i.MX 6 Series Firmware Guide

Document Number: IMX6FG
Rev. 0, 11/2012

Contents

Section number | **Title** | **Page**

Chapter 1
About This Guide

1.1 About this content....25

1.2 Devices supported....25

1.3 Essential reference....25

1.4 Suggested reading....26

1.4.1 General information....26

1.4.2 Related documentation....26

1.5 Notational conventions....27

1.6 Acronyms and abbreviations....28

Chapter 2
Register Macro Usage

2.1 Register macro usage overview....33

2.2 Register macro usage advantages....33

2.3 Overview of SCT registers....34

2.3.1 Using an SCT register....34

2.3.2 Using a clear-set (CS) operation....35

2.4 Naming conventions for include files and macros....35

2.4.1 Include file naming conventions....35

2.4.2 Register macro name conventions....36

2.4.3 Bitfield macro name conventions....37

2.4.4 Register struct naming conventions....37

2.4.5 Register struct usage....38

2.5 Examples....39

2.5.1 Setting 1-Bit Wide Field....39

2.5.2 Clearing 1-Bit Wide Field....39

2.5.3 Toggling 1-Bit Wide Field....39

2.5.4 Modifying n-Bit Wide Field....39

Section number	Title	Page

2.5.5 Modifying Multiple Fields....40

2.5.6 Writing Entire Register....40

2.5.7 Reading a Bit Field....40

2.5.8 Reading Entire Register....41

2.5.9 Assembly example....41

2.6 Summary examples....41

2.6.1 Preferred syntax....41

2.6.2 Alternate syntax....41

Chapter 3 Multicore Startup

3.1 Overview....43

3.2 Boot ROM process....43

3.3 Activating the secondary cores....44

3.4 Multicore hello world example....45

3.4.1 System Reset Controller enable CPU function....46

3.4.2 Hello multicore world....46

Chapter 4 Configuring the GIC Driver

4.1 Overview....49

4.2 Feature summary....51

4.3 ARM interrupts and exceptions....51

4.3.1 GIC interrupt distributor....52

4.3.2 GIC core interfaces....53

4.4 Sample code....53

4.4.1 Handling interrupts using C....53

4.4.2 Enabling the GIC distributor....54

4.4.3 Enabling interrupt sources....55

4.4.4 Configuring interrupt priority....55

4.4.5 Targeting interrupts to specific cores....56

Section number	Title	Page

4.4.6 Using software generated interrupts (SGIs) 57

4.4.7 Enabling the GIC processor interface....57

4.4.8 Setting the CPU priority level....58

4.4.9 Reading the GIC IRQ Acknowledge 58

4.4.10 Writing the end of IRQ....58

4.4.11 GIC "hello world" example 59

4.4.12 GIC test code....59

4.5 Initializing and using the GIC driver....60

Chapter 5
Configuring the AUDMUX Driver

5.1 Overview....63

5.2 Feature summary....64

5.3 Clocks....65

5.4 IOMUX pin mapping....65

5.5 Modes of operation....65

5.5.1 Port timing mode 66

5.5.2 Port receive mode....66

5.6 Port configuration....67

5.6.1 Signal direction....67

5.6.2 AUDMUX default setting....68

5.6.2.1 Example: Port2 to Port5....68

5.7 Port configuration for SSI sync mode....69

5.8 Pseudocode for audmux_route....69

5.9 Pseudocode for audmux_port_set....70

Chapter 6
Configuring the eCSPI Driver

6.1 Overview....71

6.2 Feature summary....71

6.3 I/O signals....72

Section number	Title	Page

6.4 eCSPI controller initialization......72

6.5 eCSPI IOMUX pin mapping......73

6.6 Clocks......74

6.7 Controller initialization......74

6.8 eCSPI data transfers......75

6.9 Application program interface......76

Chapter 7
Configuring the EIM Driver

7.1 EIM overview......79

7.2 Feature summary......79

7.3 Modes of operation......80

