shows the result for each Predefined Blending Mode.

<u>Predefined Blending Modes</u>	<u>RGB</u>	<u>ALPHA</u>
NEMA_BL_SIMPLE	$Sc * Sa + Dc * (1 - Sa)$	$Sa * Sa + Da * (1 - Sa)$
NEMA_BL_CLEAR	0	0
NEMA_BL_SRC	Sc	Sa
NEMA_BL_SRC_OVER	$Sc + Dc * (1 - Sa)$	$Sa + Da * (1 - Sa)$
NEMA_BL_DST_OVER	$Sc * (1 - Da) + Dc$	$Sa * (1 - Da) + Da$
NEMA_BL_SRC_IN	$Sc * Da$	$Sa * Da$
NEMA_BL_DST_IN	$Dc * Sa$	$Da * Sa$
NEMA_BL_SRC_OUT	$Sc * (1 - Da)$	$Sa * (1 - Da)$
NEMA_BL_DST_OUT	$Dc * (1 - Sa)$	$Da * (1 - Sa)$
NEMA_BL_SRC_ATOP	$Sc * Da + Dc * (1 - Sa)$	$Sa * Da + Da * (1 - Sa)$
NEMA_BL_DST_ATOP	$Sc * (1 - Da) + Dc * Sa$	$Sa * (1 - Da) + Da * Sa$
NEMA_BL_ADD	$Sc + Dc$	$Sa + Da$
NEMA_BL_XOR	$Sc * (1 - Da) + Dc * (1 - Sa)$	$Sa * (1 - Da) + Da * (1 - Sa)$

Table 126: Predefined Blending Modes

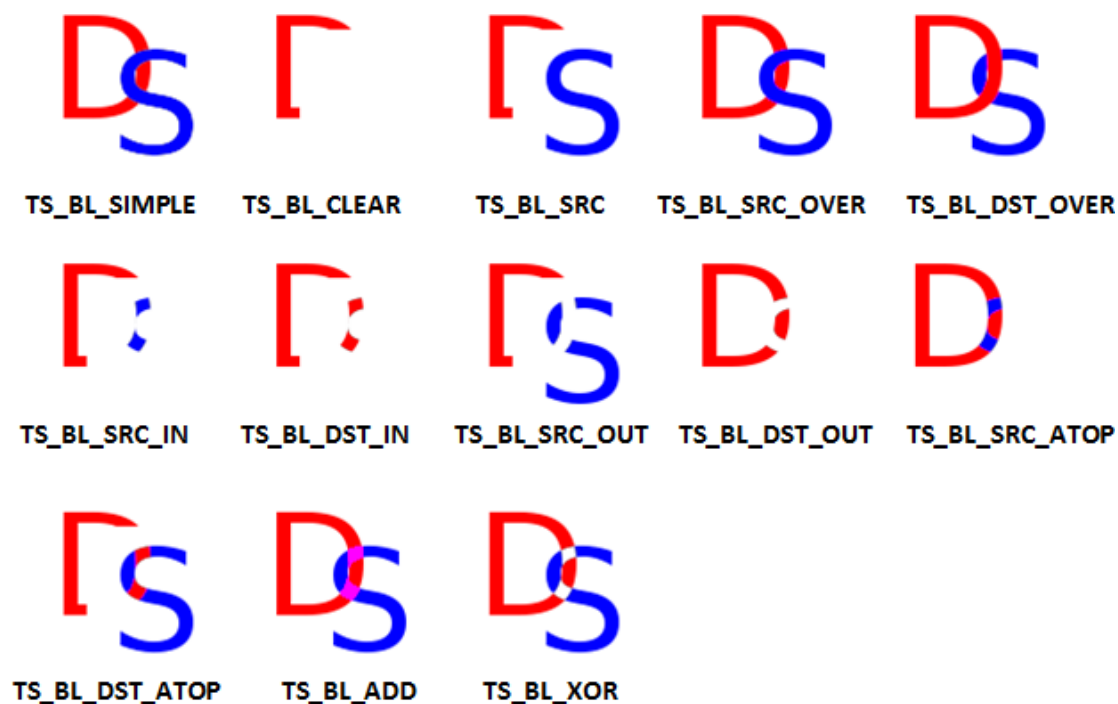


Figure 129. Predefined Blending Modes

For instance, drawing a translucent red rectangle would require the following calls:

```
nema_set_blend_fill(NEMA_BL_SIMPLE);
nema_fill_rect(52, 52, 14, 4, nema_rgba(0xff, 0, 0, 0x80));
```

Tip: When in doubt, usually the NEMA_BL_SIMPLE blending mode is the safest choice.

The overall process starts with an empty Framebuffer, shown in Figure 130. Next, the process continues by blitting the textures shown in Figure 131.

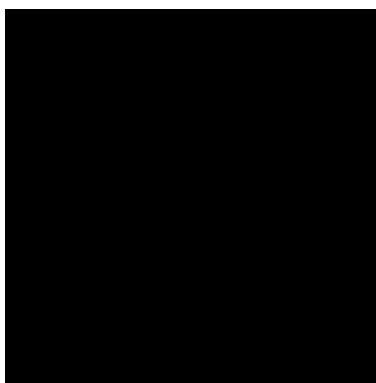


Figure 130. Original Framebuffer before Blending



Figure 131. Scene Textures

22.9.4 User Defined Modes

Developers are able to create custom blending modes by using different factors for the source (foreground) and destination (background) color, through the following function:

```
nema_blend_mode_t nema_blending_mode(nema_blend_factor_t src,
                                     nema_blend_factor_t dst,
                                     nema_blend_op_t ops)
```

The ops argument of the above function refers to additional operations, presented in Section 22.9.5 on page 376, therefore for the time being should be set to zero (0). The calculations result to the final color, using the following two equations:

$$F_c = S_c \cdot S_f + D_c \cdot D_f \quad (1)$$

$$F_a = S_a \cdot S_f + D_a \cdot D_f \quad (2)$$

The available Blend Factors are listed in Table 127. Figure 132 shows the available custom blending modes. As a result, the previous example is possible to be rewritten as:

```
nema_set_blend_fill(nema_blending_mode(NEMA_BF_SRCALPHA, NEMA_BF_INVSRCALPHA, 0));
nema_fill_rect(52, 52, 14, 4, nema_rgba(0xff, 0, 0, 0x80));
```

Blend Factors (S_f or D_f)

NEMA_BF_ZERO	0
NEMA_BF_ONE	1
NEMA_BF_SRCCOLOR	S_c
NEMA_BF_INVSRCOLOR	$(1 - S_c)$
NEMA_BF_SRCALPHA	S_a
NEMA_BF_SRC_INVSRCALPHA	$(1 - S_a)$
NEMA_BF_DESTALPHA	D_a

NEMA_BF_INVDESTALPHA	$(1 - D_a)$
NEMA_BF_DESTCOLOR	D_c
NEMA_BF_INVDESTCOLOR	$(1 - D_c)$
NEMA_BF_CONSTCOLOR	C_c
NEMA_BF_CONSTALPHA	C_a

Table 127: Blend Factors

DESTINATION SOURCE	TS_BF_ZERO	TS_BF_ONE	TS_BF_SRCOLOR	TS_BF_INVSRCOLOR	TS_BF_SRCALPHA	TS_BF_INVSRCALPHA	TS_BF_DESTALPHA	TS_BF_INVDESTALPHA	TS_BF_DESTCOLOR	TS_BF_INVDESTCOLOR
TS_BF_ZERO	I	D	I :	D :	I :	D :	D	I	D	I
TS_BF_ONE	I S	D S	I S	D S	I S	D S	D S	I S	D S	I S
TS_BF_SRCOLOR	I S	D S	I S	D S	I S	D S	D S	I S	D S	I S
TS_BF_INVSRCOLOR	I	D	I :	D :	I :	D :	D	I	D	I
TS_BF_SRCALPHA	I S	D S	I S	D S	I S	D S	D S	I S	D S	I S
TS_BF_INVSRCALPHA	I	D	I :	D :	I :	D :	D	I	D	I
TS_BF_DESTALPHA	I :	D	I :	D	I :	D	D	I :	D	I :
TS_BF_INVDESTALPHA	I S	D S	I S	D S	I S	D S	D S	I S	D S	I S
TS_BF_DESTCOLOR	I :	D	I :	D	I :	D	D	I :	D	I :
TS_BF_INVDESTCOLOR	I S	D S	I S	D S	I S	D S	D S	I S	D S	I S

Figure 132. User-defined Blending Modes

22.9.5 Additional Operations

The GFX library allows the following operations, which can be applied together with the previously mentioned blending modes through the aforementioned function:


```
nema_blend_mode_t nema_blending_mode(nema_blend_factor_t src,  
                                     nema_blend_factor_t dst,  
                                     nema_blend_op_t ops)
```

The additional supported operations are listed in Table 128. An example of the overall process can be found in Figure 133 and Figure 134.

<u>ops Arguments</u>	<u>Description</u>
SRC_MODULATE_A	Multiply source alpha channel with <i>Ca</i> constant before blending. <i>Ca</i> is defined by calling <code>NemaGFX_set_const_color()</code> .
SRC_FORCE_A	Replace source alpha channel with <i>Ca</i> before blending. Overrides SRC_MODULATE_A option. <i>Ca</i> is defined by calling <code>NemaGFX_set_const_color()</code> .
SRC_COLORIZE	Multiply source color channels (RGB) with <i>Cc</i> before blending. <i>Cc</i> is defined by calling <code>NemaGFX_set_const_color()</code> .
SRC_COLORKEY	Ignore fragment when source color matches the source color key, which is defined by calling <code>NemaGFX_set_src_color_key()</code> .
DST_COLORKEY	Ignore fragment when destination color matches the destination color key, which is defined by calling <code>NemaGFX_set_dst_color_key()</code> .

Table 128: ops Arguments



Figure 133. Source Textures

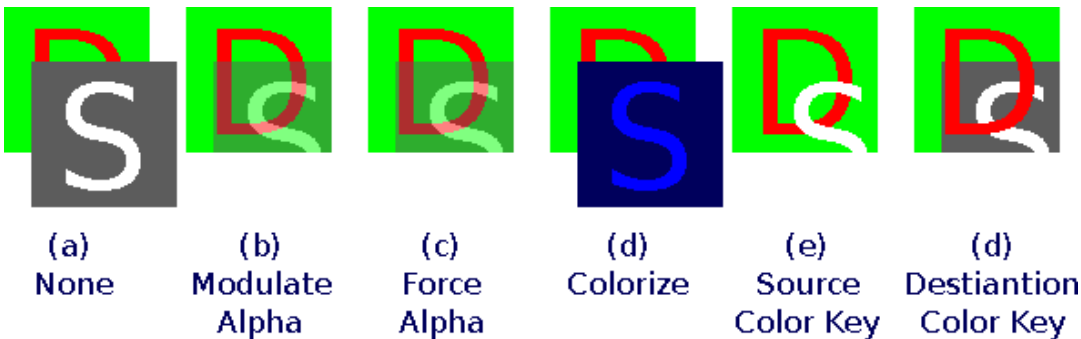


Figure 134. Additional Operations Example

22.10 Fonts

Drawing text on the screen is an important element of any Graphical User Interface. To draw a string you will need a Typeface, the text to be drawn and some attributes on how the text is to be displayed. Typefaces are sourced in TrueType (TTF) file type, which contains scalable representations of typefaces described as vector curves. Scalable fonts are converted to raster fonts (bitmap fonts) by rasterization to a particular size and format. Raster fonts are drawn on the screen as a series of images with each letter drawn after the other using the correct letter width. To facilitate this process, the GFX library handles text display and alignment using special functions.

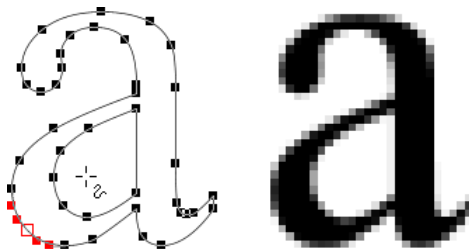


Figure 135. Vector and Bitmap Fonts

The first step is to convert a TrueType font to Bitmap font. This is done off-line with the NemaGFX_FontUtil tool.

```
./nema_font_convert <font.ttf> -size <size> -format <A1|A2|RGBA>
-output <font\_bitmap>
```

The conversion generates a font_bitmap.h and font_bitmap.bin file at the specified size and format. The <size> parameter defines the height of the font. Fonts can be mono-spaced (fixed-width) or not.

Typefaces can be converted to the following formats:

A1:1 bit per pixel

A2:2 bits per pixel antialiased

A4:4 bits per pixel antialiased

A8:8 bits per pixel antialiased RGBA32:Sub-pixel Antialiased

Before drawing text we need to bind the data structure of a typeface.

```
void nema_bind_font (font_t *fontdata)
```

The text can be drawn through the following function:

```
void nema_draw_text (char *text, int *x, int *y, int clip_x, int clip_y, int clip_w,  
    int clip_h, uint32_t color, font_attr_t attributes)
```

The arguments of the above function refer to:

- The text to be drawn
- The (x, y) screen coordinates that the text should be drawn to
- The clipping box (x, y, width, height) in case some part of the text should be hidden
- The color of the text
- Additional font attributes

The additional font attributes are:

- JUST_LEFT:Justify Left
- JUST_CENTRE:Justify Centre
- JUST_RIGHT:Justify Right
- WRAP:Wrap to a new line if exceeding

Arguments *x, *y return the position of the cursor after the text is drawn.

22.11 GFX Library Platform Porting

The GFX library has been designed to be easily portable in a variety of different platforms. This includes systems with or without an operating system. In order to port the GFX library successfully, one must take into account the target platform and adapt the HAL (Hardware Abstraction Layer) accordingly.

The HAL is a thin layer of the library that:

- Communicates directly with the system hardware and the GPU drivers (when drivers are available)
- Performs the communication between the host and the GPU (access to the GPU registers)
- Handles interrupts
- Implements the memory management scheme (memory allocation, deallocation, mapping and un-mapping)

22.11.1 Platform Specific HAL

Each target platform has:

- A *nema_sys_defs.h* header file located in *NemaGFX_SDK/common/platforms* folder
- A separate *nema_hal.c* file, located in *NemaGFX_SDK/NemaGFX/platforms/* folder

In order to port the GFX library to a new platform, it is advised that the corresponding source files of an already ported platform are used as templates.

22.11.2 *nema_sys_defs.h*

nema_sys_defs.h contains all the global definitions and inclusions that are platform specific. For example, the GFX library uses a set of integer types defined inside the C standard *stdint.h* header. If the platform's compiler supports the *stdint.h*, then it should be included in the *nema_sys_defs.h* header file. If the compiler does not support the *stdint.h*, then the following types should be defined:

- *int8_t*, *uint8_t*,
- *int16_t*, *uint16_t*,
- *int32_t*, *uint32_t*,
- *int64_t*, *uint64_t*

If the GFX library is compiled for a platform that runs multiple processes and/or multiple threads, the following definitions should be added respectively:

```
#define NEMA_MULTI_PROCESS
#define NEMA_MULTI_THREAD
```

22.11.3 *nema_hal.c*

nema_hal.c contains all the platform specific functions that implement hardware register read/write operations, interrupt handling, memory management and mutex support. It acts as a thin layer which incorporates all the platform dependent portions of a the GFX library implementation.

22.11.3.1 System Initialization

The *nema_init()* function initializes the GFX library and calls the *nema_sys_init()* function which is responsible for the system initialization. The system initialization includes:

- GPU register memory mapping

- Graphics Memory mapping
- Mutex initialization
- Ring Buffer allocation and initialization

22.11.3.2 Register Read/Write

The host CPU writes to and reads from the GPU's configuration registers. The functions *nema_reg_read()* and *nema_reg_write()* are used for the communication of the CPU with the GPU.

When the target platform does not need memory virtualization (e.g. bare-metal systems), the access to the GPU's registers is straightforward by using the register's physical memory address. The only prerequisite in this case, is the appropriate memory mapping of the GPU registers to the system's main memory.

The following examples, illustrate the register read and write operations for bare-metal systems.

```
uint32_t nema_reg_read(uint32_t reg) {
    uint32_t *ptr = (uint32_t *) (nema_regs + reg);
    return *ptr;
}

void nema_reg_write(uint32_t reg, uint32_t value) {
    uint32_t *ptr = (uint32_t *) (nema_registers_base_addr + reg);
    *ptr = value;
}
```

On platforms that support virtual memory (i.e. Linux, Android), the GPU registers' physical addresses must be mapped to the virtual memory addresses before an access is attempted. This can be performed in three ways.

The first one is to utilize the GPU driver (if applicable), and perform the memory mapping using the *mmap* system call.

```
nema_regs_base_virt = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED, nema_fd, 0);
```

The second way in Linux systems, is to use the */dev/mem* instead of the *unema* driver as shown in the following example.

```
// Map and initialize Graphics Memory
nema_devmem_mmap(&gmem_base_virt, gmem_base_phys, gmem_size, "VideoMemory");

// Memory map nema registers
nema_devmem_mmap((void *)&nema_regs, nema_regs_base_phys, 0x1000, "NemaRegisters");
```

The third way in Linux systems, is to perform the read/write operations to the registers via respective *IOCTL* calls to the *unema* driver.

```

uint32_t nema_reg_read(uint32_t reg) {
    unema_ioctl_readwrite_t ioctl_rw;
    ioctl_rw.reg = reg;
    ioctl(nema_fd, UNEMA_IOCTL_REG_READ, &ioctl_rw);

    return ioctl_rw.value;
}

void nema_reg_write(uint32_t reg, uint32_t value) {
    unema_ioctl_readwrite_t ioctl_rw;
    ioctl_rw.reg = reg;
    ioctl_rw.value = value;
    ioctl(nema_fd, UNEMA_IOCTL_REG_WRITE, &ioctl_rw);
}

```

22.11.3.3 Interrupt Handling

The interrupt handler is executed when an interrupt is triggered by the GPU. Its purpose is to awaken suspended processes and clear the interrupt.

When the GPU kernel driver is available (Linux and Android systems), the interrupt handler is defined in the unema driver. When the GPU kernel driver is not available (bare-metal and RTOS based systems), the interrupt handler has to be defined in the `nema_hal.c` file.

A typical interrupt handler for bare-metal systems is the following:

```

void irq_handler (void)
{
    //Clear the interrupt
    nema_reg_write(NEMA_INTERRUPT, 0);
}

```

The HAL should implement the `nema_wait_irq` function for interrupt handling. Its purpose is to suspend the process (put to sleep) if the GPU is idle or until the GPU signals an interrupt. Its implementation is platform (CPU) dependent. If the kernel driver is available, then IOCTL calls are used, otherwise it is manually defined according to the CPU platform.

```

void nema_wait_irq(void)
{

```

Typically, the GPU will raise an interrupt when it has finished executing a Command List. The ID of the last Command List that has been executed can be read from the Configuration Register `NEMA_CLID`. The function `nema_wait_irq_cl(int cl_id)` should wait until the content of the `NEMA_CLID` register is greater or equal than the `cl_id` argument.

22.11.3.4 Memory Management

At this stage, the memory management scheme should be implemented. Memory can be considered to consist of two parts: host memory (memory available only to the CPU) and graphics memory (memory available both to the GPU and the CPU).

Host memory can be allocated by using systems' default `malloc` method:

```
//System malloc

void nema_host_free      (void *ptr ) {
    free(ptr);
}

void *nema_host_malloc   (unsigned size) {
    return malloc(size);
}
```

If such a method is not available, the graphics memory allocator can be used:

```
//Think Silicon's graphics memory allocator

void nema_host_free      (void *ptr ) {
    tsi_free(ptr);
}

void *nema_host_malloc   (unsigned size) {
    return tsi_malloc(size);
}
```

Graphics memory is the portion of system memory that the GPU is allowed to have access. In this case, it is necessary that the graphics memory occupies a contiguous physical memory space. On systems that do not support virtual memory (e.g. bare-metal systems), graphics memory can be allocated in the same way as host memory (using *malloc*, *tsi_malloc* or other custom memory allocator).

The GFX library includes the following API calls for memory allocation, deallocation and mapping:

- ^:nema_buffer_create() - Allocate memory
- ^:nema_buffer_create_pool() - Allocate memory from specific memory pool
- ^:nema_buffer_map() - Map allocated memory space for CPU access
- ^:nema_buffer_unmap() - Unmap previously mapped memory space
- ^:nema_buffer_destroy() - Deallocate memory space

A reference example that illustrates this memory allocation scheme can be found in:

Software/NemaGFX_SDK/NemaGFX/platforms/lattice_mico32_no_OS/nema_hal.c.

In this file, functions *nema_buffer_create* and *nema_buffer_destroy* adopt this specific scheme.

When virtual memory is used (e.g. Linux or Android systems), graphics memory should be allocated by the system's contiguous memory allocator (e.g. *ION* for Android, *CMA* for Linux).

Linux based system: The *unema* kernel driver pre-allocates a contiguous physical memory space using Linux *CMA*. When the GFX library is initialized, this memory space is mapped to the process' virtual space.

Android based system: When the GFX library is initialized, the *ION* kernel module is opened. On each subsequent graphics memory allocation or deallocation (*nema_buffer_create* or *nema_buffer_destroy*), the corresponding *IOCTL* to the *ION* module is called.

The GFX library supports multiple memory pools. This can be useful for systems with non-uniform memory hierarchy that have different characteristics (e.g. latency, throughput, etc). If such a feature is not needed, *nema_buffer_create_pool()* can be set to redirect to *nema_buffer_create()*:


```

nema_buffer_t nema_buffer_create_pool(int pool, unsigned size) {
#ifdef NEMA_MULTI_MEM_POOLS //defined in nema_sys_defs.h
    nema_mutex_lock(MUTEX_MALLOC);

    nema_buffer_t bo;
    bo.base_virt = tsi_malloc_pool(pool, size);
    bo.base_phys = (uint32_t) tsi_virt2phys(bo.base_virt);
    bo.size      = size;
    bo.fd        = 0;

    nema_mutex_unlock(MUTEX_MALLOC);
    return bo;
#else
    return nema_buffer_create(size);
#endif
}

```

22.11.3.5 Support for Multi-Process Multi-Threaded systems

The GFX library is designed to support a wide variety of systems, from bare-metal to Linux platforms. These systems might also support multiple processes and/or multiple threads within a process. the GFX library is a graphics API that can manage resource sharing among multiple processes/threads if needed, using mutices and thread local storage (TLS). To support multiple processes, only mutices are necessary. For multiple threads, both mutices and TSL are needed.

The most obvious shared resource is the GPU itself. Multiple processes/threads send work to the GPU using Command Lists (CL). When a CL is executed by the GPU, it is guaranteed that it will not be interrupted by another CL. Each CL also needs to set the entire state of the GPU and not rely on previous CLs. So the only thing that needs to be taken care of, when multiple processes/threads are running, is the submission of a CL for execution. In this case, a simple mutex is used by the library.

The same applies for memory management. When a buffer is created or destroyed, a mutex ensures that no allocation or deallocation is performed by two processes or threads concurrently.

During system initialization, `nema_init()` should call `nema_sys_init()` only once for each process. New threads within the process must not call `nema_init()` again. The `nema_sys_init()` should not reinitialize the memory allocator nor the Ring Buffer, unless no other running process performed the aforementioned initializations.

For multi-threaded environments, `TLS_VAR` should be defined inside `nema_sys_defs.h`. The GFX library uses `TLS_VAR` as a prefix to declare thread local variables. For example, when using GCC the following definition should be added:

```

#ifdef NEMA_MULTI_THREAD
#define TLS_VAR __thread
#else
#define TLS_VAR
#endif

```

22.11.4 Bare Metal Display Control Library

The API for the NEMA functions to communicate with the display controller in bare metal systems is described in the following sections.

22.11.4.1 *nemadc_get_config*

Read Configuration Register

Return Value:

uint32_t: return implemented configuration

22.11.4.2 *nemadc_get_crc*

Read CRC Checksum Register

Return Value:

unsigned int: return crc checksum of last frame for testing

22.11.4.3 *nemadc_init*

Initializes the Display Controller and structures

Return Value:

int: true if Display Controller is found

22.11.4.4 *nemadc_set_bgcolor*

Set Background Color before layer overlays

Arguments:

int: Colour as a 32-bit rgba value

Return Value:

void: void

22.11.4.5 *nemadc_timing*

Sets Display Timing parameters and activates display

Arguments:

int resx: Resolution X

int resy: Resolution Y

int fpx: frontporch X

int fpy: frontporch Y

int blx: blanking X

int bly: blanking Y

int bpx: backporch X

int bpy: backporch Y

Return Value:

void: void

22.11.4.6 *nemadc_set_mode*

Set Operation Mode

Arguments:

int mode: Mode of Operation (reg00)

Return Value:

void: void

22.11.4.7 *nemadc_get_status*

Get STATUS

Arguments:

void: void

Return Value:

uint32_t: Status

22.11.4.8 *nemadc_request_vsync_non_blocking*

Request a VSync Interrupt without blocking

Return Value:

void: void

22.11.4.9 *nemadc_set_layer*

Set the Layer mode. This function can enable a layer and set attributes to it

Arguments:

int layer_no: The layer number

display_layer layer: layer attributes structure

Return Value:

void: void

22.11.4.10 *nemadc_clkdiv*

Sets the built-in Clock Dividers and DMA Line Prefetch. (See Configuration Register 0x4)

Arguments:

int div: Sets Divider 1

int div2: Sets Divider 2

int dma_prefetch: Sets number of lines for the dma to prefetch

Return Value:

void: void

22.11.4.11 *nemadc_clkctrl*

Control the Clock Gaters

Arguments:

int div: Sets the Clock Gater

Return Value:

void: void

22.11.4.12 *nemadc_layer_enable*

Enables a layer

Arguments:

int layer_no: layer to enable

Return Value:

void: void

22.11.4.13 *nemadc_layer_disable*

Disables a layer

Arguments:

int layer_no: layer to disable

Return Value:

void: void

22.11.4.14 *nemadc_get_col_mode*

Read Color Mode Register

Return Value:

uint32_t: return color mode register

22.11.4.15 *nemadc_set_palette*

Sets an entry in the lut8 Palatte Gamma table

Arguments:

int index: Color Index

int colour: 32-bit RGBA colour value or Gamma index

Return Value:

void: void

22.11.4.16 *nemadc_get_palette*

Reads an entry from the lut8 Palatte Gamma table

Arguments:

int index: Color Index

Return Value:

int: Return Colour for given palette index

22.11.4.17 *nemadc_set_layer_gamma_lut*

Sets an entry in the lut8 Palatte Gamma table for a layer

Arguments:

int layer: Layer to set the entry

int index: Color Index

int colour: 32-bit RGBA colour value or Gamma index

Return Value:

void: void

22.11.4.18 *nemadc_get_layer_gamma_lut*

Sets an entry in the lut8 Palatte Gamma table for a layer

Arguments:

int layer: layer to read Gammalut

int index: Color Index

Return Value:

int: Return Pallete index

22.11.4.19 *nemadc_MIPI_out*

Send command or data to MIPI Interface

Arguments:

int cmd: command

Return Value:

void: void

22.11.4.20 *nemadc_MIPI_CFG_out*

Send command or data to MIPI Interface

Arguments:

int cfg: configuration mode

Return Value:

void: void

22.11.4.21 *nemadc_MIPI_in*

Read data from MIPI Interface

Arguments:

void: void

Return Value:

void: void

22.11.4.22 *nemadc_MIPI_updateregion*

Does Partial Update in MIPI

Arguments:

int start_x: start x coordinate

int start_y: start y coordinate

int end_x: end x coordinate

int end_y: end y coordinate

int mode: mode of operation

Return Value:

void: void

22.12 GFX Library Functions

This section provides an overview of the implemented functions of the GFX library.

