PROGRAMMER'S GUIDE

Applicable to Apollo4 Family SoCs: Apollo4 SoC, Apollo4 Blue SoC, Apollo4 Plus SoC, Apollo4 Blue Plus SoC

Ultra-low Power Apollo SoC Family

Doc. ID: PG-A4-7p0

Doc. Revision: 7.0, October 2022

Legal Information and Disclaimers

AMBIQ MICRO INTENDS FOR THE CONTENT CONTAINED IN THE DOCUMENT TO BE ACCURATE AND RELIABLE. THIS CONTENT MAY, HOWEVER, CONTAIN TECHNICAL INACCURACIES, TYPOGRAPHICAL ERRORS OR OTHER MISTAKES. AMBIQ MICRO MAY MAKE CORRECTIONS OR OTHER CHANGES TO THIS CONTENT AT ANY TIME. AMBIQ MICRO AND ITS SUPPLIERS RESERVE THE RIGHT TO MAKE CORRECTIONS, MODIFICATIONS, ENHANCEMENTS, IMPROVEMENTS AND OTHER CHANGES TO ITS PRODUCTS, PROGRAMS AND SERVICES AT ANY TIME OR TO DISCONTINUE ANY PRODUCTS, PROGRAMS, OR SERVICES WITHOUT NOTICE.

THE CONTENT IN THIS DOCUMENT IS PROVIDED "AS IS". AMBIQ MICRO AND ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THIS CONTENT FOR ANY PURPOSE AND DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS CONTENT, INCLUDING BUT NOT LIMITED TO, ALL IMPLIED WARRANTIES AND CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHT.

AMBIQ MICRO DOES NOT WARRANT OR REPRESENT THAT ANY LICENSE, EITHER EXPRESS OR IMPLIED, IS GRANTED UNDER ANY PATENT RIGHT, COPYRIGHT, MASK WORK RIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT OF AMBIQ MICRO COVERING OR RELATING TO THIS CONTENT OR ANY COMBINATION, MACHINE, OR PROCESS TO WHICH THIS CONTENT RELATE OR WITH WHICH THIS CONTENT MAY BE USED.

USE OF THE INFORMATION IN THIS DOCUMENT MAY REQUIRE A LICENSE FROM A THIRD PARTY UNDER THE PATENTS OR OTHER INTELLECTUAL PROPERTY OF THAT THIRD PARTY, OR A LICENSE FROM AMBIQ MICRO UNDER THE PATENTS OR OTHER INTELLECTUAL PROPERTY OF AMBIQ MICRO.

INFORMATION IN THIS DOCUMENT IS PROVIDED SOLELY TO ENABLE SYSTEM AND SOFTWARE IMPLEMENTERS TO USE AMBIQ MICRO PRODUCTS. THERE ARE NO EXPRESS OR IMPLIED COPYRIGHT LICENSES GRANTED HEREUNDER TO DESIGN OR FABRICATE ANY INTEGRATED CIRCUITS OR INTEGRATED CIRCUITS BASED ON THE INFORMATION IN THIS DOCUMENT. AMBIQ MICRO RESERVES THE RIGHT TO MAKE CHANGES WITHOUT FURTHER NOTICE TO ANY PRODUCTS HEREIN. AMBIQ MICRO MAKES NO WARRANTY, REPRESENTATION OR GUARANTEE REGARDING THE SUITABILITY OF ITS PRODUCTS FOR ANY PARTICULAR PURPOSE, NOR DOES AMBIQ MICRO ASSUME ANY LIABILITY ARISING OUT OF THE APPLICATION OR USE OF ANY PRODUCT OR CIRCUIT, AND SPECIFICALLY DISCLAIMS ANY AND ALL LIABILITY, INCLUDING WITHOUT LIMITATION CONSEQUENTIAL OR INCIDENTAL DAMAGES. “TYPICAL” PARAMETERS WHICH MAY BE PROVIDED IN AMBIQ MICRO DATA SHEETS AND/OR SPECIFICATIONS CAN AND DO VARY IN DIFFERENT APPLICATIONS AND ACTUAL PERFORMANCE MAY VARY OVER TIME. ALL OPERATING PARAMETERS, INCLUDING “TYPICALS” MUST BE VALIDATED FOR EACH CUSTOMER APPLICATION BY CUSTOMER’S TECHNICAL EXPERTS. AMBIQ MICRO DOES NOT CONVEY ANY LICENSE UNDER NEITHER ITS PATENT RIGHTS NOR THE RIGHTS OF OTHERS. AMBIQ MICRO PRODUCTS ARE NOT DESIGNED, INTENDED, OR AUTHORIZED FOR USE AS COMPONENTS IN SYSTEMS INTENDED FOR SURGICAL IMPLANT INTO THE BODY, OR OTHER APPLICATIONS INTENDED TO SUPPORT OR SUSTAIN LIFE, OR FOR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE AMBIQ MICRO PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR. SHOULD BUYER PURCHASE OR USE AMBIQ MICRO PRODUCTS FOR ANY SUCH UNINTENDED OR UNAUTHORIZED APPLICATION, BUYER SHALL INDEMNIFY AND HOLD AMBIQ MICRO AND ITS OFFICERS, EMPLOYEES, SUBSIDIARIES, AFFILIATES, AND DISTRIBUTORS HARMLESS AGAINST ALL CLAIMS, COSTS, DAMAGES, AND EXPENSES, AND REASONABLE ATTORNEY FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PERSONAL INJURY OR DEATH ASSOCIATED WITH SUCH UNINTENDED OR UNAUTHORIZED USE, EVEN IF SUCH CLAIM ALLEGES THAT AMBIQ MICRO WAS NEGLIGENT REGARDING THE DESIGN OR MANUFACTURE OF THE PART.

Table of Content

List of Figures

List of Tables

1. Introduction

The purpose of this Programmer's Guide is to provide developers using an Apollo4 family SoC with additional information to that found in the datasheets, specifically as it relates to programming the device for intended operation. The current SoCs that make up the Apollo4 family include the following:

- Apollo4
- Apollo4 Blue
- Apollo4 Plus
- Apollo4 Blue Plus

The Programmer's Guide and the relevant datasheet(s) are intended to supplement each other and the user should have ready access to one when referencing the other. This Programmer's Guide also refers to the AmbiqSuite SDK extensively, as the SDK is the programmers' primary source of software to use and control Apollo4 operation properly.

Unless stated otherwise, references in this guide to "Apollo4 SoC" pertain to all members of the Apollo4 family.

2. Document Revision History

Table 1: Document Revision History

Revision	Date	Description
1.0	Apr 2020	Document initial release
2.0	Apr 2020	Added chapters: WDT, UART, DC, BLE Controller (appendix) Updated chapters: - GPIO: Updated Pin Muxing Table (CSP package pins) - PDM: Added "Supported Data Formats" section
3.0	Jul 2020	Initial version for silicon revision B Added chapters: USB Updated chapters: - GPIO: Pin Mapping, Pad Function Color Code, and Special Pad Types tables updated. - DC: Corrected pin names in "Serial Formats" section. - BLE Controller Appendix: Minor edits to transaction diagrams for clarity.
3.1	Oct 2020	- CLKGEN: Removed digital calibration procedures for XT/LFRC and auto-calibration of LFRC, which have been deprecated. - RTC: Removed references to deprecated features (12-hour clock mode, read error status (CKERR). - GPIO: Removed unavailable external clocks in Pin Mapping Tables. - DC: Display Formats content added from datasheet
4.0	Oct 2020	- I2C/SPI Slave Module: Direct Area Functions section updated to include Address Pointer Wrap Mode.
4.1	Dec 2020	- CLKGEN: HFRC Auto-adjustment section updated. - TIMER: References to deprecated CONTINUOUS mode removed. - GPIO: Added replicated I2S0/I2S1 functions for sharing I2S ports on different devices in Rev B Pin Mapping Table. IO Master 0 4-wire SPI Connection section updated. - IOM: SPI Configuration section updated.
5.0	Jun 2021	- SoC Architecture: "DAXI Considerations" section added. - CLKGEN: "Clock Enablement for Modules with a Special Mux" section added. - TIMER: Updated modes of operation. - IOSLAVE: Added detail in "Wakeup Using the I2C/SPI Slave" section.
6.0	Sep 2021	- SoC Architecture: Removed unsupported entries in "ARM Cortex-M4 Vector Table for Apollo4 SoC" and "SoC Interrupt Assignments" - TIMER: Removed references to STIMER selection as a timer's trigger.

Table 1: Document Revision History

Revision	Date	Description
7.0	Oct 2022	All: - Apollo4 and Apollo4 Plus Programmer's Guides have been consolidated to include information/support for Apollo4 derivatives Apollo4, Apollo4 Blue, Apollo4 Plus, Apollo4 Blue Plus. SoC Architecture: - Added note about mode transitioning requirements described in ERR076. - Updated "ARM Cortex-M4 Vector Table for Apollo4 SoC" to remove Secure Fault ISR and exception numbers above 99. - Updated "DAXI Buffers and Flush/Invalidate Operations" section. TIMER: - Updated Edge mode operation in the "Timer Modes" table. - Corrected pattern lengths of SINGLEPATTERN and REPEATPATTERN. - Corrected "Terminating a Repeat Operation" section". GPIO: - Added note specifying that the DPI-2 interface is not supported. - "Module-specific Pad Configuration" section updated to cover all Apollo4 devices. GPADC: - Updated "Clock Source and Divider" section and specified that only 24 MHz ADC clock is supported and noted that a minimum of 37 sampling/tracking cycles should be used. - Added note specifying that a delay is required after CNVCMP ISR is asserted and before reading the FIFO. MSPI: - Added section "Clocking and Other Limitations" - On Apollo4 / Apollo4 Blue the maximum clock rate for MSPI2 for quad and octal data widths has been updated to 24MHz SDR and 12MHz DDR. - Added note that MSPI2_4 cannot be used as the clock line. USB: - Added note specifying limitations for High Speed mode. SDIO: - Added. Display Controller: - "Video Timing Generator" section renamed and updated. I2S: - Added. UART: - Clock selection updated to include 48 MHz.

3. SoC Architecture

3.1 Interrupts

Within the SoC, multiple peripherals can generate interrupts. In some cases, a single peripheral may be able to generate multiple different interrupts. Each interrupt signal generated by a peripheral is connected back to the M4 core in two places. First, the interrupts are connected to the Nested Vectored Interrupt Controller, NVIC, in the core. This connection provides the standard changes to program flow associated with interrupt processing. Additionally, they are connected to the WIC outside of the core, allowing the interrupt sources to wake the M4 core when it is in a deep sleep (SRPG) mode.

The SoC supports the M4 NMI as well as the normal interrupt types. For details on the Interrupt model of the M4, please see the "**Cortex-M4 Devices Generic User Guide,**" document number DUI0553A.

Below is the M4 Vector Table for the Apollo4 family. Note that not all modules listed in the table are on all family derivatives.

Table 2: ARM Cortex-M4 Vector Table for the Apollo4 Family

Exception Number	IRQ	Offset	Vector	Description
99	83	0x18C	IRQ83	CPU (cache/DAXI)
98	82	0x168	IRQ82	Timer 15
97	81	0x184	IRQ81	Timer 14
96	80	0x180	IRQ80	Timer 13
95	79	0x17C	IRQ79	Timer 12
94	78	0x178	IRQ78	Timer 11
93	77	0x174	IRQ77	Timer 10
92	76	0x170	IRQ76	Timer 9
91	75	0x16C	IRQ75	Timer 8
90	74	0x168	IRQ74	Timer 7
89	73	0x164	IRQ73	Timer 6
88	72	0x160	IRQ72	Timer 5
87	71	0x15C	IRQ71	Timer 4
86	70	0x158	IRQ70	Timer 3
85	69	0x154	IRQ69	Timer 2
84	68	0x150	IRQ68	Timer 1
83	67	0x14C	IRQ67	Timer 0
82	66	0x148	IRQ66	Reserved
81	65	0x144	IRQ65	Reserved
80	64	0x140	IRQ64	Reserved
72-79	56-63	0x120-0x13C	IRQ56-IRQ63	GPIO
71	55	0x11C	IRQ55	Reserved
70	54	0x118	IRQ54	Reserved
69	53	0x114	IRQ53	Reserved
68	52	0x110	IRQ52	Reserved
67	51	0x10C	IRQ51	PDM3
66	50	0x108	IRQ50	PDM2

Table 2: ARM Cortex-M4 Vector Table for the Apollo4 Family

Exception Number	IRQ	Offset	Vector	Description
65	49	0x104	IRQ49	PDM1
64	48	0x100	IRQ48	PDM0
63	47	0xFC	IRQ47	Reserved
62	46	0xF8	IRQ46	Reserved
61	45	0xF4	IRQ45	I2S1
60	44	0xF0	IRQ44	I2S0
59	43	0xEC	IRQ43	Reserved
58	42	0xE8	IRQ42	LP-ADC
57	41	0xE4	IRQ41	Reserved
56	40	0xE0	IRQ40	Stimer Capture/Overflow
48-55	32-39	0xC0-0xDC	IRQ32-IRQ39	Stimer Compare[0:7]
47	31	0xBC	IRQ31	Reserved
46	30	0xB8	IRQ30	DSI
45	29	0xB4	IRQ29	Display
44	28	0xB0	IRQ28	Graphics
43	27	0xAC	IRQ27	USB
42	26	0xA8	IRQ26	SDIO
41	25	0xA4	IRQ25	Reserved
40	24	0xA0	IRQ24	CRYPTO Secure
39	23	0x9C	IRQ23	Clock Control
38	22	0x98	IRQ22	MSPI2
37	21	0x94	IRQ21	MSPI1
36	20	0x90	IRQ20	MSPI0
35	19	0x8C	IRQ19	GP-ADC
34	18	0x88	IRQ18	UART3
33	17	0x84	1RQ17	UART2
32	16	0x80	IRQ16	UART1
31	15	0x7C	IRQ15	UART0
30	14	0x78	IRQ14	Counter/Timers (combined - also see IRQ 67-82)
29	13	0x74	IRQ13	I2C/SPI Master7
28	12	0x70	IRQ12	I2C/SPI Master6
27	11	0x6C	IRQ11	I2C/SPI Master5
26	10	0x68	IRQ10	I2C/SPI Master4
25	9	0x64	IRQ9	I2C/SPI Master3
24	8	0x60	IRQ8	I2C/SPI Master2
23	7	0x5C	IRQ7	I2C/SPI Master1
22	6	0x58	IRQ6	I2C/SPI Master0
21	5	0x54	IRQ5	I2C/SPI Slave Register Access
20	4	0x50	IRQ4	I2C/SPI Slave
19	3	0x4C	IRQ3	Voltage Comparator

Table 2: ARM Cortex-M4 Vector Table for the Apollo4 Family

Exception Number	IRQ	Offset	Vector	Description
18	2	0x48	IRQ2	RTC
17	1	0x44	IRQ1	Watchdog Timer
16	0	0x40	IRQ0	Brownout Detection
15	-1	0x3C	Systick_S	
14	-2	0x38	PendSV_S	
13	-	0x34	Reserved	
12	-4	0x30	DebugMonitor	
11	-5	0x2C	SVCall_S	
10	-	0x28	Reserved	
9	-	0x24		
8	-	0x20		
7	-	0x1C		
6	-10	0x18	UsageFault_S	Usage Fault
5	-11	0x14	BusFault_S	Bus Fault
4	-12	0x10	MemoryManage_S	Memory Management Fault
3	-13	0xC	HardFault_S	Hard Fault
2	-14	0x8	NMI_S	Unused
1	-	0x4	Reset	
		0x0	Initial SP	

NOTE

MSPI1 is not available on the Apollo4 Blue SoC or on the KBR package of the Apollo4 Blue Plus SoC.

Hardware interrupts are assigned in the SoC to the M4 NVIC as shown below.

Table 3: SoC Interrupt Assignments

IRQ	Peripheral/Description
NMI	Unused
IRQ0	Brownout Detection
IRQ1	Watchdog Timer
IRQ2	RTC
IRQ3	Voltage Comparator
IRQ4	I^2C / SPI Slave
IRQ5	I^2C / SPI Slave Register Access
IRQ6-IRQ13	I^2C / SPI Master0-7
IRQ14	Counter/Timers
IRQ15-IRQ18	UART0-UART3
IRQ19	GP-ADC
IRQ20-IRQ22	MSPI0-MSPI2
IRQ23	Clock Control
IRQ24	CRYPTO Secure
IRQ25	Reserved
IRQ26	SDIO
IRQ27	USB
IRQ28	Graphics
IRQ29	Display
IRQ30	DSI
IRQ31	Reserved
IRQ32-IRQ39	Stimer Compare[0:7]
IRQ40	Stimer Capture/Overflow
IRQ41	Reserved
IRQ42	LP-ADC
IRQ43	Reserved
IRQ44-IRQ45	I2S0-I2S1
IRQ46-IRQ47	Reserved
IRQ48-IRQ51	PDM0-PDM3

Table 3: SoC Interrupt Assignments

IRQ	Peripheral/Description
IRQ52	Reserved
IRQ53	Reserved
IRQ54	Reserved
IRQ55	Reserved
IRQ56-IRQ63	GPIO
IRQ64	Reserved
IRQ65	Reserved
IRQ66	Reserved
IRQ67-IRQ82	Timer0-Timer15
IRQ83	CPU (cache/DAXI)

3.2 Memory Map

The high-level SoC-specific memory map is as shown in the following tables.

Table 4: High-level Apollo4 / Apollo4 Blue Memory Map

Address	Name	Executable[a]	Description
0x00000000 – 0x001FFFFF	MRAM	Y	Internal Non-volatile MRAM memory
0x00200000 - 0x07FFFFFF	Reserved	X	No device at this address range
0x08000000 – 0x08000FFF	Boot Loader ROM	Y	Execute Only Boot Loader and MRAM Helper Functions.
0x08001000 – 0x0FFFFFFF	Reserved	X	No device at this address range
0x10000000 – 0x101D7FFF	SRAM	Y/N	SRAM Memory (execution depends on which memory is being referenced)
0x101D8000 – 0x13FFFFFF	Reserved	X	Reserved
0x14000000 – 0x1FFFFFFF	External NVM	Y	External Memory
0x20000000 – 0x3FFFFFFF	Reserved	X	Reserved
0x40000000 – 0x4FFFFFFF	Peripheral	N	Peripheral devices
0x50000000 – 0xDFFFFFFF	Reserved	X	No device at this address range
0xE0000000 – 0xE00FFFFF	PPB	N	NVIC, System timers, System Control Block
0xE0100000 – 0xFFFFFFFF	Reserved	X	No device at this address range

a. "Y" = Yes, "N" = No, "X" = Does not apply

Table 5: High-level Apollo4 Plus / Apollo4 Blue Plus Memory Map

Address	Name	Executable[a]	Description
0x00000000 – 0x001FFFFF	MRAM	Y	Internal Non-volatile MRAM memory
0x00200000 - 0x07FFFFFF	Reserved	X	No device at this address range
0x08000000 – 0x08000FFF	Boot Loader ROM	Y	Execute Only Boot Loader and MRAM Helper Functions.
0x08001000 – 0x0FFFFFFF	Reserved	X	No device at this address range
0x10000000 – 0x102BFFFF	SRAM	Y/N	SRAM Memory (execution depends on which memory is being referenced)
0x102C0000 – 0x13FFFFFF	Reserved	X	Reserved
0x14000000 – 0x1FFFFFFF	External NVM	Y	External Memory
0x20000000 – 0x3FFFFFFF	Reserved	X	Reserved
0x40000000 – 0x4FFFFFFF	Peripheral	N	Peripheral devices
0x50000000 – 0xDFFFFFFF	Reserved	X	No device at this address range
0xE0000000 – 0xE00FFFFF	PPB	N	NVIC, System timers, System Control Block
0xE0100000 – 0xFFFFFFFF	Reserved	X	No device at this address range

a. "Y" = Yes, "N" = No, "X" = Does not apply

A more detailed view of the device-specific memory map, which breaks out the various memory types and locations, is as shown in the following tables.

Table 6: Detailed Apollo4 / Apollo4 Blue Memory Map

Address	Name	Executable[a]	Cacheable[a]	Description
0x00000000 – 0x001FFFFF	MRAM	Y	I & D	Internal Non-volatile MRAM Memory
0x00200000 - 0x07FFFFFF	Reserved	X	X	No device at this address range
0x08000000 – 0x08000FFF	Boot Loader ROM	Y	N	Execute Only Boot Loader and MRAM Helper Functions.
0x08001000 – 0x0FFFFFFF	Reserved	X	X	No device at this address range
0x10000000 – 0x1005FFFF	CPU SRAM (TCM)	Y	N	Low-power / Low Latency SRAM (TCM)
0x10060000 – 0x1015FFFF	System SRAM	Y	I only	Shared System SRAM (SSRAM)
0x10160000 – 0x101D7FFF	Extended SRAM	N	I only	Extended Memory
0x101D8000 – 0x13FFFFFF	Reserved	X	X	Reserved
0x14000000 – 0x17FFFFFF	MSPI0	N	I only	Memory Mapped MSPI 0 Aperture
0x18000000 – 0x1BFFFFFF	MSPI1	N	I only	Memory Mapped MSPI 1 Aperture
0x1C000000 – 0x1FFFFFFF	MSPI2	N	I only	Memory Mapped MSPI 2 Aperture
0x20000000 – 0x3FFFFFFF	Reserved	X	X	Reserved
0x40000000 – 0x4FFFFFFF	Peripheral	N	N	Peripheral devices
0x50000000 – 0xDFFFFFFF	Reserved	X	X	Reserved
0xE0000000 – 0xE00FFFFF	PPB	N	N	NVIC, System timers, System Control Block
0xE0100000 – 0xFFFFFFFF	Reserved	X	X	No device at this address range

a. "Y" = Yes, "N" = No, "X" = Does not apply

Table 7: Detailed Apollo4 Plus / Apollo4 Blue Plus Memory Map

Address	Name	Executable	Cacheable	Daxiable	Description
0x00000000 – 0x001FFFFF	MRAM	Y	I & D	X	Internal Non-volatile MRAM Memory
0x00200000 - 0x03FFFFFF	Reserved	X	X	X	No device at this address range
0x04000000 – 0x07FFFFFF	External NVM	Y	I & D	D only	External NVM XIP
0x08000000 – 0x08000FFF	Boot Loader ROM	Y	N	X	Execute-only Boot Loader and MRAM Helper Functions.
0x08001000 – 0x0FFFFFFF	Reserved	X	X	X	No device at this address range
0x10000000 – 0x1005FFFF	CPU SRAM (TCM)	Y	N	X	Low-power / Low Latency SRAM (TCM) [384 kB]
0x10060000 – 0x1015FFFF	System SRAM	Y	I only	D only	Shared System SRAM (SSRAM0) [1 MB]
0x10160000 – 0x101BFFFF	Extended Memory	Y	I only	D only	Extended SRAM [384 kB]
0x101C0000 – 0x102BFFFF	Extended Memory	Y	I only	D only	Shared System SRAM (SSRAM1) [1 MB]
0x102C0000 – 0x13FFFFFF	Reserved	X	X	X	Reserved
0x14000000 – 0x17FFFFFF	MSPI0	Y	I only	D only	Memory Mapped MSPI 0 Aperture [64 MB]

Address	Name	Executable	Cacheable	Daxiable	Description
0x18000000 – 0x1BFFFFFF	MSPI1	Y	I only	D only	Memory Mapped MSPI 1 Aperture [64 MB]
0x1C000000 – 0x1FFFFFFF	MSPI2	Y	I only	D only	Memory Mapped MSPI 2 Aperture [64 MB]
0x20000000 – 0x3FFFFFFF	Reserved	X	X	X	Reserved
0x40000000 – 0x4FFFFFFF	Peripheral	N	N	N	Peripheral devices
0x50000000 – 0x80FFFFFF	Reserved	X	X	X	No device at this address range
0x81000000 - 0x81001FFF	Reserved	Y	N	N	Reserved
0x81002000– 0xDFFFFFFF	Reserved	X	X	X	No device at this address range
0xE0000000 – 0xE00FFFFF	PPB	N	N	N	NVIC, System timers, System Control Block
0xE0100000 – 0xFFFFFFFF	Reserved	X	X	X	No device at this address range

NOTE

MSPI1 is not available on the Apollo4 Blue SoC or on the KBR package of the Apollo4 Blue Plus SoC.

Peripheral devices within the memory map are allocated on 4 kB boundaries, allowing each device up to 1024 32-bit control and status registers. Peripherals will 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, will always accept any write data sent to their registers without attempting to return an ERROR response. Specifically, a write to a read-only register would just become a don't-care write.

Table 8 shows the address mapping for the peripheral devices of the Base Platform.

Table 8: SoC Peripheral Device Memory Map

Address	Device
0x40000000 – 0x400003FF	Reset/BoD Control
0x40000400 – 0x40003FFF	Reserved
0x40004000 – 0x400041FF	Clock Generator
0x40004200 – 0x400047FF	Reserved
0x40004800 – 0x40004BFF	RTC
0x40004C00 – 0x40007FFF	Reserved
0x40008000 – 0x400083FF	Timers
0x40008400 – 0x4000BFFF	Reserved
0x4000C000 – 0x4000C3FF	Voltage Comparator
0x4000C400 – 0x4000FFFF	Reserved
0x40010000 – 0x400107FF	GPIO Control
0x40010800 – 0x40010FFF	Fast GPIO Control
0x40011000 – 0x400113FF	Reserved
0x40011400 – 0x40013FFF	Reserved

Table 8: SoC Peripheral Device Memory Map

Address	Device
0x40014000 – 0x400143FF	NVM Cache Control
0x40014400 – 0x4001BFFF	Reserved
0x4001C000 – 0x4001C3FF	UART0
0x4001C400 – 0x4001CFFF	Reserved
0x4001D000 – 0x4001D3FF	UART1
0x4001D400 – 0x4001DFFF	Reserved
0x4001E000 – 0x4001E3FF	UART2
0x4001E400 – 0x4001EFFF	Reserved
0x4001F000 – 0x4001F3FF	UART3
0x4001F400 – 0x4001FFFF	Reserved
0x40020000 – 0x400203FF	Miscellaneous Control
0x40020400 – 0x40020FFF	Reserved
0x40021000 – 0x400213FF	Power Control
0x40021400 – 0x40023FFF	Reserved
0x40024000 – 0x400243FF	Watchdog Timer
0x40024400 – 0x4002FFFF	Reserved
0x40030000 - 0x400303FF	Security
0x40030400 - 0x40033FFF	Reserved
0x40034000 - 0x400343FF	I2C / SPI Slave
0x40034400 - 0x40037FFF	Reserved
0x40038000 - 0x400383FF	GP-ADC
0x40038400 - 0x4003FFFF	Reserved
0x40040000 - 0x400403FF	Miscellaneous Control - NonCV
0x40040400 - 0x4004FFFF	Reserved
0x40050000 - 0x40050FFF	I2C / SPI Master0
0x40051000 - 0x40051FFF	I2C / SPI Master1
0x40052000 - 0x40052FFF	I2C / SPI Master2
0x40053000 - 0x40053FFF	I2C / SPI Master3
0x40054000 - 0x40054FFF	I2C / SPI Master4
0x40055000 - 0x40055FFF	I2C / SPI Master5
0x40056000 - 0x40056FFF	I2C / SPI Master6
0x40057000 - 0x40057FFF	I2C / SPI Master7
0x40058000 - 0x4005FFFF	Reserved
0x40060000 - 0x400603FF	MSPI Master0
0x40060400 - 0x40060FFF	Reserved
0x40061000 - 0x400613FF	MSPI Master1
0x40061400 - 0x40061FFF	Reserved

Table 8: SoC Peripheral Device Memory Map

Address	Device
0x40062000 - 0x400623FF	MSPI Master2
0x40062400 - 0x4006FFFF	Reserved
0x40070000 - 0x4007FFFF	SDIO
0x40080000 - 0x40087FFF	Debug Subsystem
0x40088000 - 0x400AFFFF	Reserved
0x40090000 - 0x4009FFFF	Graphics Subsystem
0x400A0000 - 0x400A7FFF	Display Controller
0x400A8000 - 0x400AFFFF	Display PHY
0x400B0000 - 0x400B3FFF	USB
0x400B4000 - 0x400B7FFF	USB PHY
0x400B8000 - 0x400BFFFF	Reserved
0x400C0000 – 0x400C3FFF	Crypto
0x400C4000 – 0x40200FFF	Reserved
0x40201000 – 0x402013FF	PDM0
0x40201400 – 0x40201FFF	Reserved
0x40202000 – 0x402023FF	PDM1
0x40202400 – 0x40202FFF	Reserved
0x40203000 – 0x402033FF	PDM2
0x40203400 – 0x40203FFF	Reserved
0x40204000 – 0x402043FF	PDM3
0x40204400 – 0x40207FFF	Reserved
0x40208000 – 0x402083FF	I2S0
0x40208400 – 0x40208FFF	Reserved
0x40209000 – 0x402093FF	I2S1
0x40209400 – 0x4020FFFF	Reserved
0x40210000 – 0x4024103FF	Audio ADC
0x40210400 – 0x41FFFFFF	Reserved
0x42000000 – 0x4200FFFF	NVM OTP
0x42010000 – 0x4FFFFFFF	Reserved

3.2.1 Memory Access Restriction

There is a restriction when a DMA write crosses the boundary between SRAM and Extended RAM. The previously shown Detailed SoC-specific Memory Map in Table 7 lists the memory ranges for Shared System SRAM (SSRAM0) from 0x10060000 to 0x1015FFFF and Extended SRAM starting at 0x10160000. When doing a single, continuous DMA write across the SSRAM0 / Extended SRAM boundary at 0x10160000, the DMA start address must be no closer than 32 bytes before the boundary (0x1015FFE0).

WARNING

Failure to adhere to the above boundary-crossing access restriction may result in an SoC hang.

3.3 Power Management Programming

The transition between power modes is largely directed by software and the workload/task requirements.

3.3.1 CPU

Transitioning to different CPU power modes is initiated through a couple methods depending on the software intent and performance/power/latency requirements. The different mode transitions are described in Table 9.

Table 9: CPU Power Mode Transitions

Mode	Deep Sleep	Sleep	Active - LP	Active - HP
Sleep	X	X	WAKE+LP	WAKE+HP
Active - LP	WFI (DS)	WFI (S)	X	HP
Active - HP	WFI (DS)[a]	WFI (S)	LP	X

a. Transitioning from Active-HP mode to Deep Sleep mode requires an intermediate transition to Active-LP mode.

WFI (DS): Issue WFI instruction Deep Sleep

WFI (S): Issue WFI instruction Sleep

LP: Write to PWRCTRL_MCUPERFREQ_MCUPERFREQ field indicating "LP". Check PWRCTRL_MCUPERFREQ_MCUPERFSTATUS to check if mode switch completed and if performance mode is available. Continue execution.

HP: Write to PWRCTRL_MCUPERFREQ_MCUPERFREQ field indicating "HP". Check PWRCTRL_MCUPERFREQ_MCUPERFSTATUS to check if mode switch completed and if performance mode is available. Continue execution.

NOTE

In the Apollo4/Apollo4 Blue Errata Lists, ERR076 addresses an issue that the Power Controller in the Apollo4 and Apollo4 Blue does not allow adequate time for voltages to settle before re-enabling clocks when switching between high performance and low power modes. The AmbiqSuite SDK uses a TIMER interrupt (TIMER13) configured as the highest priority (0) interrupt to prevent unintentional break out by other interrupts. **In order for this to work reliably, it is required that all other interrupts in the system be set at a lower priority, reserving the highest priority interrupt exclusively for this workaround.**

3.3.2 I/O

The I/O power modes are determined by how the I/O controller is configured. For example:

- OFF: Controller power domain is OFF, "device enable" bit not set in power controller. This transition can also be made by the I/O PMU if auto-poweroff is supported.
- Sleep: Controller is "device enabled" but the controller interface has not been enabled/activated (AKA IDLE).
- Active: Controller is "active", meaning it is enabled and configured for active transactions.

3.3.3 Power Control

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 also supports event notification to indicate peripheral power transition completion. Event notification is the preferred power-optimized method in lieu of status polling.

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.

3.4 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.

3.5 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.

3.6 Memory Controller Programming

The Cortex-M4 supports the ARMv7 ISA. Details on the instruction set and overall programming model can be referenced in the "ARM Cortex-M4 Devices Generic User Guide" and the "ARM Cortex-M4 Processor Technical Reference Manual".

The following subsections describe the set of CPU registers to interface with and control memory. The CPU registers are CM-4 Complex registers which set up and control the use of cache, TCM and DAXI (CM-4's local AHB-D interface to the primary AXI crossbar for accesses outside of the NVM and local TCM interfaces).

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

3.6.1 DAXI Considerations

3.6.1.1 DAXI Controls

The DAXI operates fairly autonomously with little programming intervention after the initial setup. The only control provided is the ability for the CPU to flush and invalidate the DAXI's line buffers via operations on the system bus to the APB interface. Mode bits allow for fine-tuning the DAXI settings.

A couple registers are available for configuring for, and controlling, the DAXI operation.

CPU_DAXICTRL Register: This register provides controls to flush and invalidate buffers in the DAXI interface.

DAXIFLUSHWRITE:

- Writing a 1 to this write-only single-bit field forces a flush of all buffers in the WRITE or MODIFIED state, which is useful for making modified data visible to the rest of the system.

DAXIINVALIDATE:

- Writing a 1 to this write-only single-bit field invalidates all buffers in the SHARED state, which is useful for making visible data modified by other parts of the system.

NOTE
Setting these bits in the register just initiates the respective operation. The operation is not atomic and blocking, and could take a variable amount of time depending on the content of the DAXI buffers at that time. The SDK HAL implementation incorporates additional steps to ensure the flush/invalidate has finished.

CPU_DAXICFG Register: This register allows configuration of the aging counter, buffers and flush level.

AGINGCOUNTER:

- An 8-bit R/W field which specifies the time in CPU clock cycles that DAXI buffers may remain unused before being flushed. The setting indicates the number of clock cycles between increments of the aging counter. For Apollo4, buffers will generally be flushed in 1-2 AGINGCOUNTER time steps. The Apollo4 Plus allows for up to 65,535 steps per age with a default value of 2048.

BUFFERENABLE:

- A 2-bit (Apollo4) or 4-bit (Apollo4 Plus) R/W field that enables a number of DAXI buffers from 1 to 4 (Apollo4) or from 1 to 32 (Apollo4 Plus). The default is single-buffer mode (Apollo4) or 4 buffers (Apollo4 Plus).

FLUSHLEVEL:

- For Apollo4, a single-bit R/W field that sets the number of buffers to try to keep free. When set to 0 (default), the DAXI will attempt to maintain two free buffers at all times. When set to 1, the DAXI will attempt to keep only a single buffer free.
- For Apollo4 Plus, a single-bit R/W field that sets the number of buffers to try to keep free. When set to 0 (default), flushing out dirty buffers occurs if 3 or more buffers are enabled and less than two are free, or if 2 buffers are enabled and none are free. When set to 1, flushing out dirty buffers occurs if 3 or more buffers are enabled and less than three are free, or if 2 buffers are enabled and less then two are free.

NOTE

Due to an existing limitation on Apollo4, the DAXI must be used with either 2 buffers or 1 buffer enabled and, in either case, the flush level must be set to 0.

3.6.1.2 DAXI Buffers and Flush/Invalidate Operations

A few considerations need to be followed in software when working with AXI memory (SSRAM, Extended RAM and XIP RAM) in order to maintain RAM coherency or when going into a sleep mode. RAM coherency must be maintained when other masters access or change data that is in the DAXI buffer. Other masters include DMA to/from any peripheral such as IOM (SPI/I2C), MSPI, ADC, I2S, PDM, SDIO, CRC Engine, Crypto, GPU and Display Controller.

The DAXI subsystem requires that a few guidelines must be followed in order to ensure software coherency and proper functionality of the DAXI buffers. In user applications the following actions must be taken.

1. For DMA Reads from a shared access memory written by the CPU:
   - Flush the DAXI buffer before starting the DMA transfer:

     am_hal_daxi_control(AM_HAL_DAXI_CONTROL_FLUSH, 0);

   - Practical integration point in software: When the CPU writes/modifies data in a buffer which will be read by another master, the flush can be performed at the point where the CPU completes its writing before signaling the other master read to start.

2. For DMA Writes to a shared access memory which needs to be consumed by the CPU:
   - Invalidate the DAXI buffer after the DMA completes, before the CPU reads it:

     am_hal_daxi_control(AM_HAL_DAXI_CONTROL_INVALIDATE, 0);

   - Practical integration point in software: When the DMA write transfer completes, a DAXI buffer invalidate must be performed before signaling to the CPU that the data is available.

3. Alignment restrictions:
   - All DMA buffers which are to be accessed by non-CPU masters need to be padded to 128 bits.
   - The buffers should be aligned such that the CPU and an external agent are never in control of the same 128-bit segment of the buffer at any point in time.

4. Entering normal sleep or deepsleep:
   - A DAXI flush must be performed prior to entering sleep.

When using the AmbiqSuite HAL implementation to manage DAXI, in addition to the above requirements, user applications need to make a call as part of initialization.

For Apollo4 and Apollo4 Blue, the call is:

am_hal_daxi_control(AM_HAL_DAXI_CONTROL_AXIMEM, pBuf);

For Apollo4 Plus and Apollo4 Blue Plus, the call is:

am_hal_daxi_control(AM_HAL_DAXI_CONTROL_ENABLE, pBuf);

The pBuf parameter in both cases is a 16-byte aligned memory of at least 64 bytes in SSRAM/Extended SRAM.

It is expected that if any of the AXI mapped memory is in use, this memory is enabled. For example, if pBuf is in SSRAM memory, there is never a case where MSPI XIP or Extended SRAM is being used but SSRAM is turned off. This function provides scratch pad space in the user application for proper execution of DAXI flushing, and keeps the HAL independent of linker configurations and customer memory layout.

Transitions to sleep, as well as all of the internal transition points of buffer hand-off, are taken care of inside of the HAL. For any instance not handled internal to the HAL, such as ADC, AUDADC, PDM, Crypto, and Display Controller (if Display Controller is using buffers composed by the CPU directly, bypassing the GPU), the application must follow the above DAXI management guidelines.

For ADC, AUDADC and PDM, before reading the received DMA data the application must directly handle the interrupts and call:

am_hal_daxi_control(AM_HAL_DAXI_CONTROL_INVALIDATE, 0);

For Crypto, when operating on non-TCM buffers, the application needs to call:

am_hal_daxi_control(AM_HAL_DAXI_CONTROL_FLUSH, 0);

or

am_hal_daxi_control(AM_HAL_DAXI_CONTROL_INVALIDATE, 0);

For Display Controller, when operating on non-TCM buffers and in scenarios where using buffers that are composed by the CPU directly (and not through the GPU), applications must call flush:

am_hal_daxi_control(AM_HAL_DAXI_CONTROL_FLUSH, 0);

As described in section "4.3.1 DAXI" of the Apollo4 Datasheet version 0.9.0, these flush and invalidate operations help ensure software coherency in the end application. In the case of the flush, it ensures that any data meant for non-CPU agents has made it to the actual destination before signaling the agent to retrieve it. Similarly, the invalidate operation ensures that no stale data is read when fetching a buffer from an external agent, by invalidating any DAXI buffer contents and re-fetching the data from memory.

The following considerations must be followed in software when using XIP memory:

- Ensure all XIP transactions have completed before disabling XIP, and before disabling or powering off the corresponding MSPI.

- The HAL implementation calls DAXI flush before any of these operations. However, this just ensures that the transaction has been flushed out of DAXI.
- To ensure any outstanding Writes to XIPMM have completed, it must be ensured that they have made it to the external device.
- Additional delays need to be provisioned between last usage of XIP Write, and disabling of XIP/MSPI.

4. Clock Generation (CLKGEN)

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

4.1 HFRC Auto-adjustment

In some applications it is important that the High Frequency RC Oscillator (HFRC) frequency be more accurate than the ±2% variation typically seen, particularly in cases where the temperature may vary widely. A good 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.

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.

4.2 Generating 100 Hz

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) to produce a maximum jitter of less than 4 ms.

4.3 Clock Enablement for Modules with a Special Mux

NOTE
This section and sub-sections apply specifically to Apollo4 and Apollo4 Blue.

Module clock selection should be performed with system-wide coordination of enabling only clock sources that are needed while keeping unneeded clock sources powered off. Note that not all modules are pinned out on all Apollo4 packages.

The I2Sn, IOMn, ADC and AUDADC modules use a special clock source selection mux which requires a specific selection procedure to avoid causing clock glitches that may affect clocking/operation of any module using the same clock. These modules may include up to three possible clock sources - HFRC, HFRC2 and high-speed crystal oscillator (XTHS). However, only the usage (enabling/disabling) of HFRC2 and XTHS clock sources is required to be monitored and controlled, as HFRC is enabled always and does not require or support software enabling/disabling.

Throughout this section, reference to the FSEL clock source means the source clock selection when FSEL = 0. The FSEL = 0 clock source is the clock that samples FSEL when changing the clock for a module. Note that for the AUDADC module, the FSEL = 0 clock source selection, as indicated in the CFG_CLKSEL field, is clock off. In this case, the HFRC clock is used as the FSEL sampling clock.

Clock source selection for the IOMn (and ADC modules for Apollo4) is included in this discussion because each IOM has the special mux and requires a 40 µs delay after switching the clock source frequency selection FSEL. However, the only clock source option for these modules is HFRC which, as mentioned above, does not require any special software action to enable or disable this clock source.

The PDMn modules do not have this special mux, but clock selection for these modules is discussed here since they can be clocked with the HFRC2 or XTHS clock source and therefore should be considered when changing clock sources for any of these modules.

For the affected modules, when switching from the currently selected clock source to a new clock source, up to 3 clocks must be running during the clock switch – the currently selected clock source, the new clock source and the FSEL/CLKSEL sample clock source, which may or may not be the default selection, and may or may not be either the currently selected clock source or the new clock source. The FSEL sample clock source for a module is always the clock source of FSEL = 0 in the clock select register, and that clock source must be enabled to sample the changing FSEL clock selection. If the FSEL = 0 clock source is not enabled, then switching the clock source cannot occur.

The final step in the procedure for selecting a (new) clock for a module is to power down the replaced clock source and the FSEL sample clock source if:

1. Either is HFRC2 or XTHS
2. Either is *not* the new clock
3. Neither is used elsewhere

To make the determination as to whether any clock source is not used elsewhere and can be powered off, it is necessary to either maintain a Clock Usage Table to keep track of clocks currently being used by all modules, or to individually check the clock source setting of each module which can use either HFRC2 or XTHS as its clock source to see if in fact either is being used as a clock source.

When switching between clocks, it is important to allow a settling delay after enabling and after switching to the clock source, and before disabling the replaced clock source to ensure the new clock has settled and is clocking properly.

Settings required to use the XTHS clock include the adjustment of several trim settings, whether using the internal high-speed crystal oscillator or an external high-speed oscillator. This sequence of settings, as

well as a number of field settings in the MCUCTRL_XTALHSCTRL register, follow the procedure found in the AmbiqSuite SDK's am_hal_mcuctrl_control () function of the am_hal_mcuctrl.c file. In particular, the EXTCLK32M_KICK_START case of that function is used to enable the 32 MHz circuit/clock, and the EXTCLK32M_DISABLE case is used to disable this clock.

NOTE
The XTHS clock requires that the VDDAUDA supply is provided. Consult the Electrical Characteristics in the Apollo4 Datasheet for supply specifications.

4.3.1 Clock Switching Procedure

Selecting or changing the clock source for a module requires enabling up to three clock sources - the currently selected clock source, the FSEL sample clock source and the new clock source.

1. Enable each of the three possible clock sources as follows.

HFRC clock:

- No software-triggered actions required.

HFRC2 clock:

- Set CLKGEN_MISC_FRCHFRC2 = 1 if not already set.

XTHS clock:

- Set MCUCTRL_XTALHSTRIMS register to 0x0FFF8D2C, resulting in the individual field settings:
  - XTALHSCAP2TRIM = 44
  - XTALHSCAPTRIM = 4
  - XTALHSDRIVETRIM = 3
  - XTALHSDRIVERSTRENGTH = 0
  - XTALHSIBIASCOMP2TRIM = 3
  - XTALHSIBIASCOMPTRIM = 15
  - XTALHSIBIASTRIM = 127
  - XTALHSRSTRIM = 0
  - XTALHSSPARE = 0
- Set MCUCTRL_XTALHSCTRL register to 0x0A2, resulting in the individual field settings:
  - XTALHSCOMPPDNB = 1
  - XTALHSPDNPNIMPROVE = 1
  - XTALHSPADOUTEN = 1
- Set MCUCTRL_XTALHSCTRL_XTALHSPDNB = 1
- Set MCUCTRL_XTALHSCTRL_XTALHSINJECTIONENABLE = 1
- Set MCUCTRL_XTALHSCTRL_XTALHSIBSTENABLE = 1
- Delay 5 µs.
- Set MCUCTRL_XTALHSCTRL_XTALHSINJECTIONENABLE = 0
- Delay AM_HAL_MCUCTRL_CRYSTAL_IBST_DURATION µs.
- Set MCUCTRL_XTALHSTRIMS register to 0x17118D1C, resulting in the individual field settings:
  - XTALHSCAP2TRIM = 28
  - XTALHSCAPTRIM = 4
  - XTALHSDRIVETRIM = 3
  - XTALHSDRIVERSTRENGTH = 0
  - XTALHSIBIASCOMP2TRIM = 3
  - XTALHSIBIASCOMPTRIM = 8

  - XTALHSIBIASTRIM = 56
  - XTALHSRSTRIM = 1
  - XTALHSSPARE = 0
- If using an external oscillator, set the MCUCTRL_XTALHSCTRL register fields:
  - XTALHSIBSTENABLE = 0
  - XTALHSEXTERNALCLOCK = 1
  - XTALHSPDNB = 0
- Else if using internal crystal oscillator, set the MCUCTRL_XTALHSCTRL register fields:
  - XTALHSPDNPNIMPROVE = 0
  - XTALHSIBSTENABLE = 1

2. Delay(s) for clock power-up when required. If the FSEL sample clock source or the new clock source (if different) requires power-up, delay the following times after enable/power-up.

HFRC2 clock:

- Delay 10 µs.

XTHS using internal crystal oscillator:

- Delay at least 520 µs.

XTHS using an external oscillator:

- Delay the specified start-up time required for the external oscillator plus some additional margin (> 10 µs).

3. Set the module's clock select field (FSEL) to the desired new clock/divider selection.

4. Delay 40 µs to ensure new clock is running.

5. Disable unused clock sources if not used anywhere else by referencing the Clock Usage Table.

HFRC clock:

- No software-triggered actions required.

HFRC2 clock:

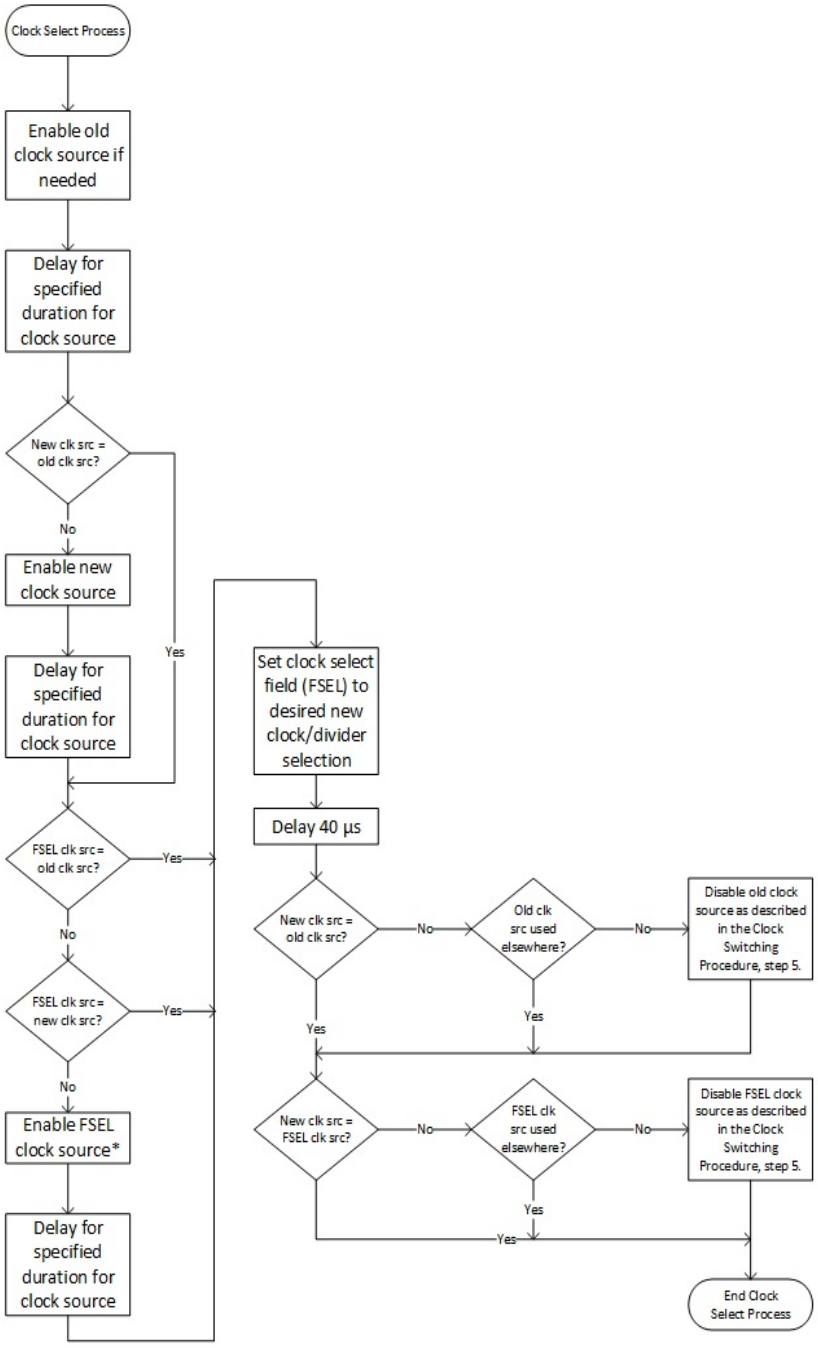
- Set CLKGEN_MISC_FRCHFRC2= 0.

XTHS clock:

- Set MCUCTRL_XTALHSTRIMS register to 0x0311F12C, resulting in the individual field settings:
  - XTALHSCAP2TRIM = 44
  - XTALHSCAPTRIM = 4
  - XTALHSDRIVETRIM = 0
  - XTALHSDRIVERSTRENGTH = 7
  - XTALHSIBIASCOMP2TRIM = 3
  - XTALHSIBIASCOMPTRIM = 8
  - XTALHSIBIASTRIM = 24
  - XTALHSRSTRIM = 0
  - XTALHSSPARE = 0
- Set MCUCTRL_XTALHSCTRL register to 0x0A2, resulting in the individual field settings:
  - XTALHSEXTERNALCLOCK = 0
  - XTALHSPADOUTEN = 1
  - XTALHSPDNPNIMPROVE = 1
  - XTALHSIBSTENABLE = 0
  - XTALHSCOMPPDNB = 1
  - XTALHSPDNB = 0

4.3.2 Clock Switching Process Flowchart

Figure 1 shows a flowchart of the clock source selection process, including clock source enabling, necessary delays and unused clock source disabling.



*References to FSEL clock source means the source clock selection when FSEL = 0. Note that for the AUDADC module, the FSEL = 0 source clock (actually called CFG_CLKSEL) is clock off. In this case the module uses the HFRC clock as the FSEL sampling clock, and for HFRC there is no FSEL clock enabling required.

Figure 1. Clock Source Selection Flowchart

4.3.3 Clock Usage Table

The following table may be used as a clock selection reference and/or a software lookup table to keep track of HFRC2 and XTHS source clock usage by module so that a used clock type is not inadvertently disabled and any unused clock type can be disabled to save power. Where there are multiple instances of a module, each module has its own clock source setting. They are represented in the table on a single line for brevity. In addition to these modules which have the option to use one or more of the clock sources, the table also includes pads which support CLKOUT function selection for which these clock sources can be selected.

Table 10: Clock Usage Table

Module/Pad	Selectable Clock Source[a]		
	HFRC2	HFRC	XTHS
I2S0 - I2S1	**X**^{S}	**X**	**X**
AUDADC	**X**	**X**^{S}	**X**
ADC		**X**^{S}	
IOM0 – IOM7		**X**^{S}	
PDM0 – PDM3	**X**	**X**	**X**
CLKOUT pads[b] (GPIO33, GPIO63, GPIO66, GPIO67, GPIO71, GPIO72, GPIO80 and/or GPIO81)	**X** When CLKGEN_ CLKOUT_CKSEL = 0x3A - 0x3C (HFRC2 divisors)	**X** When CLKGEN_ CLKOUT_CKSEL = 0x19 – 0x20, 0x2F – 0x33, 0x39 (HFRC divisors)	
CLKOUT_32M pad: GPIO46[c]			**X**

a. **X**^{S} indicates the clock source which samples FSEL when changing the clock for a module. Referred to as "FSEL sample clock source"; applies only to modules with the special muxes. Modules not using the special mux (but may select these clock sources) are also listed here to note where each clock source may be in use in order to prevent inadvertent power-down of a clock source in use.
b. Valid when CLKGEN_CLKOUT_CKEN = 1 and GPIO_PINCFGn_FNCSELn = CLKOUT for the pad.
c. Valid when GPIO_PINCFG46_FNCSEL46 = CLKOUT_32M, the XTHS enable settings of the Clock Switching Procedure are set.

4.3.4 Clock Selection Examples

Refer to the latest register set for register field selections.

1. Change I2S0 clock selection from HFRC_375kHz to XTHS_EXTREF_CLK.
   A. HFRC is the currently selected clock source and is presently enabled (I2S0_CLKCFG_FSEL = HFRC_375kHz (0x0e).
   B. If HFRC2, the FSEL sample clock source, requires power up, then enable HFRC2 per the **Clock Switching Procedure** and wait 10 µs.
   C. If XTHS, the new clock source, requires power up, then enable XTHS per the **Clock Switching Procedure** above and wait the specified start-up time for the external oscillator.
   D. Set I2S0_CLKCFG_FSEL = XTHS_EXTREF_CLK (0x0f) and wait 40 µs.
   E. If the HFRC2 clock source is no longer required per the **Clock Usage Table**, disable the HFRC2 clock source per the **Clock Switching Procedure**.

2. Change AUDADC clock selection from HFRC2_48MHz to XTALHS_24MHz.
   1. HFRC2 is the currently selected clock source and is presently enabled (AUDADC_CFG_CLKSEL = HFRC2_48MHz (0x3).
   2. HFRC is the FSEL sample clock source by default (even though AUDADC_CFG_CLKSEL = OFF for CLKSEL = 0x0). HFRC does not require software power up.
   3. If XTHS, the new clock source, requires power up, then enable XTHS per the **Clock Switching Procedure** and wait 800 µs for the internal crystal oscillator to settle.
   4. Set AUDADC_CFG_CLKSEL = XTALHS_24MHz (0x3) and wait 40 µs.
   5. If the HFRC2 clock source is no longer required per the **Clock Usage Table**, disable the HFRC2 clock source per the **Clock Switching Procedure**.