7.4 Clocks......80

7.5 IOMUX pin mapping......81

7.6 Resets and interrupts......82

7.7 Initializing the driver......82

7.8 Testing the driver......83

Chapter 8
Configuring the EPIT Driver

8.1 Overview......85

8.2 Feature summary......85

8.3 Modes of operation......85

8.4 Output compare event......86

8.5 Clocks......86

8.6 IOMUX pin mapping......88

8.7 Resets and interrupts......88

8.8 Initializing the EPIT driver......88

8.9 Testing the EPIT driver......90

8.9.1 Delay test......90

8.9.2 Tick test......90

Section number	Title	Page

Chapter 9
Configuring the ESAI Driver

9.1 ESAI overview....................91

9.2 Feature summary....................91

9.3 Clocks....................92

9.4 IOMUX pin mapping....................92

9.5 External ESAI signal description....................93

9.6 Audio framework....................94

9.6.1 audio_card_t data structure....................94

9.6.2 audio_ctrl_t data structure....................94

9.6.3 audio_codec_t data structure....................94

9.6.4 audio_dev_ops_t data structure....................95

9.6.5 audio_dev_para_t data structure....................95

9.7 ESAI driver functions....................95

9.7.1 Resetting the ESAI....................96

9.7.2 Obtaining ESAI parameters....................96

9.7.3 Setting ESAI parameters....................96

9.7.4 Obtaining ESAI status....................96

9.7.5 Enabling ESAI submodules....................97

9.7.6 Initializing the ESAI....................97

9.7.7 Configuring the ESAI....................97

9.7.8 Playback through ESAI....................98

9.7.9 ESAI de-initialization....................98

9.8 CS42888 driver....................98

9.9 Testing the unit....................98

Chapter 10
Configuring the Ethernet Driver

10.1 Overview....................101

10.2 Feature summary....................101

Section number	Title	Page

10.3 Modes of operation....103

10.4 Clocks....104

10.5 IOMUX pin mapping....104

10.6 Resets and interrupts....105

10.7 Initializing the driver....106

10.8 Testing the driver....106

Chapter 11
Configuring the FlexCAN Modules

11.1 Overview....107

11.2 Feature summary....108

11.3 Modes of operation....109

11.4 Clocks....109

11.5 Module timing....110

11.6 IOMUX pin mapping....110

11.7 Resets and interrupts....111

11.7.1 Module reset....111

11.7.2 Module interrupts....111

11.8 Initializing the FlexCAN module....112

11.9 Testing the driver....113

Chapter 12
Configuring the GPU3D Driver

12.1 Overview....117

12.2 Feature summary....117

12.3 Modes of operation....118

12.4 Clocks....118

12.5 IOMUX pin mapping....118

12.6 Resets and interrupts....118

12.7 Initializing the GPU3D driver....119

12.8 Testing the GPU3D driver....119

Section number	Title	Page

Chapter 13
Configuring the GPMI Controller

13.1 Overview....121

13.2 Feature summary....122

13.3 Modes of operation....123

13.4 Basic NAND timing....124

13.5 Clocks....125

13.6 IOMUX pin mapping....125

13.7 APBH DMA....127

13.8 BCH ECC....128

13.9 NAND FLASH WRITE example code....129

13.10 NAND FLASH READ example code....133

13.11 NAND FLASH ERASE example code....136

Chapter 14
Configuring the GPT Driver

14.1 Overview....139

14.2 Feature summary....139

14.3 Modes of operation....140

14.4 Events....140

14.4.1 Output compare event....140

14.4.2 Input capture event....140

14.4.3 Rollover event....141