22.12.1 *nema_blender.h* File Reference

Enumerations

- `enum nema_blend_factors_t {`
 - `NEMA_BF_ZERO,`
 - `NEMA_BF_ONE,`
 - `NEMA_BF_SRCCOLOR,`
 - `NEMA_BF_INVSRCCOLOR,`
 - `NEMA_BF_SRCALPHA,`
 - `NEMA_BF_INVSRCALPHA,`
 - `NEMA_BF_DESTALPHA,`
 - `NEMA_BF_INVDESTALPHA,`
 - `NEMA_BF_DESTCOLOR,`
 - `NEMA_BF_INVDESTCOLOR,`
 - `NEMA_BF_CONSTCOLOR,`
 - `NEMA_BF_CONSTALPHA }`

- `enum nema_blend_mode_t {`
 - `NEMA_BL_SIMPLE,`
 - `NEMA_BL_CLEAR,`
 - `NEMA_BL_SRC,`
 - `NEMA_BL_SRC_OVER,`
 - `NEMA_BL_DST_OVER,`
 - `NEMA_BL_SRC_IN,`
 - `NEMA_BL_DST_IN,`
 - `NEMA_BL_SRC_OUT,`
 - `NEMA_BL_DST_OUT,`
 - `NEMA_BL_SRC_ATOP,`
 - `NEMA_BL_DST_ATOP,`
 - `NEMA_BL_ADD,`
 - `NEMA_BL_XOR }`

- `enum nema_blend_op_t {`
 `NEMA_BLOP_NONE,`
 `NEMA_BLOP_NO_USE_ROPBL,`
 `NEMA_BLOP_DST_CKEY_NEG,`
 `NEMA_BLOP_SRC_PREMULT,`
 `NEMA_BLOP_MODULATE_A,`
 `NEMA_BLOP_FORCE_A,`
 `NEMA_BLOP_MODULATE_RGB,`
 `NEMA_BLOP_SRC_CKEY,`
 `NEMA_BLOP_DST_CKEY,`
 `NEMA_BLOP_MASK}`

Functions

- `static uint32_t nema_blending_mode(nema_blend_factors_t src, nema_blend_factors_t dst, nema_blend_op_t ops)`
 Return blending mode given source and destination blending factors and additional blending operations.
- `void nema_set_blend(uint32_t blending_mode, nema_tex_t dst_tex, nema_tex_t fg_tex, nema_tex_t bg_ex)`
 Set blending mode.
- `static void nema_set_blend_fill(uint32_t blending_mode)`
 Set blending mode for filling.
- `static void nema_set_blend_fill_compose(uint32_t blending_mode)`
 Set blending mode for filling with composing.
- `static void nema_set_blend_blit(uint32_t blending_mode)`
 Set blending mode for blitting.
- `static void nema_set_blend_blit_compose(uint32_t blending_mode)`
 Set blending mode for blitting with composing.
- `void nema_set_const_color(uint32_t rgba)`
 Set constant color.
- `static void nema_set_src_color_key(uint32_t rgba)`
 Set source color key.
- `void nema_set_dst_color_key(uint32_t rgba)`
 Set destination color key.

22.12.1.1 Enumeration Type Documentation

22.12.1.1.1 `enum nema_blend_factors_t`

Enumerator

NEMA_BF_ZERO 0
 NEMA_BF_ONE 1
 NEMA_BF_SRCCOLOR S_c
 NEMA_BF_INVSRCOLOR $(1-S_c)$
 NEMA_BF_SRCALPHA S_a
 NEMA_BF_INVSRCALPHA $(1-S_a)$
 NEMA_BF_DESTALPHA D_a
 NEMA_BF_INVDESTALPHA $(1-D_a)$
 NEMA_BF_DESTCOLOR D_c
 NEMA_BF_INVDESTCOLOR $(1-D_c)$
 NEMA_BF_CONSTCOLOR C_c
 NEMA_BF_CONSTALPHA C_a

22.12.1.1.2 enum nema_blend_mode_t**Enumerator**

<i>NEMA_BL_SIMPLE</i>	$S_a * S_a + D_a * (1 - S_a)$
<i>NEMA_BL_CLEAR</i>	0
<i>NEMA_BL_SRC</i>	S_a
<i>NEMA_BL_SRC_OVER</i>	$S_a + D_a * (1 - S_a)$
<i>NEMA_BL_DST_OVER</i>	$S_a * (1 - D_a) + D_a$
<i>NEMA_BL_SRC_IN</i>	$S_a * D_a$
<i>NEMA_BL_DST_IN</i>	$D_a * S_a$
<i>NEMA_BL_SRC_OUT</i>	$S_a * (1 - D_a)$
<i>NEMA_BL_DST_OUT</i>	$D_a * (1 - S_a)$
<i>NEMA_BL_SRC_ATOP</i>	$S_a * D_a + D_a * (1 - S_a)$
<i>NEMA_BL_DST_ATOP</i>	$S_a * (1 - D_a) + D_a * S_a$
<i>NEMA_BL_ADD</i>	$S_a + D_a$
<i>NEMA_BL_XOR</i>	$S_a * (1 - D_a) + D_a * (1 - S_a)$

22.12.1.1.3 enum nema_blend_op_t**Enumerator**

- *NEMA_BLOP_NONE*
No extra blending operation
- *NEMA_BLOP_NO_USE_ROPBL*

Don't use Rop Blender even if present

- **NEMA_BLOP_DST_CKEY_NEG**

Apply Inverse Destination Color Keying - draw only when dst color doesn't match colorkey

- **NEMA_BLOP_SRC_PREMULT**

Pre-multiply Source Color with Source Alpha (cannot be used with NEMA_BLOP_MODULATE_RGB)

- **NEMA_BLOP_MODULATE_A**

Modulate by Constant Alpha value **NEMA_BLOP_FORCE_A** Force Constant Alpha value
NEMA_BLOP_MODULATE_RGB Modulate by Constant Color (RGB) values

- **NEMA_BLOP_SRC_CKEY**

Apply Source Color Keying - draw only when src color doesn't match colorkey

- **NEMA_BLOP_DST_CKEY**

Apply Destination Color Keying - draw only when dst color matches colorkey

- **NEMA_BLOP_MASK**

22.12.1.2 Function Documentation

22.12.1.2.1 **static uint32_t nema_blending_mode (nema_blend_factors_t src, nema_blend_factors_t dst, nema_blend_op_t ops) [inline], [static]**

Return blending mode given source and destination blending factors and additional blending operations.

Parameters

<i>src</i>	Source Blending Factor
<i>dst</i>	Destination Blending Factor
<i>ops</i>	Additional Blending Operations

Returns

Final Blending Mode

22.12.1.2.2 **void nema_set_blend(uint32_t blending_mode, nema_tex_t dst_tex, nema_tex_t fg_tex, nema_tex_t bg_tex)**

Set blending mode.

Parameters

<i>blending_mode</i>	Blending mode to be set
<i>dst_tex</i>	Destination Texture
<i>fg_tex</i>	Foreground (source) Texture
<i>bg_tex</i>	Background (source2) Texture

22.12.1.2.3 static void nema_set_blend_blit(uint32_t blending_mode) [inline], [static]

Set blending mode for blitting.

Parameters

<i>blending_mode</i>	Blending mode to be set

22.12.1.2.4 static void nema_set_blend_blit_compose(uint32_t blending_mode) [inline], [static]

Set blending mode for blitting with composing.

Parameters

<i>blending_mode</i>	Blending mode to be set

22.12.1.2.5 static void nema_set_blend_fill(uint32_t blending_mode) [inline], [static]

Set blending mode for filling.

Parameters

<i>blending_mode</i>	Blending mode to be set

22.12.1.2.6 static void nema_set_blend_fill_compose(uint32_t blending_mode) [inline], [static]

Set blending mode for filling with composing.

Parameters

<i>blending_mode</i>	Blending mode to be set

22.12.1.2.7 void nema_set_const_color(uint32_t rgba)

Set constant color.

Parameters

<i>rgba</i>	RGBA color

See Also

nema_rgba()

22.12.1.2.8 void nema_set_dst_color_key (uint32_t rgba)

Set destination color key.

Parameters

<i>rgba</i>	RGBA color key

See Also

nema_rgba()

22.12.1.2.9 static void nema_set_src_color_key (uint32_t rgba) [inline], [static]

Set source color key.

Parameters

<i>rgba</i>	RGBA color key

See Also

nema_rgba()

22.12.1.2.10 nema_cmdlist.h File Reference**Data Structures**

- struct nema_cmdlist_t

Functions

- `nema_cmdlist_t nema_cl_create_prealloc (nema_buffer_t *bo)`
Create a new Command List into a preallocated space.
- `nema_cmdlist_t nema_cl_create_sized (uint32_t size_bytes)`
Create a new, non expandable Command List of specific size.
- `nema_cmdlist_t nema_cl_create (void)`
Create a new expandable Command List.
- `void nema_cl_destroy (nema_cmdlist_t *cl)`
Destroy/Free a Command List.
- `void nema_cl_rewind (nema_cmdlist_t *cl)`
Reset position of next command to be written to the beginning. Doesn't clear the List's contents.
- `void nema_cl_bind (nema_cmdlist_t *cl)`
Define in which Command List each subsequent commands are going to be inserted.
- `void nema_cl_unbind (void)`
Unbind current bound Command List, if any.

- `void nema_cl_submit (nema_cmdlist_t *cl)`
Enqueue Command List to the Ring Buffer for execution.
- `void nema_cl_wait (nema_cmdlist_t *cl)`
Wait for Command List to finish.
- `int nema_cl_add_cmd (uint32_t reg, uint32_t data)`
Add a command to the bound Command List.
- `int nema_cl_add_multiple_cmds (int cmd_no, uint32_t *cmd)`
Add multiple commands to the bound Command List.
- `uint32_t *nema_cl_get_space (int cmd_no)`
- `void nema_cl_branch (nema_cmdlist_t *cl)`
Branch from the bound Command List to a different one. Return is implied.
- `void nema_cl_jump (nema_cmdlist_t *cl)`
Jump from the bound Command List to a different one. No return is implied.
- `void nema_cl_return (void)`
Add an explicit return command to the bound Command List.

22.12.1.3 Function Documentation

22.12.1.3.1 *int nema_cl_add_cmd (uint32_t reg, uint32_t data)*

Add a command to the bound Command List.

Parameters

<i>reg</i>	Hardware register to be written
<i>data</i>	Data to be written

Returns

0 if no error has occurred

22.12.1.3.2 *int nema_cl_add_multiple_cmds (int cmd_no, uint32_t *cmd)*

Add multiple commands to the bound Command List.

Parameters

<i>cmd_no</i>	Numbers of commands to add
<i>cmd</i>	Pointer to the commands to be added

Returns

0 if no error has occurred

22.12.1.3.3 void nema_cl_bind (nema_cmdlist_t * cl)

Define in which Command List each subsequent commands are going to be inserted.

Parameters

<i>cl</i>	Pointer to the Command List

22.12.1.3.4 void nema_cl_branch (nema_cmdlist_t * cl)

Branch from the bound Command List to a different one. Return is implied.

Parameters

<i>cl</i>	Pointer to the Command List to branch to

22.12.1.3.5 nema_cmdlist_t nema_cl_create (void)

Create a new expandable Command List. Returns

The instance of the new Command List

22.12.1.3.6 nema_cmdlist_t nema_cl_create_prealloc (nema_buffer_t * bo)

Create a new Command List into a preallocated space.

Parameters

<i>addr_virt</i>	Command List's address (preallocated)
<i>size_bytes</i>	Command List's size in bytes

Returns

The instance of the new Command List

22.12.1.3.7 nema_cmdlist_t nema_cl_create_sized (uint32_t size_bytes)

Create a new, non expandable Command List of specific size.

Parameters

<i>size_bytes</i>	Command List's size in bytes

Returns

The instance of the new Command List

22.12.1.3.8 void nema_cl_destroy (nema_cmdlist_t * cl)

Destroy/Free a Command List.

Parameters

<i>cl</i>	Pointer to the Command List

22.12.1.3.9 *uint32_t* nema_cl_get_space (int cmd_no)*

private

22.12.1.3.10 *void nema_cl_jump (nema_cmdlist_t * cl)*

Jump from the bound Command List to a different one. No return is implied.

Parameters

<i>cl</i>	Pointer to the Command List to jump to

22.12.1.3.11 *void nema_cl_return(void)*

Add an explicit return command to the bound Command List.

22.12.1.3.12 *void nema_cl_rewind (nema_cmdlist_t * cl)*

Reset position of next command to be written to the beginning. Doesn't clear the List's contents.

Parameters

<i>cl</i>	Pointer to the Command List

22.12.1.3.13 *void nema_cl_submit (nema_cmdlist_t * cl)*

Enqueue Command List to the Ring Buffer for execution.

Parameters

<i>cl</i>	Pointer to the Command List

22.12.1.3.14 *void nema_cl_unbind(void)*

Unbind current bound Command List, if any.

22.12.1.3.15 *void nema_cl_wait (nema_cmdlist_t * cl)*

Wait for Command List to finish.

Parameters

<i>cl</i>	Pointer to the Command List

22.12.1.3.16 *nema_easing.h* File Reference

Functions

- float `nema_ez_linear` (float p)
Linear easing, no acceleration.
- float `nema_ez_quad_in` (float p)
Quadratic easing in, accelerate from zero.
- float `nema_ez_quad_out` (float p)
Quadratic easing out, decelerate to zero velocity.
- float `nema_ez_quad_in_out` (float p)
Quadratic easing in and out, accelerate to halfway, then decelerate.
- float `nema_ez_cub_in` (float p)
Cubic easing in, accelerate from zero.
- float `nema_ez_cub_out` (float p)
Cubic easing out, decelerate to zero velocity.
- float `nema_ez_cub_in_out` (float p)
Cubic easing in and out, accelerate to halfway, then decelerate.
- float `nema_ez_quar_in` (float p)
Quartic easing in, accelerate from zero.
- float `nema_ez_quar_out` (float p)
Quartic easing out, decelerate to zero velocity.
- float `nema_ez_quar_in_out` (float p)
Quartic easing in and out, accelerate to halfway, then decelerate.
- float `nema_ez_quin_in` (float p)
Quintic easing in, accelerate from zero.
- float `nema_ez_quin_out` (float p)
Quintic easing out, decelerate to zero velocity.
- float `nema_ez_quin_in_out` (float p)
Quintic easing in and out, accelerate to halfway, then decelerate.
- float `nema_ez_sin_in` (float p)
Sinusoidal easing in, accelerate from zero.
- float `nema_ez_sin_out` (float p)
Sinusoidal easing out, decelerate to zero velocity.
- float `nema_ez_sin_in_out` (float p)
Sinusoidal easing in and out, accelerate to halfway, then decelerate.
- float `nema_ez_circ_in` (float p)
Circular easing in, accelerate from zero.

- `float nema_ez_circ_out (float p)`
Circular easing out, decelerate to zero velocity.
- `float nema_ez_circ_in_out (float p)`
Circular easing in and out, accelerate to halfway, then decelerate.
- `float nema_ez_exp_in (float p)`
Exponential easing in, accelerate from zero.
- `float nema_ez_exp_out (float p)`
Exponential easing out, decelerate to zero velocity.
- `float nema_ez_exp_in_out (float p)`
Exponential easing in and out, accelerate to halfway, then decelerate.
- `float nema_ez_elast_in (float p)`
Elastic easing in, accelerate from zero.
- `float nema_ez_elast_out (float p)`
Elastic easing out, decelerate to zero velocity.
- `float nema_ez_elast_in_out (float p)`
Elastic easing in and out, accelerate to halfway, then decelerate.
- `float nema_ez_back_in (float p)`
Overshooting easing in, accelerate from zero.
- `float nema_ez_back_out (float p)`
Overshooting easing out, decelerate to zero velocity.
- `float nema_ez_back_in_out (float p)`
Overshooting easing in and out, accelerate to halfway, then decelerate.
- `float nema_ez_bounce_out (float p)`
Bouncing easing in, accelerate from zero.
- `float nema_ez_bounce_in (float p)`
Bouncing easing out, decelerate to zero velocity.
- `float nema_ez_bounce_in_out (float p)`
Bouncing easing in and out, accelerate to halfway, then decelerate.
- `float nema_ez (float A, float B, float steps, float cur_step, float(*ez_func)(float p))`
Convenience function to perform easing between two values given number of steps, current step and easing function.

22.12.1.4 Function Documentation

?

22.12.1.4.1 `float nema_ez (float A, float B, float steps, float cur_step, float(*ez_func)(float p))`

Convenience function to perform easing between two values given number of steps, current step and easing function.

Parameters

<i>A</i>	Initial value within range [0, 1]
<i>B</i>	Finale value within range [0, 1]
<i>steps</i>	Total number of steps
<i>cur_step</i>	Current Step
<i>ez_func</i>	pointer to the desired easing function

Returns

Eased value

22.12.1.4.2 float nema_ez_back_in (float p)

Overshooting easing in, accelerate from zero.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.3 float nema_ez_back_in_out (float p)

Overshooting easing in and out, accelerate to halfway, then decelerate.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.4 float nema_ez_back_out (float p)

Overshooting easing out, decelerate to zero velocity.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.5 float nema_ez_bounce_in (float p)

Bouncing easing out, decelerate to zero velocity.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.6 float nema_ez_bounce_in_out (float p)

Bouncing easing in and out, accelerate to halfway, then decelerate.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.7 float nema_ez_bounce_out (float p)

Bouncing easing in, accelerate from zero.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.8 float nema_ez_circ_in (float p)

Circular easing in, accelerate from zero.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.9 float nema_ez_circ_in_out (float p)

Circular easing in and out, accelerate to halfway, then decelerate. [Parameters](#)

p	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.10 float nema_ez_circ_out (float p)

Circular easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.11 float nema_ez_cub_in (float p)

Cubic easing in, accelerate from zero.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.12 float nema_ez_cub_in_out (float p)

Cubic easing in and out, accelerate to halfway, then decelerate.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.13 float nema_ez_cub_out (float p)

Cubic easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns
Eased value

22.12.1.4.14 float nema_ez_elast_in (float p)

Elastic easing in, accelerate from zero.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns
Eased value

22.12.1.4.15 float nema_ez_elast_in_out (float p)

Elastic easing in and out, accelerate to halfway, then decelerate.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns
Eased value

22.12.1.4.16 float nema_ez_elast_out (float p)

Elastic easing out, decelerate to zero velocity.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns
Eased value

22.12.1.4.17 float nema_ez_exp_in (float p)

Exponential easing in, accelerate from zero.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.18 float nema_ez_exp_in_out (float p)

Exponential easing in and out, accelerate to halfway, then decelerate.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.19 float nema_ez_exp_out (float p)

Exponential easing out, decelerate to zero velocity.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.20 float nema_ez_linear (float p)

Linear easing, no acceleration.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.21 float nema_ez_quad_in (float p)

Quadratic easing in, accelerate from zero.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.22 float nema_ez_quad_in_out (float p)

Quadratic easing in and out, accelerate to halfway, then decelerate.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.23 float nema_ez_quad_out (float p)

Quadratic easing out, decelerate to zero velocity.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.24 float nema_ez_quar_in (float p)

Quartic easing in, accelerate from zero.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.25 float nema_ez_quar_in_out (float p)

Quartic easing in and out, accelerate to halfway, then decelerate.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.26 float nema_ez_quar_out (float p)

Quartic easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.27 float nema_ez_quin_in (float p)

Quintic easing in, accelerate from zero.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.28 float nema_ez_quin_in_out (float p)

Quintic easing in and out, accelerate to halfway, then decelerate.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.29 float nema_ez_quin_out (float p)

Quintic easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.30 float nema_ez_sin_in (float p)

Sinusoidal easing in, accelerate from zero

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.31 float nema_ez_sin_in_out (float p)

Sinusoidal easing in and out, accelerate to halfway, then decelerate.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.1.4.32 float nema_ez_sin_out (float p)

Sinusoidal easing out, decelerate to zero velocity.

Parameters

<i>p</i>	Input value, typically within the [0, 1] range

Returns

Eased value

22.12.2 nema_font.h File Reference

Data Structures

- struct nema_glyph_t
- struct nema_font_t

Macros

- #define NEMA_ALIGNX_CENTER - Align horizontally centered
- #define NEMA_ALIGNX_JUSTIFY - Justify horizontally
- #define NEMA_ALIGNX_LEFT - Align horizontally to the left
- #define NEMA_ALIGNX_MASK - Horizontal alignment mask
- #define NEMA_ALIGNX_RIGHT - Align horizontally to the right
- #define NEMA_ALIGNY_BOTTOM - Align vertically to the bottom
- #define NEMA_ALIGNY_CENTER - Align vertically centered
- #define NEMA_ALIGNY_JUSTIFY - Justify vertically
- #define NEMA_ALIGNY_MASK - Vertical alignment mask
- #define NEMA_ALIGNY_TOP - Align vertically to the top
- #define NEMA_TEXT_WRAP - Use text wrapping

Functions

- `void nema_bind_font (nema_font_t *font)`
Bind the font to use in future `nema_print()` calls.
- `int nema_string_get_bbox (char *str, int *w, int *h, int max_w)`
Get the bounding box's width and height of a string.
- `void nema_print (char *str, int *x, int *y, int w, int h, uint32_t fg_col, uint32_t align)`
Print pre-formatted text.

22.12.2.1 Function Documentation

22.12.2.1.1 void nema_bind_font (nema_font_t * font)

Bind the font to use in future `nema_print()` calls.

Parameters

<i>font</i>	Pointer to font

22.12.2.1.2 void nema_print (char * str, int * x, int * y, int w, int h, uint32_t fg_col, uint32_t align)

Print pre-formatted text.

Parameters

<i>str</i>	Pointer to string
<i>x</i>	X coordinate of the starting point. After execution, x contains the starting point for the next string to be drawn at.
<i>y</i>	Y coordinate of the starting point. After execution, y contains the starting point for the next string to be drawn at.
<i>w</i>	Width of the drawing area
<i>h</i>	Height of the drawing area
<i>fg_col</i>	Foreground color of text
<i>align</i>	Alignment and wrapping mode

22.12.2.1.3 int nema_string_get_bbox (char * str, int * w, int * h, int max_w)

Get the bounding box's width and height of a string.

Parameters

<i>str</i>	Pointer to string
<i>w</i>	Pointer to variable where width should be written
<i>h</i>	Pointer to variable where height should be written
<i>max_w</i>	Max allowed width

Returns

Number of carriage returns

22.12.3 *nema_graphics.h* File Reference

Enumerations

- enum nema_rotation_t {
 NEMA_ROT_000_CCW,
 NEMA_ROT_090_CCW,
 NEMA_ROT_180_CCW,
 NEMA_ROT_270_CCW,
 NEMA_ROT_000_CW,
 NEMA_ROT_270_CW,
 NEMA_ROT_180_CW,
 NEMA_ROT_090_CW,
 NEMA_MIR_VERT,
 NEMA_MIR_HOR}

Functions

- void nema_emulate_p (void)
 Emulate GPU functionality when using the GFX fragment shaders (blending modes).
- uint32_t nema_rgba (unsigned char R, unsigned char G, unsigned char B, unsigned char A)
 Return Nema internal RGBA color.
- uint32_t nema_premultiply_rgba (uint32_t rgba)
 Pre-multiply RGB channels with Alpha channel.
- int nema_init (void)
 Initialize NemaGFX library.
- void nema_bind_src_tex (uint32_t baseaddr_phys, uint32_t width, uint32_t height, nema_tex_format_t format, int32_t stride, nema_tex_mode_t mode)
 Program Texture Unit with a foreground (source) texture (NEMA_TEX1)
- void nema_bind_src2_tex (uint32_t baseaddr_phys, uint32_t width, uint32_t height, nema_tex_format_t format, int32_t stride, nema_tex_mode_t mode)
 Program Texture Unit with a background texture ((NEMA_TEX2)
- void nema_bind_dst_tex (uint32_t baseaddr_phys, uint32_t width, uint32_t height, nema_tex_format_t format, int32_t stride)
 Program Texture Unit with a destination texture (NEMA_TEX0)
- void nema_bind_depth_buffer (uint32_t baseaddr_phys, uint32_t width, uint32_t height)

Bind Depth Buffer.