4.4 External 32.768 kHz Clock Input on X0

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 VDDH/2 to 3.3V 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

5. Real Time Clock (RTC)

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

5.1 Calendar Counters

The real time is held in a set of eight Calendar Counters, which hold the current 1/100th of a second (RTC_CTRLOW_CTR100), the current second (RTC_CTRLOW_CTRSEC), the minute (RTC_CTRLOW_CTRMIN), the hour (RTC_CTRLOW_CTRHR), the current day of the month (RTC_CTRUP_CTRDATE), the current day of the week (RTC_CTRUP_CTRWKDY), the current month (RTC_CTRUP_CTRMO), the current year (RTC_CTRUP_CTRYR) and the current century (RTC_CTRUP_CB), all in BCD format. In order to insure 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.

5.2 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.

5.3 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.

5.4 Century Control and Leap Year Management

The RTC_CTRUP_CB bit indicates the current century. A value of 0 indicates the 20^{th} century, and a value of 1 indicates the 19^{th} or 21^{st} century. The CB value will toggle when the Years counter rolls over from 99 to 0 if the RTC_CTRUP_CEB bit is set, and will remain constant if CEB is clear. 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.

5.5 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.

6. Counter/Timer Module (TIMER)

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

6.1 Counter/Timer Functions

The Timer Module contains sixteen 32-bit counters that can be used for a number of generic counting or waveform generation functions as described in Table 12 and the following sub-sections.

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): Counts up from zero and stops when CMP0 s reached. TMRnLMT: Has no effect in this mode. TIMERn counts up 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). 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 (TMRnLMT = 0). Timer outputs will be a pulse of one source clock period on the output when the TIMER reaches the associated CMP value. 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
DOWNCOUNT **(Apollo4 / Apollo4 Blue only)**	OUT0 pulses for 1 clock cycle when TIMERn = 0; repeats per TMRnLMT setting. OUT1 pulses for 1 clock cycle when TIMERn = CMP1 (if CMP1 < CMP0).	Repeatable Down-counter (0x6): Counts down from CMP0 to zero and stops (TMRnLMT = 1), repeat N times (TMRnLMT = 2 - 255), or repeat indefinitely (TMRnLMT = 0). Timer outputs will be a pulse of one source clock period on the output when the TIMER reaches the associated CMP value. The down-count version can be useful for irregular intervals since software can reprogram the CMP0 value after each interrupt to change the next reload value. Trigger: Timer does not start until trigger occurs (if enabled). Subsequent triggers ignored.
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 bit-shifted to form output pattern on OUT0 and OUT1, or concatenation of CMP1:CMP0 bit shifted to form 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 restart 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].	Repeated Pattern Generation (0xD): CMP0/CMP1 bit-shifted to form output pattern on OUT0, reset at TMRnLMT back to 0 and repeat pattern. Same as SINGLEPATTERN with the exception that the timer repeats the pattern after TMRnLMT+1 bits have been streamed out. Since TMRnLMT is used for the repeat pattern length, there is no option to repeat the pattern N times.
EVENTTIMER **(Apollo4 / Apollo4 Blue only)**	OUT0/OUT1: Unused	Single-run Edge Event Timer (0xE): Counts up from zero upon trigger event, stops at first clock edge. TIMERn clock = bus clock. Start counter at trigger event and stop counter at first transition of edge source (CTRLn_TMRnCLK). Trigger: Timer does not start until trigger occurs (if enabled). Subsequent triggers ignored.

NOTE

For the Apollo4 and Apollo4 Blue SoCs, please refer to ERR059 in the errata list for a comprehensive list of Counter/Timer errata limiting functionality or timer selection.

NOTE

On the Apollo4 and Apollo4 Blue SoCs, stopping the timer by de-asserting the TIMER_CTRLn_TMRnEN bit after the timer has completed a mode operation may not always retain/clear the timer count (TIMERn register). The counter should be read before disabling the timer if the count is of interest.

On the Apollo4 Plus and Apollo4 Blue Plus SoCs, 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

On the Apollo4 Plus and Apollo4 Blue Plus SoCs, for EDGE, UPCOUNT and PWM timer modes, the TMRnmINT 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

On the Apollo4 Plus and Apollo4 Blue Plus SoCs,CMP0/CMP1 values should not be changed when the TIMER is running to prevent corrupting the counter.

NOTE

On the Apollo4 Plus and Apollo4 Blue Plus SoCs, the TMRnLMTVAL register holds the instantaneous value of a counter for TMRnLMT and can be read to get either (1) the number of executed iterations of the mode in EDGE, UPCOUNT or PWM mode, or (2) the number of bits already streamed out of the TMRnLMT+1 total bits in SINGLEPATTERN OR REPEATPATTERN mode.

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

This 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 2), 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 3).
- 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.

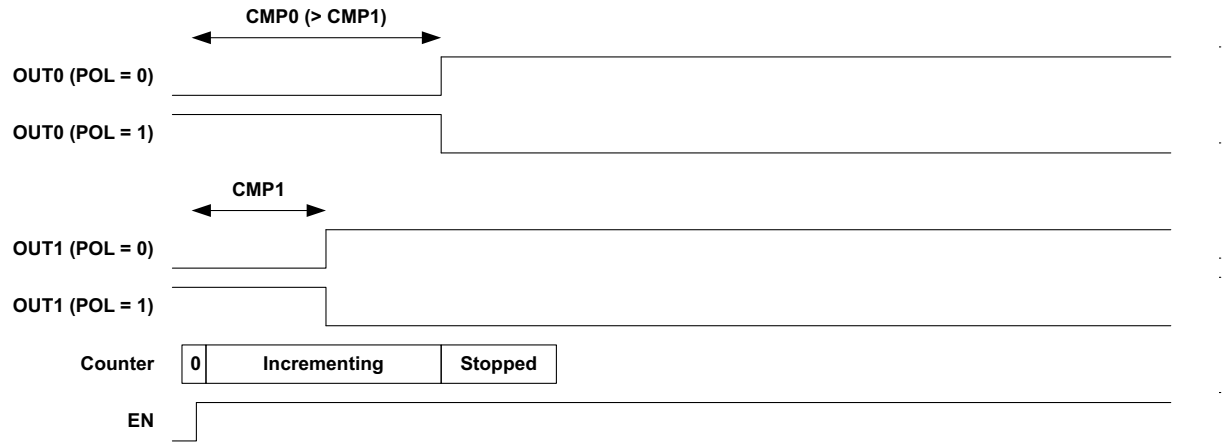


Figure 2. Timer Edge Mode *(TMRnFN* = 0x1) - CMP0 > CMP1

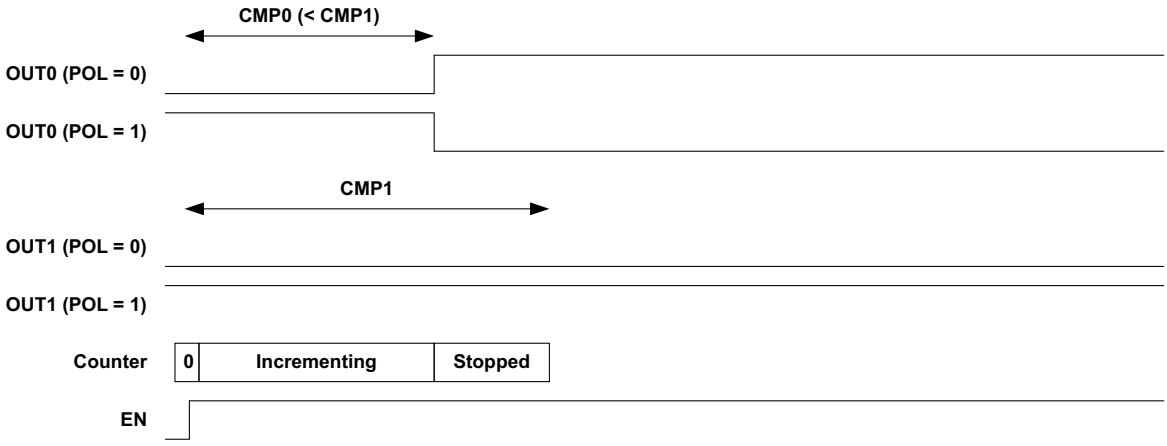


Figure 3. Timer Edge Mode *(TMRnFN* = 0x1) - CMP0 < CMP1

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

This 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 4.
- 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.

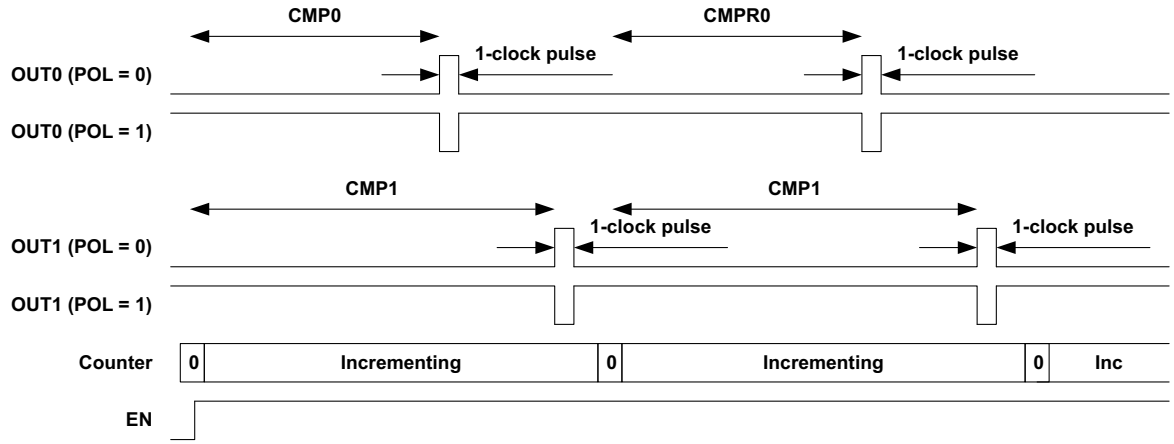


Figure 4. Timer Repeated Pulse Up-counter Compare Mode (TMRnFN = 0x2)

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

This 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 5.
- 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.

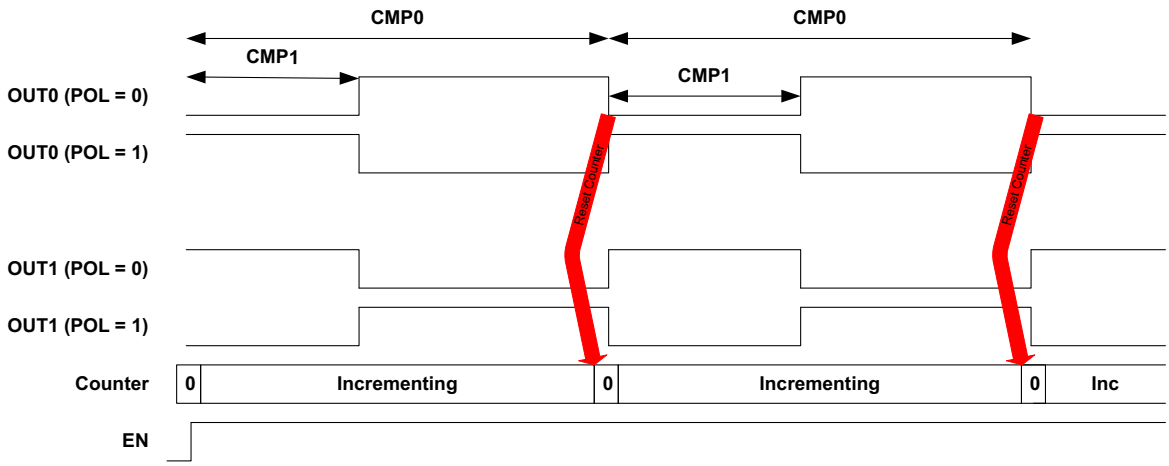


Figure 5. Timer PWM Mode (TMRnFN = 0x4)

6.1.4 Repeatable Pulse-generating Down-counter (TMRnFN = 0x6)

Supported on the Apollo4 and Apollo4 Blue SoCs only

This timer mode is selected for TIMERn by setting the CTRLn_TMRnFN field to 0x6 (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).
  - If the polarity for an output is set to NORMAL (CTRLn_TMRnPOLm = 0x0), then OUTm pulses high when TIMERn = TMRnCMPm or 0. When an output is set to INVERTED, then the generated pulses occur at the same times but are the opposite in polarity.
- TIMERn is set to the value in the TMRnCMP0 register when CTRLn_TMRnCLR is asserted after the TMRnFN field is set for this mode.
- 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 down on each clock, the source of which is selected by CTRLn_TMRnCLK field.
- When TIMERn =0, a pulse of polarity opposite of the start state of OUT0 is generated, after which OUT0 returns to its original state.
  - The pulse is of 1 clock cycle in duration.
  - TMRnCMP0 must be at least 1 so that the repeat interval is two clock cycles.
- If TMRnCMP1 is set to a value less than TMRnCMP0, then when the counter count-down hits the value in TMRnCMP1, a pulse with a polarity opposite of the start state of OUT1 is generated, after which OUT1 returns to its original state. Again, the pulse is of 1 clock cycle in duration.
- 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 until it is disabled. When set for at least one repeat (CTRLn_TMRnLMT >= 2),
- TIMERn continues counting after reaching zero by loading the value of TMRnCMP0 in TIMERn and continuing counting down from there. This results in pulses being generated on OUT0 (and OUT1 if TMRnCMP1 < TMRnCMP0), creating a stream of pulses or interrupts at a fixed interval until CTRLn_TMRnLMT is reached or TIMERn is stopped. This operation is as shown in Figure 6.
- When stopped or disabled, TIMERn will stop counting but will not be cleared. Asserting CTRLn_TMRnCLR will reset TIMERn to zero.
- The interrupt, if enabled by setting the INTEN_TMRn0INT bit, may be cleared by writing a one to the corresponding INTCLR_TMRn0INT bit.

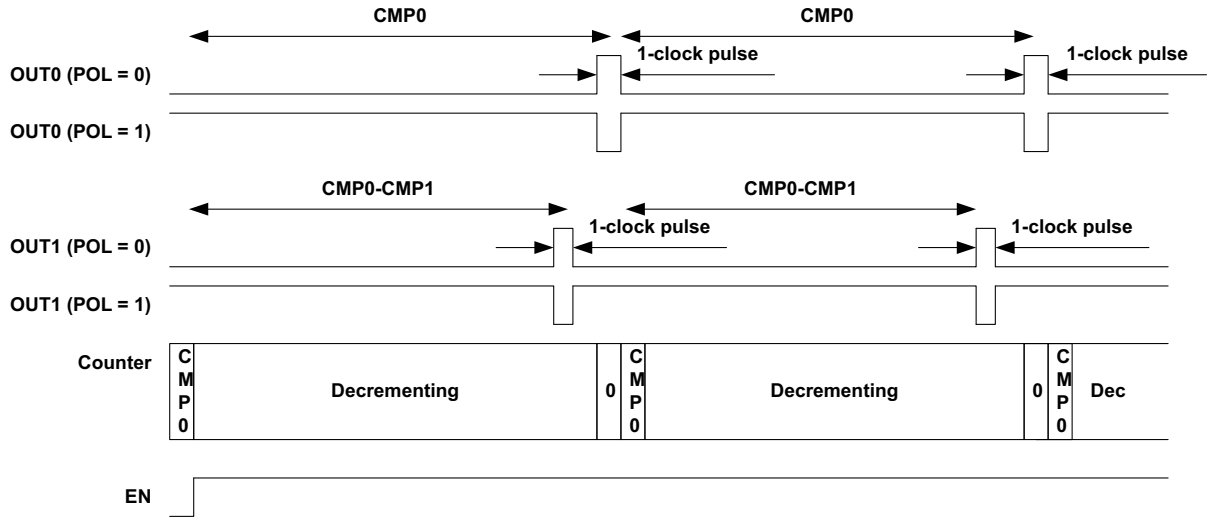


Figure 6. Timer Repeated Pulse Down-counter Compare Mode (TMRnFN = 0x6)

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

This 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 7.
- 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 8.

- 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.

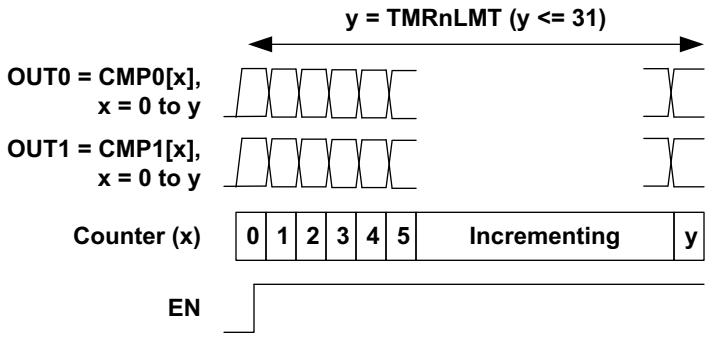


Figure 7. Timer Single 32-bit Pattern Output (TMRnFN = 0xC)

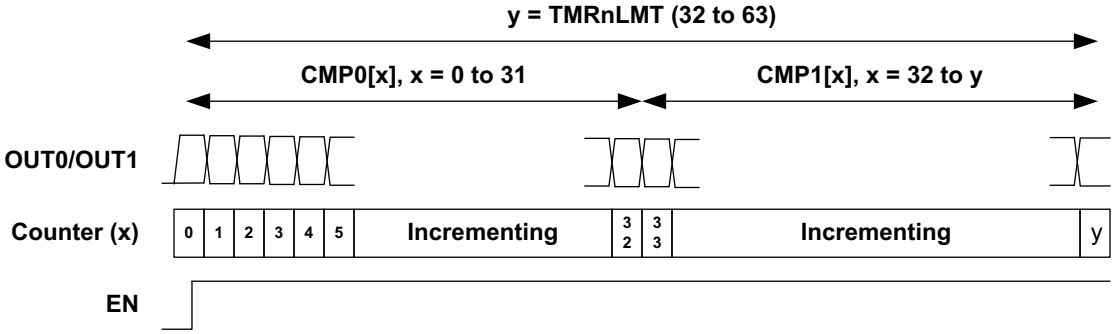


Figure 8. Timer Single 64-bit Pattern Output (TMRnFN = 0xC)

6.1.5.1 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.
  - A maximum of 32 bits (TMRnLMT+1) are indexed.

The first 25 bits for the output waveforms are as shown in Figure 9.

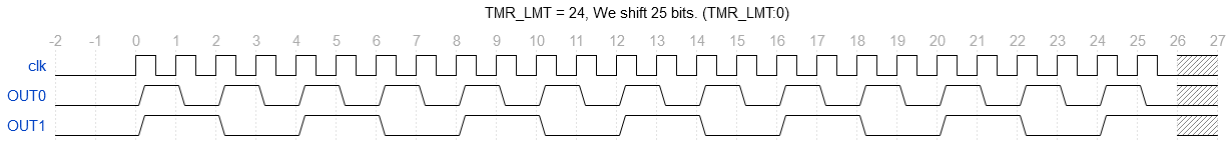


Figure 9. Single 32-bit Pattern Output

6.1.5.2 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 first 25 bits for the output waveforms are as shown in Figure 10.

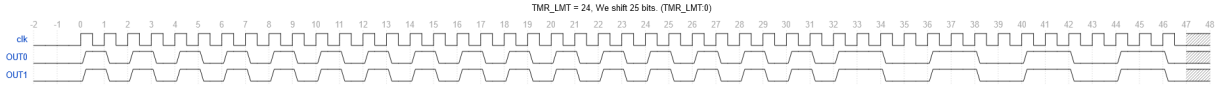


Figure 10. Single 64-bit Pattern Output

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

This 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.

6.1.7 Single Run Edge Event Timer Up-counter (TMRnFN = 0xE)

Supported on the Apollo4 and Apollo4 Blue SoCs only

This timer mode is selected for TIMERn by setting the CTRLn_TMRnFN field to 0xE (n = 0 to 15).

- TIMERn outputs OUT0 and OUT1 are not used in this mode and are not driven.
- 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 *bus* clock.

- TIMERn stops counting upon detection of an input clock edge, the source of which is as specified by the setting of the CTRLn_TMRnCLK field. See Figure 11.
- An interrupt, if enabled by setting the INTEN_TMRn0INT bit, may be cleared by writing a one to the corresponding INTCLR_TMRn0INT bit.
- After event capture and the clearing of the timer by asserting CTRLn_TMRnCLR, subsequent triggers start TIMERn counting again.

This timer mode is typically used to measure time between transitions on the same GPIO input or different inputs, as any GPIO can be used to trigger the event counter and any GPIO can be used as the timed event source (GPIO or clock).

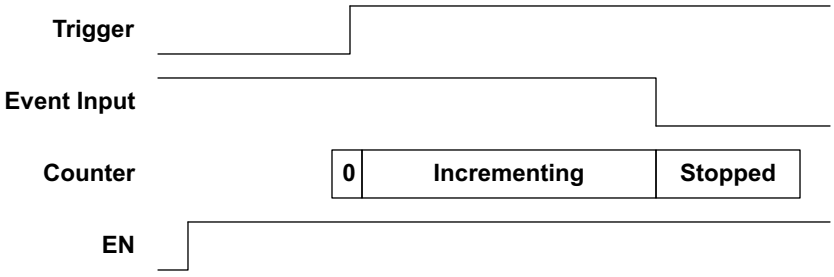


Figure 11. Timer Event Counter (TMRnFN = 0xE)

6.2 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.

NOTE

On the Apollo4 Plus and Apollo4 Blue Plus SoCs, when using a trigger in a non-repeat mode, only a single mode iteration can be triggered. After the mode iteration completes and before the next trigger occurs, the timer must be cleared (by setting and then clearing the CTRLn_TMRnCLR bit) and reconfigured properly for the next trigger to work.

NOTE

Due to an Apollo4 device erratum, there is no support for an STIMER capture/compare event to be the trigger for the start of a timer. This functionality has been deprecated on Apollo4 Plus.

6.2.1 Initiating a One-shot Operation

If the mode of a TIMER is as a one-shot (EDGE, SINGLEPATTERN and EVENTTIMER modes on Apollo4 / Apollo4 Blue; EDGE and SINGLEPATTERN on Apollo4 Plus / Apollo4 Blue Plus), 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.

6.2.2 Terminating a Repeat Operation

The Repeat Pattern mode and the repeatable modes (UPCOUNT, PWM, and DOWNCOUNT on Apollo4 / Apollo4 Blue only) 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.

6.3 Clocking Timer/Counters with Other Counter/Timer Outputs

There are cases where it is very valuable 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.

6.4 Global Timer/Counter Enable

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 CTIMER_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.

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.

6.5 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 Repeatable 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.

6.6 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 Repeatable 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.

6.7 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.

NOTE

Software should assert TMRnCLR each time before asserting TMRnEN. If not done, then stale values will remain.

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 increment on the next clock and then hold its current value. When TMRnEN is set to 1, the Counter will resume counting on the second following edge.

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.

6.8 NOSYNC Function

Each TIMERn clock is supplied directly by the clock selected in the TIMER_CTRLn_TMRnCLK field. Timers run synchronously on the bus clock based on sampling the source clock, which means the edges will have about 20ns of jitter.

6.9 Counter Functions

A TIMER operates in counter mode when the TMRnCLK field selects either an external pad input (if 0x00-0xFF) or a buck pulse input (if 0x1C-0x1F). The different clock selections provide different functions.

6.9.1 Counting External Edges

If the TMRnCLK field is 0x80-0xFF, the TIMER clock input comes from an external pad. This allows the TIMER to monitor pulses or edges on an external signal.

6.9.2 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.

6.10 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.

7. System Timer (STIMER)

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

7.1 Writing to the STIMER Compare Register on the Apollo4 Plus

STIMER logic has been updated on Apollo4 Plus 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 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 a number of STIMER clocks to update the SCMPR register.

Specifically, hardware takes four STIMER clock cycles to update the value of the SCMPRn register. Do the following calculations to determine the length of this delay:

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

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

8. Watch Dog Timer (WDT)

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

8.1 Basic Operation

The basic operation of the watchdog timers is to configure the clock, program the two 8-bit registers that are used as counter thresholds for the timer (INTVAL, RESVAL), and set the WDTEN to enable the timer. As the watchdog counter increments and reaches the interrupt value (INTVAL), the watchdog timer will issue an interrupt to the respective CPU (assuming INTEN is set). The counter will then continue counting to the RESVAL value unless the CPU writes the RESTART KEY to the RSTART register, which will reset the counter to zero and continue counting (the CPU cannot otherwise influence the counter). If the CPU fails to respond to the interrupt and reset the timer, the timer will reach the reset (RESVAL) value. At this point, the watchdog timer will issue a reset to the appropriate reset controller to reset the CPU.

8.2 Register Functions

A summary of the registers is listed below.

- CFG: The config register contains the INTVAL, RESVAL, and the CLKSEL (ARM) fields. It also contains the WDTEN to enable the timer, INTEN to enable the interrupt comparator, and RESEN to enable the reset comparator.
- RSTRT: The "restart" register allows the processor to reset its watchdog counter to zero by writing the 8-bit key. Failure to restart the counter before the RESVAL will cause a reset.
- TLOCK: The "timer lock" register allows the CPU to irrevocably enable its timer by writing the LOCK key to this register. Locking the timer will automatically enable the timer and prevent subsequent writes to the associated CFG register, therefore it is important that software ensure that it is programmed properly before locking the timer.
- COUNT: The count register returns the current counter value.

The watchdog timer also contains a standard Ambiq interrupt register block for interrupts for each of the processors. The interrupt registers have been split into separate register banks to avoid issues with contention between processors when servicing interrupts.

9. General Purpose Input/Output (GPIO)

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

9.1 General Purpose I/O (GPIO) Functions

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

For each pad, if the FNCSEL0n field is set to 0x3 the pad is connected to the corresponding GPIO signal. This section describes the configuration functions specific to GPIO pads.

9.1.1 Configuring the GPIO Functions

Each GPIO must be configured in the REG_GPIO_PINCFGn (n = GPIO number) Registers 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 REG_GPIO_PINCFGn Registers. Each output may be push-pull, open drain, disabled, or tri-stated as selected by the REG_GPIO_PINCFGn_OUTCFGn field. If the output is configured as push-pull, the pad will be driven with the corresponding bit in the REG_GPIO_WTn (n = 0-3) 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 REG_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.

9.1.2 Reading from a GPIO Pad

All GPIO inputs are readable at all times provided 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 0 to 31 are read in the REG_GPIO_RD0 Register, the current values of pads 32 to 63 are read in the REG_GPIO_RD1 Register, the current values of pads 64 to 95 are read in the REG_GPIO_RD2 Register, and the current values of pads 96 to 127[1] are read in the REG_GPIO_RD3 Register.

9.1.3 Writing to a GPIO Pad

The GPIO pad outputs are controlled by the REG_GPIO_WT0-WT3 Registers and the REG_GPIO_EN0-EN3 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 REG_GPIO_WTS0-WTS3 Register is written, the corresponding bit in WT0-WT3 will be set if the write data is 1, otherwise the WT0-WT3 bit will not be changed. If the REG_GPIO_WTC0-WTC3 Register is written, the corresponding bit in WT0-WT3 will be cleared if the write data is 1, otherwise the WT0-WT3 bit will not be changed.

If a GPIO pad is configured for tri-state output mode, the EN0-EN3 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-ENS3 or ENC0-ENC3 Registers with a 1 in the desired bit position.

1. GPIO105-127 are reserved and unavailable for use.

9.1.4 GPIO Interrupts

Each GPIO pad can be configured to generate an interrupt on a high-to-low transition or a low-to-high transition (or either), as selected by setting the REG_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 REG_GPIO_MCUN0INTnEN, REG_GPIO_MCUN0INTnSTAT, REG_GPIO_MCUN0INTnCLR and REG_GPIO_MCUN0INTnSET, where n = 0 to 3 for GPIO pads 0 to 31, 32 to 63, 64 to 95 and 96 to 127, respectively. The N0 designation in these registers indicates that these are interrupt register set 0. A duplicate set of registers, with a designation N1, is available and these registers are similarly named REG_GPIO_MCUN1INTnEN, REG_GPIO_MCUN1INTnSTAT, REG_GPIO_MCUN1INTnCLR and REG_GPIO_MCUN1INTnSET. The purpose of this dual set of registers is to enable 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.

9.2 Pad Connection Summary

Figure 12 shows the detailed implementation of each pad. Each element will be described in detail.

9.2.1 Output Selection

There is a multiplexer which selects the module signal to be driven to the output based on the REG_GPIO_PINCFGn_FNCSELn (n = GPIO number) field. This implements the multiplexing 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.

9.2.2 Output Control

The pad 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 OUTDATSEL and OUTENSEL signals which are controlled by the selection of the output type as shown in the Special Pad Types table in the GPIO section of the applicable datasheet, as set in the REG_GPIO_PINCFGn_PULLCFGn field.

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.

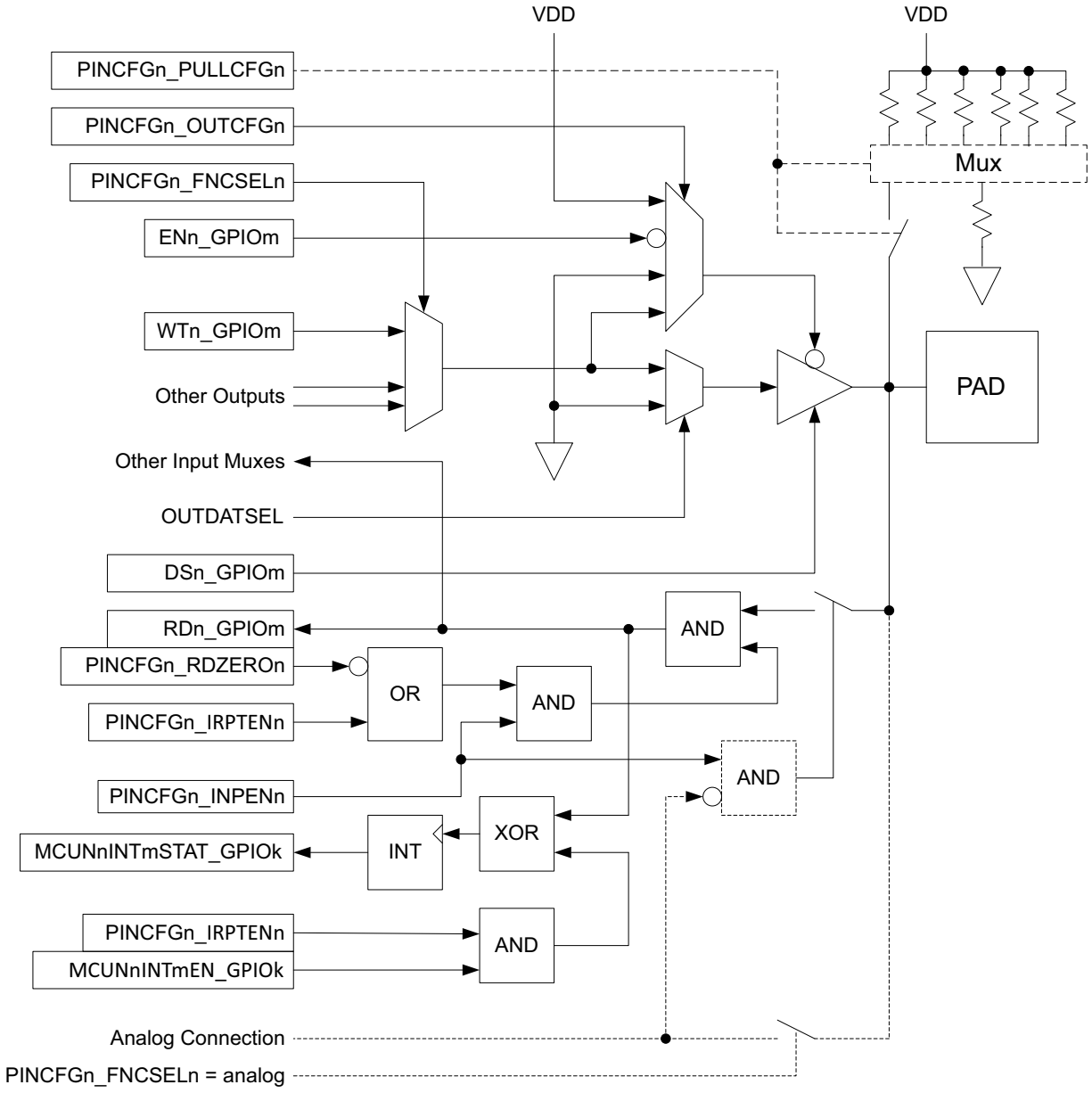


Figure 12. Pad Connection Details

OUTENSEL normally selects a ground signal to keep the pad driver enabled. If the 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 9.1 on page 57.

9.2.3 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 I^2C/SPI Slave nCE signal is driven low by the Interface Host.

9.2.4 Pull-up/Pull-down Control

If PULLCFGn is set to a pull-up or pull-down value, a pull-up/pull-down resistor is connected between the pad and VDDH/VSSH.

9.2.5 Analog Pad Configuration

Pads which may have analog connections (all pads with ADC or VCOMP functions shown in the device-specific Pin Mapping Table) include the circuitry shown with the dotted lines of Figure 12. If the 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, OUTENSEL is forced high to disable the pad output, and the input of the pad is disabled independent of the value of INPENn.

9.3 Module-specific Pad Configuration

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

9.3.1 Implementing IO Master Connections

The eight IO Master modules must be correctly connected to the appropriate pads in order to operate. Reference the device-specific Pin Mapping Table for pads which can used for the various signals for each IO Master instance: MnSCK, MnMOSI and MnMISO for SPI operation, and MnSCL and MnSDAWIRE3 for I^2C 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.

9.3.1.1 IO Master I^2C Connection

I^2C mode of IO Master 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 I^2C 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 Master 0 I^2C Configuration

Field	Value
FNCSEL5	0
FNCSEL6	0

This same setup for I^2C mode is done for each of the other seven instances of the IO Master 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 Master instance.

9.3.1.2 IO Master 4-wire SPI Connection

Four-wire SPI mode of IO Master 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 Master IO instances. This same setup for 4-wire SPI mode is done for each of the other seven instances of the IO Master as needed. Reference the device-specific Pin Mapping Table for pads which are used for the MnSCK, MnMOSI and MnMISO signals for each IO Master 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 Master 0 4-wire SPI Configuration

Field	Value
FNCSEL5	1
FNCSEL6	1
FNCSEL7	0

9.3.1.3 IO Master 0 3-wire SPI Connection

Three-wire SPI mode of IO Master 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 Master 0 3-wire SPI Configuration

Field	Value
FNCSEL5	1
FNCSEL6	0

A variety of pads may be used for up to four nCE signals to select up to four separate slaves. 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 Master IO instances.

This same setup for 3-wire SPI mode is done for each of the other seven instances of the IO Master 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 Master instance.

9.3.1.4 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 the desired pin in the IOMxIRQ register (x = 0 to 7).

9.3.1.5 Implementing IO Slave Connections

The IO Master module must be correctly connected to the appropriate pads in order to operate.

9.3.1.6 IO Slave I^2C Connection

I^2C mode of the IO Slave uses pad 0 as SCL and pad 1 as SDA. This mode is configured by setting the FNCSELn fields as shown in Table 17. The INPEN0 and INPEN1 bits must be set. PULLCFG0 and PULLCFG1 should be cleared.

Table 17: IO Slave I^2C Configuration

Field	Value
FNCSEL0	1
FNCSEL1	1

9.3.1.7 IO Slave 4-wire SPI Connection

Four-wire SPI mode of the IO Slave uses pad 0 as SCK, pad 1 as MOSI, pad 2 as MISO and pad 3 as nCE. 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 18: IO Slave 4-wire SPI Configuration

Field	Value
FNCSEL0	2
FNCSEL1	2

Table 18: IO Slave 4-wire SPI Configuration

Field	Value
FNCSEL2	1
FNCSEL3	1

9.3.1.8 IO Slave 3-wire SPI Connection

Three-wire SPI mode of the IO Slave 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 19: IO Slave 3-wire SPI Configuration

Field	Value
FNCSEL0	1
FNCSEL1	1
FNCSEL3	1

9.3.1.9 IO Slave Interrupt Connection

The IO Slave 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. FNCSEL4 must be set to 1, and INPEN4 and PULLCFG4 should be cleared.

NOTE

The BLE Controller in the Apollo4 Blue Plus KXR package uses GPIO04 for its 32 kHz clock. This pad cannot be configured for the IO Slave interrupt or any other function.

9.3.2 Implementing Display Controller Connections

The Display Controller (DC) supports connection to displays via two serial ports. 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 used for the DC signals.

The functions of the DC signals are as follows, grouped by interface. See Display Controller chapter for description of signals.

SPI

- DISP_SPI_SCK
- DISP_SPI_SD
- DISP_SPI_DCX
- DC_SPI_CS_N*

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

INPENn and PULLCFGn should be cleared for output signals. For input and bi-directional signals, INPENn and PULLCFGn should be set.

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.

Table 20: DISP Interface Configuration

Field	Value	Signal	Direction	Pad
FNCSEL78	9	DISP_SPI_SCK	Output	78
FNCSEL74	9	DISP_SPI_SD	Bi-directional	74
FNCSEL75	9	DISP_SPI_DCX	Output	75
FNCSEL74	2	DISP_QSPI_D0	Bi-directional	74
FNCSEL75	2	DISP_QSPI_D1	Bi-directional	75
FNCSEL76	2	DISP_QSPI_D2	Bi-directional	76
FNCSEL77	2	DISP_QSPI_D3	Bi-directional	77
FNCSEL78	2	DISP_QSPI_SCK	Output	78

9.3.3 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.

Refer to the "Pad Connections from the Timer/Counter" section in the Counter/Timer Module chapter for more information on configuring a timer's input/output pad.

9.3.4 Implementing UART Connections

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

9.3.4.1 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 21 shows the connections for TX, which should have the corresponding

INPENn and PULLCFGn fields clear. Table 22 shows the connections for RX, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 21: UART0 TX Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
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 22: UART0 RX Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
2	4	✓	✓	✓	✓	✓
22	4	✓	✓	✓	✓	✓
32	4	✓	✓	✓	✓	✓
36	4	✓		✓		
43	4	✓		✓		✓
47	4	✓	✓	✓	✓	✓
55	4	✓	✓	✓	✓	
62	4	✓	✓	✓	✓	✓
68	4	✓	✓	✓	✓	✓
74	4	✓	✓	✓	✓	✓

9.3.4.2 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 23 shows the connections for RTS, which should have the corresponding INPENn and PULLCFGn fields clear. Table 24 shows the connections for CTS, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 23: UART0 RTS Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
4	4	✓	✓	✓	✓	
24	4	✓	✓	✓	✓	✓
38	4	✓	✓	✓	✓	✓
49	4	✓	✓	✓	✓	✓
57	4	✓	✓	✓	✓	✓
58	4	✓	✓	✓	✓	✓
64	4	✓	✓	✓	✓	✓
70	4	✓	✓	✓	✓	✓
76	4	✓	✓	✓	✓	✓

Table 24: UART0 CTS Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
6	4	✓	✓	✓	✓	✓
18	4	✓	✓	✓	✓	✓
40	4	✓		✓		✓
51	4	✓	✓	✓	✓	✓
59	4	✓	✓	✓	✓	
65	4	✓	✓	✓	✓	✓
71	4	✓	✓	✓	✓	✓
77	4	✓	✓	✓	✓	✓

9.3.4.3 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 25 shows the connections for TX, which should have the corresponding INPENn and PULLCFGn fields clear. Table 26 shows the connections for RX, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 25: UART1 TX Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
0	5	✓	✓	✓	✓	✓
12	5	✓	✓	✓	✓	✓

Table 25: UART1 TX Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
20	5	✓	✓	✓	✓	✓
53	5	✓	✓	✓	✓	

Table 26: UART1 RX Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
2	5	✓	✓	✓	✓	✓
14	5	✓	✓	✓	✓	✓
22	5	✓	✓	✓	✓	✓
36	5	✓		✓		
47	5	✓	✓	✓	✓	✓
55	5	✓	✓	✓	✓	
62	5	✓	✓	✓	✓	✓

9.3.4.4 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 27 shows the connections for RTS, which should have the corresponding INPENn and PULLCFGn fields clear. Table 28 shows the connections for CTS, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 27: UART1 RTS Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
4	5	✓	✓	✓	✓	
16	5	✓	✓	✓	✓	✓
24	5	✓	✓	✓	✓	✓
49	5	✓	✓	✓	✓	✓
57	5	✓	✓	✓	✓	✓

Table 28: UART1 CTS Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
6	5	✓	✓	✓	✓	✓
18	5	✓	✓	✓	✓	✓
29	4	✓	✓	✓	✓	✓

Table 28: UART1 CTS Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
51	5	✓	✓	✓	✓	✓
59	5	✓	✓	✓	✓	

9.3.4.5 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 29 shows the connections for TX, which should have the corresponding INPENn and PULLCFGn fields clear. Table 30 shows the connections for RX, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 29: UART2 TX Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
1	4	✓	✓	✓	✓	✓
13	4	✓	✓	✓	✓	✓
21	4	✓	✓	✓	✓	✓
31	4	✓	✓	✓	✓	✓
35	4	✓		✓		
42	4	✓		✓		✓
46	4	✓		✓		
54	4	✓	✓	✓	✓	
61	4	✓	✓	✓	✓	✓
67	4	✓	✓	✓	✓	✓
73	4	✓	✓	✓	✓	✓

Table 30: UART2 RX Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
3	4	✓	✓	✓	✓	✓
11	4	✓	✓	✓	✓	✓
23	4	✓	✓	✓	✓	✓
33	4	✓	✓	✓	✓	✓
37	4	✓	✓	✓	✓	✓
44	4	✓		✓		✓
48	4	✓	✓	✓	✓	✓
56	4	✓	✓	✓	✓	✓

Table 30: UART2 RX Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
63	4	✓	✓	✓	✓	✓
69	4	✓	✓	✓	✓	✓
75	4	✓	✓	✓	✓	✓

9.3.4.6 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 31 shows the connections for RTS, which should have the corresponding INPENn and PULLCFGn fields clear. Table 32 shows the connections for CTS, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 31: UART2 RTS Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
5	4	✓	✓	✓	✓	✓
39	4	✓		✓		✓
50	4	✓	✓	✓	✓	✓

Table 32: UART2 CTS Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
7	4	✓	✓	✓	✓	✓
19	4	✓	✓	✓	✓	✓
28	4	✓	✓	✓	✓	✓
52	4	✓	✓	✓	✓	

9.3.4.7 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 33 shows the connections for TX, which should have the corresponding INPENn and PULLCFGn fields clear. Table 34 shows the connections for RX, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 33: UART3 TX Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
1	5	✓	✓	✓	✓	✓
13	5	✓	✓	✓	✓	✓
21	5	✓	✓	✓	✓	✓
35	5	✓		✓		
46	5	✓		✓		
54	5	✓	✓	✓	✓	
61	5	✓	✓	✓	✓	✓

Table 34: UART3 RX Configuration

Pad	Value	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
3	5	✓	✓	✓	✓	✓
11	5	✓	✓	✓	✓	✓
15	5	✓	✓	✓	✓	✓
23	5	✓	✓	✓	✓	✓
48	5	✓	✓	✓	✓	✓
56	5	✓	✓	✓	✓	✓
63	5	✓	✓	✓	✓	✓

9.3.4.8 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 35 shows the connections for RTS, which should have the corresponding INPENn and PULLCFGn fields clear. Table 36 shows the connections for CTS, which must have the corresponding INPENn field set and should have the corresponding PULLCFGn field clear.

Table 35: UART3 RTS Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
5	5	✓	✓	✓	✓	✓
17	5	✓	✓	✓	✓	✓
50	5	✓	✓	✓	✓	✓
58	5	✓	✓	✓	✓	

Table 36: UART3 CTS Configuration

Pad	FNCSEL	Apollo4	Apollo4 Blue	Apollo4 Plus	Apollo4 Blue Plus KBR	Apollo4 Blue Plus KXR
7	5	✓	✓	✓	✓	✓
19	5	✓	✓	✓	✓	✓
52	5	✓	✓	✓	✓	
60	5	✓	✓	✓	✓	

9.3.5 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.

9.3.5.1 PDM Connections

PDM CLK and DATA signals for each available PDM are connected to separate GPIO pads. Note that CLK and DATA are enabled independently. Table 37 shows the connections for PDM CLK, which should have the corresponding INPENn and PULLCFGn fields clear. Table 38 shows the pins for PDM DATA, which must have the corresponding INPENn field set and the corresponding PULLCFGn field clear.

Table 37: PDM CLK Configuration

PDM Instance	Field	FNCSEL Value	Pad
PDM0	FNCSEL50	0	50
PDM1	FNCSEL52	0	52
PDM2	FNCSEL54	0	54
PDM3	FNCSEL56	0	56

Table 38: PDM DATA Configuration

PDM Instance	Field	FNCSEL Value	Pad
PDM0	FNCSEL51	0	51
PDM1	FNCSEL53	0	53
PDM2	FNCSEL54	0	55
PDM3	FNCSEL57	0	57

9.3.5.2 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 39 shows the connections for I2S CLK, which should have the corresponding PADnINPEN set and the corresponding PULLCFGn field clear.
- Table 40 shows the connections for I2S WS, which should have the corresponding INPENn set and the PULLCFGn field clear.
- Table 41 shows the connections for I2S DATA, which must have the corresponding INPENn and PULL-CFGn fields clear.

- Table 42 shows the connections for I2S SDIN, which must have the corresponding INPENn and PULLCFGn fields clear.
- Table 43 shows the connections for I2S SDOUT, which must have the corresponding INPENn and PULLCFGn fields clear.

> NOTE
>
> The Apollo4 Lite and the Apollo4 Blue Lite include a single I2S module, I2S0. Disregard all I2S1 function selections in the tables below for these devices.

Table 39: I2S CLK Configuration

I2S Instance	Field	FNCSEL Value	Pad
I2S0	FNCSEL5	2	5
I2S0	FNCSEL11	2	11
I2S0	FNCSEL28	2	28
I2S0	FNCSEL47	10	47
I2S1	FNCSEL5	10	5
I2S1	FNCSEL16	2	16
I2S1	FNCSEL47	2	47
I2S1	FNCSEL61	2	61

Table 40: I2S WS Configuration

Instance	Field	FNCSEL Value	Pad
I2S0	FNCSEL7	2	7
I2S0	FNCSEL13	2	13
I2S0	FNCSEL30	2	30
I2S0	FNCSEL49	10	49
I2S1	FNCSEL7	10	7
I2S1	FNCSEL18	2	18
I2S1	FNCSEL49	2	49
I2S1	FNCSEL63	2	63

Table 41: I2S DATA Configuration

I2S Instance	Field	FNCSEL Value	Pad
I2S0	FNCSEL6	2	6
I2S0	FNCSEL12	2	12
I2S0	FNCSEL29	2	29
I2S1	FNCSEL17	2	17
I2S1	FNCSEL48	2	48
I2S1	FNCSEL62	2	62

Table 42: I2S SDIN Configuration

I2S Instance	Field	FNCSEL Value	Pad
I2S0	FNCSEL4	9	4[1]
I2S0	FNCSEL14	9	14
I2S0	FNCSEL27	9	27
I2S0	FNCSEL46	10	46[2]
I2S1	FNCSEL4	10	4[1]
I2S1	FNCSEL19	9	19
I2S1	FNCSEL46	9	46[2]
I2S1	FNCSEL64	9	64

1. The BLE Controller in the Apollo4 Blue Plus KXR package uses GPIO04 for its 32 kHz clock. This pad cannot be configured for the I2S0/I2S1 SDIN function or any other function.
2. The BLE Controller in the Apollo4 Blue Plus KXR package uses GPIO46 for its 32 MHz clock. This pad cannot be configured for the I2S0/I2S1 SDIN function or any other function.

Table 43: I2S SDOUT Configuration

I2S Instance	Field	FNCSEL Value	Pad
I2S0	FNCSEL6	9	6
I2S0	FNCSEL12	10	12
I2S0	FNCSEL29	9	29
I2S0	FNCSEL48	10	48
I2S1	FNCSEL6	10	6
I2S1	FNCSEL17	9	17
I2S1	FNCSEL48	9	48
I2S1	FNCSEL62	9	62

9.3.6 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 one set of pads on the device. Table 44 shows the pads used for SDIO data lines, clock and command signals. The pads for the clock (CLKOUT) and command (CMD) signal 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 44: SDIO/SDIF Configuration

Signal	Field	FNCSEL Value	Pad
SDIF_DATA0	FNCSEL84	2	84
SDIF_DATA1	FNCSEL85	2	85
SDIF_DATA2	FNCSEL86	2	86
SDIF_DATA3	FNCSEL87	2	87
SDIF_DATA4	FNCSEL79	2	79
SDIF_DATA5	FNCSEL80	2	80
SDIF_DATA6	FNCSEL81	2	81
SDIF_DATA7	FNCSEL82	2	82
SDIF_CMD	FNCSEL83	2	83
SDIF_CLKOUT	FNCSEL88	2	88

9.3.7 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.

9.3.8 Implementing CLKOUT Connections

The flexible clock output of the Clock Generator module, CLKOUT, may be configured on several pads as shown in Table 45. INPENn and PULLCFGn should be cleared in each case.

Table 45: CLKOUT Configuration

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

9.3.9 Implementing 32 kHz CLKOUT Connections

In addition to the CLKOUT mux output, there is also a dedicated 32 kHz clock output. 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 46. INPENn and PULLCFGn should be cleared in each case.

Table 46: 32 kHz CLKOUT Configuration

Field	FNCSEL Value	Pad
FNCSEL4	2	4[1]
FNCSEL37	2	37
FNCSEL45	2	45[2]
FNCSEL64	1	64
FNCSEL65	1	65
FNCSEL69	1	69
FNCSEL70	1	70
FNCSEL75	1	75
FNCSEL76	1	76
FNCSEL82	1	82
FNCSEL83	1	83

1. The BLE Controller in the Apollo4 Blue Plus KXR package uses GPIO04 for its 32 kHz clock. This pad cannot be configured for the 32KHzXT clock output or any other function.
2. GPIO45 is not pinned out on the Apollo4 Blue Plus KBR package and cannot be used as the 32KHzXT clock output.

9.3.10 Implementing 32 MHz CLKOUT Connections

There is also a dedicated 32 MHz 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 47. INPENn and PULLCFGn should be cleared in this case.

Table 47: CLKOUT_32M Configuration

Field	Value	Pad
FNCSEL46	2	46[1]

1. GPIO46 is not pinned out on either package of the Apollo4 Blue Plus.

9.3.11 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 48. 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 49. 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 48: ADC Analog Input Configuration

Field	FNCSEL Value	Input	Pad
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 49: ADC Trigger Input Configuration

Field	FNCSEL Value	Input	Pad
FNCSEL7	1	TRIG0	7
FNCSEL11	1	TRIG0	11
FNCSEL15	1	TRIG0	15
FNCSEL27	1	TRIG0	27
FNCSEL29	0	TRIG0	29
FNCSEL36	1	TRIG0	36[1]
FNCSEL41	1	TRIG0	41
FNCSEL49	1	TRIG0	49
FNCSEL50	1	TRIG0	50
FNCSEL54	1	TRIG0	54
FNCSEL59	1	TRIG0	59[2]
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[3]
FNCSEL44	1	TRIG1	44[4]
FNCSEL51	1	TRIG1	51
FNCSEL55	1	TRIG1	55[5]
FNCSEL60	1	TRIG1	60[6]
FNCSEL9	1	TRIG2	9
FNCSEL13	1	TRIG2	13

Table 49: ADC Trigger Input Configuration

Field	FNCSEL Value	Input	Pad
FNCSEL17	1	TRIG2	17
FNCSEL21	1	TRIG2	21
FNCSEL38	1	TRIG2	38
FNCSEL42	1	TRIG2	42[7]
FNCSEL45	1	TRIG2	45[8]
FNCSEL52	1	TRIG2	52[9]
FNCSEL56	1	TRIG2	56
FNCSEL10	1	TRIG3	10
FNCSEL14	1	TRIG3	14
FNCSEL24	1	TRIG3	24
FNCSEL39	1	TRIG3	39[10]
FNCSEL43	1	TRIG3	43[11]
FNCSEL46	1	TRIG3	46[12]
FNCSEL53	1	TRIG3	53[13]
FNCSEL57	1	TRIG3	57

1. GPIO36 is not pinned out on the Apollo4 Blue, either package of the Apollo4 Blue Plus.
2. GPIO59 is not pinned out on the Apollo4 Blue Plus KXR package.
3. GPIO40 is not pinned out on the Apollo4 Blue or the Apollo4 Blue Plus KBR package.
4. GPIO44 is not pinned out on the Apollo4 Blue or the Apollo4 Blue Plus KBR package.
5. GPIO55 is not pinned out on the Apollo4 Blue Plus KXR package.
6. GPIO60 is not pinned out on the Apollo4 Blue Plus KXR package.
7. GPIO42 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.
8. GPIO45 is not pinned out on the Apollo4 Blue Plus KBR package.
9. GPIO52 is not pinned out on the Apollo4 Blue Plus KXR package.
10. GPIO39 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.
11. GPIO43 is not pinned out on the Apollo4 Blue or the Apollo4 Blue Plus KBR package.
12. GPIO46 is not pinned out on the Apollo4 Blue or either package of the Apollo4 Blue Plus.
13. GPIO53 is not pinned out on the Apollo4 Blue Plus KXR package.

9.3.12 Implementing Voltage Comparator Connections

Two types of pad connections may be made for the Voltage Comparator (VCOMP) module. Three reference voltages may be used for the comparator negative input as shown in Table 50. The voltage to be applied to the comparator positive input are shown in Table 51. 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 52. In each case INPENn and PULLCFGn should be cleared.

Table 50: Voltage Comparator Reference Configuration

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

Table 51: Voltage Comparator Input Configuration

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

Table 52: Voltage Comparator Output Configuration

Field	FNCSEL Value	Output	Pad
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[1]
FNCSEL35	9	VCMPO	35[2]
FNCSEL44	9	VCMPO	44[3]
FNCSEL52	9	VCMPO	52[4]
FNCSEL57	9	VCMPO	57
FNCSEL72	9	VCMPO	72
FNCSEL90	9	VCMPO	90[5]
FNCSEL91	9	VCMPO	91
FNCSEL92	9	VCMPO	92[6]
FNCSEL93	9	VCMPO	93
FNCSEL94	9	VCMPO	94[7]

1. GPIO34 is not pinned out on the Apollo4 Blue, either package of the Apollo4 Blue Plus.
2. GPIO35 is not pinned out on the Apollo4 Blue, either package of the Apollo4 Blue Plus.
3. GPIO44 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.
4. GPIO52 is not pinned out on the Apollo4 Blue Plus KXR package.
5. GPIO90 is not pinned out on the Apollo4 Blue Plus KXR package.
6. GPIO92 is not pinned out on either package of the Apollo4 Blue Plus.
7. GPIO94 is not pinned out on either package of the Apollo4 Blue Plus.

9.3.13 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 53, and INPENn and PULLCFGn should be cleared for the selected pad.