14.5 Clocks....141

14.6 IOMUX pin mapping....143

14.7 Resets and interrupts....143

14.8 Initializing the GPT driver....144

14.9 Testing the GPT driver....145

14.9.1 Output compare test....146

14.9.2 Input compare test....146

Section number	Title	Page

Chapter 15
Configuring the HDMI Tx Module

15.1 Overview....147

15.2 Feature summary....148

15.3 Modes of operation....149

15.4 Events....149

15.5 Clocks....149

15.5.1 Video input interface clock....150

15.5.2 System and slave register interface clocks....150

15.5.3 EDID I2C E-DDC interface clock....151

15.5.4 CEC interface clock....151

15.5.5 HDMI Tx PHY interface....152

15.6 IOMUX pin mapping....152

15.7 Resets and interrupts....153

15.8 Initializing the driver....153

15.8.1 Setting up the video input....153

15.8.2 Setting up the video sampler....154

15.8.3 Setting up the CSC (color space converter)....155

15.8.4 Setting up the video packetizer....155

15.8.5 Setting up the frame composer....156

15.8.6 Setting up HDMI Tx PHY....157

15.9 Testing the driver....157

Chapter 16
Configuring the I2C Controller as a Master Device

16.1 Overview....159

16.2 Initializing the I2C controller....159

16.2.1 IOMUX pin configuration....160

16.2.2 Clocks....160

16.2.3 Configuring the programming frequency divider register (IFDR)....161

Section number	Title	Page

16.3 I2C protocol....163

16.3.1 START signal....163

16.3.2 Slave address transmission....163

16.3.3 Data transfer....164

16.3.4 STOP signal....165

16.3.5 Repeat start....165

16.4 Programming controller registers for I2C data transfers....166

16.4.1 Function to initialize the I2C controller....166

16.4.2 Programming the I2C controller for I2C Read....166

16.4.3 Code used for I2C read operations....168

16.4.4 Programming the I2C controller for I2C Write....169

16.4.5 Code used for I2C write operations....171

Chapter 17 Configuring the I2C Controller as a Slave Device

17.1 Overview....173

17.2 Feature summary....173

17.3 Modes of operation....173

17.4 Clocks....174

17.5 Resets and interrupts....175

17.6 Initializing the driver....175

17.7 Testing the driver....177

17.7.1 Running the test....177

Chapter 18 Configuring the IPU Driver

18.1 Overview....179

18.2 IPU task management....181

18.3 Image rendering....183

18.3.1 IDMAC....183

18.3.2 DMFC....183

Section number	Title	Page

18.3.3 Display Processor (DP)....184
18.3.4 Display controller (DC)....185
18.3.5 Display interface (DI)....186
18.4 Image processing....186
18.4.1 Downsizing....187
18.4.2 Main processing....188
18.4.3 Rotation....188
18.5 CSI preview....189
18.5.1 CSI interfaces....189
18.5.1.1 Parallel interface....189
18.5.1.2 High-speed serial interface-MIPI (mobile industry processor interface)....189
18.5.2 CSI modes....190
18.5.2.1 Gated mode....190
18.5.2.2 Non-gated mode....190
18.5.2.3 BT656 mode....191
18.5.2.4 BT1120 mode....191
18.6 CSI capture....192
18.7 Mixed task....193
18.8 Clocks....193
18.8.1 High-speed processing clock (HSP_CLK)....193
18.8.2 Display interface clocks (DI_CLKn)....194
18.9 IOMUX pin mapping....195
18.10 Use cases....196
18.10.1 Single image rendering example....196
18.10.1.1 Configuring the IPU DMA channel (single image rendering)....197
18.10.1.2 Allocating the DMFC block....197
18.10.1.3 Configuring the DP block....197
18.10.1.4 Configuring the DC block....198
18.10.1.5 Configuring the DI block....198