- `void nema_clear (uint32_t rgba8888)`
Clear destination texture with color.
- `void nema_clear_depth (uint32_t val)`
Clear depth buffer with specified value.
- `void nema_draw_line (int x0, int y0, int x1, int y1, uint32_t rgba8888)`
Draw a colored line.
- `void nema_draw_circle (int x0, int y0, int r, uint32_t rgba8888)`
Draw a colored circle.
- `void nema_draw_rounded_rect (int x0, int y0, int w, int h, int r, uint32_t rgba8888)`
Draw a colored rectangle with rounded edges.
- `void nema_draw_rect (int x, int y, int w, int h, uint32_t rgba8888)`
Draw a colored rectangle.
- `void nema_fill_circle (int x, int y, int r, uint32_t rgba8888)`
Fill a circle with color.
- `void nema_fill_triangle (int x0, int y0, int x1, int y1, int x2, int y2, uint32_t rgba8888)`
Fill a triangle with color.
- `void nema_fill_rounded_rect (int x0, int y0, int w, int h, int r, uint32_t rgba8888)`
Fill a rectangle with rounded edges with color.
- `void nema_fill_rect (int x, int y, int w, int h, uint32_t rgba8888)`
Fill a rectangle with color.
- `void nema_fill_quad (int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3, uint32_t rgba8888)`
Fill a quadrilateral with color.
- `void nema_blit (int x, int y)`
Blit source texture to destination texture.
- `void nema_blit_rect (int x, int y, int w, int h)`
Blit source texture to destination's specified rectangle (crop or wrap when needed)
- `void nema_blit_rect_fit (int x, int y, int w, int h)`
Blit source texture to destination. Fit (scale) texture to specified rectangle.
- `void nema_blit_rotate (int x, int y, nema_rotation_t rotation)`
Rotate and Blit source texture to destination.
- `void nema_blit_rotate_partial (int sx, int sy, int sw, int sh, int x, int y, nema_rotation_t rotation)`
Rotate and Blit partial source texture to destination.
- `void nema_blit_quad_fit (float dx0, float dy0, float dx1, float dy1, float dx2, float dy2, float dx3, float dy3)`
Blit source texture to destination. Fit texture to specified quadrilateral.

22.12.3.1 Enumeration Type Documentation

22.12.3.1.1 enum nema_rotation_t

Enumerator

<i>NEMA_ROT_000_CCW</i>	No rotation
<i>NEMA_ROT_090_CCW</i>	Rotate 90 degrees counter-clockwise
<i>NEMA_ROT_180_CCW</i>	Rotate 180 degrees counter-clockwise
<i>NEMA_ROT_270_CCW</i>	Rotate 270 degrees counter-clockwise
<i>NEMA_ROT_000_CW</i>	No rotation
<i>NEMA_ROT_270_CW</i>	Rotate 270 degrees clockwise
<i>NEMA_ROT_180_CW</i>	Rotate 180 degrees clockwise
<i>NEMA_ROT_090_CW</i>	Rotate 90 degrees clockwise
<i>NEMA_MIR_VERT</i>	Mirror Vertically
<i>NEMA_MIR_HOR</i>	Mirror Horizontally

22.12.3.2 Function Documentation

22.12.3.2.1 void nema_bind_depth_buffer (uint32_t baseaddr_phys, uint32_t width, uint32_t height)

Bind Depth Buffer.

Parameters

<i>baseaddr_phys</i>	Address of the depth buffer, as seen by the GPU
<i>width</i>	Buffer width
<i>height</i>	Buffer height

22.12.3.2.2 void nema_bind_dst_tex (uint32_t baseaddr_phys, uint32_t width, uint32_t height, nema_tex_format_t format, int32_t stride)

Program Texture Unit with a destination texture (NEMA_TEX0)

Parameters

<i>baseaddr_phys</i>	Address of the destination texture, as seen by the GPU
<i>width</i>	Texture width
<i>height</i>	Texture height
<i>format</i>	Texture format
<i>stride</i>	Texture stride. If negative, it's calculated internally.

22.12.3.2.3 void nema_bind_src2_tex (uint32_t baseaddr_phys,uint32_t width,uint32_t height, nema_tex_format_t format,int32_t stride,nema_tex_mode_t mode)

Program Texture Unit with a background texture ((NEMA_TEX2)

Parameters

<i>baseaddr_phys</i>	Address of the source2 texture, as seen by the GPU
<i>width</i>	Texture width
<i>height</i>	Texture height
<i>format</i>	Texture format
<i>stride</i>	Texture stride. If negative, it's calculated internally.
<i>mode</i>	Wrapping and Filtering mode

22.12.3.2.4 void nema_bind_src_tex (uint32_t baseaddr_phys,uint32_t width,uint32_t height, nema_tex_format_t format,int32_t stride,nema_tex_mode_t mode)

Program Texture Unit with a foreground (source) texture (NEMA_TEX1)

Parameters

<i>baseaddr_phys</i>	Address of the source texture, as seen by the GPU
<i>width</i>	Texture width
<i>height</i>	Texture height
<i>format</i>	Texture format
<i>stride</i>	Texture stride. If negative, it's calculated internally.
<i>mode</i>	Wrapping and Filtering mode

22.12.3.2.5 void nema_blit (int x, int y)

Blit source texture to destination texture.

Parameters

<i>x</i>	destination x coordinate
<i>y</i>	destination y coordinate

See Also

nema_set_blend_fill()

22.12.3.2.6 void nema_blit_quad_fit (float dx0,float dy0,float dx1,float dy1,float dx2, float dy2, float dx3, float dy3)

Blit source texture to destination. Fit texture to specified quadrilateral.

Parameters

<i>dx0</i>	x coordinate at the first vertex of the quadrilateral
<i>dy0</i>	y coordinate at the first vertex of the quadrilateral

<i>dx1</i>	x coordinate at the second vertex of the quadrilateral
<i>dy1</i>	y coordinate at the second vertex of the quadrilateral
<i>dx2</i>	x coordinate at the third vertex of the quadrilateral
<i>dy2</i>	y coordinate at the third vertex of the quadrilateral
<i>dx3</i>	x coordinate at the fourth vertex of the quadrilateral
<i>dy3</i>	y coordinate at the fourth vertex of the quadrilateral

See Also

`nema_set_blend_blit()`

22.12.3.2.7 void *nema_blit_rect* (int x, int y, int w, int h)

Blit source texture to destination's specified rectangle (crop or wrap when needed)

Parameters

<i>x</i>	destination x coordinate
<i>y</i>	destination y coordinate
<i>w</i>	destination width
<i>h</i>	destination height

See Also

`nema_set_blend_blit()`

22.12.3.2.8 void *nema_blit_rect_fit* (int x, int y, int w, int h)

Blit source texture to destination. Fit (scale) texture to specified rectangle.

Parameters

<i>x</i>	destination x coordinate
<i>y</i>	destination y coordinate
<i>w</i>	destination width
<i>h</i>	destination height

See Also

`nema_set_blend_blit()`

22.12.3.2.9 void *nema_blit_rotate* (int x, int y, nema_rotation_t rotation)

Rotate and Blit source texture to destination.

Parameters

<i>x</i>	destination x coordinate
<i>y</i>	destination y coordinate
<i>rotation</i>	Rotation to be done

See Also

nema_set_blend_blit()

22.12.3.2.10 void nema_blit_rotate_partial (int sx,int sy,int sw,int sh,int x,int y,nema_rotation_t rotation)

Rotate and Blit partial source texture to destination.

Parameters

<i>sx</i>	source upper left x coordinate
<i>sy</i>	source upper left y coordinate
<i>sw</i>	source width of partial region
<i>sh</i>	source height of partial region
<i>x</i>	destination x coordinate
<i>y</i>	destination y coordinate
<i>rotation</i>	Rotation to be done

See Also

nema_set_blend_blit()

22.12.3.2.11 void nema_clear (uint32_t rgba8888)

Clear destination texture with color.

Parameters

<i>rgba8888</i>	32-bit RGBA color

See Also

nema_rgba()

22.12.3.2.12 void nema_clear_depth (uint32_t val)

Clear depth buffer with specified value.

Parameters

<i>val</i>	Clear value

22.12.3.2.13 void nema_draw_circle (int x0, int y0, int r, uint32_t rgba8888)

Draw a colored circle.

Parameters

<i>x0</i>	x coordinate of the circle's center
<i>y0</i>	y coordinate of the circle's center
<i>r</i>	circle's radius
<i>rgba8888</i>	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

22.12.3.2.14 void nema_draw_line (int x0, int y0, int x1, int y1, uint32_t rgba8888)

Draw a colored line.

Parameters

<i>x0</i>	x coordinate at the beginning of the line
<i>y0</i>	y coordinate at the beginning of the line
<i>x1</i>	x coordinate at the end of the line
<i>y1</i>	y coordinate at the end of the line
<i>rgba8888</i>	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

22.12.3.2.15 void nema_draw_rect (int x, int y, int w, int h, uint32_t rgba8888)

Draw a colored rectangle.

Parameters

<i>x</i>	x coordinate of the upper left vertex of the rectangle
<i>y</i>	y coordinate at the upper left vertex of the rectangle
<i>w</i>	width of the rectangle
<i>h</i>	height of the rectangle
<i>rgba8888</i>	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

void nema_draw_rounded_rect (int x0,int y0,int w,int h,int r,uint32_t rgba8888)

Draw a colored rectangle with rounded edges.

Parameters

<i>x0</i>	x coordinate of the upper left vertex of the rectangle

<i>y0</i>	y coordinate at the upper left vertex of the rectangle
<i>w</i>	width of the rectangle
<i>h</i>	height of the rectangle
<i>r</i>	corner radius
<i>rgba8888</i>	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

22.12.3.2.16 void nema_emulate_p(void)

Emulate NemaJP functionality when using NemaGFX fragment shaders (blending modes) .

22.12.3.2.17 void nema_fill_circle (int x, int y, int r, uint32_t rgba8888)

Fill a circle with color.

Parameters

<i>x</i>	x coordinate of the circle's center
<i>y</i>	y coordinate of the circle's center
<i>r</i>	circle's radius
<i>rgba8888</i>	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

22.12.3.2.18 void nema_fill_quad (int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3, uint32_t rgba8888)

Fill a quadrilateral with color.

Parameters

<i>x0</i>	x coordinate at the first vertex of the quadrilateral
<i>y0</i>	y coordinate at the first vertex of the quadrilateral
<i>x1</i>	x coordinate at the second vertex of the quadrilateral
<i>y1</i>	y coordinate at the second vertex of the quadrilateral
<i>x2</i>	x coordinate at the third vertex of the quadrilateral
<i>y2</i>	y coordinate at the third vertex of the quadrilateral
<i>x3</i>	x coordinate at the fourth vertex of the quadrilateral
<i>y3</i>	y coordinate at the fourth vertex of the quadrilateral
<i>rgba8888</i>	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

22.12.3.2.19 void nema_fill_rect (int x, int y, int w, int h, uint32_t rgba8888)

Fill a rectangle with color.

Parameters

<i>x</i>	x coordinate of the upper left vertex of the rectangle
<i>y</i>	y coordinate at the upper left vertex of the rectangle
<i>w</i>	width of the rectangle
<i>h</i>	height of the rectangle
<i>rgba8888</i>	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

22.12.3.2.20 void nema_fill_rounded_rect (int x0, int y0, int w, int h, int r, uint32_t rgba8888)

Fill a rectangle with rounded edges with color.

Parameters

<i>h</i>	height of the rectangle
<i>r</i>	corner radius
<i>rgba8888</i>	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

22.12.3.2.21 void nema_fill_triangle (int x0, int y0, int x1, int y1, int x2, int y2, uint32_t rgba8888)

Fill a triangle with color.

Parameters

<i>x0</i>	x coordinate at the first vertex of the triangle
<i>y0</i>	y coordinate at the first vertex of the triangle
<i>x1</i>	x coordinate at the second vertex of the triangle
<i>y1</i>	y coordinate at the second vertex of the triangle
<i>x2</i>	x coordinate at the third vertex of the triangle
<i>y2</i>	y coordinate at the third vertex of the triangle
<i>rgba8888</i>	Color to be used

See Also

nema_set_blend_fill()

22.12.3.2.22 *int nema_init(void)*

Initialize NemaGFX library.

Returns

-1 on error

22.12.3.2.23 *uint32_t nema_premultiply_rgba(uint32_t rgba)*

Pre-multiply RGB channels with Alpha channel.

Parameters

<i>rgba</i>	RGBA color

Returns

Pre-multiplied RGBA color

22.12.3.2.24 *uint32_t nema_rgba(unsigned char R, unsigned char G, unsigned char B, unsigned char A)*

Return Nema internal RGBA color.

Parameters

<i>R</i>	Red component
<i>G</i>	Green component
<i>B</i>	Blue component
<i>A</i>	Alpha component

Returns

RGBA value

22.12.4 *nema_hal.h* File Reference**Data Structures**

- `struct nema_buffer_t`

Macros

- `#define MUTEX_RB`
- `#define MUTEX_MALLOC`

Functions

- `int32_t nema_sys_init (void)`
Initialize system. Implementor defined. Called in `nema_init()`.
- `void nema_wait_irq (void)`
Wait for interrupt from the GPU.
- `void nema_wait_irq_cl (int cl_id)`
Wait for a Command List to finish.
- `uint32_t nema_reg_read (uint32_t reg)`
Read Hardware register.
- `void nema_reg_write (uint32_t reg, uint32_t value)`
Write Hardware Register.
- `nema_buffer_t nema_buffer_create (unsigned size)`
Create memory buffer.
- `nema_buffer_t nema_buffer_create_pool (int pool, unsigned size)`
Create memory buffer at a specific pool.
- `void *nema_buffer_map (nema_buffer_t *bo)`
Maps buffer.
- `void nema_buffer_unmap (nema_buffer_t *bo)`
Unmaps buffer.
- `void nema_buffer_destroy (nema_buffer_t *bo)`
Destroy/deallocate buffer.
- `uint32_t nema_buffer_phys (nema_buffer_t *bo)`
Get physical (GPU) base address of a given buffer.
- `void *nema_host_malloc (unsigned size)`
Allocate memory for CPU to use (typically, standard `malloc()` is called)
- `void nema_host_free (void *ptr)`
Free memory previously allocated with `nema_host_malloc()`
- `int nema_mutex_lock (int mutex_id)`
Mutex Lock for multiple processes/threads.
- `int nema_mutex_unlock (int mutex_id)`
Mutex Unlock for multiple processes/threads.

22.12.4.1 Function Documentation

22.12.4.1.1 `nema_buffer_t nema_buffer_create (unsigned size)`

Create memory buffer.

Parameters

<i>size</i>	Size of buffer in bytes

Returns

nema_buffer_t struct

22.12.4.1.2 *nema_buffer_t* *nema_buffer_create_pool* (int pool, unsigned size)

Create memory buffer at a specific pool.

Parameters

<i>pool</i>	ID of the desired memory pool
<i>size</i>	Size of buffer in bytes

Returns

nema_buffer_t struct

22.12.4.1.3 *void* *nema_buffer_destroy* (*nema_buffer_t* * bo)

Destroy/deallocate buffer.

Parameters

<i>bo</i>	Pointer to buffer struct

Returns

void

22.12.4.1.4 *void *nema_buffer_map* (*nema_buffer_t* * bo)**

Maps buffer.

Parameters

<i>bo</i>	Pointer to buffer struct

Returns

Virtual pointer of the buffer (same as in bo->base_virt)

22.12.4.1.5 *uint32_t* *nema_buffer_phys* (*nema_buffer_t* * bo)

Get physical (GPU) base address of a given buffer.

Parameters

<i>bo</i>	Pointer to buffer struct

Returns

Physical base address of a given `buffer`

22.12.4.1.6 void nema_buffer_unmap (nema_buffer_t * bo)

Unmaps buffer.

Parameters

<i>bo</i>	Pointer to buffer struct

Returns

void

22.12.4.1.7 void nema_host_free (void * ptr)

Free memory previously allocated with `nema_host_malloc()`

Parameters

<i>ptr</i>	Pointer to allocated memory (virtual)

Returns

void

SeeAlso `nema_host_malloc()`

22.12.4.1.8 void* nema_host_malloc (unsigned size)

Allocate memory for CPU to use (typically, standard `malloc()` is called)

Parameters

<i>size</i>	Size in bytes

Returns

Pointer to allocated memory (virtual)

See Also `nema_host_free()`

22.12.4.1.9 int nema_mutex_lock (int mutex_id)

Mutex Lock for multiple processes/threads.

Parameters

<i>MUTEX_RB</i>	or MUTEX_MALLOC

Returns

int

22.12.4.1.10 int nema_mutex_unlock (int mutex_id)

Mutex Unlock for multiple processes/threads.

Parameters

<i>MUTEX_RB</i>	or MUTEX_MALLOC

Returns

int

22.12.4.1.11 uint32_t nema_reg_read (uint32_t reg)

Read Hardware register.

Parameters

<i>reg</i>	Register to read

Returns

Value read from the register

See Also nema_reg_write

22.12.4.1.12 void nema_reg_write (uint32_t reg, uint32_t value)

Write Hardware Register.

Parameters

<i>reg</i>	Register to write
<i>value</i>	Value to be written

Returns

void()

See Also nema_reg_read()

22.12.4.1.13 *int32_t nema_sys_init(void)*

Initialize system. Implementor defined. Called in `nema_init()`.

Parameters

<i>void</i>	

Returns

0 if no errors occurred

See Also `nema_init()`

22.12.4.1.14 *void nema_wait_irq(void)*

Wait for interrupt from the GPU.

Parameters

<i>void</i>	

Returns

`void`

22.12.4.1.15 *void nema_wait_irq_cl(int cl_id)*

Wait for a Command List to finish.

Parameters

<i>cl_id</i>	Command List ID

Returns

`void`

22.12.5 *nema_math.h* File Reference

Macros

- `#define NEMA_E`
- `#define NEMA_LOG2E`
- `#define NEMA_LOG10E`
- `#define NEMA_LN2`
- `#define NEMA_LN10`
- `#define NEMA_PI`
- `#define NEMA_PI_2`
- `#define NEMA_PI_4`
- `#define NEMA_1_PI`
- `#define NEMA_2_PI`

- `#define NEMA_2_SQRTPI`
- `#define NEMA_SQRT2`
- `#define NEMA_SQRT1_2`
- `#define nema_min2 (a, b)`
Find the minimum of two values.
- `#define nema_max2 (a, b)`
Find the maximum of two values.
- `#define nema_clamp (val, min, max)`
Clamp value.
- `#define nema_abs (a)`
Calculate the absolute value.
- `#define nema_floats_equal (x, y)`
Compare two floats.
- `#define nema_float_is_zero (x)`
Checks if value x is zero.
- `#define nema_deg_to_rad (d)`
Convert degrees to radians.
- `#define nema_rad_to_deg(r)`
Convert radians to degrees.
- `#define nema_i2fx (a)`
Convert integer to 16.16 fixed point.
- `#define nema_f2fx(a)`
Convert float to 16.16 fixed point.
- `#define nema_floor(f)`
Floor function.
- `#define nema_ceil(f)`
Ceiling function.

Functions

- `float nema_sin (float angle_degrees)`
Fast sine approximation of a given angle.
- `float nema_cos (float angle_degrees)`
Fast cosine approximation of a given angle.
- `float nema_tan (float angle_degrees)`
Fast tangent approximation of a given angle.
- `float nema_pow (float x, float y)`
A rough approximation of x raised to the power of y. USE WITH CAUTION!

- float nema_sqrt (float x)
A rough approximation of the square root of x. USE WITH CAUTION!
- float nema_atan (float x)
A floating-point approximation of the inverse tangent of x.

22.12.5.1 Macro Definition Documentation

22.12.5.1.1 #define NEMA_1_PI

1/pi

22.12.5.1.2 #define NEMA_2_PI

2/pi

22.12.5.1.3 #define NEMA_2_SQRTPI

2/sqrt(pi)

22.12.5.1.4 #define nema_abs(a)

Calculate the absolute value.

Parameters

a	Value

Returns

The absolute value of a

22.12.5.1.5 #define nema_ceil(f)

Ceiling function.

Parameters

a	Value to be ceiled

Returns

ceiled value

22.12.5.1.6 #define nema_clamp(val,min,max)

Clamp value.

Parameters

<i>val</i>	Value to clamp
<i>min</i>	Minimum value
<i>max</i>	Minimum value

Returns

Clamped value

22.12.5.1.7 #define nema_deg_to_rad(d)

Convert degrees to radians.

Parameters

<i>d</i>	Angle in degrees

Returns

Angle in radians

22.12.5.1.8 #define NEMA_E

e

22.12.5.1.9 #define nema_f2fx(a)

Convert float to 16.16 fixed point.

Parameters

<i>a</i>	Value to be converted to 16.16 fixed point

Returns

16.16 fixed point value

22.12.5.1.10 #define nema_float_is_zero(x)

Checks if value x is zero.

Parameters

<i>x</i>	X value

Returns

1 if x == 0, 0 if x != 0

22.12.5.1.11 #define nema_floats_equal(x,y)

Compare two floats.

Parameters

x	First float
y	Second float

Returns

1 if x == y, 0 if x != y

22.12.5.1.12 #define nema_floor(f)

Floor function.

Parameters

a	Value to be floored

Returns

floored value

22.12.5.1.13 #define nema_i2fx(a)

Convert integer to 16.16 fixed point.

Parameters

a	Value to be converted

Returns

16.16 fixed point value

22.12.5.1.14 #define NEMA_LN10

ln(10)

22.12.5.1.15 #define NEMA_LN2

ln(2)

22.12.5.1.16 #define NEMA_LOG10E

log10(e)

22.12.5.1.17 #define NEMA_LOG2E

$\log_2(e)$

22.12.5.1.18 #define nema_max2(a,b)

Find the maximum of two values.

Parameters

<i>a</i>	First value
<i>b</i>	Second value

Returns

The maximum of *a* and *b*

22.12.5.1.19 #define nema_min2(a,b)

Find the minimum of two values.

Parameters

<i>a</i>	First value
<i>b</i>	Second value

Returns

The minimum of *a* and *b*

22.12.5.1.20 #define NEMA_PI

π

22.12.5.1.21 #define NEMA_PI_2

$\pi/2$

22.12.5.1.22 #define NEMA_PI_4

$\pi/4$

22.12.5.1.23 #define nema_rad_to_deg(r)

Convert radians to degrees.

Parameters

<i>r</i>	Angle in radians

Returns

Angle in degrees

22.12.5.1.24 **#define NEMA_SQRT1_2**

1/sqrt(2)

22.12.5.1.25**#define NEMA_SQRT2**

sqrt(2)

22.12.5.2 **Function Documentation**

22.12.5.2.1 **float nema_atan (float x)**

A floating-point approximation of the inverse tangent of x.

Parameters

<i>x</i>	X value

Returns

Inverse tangent (angle) of x in degrees

22.12.5.2.2 **float nema_cos (float angle_degrees)**

Fast cosine approximation of a given angle.

Parameters

<i>angle_degrees</i>	Angle in degrees

Returns

Cosine of the given angle

22.12.5.2.3 **float nema_pow (float x, float y)**

A rough approximation of x raised to the power of y. USE WITH CAUTION!

Parameters

<i>x</i>	base value. Must be non negative.

y	power value

Returns

the result of raising x to the power y

22.12.5.2.4 float nema_sin (float angle_degrees)

Fast sine approximation of a given angle.

Parameters

angle_degrees	Angle in degrees

Returns

Sine of the given angle

22.12.5.2.5 float nema_sqrt (float x)

A rough approximation of the square root of x. USE WITH CAUTION!

Parameters

x	X value. Must be non negative
\return	The square root of x

22.12.5.2.6 float nema_tan (float angle_degrees)

Fast tangent approximation of a given angle.

Parameters

angle_degrees	Angle in degrees

Returns

Tangent of the given angle

22.12.6 nema_matrix3x3.h File Reference

Typedefs

- typedef float nema_matrix3x3_t [3][3]

Functions

- void nema_mat3x3_load_identity (nema_matrix3x3_t m)

Load Identity Matrix.

- `void nema_mat3x3_translate (nema_matrix3x3_t m, float tx, float ty)`
Apply translate transformation.
- `void nema_mat3x3_scale (nema_matrix3x3_t m, float sx, float sy)`
Apply scale transformation.
- `void nema_mat3x3_shear (nema_matrix3x3_t m, float shx, float shy)`
Apply shear transformation.
- `void nema_mat3x3_mirror (nema_matrix3x3_t m, int mx, int my)`
Apply mirror transformation.
- `void nema_mat3x3_rotate (nema_matrix3x3_t m, float angle_degrees)`
Apply rotation transformation.
- `void nema_mat3x3_mul (nema_matrix3x3_t m, nema_matrix3x3_t _m)`
Multiply two 3x3 matrices ($m = m * _m$)
- `void nema_mat3x3_mul_vec (nema_matrix3x3_t m, float *x, float *y)`
Multiply vector with matrix.
- `void nema_mat3x3_mul_vec_affine (nema_matrix3x3_t m, float *x, float *y)`
Multiply vector with affine matrix.
- `void nema_mat3x3_adj (nema_matrix3x3_t m)`
Calculate adjoint.
- `void nema_mat3x3_div_scalar (nema_matrix3x3_t m, floats)`
Divide matrix with scalar value.
- `int nema_mat3x3_invert (nema_matrix3x3_t m)`
Invert matrix.
- `int nema_mat3x3_quad_to_rect (int width, int height, float sx0, float sy0, float sx1, float sy1, float sx2, float sy2, float sx3, float sy3, nema_matrix3x3_t m)`
Map rectangle to quadrilateral.

22.12.6.1 Function Documentation

22.12.6.1.1 `void nema_mat3x3_adj (nema_matrix3x3_t m)`

Calculate adjoint.

Parameters

<i>m</i>	Matrix

22.12.6.1.2 `void nema_mat3x3_div_scalar (nema_matrix3x3_t m, float s)`

Divide matrix with scalar value.

Parameters

<i>m</i>	Matrix to divide
<i>s</i>	scalar value

22.12.6.1.3 int nema_mat3x3_invert (nema_matrix3x3_t m)

Invert matrix.

Parameters

<i>m</i>	Matrix to invert

22.12.6.1.4 void nema_mat3x3_load_identity (nema_matrix3x3_t m)

Load Identity Matrix.

Parameters

<i>m</i>	Matrix to be loaded

22.12.6.1.5 void nema_mat3x3_mirror (nema_matrix3x3_t m, int mx, int my)

Apply mirror transformation.

Parameters

<i>m</i>	Matrix to apply transformation
<i>mx</i>	if non-zero, mirror horizontally
<i>my</i>	if non-zero, mirror vertically

22.12.6.1.6 void nema_mat3x3_mul (nema_matrix3x3_t m, nema_matrix3x3_t _m)

Multiply two 3x3 matrices (m = m*_m)

Parameters

<i>m</i>	left matrix, will be overwritten by the result
<i>m</i>	right matrix

22.12.6.1.7 void nema_mat3x3_mul_vec (nema_matrix3x3_t m, float * x, float * y)

Multiply vector with matrix.