Table 53: SWO Configuration

Field	FNCSEL Value	Pad
FNCSEL3	2	3
FNCSEL22	2	22
FNCSEL23	2	23
FNCSEL24	2	24
FNCSEL28	0	28
FNCSEL34	2	34[1]
FNCSEL35	2	35[2]
FNCSEL36	2	36[3]
FNCSEL41	9	41[4]
FNCSEL44	2	44[5]
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

1. GPIO34 is not pinned out on the Apollo4 Blue, either package of the Apollo4 Blue Plus.
2. GPIO35 is not pinned out on the Apollo4 Blue, either package of the Apollo4 Blue Plus.
3. GPIO36 is not pinned out on the Apollo4 Blue, either package of the Apollo4 Blue Plus.
4. GPIO41 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.
5. GPIO44 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.

Optional debug trace output signals, SWTRACE0-SWTRACE3, as well as the trace clock (SWTRACECLK) and trace control signal (SWTRACECTL), may be configured on a variety of pads as shown in Table 54. INPENn and PULLCFGn should be cleared for the selected pads.

Table 54: SW Trace Configuration

Field	FNCSEL Value	Signal	Pad
FNCSEL1	0	SWTRACE0	1
FNCSEL39	2	SWTRACE0	39[1]
FNCSEL51	2	SWTRACE0	51
FNCSEL70	2	SWTRACE0	70
FNCSEL80	4	SWTRACE0	80
FNCSEL2	0	SWTRACE1	2
FNCSEL40	2	SWTRACE1	40[2]
FNCSEL52	2	SWTRACE1	52[3]
FNCSEL71	2	SWTRACE1	71
FNCSEL81	4	SWTRACE1	81
FNCSEL3	0	SWTRACE2	3
FNCSEL41	2	SWTRACE2	41[4]
FNCSEL53	2	SWTRACE2	53[5]
FNCSEL72	2	SWTRACE2	72
FNCSEL82	4	SWTRACE2	82
FNCSEL4	0	SWTRACE3	4[6]
FNCSEL42	2	SWTRACE3	42[7]
FNCSEL54	2	SWTRACE3	54
FNCSEL73	2	SWTRACE3	73
FNCSEL83	4	SWTRACE3	83
FNCSEL0	0	SWTRACECLK	0
FNCSEL38	2	SWTRACECLK	38
FNCSEL50	2	SWTRACECLK	50
FNCSEL43	0	SWTRACECTL	43[8]

Table 54: SW Trace Configuration

Field	FNCSEL Value	Signal	Pad
FNCSEL55	0	SWTRACECTL	55[9]

1. GPIO39 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.
2. GPIO40 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.
3. GPIO52 is not pinned out on the Apollo4 Blue Plus KXR package.
4. GPIO41 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.
5. GPIO53 is not pinned out on the Apollo4 Blue Plus KXR package.
6. The BLE Controller in the Apollo4 Blue Plus KXR package uses GPIO04 for its 32 kHz clock. This pad cannot be configured for the 32KHzXT clock output or any other function.
7. GPIO42 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.
8. GPIO43 is not pinned out on the Apollo4 Blue or the KBR package of the Apollo4 Blue Plus.
9. GPIO55 is not pinned out on the Apollo4 Blue Plus KXR package.

9.3.14 Fast GPIO

9.3.14.1 Description

Access to GPIO pin registers on the device can be multiple CPU cycles to complete. To support certain functions that require shorter latency access, a fast GPIO interface is supported. The fast GPIO is accessed via the fast GPIO registers shown in the next section.

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

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

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

10.1 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 all Apollo4 family SoCs offers four options for the reference clock source and frequency via the GPADC's CFG_CLKSEL field:

1. HFRC at 48 MHz (default setting)
2. Inverted HFRC at 48 MHz
3. HFRC at 24 MHz
4. HFRC2 at 48 MHz

NOTE
Due to an erratum, ERR091, only the 24 MHz HFRC setting (CLKSEL = 0x2) is supported. The other three clock options should not be used. Also, due to erratum ERR113, setting additional input signal sampling/tracking time of at least 32 ADC clock cycles (SLnCFG_TRKCYCn = 32), for a total minimum of 37 sampling/tracking cycles, should be used.

10.1.1 11 Channel Analog Mux

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_DIV3 internal voltage divide by 3 connection to the input power rail.
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.

10.1.2 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 Comparator trigger.
5. <Reserved>
6. <Reserved>
7. ADC_SWT software trigger.

Refer to the CFG Register in the GPADC register set for the Apollo4. 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.

10.1.3 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.

10.1.4 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. Slots are enabled when the LSB of the slot configuration is set to one. See "One SLOT Configuration Register" on page 85.

Table 55: One SLOT Configuration Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Reserved					# Samples to Accum.			Reserved												CHANNEL SELECT				Reserved						WINDOW_COMP	SLOT_ENABLE

The window comparator enable is discussed in a subsequent section below. See "Window Comparator" on page 89. The number of samples to accumulate is explained in a subsequent section. See "Automatic Sample Accumulation and Scaling" on page 85.

As described above, the channel select bit field specifies which one of the analog multiplexer channels will be used for the conversions requested for an individual slot. See "11 Channel Analog Mux" on page 83.

Each of the eight conversion slots can independently specify:

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

10.1.5 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. Table 56 shows the format that is used by all conversions. This is a scaled integer format with a 6-bit fractional part. The precision mode for each determines the format for the FIFO data. 12-bit, 10-bit and 8-bit precision modes respectively correspond to 12.6, 10.6 and 8.6 formats.

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.

Table 56: 14.6 GPADC Sample Format

19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
14-bit Integer														6-bit Fraction					

Each slot contains a 21-bit accumulator as shown in Table 57, "Per Slot Sample Accumulator," on page 86. 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.

If a slot is set to accumulate 128 samples per result then the accumulator could reach a maximum value of:

$128*(2^{14}-1) = 128*16383 = 2097024 = 2^{21} - 128$, hence the 21 bit accumulator.

Table 57: Per Slot Sample Accumulator

20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Accumulator																				

Table 59 shows the maximum possible accumulated values. Note that 64 sample accumulation produces a result that is exactly correct for the 14.6 format results so it is copied unscaled in to the FIFO.

Note that 128 sample accumulation can produce a result that is too large for the 14.6 format since it may result in 7 bits of valid fractional data. All of the remaining sample accumulation settings must have their results left shifted to produce the desired 14.6 format.

Finally, note that for the 128 sample accumulation case, the LSB of the accumulator is discarded when the results are written to the FIFO.

Table 58: Accumulator Scaling

# Samples	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
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									

Table 59: Accumulator Scaling

# Samples	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
128	14.6																				0

Table 59: Accumulator Scaling

# Samples	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
64	X	14.6																			
32	X	X	14.5																		
16	X	X	X	14.4																	
8	X	X	X	X	14.3																
4	X	X	X	X	X	14.2															
2	X	X	X	X	X	X	14.1														
1	X	X	X	X	X	X	X	14													

10.1.6 Sixteen Entry Result FIFO

All results written to the FIFO have exactly the same format as shown in Table 60. The properly scaled accumulation results are written the lower half word in the aforementioned 14.6 format. Since each slot can produce results at a different rate, the slot number generating the result is also written to the FIFO along with the total valid entry count within the FIFO.

Table 60: FIFO Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RSV	Slot Number.			FIFO Count								FIFO DATA																			

Table 61: 12-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	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

Table 62: 10-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	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 63: 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
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
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 µs 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 conver-

sion 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 FIFOOVR1 (FIFO 75% full) interrupt status bit will be set. When a conversion completes resulting in hardware populating the 16th valid FIFO entry, the FIFOOVR2 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.

10.1.7 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 64: Window Comparator Lower Limit Register

19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Lower Limit																			

Table 65: Window Comparator Upper Limit Register

19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
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 Lower Limit <= GPADC SAMPLE <= 14.6 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 by 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.

10.2 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 over-riding 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 reseting 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 13

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

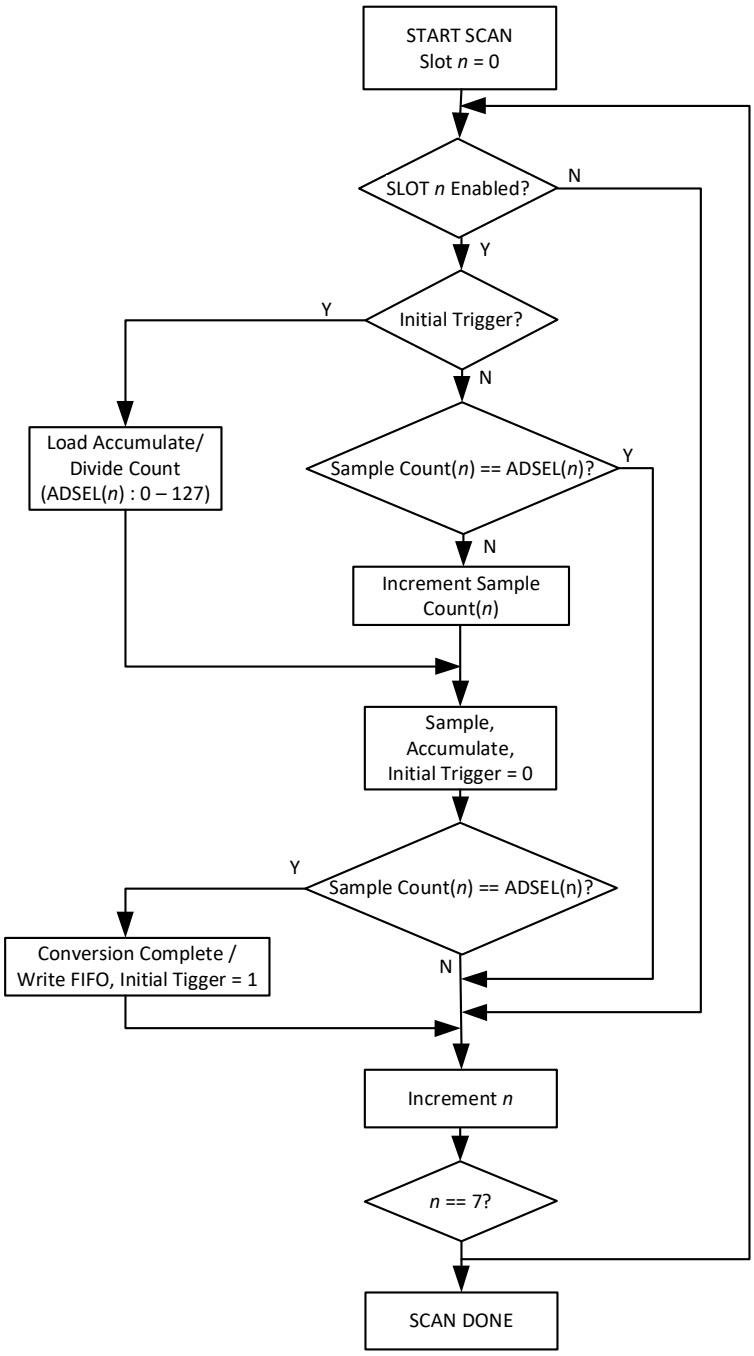


Figure 13. Scan Flowchart

10.2.1 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.

10.2.2 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.

10.2.3 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 66: GPADC Power Modes

LPMODE	Definition	Entry Latency
0	GPADC is kept active continuously (used in continuous sampling scenarios)	0 (requires initial calibration)
1	GPADC is mostly powered off between samples, HFRC is duty cycled between samples. No calibration required after initial calibration)	<70µs (shorter for lower resolution)
2	GPADC is completely powered off between samples, HFRC is duty cycled between samples. Requires recalibration for each conversion.	<660µs

10.2.3.1 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 triggers[1].

10.2.3.2 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 while maintaining

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

GPADC calibration data. This mode may operate autonomously without CPU interaction, even while the CPU is in sleep or deepsleep 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.

10.2.3.3 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 and a calibration sequence 100's of microseconds, nominally. In this mode, the GPADC must be reconfigured prior to any subsequent GPADC operation.

10.3 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 89.

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 µs is required before reading the FIFO, otherwise the FIFO data count may be incorrect.

10.4 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 Repeatable 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.

11. Multi-bit Serial Peripheral Interface Master Module (MSPI)

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

> NOTE
>
> Only two MSPI instances are pinned out on the Apollo4 Blue and the KBR package of the Apollo4 Blue Plus, as MSPI1 is used for internal communication between the Apollo4 MCU and the BLE Controller module. Consult the datasheet for possible operating limitations of the individual MSPI instances.

> NOTE
>
> Use of Device1 is not supported on the MCU. Disregard any reference to or inference of multiple MSPI devices.

> NOTE
>
> Check the datasheet of the applicable device for MSPI clocking and functional restrictions.

11.1 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 reference frequency is 96 MHz, so the resulting clock frequency is 96/CLKDIV value.
- Configure MSPIn transfer characteristics (DEV0CFG register) to initialize the device (usually mode 0, serial transfers).
- Configure MSPIn PADOUTEN to enable the desired bits on the MSPIn bus (clock plus relevant data bits).

> NOTE
>
> Enabling unused data lines will impact the values present on those pads even if the GPIO function select is not set to MSPI.

- Program external flash 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).
- Write 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 enable 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.

11.2 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_SENDI and CTRL_SENDA 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_SENDI_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];
  }
 }
```

11.3 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 it's 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).

11.3.1 Configuring MSPI as a DMA Target and a DMA Client Concurrently

A DMA deadlock may occur when there is heavy traffic of concurrent DMA accesses such as when the MSPI is used as both a "DMA client", where MSPI is sourcing or sinking data through the DMA, and a "DMA target", where MSPI is a memory-mapped source or destination for other peripheral DMA. For example, a situation may exist when the ADC is targeting a memory device through the MSPI XIPMM aperture as a the ADC sample "DMA target" at the same time that the MSPI is using DMA itself to target SSRAM or TCM. This condition may result in a DMA deadlock due to a circular dependency when the APBDMA-AXI, MSPI-AXI, MSPI-XIPDMA and APBDMA-ARBITOR states are blocking or waiting for DMA resources.

To avoid this potential problem, software should control the DMA's configuration to alternate between MSPI as a DMA client and as a DMA target so as not to allow overlap of these DMA accesses. Note that there should not be a threat of this deadlock situation when short CPU accesses such as XIP or memory mapped MSPI are occurring, due to the location of arbitration against DMA traffic.

11.4 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.

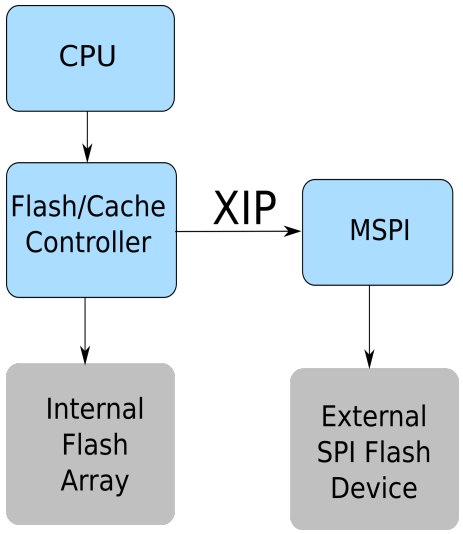


Figure 14. XIP Block Diagram

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.

11.4.1 XIP Address Mapping and Accesses

The MSPI additionally supports a memory-mapped XIP mode (XIP) that enables full read/write mapping of an MSPI device such as a PSRAM to the CPU's peripheral address map at offset 0x51000000-0x14000000-0x1FFFFFFF. This is mapped to offset 0 of the device on the MSPI bus and is not cached (unlike XIP space) and thus can be used as an extension to system SRAM. MSPI1 and MSPI2 are similarly mapped at 0x18000000 and 0x1C000000, respectively. The MSPI device can be accessed by both XIP and XIP accesses (the regions overlap), but it is recommended that XIP used for static data/ instructions and that a separate area of the MSPI device is used for read/write operations to avoid having stale data visible in the cache.

XIP seamlessly supports word, halfword, and byte read and write accesses, however, there are a few restrictions and caveats:

- As mentioned above, writes to XIP do not flush cached data to the same address.
- 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 XIP should be used for infrequently used data.

11.4.2 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.

11.4.3 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.

11.5 Command Queueing (CQ)

The MSPI's command queuing (CQ) interface is similar to command queuing 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 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).

11.5.1 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 67: 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;
```

11.5.2 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.

11.5.3 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. The table below lists the flags available in the MSPI:

Table 68: 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.
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.

Table 68: CQFLAGS

Bit	Type	Mnemonic	Description/Use
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.

11.5.4 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.

11.5.5 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 supports 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, it's 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.

11.6 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).

11.7 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.

11.8 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.

NOTE

An MSPI timing window scan function is available to determine optimum values for MSPI timing parameters such as RXDQSDELAY, RXNEG, RXCAP, etc., and other timing settings on the Apollo4 and Apollo4 Plus SoCs.

For Apollo4, it is recommended to run the window function and capture the timing values on a "per device" basis, meaning that these values should be ascertained and used for each MSPI instance, per board.

For Apollo4 Plus, timing values can be determined on a "per design" basis using a population of boards of the same design. Once the timing values are determined for each MSPI instance in the design, they can be applied to all boards without having to derive the timing settings on each board. The exception to this guidance is when an MSPI interface is configured for non-DQS mode using any data width. In this case, the window scan should be used on a "per device" basis.

Consult the Apollo4 or the Apollo4 Plus datasheet for timing and mode limitations for each MSPI instance on the applicable device

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 master (MSPI) and target (flash) device. However, the presence of delays in the system complicates the timing, and the timing diagram shown in Figure 15 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.)

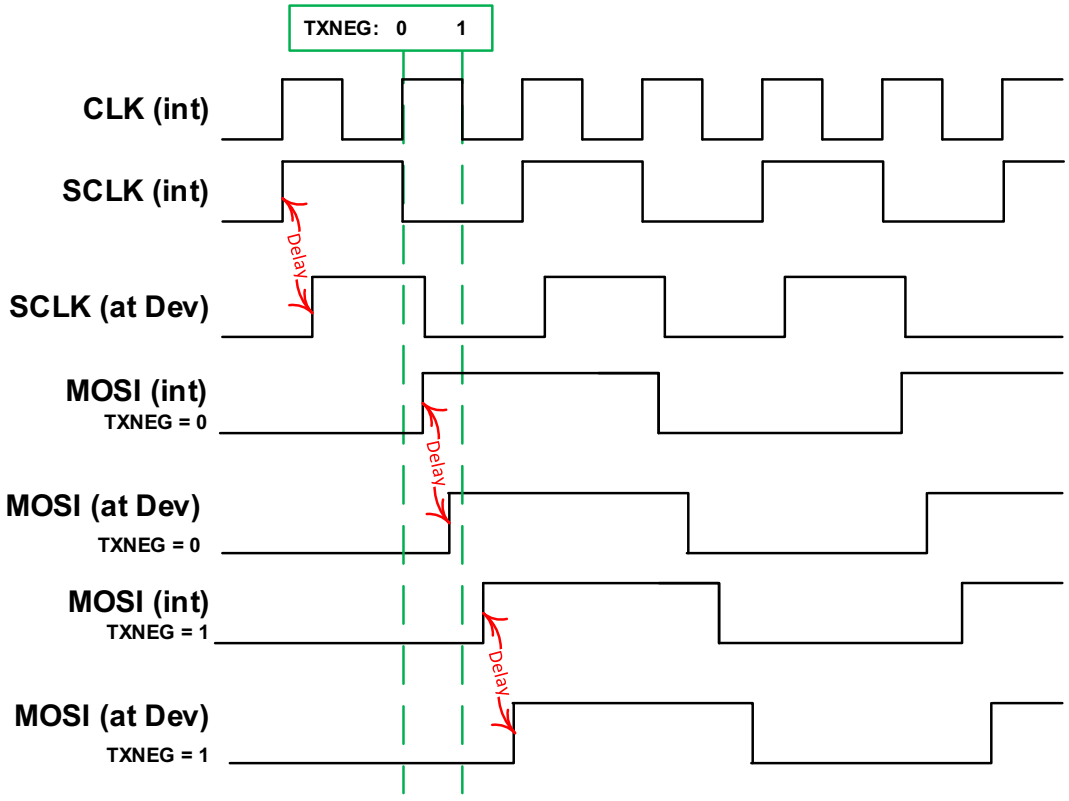


Figure 15. MSPI TX Interface Timing at SCLK = 48 MHz

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.

Target to master (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 master to target, 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 16). 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.

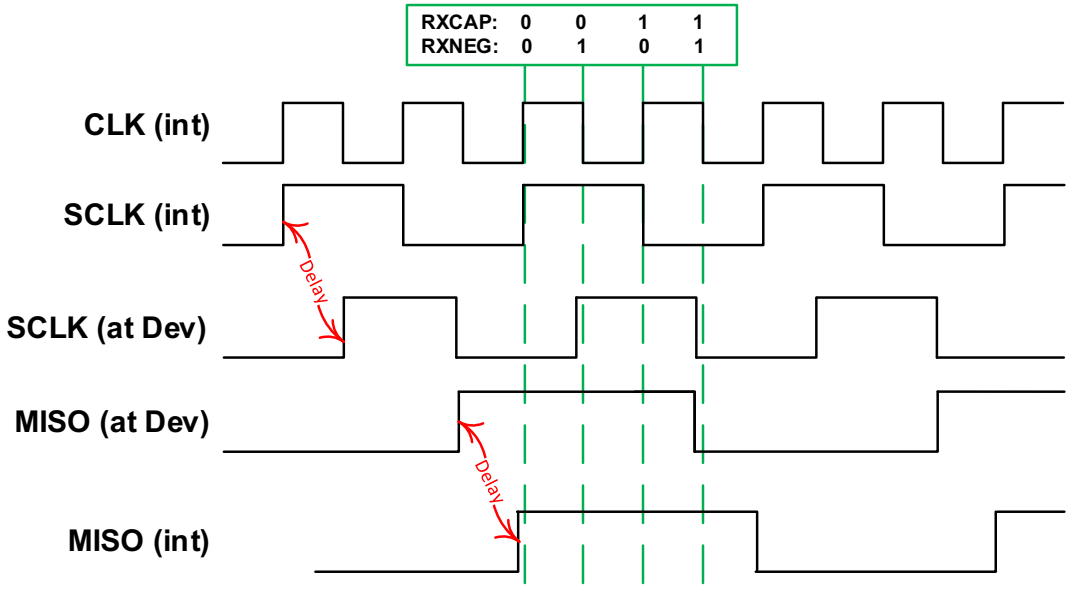


Figure 16. MSPI RX Interface Timing at SCLK = 48 MHz

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.

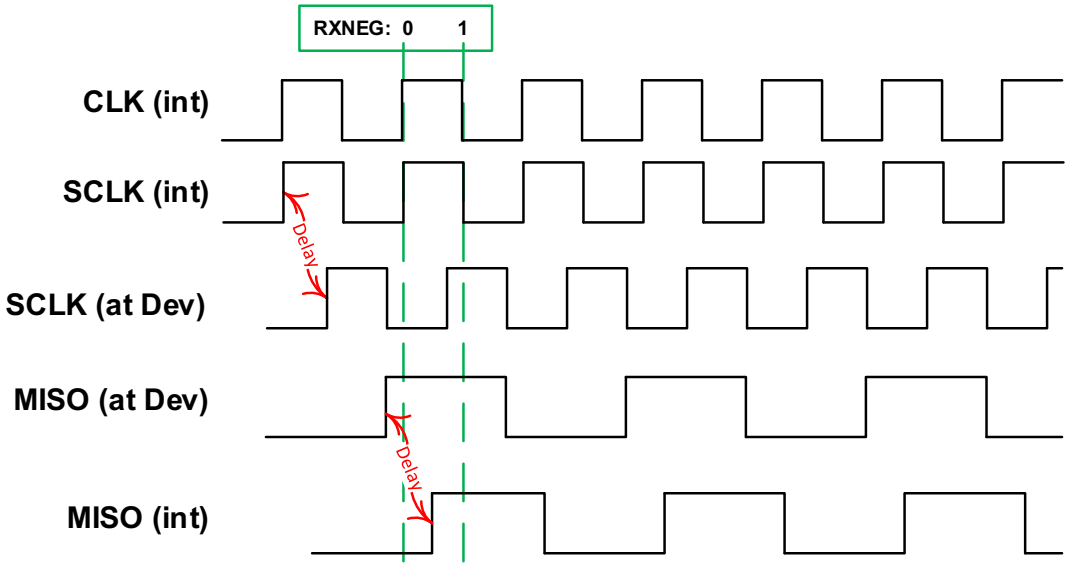


Figure 17. MSPI RX Interface Timing at SCLK = 96 MHz

11.8.1 Other MSPI Pin Timing Considerations for Apollo4 Plus

The MSPI logic contains additional controls to adjust I/O timings to accommodate differences in board or device timings through various register settings.

Derived from the HFRC clock, the aclk is the internal asynchronous clock input to MSPI running at 96 MHz. The aclk goes through the TX clock divider which is set by DEVnCFG_CLKDIVn. The output of the divider, mspi_clk, is routed all the way to clock pad at the chip. The gpio padmux out enable is always high since we need this clock in both TX and RX modes. mspi_clk or SCLK is active in command, address and data phase. The MSPI receives or sends data only if mspi_clk or SCLK is active.

11.8.1.1 Timing control updates on Apollo4 Plus

There are several changes and additional controls over those found on Apollo4 SoC to fine-tune the I/O timing on Apollo4 Plus. Table 69 lists the differences and additions to help in the migration from Apollo4 to Apollo4 Plus. Note that DEV1 registers are not shown, as DEV1 is not supported on Apollo4 or Apollo4 Plus.

Table 69: Registers/Fields Changed from Apollo4 to Apollo4 Plus

Register	Field	Change
CTRL1	PIOMIXED	Field moved from CTRL to new CTRL1 register.
PADOUTEN	CLKOND4	Field position changed.
	OUTEN	Field size changed.
PADOVEREN	OVERRIDEEN	Field size changed.
PADOVER	OVERRIDE	Field size changed.
DEV0CFG	SEPIO0	Field position changed.
	ISIZE0	Field position changed.
	ASIZE0	Field position changed.
	DEVCFG0	Field size changed.
DEV0DDR (new fields)	RXDQSDELAYHIEN0	New field. When set, RXDQSDELAYHI and RXDQSDELAYNEGHI are used to fine-tune timing of edges for the second DQS in Hex mode.
	RXDQSDELAYNEGHI0	New field. Used to set an offset to the computed value (should be set to 0 by default) of falling edge for the second DQS in Hex mode.
	RXDQSDELAYHI0	New field. Used to set offset to the computed value (should be set to 0 by default) of rising edge for second DQS in Hex mode.
	RXDQSDELAYNEGEN0	New field. When 1, RXDQSDELAYNEG0 is used to fine-tune falling edge of DQS.
	RXDQSDELAYNEG0	New field. Used to set an offset to the computed value (should be set to 0 by default) of falling edge of DQS.

Table 69: Registers/Fields Changed from Apollo4 to Apollo4 Plus

Register	Field	Change
DEV0DDR (removed or deprecated fields)	OVERRIDERXDQSDELAY0	Field removed.
	OVERRIDEDDRCLKOUTDELAY0	Field removed.
	QUADDDR0	Field deprecated. No effect on Apollo4 Plus. Use DEV0CFG for quad devices.
DEV0DDR (changed position)	ENABLEFINEDELAY0	Field position changed.
	RXDQSDELAY0	Field position changed.
	TXDQSDELAY0	Field position changed.
DEV0XIP	XIPMIXED0 and other fields	XIPMIXED field size changed. XIPWRITELATENCY0, XIPTURNAROUND0, XIPENWLAT0 and XIPENDCX0 fields changed position.
DEV0BOUNDARY	DMATIMELIMIT	Step size changed.
DEV0XIPMISC	XIPBOUNDARY	Field deprecated. No effect on Apollo4 Plus.
DMACFG	DMATXEMPT	New field. For DMA_M2P, only start when DMA fifo is not empty.
INTEN, INTSTAT, INTCLR, INTSET	APBDMAERR	New field. Interrupt signifying that MSPI is DMA target as well as DMA source, which may result in deadlock.
DEV0CFG1 (new register)	DQSTURN0	New field. In DQS mode, the internal cycle count to enable DQS path.
	RXHI0	New field. Force st_rx to start at clock high of mspi_clk.
	TAFOURTH0	New field. Add 1/4 mspi_clk in DDR to turnaround. Recommend setting this to 1 when EMULATEDDR is set to non-DQS mode (ENABLEDQS = 0).
	HYPERIO0	New field. When using Winbond, set this bit to 1 to generate CA[47:0] in hardware.
	RXSMP0	New field. Sampling edge based on sclk edge. No effect when div1.
	RBX0	New field. Enable the support of RBX (page boundary crossing on read).
	WBX0	New field. Enable the support of WBX (page boundary crossing on write).
	SCLKRXHALT0	New field. Halt sclk based on xfer_count.
	RXCAPEXT0	New field. Specify the number of apb_clk's of RX capture phase (RXCAPEXT, RXCAP).
	SFTURN0	New field. Subtract from internal counter of write latency and turnaround.

For MSPI0, there are 4 separate delay lines for each byte captured at the respective active clock edge. If needed, RXDQSDELAYNEG can be set to a different value from RXDQSDELAY if an imbalanced duty cycle is needed. Note that this is not chip boundary timing. It is internal data (propagated from DQ) setup time to internal falling/rising sampling clock (propagated from DQS) assuming DQ and DQS arrive at the pads simultaneously.

11.8.1.2 Timing Adjustments in DQS and Non-DQS modes

It is recommended to tune MSPI for proper alignment before programming or reading external memories.

11.8.1.2.1 DQS Mode

In TX mode and in command mode, DQS/DM is the output of MSPI to the device and is used for data masking. DQS/DM is affected by internal write strobes coming from the AXI subsystem to MSPI and by the number of bytes (XFERCOUNT) to be transferred out in DMA mode.

In RX mode, DQS is the capture clock (capture_clk = DQS). DQS/DM or DS, if supported by the device, is the data strobe which indicates availability of data at the pads. The MSPI module picks up the data at DQS edges and stores it in the internal FIFO before transferring to internal memories. Sometimes, due to process variations and/or internal propagation delays, DQS edges may end up at MSPI before or after valid DQ leading to improper data capture and store in MSPI FIFOs. Ideally DQS would align with the center of the valid data window.

There are two situation where data capture fails in DQS mode:

- DQS arrives at MSPI later than DQ

In this case tuning can be done by reducing XIPTURNAROUND/TURNAROUND in steps of 1 and by adding RXDQSDELAY where (RXDQSDELAY × gate_delay) < (mspi_clk period × n, where n = number of cycles reduced in XIPTURNAROUND/TURNAROUND).

- DQS arrives at MSPI sooner than DQ

This case requires that DQS be tuned internally by adding RXDQSDELAY × gate_delay to it in steps of 1 until all the data captured is as expected. In the end, adding delays ensures that capture clock at MSPI aligns to the canter of valid DQ window.

NOTE
Use and setting of the DEV0DDR_ENABLEFINEDELAYn is not needed on any Apollo4 family SoC. The enabling of DQS by setting DQSENABLEn automatically uses the delay selected by the RXDQSDELAYn setting.

Figure 18 shows a situation where DQ and DQS from outside the chip does not align when they arrive at MSPI RX for a proper data capture (rise and fall times exaggerated for clarity). Here DQS alignment with data is tuned by configuring DEV0CFG_RXDQSDELAY by adding n delay gates to DQS where n = RXDQSDELAY. The delay of each delay gate may range from 300 ps to 600 ps. For SDR 96 MHz, this delay must not exceed 10.42 ns due to the risk of losing the capture window.

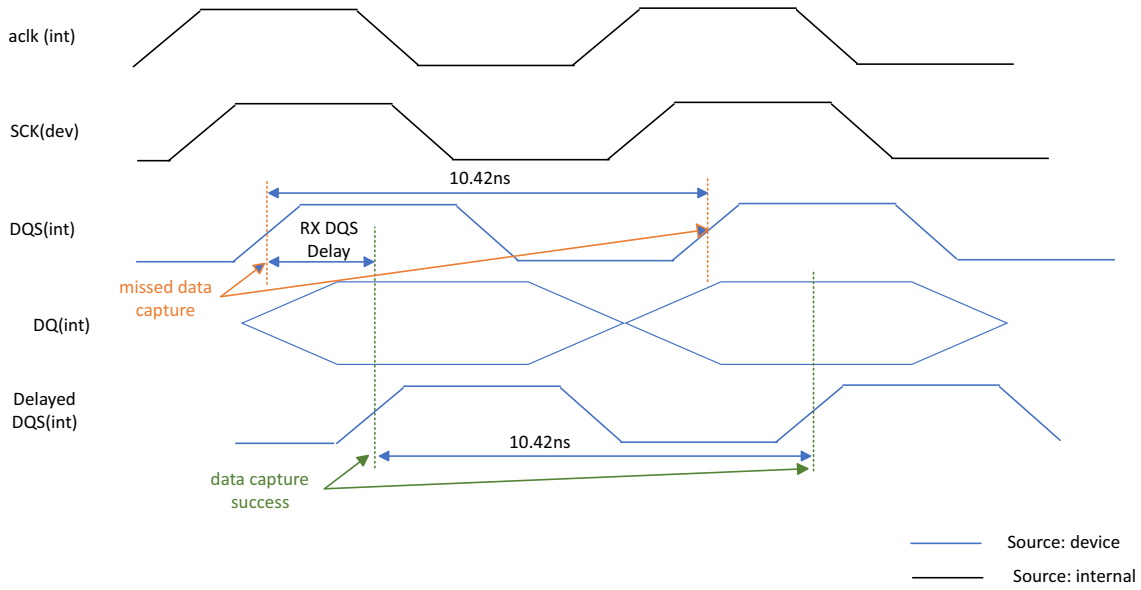


Figure 18. RX DQS Delay in SDR DQS Mode (96MHz)

11.8.1.2.2 Non-DQS Mode

In TX mode, the operation is the same as for DQS mode.

In RX mode, DQS from the device is ignored. Instead the internal aclk is used as the capture clock (capture_clk = aclk). Some devices may not support DQS so this mode was included for such devices. The aclk, by definition, is asynchronous and is internal. MSPI uses the aclk to capture data right after TURNAROUND/XIPTURNAROUND has expired. At the expiration of turnaround device starts sending data on DQ, the aclk needs to be in alignment with the valid DQ window. Because of its asynchronous nature, the aclk edges may not align with valid DQ window.

Data capture may fail in non-DQS mode similar to DQS mode and DQS tuning methods can be applied in each case:

- capture_clk arrives at MSPI later than DQ
- capture_clk arrives at MSPI sooner than DQ

In addition to adjusting turnaround cycles and RXDQSDELAY, non-DQS mode allows more fine tuning using RXCAP and RXNEG described in the sections below.

Figure 19 shows the effect of applying a delay via the setting of RXDQSDELAY.

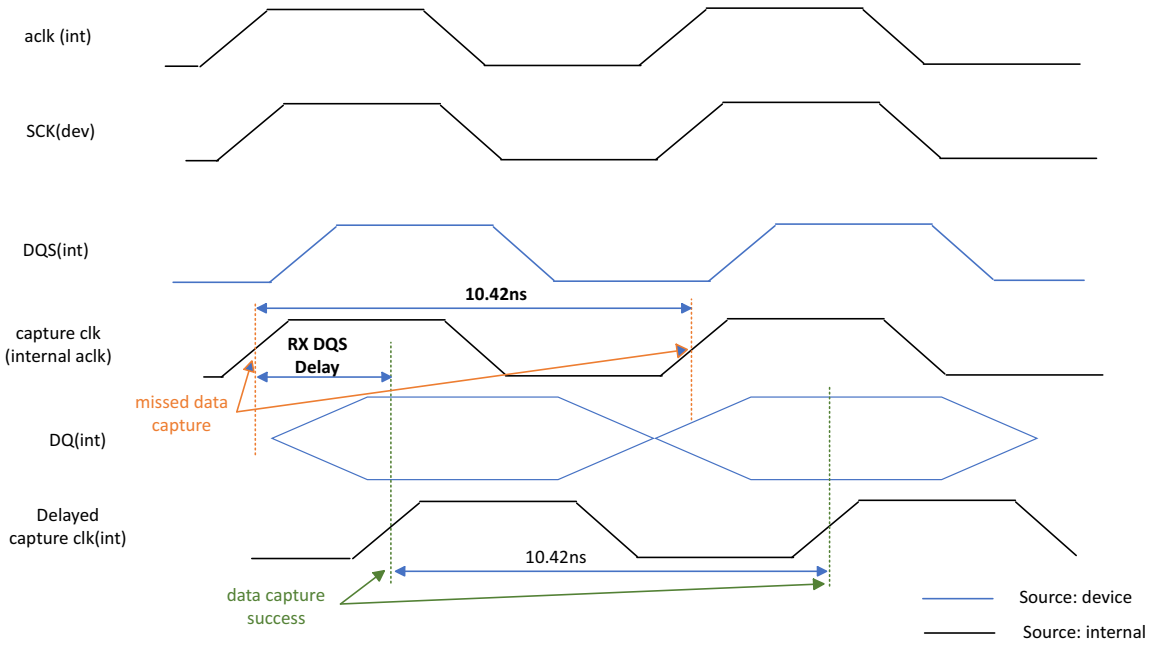


Figure 19. RX DQS Delay in SDR Non-DQS Mode (96MHz)

11.8.1.2.3 Clock Stopping

Another feature in non-DQS mode is clock stopping. If MSPI traffic is heavy or if the internal memory subsystem is slow in sending or accepting requests, MSPI FIFOs overflow (RX) or underflow (TX) and are not able to send or capture data continuously. These conditions are indicated by FIFOFULL or FIFOEMPTY interrupts. MSPI pauses the mspi_clk or SCLK until the FIFOs are not empty or full.

When the clock is paused, the device stops incrementing the current address in TX mode. In RX mode, the device stops sending data at the next address when MSPI is not ready to capture.