18.10.1.6 Enabling the blocks involved in the display flow....201

18.10.2 Image combining example....201

18.10.2.1 Configuring the IPU DMA channel....203

18.10.2.2 Allocating the DMFC....203

18.10.2.3 Configuring the DP module....203

18.10.2.4 Other modules....204

18.10.3 Image rotate example....204

18.10.3.1 Configuring IDMAC channels for IC tasks (IC rotate)....205

18.10.3.2 Configuring the IC task....205

18.10.3.3 Setting IDMAC buffer ready....206

18.10.3.4 Image rendering process (IDMAC)....206

18.10.4 Image resizing example....206

18.10.4.1 IPU process flow....206

18.10.4.2 Configuring IDMAC channels for IC resize tasks....207

18.10.4.3 Configuring the IC resize tasks....208

18.10.4.4 Setting IDMAC buffer ready (image rotation)....209

18.10.4.5 Image rendering process....209

18.10.5 Color space conversion example....210

18.10.5.1 IPU process flow (color space conversion)....210

18.10.5.2 Configuring IDMAC channels for IC tasks....210

18.10.5.3 Configuring IC tasks....211

18.10.5.4 IPU configurations for the DP task....212

Chapter 19
Configuring the Keypad Controller

19.1 Overview....215

19.2 Feature summary....215

19.3 Modes of operation....215

19.4 Clocks....216

19.5 IOMUX pin mapping....216

Section number	Title	Page

19.6 Resets and interrupts....217

19.7 Initializing the driver....217

19.7.1 Closing the keypad port....218

19.7.2 Waiting for or obtaining a key press event....218

19.7.3 Waiting for all keys to release....218

19.8 Testing the driver....219

Chapter 20
Configuring the LDB Driver

20.1 Overview....221

20.2 Feature summary....223

20.3 Input and output ports....223

20.4 Modes of operation....223

20.4.1 Single display mode....224

20.4.2 Dual display mode....224

20.4.3 Separate display mode....224

20.4.4 Split mode....225

20.5 LDB Processing....225

20.5.1 SPWG mapping....225

20.5.2 JEIDA mapping....225

20.6 Clocks....226

20.6.1 Data serialization clocking....227

20.7 Configuring the LDB_CTRL register....228

20.8 Use cases....228

Chapter 21
Configuring the Camera Preview Driver

21.1 Overview....231

21.2 Feature summary....233

21.2.1 Synchronization performance details....233

21.2.2 Simultaneous functionality support....234

Section number	Title	Page

21.2.3 Data rate support....234

21.3 Modes of operation....234

21.3.1 Interface modes....234

21.3.2 Work modes....235

21.4 Clocks....235

21.5 IOMUX pin mapping....236

21.5.1 IOMUX pin mapping for CSI0/CIS1 parallel interface....237

21.5.2 IOMUX pin mapping for the MIPI CSI-2 interface 237

21.6 Resets and interrupts....238

21.6.1 Resets....238

21.6.2 Interrupts....238

21.7 Initializing the driver....239

21.7.1 Configuring the IDMAC channel for CSI....239

21.7.2 Allocating SMFC....239

21.7.3 Configuring CSI....240

21.7.4 Configuring the sensor....240

21.7.5 Image rendering....241

21.8 Testing the driver....241

Chapter 22
Configuring the MIPI CSI-2 Driver

22.1 Overview....243

22.2 Feature summary....245

22.3 Modes of operation....245

22.4 Clocks....245

22.4.1 Output clock....246

22.4.2 Input clock 246

22.5 IOMUX pin mapping....248

22.6 Resets and interrupts....248

22.7 Initializing the driver....249

Section number	Title	Page

22.8 Testing the driver....250

Chapter 23
Configuring the MIPI DSI driver

23.1 Overview....251

23.2 Feature summary....252

23.3 Modes of operation....253

23.4 Clocks....254

23.5 IOMUX pin mapping....255

23.6 Resets and Interrupts....255

23.7 Initializing the driver....255

23.7.1 Initializing the DSI controller....255

23.7.1.1 Global configuration....255

23.7.1.2 Configure the DPI interface....256

23.7.1.3 Select the video transmission mode....256

23.7.1.4 Define the DPI horizontal timing configuration....258

23.7.1.5 Define the vertical line configuration....258

23.7.2 Initializing the D-PHY....259

23.8 Testing the driver....260

Chapter 24
Configuring the Power Modes

24.1 Overview....261

24.2 Feature summary....261

24.3 Modes of operation....261

24.4 Clocks....261

24.5 IOMUX pin mapping....262

24.6 Resets and interrupts....262

24.7 Using the driver....262

24.8 Testing the driver....263

24.9 Running the test....263

Section number	Title	Page

Chapter 25
Configuring the OCOTP Driver

25.1 Overview....265
25.2 Feature summary....265
25.3 Modes of operation....265
25.4 Clocks....266
25.5 IOMUX pin mapping....266
25.6 Resets and interrupts....266
25.7 Initializing the driver....266
25.8 Testing the driver....267
25.9 Running the test....267

Chapter 26
Configuring the PCI Express Driver

26.1 Overview....269
26.2 Feature summary....269
26.3 Modes of operation....270
26.4 Clocks....271
26.5 IOMUX pin mapping....271
26.6 Resets and interrupts....271
26.7 Initializing the driver....271
26.8 Testing the driver....271