Parameters

<i>m</i>	Matrix to multiply with
<i>x</i>	Vector x coefficient
<i>y</i>	Vector y coefficient

22.12.6.1.8 void nema_mat3x3_mul_vec_affine (nema_matrix3x3_t m, float * x, float * y)

Multiply vector with affine matrix.

Parameters

<i>m</i>	Matrix to multiply with
<i>x</i>	Vector x coefficient
<i>y</i>	Vector y coefficient

22.12.6.1.9 int nema_mat3x3_quad_to_rect (int width, int height, float sx0, float sy0, float sx1, float sy1, float sx2, float sy2, float sx3, float sy3, nema_matrix3x3_t m)

Map rectangle to quadrilateral.

Parameters

<i>width</i>	Rectangle width
<i>height</i>	Rectangle height
<i>sx0</i>	x coordinate at the first vertex of the quadrilateral
<i>sy0</i>	y coordinate at the first vertex of the quadrilateral
<i>sx1</i>	x coordinate at the second vertex of the quadrilateral
<i>sy1</i>	y coordinate at the second vertex of the quadrilateral
<i>sx2</i>	x coordinate at the third vertex of the quadrilateral
<i>sy2</i>	y coordinate at the third vertex of the quadrilateral
<i>sx3</i>	x coordinate at the fourth vertex of the quadrilateral
<i>sy3</i>	y coordinate at the fourth vertex of the quadrilateral
<i>m</i>	Mapping matrix

22.12.6.1.10 void nema_mat3x3_rotate (nema_matrix3x3_t m, float angle_degrees)

Apply rotation transformation.

Parameters

<i>m</i>	Matrix to apply transformation
<i>angle_degrees</i>	Angle to rotate in degrees

22.12.6.1.11 void nema_mat3x3_shear (nema_matrix3x3_t m, float shx, float shy)

Apply shear transformation.

Parameters

<i>m</i>	Matrix to apply transformation
<i>shx</i>	X shearing factor
<i>shy</i>	Y shearing factor

22.12.6.1.12 void nema_mat3x3_translate (nema_matrix3x3_t m, float tx, float ty)

Apply translate transformation.

Parameters

<i>m</i>	Matrix to apply transformation
<i>tx</i>	X translation factor
<i>ty</i>	Y translation factor

22.12.7 nema_matrix4x4.h File Reference**Typedefs**

- typedef float nema_matrix4x4_t [4][4]

Functions

- void nema_mat4x4_load_identity (nema_matrix4x4_t m)
Load a 4x4 Identity Matrix.
- void nema_mat4x4_mul (nema_matrix4x4_t m, nema_matrix4x4_t m_l, nema_matrix4x4_t m_r)
Multiply two 4x4 matrices.
- void nema_mat4x4_mul_vec (nema_matrix4x4_t m, float *x, float *y, float *z, float *w)
Multiply a 4x1 vector with a 4x4 matrix.
- void nema_mat4x4_translate (nema_matrix4x4_t m, float tx, float ty, float tz)
Apply translate transformation.
- void nema_mat4x4_scale (nema_matrix4x4_t m, float sx, float sy, float sz)
Apply scale transformation.
- void nema_mat4x4_rotate_X (nema_matrix4x4_t m, float angle_degrees)
Apply rotate transformation around X axis.
- void nema_mat4x4_rotate_Y (nema_matrix4x4_t m, float angle_degrees)
Apply rotate transformation around Y axis.
- void nema_mat4x4_rotate_Z (nema_matrix4x4_t m, float angle_degrees)
Apply rotate transformation around Z axis.
- void nema_mat4x4_load_perspective (nema_matrix4x4_t m, float fovy_degrees, float aspect, float nearVal, float farVal)
Set up a perspective projection matrix.
- void nema_mat4x4_load_ortho (nema_matrix4x4_t m, float left, float right, float bottom, float top, float nearVal, float farVal)
Set up an orthographic projection matrix.

- `void nema_mat4x4_load_ortho_2d (nema_matrix4x4_t m, float left, float right, float bottom, float top)`
Set up a 2D orthographic projection matrix.
- `int nema_mat4x4_obj_to_win_coords (nema_matrix4x4_t mvp, float x_orig, float y_orig, float width, float height, float nearVal, float farVal, float *x, float *y, float *z, float *w)`
Convenience Function to calculate window coordinates from object coordinates.

22.12.7.1 Function Documentation

22.12.7.1.1 void *nema_mat4x4_load_identity* (*nema_matrix4x4_t m*)

Load a 4x4 Identity Matrix.

Parameters

<i>m</i>	Matrix to be loaded

22.12.7.1.2 void *nema_mat4x4_load_ortho* (*nema_matrix4x4_t m*, *float left*, *float right*, *float bottom*, *float top*, *float nearVal*, *float farVal*)

Set up an orthographic projection matrix.

Parameters

<i>m</i>	A 4x4 Matrix
<i>left</i>	Left vertical clipping plane
<i>right</i>	Right vertical clipping plane
<i>bottom</i>	bottom horizontal clipping plane
<i>top</i>	Top horizontal clipping plane
<i>nearVal</i>	Distance from the viewer to the near clipping plane (always positive)
<i>farVal</i>	Distance from the viewer to the far clipping plane (always positive)

22.12.7.1.3 void *nema_mat4x4_load_ortho_2d* (*nema_matrix4x4_t m*, *float left*, *float right*, *float bottom*, *float top*)

Set up a 2D orthographic projection matrix.

Parameters

<i>m</i>	A 4x4 Matrix
<i>left</i>	Left vertical clipping plane
<i>right</i>	Right vertical clipping plane
<i>bottom</i>	bottom horizontal clipping plane
<i>top</i>	Top horizontal clipping plane

22.12.7.1.4 void *nema_mat4x4_load_perspective* (*nema_matrix4x4_t m*, *float fovy_degrees*, *float aspect*, *float nearVal*, *float farVal*)

Set up a perspective projection matrix.

Parameters

<i>m</i>	A 4x4 Matrix
<i>fovy_degrees</i>	Field of View in degrees
<i>aspect</i>	Aspect ratio that determines the field of view in the x direction.
<i>nearVal</i>	Distance from the viewer to the near clipping plane (always positive)
<i>farVal</i>	Distance from the viewer to the far clipping plane (always positive)

22.12.7.1.5 void *nema_mat4x4_mul* (*nema_matrix4x4_t m*, *nema_matrix4x4_tm_l*, *nema_matrix4x4_t m_r*)

Multiply two 4x4 matrices.

Parameters

<i>m</i>	Result Matrix
<i>m_l</i>	Left operand
<i>m_r</i>	Right operand

22.12.7.1.6 void *nema_mat4x4_mul_vec* (*nema_matrix4x4_t m*, *float * x*, *float * y*, *float * z*, *float * w*)

Multiply a 4x1 vector with a 4x4 matrix.

Parameters

<i>m</i>	Matrix to be multiplied
<i>x</i>	Vector first element
<i>y</i>	Vector second element
<i>z</i>	Vector third element
<i>w</i>	Vector forth element

22.12.7.1.7 int *nema_mat4x4_obj_to_win_coords* (*nema_matrix4x4_tmvp*, *float x_orig*, *float y_orig*, *float width*, *float height*, *float nearVal*, *float farVal*, *float * x*, *float * y*, *float * z*, *float * w*)

Convenience Function to calculate window coordinates from object coordinates.

Parameters

<i>mvp</i>	Model, View and Projection Matrix
<i>x_orig</i>	Window top left X coordinate
<i>y_orig</i>	Window top left Y coordinate
<i>width</i>	Window width
<i>height</i>	Window height
<i>nearVal</i>	Distance from the viewer to the near clipping plane (always positive)
<i>farVal</i>	Distance from the viewer to the far clipping plane (always positive)
<i>x</i>	X object coordinate
<i>y</i>	Y object coordinate
<i>z</i>	Z object coordinate

<i>w</i>	W object coordinate

Returns

1 if vertex is outside frustum (should be clipped)

22.12.7.1.8 void *nema_mat4x4_rotate_X* (*nema_matrix4x4_t m*, float *angle_degrees*)

Apply rotate transformation around X axis.

Parameters

<i>m</i>	Matrix to apply transformation
<i>angle_degrees</i>	Angle to rotate in degrees

22.12.7.1.9 void *nema_mat4x4_rotate_Y* (*nema_matrix4x4_t m*, float *angle_degrees*)

Apply rotate transformation around Y axis.

Parameters

<i>m</i>	Matrix to apply transformation
<i>angle_degrees</i>	Angle to rotate in degrees

22.12.7.1.10 void *nema_mat4x4_rotate_Z* (*nema_matrix4x4_t m*, float *angle_degrees*)

Apply rotate transformation around Z axis.

Parameters

<i>m</i>	Matrix to apply transformation
<i>angle_degrees</i>	Angle to rotate in degrees

22.12.7.1.11 void *nema_mat4x4_scale* (*nema_matrix4x4_t m*, float *sx*, float *sy*, float *sz*)

Apply scale transformation.

Parameters

<i>m</i>	Matrix to apply transformation
<i>sx</i>	X scaling factor
<i>sy</i>	Y scaling factor
<i>sz</i>	Z scaling factor

22.12.7.1.12 void *nema_mat4x4_translate* (*nema_matrix4x4_t m*, float *tx*, float *ty*, float *tz*)

Apply translate transformation.

Parameters

<i>m</i>	Matrix to apply transformation

<i>tx</i>	X	translation factor
<i>ty</i>	Y	translation factor
<i>tz</i>	Z	translation factor

22.12.8 *nema_programHW.h* File Reference

Enumerations

- enum `nema_tex_t`{
 - NEMA_NOTEX,
 - NEMA_TEX0,
 - NEMA_TEX1,
 - NEMA_TEX2,
 - NEMA_TEX3}

- enum `nema_tex_format_t`{
 - NEMA_RGBX8888,
 - NEMA_RGBA8888,
 - NEMA_XRGB8888,
 - NEMA_ARGB8888,
 - NEMA_RGB565,
 - NEMA_RGBA5650,
 - NEMA_RGBA5551,
 - NEMA_RGBA4444,
 - NEMA_RGBA0800,
 - NEMA_A8,
 - NEMA_RGBA0008,
 - NEMA_L8,
 - NEMA_RGBA3320,
 - NEMA_BW1,
 - NEMA_UYVY,
 - NEMA_ABGR8888,
 - NEMA_BGRA8888,
 - NEMA_BGRX8888,
 - NEMA_TSC4,
 - NEMA_TSC6,
 - NEMA_TSC6A,
 - NEMA_RY,

```
NEMA_GU,  
NEMA_BV,  
NEMA_YUV,  
NEMA_Z24_8,  
NEMA_Z16,  
NEMA_UV,  
NEMA_L2,  
NEMA_L4,  
NEMA_ASTC4_4,  
NEMA_ASTC8_8 }
```

- `enum nema_tex_mode_t {`
 `NEMA_FILTER_PS,`
 `NEMA_FILTER_BL,`
 `NEMA_TEX_CLAMP,`
 `NEMA_TEX_REPEAT,`
 `NEMA_TEX_BORDER,`
 `NEMA_TEX_MIRROR,`
 `NEMA_TEX_RANGE_0_1,`
 `NEMA_TEX_LEFT_HANDED }`

- `enum nema_tri_cull_t {`
 `NEMA_CULL_NONE,`
 `NEMA_CULL_CW,`
 `NEMA_CULL_CCW,`
 `NEMA_CULL_ALL }`

Functions

- `int nema_checkGPUPresence (void)`
 Check if a known GPU is present.
- `void nema_bind_tex (nema_tex_t texid, uint32_t addr_gpu, uint32_t width, uint32_t height, nema_tex_format_t format, int32_t stride, nema_tex_mode_t wrap_mode)`
 Program a Texture Unit.
- `void nema_set_tex_color (uint32_t color)`
 Set Texture Mapping default color.
- `void nema_set_matrix (nema_matrix3x3_t m)`

Load GPU's Matrix Multiplier with a given 3x3 matrix.

- `void nema_set_matrix_scale (float src_xres, float src_yres, float dst_xres, float dst_yres, float dst_x, float dst_y)`

Load GPU's Matrix Multiplier for scaling.

- `void nema_set_matrix_translate (float dst_x, float dst_y)`

Load GPU's Matrix Multiplier for a simple Blit (affine translation)

- `void nema_set_const_reg (int reg, uint32_t value)`

Write a value to a Constant Register of the GPU.

- `void nema_set_clip (uint32_t x, uint32_t y, uint32_t w, uint32_t h)`

Sets the drawing area's Clipping Rectangle.

- `void nema_load_frag_shader (const uint32_t *cmd, uint32_t count, uint32_t codeptr)`

Load a precompiled Shader to the GPU's internal memory.

- `void nema_set_frag_ptr (uint32_t ptr)`

Set the Internal Memory address of the fragment shader to be executed.

- `void nema_load_frag_shader_ptr (const uint32_t cmd, uint32_t count, uint32_t codeptr, uint32_t ptr)`

Load a precompiled Shader to the GPU's internal memory and set fragment pointer.

- `void nema_set_rop_blend_mode (uint32_t bl_mode)`

Set ROP blending mode.

- `void nema_set_rop_dst_color_key (uint32_t rgba)`

Set ROP destination color key.

- `void nema_set_rop_const_color (uint32_t rgba)`

Set ROP constant color.

- `void nema_tri_cull (nema_tri_cull_t cull)`

Set triangle/quadrilateral culling mode.

- `void nema_set_raster_color (uint32_t rgba8888)`

Set the color which will be used when drawing primitives (lines, rectangles etc)

- `void nema_raster_pixel (int x, int y)`

Program Rasterizer to generate a pixel.

- `void nema_raster_line (int x0, int y0, int x1, int y1)`

Program Rasterizer to draw a line.

- `void nema_raster_triangle_fx (int x0fx, int y0fx, int x1fx, int y1fx, int x2fx, int y2fx)`

Program Rasterizer to draw a triangle.

- `void nema_raster_rect (int x, int y, int w, int h)`

Program Rasterizer to generate a triangle.

- `void nema_raster_rounded_rect (int x0, int y0, int w, int h, int r)`

Program Rasterizer to draw a rectangle with rounded edges.

- `void nema_raster_triangle_f (float x0, float y0, float z0, float w0, float x1, float y1, float z1, float w1, float x2, float y2, float z2, float w2)`
- `void nema_raster_quad_fx(intx0fx, inty0fx, intx1fx, inty1fx, intx2fx, inty2fx, intx3fx, inty3fx)`
Program Rasterizer to draw a quadrilateral.
- `void nema_get_dirty_region (int *minx, int *miny, int *maxx, int *maxy)`
Returns the bounding rectangle of all the pixels that have been modified since its previous call.
- `int nema_format_size (nema_tex_format_t format)`
Return pixel size in bytes.
- `int nema_stride_size (nema_tex_format_t format, int width)`
Return stride in bytes.
- `int nema_texture_size (nema_tex_format_t format, int width, int height)`
Return texture size in bytes.
- `void nema_enable_tiling (uint32_t enable)`
- `void nema_set_depth_range (float min_depth, float max_depth)`
Set maximum and minimum values for depth buffer. Available only for Nema|T.
- `void nema_set_viewport (float x, float y, float w, float h)`
Sets Viewport parameters for vertex shader. Available only for Nema|T.

22.12.8.1 Enumeration Type Documentation

22.12.8.1.1 enum nema_tex_format_t

Enumerator

`NEMA_RGBX8888` RGBX8888
`NEMA_RGBA8888` RGBA8888
`NEMA_XRGB8888` XRGB8888
`NEMA_ARGB8888` ARGB8888
`NEMA_RGB565` RGBA5650
`NEMA_RGBA5650` RGBA5650
`NEMA_RGBA5551` RGBA5551
`NEMA_RGBA4444` RGBA4444
`NEMA_RGBA0800` RGBA0800
`NEMA_A8` RGBA0008
`NEMA_RGBA0008` RGBA0008
`NEMA_L8` L8
`NEMA_RGBA3320` RGBA3320 (source only)
`NEMA_BW1` BW1 (source only)

NEMA_UYVY UYVY
NEMA_ABGR8888 ABGR8888
NEMA_BGRA8888 BGRA
NEMA_BGRX8888 BGRX
NEMA_TSC4 TSC4
NEMA_TSC6 TSC6
NEMA_TSC6A TSC6A
NEMA_TSC12 TSC12
NEMA_TSC12A TSC12A
NEMA_RYRY
NEMA_GU GU
NEMA_BV BV
NEMA_YUV YUV
NEMA_Z24_8 Z24_8
NEMA_Z16 Z16
NEMA_UV UV
NEMA_L2 L2
NEMA_L4 L4
NEMA_ASTC4_4 ASTC4_4
NEMA_ASTC8_8 ASTC8_8

22.12.8.1.2 *enum nema_tex_mode_t*

Enumerator

NEMA_FILTER_PS Point Sampling.
NEMA_FILTER_BL Bilinear filtering.
NEMA_TEX_CLAMP Clamp
NEMA_TEX_REPEAT Repeat
NEMA_TEX_BORDER Border
NEMA_TEX_MIRROR Mirror
NEMA_TEX_RANGE_0_1 Interpolated Coordinates range: 0-1
NEMA_TEX_LEFT_HANDED (0,0) is bottom leftcorner

22.12.8.1.3 *enum nema_tex_t*

Enumerator

NEMA_NOTEX No Texture

NEMA_TEX0 Texture 0

NEMA_TEX1 Texture 1

NEMA_TEX2 Texture 2

NEMA_TEX3 Texture 3

22.12.8.1.4 enum *nema_tri_cull_t*

Enumerator

NEMA_CULL_NONE Disable Triangle/Quadrilateral Culling

NEMA_CULL_CW Cull clockwise Triangles/Quadrilaterals

NEMA_CULL_CCW Cull anti-clockwise Triangles/Quadrilaterals

NEMA_CULL_ALL Cull all

22.12.8.2 Function Documentation

22.12.8.2.1 void *nema_bind_tex* (*nema_tex_t* *texid*, *uint32_t* *addr_gpu*, *uint32_t* *width*, *uint32_t* *height*, *nema_tex_format_t* *format*, *int32_t* *stride*, *nema_tex_mode_t* *wrap_mode*)

Program a Texture Unit.

Parameters

<i>texid</i>	Texture unit to be programmed
<i>addr_gpu</i>	Texture's address as seen by the GPU
<i>width</i>	Texture's width
<i>height</i>	Texture's height
<i>format</i>	Texture's format
<i>stride</i>	Texture's stride. If stride < 0, it's left to be calculated
<i>wrap_mode</i>	Wrap/Repeat mode to be used

22.12.8.2.2 int *nema_checkGPUPresence* (void)

Check if a known GPU is present. Returns -1 if no known GPU is present

22.12.8.2.3 void *nema_enable_tiling* (*uint32_t* *enable*)

22.12.8.2.4 int *nema_format_size* (*nema_tex_format_t* *format*)

Return pixel size in bytes.

Parameters

<i>format</i>	Color format

Returns

Pixel size in bytes

22.12.8.2.5 void *nema_get_dirty_region* (int ? minx,int ? miny,int ? maxx,int ? maxy)

Returns the bounding rectangle of all the pixels that have been modified since its previous call.

Parameters

<i>minx</i>	x coordinate of the upper left corner of the dirty region
<i>miny</i>	y coordinate of the upper left corner of the dirty region
<i>maxx</i>	x coordinate of the lower right corner of the dirty region
<i>maxy</i>	y coordinate of the lower right corner of the dirty region

?

22.12.8.2.6 void *nema_load_frag_shader* (const uint32_t cmd, uint32_t count, uint32_t codeptr)

Load a precompiled Shader to the GPU's internal memory.

Parameters

<i>cmd</i>	Pointer to the shader
<i>count</i>	Number of commands
<i>codeptr</i>	Internal Memory address to be written (default is 0)

22.12.8.2.7 void *nema_load_frag_shader_ptr* (const uint32_t cmd, uint32_t count, uint32_t codeptr, uint32_t ptr)

Load a precompiled Shader to the GPU's internal memory and set fragment pointer.

Parameters

<i>cmd</i>	Pointer to the shader
<i>count</i>	Number of commands
<i>codeptr</i>	Internal Memory address to be written (default is 0)
<i>ptr</i>	Internal Memory address of the fragment shader

22.12.8.2.8 void *nema_raster_line* (int x0, int y0, int x1, int y1)

Program Rasterizer to draw a line.

Parameters

<i>x0</i>	x coordinate at the beginning of the line
<i>y0</i>	y coordinate at the beginning of the line
<i>x1</i>	x coordinate at the end of the line
	y coordinate at the end of the line

<i>y1</i>	y coordinate at the end of the line

22.12.8.2.9 void *nema_raster_pixel* (int x, int y)

Program Rasterizer to generate a pixel.

Parameters

<i>x</i>	x coordinate of the pixel
<i>y</i>	y coordinate of the pixel

22.12.8.2.10 void *nema_raster_quad_fx* (int x0fx, int y0fx, int x1fx, int y1fx, int x2fx, int y2fx, int x3fx, int y3fx)

Program Rasterizer to draw a quadrilateral.

Parameters

<i>x0fx</i>	x coordinate at the first vertex of the quadrilateral (fixed point 16.16)
<i>y0fx</i>	y coordinate at the first vertex of the quadrilateral (fixed point 16.16)
<i>x1fx</i>	x coordinate at the second vertex of the quadrilateral (fixed point 16.16)
<i>y1fx</i>	y coordinate at the second vertex of the quadrilateral (fixed point 16.16)
<i>x2fx</i>	x coordinate at the third vertex of the quadrilateral (fixed point 16.16)
<i>y2fx</i>	y coordinate at the third vertex of the quadrilateral (fixed point 16.16)
<i>x3fx</i>	x coordinate at the fourth vertex of the quadrilateral (fixed point 16.16)
<i>y3fx</i>	y coordinate at the fourth vertex of the quadrilateral (fixed point 16.16)

22.12.8.2.11 void *nema_raster_rect* (int x, int y, int w, int h)

Program Rasterizer to generate a rectangle.

Parameters

<i>x</i>	x coordinate of the upper left vertex of the rectangle
<i>y</i>	y coordinate at the upper left vertex of the rectangle
<i>w</i>	width of the rectangle
<i>h</i>	height of the rectangle

22.12.8.2.12 void *nema_raster_rounded_rect* (int x0, int y0, int w, int h, int r)

Program Rasterizer to draw a rectangle with rounded edges.

Parameters

<i>x0</i>	x coordinate of the upper left vertex of the rectangle
<i>y0</i>	y coordinate at the upper left vertex of the rectangle
<i>w</i>	width of the rectangle

<i>h</i>	height of the rectangle
<i>r</i>	corner radius

22.12.8.2.13 void nema_raster_triangle_f (float x0,float y0,float z0,float w0,float x1, float y1, float z1, float w1, float x2, float y2, float z2, float w2)

22.12.8.2.14 void nema_raster_triangle_fx (int x0fx, int y0fx, int x1fx, int y1fx, int x2fx, int y2fx)

Program Rasterizer to draw a triangle.

Parameters

<i>x0fx</i>	x coordinate at the first vertex of the triangle (fixed point 16.16)
<i>y0fx</i>	y coordinate at the first vertex of the triangle (fixed point 16.16)
<i>x1fx</i>	x coordinate at the second vertex of the triangle (fixed point 16.16)
<i>y1fx</i>	y coordinate at the second vertex of the triangle (fixed point 16.16)
<i>x2fx</i>	x coordinate at the third vertex of the triangle (fixed point 16.16)
<i>y2fx</i>	y coordinate at the third vertex of the triangle (fixed point 16.16)

22.12.8.2.15 void nema_set_clip (uint32_t x, uint32_t y, uint32_t w, uint32_t h)

Sets the drawing area's Clipping Rectangle.

Parameters

<i>x</i>	Clip Window top-left x coordinate
<i>y</i>	Clip Window minimum y
<i>w</i>	Clip Window width
<i>h</i>	Clip Window height

22.12.8.2.16 void nema_set_const_reg (int reg, uint32_t value)

Write a value to a Constant Register of the GPU.

Parameters

<i>reg</i>	Constant Register to be written
<i>value</i>	Value to be written

22.12.8.2.17 void nema_set_depth_range (float min_depth, float max_depth)

Set maximum and minimum values for depth buffer. Available only for Nema|T.

Parameters

<i>min_depth</i>	Minimum value

<i>max_depth</i>	Maximum value

22.12.8.2.18 void nema_set_frag_ptr (uint32_t ptr)

Set the Internal Memory address of the fragment shader to be executed.

Parameters

<i>ptr</i>	Internal Memory address of the fragment shader

22.12.8.2.19 void nema_set_matrix (nema_matrix3x3_t m)

Load GPU's Matrix Multiplier with a given 3x3 matrix.

Parameters

<i>m</i>	Matrix to be loaded

22.12.8.2.20 void nema_set_matrix_scale (float src_xres, float src_yres, float dst_xres, float dst_yres, float dst_x, float dst_y)

Load GPU's Matrix Multiplier for scaling.

Parameters

<i>src_xres</i>	Width of source rectangular area
<i>src_yres</i>	Height of source rectangular area
<i>dst_xres</i>	Width of destination rectangular area
<i>dst_yres</i>	Height of destination rectangular area
<i>dst_x</i>	X coordinate of upper-left vertex of the destination
<i>dst_y</i>	Y coordinate of upper-left vertex of the destination

22.12.8.2.21 void nema_set_matrix_translate (float dst_x, float dst_y)

Load GPU's Matrix Multiplier for a simple Blit (affine translation)

Parameters

<i>dst_x</i>	X coordinate of upper-left vertex of the destination
<i>dst_y</i>	Y coordinate of upper-left vertex of the destination

22.12.8.2.22 void nema_set_raster_color (uint32_t rgba8888)

Set the color which will be used when drawing primitives (lines, rectangles etc)

Parameters

<i>rgba8888</i>	Color to be used

See Also

nema_rgba()

22.12.8.2.23 void nema_set_rop_blend_mode (uint32_t bl_mode)

Set ROP blending mode.

Parameters

<i>bl_mode</i>	Blending mode

22.12.8.2.24 void nema_set_rop_const_color (uint32_t rgba)

Set ROP constant color.