12. I²C/SPI Master (IOM)

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

12.1 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);
```

12.2 Interface Clock Generation

The I²C/SPI Master (IOM) can generate a wide range of I/O interface clocks, as shown in Figure 20. 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.

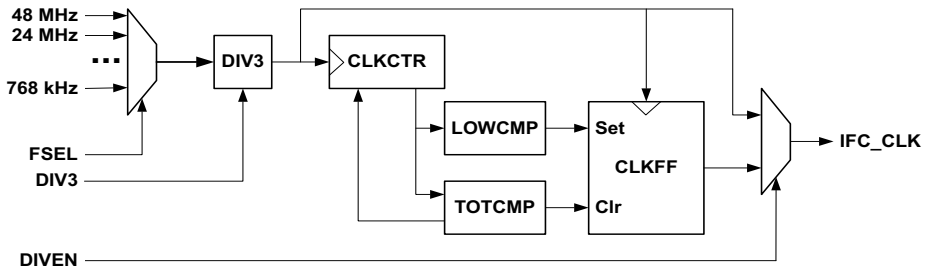


Figure 20. I²C/SPI Master Clock Generation

12.3 Command Operation

In order to minimize the amount of time the CPU must be awake during I²C/SPI Master operations, the architecture of the I²C/SPI Master 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 slave address or the SPI channel select, the desired address offset and the length of the transfer. At that point the I²C/SPI Master executes the entire transfer, so the CPU can go to sleep. If more than 64 bytes are to be transferred, the Master 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 Master 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 Master initiates the read and transfers read data to the FIFO. If the FIFOSZ value exceeds the read threshold IOMn_FIFOTHR_FIFORTHR, 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 Master 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 Master 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 Master will generate an ICMD error interrupt.

12.4 FIFO

The I2C/SPI Master 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, FIFORTHR and FIFOWTHR indicate when a THR interrupt should be generated to signal the processor that data should be transferred.

12.5 I2C Interface

The I2C/SPI Master 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 "master". The devices that are controlled by the master are called "slaves". The I2C Master is always a master 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 21) and are described in the following sections

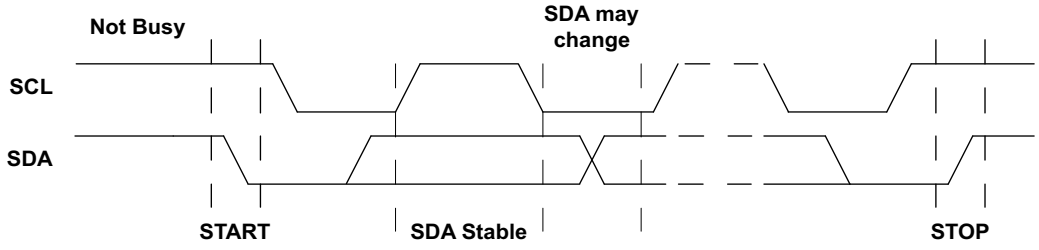


Figure 21. Basic I2C Conditions

I2C operations may transfer up to 512 bytes in a single transfer.

12.5.1 Bus Not Busy

Both SDA and SCL remain high.

12.5.2 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.

12.5.3 Stop Data Transfer

A change in the state of SDA from low to high, while SCL is high, defines the STOP condition.

12.5.4 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.

12.5.5 Acknowledge

Each byte of eight bits is followed by one acknowledge (ACK) bit as shown in Figure 22. This acknowledge bit is a low level driven onto SDA by the receiver, whereas the master generates an extra acknowledge related SCL pulse. A slave receiver which is addressed is obliged to generate an acknowledge after the reception of each byte. Also, on a read transfer, a master receiver must generate an acknowledge after the reception of each byte that has been clocked out of the slave 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 master receiver must signal an end-of-data to the slave transmitter by not generating an acknowledge (a NAK) on the last byte that has been clocked out of the slave. In this case, the transmitter must leave the data line high to enable the master to generate the STOP condition. If I/O Host attempts an I^2C operation but no slave device generates an ACK, or if a slave fails to generate an ACK on a data byte before the transfer is complete, a NAK interrupt will be generated.

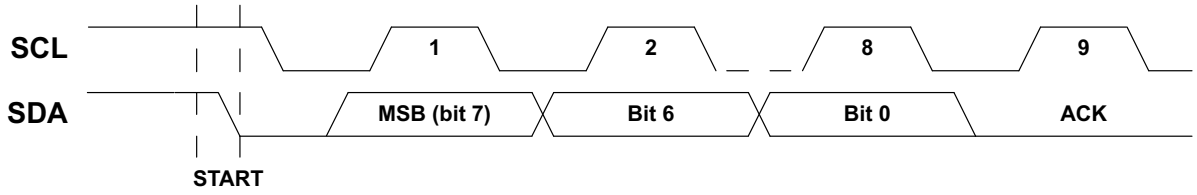


Figure 22. I^2C Acknowledge

12.5.6 I^2C Slave Addressing

For normal I^2C 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 23 shows the operation in 7-bit mode in which the master addresses the slave 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 slave device, the eighth bit indicates a write (RW = 0) or a read (RW = 1) operation and the slave supplies the ACK. If no slave acknowledges the address, the transfer is terminated and a NAK error interrupt is generated.

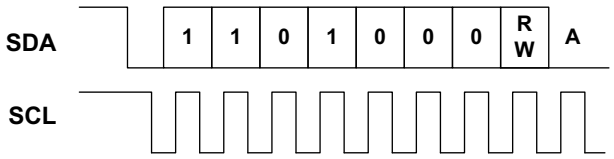


Figure 23. I²C 7-bit Address Operation

Figure 24 shows the operation with which the master addresses the slave 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 slave 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 slave again supplies the ACK. If no slave acknowledges either address byte, the transfer is terminated and a NAK error interrupt is generated.

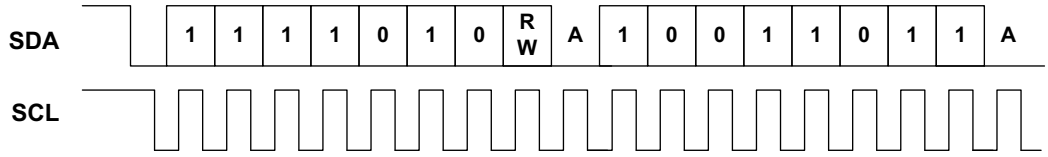


Figure 24. I²C 10-bit Address Operation

12.5.7 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 Master 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 25. The Offset Address is loaded into the Address Pointer of the slave.

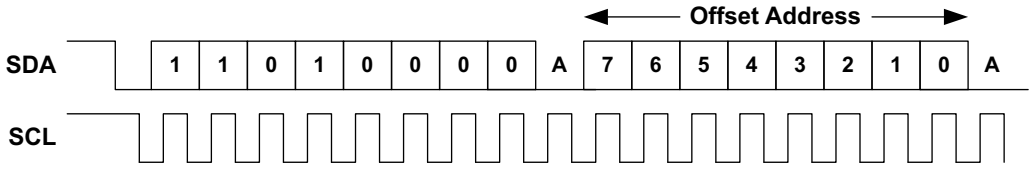


Figure 25. I²C Offset Address Transmission

12.5.8 I²C Write Operation with Address Offset

In a write operation the I²C/SPI Master transmits to a slave receiver. The Address Operation has a RW value of 0, and the second byte contains the Offset Address, as in Figure 25. The next byte is written to the slave 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 26.

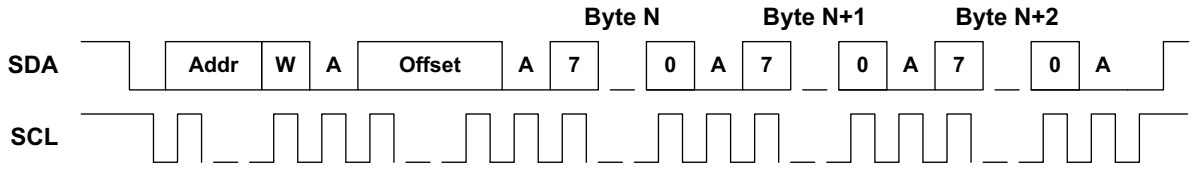


Figure 26. I²C Write Operation with Address Offset

12.5.9 I²C Read Operation with Address Offset

If a read operation with address offset is selected, the I²C/SPI Master first executes an Offset Address Transmission to load the Address Pointer of the slave with the desired Offset Address.

A subsequent operation will again issue the address of the slave but with the RW bit as a 1 indicating a read operation. As shown in Figure 27, this transaction begins with a RESTART condition so that the interface will be held in a multi-master environment. After the address operation, the slave 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 Master receiver responds with a NAK and a STOP to complete the operation.

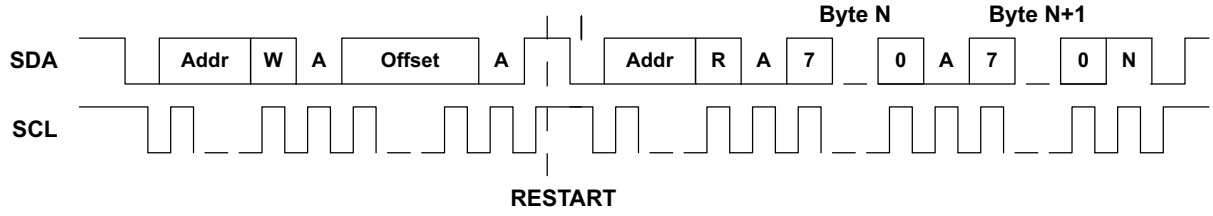


Figure 27. I²C Read Operation with Address Offset

12.5.10 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 Master does not execute the Offset Address Transmission, but simply begins transferring bytes as shown in Figure 28. This provides support for slave devices which do not implement the standard offset address architecture. The OFFSETHI and OFFSETLO fields are not used in this case.

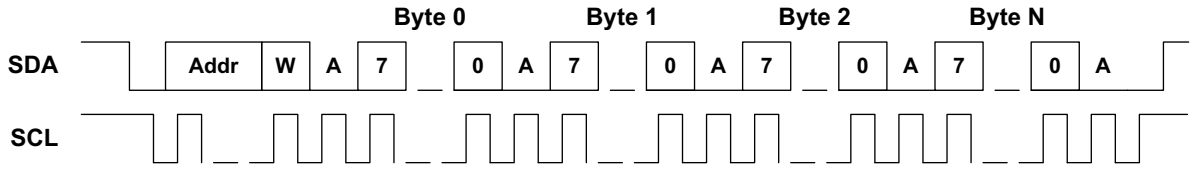


Figure 28. I²C Write Operation with No Address Offset

12.5.11 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 Master does not execute the Offset Address Transmission, but simply begins transferring bytes with a read as shown in Figure 29. This is important for slave 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.

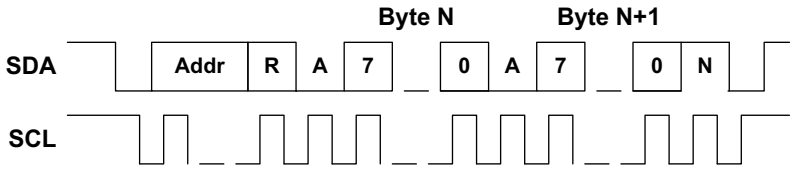


Figure 29. I2C Read Operation with No Address Offset

12.5.12 Holding the Interface with CONT

In all of the previously described transactions, the I2C/SPI Master terminates the I2C operation with a STOP sequence. In environments where there are other masters connected to the I2C interface, it may be necessary for the MCU to hold the interface between Commands to insure that another master does not inadvertently access the same slave 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 Master 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 it CONT is set the read transaction will not terminate with a STOP.

12.5.13 I2C Multi-master Arbitration

The I2C/SPI Master supports multi-master arbitration in I2C mode. There are two cases which must be handled.

The first is the case where another master initiates an I2C operation when the MCU Master is inactive. In this case the I2C/SPI Master 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 master initiates an operation at the same time as the MCU. In this case there will be a point where one master detects that it is not driving SDA low but the bus signal is low, and that master loses the arbitration to the other master. If the MCU I2C/SPI Master 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.

12.6 SPI Operations

12.6.1 SPI Configuration

The I2C/SPI Master 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.

12.6.2 SPI Slave Addressing

In SPI mode, the Command specifies the slave channel to be used in the CMDSEL field. The I2C/SPI Master supports up to four slaves, each of which has its own nCE signal which can be configured on an IO pin. Additional slaves may be supported using GPIO pins and external decoding.

12.6.3 SPI Write with Address Offset

Figure 30 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 Master pulls one of the four nCE signals low. At that point the I2C/SPI Master begins generating the clock on SCK and the offset address is transmitted from the master 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 slave expects a RW bit. If the slave 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 Master 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 Master.

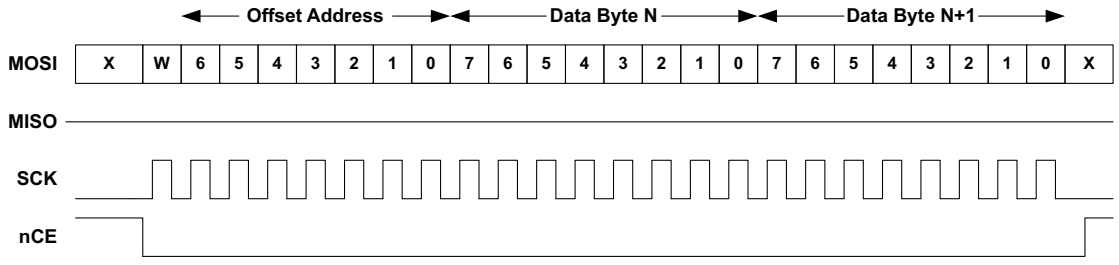


Figure 30. SPI Normal Write Operation (Single-byte Offset Address)

12.6.4 SPI Read with Address Offset

Figure 31 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 Master pulls one of the four nCE signals low. At that point the I2C/SPI Master begins driving the clock onto SCK and the address is transferred from the master to the slave 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 Master stops driving the MOSI line and begins loading the FIFO with the data on the MISO line. The transfer continues until the I2C/SPI Master brings the nCE line high.

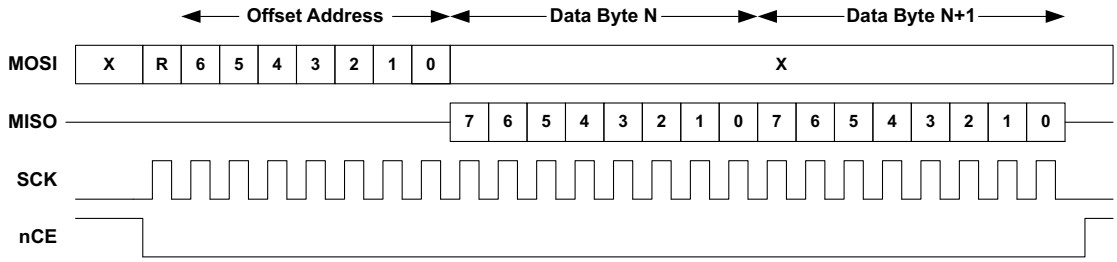


Figure 31. SPI Normal Read Operation

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 slave 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.

12.6.5 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 32. The OFFSET field is not used in this case.

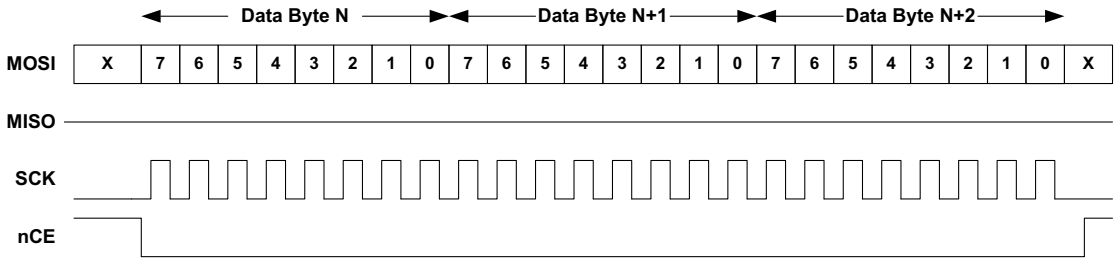


Figure 32. SPI Raw Write Operation

12.6.6 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 33. The OFFSET field is not used in this case.

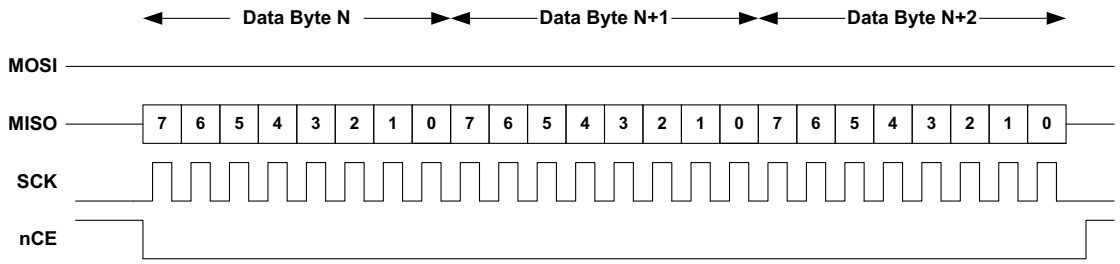


Figure 33. SPI Raw Read Operation

12.6.7 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 Master) 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.

12.6.8 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 34 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.

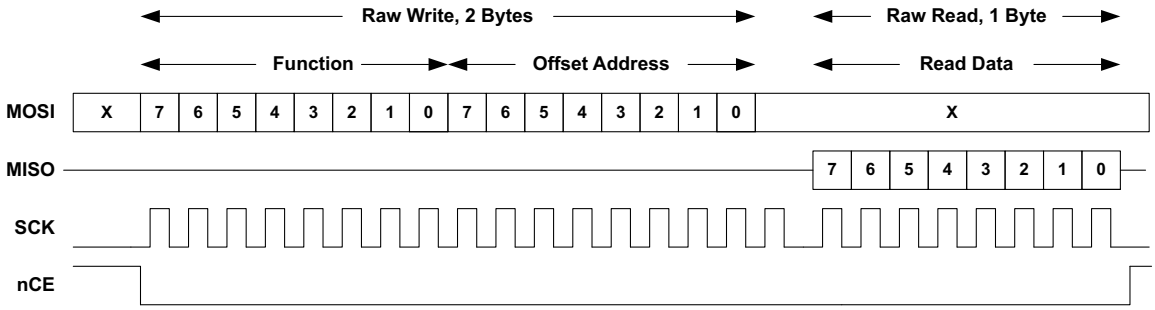


Figure 34. SPI Combined Operation

12.6.9 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 35 shows how these two bits affect the interface signal behavior.

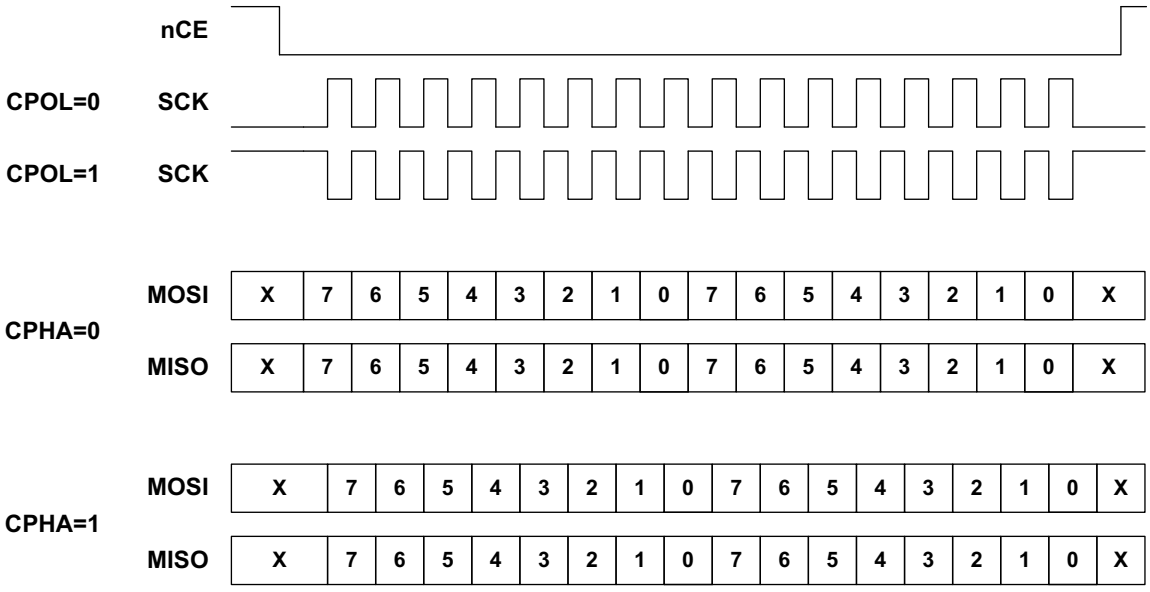


Figure 35. SPI CPOL and CPHA

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 slave 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.

12.7 Bit Orientation

In both I^2C and SPI modes, the I^2C/SPI Master 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.

12.8 SPI Flow Control

The I^2C/SPI Master supports flow control from the slave, 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 I^2C/SPI Master 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 I^2C/SPI Master. 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.

Slave 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 36 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 ½ 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.

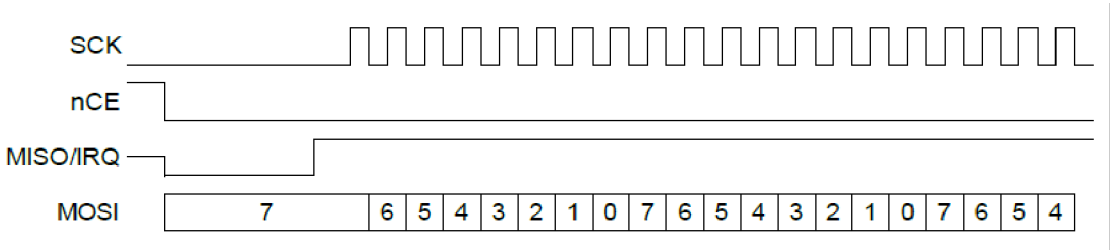


Figure 36. Flow Control at Beginning of a Write Transfer

Figure 37 shows the operation of flow control at the beginning of a raw read transfer. IRQ must be deasserted low within ½ 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.

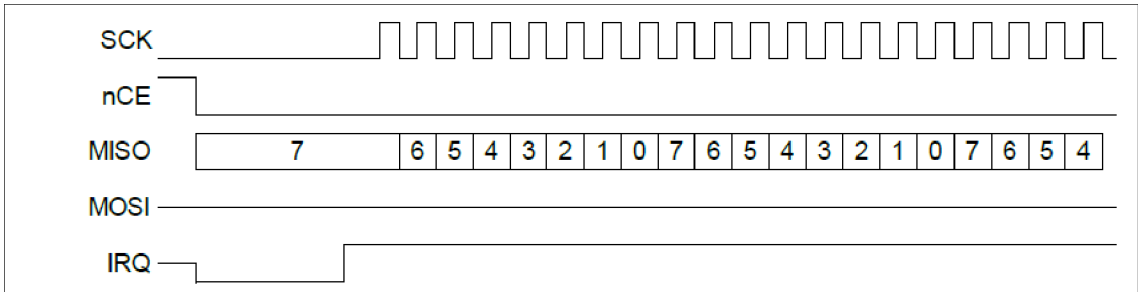


Figure 37. Flow Control at Beginning of a Raw Read Transfer

Figure 38 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.

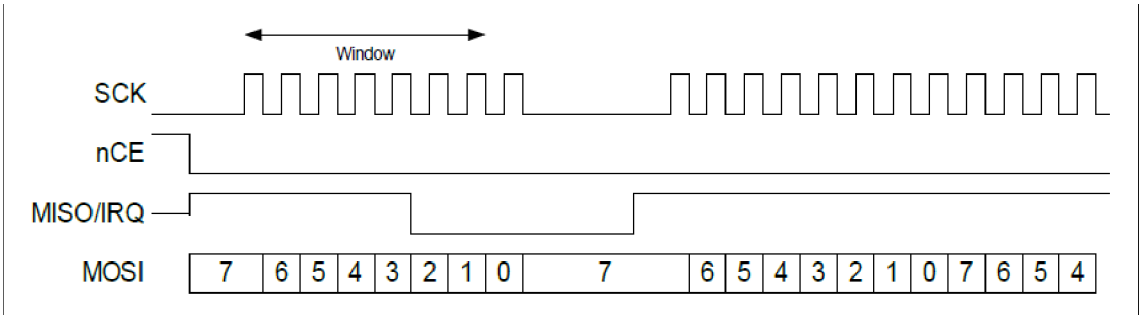


Figure 38. Flow Control in the Middle of a Write Transfer

Figure 39 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.

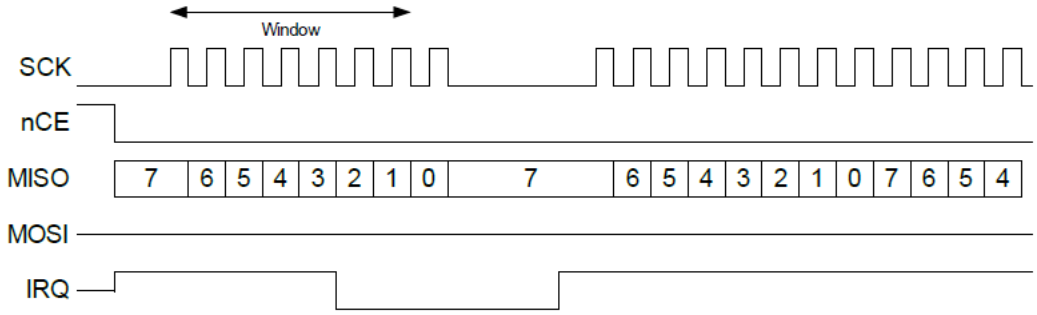


Figure 39. Flow Control in the Middle of a Read Transfer

12.9 Minimizing Power

Each I^2C/SPI Master 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.

13. I²C/SPI Slave (IOS)

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

13.1 Local RAM Allocation

The I²C/SPI Slave (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 Slave 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 (IOSLAVE_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 Slave Read-only starting at any 8-byte segment defined by IOSLAVE_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, IOSLAVE_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 Slave. The RAM area is from the LRAM address calculated from the value in the FIFOMAX field of the FIFOCFG register, IOSLAVE_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 40 below shows the LRAM address mapping between the I/O interface and the AHB.

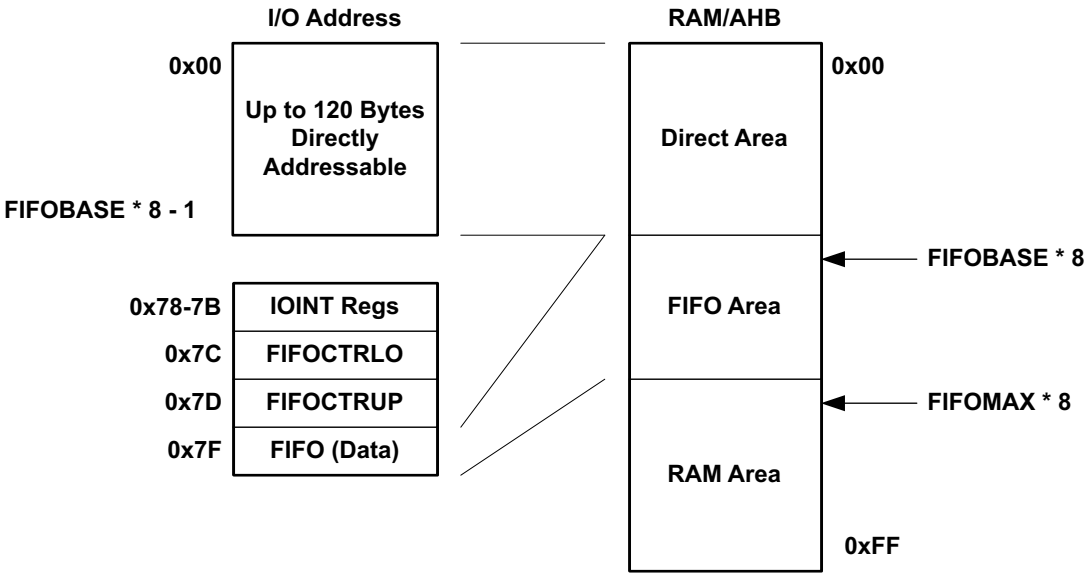


Figure 40. I²C/SPI Slave Module LRAM Addressing in Standard Address Mode

13.2 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 70 in Section 13.2.2 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).

13.2.1 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 IOSLAVE_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 IOSLAVE_FIFOCFG_FIFOBASE = 0xF), as undefined results will occur.

For burst writes > 248 bytes, the end address of the FIFO, as defined by IOSLAVE_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 70 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 Slave Read-Only address mapping, as specified by the IOSLAVE_ FIFOCFG_ROBASE setting, is not supported in Host Burst Write Mode (IOSLAVE_CFG_WRAPPTR = 1).

13.2.2 Mapping of Direct Area Access Interrupts

Table 70 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 70: Mapping of Direct Area Access Interrupts 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
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 IOSLAVE_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 (FIFOBASE * 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.

13.3 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 I^2C/SPI Slave on demand. Several hardware features support this operation.

Figure 41 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 Slave. The IOSLAVE_FIFOPTR_FIFOPTR register points to the next byte to be read on the I/O interface. IOSLAVE_FIFOPTR_FIFOSIZ holds the current number of valid bytes in the FIFO on the I^2C/SPI Slave, and FIFOCTR holds the total number of bytes in the FIFO. The value in IOSLAVE_FIFOCTR may be read indirectly at any time by the Host processor via the FIFOCTRUP_FIFOCTRLO 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 FIFOCTR value via FIFOCTRUP_FIFOCTRLO is not synchronized to the write clock. So if the host read happens during a FIFOCTR update (either through a read-modify-write of FIFOCTR register or an automatic update because of a write to the FIFOINC register by the Slave CPU), it is possible for the count value to be out of sync, impacting the value read in either or both the upper (FIFOCTRUP) and lower (FIFOCTRLO) bytes. This is a very rare case, but proper code would have the host read the two registers for the FIFOCTR value multiple times until consecutive reads are the same.

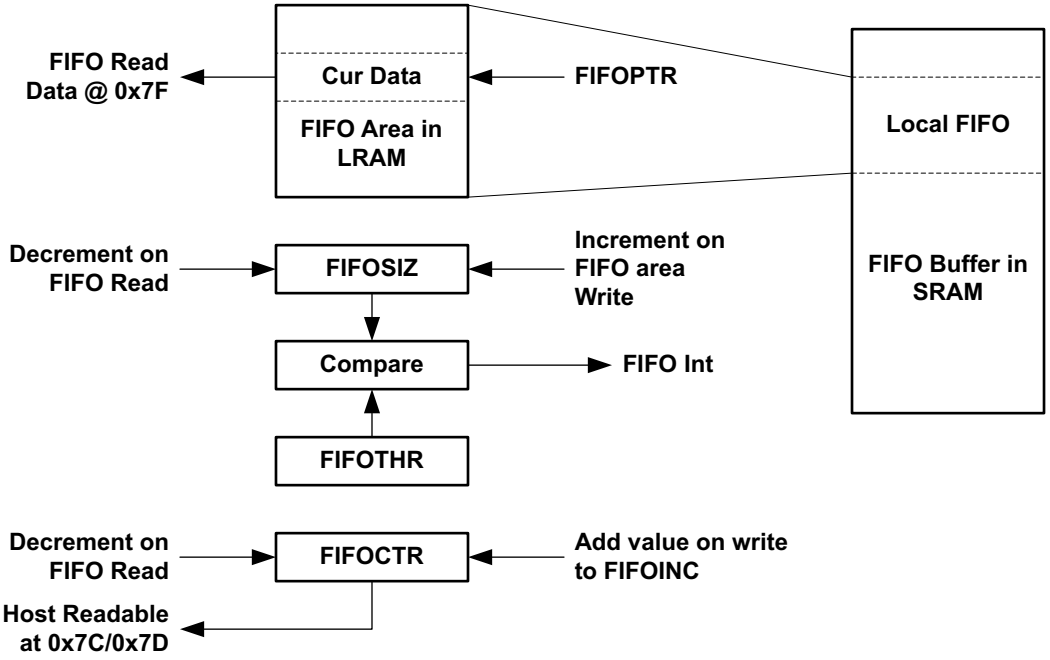


Figure 41. I²C/SPI Slave Module FIFO

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 Slave interrupt block and in the Host interrupt block.

When FIFOSIZ drops below the configured threshold IOSLAVE_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 Slave 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 IOSLAVE_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 Slave, 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.

13.4 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 IOSLAVE_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 IOSLAVE_FUPD_FIFOUPD 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 FIFOUPD bit is cleared.

If the Host attempts a FIFO read operation while the FIFOUPD 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.

13.5 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 Slave 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 71 summarizes these I/O interface interrupts and how they can be controlled and read.

Table 71: 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 Slave 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.

13.6 Command Completion Interrupts

Four interrupts in the I2C/SPI Slave 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.

13.7 Host Address Space and Registers

The Host of the I/O interface can access 128 bytes in the I^2C/SPI Slave in either I^2C 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 I^2C/SPI Slave. The R/W indicator is referring to accesses from the Host.

13.8 I2C Interface

The MCU I2C Slave interface operates as a standard slave. The device is accessed at an address configured in the IOSLAVE_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 IOSLAVE_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 "master". The devices that are controlled by the master are called "slaves". The SoC's I2C Slave is always a slave 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 42) and are described in the following sections.

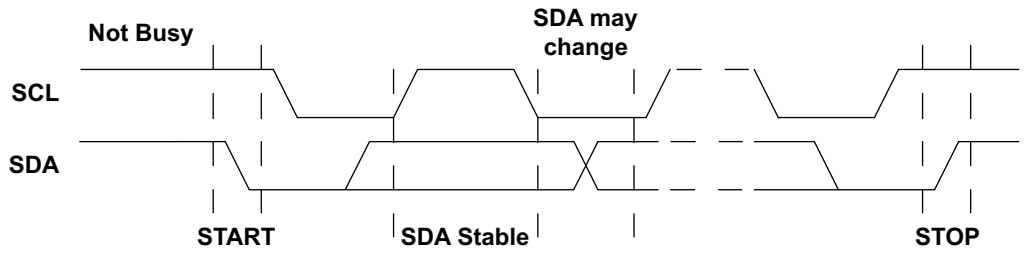


Figure 42. Basic I2C Conditions

13.8.1 Bus Not Busy

Both SDA and SCL remain high.

13.8.2 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.

13.8.3 Stop Data Transfer

A change in the state of SDA from low to high, while SCL is high, defines the STOP condition.

13.8.4 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.

13.8.5 Acknowledge

Each byte of eight bits is followed by one Acknowledge (ACK) bit as shown in Figure 43. This Acknowledge bit is a low level driven onto SDA by the receiver, whereas the master generates an extra ACK related SCL pulse. A slave receiver which is addressed is obliged to generate an Acknowledge after the reception of each byte. Also, on a read transfer a master receiver must generate an Acknowledge after the reception of each byte that has been clocked out of the slave 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 master receiver must signal an end-of-data to the slave transmitter by not generating an Acknowledge (a NAK) on the last byte that has been clocked out of the slave. In this case, the transmitter must leave the data line high to enable the master to generate the STOP condition.

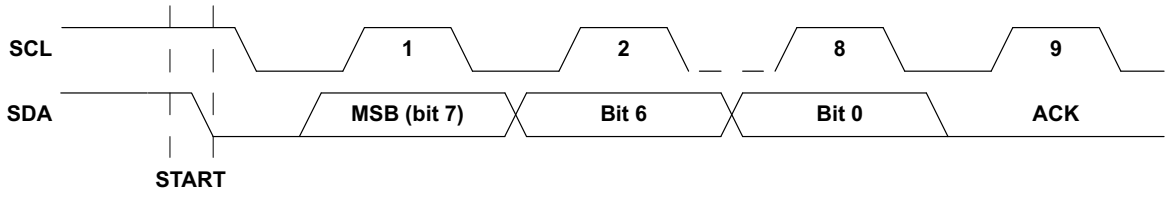


Figure 43. I²C Acknowledge

13.8.6 Address Operation

In I²C mode, the I²C/SPI Slave supports either 7-bit or 10-bit addressing, selected by the 10BIT bit in the IOSCFG Register. Figure 44 shows the operation in 7-bit mode in which the master 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.

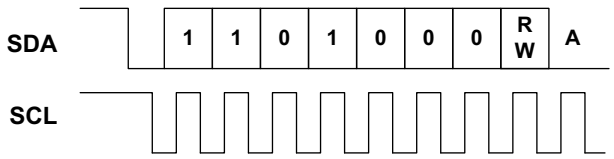


Figure 44. I²C 7-bit Address Operation

Figure 45 shows the operation with which the master 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 Slave 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 Slave ignores all other address values and does not respond with an ACK.

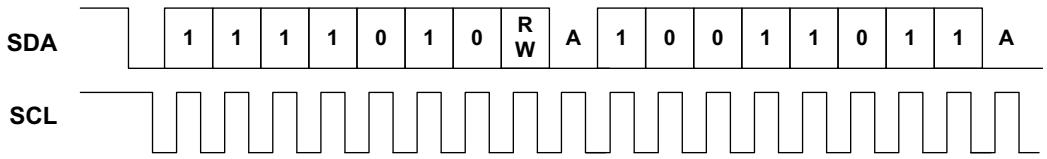


Figure 45. I2C 10-bit Address Operation

13.8.7 Offset Address Transmission

If the RW bit of the Address Operation indicates a write, the next byte transmitted from the master is the Offset Address as shown in Figure 46. This value is loaded into the Address Pointer of the I2C/SPI Slave.

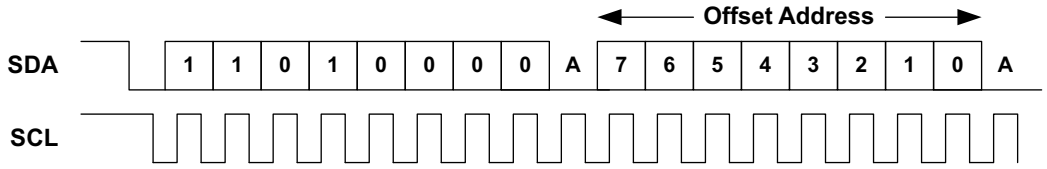


Figure 46. I2C Offset Address Transmission

13.8.8 Write Operation

In a write operation the master transmitter transmits to the MCU slave receiver. The Address Operation has a RW value of 0, and the second byte contains the Offset Address as in Figure 46. 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 47. Note that if the Address Pointer is at 0x7F, it will not increment on the write.

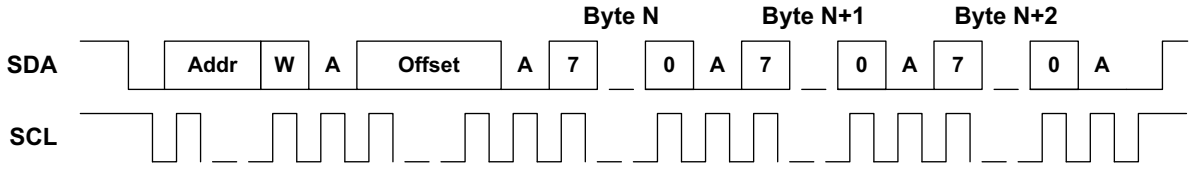


Figure 47. I2C Write Operation

13.8.9 Read Operation

In a read operation, the master 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 48 shows this transaction beginning with a RESTART condition, although a STOP followed by a START may also be used. After the address operation, the slave 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 master receiver responds with a NAK and a STOP to complete the operation. Because the Address Pointer holds a valid register address, the master 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.

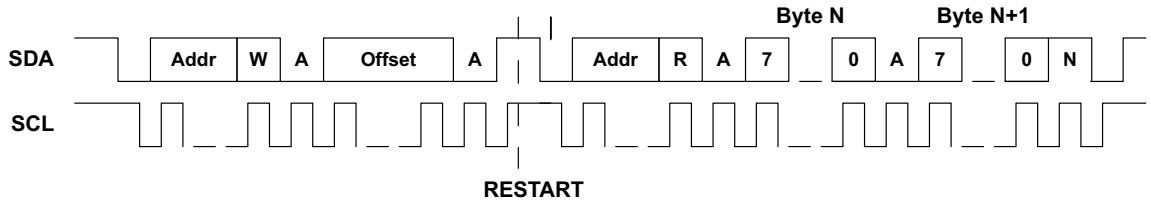


Figure 48. I2C Read Operation

13.8.10 General Address Detection

The I2C/SPI Slave 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 IOSLAVE_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 Slave.

13.9 SPI Interface

The I2C/SPI Slave 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 Slave may be connected to a master 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 "master". The devices that are controlled by the master are called "slaves". The I2C/SPI Slave SPI Slave is always a slave device.

The nCE input is used to initiate and terminate a data transfer. The SCL input is used to synchronize data transfer between the master and the slave devices via the MOSI (master to slave) and MISO (slave to master) lines. The SCL input, which is generated by the master, is active only during address and data transfer to any device on the SPI bus.

The I2C/SPI Slave 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.

13.9.1 Write Operation

Figure 49 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 master 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 Slave.

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 master 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 Slave. Note also that if the Address Pointer is 0x7F, it does not increment on the read.

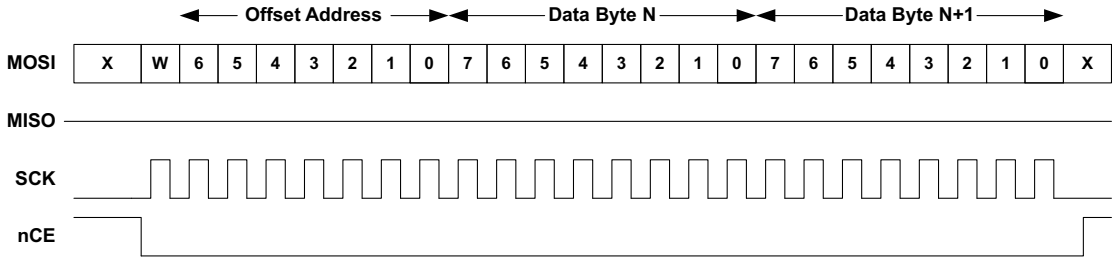


Figure 49. SPI Write Operation

13.9.2 Read Operation

Figure 50 shows a read operation. The address is transferred from the master to the slave 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 I^{2}C/SPI Slave 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 master brings the nCE line high. Note that if the Address Pointer is 0x7F, it does not increment on the read.

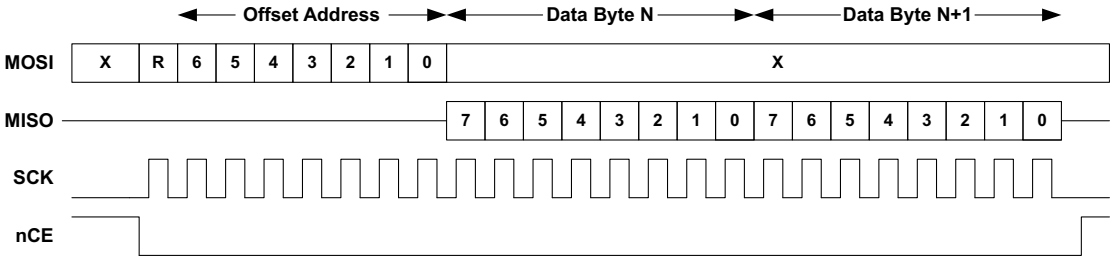


Figure 50. SPI Read Operation