Chapter 27
Configuring the PWM driver

27.1 Overview....273
27.2 Feature summary....273
27.3 Clocks 274
27.4 IOMUX pin mapping....274
27.5 Resets and interrupts....275

Section number	Title	Page

27.6 Initializing the driver......276

27.6.1 Configuring the PWM output......276

27.6.1.1 Generating the pulse width......276

27.6.1.2 Generating the duty cycle......277

27.6.2 Enabling PWM output......277

27.7 Application program interface......277

Chapter 28
Using the SATA SDK

28.1 Overview......279

28.2 Feature summary......279

28.3 Modes of operation......280

28.4 Clocks......280

28.5 IOMUX pin mapping......281

28.6 Resets and Interrupts......281

28.7 Initializing the driver......281

28.8 Testing the driver......281

Chapter 29
Configuring the SDMA Driver

29.1 Overview......283

29.2 IOMUX pin mapping......283

29.3 Scripts......285

29.4 Channels and channel descriptor......285

29.5 Buffer descriptor and BD chain......286

29.6 Application programming interface......287

29.7 Using the API......290

Chapter 30
Configuring the SPDIF Driver

30.1 Overview......291

30.2 Feature summary......291

30.3 Clocks......292

Section number	Title	Page

30.4 IOMUX pin mapping....292

30.5 Audio framework....293

30.5.1 audio_card_t data structure....293

30.5.2 audio_ctrl_t data structure....294

30.5.3 audio_codec_t data structure....294

30.5.4 audio_dev_ops_t data structure....294

30.5.5 audio_dev_para_t data structure....294

30.6 Using SPDIF driver functions....295

30.6.1 Soft resetting SPDIF....295

30.6.2 Dumping readable SPDIF registers....295

30.6.3 Obtaining SPDIF setting and status....296

30.6.4 Initializing SPDIF....296

30.6.5 Configuring SPDIF....297

30.6.6 Playback through SPDIF....297

30.6.7 De-initializing SPDIF....297

30.7 Testing the SPDIF driver....298

Chapter 31
Using the SNVS RTC/SRTC Driver

31.1 Overview....299

31.2 Feature summary....300

31.3 Modes of operation....300

31.4 Clocks....301

31.5 Counters....301

31.5.1 Non-Secured Real Time Counter....301

31.5.1.1 Non-Secured Real Time Counter Alarm....301

31.5.1.2 Non-Secured Real Periodic Interrupt....302

31.5.2 Secure Real Time Counter....302

Section number	Title	Page

31.6 Driver API....303

31.6.1 SNVS lower level driver APIs....303

31.6.1.1 Enable/Disable SNVS non-secured real time counter....303

31.6.1.2 Enable/Disable SNVS non-secured time alarm....304

31.6.1.3 Enable/Disable SNVS periodic interrupt....304

31.6.1.4 Set SNVS non-secure real time counter registers....305

31.6.1.5 Set SNVS non-secure RTC time alarm registers....306

31.6.1.6 Enable/Disable SNVS secure real time counter....306

31.6.1.7 Enable/Disable SNVS secure time alarm....307

31.6.1.8 Set SNVS secured real time counter registers....307

31.6.1.9 Set SNVS non-secure time alarm register....308

31.6.2 RTC upper layer driver APIs....308

31.6.2.1 Initialize RTC....308

31.6.2.2 De-initialize RTC....309

31.6.2.3 Setup RTC one time alarm....309

31.6.2.4 Setup RTC periodic time alarm 310

31.6.2.5 Disable RTC periodic alarm....310

31.6.3 SRTC upper layer driver APIs....311

31.6.4 Initialize SRTC....311

31.6.5 De-initialize SRTC....311

31.6.6 Setup SRTC one time alarm....312

31.6.7 Testing the SNVS SRTC/RTC driver....312

Chapter 32
Configuring the SSI Driver

32.1 SSI overview....315

32.2 Feature summary....316

32.3 Clocks....317

32.4 IOMUX pin mapping 317

Section number	Title	Page

32.5 Audio framework....318

32.5.1 audio_card_t data structure....319

32.5.2 audio_ctrl_t data structure....319

32.5.3 audio_codec_t data structure....319

32.5.4 audio_dev_ops_t data structure....319

32.5.5 audio_dev_para_t data structure....320

32.6 SSI driver functions....320

32.6.1 Resetting the SSI....321

32.6.2 Obtaining SSI setting and status values....321

32.6.3 Setting SSI parameters....321

32.6.4 Enabling SSI sub-modules....322

32.6.5 Initializing the SSI driver....322

32.6.6 Configuring the SSI....322

32.6.6.1 Playback through SSI....323

32.7 sgtl5000 driver....323

32.8 Testing the unit....323

32.9 Functions....324

32.9.1 Local functions....324

32.9.2 APIs....325

Chapter 33
Configuring the UART Driver

33.1 Overview....327