Parameters

<i>rgba</i>	Constant color

See Also

nema_rgba()

22.12.8.2.25 void nema_set_rop_dst_color_key (uint32_t rgba)

Set ROP destination color key.

Parameters

<i>rgba</i>	Destination Color Key

See Also

nema_rgba()

22.12.8.2.26 void nema_set_tex_color (uint32_t color)

Set Texture Mapping default color.

Parameters

<i>color</i>	default color in 32-bit RGBA format

See Also

nema_rgba()

22.12.8.2.27 void nema_set_viewport (float x, float y, float w, float h)

Sets Viewport parameters for vertex shader. Available only for Nema|T.

Parameters

<i>x</i>	Start X coordinate
<i>y</i>	Start Y coordinate
<i>w</i>	Width of the Viewport
<i>h</i>	Height of the Viewport

22.12.8.2.28 int nema_stride_size (nema_tex_format_t format, int width)

Return stride in bytes.

Parameters

<i>format</i>	Color format
<i>width</i>	Texture color format

Returns

Stride in bytes

22.12.8.2.29 int nema_texture_size (nema_tex_format_t format, int width, int height)

Return texture size in bytes.

Parameters

<i>format</i>	Texture color format
<i>width</i>	Texture width
<i>height</i>	Texture height

Returns

Texture size in bytes

22.12.8.2.30 void nema_tri_cull (nema_tri_cull_t cull)

Set triangle/quadrilateral culling mode.

Parameters

<i>cull</i>	Culling mode

23. PDM-to-PCM Converter (PDM)

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

23.1 PDM Clock Configuration

The PDM mic can be configured for LoFi (AoN) or HiFi (telco) audio modes via the `MCUCTRL_PLLMUXCTL_PDMPLLCLKSEL` field. For LoFi the source clock will be low-power HFRC, a crystal, or `EXTREFCLK` (Apollo510 SoC only), and for HiFi the source clock will be the PLL.

There are 4 clocks or frequencies of relevance for the PDM interface:

- Interface core main clock, `PDM_CLK`.
- PDM output clock, `PDM_CKO`, whose frequency is selected to meet microphone specification.
- Desired sampling frequency at the microphone, F_{sin} , also determined by the requirements of the microphone.
- Desired sample rate, F_{sout} , of the PCM stream from the interface determined by the requirements of the application.

The Oversample Ratio (OSR), which is the ratio of the input sample frequency divided by the output sample frequency, or $F_{\text{sin}} / F_{\text{sout}}$, is an important parameter to determine and is derived based on several factors. By definition, the `PDM_CKO` clock to the microphone(s) is the same as F_{sin} . The Nyquist Sampling Rate requires sampling at, or greater than, twice the bandwidth (BW) of the signal. For example, if a bandwidth of 8 kHz is targeted, then a sampling rate of 16 kHz is required. Oversampling is sampling beyond the Nyquist rate such that $\text{OSR} = F_{\text{sin}} / \text{Nyquist Rate} = F_{\text{sin}} / (2 \times \text{BW}) = F_{\text{sin}} / F_{\text{sout}}$. For example, 8 kHz BW, sampled at 768 kHz, has an OSR of $768 \text{ kHz} / 16 \text{ kHz} = 48$.

The OSR is selected by setting the SINC decimation rate in the `SINCRATE` field in the `PDMn_CORECFG0` register, such that $\text{OSR} = 2 \times \text{SINCRATE}$.

The `PDM_CLK` and divider ratio are selected in the `PDMn_CTRL_CLKSEL` bit field on Apollo510 and Apollo510B SoCs. On Apollo510 Lite series and Apollo330 Plus series SoCs, the source of `PDM_CLK` is selected in the `DSPPDM0CLKSEL` field of `CRM_DSPPDM0CRM` register. The divider ratio is written by the user in the `DSPPDM0CLKDIV` field of the same register. The `CLKSEL` field of the `PDMn_CTRL` register is not used on these SoCs..

On the Apollo510 SoC, the `PDMn_CLK` frequency can be set via the `CLKSEL` field, which allows for clocking from HFRC2, the high frequency crystal, or an external reference clock (`EXTREFCLK`). In addition, the `SYSPLL` can be used to clock the PDM by setting the `MCUCTRL_PLLMUXCTL_PDMPLLCLKSEL` register field to "PLL". This allows for a more general high frequency crystal value to be chosen for use with other peripherals (e.g., 48 MHz to clock USB), while still enabling specific frequencies required by the audio subsystem.

The microphone clock frequency, `PDM_CKO` or F_{sin} , is determined by the requirements of the microphone. This clock is typically set to 768 kHz for lower power operation or to 1536 kHz for slightly better microphone performance. As mentioned, F_{sout} is determined by the requirements of the application, typically set for 16 kHz or 32 kHz.

The `PDM_CKO` is configured through the `SINCRATE` and `MCLKDIV` fields of the `PDMn_CORECFG0` register, and the `DIVMCLKQ` field of the `PDMn_CORECFG1` register. See PDM chapter of the Apollo5 datasheet and the PDM register set for the relationships between the clocks and their dividers.

The following table shows settings which are recommended to produce commonly used PDM bit data clock frequencies (PDMA_CKO) and PCM sample rates as a function of F_{sin} and OSR for the various operating modes, at a PDM_CLK of 24.576 MHz.

Table 129: PDMA_CKO and OSR Settings for Different Sampling Frequencies

OPERATING MODE	$F_{\text{PDMA\_CKO}}$ (MHz)	F_{sin} (kHz)	OSR	DIV_MCLKQ [1:0]	MCLKDIV [3:0]	SINCRATE [6:0]	Maximum SINAD (dB)	Maximum DR (dB)
High Performance Mode	6.144	96	64	1	1	32	103	110.8
	3.072	48	64	1	3	32	105.5	108.7
	3.072	24	128	1	3	64	122.8	120.9
	3.072	16	192	1	3	96	116.1	120.4
	1.536	16	96	1	7	48	115.4	120.5
Reduced Performance Mode	3.072	96	32	1	3	16	86	87.9
	1.536	48	32	1	7	16	83.2	88.8
	0.768	16	48	1	15	24	89.7	97.2
Mid Performance Mode	1.536	24	64	1	7	32	101	107.6
	1.024	16	64	1	11	32	100	106.8

Notes:

1. Assumes PDM_CLK of 24.576 MHz.
2. The above frequency combinations are recommended values, where **DIV_MCLKQ = 2'b01**. User may determine other proper values according to actual main clock rate and digital microphones implemented in system design.
3. **Maximum SINAD** means maximum ratio of signal to noise plus the first N harmonics of THD.
4. **Maximum DR** means maximum dynamic range, which is measured as SINAD - (-60 dB) in this table.

To derive the settings, use the steps and equations below:

- For a PDM_CLK = 24.576 MHz, set PDMx_CTRL_CLKSEL to 1.
- For oversampling where $\text{OSR} = F_{\text{sin}} / F_{\text{sout}}$, OSR should typically be between 32 and 128, or 192.
- To select the SINC rate to be equal to $\text{OSR} / 2$, set PDMx_CORECFG0_SINCRATE field to typically be between 16 and 64, or 96 ONLY for OSR of 192.
- To select the recommended setting of 1 for DIV_MCLKQ where PDMn_MCLKQ is the same clock frequency as PDMn_CLK, set the PDMx_CORECFG1_DIVMCLKQ field to 1.
- To select the PDM output clock, PDMn_CKO, select the divider of the MCLKQ clock by setting the PDMx_CORECFG0_MCLKDIV field to get the desired value of PDMn_CKO, where $\text{PDMn\_CKO} = \text{PDMn\_MCLKQ} / (\text{PDMn\_MCLKDIV} + 1)$.

23.1.1 PDM Clock Duty Cycle

PDMA_CKO duty cycle is determined by:

- MCLKDIV divider when MCLKDIV is not 0
- DIVMCLKQ divider when MCLKDIV is 0

Divider setting DIVMCLKQ and MCLKDIV value N corresponds to divide-by-(N+1). Even divide ratios generate approximately 50% duty cycle while odd divide ratios generate asymmetric duty cycles. For divide-by-(2K+1):

- High width = K
- Low width = K+1
- Duty cycle = $K / (2K+1)$

For example, if MCLKDIV = 2 and the actual divide ratio = 3, then the duty cycle = 1/3 (33%). PDMA_CKO does not guarantee a 50% duty cycle when configured for odd divide ratios. The following equation can be used for calculating the duty cycle when MCLKDIV or DIVQ (when MCLKDIV=0) is N (actual divide = N+1):

$$\text{Duty cycle} = \text{FLOOR}((N+1)/2) / (N+1)$$

The following duty cycles result:

- MCLKDIV = 1 (divider = 2) => Duty Cycle = 1/2
- MCLKDIV = 2 (divider = 3) => Duty Cycle = 1/3
- MCLKDIV = 3 (divider = 4) => Duty Cycle = 1/2
- MCLKDIV = 4 (divider = 5) => Duty Cycle = 2/5
- ...
- MCLKDIV = 8 (divider = 9) => Duty Cycle = 4/9
- ...
- MCLKDIV = 15 (divider = 16) => Duty Cycle = 1/2
- MCLKDIV = 16 (divider = 17) => Duty Cycle = 8/17
- ...
- MCLKDIV = 30 (divider = 31) => Duty Cycle = 15/31
- MCLKDIV = 31 (divider = 32) => Duty Cycle = 1/2

23.1.2 Clock Gating and Data Synchronization

For low power implementation, PDM_CLK is sourced from the MCU and divided down internally for each conversion channel with clock gating. Figure 136 shows the clock tree and internal synchronization from PDM_CLK clock domain to the MCLKQ clock domain. The DIVIDER block is user-controlled by the DIVMCLKQ register field to generate MCLKQ, which in turn is divided by MCLKDIV and fed to the CLOCK GENERATOR block to generate PDMA_CKO.

MCLK_L is always on in both mono and stereo mode. That is, in mono right-channel operation, it actually uses the left channel of PDM-to-PCM core conversion for recording conversion process. LRSWAP must be set to “1” in mono right mode operation, and “0” in mono left mode operation. Please also refer to “Operating Modes” Section for more detailed description.

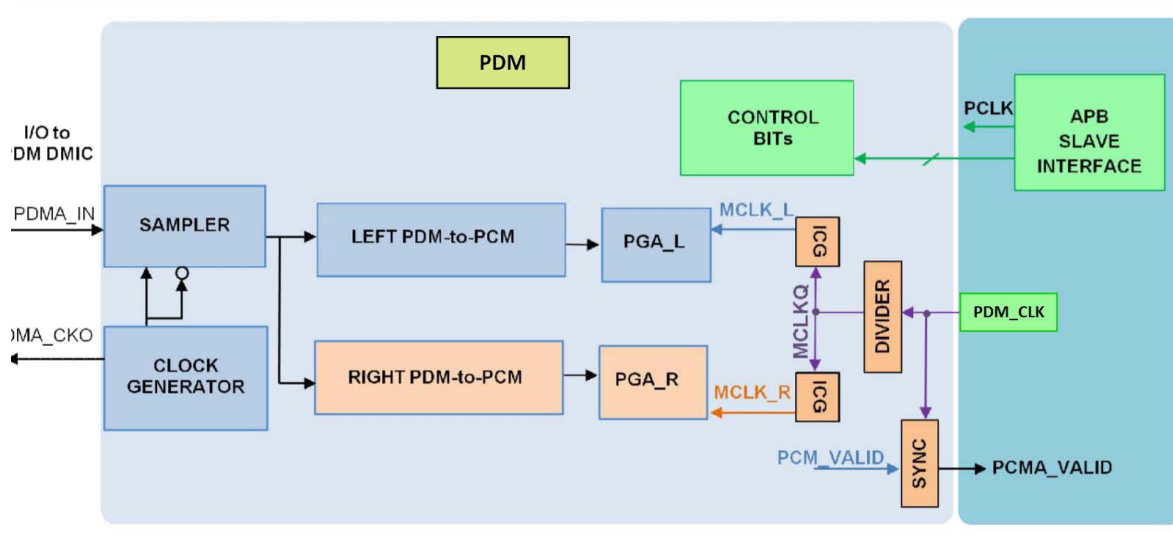


Figure 136. Clock Path and Data Synchronization Diagram

Figure 137 illustrates the clock gating scheme for MCLK_L and MCLK_R internal main clocks through the ICG (integrated clock gating) cells. PDM_LEFT_EN and PDM_STEREO_EN signals are controlled by the setting of the CORECFG1_PCMCHSET field.

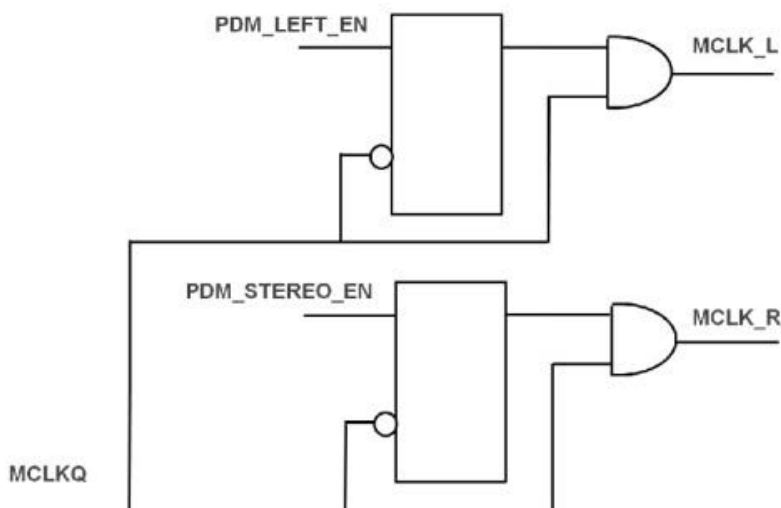


Figure 137. PDM Converter Core Local Clock Gating

23.2 System Gain and Transfer Function

The block diagram in Figure 136 above can also be referenced to show the location of the gain stages, PGA_L and PGA_R, between the DMICs connected to PDMA_IN and PDMA_CKO on the left side and the left/right PCM data read by the CPU on the right side. It is important to note that there is extra conversion gain inside the PDM-to-PCM core conversion block in the diagram. This extra gain is different from the settable PGA_L/R gain (-12 to +34.5 dB).

In many digital MEMS microphones, the maximum specified acoustic input is mapped to a PCM output slightly below 0 dBFS (e.g., -1 to -3 dBFS) to preserve delta-sigma modulator stability and avoid overload-related artifacts. To compensate for this intentional headroom, the DMIC interface includes a built-in Compensation Gain mechanism. This mechanism scales the decimated PCM signal such that the maximum valid PDM input level maps to a PCM output level close to 0 dBFS, allowing the CPU to utilize the full available digital dynamic range. Without this compensation, the maximum achievable PCM output level would remain several decibels below full scale (approximately -3 dBFS).

The magnitude transfer function from the digital microphones (DMICs) to the left/right PCM output samples is given by:

$$PCM\_Output = DMIC\_Input + PGA\_Gain + Compensation\_Gain$$

where DMIC_Input represents the effective signal level derived from the PDM bitstream after decimation.

The Compensation Gain for this block is approximately +3.6 dB (nominally +3.52 dB).

When the PGA_L/R registers are set to 0x0, the programmable gain amplifier applies -12 dB of gain. When PGA_L/R are set to 0x8, the programmable gain is 0 dB, which is regarded as unity gain from DMIC input to PCM output, except for the additional Compensation Gain described above.

23.2.1 Compensation Gain Questions and Examples

Question 1: There is an additional gain in the system of approximately 3.5 dB. Where in the data path, from the initial DMIC output to the final data read by the CPU, could ~3.5 dB of gain be coming from? The controls for PGA_L and PGA_R are set to a range of -12 to +34.5 dB.

- A. The extra conversion gain is an adjustment inside the PDM-to-PCM core conversion blocks in Figure 132. This extra gain is different from PGA_L/R, and it allows access to the full dynamic range of the PDM microphone. As the max amplitude of a typical PDM input data cannot produce 0 dBFS PCM output data due to the stability limitation of the delta-sigma conversion inside the digital microphone when the PDM-to-PCM converter includes a build-in gain-adjustment mechanism to map such maximum PDM input to close to 0 dBFS PCM output so that the CPU can utilize the full dynamic range. Otherwise, the max PCM output signal level would be less than -3 dBFS.

Question 2: What is the complete transfer function from the initial DMIC output to the final data read from the module registers by the CPU?

- A. The magnitude transfer function from the initial DMIC output to the final data read from the registers by the CPU is: $PCM_{Final} = PDM_{Init} + PGA + Compensation\_Gain$, where Compensation_Gain is ~3.5 dB in the PDM-to-PCM converter.

Question 3: Is unity gain expected when all values are set to 0?

- A. When PGA_L/R registers are set to 0, the gain is -12 dB; when PGA_L/R are set to 0x08, the gain is 0 dB which is regarded as unity gain from input to output except for the aforementioned extra compensation gain (~3.5 dB).

Question 4: What does it mean when the input signal is very high? Does it mean this compensation is input level dependent?

- A. No, the compensation is independent of level. Low amplitude and high amplitude are ‘scaled’ by the same amount. The reference to ‘very high’ was meant to explain why it’s needed. When the signal into the microphone is ‘too high’ then the resulting level signal out will scale nonlinearly as its amplitude gets “too close to the top” of the maximum range the microphone can produce. Therefore, if the definition of “high level signal” is moved down slightly, then even when the amplitude is high, the signal level will not be clipped.

Question 5: The formula specifies that, besides the compensation gain, there is PGA gain added too. Does the ~3.5 dB include both? How did it break down and does it matter?

- A. No, the PGA gain blocks are separate.

Question 6: Why does having this compensation allow us to see the full dynamic range? If it’s a linear gain and not input-level-dependent, by adding the gain to match to 0 dBFS at the maximum, the lowest level is added to the gain too. The dynamic range will still remain the same.

- A. If this ‘compensation gain’ is not added to the input path then the full dynamic range will be smaller. Figure 138 below indicates the purpose.

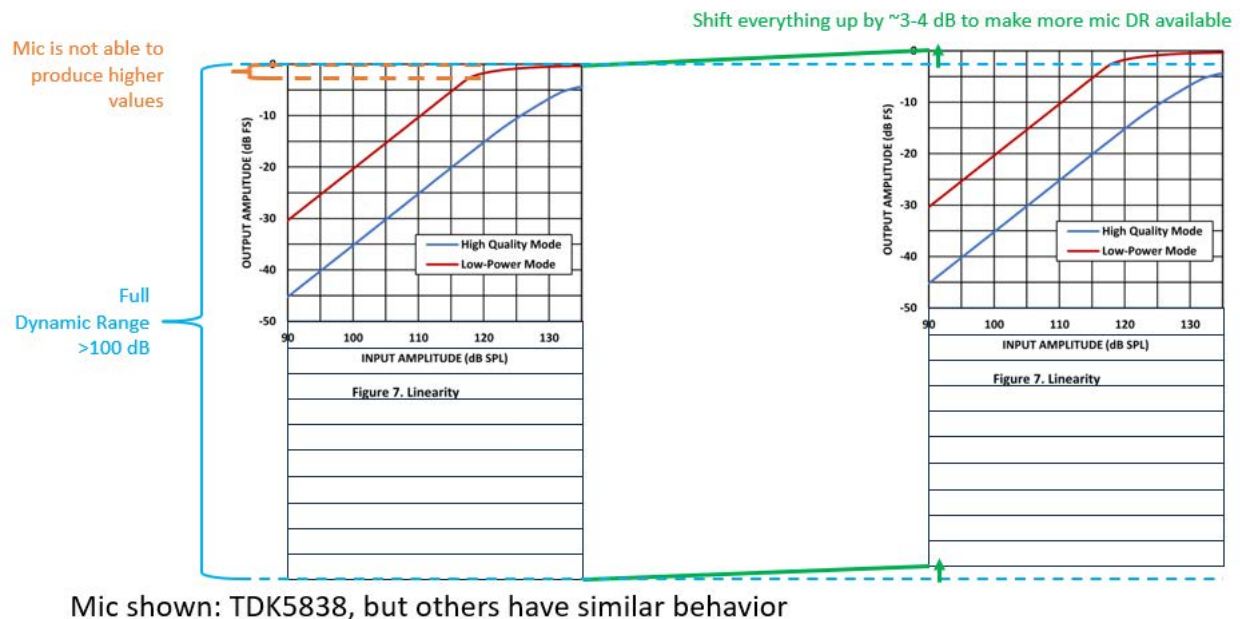


Figure 138. Compensation Gain Adjustment

23.3 Operating Modes

The PDM may operate in stereo or mono mode in normal operation, and system reset or power down mode when not in use. Each mode can be programmed by control registers. Note that mono mode operation always uses left channel conversion path.

Table 130: PDM-to-PCM Converter Operating Mode

Mode	CORE- CFG1_ PCM- CHSET [1:0]	CORE- CFG0_ LRSWAP	CTRL_EN	PDMA_ CKO	CTRL_ CLKEN	CTRL_ RSTB	PCMA_ DATA_L	PCMA_ DATA_R	PCMA_ _VALID
Stereo Non- swap	3	0	1	Active	Active	1	Normal Left	Normal Right	Normal
Stereo Swap	3	1	1	Active	Active	1	Normal Right Swapped	Normal Left Swapped	Normal
Mono Left	1	0	1	Active	Active	1	Normal Left	0	Normal
Mono Right	2	1	1	Active	Active	1	Normal Right Swapped	0	Normal
System Reset	X	X	X	X	X	0	0	0	0
Power Down	X	X	0	0	X	X	0	0	0

Notes:

1. "Active" means clock toggling at normal frequency.
2. Reasonable phase delay between PDMA_CKO and PDM_CLK is allowable.
3. RSTB is asynchronous reset signal.
4. At Power-Down mode, internal main clocks (MCLK_L and MCLK_R) are set to "0" to avoid toggling.
5. PCMA_VALID normal output pulse width is one PDM_CLK clock period.

23.4 Supported Data Format

The PDM module supports 24-bit unpacked data format. The following table shows the word sequence for single-channel (A) and dual-channel (A and B) audio.

Table 131: 24-bit Unpacked Data

Word Sequence	Single-channel		Dual-channel	
	Upper 8 bits	Lower 24 bits	Upper 8 bits	Lower 24 bits
1	0x00	A_t0[23:0]	0x00	A_t0[23:0]
2	0x00	A_t1[23:0]	0x00	B_t0[23:0]
3	0x00	A_t2[23:0]	0x00	A_t1[23:0]
4	0x00	A_t3[23:0]	0x00	B_t1[23:0]

23.5 Digital Volume Control

The PDM supports digital volume-control with a PGA gain range from -12 dB to +34.5 dB in 1.5 dB/step. It is programmed by register PGAL/PGAR for left/right channel respectively. PGA gain may be changed on-the-fly during normal operation. In order to reduce zipper or clip noise during gain transition, a built-in volume smoother is implemented with fine gain steps that enables softly ramp up or ramp down volume levels. The fine gain is set by register bit SELSTEP to 0.13 dB or 0.26 dB internally.

Table 132: PGA Gain Control

Register	Default	Description
CORECFG0_PGAL[4:0]	01000	Left Channel PGA Gain: +1.5dB/step, -12 dB ~ +34.5 dB
		00000 = -12 dB; 00001 = -10.5dB; ...
		01000 = 0 dB; ...
		11111 = +34.5 dB
CORECFG0_PGAR[4:0]	01000	Right Channel PGA Gain: +1.5 dB/step, -12 dB ~ +34.5 dB
		00000 = -12 dB; 00001 = -10.5 dB; ...
		01000 = 0 dB; ...
		11111 = +34.5 dB
CORECFG1_SELSTEP	0	Fine grain step size for smooth PGA or soft mute attenuation transition: 0 = 0.13 dB step size 1 = 0.26 dB step size

23.5.1 Soft Mute

The PDM contains a built-in software controlled mute function that digitally attenuates signals to imperceptible levels or zero. When mute function is enabled by setting SOFTMUTE = 1, the corresponding digital signal output is decreased from current level to mute state through predefined granular gain step per time constant transition. The time constant is set through register SCYCLES[2:0]. The slope of mute transition process is determined by SCYCLES and gain step. During soft-mute, the PDM is still on and clocks are toggling while user may desire to mute the recording tentatively. The soft mute configuration delivers smooth transition and reduces clip/zipper noise during the mute transition.

When the mute function is disabled by setting SOFTMUTE = 0, the mute function is off and the PDM goes back to normal operation when the output signal level returns to normal with the existing PGA gain.

Table 133: SOFTMUTE Register Configuration

Register/Field	Default	Description
CORECFG0_SOFTMUTE	0	Sets Soft-Mute function: 0 = Disable 1 = Enable
CORECFG0_SCYCLES[2:0]	001	Sets number of steps (PDMA_CKO cycles) during PGA gain setting changes or soft mute operation. 000 = 64 steps 001 = 96 steps 010 = 128 steps 011 = 160 steps 100 = 192 steps 101 = 256 steps 110 = 384 steps 111 = 512 steps
CORECFG1_SELSTEP	1	Sets fine gain step for smooth PGA or Soft-Mute attenuation transition. 0 = 0.13 dB 1 = 0.26 dB

23.6 Low Pass Filter (LPF)

The PDM's internal low pass filters attenuate the out-of-band noise at predefined band width and corners. Table 134 below describes the parameters of LPF performance.

Table 134: LPF Parameters

Parameter	Symbol	Test Condition	Min	Typ	Max	Unit
Pass band corner frequency	F _{PASS}		0.02	0.417*Fs	20	kHz
Pass band ripple		Peak-to-Peak		0.3	0.5	dB
Stop band corner frequency	F _{STCUT}			0.6		Fs
Stop band rejection	STA			-70		dB

23.7 High Pass Filter (HPF)

The PDM's high pass filter blocks DC offset and low frequency noise in the signal band. The filter response for the high pass filter is characterized as:

$$H(Z) = (1 - Z^{-1}) / [1 - (1 - 2^{-\text{HPGAIN}}) Z^{-1}]$$

In default mode, HPGAIN = 1011, so the high pass filter can be formulated by the polynomial:

$$H(Z) = (1 - Z^{-1}) / [1 - 0.99951Z^{-1}]$$

The user may tune the HPGAIN value to adjust the high pass filter cutoff corner frequency for better system configuration.

