i.MX 6 Series Firmware Guide

Document Number: IMX6DQFG
Rev. B, 12/2011

freescale™

Contents

Section Number **Title** **Page**

Section Number	Title	Page

Chapter 1
About This Guide

1.1 About this content

This document's purpose is to help software engineers quickly test and bring up their own custom boards for the i.MX6 series of applications processors. It provides example driver code to demonstrate the proper usage of the peripherals. Engineers are expected to have a working understanding of the ARM processor programming model, the C programming language, tools such as compilers and assemblers, and program build tools such as MAKE. Familiarity with the use of commonly available hardware test and debug tools such as oscilloscopes and logic analyzers is assumed. An understanding of the architecture of the i.MX6 series of application processors is also assumed. This document is intended as a companion to the i.MX6 series chip reference manuals and data sheets.

1.2 Essential reference

You should have access to an electronic copy of the latest version of the *i.MX 6Dual/ 6Quad Multimedia Applications Processor Reference Manual* (IMX6DQRM). Contact your local FAE for assistance.

1.3 Suggested reading

This section lists additional reading that provides background for the information in this manual as well as general information about the architecture.

1.3.1 General information

The following documentation provides useful information about the ARM processor architecture and computer architecture in general:

- For information about the ARM Cortex-A9 processor see http://www.arm.com/products/processors/cortex-a/cortex-a9.php
- Computer Architecture: A Quantitative Approach, Fourth Edition, by John L. Hennessy and David A. Patterson
- *Computer Organization and Design: The Hardware/Software Interface*, Second Edition, by David A. Patterson and John L. Hennessy

1.3.2 Related documentation

Freescale documentation is available from the sources listed on the back page of this guide.

Additional literature is published as new Freescale products become available. For a current list of documentation, refer to www.freescale.com.

1.4 Notational conventions

This table shows notational conventions used in this content.

Table 1-1. Notational conventions

Convention	Definition
General	
Cleared	When a bit takes the value zero, it is said to be cleared.
Set	When a bit takes the value one, it is said to be set.
mnemonics	Instruction mnemonics are shown in lowercase bold.
italics	Italics can indicate the following: • Variable command parameters, for example, **bcctr***x* • Titles of publications • Internal signals, for example, $\overline{\textit{core int}}$
h	Suffix to denote hexadecimal number. Note that numbers in code may use the alternate 0x prefix convention to denote hexadecimal instead.
b	Suffix to denote binary number. Note that numbers in code may use the alternate 0b prefix convention to denote binary instead.
rA, **r**B	Instruction syntax used to identify a source GPR
rD	Instruction syntax used to identify a destination GPR
REGISTER[FIELD]	Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.
x	In some contexts, such as signal encodings, an unitalicized x indicates a don't care.
x	An italicized *x* indicates an alphanumeric variable

Table continues on the next page...

Table 1-1. Notational conventions (continued)

Convention	Definition
n	An italicized *n* indicates either: • An integer variable • A general-purpose bitfield unknown
Â¬	NOT logical operator
&	AND logical operator
|	OR logical operator
||	Concatenation, for example, TCR[WPEXT] || TCR[WP]
Signals	
$\overline{\text{OVERBAR}}$	An overbar indicates that a signal is active-low.
lowercase_italics	Lowercase italics is used to indicate internal signals
lowercase_plaintex t	Lowercase plain text is used to indicate signals that are used for configuration.
Register access	
Reserved	Ignored for the purposes of determining access type
R/W	Indicates that all non-reserved fields in a register are read/write
R	Indicates that all non-reserved fields in a register are read only
W	Indicates that all non-reserved fields in a register are write only
w1c	Indicates that all non-reserved fields in a register are cleared by writing ones to them

1.4.1 Signal conventions

$\overline{\text{PWR\_ON\_RESET}}$ An overbar indicates that a signal is active when low

_b, _B Alternate notation indicating an active-low signal

signal_name Lowercase italics is used to indicate internal signals

1.5 Acronyms and abbreviations

The following table defines the acronyms and abbreviations used in this document.

Definitions and acronyms

Term	Definition
Address Translation	Address conversion from virtual domain to physical domain
API	Application Programming Interface

Table continues on the next page...

Term	Definition
ARMÂ®	Advanced RISC Machines processor architecture
AUDMUX	Digital audio multiplexer-provides a programmable interconnection for voice, audio, and synchronous data routing between host serial interfaces and peripheral serial interfaces.
BCD	Binary Coded Decimal
Bus	A path between several devices through data lines.
Bus load	The percentage of time a bus is busy.
CODEC	Coder/decoder or compression/decompression algorithm-Used to encode and decode (or compress and decompress) various types of data.
CPU	Central Processing Unit-generic term used to describe a processing core.
CRC	Cyclic Redundancy Check-Bit error protection method for data communication.
CSI	Camera Sensor Interface
DMA	Direct Memory Access-an independent block that can initiate memory-to-memory data transfers.
DRAM	Dynamic Random Access Memory
EMI	External Memory Interface-controls all IC external memory accesses (read/write/erase/program) from all the masters in the system.
Endian	Refers to byte ordering of data in memory. Little Endian means that the least significant byte of the data is stored in a lower address than the most significant byte. In Big Endian, the order of the bytes is reversed.
EPD	Electronic Paper Display
EPIT	Enhanced Periodic Interrupt Timer-a 32-bit set and forget timer capable of providing precise interrupts at regular intervals with minimal processor intervention.
ePXP	Enhanced Pixel Pipeline
FCS	Frame Checker Sequence
FIFO	First In First Out
FIPS	Federal Information Processing Standards-United States Government technical standards published by the National Institute of Standards and Technology (NIST). NIST develops FIPS when there are compelling Federal government requirements such as for security and interoperability but no acceptable industry standards or solutions.
FIPS-140	Security requirements for cryptographic modules-Federal Information Processing Standard 140-2(FIPS 140-2) is a standard that describes US Federal government requirements that IT products should meet for Sensitive, But Unclassified (SBU) use.
Flash	A non-volatile storage device similar to EEPROM, but where erasing can only be done in blocks of the entire chip.
Flash path	Path within ROM bootstrap pointing to an executable Flash application.
Flush	A procedure to reach cache coherency. Refers to removing a data line from cache. This process includes cleaning the line, invalidating its VBR and resetting the tag valid indicator. The flush is triggered by a software command.
GPIO	General Purpose Input/Output
Hash	Hash values are produced to access secure data. A hash value (or simply hash), also called a message digest, is a number generated from a string of text. The hash is substantially smaller than the text itself, and is generated by a formula in such a way that it is extremely unlikely that some other text will produce the same hash value.

Table continues on the next page...

Term	Definition
I/O	Input/Output
ICE	In-Circuit Emulation
IP	Intellectual Property.
IrDA	Infrared Data Association-a nonprofit organization whose goal is to develop globally adopted specifications for infrared wireless communication.
ISR	Interrupt Service Routine.
JTAG	JTAG (IEEE Standard 1149.1) A standard specifying how to control and monitor the pins of compliant devices on a printed circuit board.
Kill	Abort a memory access.
KPP	KeyPad Port-a 16-bit peripheral that can be used as a keypad matrix interface or as general purpose input/output (I/O).
line	Refers to a unit of information in the cache that is associated with a tag.
LRU	Least Recently Used-a policy for line replacement in the cache.
MMU	Memory Management Unit-a component responsible for memory protection and address translation.
MPEG	Moving Picture Experts Group-an ISO committee that generates standards for digital video compression and audio. It is also the name of the algorithms used to compress moving pictures and video.
MPEG standards	There are several standards of compression for moving pictures and video. MPEG-1 is optimized for CD-ROM and is the basis for MP3. MPEG-2 is defined for broadcast quality video in applications such as digital television set-top boxes and DVD. MPEG-3 was merged into MPEG-2. MPEG-4 is a standard for low-bandwidth video telephony and multimedia on the World-Wide Web.
MQSPI	Multiple Queue Serial Peripheral Interface-used to perform serial programming operations necessary to configure radio subsystems and selected peripherals.
MSHC	Memory Stick Host Controller
NAND Flash	Flash ROM technology-NAND Flash architecture is one of two flash technologies (the other being NOR) used in memory cards such as the Compact Flash cards. NAND is best suited to flash devices requiring high capacity data storage. NAND flash devices offer storage space up to 512-Mbyte and offer faster erase, write, and read capabilities over NOR architecture.
NOR Flash	See NAND Flash.
PCMCIA	Personal Computer Memory Card International Association-a multi-company organization that has developed a standard for small, credit card-sized devices, called PC Cards. There are three types of PCMCIA cards that have the same rectangular size (85.6 by 54 millimeters), but different widths.
Physical address	The address by which the memory in the system is physically accessed.
PLL	Phase Locked Loop-an electronic circuit controlling an oscillator so that it maintains a constant phase angle (a lock) on the frequency of an input, or reference, signal.
RAM	Random Access Memory
RAM path	Path within ROM bootstrap leading to the downloading and the execution of a RAM application
RGB	The RGB color model is based on the additive model in which Red, Green, and Blue light are combined in various ways to create other colors. The abbreviation RGB come from the three primary colors in additive light models.

Table continues on the next page...

Term	Definition
RGBA	RGBA color space stands for Red Green Blue Alpha. The alpha channel is the transparency channel, and is unique to this color space. RGBA, like RGB, is an additive color space, so the more of a color you place, the lighter the picture gets. PNG is the best known image format that uses the RGBA color space.
RNGA	Random Number Generator Accelerator-a security hardware module that produces 32-bit pseudo random numbers as part of the security module.
ROM	Read Only Memory
ROM bootstrap	Internal boot code encompassing the main boot flow as well as exception vectors.
RTIC	Real-time integrity checker-a security hardware module
SCC	SeCurity Controller-a security hardware module
SDMA	Smart Direct Memory Access
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on a Chip
SPBA	Shared Peripheral Bus Arbiter-a three-to-one IP-Bus arbiter, with a resource-locking mechanism.
SPI	Serial Peripheral Interface-a full-duplex synchronous serial interface for connecting low-/medium-bandwidth external devices using four wires. SPI devices communicate using a master/slave relationship over two data lines and two control lines: Also see SS, SCLK, MISO, and MOSI.
SRAM	Static Random Access Memory
SSI	Synchronous-Serial Interface-standardized interface for serial data transfer
TBD	To Be Determined
UART	Universal Asynchronous Receiver/Transmitter-this module provides asynchronous serial communication to external devices.
UID	Unique ID-a field in the processor and CSF identifying a device or group of devices
USB	Universal Serial Bus-an external bus standard that supports high speed data transfers. The USB 1.1 specification supports data transfer rates of up to 12Mb/s and USB 2.0 has a maximum transfer rate of 480 Mbps. A single USB port can be used to connect up to 127 peripheral devices, such as mice, modems, and keyboards. USB also supports Plug-and-Play installation and hot plugging.
USBOTG	USB On The Go-an extension of the USB 2.0 specification for connecting peripheral devices to each other. USBOTG devices, also known as dual-role peripherals, can act as limited hosts or peripherals themselves depending on how the cables are connected to the devices, and they also can connect to a host PC.
Word	A group of bits comprising 32 bits

Chapter 2
Multicore Startup

2.1 Overview

This chip can include up to four Cortex-A9 cores. Regardless of how many cores are available on the part, only core0 will be automatically released from reset upon initial power-up. All other available secondary cores will remain in a low-power reset state. The firmware must initialize all secondary cores. This chapter explains how to enable the available secondary cores.

2.2 Boot ROM process

Once it has been released from reset, each Cortex-A9 core attempts to execute at the ARM reset exception vector upon initial power-up. This vector in the i.MX 6Dual/6Quad memory map (at 0000 0000h) is part of the on-chip boot ROM. The boot ROM code uses the state of the eFuses and/or boot GPIO settings to determine the boot behavior of the device using core0 (where core0- core3's availability on the chip depends on whether i.MX 6Quad or i.MX 6Dual is being used).

To distinguish which core is currently booting up, the boot ROM checks the CPU ID stored in the CortexA9 Multiprocessor Affinity register. See the CortexA9 technical reference manual for further details.

If core0 is booting up, the boot ROM enters the boot process and determines where to boot the image from. It loads and executes the image after completing all HAB checks. See the "System Boot" chapter of the i.MX6DQ reference manual for further details.

If the core booting is not core0, the boot ROM checks the persistent bits to determine whether the core has a valid pointer that the boot ROM can execute from. The persistent bits are a collection of general-purpose registers in the System Reset Controller (SRC).

The boot ROM code expects to find valid pointers to executable regions and functions for each core stored in these registers. These registers are used because they retain their values even after a warm reset. See table below for the list of registers.

For full details on boot process and the persistent bits, refer to the i.MX6DQ reference manual.

Table 2-1. Function pointers used in boot ROM

Register	Description
SRC_GPR3	Entry function pointer for CPU1
SRC_GPR4	Argument for entry function for CPU1
SRC_GPR5	Entry function pointer for CPU2 (i.MX 6Quad only)
SRC_GPR6	Argument for entry function for CPU2
SRC_GPR7	Entry function pointer for CPU3 (i.MX 6Quad only)
SRC_GPR8	Argument for entry function for CPU3

In addition to the entry function pointer for each core, the presistent bit registers also provide an argument register that is passed into the entry function as an argument pre-loaded to the Cortex-A9 register 0 (r0) for that core. The SDK uses the entry function to point to the startup routine that initializes the core, cache, and stacks. Then it uses the argument value in r0 as a pointer to a function that the core will jump to once general initialization is complete.

2.3 Activating the secondary cores

Although multiple cores are available on this processor, only core0 automatically activates during the initial boot process. The System Reset Controller (SRC) block handles the reset signal for each core and by default, the SRC keeps the secondary cores in a reset state after boot. Therefore, the application needs to enable the other available cores.

To enable the other available cores:

1. Initialize persistent bits for the secondary core being activated.
2. Set the core_enable signal for each of the cores in the SRC Control Register (bits 22:24, for core1, core2, and core3 respectively).
3. Once the enable bits are set, the corresponding core is released from its reset state, and it executes the boot ROM (at 0000 0000h).
4. The boot room determines if it is a secondary core and uses the presistent bit registers to determine what to execute next.

This boot process is described in detail in the "System Boot" chapter of the reference manual.

2.4 Multicore hello world example

Here is an example startup routine written in ARM assembly that shows how the argument registers can be used.

```
ResetHandler
   mov     r4, r0       ; save r0 for cores 1-3, r0 arg field passed by ROM
                        ; r0 is a function pointer for secondary cpus
   ldr                r0, =STACK_BASE
   ldr                r1, =STACK_SIZE
   ; get cpu id, and subtract the offset from the stacks base address
   mrc     p15,0,r2,c0,c0,5  ; read multiprocessor affinity register
   and     r2, r2, #3        ; mask off, leaving CPU ID field
   mov     r5, r2       ; save cpu id for later
   mul     r3, r2, r1
   sub     r0, r0, r3

   mov     r1, r1, lsl #2

   ; set stack for SVC mode
   mov     sp, r0
   ; set stacks for all other modes
   msr     CPSR_c, #Mode_FIQ :OR: I_Bit :OR: F_Bit
   sub     r0, r0, r1
   mov     sp, r0

   msr     CPSR_c, #Mode_IRQ :OR: I_Bit :OR: F_Bit
   sub     r0, r0, r1
   mov     sp, r0

   msr     CPSR_c, #Mode_ABT :OR: I_Bit :OR: F_Bit
   sub     r0, r0, r1
   mov     sp, r0

   msr     CPSR_c, #Mode_UND :OR: I_Bit :OR: F_Bit
   sub     r0, r0, r1
   mov     sp, r0

   msr     CPSR_c, #Mode_SYS :OR: I_Bit :OR: F_Bit
   sub     r0, r0, r1
   mov     sp, r0

   ; go back to SVC mode and enable interrupts
   msr     CPSR_c, #Mode_SVC

   bl         freq_populate
   ; Disable caches
   bl      disable_caches
   ; Invalidate caches
   bl      invalidate_caches
   ; Invalidate Unified TLB
   bl      invalidate_unified_tlb
   ; Enable MMU
   bl      invalidate_unified_tlb
   ; check cpu id - for cores 1-3 jump to user code, continue otherwise
   cmp r5, #0
   bleq primary_cpu_init
             blne secondary_cpus_init
          primary_cpu_init
   bl enable_scu
   bl enable_GIC
   bl enable_gic_processor_interface
   bl hello_mpcore     ; jump to main application
secondary_cpus_init
   bl enable_gic_processor_interface
   bx r4                     ; jump to argument function pointer passed in by ROM
  END
```

This ResetHandler routine is written with the expectation that the secondary cores will be enabled by setting the Entry Function Pointer persistent bits for all the cores to point to the address of the ResetHandler routine. Initially, all of the stack pointers for all of the cores need to be initialized; therefore, they can all execute a common startup routine.

2.4.1 System Reset Controller enable CPU function

This example uses a start secondary core function that writes the boot function pointers to the ROM persistent bits and releases the secondary core from reset.

```
void start_secondary_cpu(int cpu_num, unsigned int *ptr){
          /* prepare pointers for ROM code */
          writel((u32)&ResetHandler, SRC_BASE_ADDR + (SRC_GPR1_OFFSET + cpu_num*8));
          writel((u32)ptr, SRC_BASE_ADDR + (SRC_GPR2_OFFSET + cpu_num*8));
          /* start core */
          if (cpu_num > 0){
                    writel( (readl(SRC_BASE_ADDR + SRC_SCR_OFFSET) | (1 << (21 + cpu_num))),
(SRC_BASE_ADDR + SRC_SCR_OFFSET));
          }
}
```

In this function, the main Entry Function (in the persistent bits) is always be set to the ResetHandler startup routine for all cores. The second argument is a pointer (*ptr) to the function that will execute after the Entry Function. This is the argument that will be passed in at r0 to the ResetHandler startup function. After the startup routine finishes initializing the CortexA9, it jumps to this pointer.

2.4.2 Hello multicore world

The example hello world routine is shown below.

```
void hello_mpcore(){
          int cpu_id, i;
          cpu_id = getCPUnum();
          if (cpu_id == 0){
                    debug_uart_iomux();
                    debug_uart->freq = 80000000;
                    init_debug_uart(debug_uart, 115200);
                    printf("####################################################\n");
          }
          printf("Hello from CPU %d\n", cpu_id);
          if (cpu_id < (TOTAL_CORES-1)){  //start the next core
                    start_secondary_cpu(cpu_id+1, (unsigned int *)&hello_mpcore);
          }
          while(1);
}
```

This function first determines which core is executing the routine. If it is core0, the function initializes the UART port so that the printf statements are sent out to the debug UART port on the i.MX 6Dual/6Quad hardware. If it is core1-3, the function prints the statement "Hello from CPU *n*."

After the function communicates its hello statement, it enables the next available core using the start_secondary_cpu routine. Here the persistent bits argument function pointer is set to call the hello_mpcore routine so that each core prints out its equivalent "hello world" statement.

When executing the example using the i.MX 6Quad chip, these are the expected results:

```
#####################################################
Hello from CPU 0
Hello from CPU 1
Hello from CPU 2
Hello from CPU 3
```

Chapter 3
Configuring the GIC Driver

3.1 Overview

This chapter explains how to enable interrupts in the processor. The general interrupt controller (GIC) supports up to 128 shared peripheral interrupt (SPI) sources and 16 software generated interrupt (SGI) sources. The GIC is split into a main interrupt distributor block and individual core interface blocks, one for each Cortex-A9 core present in the system. Refer to the *GIC Architecture Specification* from ARM for a complete description of the GIC.

The block diagram for GIC is as follows:

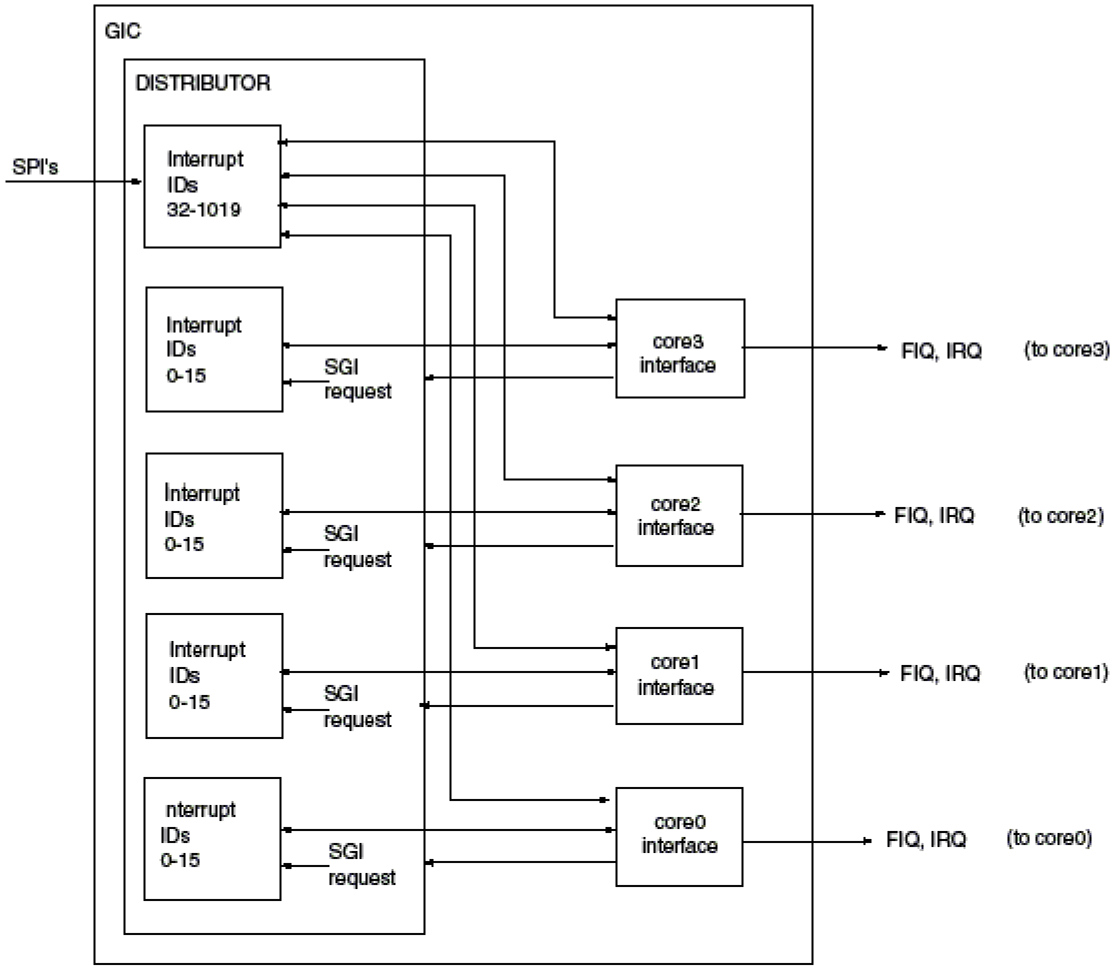


Figure 3-1. GIC simplified block diagram

The interrupt controller is memory mapped. Each core can access these global control registers by using a private interface through the snoop control unit (SCU). The base address can be determined by reading the Cortex-A9 CP15 configuration base address, which stores the value of this location. On this processor, the SCU base address (aka PERIPHBASE address) starts at 00A0 0000h. The following table shows the general high-level private memory map.

Table 3-1. Cortex A-9 general MPCore memory map

Offset from (00A0 0000h)	Cortex-A9 MPCore Module
0000h-00FCh	SCU registers
0100h-01FFh	GIC core interrupt interfaces
0200h-02FFh	Global timer
0600h-06FFh	Private timers and watchdogs
1000h-1FFFh	GIC interrupt distributor

3.2 Feature summary

The GIC's main functions are to:

- Globally enable the GIC distributor
- Enable individual interrupt sources (IDs)
- Set individual interrupt ID priority levels
- Set the interrupt source targeted core
- Send software generated interrupts between the cores

3.3 ARM interrupts and exceptions

All of the i.MX applications processors use the standard ARM exception vectors, where by default the exception vector table resides at 0000 0000h. The standard exception vector table is shown in the following table.

Table 3-2. Standard exception vectors

Address	ARM Mode	Exception Description
0000 0000h	SVC	Reset
0000 0004h	UND	Undefined Instruction
0000 0008h	SVC	Software Interrupt
0000 000Ch	ABT	Prefetch Abort
0000 0010h	ABT	Data Abort
0000 0014h	-	Not assigned
0000 0018h	IRQ	IRQ
0000 001Ch	FIQ	FIQ

When there is an exception, the ARM core jump to the associated exception vector and switches to the corresponding mode. Therefore, the stack pointers for all ARM modes should be initialized to a valid address during the start-up routine.

NOTE

If the stacks are not initialized when servicing an interrupt, pushing registers to the stack causes a memory access violation, triggers a Data Abort exception, and ultimately crashes the system.

On i.MX processors, the default ARM exception table region is allocated to the boot ROM region and cannot be overwritten. Upon power up, the boot ROM sets up this table to jump to locations in the internal RAM space (iRAM).

To modify the table to execute custom interrupt/exception handlers, update the locations that the iRAM vectors point to. On the i.MX 6Dual/6Quad processor, the boot ROM locates the IRQ jump pointer address at 0093 FFF4h and the FIQ pointer at 0093 FFF8h. To have an interrupt exception execute a custom handler, update the pointer to the IRQ and FIQ to point to the custom function.

3.3.1 GIC interrupt distributor

The distributor block performs interrupt prioritization and distribution of interrupts to the core interface blocks. Any interrupt (IRQ, FIQ) that is triggered from any peripheral must follow this sequence:

1. The GIC distributor determines the priority of each interrupt.
2. It forwards the highest priority interrupt to the available core interface blocks. Each interrupt source can be targeted to a single or multiple cores through GIC distributor CPU target registers.
3. Hardware ensures that if an interrupt is targeted to several of the available cores, only one of the cores handles it.

The GIC distributor registers used in the i,MX 6Dual/6Quad interrupt example are shown in the following table. For a full list of available registers/features, refer to the ARM GIC architecture specification document.

Table 3-3. GIC distributor registers

Offset from GIC interrupt distributor base (00A0 1000h)	Register name	Description
000h	ICDDCR	Distributor Control Register
080h-0FCh	ICDISR	Interrupt Security Registers (n-interrupts/32, registers)
100h-17Ch	ICDISER	Interrupt Set Enable Registers (n-interrupts/32, registers)
180h-1FCh	ICDICER	Interrupt Clear Enable Registers (n-interrupts/32, registers)
400h-7F8h	ICDIPR	Interrupt Priority Registers (n-interrupts/4, registers)
800h-BF8h	ICDITR	Interrupt CPU Target Register (n-interrupts/4, registers)

3.3.2 GIC core interfaces

The core interface blocks provide a separate interface between each available core and the GIC distributor. Their main functions are to:

- Enable the GIC CPU interface (to allow it to send interrupts to the CPU it is connected to) and set the CPU priority level
- Read the acknowledge register to send an ack signal to the GIC Distributor
- Write to the end of interrupt register

When enabled, the interface takes the highest priority pending interrupt available from the distributor and determines whether the interrupt source has sufficient priority to interrupt the core to which it is connected. The core interface needs to be enabled to send interrupt requests to the core to which it is connected.

In the main interrupt service routine, if the interrupt source is sent to a core, that core must use its GIC core interface to send the GIC distributor an acknowledge signal. Similarly, when the interrupt finishes being serviced, the core interface must be used to send an end of interrupt signal to the distributor block.

The registers used in this example are shown in the following table. For a full list of available registers and features, refer to the *ARM GIC Architecture Specification*.

Table 3-4. GIC CPU interface registers

Offset from GIC CPU Interface Base (00A0 0100h)	Register Name	Description
00h	ICCICR	CPU Interface Control Register
04h	ICCPMR	CUP Interrupt Priority Mask Register
0Ch	ICCIAR	Interrupt acknowledge Register
10h	ICCEOIR	End of Interrupt Register

Note that each core interface is memory mapped to the same address space, but is unique for each available core interface. For example, in the i.MX6Quad, each core must perform writes to these registers to configure the associated GIC core interface.

3.4 Sample code

3.4.1 Handling interrupts using C

This example shows how to implement the interrupt support by creating an interrupt vector array. Using C, you can create an array of function pointers (pointers to the individual ISR routines) where the array index corresponds to the interrupt source ID.

```
typedef void (*funct_t)(void);       // define a pointer to a function
funct_t vect_IRQ[160];
```

The i.MX6Quad/Dual processor implements the interrupt sources as follows:

- interrupt IDs[0:15] are used for the 16 software generated interrupt sources
- interrupt IDs [15:31] are unused and left as reserved
- interrupt IDs [32:160] are used for the 128 shared peripheral interrupt sources

Therefore, there are 160 corresponding entries in the interrupt vector array. A register IRQ function can be used to set the corresponding device driver interrupt service routine to the position in the array that corresponds to the device interrupt source ID.

```
// set funcISR as the ISR function for the source ID #
void registerIRQ(int ID, funct_t funcISR){
          vect_IRQ[ID] = funcISR;
}
```

In the actual interrupt service routine that is executed when the ARM jumps to an IRQ exception, determine the interrupt source ID and use that as the index to the interrupt vector array.

To follow GIC guidelines for handling interrupts:

- Read the IAR register to send the ack signal
- After the actual targeted interrupt service routine is finished, the interrupt handler must send the end of IRQ to the distributor.

```
// IRQ_Handler, this functions handles IRQ exceptions
void __irq C_IRQ_Handler(void){
          unsigned int vectNum;
          vectNum = read_irq_ack(); // send ack, get ID source #
          // vectNum now contains source ID in bits [9:0]
// Check if ID is 1023 or 1022 (spurious interrupt)
if (vectNum & 0x0200){
                    write_end_of_irq(vectNum); // if spurious, send end of irq
          }
          else{
                    vect_IRQ[vectNum & 0x1FF](); // jump to ISR in the look up table
                    write_end_of_irq(vectNum); // send end of irq
      }
}
```

3.4.2 Enabling the GIC distributor

To enable the distributor, set bit 0 of the GIC distributor control register (ICDDCR). This enables the distributor to forward pending interrupts to the enabled GIC core interface blocks.

```
; void enable_GIC(void);
enable_GIC PROC
  MRC     p15, 4, r0, c15, c0, 0  ; Read periph base address
  ADD     r0, r0, #0x1000         ; Add the GIC Distributor offset
  LDR     r1, [r0]                ; Read the GIC's Enable Register  (ICDDCR)
  ORR     r1, r1, #0x03           ; the enable bits
  STR     r1, [r0]                ; Write the GIC's Enable Register  (ICDDCR)
  BX      lr
  ENDP
```

3.4.3 Enabling interrupt sources

To enable an interrupt source, the GIC distributor provides Interrupt Set Enable Registers (ICDISER) for the interrupt sources. Each bit in the registers corresponds to an available interrupt source. The number of ICDISER registers are implementation dependent and vary depending on how many interrupts the system supports.

To enable the GIC distributor so that it can forward the corresponding interrupt to the GIC CPU interfaces, set the corresponding interrupt source bit.

```
; void enable_irq_id(unsigned int ID);
enable_irq_id PROC
  MOV     r1, r0                  ; Back up passed in ID value
  MRC     p15, 4, r0, c15, c0, 0  ; Read periph base address
  ; First, we need to identify which 32 bit block the interrupt lives in
  MOV     r2, r1              ; Make working copy of ID in r2
  MOV     r2, r2, LSR #5      ; LSR by 5 places, affective divide by 32
                              ; r2 now contains the 32 bit block for the ID
  MOV     r2, r2, LSL #2      ; mult by 4, to convert offset into an address offset (four 
bytes
per reg)
  ; Now work out which bit within the 32 bit block the ID is
  AND     r1, r1, #0x1F      ; Mask off to give offset within 32bit block
  MOV     r3, #1             ; Move enable value into r3
  MOV     r3, r3, LSL r1     ; Shift it left to position of ID
  ADD     r2, r2, #0x1100    ; Add r2 offset, to get (ICDISER) register
  STR     r3, [r0, r2]       ; Store r3 to (ICDISER)
  BX      lr
  ENDP
```

3.4.4 Configuring interrupt priority

The GIC distributor provides a set of Interrupt Priority Registers (ICDIPR) for the interrupt sources. Each byte in the registers corresponds to the priority level of an interrupt source. Therefore, there are four priority bit fields per ICDIPR register. Each 8-bit priority field within the priority registers can have possible values of 00h-FFh, where 00h is the highest possible priority and FFh is the lowest. The individual interrupt priority level must be set to a higher priority level than the core priority level to be able to interrupt the ARM core.

```
; void set_irq_priority(unsigned int ID, unsigned int priority);
; r0 = ID, r1 = priority
set_irq_priority PROC
  ; Get base address of private peripheral space
  MOV     r2, r0                  ; Back up passed in ID value
  MRC     p15, 4, r0, c15, c0, 0  ; Read periph base address
; Make sure that priority value is only 8 bits
  AND     r1, r1, #0xFF
  ; Find which priority register this ID lives in
  BIC     r3, r2, #0x03   ; copy ID to r3 clearing off the bottom two bits
                                  ; There are four IDs per reg, by clearing the bottom two
bits
we get an address offset
  ADD     r3, r3, #0x1400         ; Now add the offset of the Priority Level registers from
the
base of the private peripheral space
  ADD     r0, r0, r3              ; Now add in the base address of the private peripheral
space,
giving us the absolute address
  ; Now work out which ID in the register it is
  AND     r2, r2, #0x03           ; Clear all but the bottom four bits, leaves which ID in
the
reg it is (which byte)
  MOV     r2, r2, LSL #3          ; Multiply by 8, this gives a bit offset
  ; Read -> Modify -> Write
  MOV     r12, #0xFF              ; Mask (8 bits)
  MOV     r12, r12, LSL r2        ; Move mask into correct bit position
  MOV     r1, r1, LSL r2          ; Also, move passed in priority value into correct bit
position
  LDR     r3, [r0]                ; Read current value of the Priority Level register
(ICDIPR)
  BIC     r3, r3, r12             ; Clear appropriate field
  ORR     r3, r3, r1              ; Now OR in the priority value
  STR     r3, [r0]                ; And store it back again  (ICDIPR)
  BX      lr
  ENDP
```

3.4.5 Targeting interrupts to specific cores

The GIC distributor provides a set of Interrupt Target Registers (ICDITR) for the interrupt sources. Each byte in the registers corresponds to the core targets of an interrupt source. Therefore, there are four core target bit fields per ICDITR register.

For the i.MX6Quad/Dual processor, each 8-bit CPUT target field within the priority registers can have possible values of 00h-0Fh because there are only up to four available cores. Each bit corresponds to one core (where core-0 = bit0, core-1 = bit1, and so on…).

Using the core target registers, the GIC distributor can distribute the load between cores effectively. If a triggered interrupt is targeted to multiple cores, the GIC distributor has options for where it can send the interrupt. Thus, a delay is less likely because if one of the targeted cores is busy, the GIC distributor can send the interrupt to a free core instead.

```
; void enable_interrupt_target_cpu(unsigned int ID, unsigned int target_cpu);
enable_interrupt_target_cpu PROC
  MOV     r2, r0                  ; Back up passed in ID value
  MRC     p15, 4, r0, c15, c0, 0  ; Read periph base address

  ; Make sure that cpu value is only 2 bits max CPU value is 3 (0-3)
  AND     r1, r1, #0x3
  ; Find which cpu_target register this ID lives in
  BIC     r3, r2, #0x03  ; copy the ID, clearing off the bottom two bits
                                  ; There are four IDs per reg, by clearing the bottom two
bits
we get an address offset
  ADD     r3, r3, #0x1800   ; Now add the offset of the Target CPU registers from the base
of
the private peripheral space
  ADD     r0, r0, r3        ; Now add in the base address of the private peripheral space,
giving us the absolute address
```

```
  ; Now work out which ID in the register it is
  AND     r2, r2, #0x03           ; Clear all but the bottom four bits, leaves which ID in
the
reg it is (which byte)
  MOV     r2, r2, LSL #3          ; Multiply by 8, this gives a bit offset
  MOV     r4, #1                  ; Move enable value into r4
  MOV     r4, r4, LSL r1          ; Shift it left to position of CPU target
  MOV     r4, r4, LSL r2          ; move it to correct bit ID offset position

  LDR     r3, [r0]   ;read current value of the CPU Target register (ICDITR)
  ORR     r3, r3, r4              ; Now OR in the CPU Target value
  STR     r3, [r0]                ; And store it back again  (ICDITR)
  BX      lr
  ENDP
```

3.4.6 Using software generated interrupts (SGIs)

The GIC distributor also allows the use of software generated interrupts (SGIs) for interprocessor communication. SGIs allow a core to interrupt other cores directly.

This processor supports 16 SGI interrupt sources. To issue an SGI, write to the SGIR distributor register and set the SGI_ID and CPUTarget bit fields. As with normal interrupts, each SGI can target multiple cores.

```
; void send_sgi(unsigned int ID, unsigned int target_list, unsigned int filter_list)
send_sgi PROC
  AND     r3, r0, #0x0F        ; Mask off unused bits of ID, and move to r3
  AND     r1, r1, #0x0F        ; Mask off unused bits of target_filter
  AND     r2, r2, #0x0F        ; Mask off unused bits of filter_list
  ORR     r3, r3, r1, LSL #16     ; Combine ID and target_filter
  ORR     r3, r3, r2, LSL #24     ; and now the filter list
  ; Get the address of the GIC
  MRC     p15, 4, r0, c15, c0, 0  ; Read periph base address
  ADD     r0, r0, #0x1F00         ; Add offset of the sgi_trigger reg
  STR     r3, [r0]                ; Write to the SGI Register  (ICDSGIR)
  BX      lr
  ENDP
```

3.4.7 Enabling the GIC processor interface

To enable the GIC processor interface, write to the bottom two bits of the core Interface Control Register, where bit:0 enables secure interrupts, and bit:1 enables non-secure interrupts.

```
; void enable_gic_processor_interface(void);
enable_gic_processor_interface PROC
  MRC     p15, 4, r0, c15, c0, 0  ; Read periph base address
  LDR     r1, [r0, #0x100] ; Read CPU Interface Control reg (ICCICR/ICPICR)
  ORR     r1,  r1, #0x07   ; Bit 0:secure interrupts, bit 1: Non-Secure
  STR     r1, [r0, #0x100] ; Write CPU Interface Control reg (ICCICR/ICPICR)
  BX      lr
  ENDP
```

3.4.8 Setting the CPU priority level

Each core can have different priority levels set with the core Interface Priority Mask Register. After reset, the value of the priority mask registers for each core interface is set to mask all interrupts. Therefore, this register needs to be configured to allow the core to service interrupts. The associated core can only be interrupted by interrupt sources with higher priority levels than the core mask priority level.

```
; void set_cpu_priority_mask(unsigned int priority);
set_cpu_priority_mask PROC
  MRC     p15, 4, r1, c15, c0, 0  ; Read periph base address to r1
  STR     r0, [r1, #0x0104] ; Write the priority mask reg (ICCPMR/ICCIPMR)
  BX      lr
  ENDP
```

3.4.9 Reading the GIC IRQ Acknowledge

After an interrupt is sent to a core, the core must read the Interface IRQ Acknowledge Register (ICCIAR) to determine the interrupt source. This read effectively acts as an acknowledge for the interrupt to the GIC distributor. The ICCIAR register contains the interrupt ID for normal interrupts in the bottom 10 bits, and the core ID for any software generated interrupts in bits 13-10.

```
; unsigned int read_irq_ack(void);
read_irq_ack PROC
  MRC     p15, 4, r0, c15, c0, 0  ; Read periph base address
  LDR     r0, [r0, #0x010C]   ; Read the Interrupt Acknowledge reg (ICCIAR)
                                                     ; value gets returned in r0
  BX      lr
  ENDP
```

3.4.10 Writing the end of IRQ

After the core finishes servicing the interrupt, it must write the interrupt source ID to the CPU Interface End of Interrupt Register (ICCEOIR). For every read of a valid ID from the ICCIAR register, the core must perform a matching write to the ICCEOIR register. The value written to the EOIR must be the interrupt ID read from the IAR register.

```
; void write_end_of_irq(unsigned int ID)
write_end_of_irq PROC
  MRC     p15, 4, r1, c15, c0, 0  ; Read periph base address to r1
  STR     r0, [r1, #0x0110] ; Write ID(r0) to the End of Interrupt register
  BX      lr
  ENDP
```

3.4.11 GIC "hello world" example

This simple "hello world" example uses software generated interrupts. We arbitrarily chose SGI ID 3, which in the example is defined as SW_INTERRUPT_3.

1. In the main routine, all four cores are initialized with the system reset controller.
2. When core-3 completes the main routine, it triggers the SGI3 interrupt to core-0.
3. The SGI service routine initiates a loop that tells all cores to print hello to the terminal because of the SGI ISR routine is written such that after the SGI prints hello to the terminal, it triggers another SGI to the next core, as shown below:

```
void SGI3_ISR(void){
          int cpu_id;
          cpu_id = getCPUnum();
          printf("Hello from CPU %d\n", cpu_id);
          if(cpu_id < 4){
                    send_sgi(SW_INTERRUPT_3, ( 1 << (cpu_id+1) ), 0);
          }
}
```

When executing the example using the i.MX 6Quad chip, these are the expected results:

```
####################################################
Hello from CPU 0
Hello from CPU 1
Hello from CPU 2
Hello from CPU 3
```

3.4.12 GIC test code

```
#include "hardware.h"
//globals used for gic_test
unsigned int gicTestDone;
//unsigned int uartFREE;
extern void startup_imx6x(void);    // entry function, startup routine, defined in startup.s
extern uint32_t getCPUnum(void);
void SGI3_ISR(void)
{
    uint32_t cpu_id;
    cpu_id = getCPUnum();
    //while(1); // debug
    printf("Hello from CPU %d\n", cpu_id);
    if (cpu_id < 4) {
        send_sgi(SW_INTERRUPT_3, (1 << (cpu_id + 1)), 0);   // send to cpu_0 to start sgi
loop;
    }
    if (cpu_id == 3) {
        gicTestDone = 0;        // test complete
    }
}
void start_secondary_cpu(uint32_t cpu_num, void functPtr(void))
{
    //printf("start sedondary %d\n", cpu_num);
    //printf("ptr 0x%x\n",(uint32_t)functPtr);
    /* prepare pointers for ROM code */
    writel((uint32_t) & startup_imx6x, SRC_BASE_ADDR + (SRC_GPR1_OFFSET + cpu_num * 8));
    writel((uint32_t) functPtr, SRC_BASE_ADDR + (SRC_GPR2_OFFSET + cpu_num * 8));
    /* start core */
    if (cpu_num > 0) {
        writel((readl(SRC_BASE_ADDR + SRC_SCR_OFFSET) | (1 << (21 + cpu_num))),
               (SRC_BASE_ADDR + SRC_SCR_OFFSET));
    }
}
// only primary cpu will run gic_test
void gic_test(void)
{
    uint32_t cpu_id;
    cpu_id = getCPUnum();
    if (cpu_id == 0) {
        gicTestDone = 1;
        //uartFREE = 1;
        register_interrupt_routine(SW_INTERRUPT_3, SGI3_ISR);   // register sgi isr
        printf("Running the GIC Test \n");
        printf("Starting and sending SGIs to secondary CPUs for \"hello world\" \n\n");
        // start second cpu
        start_secondary_cpu(1, &gic_test);
        while (gicTestDone) ;   //cpu0 wait until test is done, that is until cpu3 completes
its SGI.
        //writel((readl(SRC_BASE_ADDR + SRC_SCR_OFFSET) & ~(7 << 22)),
        //       (SRC_BASE_ADDR + SRC_SCR_OFFSET));  // put other cores back into reset with
SRC module
        printf("\nEND of GIC Test \n");
    } else {                    //other cpus
        //printf("secondary main cpu: %d\n", cpu_id);
        if (cpu_id == 3) {
            //void send_sgi(unsigned int ID, unsigned int target_list, unsigned int
filter_list);
            send_sgi(SW_INTERRUPT_3, 1, 0); // send to cpu_0 to start sgi loop;
        } else {
            start_secondary_cpu(cpu_id + 1, &gic_test);
        }
        while (1) ;             //do nothing wait to be interrupted
    }
}
```

3.5 Initializing and using the GIC driver

Typically, the startup routine (in the case of PLATLIB, startup_imx6x) is used to initialize the GIC distributor and each of the available GIC CPU interfaces for both the primary and secondary cores. The startup routine uses the available GIC driver functions to initialize the GIC. When executing startup on the primary core, the GIC distributor and the GIC CPU interface for the primary core are initialized as shown below:

```
mov r0, #0xFF   @ 0xFF is lowest priority level
bl set_cpu_priority_mask

bl enable_gic_processor_interface
bl enable_GIC
```

When a secondary core is brought up and executes the startup routine, only its GIC CPU interface needs to be initialized because the GIC distrubutor only needs to be initialized once. An example of this is shown below:

Example GIC secondary CPU initialization

```
secondary_cpus_init:
  mov r0, #0xFF   @ 0xFF is lowest priority level
  bl set_cpu_priority_mask
  bl enable_gic_processor_interface   enable_gic_processor_interface
```

Since the GIC distributor and CPU interfaces are initialized during startup, each module driver does not need to access any of these functions to initialize these GIC interfaces. Because the low-level initialization is already taken care of, only the following items need to be initialized to enable interrupts for a given source:

- Enable interrupt sources (unmask interrupts) at the module level.
- Register the module interrupt service routine (ISR).
- Enable the interrupt to one of the available CPUs.

The following shows a generic example:

Example -3. Initializing module interrupts

```
enable_module_interrupt();
register_interrupt_routine(module_IRQ_ID, module_ISR);  //register ISR
enable_interrupt(module_IRQ_ID, CPU_0, 0); //gic function to enable interrupt source
                                           //init to CPU_0, with max priority
```

3.6 Source code structure

Table 3-5. Source code file locations

Description	Location
Low-level driver source	./src/sdk/cortex_a9/gic.s
Low-level driver header	./src/include/gic.h
Low-level driver source	./src/mx61/interrupt.c
Module unit test	./src/sdk/gic/test/gic_test.c

Chapter 4
Configuring the AUDMUX Driver

4.1 Overview

The Digital Audio Multiplexer (AUDMUX) provides a programmable interconnect device for voice, audio, and synchronous data routing between host serial interfaces (such as the Synchronous Serial Interface Controller (SSI)) and peripheral serial interfaces (audio and voice codecs, also known as coder-decoders). This chapter explains how to configure the AUDMUX driver.

The AUDMUX is dedicated to the SSI only. With the AUDMUX, SSI signals can be multiplexed to different ports without changing the PCB layout.

AUDMUX includes two types of interfaces: internal and external ports. Internal ports connect to the processor serial interfaces, and external ports connect to off-chip audio devices and the serial interfaces of other processors. The desired connectivity is achieved by configuring the appropriate internal and external ports.

AUDMUX includes three internal ports (Port1, Port2, and Port3) and four external ports (Port3, Port4, Port5, and Port6). Each port can be programmed to a full 6-wire SSI interface for asynchronous receive and transmit, or 4-wire (synchronous) or 6-wire (asynchronous) peripheral interfaces.

The following figure shows the structure of the AUDMUX.

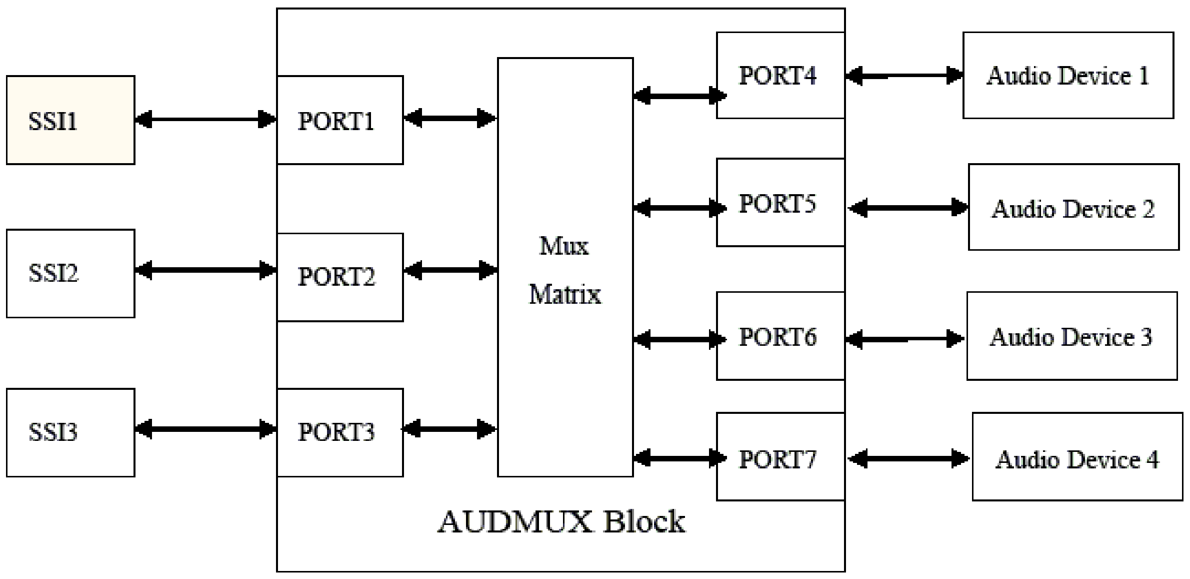


Figure 4-1. AUDMUX structure

The only instance of AUDMUX in i.MX 6Dual/6Quad is located in the memory at the following address:

- AUDMUX base address = 021D 8000h

For each port, the AUDMUX interface provides the following two programmable, 32-bit registers:

- Port Timing Control Register (AUDMUX_PTCRn)
- Port Data Control Register (AUDMUX_PDCRn)

For AUDMUX register definition details, see the i.MX 6Dual/6Quad Reference Manual.

4.2 Features summary

The AUDMUX driver supports:

- Three internal ports
- Four external ports
- Full 6-wire SSI interfaces for asynchronous receive and transmit
- Configurable 4-wire (synchronous) or 6-wire (asynchronous) peripheral interfaces
- Each host interface's capability to connect to any other host or peripheral interface in a point-to-point or point-to-multipoint (network mode)
- Transmit and Receive Data switching to support external network mode

4.3 Clocks

The AUDMUX only requires a peripheral clock and places no restrictions on the clock frequency. Before accessing the AUDMUX register, the peripheral clock must be gated on. Please see the CCM chapter of the i.MX 6Dual/6Quad reference manual for details.

4.4 IOMUX pin mapping

For the i.MX53 SMD board, PORT5 was connected with the SSI codec sgtl5000 in SYNC mode, and the IOMUX pin configuration appeared as shown in the following table:

NOTE

The i.MX 6Dual/6Quad ARD Board does not support audio driver functionality. The i.MX53 SMD board is utilized for functional purposes. When using another board, please check the board schematic for correct pin assignments.

Table 4-1. IOMUX pin map

Signal name	Pin name	ALT
AUD5_RXD	KEY_ROW1	ALT2
AUD5_TXD	KEY_ROW0	ALT2
AUD5_TXC	KEY_COL0	ALT2
AUD5_TXFS	KEY_COL1	ALT2

4.5 Modes of operation

The following table explains the AUDMUX modes of operation:

Table 4-2. Modes of operation

Mode	Description	Configuration
Asynchronous	This port has a 6-wire interface (meaning RxD, TxD, TxCLK,TxFS, RxCLK, RxFS). This mode has additional receive clock (RxCLK) and frame sync (RxFS) signals for receiving (as compared to the synchronous 4-wire interface.)	AUDMUX_PTCR[SYN] = 0b
Synchronous	This port has a 4-wire interface (that is, RxD, TxD, TxCLK, and TxFS). The receive data timing is determined by TxCLK and TxFS.	AUDMUX_PTCR[SYN] = 1b

Table continues on the next page...

Table 4-2. Modes of operation (continued)

Mode	Description	Configuration
Normal	This port is connected in a point-to-point configuration (as a master or a slave). The RXDSEL [2:0] setting selects the transmit signal from any port.	AUDMUX_PDCR1[MODE] = 0b
Internal Network	The output of the AND gate is routed (via the output of the port) to the RxD signal of the corresponding host interface. The INMMASK bit vector selects the transmit signals of the ports that are to be connected in network mode. An AND Operator receives the transmit signals from the AUDMUX ports (TxDn_in) to form the output. In internal network mode, only one device can transmit in its predesignated timeslot and all other transmit signals must remain in high-impedance state and pulled-up.	AUDMUX_PDCR1[MODE] = 1b

4.5.1 Port timing mode

All ports can be configured in one of two timing modes: synchronous (SYNC) and asynchronous (ASYNC). Both timing modes affect the usage of RxCLK and RxFS. AUDMUX_PTCR[SYN] can set SYNC and ASYNC modes.

4.5.2 Port receive mode

Each port has two receive modes (normal mode and internal network mode) that affect which data lines are used to create the RxD line for the corresponding host interface.

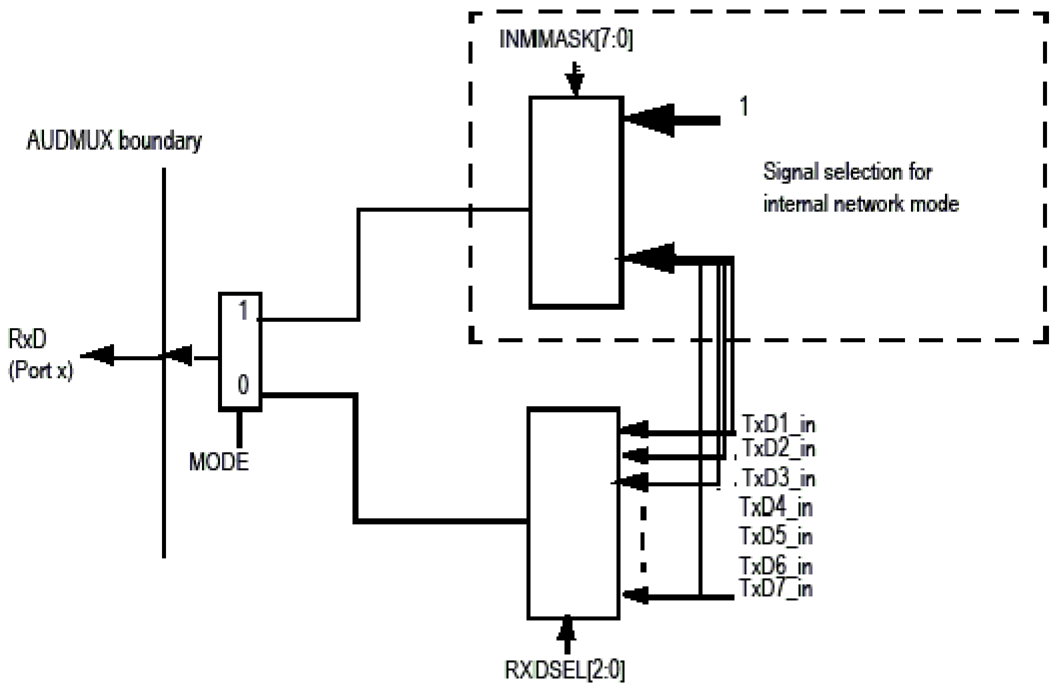


Figure 4-2. Port receive mode

Normal mode or internal network mode can be selected using AUDMUX_PDCRn[MODE]. When internal network mode is selected, AUDMUX_PDCRn[RXDSEL] is ignored and AUDMUX_PDCRn[INMMASK] determines which RxD signals are ANDed together.

4.6 Port configuration

4.6.1 Signal direction

The direction of TxFS, TxCLK, RxFS, and RxCLK can be programmed to configure the SSI interface connecting to the internal port of AUDMUX as a master or a slave of the bus. The following fields control direction of TxFS, TxCLK, RxFS, and RxCLK:

- AUDMUX_PTCRn[TFSDIR]
- AUDMUX_PTCRn[TCLKDIR]
- AUDMUX_PTCRn[RFSDIR]
- AUDMUX_PTCRn[RCLKDIR]

If the signal of the port is set as an output, then a source signal should be selected. For example, if AUDMUX_PTCRn[TFSDIR] is set, the AUDMUX_PTCRn[TFSEL] bits should be programmed to select which port will supply the source TxFS signal.

4.6.2 AUDMUX default setting

The default port-to-port connections are as follows:

- Port 1 to Port 6-Port 6 provides the clock and frame sync.
- Port 2 to Port 5-Port 5 provides the clock and frame sync.
- Port 3 to Port 4-Port 4 provides the clock and frame sync.
- Port 7 to Port 7 (in data loopback mode).

4.6.2.1 Example: Port 2 to Port 5

Assume that the SSI audio codec is connected to port5 (external). Using the default setting, SSI controller 2 (connecting with the internal port2) and the codec will be connected together. The SSI controller 2 is the slave of the SSI bus and the codec is the master.

The registers, related to port2 and port5, and their reset values are as follows:

Table 4-3. Port2 and Port5 Example

Register	Reset Value
AUDMUX_PTCR2	A500_0800h
AUDMUX_PDCR2	0000_8000h
AUDMUX_PTCR5	0000_0800h
AUDMUX_PDCR5	0000_2000h

The following figure shows the multiplexing and direction of the signals related to port2 and port5.

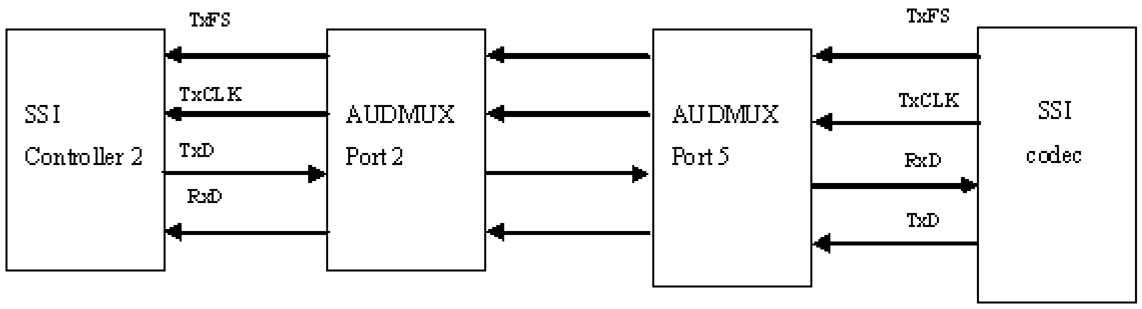


Figure 4-3. Signal muxing and direction of port2 & port5 under default setting

4.7 Port configuration for SSI sync mode

For most audio codecs, the SSI sync mode will be utilized. A function named audmux_route configures the audmux ports for SSI sync mode. In this case, the timing mode of the ports is set as synchronous mode, and the receive mode of the port is set as normal mode. The direction of the clock signals (TxClk, TxFS) depends on the SSI controller being master or not.