33.2 Feature summary....327

33.3 Modes of operation....327

33.4 Clocks....328

33.5 IOMUX pin mapping....329

33.6 Resets and interrupts....329

33.7 Initializing the UART driver....329

Section number	Title	Page

33.8 Testing the UART driver....331

33.8.1 Echo test....331

33.8.2 SDMA test....331

33.8.3 Running the UART test....331

Chapter 34
Configuring the USB Host Controller Driver

34.1 Overview....333

34.2 Feature summary 333

34.3 Modes of operation....334

34.4 Clocks....334

34.5 IOMUX pin mapping....335

34.6 Resets and interrupts....335

34.7 Initializing the host driver....336

34.8 Initializing the device driver....337

34.9 Testing the host mode....338

34.10 Testing the device mode....338

34.11 PHY and clocks API....339

34.12 USB host API....339

34.13 USB device API....341

34.14 Source code and structure....342

Chapter 35
Configuring the uSDHC Driver

35.1 Overview....345

35.2 Clocks....345

35.3 IOMUX pin mapping....346

35.4 Initializing the uSDHC controller 347

35.4.1 Initializing the SD/MMC card....347

35.4.2 Frequency divider configuration....347

35.4.3 Send command to card flow chart....348

Section number	Title	Page

35.4.4 SD voltage validation flow chart....349

35.4.5 SD card initialization flow chart....350

35.4.6 MMC voltage validation flow chart....351

35.4.7 MMC card initialization flow chart....352

35.5 Transferring data with the uSDHC....353

35.5.1 Reading data from the card....353

35.5.2 Writing data to the card....354

35.6 Application programming interfaces....355

35.6.1 card_init API....355

35.6.2 card_data_read API....355

35.6.3 card_data_write API....355

Chapter 36
Configuring the VDOA Driver

36.1 Overview....357

36.2 Feature summary....357

36.3 Modes of operation....357

36.4 Clocks....358

36.5 Resets and interrupts....358

36.6 Initializing the driver....358

36.7 Testing the driver....359

Chapter 37
Configuring the VPU Driver

37.1 Overview....361

37.2 Feature summary....362

37.3 Modes of operation....363

37.3.1 Using the input stream modes....363

37.3.2 Using the output stream modes....363

37.4 Clocks....365

37.5 Resets and interrupts....366

Section number	Title	Page

37.6 Initializing the driver....................................366

37.6.1 Initializing the VPU for the first time....................................366

37.6.2 Initializing the VPU decoder....................................367

37.6.3 VPU encoder initialization....................................367

37.6.4 Using the multi-instance operation....................................369

37.7 Testing the driver....................................369

37.7.1 Testing the decoder....................................369

37.7.2 Testing the encoder....................................369

37.7.3 Running the multi-instance demo....................................370

Chapter 38
Configuring the Watchdog Driver

38.1 Overview....................................373

38.2 Feature summary....................................375

38.3 Modes of operation....................................375

38.4 Signals....................................375

38.5 Resets and interrupts....................................376

38.6 Initializing the driver....................................376

38.7 Testing the driver....................................376

Chapter 1
About This Guide

1.1 About this content

This document's purpose is to help software engineers create board bring up and test code for their own custom boards based on the i.MX 6 series of application processors. It provides example driver code that demonstrates the proper initialization, boot up and basic I/O operation of i.MX 6 peripherals and controllers. This code can be implemented into test suites or boot code to help ensure proper board functionality.

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.MX 6 series of application processors is also assumed.

This guide is released along with the i.MX 6 series Platform SDK release package, which consists of a working set of sample drivers and test code for customer reference and use. The README.pdf document in the package describes how to build and run the drivers and the test code.