13.9.3 Configuring 3-wire vs. 4-wire SPI Mode

The I^{2}C/SPI Slave 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 I^{2}C/SPI Slave itself.

13.9.4 SPI Polarity and Phase

The I^{2}C/SPI Slave supports all combinations of CPOL (clock polarity) and CPHA (data phase) in SPI mode. Figure 51 shows how these two bits affect the interface signal behavior.

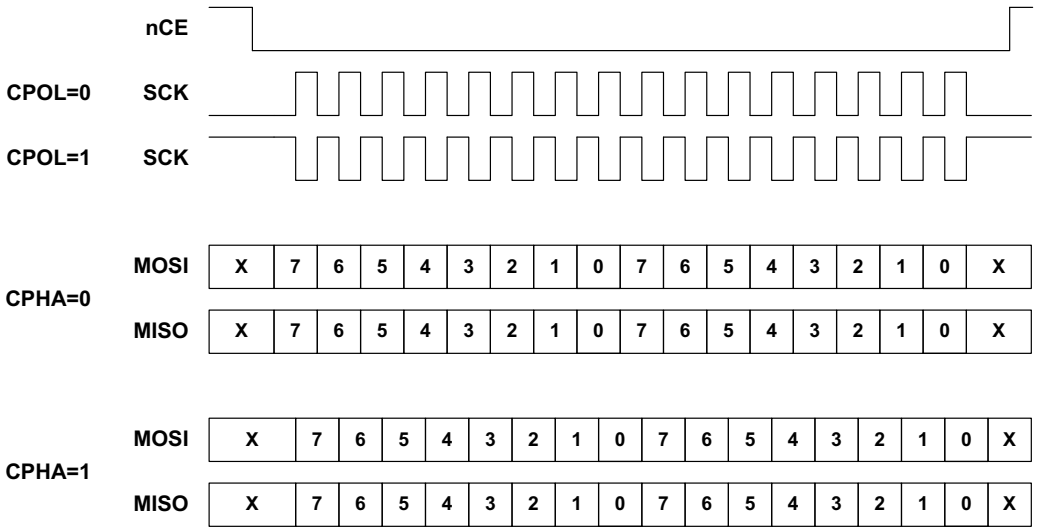


Figure 51. SPI CPOL and CPHA

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 I^2C/SPI Slave has only a single SPOL bit to control the polarity. If CPOL = CPHA, IOSLAVE_IOSCFG_SPOL must be set to 0. If CPOL ≠ CPHA, SPOL must be set to 1.

13.10 Bit Orientation

In both I^2C and SPI modes, the I^2C/SPI Slave supports data transmission either LSB first or MSB first as configured by the IOSLAVE_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.

13.11 Wakeup Using the I^2C/SPI Slave

The I^2C/SPI Slave can continue to operate even if the SoC's CPU is in Sleep or Deep Sleep mode. The hardware will enable and disable the I^2C/SPI Slave 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 master attempts to access the I^2C/SPI Slave. 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 µs, and is initiated by nCE going low in SPI mode or by the detection of a START in I^2C 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 I^2C applications, the time delay is typically not relevant. At the fastest system clock of 1 MHz, the master must transfer 9 bits of address plus 9 bits of offset before any FIFO access can occur, and that is a minimum of 18 µs. The clocks will have started prior to that point in every case.

For SPI applications with fast interface clocks (faster than 1 MHz), the master must be programmed to pull nCE low at least 10 µs prior to sending the first clock. If a master is unable to control the timing of nCE in this way, then there are other options for ensuring that master commands/data are received reliably by the slave 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 configure to wake up the slave 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 slave 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 I^2C/SPI Slave clock will start immediately when nCE goes low.

13.12 Host Side Address Space and Register

13.12.1 Host Address Space and Registers

The Host of the I/O interface can access 128 bytes in the I^{2}C/SPI Slave in either I^{2}C 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 I^{2}C/SPI Slave. The R/W indicator refers to accesses from the Host.

13.12.1.1 HOST_IER Register

Host Interrupt Enable

OFFSET: 0x78

This register enables the FIFO read interrupts.

Table 72: HOST_IER Register

07	06	05	04	03	02	01	00
FUNDFLEN	RDERREN	SWINT5EN	SWINT4EN	SWINT3EN	SWINT2EN	SWINT1EN	SWINT0EN

Table 73: 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

13.12.1.2 HOST_ISR Register

Host Interrupt Status Register

OFFSET: 0x79

The host uses this register to read interrupt status.

Table 74: HOST_ISR Register

07	06	05	04	03	02	01	00
FUNDFLSTAT	RDERRSTAT	SWINT5STAT	SWINT4STAT	SWINT3STAT	SWINT2STAT	SWINT1STAT	SWINT0STAT

Table 75: 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 FIFOUPD 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.

13.12.1.3 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 76: HOST_WCR Register

07	06	05	04	03	02	01	00
FUNDFLWC	RDERRWC	SWINT5WC	SWINT4WC	SWINT3WC	SWINT2WC	SWINT1WC	SWINT0WC

Table 77: 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

13.12.1.4 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 78: HOST_WCS Register

07	06	05	04	03	02	01	00
FUNDFLWS	RDERRWS	SWINT5WS	SWINT4WS	SWINT3WS	SWINT2WS	SWINT1WS	SWINT0WS

Table 79: 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

13.12.1.5 FIFOCTRLO Register

FIFOCTR Low Byte

OFFSET: 0x7C

This register allows the host to read the lower eight bits of the FIFOCTR register.

Table 80: FIFOCTRLO Register

07	06	05	04	03	02	01	00
FIFOCTRLO							

Table 81: FIFOCTRLO Register Bits

Bit	Name	Reset	RW	Description
7:0	FIFOCTRLO	0x0	RO	Reads the lower eight bits of FIFOCTR

13.12.1.6 FIFOCTRUP Register

FIFOCTR Upper Byte

OFFSET: 0x7D

This register allows the host to read the upper two bits of the FIFOCTR register.

Table 82: FIFOCTRUP Register

07	06	05	04	03	02	01	00
RSVD						FIFOCTRUP	

Table 83: FIFOCTRUP Register Bits

Bit	Name	Reset	RW	Description
1:0	FIFOCTRUP	0x0	RO	Reads the upper two bits of FIFOCTR

13.12.1.7 FIFO Register

FIFO Read Data

OFFSET: 0x7F

Read this register for FIFO data.

Table 84: FIFO Register

07	06	05	04	03	02	01	00
FIFO							

Table 85: FIFO Register Bits

Bit	Name	Reset	RW	Description
7:0	FIFO	0x0	RO	Reads the top byte of the FIFO

14. Universal Asynchronous Receiver/Transmitter (UART)

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

14.1 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. A higher UART_clk frequency can produce more precise 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.

14.2 Configuration

The UART Register Block may be set 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 is configured using the integer UART_IBRD and UART_FBRD registers. The correct values for UART_IBRD and UART_FBRD may be determined according to the following equation:

$F_{UART}/(16 \cdot BR) = IBRD + FBRD$

F_{UART} is the frequency of the UART clock. BR is the desired baud rate. IBRD is the integer portion of the baud rate divisor. FBRD is the fractional portion of the baud rate divisor.

The UART Module supports independent CTS and RTS hardware flow control. All flow control configuration may be set using the UART_CR register.

14.3 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.

15. Universal Serial Bus (USB)

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

15.1 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.

The MCU 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.

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 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_SoftConn bit. When the SoftConn 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 SoftConn 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.

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.

NOTE
Due to the absence of DMA to/from the USB interface, USB high-speed (HS) mode is feasible on an Apollo4 family SoC only in a use case where the CPU can be completely dedicated to and fully occupied by the USB task. HS mode generates 60 MB/s traffic which uses the majority of the CPU bandwidth. One such use case which could utilize HS mode effectively is when the CPU does nothing but data transfer to/from the USB endpoint, such as when programming eMMC over USB.

15.2 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.

15.3 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.

15.4 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.

15.5 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 52.

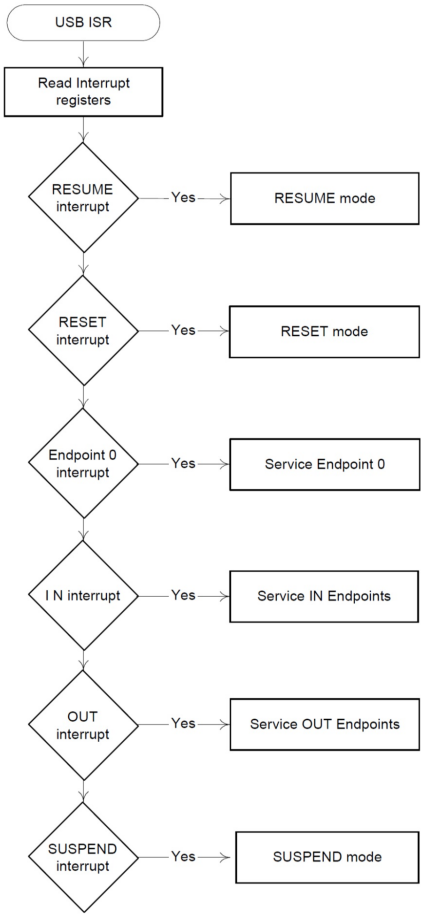


Figure 52. USB Interrupt Service Routine

15.6 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 86 below lists the functionality for these fields. See the field descriptions in the IDX0 register for more information.

Table 86: 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 86: 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 generate when this bit is set (if enabled). CPU clears this bit by setting the ServicedOutPktRdy 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 generate (if enabled) when the bit is cleared.

15.7 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.

15.7.1 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.

15.7.2 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.

15.8 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.

15.8.1 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.

15.8.2 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.

15.8.3 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.

15.9 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_SOFE 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 µs), 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.

15.10 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 87, 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 87: USB Controller Register Map Additions for Dynamic FIFO Sizing

Address	Register_Field	Description
1A	IDX2_INFIFOSZ	IN Endpoint FIFO size
1B	IDX2_OUTFIFOSZ	OUT Endpoint FIFO size
1C, 1D	FIFOADD_INFIFOADD	IN Endpoint FIFO address
1E, 1F	FIFOADD_OUTFIFOADD	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 x 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.

15.11 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.

15.11.1 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.

15.11.2 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 th 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.

15.11.3 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.

15.11.4 Endpoint 0 States

As illustrated in Figure 53, 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.

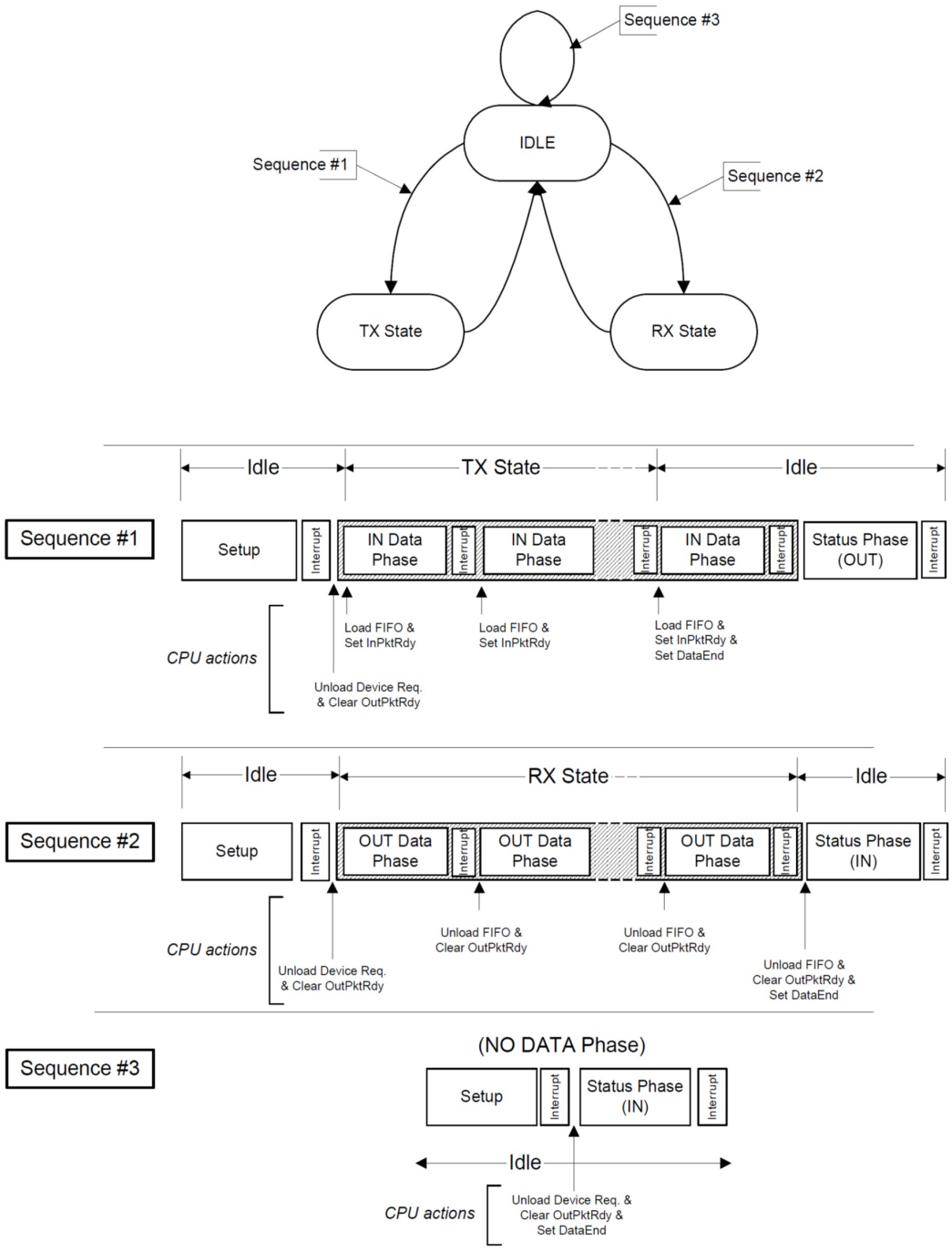


Figure 53. Endpoint 0 States

15.11.5 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 54.

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.

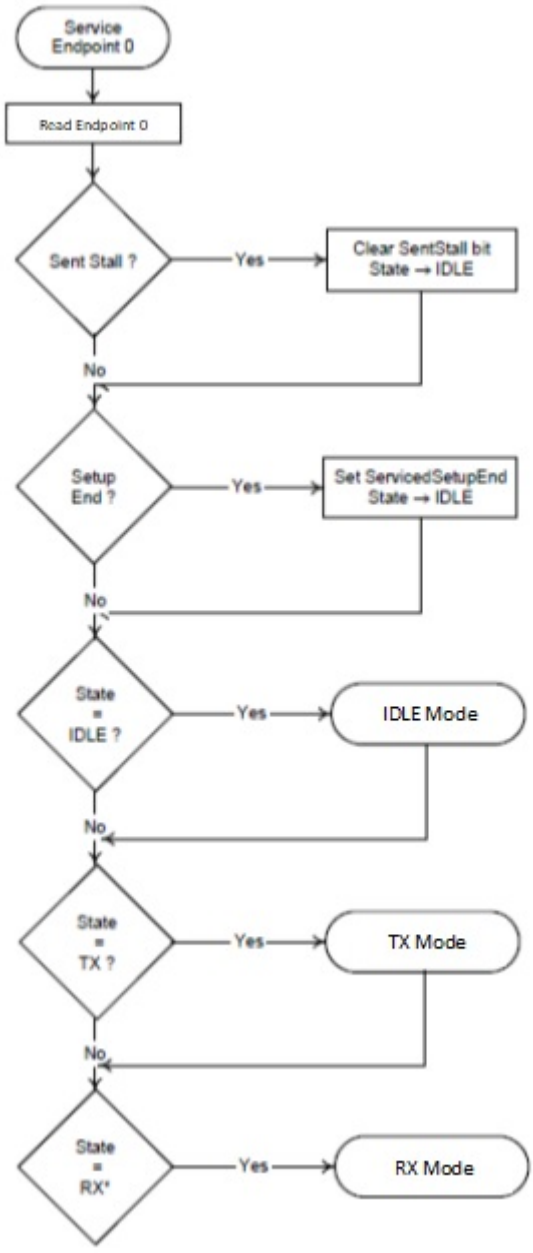


Figure 54. Endpoint 0 Service Routine

15.11.5.1 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 55 below.

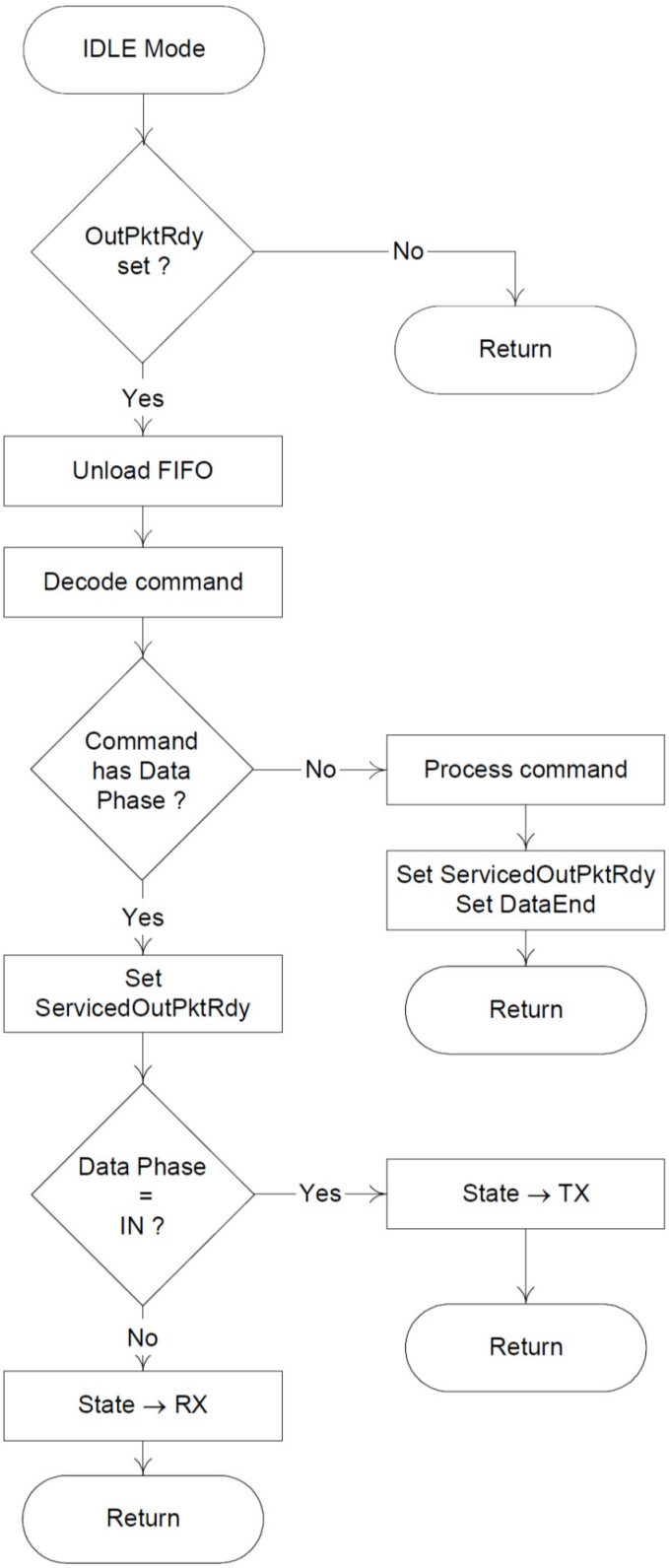


Figure 55. SETUP Phase of Control Transfer

15.11.5.2 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_MAX-PAYLOAD).
- 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.

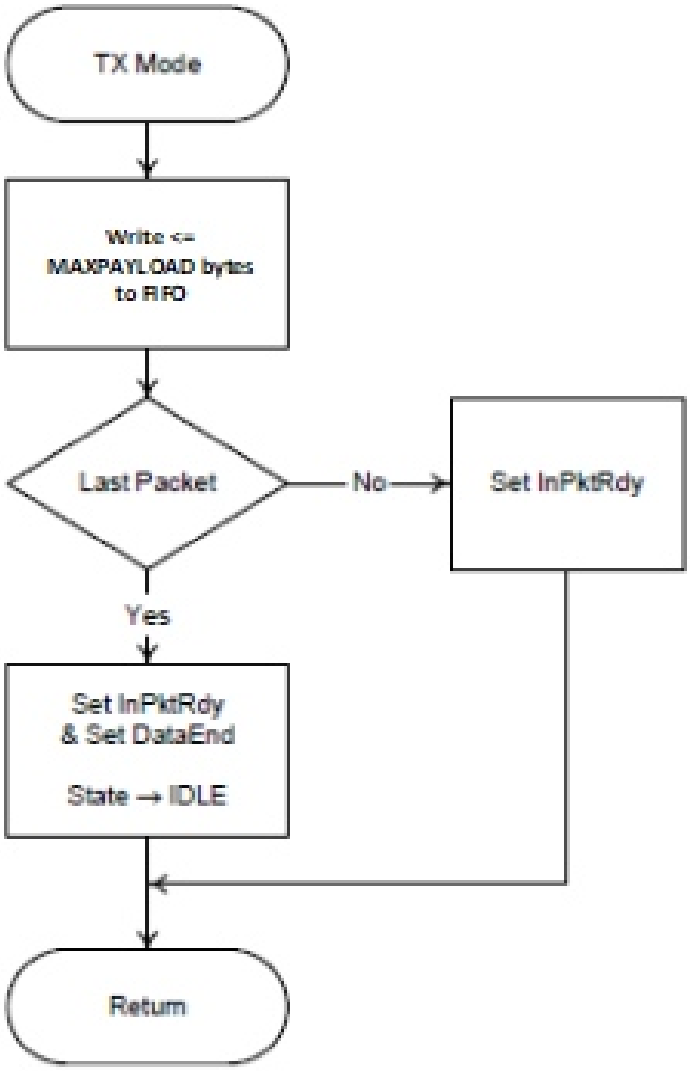


Figure 56. IN Data Phase for Control Transfer

15.11.5.3 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.

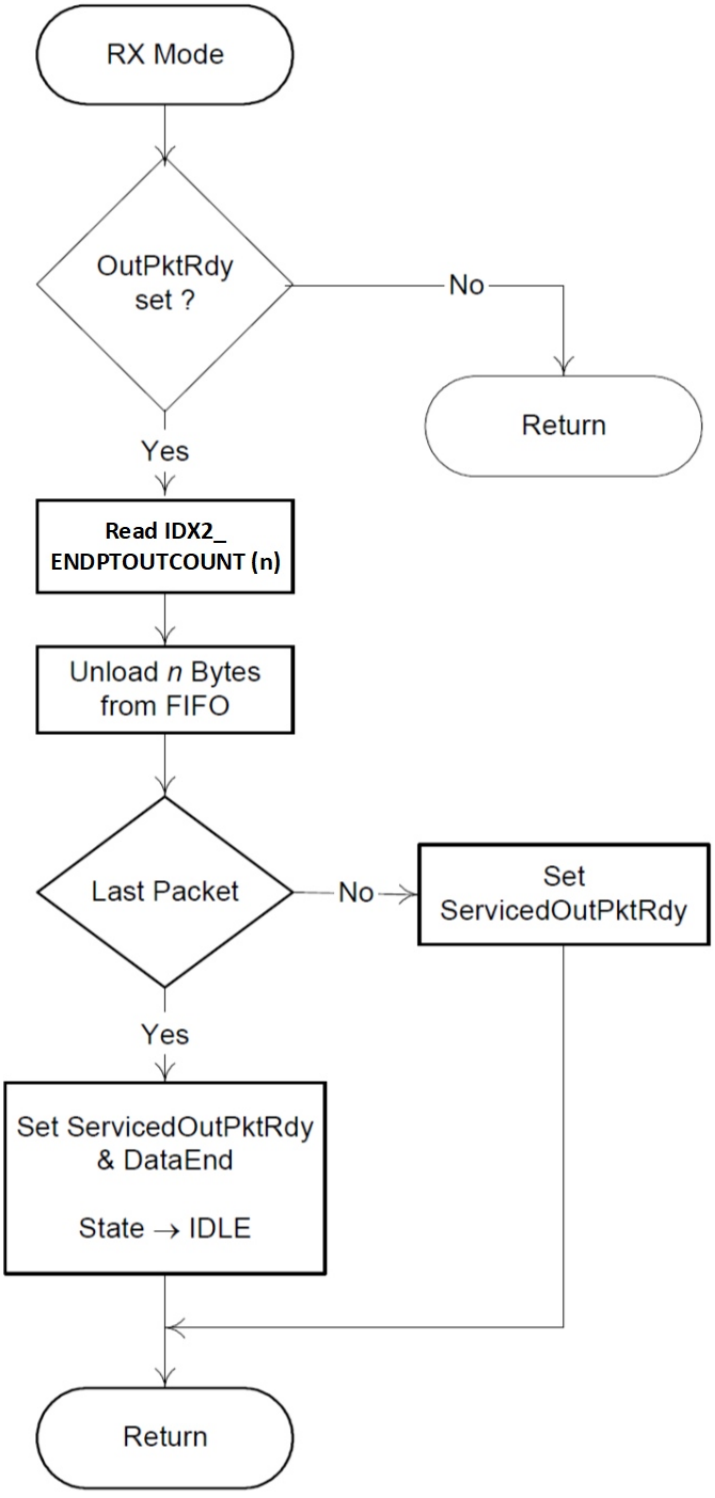


Figure 57. OUT Data Phase for Control Transfer

15.11.6 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.

15.12 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 155.) 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.

15.12.1 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.

15.12.2 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.

15.12.3 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.

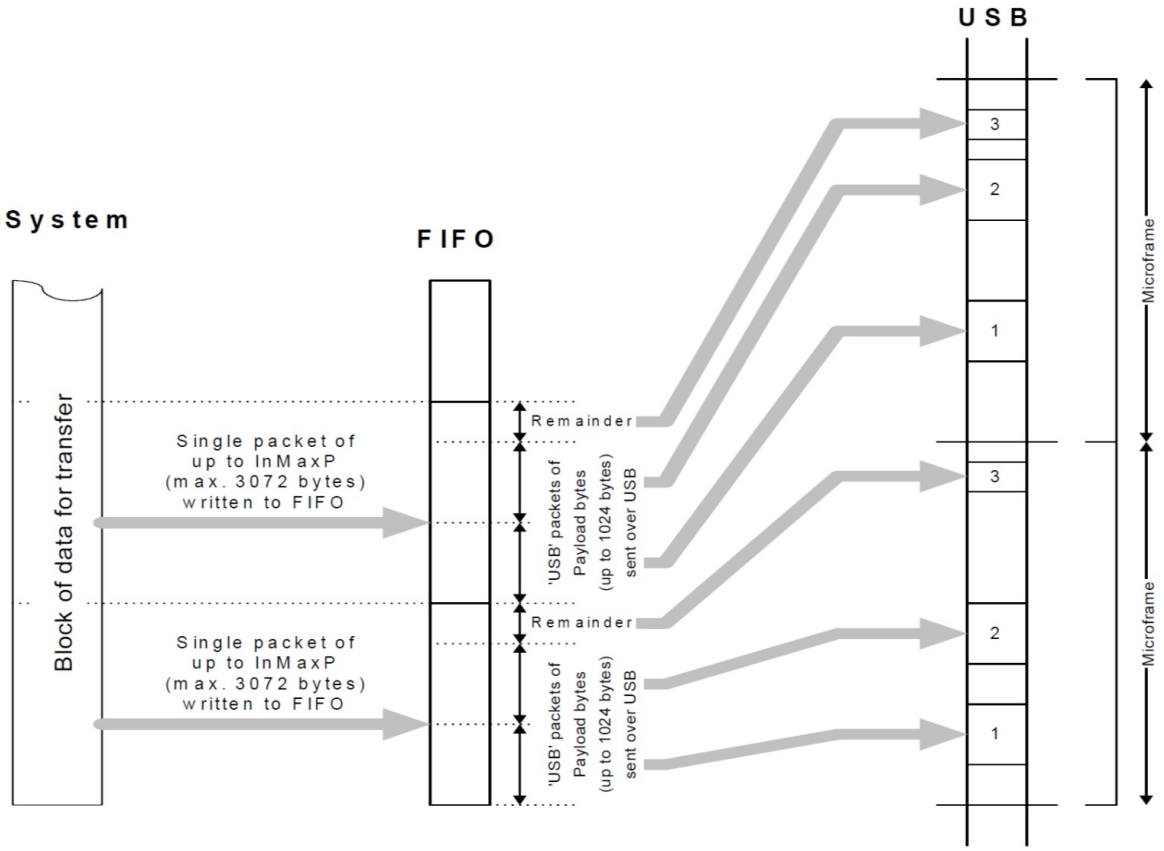


Figure 58. High-speed Isochronous IN Endpoint Transmission

15.12.4 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.

15.13 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.

15.13.1 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).

Table 88: Unloaded Packet Sizes < MAXPAYLOAD Requiring Manual Clear of OutPktRdy

Remainder (MAXPAYLOAD/4)	Actual Bytes Read	Packet Sizes Which Will Clear OutPktRdy
0 (i.e., MAXPAYLOAD = 64 bytes)	MAXPAYLOAD	MAXPAYLOAD, MAXPAYLOAD-1, MAXPAYLOAD-2, MAXPAYLOAD-3
3 (i.e., MAXPAYLOAD = 63 bytes)	MAXPAYLOAD+1	MAXPAYLOAD, MAXPAYLOAD-1, MAXPAYLOAD-2
2 (i.e., MAXPAYLOAD = 62 bytes)	MAXPAYLOAD+2	MAXPAYLOAD, MAXPAYLOAD-1
1 (i.e., MAXPAYLOAD = 61 bytes)	MAXPAYLOAD+3	MAXPAYLOAD

15.13.2 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_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.

15.13.3 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_PKTSPLITOPTION 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 is 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.

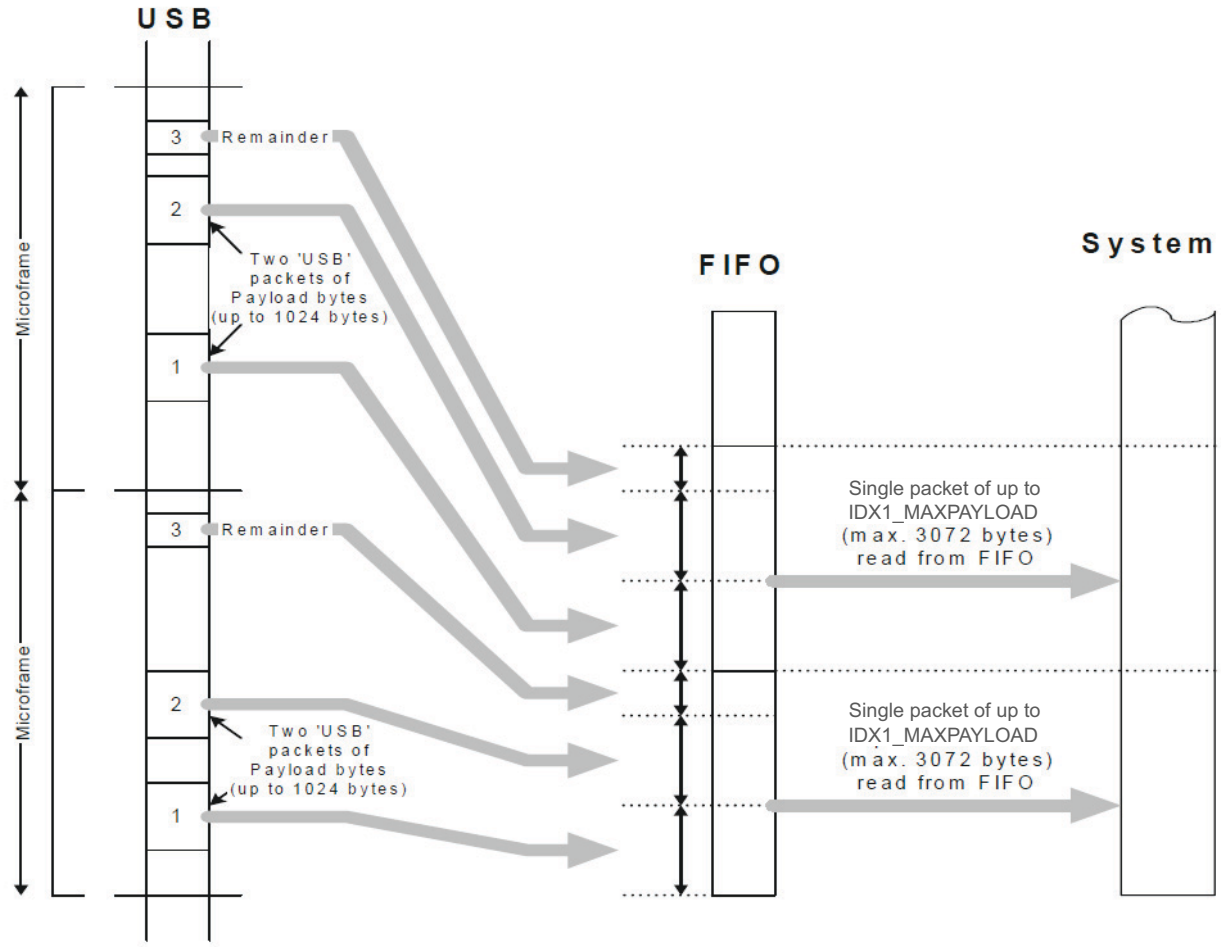


Figure 59. Packets Sent per Microframe

Table 89: 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

15.13.4 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 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).

15.14 Bulk Transactions

15.14.1 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.

15.14.1.1 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 90.

Table 90: 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.

15.14.1.2 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 × payload where IDX0_PKTSPLITOPTION = 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.

15.14.1.3 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.

15.14.2 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.

15.14.2.1 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 91. Bit D0 is unused/Read-only.

Table 91: 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.

15.14.2.2 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 × wMaxPacketSize where IDX1_PKTSPLITOPTION = 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.

15.14.2.3 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.

15.14.3 Interrupt Transactions

15.14.3.1 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.

15.14.3.2 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.

15.15 Isochronous Transactions

15.15.1 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.

15.15.1.1 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 92. (Bit D0 is unused):

Table 92: 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.

15.15.1.2 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.

15.15.1.3 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.

15.15.2 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.

15.15.2.1 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 93.

Table 93: 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.

15.15.2.2 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).

15.15.2.3 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.

15.15.2.4 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.

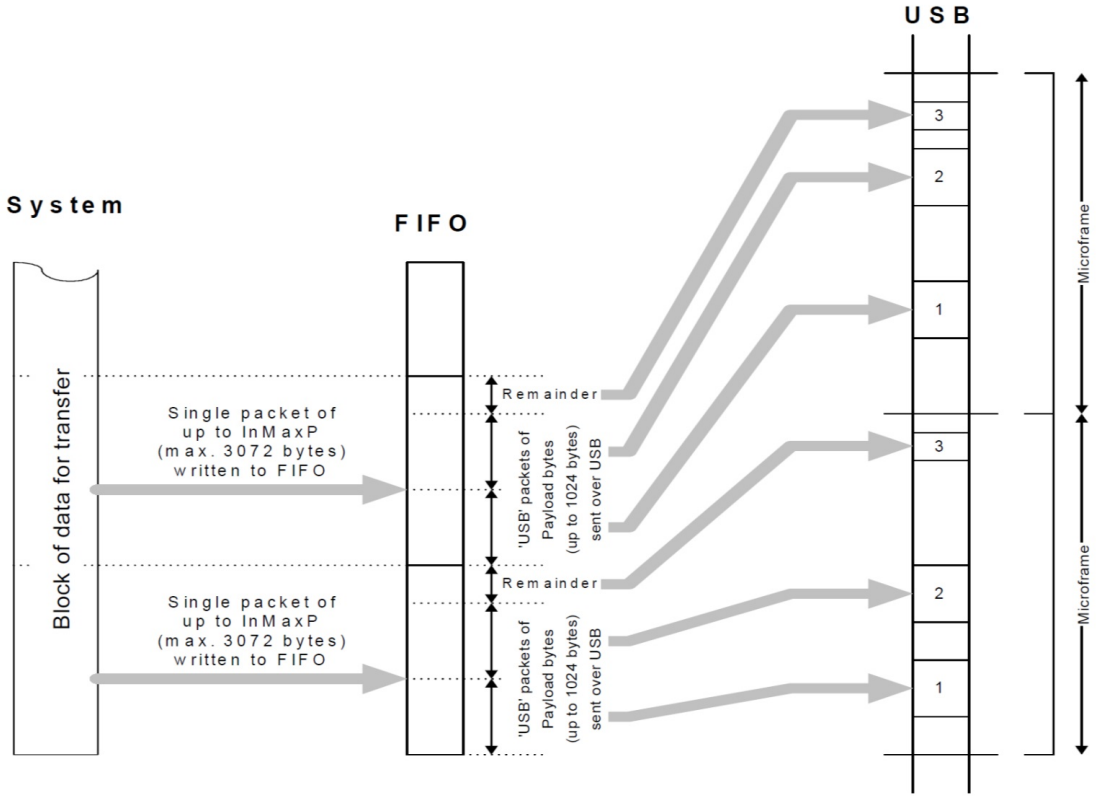


Figure 60. High-Speed High-Bandwidth IN Endpoint

15.15.3 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.

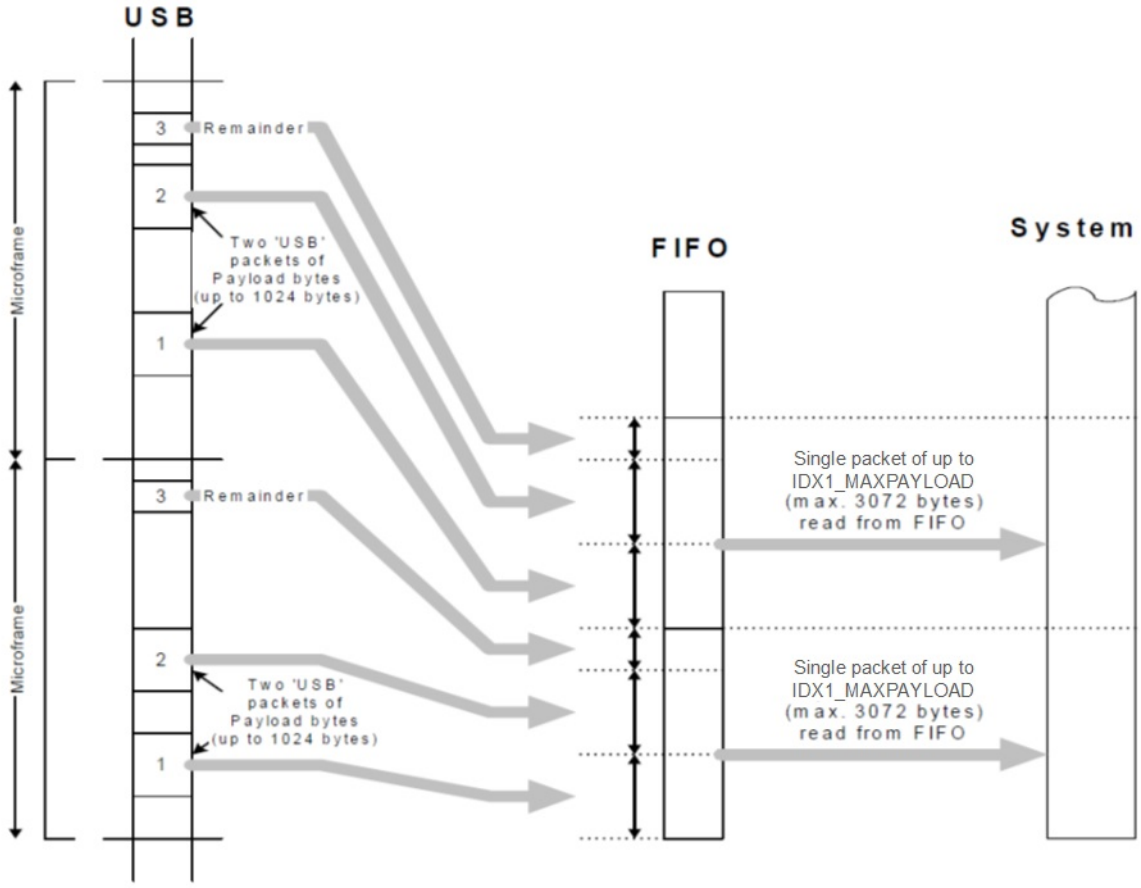


Figure 61. High-speed High-bandwidth OUT Endpoint

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 IDX1_IncompRx bit 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 IDX1_DisNye bit 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.

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 94: 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

15.16 Transaction Flows

In the following transaction flows, host actions are shown against a white background, and the USB Controller actions are shown shaded.

15.16.1 Control Transactions

15.16.1.1 Setup Phase

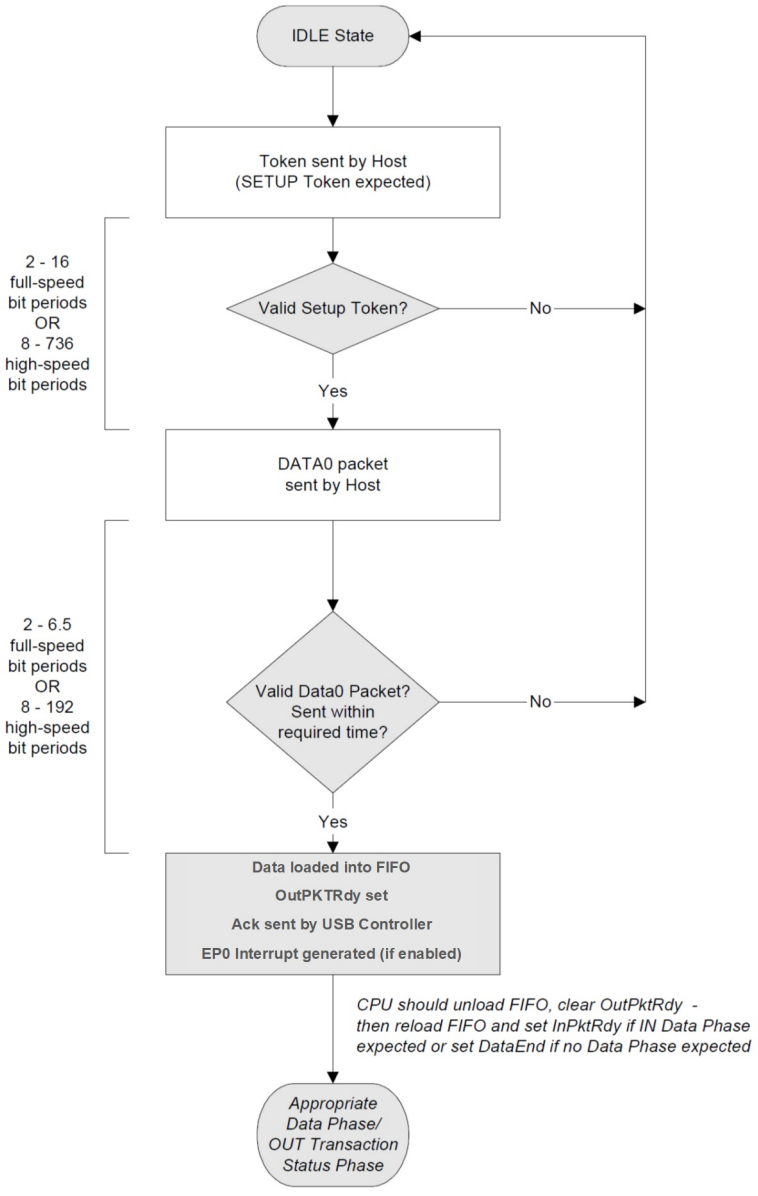


Figure 62. Setup Phase Transaction

15.16.1.2 IN Data Phase

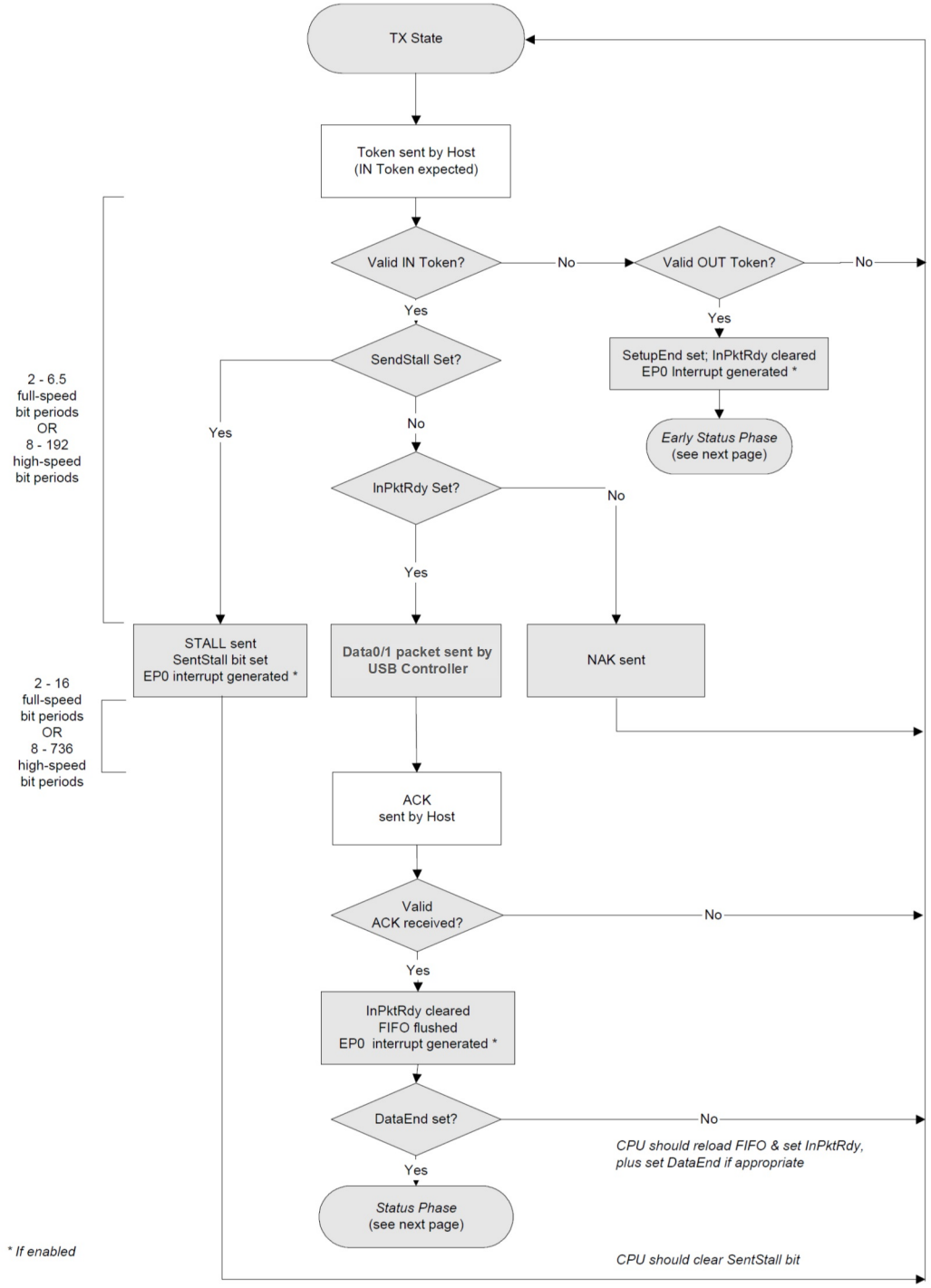


Figure 63. IN Data Phase Transaction

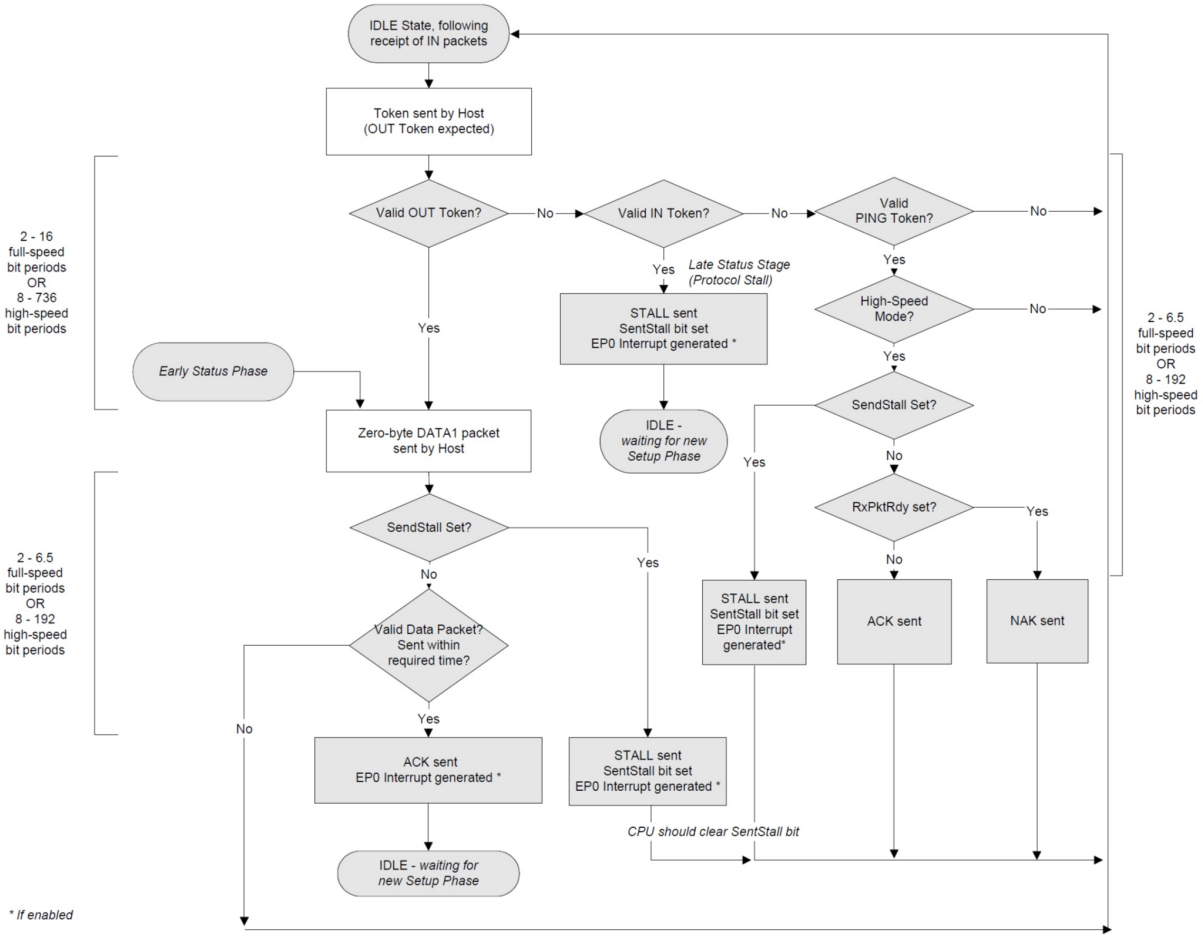


Figure 64. Status Phase Following IN Data Phase Transaction

15.16.1.3 OUT Data Phase

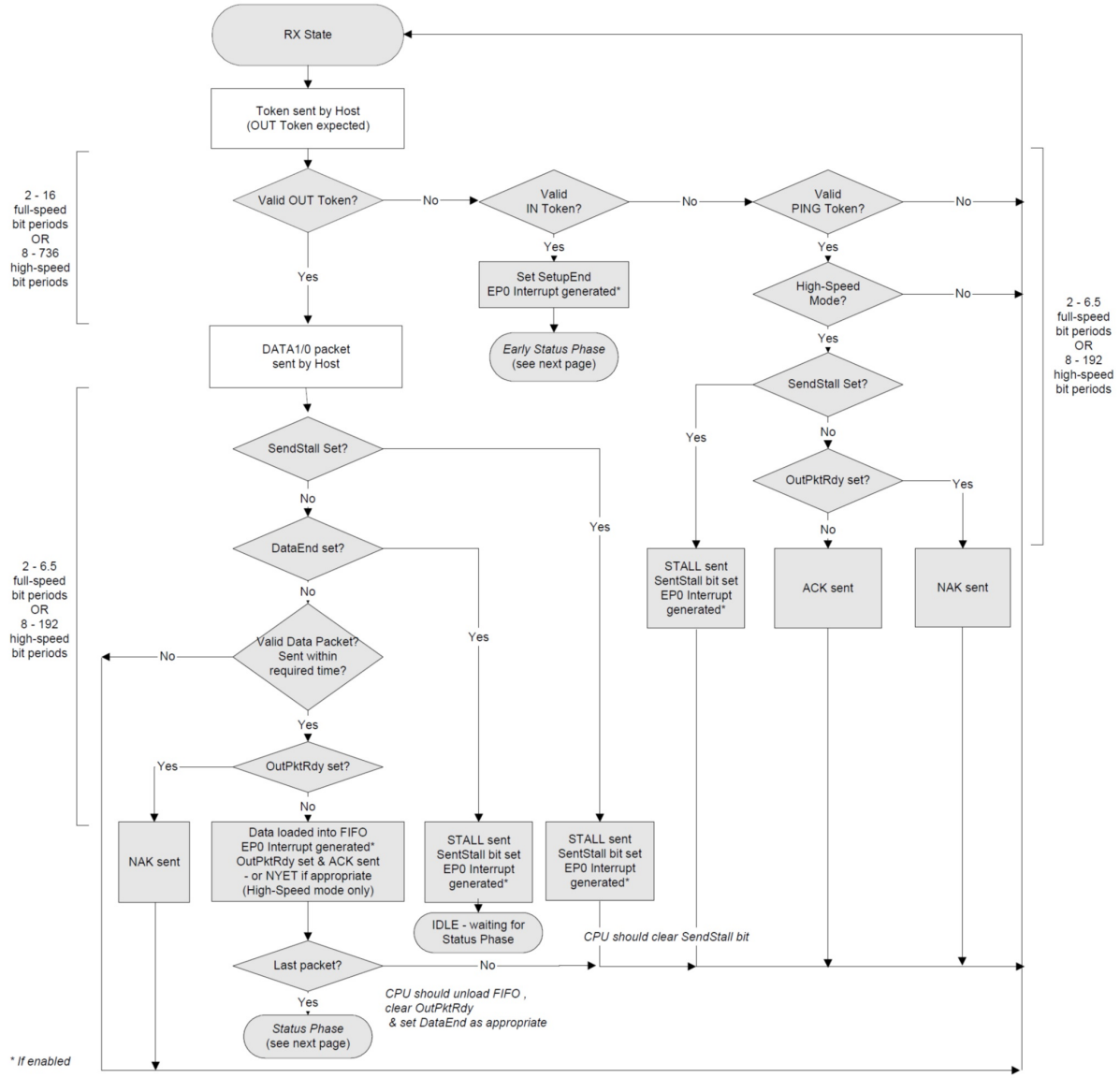


Figure 65. OUT Data Phase Transaction

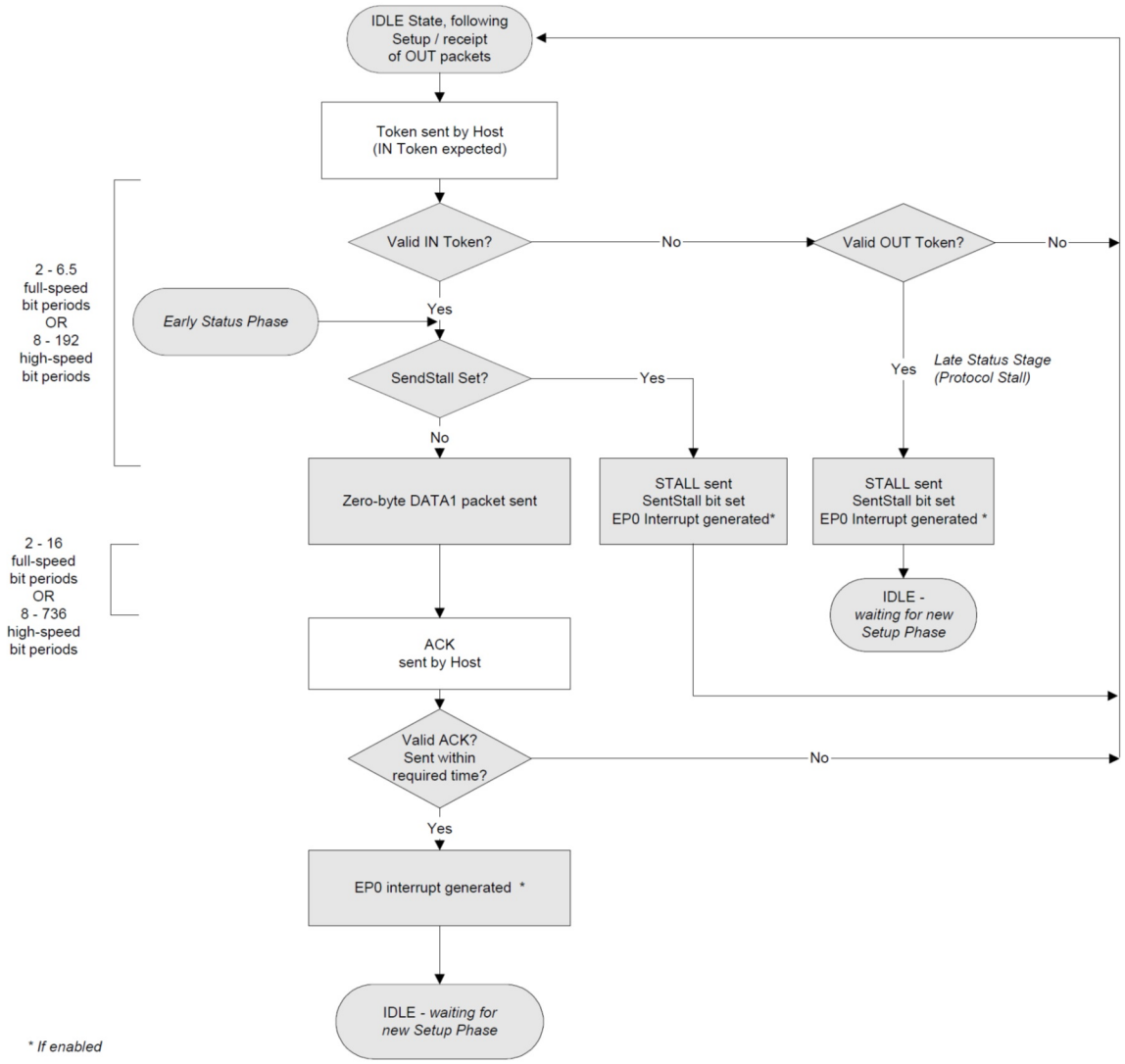


Figure 66. Status Phase Following OUT Data Phase Transaction

15.16.2 Bulk/Interrupt Transactions

15.16.2.1 IN Transactions

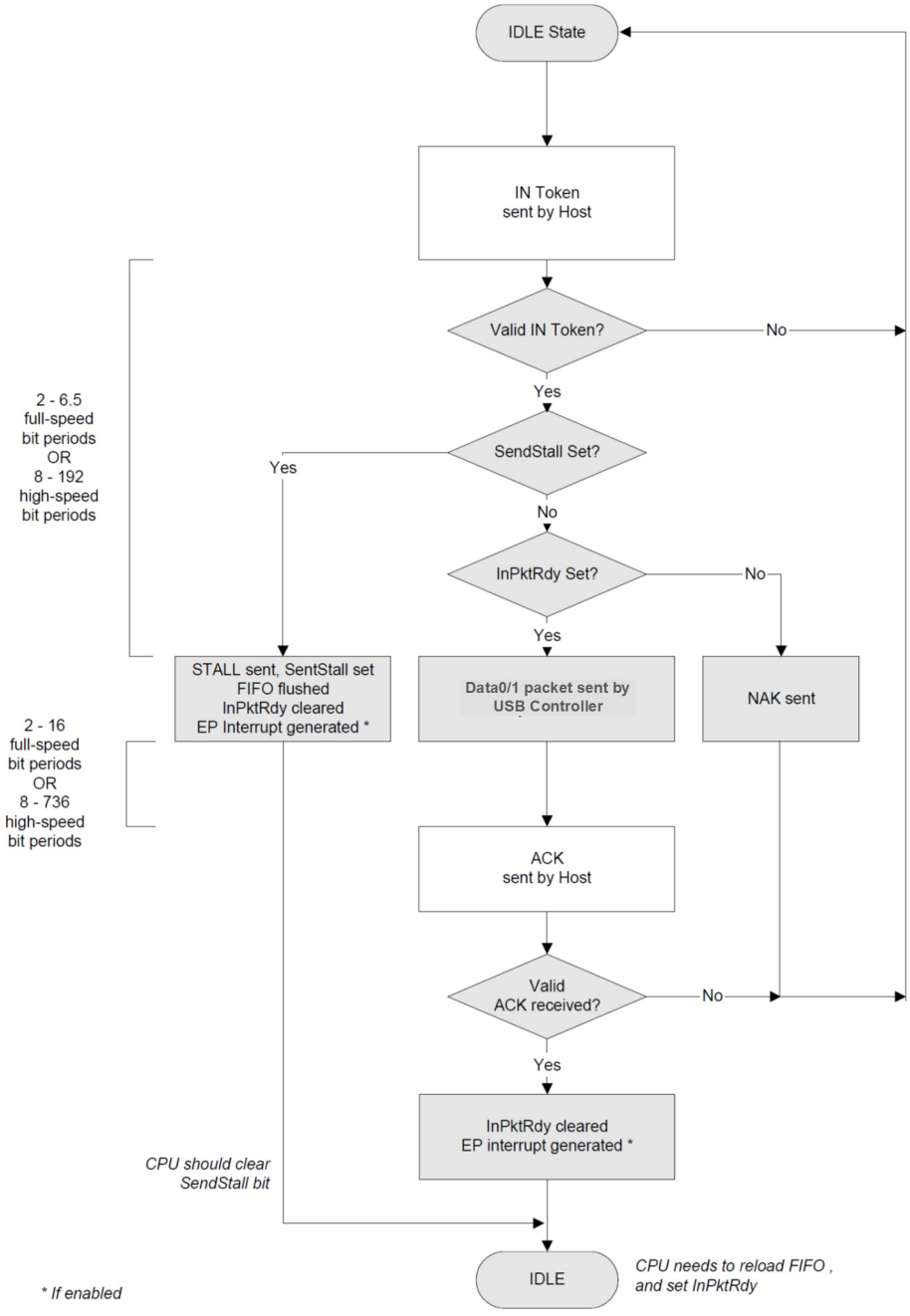


Figure 67. Bulk/Interrupt IN Transaction

15.16.2.2 OUT Transaction

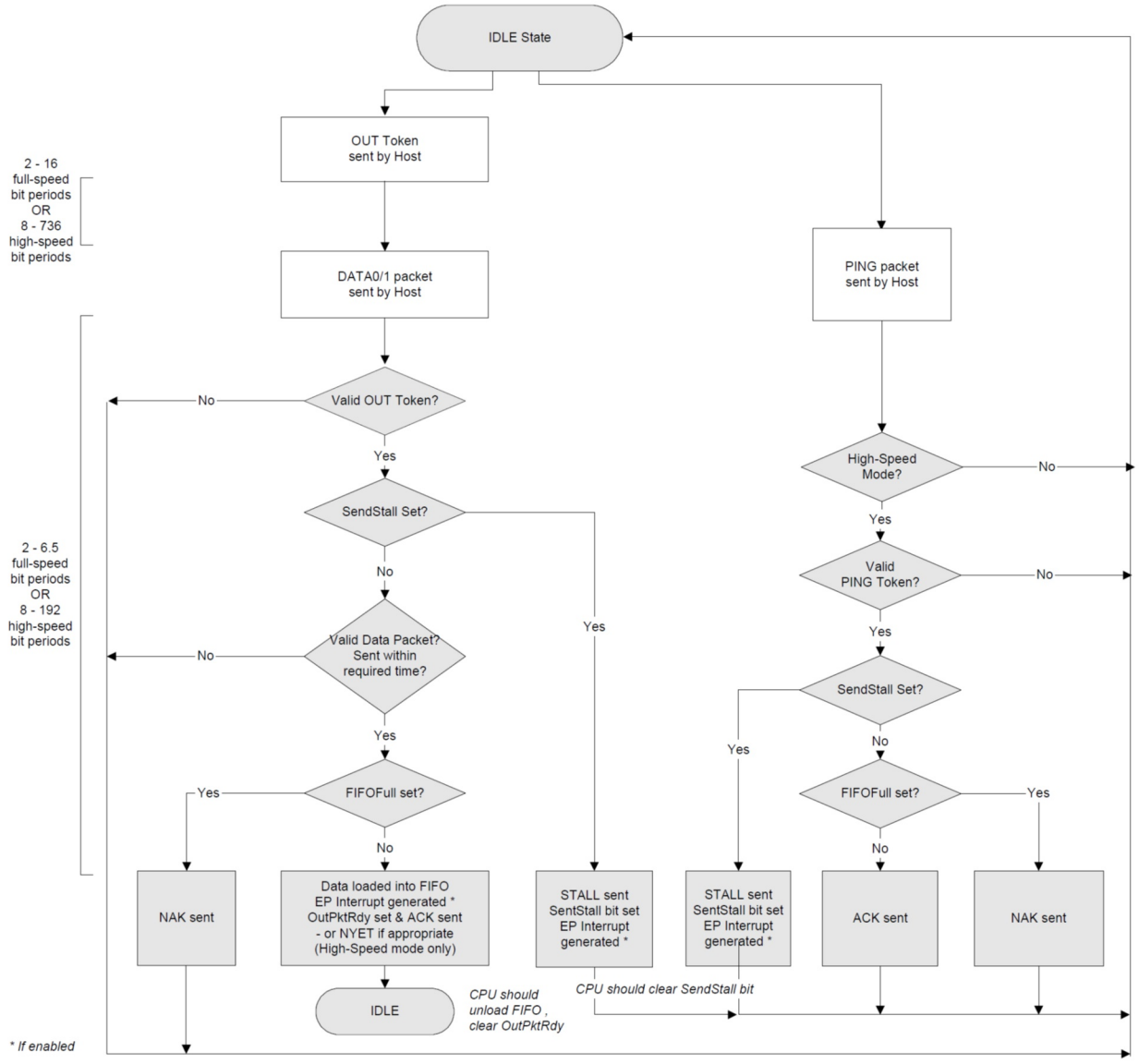


Figure 68. Bulk OUT Transaction

15.16.3 Full-speed Low-bandwidth Isochronous Transaction

15.16.3.1 IN Transaction

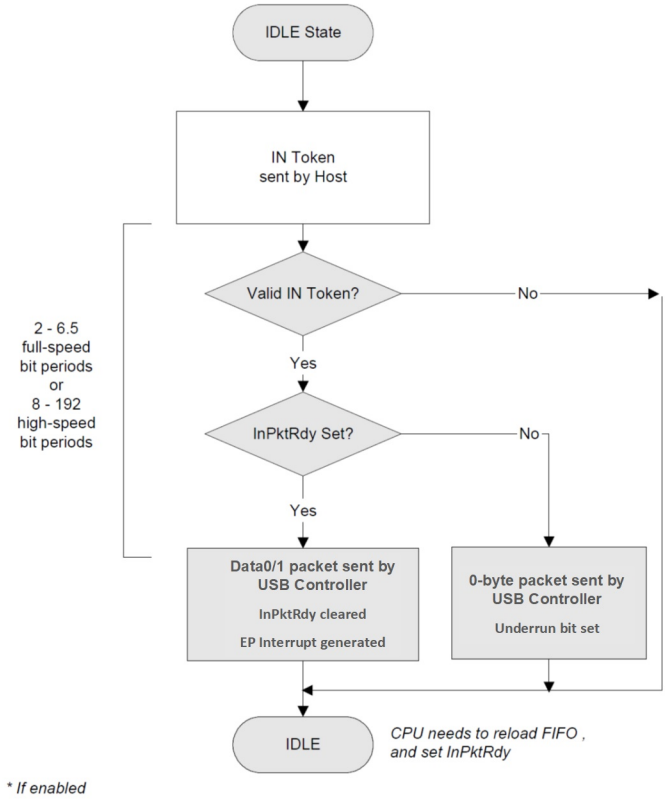


Figure 69. Full-speed Isochronous IN Transaction

15.16.3.2 OUT Transaction

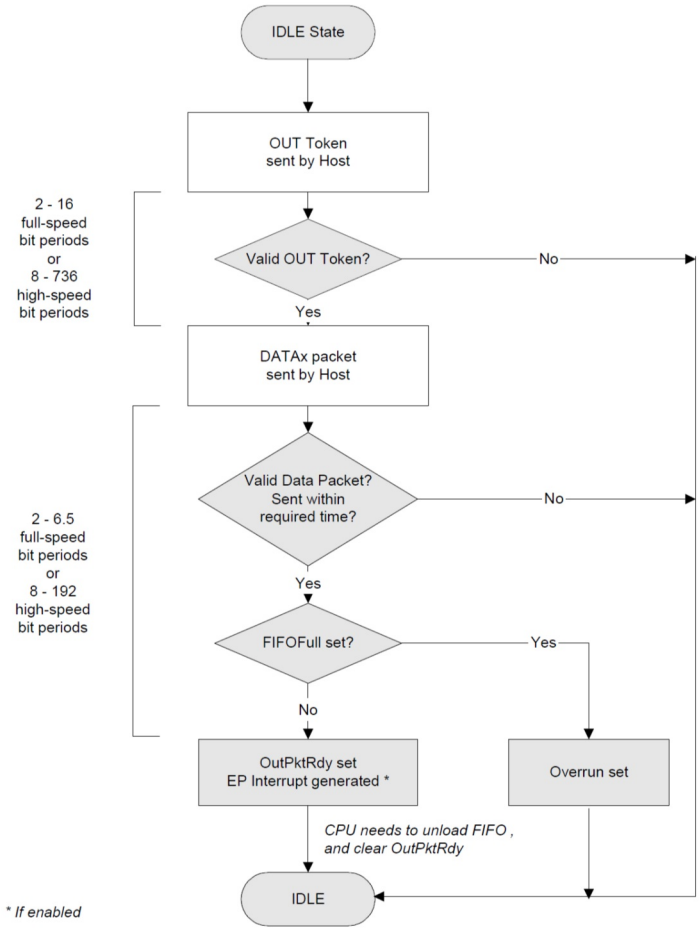


Figure 70. Full-speed Isochronous OUT Transaction

15.16.4 High-bandwidth Isochronous Transactions

15.16.4.1 IN Transaction

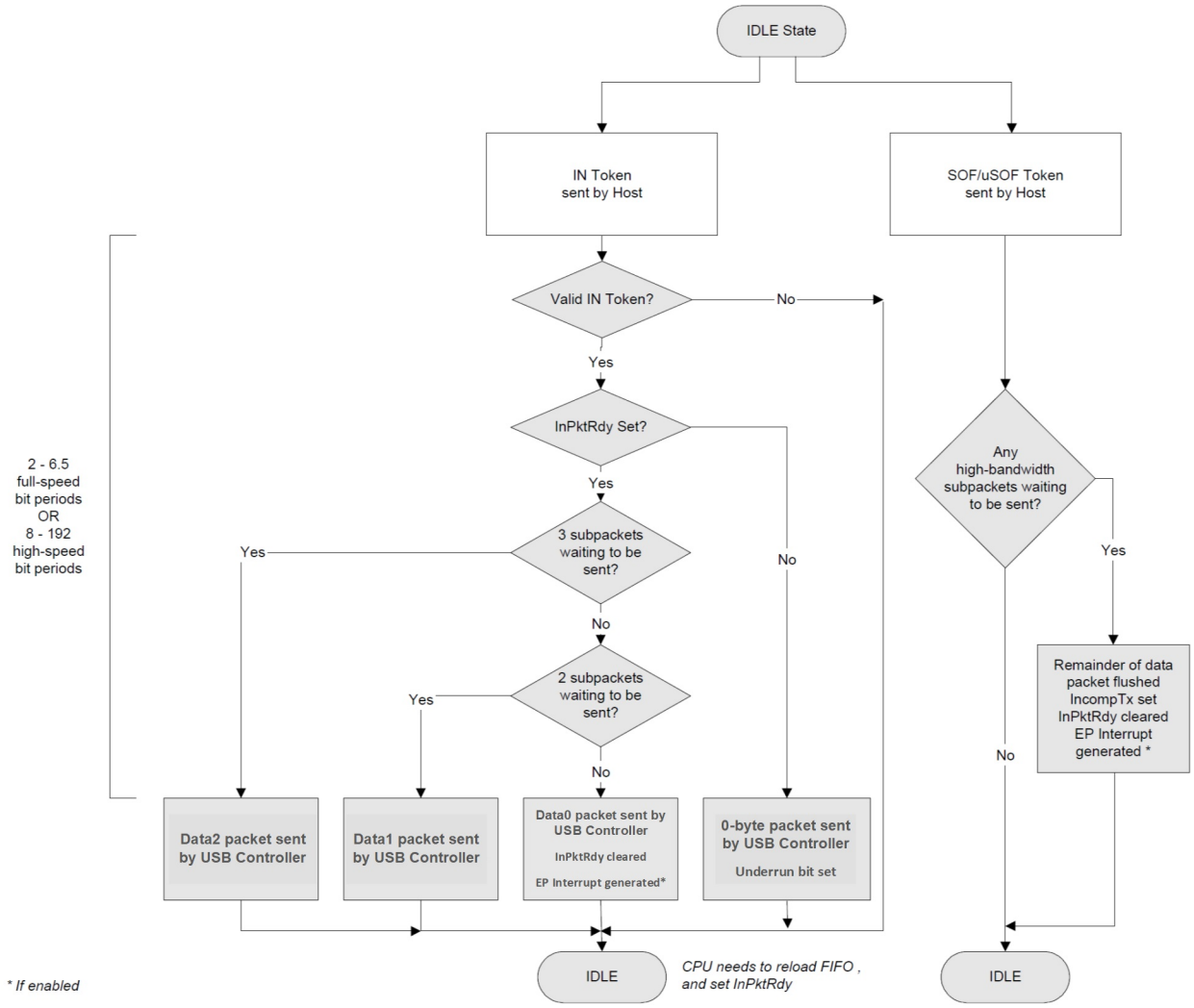


Figure 71. High-bandwidth Isochronous IN Transaction

15.16.4.2 OUT Transaction

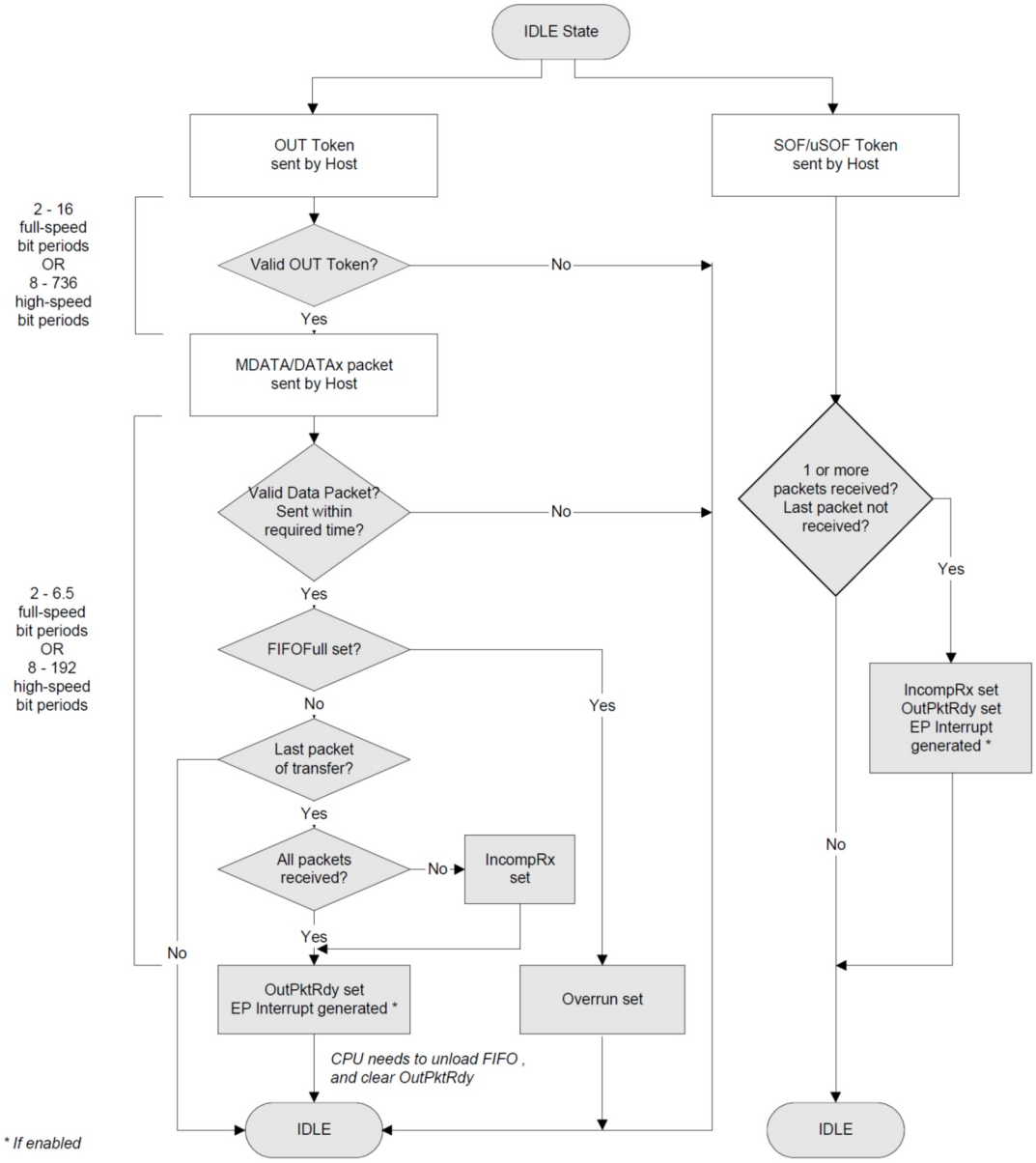


Figure 72. High-bandwidth Isochronous OUT Transaction

15.17 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.

15.17.1 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.

15.17.2 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.

15.17.3 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.

15.17.4 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.

16. Secure Digital Input Output (SDIO)

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

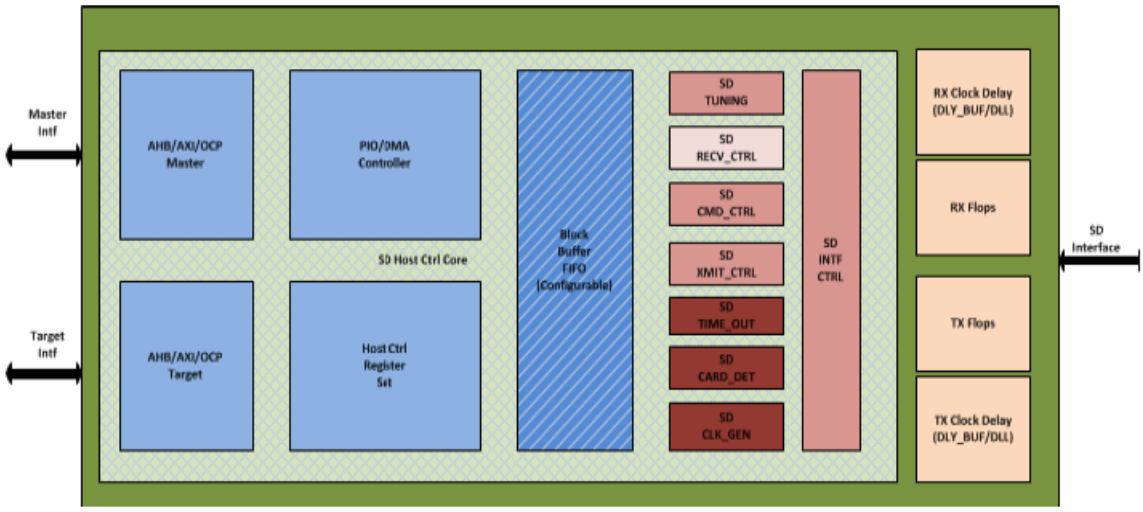


Figure 73. SD/SDIO Host Controller Interface Block Diagram

16.1 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.

16.1.1 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.

16.1.2 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) master 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 Master Interface module.

16.1.3 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.

16.1.4 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.

16.1.5 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.

16.1.6 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.

16.1.7 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.

16.1.8 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.

16.1.9 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.

16.1.10 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.

16.1.11 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.

16.2 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 master/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.

16.2.1 SD clocking Architecture

Figure 74 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.

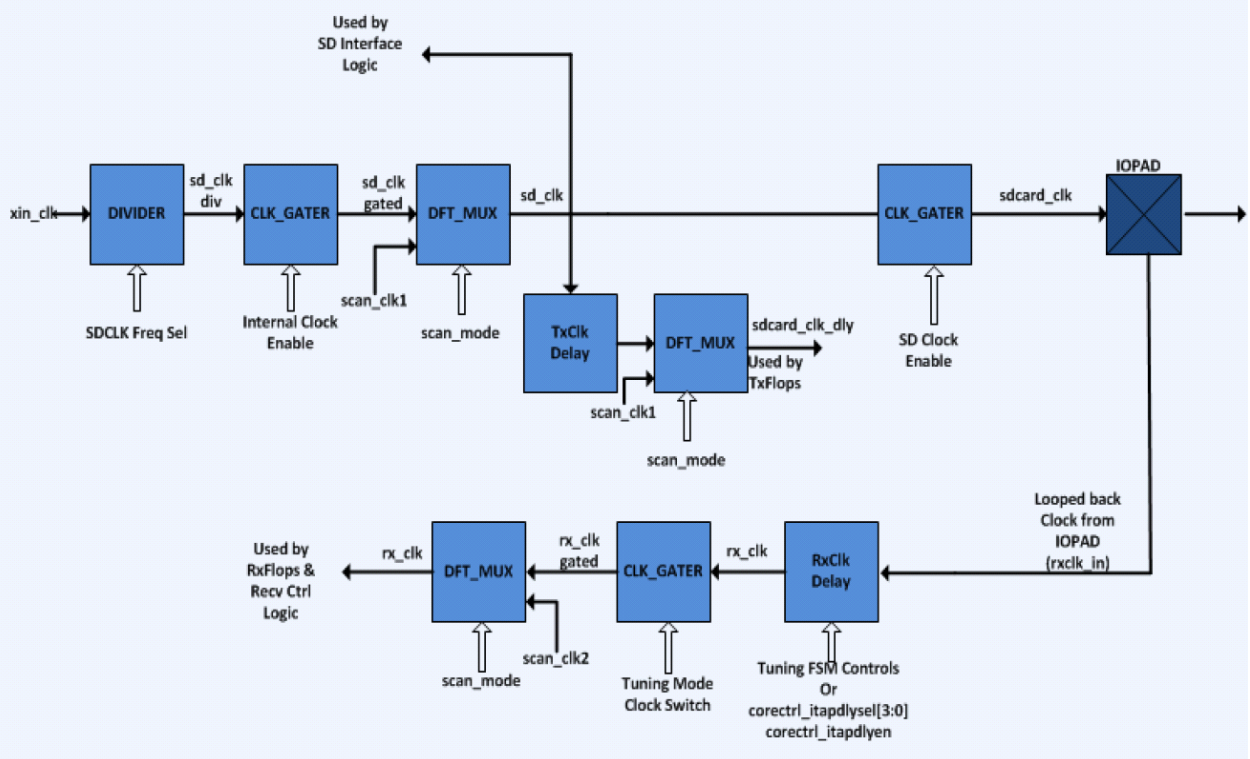


Figure 74. SDIO Module Clock Derivation

16.3 Advanced DMA

The SD Host Controller Standard Specification version 2.00 defines the ADMA (Advanced DMA) transfer algorithm. The DMA algorithm defined in the SD Host Controller Standard Specification version 1.00 is called SDMA (Single-operation DMA). SDMA has a disadvantage that a DMA interrupt is generated at every page boundary which requires the CPU to reprogram a new system address. This SDMA algorithm creates a performance bottleneck when interrupting at every page boundary. ADMA adopts a scatter-gather DMA algorithm so that higher data transfer speed is achievable. The CPU can program a list of data transfers between system memory and SD card in the descriptor table before executing ADMA. This enables ADMA to operate without interrupting the CPU. Furthermore, ADMA can support not only32-bit system memory addressing but also 64-bit system memory addressing (the 32-bit system memory addressing uses the lower 32-bit field of 64-bit address registers).

There are two types of ADMA - ADMA1 and ADMA2. ADMA1 supports data transfer of only 4KB-aligned data in system memory. ADMA2 improves upon this restriction so that data of any location and any size can be transferred in system memory. The format of the descriptor table is different between them. Version 2.00 of the SD Host Controller specification defines ADMA2 as standard ADMA and recommends supporting ADMA2 rather than ADMA1. DMA mode ADMA1 is not supported in versions 3.0 of the specification and later. When the term "ADMA" is used in this document, it means ADMA2.

16.3.1 Block Diagram of ADMA2

Figure 75 shows the ADMA2 algorithm block diagram.

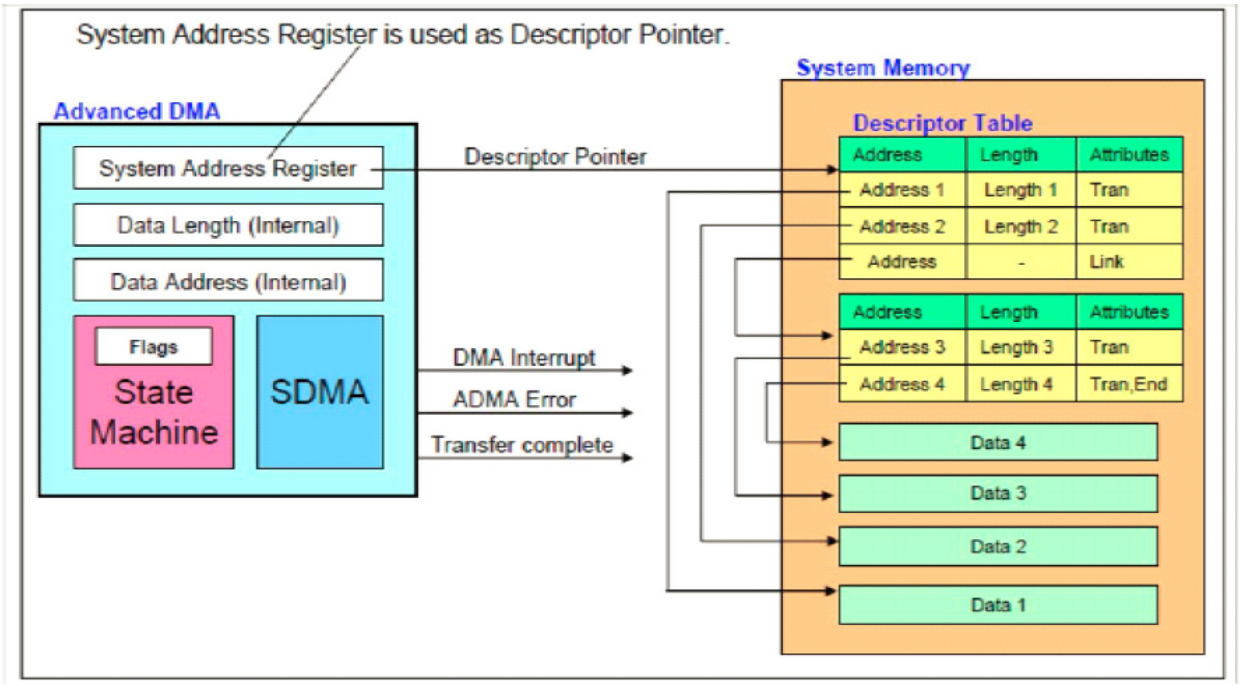


Figure 75. ADMA2 Block Diagram

A 32-bit addressing Descriptor Table is created in system memory by the CPU. Each descriptor line (one executable unit) consists of address, length and attribute fields. The attribute specifies operation of the 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).

16.3.2 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 76.

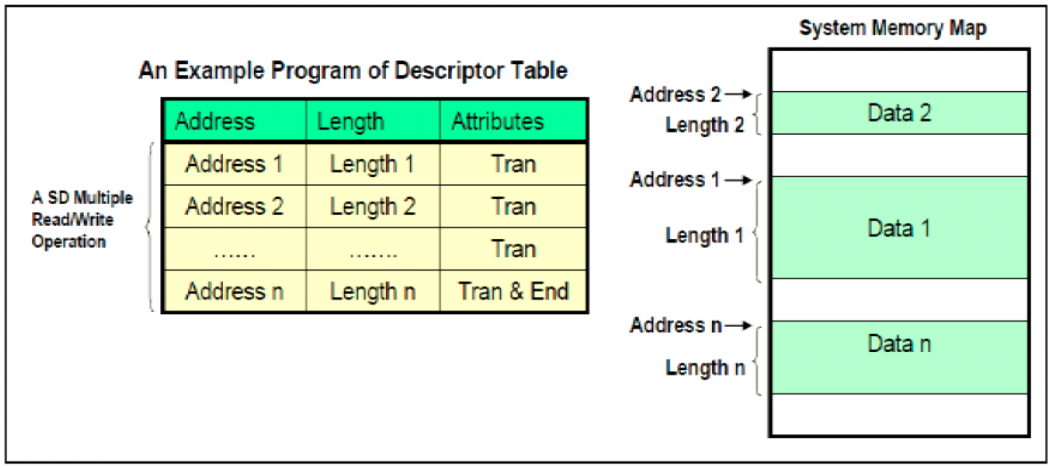


Figure 76. Example Descriptor Table

16.3.3 Data Address and Data Length Requirements

There are 3 requirements when programing the descriptor.

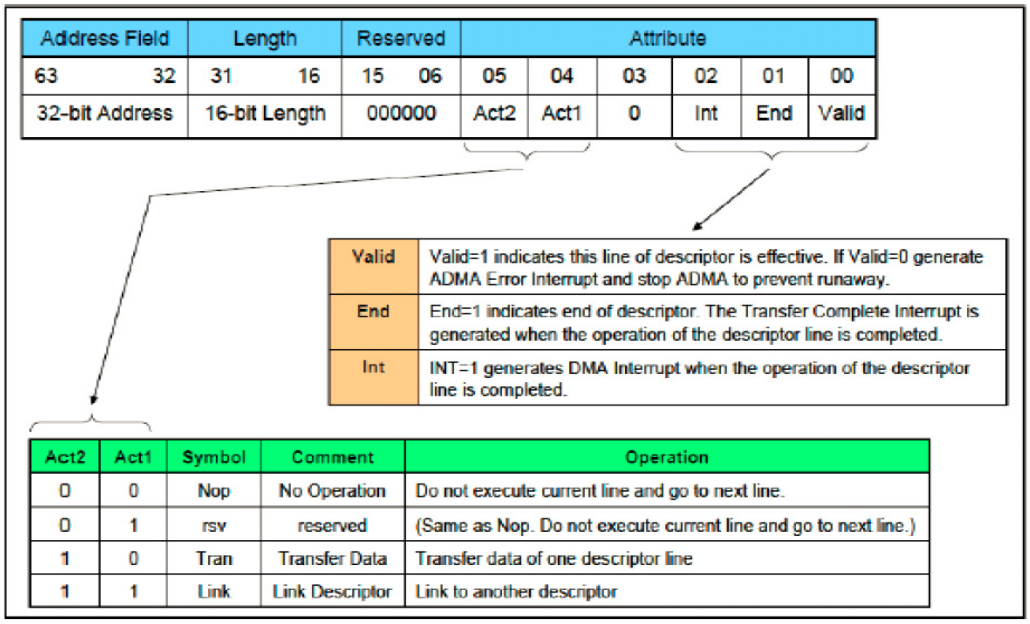
1. The minimum unit of address is 4 bytes.
2. The maximum data length of each descriptor line is less than 64KB.
3. Total Length = Length 1 + Length 2 + Length 3 + ... + Length n = multiple of Block Size.

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.

16.3.4 Descriptor Table

Figure 77 shows a 32-bit addressing Descriptor Table.



Address Field	Length	Reserved	Attribute					
63 32	31 16	15 06	05	04	03	02	01	00
32-bit Address	16-bit Length	000000	Act2	Act1	0	Int	End	Valid

Valid	Valid=1 indicates this line of descriptor is effective. If Valid=0 generate ADMA Error Interrupt and stop ADMA to prevent runaway.
End	End=1 indicates end of descriptor. The Transfer Complete Interrupt is generated when the operation of the descriptor line is completed.
Int	INT=1 generates DMA Interrupt when the operation of the descriptor line is completed.

Act2	Act1	Symbol	Comment	Operation
0	0	Nop	No Operation	Do not execute current line and go to next line.
0	1	rsv	reserved	(Same as Nop. Do not execute current line and go to next line.)
1	0	Tran	Transfer Data	Transfer data of one descriptor line
1	1	Link	Link Descriptor	Link to another descriptor

Figure 77. 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 78 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 78. Descriptor Table Length Field Definitions

16.4 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

16.4.1 Non-DMA Transaction

The sequence for transactions when not using DMA (non-DMA) is shown in Figure 79 and the steps are listed in Table 95.

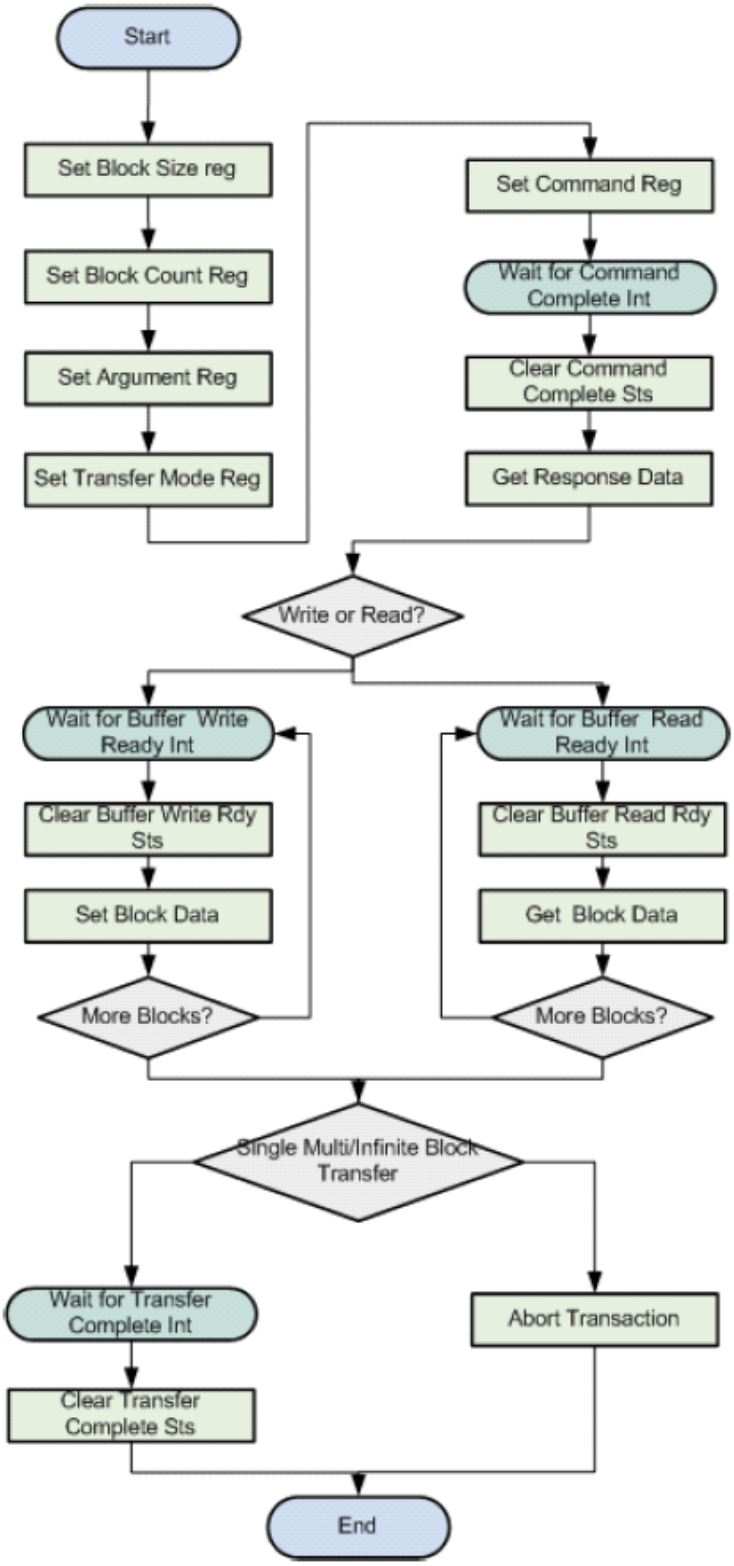


Figure 79. Non-DMA Transfer Flow Sequence

Table 95: 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.

16.4.2 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 master for a longer time.

The sequence for using DMA is shown in Figure 80 and the flow sequence steps are listed in Table 96.

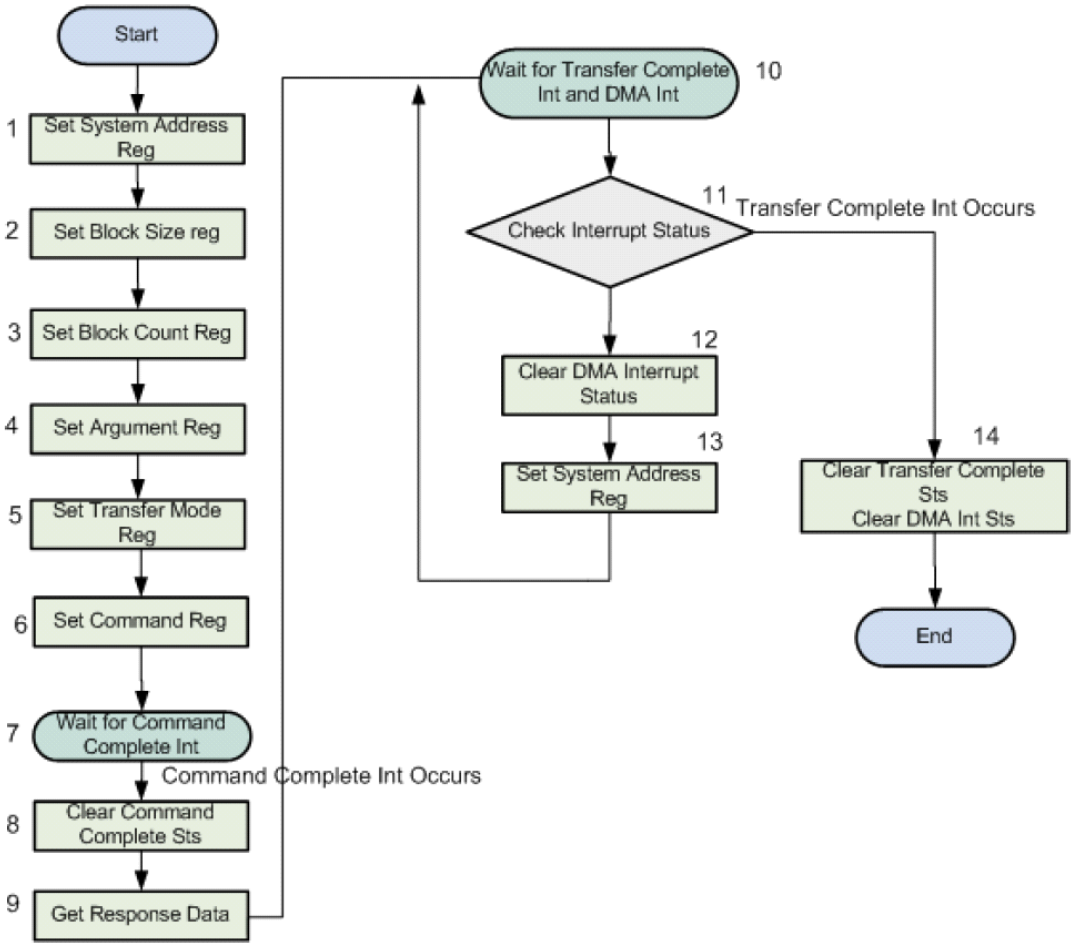


Figure 80. DMA Transfer Flow Sequence

Table 96: 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 master and transmitter residing inside the Host Controller will get the information (how much data to transfer) from these registers. Using the above information, the master 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.

16.4.3 ADMA Transactions

The sequence for using DMA is shown in Figure 80 and the flow sequence steps are listed in Table 96.

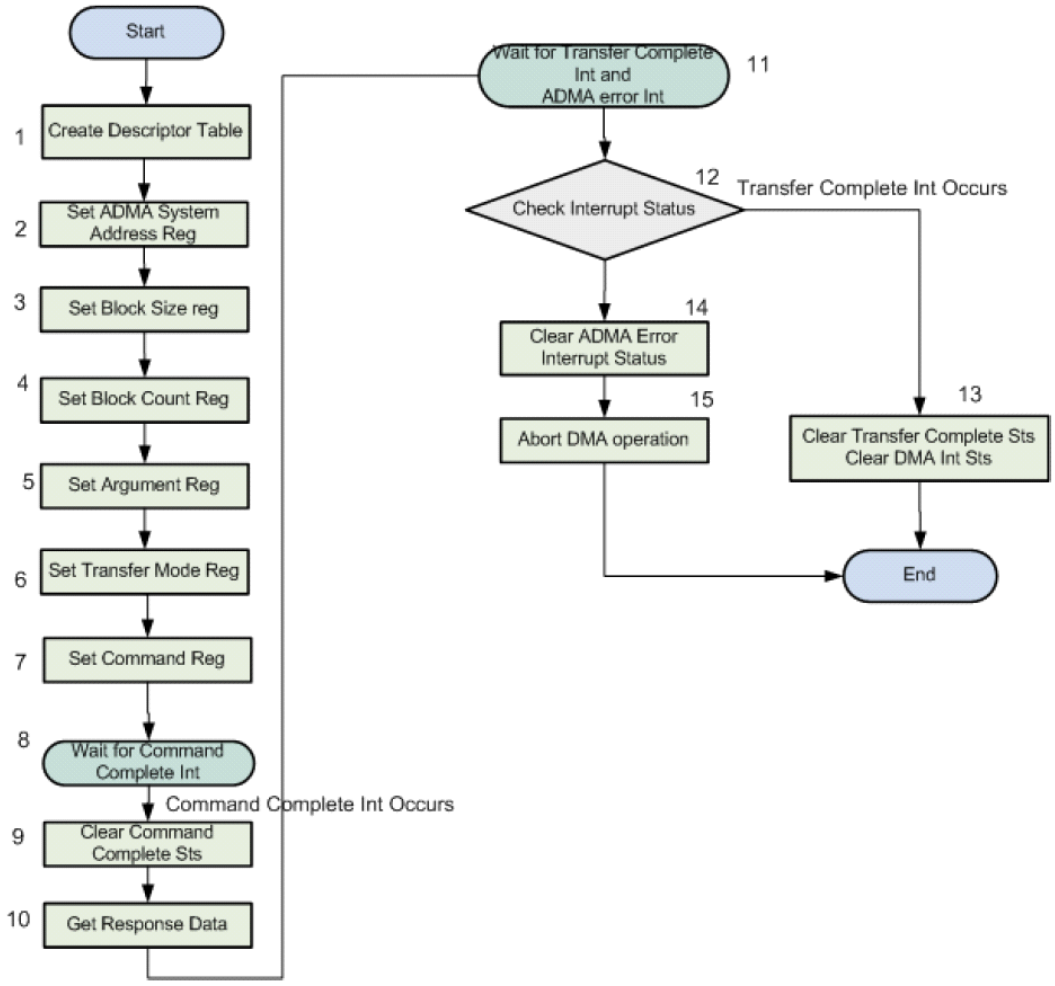


Figure 81. ADMA Transaction Flow Sequence

Table 97: 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.

16.5 Using the AmbiqSuite SDK to Program and Use the SDIO Module

In order to access the eMMC card, follow the steps listed below.

- **Define a global SD 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(AM_HAL_HOST_BUS_WIDTH_8);
```