4.8 Pseudocode for audmux_route

```
/*!
 * Set audmux port according to ssi mode (master/slave).
 * Set the audumx ports in sync mode (which is the default status for most codec).
 *
 * @param   intPort     the internal port to be set
 * @param   extPort     the external port to be set
 * @param   is_master   ssi mode(master/slave) of ssi controller, for example if
 * is_master=AUDMUX_SSI_MASTER,then the ssi controller is the master of the ssi bus. That is,
 * it supplies the bit clock frame sync signal, while the codec is the slave of the bus.
 * @return  0 if it succeeds
 *          -1 if it fails
 */
uint32_t audmux_route(uint32_t intPort, uint32_t extPort, uint32_t is_master)
{
          Check_the_Parameter_Valid();
          //Configure the internal port firstly.
          If(ssi_controller_is_master){
                    Set_clk_signals_as_input(intPort);
          }else{
                    Set_clk_signals_as_output(intPort);
                    Set_clk_sigtnals_from(extPort);
          }
          // Then configure the external port
          If(ssi_controller_is_master){
                    Set_clk_signals_as_output(extPort);
                    Set_clk_sigtnals_from(intPort);
          }else{
                    Set_clk_signals_as_input(extPort);
          }
          return 0;
}
```

4.9 Pseudocode for audmux_port_set

```
/* Set ptcr and pdcr of the audmux port
 *
 * @param   port        the port to be set
 * @param   ptcr        ptcr value to be set
 * @param   pdcr        pdcr value to be set
 * @return  0 if succeeded
 *          -1 if failed.
 */
uint32_t audmux_port_set
         (uint32_t port, uint32_t ptcr, uint32_t pdcr)
```

```
{
    uint32_t pPTCR, pPDCR;
    if ((port < AUDMUX_PORT_INDEX_MIN) || (port > AUDMUX_PORT_INDEX_MAX)) {
        return -1;
    }
    pPTCR = AUDMUX_BASE_ADDR + AUDMUX_PTCR_OFFSET(port);
    pPDCR = AUDMUX_BASE_ADDR + AUDMUX_PDCR_OFFSET(port);
    writel(ptcr, pPTCR);
    writel(pdcr, pPDCR);
    return 0;
}
```

4.10 Source code structure

Table 4-4. Source code file location

Source code	Location
Test unit driver source	`.src/sdk/audio/test/ssi_playback.c`

Chapter 5
Configuring the eCSPI Driver

5.1 Overview

This chapter provides a quick guide for firmware developers about how to write a device driver for the eCSPI controller. The enhanced configurable serial peripheral interface (eCSPI) is a full-duplex, synchronous, four-wire serial communication block. The eCSPI controller works as a device over the SPI bus, either as a master or a slave, and communicates with other devices according to the chip select (CS) signal's selections. Note that this driver does not implement slave mode.

NOTE

This chapter uses i.MX61 EVB board schematics for pin assignments. For other board types, refer to respective schematics.

There are five instances of eCSPI in the chip. They are located in the memory map at the following addresses:

- eCSPI1 base address - 0200 8000h
- eCSPI2 base address - 0200 C000h
- eCSPI3 base address - 0201 0000h
- eCSPI4 base address - 0201 4000h
- eCSPI5 base address - 0201 8000h

5.2 Features summary

This driver provides the basic eCSPI initialization functionality and R/W in master mode.

5.3 I/O signals

The eCSPI block has below I/O signals:

Table 5-1. eCSPI I/O signals

Signal	I/O	Description	Reset State	Pull Up/Down
SS[3:0]	I/O	Chip selects	1	-
SCLK	I/O	SPI clock	0	Active
MISO	I/O	Master data in; slave data out	0	Passive
MOSI	I/O	Master data out; slave data in	0	-
SPI_RDY	I	Master data out; slave data in	0	Active

The following figure shows the usage when eCSPI is functioning as an SPI master:

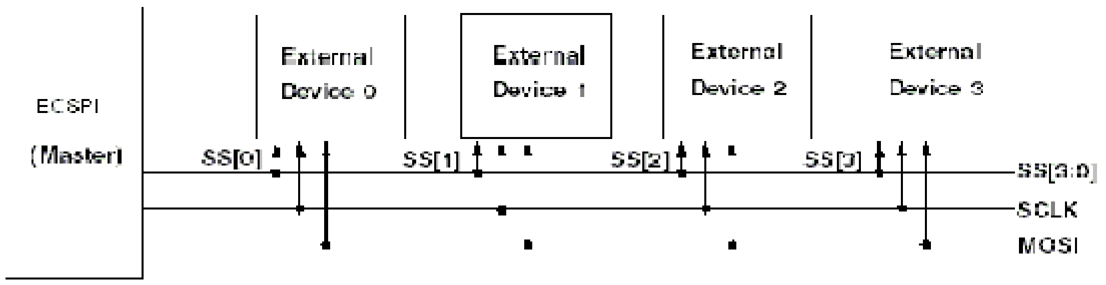


Figure 5-1. eCSPI as SPI master

5.4 eCSPI controller initialization

The necessary initialization process can be summarized as:

1. Pin-mux configuration
2. Clock configuration
3. Controller initialization
4. Controller is ready to transfer data.

5.5 IOMUX pin mapping

Refer to board schematics for correct pin assignments to configure the pin signals. The following table shows eCSPI1 for the i.MX61 EVB board as an example:

Table 5-2. eCSPI1 options

Signals	Option 1	
	PAD	MUX
MOSI	EIM_D18	ALT1
	KEY_ROW0	ALT0
RDY	GPIO_19	ALT4
SCLK	CSIO_DAT4	ALT2
	DISP0_DAT20	ALT2
	EIM_D16	ALT1
	KEY_COL0	ALT0
SS0	CSIO_DAT7	ALT2
	DISP0_DAT23	ALT2
	EIM_EB2	ALT1
	KEY_ROW1	ALT0
SS1	DISP0_DAT15	ALT2
	EIM_D19	ALT1
	KEY_COL2	ALT0
SS2	EIM_D24	ALT3
	KEY_ROW2	ALT0
SS3	EIM_D25	ALT3
	KEY_COL3	ALT0

The pad control of each pin also needs to be configured. Because the clock and data pins have pull-up resistors, these pads can be configured to open drain if the board schematic already has external pull-up resistors for them. Otherwise, they have to be configured to push-pull with a specified pull-up resistor value.

5.6 Clocks

If the eCSPI clock is gated, ungate it in the clock control module (CCM) as follows:

- For eCSPI1, set CCM_CCGR1[CG0] (bits 1-0).
- For eCSPI2, set CCM_CCGR1[CG1] (bits 2-3).
- For eCSPI3, set CCM_CCGR1[CG2] (bits 4-5).
- For eCSPI4, set CCM_CCGR1[CG3] (bits 6-7).
- For eCSPI5, set CCM_CCGR1[CG4] (bits 8-9).

The following figure shows the eCSPI clock source.

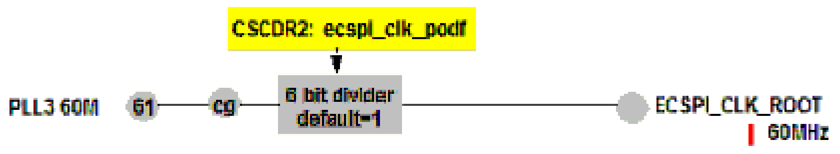
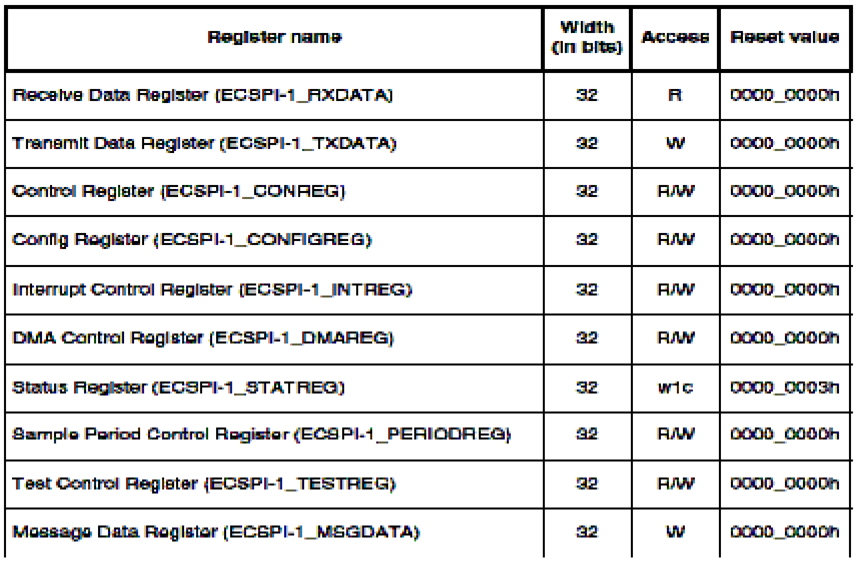


Figure 5-2. eCSPI clock source

To set the frequency of the eCSPI, Set the ecspi_clk_podf field of CCM_CSCDR2. To achieve the expected frequency, set the divider value on the control register of eCSPI controller.

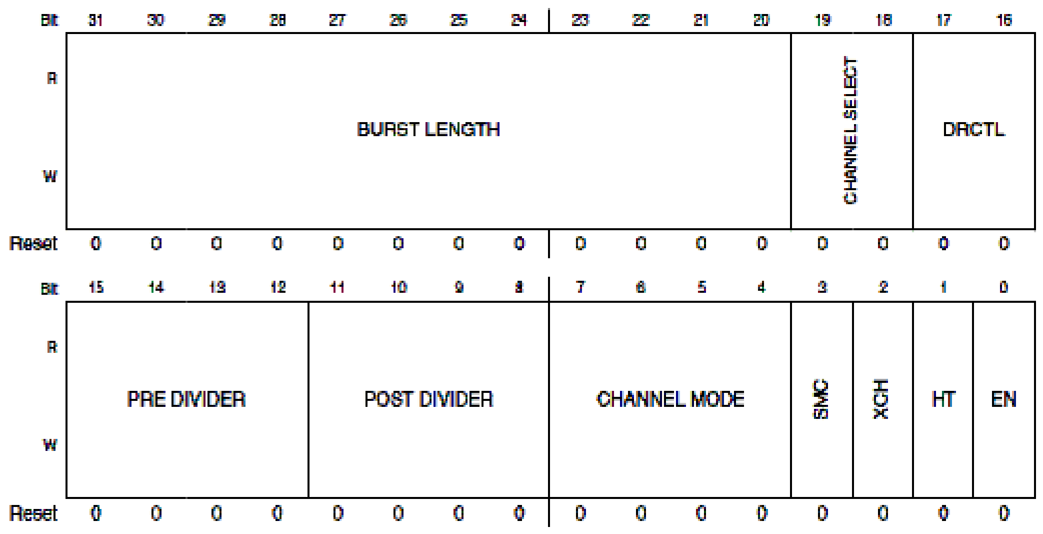
5.7 Controller initialization

The following table shows the eCSPI controller register map:



Register name	Width (in bits)	Access	Reset value
Receive Data Register (ECSPI-1_RXDATA)	32	R	0000_0000h
Transmit Data Register (ECSPI-1_TXDATA)	32	W	0000_0000h
Control Register (ECSPI-1_CONREG)	32	R/W	0000_0000h
Config Register (ECSPI-1_CONFIGREG)	32	R/W	0000_0000h
Interrupt Control Register (ECSPI-1_INTREG)	32	R/W	0000_0000h
DMA Control Register (ECSPI-1_DMAREG)	32	R/W	0000_0000h
Status Register (ECSPI-1_STATREG)	32	w1c	0000_0003h
Sample Period Control Register (ECSPI-1_PERIODREG)	32	R/W	0000_0000h
Test Control Register (ECSPI-1_TESTREG)	32	R/W	0000_0000h
Message Data Register (ECSPI-1_MSGDATA)	32	W	0000_0000h

To initialize the controller, configure the control and configuration registers. The following figure shows the control register's bit map:



The channel select value determines which chip select is used, and the channel mode value determines the mode (master or slave) for each chip select. The user must select master mode because the driver does not implement R/W slave mode functionality.

The configuration register can configure the inactive state of the clock as well as the data and polarity settings. The configuration to the controller should be aligned to the setting in device.

NOTE

The EN bit of control register should be cleared to reset the controller internal logic. Set this bit before setting the configuration register or setting to the configuration register will not have an effect.

To set the SPI bus frequency, configure the pre divider and post divider, as shown in the following equation:

Fspi = Fsource ÷ ((pre + 1) x 2post)

The following figure shows an example using the i.MX61 EVB schematic eCSPI. In this example, SS1(EIM_D19) is selected and master mode is set; therefore, the CONREG bit[19:18] should be 01b and bit 5 should be 1b.

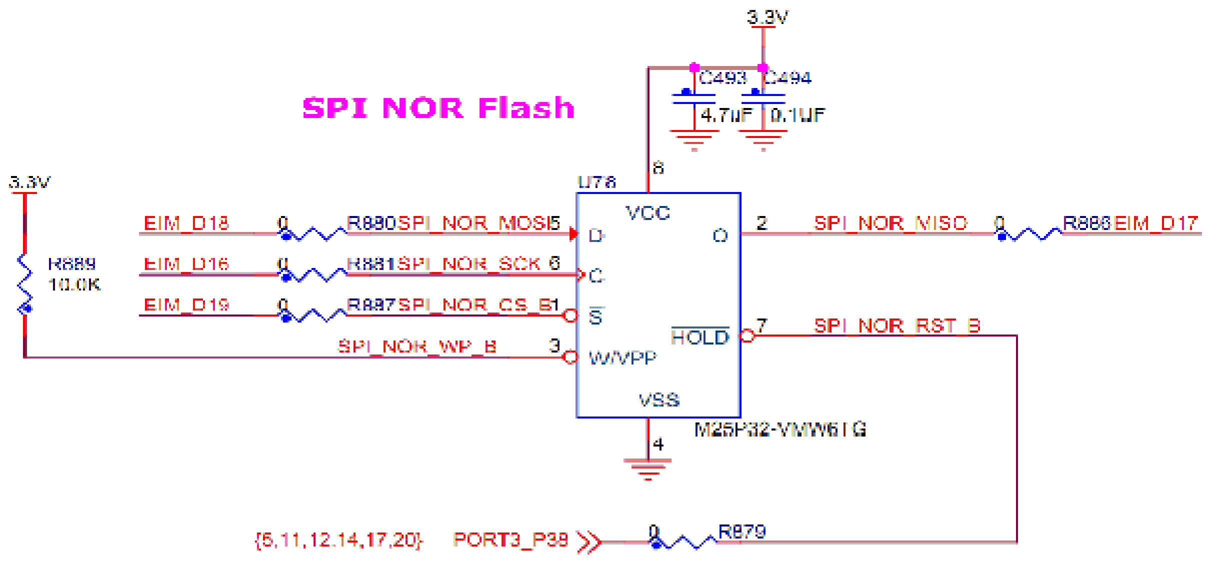


Figure 5-3. eCSPI example configuration

For further details, see the eCSPI chapter in the device reference manual.

5.8 eCSPI data transfers

This section describes how to handle data transfers between the eCSPI controller and the device. In SPI master mode, the controller initiates the transfer actively, and then reads the response from the slave.

The following figure shows the flow chart for data transfer in master mode.

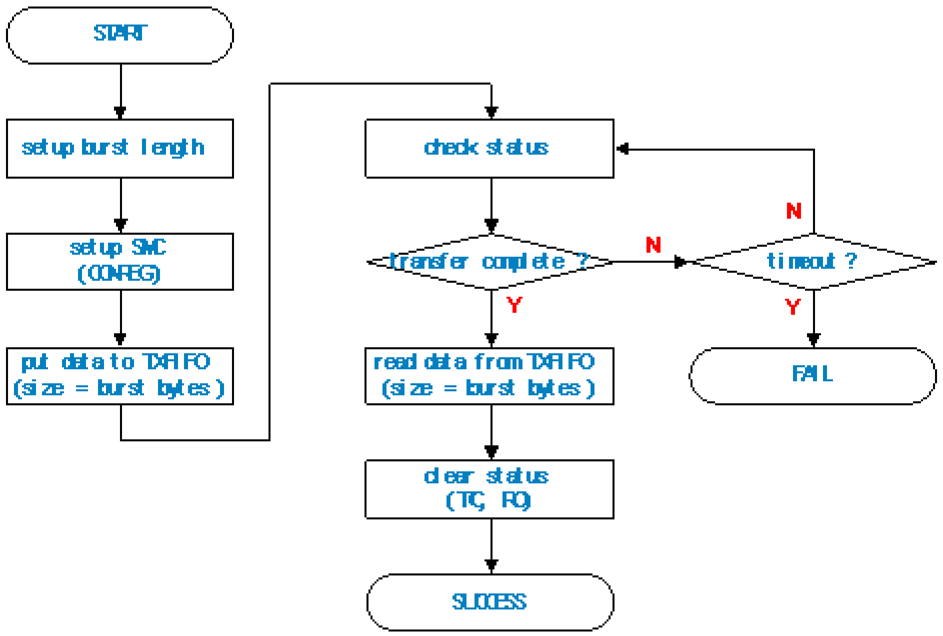


Figure 5-4. Master mode data transfer flow chart

During the SPI transfer, burst length bits of data should be written to TxFIFO.

1. Set the burst length and SMC first, and then you can write the burst length of data to TxFIFO.
2. The transfer complete (TC) bit is set only when the controller receives the same burst length of data from the slave as the burst length of data that the controller sent.
3. When this bit is set, burst length bits of data can be read from RxFIFO.

5.9 Application program interface

All the external function calls and variables are located in the inc/ecspi_ifc.h file. The following table explains the APIs.

Table 5-3. eCSPI APIs

API	Description	Parameters	Return
int ecspi_open(dev_ecspi_e device, param_ecspi_t parameter);	Initializes the eCSPI controller as specified by device; parameter specifies the configuration.	*device*: device instance/ enumeration *parameter*: device configuration. See ecspi_ifc.h for detailed bit field definitions.	• TRUE on success • FALSE on fail
int ecspi_close(dev_ecspi_e device);	Disables the eCSPI device.	*device*: device instance/ enumeration	• TRUE on success • FALSE on fail.

Table continues on the next page...

Table 5-3. eCSPI APIs (continued)

API	Description	Parameters	Return
int ecspi_ioctl(dev_ecspi_e , param_ecspi_t);	Resets the eCSPI controller. Unlike ecspi_open it does not configure the clock or IOMUX.	*device*: device instance/ enumeration *parameter*: device configuration. See ecspi_ifc.h for detailed bit field definitions.	• TRUE on success • FALSE on fail.
int ecspi_xfer(dev_ecspi_e device, uint8_t *tx_buf, uint8_t *rx_buf, uint16_t burst_len);	Initiates data transfer	*device*: device instance/ enumeration: • tx_buf: data to be sent • rx_buf: buffer to put data returned • burst_len: length in bits of the data to exchange	• TRUE on success • FALSE on fail

5.10 Source code structure

Table 5-4. Source code file locations

Description	Location
Driver	
Low level driver for ecspi controller	src/sdk/spi/drv/ecspi.c
Header file for ecspi controller	src/sdk/spi/drv/ecspi.h
Header file for ecspi external interface	src/sdk/spi/inc/ecspi_ifc.h
Unit Test	
Test file for SPI	src/sdk/spi/test/spi_test.c
Header file for SPI test	src/sdk/spi/test/spi_test.h
Driver for Numonyx SPI NOR flash	src/sdk/spi/test/spi_nor_numonyx.c
Header file for Numonyx SPI NOR flash driver	src/sdk/spi/test/spi_nor_numonyx.h

Chapter 6
Configuring the EPIT Driver

6.1 Overview

This chapter explains how to configure the EPIT driver. EPIT is a 32-bit set-and-forget timer that begins counting after it is enabled by software. It is capable of providing precise interrupts at regular intervals with minimal processor intervention.

This low-level driver helps to configure the EPIT for some functions like delay or system tick.

There are two instances of EPIT in i.MX 6Dual/6Quad. They are located in the memory map at:

- EPIT1 base address = 020D 0000h
- EPIT2 base address = 020D 4000h

6.2 Features summary

This low-level driver supports:

- The usage of three different clock sources for the 32-bit down counter.
- Set-and-forget and free-running modes
- On the fly counter reprogramming
- Can be programmed to be active in low-power and debug modes
- Interrupt generation when the counter reaches the compare value

6.3 Modes of operation

The following table explains the EPIT modes of operation:

Table 6-1. Modes of operation

Mode	What it does
Set-and-forget mode:	The EPIT counter starts the count down from the load register EPIT_EPITLR value to zero. When the counter reaches 0, it reloads the value from EPIT_EPITLR, and starts to count down towards 0. For this, the reload mode must be enabled. The counter does not have to be at 0 for it to be loaded with a different start value. It can be achieved by setting EPIT_CR[IOVW].
Free-running mode:	The EPIT counter endlessly counts down from FFFF FFFFh to 0h. The reload mode must be disabled.

6.4 Output compare event

The EPIT has the capability to change the state of an output signal (EPITO) on a compare event. The behavior of that signal is configurable in the driver and could be set to:

- OUTPUT_CMP_DISABLE = output disconnected from the external signal EPITO.
- OUTPUT_CMP_TOGGLE = toggle the output.
- OUTPUT_CMP_CLEAR = set the output to a low level.
- OUTPUT_CMP_SET = set the output to a high level.

Use the following function to generate an output event on compare:

- epit_get_compare_event()

6.5 Clocks

EPIT receives three clock signals.

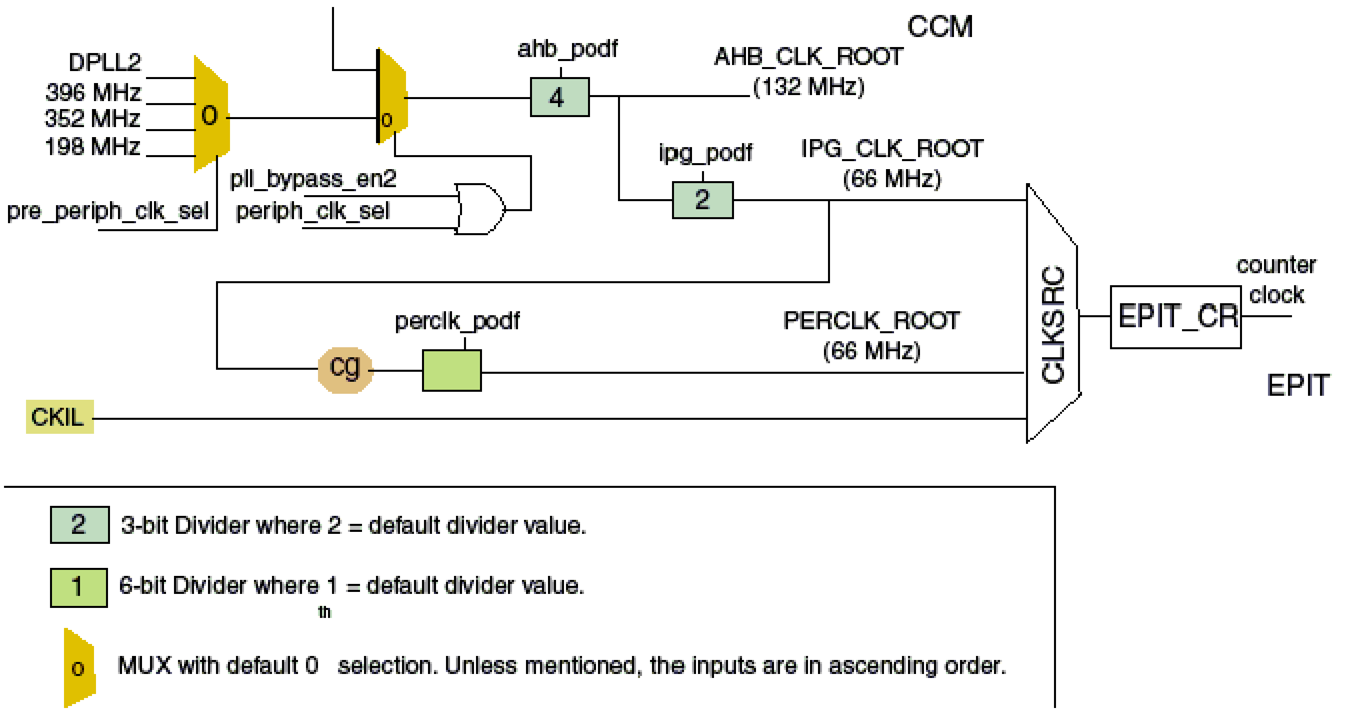


Figure 6-1. Reference clocks

The following table explains the EPIT reference clocks:

Table 6-2. Reference clocks

Clock	Name	Description
Low-frequency clock	CKIL	Global 32,768 Hz source from the CKIL input. It remains on in low power mode.
High-frequency clock	PERCLK_ROOT	Provided by the CCM. It can be disabled in low power mode.
Peripheral clock	IPG_CLK_ROOT	Typically used in normal operation. It is provided by CCM. It cannot be powered down.

Because the frequency of DPLL2 and various dividers is system dependent, the user may need to adjust the driver's frequency. To do this, change the freq member of the hw_module structure defined into ./src/include/io.h

For example, take the following non-default divider values:

- DPLL2 is set to output 396 MHz
- ahb_podf divides by 3
- ipg_podf divides by 2

In this example, IPG_CLK = 132 MHz and PERCLK = 66 MHz.

The driver handles the clock gating on the source clock.

6.6 IOMUX pin mapping

The EPIT can change the state of an output signal (EPITO) on a compare event. The IOMUX should route these signals to the appropriate pins. The IOMUX configuration is board dependent and can be handled with the IOMUX tool.

Table 6-3. Pin assignments for i.MX 6Dual/6Quad

Signal	IOMUXC Setting for EPIT1			iOMUXC Setting for EPIT2		
	PAD	MUX	SION	PAD	MUX	SION
EPITO	EIM_D19	ALT6	1	EIM_D20	ALT6	1
EPITO	GPIO_0	ALT4	1	GPIO_8	ALT2	1
EPITO	GPIO_7	ALT2	1	-	-	-

6.7 Resets and interrupts

The driver sets EPITCR[SWR] in the function epit_init() to reset the module during initialization.

The external application is responsible for creating the interrupt subroutine. The address of this routine is passed through the structure hw_module defined in ./src/include/io.h. It is initialized by the application and used by the driver for various configurations.

All interrupt sources are listed in the "Interrupts and DMA Events" chapter of the device reference manualIn the SDK, the list is provided into ./src/include/mx61/soc_memory_map.h.

6.8 Initializing the EPIT driver

Before using the EPIT timer in a system, prepare a structure that provides the essential system parameters to the driver. This is done through the hw_module structure defined into ./src/include/io.h.

Example for the EPIT used as system timer:

```
struct hw_module g_system_timer = {
    "EPIT1 used as system timer",
    EPIT1_BASE_ADDR,
    66000000,
    IMX_INT_EPIT1,
    &default_interrupt_routine,
};
```

The address of this structure is used by most functions listed below.

```
/*!
 * Initialize the EPIT timer.
 *
 * @param   port - pointer to the EPIT module structure.
 * @param   clock_src - source clock of the counter: CLKSRC_OFF,
 *                      CLKSRC_IPG_CLK, CLKSRC_PER_CLK, CLKSRC_CKIL.
 * @param   prescaler - prescaler of source clock from 1 to 4096.
 * @param   reload_mode - counter reload mode: FREE_RUNNING or
 *                        SET_AND_FORGET.
 * @param   load_val - load value from where the counter start.
 * @param   low_power_mode - low power during which the timer is enabled:
 *                           WAIT_MODE_EN and/or STOP_MODE_EN.
 */
void epit_init(struct hw_module *port, uint32_t clock_src,
               uint32_t prescaler, uint32_t reload_mode,
               uint32_t load_val, uint32_t low_power_mode)
/*!
 * Setup EPIT interrupt. It enables or disables the related HW module
 * interrupt, and attached the related sub-routine into the vector table.
 *
 * @param   port - pointer to the EPIT module structure.
 */
void epit_setup_interrupt(struct hw_module *port, uint8_t state)
/*!
 * Enable the EPIT module. Used for instance when the epit_init is done, and
 * other interrupt related settings are ready.
 *
 * @param   port - pointer to the EPIT module structure.
 * @param   load_val - load value from where the counter starts.
 * @param   irq_mode - interrupt mode: IRQ_MODE or POLLING_MODE.
 */
void epit_counter_enable(struct hw_module *port, uint32_t load_val,
                         uint32_t irq_mode)
/*!
 * Disable the counter. It saves energy when not used.
 *
 * @param   port - pointer to the EPIT module structure.
 */
void epit_counter_disable(struct hw_module *port)
/*!
 * Get the output compare status flag an clear if sets.
 * This function is typically used for polling method.
 *
 * @param   port - pointer to the EPIT module structure.
 * @return  the value of the compare event flag.
 */
uint32_t epit_get_compare_event(struct hw_module *port)
/*!
 * Reload the counter with a known value.
 *
 * @param   port - pointer to the EPIT module structure.
 */
void epit_reload_counter(struct hw_module *port, uint32_t load_val)
```

6.9 Testing the EPIT driver

The EPIT driver runs the following tests:

- Delay test
- Tick test

6.9.1 Delay test

The delay test shows the usage of the EPIT as a timer for a delay function hal_delay_us(), which is programmed into ./src/sdk/timer/drv/imx_timer/timer.c. This function serves as a use case example of EPIT in a system. The test displays the elapsed numbers of seconds. This test runs for 10 seconds and then returns to the main test menu.

6.9.2 Tick test

In the tick test, the EPIT is configured to generate an interrupt every 10 ms. This is similar to an operating system tick timer with one hundred interrupts occurring per second. After each second has elapsed, the test displays the equivalent number of received ticks. This runs for 10 seconds and then returns to the main test menu.

6.9.3 Running the tests

To run the EPIT tests, the SDK builds the test with the following command:

```
./tools/build_sdk -target mx61 -board ard -board_rev 1 -test epit
```

This generates the following ELF and binary files:

- `./output/mx61/bin/mx61ard-epit-sdk.elf`
- `./output/mx61/bin/mx61ard-epit-sdk.bin`

6.10 Source code structure

Table 6-4. Source code file locations

Description	Location
Low-level driver source	`./src/sdk/timer/drv/imx_timer/epit.c`
Low-level driver header	`./src/include/epit.h`

Unit tests	`./src/sdk/timer/test/epit_test.c`
	`./src/sdk/timer/test/epit_test.h`

SDK common function for delay	`./src/sdk/timer/drv/imx_driver/timer.c`
	`./src/include/timer.c`

Chapter 7
Configuring the FlexCAN Modules

7.1 Overview

This chapter explains how to configure and use the FlexCAN modules. The FlexCAN (flexible controller area network) module is a communication controller that implements the CAN protocol (CAN 2.0B).

The following figure shows a general block diagram, which illustrates the main subblocks implemented in the FlexCAN module. This includes two embedded memories: one for supporting message buffers (MB) and another for storing Rx individual mask registers. For more details, refer to the FlexCAN chapter of the reference manual.

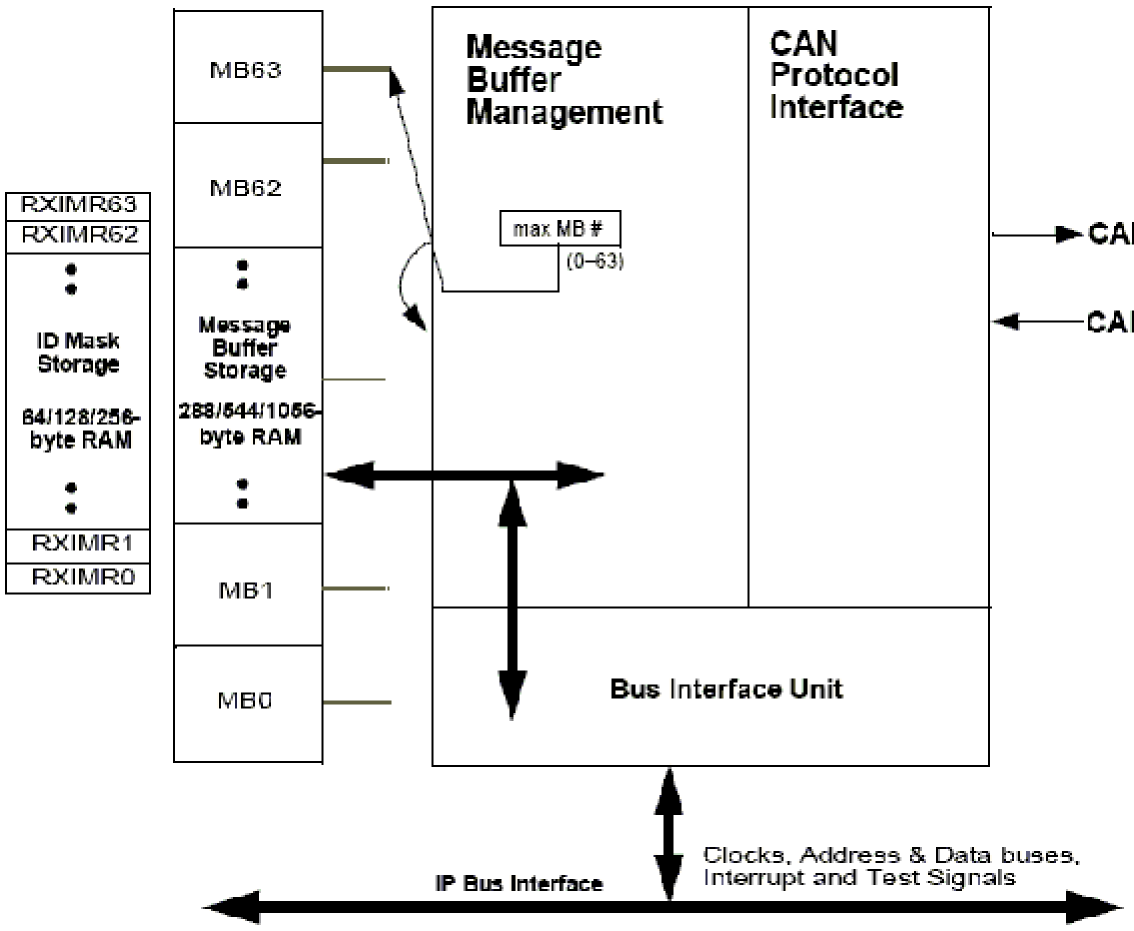


Figure 7-1. FlexCAN block diagram

There are two module instances of the FlexCAN module in the chip, which are memory mapped to locations:

- CAN1 base address = 0209 0000h
- CAN2 base address = 0209 4000h

7.2 Features summary

This low-level driver supports:

- Up to 64 message buffers
- Standard CAN initialization routine
- Maskable interrupts for each message buffer
- Use of data structures to define the module register memory map

7.3 Modes of operation

The following table summarizes the FlexCAN modes of operation.

Table 7-1. FlexCAN modes of operation

Mode	What it does
Normal mode (User or Supervisor)	The module can receive and transmit message frames; errors are handled normally, and all CAN protocol functions are enabled. User and supervisor mode differ in given access to some restricted control registers.
Freeze mode	The module cannot transmit or receive message frames, and synchronicity to the CAN bus is lost.
Listen-only mode	Transmission is disabled, and all error counters are frozen. The module operates in a CAN error passive mode. Only messages acknowledged by another CAN station are received.
Loopback mode	The module performs an internal loopback that can be used for a self-test operation. The bit stream output of the transmitter is internally fed back to the receiver input.
Module disable	This is a low power mode in which the clocks to the flexCAN module are disabled.
Stop mode	This is a low power mode. The module puts itself into an inactive state then it informs the ARM core that the clocks can be shutdown globally. Exit from this mode can be achieved when activity is detected on the CAN bus.

7.4 Clocks

The main clock source input for the FlexCAN module is the CAN_CLK_ROOT clock, which is derived as shown in the following image. Additionally, the CCM module has two CAN-related clock gating signals that are configurable with the CCM CGR0 register. The can*_serial_clock_enable and can*_clock_enable signals must both be gated on for the FlexCAN module to function properly.

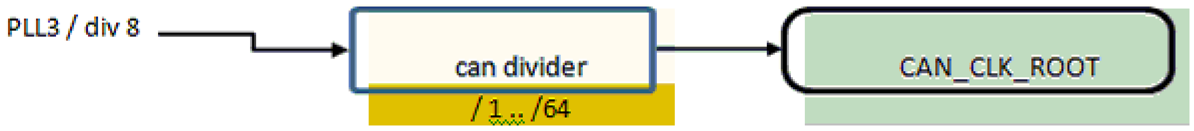


Figure 7-2. Clocking figure

Table 7-2. Clock sources

Clock	Name	Description
CAN protocol engine (PE) clock (also called the CAN protocol interface clock, or CPI)	CAN_CLK_ROOT	FlexCAN module main clock (PE clock).
Serial clock	Sclock	The Sclock period defines the time quantum of the CAN protocol. The prescaler division factor (PRESDIV bit field in the CTRL register) determines the ratio between CAN_CLK_ROOT and the serial clock.

7.5 IOMUX pin mapping

The IOMUX configuration is board dependent and can be handled by the IOMUX tool. The following table shows the available mux options for the FlexCAN module signals.

Table 7-3. FlexCAN IOMUX options

Signals	Option 1		Option 2		Option 3	
	PAD	MUX	PAD	MUX	PAD	MUX
Module Instance: CAN1						
TXCAN	KEY_COL2	ALT2	SD3_CMD	ALT2	GPIO_7	ALT3
RXCAN	KEY_ROW2	ALT2	SD3_CLK	ALT2	GPIO_8	ALT3
Module Instance: CAN2						
TXCAN	KEY_COL4	ALT0	SD3_DAT0	ALT2	-	-
RXCAN	KEY_ROW4	ALT0	SD3_DAT1	ALT2	-	-

NOTE

: Daisy Chain Configuration Required. Because the RXCAN input signals have multiple mux options, users must also configure the associated daisy chain select registers. Refer to the IOMUXC chapter of the reference manual for more details.

7.6 Resets and interrupts

7.6.1 Module reset

The FlexCAN module can be reset in two ways. First, the FlexCAN module is reset when the system powers up (and/or there is a system reset). Additionally, the FlexCAN module may also be reset by asserting the software reset bit (bit 25) in the Module Configuration Register. The firmware driver provides the following software reset function:

Software reset

```
void can_sw_reset(struct hw_module *port){
    volatile struct mx_can_control *can_ctl = (volatile struct mx_can_control *)port->base;

    can_ctl->mcr |= (1<<25); //assert SOFT_RST
    while(can_ctl->mcr & (1<<25)); // poll until complete
}
```

7.6.2 Module interrupts

The FlexCAN module can generate an interrupt from 70 interrupt sources:

- 64 interrupts, one from each message buffer
- 6 other general sources (MBs OR'ed together, Bus Off, Error, Tx warning, Rx Warning, Wake-Up)

All FlexCAN interrupt sources are OR'ed together to a single interrupt source to the ARM GIC. The interrupt service routine determines which event actually triggered the interrupt. Similarly, each message buffer can generate an interrupt on either a Tx or Rx event, but both events (Rx/Tx events) trigger a single interrupt source. The interrupt service routine needs to read the message buffer that triggered the event in order to distinguish which type of event occurred.

The firmware driver provides functions for enabling or disabling (setting or clearing each interrupt mask or imask bit) interrupts for each individual message buffer as shown in the following example:

Message buffer interrupts

```
void can_enable_mb_interrupt(struct hw_module *port, uint32_t mbID)
{
    volatile struct mx_can_control *can_ctl = (volatile struct mx_can_control *)port->base;
    if (mbID < 32) {
        can_ctl->imask1 |= (1 << mbID);
    } else if (mbID < 64) {
        can_ctl->imask2 |= (1 << (mbID - 32));
    }
}
void can_disable_mb_interrupt(struct hw_module *port, uint32_t mbID)
{
    volatile struct mx_can_control *can_ctl = (volatile struct mx_can_control *)port->base;
    if (mbID < 32) {
        can_ctl->imask1 &= ~(1 << mbID);
    } else if (mbID < 64) {
        can_ctl->imask2 &= ~(1 << (mbID - 32));
    }
}
```

To enable FlexCAN interrupts to the system to one of the available cores, however, use the GIC-related functions for enabling interrupts. See Testing the driver, for an example.

7.7 Initializing the FlexCAN module

To initialize the FlexCAN module, use the following steps:

1. Run the can_init function, which will configure iomux, issue a module software reset, initialize the configuration register, initialize the control register, initialize the message buffers to zero, and disable all message buffer interrupt mask registers.
2. Run the set_can_mb function to initialize the message buffers.
3. Run the can_exit_freeze function to exit freeze mode and allow module to transmit or receive data.

7.8 Testing the driver

The FlexCAN module test is set up to transmit eight message buffers, each with different byte lengths of data. This test requires that the board has both FlexCAN modules available and that the two ports are connected together to complete the loopback. In this example, the CAN1 module is used to transmit the message buffers, while CAN2 is used to receive them. This is shown below:

FlexCAN unit test

```
/* CAN module data structures */
static struct hw_module can1_port = {
    "CAN1",
    CAN1_BASE_ADDR,
    30000000,
    IMX_INT_CAN1,
};
static struct hw_module can2_port = {
    "CAN2",
    CAN2_BASE_ADDR,
    30000000,
    IMX_INT_CAN2,
    &can2_rx_handler,
};
uint32_t can_test_count;
/*! ------------------------------------------------------------
 * CAN Test (loopback can1/can2 ports)
 * ------------------------------------------------------------
 */
void flexcan_test(void)
{
    int i;
    printf("\n---- Running CAN1/2 loopback test ----\n");
    can_test_count = 0;
    can_init(&can1_port, CAN_LAST_MB);  // max 64 MB 0-63
    can_init(&can2_port, CAN_LAST_MB);  // last mb is MB[63]
    printf("CAN1-TX and CAN2-RX\n");
    // configure CAN1 MBs as Tx, and CAN2 MBs as Rx
    // set-up 8 MBs for the test
    for (i = 1; i < 9; i++) {
        set_can_mb(&can1_port, i, 0x0c000000 + (i << 16), 0x0a000000 + (i << 20), 0x12345678,
                   0x87654321);
        set_can_mb(&can2_port, i, 0x04000000 + (i << 16), 0x0a000000 + (i << 20), 0, 0);
        can_enable_mb_interrupt(&can2_port, i); // enable MB interrupt for idMB=i
    }
    //enable CAN2 interrupt
    register_interrupt_routine(can2_port.irq_id, can2_port.irq_subroutine);
    enable_interrupt(can2_port.irq_id, CPU_0, 0);   // to cpu0, max priority (0)
    // init CAN1 MB0
    can_exit_freeze(&can2_port);    // Rx
    can_exit_freeze(&can1_port);    // Tx
    while (!(can_test_count)) ;
    can_freeze(&can2_port);     // Rx
    can_freeze(&can1_port);     // Tx
    printf("%d MBs were transmitted \n", can_test_count);
    printf("---- CAN1/2 test complete ----\n");
}
```

As shown above, the GIC functions **register_interrupt_routine** and **enable_interrupt** enabled the CAN2 interrupt source to CPU_0. The CAN2 interrupt service routine is then checked to see which message buffer triggered the interrupt. It prints the message buffer to the terminal as shown below:

FlexCAN test interrupt service routine

```
/*
 * Can2 receive ISR function
*/
void can2_rx_handler(void)
{
  int i = 0;
  volatile struct mx_can_control *can_ctl = (volatile struct mx_can_control *)can2_port.base;
    if (can_ctl->iflag1 != 0) {
        for (i = 0; i < 32; i++) {
            if (can_ctl->iflag1 & (1 << i)) {
                can_ctl->iflag1 = (1 << i); //clear interrupt flag
                printf("\tCAN2 MB:%d Recieved:\n", i);
                print_can_mb(&can2_port, i);
                can_test_count++;
            }
        }
    } else if (can_ctl->iflag2 != 0) {
        for (i = 0; i < 32; i++) {
            if (can_ctl->iflag2 & (1 << i)) {
                can_ctl->iflag1 = (1 << i); //clear interrupt flag
                printf("\tCAN2 MB:%d Recieved:\n", i + 32);
                print_can_mb(&can2_port, i + 32);
                can_test_count++;
            }
        }
    }
}
```

The expected output from this test is:

```
---- Running CAN1/2 loopback test ----
CAN1-TX and CAN2-RX
        CAN2 MB:1 Recieved:
        MB[1].cs    = 0x201000f
        MB[1].id    = 0xa100000
        MB[1].data0 = 0x12000000
        MB[1].data1 = 0x353deb2
        CAN2 MB:2 Recieved:
        MB[2].cs    = 0x2020048
        MB[2].id    = 0xa200000
        MB[2].data0 = 0x12340000
        MB[2].data1 = 0x353deb2
        CAN2 MB:3 Recieved:
        MB[3].cs    = 0x2030088
        MB[3].id    = 0xa300000
        MB[3].data0 = 0x12345600
        MB[3].data1 = 0x353deb2
        CAN2 MB:4 Recieved:
        MB[4].cs    = 0x20400d0
        MB[4].id    = 0xa400000
        MB[4].data0 = 0x12345678
        MB[4].data1 = 0x353deb2
        CAN2 MB:5 Recieved:
        MB[5].cs    = 0x2050122
        MB[5].id    = 0xa500000
        MB[5].data0 = 0x12345678
        MB[5].data1 = 0x87000000
        CAN2 MB:6 Recieved:
        MB[6].cs    = 0x206017b
        MB[6].id    = 0xa600000
        MB[6].data0 = 0x12345678
        MB[6].data1 = 0x87650000
        CAN2 MB:7 Recieved:
        MB[7].cs    = 0x20701db
        MB[7].id    = 0xa700000
        MB[7].data0 = 0x12345678
        MB[7].data1 = 0x87654300
        CAN2 MB:8 Recieved:
        MB[8].cs    = 0x2080244
        MB[8].id    = 0xa800000
        MB[8].data0 = 0x12345678
        MB[8].data1 = 0x87654321
8 MBs were transmitted
---- CAN1/2 test complete ----
```

7.9 Source code structure

Table 7-4. Source code file locations

Description	Location
Low-level driver source	./src/sdk/flexcan/drv/can.c
Low-level driver header	./src/sdk/flexcan/drv/can.c
Module unit test	./src/sdk/flexcan/test/can_test.c

Chapter 8
Configuring the GPMI Controller

8.1 Overview

This chapter explains how to configure and use the general-purpose media interface (GPMI) controller. The GPMI controller is a flexible interface for up to eight NAND Flash devices. It is compatible with the ONFI 2.2 standard and Samsung's Toggle NAND protocol.

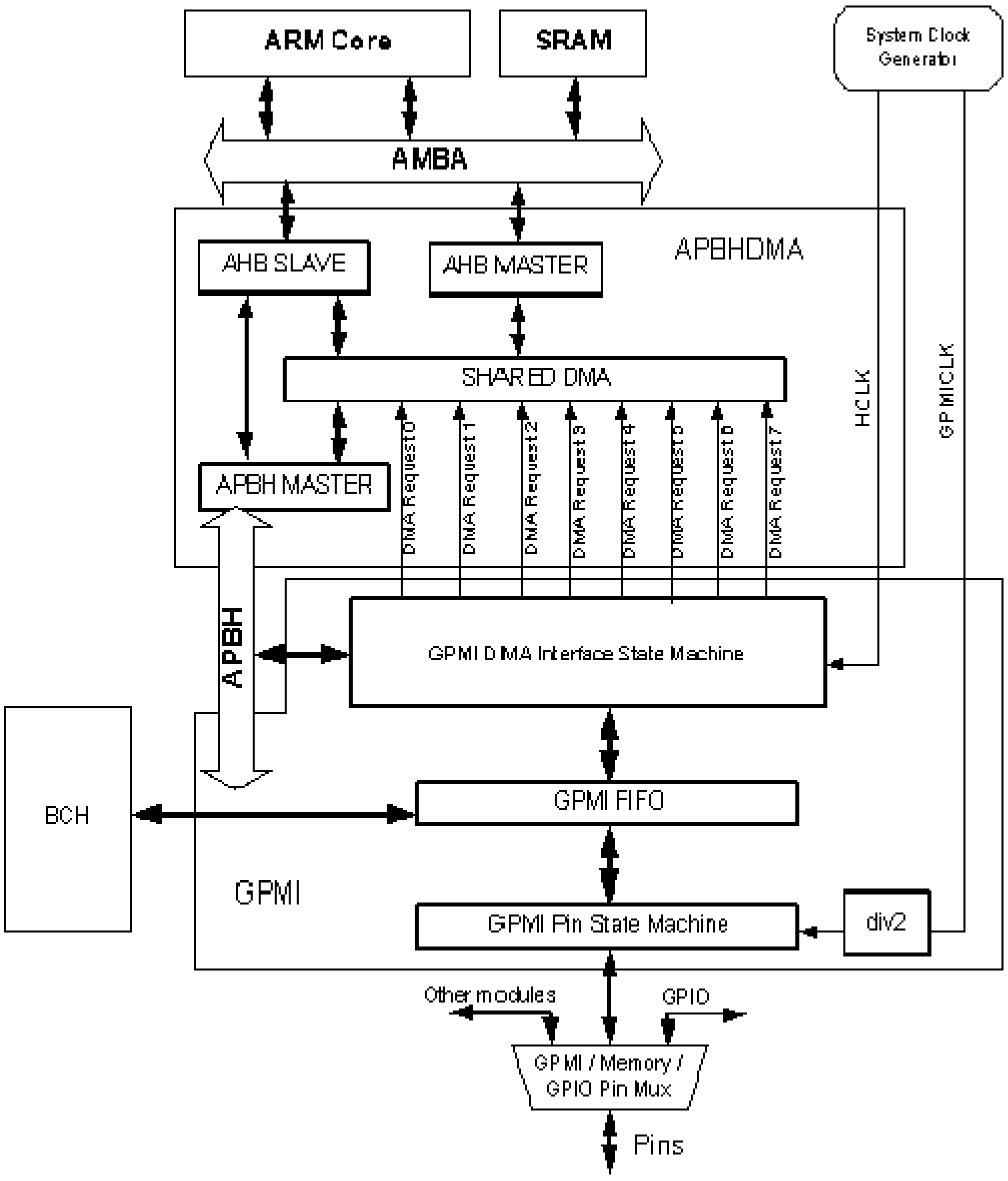


Figure 8-1. GPMI block diagram

The GPMI resides on the APBH. The GPMI also provides an interface to the BCH module to allow direct parity processing.

This chip has a single instance of the GPMI module, which is memory mapped to location:

- GPMI base address = 0011 2000h

8.2 Feature summary

The key features are:

- Individual chip select and ready/busy pins for up to eight NAND devices
- Individual state machine and DMA channel for each chip select
- Special command modes that work with DMA controller to perform all normal NAND functions without CPU intervention
- Configurable timing based on a dedicated clock allows optimal balance of high NAND performance and low system power

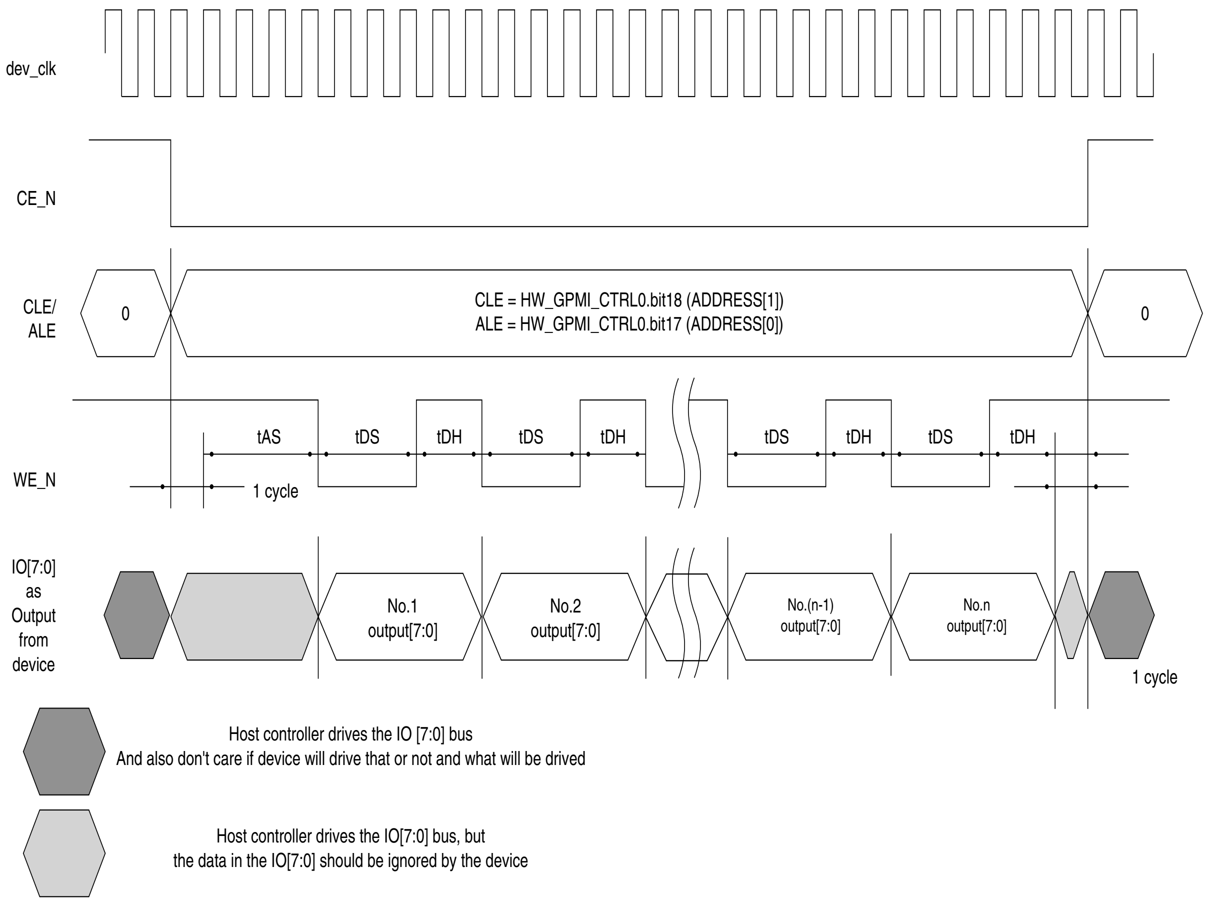
GPMI and DMA have been designed to handle complex multi-page operations without CPU intervention. The DMA uses a linked descriptor function with branching capability to automatically handle all of the operations needed to read/write multiple pages.

8.3 Modes of operation

Table 8-1. GPMI modes of operation

Mode	What it does
Data/Register Read/ Write	In this mode, the GPMI can be programmed to read or write multiple cycles to the NAND address, command, or data registers.
Wait for NAND ready	This mode can monitor the ready/busy signal of a single NAND flash and signal to the DMA when the device is ready. It also has a timeout counter and can indicate to the DMA that a timeout error has occurred. The DMAs can conditionally branch to a different descriptor in the case of an error.
Check status	This mode allows the GPMI to check NAND status against a reference. If an error is found, the GPMI can instruct the DMA to branch to an alternate descriptor, which attempts to fix the problem or asserts a CPU IRQ.

8.4 Basic NAND timing



Host controller drives the IO [7:0] bus
And also don't care if device will drive that or not and what will be drived

Host controller drives the IO[7:0] bus, but
the data in the IO[7:0] should be ignored by the device

- tAS is configurable by programming HW_GPMI_TIMING0 Address_Setup: in this example, Address_Setup = 4, tAS is equal to 4 dev_clk cycles.
- tDS is configurable by programming HW_GPMI_TIMING0 Data_Setup; in this example, Data_Setup = 3, tDS is equal to 3 dev_clk cycles
- tDH is configuarble by programming HW_GPMI_TIMING0 Data_Hold: in this example, Data_Hold = 2, tDH is equal to 2 dev_clk cycles
- tAS/tDS/tDH will extend, if the output data is not ready in device fifo.

For additional timing information, see the "Basic NAND timing" section of the GPMI chapter in your device reference manual.

8.5 Clocks

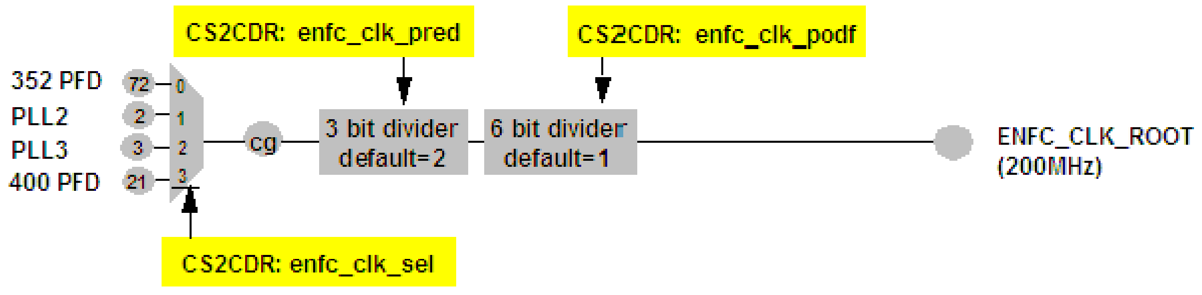


Figure 8-3. GPMI clock tree

The dedicated clock, GPMICLK(ENFC_CLK_ROOT), is used as a timing reference for NAND Flash I/O. Because various NANDs have different timing requirements, GPMICLK may need to be adjusted for each application. While the actual pin timings are limited by the NAND chips used, the GPMI can support data bus speeds of up to 200 MHz x 8 bits.

8.6 IOMUX pin mapping

The IOMUX configuration is board dependent. The IOMUX tool can be used to auto-generate an IOMUX configuration code for a particular board. The following table shows this chip's available mux options.