Table 135 shows the approximate -3dB cutoff corner frequencies for Sample Rate Fs = 48 kHz.

Table 135: PDM High Pass Filter -3 dB Cutoff Frequency Table

HPGAIN	$1 - 2^{-\text{HPGAIN}}$	-3 dB Cutoff Frequency (Hz)
0	0	12110
1	0.5	4979
2	0.75	2197
3	0.875	1031
4	0.9375	499
5	0.96875	244
6	0.984375	121
7	0.9921875	60
8	0.99609375	30.6
9	0.998046875	15.2
10	0.999023438	7.64
11	0.999511719	3.79
12	0.999755859	1.87
13	0.99987793	0.91
14	0.999938965	0.45
15	0.999969482	0.22

If F_s is different from 48 kHz, please scale the -3 dB cutoff frequency according to the ratio between the desired sample rate and the 48 kHz as a baseline.

24. Low Power Analog Audio Interface

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

This module is available on the Apollo510 and Apollo510B SoCs only.

24.1 Functional Overview

The Apollo510 and Apollo510B SoCs integrate a sophisticated 12-bit successive approximation Analog to Digital Converter (AUDADC) block for sensing both internal and external voltages. The block provides eight separately managed conversion requests, called slots which are serially sequenced. The result of each conversion request is delivered to a 16-deep FIFO. Firmware can utilize various interrupt notifications to determine when to collect the sampled data from the FIFO or from a buffer written by DMA. This block is extremely effective at automatically managing its power states and its clock sources.

The AUDADC can be configured for High Performance (Telco) mode or Low Power (Always On) mode. In High Performance mode, the PLL should be configured to provide a 48 MHz source clock by setting the MCUCTRL_PLLMUXCTL_AUDADCPLLCLKSEL register to PLL, with AUDADC_CFG_CLKSEL set to OFF.

In Low Power mode, the low power HFRC_48MHz clock source should be used by setting the AUDADC_CFG_CLKSEL to HFRC_48MHz.

The recommended sampling rate for both modes is 16 kHz for best performance.

Conversion of unsigned samples to signed data or DC offset adjustment may be performed using the 12-bit inline adder on data entering the FIFO using the OFFSET register.

As with prior Ambiq ADC designs, there are a number of off-chip and on-chip trigger sources. For repeating periodic triggers, one of the timer peripherals may be used, or, as is preferred for high quality audio sampling, the internal trigger timer may be employed as a low-jitter repeating trigger source that is synchronous to the ADC clock. It takes the ADC clock source selected via CLKSEL and may apply both a power-of-2 divider and an integer divider, if desired. See the INTTRIGTIMER register for appropriate bit fields.

24.1.1 Clock Source and Dividers

The Audio ADC clock can be sourced by HFRC_48MHz, HFRC2-derived 31.25 MHz, XTAL_HS (Apollo510 only), EXTREFCLK, or the FOUTPOSTDIV output of the PLL. When the Audio ADC block is enabled and has an active scan in progress, it requests a clock source. There is an automatic hardware hand shake between the clock generator and the Audio ADC. If the Audio ADC is the only block requesting an HFRC based clock, then the HFRC will be automatically started. The Audio ADC can be configured to completely power down the HFRC between scans if the startup latency is acceptable or it can leave the HFRC powered on between scans if the application requires low latency between successive conversions.

24.1.2 Voltage Reference Source

The Audio ADC supports an external 1.2 V reference source, which has an internal buffer, to be used for the analog to digital conversion step.

24.1.3 Eight Automatically Managed Conversion Slots

The Audio ADC block contains eight conversion slot control registers, one for each of the eight slots. These can be thought of as time slots in the conversion process. When a slot is enabled, it participates in a conversion cycle. The Audio ADC's mode controller cycles through up to eight time slots each time it is

triggered. For each slot that is enabled, a conversion cycle is performed based on the settings in the slot configuration register for that slot. A slot is enabled when the SLEN_n field ($n = 0$ to 7) of the Slot_n Configuration register is set to one. See Table 136 below.

Table 136: SLOT Configuration Register

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	0 9	0 8	0 7	0 6	0 5	0 4	0 3	0 2	0 1	0 0
Reserved				ADSEL (# Samples to Accum)		TRKCYC				PRMODE	Reserved				CHSEL		Reserved				WCEN	SLEN									

As described above, the Channel Select bit field, CHSEL_n, specifies which one of the analog multiplexer channels will be used for the conversions requested for an individual slot.

The Precision Mode field, PRMODE, enables the selection of the slot's precision - 8-bit, 10-bit or 12-bit.

An additional number of AUDADC clock cycles may be added to the default value of 5 cycles to specify the input signal sampling/tracking time by setting the TRKCYC field. Five cycles are added to the value set in this field to determine the total number of sampling/tracking cycles.

Each of the eight conversion slots can independently specify:

- Analog Multiplexer Channel Selection
- Automatic Sample Accumulation

24.2 Main Configuration

There is only one supported setting for the sample mode as set in the SAMPMODE field of the ADCCFG register. It is PGA = 0x0 (PGA gain is programmable and can be selected in dB steps. A setting of 0x1 is not supported. PGA gain is set in the Gain register.

There are multiple off-chip and on-chip trigger sources. For repeating periodic triggers, one of the timer peripherals may be used, or, as is preferred for high quality audio sampling, the internal trigger timer may be employed as a low-jitter repeating trigger source that is synchronous to the ADC clock. It takes the ADC clock source selected via the CLKSEL field of the CFG register and may apply both a power-of-2 divider and an integer divider if desired. See the INTTRIGTIMER register for applicable bit fields.

24.2.1 Slot Configuration

The only input channel that can be selected in a slot configuration is channel 0 (PGA A0). Sample precision may be controlled on a per-slot basis and may be 8-, 10-, or 12-bits via the PRMODEN field of the SLNCFG registers ($n = 0$ to 7). However, it is recommended that all slots be configured to match since the ZX and saturation comparators do not normalize samples based on sample precision mode.

The TRKCYC_n field of the SLNCFG register control enables additional input signal sampling/tracking time in ADC clock cycles, if desired, before the sample is taken.

24.2.2 Sample Data Format

The FIFO and DMA sample data format contains a single low-gain (even slot number) sample in LP mode or both a high-gain (odd slot number) and low-gain sample from a single audio channel in MED mode. It also contains information about which microphone the sample was sourced from and low gain "meta" information.

Unlike the General Purpose ADC, FIFO fullness information is not provided in the sample data. If the FIFO is empty, it will read 0xFFFFFFFF. Additionally, the number of entries in the FIFO may be read from the FIFOSTAT register.

24.3 Automatic Sample Accumulation and Scaling

The Audio ADC block offers a facility for the automatic accumulation of samples without requiring core involvement. Thus up to 128 samples per slot can be accumulated without waking the core. This facilitates averaging algorithms to smooth out the data samples. Each slot can request from 1 to 128 samples in power-of-2 increments to be accumulated before producing a result in the FIFO.

NOTE

Each slot can independently specify how many samples to accumulate so results can enter the FIFO from different slots at different rates.

All slots write their accumulated results to the FIFO in exactly the same format regardless of how many samples were accumulated to produce the results. The precision mode for each determines the format for the FIFO data. 12-bit, 10-bit and 8-bit precision modes correspond to 12.6, 10.6 and 8.6 formats, respectively.

Each slot contains a 21-bit accumulator as shown in Table 137. When the Audio ADC is triggered for the last sample of an accumulation, the accumulator is cleared and the FIFO will be written with the final average value. When each active slot obtains a sample from the Audio ADC, it is added to the value in its accumulator.

If a slot is set to accumulate 128 samples per result, then the accumulator could reach a maximum value of:

$$128 * (2^{14} - 1) = 128 * 16383 = 2097024 = 2^{21} - 128, \text{ hence the 21 bit accumulator.}$$

Table 137: Per Slot Sample Accumulator

2	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
Accumulator																				

Table 138 shows the maximum possible accumulated values. Note that for the 128 sample accumulation case, the LSB of the accumulator is discarded when the results are written to the FIFO.

Table 141: 10-bit FIFO Data Format

# Samples	1	1	1	1	1	1	1	1	1	1	0	9	8	7	6	5	4	3	2	1	0
128	0	0	0	0																	
64	0	0	0	0																	
32	0	0	0	0																	X
16	0	0	0	0																X	X
8	0	0	0	0															X	X	X
4	0	0	0	0														X	X	X	X
2	0	0	0	0												1	X	X	X	X	X
1	0	0	0	0												X	X	X	X	X	X

Table 142: 8-bit FIFO Data Format

# Samples	1	1	1	1	1	1	1	1	1	1	0	9	8	7	6	5	4	3	2	1	0
128	0	0	0	0	0	0															
64	0	0	0	0	0	0															
32	0	0	0	0	0	0															X
16	0	0	0	0	0	0														X	X
8	0	0	0	0	0	0													X	X	X
4	0	0	0	0	0	0												X	X	X	X
2	0	0	0	0	0	0											X	X	X	X	X
1	0	0	0	0	0	0										X	X	X	X	X	X

Software accesses the contents of the FIFO through the FIFO register. This register will be written by the Audio ADC digital controller simultaneous with the conversion complete interrupt (if enabled) after accumulating the number of samples to average configured for the slot. The FIFO register contains the earliest written data, the number of valid entries within the FIFO and the slot number associated with the FIFO data. Thus the interrupt handler servicing Audio ADC interrupts can easily distribute results to different RTOS tasks by simply looking up the target task using the slot number from the FIFO register.

Three other features greatly simplify the task faced by firmware developers of interrupt service routines for the Audio ADC block:

1. The FIFO count bit field is not really stored in the FIFO. Instead it is a live count of the number of valid entries currently residing in the FIFO. If the interrupt service routine was entered because of a conversion then this value will be at least one. When the interrupts routine is entered it can pull successive sample values from the FIFO until this bit field goes to zero. Thus avoiding wasteful re-entry of the interrupt service routine. Note that no further I/O bus read is required to determine the FIFO depth.
2. This FIFO has no read side effects. This is important to firmware for a number of reasons. One important result is that the FIFO register can be freely read repetitively by a debugger without affecting the state of the FIFO. In order to pop this FIFO and look at the next result, if any, one simply writes any value to this register. Any time the FIFO is read, then the compiler has gone to the trouble of generat-

ing an address for the read. To pop the FIFO, one simply writes to that same address with any value. This give firmware a positive handshake mechanism to control exactly when the FIFO pops.

3. When a conversion completes resulting in hardware populating the 12th valid FIFO entry, the FIFO_OVR1 (FIFO 75% full) interrupt status bit will be set. When a conversion completes resulting in hardware populating the 16th valid FIFO entry, the FIFO_OVR2 interrupt status bit will be set. In a FIFO full condition with 16 valid entries, the Audio ADC will not overwrite existing valid FIFO contents. Before subsequent conversions will populate the FIFO with conversion data, software must free an open FIFO entry by writing to the FIFO Register or by resetting the Audio ADC by disabling and enabling the Audio ADC using the CFG register.

24.5 DMA

When enabled, the Audio ADC can use DMA to keep its FIFO serviced and transfers samples to SRAM. Generally, DMA should be used when the desired use case is autonomous recording of samples to a pre-allocated buffer in SRAM. The buffer may be byte-aligned but must be a word-multiple in size.

The general steps to enabling Audio ADC DMA are as follows:

1. Ensure SRAM target(s) are powered up.
2. Power up the Audio ADC if it's not already on.
3. Configure ADC slots and CFG register.
4. Set DMATOTCOUNT to the total amount of data to transfer. While the DMA is in progress, this register contains a live count of the remaining data to transfer.
5. Configure DMATARGADDR, the SRAM target byte address, for the location in memory of the first sample to be written by DMA.
6. Select a DMA trigger level by configuring DMATRIGEN to either FIFO 100% full or FIFO 75% full. This defines what conditions will initiate a DMA transfer.
7. Configure DMACFG, including setting DMAEN.
8. Trigger the Audio ADC multiple times, using either the timer trigger (when using repeat mode), multiple SW triggers, or multiple external triggers.

Each time the FIFO fills to the appropriate level, the DMA will start and the FIFO will be drained. During this time, depending on the particular use case, it may be appropriate to put the SoC to sleep or deep sleep.

To monitor progress of the DMA, there is a DMASTAT status register. When the DMA is actively transferring data from the Audio ADC FIFO to SRAM, DMATIP will be asserted. At the end of an entire transfer (DMATOTCOUNT reaches 0), then DMACPL will be set. Last, but not least, if an error occurs due to the DMA being asked to perform an illegal operation, DMAERR will be asserted. Causes of a DMA error include:

- DMA transfers to address outside SRAM memory region
- Popping from the FIFO while the DMA is underway

Care must be taken to avoid powering down SRAM that the DMA wants to write to.

If the DMA complete interrupt is enabled, this can be used to wake the SoC from sleep or deep sleep and communicate that the SRAM buffer has been filled and is ready for processing. The DMA error interrupt may also be used to signal the SoC that there is a problem with the DMA configuration.

To recover from a DMA error, disable any repeating trigger, disable the DMA via DMACFG's DMAEN field, and manually drain the Audio ADC FIFO. Then follow the procedure described above for enabling Audio ADC DMA while correcting the configuration issue.

Some additional capabilities of the DMA include:

- Audio ADC auto-power-off upon DMA completion: This feature, enabled via the DMACFG register's DPWROFF field, allows the Audio ADC to power off once DMATOTCOUNT reaches zero. Note that this feature is incompatible with waking the SoC from sleep or deep sleep using the DMA complete interrupt.
- Masking FIFOCNT and SLOTNUM data from FIFO data: The DMA engine can be configured to write only samples to SRAM without the FIFOCNT and SLOTNUM data. This allows the SoC to skip the manual process of masking the potentially undesirable upper bits of each data value written to SRAM.

24.6 Operating Modes and the Mode Controller

The mode controller is a sophisticated state machine that manages not only the time slot conversions but also the power state of the Audio ADC analog components and the hand shake with the clock generator to start the HFRC clock source if required. Thus once the various control registers are initialized, the core can go to sleep and only wake up when there are valid samples in the FIFO for the interrupt service routine to distribute. Firmware does not have to keep track of which block is using the HFRC clock source since the devices in conjunction with the clock generator manage this automatically. The Audio ADC block's mode controller participates in this clock management protocol.

From a firmware perspective, the Audio ADC mode controller is controlled from bit fields in the Audio ADC configuration register and from the various bit fields in the eight slot configuration registers.

The most over-riding control is the PWRENAUDADC enable bit in the PWRCTRL_AUDSSPWREN register of the power control block. This bit must be set to '1' to enable power to the Audio ADC subsystem. Furthermore, the ADCEN bit in the Audio ADC configuration register is a global functional enable bit for general Audio ADC operation. Setting this bit to zero has many of the effects of a software reset, such as resetting the FIFO pointers. Setting this bit to one enables the mode controller to examine its inputs and proceed to autonomously handle analog to digital conversions.

An Audio ADC scan is the process of sampling the analog voltages at each input of the Audio ADC following a trigger event. If the Audio ADC is enabled and one or more slots are enabled, a scan is initiated after the Audio ADC receives a trigger through one of the configured trigger sources. The scan flowchart diagram can be found in Figure 139

An Audio ADC conversion is the process of averaging measurements following one or more scans for each slot that is enabled.

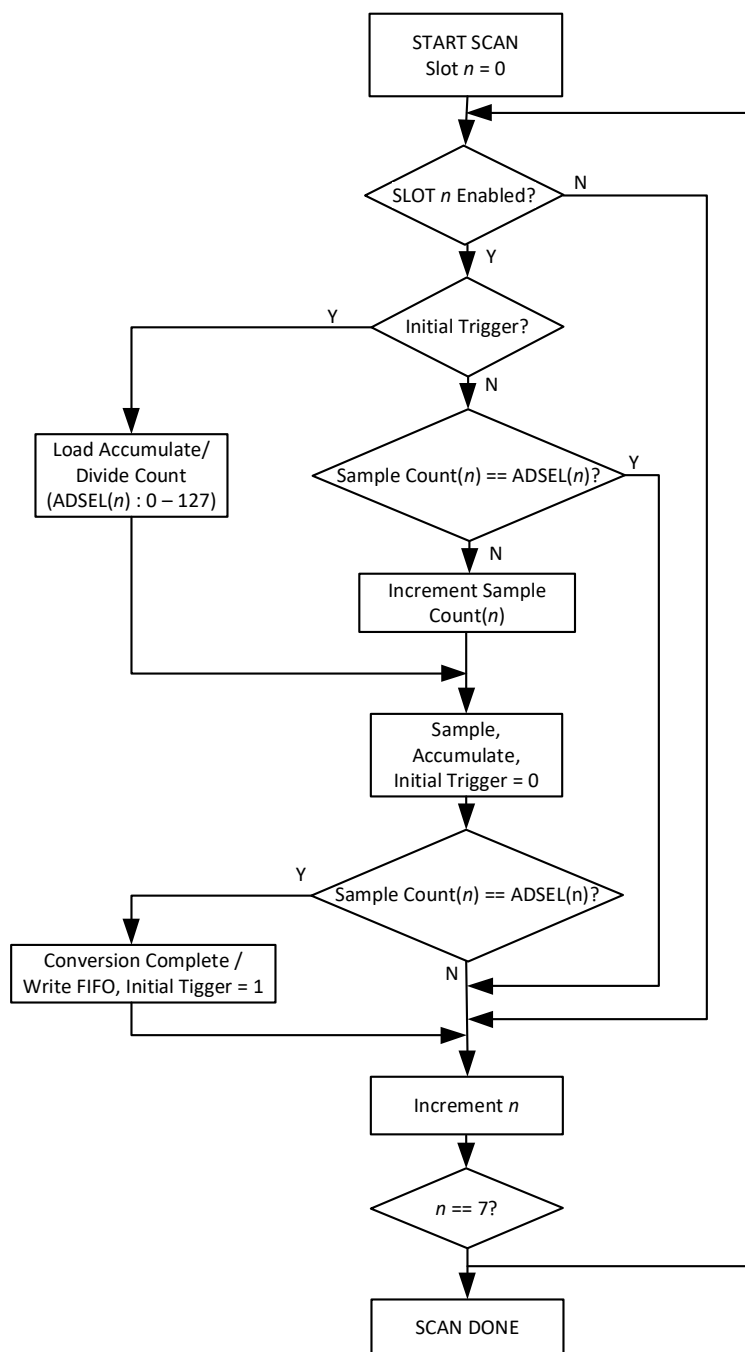


Figure 139. Scan Flowchart

24.6.1 Single Mode

In single mode, one trigger event produces one scan of all enabled slots. Depending on the settings of the accumulate and scale bit field for the active slots, this may or may not result in writing a result to the FIFO. When the trigger source is an external pin then one external pin transition of the proper polarity will result in one complete scan of all enabled slots. If the external pin is connected to a repetitive pulse source then repeating scans of all enabled slots are run at the input trigger rate.

24.6.2 Repeat Mode

Counter/Timer 7 has a bit in its configuration register that allows it to be a source of repetitive triggers for the Audio ADC. If counter/timer 7 is initialized for this purpose then one only needs to turn on the RPTEN bit in the Audio ADC configuration register to enable this mode in the Audio ADC.

NOTE

The mode controller does **not** process these repetitive triggers from the counter/timer until a first triggering event occurs from the normal trigger sources. Thus one can select software triggering in the TRIGSEL field and set up all of the other Audio ADC registers for the desired sample acquisitions. Then one can write to the software trigger register and the mode controller will enter REPEAT mode. In repeat mode, the mode controller waits only for each successive counter/timer 3A input to launch a scan of all enabled slots.

24.6.3 Low Power Modes

An application may use the Audio ADC in one of three power modes. Each mode has different implications from overall energy perspective relative to the startup latency from trigger-to-data as well as the standby power consumed. The table below is intended to provide guidance on which mode may be more effective based on latency tolerance. This table should only be used as a reference.

Table 143: Audio ADC Power Modes

LPMODE	Definition	Entry Latency
0	Audio ADC is kept active continuously (used in continuous sampling scenarios)	0
1	Audio ADC is mostly powered off between samples, HFRC is duty cycled between samples.	<70 μ s (shorter for lower resolution)
2	Audio ADC is completely powered off between samples, HFRC is duty cycled between samples.	<660 μ s

24.6.3.1 Low Power Mode 0

Low Power Mode 0 (LPMODE0) enables the lowest latency from trigger to conversion data available. This mode leaves the reference buffer powered on between scans to bypass any startup latency between triggers¹.

24.6.3.2 Low Power Mode 1

Low power mode 1 (LPMODE1) is a power mode whereby the Audio ADC Digital Controller will automatically power off the Audio ADC clocks, analog Audio ADC and reference buffer between scans. This mode may operate autonomously without CPU interaction, even while the CPU is in sleep or deep sleep mode for repeat mode triggers or hardware triggers. While operating in this mode, the Audio ADC Digital Controller may be used to burst through multiple scans enabling max sample rate data collection if the triggers are running at a rate at least 2x the maximum sample rate until the final scan has completed. When a scan completes without a pending trigger latched, the Audio ADC subsystem will enter a low power state until the next trigger event.

¹. The reference buffer will not be powered on when the Audio ADC is configured for external reference

24.6.3.3 Low Power Mode 2

If desirable, for applications requiring infrequent conversions, software may choose to operate the Audio ADC in LPMODE2, whereby the full Audio ADC Analog and Digital subsystem remains completely powered off between samples. In this use case, the software configures the power control Audio ADC enable register followed by configuring the Audio ADC slots and the Audio ADC configuration register between conversion data collections, followed by disabling the Audio ADC in the power control Audio ADC enable register. Although this mode provides extremely low power operation, using the Audio ADC in this mode will result in a cold start latency including reference buffer stabilization delay.

IMPORTANT: the Audio ADC Digital Control logic is powered off when the Power Control Audio ADC enable bit (PWRCTRL_AUDSSPWREN_PWRENAUDADC) is set to zero. This clears any previously configured registers in the Audio ADC Digital Module necessitating re-configuration for any subsequent Audio ADC operation. In this mode, the Audio ADC must be reconfigured prior to any subsequent Audio ADC operation.

24.7 Interrupts

The Audio ADC has 8 interrupt status bits with corresponding interrupt enable bits, as follows:

1. Conversion Complete Interrupt
2. Scan Complete Interrupt
3. FIFO Overflow Level 1
4. FIFO Overflow Level 2
5. DMA Complete (DCMP)
6. DMA Error (DERR)
7. DMA transfer complete
8. DMA error condition

There are two interrupts based on the *fullness* of the FIFO. When the respective interrupts are enabled, Overflow 1 fires when the FIFO reaches 75% full, viz. 6 entries. Overflow 2 fires when the FIFO is completely full.

When enabled, the conversion complete interrupt fires when a single slot completes its conversion and the resulting conversion data is pushed into the FIFO.

When enabled, the scan complete interrupt indicates that all enabled slots have sampled their respective channels following a trigger event.

When a single slot is enabled and programmed to average over exactly one measurement and the scan complete and conversion complete interrupts are enabled, a trigger event will result in the conversion complete and scan complete interrupts firing simultaneously upon completion of the Audio ADC scan. Again, if both respective interrupts are enabled and a single slot is enabled and programmed to average over 128 measurements, 128 trigger events result in 128 scan complete interrupts and exactly one conversion complete interrupt following the 128 Audio ADC scans. When multiple slots are enabled with different settings for the number of measurements to average, the conversion complete interrupt signifies that one or more of the conversions have completed and the FIFO contains valid data for one or more of the slot conversions.

The DMA transfer complete interrupt is triggered upon completion of the currently configured DMA.

The DMA error interrupt is triggered if the DMA has been instructed to perform an illegal operation such as:

- Writing outside SRAM
- Writing to powered-down SRAM
- Popping from the FIFO while DMA is underway

24.8 Generating the Sample Rate for the Audio ADC

TIMER6 of the Timer Module has a special function which allows it to function as the sample trigger generator for the Audio ADC. If the `TIMER_GLOBEN_AUDADCEN` bit is set, the output of the timer is sent to the Audio ADC which uses it as a trigger. Typically, TIMER6 is configured in Repeatable Up-counter Compare (UPCOUNT - FN = 2) mode. `INTEN_TMR60INT` may be set to generate an interrupt whenever the trigger occurs, but typically the Audio ADC interrupt will be used for this purpose.