- **Find and initialize the SD card host**

Currently only one SD card host is supported, so always use 'AM_HAL_SDHC_CARD_HOST', 'true' parameter. This forces the SD card host to reinitialize from scratch. Using 'false' will not force the SD card host to do reinitialization.

```
//
// Get the uderlying SDHC card host instance
//
pSdhcCardHost = am_hal_get_card_host(AM_HAL_SDHC_CARD_HOST, true);

if (pSdhcCardHost == NULL)
{
    am_util_stdio_printf("No such card host and stop\n");
    while(1);
}
am_util_stdio_printf("card host is found\n");
```

- **Define an eMMC card and check if it is present**

Currently only the eMMC card type is supported.

```
am_hal_card_t eMMCard;

//
// check if card is present
//
while (am_hal_card_host_find_card(pSdhcCardHost, &eMMCard) != AM_HAL_STATUS_SUCCESS)
{
    am_util_stdio_printf("No card is present now\n");
    am_util_delay_ms(1000);
    am_util_stdio_printf("Checking if card is available again\n");
}
```

- **Initialize the eMMC card**

Optionally, the user can provide a card power control callback function during card initialization, as well as establish a card "power off" policy. Two policies are defined - one powers off only the SDHC controller and the other powers off both the SDHC controller and the eMMC card.

After initialization, the eMMC card is running with the default setting - 1-bit bus width and lowest clock speed

```
uint32_t
am_hal_card_init(am_hal_card_t *pCard,
                 am_hal_card_pwr_ctrl_func pCardPwrCtrlFunc,
                 am_hal_card_pwr_ctrl_policy_e eCardPwrCtrlPolicy)
```

```
while (am_hal_card_init(&eMMCard, NULL, AM_HAL_CARD_PWR_CTRL_SDHC_OFF) != AM_HAL_STATUS_SUCCESS)
{
    am_util_delay_ms(1000);
    am_util_stdio_printf("card init failed, try again\n");
}
```

- **Configure the eMMC card**

Configure the bus width, bus speed, bus voltage and eMMC bus mode according to the hardware design and eMMC chip.

```
//
// 48MHz, 4-bit DDR mode for read and write
//
while (am_hal_card_cfg_set(&eMMCard, AM_HAL_CARD_TYPE_EMMC,
    AM_HAL_HOST_BUS_WIDTH_4, 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");
}
```

- **Perform synchronous block read or block write**

A synchronous block read or block write function needs to specify the start sector (or block) and sector (block) numbers.

The sector (or block) size is **512** bytes.

The read & write buffer size should be **512*BLK_NUM**.

The return value, **ui32Status**, includes the successful transferred sector (or block) number in the upper 16 bits and the transfer status in the lower 16 bits.

```
ui32Status = am_hal_card_block_read_sync(&eMMCard, START_BLK, BLK_NUM, (uint8_t *)ui8RdBuf);
am_util_stdio_printf("Synchronous Reading %d blocks is done, Xfer Status %d\n", ui32Status >> 16, ui32Status & 0xffff);
```

```
ui32Status = am_hal_card_block_write_sync(&eMMCard, START_BLK, BLK_NUM, (uint8_t *)ui8WrBuf);
am_util_stdio_printf("Synchronous Writing %d blocks is done, Xfer Status %d\n", ui32Status >> 16, ui32Status & 0xffff);
```

- **Perform asynchronous block read or block write**

An asynchronous block read or block write 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(&eMMCard, am_hal_card_event_test_cb);
```

```
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;

    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);
    }

    if (AM_HAL_EVT_XFER_COMPLETE == pEvt->eType &&
        pHost->AsyncCmdData.dir == AM_HAL_DATA_DIR_WRITE)
    {
        bAsyncWriteIsDone = true;
        am_util_stdio_printf("Last Write Xfered block %d\n", pEvt->ui32BlkCnt);
    }

    if (AM_HAL_EVT_SDMA_DONE == pEvt->eType &&
        pHost->AsyncCmdData.dir == AM_HAL_DATA_DIR_READ)
    {
        am_util_stdio_printf("SDMA Read Xfered block %d\n", pEvt->ui32BlkCnt);
    }

    if (AM_HAL_EVT_SDMA_DONE == pEvt->eType &&
        pHost->AsyncCmdData.dir == AM_HAL_DATA_DIR_WRITE)
    {
        am_util_stdio_printf("SDMA Write Xfered block %d\n", pEvt->ui32BlkCnt);
    }

    if (AM_HAL_EVT_DAT_ERR == pEvt->eType)
    {
        am_util_stdio_printf("Data error type %d\n", pHost->AsyncCmdData.eDataError);
    }

    if (AM_HAL_EVT_CARD_PRESENT == pEvt->eType)
```

```
//
// Write 'BLK_NUM' blocks to the eMMC flash
//
bAsyncWriteIsDone = false;
am_hal_card_block_write_async(&eMMCard, START_BLK, BLK_NUM, (uint8_t *)ui8WrBuf);
```

```
//
// Read 'BLK_NUM' blocks to the eMMC flash
//
bAsyncReadIsDone = false;
am_hal_card_block_read_async(&eMMCard, START_BLK, BLK_NUM, (uint8_t *)ui8RdBuf);
```

16.5.1 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.

After calling this function, the Tx/Rx settings will take effective immediately.

```
am_hal_card_emmc_calibrate(AM_HAL_HOST_UHS_DDR50, 48000000, AM_HAL_HOST_BUS_WIDTH_4,
    (uint8_t *)ui8WrBuf, START_BLK, 2, ui8TxRxDelays);

am_util_stdio_printf("SDIO TX delay - %d, RX Delay - %d\n", ui8TxRxDelays[0], ui8TxRxDelays[1]);
```

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);
```

17. Display Controller (DC)

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.

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.

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

17.1 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.

17.1.1 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

17.2 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 82 shows how the parameters are defined.

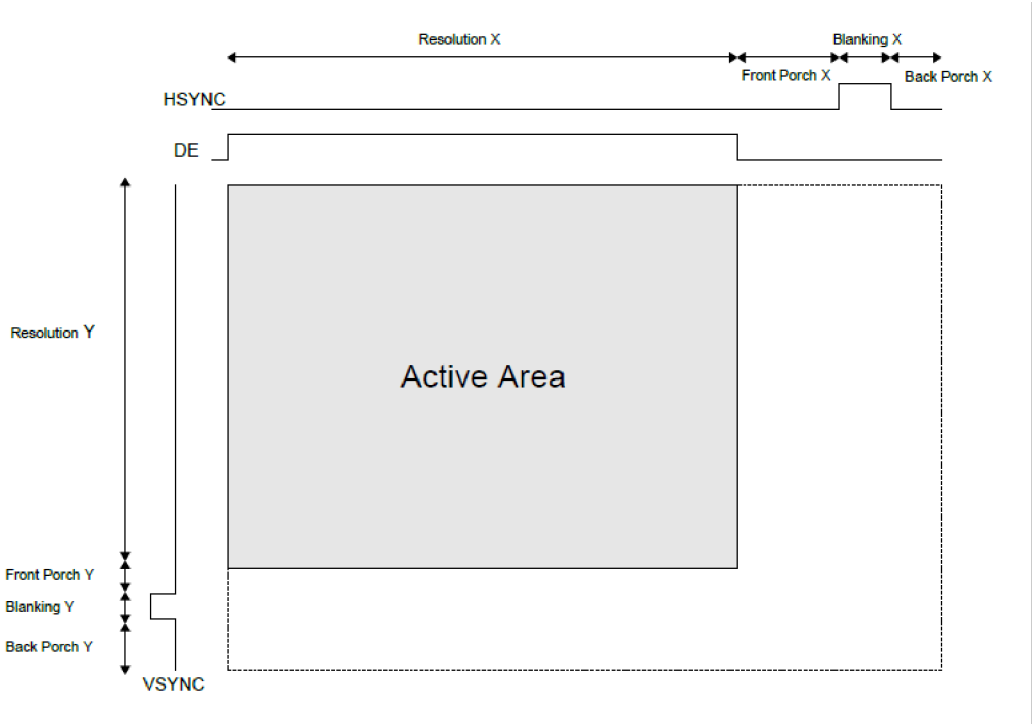


Figure 82. Display Controller Video Timing

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.

Equations 1-3 determine in pixel clock cycles, the X11 Modeline horizontal:

Front Porch (FRONTPORCHXY_FPCLKCYCLES field)

$$FRONTPORCHXY\_FPCLKCYCLES = ResolutionX + FrontP\ orchX \quad (1)$$

Blanking Period (BLANKINGXY_HSYNCPULSE field)

$$BLANKINGXY\_HSYNCPULSE = ResolutionX + FrontPorchX + BlankingX \quad (2)$$

Back Porch (BACKPORCHXY_BPCLKCYCLES field)

$$BACKPORCHXY\_BPCLKCYCLES = ResolutionX + FrontP\ orchX + BlankingX + BackPorchX \quad (3)$$

Equations 4-6 determine in number of lines, the X11 Modeline vertical:

Front Porch (FRONTPORCHXY_FLINES field)

$$FRONTPORCHXY\_FLINES = ResolutionY + FrontPorchY \quad (4)$$

Blanking Period (BLANKINGXY_VSYNCLINES field)

$$BLAN\,K\_XY\_VSYNCLINES = ResolutionY + FrontP\ orchY + BlankingY \quad (5)$$

Back Porch (BACKPORCHXY_BLINES field)

$$BACKPORCHXY\_BLINES = ResolutionY + FrontP\ orchY + BlankingY + BackP\ orchY \quad (6)$$

17.3 Layer Overlays

The Display Controller supports four layers (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 83, which is applied on the entire screen. If it is not needed the background register BGCOLOR field is set to 0.

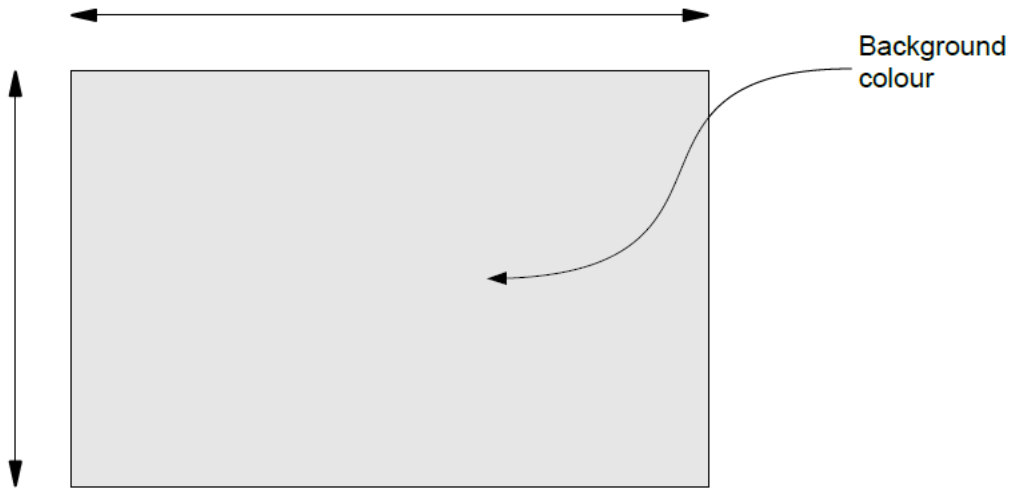


Figure 83. Display Controller RGBA Background Color

Layer 0 is applied on top of that with a requested blending method as shown in Figure 84. 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.

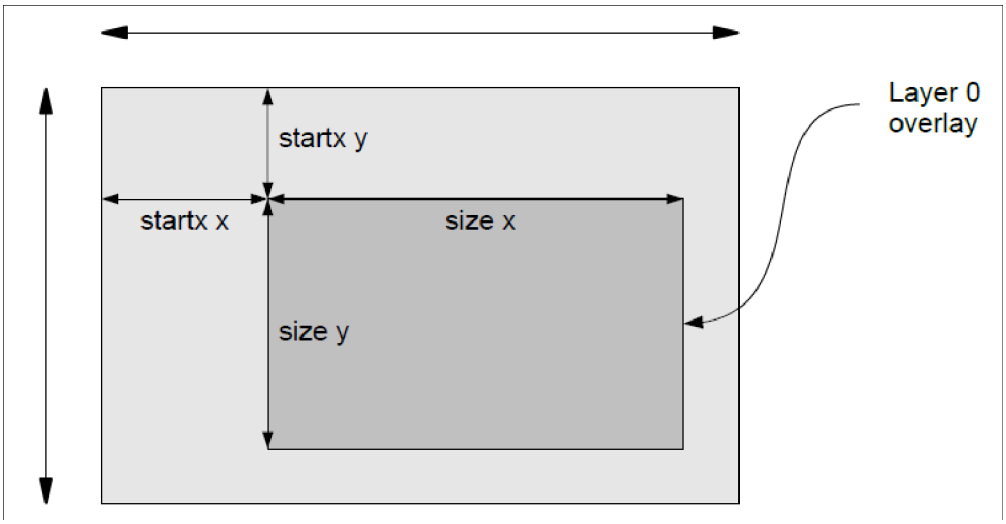


Figure 84. Display Controller First Layer

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 98:

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 98: 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 85 shows how the second layer is overlayed.

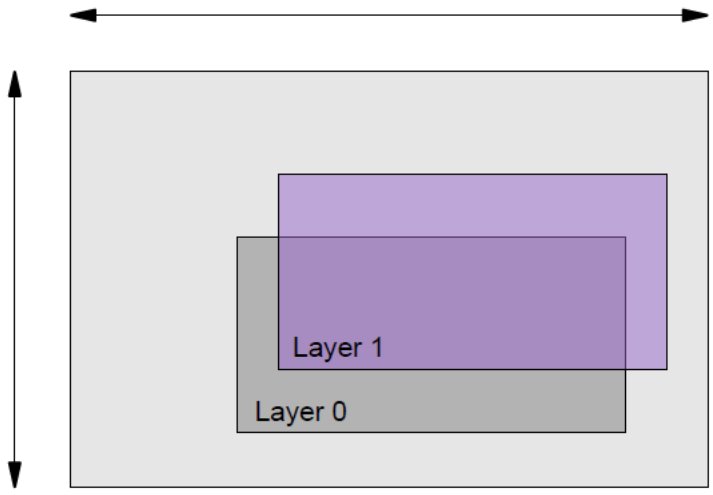


Figure 85. Display Controller Second Layer

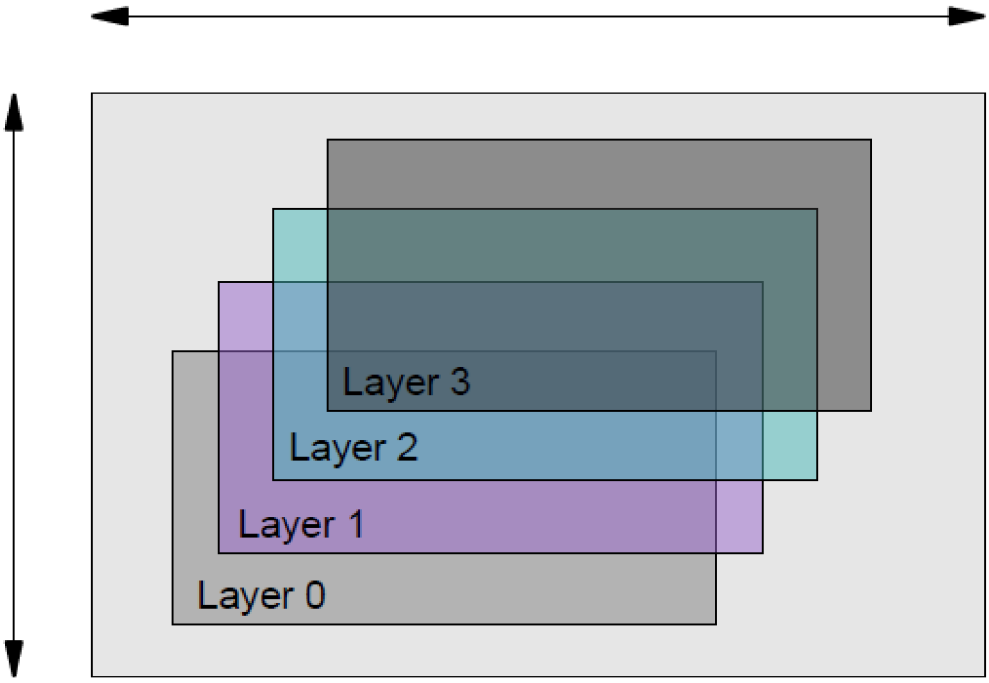


Figure 86. Display Controller Third and Fourth Layers

17.4 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 99 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 99: Blending Modes

Blending is enabled when the LAYERnMODE_LAYERnDBLEND field is set with the respective values of Table 99. The destination blending function refers to the current layer while the source blending function refers to the previous layer.

17.5 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 87 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.

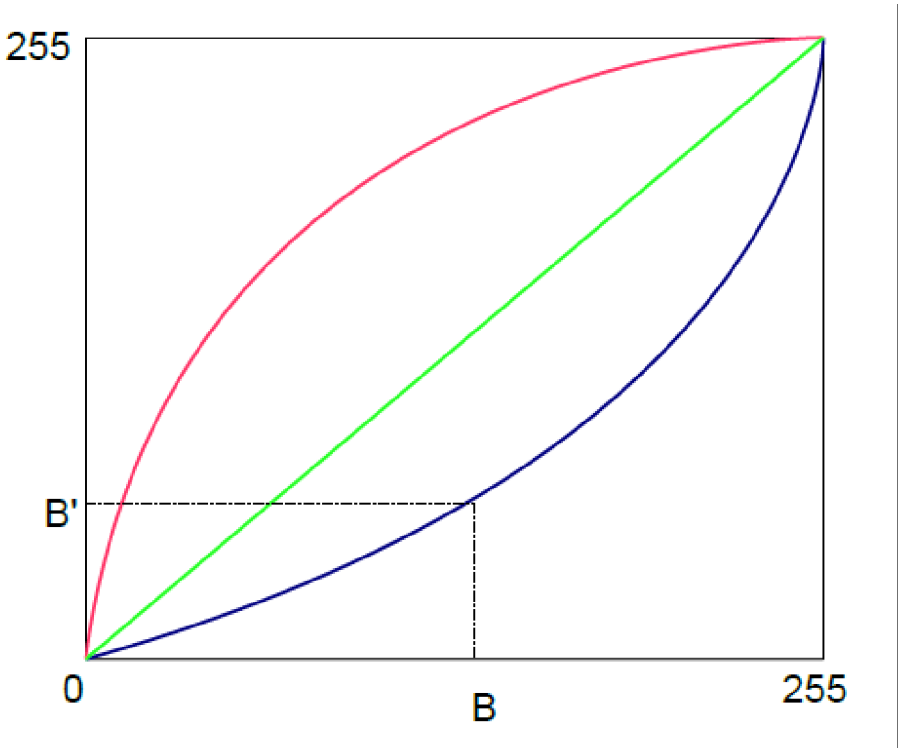


Figure 87. Gamma Correction of RGB Values

17.6 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) RGBA444 with dithering**

Figure 88. Dithering on Limited Color Palette

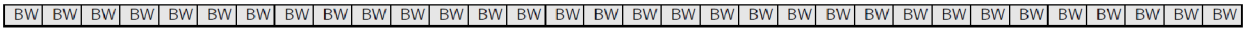
17.7 Color Modes

The DC supports multiple color formats. The supported color formats are described in the following sections.

17.7.1 Input Formats

17.7.1.1 Binary BW 1-bit

Values range from 0 (white) to 1 (black)

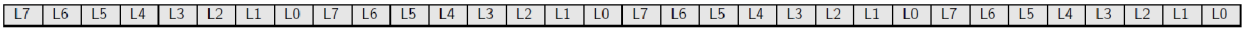


17.7.1.2 Grayscale 4-bit

Values range from 0 (black) to 16 (white)

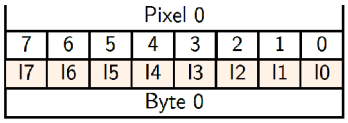
17.7.1.3 Grayscale 8-bit

Values range from 0 (black) to 255 (white)

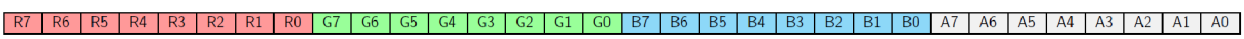


17.7.1.4 Palette 256 8-bit (LUT8)

Values range from 0 to 255. These are looked-up from the Palette Color Table.



17.7.1.5 RGBA 8888 32-bits



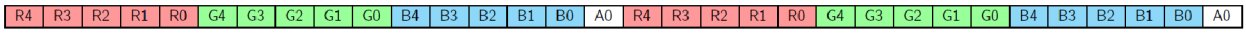
17.7.1.6 ARGB 8888 32-bit

17.7.1.7 ABGR 8888 32-bit

17.7.1.8 BGRA 8888 32-bit

17.7.1.9 RGB 888 24-bit

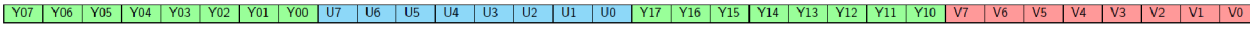
17.7.1.10 RGBA 5551 16-bit



17.7.1.11 RGB 565 16-bit

17.7.1.12 RGB 332 8-bit

17.7.1.13 YUYV 32-bit 2-pixels



17.7.1.14 YUY2 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

17.7.1.15 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

17.7.1.16 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

17.7.1.17 TSc4 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

17.7.1.18 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

17.7.1.19 Custom formats

Additional color formats can be easily supported upon our client's requests. Please do not hesitate to contact us for more information.

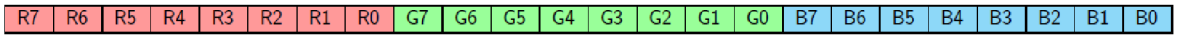
17.7.1.20 Color expansion

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:

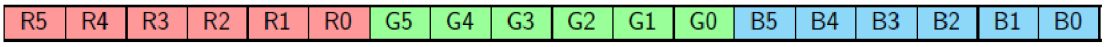
C[7:0] = {C[4:0];C[4:2]}

17.7.2 Output Formats

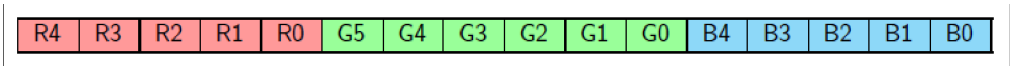
17.7.2.1 RGB 888 24-bit



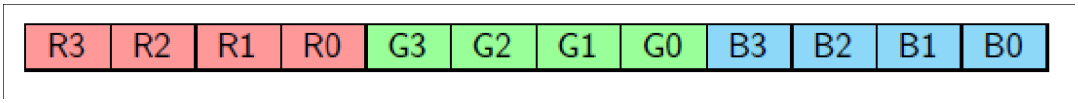
17.7.2.2 RGB 666 18-bit



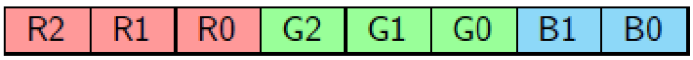
17.7.2.3 RGB 565 16-bit



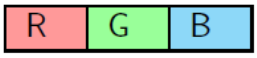
17.7.2.4 RGB 444 12-bit



17.7.2.5 RGB 332 8-bit



17.7.2.6 RGB 111 3-bit



17.7.2.7 Binary

Values range from 0 (black or white) to 1 (black or white)

17.8 TSc Framebuffer Decompression

TS 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 4:1 compression (6bpp) 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 89 shows the TS compression operation. The output of the TSC™4 compression is 64 bits per 4x4 block of pixels and the output of the TSC™6 compression is 96 bits per 4x4 block of pixels.

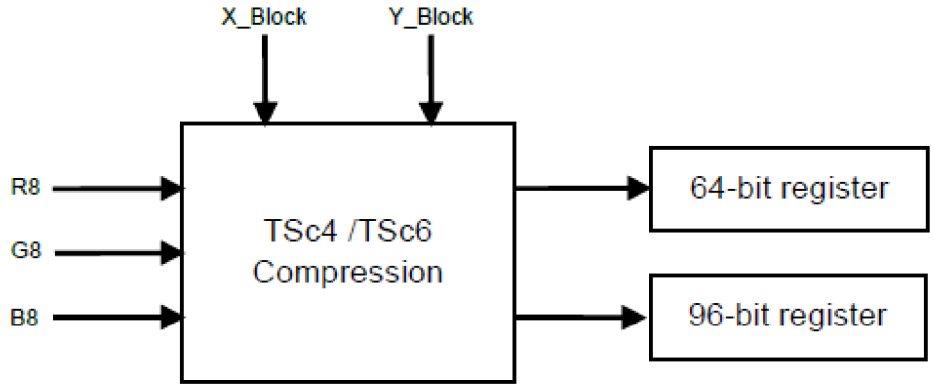


Figure 89. FTSC™4 /TSC™6 Framebuffer Compression Module

17.9 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.

NOTE
Use of the DPI-2 interface is not supported and therefore is not described in this Display Formats section.

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.

17.9.1 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 90 depicts the write sequence timing waveforms. Note that blue line indicates the high impedance state.

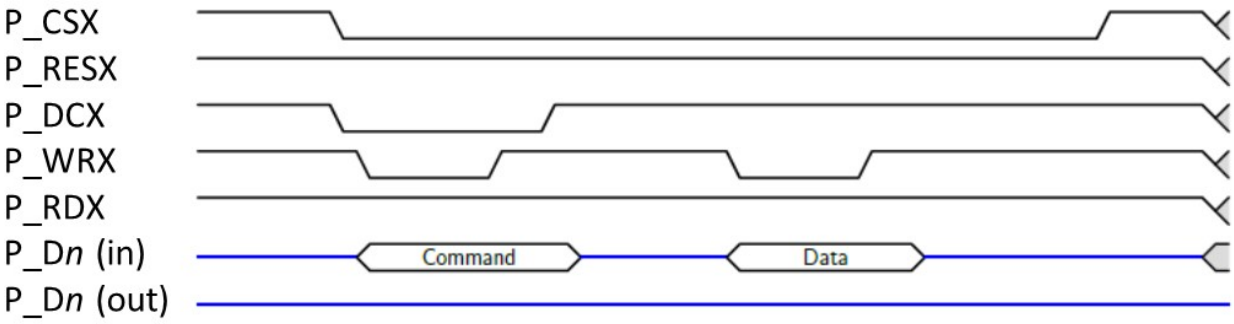


Figure 90. DBI-Type B Write Sequence

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 91 depicts the read sequence timing waveforms. Note that blue line indicates the high impedance state.

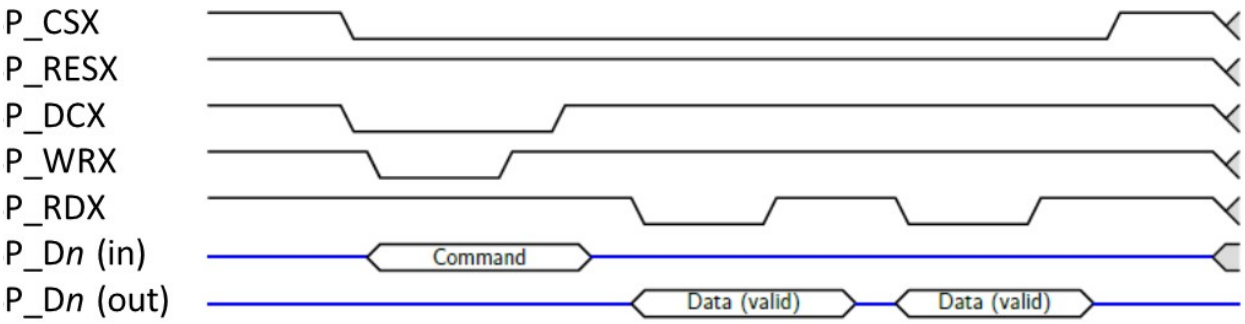


Figure 91. DBI-Type B Read Sequence

DBI-Type B interface is enabled through the MODE_DBITYPEBEN 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.

17.9.1.1 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.

17.9.1.1.1 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.

17.9.1.1.2 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.

17.9.1.1.3 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.

17.9.1.1.4 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.

17.9.1.1.5 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.

17.9.1.2 DBI-Type B Output Formats - 9bit Interface

The following figures show the color coding DBI-Type B Interface supports when Data Bus (DBIB_DB) is 9 bits wide:

17.9.1.2.1 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.

17.9.1.3 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, Bit2
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, Bit2
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, Bit2

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.

17.9.1.3.1 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.

17.9.1.3.2 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.

17.9.1.3.3 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.

17.9.1.3.4 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.

17.9.1.3.5 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.

17.9.1.3.6 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.

17.9.2 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 (bi-directional 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 (bi-directional 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.

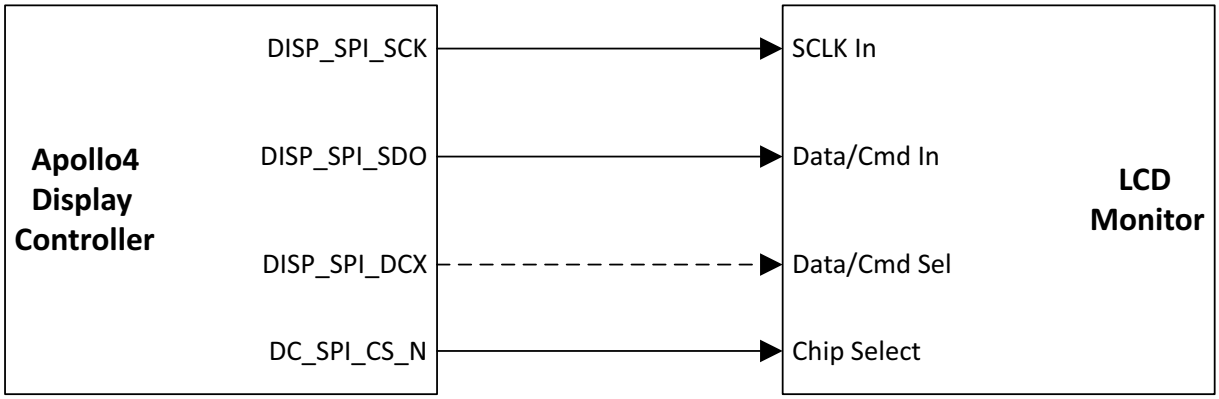


Figure 92. SPI 3- or 4-wire Interface (Dashed Line Used in 4-wire) - Data Out Only

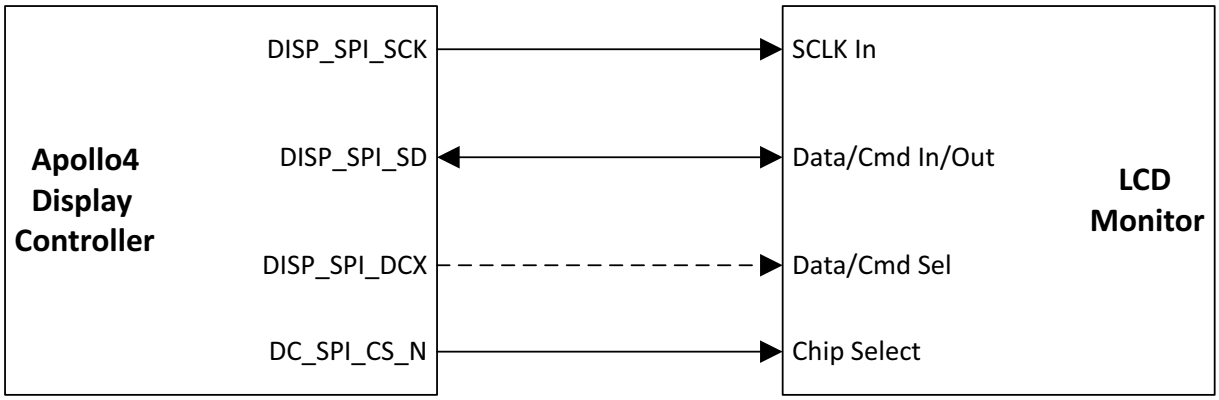


Figure 93. SPI 3- or 4-wire Interface (Dashed Line Used in 4-wire) - Bi-directional Data

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 94 and Figure 95 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 95 when low, indicates a command and when high, indicates data.

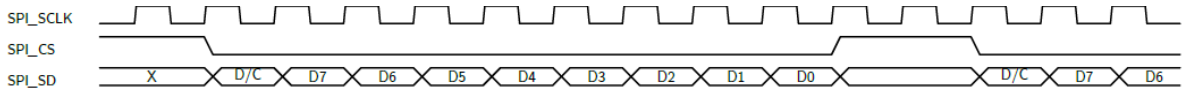


Figure 94. SPI 3-wire Serial Write Transmission

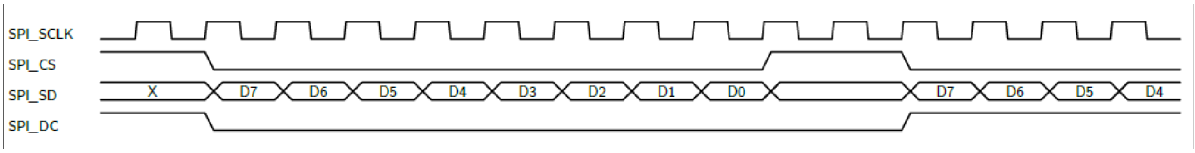


Figure 95. SPI 4-wire Serial Write Transmission

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 96 shows one line transmission data.

Figure 96. 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 97 shows multiple lines transmission data.

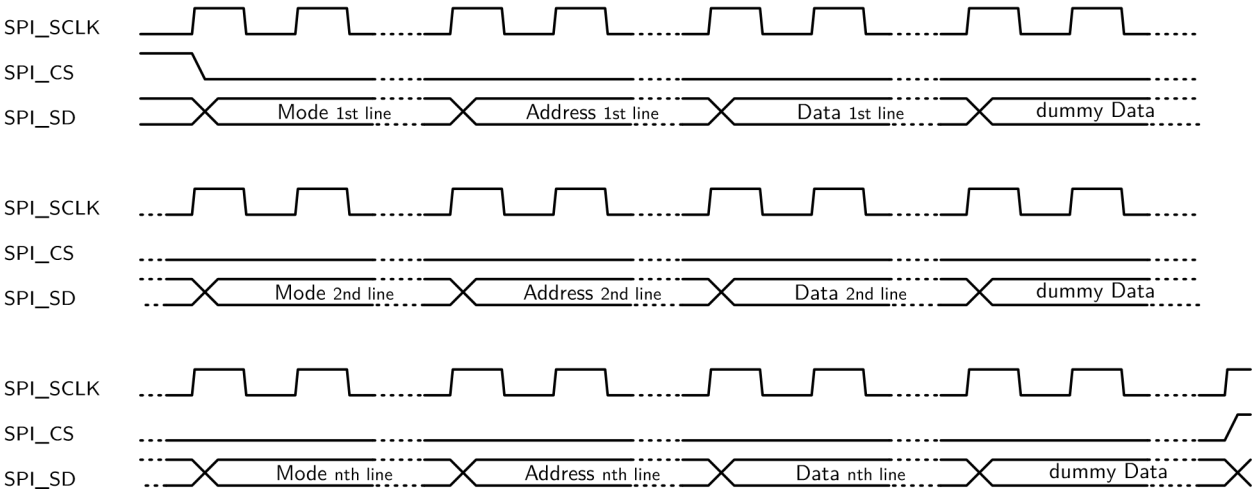


Figure 97. Multiple Lines Update Mode

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.