Table 8-2. GPMI IOMUX pin mapping options

Signal	GPMI		
	PAD	MUX	SION
ALE	NANDF_ALE	ALT0	0
CLE	NANDF_CLE	ALT0	0
Reset	NANDF_WP_B	ALT0	0
Ready/busy	NANDF_RB0	ALT0	0
CE0N	NANDF_CS0	ALT0	0
CE1N	NANDF_CS1	ALT0	0
CE2N	NANDF_CS2	ALT0	0

Table continues on the next page...

Table 8-2. GPMI IOMUX pin mapping options (continued)

Signal	GPMI		
	PAD	MUX	SION
CE3N	NANDF_CS3	ALT0	0
RDN	SD4_CMD	ALT1	0
WRN	SD4_CLK	ALT1	0
D0	NANDF_D0	ALT0	0
D1	NANDF_D1	ALT0	0
D2	NANDF_D2	ALT0	0
D3	NANDF_D3	ALT0	0
D4	NANDF_D4	ALT0	0
D5	NANDF_D5	ALT0	0
D6	NANDF_D6	ALT0	0
D7	NANDF_D7	ALT0	0
D8	SD4_DAT0	ALT0	0
D9	SD4_DAT1	ALT0	0
D10	SD4_DAT2	ALT0	0
D11	SD4_DAT3	ALT0	0
D12	SD4_DAT4	ALT0	0
D13	SD4_DAT5	ALT0	0
D14	SD4_DAT6	ALT0	0
D15	SD4_DAT7	ALT0	0

NOTE

Most of the GPMI signals have their own dedicated pins and therefore are not listed in the IOMUX pin mapping table. Refer to the device reference manual for the complete listing of pin mappings.

8.7 APBH DMA

APBH DMA is the only recommended way to transfer data between NAND flash and RAM.

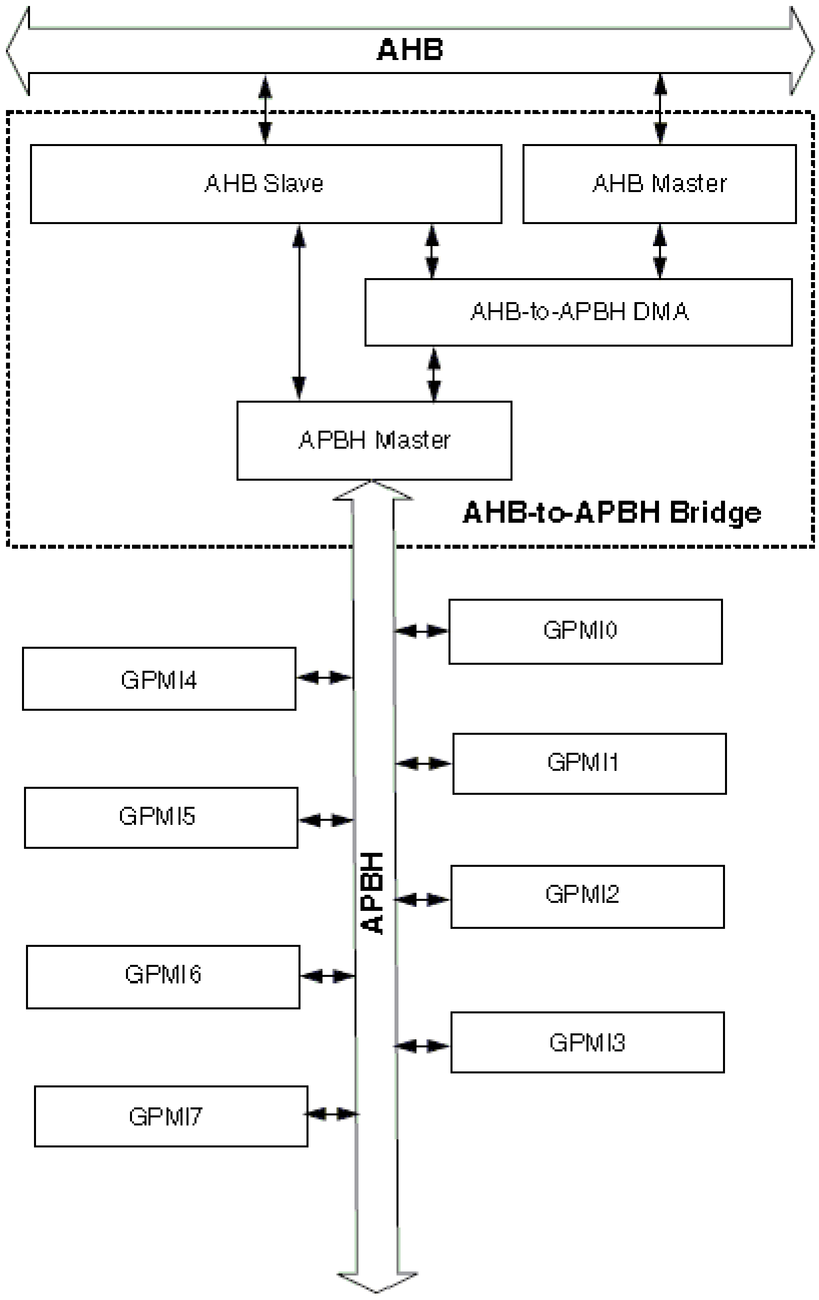


Figure 8-4. APBH DMA block diagram

The AHB-to-APBH bridge includes:

- The AHB-to-APB PIO bridge for a memory-mapped I/O to the APB devices
- A central DMA facility for devices on this bus
- A vectored interrupt controller for the ARM core

The following figure shows the structure for the channel command word. This single command structure specifies a number of operations to be performed by the DMA in support of a given device. Using this structure, the ARM platform can set up large units of work, chaining together many DMA channel command words and passing them off to the DMA. The ARM platform has no further concern for the device until the DMA completion interrupt occurs. The goal is to have enough intelligence in the DMA and the devices to keep the interrupt frequency from any device below 1 KHz (arrival intervals longer than 1 ms).

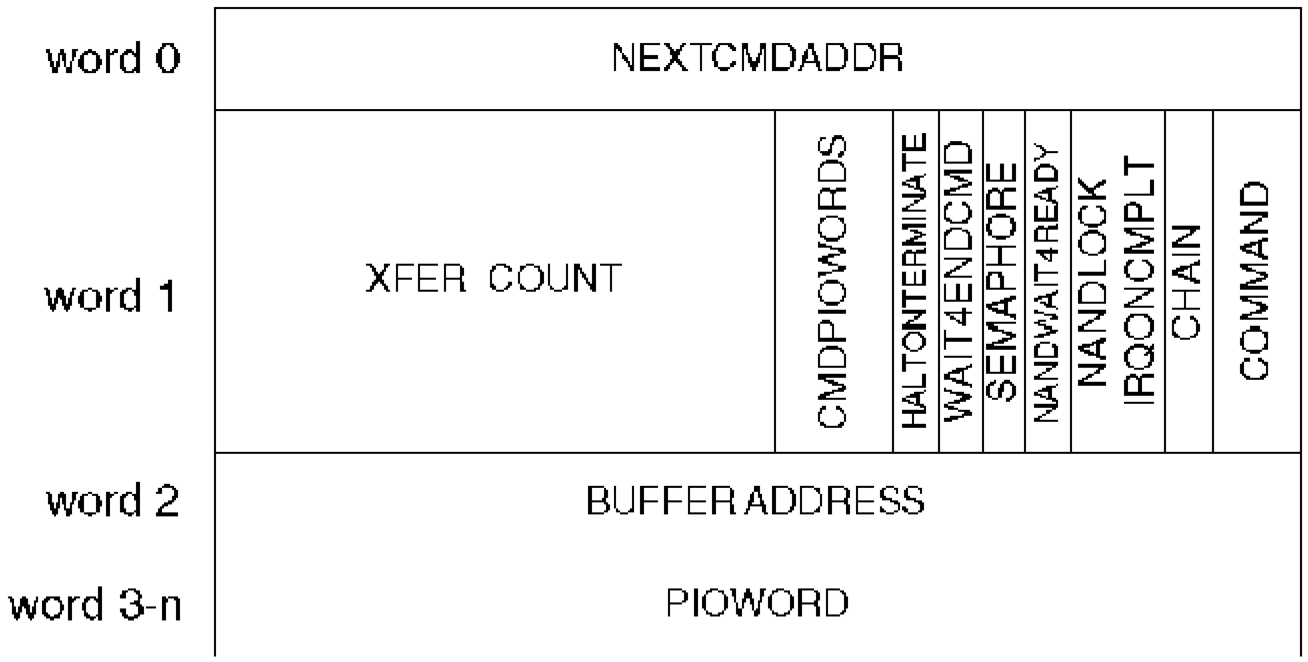


Figure 8-5. Channel-command word structure

8.8 NAND FLASH WRITE example code

The following example code illustrates the code for writing 4096 byte page data to NAND Flash with no error correction.

```
//-----------------------------------------------------------------------------
// generic DMA/GPMI/ECC descriptor struct, order sensitive!
//-----------------------------------------------------------------------------
typedef struct {
// DMA related fields
unsigned int dma_nxtcmdar;
unsigned int dma_cmd;
unsigned int dma_bar;
// GPMI related fields
unsigned int gpmi_ctrl0;
unsigned int gpmi_compare;
unsigned int gpmi_eccctrl;
unsigned int gpmi_ecccount;
unsigned int gpmi_data_ptr;
unsigned int gpmi_aux_ptr;
} GENERIC_DESCRIPTOR;
//-----------------------------------------------------------------------------
// allocate 10 descriptors for doing a NAND ECC Write
//-----------------------------------------------------------------------------
```

```
GENERIC_DESCRIPTOR write[10];
//----------------------------------------------------------------------------
// DMA descriptor pointer to handle error conditions from psense checks
//----------------------------------------------------------------------------
unsigned int * dma_error_handler;
//----------------------------------------------------------------------------
// 8 byte NAND command and address buffer
// any alignment is ok, it is read by the GPMI DMA
// byte 0 is write setup command
// bytes 1-5 is the NAND address
// byte 6 is write execute command
// byte 7 is status command
//----------------------------------------------------------------------------
unsigned char nand_cmd_addr_buffer[8];
//----------------------------------------------------------------------------
// 4096 byte payload buffer used for reads or writes
// needs to be word aligned
//----------------------------------------------------------------------------
unsigned int write_payload_buffer[(4096/4)];
//----------------------------------------------------------------------------
// 65 byte meta-data to be written to NAND
// needs to be word aligned
//----------------------------------------------------------------------------
unsigned int write_aux_buffer[65];
//----------------------------------------------------------------------------
// Descriptor 1: issue NAND write setup command (CLE/ALE)
//----------------------------------------------------------------------------
write[0].dma_nxtcmdar = &write[1]; // point to the next descriptor
write[0].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (1 + 5)| // 1 byte command, 5 byte address
BF_APBH_CHn_CMD_CMDWORDS (3) | // send 3 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_READ); // read data from DMA, write to NAND
write[0].dma_bar = &nand_cmd_addr_buffer; // byte 0 write setup, bytes 1 - 5 NAND address
// 3 words sent to the GPMI
write[0].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WRITE) | // write to the NAND
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_CLE) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (1) | // send command and address
BF_GPMI_CTRL0_XFER_COUNT (1 + 5); // 1 byte command, 5 byte address
write[0].gpmi_compare = NULL; // field not used but necessary to set eccctrl
write[0].gpmi_eccctrl = BV_FLD(GPMI_ECCCTRL, ENABLE_ECC, DISABLE); // disable the ECC block
//----------------------------------------------------------------------------
// Descriptor 2: write the data payload (DATA)
//----------------------------------------------------------------------------
write[1].dma_nxtcmdar = &write[2]; // point to the next descriptor
write[1].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (4096+128)| // page size + spare size
BF_APBH_CHn_CMD_CMDWORDS (4)| // send 4 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1)| // Wait to end
BF_APBH_CHn_CMD_SEMAPHORE (0)|
BF_APBH_CHn_CMD_NANDWAIT4READY (0)|
BF_APBH_CHn_CMD_NANDLOCK (1)| // maintain resource lock
BF_APBH_CHn_CMD_IRQONCMPLT (0)|
BF_APBH_CHn_CMD_CHAIN (1)| // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, READ); //
write[1].dma_bar = &write_payload_buffer; // pointer for the 4K byte data area
// 4 words sent to the GPMI
write[1].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WRITE) | // write to the NAND
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_DATA)|
BF_GPMI_CTRL0_ADDRESS_INCREMENT (0) |
BF_GPMI_CTRL0_XFER_COUNT (4096+128); //
```

```
// DMA transferred to GPMI via DMA (0)!
write[1].gpmi_compare = NULL; // field not used but necessary to set eccctrl
write[1].gpmi_eccctrl = BV_FLD(GPMI_ECCCTRL, ECC_CMD, ENCODE_8_BIT) | // specify t = 8
mode
BV_FLD(GPMI_ECCCTRL, ENABLE_ECC, DISABLE) | // enable ECC module
BF_GPMI_ECCCTRL_BUFFER_MASK (0x1FF); // write all 8 data blocks
// and 1 aux block
write[1].gpmi_ecccount = 0; // disable ecc
write[1].gpmi_data_pointer = (write_payload_pointer)&0xFFFFFFFC; // data buffer address
write[1].gpmi_aux_pointer = (write_aux_pointer)&0xFFFFFFFC; // metadata pointer
//----------------------------------------------------------------------------
// Descriptor 3: issue NAND write execute command (CLE)
//----------------------------------------------------------------------------
write[2].dma_nxtcmdar = &write[3]; // point to the next descriptor
write[2].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (1) | // 1 byte command
BF_APBH_CHn_CMD_CMDWORDS (3) | // send 3 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // maintain resource lock
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_READ); // read data from DMA, write to NAND
write[2].dma_bar = &nand_cmd_addr_buffer[6]; // point to byte 6, write execute command
// 3 words sent to the GPMI
write[2].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WRITE) | // write to the NAND
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_CLE) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (0) |
BF_GPMI_CTRL0_XFER_COUNT (1); // 1 byte command
write[2].gpmi_compare = NULL; // field not used but necessary to set eccctrl
write[2].gpmi_eccctrl = BV_FLD(GPMI_ECCCTRL, ENABLE_ECC, DISABLE); // disable the ECC  //
block
//----------------------------------------------------------------------------
// Descriptor 4: wait for ready (CLE)
//----------------------------------------------------------------------------
write[3].dma_nxtcmdar = &write[4]; // point to the next descriptor
write[3].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (1) | // send 1 word to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(1) | // wait for nand to be ready
BF_APBH_CHn_CMD_NANDLOCK (0) | // relinquish nand lock
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, NO_DMA_XFER); // no dma transfer
write[3].dma_bar = NULL; // field not used
// 1 word sent to the GPMI
write[3].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WAIT_FOR_READY) | // wait //for NAND
ready
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, DISABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_DATA) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (0) |
BF_GPMI_CTRL0_XFER_COUNT (0);
//----------------------------------------------------------------------------
// Descriptor 5: psense compare (time out check)
//----------------------------------------------------------------------------
write[4].dma_nxtcmdar = &write[5]; // point to the next descriptor
write[4].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (0) | // no words sent to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // do not wait to continue
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (0) |
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
```

```
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_SENSE); // perform a sense check
write[4].dma_bar = dma_error_handler; // if sense check fails, branch to error handler
//----------------------------------------------------------------------------
// Descriptor 6: issue NAND status command (CLE)
//----------------------------------------------------------------------------
write[5].dma_nxtcmdar = &write[6]; // point to the next descriptor
write[5].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (1) | // 1 byte command
BF_APBH_CHn_CMD_CMDWORDS (3) | // send 3 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_READ); // read data from DMA, write to NAND
write[5].dma_bar = &nand_cmd_addr_buffer[7]; // point to byte 7, status
command
write[5].gpmi_compare = NULL; // field not used but necessary to set
eccctrl
write[5].gpmi_eccctrl = BV_FLD(GPMI_ECCCTRL, ENABLE_ECC, DISABLE); // disable the ECC block
// 3 words sent to the GPMI
write[5].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WRITE) | // write to the NAND
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_CLE) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (0) |
BF_GPMI_CTRL0_XFER_COUNT (1); // 1 byte command
//----------------------------------------------------------------------------
// Descriptor 7: read status and compare (DATA)
//----------------------------------------------------------------------------
write[6].dma_nxtcmdar = &write[7]; // point to the next descriptor
write[6].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (0) | // send 2 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before
// continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // maintain resource lock
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_WRITE); // no dma transfer
write[6].dma_bar = NULL; // field not used
//----------------------------------------------------------------------------
// Descriptor 8: psense compare (time out check)
//----------------------------------------------------------------------------
write[7].dma_nxtcmdar = &write[8]; // point to the next descriptor
write[7].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (0) | // no words sent to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // do not wait to continue
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (0) | // relinquish nand lock
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_SENSE); // perform a sense check
write[7].dma_bar = &write[9]; // if sense check fails, branch to error handler
//----------------------------------------------------------------------------
// Descriptor 9: success handler
//----------------------------------------------------------------------------
write[8].dma_nxtcmdar = NULL; // not used since this is last descriptor
write[8].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (0) | // no words sent to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // do not wait to continue
BF_APBH_CHn_CMD_SEMAPHORE (1) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (0) |
BF_APBH_CHn_CMD_IRQONCMPLT (1) | // emit GPMI interrupt
BF_APBH_CHn_CMD_CHAIN (0) | // terminate DMA chain processing
BV_FLD(APBH_CHn_CMD, COMMAND, NO_DMA_XFER); // no dma transfer
```

```
write[8].dma_bar = (void*) SUCCESS;
//------------------------------------------------------------------------------
// Descriptor 10: failure handler
//------------------------------------------------------------------------------
write[9].dma_nxtcmdar = NULL; // not used since this is last descriptor
write[9].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (0) | // no words sent to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // do not wait to continue
BF_APBH_CHn_CMD_SEMAPHORE (1) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (0) |
BF_APBH_CHn_CMD_IRQONCMPLT (1) | // emit GPMI interrupt
BF_APBH_CHn_CMD_CHAIN (0) | // terminate DMA chain processing
BV_FLD(APBH_CHn_CMD, COMMAND, NO_DMA_XFER); // no dma transfer
write[9].dma_bar = (void *) FAILURE;
```

8.9 NAND FLASH READ example code

The following example code illustrates the code for reading 4096 bytes of page data from NAND Flash to RAM address with no error correction.

```
//------------------------------------------------------------------------------
// Descriptor 1: issue NAND read setup command (CLE/ALE)
//------------------------------------------------------------------------------
read[0].dma_nxtcmdar = &read[1]; // point to the next descriptor
read[0].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (1 + 5) | // 1 byte command, 5 byte //address
BF_APBH_CHn_CMD_CMDWORDS (3) | // send 3 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_READ); // read data from DMA, write to NAND
read[0].dma_bar = &nand_cmd_addr_buffer; // byte 0 read setup, bytes 1 - 5 NAND address
// 3 words sent to the GPMI
read[0].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WRITE) | // write to the NAND
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_CLE) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (1) | // send command and address
BF_GPMI_CTRL0_XFER_COUNT (1 + 5); // 1 byte command, 5 byte address
read[0].gpmi_compare = NULL; // field not used but necessary to set eccctrl
read[0].gpmi_eccctrl = BV_FLD(GPMI_ECCCTRL, ENABLE_ECC, DISABLE); // disable the ECC block
//------------------------------------------------------------------------------
// Descriptor 2: issue NAND read execute command (CLE)
//------------------------------------------------------------------------------
read[1].dma_nxtcmdar = &read[2]; // point to the next descriptor
read[1].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (1) | // 1 byte read command
BF_APBH_CHn_CMD_CMDWORDS (1) | // send 1 word to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_READ); // read data from DMA, write to NAND
read[1].dma_bar = &nand_cmd_addr_buffer[6]; // point to byte 6, read execute command
// 1 word sent to the GPMI
read[1].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WRITE) | // write to the NAND
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, DISABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
```

```
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_CLE) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (0) |
BF_GPMI_CTRL0_XFER_COUNT (1); // 1 byte command
//----------------------------------------------------------------------------
// Descriptor 3: wait for ready (DATA)
//----------------------------------------------------------------------------
read[2].dma_nxtcmdar = &read[3]; // point to the next descriptor
read[2].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (1) | // send 1 word to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(1) | // wait for nand to be ready
BF_APBH_CHn_CMD_NANDLOCK (0) | // relinquish nand lock
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, NO_DMA_XFER); // no dma transfer
read[2].dma_bar = NULL; // field not used 1 word sent to the GPMI
read[2].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WAIT_FOR_READY) |
// wait for NAND ready
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, DISABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_DATA) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (0) |
BF_GPMI_CTRL0_XFER_COUNT (0);
//----------------------------------------------------------------------------
// Descriptor 4: psense compare (time out check)
//----------------------------------------------------------------------------
read[3].dma_nxtcmdar = &read[4]; // point to the next descriptor
read[3].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (0) | // no words sent to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // do not wait to continue
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (0) |
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_SENSE); // perform a sense check
read[3].dma_bar = dma_error_handler; // if sense check fails, branch to error handler
//----------------------------------------------------------------------------
// Descriptor 5: read 4K page from Nand flash
//----------------------------------------------------------------------------
read[4].dma_nxtcmdar = &read[5]; // point to the next descriptor
read[4].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (4096+128) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (6) | // send 6 words to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) | // ECC block generates BCH interrupt on completion
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_WRITE); // DMA write,
// ECC block handles transfer
read[4].dma_bar = NULL; // field not used 6 words sent to the GPMI
read[4].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, READ) | // read from the NAND
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, DISABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_DATA) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (0) |
BF_GPMI_CTRL0_XFER_COUNT (4096+218); // eight 512 byte data blocks
// metadata, and parity
read[4].gpmi_compare = NULL; // field not used but necessary to set eccctrl
// Disable ECC
read[4].gpmi_eccctrl = 0;// disable ECC module
read[4].gpmi_ecccount = 0; // specify number of bytes
// read from NAND
read[4].gpmi_data_ptr = (&read_payload_buffer)&0xFFFFFFFC; // pointer for the 4K byte data
area
read[4].gpmi_aux_ptr = (&read_aux_buffer)&0xFFFFFFFC; // pointer for the 65 byte aux area +
parity and syndrome
//----------------------------------------------------------------------------
// Descriptor 6: wait for done
//----------------------------------------------------------------------------
read[5].dma_nxtcmdar = &read[6]; // point to the next descriptor
read[5].dma_bar =&read[7];//if error, jump to error handler
read[5].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (3) | // send 3 words to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(1) | // wait for nand to be ready
BF_APBH_CHn_CMD_NANDLOCK (1) | // need nand lock to be thread safe while turn-off BCH
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, NO_DMA_XFER); // no dma transfer
read[5].dma_bar = NULL; // field not used 3 words sent to the GPMI
read[5].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WAIT_READY) |
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, DISABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_DATA) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (0) |
BF_GPMI_CTRL0_XFER_COUNT (0);
read[5].gpmi_compare = NULL; // field not used but necessary to set eccctrl
read[5].gpmi_eccctrl = BV_FLD(GPMI_ECCCTRL, ENABLE_ECC, DISABLE); // disable the ECC block
//----------------------------------------------------------------------------
// Descriptor 7: success handler
//----------------------------------------------------------------------------
read[6].dma_nxtcmdar = NULL; // not used since this is last descriptor
read[6].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (0) | // no words sent to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (1) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (0) | // relinquish nand lock
BF_APBH_CHn_CMD_IRQONCMPLT (1) | //
BF_APBH_CHn_CMD_CHAIN (0) | // terminate DMA chain processing
BV_FLD(APBH_CHn_CMD, COMMAND, NO_DMA_XFER); // no dma transfer
read[6].dma_bar =(void *)SUCCESS;
//----------------------------------------------------------------------------
// Descriptor 8: FAILURE handler
//----------------------------------------------------------------------------
read[6].dma_nxtcmdar = NULL; // not used since this is last descriptor
read[6].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // no dma transfer
BF_APBH_CHn_CMD_CMDWORDS (0) | // no words sent to GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (1) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (0) | // relinquish nand lock
BF_APBH_CHn_CMD_IRQONCMPLT (1) | //
BF_APBH_CHn_CMD_CHAIN (0) | // terminate DMA chain processing
BV_FLD(APBH_CHn_CMD, COMMAND, NO_DMA_XFER); // no dma transfer
read[6].dma_bar =(void *)FAILURE;
```

8.10 NAND FLASH ERASE example code

The following code illustrates the flow of Nand Erase Block command.

```
//----------------------------------------------------------------------------
// Descriptor 1: send ERASE setup command and 3 row address cycles
//----------------------------------------------------------------------------
erase[0].dma_nxtcmdar = &erase[1]; // point to the next descriptor
erase[0].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (1 + 3) | // 1 byte command, 3 byte //address
BF_APBH_CHn_CMD_CMDWORDS (1) | // send 3 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
```

```
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_READ); // read data from DMA, write to NAND
read[0].dma_bar = 0x60; // NAND erase command
// 1 words sent to the GPMI
read[0].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WRITE) |
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_CLE) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (1) | // send command and address
BF_GPMI_CTRL0_XFER_COUNT (1 + 3); // 1 byte command, 3 byte address
//----------------------------------------------------------------------------
// Descriptor 2: Fill ERASE confirm command
//----------------------------------------------------------------------------
erase[1].dma_nxtcmdar = &erase[2]; // point to the next descriptor
erase[1].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (1 ) | // 1 byte command
BF_APBH_CHn_CMD_CMDWORDS (1) | // send 1 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_READ); // read data from DMA, write to NAND
read[1].dma_bar = 0xD0; // NAND erase confirm command
// 1 words sent to the GPMI
read[1].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WRITE) |
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_CLE) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (1) | // send command and address
BF_GPMI_CTRL0_XFER_COUNT (1 ); // 1 byte command
//----------------------------------------------------------------------------
// Descriptor 3: Check NAND Status start
//----------------------------------------------------------------------------
erase[2].dma_nxtcmdar = &erase[3]; // point to the next descriptor
erase[2].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // 0 byte command
BF_APBH_CHn_CMD_CMDWORDS (1) | // send 1 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(1) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, NO_TRANSFER);
read[2].dma_bar = NULL
// 1 words sent to the GPMI
read[2].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, CMD_WAIT_READY) |
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_DATA) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (1) | // send command and address
BF_GPMI_CTRL0_XFER_COUNT (0); // 0 byte command
//----------------------------------------------------------------------------
// Descriptor 4: Check status conditional branch
//----------------------------------------------------------------------------
erase[3].dma_nxtcmdar = &erase[4]; // point to the next descriptor
erase[3].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // 0 byte command
BF_APBH_CHn_CMD_CMDWORDS (0) | // send 1 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_SENSE);
read[3].dma_bar = &erase[7]; //if fail, jump to error handler
//----------------------------------------------------------------------------
// Descriptor 5: send read status command - 0x70
//----------------------------------------------------------------------------
erase[4].dma_nxtcmdar = &erase[5]; // point to the next descriptor
erase[4].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (1) | // 1 byte command
BF_APBH_CHn_CMD_CMDWORDS (1) | // send 1 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_READ);
read[4].dma_bar = &erase[7]; //if fail, jump to error handler
// 1 words sent to the GPMI
read[4].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, WRITE) |
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_CLE) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (1) | // send command and address
BF_GPMI_CTRL0_XFER_COUNT (1); // 1 byte command
//----------------------------------------------------------------------------
// Descriptor 6: read status value
//----------------------------------------------------------------------------
erase[5].dma_nxtcmdar = &erase[6]; // point to the next descriptor
erase[5].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (1) | // 1 byte command
BF_APBH_CHn_CMD_CMDWORDS (1) | // send 1 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (1) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (1) | // follow chain to next command
BV_FLD(APBH_CHn_CMD, COMMAND, DMA_WRITE);
read[5].dma_bar = &erase[7]; //if fail, jump to error handler
// 1 words sent to the GPMI
read[5].gpmi_ctrl0 = BV_FLD(GPMI_CTRL0, COMMAND_MODE, READ) |
BV_FLD(GPMI_CTRL0, WORD_LENGTH, 8_BIT) |
BV_FLD(GPMI_CTRL0, LOCK_CS, ENABLED) |
BF_GPMI_CTRL0_CS (0) | // must correspond to NAND CS used
BV_FLD(GPMI_CTRL0, ADDRESS, NAND_DATA) |
BF_GPMI_CTRL0_ADDRESS_INCREMENT (1) | // send command and address
BF_GPMI_CTRL0_XFER_COUNT (1); // 1 byte command
//----------------------------------------------------------------------------
// Descriptor 7: success handler
//----------------------------------------------------------------------------
erase[6].dma_nxtcmdar = NULL; // point to the next descriptor
erase[6].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // 0 byte command
BF_APBH_CHn_CMD_CMDWORDS (0) | // send 1 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (0) | // the last command, no need to chain
BV_FLD(APBH_CHn_CMD, COMMAND, NO_TRANSFER);
read[5].dma_bar = (void *)SUCCESS;
//----------------------------------------------------------------------------
// Descriptor 8: failer handler
//----------------------------------------------------------------------------
erase[6].dma_nxtcmdar = NULL; // point to the next descriptor
erase[6].dma_cmd = BF_APBH_CHn_CMD_XFER_COUNT (0) | // 0 byte command
BF_APBH_CHn_CMD_CMDWORDS (0) | // send 1 words to the GPMI
BF_APBH_CHn_CMD_WAIT4ENDCMD (0) | // wait for command to finish before continuing
BF_APBH_CHn_CMD_SEMAPHORE (0) |
BF_APBH_CHn_CMD_NANDWAIT4READY(0) |
BF_APBH_CHn_CMD_NANDLOCK (1) | // prevent other DMA channels from taking over
BF_APBH_CHn_CMD_IRQONCMPLT (0) |
BF_APBH_CHn_CMD_CHAIN (0) | // the last command, no need to chain
BV_FLD(APBH_CHn_CMD, COMMAND, NO_TRANSFER);
read[5].dma_bar = (void *)FAILURE;
```

Chapter 9
Configuring the GPT Driver

9.1 Overview

This chapter explains how to configure the GPT driver. GPT is a 32-bit up-counter that uses four clock sources, one of which is external. The timer counter value can be captured in a register using an event on an external pin. The capture trigger can be programmed to be a rising or/and falling edge. The GPT can also generate an event on the CMPOUT*n* pins and an interrupt when the timer reaches a programmed value.

This chapter uses i.MX 6Dual/6Quad ARD board schematics for pin assignments. For other board types, please refer to respective schematics.

There is one instance of GPT in the i.MX 6Dual/6Quad processor, and it is located in the memory map at the GPT base address, 0209 8000h.

9.2 Features summary

This low-level driver supports:

- Usage of four different clock sources for the counter
- Restart and free-run modes for counter operations
- Two input capture channels with a programmable trigger edge
- Three output compare channels with a programmable output mode; a forced compare feature is also available
- Ability to be programmed to be active in low power and debug modes
- Interrupt generation at capture, compare, and rollover events

9.3 Modes of operation

The following table explains the GPT modes of operation:

Table 9-1. Modes of operation

Mode	What it does
Restart mode	The GPT counter starts to count from 0h. When it reaches the compare value, it generates an event and restarts to the initial value. Any write access to the Compare register of Channel 1 will reset the GPT counter. This is done to avoid possibly missing a compare event when compare value is changed from a higher value to lower value while counting is proceeding. For the other two compare channels, when the compare event occurs the counter is not reset.
Free-run mode	The GPT counter starts to count from 0h. When it reaches the compare value, it generates an event and continues to run. Once it reaches FFFF FFFFh , it rolls over to zero.

9.4 Events

9.4.1 Output compare event

The GPT can change the state of an output signal (CMPOUTx - x = ,2,3) based on a programmable compare value. The behavior of that signal is configurable in the driver and can be set to:

- `OUTPUT_CMP_DISABLE` = output disconnected from the external signal CMPOUTx.
- `OUTPUT_CMP_TOGGLE` = toggle the output.
- `OUTPUT_CMP_CLEAR` = set the output to a low level.
- `OUTPUT_CMP_SET` = set the output to a high level.
- `OUTPUT_CMP_LOWPULSE` = low pulse generated on the output.

Use the following functions to trigger the output compare event:

- `gpt_get_compare_event()`
- `gpt_set_compare_event()`

9.4.2 Input capture event

The GPT can capture the counter's value when an external input event occur on signals (CAPINx - x = 1,2). The behavior of that signal is configurable in the driver and can be set to:

- `INPUT_CAP_DISABLE` = input capture disabled.
- `INPUT_CAP_RISING_EDGE` = input capture on rising edge.
- `INPUT_CAP_FALLING_EDGE` = input capture on falling edge.
- `INPUT_CAP_BOTH_EDGE` = input capture on both edges

The following function can be used to capture input events:

```
gpt_get_capture_event()
```

9.4.3 Rollover event

The GPT generates an event when the counter rolls over from FFFF FFFFh to 0h.

The following function checks if this event occurred.

```
gpt_get_rollover_event()
```

9.5 Clocks

The GPT receives four clocks: three from the CCM and one external clock through CLKIN I/O.

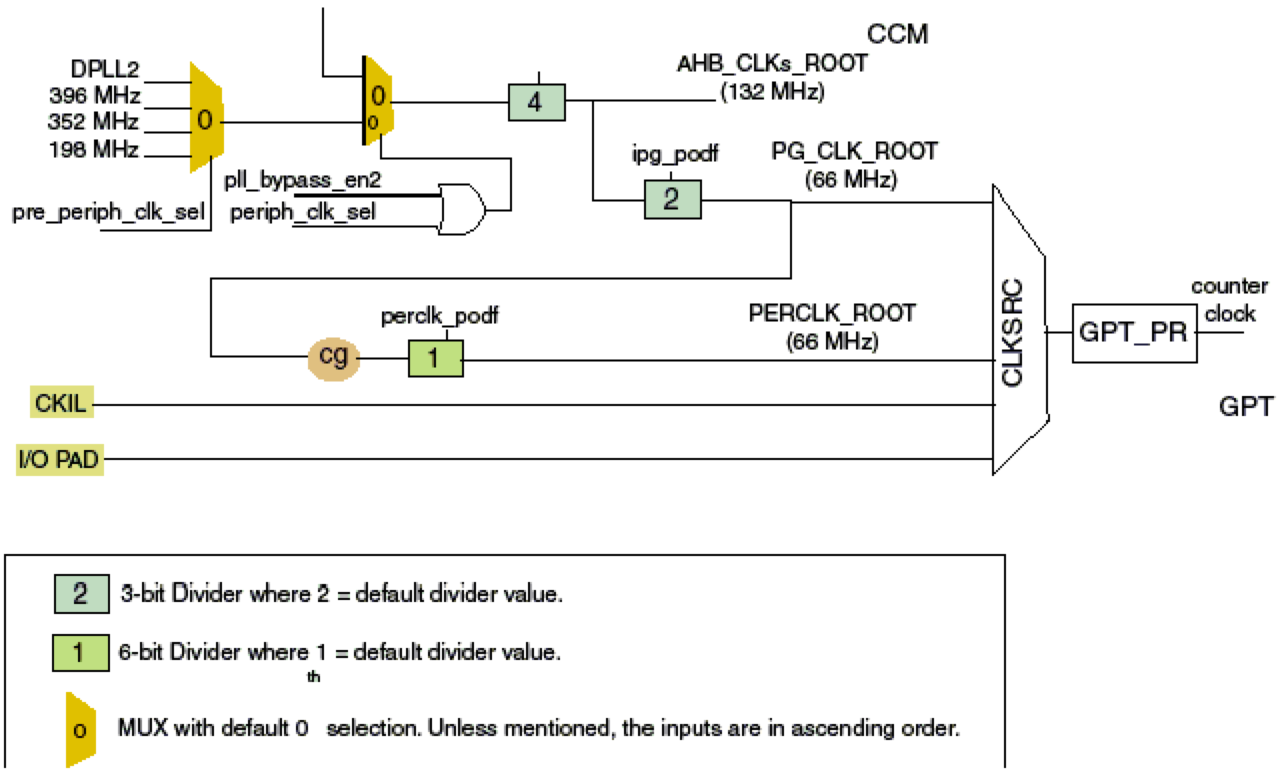


Figure 9-1. Reference clocks

The following table explains the GPT reference clocks:

Table 9-2. Reference clocks

Clock	Name	What it does
Low-frequency clock	CKIL	This 32768 Hz low reference clock is intended to be ON in Low Power mode when ipg_clk is off
High-frequency clock	PERCLK_ROOT	GPT operates on PERCLK in normal power mode when ipg_clk is off.
Peripheral clock	IPG_CLK_ROOT	In low power modes, if the GPT is disabled, then ipg_clk can be switched off.
External clock	CLKIN	External clock source synchronized to ipg_clk inside GPT. it's frequency should be < 1/4 (ipg_clk).

Because the frequency of PLL2 and various dividers is system dependent, the user may need to adjust the driver's frequency. To do this change the freq member of the hw_module structure defined into ./src/include/io.h .

For example, take the following non-default divider values:

- PLL2 is set to output 396 MHz
- ahb_podf divides by 3
- ipg_podf divides by 2

In this example, IPG_CLK = 132 MHz and PERCLK = 66 MHz.

The driver handles the clock gating on the source clock.

9.6 IOMUX pin mapping

The GPT can change the state of the compare outputs (CMPOUT, CMPOUT2, CMPOUT3) on a compare event. The IOMUX should route the signals to the appropriate pins. The IOMUX configuration is board dependent and can be handled with the IOMUX tool.

Table 9-3. Pin assignments for i.MX 6Dual/6Quad

Signal	IOMUXC setting for GPT		
	PAD	MUX	SION
CLKIN	SD1_CLK	ALT3	1
CAPIN1	SD1_DAT0	ALT3	1
CAPIN2	SD1_DAT1	ALT3	1
CMPOUT1	SD1_CMD	ALT3	-
CMPOUT2	SD1_DAT2	ALT2	-
CMPOUT3	SD1_DAT3	ALT2	-

9.7 Resets and interrupts

The driver resets the module during the initialization by setting GPT_CR[SWR] in the function **gpt_init**().

The external application is responsible for creating the interrupt subroutine. The address of this routine is passed through the structure hw_module defined in ./src/include/io.h. It is initialized by the application and used by the driver for various configurations.

All interrupt sources are listed in the "Interrupts and DMA Events" chapter of the device reference manual. In the SDK, the list is provided in ./src/include/mx61/ soc_memory_map.h.

9.8 Initializing the GPT driver

Before using the GPT timer in a system, prepare a structure that provides the essential system parameters to the driver. This is done through the hw_module structure defined into ./src/include/io.h.

Example:

```
struct hw_module g_test_timer = {
    "GPT used for test",
    GPT_BASE_ADDR,
    66000000,
    IMX_INT_GPT,
    &default_interrupt_routine,
};
```

The address of this structure is used by most functions listed below.

```
/*!
 * Initialize the GPT timer.
 *
 * @param   port - pointer to the GPT module structure.
 * @param   clock_src - source clock of the counter: CLKSRC_OFF, CLKSRC_IPG_CLK,
 *                      CLKSRC_PER_CLK, CLKSRC_CKIL, CLKSRC_CLKIN.
 * @param   prescaler - prescaler of the source clock from 1 to 4096.
 * @param   counter_mode - counter mode: FREE_RUN_MODE or RESTART_MODE.
 * @param   low_power_mode - low power during which the timer is enabled:
 *                           WAIT_MODE_EN and/or STOP_MODE_EN.
 */
void gpt_init(struct hw_module *port, uint32_t clock_src, uint32_t prescaler,
              uint32_t counter_mode, uint32_t low_power_mode)
/*!
 * Setup GPT interrupt. It enables or disables the related HW module
 * interrupt, and attached the related sub-routine into the vector table.
 *
 * @param   port - pointer to the GPT module structure.
 */
void gpt_setup_interrupt(struct hw_module *port, uint8_t state)

/*!
```

```
 * Enable the GPT module. Used typically when the gpt_init is done, and
 * other interrupt related settings are ready.
 *
 * @param   port - pointer to the GPT module structure.
 * @param   irq_mode - interrupt mode: list of enabled IRQ such GPTSR_ROVIE,
 *                     or (GPTSR_IF1IE | GPTSR_OF3IE), ... or POLLING_MODE.
 */
void gpt_counter_enable(struct hw_module *port, uint32_t irq_mode)

/*!
 * Disable the counter. It saves energy when not used.
 *
 * @param   port - pointer to the GPT module structure.
 */
void gpt_counter_disable(struct hw_module *port)

/*!
 * Get a compare event flag and clear it if set.
 * This function can typically be used for polling method.
 *
 * @param   port - pointer to the GPT module structure.
 * @param   flag - checked compare event flag such GPTSR_OF1, GPTSR_OF2, GPTSR_OF3.
 * @return  the value of the compare event flag.
 */
uint32_t gpt_get_compare_event(struct hw_module *port, uint8_t flag)

/*!
 * Set a compare event by programming the compare register and
 * compare output mode.
 *
 * @param   port - pointer to the GPT module structure.
 * @param   cmp_output - compare output: CMP_OUTPUT1, CMP_OUTPU2, CMP_OUTPUT3.
 * @param   cmp_output_mode - compare output mode: OUTPUT_CMP_DISABLE, OUTPUT_CMP_TOGGLE,
 *                            OUTPUT_CMP_CLEAR, OUTPUT_CMP_SET, OUTPUT_CMP_LOWPULSE.
 * @param   cmp_value - compare value for the compare register.
 */
void gpt_set_compare_event(struct hw_module *port, uint8_t cmp_output,
                           uint8_t cmp_output_mode, uint32_t cmp_value)
/*!
 * Set the input capture mode.
 *
 * @param   port - pointer to the GPT module structure.
 * @param   cap_input - capture input: CAP_INPUT1, CAP_INPUT2.
 * @param   cap_input_mode - capture input mode: INPUT_CAP_DISABLE, INPUT_CAP_BOTH_EDGE,
 *                           INPUT_CAP_FALLING_EDGE, INPUT_CAP_RISING_EDGE.
 */
void gpt_set_capture_event(struct hw_module *port, uint8_t cap_input,
                           uint8_t cap_input_mode)
/*!
 * Get a captured value when an event occured, and clear the flag if set.
 *
 * @param   port - pointer to the GPT module structure.
 * @param   flag - checked capture event flag such GPTSR_IF1, GPTSR_IF2.
 * @param   capture_val - the capture register value is returned there if the event is true.
 * @return  the value of the capture event flag.
 */
uint32_t gpt_get_capture_event(struct hw_module *port, uint8_t flag,
                               uint32_t * capture_val)
/*!
 * Get rollover event flag and clear it if set.
 * This function can typically be used for polling method, but
 * is also used to clear the status compare flag in IRQ mode.
 * @param   port - pointer to the GPT module structure.
 * @return  the value of the rollover event flag.
 */
uint32_t gpt_get_rollover_event(struct hw_module *port)
```

9.9 Testing the GPT driver

GPT can run the following tests:

- Output compare test
- Input capture test

9.9.1 Output compare test

The test enables the three compare channels. A first event occurs after 1s; the second occurs after 2s; and the third after 3s. The last event is generated by the compare channel 1, which is the only compare channel that can restart the counter to 0h after an event. This restarts for a programmed number of seconds.

Output compare I/Os are not enabled in this test, although enabling them can be done by configuring the IOMUX settings to enable the feature.

9.9.2 Input compare test

This test enables an input capture. An I/O is used to monitor an event that stores the counter value into a GPT input capture register when it occurs. The test displays the amount of time that elapsed since the test started and the moment the capture finished. It uses the rollover interrupt event, too, because if the counter is used for a sufficient time, it rolls over. That information is requested to calculate the exact number of seconds.

9.9.3 Running the tests

To run the GPT tests, SDK uses the following command:

```
./tools/build_sdk -target mx61 -board ard -board_rev 1 -test gpt
```

This command generates the following ELF and binary files:

- `./output/mx61/bin/mx61ard-gpt-sdk.elf`
- `./output/mx61/bin/mx61ard-gpt-sdk.bin`

9.10 Source code structure

Table 9-4. Source code file locations

Description	Location
Low-level driver source	./src/sdk/timer/drv/imx_timer/gpt.c
Low-level driver header	./src/include/gpt.h
Unit tests	./src/sdk/timer/test/gpt_test.c ./src/sdk/timer/test/gpt_test.h

Chapter 10
Configuring the HDMI TX Module

10.1 Overview

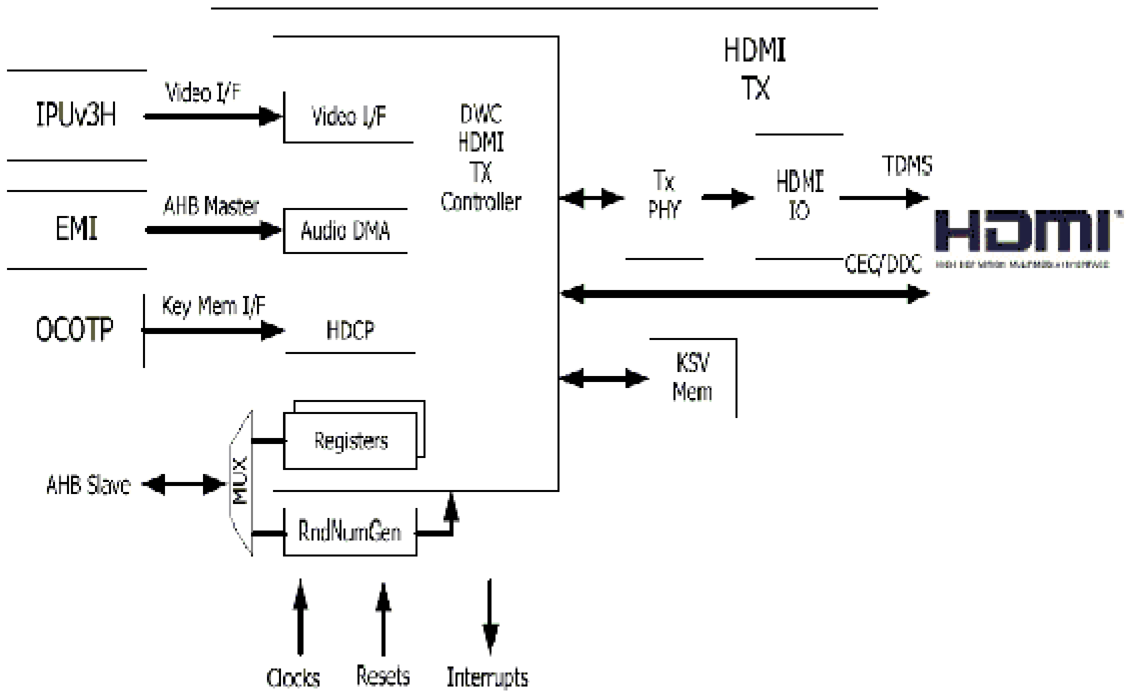


Figure 10-1. HDMI TX block diagram

This chapter explains how to configure and use the high-definition multimedia interface transmitter (HDMI TX) module. The HDMI TX module is an audio/video interface for transmitting uncompressed digital video data and uncompressed or compressed digital audio data. The HDMI TX module consists of two main parts: the HDMI TX controller and the HDMI TX PHY.

This chip uses a single instance of the HDMI TX module, which is memory mapped to the following location:

- HDMI base address = 0012 0000h

10.2 Features summary

The key features are:

- Compliant with HDMI v1.4a (DVI 1.0), HDCP 1.4
- Consumer Electronic Control (CEC) supported
- Video resolution up to 1080p at 120 Hz HDTV display
- TMDS core frequency from 25 MHz up to 340 MHz
- Monitor Detection: Hot plug/unplug detection

10.3 External Signal Descriptions

This table summarizes the HDMI TX available external signals.

Table 10-1. HDMI TX signal overview

Signal name	I/O	Description
HDMI_VP	Power	1.1 V analog power supply
HDMI_VPH	Power	2.5 V analog power supply
HDMI_REF	I/O	Reference resistor connection
HDMI_HPD	I	Hot plug detect
HDMI_DDCCEC	O	Ground Reference for HDMI_HPD signal
HDMI_CLK(M/P)	O	Minus/plus differential line clock output
HDMI_D[2:0](M/P)	O	Minus/plus differential lines for data channels (0,1, and 2)
HDMI_CEC_LINE	I/O	CEC data bus
HDMI_DDC_SDA	I/O	HDMI I2C data bus
HDMI_DDC_SCL	I/O	HDMI I2C clock

10.4 Clocks

HDMI TX's main clock source module is the pixel clock output from the image processing unit (IPU). Using the pixel clock, the HDMI PHY provides the PLL/MPLL that synthesizes the high-speed HDMI serial bit clock. The serial bit clock can be configured in the PLL/MPLL registers in the HDMI PHY.

HDMI TX also uses the HDMI internal register configuration clock, which is referred to as isfrclk in the HDMI_TX reference manual chapter. This clock can be gated on/off from the clock control module (CCM) using the CCM clock gate register

(CCM_CCGR2). The only available clock source for the isfrclk clock is VIDEO_27M_CLK_ROOT (see the CCM chapter clock tree diagram). This is the clock that generates isfrclk. The following figure shows the VIDEO_27M_CLK_ROOT clock tree.

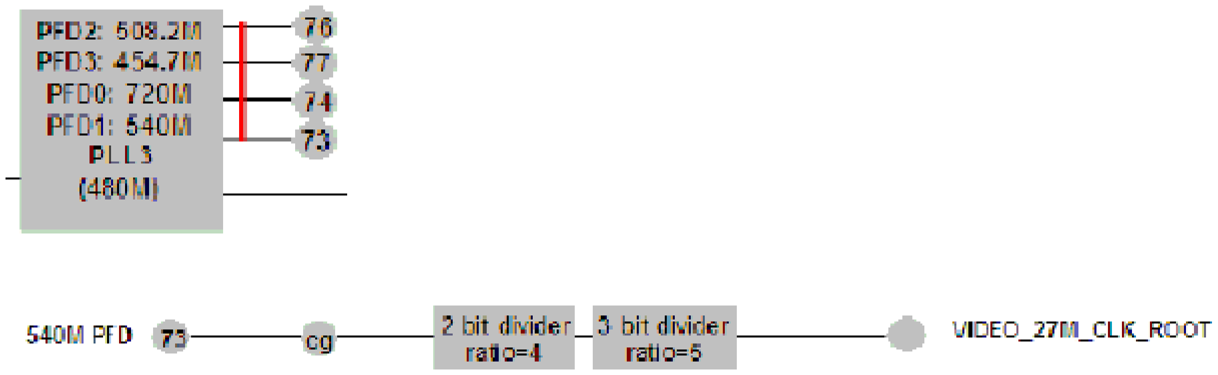


Figure 10-2. VIDEO_27M_CLK_ROOT clock tree

Similarly, the CCM_CCGR2 register also has a clock gate bit that is used to gate on/off the hdmi_tx_iahbclk signal. This clock's only available clock source is from the ahb_clk_root in the CCM, and it is needed for the AMBA AHB bus interface of the HDMITX module.

Table 10-2. HDMI TX clocks

Clock	Description
ipixelclk	Pixel clock input from the selected IPU display interface output which gets input to the HDMI_TX module.
isfrclk	Derived from video_27M_clk_root clock from CCM clock tree.
ihclk	Derived from ahb_clk from the CCM clock tree.
iahbclk	Derived from ahb_clk from the CCM clock tree.
icecclk	Derived from external 32.768 KHz reference clock.

NOTE

Both VIDEO_27M_CLK_ROOT and hdmi_tx_iahbclk need to be gated on (enabled) for HDMI TX to operate correctly.

10.5 IOMUX pin mapping

The IOMUX configuration is board dependent and can be handled by the IOMUX tool. The following table shows this chip's available mux options for the HDMI TX module signals.

Table 10-3. HDMI_TX IOMUX pin mapping

Signals	Option 1		Option 2	
	PAD	MUX	PAD	MUX
Module Instance: HDMI TX				
CEC_LINE	EIM_A25	ALT6	KEY_ROW2	ALT6
DDC_SCL	EIM_EB2	ALT4	KEY_COL3	ALT2
DDC_SDA	EIM_D16	ALT4	KEY_ROW3	ALT2

: HDMI TX dedicated pins

NOTE

Most of the HDMI TX signals have their own dedicated pins and are not listed in the IOMUX pin mapping table.

10.6 Setting up the HDMI video input

The video input source to the HDMI TX module can be any output stream from the IPU module. Therefore, there are four possible inputs:

- IPU1 display interface 0 (IPU1-DI0)
- IPU1 display interface 1 (IPU1-DI1)
- IPU2 display interface 0 (IPU2-DI0)
- IPU2 display interface 1 (IPU2-DI1)

To obtain video output from the HDMI TX module, the IPU input to HDMI TX must be properly configured. The video input source must be set up for RGB4:4:4, YCbCr4:2:2, or YCbCr4:4:4; therefore, the IPU output must be configured for one of those formats.

To configure the input to the HDMI TX module, use the IOMUXC_GPR3 register. Bits 3-2 (in the HDMI_MUX_CTL bit field) control the mux that selects which of the available IPU display interface outputs is used. The possible settings are:

- 00b-IPU1-DI0
- 01b-IPU1-DI1
- 10b-IPU2-DI0
- 11b-IPU2-DI1

The following figure shows the available mux configurations.

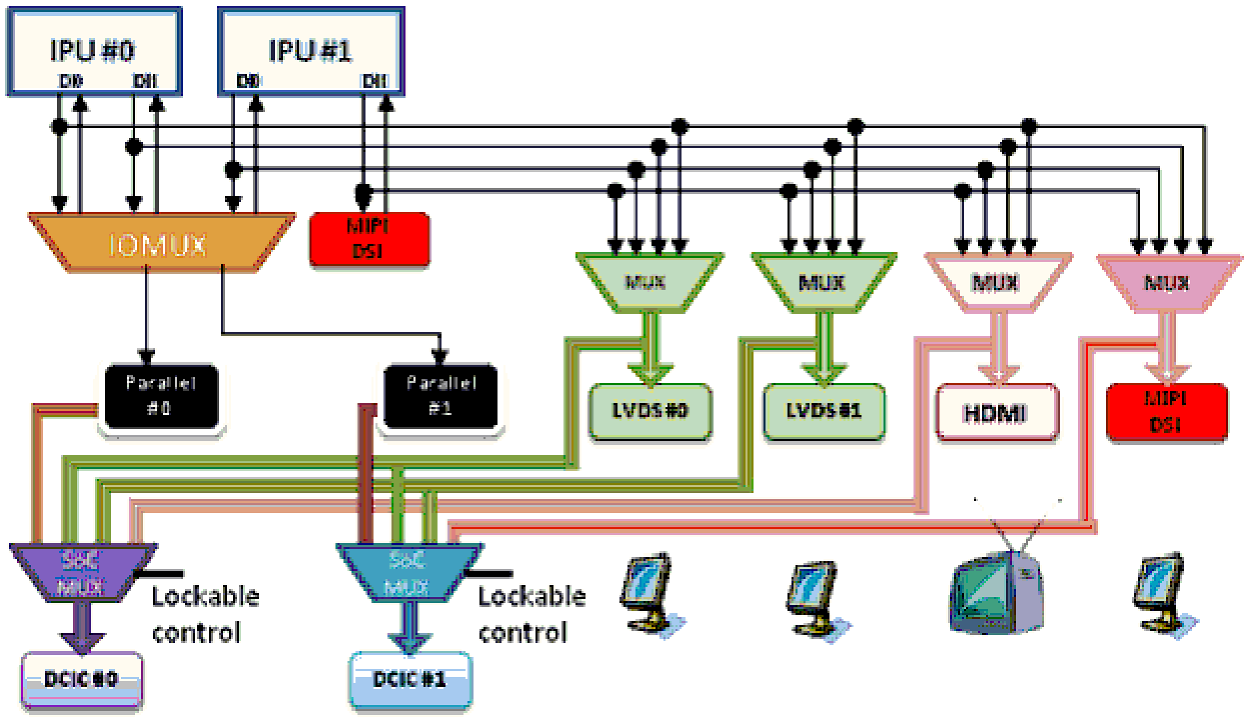


Figure 10-3. Available mux configurations

10.7 Using HDMI audio

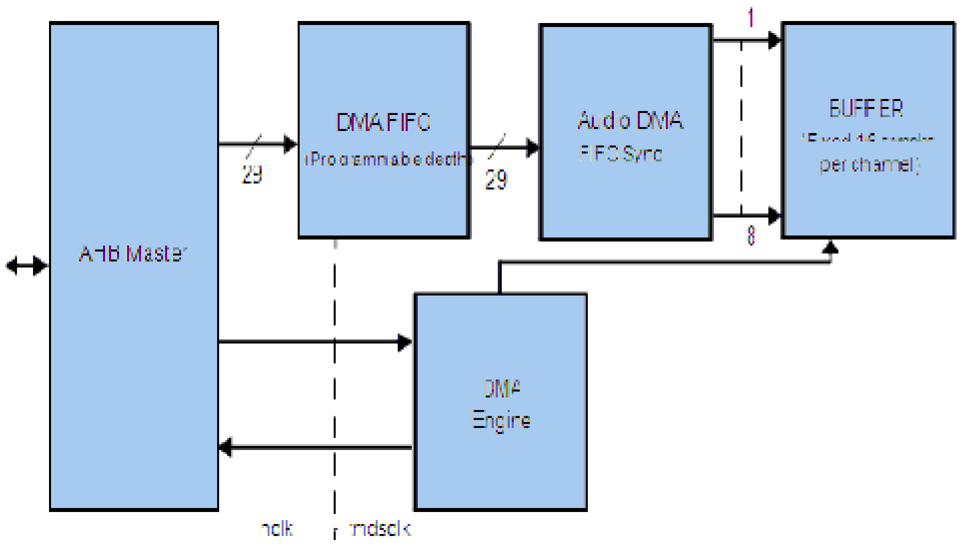


Figure 10-4. Audio DMA interface block diagram

HDMI TX uses a direct memory access (DMA) interface to receive the incoming samples directly from external memory. No other interfaces are supported for this purpose.

The DMA interface supports up to eight channels of L-PCM/NL-PCM and HBR audio, which allows all audio formats. \This requires preformatting the audio data in memory for the specific audio mode that the HDMI TX module is configured for.