25. Inter-IC Sound (I²S)

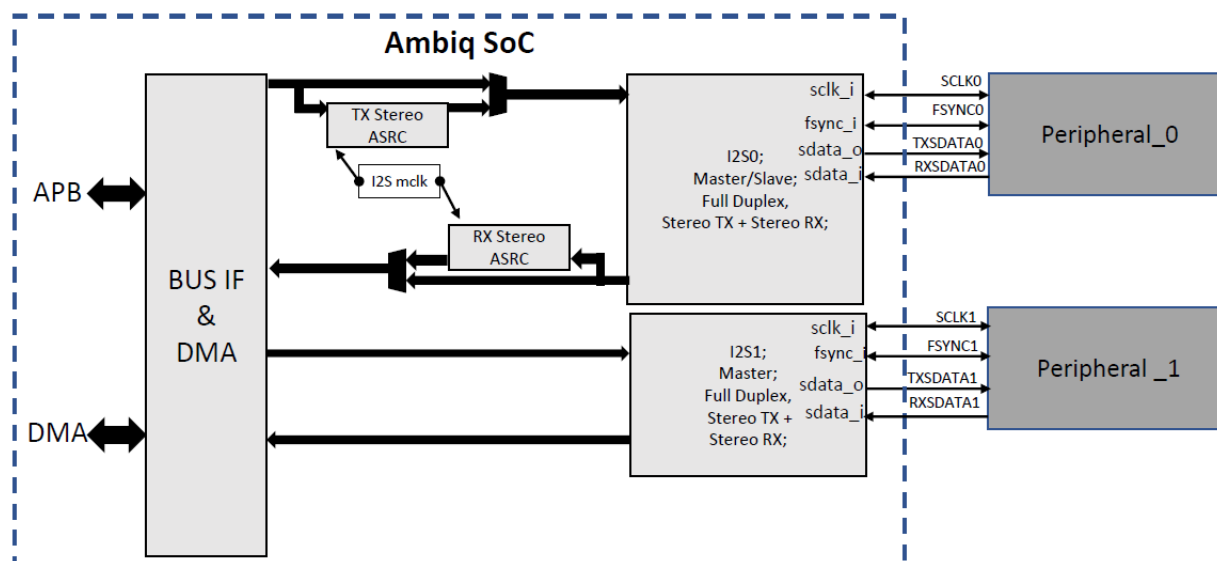
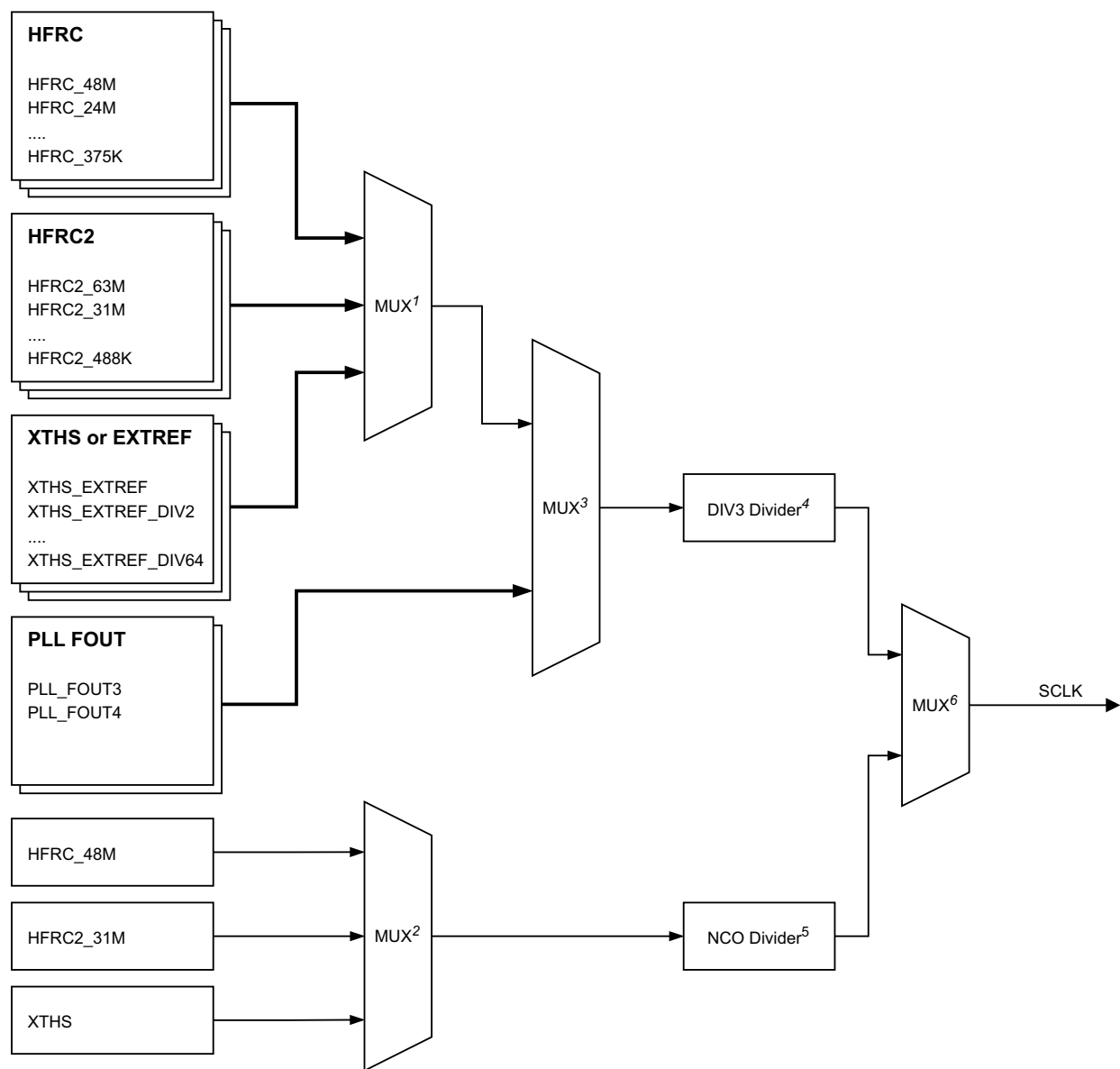


Figure 140. I²S Block Diagram

Please refer to this module's feature list and functional overview sections in the datasheet for an overview of the module's capabilities. Also refer to the latest SoC errata list posted in the Ambiq Content Portal as well as the module's register set included in the latest version of the AmbiqSuite SDK for the Apollo5 family's product(s) of interest.

25.1 I²S Clock Management

Clock Management consists of a main clock source selection and the two-stage divider configuration. Figure 141 illustrates the serial clock generation in manager/master mode.



¹A multiplexer controlled by the FSEL field of the I2Sn_CLKCFG register.

²A multiplexer controlled by the REFSEL field of the I2Sn_CLKCFG register.

³A multiplexer controlled by the I2SnPLLCLKSEL field of the MCUCTRL_PLLMUXCTRL register.

⁴A clock divider with a fixed divide ratio of 3, controlled by the DIV3 field of the I2Sn_CLKCFG register.

⁵A fixed-point fractional frequency divider with a minimum divide ratio of 4.0, controlled by the I2Sn_INTDIV and I2Sn_FRACDIV registers.

⁶A multiplexer controlled by the MCLKSEL field of the I2Sn_AMQCFG register.

Figure 141. I²S Serial Clock Generation in Manager Mode

25.2 Transmit and Receive FIFOs

The Receive FIFO and Transmit FIFO decouple the I²S clock domain and the back-end bus interface clock domain. FIFO status and current number of samples are accessible by means of memory-mapped transmit and receive FIFO status registers. The back-end interface supports blocking transactions.

The user-accessible register set contains configuration and information registers such as the number of words currently in the FIFO, its maximum capacity, the programmable RXUPPERLIMIT register, etc.

25.3 DMA

DMA is also provided to allow the automatic transfer of a block of data to and from the module. The interface multiplexes the full duplex I²S data between the system memory and the external pins using a single DMA interface internally. Data is provided to the DMA logic by either the I²S module or the system fabric, and sent to either the I²S module or system fabric.

See “Two-Stage DMA Request Pipeline Feature” on page 24 for modules supporting two-stage DMA request pipeline.

Enabling the DMA operation requires setup in both sub modules. The I²S module must be programmed with the I²S setup, and to activate the FIFO watermark signals to the interface logic. The interface logic must be updated with the DMA addressing, priority and size parameters, and then enabled for the desired directions. Interrupts should also be enabled to alert the processor on various conditions.

Once programmed and enabled, the DMA data transfer is performed using a series of transfers in a fixed block size of 8 32-bit samples. The internal DMA/APB bus is used for this transfer and, if both TX and RX DMA are enabled, will arbitrate between these as needed to store or fetch the appropriate data.

Arbitration is done in a round robin fashion between the TX and RX when both are active simultaneously. The arbitration occurs before each block of DMA (8 samples) is transferred, and will grant either the alternate direction between transfers if there are active requests for the transfer. There is no mandated order for which DMA direction is done other than the alternate direction is selected if the request is active and was not selected for the previous block of data transfer.

DMA request are created within the I²S module based on FIFO space availability (TX) or filled level (RX). These two FIFOs in the I²S module are separate and independent. Two registers, TXLOWERLIMIT register and the RXUPPERLIMIT register, are used to program the DMA transfer trigger condition and interrupt condition when TXFFM or RXFFM is set. During TX operation, if the TX FIFO falls below the number of samples programmed into TXLOWERLIMIT, it will assert a request to the DMA logic to fetch data and store this into the TX FIFO for transmission out of the chip. Similarly, when there is the same or more samples in the RX FIFO as set in RXUPPERLIMIT, an RX transfer request will be asserted to transfer data from the RX FIFO to the internal system memory.

Due to the fixed-size block transfer of 8 samples (8 32-bit words), TXLOWERLIMIT must not be programmed to more than TX FIFO SIZE - 8, as this could cause overruns.

If RXDMATOTCNT is greater than or equal to 8, set RXUPPERLIMIT to within the range of 7 to 63. It is recommended to set RXUPPERLIMIT to 7 and configure RXDMATOTCNT to a multiple of 8. With this configuration, the receiver's DMACPL interrupt can be triggered immediately after the I²S transmitter has sent sufficient data. Otherwise, the receiver must receive a small amount of additional data before the DMACPL interrupt is triggered.

if RXDMATOTCNT is less than 8, set RXUPPERLIMIT to RXDMATOTCNT - 1.

The FIFO size for both the TX and RX FIFOs is 64 32-bit samples.

The address and size of the DMA must be programmed into the interface registers prior to enabling the DMA operation. DMA operations for TX and RX are independent and can operate alone or simultaneously with the other I²S DMA operation. Once the DMA is enabled, the system will continue to transfer data in blocks of 8 samples until the TOTCNT number of samples (TXTOTCNT in the TXDMATOTCNT register or RXTOTCNT in the RXDMATOTCNT register) is transferred. Note that the TOTCNT value need not be a multiple of 8. Transfers will be done in units of 8, and the last transfer of the DMA will be 8 or fewer samples, as needed to complete the TOTCNT to zero.

The DMAADDRESS registers and the TXDMATOTCNT/RXDMATOTCNT registers are updated with each block of data transferred and can be read at any time by the processor without affecting ongoing DMA operations. However, the registers cannot be written unless the DMA is disabled by writing the TXDMAEN

or RXDMAEN bits in the DMACFG register. Once the DMA is disabled, it will stop making requests and transferring data. The maximum size of a single DMA is 4K samples (32-bit words), which would require 16 kB of storage space.

For each direction (TX or RX), interrupts can be used to alert the processor(s) to the completion of a DMA operation, or after a programmable number of blocks have been transferred.

Once completed, software must first write the DMACFG register to 0 prior to making any updates to either DMAADDR register or either DMATOTCNT register.

NOTE

If the I2S is operating in 8-bit or 16-bit mode, before restarting a RX DMA, it is necessary to power cycle the I2S for Apollo510, and to reset the I2S through the CRM block for the Apollo510 Lite Series and Apollo330 Plus Series.

25.4 Interrupts

The interface control logic provides interrupts for DMA operations, and forwards the interrupt from the I²S module through the single interrupt signal from the module instance. There are 5 main interrupts available, of 3 different types.

The RX/TX DMACPL interrupt asserts when the programmed DMA completes, or end with an error condition. An error condition is indicated with a 1 in the DMA status registers (TXDMASTAT and RXDMASTAT). If an error condition did occur during a DMA operation, the DMA must first be disabled, then the DMAERR bit cleared by writing a '0' to it. Once this is done, the DMA operation can be reprogrammed and restarted again.

The TX/RX REQCNT interrupt will assert every time the DMA engine transfers a certain number of data blocks (8 samples) of the DMA. The count is programmed into the DMACFG register prior to starting the DMA. If the count register is written with 0 or 1, it will assert for each transfer of 8 samples for the DMA. If programmed to '3' for example, it would assert the interrupt on the completion of every 3rd transfer block.

The last interrupt, IPB, is a level interrupt from the I²S module and is controlled with register settings within this module. This is a level based interrupt, and is used within the edge trigger interrupt system, so care must be taken to ensure that the level is de-asserted prior to re-enabling this interrupt.

25.5 Data Configurations

Table 144 lists the configurations supported by the I²S module.

Table 144: Configurations Supported by the I²S Module

Feature	Non-ASRC mode	ASRC mode
Audio sample sizes (bits)	8, 16, 24, 32	8, 16, 24, 32
Maximum number of channels	8	2
Manager/Subordinate modes	manager or subordinate	subordinate ¹
I ² S/TDM modes	Any supported by the I ² S, with the limitations indicated above	Any supported by the I ² S, with the limitations indicated above

1. The I²S also can be configured in manager mode. However, since the *mclk* signal is used to generate the *sclk* signal, the source ratio will be always 1:1.

Figure 142 and Figure 143 present the packing of the audio samples in the DMA, for all possible audio sample sizes. Those figures illustrate the waveforms for the I²S mode (slots of 32 bits, left justification and one delay bit). However, the module can support all I²S and TDM modes described in this datasheet, with the limitations indicated in Table 144.

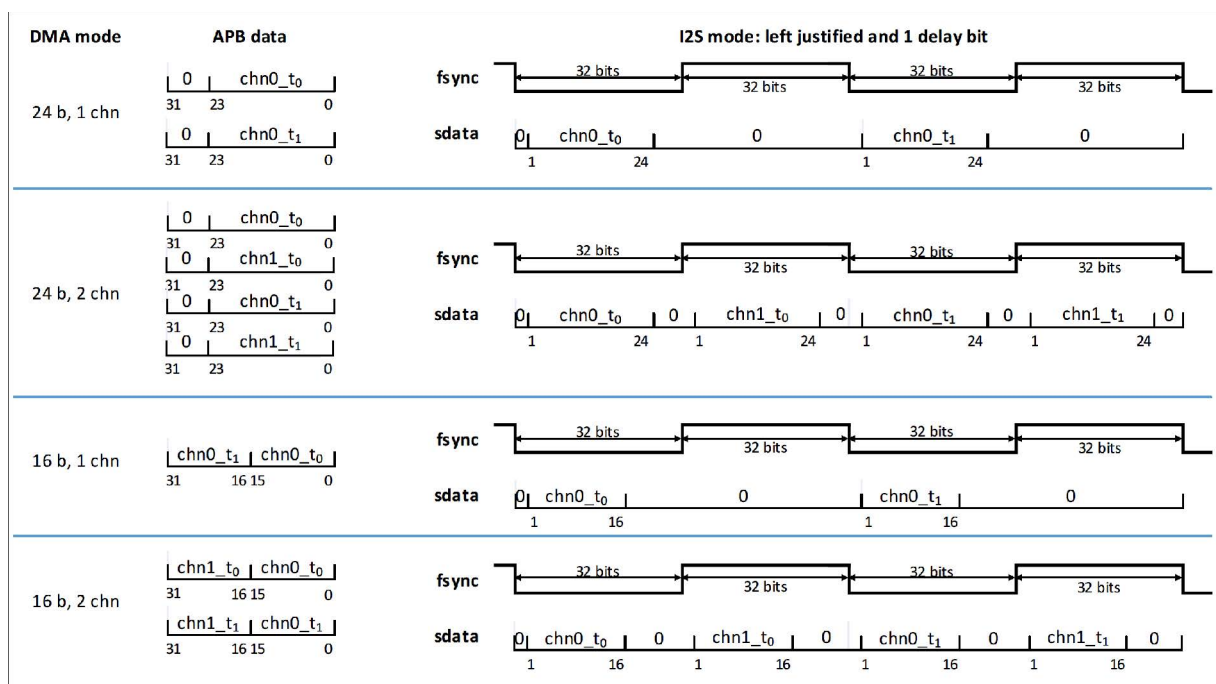


Figure 142. DMA Configurations for 16 Bits and 24 Bits

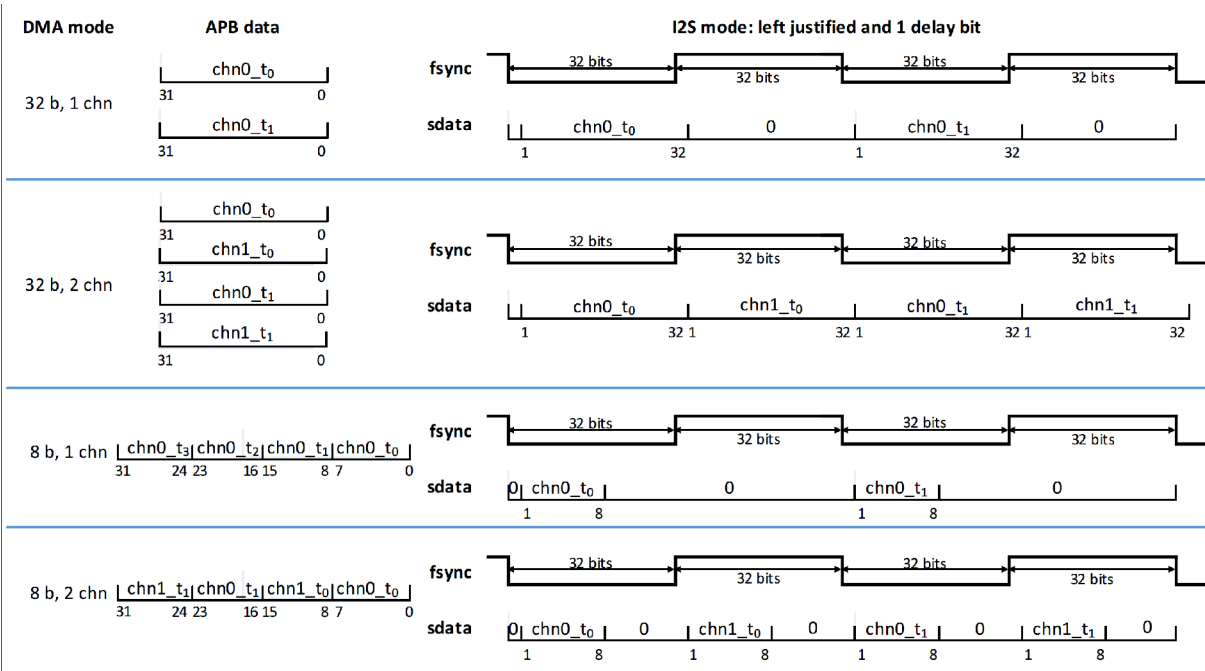


Figure 143. DMA Configurations for 8 Bits and 32 Bits

25.6 Configuration and Control

Each I²S instance has several configuration and control options in the register set for software driver development and use. Three aspects of configuration and control are important: manager/subordinate configurations, I²S initialization, and data flow control using receive and transmit FIFOs.

25.6.1 Manager/Subordinate Configurations

Each I²S instance can be configured to operate in subordinate or manager mode. At power-up, they are configured to act as subordinates to avoid signal collisions with other modules on the chip. The operating mode can be changed by writing the appropriate value to the MSL (Manager/Subordinate Configuration) bit in the I2S IO Configuration (I2SIOCFG) register.

25.6.2 Module Initialization

The following steps outline the procedure for initializing the I²S interface. Unless otherwise noted, referenced registers are in the I²S register documentation.

1. Reset the serial receiver or transmitter by asserting bits RXRST and/or TXRST in the I2SCTL register.
2. Specify the polarity of the I²S IO signals by writing the desired configuration to the I²S IO Configuration (I2SIOCFG) register.
3. Specify the serial data format in the I²S Data Format Configuration (I2SDATACFG) register.
4. To generate interrupts triggered by the number of samples in the FIFOs, do as follows:
 - A. Read the Receive FIFO Size register (RXFIFOSIZE) or the Transmit FIFO Size register (TXFIFOSIZE) to know its size, if not known already.
 - B. Write the desired values to the Receive FIFO Upper Limit register (RXUPPERLIMIT) and/or the Transmit FIFO Lower Limit register (TXLOWERLIMIT).
 - C. Unmask the RXFFI or TXFFI interrupt bits by asserting the RXFFM or TXFFM mask bits in the IPB Interrupt Mask register (IPBIRPT).
5. To generate interrupts when the next channel to be received becomes uncertain, due to a Transmit FIFO underrun or Receive FIFO overrun condition, unmask the TXEI and RXFI interrupt bits by asserting the TXEM or RXFM mask bits in the IPB Interrupt Mask register (IPBIRPT).
6. Write samples to the Transmit FIFO so that the transmit FIFO has samples above the FIFO Lower Limit value.
7. To start receiving and/or transmitting audio streams, enable the receiver and/or transmitter by asserting the RXEN and/or TXEN bits in the I²S Control register (I2SCTL). Note that for full-duplex operation, the TXEN and RXEN bits must be asserted in a single register access.

25.6.3 Data Flow Control

25.6.3.1 Receiver Data Flow Control

1. Poll the number of samples field or the Empty bit in the Receive FIFO Status register (RXFIFOSTATUS), and take action to prevent the FIFO from becoming full.
2. Alternatively, set the RXFFM bit in the IPB Interrupt Mask register (IPBIRPT) to enable the RXFFI interrupt bit when the number of samples in the Receive FIFO exceeds the value in the Receive FIFO Upper Limit register (RXUPPERLIMIT). The RXFFI interrupt bit will clear after the number of samples in the Receive FIFO is below the value in RXUPPERLIMIT.
3. If the Receive FIFO does become full, the newly received samples are dropped, and an overrun condition is triggered by means of the RXFI interrupt bit, if the RXFM mask bit is enabled. The overrun condition means that the order of the received channels (channel synchronization) has become uncertain.

4. To regain synchronicity, disable the receiver by de-asserting the RXEN bit, reset the receiver by asserting the RXRST bit, and re-enable it by asserting the RXEN bit; these bits are all in the I²S Control register (I2SCTL). After this operation, the module will resume receiving samples starting from the first channel of a new frame.
5. During normal operation (before an overrun condition has occurred) use the Receive Channel ID register (RXCHANID) to know the last channel received. After an overrun condition has occurred, the contents of RXCHANID become meaningless.

25.6.3.2 Transmitter Data Flow Control

1. Poll the number of samples field or the Full bit in the Transmit FIFO Status register (TXFIFOSTATUS), and take action to prevent the Transmit FIFO from becoming empty.
2. Alternatively, set the TXFFM bit in the IPB Interrupt Mask register (IPBIRPT) to enable the TXFFI interrupt bit when the number of samples in the Transmit FIFO drops below the value in the Transmit FIFO Lower Limit register (TXLOWERLIMIT). The TXFFI interrupt bit will clear after the number of samples in the Transmit FIFO is above the value in TXLOWERLIMIT.
3. If the Transmit FIFO does become empty, a zero is transmitted, and an underrun condition is triggered by means of the TXEI interrupt bit, if the TXEM mask bit is enabled. The underrun condition means that the order of the transmitted channels (channel synchronization) has become uncertain.
4. To regain synchronicity, disable the transmitter by de-asserting the TXEN bit, reset the transmitter by asserting the TXRST bit, clear the TXRST bit to take the transmitter out of the reset state, write samples to the Transmit FIFO until it contains more samples than the value in the Transmit FIFO Lower Limit register (TXLOWERLIMIT), and finally re-enable the transmitter by asserting the TXEN bit; the TXEN and TXRST bits are all in the I²S Control register (I2SCTL). After this operation, the module will resume receiving samples starting from the first channel of a new frame.
5. During normal operation (before an overrun condition has occurred) use the Transmit Channel Identification register to know the next channel to be transmitted. After an underrun condition has occurred, the contents of the Transmit Channel Identification register become meaningless.

25.7 Serial Audio Interface

25.7.1 TDM Serial Audio Format

Figure 144 illustrates the relationship between the behavior of the frame sync, *fsync*, signal and the configuration parameters *FPER* (Frame Period) and *FWID* (*fsync* width) in the I²S IO Configuration (I2SIOCFG) register. The *fsync* signal comes out of the Apollo5 on one of the pins offering I2Sx_WS functionality. Refer to the Pin Mapping Table in the GPIO chapter.

The *FPER* parameter has a maximum value of 4095 and measures the distance in *sclk* cycles between two consecutive *fsync* assertions. The *FWID* parameter has a maximum value of 255 and indicates the number of *sclk* cycles during which *fsync* will stay asserted. When the module is configured as a subordinate, these registers are not used.

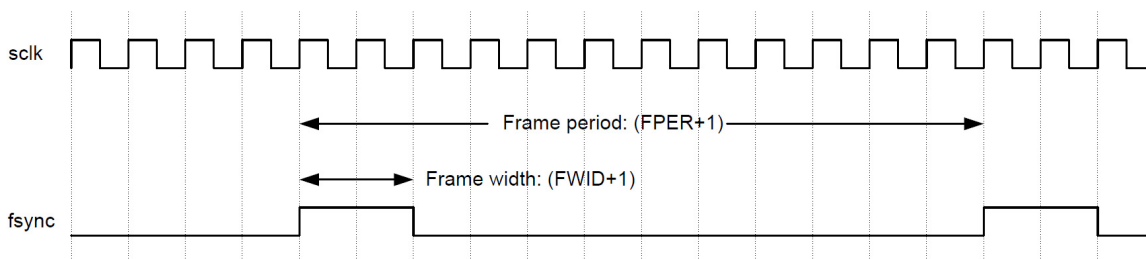


Figure 144. Programmable Frame Period and Width

A timing diagram of a single channel 8-bit sample audio serial interface signal can be observed in Figure 145. A frame starts in the first clock cycle in which the frame synchronization, *fsync*, is active. The data contents can be delayed with respect to the frame start using the *DATADLY* parameter in the Control register, which can be configured for 0-bit, 1-bit or 2-bit delay as shown in the figure.

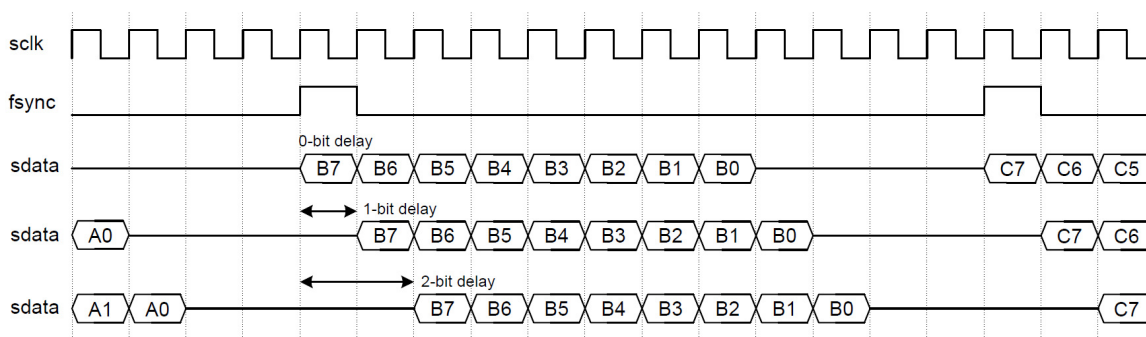


Figure 145. Data Delay

For example, for a 1-bit delay, the module must be configured as follows [see the I²S Data Format Configuration (I2SDATACFG) and I²S IO Configuration (I2SIOCFG) registers]:

- *PH* = 0, specifying a single-phase frame
- *FRLN1* = 0, specifying one channel per frame
- *WDLEN1* = 1h, specifying 12 bits per channel
- *DATADLY* = 1, specifying a 1-bit data delay
- *SSZ1* = 0, specifying an 8 bit audio sample for each channel

- FPER = 0Bh, specifying a frame period of 12 cycles
- FWID = 0h, specifying a 1-cycle frame sync pulse
- PRx = 0, specifying sclk rising edge sampling
- JUST = 0, specifying that the receive data is left-justified
- FSP = 1, specifying that an active high fsync signal is used

NOTE

In the examples in this section, 12 bits per channel sample size is used as the WDLEN1 parameter, even though only 8, 16, 24 and 32 bits per channel sample size is supported in the I²S modules on this device.