1.2 Devices supported

The firmware guide currently supports the i.MX 6Quad and 6Dual processor families.

1.3 Essential reference

This guide intended as a companion to the i.MX 6 series chip reference manuals and data sheets. You should have access to an electronic copy of the latest versions of your chip-specific reference manual and data sheet. A non-disclosure agreement may be required.

At the time of publication, the following reference manual is available:

- *i.MX 6Dual/6Quad Multimedia Applications Processor Reference Manual* (IMX6DQRM).

At the time of publication, the following data sheets are available:

- i.MX 6Dual/6Quad Automotive and Infotainment Applications Processors (IMX6DQAEC)
- i.MX 6Dual/6Quad Applications Processors for Consumer Products (IMX6DQCEC)

Contact your local FAE if you need assistance obtaining an electronic copy or an updated list of available chip-specific reference manuals.

1.4 Suggested reading

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

1.4.1 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

1.4.2 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.

1.5 Notational conventions

This table shows notational conventions used in this content.

Table 1-1. Notational conventions

<table>
<tr><th>Convention</th><th>Definition</th></tr>
<tr><td colspan="2">General</td></tr>
<tr><td>Cleared</td><td>When a bit takes the value zero, it is said to be cleared.</td></tr>
<tr><td>Set</td><td>When a bit takes the value one, it is said to be set.</td></tr>
<tr><td>mnemonics</td><td>Instruction mnemonics are shown in lowercase bold.</td></tr>
<tr><td>italics</td><td>Italics can indicate the following:
• Variable command parameters, for example, bcctrx
• Titles of publications
• Internal signals, for example, $\overline{\textit{core int}}$</td></tr>
<tr><td>h</td><td>Suffix to denote hexadecimal number. Note that numbers in code may use the alternate 0x prefix convention to denote hexadecimal instead.</td></tr>
<tr><td>b</td><td>Suffix to denote binary number. Note that numbers in code may use the alternate 0b prefix convention to denote binary instead.</td></tr>
<tr><td>REGISTER[FIELD]</td><td>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.</td></tr>
<tr><td>x</td><td>In some contexts, such as signal encodings, an unitalicized x indicates a don't care.</td></tr>
<tr><td>x</td><td>An italicized x indicates an alphanumeric variable</td></tr>
<tr><td>n</td><td>An italicized n indicates either:
• An integer variable
• A general-purpose bitfield unknown</td></tr>
<tr><td>~</td><td>NOT logical operator</td></tr>
<tr><td>&</td><td>AND logical operator</td></tr>
<tr><td>|</td><td>OR logical operator</td></tr>
<tr><td>||</td><td>Concatenation, for example, TCR[WPEXT] || TCR[WP]</td></tr>
<tr><td colspan="2">Signals</td></tr>
<tr><td>$\overline{\text{OVERBAR}}$</td><td>An overbar indicates that a signal is active-low.</td></tr>
<tr><td>_b, _B</td><td>Alternate notation to indicate that a signal is active-low</td></tr>
<tr><td>lowercase_italics</td><td>Lowercase italics is used to indicate internal signals</td></tr>
<tr><td>lowercase_plaintext</td><td>Lowercase plain text is used to indicate signals that are used for configuration.</td></tr>
<tr><td colspan="2">Register access</td></tr>
<tr><td>Reserved</td><td>Ignored for the purposes of determining access type</td></tr>
<tr><td>R/W</td><td>Indicates that all non-reserved fields in a register are read/write</td></tr>
<tr><td>R</td><td>Indicates that all non-reserved fields in a register are read only</td></tr>
<tr><td>W</td><td>Indicates that all non-reserved fields in a register are write only</td></tr>
<tr><td>w1c</td><td>Indicates that all non-reserved fields in a register are cleared by writing ones to them</td></tr>
</table>

1.6 Acronyms and abbreviations

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

Table 1-2. Definitions and acronyms

Term	Definition
Address Translation	Address conversion from virtual domain to physical domain
API	Application programming interface
ARM®	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	Path between several devices through data lines.
Bus load	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. • Big endian means that the least significant byte of the data is stored in a higher address than the most significant byte (the reverse of little endian)
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

Table continues on the next page...