10.8 Using HDCP

HDMI TX implements high-bandwidth digital copy protection (HDCP). The HDCP transmitter implements the three layers of the HDCP cipher, including LFSR and other functions required to generate encryption key bytes. These bytes are then XORed with the data.

The process is as follows:

1. After the system has been properly configured, the HDMI TX controller unit authenticates the slave device through the I^{2}C link.
2. After authentication, HDMI TX encrypts the TMDS channels; it signals the slave device when it begins this process.
3. Periodically, the HDCP port is checked for integrity by ensuring the slave side is properly decrypting the content.

10.9 Initializing the driver

The HDMI TX module driver works by using data structures that provide the available configurability of the module. Before using the HDMI TX module driver in a system, instances of these data structures must be instantiated and initialized. The driver functions then use these data structures as parameters to properly initialize the HDMI TX module.

In addition, the input video source must be set-up as described in Setting up the HDMI video input, using the following function:

```
void hdmi_config_input_source(uint32_t mux_value);
```

The primary data structures are defined in the ./src/sdk/hdmi/drv/hdmi_tx.h file. They are as follows:

```
typedef struct hdmi_vmode {
    unsigned int mCode;
    unsigned int mHdmiDviSel;
    unsigned int mRVBlankInOSC;
    unsigned int mRefreshRate;
    unsigned int mHImageSize;
    unsigned int mVImageSize;
    unsigned int mHActive;
    unsigned int mVActive;
    unsigned int mHBlanking;
    unsigned int mVBlanking;
    unsigned int mHSyncOffset;
    unsigned int mVSyncOffset;
    unsigned int mHSyncPulseWidth;
    unsigned int mVSyncPulseWidth;
    unsigned int mHSyncPolarity;
    unsigned int mVSyncPolarity;
    unsigned int mDataEnablePolarity;
    unsigned int mInterlaced;
    unsigned int mPixelClock;
    unsigned int mHBorder;
    unsigned int mVBorder;
    unsigned int mPixelRepetitionInput;
} hdmi_vmode_s;
typedef struct hdmi_data_info {
    unsigned int enc_in_format;
    unsigned int enc_out_format;
    unsigned int enc_color_depth;
    unsigned int colorimetry;
    unsigned int pix_repet_factor;
    unsigned int hdcp_enable;
    hdmi_vmode_s * video_mode;
} hdmi_data_info_s;
typedef struct hdmi_AudioParam {
          unsigned char IecCgmsA;
          int IecCopyright;
          unsigned char IecCategoryCode;
          unsigned char IecPcmMode;
          unsigned char IecSourceNumber;
          unsigned char IecClockAccuracy;
          unsigned int OriginalSamplingFrequency;
          unsigned char ChannelAllocation;
          unsigned int SamplingFrequency;
          unsigned char SampleSize;
} hdmi_audioparam_s;
Once these data structures are declared and initialzed, they can be passed in directly to
the
HDMI TX driver functions to complete the initialization.  The primary HDMI TX initialization
functions include:
void hdmi_av_frame_composer(hdmi_data_info_s * hdmi_instance);
void hdmi_video_packetize(hdmi_data_info_s hdmi_instance);
void hdmi_video_csc(hdmi_data_info_s hdmi_instance);
void hdmi_video_sample(hdmi_data_info_s hdmi_instance);
void hdmi_tx_hdcp_config(uint32_t de);
void hdmi_phy_init(uint8_t de, uint16_t pclk);
int audio_Configure(hdmi_audioparam_s audioparam, uint16_t pixelClk, unsigned ratioClk);
```

10.10 Testing the driver

The HDMI TX unit test demonstrates how to output both audio and video out of the HDMI signals. The test utilizes the IPU unit test to initialize the IPU to output a Freescale logo picture to the HDMI block. It also sets up a sine wave audio sample which it outputs out of the HDMI signals along with the IPU output. This is done in the ./src/sdk/hdmi/test/hdmi_test.c file:

```
// declare hdmi module object data structures
hdmi_data_info_s myHDMI_info = { 0 };
hdmi_vmode_s myHDMI_vmode_info = { 0 };
myHDMI_info.video_mode = &myHDMI_vmode_info;
hdmi_audioparam_s myHDMI_audio_info = { 0 };
//initialize HDMI TX data structures for 1080p
myHDMI_info.enc_in_format = eRGB;
myHDMI_info.enc_out_format = eRGB;
myHDMI_info.enc_color_depth = 8;
myHDMI_info.colorimetry = eITU601;
myHDMI_info.pix_repet_factor = 0;
myHDMI_info.hdcp_enable = 0;
myHDMI_info.video_mode->mCode = 16; //1920x1080p @ 59.94/60Hz 16:9
myHDMI_info.video_mode->mHdmiDviSel = TRUE;
myHDMI_info.video_mode->mRVBlankInOSC = FALSE;
myHDMI_info.video_mode->mRefreshRate = 60000;
myHDMI_info.video_mode->mDataEnablePolarity = TRUE;
myHDMI_audio_info.IecCgmsA = 0;
myHDMI_audio_info.IecCopyright = TRUE;
myHDMI_audio_info.IecCategoryCode = 0;
myHDMI_audio_info.IecPcmMode = 0;
myHDMI_audio_info.IecSourceNumber = 1;
myHDMI_audio_info.IecClockAccuracy = 0;
myHDMI_audio_info.OriginalSamplingFrequency = 0;
myHDMI_audio_info.ChannelAllocation = 0xf;
myHDMI_audio_info.SamplingFrequency = 32000;
myHDMI_audio_info.SampleSize = 16;
//use hdmi driver functions to initialize the module
hdmi_av_frame_composer(&myHDMI_info);
hdmi_video_packetize(myHDMI_info);
hdmi_video_csc(myHDMI_info);
hdmi_video_sample(myHDMI_info);
hdmi_audio_mute(TRUE);
hdmi_tx_hdcp_config(myHDMI_info.video_mode->mDataEnablePolarity);
hdmi_phy_init(TRUE, myHDMI_info.video_mode->mPixelClock);
hdmi_config_input_source(IPU1_DI0); // configure input source to HDMI block
// set-up Audio for test
init_dma_data(32000, 4);
//enable HDMI_TX interrupt for audio DMA
register_interrupt_routine(IMX_INT_HDMI_TX, hdmi_tx_ISR);
init_hdmi_interrupt();
enable_interrupt(IMX_INT_HDMI_TX, CPU_0, 0);    // to cpu0, max priority (0)
audio_Configure(myHDMI_audio_info, 14850, 100);
writeb(0x00, HDMI_IH_MUTE_AHBDMAAUD_STAT0);
audio_Configure_DMA(AUDIO_BUF_START, (AUDIO_BUF_START + 0x17ff), 1, 4, 64, 4, 0x7f);
writeb(0x2, HDMI_IH_MUTE);  //reg8_write(HDMI_IH_MUTE,0x2);
//use IPU unit test to provide video input for HDMI block
if (ips_hdmi_stream()) {    // set up ipu1 disp0  1080P60 display stream
    printf("HDMI video test PASS\n");
    printf("Audio will play for 2 more seconds\n");
    hal_delay_us(2000000);  // play hdmi audio for 2 seconds
    disable_interrupt(IMX_INT_HDMI_TX, CPU_0);
    printf("---- END of HDMI_TX test ----\n");
} else {
    printf("HDMI video test FAIL\n");
    disable_interrupt(IMX_INT_HDMI_TX, CPU_0);
    printf("---- END of HDMI_TX test ----\n");
}
```

10.11 Source code structure

Table 10-4. Source code file locations

Description	Location
Low-level driver source for video blocks	./src/sdk/hdmi/drv/hdmi_tx.c
Low-level driver header	./src/sdk/hdmi/drv/hdmi_tx.h
Low-level driver source for Audio blocks	./src/sdk/hdmi/drv/hdm_tx_audio.c
Low-level driver source for PHY block	./src/sdk/hdmi/drv/hdmi_tx_phy.c
Comon functions used by driver	./src/sdk/hdmi/drv/hdmi_common.c

Table continues on the next page...

Table 10-4. Source code file locations (continued)

HDMI TX unit test	./src/sdk/hdmi/test/hdmi_test.c
Audio test set-up function and audio sample	./src/sdk/hdmi/drv/init_dma_data.c

Chapter 11
Configuring the I2C Controller as a Master Device

11.1 Overview

This chapter provides a quick guide for firmware developers about how to write the driver for the I2C controller, which provides an efficient interface to the I2C bus. The I2C bus is a two-wire, bidirectional serial bus that provides an efficient method of data exchange to minimize the interconnection between devices. The I2C controller provides the functionality of standard I2C slave and master. This guide targets the development of the I2C master mode driver.

There are three instances of I2C in the chip. They are located in memory map at the following addresses:

- I2C1 base address-021A 0000h
- I2C2 base address-021A 4000h
- I2C3 base address-021A 8000h

For register definitions and information, refer to the chip reference manual.

This chapter assumes an understanding of the I2C bus specification, version 2.1. However, a brief introduction to I2C protocol is discussed in I2C protocol.

NOTE

This chapter uses i.MX 6Dual/6Quad ARD board schematics as its reference for pin assignments. For other board types, refer to the appropriate schematics.

11.2 Initializing the I2C controller

To initialize the I2C controller, configure the following two I^{2}C signals: clock initialization and the programming frequency divider register (I2Cn_IFDR). The following subsections explain how to do this.

11.2.1 IOMUX pin configuration

Refer to your board schematics for correct pin assignments. The following table shows the contacts assigned to the signals that the three I2C blocks use:

Table 11-1. I2C pin assignments

Signals	I2C1			I2C2			I2C3		
	PAD	MUX	SION	PAD	MUX	SION	PAD	MUX	SION
SDA	EIM_D28	ALT1	1	EIM_D16	ALT6	1	EIM_D18	ALT6	1
	CSI0_DAT8	ALT4	1	KEY_ROW3	ALT4	1	GPIO_6	ALT2	1
							GPIO_16	ALT6	1
SCL	EIM_D21	ALT6	1	EIM_EB2	ALT6	1	EIM_D17	ALT6	1
	CSI0_DAT9	ALT4	1	KEY_ROW3	ALT4	1	GPIO_3	ALT2	1
							GPIO_5	ALT6	1

NOTE

Set the SION (Software Input ON) bit of the software MUX control register to force MUX input path.

Program the pad setting register to have pull up enabled in open drain mode or ensure that pull ups are connected externally to each lines .

For more information about the IOMUX controller, refer to the IOMUXC chapter of the i.MX 6Dual/6Quad reference manual.

11.2.2 Clocks

The following figure shows the clock control signals.

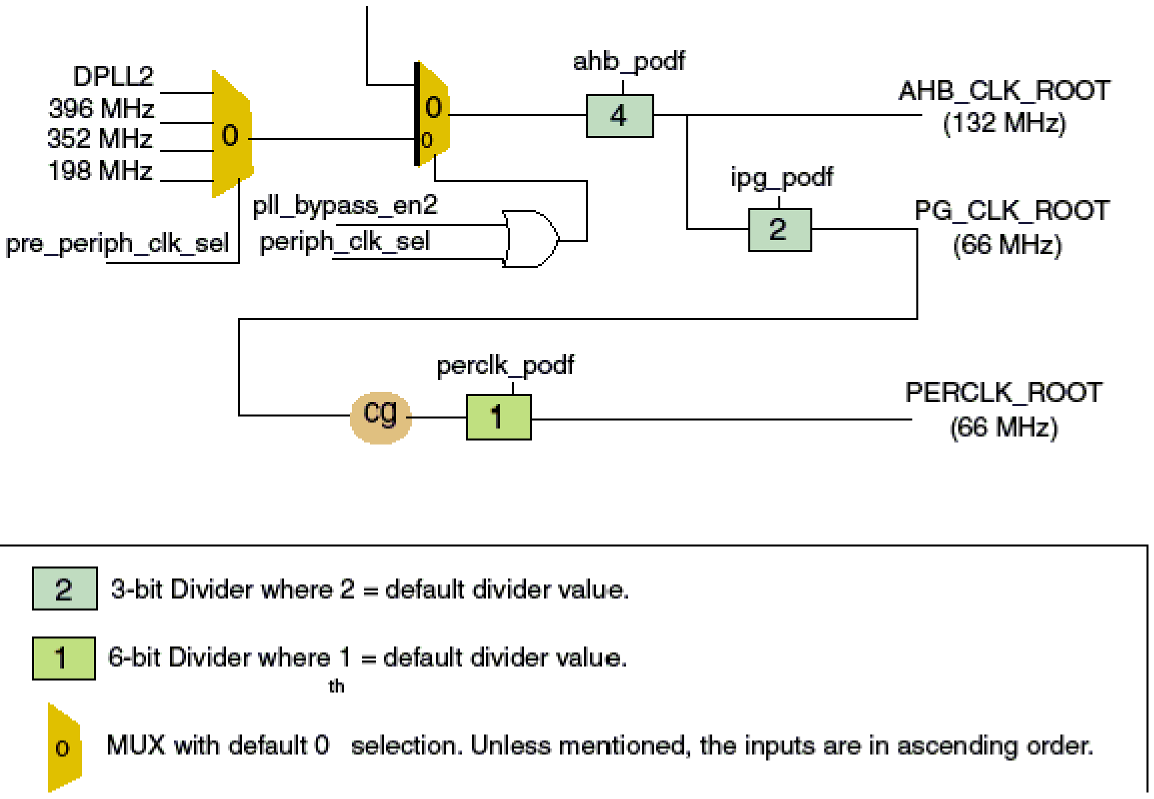

DPLL2
396 MHz
352 MHz
198 MHz
0
pre_periph_clk_sel
0
0
pll_bypass_en2
periph_clk_sel
ahb_podf
4
AHB_CLK_ROOT
(132 MHz)
ipg_podf
2
PG_CLK_ROOT
(66 MHz)
perclk_podf
cg
1
PERCLK_ROOT
(66 MHz)
2
3-bit Divider where 2 = default divider value.
1
6-bit Divider where 1 = default divider value.
0
MUX with default 0th selection. Unless mentioned, the inputs are in ascending order.

Figure 11-1. Clock control signals for I2C blocks

The I2C uses PERCLK_ROOT as its clock source. PERCLK_ROOT is derived from IPG_CLK_ROOT. The IPG_CLK_ROOT runs at 66 MHz with default dividers, as shown in the above figure.

IPG_CLK_ROOT is derived from PLL2, which typically runs at 528 MHz. If PLL2 is programmed to run at a speed other than 528 MHz, the IPG_CLK_ROOT output speed will also be different. To set the desired source speed for I2C clock, adjust the dividers by setting the fields ahb_podf and ipg_podf of CCM_CBCDR and perclk_podf of CCM_CSCMR1. Refer to the register description for further information.

If the I2C clock is gated, ungate it as follows:

- For I2C1, set bits CCM_CCGR2[7:6]
- For I2C2, set bits CCM_CCGR2[9:8]
- For I2C3, set bits CCM_CCGR2[11:10]

Refer to the CCM chapter of the i.MX 6Dual/6Quad Reference Manual for more information about programming clocks.

11.2.3 Configuring the programming frequency divider register (IFDR)

The I2C module can operate at speeds up to 400 kbps. The driver calculates the SCL frequency automatically by passing the desired baud rate to the initialization function. Nevertheless, the following steps can be used to set the I2C frequency divider to get appropriate transfer speed.

1. Software reset I2C block before changing the I2C frequency divider register by clearing the I2C*n*_I2CR register.

   ```
   write(0, I2Cn_I2CR);
   ```

   - For 100 kbps speed, program I2C*n*_IFDR to 14h.
   - For 400 kbps, program IFDR to a value of Eh.

     The source clock for I2C is PERCLK_ROOT running at 66 MHz. According to the frequency divider table, a value of 14h set to the IFDR register results in a divider value of 576, and I2C_CLK = 66 MHz ÷ 576 = 100 Kbps. Refer to table below for the IFDR frequency divider values.

     ```
     write(0x14, i2c_base_register_address + I2C_IFDR);
     ```
2. Enable the I2C module by setting I2C_I2CR[IEN].

```
write(IEN, i2c_base_register_address + I2C_I2CR);
```

The following table shows the divider values for I2C*n*_IFDR register settings.

Table 11-2. I2Cn_IFDR[5:0] Register Field Values

IC	Divider	IC	Divider	IC	Divider	IC	Divider
00h	30	10h	288	20h	22	30h	160
01h	32	11h	320	21h	24	31h	192
02h	36	12h	384	22h	26	32h	224
03h	42	13h	480	23h	28	33h	256
04h	48	14h	576	24h	32	34h	320
05h	52	15h	640	25h	36	35h	384
06h	60	16h	768	26h	40	36h	448
07h	72	17h	960	27h	44	37h	512
08h	80	18h	1152	28h	48	38h	640
09h	88	19h	1280	29h	56	39h	768
0Ah	104	1Ah	1536	2Ah	64	3Ah	896
0Bh	128	1Bh	1920	2Bh	72	3Bh	1024
0Ch	144	1Ch	2304	2Ch	80	3Ch	1280
0Dh	160	1Dh	2560	2Dh	96	3Dh	1536

Table continues on the next page...

Table 11-2. I2Cn_IFDR[5:0] Register Field Values (continued)

IC	Divider	IC	Divider	IC	Divider	IC	Divider
0Eh	192	1Eh	3072	2Eh	112	3Eh	1792
0Fh	240	1Fh	3840	2Fh	128	3Fh	2048

11.3 I2C protocol

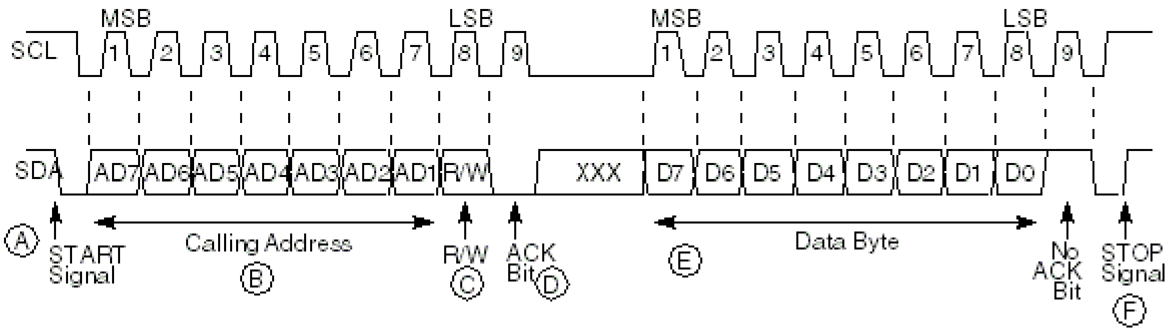


Figure 11-2. I2C standard communication protocol

The I2C communication protocol consists of the following six components:

- START
- Data Source/Recipient
- Data Direction
- Slave Acknowledge
- Data Acknowledge
- STOP

11.3.1 START signal

When no other device is a bus master (both SCL and SDA lines are at logic high), a device can initiate communication by sending a START signal. A START signal is defined as a high-to-low transition of SDA while SCL is high. This signal denotes the beginning of a data transfer (each data transfer can be several bytes long) and awakens all slaves.

NOTE

Setting the MSTA bit of the I2CR register generates a START on the bus and selects master mode.

11.3.2 Slave address transmission

The master sends the slave address in the first byte after the START signal (B). After the seven-bit calling address, it sends the R/W bit (C), which tells the slave the data transfer direction.

Each slave must have a unique address. An I2C master must not transmit an address that is the same as its slave address; it cannot be master and slave at the same time.

The slave whose address matches that sent by the master pulls SDA low at the ninth clock (D) to return an acknowledge bit.

NOTE

The slave address is sent along with the R/W bit using the I2C*n*_I2DR register. When cleared, I2C*n*_I2SR[RXAK] denotes the ACK bit received.

11.3.3 Data transfer

When successful slave addressing is achieved, the data transfer can proceed (E) on a byte-by-byte basis in the direction specified by the R/W bit sent by the calling master in a slave address transmission.

Data can be changed only while SCL is low and must be held stable while SCL is high. SCL is pulsed once for each data bit, most-significant bit first. The receiving device must acknowledge each byte by pulling SDA low at the ninth clock; therefore, a data byte transfer takes nine clock pulses.

If it does not acknowledge the master, the slave receiver must leave SDA high. The master can then generate a STOP signal to abort the data transfer or generate a START signal (a repeated start) to start a new calling sequence.

If the master receiver does not acknowledge the slave transmitter after a byte transmission, it means end-of-data to the slave. The slave releases SDA for the master to generate a STOP or START signal.

NOTE

Writing to the data register triggers the transmit operation.

Transmit data should always be written after I2C*n*_I2CR[MTX] bit is programmed. Transmit data is not latched inside until the transfer is initiated on the interface bus.

After the transmit data write in I2C, software can either wait for a transfer-done interrupt or it can poll I2C*n*_I2SR[ICF] for zero if new data had to be written during the previous data transfer. I2C*n*_I2SR[IIF] may not be polled if I2C*n*_I2CR[IIEN] is set because the I2C generates an interrupt when IIF is set.

NOTE

When cleared, I2C*n*_I2SR[RXAK] denote the ACK bit was received.

11.3.4 STOP signal

The master can terminate communication by generating a STOP signal to free the bus. A STOP signal is defined as a low-to-high transition of SDA while SCL is at logical high (F).

NOTE

A master can generate a STOP even if the slave has made an acknowledgment, at which point the slave must release the bus. Clearing the I2C*n*_I2CR[MSTA] bit generates a STOP and selects slave mode.

11.3.5 Repeat start

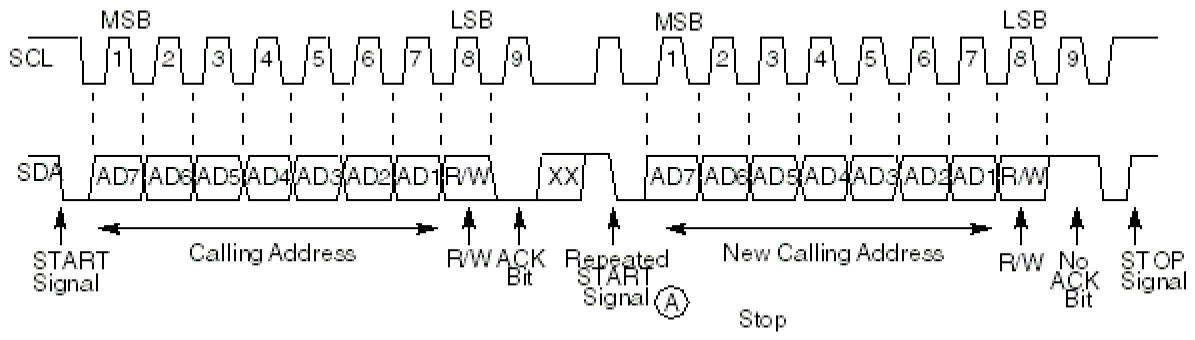


Figure 11-3. Repeated START

Instead of signaling a STOP, the master can repeat the START signal, followed by a calling command. A repeated START occurs when a START signal is generated without first generating a STOP signal to end the communication. The master uses a repeated START to communicate with another slave or with the same slave in a different mode (transmit/receive mode) without releasing the bus.

NOTE

Setting I2C*n*_I2CR[RSTA] bit generates a repeat start condition.

11.4 Programming controller registers for I2C data transfers

his section describes how to program I2C controller registers I2C*n*_I2CR, I2C*n*_I2SR, and I2C*n*_I2DR for transferring data on an I2C bus. Pseudocode is provided wherever necessary.

11.4.1 Function to initialize the I2C controller

This initialization function uses two parameters: the base address of the initialized controller and the desired baud rate used for the I2C bus. The function:

- Manages the controller's clock gating
- Calculates the divider to get the desired frequency of SCL
- Enables the controller

```
/*!
 * Initialize the I2C module -- mainly enable the I2C clock, module
 * itself and the I2C clock prescaler.
 *
 * @param   base        base address of I2C module (also assigned for I2Cx_CLK)
 * @param   baud        the desired data rate in bps
 *
 * @return  0 if successful; non-zero otherwise
 */
int i2c_init(uint32_t base, uint32_t baud)
```

11.4.2 Programming the I2C controller for I2C Read

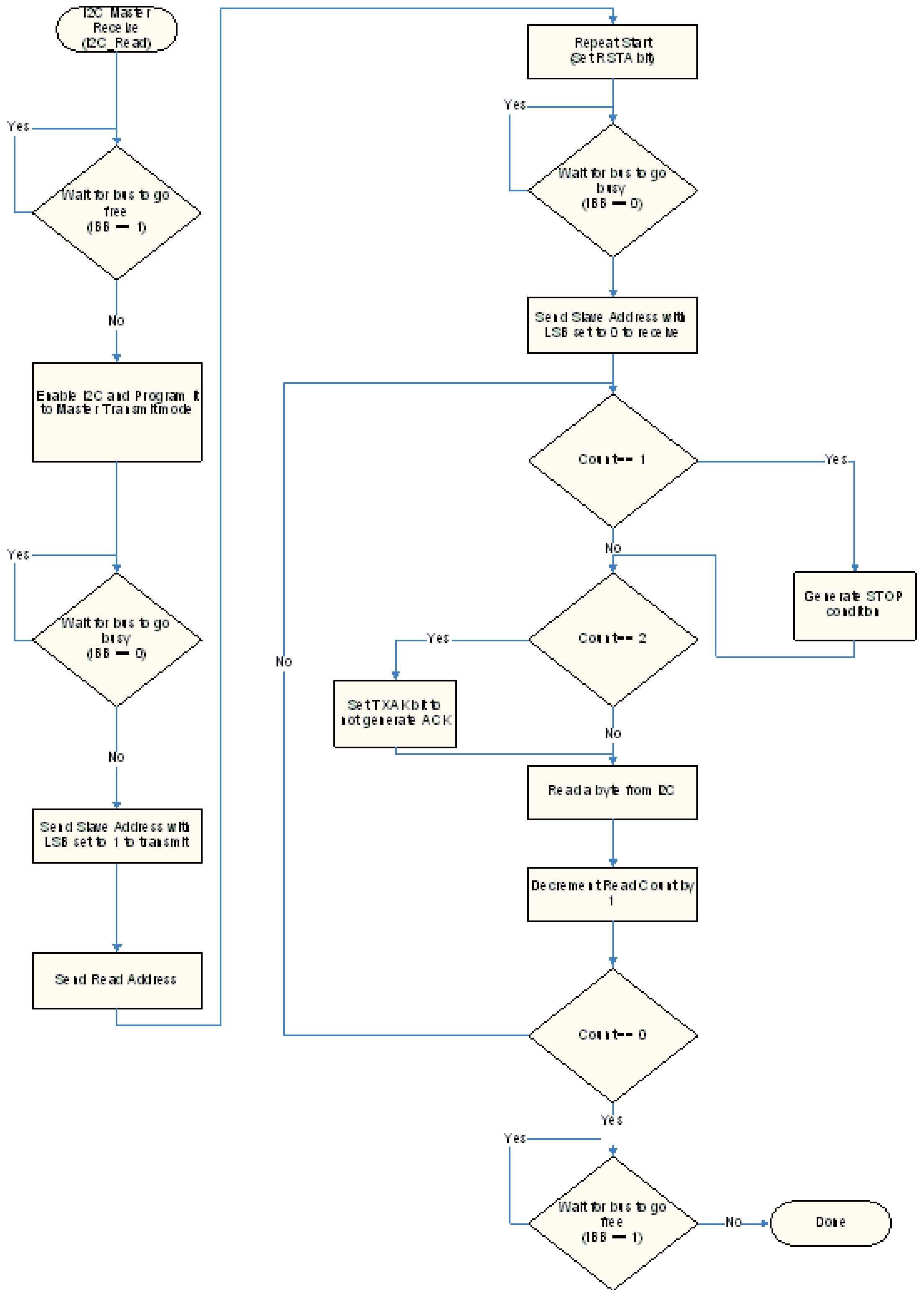

I2C Master Receive (I2C_Read)
Yes
Wait for bus to go free (IBB == 1)
No
Enable I2C and Program it to Master Transmit mode
Yes
Wait for bus to go busy (IBB == 0)
No
Send Slave Address with LSB set to 1 to transmit
Send Read Address
Repeat Start (set RSTA bit)
Yes
Wait for bus to go busy (IBB == 0)
Send Slave Address with LSB set to 0 to receive
Count == 1
Yes
Generate STOP condition
No
No
Count == 2
Yes
Set TXAK bit to not generate ACK
No
Read a byte from I2C
Decrement Read Count by 1
Count == 0
Yes
Yes
Wait for bus to go free (IBB == 1)
No
Done

Figure 11-4. Flow chart for I2C read

Program the I2C controller according to the following sequence to receive bytes (read) a device.

```
<WAITBUSFREE><START><WAITBUSBUSY><SLV ADR><W><ADR MSB><ADR LSB><RPT START><WAITBUSBUSY><SLV
ADR><R><READ BYTE><ACK><READ BYTE><STOP>
```

<WAITBUSFREE> Wait until I2C*n*_I2SR[IBB] (I2C bus busy) bit is high. Wait for bus to go free.

<START> Start signaled by master

<WAITBUSBUSY> Wait until I2C*n*_I2SR[IBB] (I2C bus busy) bit is low. Wait for bus to go busy.

<SLV ADR><W> 7-bit slave address and last bit set to 1. This indicates to the slave device with the matching slave address that a transmit operation from master to slave is being issued by master. Slave responds with ACK; software can read the ACK using I2C*n*_I2SR[RXAK] bit. If it is 1, then no ACK was received and software can issue stop signal.

<ADR MSB> .. . <ADR LSB> Depending on device type this could be a memory offset, a command, or a register address. The address may be a byte or multiple bytes depending on device type. Master should be programmed to transmit this data byte at a time using I2C*n*_I2DR. This address tells the slave I2C device what data master is requesting. Slave ACK after receiving each byte and software should make sure I2C*n*_I2SR[RXAK] bit is set in order to confirm ACK is received.

<RPT START> Repeat start signaled by master

<SLA ADR><R> 7 bit slave address and last bit set to 0, this will indicate to the slave device that master is ready to receive data.

<READ BYTE><ACK> , Master reads data from slave byte at a time. The I2C*n*_I2DR register is used by software to read the byte. The I2C controller issues the ACK bit upon reading each byte; once the specified number of bytes are received, software should program the I2C controller's I2C*n*_I2CR[TXAK] bit to not generate ACK.

<STOP> Stop signaled by master

11.4.3 Code used for I2C read operations

This section provides the functions used in an I2C read operation.

The function i2c_xfer is used with the I2C_READ parameter.

```
/*!
 * This is a rather simple function that can be used for most I2C devices.
 * Common steps for both READ and WRITE:
```

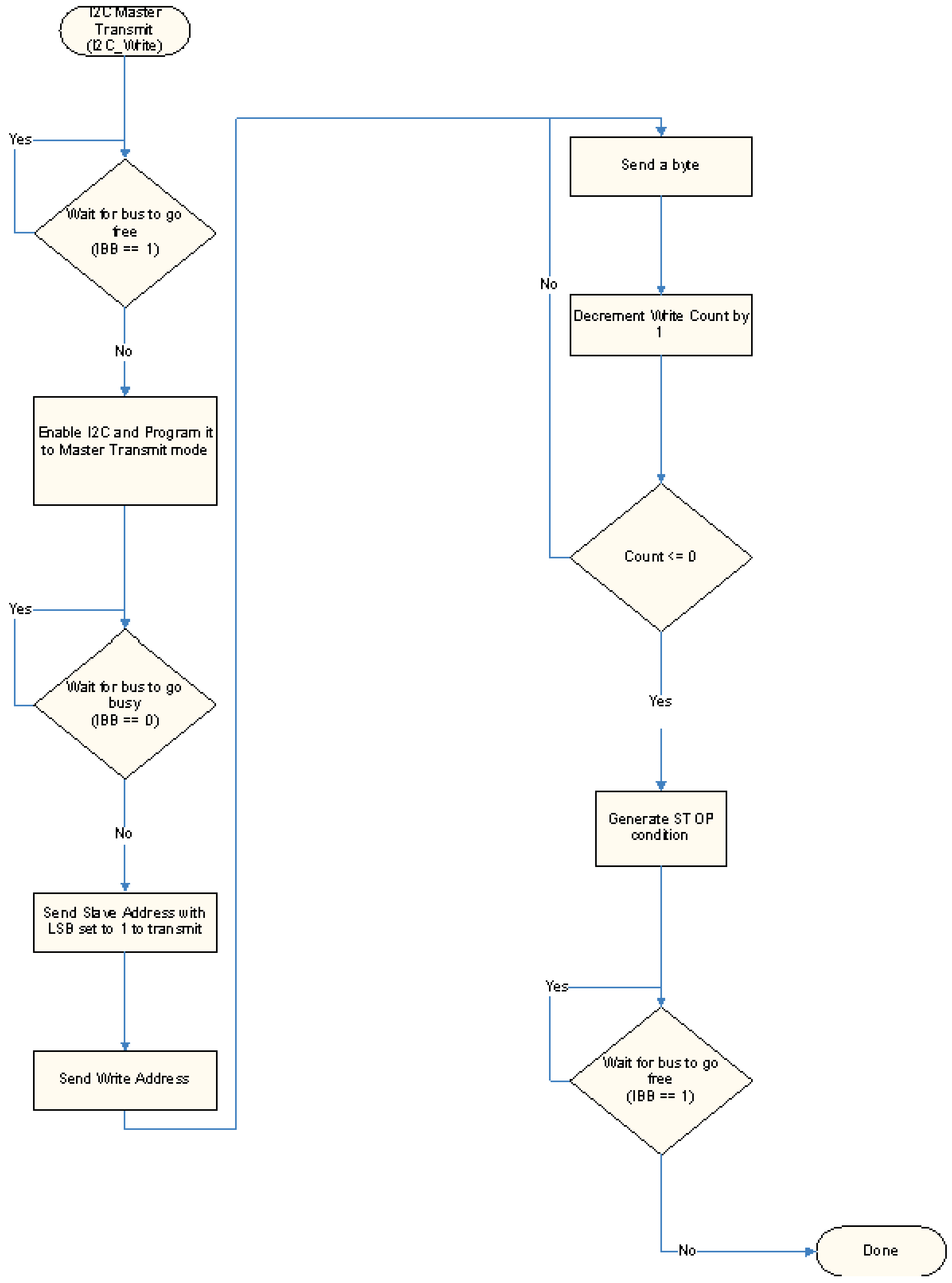
```
 *      step 1: issue start signal
 *      step 2: put I2C device addr on the bus (always 1 byte write. the dir always
I2C_WRITE)
 *      step 3: offset of the I2C device write (offset within the device. can be 1-4 bytes)
 * For READ:
 *      step 4: do repeat-start
 *      step 5: send slave address again, but indicate a READ operation by setting LSB bit
 *      Step 6: change to receive mode
 *      Step 7: dummy read
 *      Step 8: reading
 * For WRITE:
 *      Step 4: do data write
 *      Step 5: generate STOP by clearing MSTA bit
 *
 * @param   rq        pointer to struct imx_i2c_request
 * @param   dir       I2C_READ/I2C_WRITE
 *
 * @return  0 on success; non-zero otherwise
 */
int32_t i2c_xfer(struct imx_i2c_request *rq, int dir)

/*!
 * For master RX
 * Implements a loop to receive bytes from I2C slave.
 *
 * @param   base        base address of I2C module
 * @param   data        return buffer for data
 * @param   sz          number of bytes to receive
 *
 * @return  0 if successful; -1 otherwise
 */
static int rx_bytes(uint8_t * data, uint32_t base, int sz)

/*!
 * For master TX
 * Implements a loop to send a byte to I2C slave.
 * Always expect a RXAK signal to be set!
 *
 * @param   base        base address of I2C module
 * @param   data        return buffer for data
 *
 * @return  0 if successful; -1 otherwise
 */
static int tx_byte(uint8_t * data, uint32_t base)

/*!
 * wait for operation done
 * This function loops until we get an interrupt. On timeout it returns -1.
 * It reports arbitration lost if IAL bit of I2SR register is set
 * Clears the interrupt
 * If operation is transfer byte function will make sure we received an ack
 *
 * @param   base        base address of I2C module
 * @param   is_tx       Pass 1 for transfering, 0 for receiving
 *
 * @return  0 if successful; negative integer otherwise
 */
static int wait_op_done(uint32_t base, int is_tx)
```

11.4.4 Programming the I2C controller for I2C Write


I2C Master Transmit (I2C_Write)
Yes
Wait for bus to go free (IBB == 1)
No
Enable I2C and Program it to Master Transmit mode
Yes
Wait for bus to go busy (IBB == 0)
No
Send Slave Address with LSB set to 1 to transmit
Send Write Address
Send a byte
Decrement Write Count by 1
No
Count <= 0
Yes
Generate STOP condition
Yes
Wait for bus to go free (IBB == 1)
No
Done

Use the following sequence to program the I2C controller to send data bytes (write) to the device.

```
<WAITBUSFREE><START><WAITBUSBUSY><SLV ADR><W><ADR MSB><ADR LSB><WRITE BYTE><WRITE
BYTE><STOP>
```

<WAITBUSFREE> Wait until I2Cn_I2SR[IBB] (I2C bus busy) bit is high. Wait for bus to go free.

<START> Start signaled by master

<WAITBUSBUSY> Wait until I2C*n*_I2SR[IBB] (I2C bus busy) bit is low. Wait for bus to go busy.

<SLV ADR><W> 7 bit slave address and last bit set to 1, this will indicate to the slave device with matching slave address that a transmit operation from master to slave is being issued by master. Slave responds with ACK, software can read the ACK using I2C*n*_I2SR[RXAK] bit. If it is 1 then no ACK received and software can issue stop signal.

<ADR MSB> ... <ADR LSB> Depending on device type this could be a memory offset, a command or a register address. The address could be just a byte or multiple bytes depending on device type. Master should be programmed to transmit this data byte at a time using I2C*n*_I2DR. This address tells the slave I2C device what data master is going to send. Slave ACK after receiving each byte and software should make sure I2C*n*_I2SR[RXAK] bit is 1 to confirm ACK is received.

<WRITE BYTE> Master sends data to slave byte at a time. The I2C*n*_I2DR register is used by software to read the byte. Slave ACK after receiving each byte and software should make sure RXAK bit is 1 to confirm ACK is received.

<STOP> Stop signaled by master

11.4.5 Code used for I2C write operations

Code used for I2C read operations provides the description of the functions used in the driver.

For an I2C write operation, use the function i2c_xfer with the I2C_WRITE parameter.

Chapter 12
Configuring the I2C Controller as a Slave Device

12.1 Overview

This chapter explains how to configure the I^{2}C controller as a slave device.

There are three instances of I^{2}C in the chip, located in the memory map at the base addresses:

- I2C1 at 021A 0000h
- I2C2 at 021A 4000h
- I2C3 at 021A 8000h

12.2 Features summary

This low-level driver supports:

- Usage of an I^{2}C controller as a slave device

12.3 Modes of operation

The following table explains the I^{2}C slave driver modes of operation:

Table 12-1. I^{2}C slave driver modes of operation

Mode	What it does
Slave device	The controller is configured with a user's defined slave device ID. It executes the user's defined transmit and receive operations as commanded by an external master. The driver is able to automatically receive and transmit like a memory, with a pre-defined number of address cycles and unlimited number data cycles.

12.4 Clocks

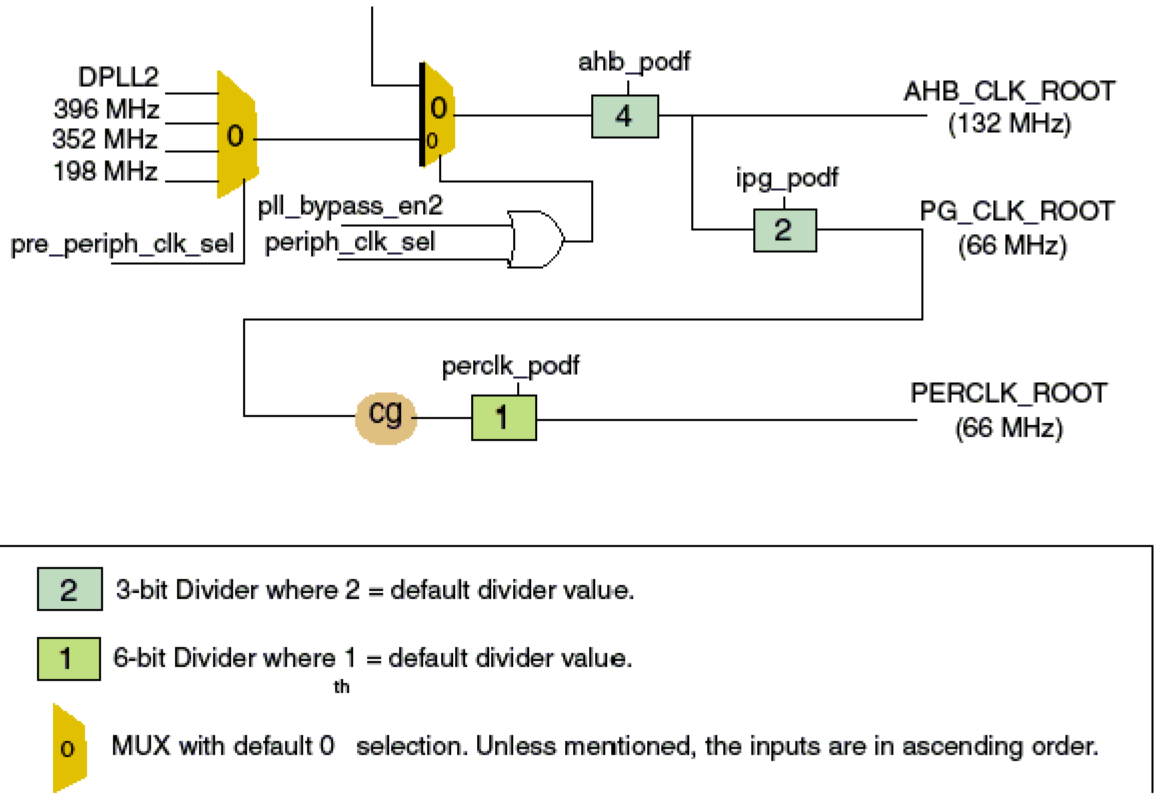

DPLL2
396 MHz
352 MHz
198 MHz
0
pre_periph_clk_sel
0
0
pll_bypass_en2
periph_clk_sel
ahb_podf
4
AHB_CLK_ROOT
(132 MHz)
ipg_podf
2
PG_CLK_ROOT
(66 MHz)
perclk_podf
cg
1
PERCLK_ROOT
(66 MHz)
2 3-bit Divider where 2 = default divider value.
1 6-bit Divider where 1 = default divider value.
0 MUX with default 0 th selection. Unless mentioned, the inputs are in ascending order.

Figure 12-1. Clock control signals for I2C blocks

This controller uses IPG_CLK as its single input clock. The frequency of the I2C bus SCL signal is calculated based on the IPG_CLK frequency and block divider defined in the I2C Frequency Divider Register (IFDR).

The SCL frequency is simply the IPG_CLK frequency divided by any of the values defined in the "I2C_IFDR Register Field Values" table (see the I^{2}C chapter in the chip reference manual)

Table 12-2. I^{2}C slave driver clocks

Clock	Name	Description
IPG_CLK	IPG_CLK	Global IPG_CLK that is typically used in normal operation. It is provided by CCM. It cannot be powered down.

12.5 Resets and Interrupts

To save power, the driver disables the controller when not using it and enables the controller to use it. When disabled, the controller is reset.

The driver provides an interrupt routine (i2c_interrupt_routine) that reads the status register for later processing, depending on the active flags, and clears the register. The application routine address is passed through the hw_module data structure, which is defined in .src/include/io.h. This data structure is initialized by the application and used by the driver for various configurations.

All interrupt sources are listed in the "Interrupts and DMA Events" chapter of the chip reference manual. In the SDK, the list is provided at ./src/include/mx61/soc_memory_map.h.

12.6 Initializing the driver

The application should use the following function to initialize the I^2C controller. This function is available in the I^2C master driver at the location: ./src/sdk/i2c/drv/imx_i2c.c

```
/*!
 * Initialize the I2C module -- mainly enable the I2C clock, module
 * itself and the I2C clock prescaler.
 *
 * @param   base        base address of I2C module (also assigned for I2Cx_CLK)
 * @param   baud        the desired data rate in bps
 *
 * @return  0 if successful; non-zero otherwise
 */
int i2c_init(uint32_t base, uint32_t baud)
```

The following structure creates an I^2C request. It is defined in ./src/include/imx_i2c.h by

```
struct imx_i2c_request {
    uint32_t ctl_addr;          // the I2C controller base address
    uint32_t dev_addr;          // the I2C DEVICE address
    uint32_t reg_addr;          // the actual REGISTER address
    uint32_t reg_addr_sz;       // number of bytes for the address of I2C device register
    uint8_t *buffer;            // buffer to hold the data
    uint32_t buffer_sz;         // the number of bytes for read/write
    int32_t (*slave_receive) (struct imx_i2c_request *rq);  // slave receive data from master
    int32_t (*slave_transmit) (struct imx_i2c_request *rq); // slave transmit data to master
};
```

This structure provides the following information to the driver:

- ctl_addr is the I2C controller base address.
- dev_addr is the slave device address ID of the i.MX6.
- reg_addr is not used.
- reg_addr_sz is the number of address cycles that the master uses.

- *buffer is a pointer used for the data transfers.
- buffer_sz is not used.
- (*slave_receive) is a pointer to the function used to handle the received data. It takes an I2C request as parameter, which is typically this request.
- (*slave_transmitter) is a pointer to the function used to handle the transmitted data. It takes an I2C request as parameter, which is typically this request.

An hw_module data structure created in the application and defined in ./src/include/io.h is used to pass the interrupt number and base address of the used I^{2}C controller. Other parameters are not used.

Once the data is ready, the transfer function can be called. It returns when the access from the master is complete.

```
/*!
 * The slave mode behaves like any device with g_addr_cycle of address + g_data_cycle of
data.
 * Master read =
 * START - SLAVE_ID/W - ACK - MEM_ADDR - ACK - START - SLAVE_ID/R - ACK - DATAx - NACK - STOP
 * Example for a 16-bit address access:
 * 1st IRQ - receive the slave address and Write flag from master.
 * 2nd IRQ - receive the lower byte of the requested 16-bit address.
 * 3rd IRQ - receive the higher byte of the requested 16-bit address.
 * 4th IRQ - receive the slave address and Read flag from master.
 * 5th and next IRQ - transmit the data as long as NACK and STOP are not asserted.
 *
 * Master write =
 * START - SLAVE_ID/W - ACK - MEM_ADDR - ACK - DATAx - NACK - STOP
 *
 * 1st IRQ - receive the slave address and Write flag from master.
 * 2nd IRQ - receive the lower byte of the requested 16-bit address.
 * 3rd IRQ - receive the higher byte of the requested 16-bit address.
 * 4th and next IRQ - receive the data as long the STOP is not asserted.
 */
/*!
 * Handle the I2C transfers in slave mode.
 *
 * @param   port - pointer to the I2C module structure.
 * @param   rq - pointer to struct imx_i2c_request
 */
void i2c_slave_xfer(struct hw_module *port, struct imx_i2c_request *rq)
```

For more functional details, the **i2c_slave_xfer** function calls the following function. This function is a software implementation of the flow chart described in the I^{2}C chapter of the chip reference manual (see the "Flow Chart for Typical I2C Polling Routine" figure for the flow chart).

```
/*!
 * I2C handler for the slave mode. The function is based on the
 * flow chart for typical I2C polling routine described in the
 * I2C controller chapter of the reference manual.
 *
 * @param   rq - pointer to struct imx_i2c_request
 */
void i2c_slave_handler(struct imx_i2c_request *rq)
```

12.7 Testing the driver

A test is available to use the slave driver as a memory device. Any of the chip addresses (for example register or memory location) can be read or written to in this usage example.

12.7.1 Running the test

To run the I^2C slave test, the SDK builds the test with the following command:

```
./tools/build_sdk -target mx61 -board ard -board_rev 1 -test i2c
```

This generates the following ELF and binary files:

```
./output/mx61/bin/mx61ard-i2c-sdk.elf
./output/mx61/bin/mx61ard-i2c-sdk.bin
```

The I^2C test allows testing the controller in two ways:

- As a master, using an EEPROM as slave
- As a slave by using an external master connected to the appropriate I^2C bus.

The I^2C slave test allows the user to choose the device ID of the chip, as well as the number of address cycles (1 to 4) that this device should support when accessed by the master.

The data size is variable and automatically adjusted by the driver until the bus is busy (no ACK or STOP are received). However, the imx6_slave_transmit and imx6_slave_receive functions of the test application can only handle up to four transmitted or received bytes whatever the address size is. With 1- and 2-byte address accesses, the data is read from a reference data buffer or written onto the console to show the received data.

Take special care when performing a 4-byte address access because the imx6_slave_transmit and imx6_slave_receive service functions access a physical memory address of the chip. These functions assume that when using 4 address cycles, the data size is 4 bytes. Therefore, the targeted memory location must be 32-bit accessible.

For example, when the master performs a read access at the address 1000 0000h, the driver transmits the value read from this address, which is the base address of the SDRAM memory.

NOTE

The master used to validate that driver is an FTDI chip FT2232H mounted on an evaluation board from FTDI - FT2232H_Mini_Module.

12.8 Source code structure

Table 12-3. Source code file locations

Description	Location
Low-level driver source - common I2C functions	`./src/sdk/i2c/drv/imx_i2c.c`
Low-level driver source - slave mode specific functions	`./src/sdk/i2c/drv/imx_i2c_slave.c`
Low-level driver header	`./src/sdk/include/imx_i2c.h`
Unit test	`./src/sdk/i2c/test/i2c_test.c`
Unit test specific to slave mode	`./src/sdk/i2c/test/imx6_i2c_device.c`

Chapter 13
Configuring the IPU Driver

13.1 Overview

The IPU is a part of the video and graphics subsystem in the application processor. The goal of the IPU is to provide comprehensive support for the flow of data from an image sensor, and/or storage, to a display device.

This support covers all aspects of these activities:

- Connectivity to relevant devices: such as displays, graphics accelerators, and TV encoders
- Related image processing and manipulation: sensor image signal processing, display processing, image conversions, etc.
- Synchronization and control capabilities (to avoid tearing artifacts)

This integrative approach leads to several significant advantages:

- Automation: The involvement of the ARM platform in image management is minimized. In particular, display refresh/update can be performed completely autonomously. The resulting benefits are reducing the overhead due to software-hardware synchronization, freeing the ARM platform to perform other tasks and reducing power consumption (when the ARM core is idle and can be powered down).
- Optimal data path: Access to system memory is minimized. In particular, significant processing can be performed on-the-fly while sending data to a display. System memory is used only when a change in pixel order or frame rate is needed. The resulting benefits are reduced load on the system bus and further reduction of power consumption.
- Resource sharing: Maximum hardware reuse for different applications, resulting in the support of a wide range of requirements with minimal hardware.

The hardware reuse is enabled by a sophisticated configurability of each hardware block. This configurability also allows the support of a wide range of external devices, data formats, and operation modes. The resulting flexibility is also important because the support requirements are evolving significantly and expected future changes need to be anticipated and accounted for.

There are two equivalent instances of IPU in i.MX 6Dual/6Quad, located in the memory map at the following addresses:

- IPU1 base address = 0240 0000h
- IPU2 base address = 0280 0000h

The following figure provides a simple block diagram of IPU:

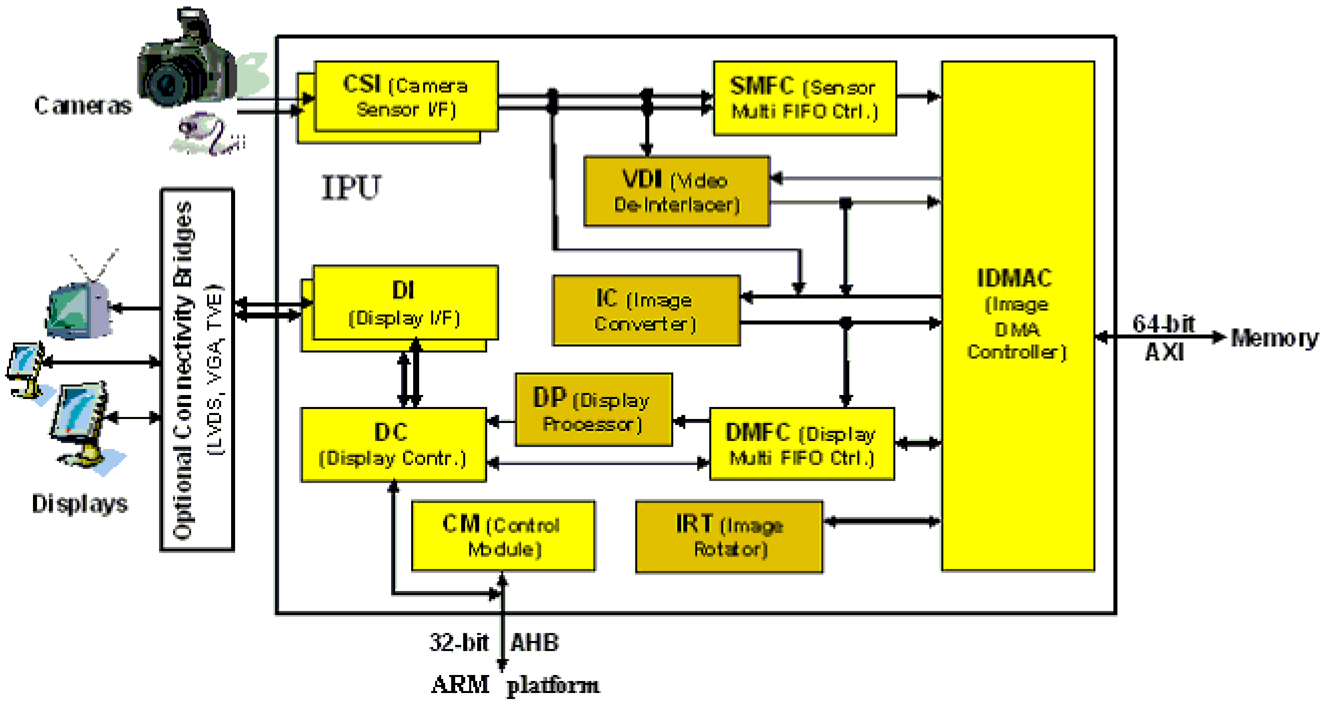


Figure 13-1. IPU block diagram

The following table describes the role of each block.

Table 13-1. IPU block descriptions

Block	Description
Camera Sensor Interface (CSI)	• Controls a camera port • Provides interface to an image sensor or a related device • Each IPU includes two CSI blocks

Table continues on the next page...

Table 13-1. IPU block descriptions (continued)

Block	Description
Display Interface (DI)	• Provides interface to displays, display controllers, and related devices • Each IPU includes two DI blocks
Display Controller (DC)	• Controls the display ports
Image Converter (IC)	• Performs resizing, color conversion/correction, combining with graphics, rotating, and horizontal inversion
Display Processor (DP)	• Performs the processing required for data sent to display.
Image Rotator (IRT)	• Performs rotation (90 or 180 degrees) and inversion (vertical/horizontal).
Image DMA Controller (IDMAC)	• Controls the memory port • Transfers data to/from system memory
Display Multi FIFO Controller (DMFC)	• Controls FIFOs for IDMAC channels related to the display system.
Sensor Multi FIFO Controller (SMFC)	• Controls FIFOs for output from the CSIs to system memory
Video De-Interlaced and Combiner (VDIC)	• Converts interlaced image into progressive and layers combination
Control Module (CM)	• Provides control and synchronization.

13.2 IPU task management

The detailed IPU diagram is shown in the following figure:

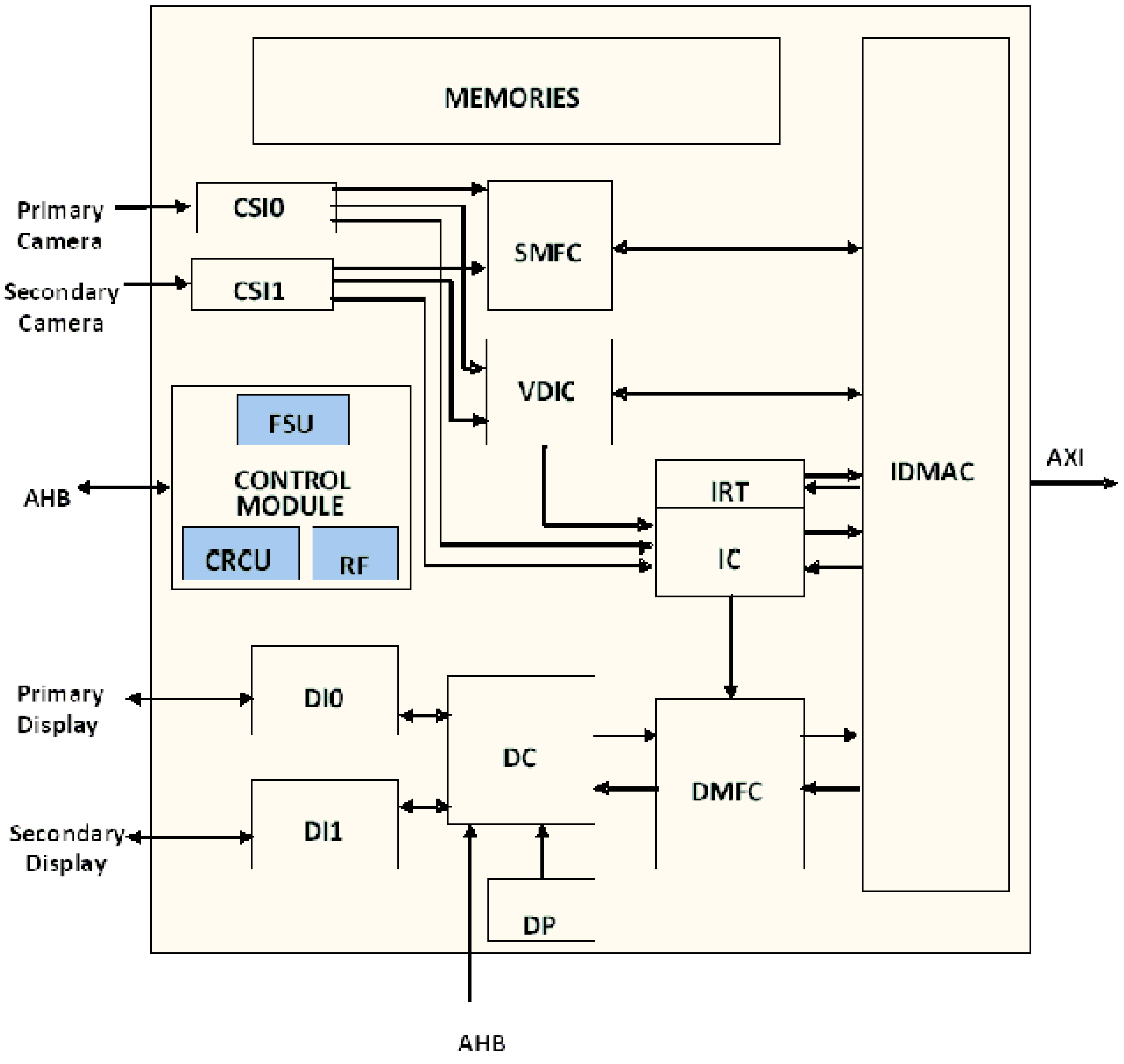


Figure 13-2. Detailed IPU block diagram

The five types of IPU tasks are listed in the following table:

Table 13-2. IPU tasks

Task	Data flow	Additional information
Image rendering	The data flow is from memory to display.	The image is provided by external devices (such as sensor, DVD player, etc.) through the CSI interface.
Image processing	The data flow is from memory to memory.	The image is provided by external devices (such as sensor, DVD player, etc.) through the CSI interface.