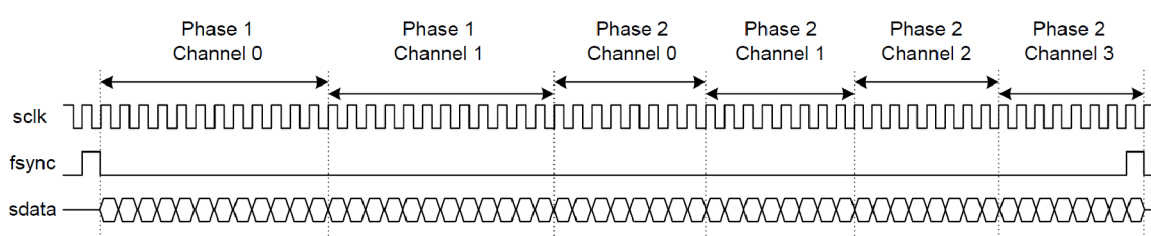


Figure 146. Dual-phase Frame Example

Figure 146 shows a TDM example with two phases. The module must be configured as follows [see the I²S Data Format Configuration (I2SDATACFG) and I²S IO Configuration (I2SIOCFG) registers]:

- PH = 1, specifying a two-phase frame
- FRLN1 = 1, specifying two channels in phase 1
- FRLN2 = 3, specifying four channels in phase 2
- WDLEN1 = 1h, specifying 12 bits per channel in phase 1
- WDLEN2 = 0h, specifying 8 bits per channel in phase 2
- DATADLY = 1, specifying a 1-bit data delay
- FPER = 38h, specifying a frame period of 56 cycles
- FWID = 0h, specifying a 1-cycle frame sync pulse
- PRx = 0, specifying sclk rising edge sampling
- JUST = 0, specifying that the receive data is left-justified
- FSP = 1, specifying that an active high fsync signal is used

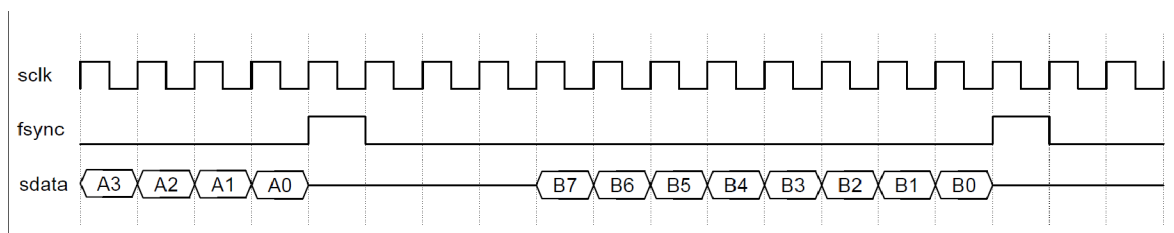


Figure 147. Right Justification

Figure 147 shows an audio frame using right justification. The module must be configured as follows [see the I²S Data Format Configuration (I2SDATACFG) and I²S IO Configuration (I2SIOCFG) registers]:

- PH = 0, specifying a single-phase frame
- FRLN1 = 0, specifying one channel per frame
- WDLN1 = 1h, specifying 12 bits per channel
- DATADLY = 0, specifying a 0-bit data delay
- SSZ1 = 0, specifying an 8-bit audio sample for each channel
- FPER = Bh, specifying a frame period of 12 cycles
- FWID = 0h, specifying a 1-cycle frame sync pulse
- PRx = 0, specifying sclk rising edge sampling
- JUST = 1, specifying that the receive data is left-justified
- FSP = 1, specifying that an active high fsync signal is used

25.7.2 I²S Serial Audio Format

Figure 148 shows an I²S formatted audio frame.

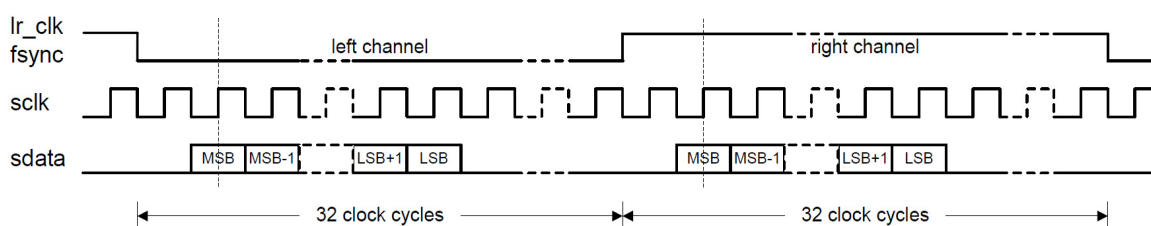


Figure 148. I²S Formatted Audio Frame

The module must be configured as follows [see the I²S Data Format Configuration (I2SDATACFG) and I²S IO Configuration (I2SIOCFG) registers]:

- PH = 0, specifying a single-phase frame
- FRLN1 = 1, specifying two channels per frame
- WDLN1 = 5, specifying 32 bits per channel
- DATADLY = 1, specifying a 1-bit data delay
- SSZ1 = 0, specifying an 8 bit audio sample for each channel
- FPER = 3Fh, specifying a frame period of 64 cycles
- FWID = 1Fh, specifying a frame sync pulse of 32 bits
- PRx = 0, specifying sclk rising edge sampling
- JUST = 0, specifying that the receive data is left-justified
- FSP = 0, specifying that active low fsync signals are used

25.7.3 Left-justified Serial Audio Format

Figure 149 shows a left-justified formatted audio frame.

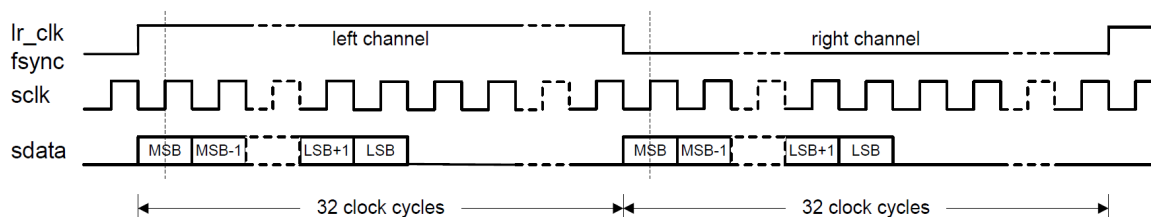


Figure 149. Left-justified Audio Frame

The module must be configured as follows [see the I²S Data Format Configuration (I2SDATACFG) and I²S IO Configuration (I2SIOCFC) registers]:

- PH = 0, specifying a single-phase frame
- FRLN1 = 1, specifying two channels per frame
- WDLN1 = 5, specifying 32 bits per channel
- DATADLY = 0, specifying a 0-bit data delay
- SSZ1 = 0, specifying an 8 bit audio sample for each channel
- FPER = 3Fh, specifying a frame period of 64 cycles
- FWID = 1Fh, specifying a frame sync pulse of 32 bits
- PRx = 0, specifying sclk falling edge sampling
- JUST = 0, specifying that the receive data is left-justified
- FSP = 1, specifying that active high fsync signals are used

25.7.4 Right-justified Serial Audio Format

Figure 150 shows a right-justified formatted audio frame.

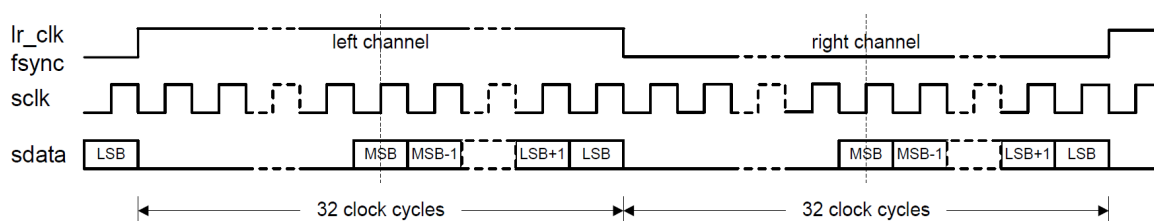


Figure 150. Right-justified Audio Frame

The module must be configured as follows [see the I²S Data Format Configuration (I2SDATACFG) and I²S IO Configuration (I2SIOCFC) registers]:

- PH = 0, specifying a single-phase frame
- FRLN1 = 1, specifying two channels per frame
- WDLN1 = 5, specifying 32 bits per channel
- DATADLY = 0, specifying a 0-bit data delay
- SSZ1 = 0, specifying an 8 bit audio sample for each channel
- FPER = 3Fh, specifying a frame period of 64 cycles
- FWID = 1Fh, specifying a frame sync pulse of 32 bits
- PRx = 0, specifying sclk falling edge sampling
- JUST = 1, specifying that the receive data is right-justified
- FSP = 1, specifying that active high fsync signals are used

26. BLE Controller

This section covers usage and programming topics of the BLE Controller. Communication between the Apollo510B SoC and the BLE Controller is handled over SPI while communication between the Apollo510 Lite Series SoCs and Apollo330 Plus Series SoCs and BLE Controller is handled over IPC.

26.1 Configuration

The BLE module should be run in one task (RadioTask).

26.2 Initialization

RadioTask initialization is as follows:

1. Boot the radio with `HciDrvRadioBoot(1)`.
 - A. Initialize the SPI module and XTAL for BLE controller operation. [Apollo510B uses the 48 MHz BLE XTAL⁽¹⁾; Apollo330 Plus and 510D Lite use a 48 MHz XTALHS crystal.]
 - B. Register the BLE Controller IRQ pin ISR and the BLE Controller ClkReq pin ISR.
 - C. For a cold boot use, generate new Bluetooth MAC address from chip ID to initialize Bluetooth MAC address, otherwise preserve existing MAC address.
 - D. Prepare the HCI (Host Controller Interface) driver for operation.
2. Initialize the main ExactLE stack.
 - A. Set up timers for the WSF scheduler with `WsfOsInit()` and `WsfTimerInit()`.
 - B. Initialize a buffer pool for WSF dynamic memory needs.
 - C. Initialize the WSF security service by calling `SecInit()`, `SecAesInit()`, `SecCmacInit()` and `SecEccInit()`.
 - D. Set up a callback function for the various layers of the ExactLE stack with `WsfOsSetNextHandler()` and `xxxHandlerInit()`, to serve as a WSF event handler for HCI-related events (HCI, DM, L2CAP, ATT, SMP, application framework and application).
3. Start the application profile, e.g., `xxxStart()`.
 - A. Register for stack callbacks.
 - B. Register for app framework callbacks.
 - C. Initialize the attribute server database.
 - D. Set service changed CCCD index.
 - E. Reset the device.
4. Enter an infinite loop to dispatch WSF OS message.

1. An internal reference clock, EXTREFCLK, uses the BLE XTAL frequency divided by 4 (for a 12 MHz clock) which is made available as a source clock for several on-chip modules.

26.3 General Use

26.3.1 Pairing

The Apollo510B can work as a BLE subordinate and/or a manager. When it works as a manager, it can call *DmSecPairReq ()* to initialize the pairing. When it works as a subordinate, it can also call *DmSecSlaveReq ()* to request the manager to initiate the pairing. And then the stack SMP state machine will process the rest of procedure automatically. The user also needs to configure the SMP parameters such as I/O capability, authentication and other security properties based on hardware capability and application requirements.

26.3.2 Send data over BLE

The user/application code uses the ATT layer API to send the data through BLE.

- When the device works as an ATT server, it can use

```
void AttsHandleValueInd (dmConnId_t connId, uint16_t handle, uint16_t valueLen, uint8_t *pValue)
```

to send an attribute protocol handle value indication to the peer device. It can also use

```
void AttsHandleValueNtf (dmConnId_t connId, uint16_t handle, uint16_t valueLen, uint8_t *pValue)
```

to send an attribute protocol handle value notification to the peer device.

- When the device works as an ATT client, it can use

```
void attcSetupReq (attcCcb_t *pCcb, attcApiMsg_t *pMsg)
```

to set up and send an attribute client request to the peer device.

26.3.3 Receive data through BLE

When the BLE receives data from the peer device, the BLE Controller will notify the host with the BLE_IRQ interrupt (GPIO), and then the host callback function will read the data from the BLE Controller. The user application code can use ATT layer API to receive the data through BLE.

1. Call *void AttHandlerInit (wsfHandlerId_t handlerId)* to initialize the ATT handler. See Initialization section.
2. Call the register function *void AttRegister (attCback_t cback)* to register the application callback function.
3. Once the ATT layer receives the data from the lower layer, it will call *attCb.cback ()*, which is the exact registered application callback function, to retrieve the data for the user/application code.

26.3.4 Disconnection

A BLE disconnection can be initiated either remotely or by the local device due to a procedure error. The BLE stack will post a DM_CONN_CLOSE_IND event with the caused error code to the application layer when the connection is terminated. Table 145 lists some general disconnection reasons. Reference the Bluetooth Core Specification for more error codes.

Table 145: BLE Disconnection Reasons

Bluetooth Disconnection Value	Disconnection Reason
0x08	The Connection Timeout error code indicates that the link supervision timeout has expired which could be caused by a continuous synchronization failure for a given connection. Low received signal strength or noisy interference may result in such failure.
0x13	The Remote User Terminated Connection error code indicates that the user on the remote device terminated the connection.
0x16	The Connection Terminated By Local Host error code indicates that the local device terminated the connection.
0x22	The LL Response Timeout error code indicates that an LL transaction failed to respond within the LL response timeout.
0x28	The Instant Passed error code indicates that an LL PDU that includes an instant cannot be performed because the instant when this would have occurred has passed. This error could occur in Connection Update, PHY Update or Channel Map Update procedures which include an instant and the instant is not larger than the current connection event counter when the stack starts to schedule the update procedure.

26.3.5 How to Use `uart_ble_bridge` to Perform DTM Testing

For Apollo510B SoC:

- Download the `uart_ble_bridge.bin` to the EVB.
- Open a serial link terminal such as SSCOM. Set the baud rate to 115200 and configure other UART parameters according to your hardware setup.
- Send the HCI command data in hex format (each data should be separated by a space) in the serial link terminal.

For Apollo330 Plus Series SoCs and Apollo510D Lite SoC:

- Use `uart_hci_bridge.bin` for DTM testing.

26.3.6 BLE Controller Firmware Update

For Apollo510B SoC:

- Put the latest `ble_fw_image_em9305.h` file into the folder `SDK\third_party\cordio\ble-host\sources\hci\ambiq\em9305`.
- Navigate to the SDK folder `\boards\apollo510b_evb\examples\ble\ble_firmware_update` and build `ble_firmware_update.bin`.
- Download `ble_firmware_update.bin` to the EVB. Wait a few seconds to finish updating the BLE Controller firmware.

For Apollo330 Plus Series SoCs and Apollo510 Lite Series SoCs:

- Some versions of these SoCs support both BLE as well as BT Classic.
- Verify the latest BLE/BTDM version is located in the SDK folder `\tools\apollo(330P or 510L)_scripts\firmware_updates\cm4_ble_updates or cm4_btdm_updates\fw_version`.
- Navigate to the target firmware version folder and run the `"jlink-load_[BLE/BT_image_version]_ota.txt"` script file to program the device using JLINK via the J16 USB.
- Jlink might request the target device setting through a pop-up message, Filter device by "AP330MPA-CBR" or "AP510DLA-CBR"
- Wait a few seconds to finish updating the BLE Controller firmware.

26.3.7 BLE Controller NVDS Configuration

NVDS configuration is NOT used for the Apollo510B SoC; however, it is used for applicable Apollo330 Plus Series SoCs and Apollo510 Lite Series SoCs. The NVDS parameters are defined in `nvds_data` array of `hci_drv_510L_radio.c` file.

26.3.8 Useful APIs of the BLE Controller

- Read a register value from the BLE core: `am_util_ble_plf_reg_read ()`.
- Write a register value to the BLE core: `am_util_ble_plf_reg_write ()`.
- Set the TX power of BLE: `am_util_ble_tx_power_set ()`.
- Write NVDS parameters to the BLE core: `am_util_ble_nvds_set ()`.
- Get the BLE Controller's trim versions: `am_util_ble_trim_version_get ()`.

26.3.9 OTA

Using the `ble_freertos_amota` example, the host firmware and secure bootloader (SBL) can be upgraded through OTA with Ambiq OTA APP.

- For the host firmware OTA, navigate to the folder `SDK\tools\apollo5_amota\scripts` and build it. Note that you may need to use `pip3` to install necessary packages such as `pycryptodome` before you build it to generate the bin file for OTA. Load the bin into Ambiq OTA APP to run the OTA.
- For the host SBL OTA, create a folder `SDK\tools\apollo510_scripts\sbl_updates\[SBL_ap510_latest_version]` and do the following:
 - Unzip *.bin from a different folder in `SDK\tools\apollo510_scripts\sbl_updates`.
 - Navigate to the folder `SDK\tools\apollo5_amota\scripts\sbl_ota` and build it to generate the bin file for OTA.
 - Load the bin into Ambiq OTA APP to run the OTA.
 - Check the start address of SBL to see if there is a difference with that address of SBL before OTA (such as address changing from `0x8000` to `0x10000` or changing from `0x10000` to `0x8000`).

For the SBL OTA, please note that the zip package file version to unzip should match the EVB board version.

26.4 Vendor Specific HCI commands

Table 146 shows vendor-specific HCI commands for the BLE Controller on Apollo510B SoC.

Table 146: HCI Commands

HCI Command Name	Opcode	OGF	OCF	Length	Parameters	Functionality	API Function
HCI_VSC_READ_PRODUCT_INFO	0xFC01	0x3F	0x01	0	None	Read product information from the device	-
HCI_VSC_SET_DEV_PUB_ADDR	0xFC43	0x3F	0x43	6	BD Address (6 bytes)	Set device public Bluetooth address	HciVscUpdateBDAddress()
HCI_VSC_SET_SLEEP_OPTION	0xFC49	0x3F	0x49	1	Enable (1 byte)	Set sleep option (enable/disable) for BLE core	HciDrvBleSleepSet()
HCI_VSC_CRC_CALCULATE	0xFC4E	0x3F	0x4E	8	CRC data (8 bytes)	Calculate CRC checksum	-
HCI_VSC_ENTER_CM	0xFC4F	0x3F	0x4F	0	None	Enter configuration mode	-
HCI_VSC_EXIT_CM	0xFC50	0x3F	0x50	0	None	Exit configuration mode	-
HCI_VSC_START_TRANSMIT_TEST	0xFCC1	0x3F	0xC1	6	Mode, Channel, DataLen, Payload, PHY, Power	Start transmitter test	HciVscTransmitTestStart()
HCI_VSC_END_TRANSMIT_TEST	0xFCC2	0x3F	0xC2	0	None	End transmitter test	HciVscTransmitTestEnd()
HCI_VSC_SET_TX_POWER_LEVEL	0xFCC4	0x3F	0xC4	1	Power Level (0-15)	Set transmit power level (4-15 recommended)	HciVscSetRfPowerLevelEx()
HCI_VSC_ENTER_SLEEP	0xFCF0	0x3F	0xF0	0	None	Trigger Apollo510 and EM9305 to enter sleep (Ambiq)	-
HCI_VSC_CALCULATE_TRIM_VALUE	0xFCF1	0x3F	0xF1	1	Trim data (1 byte)	Obtain high frequency crystal trim value (Ambiq)	-
HCI_VSC_READ_AT_ADDRESS	0xFD01	0x3F	0x101	5	Address info (5 bytes)	Read data from specific memory address	-
HCI_VSC_WRITE_AT_ADDRESS	0xFD03	0x3F	0x103	134	Address + Data (134 bytes)	Write data to specific memory address	-
HCI_VSC_NVM_ERASE_NVM_MAIN	0xFD06	0x3F	0x106	0	None	Erase NVM main area	-
HCI_VSC_NVM_ERASE_PAGE	0xFD07	0x3F	0x107	2	Page info (2 bytes)	Erase NVM page	-
HCI_VSC_SET_LOCAL_SUP_FEAT	0xFFF2	0x3F	0x3F2	8	Feature bits (8 bytes)	Set local supported Link Layer features	HciVscUpdateLinklayerFeature()
HCI_OPCODE_LE_VS_ENABLE_READ_FEAT_ON_CONN	0xFFF3	0x3F	0x3F3	N/A	TBD	Enable reading features on connection	-

Command Format Structure

Type (1 byte)...Opcode Low (1 byte)...Opcode High (1 byte)...Length (1 byte)...Parameters (0-134 bytes)

Constants

- Header Length: 4 bytes
- OGF (Opcode Group Field): 0x3F (Vendor Specific)
- Total Command Size: Header (4) + Parameter Length

API Function Reference

Table 147: API Function Reference

Function	Command Used	Parameters	Return
HciVscUpdateBDAddress()	0xFC43	None (uses global)	void
HciDrvBleSleepSet()	0xFC49	bool enable	void
HciVscSetRfPowerLevelEx()	0xFCC4	int8_t txPowerlevel	bool_t
HciVscTransmitTestStart()	0xFCC1	hciVscStartTransTestCmd_t *	void
HciVscTransmitTestEnd()	0xFCC2	None	void
HciVscUpdateLinklayerFeature()	0xFFFF2	None (uses global)	void
HciVscDisablePowerCtrl()	0xFFFF2	None	void
HciVscCarrierWaveMode()	TODO	uint8_t txchannel	void

Parameter Details by Command

- HCI_VSC_START_TRANSMIT_TEST (0xFCC1) - 6 bytes

Table 148: HCI_VSC_START_TRANSMIT_TEST Parameters

Byte	Parameter	Description
0	tx_test_mode	Test mode selection
1	tx_channel	TX channel (0-39)
2	test_data_len	Test data length
3	pkt_payl	Packet payload type
4	phy	PHY type (1M, 2M, Coded)
5	tx_power_level	TX power level (0-15)

- HCI_VSC_SET_DEV_PUB_ADDR (0xFC43) - 6 bytes

Table 149: HCI_VSC_SET_DEV_PUB_ADDR Parameters

Byte	Parameter	Description
0-5	BD_ADDR	Bluetooth Device Address (LSB first)

- HCI_VSC_SET_LOCAL_SUP_FEAT (0xFFFF2) - 8 bytes

Table 150: HCI_VSC_SET_LOCAL_SUP_FEAT Parameters

Byte	Parameter	Description
0-7	Feature Bits	Link Layer feature bits (see BT 5.4 Vol 6, Part B, pg 2844)

27. Ordering Information

Table 151: Ordering Information

Device Name	Commercial Temp Range (-20°C to 70°C)	Industrial Temp Range (-40°C to 85°C)	Package Type	GPIOs	NVM (MRAM) ¹	SRAM (SSRAM and TCM)	Connectivity	Package ² Size (mm)
Apollo510 SoC	AP510NFA-CBR	AP510NFA-IBR	BGA	183	4 MB	3.75 MB	No Connectivity	6.6 x 6.6 x 0.75(max) 225-pin BGA
Apollo510 SoC	AP510NFA-CCR	-	WLCSP	144	4 MB	3.75 MB	No Connectivity	4.9 x 4.7 x 0.513(max) 182-pin WLCSP
Apollo510B SoC	AP510BFA-CBR	-	BGA	96	4 MB	3.75 MB	BLE 5.4	5.6 x 5.6 x 0.8(max) 153-pin WFBGA
Apollo510 Lite	AP510NLA-CCR	-	CSP	68	2 MB	2 MB	No Connectivity	4.047 x 3.948 x 0.510(max) 110-pin
Apollo510 Lite	AP510NLA-CBR	-	BGA	120	2 MB	2 MB	No Connectivity	5.3 x 5.3 x 0.8(max) 169-pin
Apollo510B Lite	-	AP510BLA-ICR	CSP	68	2 MB	2 MB	BLE 5.4	4.047 x 3.948 x 0.510(max) 110-pin
Apollo510D Lite	AP510DLA-CCR	-	CSP	68	2 MB	2 MB	BLE 5.4, BT Classic	4.047 x 3.948 x 0.510(max) 110-pin
Apollo510D Lite	AP510DLA-CBR	AP510DLA-IBR	BGA	120	2 MB	2 MB	BLE 5.4, BT Classic	5.3 x 5.3 x 0.8(max) 169-pin
Apollo330 Plus	AP330NPA-CCR	-	CSP	68	2 MB	2 MB	No Connectivity	4.047 x 3.948 x 0.510 (max) 110-pin
Apollo330 Plus	-	AP330NPA-IBR	BGA	120	2 MB	2 MB	No Connectivity	5.3 x 5.3 x 0.8 (max) 169-pin
Apollo330B Plus	AP330BPA-CCR	AP330BPA-ICR	CSP	68	2 MB	2 MB	BLE 5.4	4.047 x 3.948 x 0.510 (max) 110-pin
Apollo330M Plus	AP330MPA-CCR	-	CSP	68	2 MB	2 MB	BLE 5.4, Matter/ Thread	4.047 x 3.948 x 0.510 (max) 110-pin
Apollo330M Plus	-	AP330MPA-IBR	BGA	120	2 MB	2 MB	BLE 5.4, Matter/ Thread	5.3 x 5.3 x 0.8 (max) 169-pin

1. Factory-installed Cortex-M4 firmware reduces the amount of available NVM which varies for each series derivative. The amount of NVM used will be made available in a future datasheet release.

2. Packing: Tape and Reel



©2026 Ambiq, Inc. All rights reserved.

Ambiq Micro, Inc.

6500 River Place Boulevard, Building 7,

Suite 200, Austin, TX 78730-1156

www.ambiq.com/

sales@ambiq.com

<https://support.ambiq.com>

+1 (512) 879-2850

RM-A5-1p1

Version 1.1

Jul 2026