17.9.2.1 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

17.9.2.1.1 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.

17.9.2.1.2 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.

17.9.2.1.3 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.

17.9.2.1.4 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.

17.9.2.1.5 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.

17.9.2.1.6 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.

17.9.2.1.7 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.

17.9.2.1.8 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.

17.9.2.1.9 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.

17.9.3 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 bi-directional 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 98 and Figure 99 show how the DC is connected to an LCD monitor in either QuadSPI or DualSPi mode using the aforementioned signals.

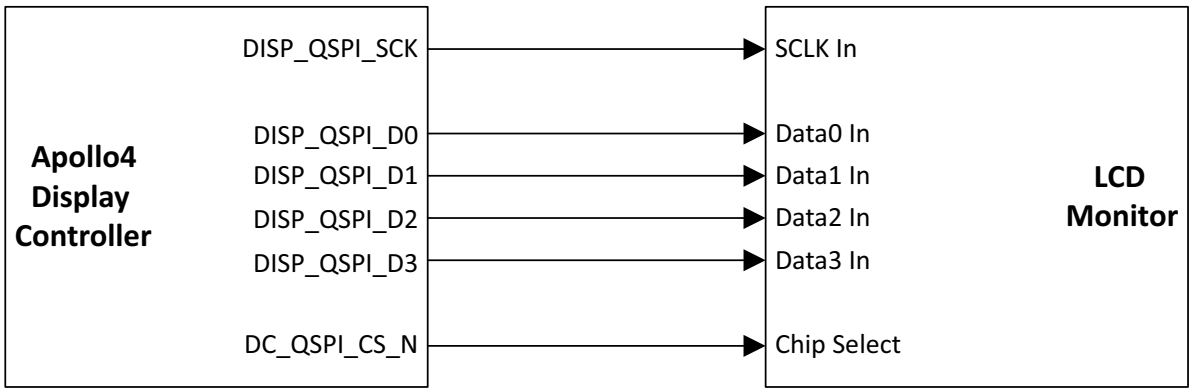


Figure 98. QuadSPI Interface

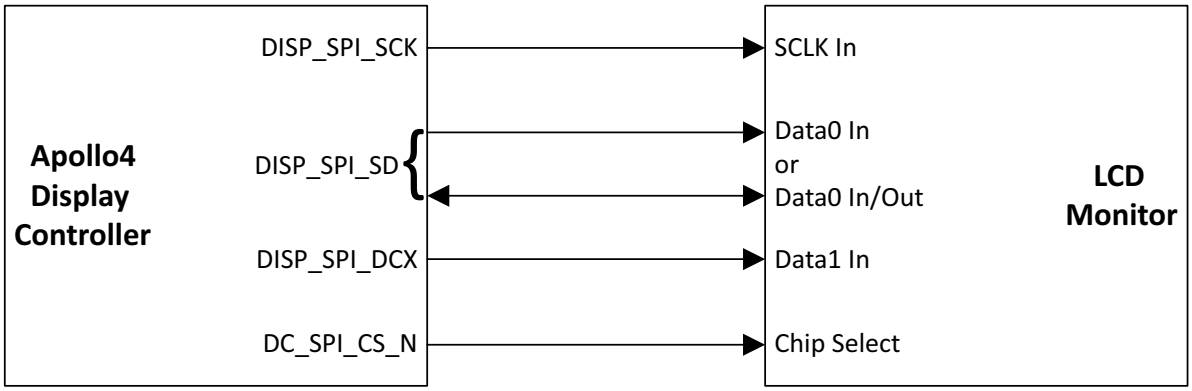


Figure 99. DualSPI Interface

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 selectsignals generally conform to the 3-wire SPI interface timing as shown and described in "SPI, 3-/4-wire Serial Peripheral Interface" on page 240.

18. Display Serial Interface (DSI)

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

19. Graphics and the GFX Library on the Apollo4 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 registers in the applicable SoC's register set, which is included in the AmbiqSuite SDK.

19.1 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.

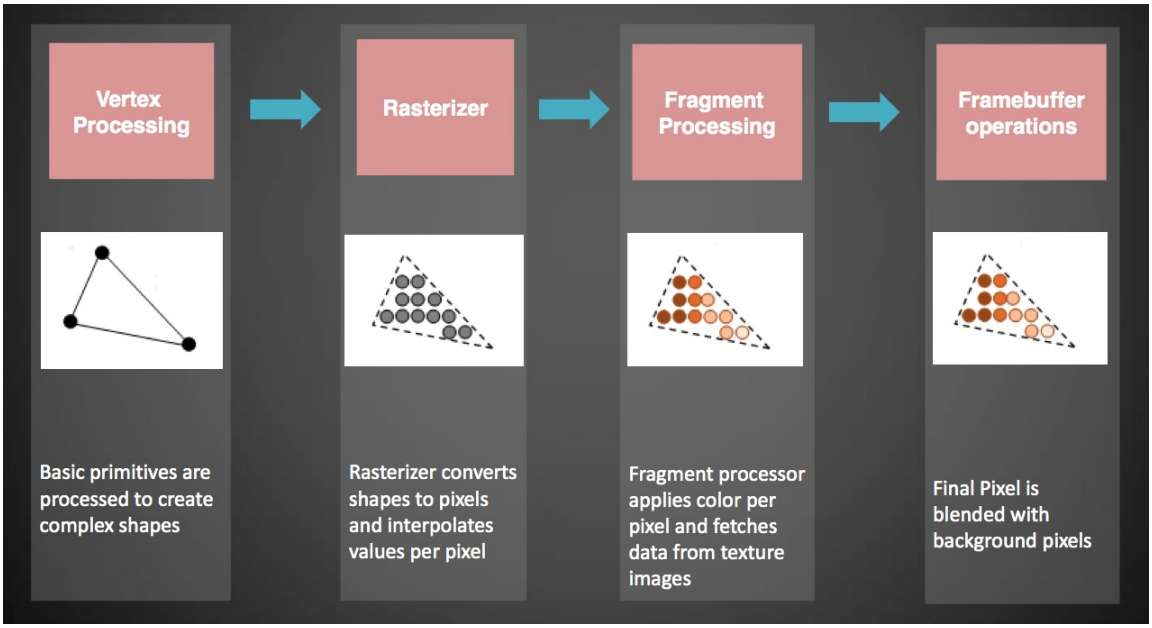


Figure 100. Rendering Flow

19.2 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
  - NEMA®|GUI-Builder

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:

- NEMA®|GUI-Builder, a graphical cross-platform software framework enabling rapid high-end Graphics User Interface (GUI) development on low resource hardware
- NEMA®| PIX-Presso, a utility software for converting images to/from formats suitable for low power embedded devices

19.3 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.

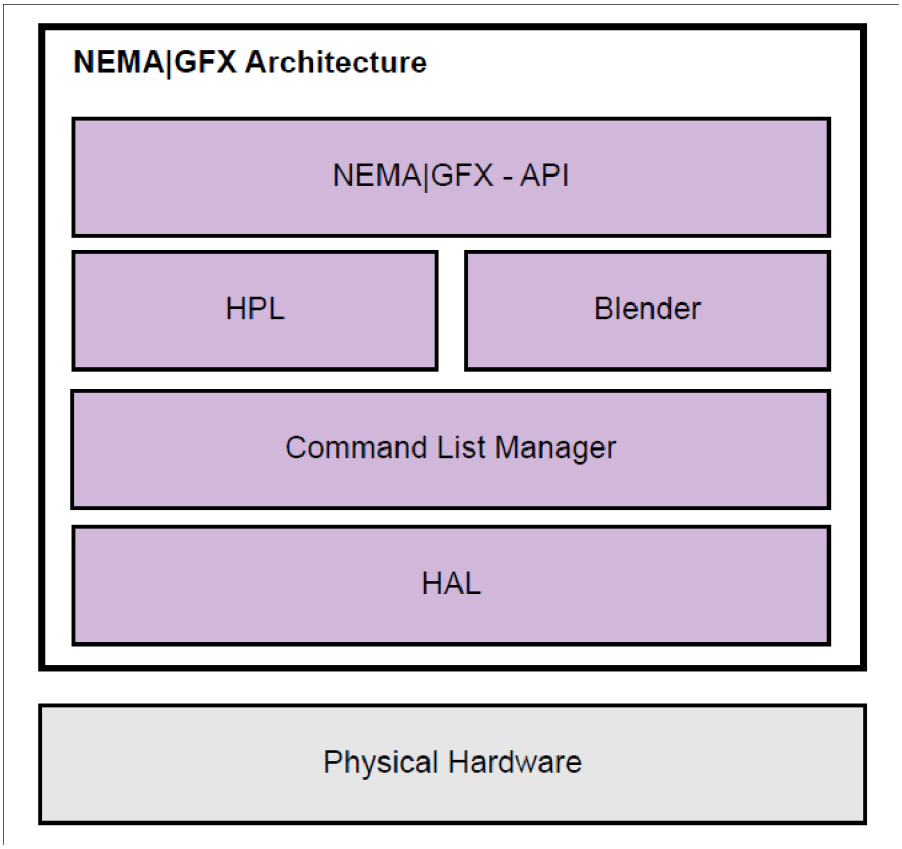


Figure 101. GFX Library Architecture

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 19.6.1 on page 267.

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.

19.4 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 utilize 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.

19.4.1 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 102. In accordance with Figure 102, 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.

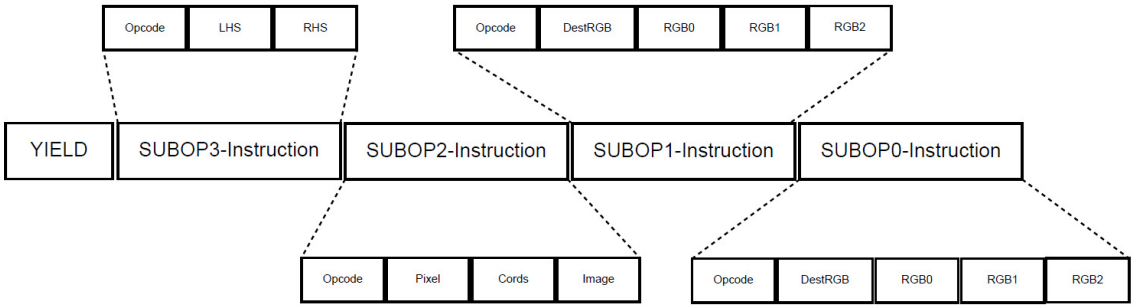


Figure 102. GPU VLIW fragment instruction format

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 103 in which the sub-operand A1 is set to 0x4 value which reads the value 1-alpha of R1 register.

19.4.1.1 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 100.

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.

Table 100: DestA Operand

The Alpha operations that can be executed are described in Table 101.

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

Table 101: SUBOP0-Instructions

The source operands are A0, A1 and A2. The configuration values are described in Table 102, Table 103 and Table 104, respectively.

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 102: A0 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 103: A1 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

Table 104: A2 Operand

19.4.1.2 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 105.

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 105: SUBOP1 Instructions

These instructions have one destination operand that can be written, see Table 106.

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 106: DestRGB Operand

The source operands are RGB0, RGB1 and RGB2. The accepted values are described in Table 107, Table 108 and Table 109, 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 107: 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 - Alpha) value from R0
0xd	R1	7:0	Read (1 - Alpha) value from R1
0xe	R2	7:0	Read (1 - Alpha) value from R2
0xf	R3	7:0	Read (1 - 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 108: 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 109: RGB2 Operand

19.4.1.3 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 110.

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 110: SUBOP2 Instructions

Table 111 provides a summary for Pixel Operand while Table 112 and Table 113 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 111: Pixel Operand

Value	Register	Bits	Summary for Coords Operand
0x0	iY, iX		Read iY, iX values for addressing.
0x1	i_R10	31:0	Read TY 4, TX 4 values for addressing.

Table 112: 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 113: TEXnBASE Coords Operand

19.4.1.4 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 114.

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 114: SUBOP3 Instructions

The source operands are LHS and RHS. The accepted values are described on Table 115.

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 115: LHS, RHS Operands

19.4.2 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).

19.4.3 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.

19.4.4 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.

19.4.5 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.

19.4.6 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.

19.4.7 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.

19.4.8 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:

$$Fc = Sc \cdot Sf + Dc \cdot Df \qquad Fa = Sa \cdot Sf + Da \cdot Df$$

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 119. Figure 110 shows the effect of the blending modes.

19.5 Frame Buffer Compression

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 4:1 compression (6bpp) 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 103 shows the TS compression operation. The output of the TSC™4 compression is 64 bits per 4x4 block of pixels and the output of the TSC™6 compression is 96 bits per 4x4 block of pixels.

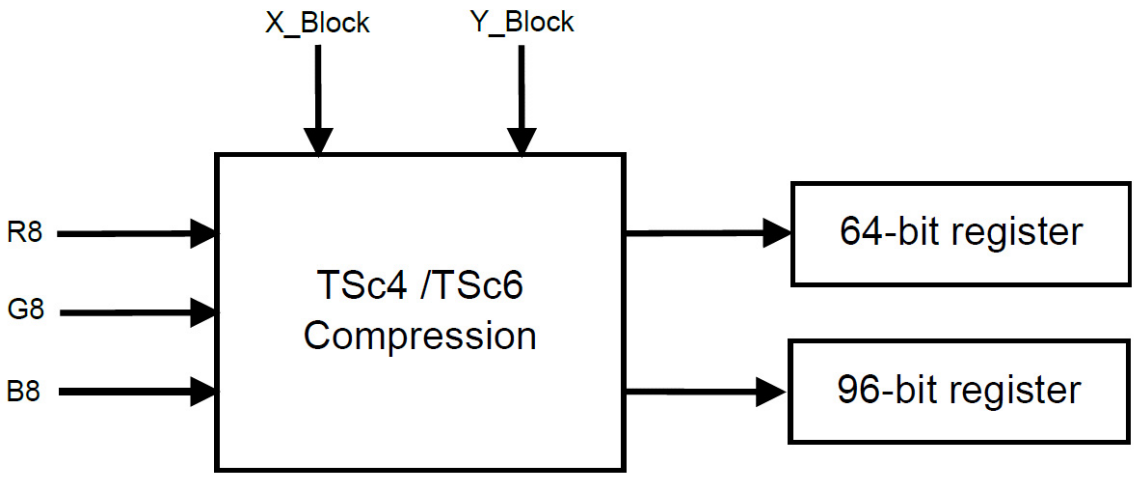


Figure 103. TSC™4 /TSC™6 Framebuffer Compression Module

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_ heigth> -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.

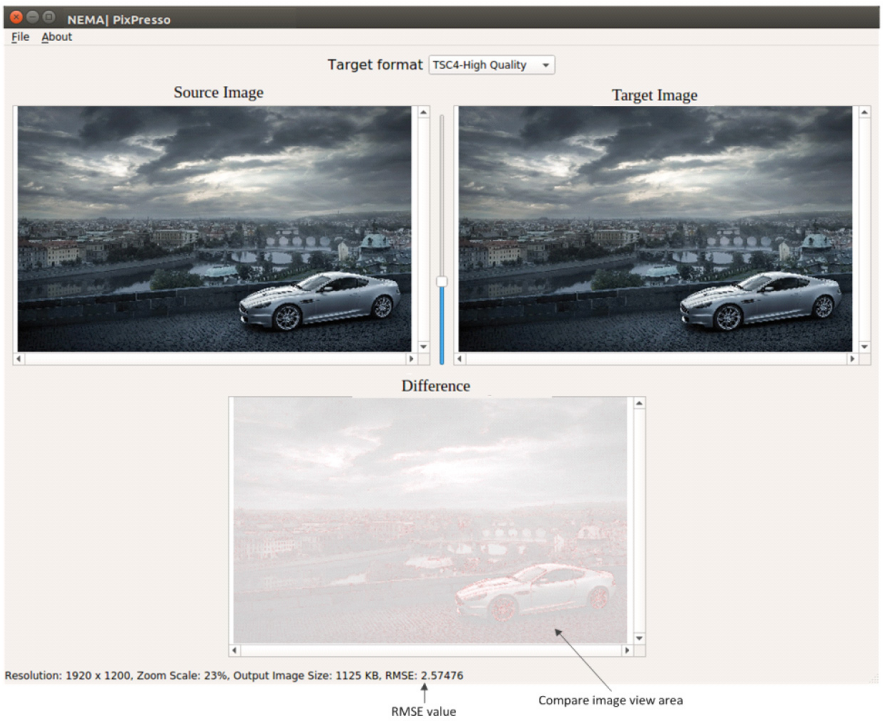


Figure 104. NEMA®| PIX-Presso State after a Conversion

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.

19.6 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.

19.6.1 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.

19.6.1.1 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.

19.6.1.2 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.

19.6.1.3 Unbind

```
nema_cl_unbind_cmdlist(void)
```

Unbind the currently bound Command List.

19.6.1.4 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
```

19.6.2 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
```

19.6.3 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
```

19.6.4 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
```

19.6.5 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.

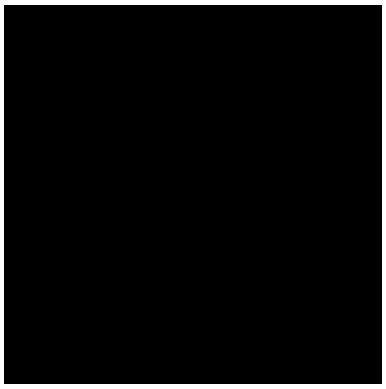


Figure 105. Original Empty Framebuffer

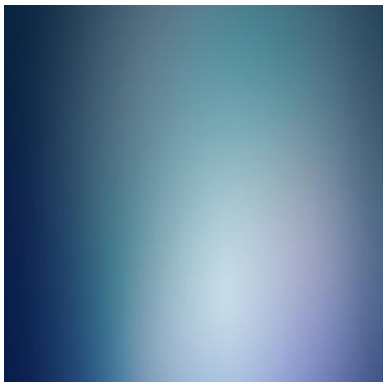


Figure 106. Image Background

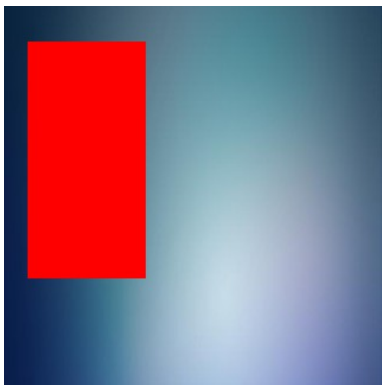


Figure 107. Final Output of the Drawing Process

19.7 Color Modes and Binding Textures

19.7.1 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 116 for quick reference purposes.

Color Mode	Description
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	4-bit proprietary compressed
YUV	YUV
Z24_8	32-bit (24+4) depth and stencil
Z16	16-bit depth

Table 116: Supported formats

19.7.2 Binding Textures

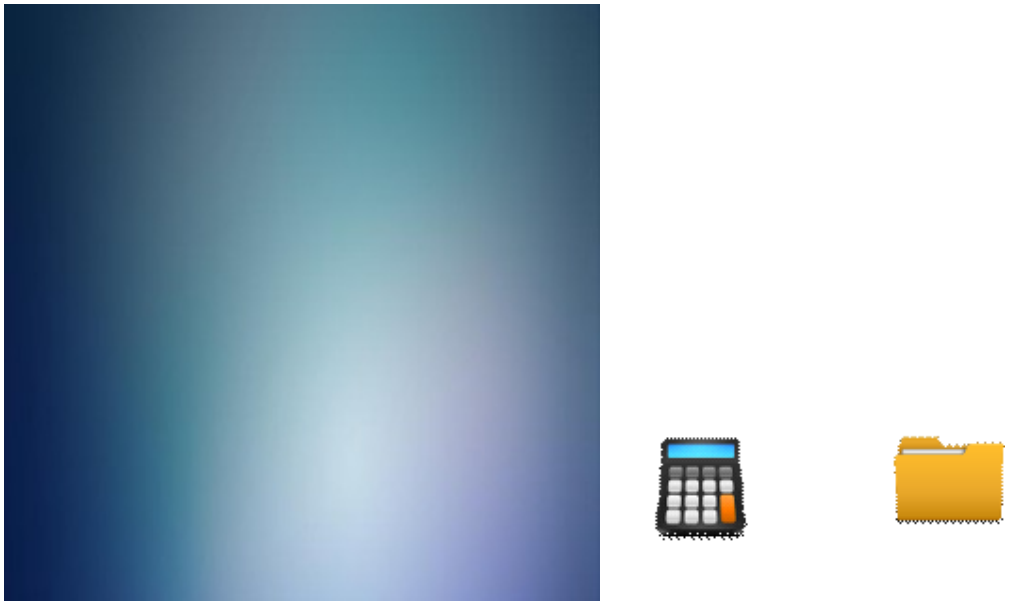
The color modes discussed in 19.7.1 can be used for both source and destination textures. The GPU incorporate 4 texture slots allowing 4 textures to be bound simultaneously. This means that the hardware allows a single Shader to read from and/or write to 4 different textures. These textures have to be 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 117.

Texture Slot	Texture Usage
NEMA_TEX0	Destination/Background Texture
NEMA_TEX1	Foreground Texture
NEMA_TEX2	Background Texture
NEMA_TEX3	Depth Buffer

Table 117: Shader conventions

For further clarifying the aforementioned conventions, let's assume the following example: We need to draw the scene shown in Figure 108, consisted of a background image and two icons. The scene requires the 3 source textures shown in Figure 109, and a Framebuffer, i.e., the destination texture.

Figure 108. Rendered Scene with Two Icons



Background Texture Icon_0 Icon_1

Figure 109. 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

19.7.2.1 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)
```

19.8 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

19.9 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.

19.9.1 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.

19.9.2 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)
```

19.9.3 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 118 shows the result for each Predefined Blending Mode.

Predefined Blending Modes	RGB	ALPHA
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 118: Predefined Blending Modes

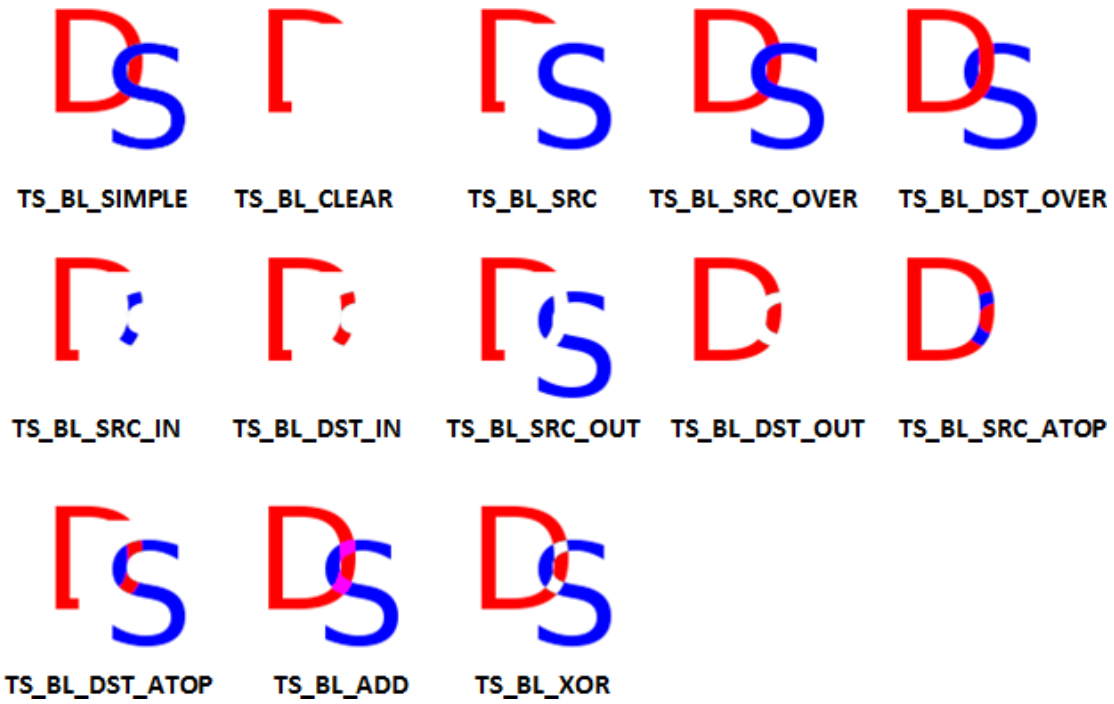


Figure 110. Predefined Blending Modes

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 111. Next, the process continues by blitting the textures shown in Figure 112.

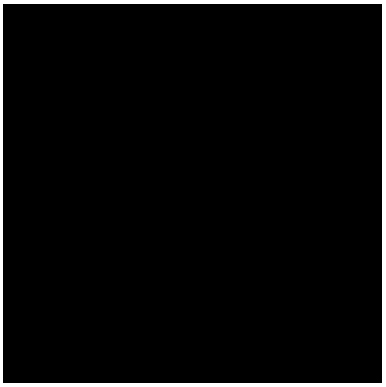


Figure 111. Original Framebuffer before Blending

Figure 112. Scene Textures

19.9.4 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 19.9.5 on page 282, therefore for the time being should be set to zero (0). The calculations result to the final color, using the following two equations:

$$Fc = Sc \cdot Sf + Dc \cdot Df \quad (1)$$

$$Fa = Sa \cdot Sf + Da \cdot Df \quad (2)$$

The available Blend Factors are listed in Table 119. Figure 113 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 (*Sf* or *Df*)

NEMA_BF_ZERO	0
NEMA_BF_ONE	1
NEMA_BF_SRCCOLOR	*Sc*
NEMA_BF_INVSRCCOLOR	(1 −*Sc*)
NEMA_BF_SRCALPHA	*Sa*
NEMA_BF_SRC_INVSRCALPHA	(1 −*Sa*)
NEMA_BF_DESTALPHA	*Da*

NEMA_BF_INVDESTALPHA	(1 −*Da*)
NEMA_BF_DESTCOLOR	*Dc*
NEMA_BF_INVDESTCOLOR	(1 −*Dc*)
NEMA_BF_CONSTCOLOR	*Cc*
NEMA_BF_CONSTALPHA	*Ca*

Table 119: Blend Factors

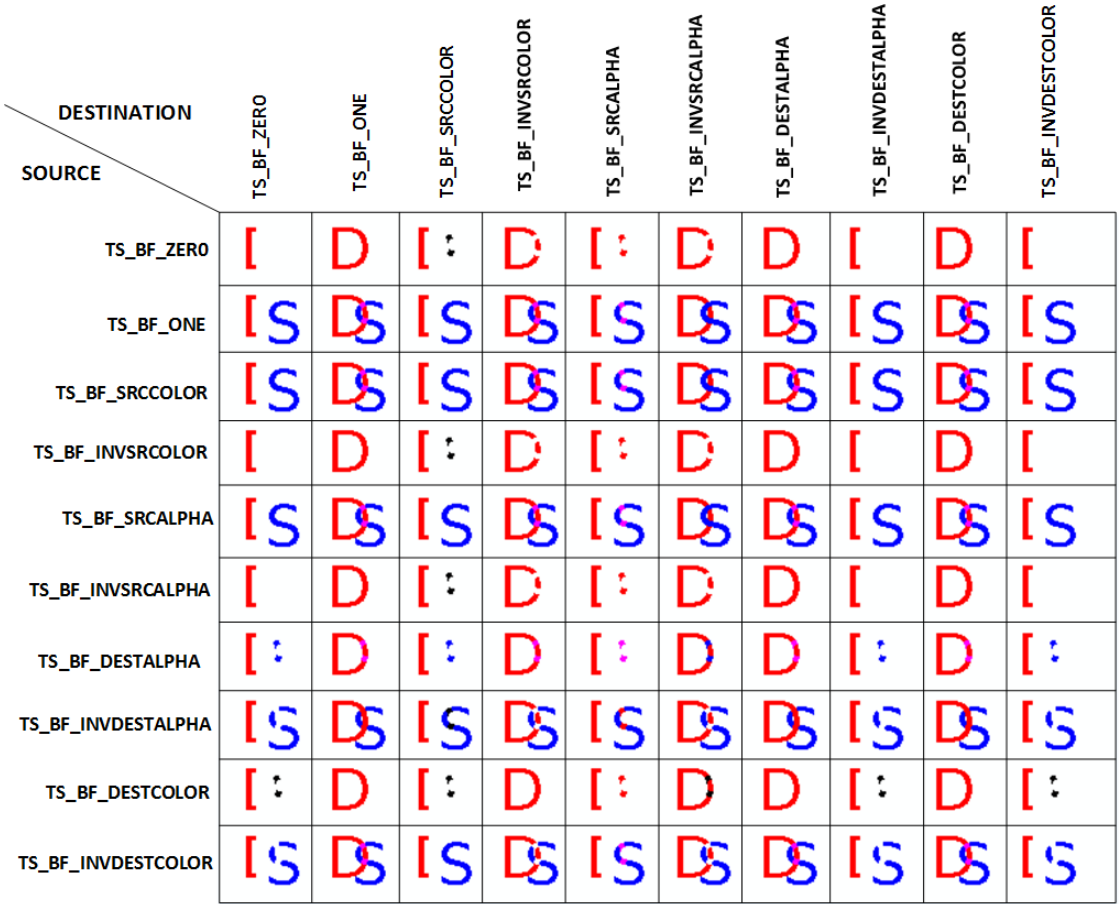


Figure 113. User-defined Blending Modes

19.9.5 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 120. An example of the overall process can be found in Figure 114 and Figure 115.

ops Arguments	Description
SRC_MODULATE_A	Multiply source alpha channel with *Ca* constant before blending. *Ca* is defined by calling NemaGFX_set_const_color().
SRC_FORCE_A	Replace source alpha channel with *Ca* before blending. Overrides SRC_MODULATE_A option. *Ca* is defined by calling NemaGFX_set_const_color().
SRC_COLORIZE	Multiply source color channels (RGB) with *Cc* before blending. *Cc* is defined by calling NemaGFX_set_const_color().
SRC_COLORKEY	Ignore fragment when source color matches the source color key, which is defined by calling NemaGFX_set_src_color_key().
DST_COLORKEY	Ignore fragment when destination color matches the destination color key, which is defined by calling NemaGFX_set_dst_color_key().

Table 120: ops Arguments

Figure 114. Source Textures

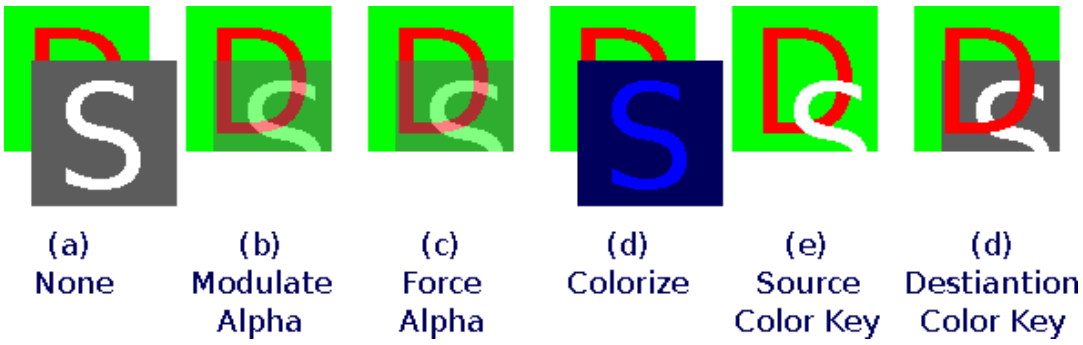


Figure 115. Additional Operations Example

19.10 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.

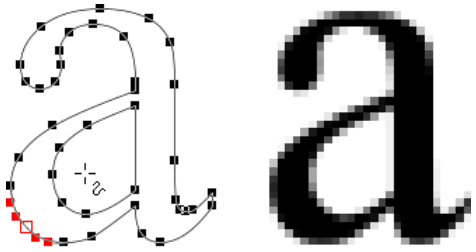


Figure 116. Vector and Bitmap Fonts

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.

19.11 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)

19.11.1 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.

19.11.2 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
```

19.11.3 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.

19.11.3.1 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

19.11.3.2 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);
}
```

19.11.3.3 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.

19.11.3.4 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
}
```

19.11.3.5 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
```

19.11.4 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.

19.11.4.1 nemadc_get_config

Read Configuration Register

Return Value:

uint32_t: return implemented configuration

19.11.4.2 nemadc_get_crc

Read CRC Checksum Register

Return Value:

unsigned int: return crc checksum of last frame for testing

19.11.4.3 nemadc_init

Initializes the Display Controller and structures

Return Value:

int: true if Display Controller is found

19.11.4.4 nemadc_set_bgcolor

Set Background Color before layer overlays

Arguments:

int: Colour as a 32-bit rgba value

Return Value:

void: void

19.11.4.5 nemadc_timing

Sets Display Timing parameters and activates display

Arguments:

int resx: Resolution X

int resy: Resolution Y

int fpx: front porch X

int fpy: front porch Y

int blx: blanking X

int bly: blanking Y

int bpx: back porch X

int bpy: back porch Y

Return Value:

void: void

19.11.4.6 nemadc_set_mode

Set Operation Mode

Arguments:

int mode: Mode of Operation (reg00)

Return Value:

void: void

19.11.4.7 nemadc_get_status

Get STATUS

Arguments:

void: void

Return Value:

uint32_t: Status

19.11.4.8 nemadc_request_vsync_non_blocking

Request a VSync Interrupt without blocking

Return Value:

void: void

19.11.4.9 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

19.11.4.10 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

19.11.4.11 nemadc_clkctrl

Control the Clock Gaters

Arguments:

int div: Sets the Clock Gater

Return Value:

void: void

19.11.4.12 nemadc_layer_enable

Enables a layer

Arguments:

int layer_no: layer to enable

Return Value:

void: void

19.11.4.13 nemadc_layer_disable

Disables a layer

Arguments:

int layer_no: layer to disable

Return Value:

void: void

19.11.4.14 nemadc_get_col_mode

Read Color Mode Register

Return Value:

uint32_t: return color mode register

19.11.4.15 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

19.11.4.16 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

19.11.4.17 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

19.11.4.18 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

19.11.4.19 nemadc_MIPI_out

Send command or data to MIPI Interface

Arguments:

int cmd: command

Return Value:

void: void

19.11.4.20 nemadc_MIPI_CFG_out

Send command or data to MIPI Interface

Arguments:

int cfg: configuration mode

Return Value:

void: void

19.11.4.21 nemadc_MIPI_in

Read data from MIPI Interface

Arguments:

void: void

Return Value:

void: void

19.11.4.22 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

19.12 GFX Library Functions

This section provides an overview of the implemented functions of the GFX library.

19.12.1 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.

19.12.1.1 Enumeration Type Documentation

19.12.1.1.1 enum nema_blend_factors_t

Enumerator

NEMA_BF_ZERO 0

NEMA_BF_ONE 1

NEMA_BF_SRCCOLOR Sc

NEMA_BF_INVSRCCOLOR (1-Sc)

NEMA_BF_SRCALPHA Sa

NEMA_BF_INVSRCALPHA (1-Sa)

NEMA_BF_DESTALPHA Da

NEMA_BF_INVDESTALPHA (1-Da)

NEMA_BF_DESTCOLOR Dc

NEMA_BF_INVDESTCOLOR (1-Dc)

NEMA_BF_CONSTCOLOR Cc

NEMA_BF_CONSTALPHA Ca

19.12.1.1.2 enum nema_blend_mode_t

Enumerator

NEMA_BL_SIMPLE	Sa * Sa + Da * (1 - Sa)
NEMA_BL_CLEAR	0
NEMA_BL_SRC	Sa
NEMA_BL_SRC_OVER	Sa + Da * (1 - Sa)
NEMA_BL_DST_OVER	Sa * (1 - Da) + Da
NEMA_BL_SRC_IN	Sa * Da
NEMA_BL_DST_IN	Da * Sa
NEMA_BL_SRC_OUT	Sa * (1 - Da)
NEMA_BL_DST_OUT	Da * (1 - Sa)
NEMA_BL_SRC_ATOP	Sa * Da + Da * (1 - Sa)
NEMA_BL_DST_ATOP	Sa * (1 - Da) + Da * Sa
NEMA_BL_ADD	Sa + Da
NEMA_BL_XOR	Sa * (1 - Da) + Da * (1 -Sa)

19.12.1.1.3 enum nema_blend_op_t

Enumerator

- *NEMA_BLOP_NONE*

  No extra blendingoperation

- *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

19.12.1.2 Function Documentation

19.12.1.2.1 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

src	Source Blending Factor
dst	Destination Blending Factor
ops	Additional Blending Operations

Returns

Final Blending Mode

19.12.1.2.2 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

blending_mode	Blending mode to be set
dst_tex	Destination Texture
fg_tex	Foreground (source) Texture
bg_tex	Background (source2) Texture

19.12.1.2.3 static void nema_set_blend_blit (uint32_t blending_mode) [inline], [static]

Set blending mode for blitting.

Parameters

blending_mode	Blending mode to be set

19.12.1.2.4 static void nema_set_blend_blit_compose (uint32_t blending_mode) [inline], [static]

Set blending mode for blitting with composing.

Parameters

blending_mode	Blending mode to be set

19.12.1.2.5 static void nema_set_blend_fill (uint32_t blending_mode) [inline], [static]

Set blending mode for filling.

Parameters

blending_mode	Blending mode to be set

19.12.1.2.6 static void nema_set_blend_fill_compose (uint32_t blending_mode) [inline], [static]

Set blending mode for filling with composing.

Parameters

blending_mode	Blending mode to be set

19.12.1.2.7 void nema_set_const_color (uint32_t rgba)

Set constant color.

Parameters

rgba	RGBA color

See Also

nema_rgba()

19.12.1.2.8 void nema_set_dst_color_key (uint32_t rgba)

Set destination color key.

Parameters

rgba	RGBA color key

See Also

nema_rgba()

19.12.1.2.9 static void nema_set_src_color_key (uint32_t rgba) [inline], [static]

Set source color key.

Parameters

rgba	RGBA color key

See Also

nema_rgba()

19.12.1.2.10 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.

19.12.1.3 Function Documentation

19.12.1.3.1 int nema_cl_add_cmd (uint32_t reg, uint32_t data)

Add a command to the bound Command List.

Parameters

reg	Hardware register to be written
data	Data to be written

Returns

0 if no error has occurred

*19.12.1.3.2 int nema_cl_add_multiple_cmds (int cmd_no, uint32_t * cmd)*

Add multiple commands to the bound Command List.

Parameters

cmd_no	Numbers of commands to add
cmd	Pointer to the commands to be added

Returns

0 if no error has occurred

*19.12.1.3.3 void nema_cl_bind (nema_cmdlist_t * cl)*

Define in which Command List each subsequent commands are going to be inserted.

Parameters

cl	Pointer to the Command List

*19.12.1.3.4 void nema_cl_branch (nema_cmdlist_t * cl)*

Branch from the bound Command List to a different one. Return is implied.

Parameters

cl	Pointer to the Command List to branch to

19.12.1.3.5 nema_cmdlist_t nema_cl_create (void)

Create a new expandable Command List. Returns

The instance of the new Command List

*19.12.1.3.6 nema_cmdlist_t nema_cl_create_prealloc (nema_buffer_t * bo)*

Create a new Command List into a preallocated space.

Parameters

addr_virt	Command List's address (preallocated)
size_bytes	Command List's size in bytes

Returns

The instance of the new Command List

19.12.1.3.7 nema_cmdlist_t nema_cl_create_sized (uint32_t size_bytes)

Create a new, non expandable Command List of specific size.

Parameters

size_bytes	Command List's size in bytes

Returns

The instance of the new Command List

*19.12.1.3.8 void nema_cl_destroy (nema_cmdlist_t * cl)*

Destroy/Free a Command List.

Parameters

cl	Pointer to the Command List

19.12.1.3.9 uint32_t nema_cl_get_space (int cmd_no)*

private

*19.12.1.3.10 void nema_cl_jump (nema_cmdlist_t * cl)*

Jump from the bound Command List to a different one. No return is implied.

Parameters

cl	Pointer to the Command List to jump to

19.12.1.3.11 void nema_cl_return (void)

Add an explicit return command to the bound Command List.

*19.12.1.3.12 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

cl	Pointer to the Command List

*19.12.1.3.13 void nema_cl_submit (nema_cmdlist_t * cl)*

Enqueue Command List to the Ring Buffer for execution.

Parameters

cl	Pointer to the Command List

19.12.1.3.14 void nema_cl_unbind (void)

Unbind current bound Command List, if any.

*19.12.1.3.15 void nema_cl_wait (nema_cmdlist_t * cl)*

Wait for Command List to finish.

Parameters

cl	Pointer to the Command List

19.12.1.3.16 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.

19.12.1.4 Function Documentation

?

19.12.1.4.1 float nema_ez (float A,float B,float steps,float cur_step,float()(float p)ez_-func)

Convenience function to perform easing between two values given number of steps, current step and easing function.

Parameters

A	Initial value within range [0, 1]
B	Finale value within range [0, 1]
steps	Total number of steps
cur_step	Current Step
ez_func	pointer to the desired easing function

Returns

Eased value

19.12.1.4.2 float nema_ez_back_in (float p)

Overshooting easing in, accelerate from zero.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.3 float nema_ez_back_in_out (float p)

Overshooting easing in and out, accelerate to halfway, then decelerate.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.4 float nema_ez_back_out (float p)

Overshooting easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.5 float nema_ez_bounce_in (float p)

Bouncing easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.6 float nema_ez_bounce_in_out (float p)

Bouncing easing in and out, accelerate to halfway, then decelerate.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.7 float nema_ez_bounce_out (float p)

Bouncing easing in, accelerate from zero.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.8 float nema_ez_circ_in (float p)

Circular easing in, accelerate from zero.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.9 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

19.12.1.4.10 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

19.12.1.4.11 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

19.12.1.4.12 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

19.12.1.4.13 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

19.12.1.4.14 float nema_ez_elast_in (float p)

Elastic easing in, accelerate from zero.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.15 float nema_ez_elast_in_out (float p)

Elastic easing in and out, accelerate to halfway, then decelerate.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.16 float nema_ez_elast_out (float p)

Elastic easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.17 float nema_ez_exp_in (float p)

Exponential easing in, accelerate from zero.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.18 float nema_ez_exp_in_out (float p)

Exponential easing in and out, accelerate to halfway, then decelerate.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.19 float nema_ez_exp_out (float p)

Exponential easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.20 float nema_ez_linear (float p)

Linear easing, no acceleration.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.21 float nema_ez_quad_in (float p)

Quadratic easing in, accelerate from zero.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.22 float nema_ez_quad_in_out (float p)

Quadratic easing in and out, accelerate to halfway, then decelerate.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.23 float nema_ez_quad_out (float p)

Quadratic easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.24 float nema_ez_quar_in (float p)

Quartic easing in, accelerate from zero.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.25 float nema_ez_quar_in_out (float p)

Quartic easing in and out, accelerate to halfway, then decelerate.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.26 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

19.12.1.4.27 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

19.12.1.4.28 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

19.12.1.4.29 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

19.12.1.4.30 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

19.12.1.4.31 float nema_ez_sin_in_out (float p)

Sinusoidal easing in and out, accelerate to halfway, then decelerate.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.1.4.32 float nema_ez_sin_out (float p)

Sinusoidal easing out, decelerate to zero velocity.

Parameters

p	Input value, typically within the [0, 1] range

Returns

Eased value

19.12.2 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.

19.12.2.1 Function Documentation

*19.12.2.1.1 void nema_bind_font (nema_font_t * font)*

Bind the font to use in future nema_print() calls.

Parameters

font	Pointer to font

*19.12.2.1.2 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

str	Pointer to string
x	X coordinate of the starting point. After execution, x contains the starting point for the next string to be drawn at.
y	Y coordinate of the starting point. After execution, y contains the starting point for the next string to be drawn at.
w	Width of the drawing area
h	Height of the drawing area
fg_col	Foreground color of text
align	Alignment and wrapping mode

*19.12.2.1.3 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

str	Pointer to string
w	Pointer to variable where width should be written
h	Pointer to variable where height should be written
max_w	Max allowed width

Returns

Number of carriage returns

19.12.3 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.

19.12.3.1 Enumeration Type Documentation

19.12.3.1.1 enum nema_rotation_t

Enumerator

NEMA_ROT_000_CCW	No rotation
NEMA_ROT_090_CCW	Rotate 90 degrees counter-clockwise
NEMA_ROT_180_CCW	Rotate 180 degrees counter-clockwise
NEMA_ROT_270_CCW	Rotate 270 degrees counter-clockwise
NEMA_ROT_000_CW	No rotation
NEMA_ROT_270_CW	Rotate 270 degrees clockwise
NEMA_ROT_180_CW	Rotate 180 degrees clockwise
NEMA_ROT_090_CW	Rotate 90 degrees clockwise
NEMA_MIR_VERT	Mirror Vertically
NEMA_MIR_HOR	Mirror Horizontally

19.12.3.2 Function Documentation

19.12.3.2.1 void nema_bind_depth_buffer (uint32_t baseaddr_phys, uint32_t width, uint32_t height)

Bind Depth Buffer.

Parameters

baseaddr_phys	Address of the depth buffer, as seen by the GPU
width	Buffer width
height	Buffer hight

19.12.3.2.2 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

baseaddr_phys	Address of the destination texture, as seen by the GPU
width	Texture width
height	Texture hight
format	Texture format
stride	Texture stride. If negative, it's calculated internally.

19.12.3.2.3 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

baseaddr_phys	Address of the source2 texture, as seen by the GPU
width	Texture width
height	Texture hight
format	Texture format
stride	Texture stride. If negative, it's calculated internally.
mode	Wrapping and Filtering mode

19.12.3.2.4 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

baseaddr_phys	Address of the source texture, as seen by the GPU
width	Texture width
height	Texture hight
format	Texture format
stride	Texture stride. If negative, it's calculated internally.
mode	Wrapping and Filtering mode

19.12.3.2.5 void nema_blit (int x, int y)

Blit source texture to destination texture.

Parameters

x	destination x coordinate
y	destination y coordinate

See Also

nema_set_blend_fill()

19.12.3.2.6 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

dx0	x coordinate at the first vertex of the quadrilateral
dy0	y coordinate at the first vertex of the quadrilateral

dx1	x coordinate at the second vertex of the quadrilateral
dy1	y coordinate at the second vertex of the quadrilateral
dx2	x coordinate at the third vertex of the quadrilateral
dy2	y coordinate at the third vertex of the quadrilateral
dx3	x coordinate at the fourth vertex of the quadrilateral
dy3	y coordinate at the fourth vertex of the quadrilateral

See Also

nema_set_blend_blit()

19.12.3.2.7 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

x	destination x coordinate
y	destination y coordinate
w	destination width
h	destination height

See Also

nema_set_blend_blit()

19.12.3.2.8 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

x	destination x coordinate
y	destination y coordinate
w	destination width
h	destination height

See Also

nema_set_blend_blit()

19.12.3.2.9 void nema_blit_rotate (int x, int y, nema_rotation_t rotation)

Rotate and Blit source texture to destination.

Parameters

x	destination x coordinate
y	destination y coordinate
rotation	Rotation to be done

See Also

nema_set_blend_blit()

19.12.3.2.10 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

sx	source upper left x coordinate
sy	source upper left y coordinate
sw	source width of partial region
sh	source height of partial region
x	destination x coordinate
y	destination y coordinate
rotation	Rotation to be done

See Also

nema_set_blend_blit()

19.12.3.2.11 void nema_clear (uint32_t rgba8888)

Clear destination texture with color.

Parameters

rgba8888	32-bit RGBA color

See Also

nema_rgba()

19.12.3.2.12 void nema_clear_depth (uint32_t val)

Clear depth buffer with specified value.

Parameters

val	Clear value

19.12.3.2.13 void nema_draw_circle (int x0, int y0, int r, uint32_t rgba8888)

Draw a colored circle.

Parameters

x0	x coordinate of the circle's center
y0	y coordinate of the circle's center
r	circle's radius
rgba8888	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

19.12.3.2.14 void nema_draw_line (int x0, int y0, int x1, int y1, uint32_t rgba8888)

Draw a colored line.

Parameters

x0	x coordinate at the beginning of the line
y0	y coordinate at the beginning of the line
x1	x coordinate at the end of the line
y1	y coordinate at the end of the line
rgba8888	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

19.12.3.2.15 void nema_draw_rect (int x, int y, int w, int h, uint32_t rgba8888)

Draw a colored rectangle.

Parameters

x	x coordinate of the upper left vertex of the rectangle
y	y coordinate at the upper left vertex of the rectangle
w	width of the rectangle
h	height of the rectangle
rgba8888	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

x0	x coordinate of the upper left vertex of the rectangle

y0	y coordinate at the upper left vertex of the rectangle
w	width of the rectangle
h	height of the rectangle
r	corner radius
rgba8888	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

19.12.3.2.16 void nema_emulate_p(void)

Emulate Nema|P functionality when using NemaGFX fragment shaders (blending modes) .

19.12.3.2.17 void nema_fill_circle (int x, int y, int r, uint32_t rgba8888)

Fill a circle with color.

Parameters

x	x coordinate of the circle's center
y	y coordinate of the circle's center
r	circle's radius
rgba8888	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

19.12.3.2.18 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

x0	x coordinate at the first vertex of the quadrilateral
y0	y coordinate at the first vertex of the quadrilateral
x1	x coordinate at the second vertex of the quadrilateral
y1	y coordinate at the second vertex of the quadrilateral
x2	x coordinate at the third vertex of the quadrilateral
y2	y coordinate at the third vertex of the quadrilateral
x3	x coordinate at the fourth vertex of the quadrilateral
y3	y coordinate at the fourth vertex of the quadrilateral
rgba8888	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

19.12.3.2.19 void nema_fill_rect (int x, int y, int w, int h, uint32_t rgba8888)

Fill a rectangle with color.

Parameters

x	x coordinate of the upper left vertex of the rectangle
y	y coordinate at the upper left vertex of the rectangle
w	width of the rectangle
h	height of the rectangle
rgba8888	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

19.12.3.2.20 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

h	height of the rectangle
r	corner radius
rgba8888	Color to be used

See Also

nema_set_blend_fill() nema_rgba()

19.12.3.2.21 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

x0	x coordinate at the first vertex of the triangle
y0	y coordinate at the first vertex of the triangle
x1	x coordinate at the second vertex of the triangle
y1	y coordinate at the second vertex of the triangle
x2	x coordinate at the third vertex of the triangle
y2	y coordinate at the third vertex of the triangle
rgba8888	Color to be used

See Also

nema_set_blend_fill()

19.12.3.2.22 int nema_init (void)

Initialize NemaGFX library.

Returns

-1 on error

19.12.3.2.23 uint32_t nema_premultiply_rgba (uint32_t rgba)

Pre-multiply RGB channels with Alpha channel.

Parameters

rgba	RGBA color

Returns

Pre-multiplied RGBA color

19.12.3.2.24 uint32_t nema_rgba (unsigned char R, unsigned char G, unsigned char B, unsigned char A)

Return Nema internal RGBA color.

Parameters

R	Red component
G	Green component
B	Blue component
A	Alpha component

Returns

RGBA value

19.12.4 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.

19.12.4.1 Function Documentation

19.12.4.1.1 nema_buffer_t nema_buffer_create (unsigned size)

Create memory buffer.

Parameters

size	Size of buffer in bytes

Returns

nema_buffer_t struct

19.12.4.1.2 nema_buffer_t nema_buffer_create_pool (int pool, unsigned size)

Create memory buffer at a specific pool.

Parameters

pool	ID of the desired memory pool
size	Size of buffer in bytes

Returns

nema_buffer_t struct

*19.12.4.1.3 void nema_buffer_destroy (nema_buffer_t * bo)*

Destroy/deallocate buffer.

Parameters

bo	Pointer to buffer struct

Returns

void

19.12.4.1.4 void nema_buffer_map (nema_buffer_t * bo)*

Maps buffer.

Parameters

bo	Pointer to buffer struct

Returns

Virtual pointer of the buffer (same as in bo->base_virt)

*19.12.4.1.5 uint32_t nema_buffer_phys (nema_buffer_t * bo)*

Get physical (GPU) base address of a given buffer.

Parameters

bo	Pointer to buffer struct

Returns

Physical base address of a given buffer

*19.12.4.1.6 void nema_buffer_unmap (nema_buffer_t * bo)*

Unmaps buffer.

Parameters

bo	Pointer to buffer struct

Returns

void

*19.12.4.1.7 void nema_host_free (void * ptr)*

Free memory previously allocated with nema_host_malloc()

Parameters

ptr	Pointer to allocated memory (virtual)

Returns

void

See Also nema_host_malloc()

19.12.4.1.8 void nema_host_malloc (unsigned size)*

Allocate memory for CPU to use (typically, standard malloc() is called)

Parameters

size	Size in bytes

Returns

Pointer to allocated memory (virtual)

See Also nema_host_free()

19.12.4.1.9 int nema_mutex_lock (int mutex_id)

Mutex Lock for multiple processes/threads.

Parameters

MUTEX_RB	or MUTEX_MALLOC

Returns

int

19.12.4.1.10 int nema_mutex_unlock (int mutex_id)

Mutex Unlock for multiple processes/threads.

Parameters

MUTEX_RB	or MUTEX_MALLOC

Returns

int

19.12.4.1.11 uint32_t nema_reg_read (uint32_t reg)

Read Hardware register.

Parameters

reg	Register to read

Returns

Value read from the register

See Also nema_reg_write

19.12.4.1.12 void nema_reg_write (uint32_t reg, uint32_t value)

Write Hardware Register.

Parameters

reg	Register to write
value	Value to be written

Returns

void()

See Also nema_reg_read()

19.12.4.1.13 int32_t nema_sys_init(void)

Initialize system. Implementor defined. Called in nema_init().

Parameters

void	

Returns

0 if no errors occurred

See Also nema_init()

19.12.4.1.14 void nema_wait_irq(void)

Wait for interrupt from the GPU.

Parameters

void	

Returns

void

19.12.4.1.15 void nema_wait_irq_cl (int cl_id)

Wait for a Command List to finish.

Parameters

cl_id	Command List ID

Returns

void

19.12.5 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.

19.12.5.1 Macro Definition Documentation

19.12.5.1.1 #define NEMA_1_PI

1/pi

19.12.5.1.2 #define NEMA_2_PI

2/pi

19.12.5.1.3 #define NEMA_2_SQRTPI

2/sqrt(pi)

19.12.5.1.4 #define nema_abs(a)

Calculate the absolute value.

Parameters

a	Value

Returns

The absolute value of a

19.12.5.1.5 #define nema_ceil(f)

Ceiling function.

Parameters

a	Value to be ceiled

Returns

ceiled value

19.12.5.1.6 #define nema_clamp(val,min,max)

Clamp value.

Parameters

val	Value to clamp
min	Minimum value
max	Minimum value

Returns

Clamped value

19.12.5.1.7 #define nema_deg_to_rad(d)

Convert degrees to radians.

Parameters

d	Angle in degrees

Returns

Angle in radians

19.12.5.1.8 #define NEMA_E

e

19.12.5.1.9 #define nema_f2fx(a)

Convert float to 16.16 fixed point.