Table continues on the next page...

Table 13-2. IPU tasks (continued)

Task	Data flow	Additional information
CSI preview	The data flow is from CSI to display on a direct path with no memory involved.	-
CSI capture	The data flow is from CSI to memory.	-
Mixed mode	Can be a combination of two or more of the above tasks.	-

13.3 Image rendering

Image rendering is the process by which the image data stored in the memory is transferred to the display device. The following are examples of supported display devices:

- Parallel dumb panel
- Smart panel
- Other further processed sinks:
    - HDMI/DVI monitor (i.MX 6Dual/6Quad provides the HDMI/DVI transmitter & PHY converts the data into serialized differentiated data lanes)
    - LVDS panels (i.MX 6Dual/6Quad provides LDB as a bridge to LVDS display)

An example of a simple display flow is: memory->IDMAC->DMFC(->DP)->DC->DI->display.

13.3.1 IDMAC

IDMAC is the DMA bridge between external memory and IPU blocks. There are 64 DMA channels inside the IPU. Each channel is dedicated as a read/write channel to/from the memory. Detailed channel descriptions can be found in the IPU spec.

The configuration parameters for each IDMAC channel are held in the CPMEM. For each channel, there are two mega-words to describe the properties. Each mega-word is 160 bits wide and includes information such as data format, frame width and height, burst size, stride line, and bit per pixel setting.

IDMAC can support interleaved mode and non-interleaved mode data transfer. The IDMAC will pack (in write direction) or unpack (in read direction) the data, no matter what format it is stored in. This means all data flow through IDMAC to other blocks of IPU will be in YUVA4444 or RGBA8888 mode.

There are several IDMAC events/interrupts for system control and debug purposes. The most import of these are EOF (end of frame) and NF (new frame). These two events are usually used to indicate the frame status and drive the whole flow.

13.3.2 DMFC

The display multi-FIFO control (DMFC) manages multi-channel FIFOs. It serves the following clients:

Table 13-3. DMFC clients

Client	Access
IDMAC	Read and write
DP	Read only
DC	Read and write
IC	Write only
AHB	Read and write

The DP and the DC read channels are physically attached to an IDMAC or an IC channel. When the input is coming from the image converter, it replaces a channel that was physically attached to the IDMAC because the image converter has only one output channel connected to the DMFC. The DMFC uses a single physical memory that serves the DP and DC read channels. The AHB accesses to the DC, and the DC's write channel (read from display), use a separate physical memory. This is used to write an external device directly through AHB bus, or to configure a smart panel.

In image rendering, DMFC is served as FIFO between either IDMAC (fetching data from external memory) or IPU subblocks (such as IC, DP, DC). The physical memory of DMFC is partitioned into eight segments. For each channel, the start address at a segment's boundary must be defined using the DMFC_ST_ADDR parameter, and the size of the FIFO allocated to a channel must be defined using the DMFC_FIFO_SIZE parameter. The FIFO must be allocated to avoid overlapping between FIFOs. The FIFO's burst length is also configurable, and it should match the IDMAC burst length for optimal performance.

There is a watermark setting to dynamically tune the channel's priority on the IDMAC's arbitration. DMFC_WM_SET and DMFC_WM_CLR are used to trigger the watermark signals.

13.3.3 Display Processor (DP)

Each IPU can support two synchronous display flows concurrently. One is through the display processor BG/FG, and the other is through the display controller.

The display processor processes the image prior to sending it to the display. The main task performed by the display processor is combining between full and partial planes. The display processor has two input FIFOs holding the data of the full plane and the partial plane. The two planes can be blended as per local or global alpha setting, based on the mode chosen by DP_GWAM_SYNC. For global alpha, the alpha value is configured in DP register DP_GWAV_SYNC. In addition, the display processor performs some image enhancement functions like gamma correction and color space conversion (including Gamut mapping).

During combining, the background is a full plane, and the foreground is a partial plane. Left and top offsets of the foreground can be set in register DP_FG_POS_SYNC. The size of the foreground is determined by the corresponding IDMAC descriptor.

The following figure shows the display processor architecture diagram:

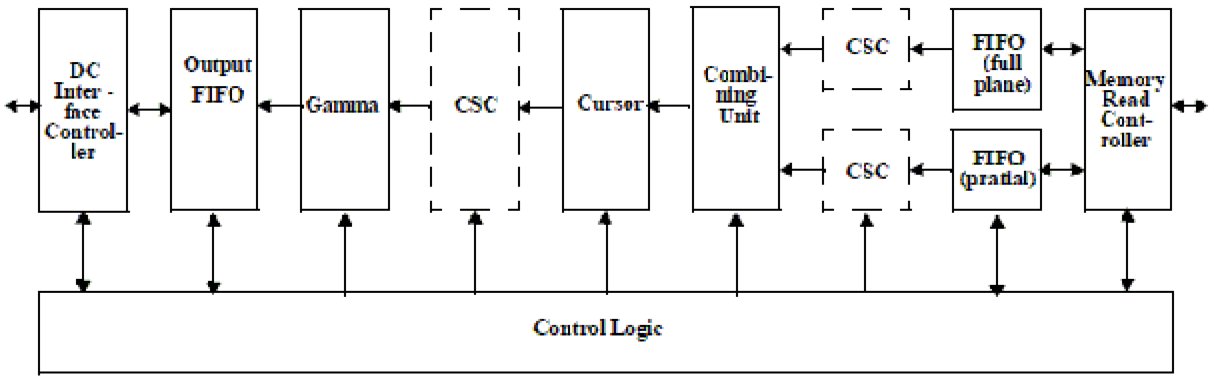


Figure 13-3. Display processor architecture diagram

The combination task can also be done in the image converter, but in that case the size of the two planes must be the same. The display processor is the first choice for two-layer blending because its combining performance is higher than the image converter's combining performance.

Note that the DP register cannot be accessed directly. For example, the shadow register SRM_DP_COM_CONF_SYNC must be accessed in order to configure the DP_COM_CONF_SYNC register. The DP_S_SRM_MODE setting indicates how the changes in shadow registers are updated in the actual registers.

13.3.4 Display controller (DC)

The display controller controls the flows coming to and from the DI port. The display controller manages the flows, decides which flows are currently active and when each flow is activated. The display controller arbitrates between the active flows, gets the data from the predefined source and distributes it to the correct DI.

The display controller's core is the microcode. The microcode contains a set of routine which is built of a set of commands stored in the template's (microcode) memory. For each event (such as new frame, end of frame etc.) a specific routine is executed. Users write the routines to describe the rules of processing, and then map them to specific events. The routine contains instructions for the display controller about how to handle the data/address/commands associated with the display. The routine can also contain information about data mapping, waveform characteristics, and more.

In the display controller block, the data coming from IDMAC is linked to a display interface. It also sets the interface format (parallel or serial, interlaced or progressive, etc.), to which the display flow is attached, and maps the data to the sink device (based on which waveform of the display interface data will be processed). The rendered image data is then sent to the display interface.

13.3.5 Display interface (DI)

The display interface provides access to up to three displays using time multiplexing. It converts data from the display controller, or the MCU (low level access for serial interface only), to a format suitable for the specific display interface. The display interface generates display clocks and other display control signals such as HSYNC, VSYNC and DRDY with programmable timings. It also outputs data to, or inputs data from, parallel and/or serial interfaces.

This module generates all the control signals sent to the display. The display controller sends the data for the display and a set of control signals to the display interface. The controls coming from the display controller are used to generate the control signals sent to the display through the display interface. One exception is serial low-level access (LLA), meaning the display controller is bypassed and the data comes directly from the MCU.

The display interface also sets the attributes of the interfaces to the display. The timing and polarity of signals are set in the display interface block according to the different types of displays.

13.4 Image processing

Image processing performs resizing, rotation, color space conversion, multi-layer combination with alpha blending, de-interlaced, gamma correction, gamut mapping, etc. The main image processing blocks are the IC (image converter) and VDIC (video de-interlaced and converter).

The image converter contains three processing sections: downsizing, main processing and rotation. The peripheral bus registers control this module. Some processing parameters are written by the MCU to the Task Parameter Memory, and the AHB bus performers writing to the memory.

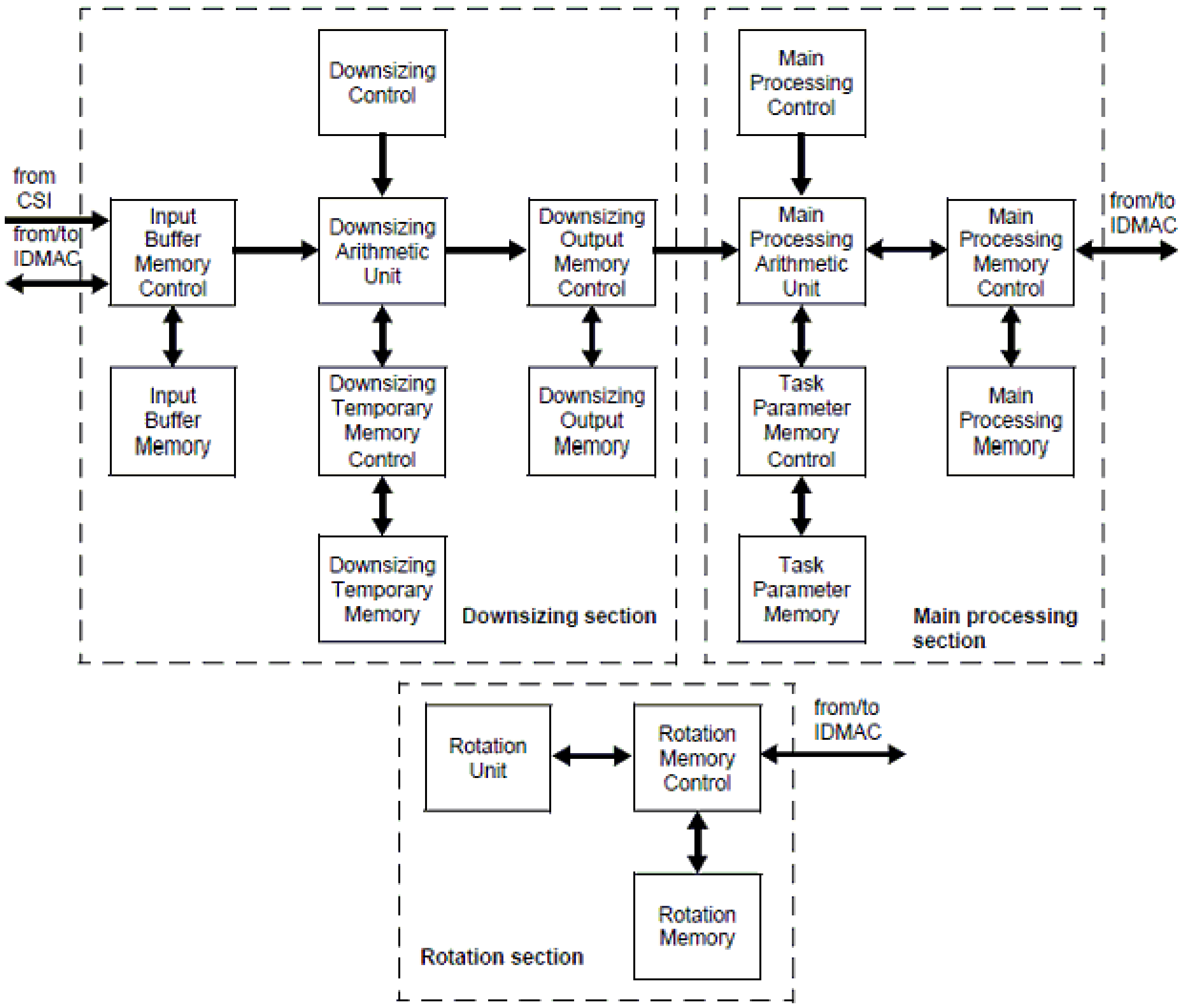


Figure 13-4. Image converter diagram

The image converter has three processing sections that can perform up to three processing tasks with time sharing mode. This means that three sets of configurations can be set at the same time, but they share the unique set of hardware accelerators. For post processing tasks, it has a dedicated input and output channel. For preprocessing tasks, encoder and viewfinder share the same input but have their own separate output channels.

13.4.1 Downsizing

In this block, the image converter performs 1x, 2x, 4x downsizing operations on the input image. The downsizing ratio can be set in DS_R_H for horizontal or DS_R_V for vertical.

13.4.2 Main processing

The main processing block reads the data from the Downsizing output and is able to perform the following operations in each task:

- Horizontal/Vertical Flip by HF/VF settings-The corresponding DMA channel descriptor should be changed accordingly.
- Horizontal/Vertical Resizing by bilinear interpolation-There is a formula to calculate the resizing ration. Resizing_ratio = floor(2^13 x (SI - 1) ÷ (SO - 1)), where SI means the input size and SO means the output size. In the Resizing block, the output should be no more than 1024 in horizontal due to the FIFO width limitation.
- Color Space Conversion-The conversion matrix is user configurable, and it can support SAT_MODE and NON_ SAT_MODE. In SAT_MODE, the range of Y is [0, 235], range of U/V is [16, 240]. In NON_SAT_MODE, the range of Y/U/V are all [0, 255].
- Combination-The image converter can support local alpha blending, global alpha blending, and use of key color. The size of the two layers for combining must be the same.

13.4.3 Rotation

The image converter and IDMAC work together to perform rotation. The image for rotation is divided into 8 x 8 blocks. The IDMAC must work in block mode and perform data rearrangement within the blocks. The image converter provides the proper rotation of the whole frame in the block unit.

The VDIC can de-interlace standard interlaced video to progressive video that is used for upsizing to HD formats or for display on progressive displays. For VDI operation, three sequential fields are necessary: F(n - 1), F(n), and F(n + 1). There is a per-designed de-interlace algorithm stored in the VDIC block as firmware. The de-interlace is performed by setting the motion level (high-motion or low-motion), and it outputs a progressive whole frame.

The VDIC can also perform on-the-fly combination and color keying. The position and size of the foreground layer are configurable.

13.5 CSI Preview

Image preview is a direct path from CSI to display. The CSI gets data from the sensor, synchronizes the data and the control signals to the IPU clock (HSP_CLK), and transfer it according to configuration of DATA_DEST register to one or more of the following: IC or SMFC. When data is transferred to the IC module then routed to display module, it is called image preview.

13.5.1 CSI interfaces

CSI supports two types of interfaces: parallel interface and high-speed serial interface. The interface is determined via the DATA_SOURCE register.

13.5.1.1 Parallel interface

In this mode, a single value arrives in each clock except when working in BT.1120 mode, in which case two values arrive in each cycle. Each value can be 8-16 bits wide according to the configuration of DATA_WIDTH. If DATA_WIDTH is configured to N, then 20-N LSB bits are ignored.

CSI can work with several data formats according to SENS_DATA_FORMAT configuration. In case the data format is YUV, the output of the CSI is always YUV444 (even if the data arrives in YUV422 format).

The polarity of the inputs can be configured using the registers SENS_PIX_CLK_POL, DATA_POL, HSYNC_POL, and VSYNC_POL.

13.5.1.2 High-speed serial interface-MIPI (mobile industry processor interface)

In MIPI interface, two values arrive in each cycle. Each value is 8 bit wide, meaning 16 MSB bits of the data bus input are treated, while 4 LSB bits are ignored.

When working in this mode, the CSI can handle up to four streams of data. Each stream is identified with DI (data identifier), including the virtual channel and the data type of this stream. Each stream that is handled is defined in registers MIPI_DI0-3. Only the main stream (MIPI_DI0) can be sent to all destination units, while the other streams are sent only to the SMFC as generic data.

In this mode SENS_DATA_FORMAT and DATA_WIDTH registers are ignored, since this information is coming to the CSI via the MCT_DI bus.

13.5.2 CSI modes

CSI can work in several timing/data mode protocols according to SENS_PRTCL configuration.

13.5.2.1 Gated mode

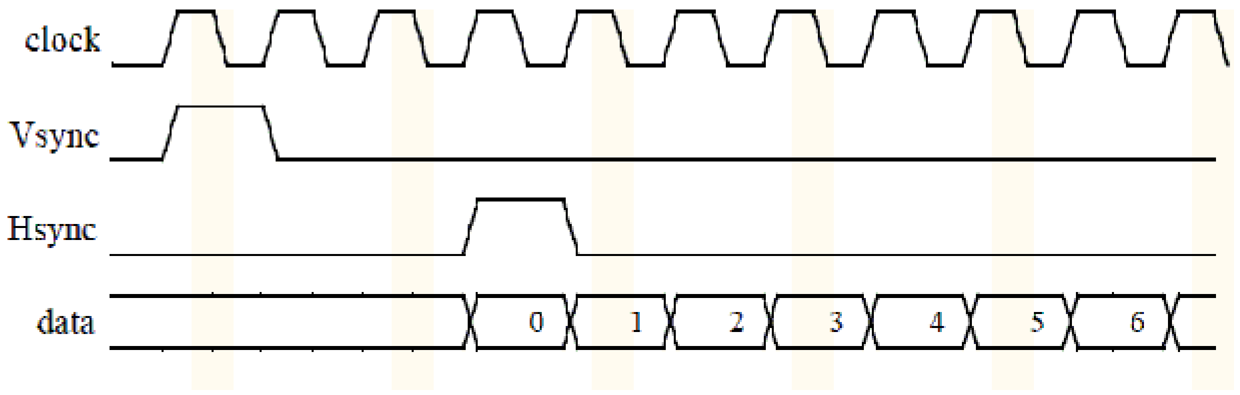


Figure 13-5. CSI gated mode

In this mode, VSYNC is used to indicate the beginning of a frame, and HSYNC is used to indicate the beginning of a raw. The sensor clock is ticking all the time.

13.5.2.2 Non-gated mode

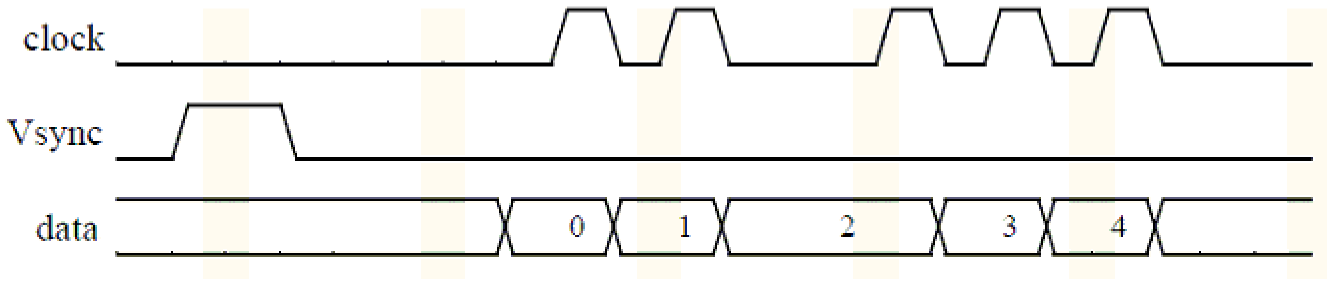


Figure 13-6. CSI non-gated mode

In this mode, VSYNC is used to indicate the beginning of a frame. The sensor clock only ticks when data is valid. HSYNC is not used.

When working with MIPI, configure the non-gated mode.

13.5.2.3 BT656 mode

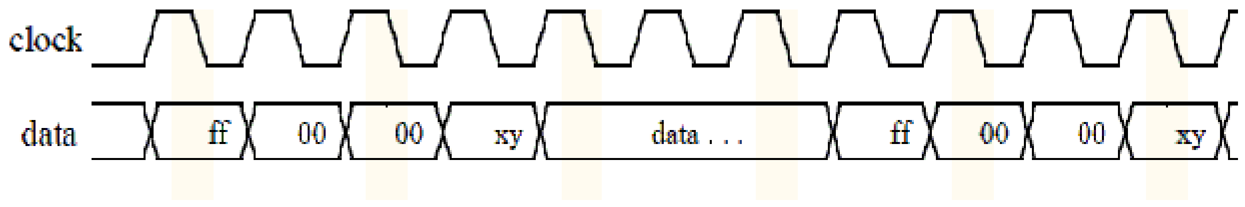


Figure 13-7. BT656 mode

BT656 describes a simple digital video protocol for streaming uncompressed PAL or NTSC Standard Definition TV (525 or 625 lines) signals. The protocol builds upon the 4:2:2 digital video encoding parameters which provide interlaced video data (streaming each field separately). It uses the YCbCr color space and a 13.5 MHz sampling frequency for pixels.

The timing reference signals (frame start, frame end, line start, line end) are embedded in the data bus input, and each timing reference signal consists of a four word sequence. The first three words are fixed and are configured in the CCIR_PRECOM register. The fourth word contains information defining the field, the state of field blanking, and the state of line blanking. These states are configured in registers CCIR_CODE_1 (for field 0) and CCIR_CODE_2 (for field 1).

For PAL mode, the CCIR_CODE can be configured as shown below:

- CCIR_CODE_1: D 07DFh
- CCIR_CODE_2: 4 0596h
- CCIR_CODE_3: FF 0000h

One value of data arrives in each cycle of the BT656 mode.

13.5.2.4 BT1120 mode

In this mode, CSI can work in SDR or DDR mode.

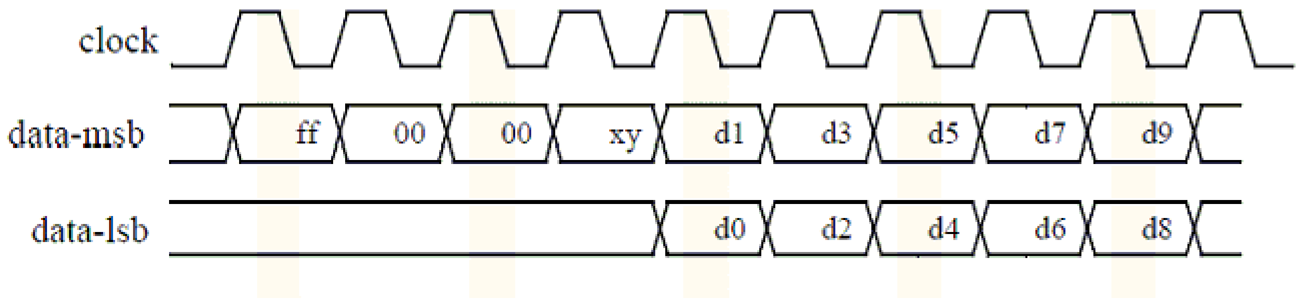


Figure 13-8. BT1120 SDR mode

In DDR mode, data will arrive on both rising and falling edge of a clock, meaning that two values of data arrive in each clock.

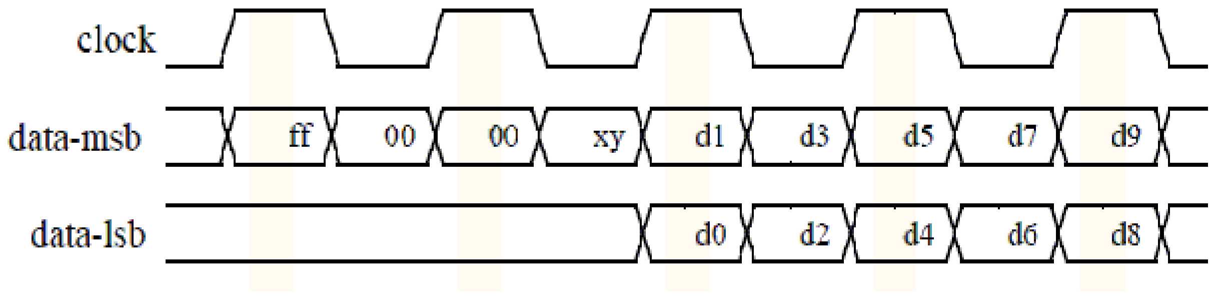


Figure 13-9. BT1120 DDR Mode

For direct path from CSI to the image converter, CSI_MEM_WR_EN and RWS_EN (located in IPU_IC_CONF) are used to choose the data flow. CSI_SEL (in IPU_CONF) determines which CSI is selected as the direct input to IC module. There is a limitation in this task: the refresh rate of the display device must be the same as the CSI input frame rate, otherwise the screen may not be functional due to the frame rate mismatch.

In the CSI block, images can be cropped by setting the actual window size. Follow these rules:

- SENS_FRM_HEIGHT ≥ VSC + ACT_FRM_HEIGHT
- SENS_FRM_WIDTH ≥ HSC + ACT_FRM_WIDTH

13.6 CSI capture

In CSI capture task, data is received from the sensor and output to the memory through SMFC and IDMAC.

The SMFC (Sensor Multi FIFO Controller) is used as a buffer between CSI and IDMAC. Both masters (CSIs) can be connected to SMFC and both can be active simultaneously.

There are four channels that can be used as CSI output channels: channels 0~3 (of the IPU DMA channels). The frame from CSI can be mapped to one of four IDMAC channels via SMFC mapping registers. Each DMA channel has a dedicated FIFO, and the burst length of the FIFO must match the DMA settings. The FIFO size attached to each DMA channel is flexible according to the number of channels required. All four channels share the whole FIFO, and if only one of them is enabled, the entire FIFO can be allocated to one channel.

13.7 Mixed task

The mixed task can be a collection of the tasks described in the sections above. For example, a CSI captured image can be stored in memory and then resized to full screen for display.

For a complex task, CM is used for the flow management in order to support automatic control without using the CPU. After the different blocks are connected together by CM, the flow will be auto-driven by internal events (such as NF, EOF, etc.).

CM configures five registers:

- IPU_FS_PROC_FLOW1
- IPU_FS_PROC_FLOW2
- IPU_FS_PROC_FLOW3
- IPU_FS_DISP_FLOW1
- IPU_FS_DISP_FLOW2

The first three set the processing tasks and the last two set the display flows. For each task, the source and destination must be configured to form a round linkage between blocks.

13.8 Clocks

The following table lists the IPU clock sources:

Table 13-4. IPU clock sources

Clock	Name	Description
High-speed processing clock	HSP_CLK	Source from the clock control module
Display interface clocks	• DI_CLK0 • DI_CLK1	Source from the clock control module or an external PLL These clocks are optional; they are needed for synchronization with interface bridges.

13.8.1 High-speed processing clock (HSP_CLK)

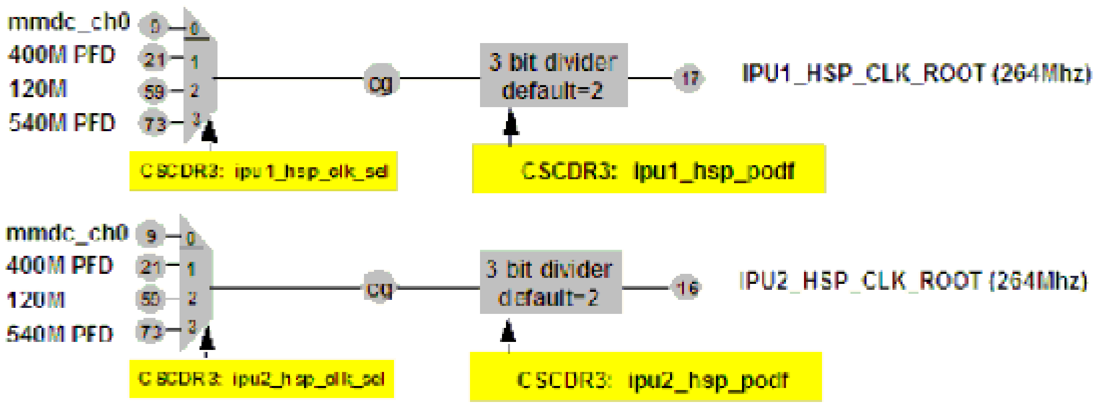


Figure 13-10. IPU HSP_CLK clock tree

The IPU main clock (HSP_CLK) is generated by the internal clock control module (CCM) (see Figure 13-10). The default HSP_CLK is divided from mmdc_ch0 by 2.

13.8.2 Display interface clocks (DI_CLK*n*)

The IPU display interface clock can be generated by either an internal clock divider or an external PLL. For example, to drive a display of XGA resolution, we need a 65 MHz pixel clock. There are two ways to obtain the clock.

- Divide from the internal IPU clock (HSP_CLK)

- Dividing the 264 MHz IPU HSP_CLK clock by 4 provides the 65 MHz pixel clock. IPU can also support fractional division, but image rendering does not usually require that precise of a clock. Clear DI_CLK_EXT to set the DI clock source to internal.
- From external PLLs
- The following figure shows the clock tree for generating IPU_DI0 clock from an external source to the IPU source (HSP_CLK). The external source is selected with the ipp_di_#_ext_clk_pin. The clock tree only works when the DI_CLK_EXT is set, which means the clock is generated externally.

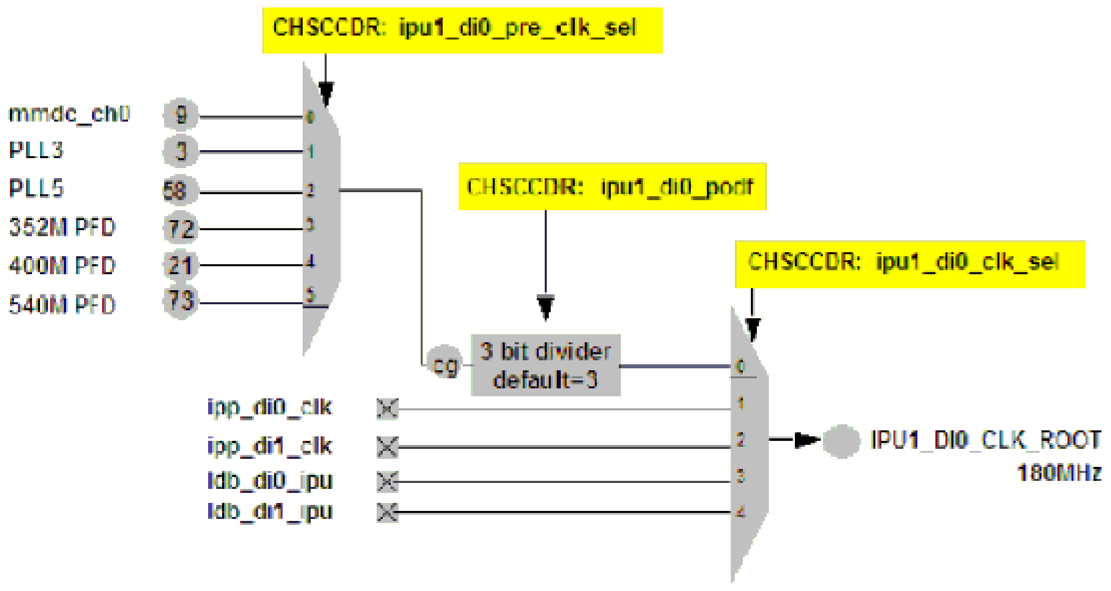


Figure 13-11. IPU DI0 clock tree

13.9 IOMUX pin mapping

PU has two sets of display interfaces. For a parallel display, IPU provides data lanes, vsync, hsync, date ready and pixel clock to drive the panel. For other further-processed displays, such as HDMI or LVDS, the IPU output signals are internally multiplexed to the relative modules.

For example, to connect the LVDS with IPU, LVDS*x*_MUX_CTL can be configured as shown below:

- 00 - IPU1 DI0, connect LVDS*x* to IPU1 DI0. The "x" means 0 or 1, and there are two sets of LVDS display interfaces.
- 01 - IPU1 DI1

- 10 - IPU2 DI0
- 11 - IPU2 DI1

To connect HDMI with IPU, HDMI_MUXCTRL can be configured as shown below:

- 00 - IPU1 DI0, connect HDMI to IPU1 DI0
- 01 - IPU1 DI1
- 10 - IPU2 DI0
- 11 - IPU2 DI1

For parallel displays, set the output signals of IPU according to the schematic and the chip data sheet. General IPU display waveform pins provide the sync signals. They must match with the waveform settings in the DI block.

The following table shows a typical IOMUX mapping for an IPU parallel panel through DI0. The exact mapping is board dependant.

Table 13-5. Typical IOMUX mapping for IPU parallel panel through DI0

Signals	Option 1	
	PAD	MUX
DI0 display clock	DI0_DISP_CLK	ALT0
DRDY	DI0_PIN15	ALT0
HSYNC	DI0_PIN2	ALT0
VSYNC	DI0_PIN3	ALT0
DI0 data0~23	DISP0_DATx	ALT0

13.10 Use cases

This section describes how to program I2C controller registers I2CR, I2SR, and I2DR for transferring data on the I2C bus. Pseudocode is provided wherever necessary.

13.10.1 Single image rendering example

Image rendering (image display) is the basic use case. This example provides a general introduction to how the IPU is configured to show an RGB image on the screen.

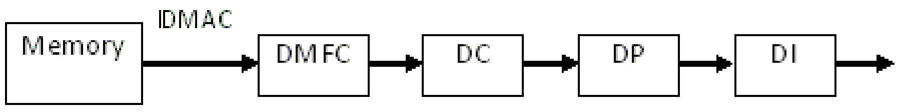


Figure 13-12. IPU process for single image rendering

As described in Image rendering, several blocks are involved in the display flow. Before setting the hardware registers, ensure you know all input and output information.

This example uses memory-to-display for the flow type and chooses the DP BG path. Hardware configuration includes the following steps:

1. Configuring the IPU DMA channel (single image rendering)
2. Allocating the DMFC block
3. Configuring the DP block
4. Configuring the DC block
5. Configuring the DI block
6. Enabling the blocks involved in the display flow

13.10.1.1 Configuring the IPU DMA channel (single image rendering)

In this step, API **ipu_disp_bg_idmac_config ()** is called. Because we chose the DP BG path for display, we must use channel 23 as the DMA channel to fetch data from memory.

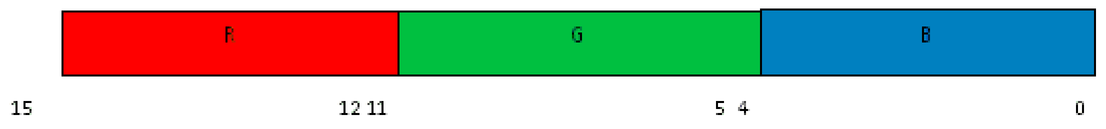
The input data format is interleaved RGB565, so the relative bit fields must be set as:

- Bpp = 0x3, which means bit per pixel is 16
- Pfs, which indicates the data format to be interleaved RGB mode.
- Wid0=5-1, off0=0;

Wid1=6-1, off1=5

Wid2=5-1, off2=11

Wid3=0, off3=16



Wid is the actual width of the component subtracting 1. Off means the start address of the component within the pixel.

- FW, which is the actual frame width - 1
- FH , which is the actual frame width - 1
- Stride line, which means the offset of the next line in bytes

The IPU DMA channel can support single buffer mode or double buffer mode by setting the MOD_SEL bit of each channel. In double buffer mode, the channel alternately fetches data from EBA0 and EBA1.

13.10.1.2 Allocating the DMFC block

The DMFC is allocated for channel 23. The FIFO is equally split for the DP and DC synchronous display channel.

13.10.1.3 Configuring the DP block

DP is the data processor for image combination, color space conversion, gamma correction, and gamut mapping. This example uses one layer, with the inputs and outputs all in RGB mode. Therefore, the data flow through the DP is bypassed with no additional processing.

13.10.1.4 Configuring the DC block

This block calls the API ipu_dc_config() and creates the following three microcodes: new data, new line, and end of line. These three events are synchronized with the DI waveform that generates the active data by setting the sync field of the microcode.

The mapping unit in DC block is used to pack the data output from DC to DI and then to the data format that the display device supports. For example, if the display can accept RGB666 mode, the RGBA8888 data flow must be packed into RGB666 format. This operation is done in ipu_dc_map(). The mapping bit field of the microcode determines which of the three available sets of data mapping units is chosen.

NOTE

As described in the IDMAC section, all data flow through the subblocks of IPU (YUVA4444 or RGBA8888) is unpacked by the IDMAC block.

Finally, ipu_dc_microcode_config() writes the microcode into a space in template memory, and ipu_dc_microcode_event() attaches it to the event. The event priority can be set individually.

The DC block also provides connection information between DI and DC. Both ipu_dc_display_config() and ipu_dc_write_channel_config() can determine which DI the DC is connected to, which format the display interface is in, what the data width is, and which port the display has selected.

13.10.1.5 Configuring the DI block

This block is the interface to the display panels or other display processing modules. The timing to display is generated by the general waveform sets inside the DI block.

For a parallel panel, IPU needs to provide pixel clock, HSYNC, VSYNC, DRDY, and data lines. The pixel clock can be generated internally or externally. In external mode, the pixel clock is always equal to the di_clk_root shown in Figure 13-12.

Each waveform generator requires several parameters to generate a proper signal, as described in the following table.

Table 13-6. Signal parameters

Signal parameter	Description
`syncWaveformGen.runValue`	Indicates the number of periods based on the reference clock
`syncWaveformGen.runResolution`	Indicates the reference clock for the waveform generator. It will trigger the counter to decrease.
`syncWaveformGen.offsetValue`	Indicates the predefined offset in the unit of offsetResolution
`syncWaveformGen.offsetResolution`	Indicates the offset reference clock.
`syncWaveformGen.cntAutoReload`	in auto-reload mode, the counter will reload the predefined value(runValue) when the counter decrease to zero
`syncWaveformGen.stepRepeat`	Valid only in non auto-reload mode. The counter will reload the predefined value (runValue) when the counter decrease to zero
`syncWaveformGen.cntClrSel`	Source to clear the non auto-reload waveform counter
`syncWaveformGen.cntPolarityGenEn = 0`	Used to clear/set the polarity of the waveform
`syncWaveformGen.cntPolarityTrigSel = 0`	Used to clear/set the polarity of the waveform
`syncWaveformGen.cntPolarityClrSel = 0`	Used to clear/set the polarity of the waveform
`syncWaveformGen.cntUp = 0`	Indicates the rising edge of the waveform
`syncWaveformGen.cntDown = 2`	Indicates the falling edge of the waveform

The following figures show how to set the key parameters to specify the timing of the display.

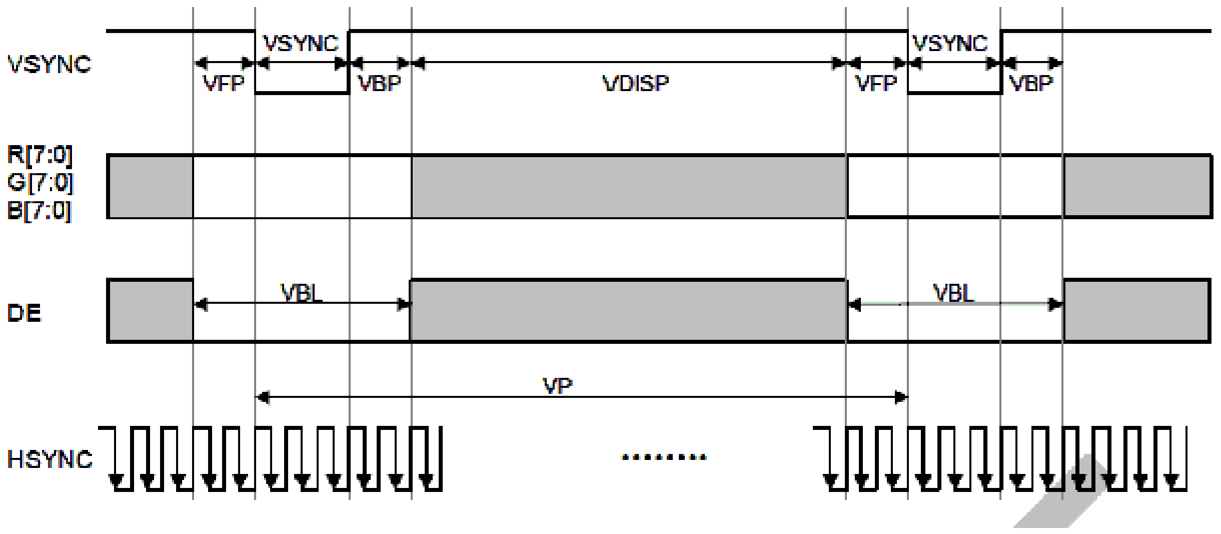


Figure 13-13. Vertical display time

Vertical time has a blanking time, VBL, between two vsync active periods. VBL can be divided into three parts:

- VFP, vertical front porch
- VBP, vertical back porch
- VSYNC, sync width in vertical

In the code, vSyncStartWidth indicates the start width of blanking in a whole vsync period, and vSyncEndWidth indicates the end width of blanking in a whole vsync period.

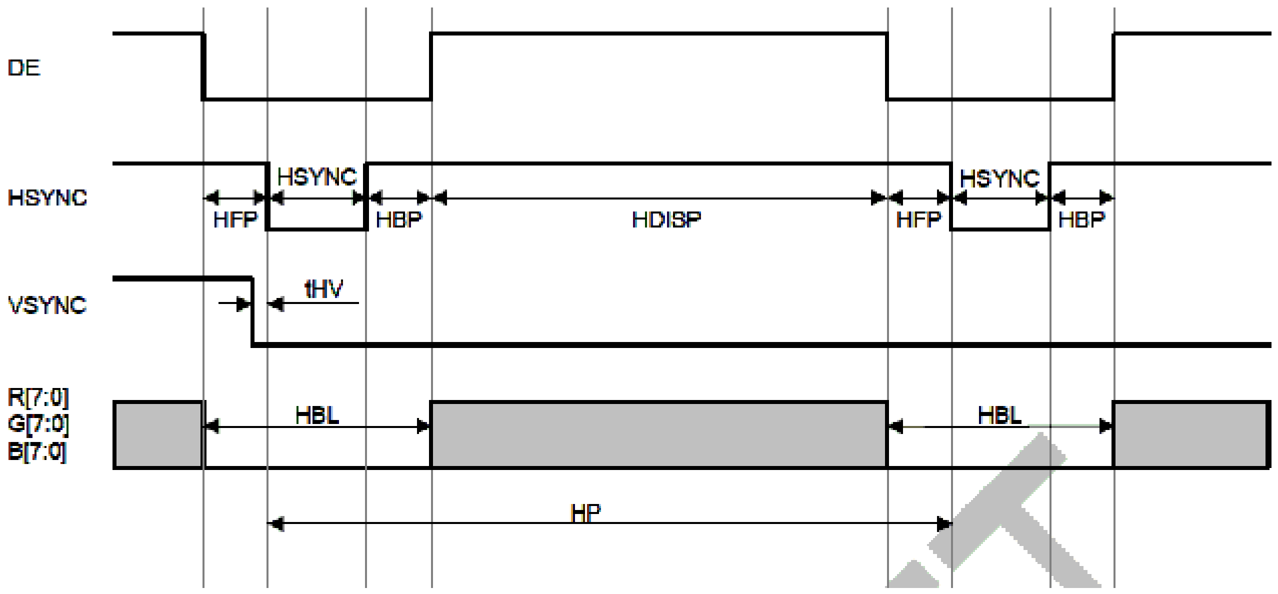


Figure 13-14. Horizontal display timing

Horizontal timing has blanking time, HBL, between two hsync active periods. Like VBL, HBL can be divided into three parts:

- HFP, horizontal front porch
- HBP, horizontal back porch
- HSYNC, sync width in horizontal

In the code, hSyncStartWidth indicates the start width of blanking in a whole hsync period, and hSyncEndWidth means the end width of blanking.

DE (or DRDY) has the same frequency as HSYNC, but the active period of DE indicates data lines that are active in that period.

Based on the timing diagram, the parameters are configured as:

- hSyncStartWidth = HSYNC + HBP;
- hSyncWidth = HSYNC;
- hSyncEndWidth = HFP;
- delayH2V = tHV;
- vSyncStartWidth = VSYNC + VBP;
- vSyncWidth = VSYNC;
- vSyncEndWidth = VFP;
- hDisp = HDISP;
- vDisp = VDISP;

The frame width and height of the screen are indicated by hDisp and vDisp.

All the waveforms in the DI block are for general usage. Some are used for internal logic, and some are used as output signals. The output pins are determined by the schematic design, and DI must bind the pins to the output by setting VSYNC_SEL and DISP_Y_SEL in the ipu_di_interface_set() function. The polarity of each output signal can also be configured in the ipu_di_interface_set().

13.10.1.6 Enabling the blocks involved in the display flow

This is the last step of hardware settings. The display flow requires the following blocks:

- IDMAC
- DMFC
- DP
- DC
- DI

All these subblocks can be selected in the IPU_CONF register.

13.10.2 Image combining example

The image combining use case illustrates combining between the full and partial planes. Each one of the planes may be a graphic or video plane. The following figure shows two planes displayed on a display.

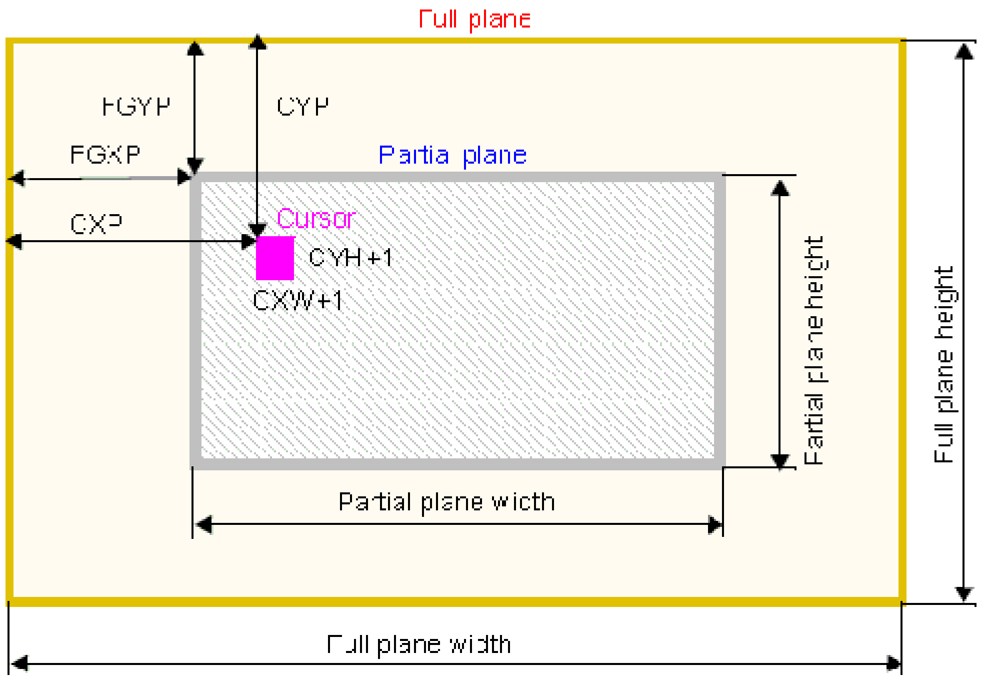


Figure 13-15. Display planes

The partial plane's position is defined relatively to the upper left corner of the full plane. The size of the partial and full planes is defined on the corresponding IDMAC's channels' FW and FH parameters. The cursor position and parameters are set in the DP_CUR_POS register.

The following figure shows the IPU process for displaying two combined images to screen from two separated memories.

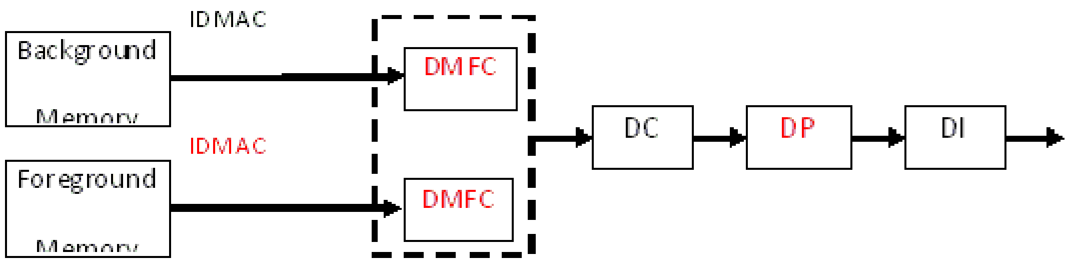


Figure 13-16. IPU process for image combining

The background image is sent to its DMFC through IDMAC main plane channel. The foreground image is send to its DMFC through IDMC auxiliary plane. The combining options are set in DP module.

Hardware configuration includes the following steps:

1. Configuring the IPU DMA channel
2. Allocating the DMFC
3. Configuring the DP module
4. Other modules

Compared to the single image rendering (Figure 13-12), IPU hardware configuration is different in the IDMAC, DMFC and DP modules.

13.10.2.1 Configuring the IPU DMA channel

The following table lists the IDMAC channels from memory to display for the main plane and auxiliary plane list.

Table 13-7. Channels for main and auxiliary planes

Channel#	Source	Destination	Purpose	Data type
23	Fmem	DP	DP primary flow-main plane	Pixel
27	Fmem	DP	DP primary flow-auxiliary plane	Pixel
31	Fmem	DP	Transparency (alpha for channel 27)	Generic
24	Fmem	DP	DP secondary flow-main plane	Pixel
29	Fmem	DP	DP secondary flow-auxiliary plane	Pixel
33	Fmem	DP	Transparency (alpha for channel 29)	Generic

This use case calls API **ipu_disp_bg_idmac_config()** to configure channel #23 for main plane and **ipu_disp_fg_idmac_config()** to configure channel #27 for auxiliary plane. This use case uses global alpha. If using local alpha, channel #31 should also be configured.

Please refer to Configuring the IPU DMA channel (single image rendering) for the relative bit fields' setting for each channel.

13.10.2.2 Allocating the DMFC

Allocate DMFC for both main plane (background) and auxiliary plane (foreground) IDMA channels.

13.10.2.3 Configuring the DP module

The DP module can set the following combining options:

- Local alpha blending
- Global alpha blending
- Use of key color
- Order of the planes (full is presented over the partial plane and vice versa)

The relative bit fields for combining are:

- DP_FGXP_SYNC / DP_FGYP_SYNC set the left upper corner position for foreground on display on screen.
- DP_FG_EN_SYNC must be set 1 to enable the partial plane channel.
- DP_GWAM_SYNC selects the use of alpha to be global or local.
- 1 Global Alpha.
- 0 Local Alpha.
- DP_GWAV_SYNC defines the global alpha value of background (main plane).

13.10.2.4 Other modules

The settings are the same as those for corresponding modules stated in Configuring the DP block, Configuring the DC block, and Configuring the DI block.

13.10.3 Image rotate example

The following figure shows the IPU process for rotating an image and displaying it on a screen.

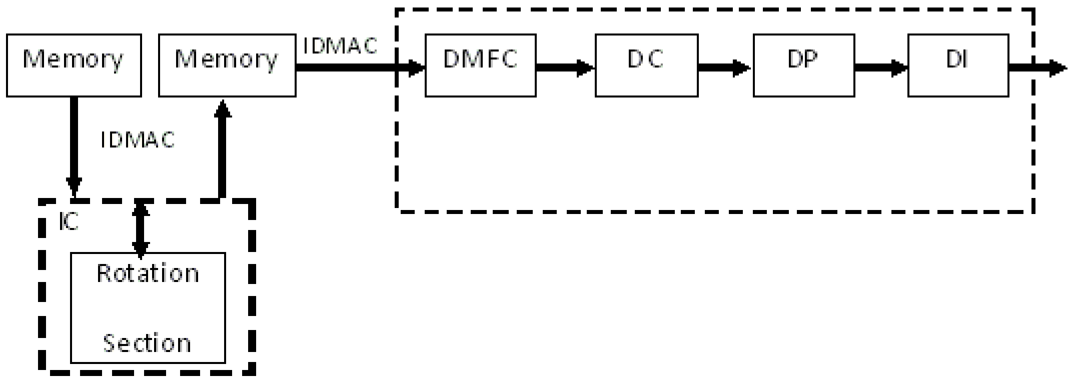


Figure 13-17. IPU process for image rotation

Rotation is performed by the IDMAC and the rotation unit inside the image converter. The frame is partitioned into 8 x 8 pixels blocks.

1. The image converter reorders the pixels within a block. The rotation unit rewrites pixels from the input block to the output FIFO with corresponding relocation of a pixel inside the block.
2. The IDMAC reorders the block according to the VF, HF, and ROT parameters of the corresponding DMA channels.

IPU configuration includes the following steps:

1. Configuring IDMAC channels for IC tasks (IC rotate)
2. Configuring the IC task
3. Setting IDMAC buffer ready
4. Image rendering process (IDMAC)

13.10.3.1 Configuring IDMAC channels for IC tasks (IC rotate)

The following table lists the IDMAC channels for the IC rotate tasks.

Table 13-8. Rotation channels

Channel #	IC channel name	R/W	Source	Destination	Purpose
45	CB10	Read	Memory	ENC ROT	Preprocessing data for rotation (encoding task)
48	CB8	Write	ENC ROT	Memory	Preprocessing data after rotation (encoding task)
46	CB11	Read	Memory	VF ROT	Preprocessing data for rotation (viewfinder task)
49	CB9	Write	VF ROT	Memory	Preprocessing data after rotation (viewfinder task)
47	CB13	Read	Memory	PP ROT	Postprocessing data for rotation
50	CB12	Write	PP ROT	Memory	Postprocessing data after rotation

This use case takes input channel #47 and output channel #50 for the postprocessing task. The API calls **ipu_rotate_idmac_config()** to set the IDMAC for IC rotation tasks.

The rotation related bit fields' setting of input channel #47 are:

- NPB (Number of pixels per burst access) must be set as 7, which means 8 pixels per burst.
- ROT (Rotation) is enabled, which means 90 degree rotation clockwise.
- BM (Block Mode) is set as 01h, which means 8 x 8 pixels blocks.

The rotation related bit fields' setting of output channel #50 are:

- NPB (Number of pixels per burst access) must be set as 7, which means 8 pixels per burst.
- ROT (Rotation) is disabled. The rotation is performed in the input channel.
- HF (Horizontal Flip) is enabled depends on the use case.
- VF (Vertical Flip) is enabled depends on the use case.
- BM (Block Mode) is set as 01h, which means 8 x 8 pixels blocks.

Please refer to Image rendering for the relative bit fields' setting for each channel.

13.10.3.2 Configuring the IC task

The rotation unit rewrites pixels from the input block to the output FIFO with the corresponding relocation of a pixel inside the block. Rotation, left/right flipping, and/or up/down flipping are enabled separately. The rotation section includes:

- Rotation memory (stores an input rectangular block of 8 x 8 pixels)
- Output FIFO (contains four pages of 8 pixels)

This example uses the postprocessing task for rotate only (without left/right or up/down flip). The settings are:

- T3_ROT is enabled, which means rotation for the postprocessing task.
- T3_ FLIP_LR is enabled depending on the use case, which means the left/right flip for the postprocessing task.
- T3_FLIP_UD is enabled depending on the use case, which means the up/down flip for the postprocessing task.

NOTE

These three fields should be the same as in IDMAC.

- PP_EN is enabled, which enables the postprocessing task.
- PP_ROT_EN is enabled, which enables postprocessing rotation task.

13.10.3.3 Setting IDMAC buffer ready

Set IDMAC buffer ready after configuring and enable the IC task. Set the output IDMAC channel buffer ready first and then the input IDMAC channel buffer.

13.10.3.4 Image rendering process (IDMAC)

Please refer to Single image rendering example for the image rendering process settings.

13.10.4 Image resizing example

13.10.4.1 IPU process flow

The image resizing is performed in the image converter module. The main processing unit reads pairs of pixels from the downsizing output memory background part. The following figure shows the IPU process for resizing an image and displaying it on a screen.

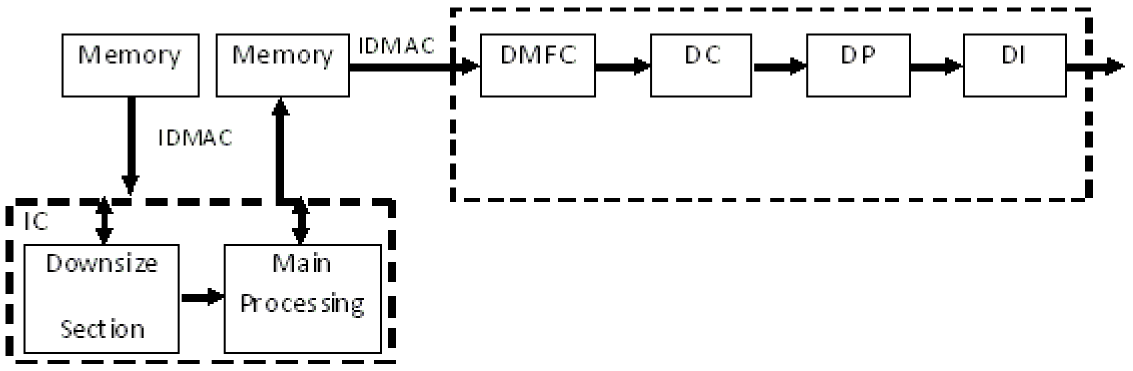


Figure 13-18. IPU process for image rotation

IPU configuration includes the following steps:

1. Configuring IDMAC channels for IC resize tasks
2. Configuring the IC resize tasks
3. Setting IDMAC buffer ready (image rotation)
4. Image rendering process

13.10.4.2 Configuring IDMAC channels for IC resize tasks

The following table lists the IDMAC channels for the IC resize tasks.

Table 13-9. Channels for resizing

Channel#	IC channel name	R/W	Source	Destination	Purpose
11	CB5	Read	Memory	IC PP	Postprocessing data from memory
22	CB2	Write	IC PP	Memory	Postprocessing data from IC to memory

Table continues on the next page...

Table 13-9. Channels for resizing (continued)

Channel#	IC channel name	R/W	Source	Destination	Purpose
12	CB6	Read	Memory	IC VF	Preprocessing data from sensor stored in memory (for example Bayer)
21	CB1	Write	IC VF	Memory/DMFC	Preprocessing data from IC (viewfinder task) to memory; This channel can be configured to send the data directly to the DMFC. This is done by programming the ic_dmfc_sel bit.
20	CB0	Write	IC ENC	Memory	Preprocessing data from IC (encoding task) to memory

This use case takes input channel #11 and output channel #22 for the postprocessing task. The API calls **ipu_resize_idmac_config()** to set the IDMAC for IC resizing tasks.

The resizing related bit fields' setting of input channel #11 are:

- NPB (Number of pixels per burst access)-determined by frame width.
- The frame width must be multiple of burst size 8 or 16 pixels as defined. If the frame width is a multiple of 16, set NPB as 16 or 8. Otherwise, NPB must be set as 8.

  The resizing related bit fields' setting of output channel #22 are:

- NPB (Number of pixels per burst access)-determined by frame width.
- The frame width must be multiple of burst size 8 or 16 pixels as defined. If the frame width is a multiple of 16, set NPB as 16 or 8. Otherwise, NPB must be set as 8.

NOTE

For both channels, the input's frame width to the image converter must be a multiple of 8 pixels.

Refer to Configuring the IPU DMA channel (single image rendering) for the other bit field settings.

13.10.4.3 Configuring the IC resize tasks

The main processing unit reads pairs of pixels from the downsizing output memory background part. The following table describes the resize task settings:

Table 13-10. Resize task settings

Setting	What it does
CB2_BURST_16	Defines the number of active cycles within a burst (burst size) coming from the IDMAC for IC's CB2 (channel #22). For pixel data, the number of pixels should match the NPB[6:2] value on the IDMAC's CPMEM.

Table continues on the next page...

Table 13-10. Resize task settings (continued)

Setting	What it does
CB5_BURST_16	Defines the number of active cycles within a burst (burst size) coming from the IDMAC for IC's CB5 (channel #11). For pixel data, the number of pixels should match the NPB[6:2] value on the IDMAC's CPMEM.
T3_FR_HEIGHT	Sets the frame height (FH) for the postprocessing task. The value of this field must be identical to the corresponding FH channel's parameters in the IDMAC's CPMEM. This parameter refers to the output's size - 1.
T3_FR_WIDTH	Sets the frame width (FW) for the post processing (PP) task. The value of this field must be identical to the corresponding FW channel's parameters in the IDMAC's CPMEM. This parameter refers to the output's size - 1.
PP_DS_R_H	Sets the postprocessing task's downsizing horizontal ratio.
PP_RS_R_H	Sets the postprocessing task's resizing horizontal ratio. Horizontal resizing is performed by bilinear interpolation between two adjacent pixels received from the downsizing output memory, according to the equation: $$HP_{R,c} = IP_{r,c} + RS\_C\_H \cdot (IP_{r+1,c} - IP_{r,c})$$ where, RS_C_H is the current horizontal resizing coefficient. The calculation result is rounded to 8 bits. The resizing coefficient is calculated by: $$RS\_C\_H = \left(\sum_{k=0}^{R-1} RS\_R\_H\right) \mathrm{mod}(8196)$$
PP_DS_R_V	Sets the postprocessing task's downsizing vertical ratio.
PP_RS_R_V	Sets the postprocessing task's resizing vertical ratio. Vertical resizing is performed by bilinear interpolation between the current and previous results of horizontal resizing. Both the current and previous results of horizontal resizing are stored in the task parameter memory. Resizing is accomplished according to the equation: $$VP_{R,C} = HP_{R,c} + RS\_C\_V \cdot (HP_{R,c+1} - HP_{R,c})$$ where, RS_C_V is the current vertical resizing coefficient. The calculation result is rounded to 8 bits. The resizing coefficient is calculated as $$RS\_C\_V = \left(\sum_{k=0}^{C-1} RS\_R\_V\right) \mathrm{mod}(8196)$$
PP_EN	Enables the postproduction task

13.10.4.4 Setting IDMAC buffer ready (image rotation)

After configuring and enabling the IC resizing task, set the IDMAC buffer to ready according to the following sequence.

1. Set the output IDMAC channel buffer ready.
2. Set the input IDMAC channel buffer ready.

13.10.4.5 Image rendering process

Refer to Configuring the IPU DMA channel (single image rendering) for the settings for the image rendering process.

13.10.5 Color space conversion example

13.10.5.1 IPU process flow (color space conversion)

The IPU contains two hardware modules that perform color space conversion (CSC): the image converter and the display processor.

The following figure shows how the image converter performs color space conversion.

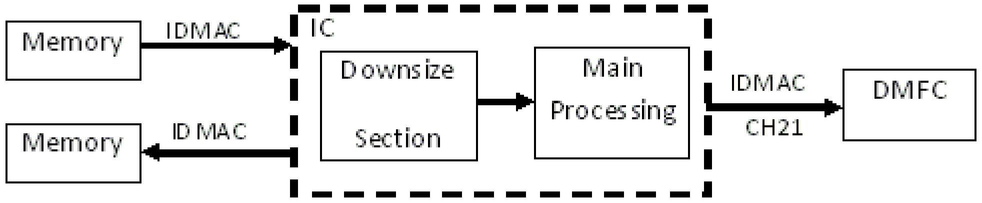


Figure 13-19. IPU process for color space conversion (IC module)

The following figure shows how the display processor performs color space conversion. The display processor connects to the display interface, so this color space conversion process is used when color space conversion is needed in the display.

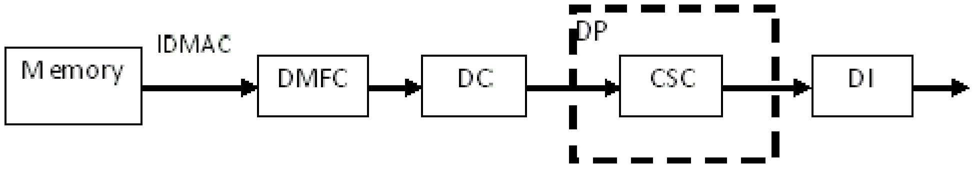


Figure 13-20. IPU process for CSC (DP module)

Color space conversion is performed in the main processing unit inside the image converter. See these sections for further details:

1. Configuring IDMAC channels for IC tasks
2. Configuring IC tasks
3. IPU configurations for the DP task

13.10.5.2 Configuring IDMAC channels for IC tasks

Refer to Configuring IDMAC channels for IC resize tasks for the IDMA channel configuration. This use case uses input channel #11 and output channel #22 for the postprocessing task.

13.10.5.3 Configuring IC tasks

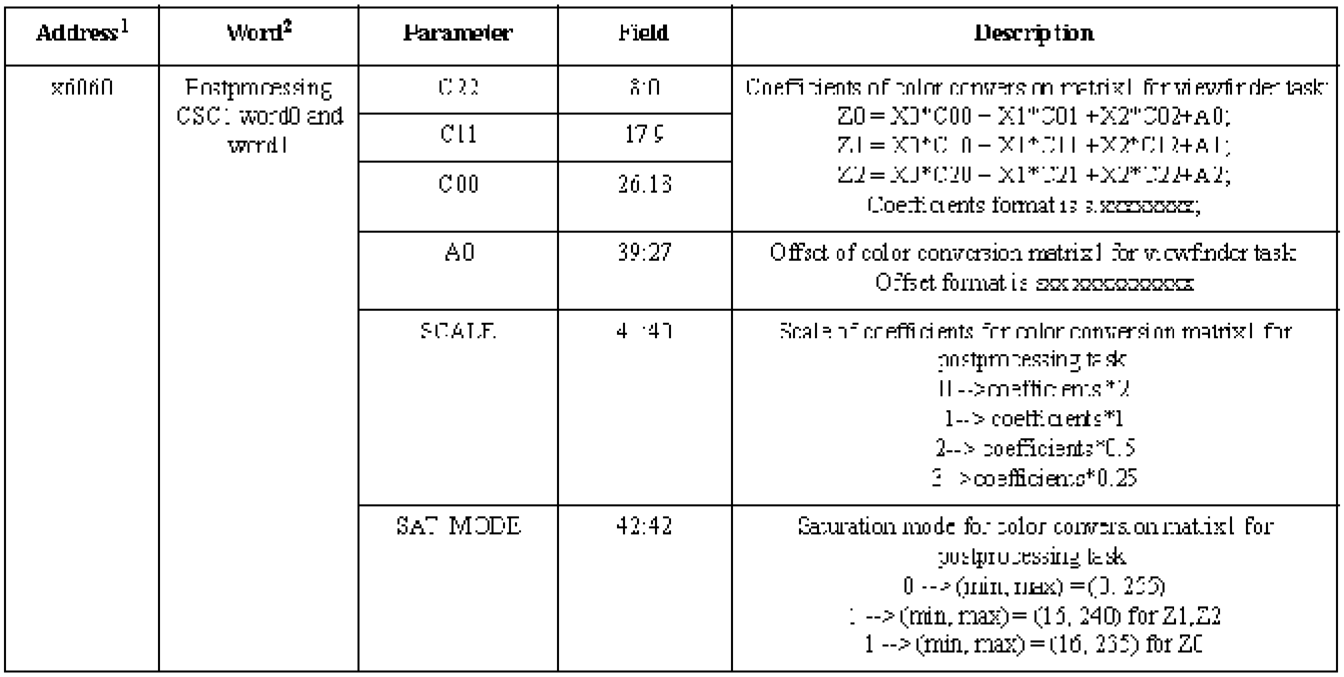
Use the conversion matrix CSC1 to perform color space conversion YUV to RGB or RGB to YUV. The conversion matrix coefficients are programmable and stored in the task parameter memory.

The conversion equations are:

$$Z_0 = 2^{SCALE-1} \cdot (X_0 \cdot C_{00} + X_1 \cdot C_{01} + X_2 \cdot C_{02} + A_0)$$
$$Z_1 = 2^{SCALE-1} \cdot (X_0 \cdot C_{10} + X_1 \cdot C_{11} + X_2 \cdot C_{12} + A_1)$$
$$Z_2 = 2^{SCALE-1} \cdot (X_0 \cdot C_{20} + X_1 \cdot C_{21} + X_2 \cdot C_{22} + A_2)$$

- For YUV to RGB:
- X0 = Y
- X1 = U
- X2 = V
- Z0 = R
- Z1 = G
- Z2 = B,
- For RGB to YUV:
- X0 = R
- X1 = G
- X2 = B
- Z0 = Y
- Z1 = U
- Z2 = V

The resizing related bit fields' setting of PP CSC1 task lists are:



Address[1]	Word[2]	Parameter	Field	Description
x0060	Postprocessing CSC1 word0 and word1	C22	8:0	Coefficients of color conversion matrix1 for viewfinder task: Z0 = X0*C00 − X1*C01 +X2*C02+A0; Z1 = X0*C10 − X1*C11 +X2*C12+A1; Z2 = X0*C20 − X1*C21 +X2*C22+A2; Coefficients format is s.xxxxxxxx;
		C11	17:9	
		C00	26:18	
		A0	39:27	Offset of color conversion matrix1 for viewfinder task Offset format is sxx.xxxxxxxxxx
		SCALE	41:40	Scale of coefficients for color conversion matrix1 for postprocessing task 0-->coefficients*2 1--> coefficients*1 2--> coefficients*0.5 3-->coefficients*0.25
		SAT MODE	42:42	Saturation mode for color conversion matrix1 for postprocessing task 0 --> (min, max) =(0, 255) 1 --> (min, max)= (16, 240) for Z1,Z2 1 --> (min, max) = (16, 235) for Z0

Figure 13-21. Postprocessing task IC parameters for color space conversion

The main processing unit reads pairs of pixels from the downsize output memory background part. Therefore, the downsize unit also needs to be configured and enabled. Refer to Configuring the IC resize tasks for help.

13.10.5.4 IPU configurations for the DP task

API calls **ipu_dp_csc_config()** to do color space conversion inside the display processor. The conversion formula is:

$$x \text{ -> } \mathrm{Clip}(\mathrm{Round}(S * 2\hat{\ }E)),\ S = Ax + B$$

Where:

- A is a 3 x 3-dimensional matrix of weights, each a 10-bit signed number with 8 fractional digits:

$$A = \begin{bmatrix} CSC\_A0 & CSC\_A1 & CSC\_A2 \\ CSC\_A3 & CSC\_A4 & CSC\_A5 \\ CSC\_A6 & CSC\_A7 & CSC\_A8 \end{bmatrix}$$

- B is a 3-dimensional vector of offsets, each a 14-bit signed number with 2 fractional digits:

$$B = \begin{bmatrix} CSC\_B0 & CSC\_B1 & CSC\_B2 \end{bmatrix}$$

- E is an exponent, assuming one of the following values: -1,0,1,2 (allowing weights up to 8):

$$E = \begin{bmatrix} CSC\_S0 & CSC\_S1 & CSC\_S2 \end{bmatrix}$$

The CSC related bit fields' settings in the display processor are:

- DP_CSC_DEF_SYNC is set to enable color space conversion.
- DP_CSC_A_SYNC_ sets the A parameter.
- DP_CSC_B0_SYNC/ DP_CSC_B1_SYNC/ DP_CSC_B2_SYNC sets the B parameter.
- DP_CSC_S0_SYNC/ DP_CSC_S1_SYNC/ DP_CSC_S2_SYNC sets the E parameter.

Chapter 14
Configuring the LDB Driver

14.1 Overview

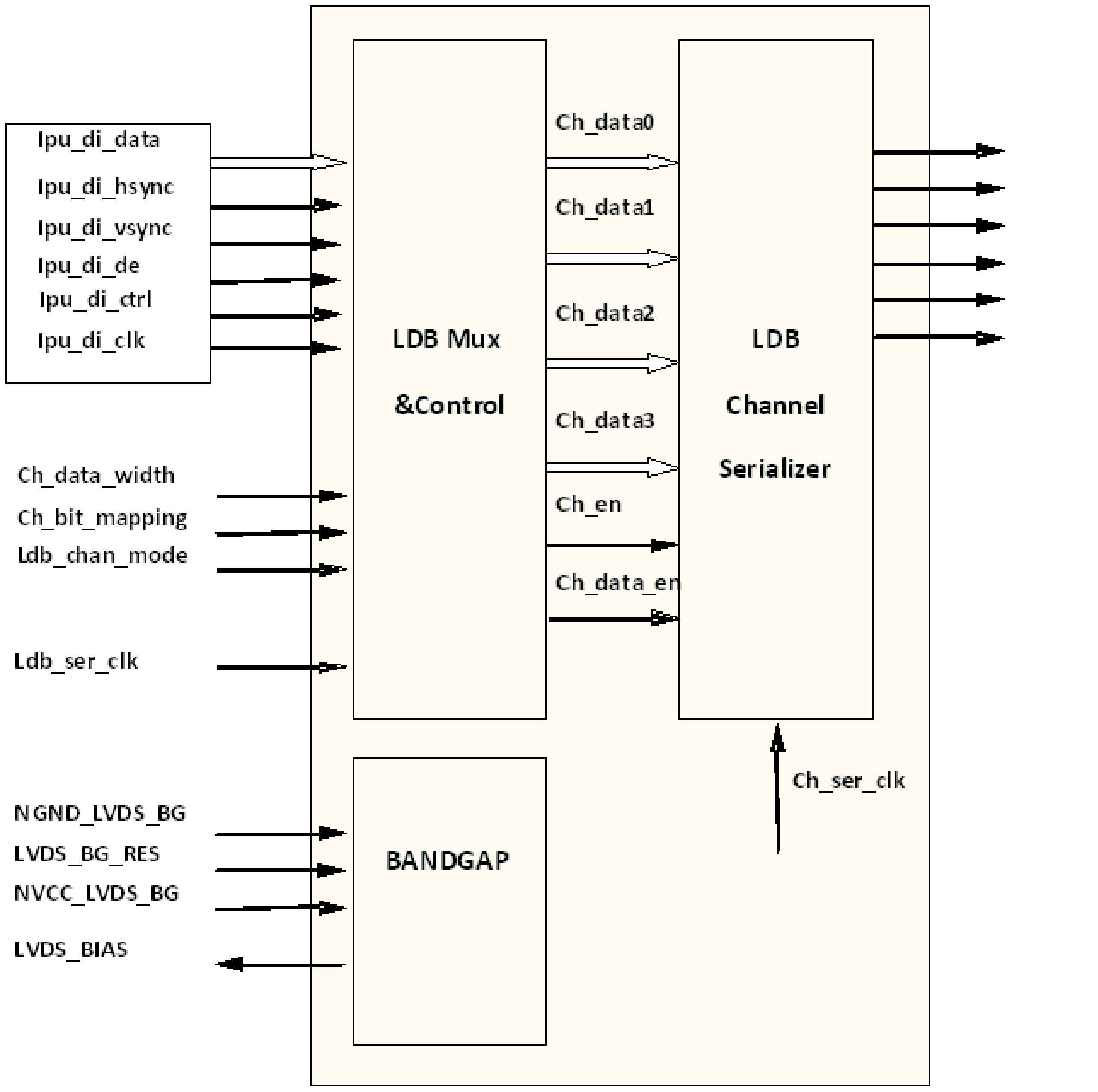
This chapter explains how to configure the LVDS display bridge (LDB), an integrated IP that is used to connect the internal IPU (image processing unit) to the external LVDS display interface. The goal of the LDB is to convert the parallel data into LVDS data lanes. It must be tested together with the IPU, which produces the parallel data, and the LVDS panel which acts as a LVDS receiver.

LVDS (low-voltage differential signaling) is an electrical digital signaling system that can run at very high speeds over inexpensive twisted-pair copper cables. It transmits information as the difference between two voltages on a pair of wires; the two-wire voltages are compared at the receiver end. The low common voltage (the average of the voltages on the paired wires, ~1.2 V) and the low differential voltage (~350 mV) allows LVDS to consume less power than other systems.

This chip has one instance of LDB. it is located in the IOMUX chapters with only one configure register named IOMUXC_IOMUXC_GPR2.

In this chip, the pins are dedicated for LVDS output with no mux.

The following figure shows the LDB block diagram.



[1] The LDB mux and control block is the interface to the IPU and system CCM module and LDB configuration registers
[2] The channel serializers convert the parallel data into serial format.
[3] BANDGAP provides the reference current for the LVDS I/O pad.

Figure 14-1. LDB block diagram

14.2 Feature summary

LDB supports:

- Connectivity to devices that have displays with LVDS receivers
- Arranging data to meet the requirements of the external display receiver and the LVDS display standards
- Synchronization and control capabilities to avoid tearing artifacts.

14.3 Input and output ports

The LDB module obtains its input from the IPU display interfaces. The LVDS channel theoretically has four choices for routing its data path because there are two IPU modules with two display ports per IPU.

However, there is no reason to connect LVDS channel 0 to IPU DI1 or LVDS channel 1 to IPU DI0 in a single display mode; those connections are only supported in dual display mode. See Modes of operation for more information.

LVDS output uses the following four pairs of wires:

- TX0_P/N
- TX1_P/N
- TX2_P/N
- TX3_P/N
- TXC_P/N

LVDS uses a current-mode driver output from a 3.5 mA current source. This drives a differential line that is terminated by a 100 Ω resistor, generating about 350 mV across the receiver. The +350 mV voltage swing is centered on a 1.2 V offset voltage.

14.4 Modes of operation

LDB supports the following modes of operation:

- Single display mode
- Dual display mode
- Split mode

The following table summarizes the channel mapping for each mode:

Table 14-1. Channel mapping

Use Case	LVDS channel 0	LVDS channel 1
Single display mode		

Table continues on the next page...

Table 14-1. Channel mapping (continued)

Use Case	LVDS channel 0	LVDS channel 1
Single channel DI0 on channel 0	DI0	Disabled
Single channel DI1 on channel 1	Disabled	DI1
Dual display mode		
Dual channels to DI0	DI0	DI0
Dual channel to DI1	DI1	DI1
Separate display mode		
Separate channels	DI0	DI1
Split mode		
Split mode to DI0	DI0 (odd pixels in line)	DI0 (even pixels in line)
Split mode to DI1	DI1 (odd pixels in line)	DI1 (even pixels in line)

14.4.1 Single display mode

In single display mode, either LVDS channel 0 or channel 1 is enabled but not both. The selected channel must be connected to the appropriate IPU display interface: channel 0 to DI0 and channel 1 to DI1.

To enable LVDS channel 0:

1. Connect LVDS channel 0 to DI0.
2. Configure LVDS0_MUX_CTL in IOMUXC_GPR3 to be 0h or 2h.
3. Enable channel 0 by setting CH0_MODE to be 1h in IOMUXC_GPR2.

To enable LVDS channel 1:

1. Connected LVDS channel 1 to DI1.
2. Configure LVDS0_MUX_CTL in IOMUXC_GPR3 to be 1h or 3h.
3. Enable channel 0 by setting CH0_MODE to be 0x3 in IOMUXC_GPR2.

14.4.2 Dual display mode

In dual display mode, LVDS channel 0 and 1 are jointly enabled. Both channels must be connected to the same IPU display interface (for example, both connected to DI1).

14.4.3 Separate display mode

In separate display mode, both channel 0 and channel 1 are enabled, but they are connected to different display interfaces. This allows users to display different content on the different displays.

14.4.4 Split mode

In split mode, the LDB has one input and two outputs. The parallel data is first serialized and then output in horizontal interlaced mode. Odd columns are output from LVDS channel0, and even columns are output from LVDS channel1.

14.5 LDB Processing

The LDB's main job is to convert the parallel data lines into differential serial data lines. It supports SPWG and JEIDA mapping modes. See Data serialization clocking for additional information.

Use the LDB_CTRL register to configure the data mapping mode and data width. See Configuring the LDB_CTRL register for further information.

14.5.1 SPWG mapping

SPWG (standard panel working group) uses a set of standard LCD panels with dimensions and interface characteristics that allow both notebook and LCD supplier industries to manage the volatile LCD supply and demand in an easier fashion. The following table shows the SPWG mapping mode.

Table 14-2. SPWG mapping mode

Serializer input	Slot 0	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6
CHx_DATA0	G0	R5	R4	R3	R2	R1	R0
CHx_DATA1	B1	B0	G5	G4	G3	G2	G1
CHx_DATA2	DE	VS	HS	B5	B4	B3	B2
CHx_DATA3 (for 24 bpp only)	CTL	B7	B6	G7	G6	R7	R6

14.5.2 JEIDA mapping

JEIDA (The Japan Electronic Industry Development Association) was an industry research, development, and standards body for electronics in Japan. JEIDA mapping mode is also popular for LVDS panels. The following table shows the JEIDA mapping mode.

Table 14-3. JEIDA mapping mode

Serializer input	Slot 0	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slot 6
CHx_DATA0	G2	R7	R6	R5	R4	R3	R2
CHx_DATA1	B3	B2	G7	G6	G5	G4	G3
CHx_DATA2	DE	VS	HS	B7	B6	B5	B4
CHx_DATA3	CTL	B1	B0	G1	G0	R1	R0

14.6 Clocks

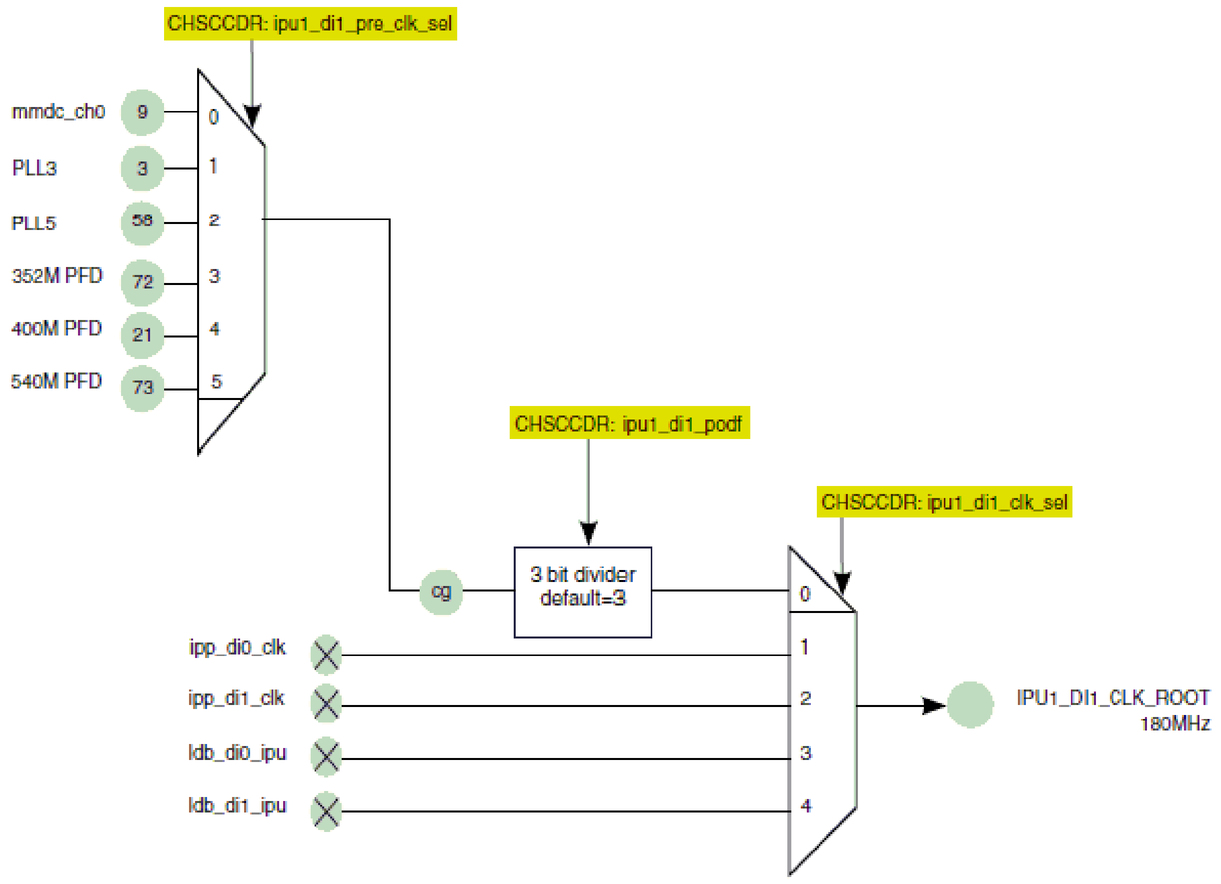


Figure 14-2. LDB clock tree

Route IPU_DI_CLK_ROOT to ipp_di_clk.

14.6.1 Data serialization clocking

The LDB module serializes the parallel 18/24 bit data output from IPU. In both SPWG and JEIDA modes, one pixel is reordered into 3 or 4 lines, with 7 bits per line.

- In non-split mode, for the IPU side, one pixel is driven to LDB during a pixel clock period, and for the LDB side, one pixel is driven to the display in 7 serialization clock periods.
- In split mode, one frame is split into two horizontal fields, and the serialization clock is x3.5 the pixel clock.

The IPU pixel clock and the serialization clock of LDB must be synchronous. To enable this:

1. Select the IPU DI clock to be external in the IPU configuration registers.
2. Choose the clock branch in CCM to root the IPU DI clock from the LDB DI clock.

The following figure shows how to generate the LDB serialization clock.

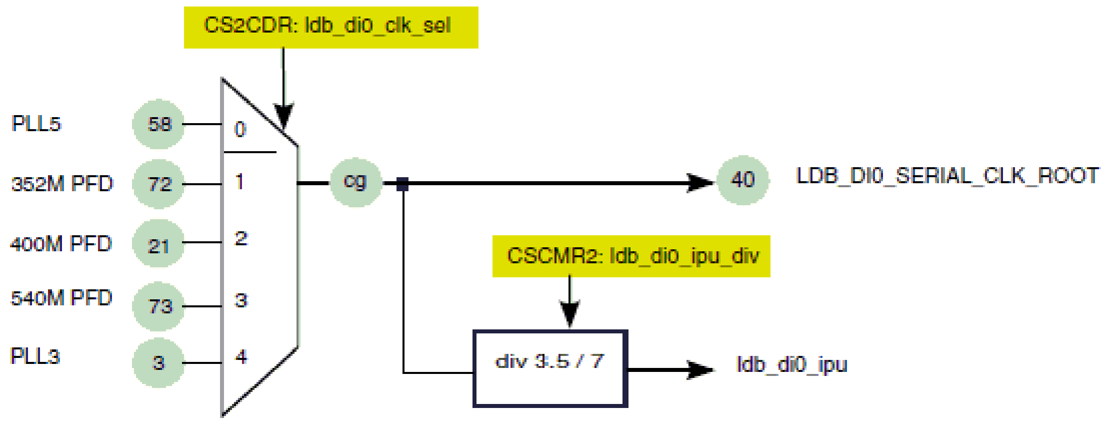


Figure 14-3. LDB serialization clock

The pixel clock is generated by dividing the clock selected by ldb_di_clk_sel (as shown in Figure 14-3, there are five clock sources available) by 3.5 if in split mode or 7 if in non-split mode.

14.7 Configuring the LDB_CTRL register

Configure the following parameters in the LDB_CTRL register for your use case:

vs_polarity Polarity of VSYNC signal; should match the IPU output

Bit_mapping Using the SPWG or the JEIDA standard

Data_width 18 bit or 24 bit selection

Split_mode Enable or disable split mode

Channel_mode Channel route to IPU DI

14.8 Use cases

This section provides example settings for:

- Image display on Hannstar HSD100PXN1 XGA panel
- Image display on CHIMEI M216H1 1080HD panel

The following table shows the implementation for the Hannstar HSD100PXN1 XGA panel use case:

Table 14-4. Hannstar HSD100PXN1 XGA panel use case

Setting	Requirements
Mode	• Single display mode
Power supply	• 3.3 V for core/IO • 5 V for backlight LED driven
Clock settings	• 65 MHz for ldb_di_clk (typical pixel clock for XGA resolution) • 455 MHz for LDB_DI_SERIAL_CLK_ROOT (ldb_di_ipu_div is set to 7 in non-split display mode and 65 x 7 = 455 MHz).
LDB configuration	• ldb_config(IPU1_DI0, LVDS _PORT0, SPWG, LVDS_PANEL_18BITS_MODE); **NOTE:** The LDB is connected to IPU1 DI0 output, and LVDS port0 is enabled. LVDS output is in SPWG standard with 18 bit width, so the TX3 lane is ignored. • ldb_config(IPU1_DI0, LVDS _DUAL_PORT, SPWG, LVDS_PANEL_18BITS_MODE); **NOTE:** In this mode, IPU output is sent to both LVDS channels, and the content is identical.

The following table shows the implementation for the CHIMEI M216H1 1080HD panel use case:

Table 14-5. CHIMEI M216H1 1080HD panel use case

Setting	Requirements
Mode	• Split display mode
Power supply	• 5 V for core/IO and backlight LED driven
Clock settings	• 74.25 MHz for ldb_di_clk (typical pixel clock for HD1080 with 30 Hz refresh rate) Note that in split mode, the panel acts as pixel interleaved mode, 960 x 1280 at 30 fps per LVDS channel. • 260 MHz for LDB_DI_SERIAL_CLK_ROOT (ldb_di_ipu_div is set to 3.5 in split display mode and 74.25 x 3.5 = 260 MHz).
LDB configuration	• ldb_config(IPU1_DI0, LVDS _SPLIT_PORT, SPWG, LVDS_PANEL_18BITS_MODE); **NOTE:** The LDB is connected to IPU1 DI0 output, and LVDS port0 is enabled. LVDS output is in SPWG standard with 18 bit width, and data is processed in split mode.

Chapter 15
Configuring the OCOTP Driver

15.1 Overview

This chapter explains how to configure the OCOTP driver. The OCOTP controller is used to read and write to the chip's OTP eFuses.

There is one instance of OCOTP in the chip, located in the memory map at the base address 021B C000h.

15.2 Features summary

This low-level driver supports:

- Read of a fuse bank/row
- Write to a fuse bank/row.

15.3 Modes of operation

The following table explains the OCOTP modes of operation:

Table 15-1. OCOTP modes of operation

Mode	What it does
Sense operation	Reads the content of a fuse location as defined by a bank and a row
Write operation	Writes a value to a fuse location as defined by a bank and a row

15.4 Clocks

This controller uses a single input clock: IPG_CLK. The read and write timings are calculated based on the IPG_CLK frequency.

Table 15-2. OCOTP reference clocks

Clock	Name	Description
IPG_CLK	IPG_CLK	Global IPG_CLK that is typically used in normal operation. It is provided by CCM. It cannot be powered down.

15.5 IOMUX pin mapping

This module has no off-chip connection.

15.6 Resets and interrupts

The external application is responsible for creating the interrupt subroutine. The address of this routine is passed through the hw_module structure, which is defined in ./src/include/io.h. The application also initializes and manages the interrupt subroutine.

All interrupt sources are listed in the "Interrupts and DMA Events" chapter of the device reference manual. In the SDK, the list is provided at ./src/include/mx61/soc_memory_map.h.

15.7 Initializing the driver

This controller does not need a special initialization procedure. The driver API is limited to the functions below. The first is used to read at a fuse location, and the second is used to program a value at a fuse location.

```
/*!
 * Read the value of fuses located at bank/row.
 *
 * @param  bank of the fuse
 * @param  row of the fuse
 * @return fuse value
 */
int32_t sense_fuse(uint32_t bank, uint32_t row)
/*!
 * Program fuses located at bank/row to value.
 *
 * @param  bank of the fuses.
 * @param  row of the fuses.
 * @param  value to program in fuses.
 */
void fuse_blow_row(uint32_t bank, uint32_t row, uint32_t value)
```

The bank and row/word of a fuse location is specified in the OCOTP register definitions, which are available in *i.MX 6Dual/6Quad Multimedia Applications Processor Reference Manual* (IMX6DQRM).

15.8 Testing the driver

A test is available to read or write at any fuse location. The test uses interactive messages to let the user choose which bank and row should be read or written to.

NOTE

All e-Fuses are one time programmable, so any misusage of the write command is irrerversible.

15.9 Running the test

To run the OCOTP test, the SDK builds the test with the following command:

```
./tools/build_sdk -target mx61 -board ard -board_rev 1 -test ocotp
```

This generates the following ELF and binary files:

```
./output/mx61/bin/mx61ard-ocotp-sdk.elf
./output/mx61/bin/mx61ard-ocotp-sdk.bin
```

15.10 Source code structure

Table 15-3. Source code file locations

Description	Location
Low-level driver source	./src/sdk/ocotp/drv/ocotp.c
Low-level driver header	./src/sdk/ocotp/drv/ocotp.h
Unit tests	./src/sdk/ocotp/test/ocotp_test.c

Chapter 16
Using the SATA SDK

16.1 Overview

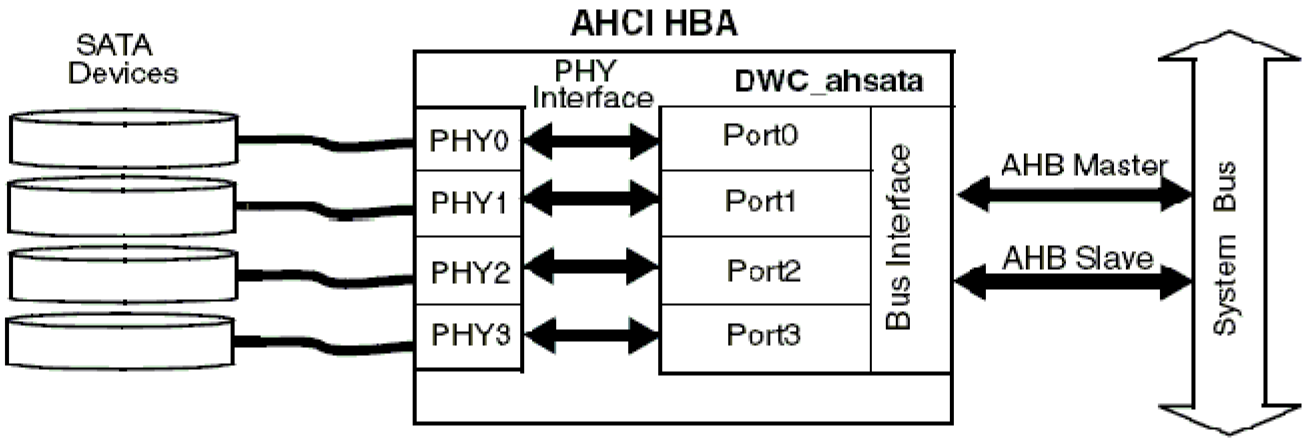


Figure 16-1. SATA system block diagram

This chapter explains how to use the SATA SDK, which provides the most basic instructions for initializing, identifying, and reading/writing of SATA. The SDK does not support all SATA features

The DWC_ahsata is an AHCI-compliant SATA AHCI host bus adaptor (HBA) that is used with a corresponding multi-port physical layer (PHY) to form a complete AHCI HBA interface. Although the block diagram shows four ports, this chip only uses PORT0 and PHY0.

16.2 Feature summary

DWC_ahsata supports the following:

- SATA 1.5-Gbps Generation 1

- Power management features including automatic partial to slumber transition
- BIST loopback modes
- One SATA device (port 0)
- Hardware-assisted native command queuing for up to 32 entries
- Port multiplier with command-based switching
- Disabling Rx and Tx Data clocks during power down modes

It also features:

- Conformity to Serial ATA Specification 2.6 and AHCI Revision 1.1 specifications.
- A highly configurable PHY interface
- Additional user defined PHY status and control ports
- Configurable AMBA AHB interface (one master and one slave).
- Internal DMA engine per port.

It has the option of featuring:

- Rx Data Buffer for recovered clock systems
- Data alignment circuitry (when Rx Data Buffer is included)
- OOB signaling detection and generation.
- Gen2 speed negotiation (when Tx OOB signaling is included)
- Asynchronous Signal Recovery, including retry polling (when Tx OOB signalling is included)
- 8b/10b encoding/decoding

16.3 Modes of operation

Table 16-1. Modes of operation

Mode	What it does
DMA	DMA mode of SATA
PIO	PIO mode of SATA

16.4 Clocks

Clock	Name	Description
Ethernet PLL	Ethernet PLL	The PLL outputs a 500 MHz clock. It also generates the SATA clock (100 MHz).
CCGR5	SATA clock gate	SATA clock gate

16.5 IOMUX pin mapping

Table 16-2. SATA pin mapping

Signals	PAD	MUX	SION	Description
IOMUXC_IOMUXC_GPR13	-	-	-	SATA PHY control
MAX7310 U19 CTRL_0	-	-	-	SATA power

16.6 Resets and Interrupts

SATA IRQ number is 71.

The SDK did not implement an interrupt mode.

16.7 Initializing the driver

```
sata_return_t sata_init(sata_ahci_regs_t * ahci)
{
sata_power_on(); /*1. Power on SATA*/
sata_clock_init(); /*2. Initialize the clock of SATA*/
/*3. Initialize SATA controller and PHY*/
}
```

16.8 Testing the driver

```
int main(void)
{
          sata_init(); /*1. Initialize SATA*/
          sata_identify() /*2. Identify SATA*/
          /*3. Read and Write test*/
}
```

16.9 Source code structure

Table 16-3. Source code file locations

Description	Location
Low-level driver source	./src/sdk/sata/drv/sata.c
Test source	./src/sdk/sata/test/sata_test.c
Header file	./src/sdk/sata/inc/atapi.h
Header file	./src/include/imx_sata.h

Chapter 17
Configuring the SDMA Driver

17.1 Overview

The smart direct memory access (SDMA) controller is composed of a RISC core, ROM, RAM, and a scheduler. It is used for programs dedicated for various kinds of DMA transfer. The SDMA controller helps maximize system performance by off-loading the ARM core in dynamic data routing.

This chapter uses i.MX 6Dual/6Quad ARD board schematics for pin assignments. For other board types refer to the respective schematics.

There is one instance of SDMA in i.MX 6Dual/6Quad. It is located in memory map at the SDMA base address of 020E C000h.

17.2 IOMUX pin mapping

Configure the IOMUX for SDMA into the iomux_config() function located in ./src/mx61/hardware.c.

Table 17-1. Pin assignments for i.MX 6Dual/6Quad

Signal	IOMUXC Setting for SDMA		
	PAD	MUX	SION
DEBUG_BUS_DEVICE[0]	DISP0_DAT21	ALT4	1
DEBUG_BUS_DEVICE[1]	DISP0_DAT22	ALT4	1
DEBUG_BUS_DEVICE[2]	DISP0_DAT23	ALT4	1
DEBUG_BUS_DEVICE[3]	ENET_MDIO	ALT3	1
DEBUG_BUS_DEVICE[4]	ENET_REF_CLK	ALT3	1
SDMA_EXT_EVENT[0]	GPIO_17	ALT3	1
SDMA_EXT_EVENT[0]	DISP0_DAT16	ALT4	1

Table continues on the next page...

Table 17-1. Pin assignments for i.MX 6Dual/6Quad (continued)

Signal	IOMUXC Setting for SDMA		
	PAD	MUX	SION
SDMA_EXT_EVENT[1]	GPIO_18	ALT3	1
SDMA_EXT_EVENT[1]	DISP0_DAT17	ALT4	1
DEBUG_EVT_CHN_LINES[0]	DISP0_DAT13	ALT4	1
DEBUG_EVT_CHN_LINES[1]	DISP0_DAT14	ALT4	1
DEBUG_EVT_CHN_LINES[2]	DISP0_DAT15	ALT4	1
DEBUG_EVT_CHN_LINES[3]	EIM_DA12	ALT4	1
DEBUG_EVT_CHN_LINES[4]	EIM_DA13	ALT4	1
DEBUG_EVT_CHN_LINES[5]	EIM_DA14	ALT4	1
DEBUG_EVT_CHN_LINES[6]	EIM_DA11	ALT4	1
DEBUG_EVT_CHN_LINES[7]	DISP0_DAT20	ALT4	1
DEBUG_PC[0]	CSI0_PIXCLK	ALT4	1
DEBUG_PC[1]	CSI0_MCLK	ALT4	1
DEBUG_PC[2]	CSI0_DATA_EN	ALT4	1
DEBUG_PC[3]	CSI0_VSYNC	ALT4	1
DEBUG_PC[4]	CSI0_DAT10	ALT4	1
DEBUG_PC[5]	CSI0_DAT11	ALT4	1
DEBUG_PC[6]	CSI0_DAT12	ALT4	1
DEBUG_PC[7]	CSI0_DAT13	ALT4	1
DEBUG_PC[8]	CSI0_DAT14	ALT4	1
DEBUG_PC[9]	CSI0_DAT15	ALT4	1
DEBUG_PC[10]	CSI0_DAT16	ALT4	1
DEBUG_PC[11]	CSI0_DAT17	ALT4	1
DEBUG_PC[12]	CSI0_DAT18	ALT4	1
DEBUG_PC[13]	CSI0_DAT19	ALT4	1
DEBUG_CORE_STATE[0]	DI0_DISP_CLK	ALT4	1
DEBUG_CORE_STATE[1]	DI0_PIN15	ALT4	1
DEBUG_CORE_STATE[2]	DI0_PIN2	ALT4	1
DEBUG_CORE_STATE[3]	DI0_PIN3	ALT4	1
DEBUG_YIELD	DI0_PIN4	ALT4	1
DEBUG_CORE_RUN	DISP0_DAT0	ALT4	1
DEBUG_EVENT_CHANNEL_SEL	DISP0_DAT1	ALT4	1
DEBUG_MODE	DISP0_DAT2	ALT4	
DEBUG_BUS_ERROR	DISP0_DAT3	ALT4	1
DEBUG_BUS_RWB	DISP0_DAT4	ALT4	1

Table continues on the next page...

Table 17-1. Pin assignments for i.MX 6Dual/6Quad (continued)

Signal	IOMUXC Setting for SDMA		
	PAD	MUX	SION
DEBUG_MATCHED_DMBUS	DISP0_DAT5	ALT4	1
DEBUG_RTBUFFER_WRITE	DISP0_DAT6	ALT4	1
DEBUG_EVENT_CHANNEL[0]	DISP0_DAT7	ALT4	1
DEBUG_EVENT_CHANNEL[1]	DISP0_DAT8	ALT4	1
DEBUG_EVENT_CHANNEL[2]	DISP0_DAT9	ALT4	1
DEBUG_EVENT_CHANNEL[3]	DISP0_DAT10	ALT4	1
DEBUG_EVENT_CHANNEL[4]	DISP0_DAT11	ALT4	1
DEBUG_EVENT_CHANNEL[5]	DISP0_DAT12	ALT4	1

17.3 Scripts

The SoC reference manual provides a set of scripts to perform DMA transfers among the memories and peripherals. Each of these scripts support one type of transfer, such as memory to peripheral and peripheral to memory. Some scripts are dedicated to specific peripherals with some feature turned on. See Appendix A in the IMX6DQRM for details.

The SoC supports three types of access:

- Burst access-to internal or external AP memories
- Per DMA through the functional unit bus-to AP peripherals
- Through the SPBA bus-to AP peripherals

Note that some peripherals reside on the Trust Zone off platform. Users need to turn off the Trust Zone to enable the SDMA access.

The scripts reside in two different places: the ROM and the RAM. The ROM contains startup scripts (boot code) and other common utilities which are referenced by the scripts in the RAM. The internal RAM is divided into a context area and a script area. Users need to download the RAM scripts into SDMA RAM through channel 0. According to the input parameters, the channel 0 script can also download other channel's context data to SDMA RAM.

17.4 Channels and channel descriptor

SDMA has up to 32 virtual DMA time-division multiplexed channels. They are executed based on channel status, its priority, DMA event map, context area (Every transfer channel requires one context area to keep the contents of all the core and unit registers while inactive) and channel control blocks (CCBs) supported. The scheduler provides hardware-based coordination among the active channels. A context area stores the SDMA core's context, and the CCB manages the buffer descriptor list.

The SDMA API provides a data structure named sdma_chan_desc_t to describe the channels.

```
typedef struct {
    unsigned int script_addr;
    unsigned int gpr[8];
    unsigned int dma_mask[2];
    unsigned char priority;
    unsigned int nbd;
} sdma_chan_desc_t, *sdma_chan_desc_p;
```

script_addr Script's address

gpr[8] Some parameters that the script uses, such as watermark. Refer to the "SDMA script" appendix in the i.MX53 reference manual (IMX53RM) for each script's details.

dma_mask[2] the event if the channel is triggered; refer to the "Interrupts and DMA Events" chapter in the SOC's reference manual for the details.

priority Priority of the channel (0-7)

nbd Number of buffer descriptors

Set up this structure before requesting a channel. The SDMA API needs this structure and a buffer descriptor to request a channel.

17.5 Buffer descriptor and BD chain

SDMA scripts use the CCB to manage the buffer descriptors. In AP software, the SDMAARM_MC0PTR register should be set to the address of CCB table of all 32 channels. In the channel script, the script knows the base address of its CCB based on the address in the SDMAARM_MC0PTR and the channel number. Because the base address of buffer descriptors is provided in the CCB, the script can read and process the commands and parameters in the buffer descriptors in order to perform the transfer. Refer to Appendix A in the i.MX6DQRM for the detailed description of the buffer descriptor usage for each script.

Typically, in the buffer descriptor data structure, the first 32 bit word is called mode word; the next two words are base and extended buffer address. The following table shows the field layout:

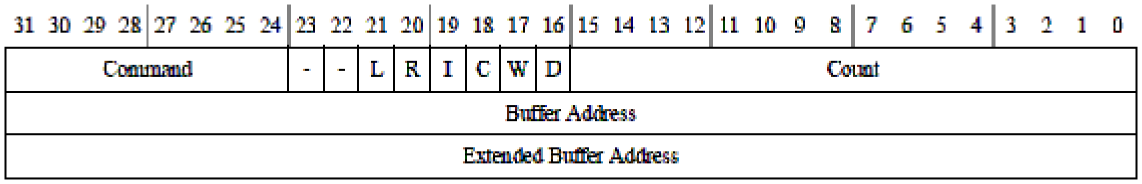


Figure 17-1. Buffer descriptor format

Field descriptions are as follows:

Count Number of bytes for this transfer

D If D = 0, SDMA has finished the transfer for this buffer descriptor. If D = 1, SDMA has not.

W Wrap. If W = 1, wraps to the base BD (pointed to by basdBDptr in CCB).

C Continuous. If C = 1 moves to the next BD after current BD is done.

I Interrupt. If I = 1 sets the corresponding bit (according to the channel number) in SDMA interrupt register after current BD is done.

R Error. If R = 1, an error occurred during the current BD transfer

L Last buffer descriptor. This bit is set in the SDMA IPC scripts to tell the receiving core that the transfer has ended.

Command This field is used to differentiate operations performed in the script. Usage of this field varies from script to script. Typically, bit 24 and 25 indicate the bus width.

If the continuous bit is set (in buffer descriptor [C]), the SDMA script finishes processing one buffer descriptor and then immediately processes the next buffer descriptor, creating a buffer descriptor chain. One channel can support up to 64 buffer descriptors in the chain. The continuous bit of the last buffer descriptor in the chain should be cleared.

17.6 Application programming interface

The API shown in this section is for the SDMA transfer control. See Using the API, for usage information.

```
int sdma_init(sdma_env_p envp, unsigned int base_addr)
```

Description: Initialize the system environment for SDMA. This function will reset SDMA controller

Setup configurations like AP DMA/SDMA clock ratio, CCB base address etc

Use channel 0 script to load the RAM scripts into SDMA RAM

Parameters: **envp** (uncacheable and unbufferable buffer allocated by user)

base_addr (base address of SDMA registers in AP)

Returns : 0 on success

-1 when fail to download RAM scripts to SDMA RAM

-2 when environment pointer is NULL

```
void sdma_deinit(void)
```

Description: De-initialize the SDMA environment. This function will close and free all the channels, clear all the EP and overrides of channels

Parameters: none

Return: none

```
int sdma_channel_
        request(sdma_chan_desc_p cdp, sdma_bd_p bdp)
```

Description: Allocates a free channel and opens it. This function will validate the input parameters, find and allocate a free channel, setup the channel overrides, DMA masks, buffer descriptors, channel priority etc, It also writes the channel context to SDMA RAM

Parameters: **cdp** (A pointer to user provided data. It includes necessary channel descriptors of: script_addr (script address) in SDMA defined in sdma_script_code.h.

gpr[8] (normally it includes the FIFO address, DMA mask, watermark etc. User could refer to the script manual released with the sdma_script_code.h for details.

dma_mask[2] (DMA mask to set in register of channel enable. Normally it's also provided in gpr[8]. We separate it here to support some special script that may have some different usage of GPRs priority the channel priority

bdp (A pointer to the user provided buffer descriptor table.)

Return: return the channel number on success

-2: at least one of the user provided pointers is NULL

-3: channel priority exceeds limitation (1-7)

-4: no free channel that could be allocated

-5: got failure when download channel context to SDMA RAM

-6: too many buffer descriptors in table (>64)

-7: SDMA is not yet initialized

`int sdma_channel_release(unsigned int channel)`

Description: Close the channel selected. This function stops and frees the channel clear the EP if set, resets the channel override, resets the channel control block

Parameters: **channel** (channel number)

Return: 0 on success,

-1 when channel number is not in range (0-31) or SDMA is not yet initialized

----

int **sdma_channel_start**(unsigned int channel)

Description: Starts the channel selected.

Parameters: **channel** (channel number)

Return: 0 on success,

-1 when channel number is out of range (0-31) or channel is free

```
----
int sdma_channel_stop(unsigned int channel)
```

Description: Stops the channel selected.

Parameters: **channel** (channel number)

Return: 0 on success,

-1 when channel number is out of range (0-31)

----

int **sdma_channel_status**(unsigned int channel, unsigned int *status)

Description:

Parameters: **channel** (channel number)

status (the pointer holds the channel's status: error, done or busy)

Return: 0 on success,

-1 on failure

```
----
int  sdma_lookup_script(script_name_e  script, unsigned int *addr )
```

Description:

Parameters: **script** (script name to lookup)

addr (the pointer holds the script's address if the function return 0)

Return: 0 on success,

-1 on failure

17.7 Using the API

The following example shows the typical usage of the API. To save space, some pseudocode is used.

```
SDMA_demo{
          Allocate uncacheable  and unbuffereable memory for BDs, buffers,etc.
          sdma_init();
          Set up channel descriptors and BDs;
          sdma_channel_request();
          sdma_channel_start();
          Wait for channel done;
          sdma_channel_stop();
          sdma_channel_release();
          sdma_deinit();
}
```

Use the following sequence:

1. Allocate a static or dynamic buffer to store the global variable used by the API.
2. • Note that this buffer is accessed by DMA and must be uncacheable and unbufferable.
3. Initialize the SDMA environment with sdma_init.
4. To Initiate an SDMA transfer, use sdma_channel_request to allocate a channel with the necessary inputs provided. These inputs are bundled in two data structures, a channel descriptor and a buffer descriptor.
5. • Provide the buffer to store these data structures and buffer descriptors. This buffer is also accessed by the SDMA and must be uncacheable and unbufferable.
   • Use sdma_script_lookup to find the script's address.
   • Set the channel attributes and necessary context contents (such as channel priority, which script to use, and GPRs) in the channel descriptor structure.
   • Refer to IMX6DQRM, Appendix A for full details of each script.
6. While initializing an SDMA transfer without DMA event (memory to memory transfer triggered by SDMAARM_HSTART register), start the SDMA transfer with **sdma_channel_start**. If a DMA event is involved, configure and start the peripheral

as well as enable peripheral DMA control. This opens up the channel. Then use **sdma_channel_start** to start the transfer.

7. To initiate this transfer again, change the data in the buffer descriptor and then restart the channel with **sdma_channel_start** again.
8. Use **sdma_channel_release** to free the channel or stop the ongoing transfer with **sdma_channel_stop**.
9. **sdma_deinit** provides a way to re-initialize the SDMA together with the **sdma_init**.

Chapter 18
Configuring the SPDIF Driver

18.1 Overview

This chapter describes module-level operation and programming for the Sony/Philips digital interface (SPDIF) audio block driver. The SPDIF audio block is a stereo transceiver that allows the processor to receive and transmit digital audio. The SPDIF transceiver allows the handling of both SPDIF channel status (CS) and user (U) data. It also includes a frequency measurement block that allows the precise measurement of an incoming sampling frequency. The pseudocode supplied in the document is based on the SPDIF driver in the diag-sdk repository.

SPDIF is typically used to transfer samples in a periodic manner. It consists of independent transmitter and receiver sections with independent FIFOs and control blocks.

SPDIF is compatible with the IEC60958 standard. Refer to IEC60958 for further details.

NOTE

This chapter uses i.MX 6Dual/6Quad EVB board schematics for pin assignments. For other board types, refer to the respective schematics.

18.2 Feature summary

The SPDIF driver supports:

- A simple framework for audio
- SPDIF APIs

18.3 Clocks

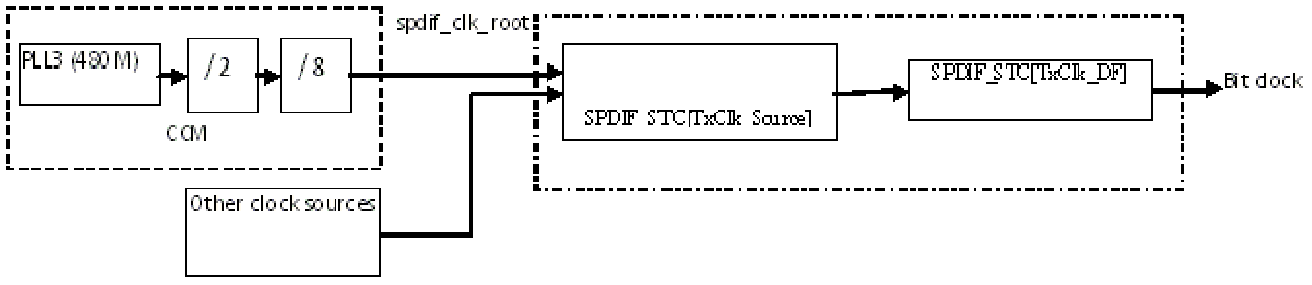


Figure 18-1. SPDIF clock tree (default)

Before using SPDIF, use CCM[CCGRx] to gate on the spdif_clock. Refer to the "Clock Controller Module" chapter in the chip reference manual for details.

By default, spdif_clk_root is sourced from PLL3, whose default value is 480 MHz. The default spdif_clk_pred value is 2 and the default spdif_clk_podf value is 8; therefore, spdif_clk_root is divided to 30 MHz.

NOTE

Any change of spdif_clk_root affects modules for which it is the source clock (such as ESAI). Therefore, we recommended using the default spdif_clk_root value (30 MHz) for the SPDIF module.

The transmit clock can be selected from several clock sources, such as ASRC_CLK or ESAI_CLK. Set SPDIF_STC[TxClk_Source] to select a specific clock source. The selected source is divided by SPDIF_STC[TxClk_DF] to generate the bit clock.

Because the SPDIFIN signal carries both clock and data, no receive clock is needed.

18.4 IOMUX pin mapping

The following table lists the IOMUX configurations for the MX61_EVB board. For other boards, refer to the appropriate board schematics for correct pin assignments.

Table 18-1. IOMUX pin mapping for SPDIF on the MX61_EVB board

Signals	PAD	MUX
SPDIFIN	KEY_COL3	ALT6
SPDIFOUT	GPIO_19	ALT2

18.5 Audio framework

Because this chip uses multiple audio controllers and audio codecs, an audio framework is needed to manage all audio modules (controllers and codecs) and to provide a uniform APIs for application programmers.