Parameters

a	Value to be converted

Returns

16.16 fixed point value

19.12.5.1.10 #define nema_float_is_zero(x)

Checks if value x is zero.

Parameters

x	X value

Returns

1 if x == 0, 0 if x != 0

19.12.5.1.11 #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

19.12.5.1.12 #define nema_floor(f)

Floor function.

Parameters

a	Value to be floored

Returns

floored value

19.12.5.1.13 #define nema_i2fx(a)

Convert integer to 16.16 fixed point.

Parameters

a	Value to be converted

Returns

16.16 fixed point value

19.12.5.1.14 #define NEMA_LN10

ln(10)

19.12.5.1.15 #define NEMA_LN2

ln(2)

19.12.5.1.16 #define NEMA_LOG10E

log10(e)

19.12.5.1.17 #define NEMA_LOG2E

log2(e)

19.12.5.1.18 #define nema_max2(a,b)

Find the maximum of two values.

Parameters

a	First value
b	Second value

Returns

The maximum of a and b

19.12.5.1.19 #define nema_min2(a,b)

Find the minimum of two values.

Parameters

a	First value
b	Second value

Returns

The minimum of a and b

19.12.5.1.20 #define NEMA_PI

pi

19.12.5.1.21 #define NEMA_PI_2

pi/2

19.12.5.1.22 #define NEMA_PI_4

pi/4

19.12.5.1.23 #define nema_rad_to_deg(r)

Convert radians to degrees.

Parameters

r	Angle in radians

Returns

Angle in degrees

19.12.5.1.24 #define NEMA_SQRT1_2

1/sqrt(2)

19.12.5.1.25#define NEMA_SQRT2

sqrt(2)

19.12.5.2 Function Documentation

19.12.5.2.1 float nema_atan (float x)

A floating-point approximation of the inverse tangent of x.

Parameters

x	X value

Returns

Inverse tangent (angle) of x in degrees

19.12.5.2.2 float nema_cos (float angle_degrees)

Fast cosine approximation of a given angle.

Parameters

angle_degrees	Angle in degrees

Returns

Cosine of the given angle

19.12.5.2.3 float nema_pow (float x, float y)

A rough approximation of x raised to the power of y. USE WITH CAUTION!

Parameters

x	base value. Must be non negative.

y	power value

Returns

the result of raising x to the power y

19.12.5.2.4 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

19.12.5.2.5 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

19.12.5.2.6 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

19.12.6 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, float s)

  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.

19.12.6.1 Function Documentation

19.12.6.1.1 void nema_mat3x3_adj (nema_matrix3x3_t m)

Calculate adjoint.

Parameters

m	Matrix

19.12.6.1.2 void nema_mat3x3_div_scalar (nema_matrix3x3_t m, float s)

Divide matrix with scalar value.

Parameters

m	Matrix to divide
s	scalar value

19.12.6.1.3 int nema_mat3x3_invert (nema_matrix3x3_t m)

Invert matrix.

Parameters

m	Matrix to invert

19.12.6.1.4 void nema_mat3x3_load_identity (nema_matrix3x3_t m)

Load Identity Matrix.

Parameters

m	Matrix to be loaded

19.12.6.1.5 void nema_mat3x3_mirror (nema_matrix3x3_t m, int mx, int my)

Apply mirror transformation.

Parameters

m	Matrix to apply transformation
mx	if non-zero, mirror horizontally
my	if non-zero, mirror vertically

19.12.6.1.6 void nema_mat3x3_mul (nema_matrix3x3_t m, nema_matrix3x3_t _m)

Multiply two 3x3 matrices (m = m * _m)

Parameters

m	left matrix, will be overwritten by the result
m	right matrix

*19.12.6.1.7 void nema_mat3x3_mul_vec (nema_matrix3x3_t m, float * x, float * y)*

Multiply vector with matrix.

Parameters

m	Matrix to multiply with
x	Vector x coefficient
y	Vector y coefficient

*19.12.6.1.8void nema_mat3x3_mul_vec_affine (nema_matrix3x3_t m,float * x,float *y*

Multiply vector with affine matrix.

Parameters

m	Matrix to multiply with
x	Vector x coefficient
y	Vector y coefficient

19.12.6.1.9 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

width	Rectangle width
height	Rectangle height
sx0	x coordinate at the first vertex of the quadrilateral
sy0	y coordinate at the first vertex of the quadrilateral
sx1	x coordinate at the second vertex of the quadrilateral
sy1	y coordinate at the second vertex of the quadrilateral
sx2	x coordinate at the third vertex of the quadrilateral
sy2	y coordinate at the third vertex of the quadrilateral
sx3	x coordinate at the fourth vertex of the quadrilateral
sy3	y coordinate at the fourth vertex of the quadrilateral
m	Mapping matrix

19.12.6.1.10 void nema_mat3x3_rotate (nema_matrix3x3_t m, float angle_degrees)

Apply rotation transformation.

Parameters

m	Matrix to apply transformation
angle_degrees	Angle to rotate in degrees

19.12.6.1.11 void nema_mat3x3_shear (nema_matrix3x3_t m, float shx, float shy)

Apply shear transformation.

Parameters

m	Matrix to apply transformation
shx	X shearing factor
shy	Y shearing factor

19.12.6.1.12 void nema_mat3x3_translate (nema_matrix3x3_t m, float tx, float ty)

Apply translate transformation.

Parameters

m	Matrix to apply transformation
tx	X translation factor
ty	Y translation factor

19.12.7 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.

19.12.7.1 Function Documentation

19.12.7.1.1 void nema_mat4x4_load_identity (nema_matrix4x4_t m)

Load a 4x4 Identity Matrix.

Parameters

m	Matrix to be loaded

19.12.7.1.2 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

m	A 4x4 Matrix
left	Left vertical clipping plane
right	Right vertical clipping plane
bottom	bottom horizontal clipping plane
top	Top horizontal clipping plane
nearVal	Distance from the viewer to the near clipping plane (always positive)
farVal	Distance from the viewer to the far clipping plane (always positive)

19.12.7.1.3 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

m	A 4x4 Matrix
left	Left vertical clipping plane
right	Right vertical clipping plane
bottom	bottom horizontal clipping plane
top	Top horizontal clipping plane

19.12.7.1.4 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

m	A 4x4 Matrix
fovy_degrees	Field of View in degrees
aspect	Aspect ratio that determines the field of view in the x direction.
nearVal	Distance from the viewer to the near clipping plane (always positive)
farVal	Distance from the viewer to the far clipping plane (always positive)

19.12.7.1.5 void nema_mat4x4_mul (nema_matrix4x4_t m, nema_matrix4x4_tm_l, nema_-matrix4x4_t m_r)

Multiply two 4x4 matrices.

Parameters

m	Result Matrix
m_l	Left operand
m_r	Right operand

*19.12.7.1.6void 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

m	Matrix to be multiplied
x	Vector first element
y	Vector second element
z	Vector third element
w	Vector forth element

*19.12.7.1.7 intnema_mat4x4_obj_to_win_coords(nema_matrix4x4_tmvp, floatx_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

mvp	Model, View and Projection Matrix
x_orig	Window top left X coordinate
y_orig	Window top left Y coordinate
width	Window width
height	Window height
nearVal	Distance from the viewer to the near clipping plane (always positive)
farVal	Distance from the viewer to the far clipping plane (always positive)
x	X object coordinate
y	Y object coordinate
z	Z object coordinate

w	W object coordinate

Returns

1 if vertex is outside frustum (should be clipped)

19.12.7.1.8 void nema_mat4x4_rotate_X (nema_matrix4x4_t m, float angle_degrees)

Apply rotate transformation around X axis.

Parameters

m	Matrix to apply transformation
angle_degrees	Angle to rotate in degrees

19.12.7.1.9 void nema_mat4x4_rotate_Y (nema_matrix4x4_t m, float angle_degrees)

Apply rotate transformation around Y axis.

Parameters

m	Matrix to apply transformation
angle_degrees	Angle to rotate in degrees

19.12.7.1.10 void nema_mat4x4_rotate_Z (nema_matrix4x4_t m, float angle_degrees)

Apply rotate transformation around Z axis.

Parameters

m	Matrix to apply transformation
angle_degrees	Angle to rotate in degrees

19.12.7.1.11 void nema_mat4x4_scale (nema_matrix4x4_t m, float sx, float sy, float sz)

Apply scale transformation.

Parameters

m	Matrix to apply transformation
sx	X scaling factor
sy	Y scaling factor
sz	Z scaling factor

19.12.7.1.12 void nema_mat4x4_translate (nema_matrix4x4_t m, float tx, float ty, float tz)

Apply translate transformation.

Parameters

m	Matrix to apply transformation

tx	X	translation factor
ty	Y	translation factor
tz	Z	translation factor

19.12.8 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 memc?y 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 (int x0fx, int y0fx, int x1fx, int y1fx, int x2fx, int y2fx, int x3fx, int y3fx)

  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 ony 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.

19.12.8.1 Enumeration Type Documentation

19.12.8.1.1 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_RY RY

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

19.12.8.1.2 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

19.12.8.1.3 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

19.12.8.1.4 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

19.12.8.2 Function Documentation

19.12.8.2.1 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

texid	Texture unit to be programmed
addr_gpu	Texture's address as seen by the GPU
width	Texture's width
height	Texture's height
format	Texture's format
stride	Texture's stride. If stride < 0, it's left to be calculated
wrap_mode	Wrap/Repeat mode to be used

19.12.8.2.2 int nema_checkGPUPresence (void)

Check if a known GPU is present. Returns -1 if no known GPU is present

19.12.8.2.3 void nema_enable_tiling (uint32_t enable)

19.12.8.2.4 int nema_format_size (nema_tex_format_t format)

Return pixel size in bytes.

Parameters

format	Color format

Returns

Pixel size in bytes

19.12.8.2.5 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

minx	x coordinate of the upper left corner of the dirty region
miny	y coordinate of the upper left corner of the dirty region
maxx	x coordinate of the lower right corner of the dirty region
maxy	y coordinate of the lower right corner of the dirty region

?

19.12.8.2.6 void nema_load_frag_shader (const uint32_tcma, uint32_t count, uint32_t codeptr)

Load a precompiled Shader to the GPU's internal memory.

Parameters

cmd	Pointer to the shader
count	Number of commands
codeptr	Internal Memory address to be written (default is 0)

?

19.12.8.2.7 void nema_load_frag_shader_ptr (const uint32_tcm d, uint32_t count, uint32_t codeptr, uint32_t ptr)

Load a precompiled Shader to the GPU's internal memory and set fragment pointer.

Parameters

cmd	Pointer to the shader
count	Number of commands
codeptr	Internal Memory address to be written (default is 0)
ptr	Internal Memory address of the fragment shader

19.12.8.2.8 void nema_raster_line (int x0, int y0, int x1, int y1)

Program Rasterizer to draw a line.

Parameters

x0	x coordinate at the beginning of the line
y0	y coordinate at the beginning of the line
x1	x coordinate at the end of the line
y1	y coordinate at the end of the line

19.12.8.2.9 void nema_raster_pixel (int x, int y)

Program Rasterizer to generate a pixel.

Parameters

x	x coordinate of the pixel
y	y coordinate of the pixel

19.12.8.2.10 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

x0fx	x coordinate at the first vertex of the quadrilateral (fixed point 16.16)
y0fx	y coordinate at the first vertex of the quadrilateral (fixed point 16.16)
x1fx	x coordinate at the second vertex of the quadrilateral (fixed point 16.16)
y1fx	y coordinate at the second vertex of the quadrilateral (fixed point 16.16)
x2fx	x coordinate at the third vertex of the quadrilateral (fixed point 16.16)
y2fx	y coordinate at the third vertex of the quadrilateral (fixed point 16.16)
x3fx	x coordinate at the fourth vertex of the quadrilateral (fixed point 16.16)
y3fx	y coordinate at the fourth vertex of the quadrilateral (fixed point 16.16)

19.12.8.2.11 void nema_raster_rect (int x, int y, int w, int h)

Program Rasterizer to generate a triangle.

Parameters

x	x coordinate of the upper left vertex of the rectangle
y	y coordinate at the upper left vertex of the rectangle
w	width of the rectangle
h	height of the rectangle

19.12.8.2.12 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

x0	x coordinate of the upper left vertex of the rectangle
y0	y coordinate at the upper left vertex of the rectangle
w	width of the rectangle
h	height of the rectangle
r	corner radius

19.12.8.2.13 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)

19.12.8.2.14 void nema_raster_triangle_fx (int x0fx, int y0fx, int x1fx, int y1fx, int x2fx, int y2fx)

Program Rasterizer to draw a triangle.

Parameters

x0fx	x coordinate at the first vertex of the triangle (fixed point 16.16)
y0fx	y coordinate at the first vertex of the triangle (fixed point 16.16)
x1fx	x coordinate at the second vertex of the triangle (fixed point 16.16)
y1fx	y coordinate at the second vertex of the triangle (fixed point 16.16)
x2fx	x coordinate at the third vertex of the triangle (fixed point 16.16)
y2fx	y coordinate at the third vertex of the triangle (fixed point 16.16)

19.12.8.2.15 void nema_set_clip (uint32_t x, uint32_t y, uint32_t w, uint32_t h)

Sets the drawing area's Clipping Rectangle.

Parameters

x	Clip Window top-left x coordinate
y	Clip Window minimum y
w	Clip Window width
h	Clip Window height

19.12.8.2.16 void nema_set_const_reg (int reg, uint32_t value)

Write a value to a Constant Register of the GPU.

Parameters

reg	Constant Register to be written
value	Value to be written

19.12.8.2.17 void nema_set_depth_range (float min_depth, float max_depth)

Set maximum and minimum values for depth buffer. Available ony for Nema|T.

Parameters

min_depth	Minimum value
max_depth	Maximum value

19.12.8.2.18 void nema_set_frag_ptr (uint32_t ptr)

Set the Internal Memory address of the fragment shader to be executed.

Parameters

ptr	Internal Memory address of the fragment shader

19.12.8.2.19 void nema_set_matrix (nema_matrix3x3_t m)

Load GPU's Matrix Multiplier with a given 3x3 matrix.

Parameters

m	Matrix to be loaded

19.12.8.2.20 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

src_xres	Width of source rectangular area
src_yres	Height of source rectangular area
dst_xres	Width of destination rectangular area
dst_yres	Height of destination rectangular area
dst_x	X coordinate of upper-left vertex of the destination
dst_y	Y coordinate of upper-left vertex of the destination

19.12.8.2.21 void nema_set_matrix_translate (float dst_x, float dst_y)

Load GPU's Matrix Multiplier for a simple Blit (affine translation)

Parameters

dst_x	X coordinate of upper-left vertex of the destination
dst_y	Y coordinate of upper-left vertex of the destination

19.12.8.2.22 void nema_set_raster_color (uint32_t rgba8888)

Set the color which will be used when drawing primitives (lines, rectangles etc)

Parameters

rgba8888	Color to be used

See Also

nema_rgba()

19.12.8.2.23 void nema_set_rop_blend_mode (uint32_t bl_mode)

Set ROP blending mode.

Parameters

bl_mode	Blending mode

19.12.8.2.24 void nema_set_rop_const_color (uint32_t rgba)

Set ROP constant color.

Parameters

rgba	Constant color

See Also

nema_rgba()

19.12.8.2.25 void nema_set_rop_dst_color_key (uint32_t rgba)

Set ROP destination color key.

Parameters

rgba	Destination Color Key

See Also

nema_rgba()

19.12.8.2.26 void nema_set_tex_color (uint32_t color)

Set Texture Mapping default color.

Parameters

color	default color in 32-bit RGBA format

See Also

nema_rgba()

19.12.8.2.27 void nema_set_viewport (float x, float y, float w, float h)

Sets Viewport parameters for vertex shader. Available only for Nema|T.

Parameters

x	Start X coordinate
y	Start Y coordinate
w	Width of the Viewport
h	Height of the Viewport

19.12.8.2.28 int nema_stride_size (nema_tex_format_t format, int width)

Return stride in bytes.

Parameters

format	Color format
width	Texture color format

Returns

Stride in bytes

19.12.8.2.29 int nema_texture_size (nema_tex_format_t format, int width, int height)

Return texture size in bytes.

Parameters

format	Texture color format
width	Texture width
height	Texture height

Returns

Texture size in bytes

19.12.8.2.30 void nema_tri_cull (nema_tri_cull_t cull)

Set triangle/quadrilateral culling mode.

Parameters

cull	Culling mode

20. PDM-to-PCM Converter (PDM)

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

20.1 PDM Clock Configuration

There are 4 clocks or frequencies of relevance for the PDM interface:

- Interface core master clock, PDM_CLK, optimally set to 24.576 MHz.
- PDM output clock, PDM_CKO, whose frequency is selected to meet microphone specification.
- Desired sampling frequency at the microphone, Fsin, also determined by the requirements of the microphone.
- Desired sample rate, Fsout, 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_{sin} / F_{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 OSR = F_{sin} / Nyquist Rate = F_{sin} /(2 x BW) = F_{sin} / F_{sout}. For example, 8 kHz BW, sampled at 768 kHz, has an OSR of 768 kHz / 16 kHz= 48.

The OSR is selected by setting the SINC decimation rate in the SINCRATE field in the PDMn_CORECFG0 register, such that OSR = 2 x SINCRATE. The PDM_CLK is determined by dividing down the PDM_APB_CLK, which is 96 MHz. The divider is selected in the CLKSEL field of the PDMn_CTRL register. If the PDMn_CLK is desired to be 24.576 MHz, then CLKSEL should set to 1.

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 Apollo4 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_S and OSR for the various operating modes, at a PDM_CLK of 24.576 MHz.

Table 121: PDMA_CKO and OSR Settings for Different Sampling Frequencies

OPERATING MODE	F_{PDMA_CKO} (MHz)	F_S (kHz)	OSR	DIV_ MCLKQ [1:0]	MCLKDIV [3:0]	SINCRATE [6:0]	SINAD (dB)	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 master clock rate and digital microphones implemented in system design.
3. **SINAD** means ratio of signal to noise plus the first N harmonics of THD.
4. **DR** means dynamic range, which is measured as SINAD - (-60dB) 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 OSR = F_{sin} / F_{sout}, OSR should typically be between 32 and 128, or 192.
- To select the SINC rate to be equal to 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 PDMn_CLKO = PDMn_MCLKQ / (PDMn_MCLKDIV+1).

20.2 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 122: PDM-to-PCM Converter Operating Mode

Mode	CORE-CFG1_PCM-CHSET	CORE-CFG0_LRSWAP	CTRL_EN	PDMA_CKO	CTRL_CLKEN	CTRL_RSTB	PCMA_DATA_L	PCMA_DATA_R	PCMA_VALID
Stereo Non- swap	2'b11	0	1	Active	Active	1	Normal Left	Normal Right	Normal
Stereo Swap	2'b11	1	1	Active	Active	1	Normal Right Swapped	Normal Left Swapped	Normal
Mono Left	2'b01	0	1	Active	Active	1	Normal Left	0	Normal
Mono Right	2'b10	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 master 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.

20.3 Supported Data Formats

There are four data formats supported by the PDM module.

- 24-bit unpacked
- 16-bit unpacked
- 16-bit packed
- 8-bit packed

The 16-bit (packed or unpacked) format is 24-bit input data truncated down to 16 bits: Data[23:0] => Data [23:8]. The 8-bit packed format is 24-bit input data truncated down to 8 bits: Data[23:0] => Data [23:16].

The following tables show the word sequences for each of the four data formats for single-channel (A), dual-channel (A and B) and 8-channel (A-H) audio.

Table 123: 24-bit Unpacked Data

Word Sequence	Single-channel		Dual-channel		8-channel	
	Upper 8 bits	Lower 24 bits	Upper 8 bits	Lower 24 bits	Upper 8 bits	Lower 24 bits
1	0x00	A_t0[23:0]	0x00	A_t0[23:0]	0x00	A_t0[23:0]
2	0x00	A_t1[23:0]	0x00	B_t0[23:0]	0x00	B_t0[23:0]
3	0x00	A_t2[23:0]	0x00	A_t1[23:0]	0x00	C_t0[23:0]
4	0x00	A_t3[23:0]	0x00	B_t1[23:0]	0x00	D_t0[23:0]
5	0x00	A_t4[23:0]	0x00	A_t2[23:0]	0x00	E_t0[23:0]
6	0x00	A_t5[23:0]	0x00	B_t2[23:0]	0x00	F_t0[23:0]
7	0x00	A_t6[23:0]	0x00	A_t3[23:0]	0x00	G_t0[23:0]
8	0x00	A_t7[23:0]	0x00	B_t3[23:0]	0x00	H_t0[23:0]

Table 124: 16-bit Unpacked Data

Word Sequence	Single-channel		Dual-channel		8-channel	
	Upper 16 bits	Lower 16 bits	Upper 16 bits	Lower 16 bits	Upper 16 bits	Lower 16 bits
1	0x0000	A_t0[23:8]	0x0000	A_t0[23:8]	0x0000	A_t0[23:8]
2	0x0000	A_t1[23:8]	0x0000	B_t0[23:8]	0x0000	B_t0[23:8]
3	0x0000	A_t2[23:8]	0x0000	A_t1[23:8]	0x0000	C_t0[23:8]
4	0x0000	A_t3[23:8]	0x0000	B_t1[23:8]	0x0000	D_t0[23:8]
5	0x0000	A_t4[23:8]	0x0000	A_t2[23:8]	0x0000	E_t0[23:8]
6	0x0000	A_t5[23:8]	0x0000	B_t2[23:8]	0x0000	F_t0[23:8]
7	0x0000	A_t6[23:8]	0x0000	A_t3[23:8]	0x0000	G_t0[23:8]
8	0x0000	A_t7[23:8]	0x0000	B_t3[23:8]	0x0000	H_t0[23:8]

Table 125: 16-bit Packed Data

Word Sequence	Single-channel		Dual-channel		8-channel	
	Upper 16 bits	Lower 16 bits	Upper 16 bits	Lower 16 bits	Upper 16 bits	Lower 16 bits
1	A_t1[23:8]	A_t0[23:8]	B_t0[23:8]	A_t0[23:8]	B_t0[23:8]	A_t0[23:8]
2	A_t3[23:8]	A_t2[23:8]	B_t1[23:8]	A_t1[23:8]	D_t0[23:8]	C_t0[23:8]
3	A_t5[23:8]	A_t4[23:8]	B_t2[23:8]	A_t2[23:8]	F_t0[23:8]	E_t0[23:8]
4	A_t7[23:8]	A_t6[23:8]	B_t3[23:8]	A_t3[23:8]	H_t0[23:8]	G_t0[23:8]
5	A_t9[23:8]	A_t8[23:8]	B_t4[23:8]	A_t4[23:8]	B_t1[23:8]	A_t1[23:8]
6	A_t11[23:8]	A_t10[23:8]	B_t5[23:8]	A_t5[23:8]	D_t1[23:8]	C_t1[23:8]
7	A_t13[23:8]	A_t12[23:8]	B_t6[23:8]	A_t6[23:8]	F_t1[23:8]	E_t1[23:8]
8	A_t15[23:8]	A_t14[23:8]	B_t7[23:8]	A_t7[23:8]	H_t1[23:8]	G_t1[23:8]

Table 126: 8-bit Packed Data - Single Channel

Word Sequence	Single-channel			
	Byte3	Byte2	Byte1	Byte0
1	A_t3[23:16]	A_t2[23:16]	A_t1[23:16]	A_t0[23:16]
2	A_t7[23:16]	A_t6[23:16]	A_t5[23:16]	A_t4[23:16]
3	A_t11[23:16]	A_t10[23:16]	A_t9[23:16]	A_t8[23:16]
4	A_t15[23:16]	A_t14[23:16]	A_t13[23:16]	A_t12[23:16]
5	A_t19[23:16]	A_t18[23:16]	A_t17[23:16]	A_t16[23:16]
6	A_t23[23:16]	A_t22[23:16]	A_t21[23:16]	A_t20[23:16]
7	A_t27[23:16]	A_t26[23:16]	A_t25[23:16]	A_t24[23:16]
8	A_t31[23:16]	A_t30[23:16]	A_t29[23:16]	A_t28[23:16]

Table 127: 8-bit Packed Data - Dual Channels

Word Sequence	Dual-channel			
	Byte3	Byte2	Byte1	Byte0
1	B_t1[23:16]	A_t1[23:16]	B_t0[23:16]	A_t0[23:16]
2	B_t3[23:16]	A_t3[23:16]	B_t2[23:16]	A_t2[23:16]
3	B_t5[23:16]	A_t5[23:16]	B_t4[23:16]	A_t4[23:16]
4	B_t7[23:16]	A_t7[23:16]	B_t6[23:16]	A_t6[23:16]
5	B_t9[23:16]	A_t9[23:16]	B_t8[23:16]	A_t8[23:16]
6	B_t11[23:16]	A_t11[23:16]	B_t10[23:16]	A_t10[23:16]
7	B_t13[23:16]	A_t13[23:16]	B_t12[23:16]	A_t12[23:16]
8	B_t15[23:16]	A_t15[23:16]	B_t14[23:16]	A_t14[23:16]

Table 128: 8-bit Packed Data - 8 Channels

Word Sequence	8-channel			
	Byte3	Byte2	Byte1	Byte0
1	D_t0[23:16]	C_t0[23:16]	B_t0[23:16]	A_t0[23:16]
2	H_t0[23:16]	G_t0[23:16]	F_t0[23:16]	E_t0[23:16]
3	D_t1[23:16]	C_t1[23:16]	B_t1[23:16]	A_t1[23:16]
4	H_t1[23:16]	G_t1[23:16]	F_t1[23:16]	E_t1[23:16]
5	D_t2[23:16]	C_t2[23:16]	B_t2[23:16]	A_t2[23:16]
6	H_t2[23:16]	G_t2[23:16]	F_t2[23:16]	E_t2[23:16]
7	D_t3[23:16]	C_t3[23:16]	B_t3[23:16]	A_t3[23:16]
8	H_t3[23:16]	G_t3[23:16]	F_t3[23:16]	E_t3[23:16]

20.4 Digital Volume Control

The PDM supports digital volume-control with a PGA gain range from -12dB to +34.5dB in 1.5dB/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 0.13dB or 0.26dB internally.

Table 129: PGA Gain Control

Register	Default	Description
CORECFG0_ PGAL[4:0]	01000	Left Channel PGA Gain: +1.5dB/step, -12dB ~ +34.5dB
		00000 = -12dB; 00001 = -10.5dB; ...
		01000 = 0dB; …
		11111= +34.5dB
CORECFG0_ PGAR[4:0]	01000	Right Channel PGA Gain: +1.5dB/step, -12dB ~ +34.5dB
		00000 = -12dB; 00001 = -10.5dB; ...
		01000 = 0dB; …
		11111= +34.5dB
CORECFG1_ SELSTEP	0	Fine grain step size for smooth PGA or Softmute attenuation transition: 0 = 0.13dB step size 1 = 0.26dB step size

20.4.1 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 130: 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 100 = 192 steps 001 = 96 steps 101 = 256 steps 010 = 128 steps 110 = 384 steps 011 = 160 steps 111 = 512 steps
CORECFG1_SELSTEP	1	Sets fine gain step for smooth PGA or Soft-Mute attenuation transition. 0 = 0.13dB 1 = 0.26dB

20.5 Low Pass Filter (LPF)

The PDM's internal low pass filters attenuate the out-of-band noise at predefined band width and corners. Table 131 below describes the parameters of LPF performance.

Table 131: 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

20.6 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:

$$H(Z) = (1 - Z^{-1}) / [1 - (1 - 2^{-HPGAIN}) Z^{-1}]$$

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.

21. Low Power Analog Audio Interface

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

21.1 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 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. Table 132 shows the format that is used by all conversions. This is a scaled integer format with a 6-bit fractional part. 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.

Table 132: 14.6 Audio ADC Sample Format

19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
14-bit Integer														6-bit Fraction					

Each slot contains a 21-bit accumulator as shown in Table 133, "Per Slot Sample Accumulator," on page 368. 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:

$128*(2^{14}-1) = 128*16383 = 2097024 = 2^{21} - 128$, hence the 21 bit accumulator.

Table 133: Per Slot Sample Accumulator

20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Accumulator																				

Table 134 shows the maximum possible accumulated values. Note that 64 sample accumulation produces a result that is exactly correct or the 14.6 format results so it is copied unscaled in to the FIFO.

Furthermore, note that 128 sample accumulation can produce a result that is too large for the 14.6 format since it may result in 7 bits of valid fractional data. All of the remaining sample accumulation settings must have their results left shifted to produce the desired 14.6 format.

Finally, note that for the 128 sample accumulation case, the LSB of the accumulator is discarded when the results are written to the FIFO.

Most importantly, note that for the 1 sample accumulation case, the 14-bit converter value is shifted left by six to produce the 14.6 format to write into the FIFO.

Table 134: Accumulator Scaling

# Samples	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
128	14.6																				0
64	X	14.6																			
32	X	X	14.5																		
16	X	X	X	14.4																	
8	X	X	X	X	14.3																
4	X	X	X	X	X	14.2															
2	X	X	X	X	X	X	14.1														
1	X	X	X	X	X	X	X	14													

21.2 Sixteen Entry Result FIFO

All results written to the FIFO have exactly the same format as shown in Table 135. The properly scaled accumulation results are written the lower half word in the aforementioned 14.6 format. Since each slot can produce results at a different rate, the slot number generating the result is also written to the FIFO along with the total valid entry count within the FIFO.

Table 135: FIFO Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
RSV	Slot Number.			FIFO Count								FIFO DATA																			

Table 136: 12-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	12.6																	
64	0	0	12.6																	
32	0	0	12.5																	X
16	0	0	12.4																X	X

Table 136: 12-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
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

Table 137: 10-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	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 138: 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
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
1	0	0	0	0	0	0	8								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 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 FIFOOVR1 (FIFO 75% full) interrupt status bit will be set. When a conversion completes resulting in hardware populating the 16th valid FIFO entry, the FIFOOVR2 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.

21.3 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 deepsleep.

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 deepsleep 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 deepsleep 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.

21.4 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 Audio ADC 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 Audio ADC 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 Audio ADC 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 139: Window Comparator Lower Limit Register

19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Lower Limit																			

Table 140: Window Comparator Upper Limit Register

19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
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 Audio ADC sample is inside if the following relation is true:

14.6 Lower Limit <= ADC SAMPLE <= 14.6 Upper Limit

Thus setting both limits to the same value, say 700.0 (0x2BC<<6 = 0xAF00), will only produce an inside interrupt when the Audio ADC 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 Audio ADC will be inside the window limits and no excursion interrupts can ever by 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 SCWLIM register.

21.5 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 reseting 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 117

An Audio ADC conversion is the process of averaging measurements following one or more scans for each slot that is enabled.

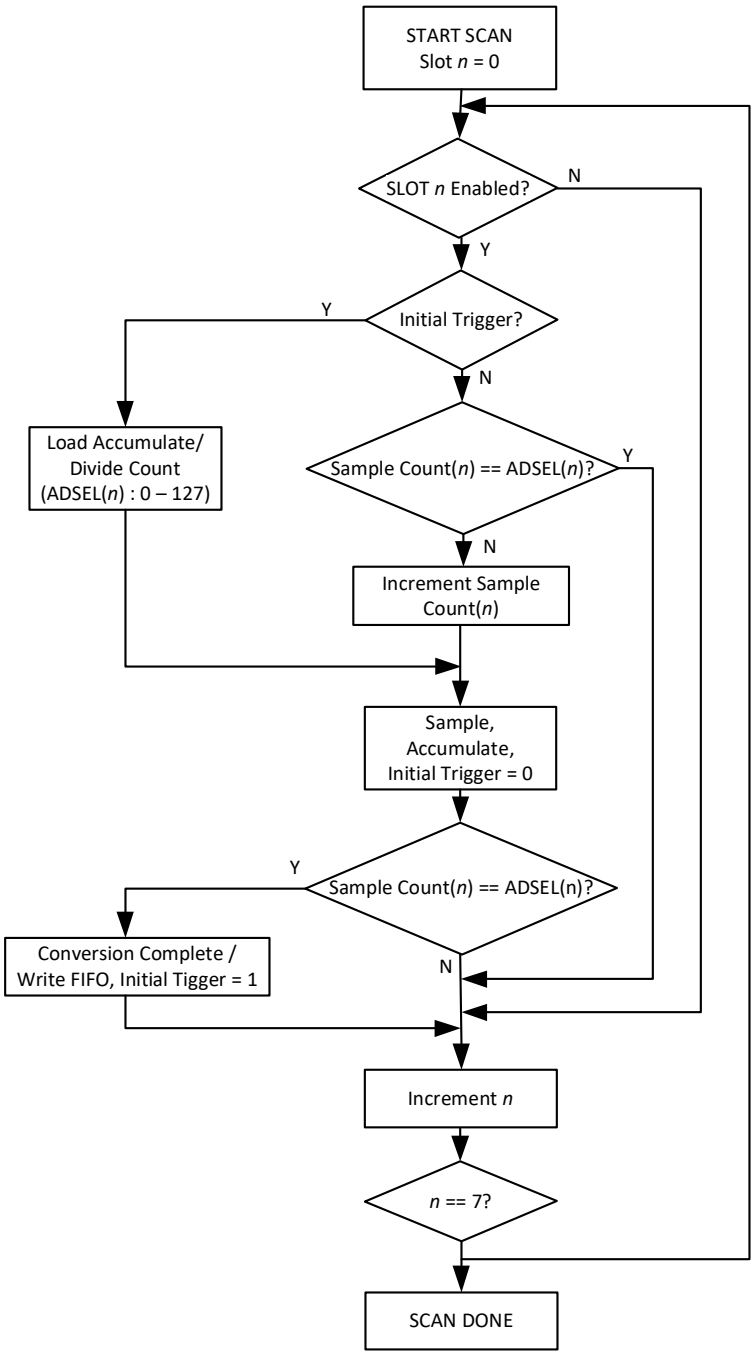


Figure 117. Scan Flowchart

21.5.1 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.

21.5.2 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.

21.5.3 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 141: Audio ADC Power Modes

LPMODE	Definition	Entry Latency
0	Audio ADC is kept active continuously (used in continuous sampling scenarios)	0 (requires initial calibration)
1	Audio ADC is mostly powered off between samples, HFRC is duty cycled between samples. No calibration required after initial calibration)	<70µs (shorter for lower resolution)
2	Audio ADC is completely powered off between samples, HFRC is duty cycled between samples. Requires recalibration for each conversion.	<660µs

21.5.3.1 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 triggers[1].

21.5.3.2 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 while maintaining Audio ADC calibration data. This mode may operate autonomously without CPU interaction, even while the CPU is in sleep or deepsleep 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

21.5.3.3 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 and a calibration sequence 100's of microseconds, nominally. 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.

21.6 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. 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)
9. DMA transfer complete
10. DMA error condition

The window comparator interrupts are discussed above. See "Window Comparator" on page 372.

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

21.7 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.

22. Inter-IC Sound (I^2S)

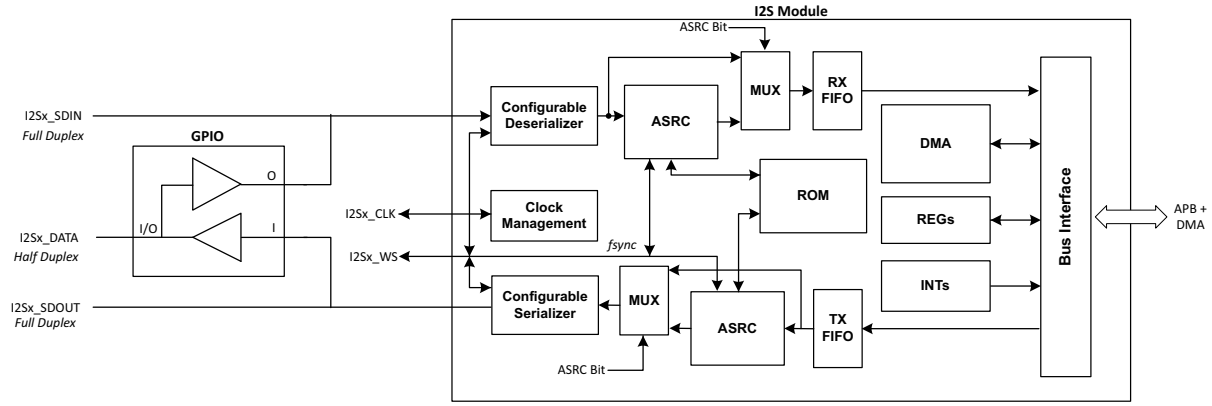


Figure 118. I^2S Block Diagram

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

22.1 I2S Clock Management

Clock Management consists of master clock source selection and the two-stage divider configuration.

> NOTE
>
> Regarding the selection of the HFRC2 to clock the I^2S module, there is the possibility of an asynchronous shutdown of the HFRC2 clock divider by the internal hardware, causing a glitch when the requesting peripheral stops requesting the HFRC2.
>
> The HFRC2 must be forced on not only when HFRC2 is being selected and while being used as the clock source but also whenever the clock source is being changed regardless of the new clock source being selected.
>
> The HFRC2 is forced on by setting the CLKGEN_MISC_FRCHFRC2 bit. The sequence for changing the clock source regardless of clock selection is to first force HFRC2 on by setting the CLKGEN_MISC_FRCHFRC2 bit, select the clock source for the module, clear the CLKGEN_MISC_FRCHFRC2 bit only if HFRC2 is NOT selected, and then engage the peripheral.
>
> If HFRC2 is the clock source, then shutting the module down cleanly requires switching to HFRC, for example, and then disabling the HFRC2 by clearing the CLKGEN_MISC_FRCHFRC2 bit.

The Receive FIFO and Transmit FIFO decouple the I^2S clock domain and the back-end bus interface clock domain. All FIFO depths are configurable, and their status and current number of samples are accessible by means of memory mapped registers and ports. 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.

22.2 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 I^2S 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 I^2S module or the system fabric, and sent to either the I^2S module or system fabric.

Enabling the DMA operation requires setup in both sub modules. The I^2S module must be programmed with the I^2S 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 I^2S module based on FIFO space availability (TX) or filled level (RX). These two FIFOs in the I^2S module are separate and independent. Two registers, TXLOWERLIMIT register and the RXUPPERLIMIT register, are used to program the interrupt condition. 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 programmed into 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. RXUPPERLIMIT must not be programmed to a value less the 8, as this could cause underruns. The FIFO size for both the TX and RX FIFOs is 64 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 I^2S 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 DMA ADDRESS 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.

22.3 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.

22.4 Data Configurations

Table 142 lists the configurations supported by the I2S module.

Table 142: 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
Master/Slave modes	master or slave	slave[1]
I2S/TDM modes	Any supported by the I2S, with the limitations indicated above	Any supported by the I2S, with the limitations indicated above

1. The I2S also can be configured in master mode. However, since the *mclk* signal is used to generate the *sclk* signal, the source ratio will be always 1:1.

Figure 119 and Figure 120 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 142.

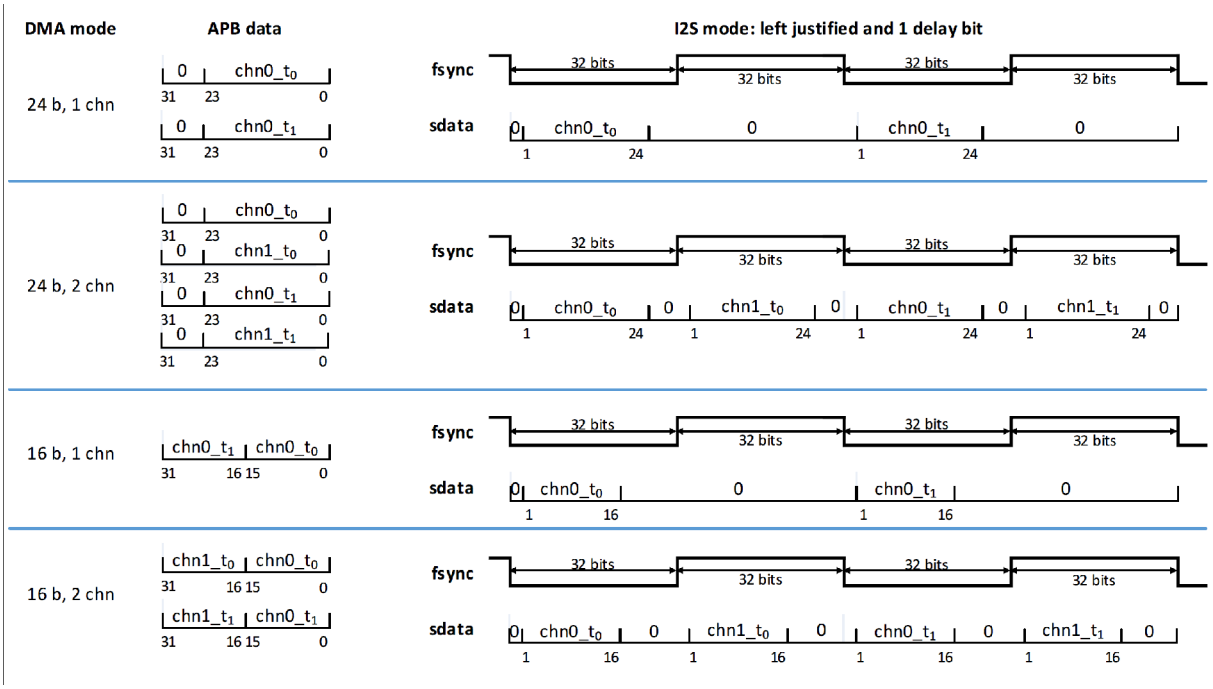


Figure 119. DMA Configurations for 16 Bits and 24 Bits

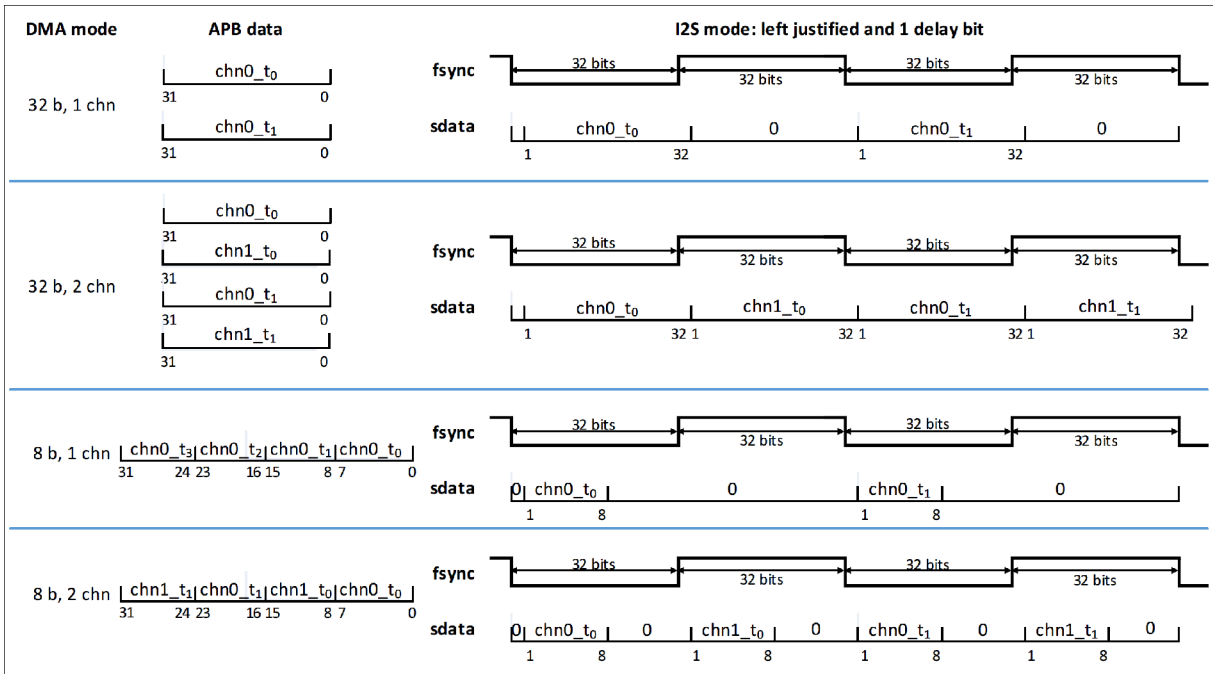


Figure 120. DMA Configurations for 8 Bits and 32 Bits

22.5 Configuration and Control

Each I^{2}S 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: master/slave configurations, I^{2}S initialization, and data flow control using receive and transmit FIFOs.

22.5.1 Master/Slave Configurations

Each I^{2}S instance can be configured to operate in slave or master mode. At power-up, they are configured to act as slaves to avoid signal collisions with other modules on the chip. The operation mode can be changed by writing the appropriate value to the MSL (Master Slave Configuration) bit in the I2S IO Configuration (I2SIOCFG) register.

22.5.2 Module Initialization

The following steps outline the procedure for initializing the I^{2}S interface. Unless otherwise noted, referenced registers are in the I^{2}S 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 I^{2}S IO signals by writing the desired configuration to the I^{2}S IO Configuration (I2SIOCFG) register.
3. Specify the serial data format in the I^{2}S 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 I^{2}S Control register (I2SCTL). Note that for full-duplex operation, the TXEN and RXEN bits must be asserted in a single register access.

22.5.3 Data Flow Control

22.5.3.1 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 I^2S 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.

22.5.3.2 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 I^2S 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.

22.6 Serial Audio Interface

22.6.1 TDM Serial Audio Format

Figure 121 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 Apollo4 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 slave, these registers are not used.

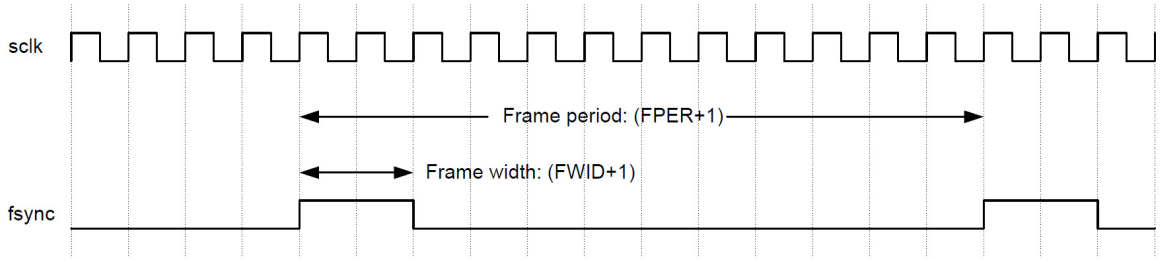


Figure 121. Programmable Frame Period and Width

A timing diagram of a single channel 8-bit sample audio serial interface signal can be observed in Figure 122. 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.

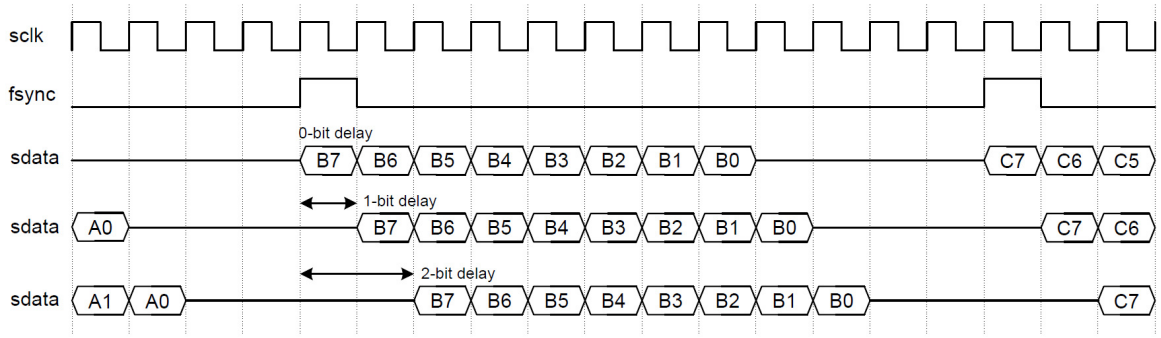


Figure 122. Data Delay

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
- FRLEN1 = 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.

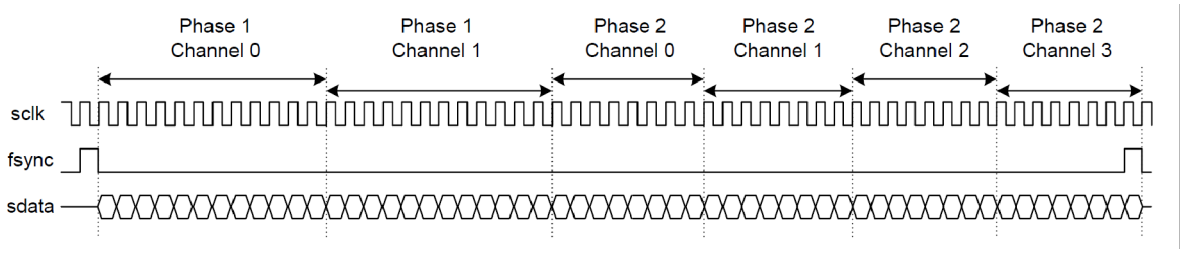


Figure 123. Dual-phase Frame Example

Figure 123 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
- FRLEN1 = 1, specifying two channels in phase 1
- FRLEN2 = 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

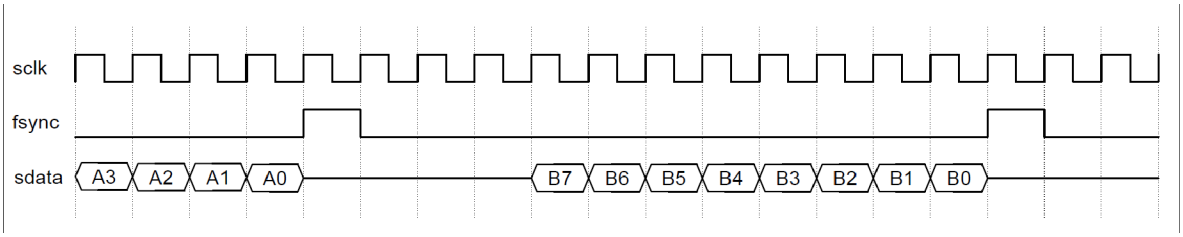


Figure 124. Right Justification

Figure 124 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
- FRLEN1 = 0, specifying one channel per frame
- WDLEN1 = 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

22.6.2 I²S Serial Audio Format

Figure 125 shows an I²S formatted audio frame.

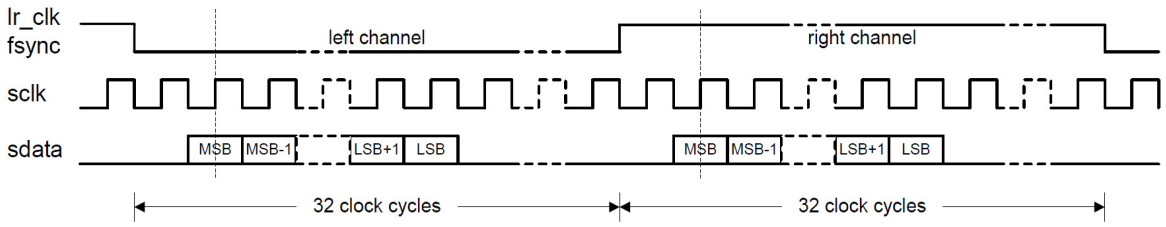


Figure 125. I²S Formatted Audio Frame

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
- FRLEN1 = 1, specifying two channels per frame
- WDLEN1 = 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

22.6.3 Left-justified Serial Audio Format

Figure 126 shows a left-justified formatted audio frame.

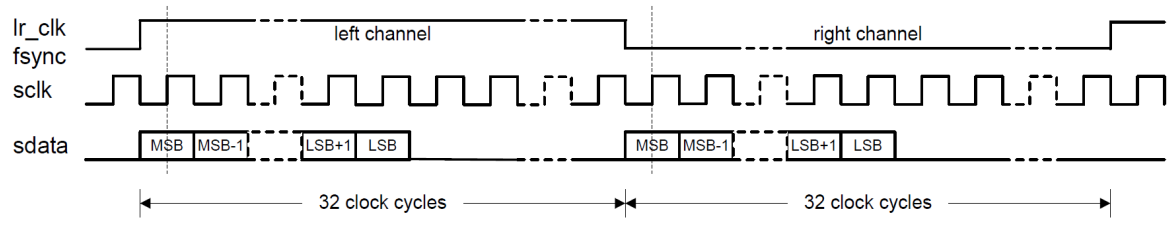


Figure 126. Left-justified Audio Frame

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
- FRLEN1 = 1, specifying two channels per frame
- WDLEN1 = 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

22.6.4 Right-justified Serial Audio Format

Figure 127 shows a right-justified formatted audio frame.

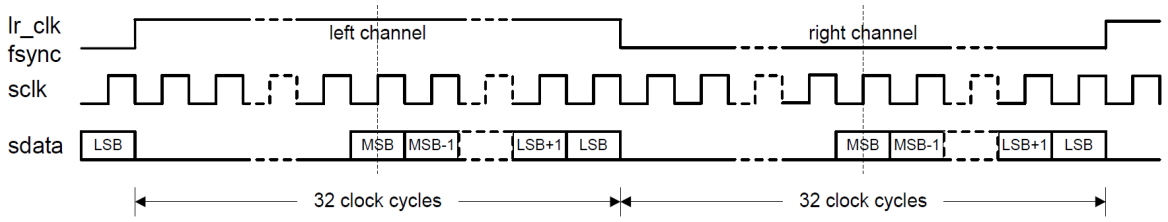


Figure 127. Right-justified Audio Frame

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
- FRLEN1 = 1, specifying two channels per frame
- WDLEN1 = 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

23. Ordering Information

Table 143: Apollo4 SoC Ordering Information

Device Name	Orderable Part Number	MRAM	RAM	Package	Packing	Temperature Range
Apollo4 SoC	AMAP42KK-KBR-B2	2 MB	1.8 MB	5.0 x 5.0 146-pin BGA	Tape and Reel	-20 to 60°C
Apollo4 Blue SoC	AMA4B2KK-KBR-B2	2 MB	1.8 MB	4.7 x 4.7 131-pin SiP BGA	Tape and Reel	-20 to 60°C

Table 144: Apollo4 Plus SoC Ordering Information

Device Name	Orderable Part Number	MRAM	RAM	Package (mm)	Packing	Temperature Range
Apollo4 Plus SoC	AMAP42KP-KBR	2 MB	2.75 MB	5.0 x 5.0 146-pin BGA	Tape and Reel	-20 to 60°C
Apollo4 Blue Plus SoC	AMA4B2KP-KBR	2 MB	2.75 MB	4.7 x 4.7 131-pin SiP BGA	Tape and Reel	–20 to 60°C
Apollo4 Blue Plus SoC	AMA4B2KP-KXR	2 MB	2.75 MB	4.7 x 4.7 131-pin SiP BGA	Tape and Reel	–20 to 60°C

Ambiq Micro, Inc.
6500 River Place Boulevard, Building 7,
Suite 200, Austin, TX 78730-1156
www.ambiq.com/
sales@ambiqmicro.com
https://support.ambiqmicro.com
+1 (512) 879-2850

PG-A4-7p0
Version 7.0
October 2022