The following three data structures create the audio framework:

- `audio_card_t`-describes the audio card
- `audio_ctrl_t`-describes the audio controller (for example, SSI or ESAI module)
- `audio_codec_t`-describes the audio codec (for example sgtl5000 or cs428888)

In addition, `audio_dev_ops_t` is the data member for the three audio framework data structures and `audio_dev_para_t` describes the audio parameter passed to the configuration function.

The audio card consists of one audio controller and one audio codec. `audio_card_t` is the only data structure that applications can access and manage.

18.5.1 `audio_card_t` data structure

This data structure describes the audio card. It is as follows:

```
typedef struct {
    const char *name;
    audio_codec_p codec; //audio codec which is included
    audio_ctrl_p ctrl;        //audio controller which is included
    audio_dev_ops_p ops; //APIs
} audio_card_t, *audio_card_p;
```

This driver defines a global variable audio_card_t snd_card_spdif to represent the SPDIF module within the chip:

```
audio_card_t snd_card_spdif = {
    .name = "i.MX SPDIF sound card",
    .codec = NULL,
    .ctrl = &imx_spdif,
    .ops = &snd_card_ops,
};
```

18.5.2 `audio_ctrl_t` data structure

This data structure describes the audio controller. It is as follows:

```
typedef struct {
    const char *name;
    uint32_t base_addr;         // the io base address of the controller
    audio_bus_type_e bus_type;  //The bus type(ssi, esai or spdif) the controller supports
    audio_bus_mode_e bus_mode;  //the bus mode(master, slave or both)the controller supports
    int irq;                    //the irq number
    int sdma_ch;                //Will be used for SDMA
    audio_dev_ops_p ops;   //APIs
} audio_ctrl_t, *audio_ctrl_p;
```

18.5.3 `audio_codec_t` data structure

This data structure describes the audio codec. It is as follows:

```
typedef struct {
    const char *name;
    uint32_t i2c_base;          //the i2c connect with the codec
    uint32_t i2c_freq;          // i2c operate freq;
    uint32_t i2c_dev_addr;      //Device address for I2C bus
    audio_bus_type_e bus_type;  //The bus type(ssi, esai or spdif) the codec supports
    audio_bus_mode_e bus_mode;  //the bus mode(master, slave or both)the codec supports
    audio_dev_ops_p ops;   //APIs
} audio_codec_t, *audio_codec_p;
```

18.5.4 audio_dev_ops_t data structure

This data structure describes the APIs of the audio devices (codec, controller, and card). It is as follows:

```
 typedef struct {
    int (*init) (void *priv);
    int (*deinit) (void *priv);
    int (*config) (void *priv, audio_dev_para_p para);
    int (*ioctl) (void *priv, uint32_t cmd, void *para);
    int (*write) (void *priv, uint8_t * buf, uint32_t byte2write, uint32_t *bytewrittern);
    int (*read) (void *priv, uint8_t * buf, uint32_t byte2read, uint32_t byteread);
} audio_dev_ops_t, *audio_dev_ops_p;
```

18.5.5 audio_dev_para_t data structure

This data structure describes the audio parameter passed to the configuration function. It is as follows:

```
typedef struct {
    audio_bus_mode_e bus_mode; //Master or slave
    audio_bus_protocol_e bus_protocol; //I2S, AC97 and so on
    audio_trans_dir_e trans_dir; //Tx, Rx or both
    audio_samplerate_e sample_rate; //32K, 44.1K , 48K, and so on
    audio_word_length_e word_length;
    unsigned int channel_number;
} audio_dev_para_t, *audio_dev_para_p;
```

18.6 Using SPDIF driver functions

The SPDIF driver has both local functions and public APIs.

The local functions are used to:

- Soft reset the driver
- Dump the SPDIF registers
- Obtain the SPDIF setting and status

The public APIs are used to:

- Initialize and de-initialize the SPDIF
- Configure the SPDIF
- Play data back through the SPDIF

18.6.1 Soft resetting SPDIF

The SPDIF_SCR [soft_reset] bit is used to soft reset the SPDIF module. When the soft reset completes, this bit is cleared automatically.

```
static int32_t spdif_soft_reset(audio_ctrl_p ctrl)
/*!
 * Get the spdif's settings.
 * @param       ctrl    a pointer of audio controller (audio_ctrl_t) that presents the spdif
module
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
```

18.6.2 Dumping readable SPDIF registers

This function dumps all readable SPDIF registers.

```
/*!
 * Dump spdif readable registers.
 * @param       ctrl    a pointer of audio controller (audio_ctrl_t) that presents the spdif
module
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
static int32_t spdif_dump(audio_ctrl_p ctrl)
/*!
 * Put the spdif to soft-reset mode, and then can be configured.
 * @param       ctrl    a pointer of audio controller (audio_ctrl_t) that presents the spdif
module
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
```

18.6.3 Obtaining SPDIF setting and status

The function can be called after SPDIF has been initialized.

```
static uint32_t spdif_get_hw_setting(audio_ctrl_p ctrl, uint32_t type)
/*!
 * Calucate the spdif's tx clock divider according the sample rate.
 * @param       ctrl    a pointer of audio controller(audio_ctrl_t) that presents the spdif
module
 *              sample_rate     sample rate to be set
 *
 * @return      the divider value
 */
static uint32_t spdif_cal_txclk_div(audio_ctrl_p ctrl, uint32_t sample_rate)
```

It returns the SPDIF setting values and status values according to the setting type:

- SPDIF_GET_FREQMEAS = 0
- SPDIF_GET_GAIL_SEL
- SPDIF_GET_RX_CCHANNEL_INFO_H
- SPDIF_GET_RX_CCHANNEL_INFO_L
- SPDIF_GET_RX_UCHANNEL_INFO
- SPDIF_GET_RX_QCHANNEL_INFO
- SPDIF_GET_INT_STATUS

18.6.4 Initializing SPDIF

Before use, SPDIF module must be initialized. Initialization requires the following:

- IOMUX setting for SPDIF signals.
- Clock setting, such as selecting the clock source, gating on clocks for SPDIF.
- Resetting the SPDIF module.

This function can be called to initialize the SPDIF module.

```
/*!
 * Initialize the spdif module and set the spdif to default status.
 * This function will be called by the snd_card driver.
 *
 * @param       priv    a pointer passed by audio card driver, spdif driver should change it
 *          to an audio_ctrl_p pointer that presents the spdif controller.
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
int32_t spdif_init(void *priv)
```

18.6.5 Configuring SPDIF

The function configures the SPDIF parameters according to the audio_dev_para provided by the audio card driver. This function:

- Writes transmit channel data to SPDIF_STCSCH and SPDIF_STCSCL.
- Configures the FIFO mode, watermark, and other settings
- Sets the transmit clock rate according the audio'ssample rate

```
/*!
 * Configure the spdif module according to the parameters that were passed by audio_card
driver.
 *
 * @param       priv    a pointer passed by audio card driver, spdif driver should change it
 *                      to an audio_ctrl_p pointer that presents the spdif controller.
 *              para    a pointer passed by audio card driver, consists of configuration
parameters
 *                      for spdif controller.
 *
 * @return      0 if succeeded
 *              -1 if failed
*/
int32_t spdif_config(void *priv, audio_dev_para_p para)
```

18.6.6 Playback through SPDIF

After initialization and configuration, data can be written to SPDIF_STL and SPDIF_STR in interleaved order to play back audio. SPDIF_SIS[TX_EMPTY] is continuously polled to determine whether TX FIFO is full. If TX FIFO is not full, data can be written to it with the following function.

```
/*!
 * Write datas to the spdif fifo in polling mode.
 * @param       priv    a pointer passed by audio card driver, spdif driver should change it
 *                      to a audio_ctrl_p pointer which presents the spdif controller.
 *      buf points to the buffer which hold the data to be written to the spdif tx fifo
 *      size    the size of the buffer pointed by buf.
 *      bytes_written   bytes be written to the spdif tx fifo
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
int32_t spdif_write_fifo(void *priv, uint8_t * buf, uint32_t size, uint32_t * bytes_written)
```

18.6.7 De-initializing SPDIF

This function de-initializes SPDIF and frees the resources that SPDIF uses.

```
/*!
 * Close the spdif module
 * @param       priv    a pointer passed by audio card driver, spdif driver should change it
 *                      to a audio_ctrl_p pointer which presents the spdif controller.
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
```

18.7 Testing the SPDIF driver

The SPDIF test unit demonstrates how to play back music using the audio framework. The test unit works as follows:

1. Initialize snd_card_spdif
2. Configure snd_card_spdif
3. Write the music file to snd_card_spdif, that is, play back music.
4. If "exit" is selected by the user, de-initialize snd_card_spdif and return

To build the SPDIF test, the SDK uses the command: ./tools/build_sdk -target mx61 -board evb -board_rev 1 -test audio

This generates the following ELF and binary files:

```
output/mx53/bin/mx61evb-audio-sdk.elf
output/mx53/bin/mx61evb-audio-sdk.bin
```

To run the test:

1. Download mx61ard-audio-sdk.elf using RV-ICE or Lauterbach or burn mx61ard-audio-sdk.bin to SD card with the following command in Windows's Command Prompt window:

cfimager-imx -o 0 -f mx61evb-audio-sdk.bin -d g:*(SD drive name in your PC)*

1. Ensure the following:
2.
   - The EVB board is mounted on the MX6QVPC board.
   - A rework was done to connect TP6[SPDIF_OUT] with PORT2_P98 on the MX6QVPC board.
   - The SPDIF_OUT socket and your PC are connected using a SPDIF recording device, such as M-AUDIO.
3. Power-up the board
4. Run the test by selecting "spdif playback" according to the prompt in the terminal.

When playback is finished, a record file should be generated on your PC.

18.8 Source code structure

Table 18-2. Source code file locataions

Description	Location
Drivers	

Table continues on the next page...

Table 18-2. Source code file locataions (continued)

Description	Location
Low-level driver for SPDIF module	src/sdk/audio/drv/imx-spdif.c
Audio card driver	src/sdk/audio/drv/snd_card.c
Low-level header file for SSI	src/sdk/audio/inc/imx-spdif.h
Data structures for audio framework	src/sdk/audio/inc/audio.h
Unit test	
Test file for all audio modules (such as SSI, ESAI, and SPDIF)	src/sdk/audio/test/audio_test.c
SPDIF unit test	src/sdk/audio/test/spdif_playback.c
PCM music file for playback	src/sdk/audio/test/wav_data.data

Chapter 19
Using the SNVS RTC/SRTC Driver

19.1 Introduction

SNVS is partitioned into two sections: a low power part (SNVS_LP) and a high power part (SNVS_HP).

The SNVS_LP block is in the always powered up domain. It is isolated from the rest of the logic by means of isolation cells, which are library-instantiated cells that insure that the powered up logic is not corrupted when power goes down in the rest of the chip.

SNVS_LP has the following functional units:

- Secure non-rollover real time counter (STRC) with alarm
- Security-related functions (see the chip security reference manual for details)

SNVS_HP is in the chip power supply domain. SNVS_HP provides an interface between SNVS_LP and the rest of the system. Access to SNVS_LP registers can only be gained through the SNVS_HP when it is powered up according to access permission policy. See the chip security reference manual for details.

SNVS_HP has the following functional units:

- IP bus interface
- SNVS_LP interface
- Non-secure real time counter (RTC) with alarm and periodic interrupt
- Control and status registers
- Security-related functions (see the chip security reference manual for details)

This chapter explains how to use the SNVS RTC/SRTC driver, which demonstrates the use of the timer alarm and periodic interrupt features of the SNVS RTC/SRTC functions. Note that because the driver is loaded with the firmware library binary in a non-secured boot environment, the high assurance boot (HAB) configures the SNVS in non-secure mode. Therfore, features that require secure boot, such as programming the zeroizable master key or validating the one-time programmable master key, cannot be demonstrated unless the user signed the firmware library binary for secure boot authentication.

The single SNVS module is located on the memory map at: 020C C000h.

Refer to the SNVS chapter of the device reference manual for description of SNVS HP and LP registers and detailed documentation of the SNVS module.

19.2 Feature summary

The SNVS module does the following:

- Provides a non-volatile real-time clock maintained by a coin-cell during system power-down for use in both secure and non-secure platforms
- Protects the real-time clock against rollback attacks in time-sensitive protocols such as DRM and PKI
- Deters replay attacks in time-independent protocols such as certificate or firmware revocation
- Other security-related functions (see the chip security reference manual for details)

19.3 Modes of operation

SNVS operates in one of two modes of operation: system power-down and system power-up.

During system power-down, SNVS_HP is powered-down and SNVS_LP is powered from the backup power supply and is electrically isolated from the rest of the SoC. In this mode, SNVS_LP continues to keep its register values and monitor the SNVS_LP tamper detection inputs.

NOTE

Backup supply mode has not been tested and depends on the hardware used.

During system power-up, SNVS_HP and SNVS_LP are both powered-up and all SNVS functions are operational.

19.4 Clocks

Table 19-1. SNVS clock sources

Clock	What it does
System Peripheral Clock	This clock is used by the SNVS internal logic, e.g. System Security Monitor. This clock can be gated outside of the module when SNVS indicates that it is not in use.
System IP Bus Access Clock	This clock is used by SNVS for clocking its registers during read/write accesses. This clock is active only during IP Bus access cycle. This clock is synchronized with System Peripheral Clock.
LP SRTC Clock	This clock is used by the secure real time counter.
HP RTC Clock	This clock is used by the real time counter.

NOTE

The counters for RTC and SRTC are incremented by the low frequency clock from the 32 KHz oscillator, which is asynchronous to the system clock.

19.5 Counters

SNVS has the following counters:

- Non-secured real time counter (RTC)
- Secured real time counter (SRTC)

NOTE

The driver does not demonstrate the clock calibration capability of the RTC and SRTC.

19.5.1 Non-Secured Real Time Counter

The SNVS_HP has an autonomous non-secured real time counter. The counter is not active and is reset when the system is powered down. The HP RTC can be used by any application and it has no privileged software access restrictions. The counter can be synchronized with the SNVS_LP SRTC by setting the HP_TS bit of SNVS_HP Control Register.

19.5.1.1 Non-Secured Real Time Counter Alarm

The SNVS_HP non-secure Real Time Counter has its own Time Alarm register. This register can be updated by any application. The SNVS_HP time alarm can generate interrupts to alert the host processor and can wake-up the host processor from one of its low-power modes (e.g. wait, doze, and stop). Note that this alarm cannot wake-up the entire system if it is powered off since this alarm would also be powered off.

19.5.1.2 Non-Secured Real Periodic Interrupt

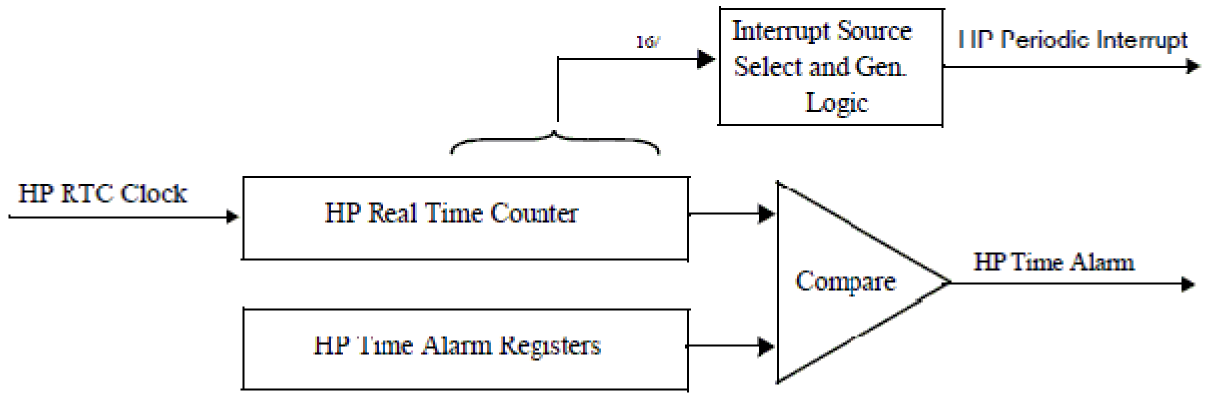


Figure 19-1. SNVS_HP real time counter, alarm, and interrupts

The SNVS_HP non-secure Real Time Counter incorporates periodic interrupt. The periodic interrupt is generated when a zero-to-one or one-to-zero transition occurs on the selected bit of the Real Time Counter. The periodic interrupt source is chosen from 16 bits of the HP Real Time Counter according to the PI_FREQ field setting in the HP Control Register. The frequency of the periodic interrupt is also defined by this bit selection.

SNVS_HP Real Time Counter and its interrupts are shown in Figure 19-1 2.

19.5.2 Secure Real Time Counter

The SNVS_LP incorporates an autonomous Secure Real Time Counter (SRTC). This is a non-rollover counter. This means that if the SRTC reaches the maximum value of all ones it will not rollover. In this case a time rollover indication is generated to the SNVS_LP Tamper Monitor, which can generates security violation and interrupt.

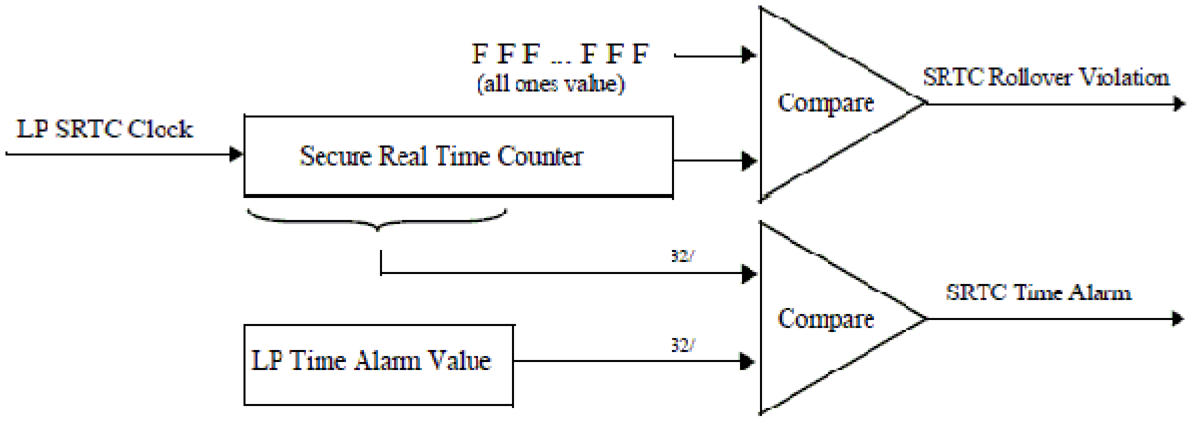


Figure 19-2. SNVS_LP secure real time counter

The SNVS_LP section has its own 32-bit Time Alarm Register. Time Alarm is generated when SRTC 32-most significant bits match with Time Alarm Register. The SNVS_LP time alarm can generate an interrupt to alert the host processor and can wake-up the host processor from one of its low-power modes (e.g. wait, doze, stop). This alarm can also wake-up the entire system in the power-down mode by asserting the wake-up external output signal.

19.6 Driver API

This driver has the following categories of APIs:

- SNVS lower level driver APIs
- RTC upper layer driver APIs
- SRTC upper layer driver APIs

19.6.1 SNVS lower level driver APIs

These low level driver API are defined in snvs.c file and are called by upper layer driver API in rtc.c and srtc.c files. These API reads or programs SNVS registers.

19.6.1.1 Enable/Disable SNVS non-secured real time counter

The API snvs_rtc_counter is used to enable or disable non-secure real time counter. The API either sets or clears RTC_EN bit of SNVS_HP control register. The API loops until the value of the register changed to the new value.

```
/*!
 * Enable or disable non-secured real time counter
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   state - 1 to enable the counter and any other value to disable it.
 */
void snvs_rtc_counter(struct hw_module *port, uint8_t state)
{
    volatile struct mx_snvs *psnvs =
        (volatile struct mx_snvs *)port->base;
    if( state == ENABLE )
    {
        /* Set RTC_EN bit in hpcr register */
        psnvs->hpcr |= HPCR_RTC_EN;
        /* Wait until the bit is set */
        while((psnvs->hpcr & HPCR_RTC_EN) == 0);
    }
    else
    {
        /* Clear RTC_EN bit in hpcr register */
        psnvs->hpcr &= ~HPCR_RTC_EN;
        /* Wait until the bit is cleared */
        while(psnvs->hpcr & HPCR_RTC_EN);
    }
}
```

19.6.1.2 Enable/Disable SNVS non-secured time alarm

The API snvs_rtc_alarm is used to enable or disable non-secure time alarm. The API either sets or clears HPTA_EN bit of SNVS_HP control register. The API loops until the value of the register changed to the new value.

```
/*!
 * Enable or disable non-secured time alarm
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   state - 1 to enable the alarm and any other value to disable it.
 */
void snvs_rtc_alarm(struct hw_module *port, uint8_t state)
{
    volatile struct mx_snvs *psnvs = (volatile struct mx_snvs *)port->base;
    if( state == ENABLE )
    {
        /* Set HPTA_EN bit of hpcr register */
        psnvs->hpcr |= HPCR_HPTA_EN;
        /* Wait until the bit is set */
        while((psnvs->hpcr & HPCR_HPTA_EN) == 0);
    }
    else
    {
        /* Clear HPTA_EN bit of hpcr register */
        psnvs->hpcr &= ~HPCR_HPTA_EN;
        /* Wait until the bit is cleared */
        while(psnvs->hpcr & HPCR_HPTA_EN);
    }
}
```

19.6.1.3 Enable/Disable SNVS periodic interrupt

The API snvs_rtc_periodic_interrupt is used to enable or disable non-secure periodic interrupt. The API either sets or clears PI_EN bit of SNVS_HP control register. The API loops until the value of the register changed to the new value. The API also needs the freq parameter to program PI_FREQ bits of HP control register. PI_FREQ can be any value from 0 to 15. CPU is interrupted whenever real time counter value at bit PI_FREQ toggles.

```
/*!
 * Enable or disable non-secured periodic interrupt
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   freq - frequence for periodic interrupt, valid values 0 to 15,
 *          a value greater than 15 will be regarded as 15.
 *
 * @param   state - 1 to enable the alarm and any other value to disable it.
 */
void snvs_rtc_periodic_interrupt(struct hw_module *port, uint8_t freq, uint8_t state)
{
    volatile struct mx_snvs *psnvs = (volatile struct mx_snvs *)port->base;
    if( state == ENABLE )
    {
        if( freq > 15 )
            freq = 15;
        /* First clear the periodic interrupt frequency bits */
        psnvs->hpcr &= ~HPCR_PI_FREQ_MASK;
        /* Set freq, SNVS interrupts the CPU whenever the
         * frequency (0-15) bit of RTC counter toggles.
         * The counter is incremented by the slow 32KHz clock.
         */
        psnvs->hpcr |= ((freq << HPCR_PI_FREQ_SHIFT) & HPCR_PI_FREQ_MASK);
        psnvs->hpcr |= HPCR_PI_EN;
        while((psnvs->hpcr & HPCR_PI_EN) == 0);
    }
    else
    {
        /* Clear freq and PI_EN bit to disable periodic interrupt */
        psnvs->hpcr &= ~HPCR_PI_FREQ_MASK;
        psnvs->hpcr &= ~HPCR_PI_EN;
        while(psnvs->hpcr & HPCR_PI_EN);
    }
}
```

19.6.1.4 Set SNVS non-secure real time counter registers

The API snvs_rtc_set_counter sets the 47-bit real time counter, it sets lower 32-bit of 64-bit argument count to HPRTCLR register and next 15 bits to HPRTCMR register. The function disables the RTC before changing the value of the counter so that the change can take effect.

```
/*!
 * Programs non-secured real time counter
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   count - 64-bit integer to program into 47-bit RTC counter register;
 *          only 47-bit LSB will be used
 */
void snvs_rtc_set_counter(struct hw_module *port, uint64_t count)
{
    volatile struct mx_snvs *psnvs = (volatile struct mx_snvs *)port->base;
    /* Disable RTC otherwise below write operation to counter registers
     * will not work
     */
    snvs_rtc_counter(port, DISABLE);
    /* Program the counter */
    psnvs->hprtclr = (uint32_t)count;
    psnvs->hprtcmr = (uint32_t)(count >> 32);
    /* Reenable RTC */
    snvs_rtc_counter(port, ENABLE);
}
```

19.6.1.5 Set SNVS non-secure RTC time alarm registers

The API snvs_rtc_set_alarm_timeout sets least significant 47-bits of timeout argument to time alarm registers. It sets lower 32-bits of 64-bit argument timeout to HPTALR register and next 15 bits to hptamr register. The function disables the RTC alarm function before changing the value of the alarm register to comply with the SNVS specifications as described in the chip reference manual. The CPU will be interrupt by SNVS when the value of counter register matches the alarm register value, the alarm is also indicated by setting of bit HPTA of status register (hpsr).

```
/*!
 * Sets non-secured RTC time alarm register
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   timeout - 64-bit integer to program into 47-bit time alarm register;
 *          only 47-bit LSB will be used
 */
void snvs_rtc_set_alarm_timeout(struct hw_module *port, uint64_t timeout)
{
    volatile struct mx_snvs *psnvs = (volatile struct mx_snvs *)port->base;
    /* Disable alarm */
    snvs_rtc_alarm(port, DISABLE);
    /* Program time alarm registers */
    psnvs->hptalr = (uint32_t)timeout;
    psnvs->hptamr = (uint32_t)(timeout >> 32);
    /* Reenable alarm */
    snvs_rtc_alarm(port, ENABLE);
}
```

19.6.1.6 Enable/Disable SNVS secure real time counter

The API snvs_rtc_counter is used to enable or disable secure real time counter. The API can set or clear RTC_EN bit of SNVS_HP control register. The API loops until the value of the register changed to the new value.

```
/*!
 * Enable or disable secure real time counter
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   state - 1 to enable the counter and any other value to disable it.
 */
void snvs_srtc_counter(struct hw_module *port, uint8_t state)
{
    volatile struct mx_snvs *psnvs = (volatile struct mx_snvs *)port->base;
    if(state == ENABLE)
    {
        psnvs->lpcr |= LPCR_RTC_EN;
        while((psnvs->lpcr & LPCR_RTC_EN) == 0);
    }
    else
    {
        psnvs->lpcr &= ~LPCR_RTC_EN;
        while(psnvs->lpcr & LPCR_RTC_EN);
    }
}
```

19.6.1.7 Enable/Disable SNVS secure time alarm

The API snvs_rtc_alarm is used to enable or disable secure time alarm. The API either sets or clears HPTA_EN bit of SNVS_HP control register. The API loops until the value of the register changed to the new value.

```
/*!
 * Enable or disable secure time alarm
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   state - 1 to enable the alarm and any other value to disable it.
 */
void snvs_srtc_alarm(struct hw_module *port, uint8_t state)
{
    volatile struct mx_snvs *psnvs = (volatile struct mx_snvs *)port->base;
    if(state == ENABLE)
    {
        psnvs->lpcr |= LPCR_LPTA_EN;
        while((psnvs->lpcr & LPCR_LPTA_EN) == 0);
    }
    else
    {
        psnvs->lpcr &= ~LPCR_LPTA_EN;
        while(psnvs->lpcr & LPCR_LPTA_EN);
    }
}
```

19.6.1.8 Set SNVS secured real time counter registers

The API snvs_srtc_set_counter sets the 47-bit real time counter, it sets lower 32-bit of 64-bit argument count to the LPRTCLR register and next 15 bits to the LPRTCMR register. The function disables the SRTC before changing the value of the counter so that the change can take effect.

```
/*!
 * Programs secure real time counter
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   count - 64-bit integer to program into 47-bit SRTC counter register;
 *          only 47-bit LSB will be used
 */
void snvs_srtc_set_counter(struct hw_module *port, uint64_t count)
{
    volatile struct mx_snvs *psnvs = (volatile struct mx_snvs *)port->base;
    /* Disable RTC */
    snvs_srtc_counter(port, DISABLE);
    /* Program the counter */
    psnvs->lpsrtclr = (uint32_t)count;
    psnvs->lpsrtcmr = (uint32_t)(count >> 32);
    /* Reenable RTC */
    snvs_srtc_counter(port, ENABLE);
}
```

19.6.1.9 Set SNVS non-secure time alarm register

The API snvs_rtc_set_alarm_timeout sets 32-bit timeout argument to 32-bit time alarm register. The function disables the RTC alarm function before changing the value of the alarm register to comply with the SNVS specifications as described in the chip reference manual. The CPU is interrupted by SNVS when the 32 MSB of counter register matches the alarm register value. The alarm is also indicated by the setting of bit LPTA of status register (LPSR).

```
/*!
 * Set secured RTC time alarm register
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   timeout - 32-bit integer to program into 32-bit time alarm register;
 */
void snvs_srtc_set_alarm_timeout(struct hw_module *port, uint32_t timeout)
{
    volatile struct mx_snvs *psnvs = (volatile struct mx_snvs *)port->base;
    /* Disable alarm */
    snvs_srtc_alarm(port, DISABLE);
    /* Program time alarm register */
    psnvs->lptar = timeout;
    /* Reenable alarm */
    snvs_srtc_alarm(port, ENABLE);
}
```

19.6.2 RTC upper layer driver APIs

The upper layer API calls into lower layer SNVS API to perform tasks like setting up periodic alarm to periodically interrupt the CPU, set up one-time alarm and also accepts callback routine to callback to higher layer application (test application) function from interrupt service routine.

19.6.2.1 Initialize RTC

This API will be called from application layer (unit test) to start the RTC counter and prepare to service requests to set one-time alarm or periodic time alarm

```
/*!
 * Initializes RTC by enabling non-secured real time counter,
 * disables alarm and periodic interrupt. It also calls internal
 * API snvs_rtc_setup_interrupt to register interrupt service handler
 */
void rtc_init(void)
{
    /* Initialize SNVS driver */
    snvs_init(snvs_rtc_module.port);
    /* Start rtc counter */
    snvs_rtc_counter(snvs_rtc_module.port, ENABLE);
    /* Keeps alarms disabled */
    snvs_rtc_alarm(snvs_rtc_module.port, DISABLE);
    snvs_rtc_periodic_interrupt(snvs_rtc_module.port, 0, DISABLE);
    /* Enable interrupt */
    snvs_rtc_setup_interrupt(snvs_rtc_module.port, ENABLE);
}
```

19.6.2.2 De-initialize RTC

This API will be called from application layer (like unit test code) to disable the real time counter.

```
/*!
 * Disables interrupt, counter, alarm and periodic alarm
 */
void rtc_deinit(void)
{
    /* Disable the interrupt */
    snvs_rtc_setup_interrupt(snvs_rtc_module.port, DISABLE);
    snvs_rtc_module.onetime_timer_callback = NULL;
    snvs_rtc_module.periodic_timer_callback = NULL;

    /* Disable the counter and alarms*/
    snvs_rtc_counter(snvs_rtc_module.port, DISABLE);
    snvs_rtc_alarm(snvs_rtc_module.port, DISABLE);
    snvs_rtc_periodic_interrupt(snvs_rtc_module.port, 0, DISABLE);
    /* Deinitialize SNVS */
    snvs_deinit(snvs_rtc_module.port);
}
```

19.6.2.3 Setup RTC one time alarm

This API will be called from application layer (like the unit test code) to set up the one time alarm, using non-secured RTC. The caller to supply the pointer to callback function. Callback will be called from interrupt service routine when SNVS interrupts CPU when alarm sets off.

```
/*!
 * Calls in appropriate low level API to setup one-time timer
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   callback - callback function to be called from isr.
 */
void rtc_setup_onetime_timer(uint64_t timeout, funct_t callback)
{
    /* Disables interrupt */
    snvs_rtc_setup_interrupt(snvs_rtc_module.port, DISABLE);
    /* Set secure real time counter to 0 */
    snvs_rtc_set_counter(snvs_rtc_module.port, 0);
    /* Disables interrupt */
    snvs_rtc_set_alarm_timeout(snvs_rtc_module.port, timeout);
    /* Set callback pointer */
    snvs_rtc_module.onetime_timer_callback = callback;
    /* Enable the interrupt */
    snvs_rtc_setup_interrupt(snvs_rtc_module.port, ENABLE);
}
```

19.6.2.4 Setup RTC periodic time alarm

This API will be called from application layer (like unit test code) to setup periodic time alarm using non-secured RTC. The caller to supply the pointer to callback function. Callback will be called from interrupt service routine whenever SNVS interrupts CPU when periodic alarm sets off.

```
/*!
 * Calls in appropriate low level API to setup periodic timer
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   periodic_bit - periodic interrupt freq (valid values 0-15)
 *
 * @param   callback - pointer to callback function
 */
void rtc_setup_periodic_timer(uint32_t periodic_bit, funct_t callback)
{
    /* Disable interrupt */
    snvs_rtc_setup_interrupt(snvs_rtc_module.port, DISABLE);

    /* Disable periodic interrupt */
    snvs_rtc_periodic_interrupt(snvs_rtc_module.port, 0, DISABLE);
    /* Set the callback pointer */
    snvs_rtc_module.periodic_timer_callback = callback;
    /* Enable counter and periodic interrupt */
    snvs_rtc_counter(snvs_rtc_module.port, ENABLE);
    snvs_rtc_periodic_interrupt(snvs_rtc_module.port, periodic_bit, ENABLE);
    /* Enable interrupt */
    snvs_rtc_setup_interrupt(snvs_rtc_module.port, ENABLE);
}
```

19.6.2.5 Disable RTC periodic alarm

This API will be called from application layer (like unit test code) to disable periodic alarm. In our example unit test the callback function to periodic alarm counts upto 10 periodic alarm interrupts and calls this API to disable periodic time alarm.

```
/*!
 * Calls in appropriate low level API to disable periodic timer
 */
```

```
void rtc_disable_periodic_timer(void)
{
    /* Disable interrupts */
    snvs_rtc_setup_interrupt(snvs_rtc_module.port, DISABLE);
    /* Disable RTC periodic interrupt */
    snvs_rtc_periodic_interrupt(snvs_rtc_module.port, 0, DISABLE);
    /* Remove callback */
    snvs_rtc_module.periodic_timer_callback = NULL;
    /* Reenable interrupts */
    snvs_rtc_setup_interrupt(snvs_rtc_module.port, ENABLE);
}
```

19.6.3 SRTC upper layer driver APIs

The upper layer API calls into lower layer SNVS API to perform tasks like setting up one-time alarm and also accepts callback routine to callback to higher layer application (unit test) function from interrupt service routine.

19.6.4 Initialize SRTC

This API will be called from application layer (unit test) to start SRTC counter and prepare to service requests to set one-time alarm.

```
/*!
 * Initializes SRTC by enabling secure real time counter and
 * disables time alarm. It also calls internal API snvs_rtc_setup_interrupt
 * to register interrupt service handler
 */
void srtc_init(void)
{
    /* Initialize SNVS driver */
    snvs_init(snvs_srtc_module.port);
    /* Start SRTC counter */
    snvs_srtc_counter(snvs_srtc_module.port, ENABLE);
    /* Keep time alarm disabled */
    snvs_srtc_alarm(snvs_srtc_module.port, DISABLE);
}
```

19.6.5 De-initialize SRTC

This API will be called from application layer (like unit test code) to disable the secure real time counter.

```
/*!
 * Disables interrupt, counter and time alarm
 */
void srtc_deinit(void)
{
    /* Disable the interrupt */
    snvs_srtc_setup_interrupt(snvs_srtc_module.port, DISABLE);
    /* Disable the counter */
    snvs_srtc_counter(snvs_srtc_module.port, DISABLE);
    snvs_srtc_alarm(snvs_srtc_module.port, DISABLE);
    /* Deinitialize SNVS */
    snvs_deinit(snvs_srtc_module.port);
}
```

19.6.6 Setup SRTC one time alarm

This API will be called from application layer (like unit test code) to setup one time alarm using secured RTC. The caller to supply the pointer to callback function. Callback will be called from interrupt service routine when SNVS interrupts CPU when alarm sets off.

```
/*!
 * Calls in appropriate low level API to setup SRTC one-time timer
 *
 * @param   port - pointer to the SNVS module structure.
 *
 * @param   callback - callback function to be called from isr.
 */
void srtc_setup_onetime_timer(uint32_t timeout, funct_t callback)
{
    /* Disables the interrupt */
    snvs_srtc_setup_interrupt(snvs_srtc_module.port, DISABLE);
    /* Clear the SRTC counter */
    snvs_srtc_set_counter(snvs_srtc_module.port, 0);
    /* Program the timeout value */
    snvs_srtc_set_alarm_timeout(snvs_srtc_module.port, timeout);
    /* Set the callback function */
    snvs_srtc_module.onetime_timer_callback = callback;
    /* Reanable the interrupt */
    snvs_srtc_setup_interrupt(snvs_srtc_module.port, ENABLE);
}
```

19.6.7 Testing the SNVS SRTC/RTC driver

There are two separate unit tests included with platform SDK code:

- snvs_rtc_test.c-demonstrates how to call into RTC driver API to setup one-time and periodic time alarms
- snvs_srtc_test_.c-demonstrates how to call into SRTC driver API to setup one-time alarm.

These unit tests demonstrate the use of callback function passed as a pointer to RTC and SRTC API and driver's interrupt service routine calls these callback function whenever an alarm is set.

Here is one example of unit test implementation that calls RTC driver API, the function initialized RTC and calls API to setup one time timer and pass in pointer to callback function. When alarm occurs the driver's interrupt service routine will call the callback function. The unit test function waits in a loop for global onetime_tick to be set by the callback function and once it is set it breaks off from the loop and sends text on uart indicating the test has passed otherwise if loop counter reaches 0 the test function will send text on uart to indicate the test failed.

```
void one_time_timer_test(void)
{
    int loop = 0xFFFFFF;
    onetime_tick = 0;
    rtc_init();

    rtc_setup_onetime_timer(10, one_time_tick_callback);
    while(loop--)
    {
        if(onetime_tick)
            break;
    }
    if(onetime_tick == 0)
        printf( "SNVS RTC Timer Test Failed!!\n");
    else
        printf( "SNVS RTC Timer Test Passed!!\n");
    rtc_deinit();
}
```

Below shows an implementation of callback one_time_tick_callback function where in it initializes global variable onetime_tick indicating that one time alarm was successful and the test function can break from the wait loop.

```
void one_time_tick_callback(void)
{
    onetime_tick = 1;
}
```

19.7 Source code structure

The driver code is divided into three files:

- snvs.c file provides low level API to enable or disable secure or non-secure counter, time alarm and periodic interrupt of SNVS HP and LP.
- rtc.c file provde API at a higher level to setup one-time timer using non-secure real time counter and setup periodic interrupt.
- srtc.c file provide API at a higher level to setup one-time timer using secure real time counter and setup periodic interrupt.

Table 19-2. Source code file locations

Description	Location
Driver	
Low-level driver source	./src/sdk/snvs/drv/snvs.c
Driver source	./src/sdk/timer/drv/imx_timer/rtc.c
Driver source	./src/sdk/timer/drv/imx_timer/srtc.c

Table continues on the next page...

Table 19-2. Source code file locations (continued)

Description	Location
Driver header	`./src/include/mx61/snvs.h`
Driver header	`./src/include/mx61/rtc.h`
Low-level driver header	`./src/include/mx61/srtc.h`

Unit tests	`./src/sdk/timer/test/snvs_rtc_test.c`
	`./src/sdk/timer/test/snvs_srtc_test.c`
	`./src/sdk/timer/test/snvs_rtc_test.h`
	`./src/sdk/timer/test/snvs_srtc_test.h`

Chapter 20
Configuring the SSI Driver

20.1 Overview

This chapter explains how to configure the synchronous serial interface (SSI) driver.

The synchronous serial interface (SSI) is a full-duplex, serial port that allows the chip to communicate with serial devices such as standard coder-decoders (CODECs), digital signal processors (DSPs), microprocessors, peripherals, or popular industry audio CODECs that implement the inter-IC sound bus standard (I2S) and Intel AC97 standard.

SSI typically transfers samples in a periodic manner. The SSI consists of independent transmitter and receiver sections with independent clock generation and frame synchronization.

The following figure illustrates the SSI organization.

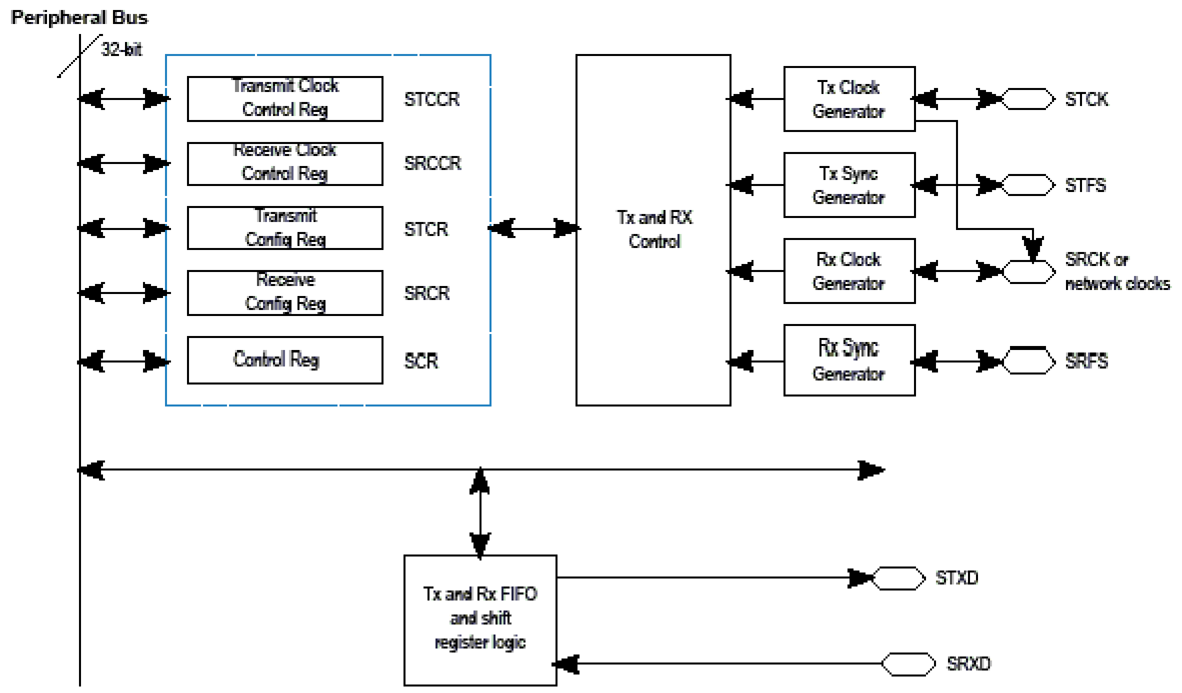


Figure 20-1. SSI block diagram

The SSI consists of:

- Control registers to set up the port
- A status register
- Two sets of transmit and receive FIFOs. Each of the four FIFOs is 15 x 32 bits. The two sets of Tx/Rx FIFOs can be used in network mode to provide two independent channels for transmission and reception. The second set of Tx and Rx FIFOs replicates the logic used for the first set of FIFOs.
- Separate serial clock and frame sync generation for transmit and receive sections

There are three SSI modules within the i.MX 6Dual/6Quad processor: SSI1, SSI2, and SSI3. The SSI signals connect to the AUDMUX, which is another module within the i.MX 6Dual/6Quad chip. For AUDMUX details, see AUDMUX.

20.2 Features summary

The SSI driver supports the following features:

- A simple framework for audio
- SSI driver supporting I2S protocol
- sgtl5000 driver supporting I2S protocol

20.3 Clocks

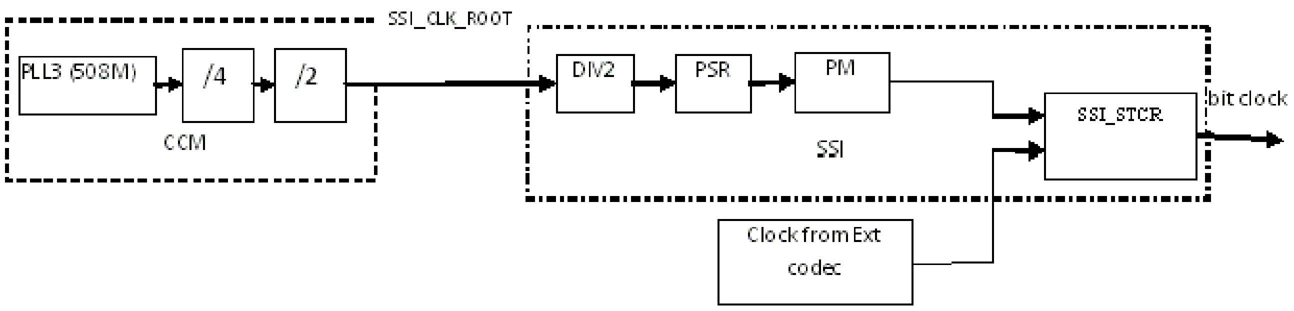


Figure 20-2. SSI clock tree

Before using SSI, gate ssi_ipg and ssi_ssi_clk on CCM_CCGR5 as follows:

- For SSI1, set CCM_CCGR5[CG9]
- For SSI2, set CCM_CCGR5[CG10]
- For SSI3, set CCM_CCGR5[CG11]

Refer to the Clock Controller Module (CCM) chapter in the i.MX 6Dual/6Quad reference manual for details.

By default, the SSI_CLK_ROOT is sourced from PLL3 which is 508 MHz. When the default ssi_clk_pred value (default 4) and ssi_clk_podf value (default 2) are used, the ssi_clk_root is divided to 63.5 MHz.

NOTE

The SSI_CLK_ROOT is the source clock for other modules, such as ESAI. Therefore, any change to SSI_CLK_ROOT can affect those modules. It is recommended that users use the default SSI_CLK_ROOT value (63.5 MHz) for the SSI module.

The bit clock (transmit bit clock or receive bit clock) can be internal (SSI_STCR[TXDIR] = 1b) or external (SSI_STCR[TXDIR] = 0b). When the bit clock is internal, it is sourced from SSI_CLK_ROOT and can be divided by SSI_STCCR[DIV2], SSI_STCCR[PSR], and SSI_STCCR[PM]. When external, the external audio codec provides the bit clock.

20.4 IOMUX pin mapping

The following figure shows the SSI signal routing.

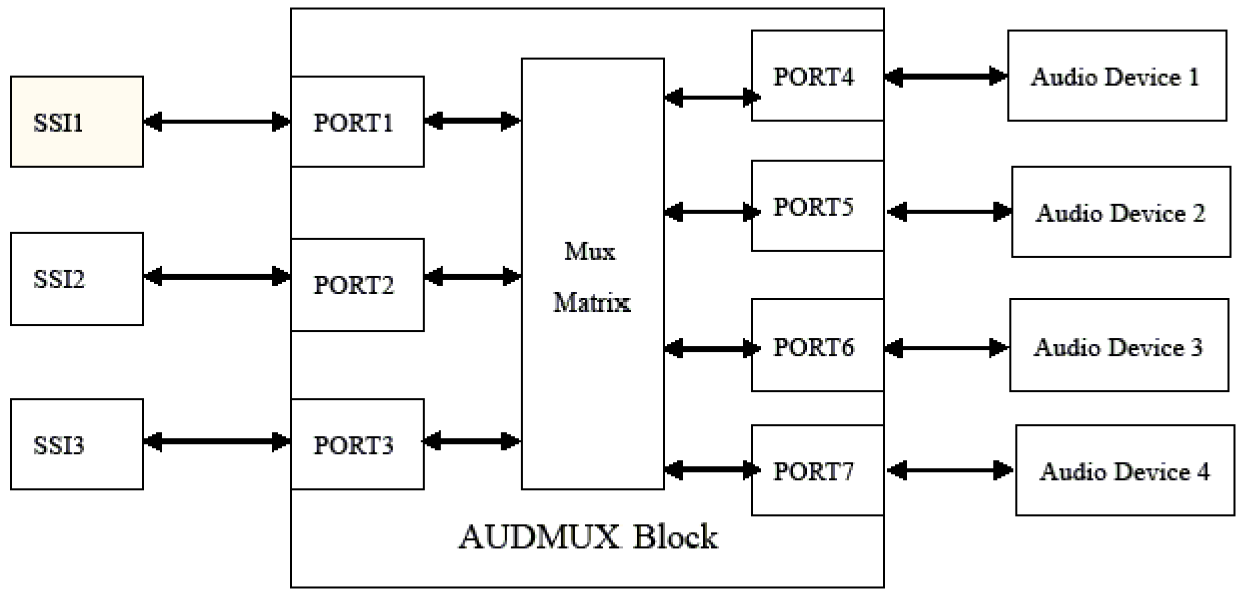


Figure 20-3. SSI signal routing

The SSI signals connect to the internal ports of the AUDMUX, which then routes them to the external pins. From there, the AUDMUX connects the signals to the external audio codec.

For i.MX53 SMD board, the PORT5 was connected with the SSI codec sgtl5000 in SYN mode, and the IOMUX pin configuration is listed in the table below

NOTE

The i.MX 6Dual/6Quad ARD Board does not support audio driver functionality. The i.MX53 SMD board is utilized for functional purposes. When using another board, please check the board schematic for correct pin assignments.

Table 20-1. IOMUX configuration of SSI2 on mx53-smd board

SSI Signal name	AUDMUX Signal name	Pin name	ALT
SSI2_SRXD	AUD5_RXD	KEY_ROW1	ALT2
SSI2_STXD	AUD5_TXD	KEY_ROW0	ALT2
SSI2_STXC	AUD5_TXC	KEY_COL0	ALT2
SSI2_STXFS	AUD5_TXFS	KEY_COL1	ALT2

20.5 Audio framework

The i.MX 6Dual/6Quad processor contains multiple audio controllers and audio codecs. The following three data structures are used to create an audio framework that abstracts all audio modules (controllers and codecs) and provides a uniform API for applications:

- audio_card_t-describes the audio card
- audio_ctrl_t-describes the audio controller (SSI, ESAI module, etc.)
- audio_codec_t-describes the audio codec (sgtl5000, cs428888, etc.)

The audio card consists of one audio controller and one audio codec. The audio_card_t data structureis the only data structure that an application can access and manage.

20.5.1 audio_card_t data structure

The audio_card_t data structure describes the audio card. It is as follows:

```
typedef struct {
    const char *name;
    audio_codec_p codec;             //audio codec which is included
    audio_ctrl_p ctrl;                   //audio controller which is included
    audio_dev_ops_p ops;           //APIs
} audio_card_t, *audio_card_p;
```

20.5.2 audio_ctrl_t data structure

The data structure audio_ctrl_t describes the audio controller. It is as follows:

```
typedef struct {
    const char *name;
    uint32_t base_addr;         // the io base address of the controller
    audio_bus_type_e bus_type;   //The bus type(ssi, esai or spdif) the controller supports
    audio_bus_mode_e bus_mode;     //the bus mode(master, slave or both)the controller
supports
    int irq;                       //the irq number
    int sdma_ch;                   //Will be used for SDMA
    audio_dev_ops_p ops;                      //APIs
} audio_ctrl_t, *audio_ctrl_p;
```

20.5.3 audio_codec_t data structure

The data structure audio_codec_t describes the audio codec. It is as follows:

```
typedef struct {
    const char *name;
    uint32_t i2c_base;          //the i2c connect with the codec
    uint32_t i2c_freq;          // i2c operate freq;
    uint32_t i2c_dev_addr;       //Device address for I2C bus
    audio_bus_type_e bus_type;   //The bus type(ssi, esai or spdif) the codec supports
    audio_bus_mode_e bus_mode;   //the bus mode(master, slave or both)the codec supports
    audio_dev_ops_p ops;                        //APIs
} audio_codec_t, *audio_codec_p;
```

20.5.4 audio_dev_ops_t data structure

The data structure audio_dev_ops_t describes the APIs of the codec, controller, and card. It is as follows:

```
 typedef struct {
    int (*init) (void *priv);
    int (*deinit) (void *priv);
    int (*config) (void *priv, audio_dev_para_p para);
    int (*ioctl) (void *priv, uint32_t cmd, void *para);
    int (*write) (void *priv, uint8_t * buf, uint32_t byte2write, uint32_t *bytewrittern);
    int (*read) (void *priv, uint8_t * buf, uint32_t byte2read, uint32_t byteread);
} audio_dev_ops_t, *audio_dev_ops_p;
```

20.5.5 audio_dev_para_t data structure

The data structure audio_dev_para_t describes the audio parameters to be passed to the configuration function. It is as follows:

```
typedef struct {
    audio_bus_mode_e bus_mode; //Master or slave
    audio_bus_protocol_e bus_protocol; //I2S, AC97 and so on
    audio_trans_dir_e trans_dir; //Tx, Rx or both
    audio_samplerate_e sample_rate; //32K, 44.1K , 48K, and so on
    audio_word_length_e word_length;
    unsigned int channel_number;
} audio_dev_para_t, *audio_dev_para_p;
```

For the imx53-smd board, sgtl5000 and SSI2 are used, so the SSI sound card should like:

```
audio_card_t snd_card_ssi = {
    .name = "i.MX SSI sound card",
    .codec = &sgtl5000,                          // the codec sgtl5000
    .ctrl = &imx_ssi_2,                              //For imx53_smd, the SSI2 was used.
    .ops = &snd_card_ops,
};
```

NOTE

The i.MX 6Dual/6Quad ARD Board does not support audio driver functionality. The i.MX53 SMD board is utilized for functional purposes. When using another board, please check the board schematic for correct pin assignments.

20.6 SSI driver functions

The SSI driver has both local functions and public APIs.

The local functions are used to:

- Reset the SSI
- Obtain the SSI setting and status values
- Set SSI parameters
- Enable SSI sub-modules

The public APIs are used to:

- Initialize the SSI driver
- Configure the SSI
- Playback through the SSI

20.6.1 Resetting the SSI

SSI_SCR[SSIEN] enables and disables the SSI. When the SSI is disabled, all SSI status bits are preset to the same state produced by the power-on reset. However, all control bits are unaffected because disabling the SSI puts it into self-reset mode and clears the contents of the Tx and Rx FIFOs.

When the SSI is disabled, all internal clocks except the register access clock are also disabled. The control registers should be modified on self-reset mode (SSI_SCR[SSIEN] = 0b).

20.6.2 Obtaining SSI setting and status values

The function uint32_t ssi_get_hw_setting(audio_ctrl_p ctrl, uint32_t type) returns the SSI setting and status values according to the setting type as follows:

```
typedef enum {
    SSI_SETTING_TX_FIFO1_DATAS_CNT,
    SSI_SETTING_TX_FIFO2_DATAS_CNT,
    SSI_SETTING_RX_FIFO1_DATAS_CNT,
    SSI_SETTING_RX_FIFO2_DATAS_CNT,
    SSI_SETTING_TX_WATERMARK,
    SSI_SETTING_RX_WATERMARK,
SSI_SETTING_TX_WORD_LEN,
SSI_SETTING_RX_WORD_LEN,
    SSI_SETTING_TX_FRAME_LENGTH,
    SSI_SETTING_RX_FRAME_LENGTH,
    SSI_SETTING_CLK_FS_DIR,
} ssi_setting_type_e;
```

The function can be called once SSI has been initialized.

20.6.3 Setting SSI parameters

The function static uint32_t ssi_set_hw_setting(audio_ctrl_p ctrl, uint32_t type, uint32_t val) sets SSI parameters according to setting type. The supported setting types are:

```
SSI_SETTING_TX_WATERMARK
SSI_SETTING_RX_WATERMARK
SSI_SETTING_TX_WORD_LEN
SSI_SETTING_RX_WORD_LEN
SSI_SETTING_TX_FRAME_LENGTH
SSI_SETTING_RX_FRAME_LENGTH
SSI_SETTING_TX_BIT_CLOCK
SSI_SETTING_RX_BIT_CLOCK
SSI_SETTING_CLK_FS_DIR
```

The function must be called when SSI is in self-reset mode (SCR[SSIEN] = 0).

20.6.4 Enabling SSI sub-modules

The SSI and its sub-modules can be enabled or disabled individually using the function static uint32_t ssi_hw_enable(audio_ctrl_p ctrl, uint32_t type, bool enable), which enables or disables SSI or its sub-modules according to enabling type as follows:

```
typedef enum {
    SSI_HW_ENABLE_SSI,
    SSI_HW_ENABLE_TX,
    SSI_HW_ENABLE_RX,
    SSI_HW_ENABLE_TXFIFO1,
    SSI_HW_ENABLE_TXFIFO2,
    SSI_HW_ENABLE_RXFIFO1,
    SSI_HW_ENABLE_RXFIFO2,
} ssi_hw_enable_type_e;
```

20.6.5 Initializing the SSI driver

Before using, initialize the SSI module as follows:

- Configure the IOMUX for external SSI signals.
- Configure the clock, including selecting the clock source and gating on clocks for SSI. Enable the external oscillator if SSI_CLK_ROOT is sourced from an external oscillator.
- Reset the SSI module and put all the registers into reset value.

The function int ssi_init(void *priv) can be called to initialize the SSI module.

20.6.6 Configuring the SSI

The function int ssi_config(void *priv, audio_dev_para_p para) configures the SSI parameters according to the descriptions in audio_dev_para. This function:

- Sets the direction of the bit clock and the frame sync clock (SSI_STCR[TXDIR] and SSI_STCR[TFDIR])
- Sets the attributes, such as polarity and frame sync length, of the bit clock and the frame sync clock.
- Sets bit clock dividers if an internal bit clock was used (SSI_STCCR[DIV2], SSI_STCCR[PSR], and SSI_STCCR[PM])
- Sets frame length (SSI_STCCR[DC])
- Sets word length (SSI_STCCR[WL])
- Sets FIFO's watermarks
- Enables SSI, FIFOs, and TX/RX

20.6.6.1 Playback through SSI

After initialization and configuration, data can be written to the SSI TX FIFO (SSI → stx0) to play back music. SSI_SFCSR[TFCNT0] polls to determine whether TX FIFO is full or not. If TX FIFO is not full, data can be written to it according to the word length (SSI_STCCR_WL).

20.7 sgtl5000 driver

The sgtl5000 is one of many codecs that have an SSI interface and thus can be used as an external audio codec for i.MX 6Dual/6Quad including sgtl5000. Refer to the sgtl5000 driver for details.

20.8 Testing the unit

The SSI test unit demonstrates how to playback music using the audio framework. The test unit works as follows:

```
audmux_route(AUDMUX_PORT_2, AUDMUX_PORT_5, AUDMUX_SSI_SLAVE);
Initialize the snd_card_ssi, which includes SSI2 and sgtl5000
Configure the snd_card_ssi
Write the music file to the snd_card_ssi, that is, playback music
If "exit" selected by the user, de-initialize the snd_card_ssi and return
```

To run the SSI test, the SDK uses the command:

```
./tools/build_sdk -target mx53 -board smd -board_rev 1 -test audio
```

The command generates the following binary and ELF files:

```
output/mx53/bin/mx53smd-audio-sdk.elf
output/mx53/bin/mx53smd-audio-sdk.bin
```

After the files have been generated, perform the following steps:

1. Download mx53smd-audio-sdk.elf using RV-ICE or Lauterbach or burn mx53smd-audio-sdk.bin to SD card with the following command in Windows's Command Prompt window:

   cfimager-imx -o 0 -f mx53smd-audio-sdk.bin -d g:(SD drive name in your PC)

2. Power up the board.
3. Select "ssi playback" according to the prompt in the terminal. This runs the SSI test unit.

   If the test passes, you will hear a voice in the headphones.

NOTE

The i.MX 6Dual/6Quad ARD Board does not support audio driver functionality. The i.MX53 SMD board is utilized for functional purposes. When using another board, please check the board schematic for correct pin assignments.

20.9 Functions

20.9.1 Local functions

```
/*!
 * Dump the ssi registers which can be readable.
 * @param       ctrl    a pointer of audio controller (audio_ctrl_t) which presents the ssi
 * module itself
 * @return      0 if succeeded
 *              -1 if failed
 */
static int ssi_dump(audio_ctrl_p ctrl)
/*!
 * Put the ssi to soft-reset mode, and then can be configured.
 * @param       ctrl    a pointer of audio controller(audio_ctrl_t) which presents the ssi
module
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
static int ssi_soft_reset(audio_ctrl_p ctrl)
/*!
```

```
 * Set all the registers to reset values, called by ssi_init.
 * @param       ctrl    a pointer of audio controller(audio_ctrl_t) which presents the ssi
module
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
static int ssi_registers_reset(audio_ctrl_p ctrl)
/*!
 * Get the ssi's settings.
 * @param       ctrl    a pointer of audio controller(audio_ctrl_t) which presents the ssi
module
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
static uint32_t ssi_get_hw_setting(audio_ctrl_p ctrl, uint32_t type)
/*!
 * Set the ssi's settings.
 * @param       ctrl    a pointer of audio controller(audio_ctrl_t) which presents the ssi
module
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
static uint32_t ssi_set_hw_setting(audio_ctrl_p ctrl, uint32_t type, uint32_t val)
```

20.9.2 APIs

```
/*!
 * Initialize the ssi module and set the ssi to default status.
 * This function will be called by the snd_card driver or application.
 *
 * @param       priv    a pointer passed by audio card driver, SSI driver should change it
 *                      to a audio_ctrl_p pointer which presents the SSI controller.
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
int ssi_init(void *priv)
/*!
 * Configure the SSI module according the parameters which was passed by audio_card driver.
 *
 * @param       priv    a pointer passed by audio card driver, SSI driver should change it
 *                      to a audio_ctrl_p pointer which presents the SSI controller.
 *              para    a pointer passed by audio card driver, consists of configuration
parameters
 *                      for SSI controller.
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
int ssi_config(void *priv, audio_dev_para_p para)
/*!
 * Write datas to the ssi fifo in polling mode.
 * @param       priv    a pointer passed by audio card driver, SSI driver should change it
 *                      to a audio_ctrl_p pointer which presents the SSI controller.
 *              buf     points to the buffer which hold the data to be written to the SSI tx
fifo
 *              size    the size of the buffer pointed by buf.
 *              bytes_written   bytes be written to the SSI tx fifo
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
int ssi_write_fifo(void *priv, uint8_t * buf, uint32_t size, uint32_t * bytes_written)
```

```
/*!
 * Close the SSI module
 * @param       priv    a pointer passed by audio card driver, SSI driver should change it
 *                      to a audio_ctrl_p pointer which presents the SSI controller.
 *
 * @return      0 if succeeded
 *              -1 if failed
 */
int ssi_deinit(void *priv)
```

20.10 Source code structure

Table 20-2. Source code file locations

Description	Location
Drivers	
Low-level driver for SSI	`src/sdk/audio/drv/imx-ssi.c`
Low-level driver for sgtl5000, an audio codec	`src/sdk/audio/drv/sgtl5000.c`
Audio card driver	`src/sdk/audio/drv/snd_card.c`
Low-level header file for SSI	`src/sdk/audio/inc/imx-ssi.h`
Low-level header file for sgtl5000	`src/sdk/audio/inc/sgtl5000.h`
Data structures for audio framework	`src/sdk/audio/inc/audio.h`
Unit Test	
Test file for all audio modules (such as SSI, ESAI, SPDIF)	`src/sdk/audio/test/audio_test.c`
SSI unit test	`src/sdk/audio/test/ssi_playback.c`
PCM music file for playback	`src/sdk/audio/test/wav_data.data`

Chapter 21
Configuring the UART Driver

21.1 Overview

This chapter explains how to configure the UART driver, which is a low-level driver that is able to handle most common uses of a RS-232 serial interface. All UART ports are controlled through this driver and all functions can be called from any place in the code.

The console/debug UART of the SDK is a usage example of this driver. Another example demonstrates the usage of the SDMA to transfer data through the UART port.

21.2 Features summary

The UART low-level driver supports:

- Interrupt-driven and SDMA-driven TX/RX of characters
- Various baud rates within the limit of the controller (4.0 Mbits/s), depending on its input clock
- Parity check and one/two stop bits
- 7-bit and 8-bit character lengths
- RTS/CTS hardware driven flow control

21.3 Modes of operation

The following table explains the UART modes of operation:

Table 21-1. Modes of operation

Mode	What it does	Related functions
DCE/DTE mode	UART can be configured for terminal mode (DTE) or device mode (DCE). The default mode is set to DCE (for example, when UART is used to output a message to a console). It is transparent from a software point of view.	-
Hardware flow control	RTS and CTS are entirely controlled by the UART. While the module allows them to be enabled or disabled, the driver does not allow using only RTS or CTS for single direction control.	The FIFO trigger level that controls CTS can be configured with the function uart_set_FIFO_mode()
DMA support	The driver allows setting the way the FIFOs are handled, though both the RX FIFO and TX FIFO could be managed by the SDMA. Above or below a certain watermark level, the FIFOs trigger a DMA request when there's sufficient data to retrieve or empty room. The watermark level of each FIFO can be set independently, and can also be enabled on only TX or RX. The external application configures the SDMA by calling the SDMA driver.	The function uart_set_FIFO_mode() allows the configuration of automatic DMA transfers on the UART side.
Interrupt support	The driver allows setting the way the FIFOs are handled, though both the RX FIFO and TX FIFO could be managed by interrupts. Above or below a certain watermark level, the FIFOs trigger an interrupt when there's sufficient data to retrieve or empty room. The watermark level of each FIFO can be set independently, and can also be enabled on only TX or RX. The external application is responsible for creating the interrupt subroutine. The address of this routine is passed through the structure hw_module, defined in .src/include/io.h. It is initialized by the application and used by the driver for various configurations.	The function uart_set_FIFO_mode() allows the configuration of the watermark level parameters.

21.4 Clocks

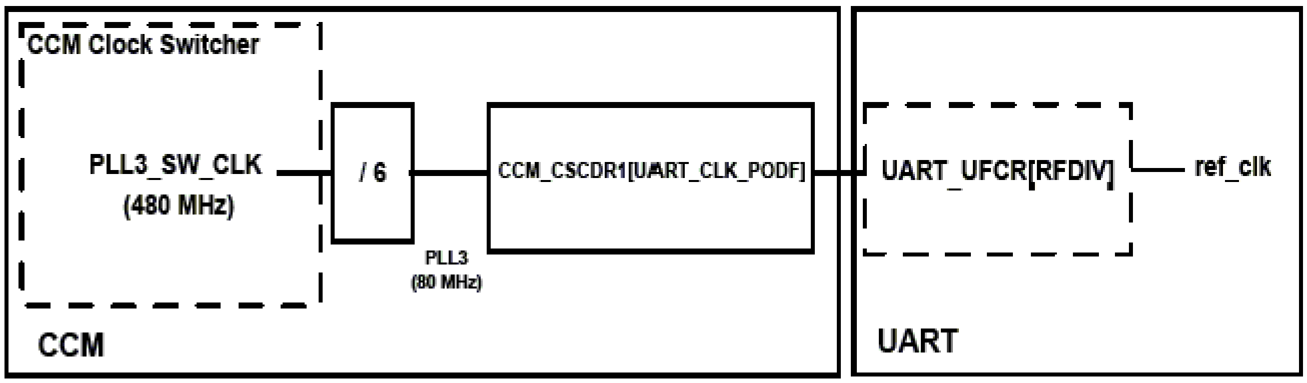


Figure 21-1. UART reference clock

The UART reference clock is used to generate the baud rate clock. This clock is derived through various dividers from the PLL3, which typically provides a 480 MHz clock. Please refer to the "Clocks" section of the UART block in the chip reference manual.

The output of PLL3 is divided with a fix divider of 6. The post divider, UART_CLK_PODF, is located in the CCM_CSCDR1 Register. The pre-divider, RFDIV, is located in the UART_UFCR Register.

The output is the ref_clk used to generate the baud rate clock according to the formula available in the section Binary Rate Multiplier (BRM) of the UART block.

21.5 IOMUX pin mapping

Although the driver calls the function that configures the IOMUX for the UART port, this is external to the driver because it depends on the board connections.

Configure the IOMUX for the UART into the iomux_config() function located in ./src/mx61/iomux/board_name/uart_iomux_config.c.

21.6 Resets and interrupts

The driver resets the module during the initialization by setting UART_UCR2[SRST] in the function uart_init().

The external application is responsible for creating the interrupt subroutine. The address of this routine is passed through the structure hw_module defined in .src/include/io.h. It is initialized by the application and used by the driver for various configurations.

All interrupt sources are listed in the "Interrupts and DMA Events" chapter in the chip reference manual. In the SDK, the list is provided in ./src/include/mx61/soc_memory_map.h.

21.7 Initializing the UART driver

Before using the UART port in a system, prepare a structure that provides the essential system parameters to the driver. This is done through the hw_module structure defined into ./src/include/io.h.

Example:

```
struct hw_module g_debug_uart = {
    "UART4 for debug",
    UART4_BASE_ADDR,
    27000000,
    IMX_INT_UART4,
    &default_interrupt_routine,
};
```

The address of this structure is used by most functions listed below.

```
/*!
 * Initialize the UART port
 *
 * @param   port - pointer to the UART module structure.
 * @param   baudrate - serial baud rate such 9600, 57600, 115200, etc.
 * @param   parity - enable parity checking: PARITY_NONE, PARITY_EVEN,
 *                   PARITY_ODD.
 * @param   stopbits - number of stop bits: STOPBITS_ONE, STOPBITS_TWO.
 * @param   datasize - number of bits in a data: SEVENBITS, EIGHTBITS,
 *                     NINEBITS (like RS-485 but not supported).
 * @param   flowcontrol - enable (RTS/CTS) hardware flow control:
 *                        FLOWCTRL_ON, FLOWCTRL_OFF.
 */
void uart_init(struct hw_module *port, uint32_t baudrate, uint8_t parity,
               uint8_t stopbits, uint8_t datasize, uint8_t flowcontrol)

/*!
 * Configure the RX or TX FIFO level and trigger mode
 *
 * @param   port - pointer to the UART module structure
 * @param   fifo - FIFO to configure: RX_FIFO or TX_FIFO.
 * @param   trigger_level - set the trigger level of the FIFO to generate
 *                          an IRQ or a DMA request: number of characters.
 * @param   service_mode - FIFO served with DMA or IRQ or polling (default).
 */
void uart_set_FIFO_mode(struct hw_module *port, uint8_t fifo, uint8_t trigger_level, uint8_t
service_mode)

/*!
 * Setup UART interrupt. It enables or disables the related HW module
 * interrupt, and attached the related sub-routine into the vector table.
 *
 * @param   port - pointer to the UART module structure.
 */
void uart_setup_interrupt(struct hw_module *port, uint8_t state)

/*!
 * Receive a character on the UART port
 *
 * @return  a character received from the UART port; if the RX FIFO
 *          is empty or errors are detected, it returns NONE_CHAR
 */
uint8_t uart_getchar(struct hw_module * port)

/*!
 * Output a character to UART port
 *
 * @param   ch - pointer to the character for output
 * @return  the character that has been sent
 */
uint8_t uart_putchar(struct hw_module * port, uint8_t * ch)

/*!
 * Enables UART loopback test mode.
 *
 * @param   port - pointer to the UART module structure
 * @param   state - enable/disable the loopback mode
 */
void uart_set_loopback_mode(struct hw_module *port, uint8_t state)
```

21.8 Testing the UART driver

The UART driver runs the following tests:

- Echo test
- SDMA test

21.8.1 Echo test

The tested UART is configured in loopback mode. Because the connection is made internally, it does not require any specific hardware.

When a character is sent through the terminal console by the user, the UART console receives it and forwards it to the tested UART TX FIFO. Once the data ready interrupt is generated, the interrupt routine reads the character from the tested UART RX FIFO and displays it through the UART console.

This test shows how to initialize the UART, how to configure the FIFO behavior, and how to set the interrupt routine.

21.8.2 SDMA test

The tested UART is configured in loopback mode. Data is sent to the TX FIFO through a DMA channel, and read from the RX FIFO through a different DMA channel.

This test shows how to initialize the UART, how to configure the FIFO behavior, and how to configure the SDMA to take care of the data transfers.

This test is available in the SDMA unit test: ./src/sdk/sdma/test/sdma_test.c.

21.8.3 Running the UART test

To run the UART tests, the SDK uses the following command to build the test:

```
./tools/build_sdk -target mx61 -board ard -board_rev 1 -test uart
```

This command generates the following ELF and binary files:

- `./output/mx61/bin/mx61ard-uart-sdk.elf`
- `./output/mx61/bin/mx61ard-uart-sdk.bin`

21.9 Source code structure

Table 21-2. Source code file locations

Description	Location
Low-level driver source	`./src/sdk/uart/drv/imx_uart/imx_uart.c`
Low-level driver header	`./src/include/imx_uart.h`

Unit tests	`./src/sdk/uart/test/uart_test.c`
	`./src/sdk/uart/test/uart_test.h`

Chapter 22
Configuring the USB Host Controller Driver

22.1 Overview

This chapter explains how to configure and use the USB controller driver.

The USB controller module contains four independent controllers: one dual role and 3 host-only controllers. In addition, there are two on-chip UTMI transceivers-one for the OTG controller and one for the HOST1 controller. Each transceiver has an associated PLL for generating the USB clocks.

The HOST2 and HOST3 controllers have an HSIC (high-speed interchip) interface for connecting to compatible on-board peripherals.

The modules related to USB are located in the memory map at the following base addresses:

- USBOTG base address = 0218 4000h
- USBH1 base address = 0218 4200h
- USBH2 base address = 0218 4400h
- USBH3 base address = 0218 4600h
- USBPLL1 base address = 020C 8010h
- USBPLL2 base address = 020C 8020h
- USBPHY1 base address = 020C 9000h
- USBPHY2 base address = 020C A000h

22.2 Features summary

This low-level driver is intended for demonstrating the configuration and basic functionality of the USB controller for host mode operation. It supports:

- Initialization of controllers and basic data structures
- Initialization of the PHY and clocks

- Host-side device enumeration
- Control transfers
- Low-level bulk transfers
- Low-level interrupt transfers

22.3 Modes of operation

The OTG controller can operate as either the host or device. Software chooses the operating mode when the controller is initialized. The host controllers-USBH1, USBH2, and USBH3-do not have device capability.

This driver does not support the OTG Host Negotiation Protocol (HNP) or Session Request Protocol (SRP).

22.4 Clocks

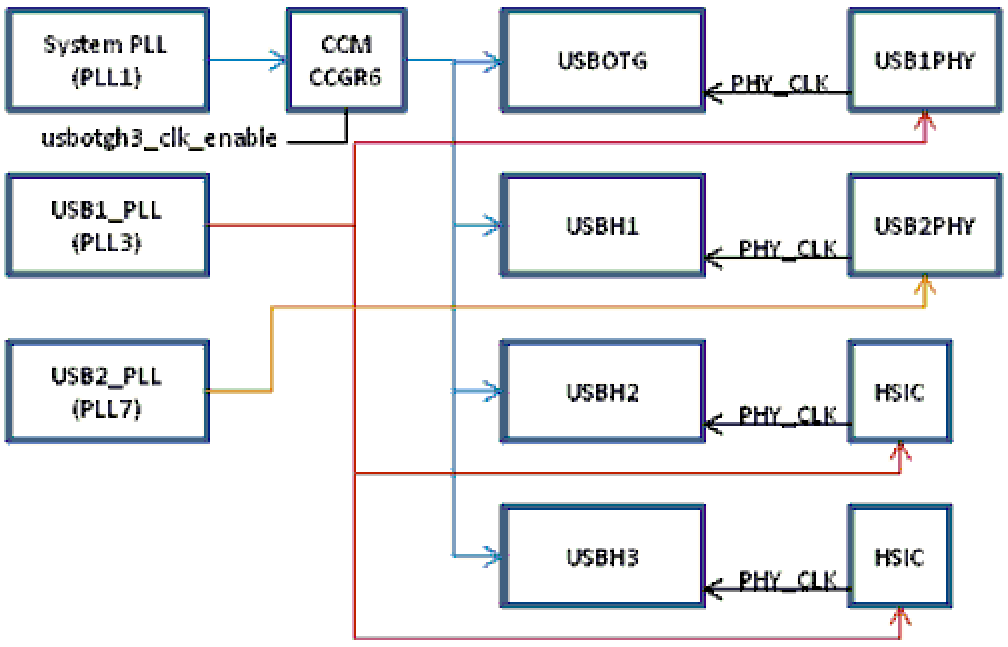


Figure 22-1. USB module clocks

The USB module uses three independent clocks: a shared clock for the control logic and DMA transfers as well as two independent dedicated transceiver clocks (PHY_CLK). The clocks are derived as follows:

- The shared clock is derived from the system PLL.

- The USBOTG, USBH2, and USBH3 controllers derive PHY_CLK from USB1_PLL.
- USBH1 and its associated PHY derive PHY_CLK from USB2_PLL.

USB1_PLL and USB2_PLL generate the PHY's 480 MHz clock., which is used for serial transmission on the USB bus. A divided version of this 480 MHz clock is used by the USB controller for the UTMI interface and protocol control logic.

22.5 IOMUX pin mapping

The pin descriptions in this section apply to the MX61 SABRE AI board. Vbus power control is implemented on I^2C port expanders, and OverCurrent inputs are implemented as GPIO. The following table shows the IOMUX settings:

Table 22-1. USB IOMUX pin mapping

Signals	Option 1		
	PAD	MUX	SION
USB_OTG_OC_B	SD4_DAT0 GPIO2[8]	ALT5	0
USB_OTG_ID	ENET_RX_ER	ALT0	0
USB_HOST1_OC_B	EIM_WAIT GPIO5[0]	ALT5	0

NOTE

USB data signals Dm/Dp and Vbus have dedicated pin functions and do not pass through the IOMUX.

Vbus PWR enable and Overcurrent I/O pass through the IOMUX, but these functions do not need to be connected to the USB controller. They can be implemented using GPIO, as is the case on the SABRE AI design.

22.6 Resets and Interrupts

All controllers in the USB module are reset to their default state by power-on reset. The driver resets each controller individually during the initialization procedure.

22.7 Initializing the driver

The driver's API contains init calls for host mode operation of the controller. These init routines initialize the controller as well as the tables and data structures (queue heads and transfers descriptor) that the controller needs. The data structure initialization provides the controller with valid pointers but does not schedule any activity. At the end of the initialization, the controller is started.

The driver's init routine performs the following steps to start the controller:

1. Enable USB clock in CCM module.
2. Configure and start USB PLL.
3. Configure and enable the PHY.
4. Set PHY type in controller's PORTSC register (UTMI for on-chip HS PHY).
5. Reset the USB controller.
6. Set the controller mode to host operation.
7. Enable Vbus power.
8. Start the controller.

At this point, the controller is running and will generate SOF tokens on the bus, but the periodic and asynchronous schedule are not yet enabled. Therefore, no data transfers are attempted.

To initialize the asynchronous schedule, the init routine creates a queue head with a dummy transfer descriptor for the control endpoint (endpoint 0). This provides an empty queue to which the application can add transfer descriptors. Additional queues can be linked to the initial queue head by the application as required.

For the periodic schedule, the init routine creates a frame list with dummy transfer descriptors to which the application can link transfer descriptors for interrupt and isochronous transfers.

The application is responsible for allocating memory for tables, data structures, and buffers. Data structures and buffers must be aligned as defined in the EHCI specification.

Please refer to PHY and clocks API and USB host API for more details.

22.8 Testing the driver

The driver comes with a test application that runs the following testing procedure:

1. The application starts the clocks and initializes the USB controller and its associated PHY.
2. The application waits for a device connection.
3. When a device is detected, the application enumerates the device.

22.9 PHY and clocks API

The functions for initializing transceivers and clocks are device specific. The API is common for all devices, but the implementation differs.

```
/*!
 * Enable Controller and transceiver clocks
 *
 * @param port
 */
int usb_clk_enable(struct usb_module *port)
/*!
 * Enable Transceiver
 * Turns transceiver power on and puts the transceiver in operational state
 *
 * @param port
 */
int enable_xcvr(struct usb_module *port)
```

22.10 USB host API

The following routines are used to initialize a controller for host operation and schedule transfers on the USB bus.

```
/*!
 * Initialize the USB host for operation.
 * This initialization sets up the USB host to detect a device connection.
 *
 * @param port      USB module to initialize
 */
int usbh_init(struct usb_module *port)
/*!
 * Initialize the periodic schedule.
 * This function creates an empty frame list for the periodic schedule,
 * points the periodic base address to the empty frame list,
 * and enables the periodic list.
 *
 * @param port              USB module to initialize
 * @param frame_list_size   size of the frame list for the periodic schedule
 * @param frame_list        pointer to the start of the allocated frame list
 */
uint32_t usbh_periodic_schedule_init(struct usb_module *port,
            uint32_t frame_list_size, uint32_t *frame_list)
/*!
 * Initialize the asynchronous schedule.
 * This function creates a queue head for the control endpoint with
 * a dummy transfer descriptor to keep the schedule idle
 * and enables the asynchronous schedule.
 *
 * @param port              USB module to initialize
 * @param queue_head        Pointer to the queue head data structure
 */
uint32_t usbh_asynchronous_schedule_init(struct usb_module *port,
            uint32_t *queue head)
/*!
 * Disable the asynchronous and periodic lists.
 *
 * @param port  USB module
 */
int usbh_disable_schedules(struct usb_module *port)
/*!
 * Initialize a queue head to service a device endpoint.
 * This function assumes the QH is the only one in the horizontal list so
 * the horizontal link pointer points to the queue head. This function
 * doesn't initialize the qTD pointer either. This must be done later.
 *
 * Parameters:
 * @param max_packet maximum packet length for the endpoint
 * @param head       mark the QH as the first in the linked list (not for interrupt QHs)
 * @param eps        end point speed
 * @param epnum      end point number
 * @param dev_addr   device address
 * @param smask      interrupt schedule mask (only used for periodic schedule QHs)
 * @param usb_qh     pointer to queue head structure
 */
int usbh_qh_init(uint32_t max_packet, uint32_t head, uint32_t ep_speed,
uint32_t ep_num,uint32_t dev_addr, uint32_t smask, USB_QH *usb_qh)
/*!
 * Issue a USB reset to the specified port.
 *
 * port      USB module to send reset
 */
void usbh_send_reset(struct usb_module *port)
/*!
 * Initialize a qTD.
 * This function initializes a transfer descriptor.
 *
           * @param trans_sz   transfer size: number of bytes to be transferred
 * @param ioc        interrupt on complete flag
 * @param pid        PID code for the transfer
 * @param buffer_ptr pointer to the data buffer
 * @param usb_qtd    Pointer to the qtd data structure
 */
int usbh_qtd_init(uint32_t trans_sz, uint32_t ioc, uint32_t pid,
                  uint32_t *buffer_ptr, USB_QTD *usb_qtd)
```

22.11 Source code and structure

Table 22-2. Source code file locations

Description	Location
Source files	
Host mode low-level driver	./src/sdk/usb/host/usbh_drvr.c
Common routines	./src/sdk/usb/common/usb_common.c
Host mode data structures	./src/sdk/usb/include/usbh_defines.h
Platform specific initialization	./src/sdk/usb/common/usb_mx61.c
Platform specific initialization header	./src/sdk/usb/include/usb_mx61.h
Test programs	
Host mode test	./src/sdk/usb/test/usbh_test.c

Chapter 23
Configuring the uSDHC Driver

23.1 Overview

This chapter provides a guide for firmware developers about how to write the device driver for the uSDHC controller. It uses i.MX 6Dual/6Quad ARD board schematics for pin assignments. For other board types, refer to their respective schematics.

The Ultra Secured Digital Host Controller (uSDHC) provides the interface between the host system and the SD(LC/HC/XC)/SDIO/MMC cards. The uSDHC acts as a bridge, passing host bus transactions to the SD(LC/HC/XC)/SDIO/MMC cards by sending commands and performing data accesses to and from the cards. It handles the SD(LC/HC/XC)/SDIO/MMC protocols at the transmission level.

There are four instances of uSDHC in the i.MX 6Dual/6Quad processor. They are located in i.MX 6Dual/6Quad memory map at the following addresses:

- uSDHC1 base address = 0219 0000h
- uSDHC2 base address = 0219 4000h
- uSDHC3 base address = 0219 8000h
- uSDHC4 base address = 0219 C000h

23.2 Clocks

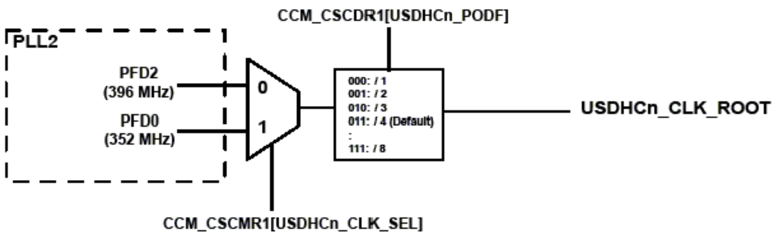


Figure 23-1. uSDHC clock tree

If the uSDHC clock is gated, ungate it in the clock control module (CCM) as follows:

- For uSDHC1, set CCM_CCGR6[CG1]
- For uSDHC2, set CCM_CCGR6[CG2]
- For uSDHC3, set CCM_CCGR6[CG3]
- For uSDHC4, set CCM_CCGR6[CG4]

Refer to the CCM chapter of the i.MX 6Dual/6Quad reference manual for more information about programming clocks.

23.3 IOMUX pin mapping

Refer to the board schematics for correct pin assignments. For the i.MX 6Dual/6Quad ARD board, the configuration for uSDHC3 is shown in the following table:

Table 23-1. uSDHC3 configuration

Port	Pad	Mode
CLK	SD3_CLK	ALT0
CMD	SD3_CMD	ALT0
DAT0	SD3_DAT0	ALT0
DAT1	SD3_DAT1	ALT0
DAT2	SD3_DAT2	ALT0
DAT3	SD3_DAT3	ALT0
DAT4	SD3_DAT4	ALT0
DAT5	SD3_DAT5	ALT0
DAT6	SD3_DAT6	ALT0
DAT7	SD3_DAT7	ALT0
RST	SD3_RST	ALT0
VSELECT	GPIO_18	ALT2
	NANDF_CS1	ALT2

NOTE

In addition to configuring the MUX control, configure the pad control of each pin. Because the pins of data and command should have pull-up resistors, they can be configured to open-drain if the board schematic already contains external pull-up resistors for them. Otherwise, they have to be configured to push-pull with a specified pull-up resistor value.

For more information about the IOMUX controller, refer to the IOMUXC chapter of the i.MX 6Dual/6Quad reference manual.

23.4 Initializing the uSDHC controller

To initialize the uSDHC controller, set up pin configuration for two uSDHC signals: clock initialization and card initialization to transfer state.

23.4.1 Initializing the SD/MMC card

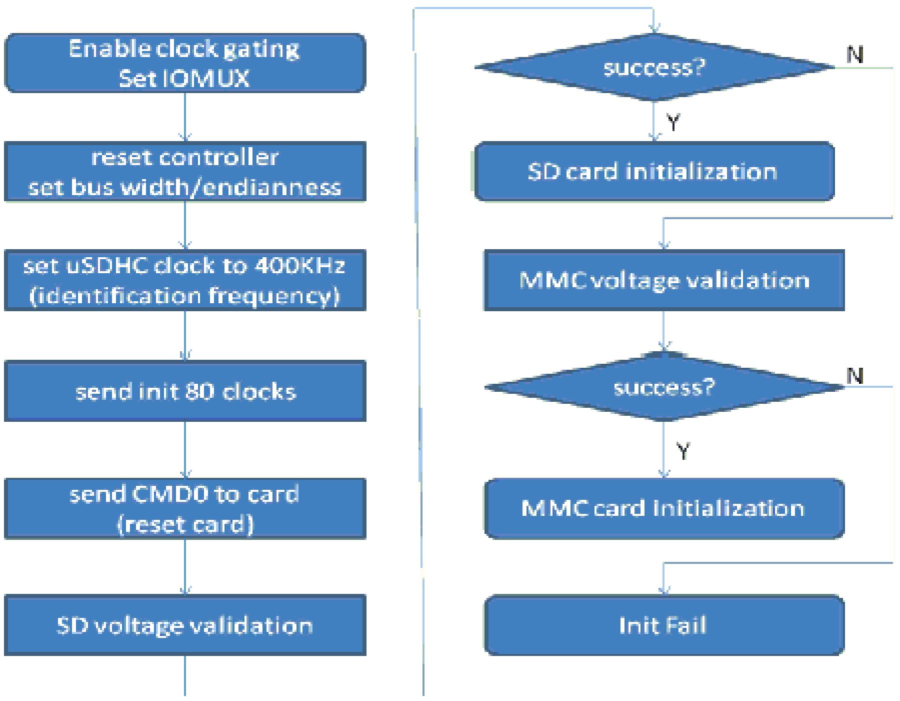


Figure 23-2. Initialization process flow chart

To initialize the SD/MMC card, perform the following procedures:

1. Controller clock setup
2. IOMUX setup
3. Controller setup and sending command to SD/MMC card for CID, RCA, bus width
4. Set the card to transfer state

23.4.2 Frequency divider configuration

The following figure shows the flow chart for the frequency divider configuration process.

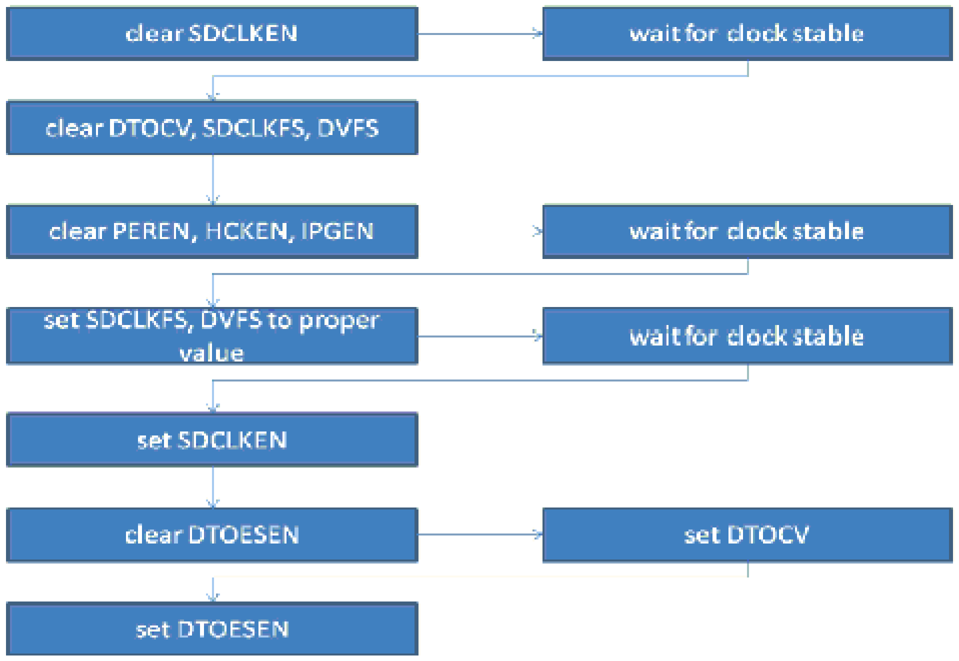


Figure 23-3. Frequency divider configuration process

For the card initialization process, configure the uSDHC clock as follows:

- Identification frequency ≤ 400 KHz
- Operating frequency ≤ 25 MHz
- High frequency ≤ 50 MHz.

Because the clock source is 200 MHz, the divider must be set to obtain the expected frequency. Use the following equation to configure the divider in the system control register (USDHC_SYS_CTRL):

Fusdhc = Fsource ÷ (DVS x SDCLKFS)

The DVS and SDCLKFS fields are set according to the value of USDHC_SYS_CTRL[DVS] and USDHC_SYS_CTRL[SDCLKFS]. See the description of the system control register for the relationship.

23.4.3 Send command to card flow chart

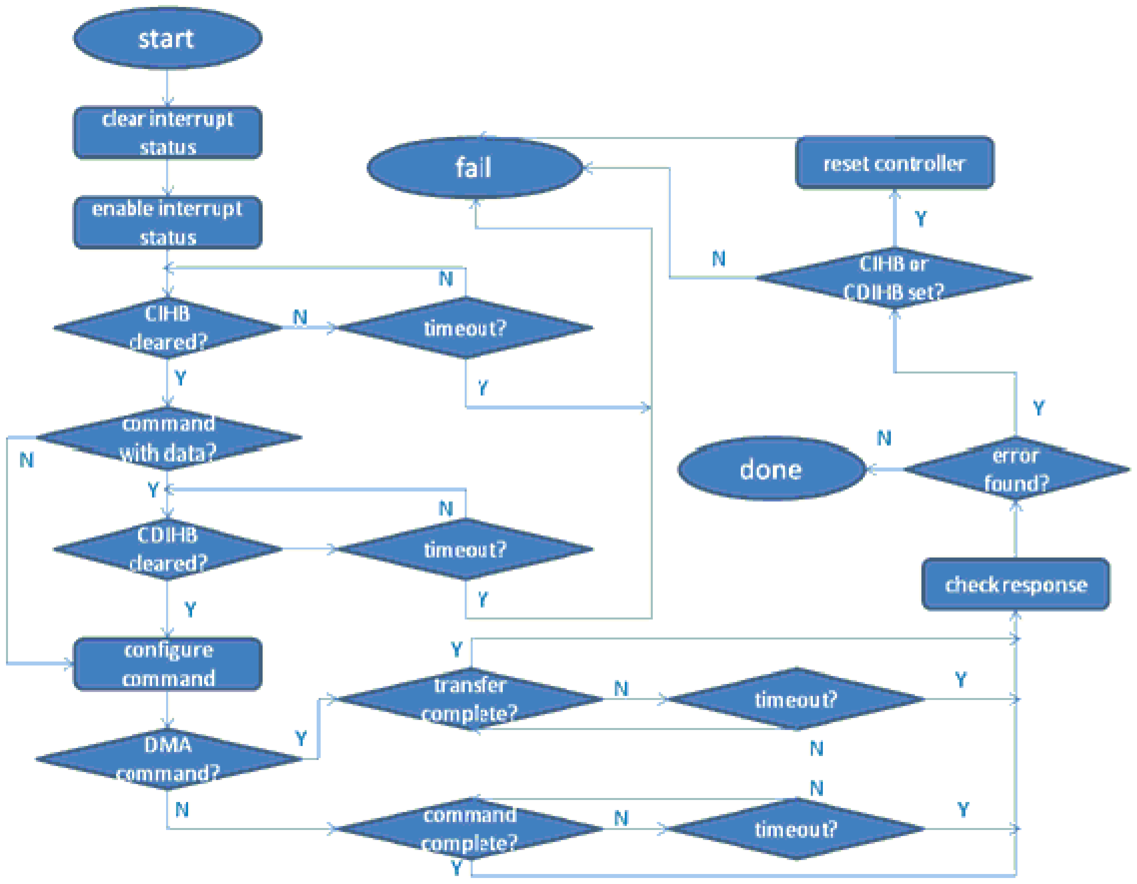


Figure 23-4. Send command to card flow chart

23.4.4 SD voltage validation flow chart

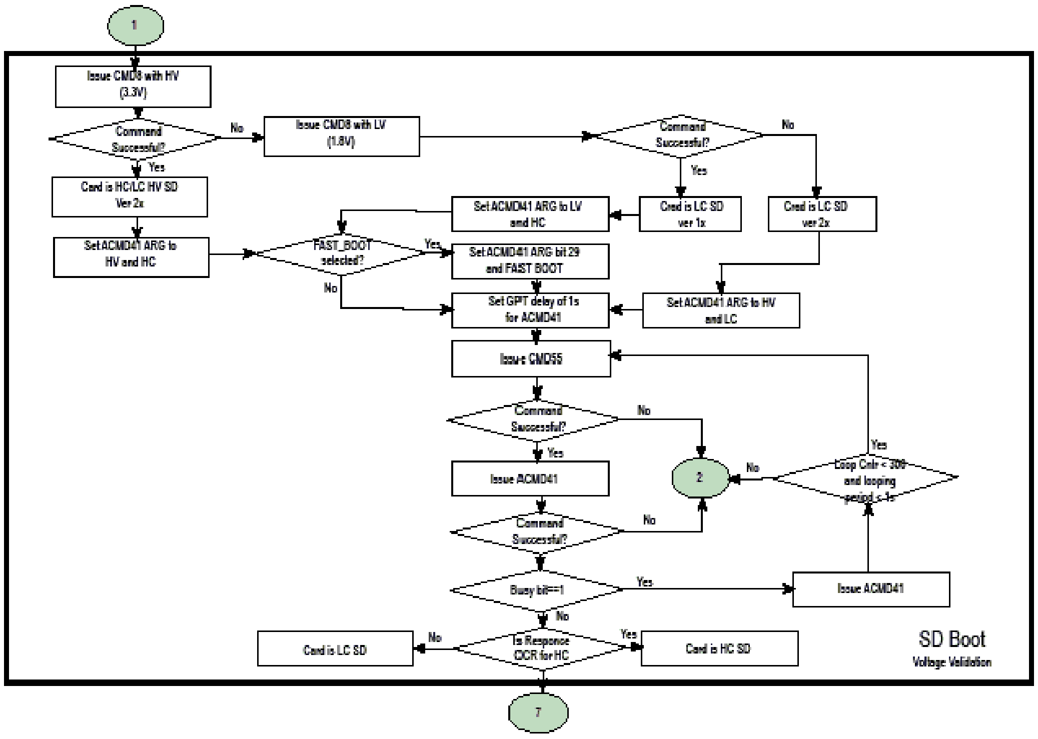


Figure 23-5. SD boot voltage validation flow chart

23.4.5 SD card initialization flow chart

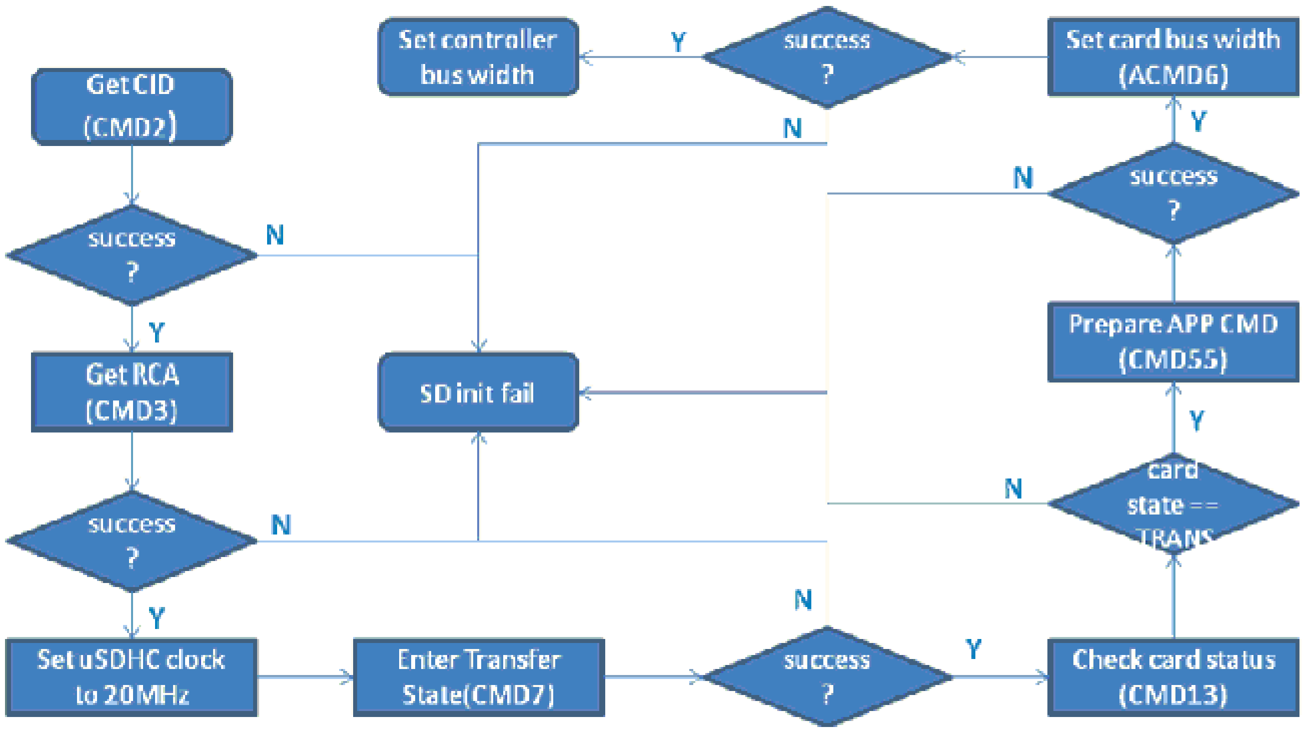


Figure 23-6. SD card initialization flow chart

23.4.6 MMC voltage validation flow chart

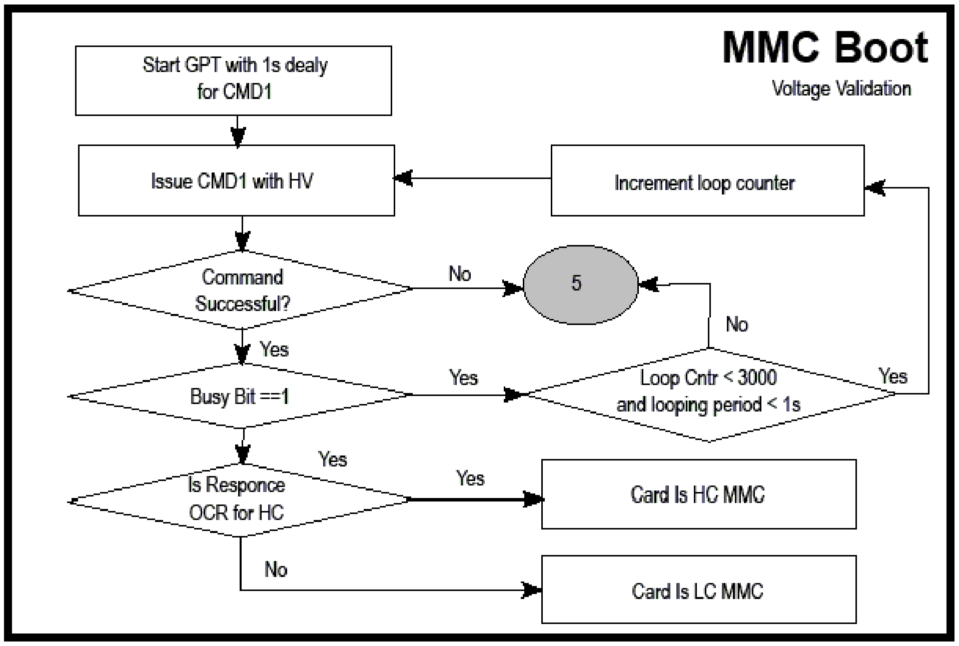


Figure 23-7. MMC voltage validation flow chart

23.4.7 MMC card initialization flow chart

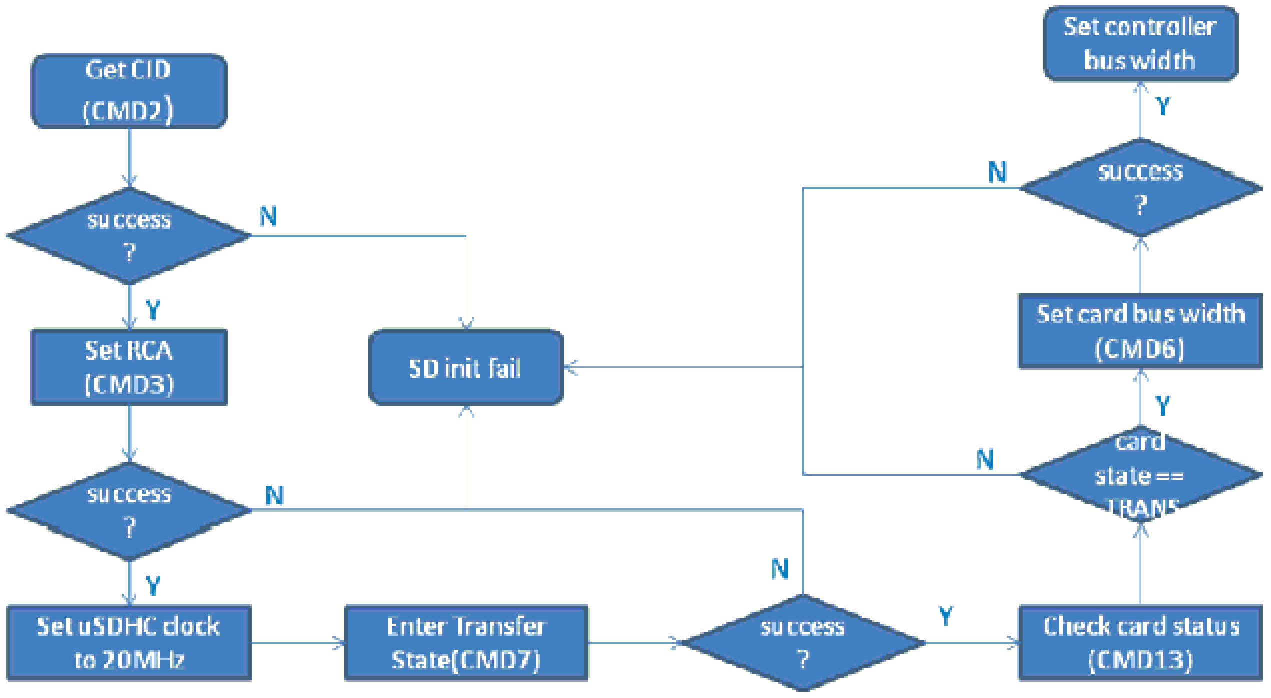


Figure 23-8. MMC card initialization flow chart

23.5 Transferring data with the uSDHC

This section describes how to read data from and write data to the SD/MMC card. Pseudocode is provided when needed.

23.5.1 Reading data from the card

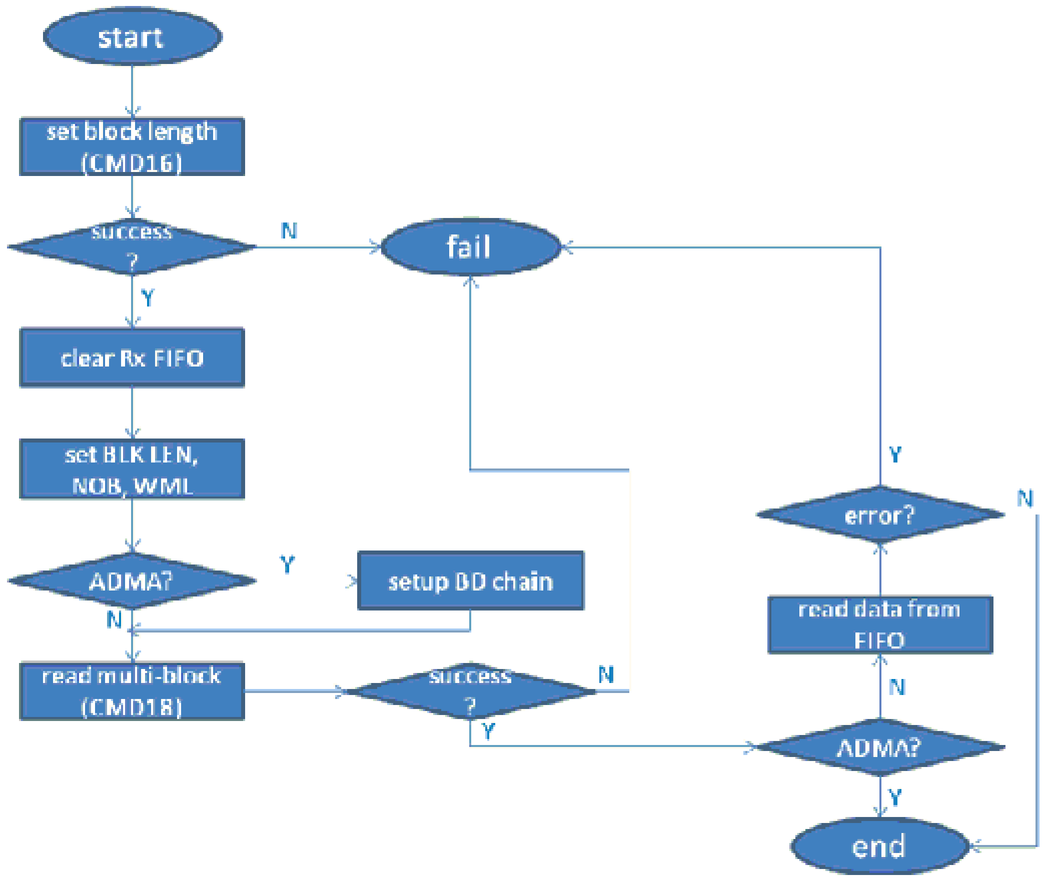


Figure 23-9. Reading data flow chart

Before reading data, use CMD16 to specify the block length to card. If the command is successful, it should also align the block length of the controller.

To read data from card, send CMD17 for one block read or CMD18 for multiblock read. The driver code uses CMD18 for reading.

The driver code supports the data transfer of polling IO and ADMA2. When using ADMA2 mode, set the buffer descriptor chain before sending the data reading command.

The buffer descriptor format is as follows:

```
typedef struct {
    unsigned char attribute; //BD attributes
    unsigned char reserved;
    unsigned short int length; //length in bytes
    unsigned int address;   //destination address
} adma_bd_t;
```

The attributes are as follows:

```
#define ESDHC_ADMA_BD_ACT             ((unsigned char)0x20)
#define ESDHC_ADMA_BD_END             ((unsigned char)0x02)
#define ESDHC_ADMA_BD_VALID         ((unsigned char)0x01)
```

For further details about the usage of ADMA2 over uSDHC, refer to the i.MX 6Dual/6Quad reference manual.

23.5.2 Writing data to the card

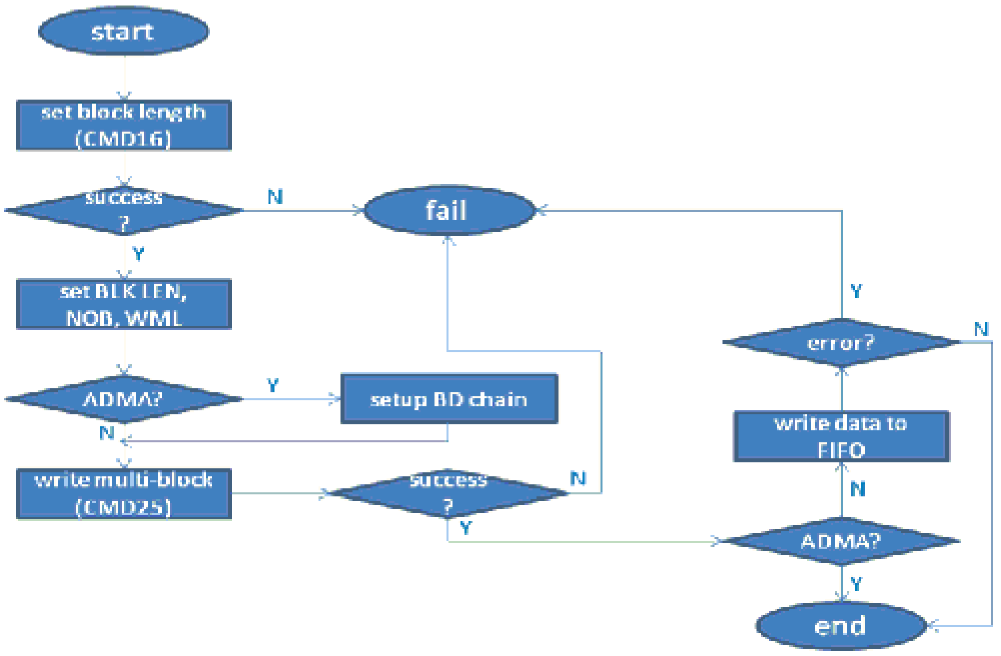


Figure 23-10. Writing data flow chart

To write data to SD/MMC card, CMD24 and CMD25 are sent. CMD24 is used to write one block while CMD25 is used to write multiblocks. In the driver code, CMD25 is used for writing.

The driver code supports polling IO and ADMA2 for writing data to the card.

23.6 Application programming interfaces

All external function calls and variables are inside inc/usdhc_ifc.h:

23.6.1 card_init API

```
int card_init(int base_address, int bus_width);
```

Description: Initialize the uSDHC controller that specified by the base_address, validate the card if inserted, initialize the card and put the card into R/W ready state.

Parameter: base_address: base address of uSDHC registers

bus_width: bus width that card will be accessed

Return: 0 on success; 1 on fail.

23.6.2 card_data_read API

```
int card_data_read(int base_address, int *dest_addr, int length, int offset);
```

Description: Read data from card to memory.

Parameter: base_address: base address of uSDHC registers

dest_addr: non-cacheable and non-bufferable area that will store the data read from card

length: number of data in bytes to be read

offset: offset in bytes that will the data be started to read from card

Return: 0 on success; 1 on fail.

23.6.3 card_data_write API

```
int card_data_write(int base_address, int *dest_addr, int length, int offset);
```

Description: Write data from memory to card.

Parameter: base_address: base address of uSDHC registers

dest_addr: non-cacheable and non-bufferable area that stores the data to write

length: number of data in bytes to write

offset: offset in bytes that will the data be started to write to card

Return: 0 on success; 1 on fail.

23.7 Source code structure

Table 23-2. uSDHC source code file locations

Description	Location
Driver	
Low level driver for card operation	`src/sdk/usdhc/drv/usdhc.c`
Header file for card operation	`src/sdk/usdhc/drv/usdhc.h`
Low level driver for uSDHC controller	`src/sdk/usdhc/drv/usdhc_host.c`
Header file for uSDHC controller	`src/sdk/usdhc/drv/usdhc_host.h`
Low level driver for MMC specific	`src/sdk/usdhc/drv/usdhc_mmc.c`
Header file for MMC specific	`src/sdk/usdhc/drv/usdhc_mmc.h`
Low level driver for SD specific	`src/sdk/usdhc/drv/usdhc_sd.c`
Header file for SD specific	`src/sdk/usdhc/drv/usdhc_sd.h`
Header file for uSDHC driver external interface	`src/sdk/usdhc/inc/usdhc_ifc.h`
Unit Test	
Test file for uSDHC	`src/sdk/usdhc/test/usdhc_test.c`
Header file for uSDHC test	`src/sdk/usdhc/test/usdhc_test.h`

Appendix A
i.MX6 Series Firmware Guide Revision History

A.1 Multicore startup revision history

The following table contains changes made to this chapter.

Topic cross-reference	Description
Overview	Updated explanation of core behavior on reset.

A.2 GIC revision history

The following table contains changes made to this chapter.

Topic cross-reference	Description
ARM interrupts and exceptions	Updated second paragraph.
GIC interrupt distributor	Updated first step.
Source code structure	Added section.
GIC Functions main file	Removed this section.
Enabling interrupt sources	Updated first line of code.
Initializing and using the GIC driver	Added section.

A.3 EPIT revision history

The following table contains changes made to this chapter.

Topic cross-reference	Description
Table 6-2	Updated high-frequency clock and peripheral clock names to "PERCLK_ROOT" and "IPG_CLK_ROOT" respectively.
Running the tests	Updated binary file name in bulleted list to "./output/mx61/bin/ mx61ard-epit-sdk.bin"
Overview	Removed note from section.

A.4 GPT revision history

The following table contains changes made to this chapter.

Topic cross-reference	Description
Table 9-2	Updated high-frequency and peripheral clock names to "PERCLK_ROOT" and "IPG_CLK_ROOT" respectively.
Table 9-3	Updated SION column values for CMPOUT1-3
Running the tests	Updated binary file name in bulleted list.

A.5 I2C as master controller revision history

The following table contains changes made to this chapter.

Topic cross-reference	Description
IOMUX pin configuration	Updated IOMUX pin configuration table values and added "or ensure that pull-up are connected externally to each lines" to second sentence of note.
Code used for I2C write operations	Updated entire section.
Code used for I2C read operations	Updated entire section.
Function to initialize the I2C controller	Updated entire section.
Repeat start	Updated note at end of section.
Configuring the programming frequency divider register (IFDR)	Clarified transfer speed procedure by adding sentences "The SCL frequency is calculated automatically by the driver by passing the desired baud rate to the initialization function. Nevertheless, the following steps can be used."
	Updated chapter title.

A.6 IPU revision history

The following table contains changes made to this chapter.

Topic cross-reference	Description
	Updated entire chapter.

A.7 UART revision history

The following table contains changes made to this chapter.

Topic cross-reference	Description
Running the UART test	Updated binary file name in bulleted list.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: IMX6FG
Rev. B, 12/2011