Table 1-2. Definitions and acronyms (continued)

Term	Definition
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.
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.

Table continues on the next page...

Table 1-2. Definitions and acronyms (continued)

Term	Definition
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 and NOR Flash architecture are the two flash technologies that are used in memory cards such as the Compact Flash cards. NAND Flash 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 than 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, reference, or 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.
RGBA	RGBA color space stands for the colors red, green, blue, and 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 pseudorandom 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

Table continues on the next page...

Table 1-2. Definitions and acronyms (continued)

Term	Definition
SPBA	Shared peripheral bus arbiter A three-to-one IP-Bus arbiter, with a resource-locking mechanism.
SPI	Serial peripheral onterface 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 12 Mb/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 depending on how the cables are connected to the devices. They also can connect to a host PC.
Word	A group of bits comprising 32 bits

Chapter 2
Register Macro Usage

2.1 Register macro usage overview

This chapter provides background on the register set and provides examples of how to use the hardware register macros generated from the chip database. The include files provide a consistent set of C defines and macros that should be used to access the hardware registers.

2.2 Register macro usage advantages

Using the register macros generated from the chip reference manual database provides the following advantages.

- The macros isolate code from specifics of the hardware such as register addresses, which makes it possible to write drivers and other code that is portable between chips.
- Using register macros produces self-documenting code. Named constants and macros clearly indicate the tasks being performed, unlike hard-coded addresses or values that are difficult to understand.
- Using register headers provided by Freescale instead of hand-coded headers ensures the most accurate values and facilitates easy updates if errors in the header files or reference manual are discovered.
- The macros ensure that the optimal sequence of operations is performed for set, clear, or toggle (SCT) or bitfield write operations, depending on whether the register has hardware SCT register instances available.
- Register macros provide multiple options for expressing the operation being performed.

2.3 Overview of SCT registers

Certain hardware registers are implemented as a set that can be used to either set, clear, or toggle (SCT) individual bits of the primary register. These registers are used to avoid time consuming read-modify-write (RMW) operations. SCT registers also provide the ability atomically change values of single-bit bitfields.

This functionality is useful because the chip has a complex architecture that uses multiple buses to segment I/O traffic and clock domains. To facilitate low power consumption, clocks are set to just meet application demands. In general, the I/O buses and associated hardware blocks run at slower speeds than the CPU's speed. Reading a hardware register may therefore incur a large number of wait cycles because the CPU must wait for the register data to travel multiple buses and bridges. The chip does provide write buffering, meaning the CPU does not wait for register write transactions to complete. From the CPU perspective, register writes occur much faster than reads.

In addition, most hardware registers are subdivided into smaller functional bit fields that are not required to align on byte or half-word boundaries. These bit fields can be any number of bits wide and are usually packed.

Without the SCT registers, the best way to update a single bit field without affecting the contents of the register's remaining fields would be to use a read-modify-write (RMW) operation. In this operation, the CPU reads the register, modifies the target field, and then writes the results back to the register. However, the initial register read makes the RMW operation expensive in terms of CPU cycles.

2.3.1 Using an SCT register

When writing to an SCT register, all set bits perform the associated operation on the primary register, while all cleared bits set are not affected. SCT registers always read back 0 and should be considered write-only.

SCT registers are not implemented if the primary register is read-only or if the register contains a single value that does not make sense to update partially (such as a memory address). In addition, only certain IP blocks support SCT registers.

NOTE

Not all macros for set, clear, or toggle (SCT) are atomic. For registers that do not provide hardware support for this functionality, these macros are implemented as a sequence of read/modify/write operations. When atomic operation is

required, the developer should pay attention to this detail, because unexpected behavior might result if an interrupt occurs in the middle of the critical section comprising the update sequence.

2.3.2 Using a clear-set (CS) operation

When SCT registers are available, it is possible to update one or more fields using only register writes. First, all bits of the target fields are cleared by a write to the associated clear register, then the desired value of the target fields is written to the set register. This sequence of two writes is referred to as a clear-set (CS) operation.

A CS operation does have one potential drawback. Whenever a field is modified, the hardware sees a value of 0 before the final value is written. For most fields, passing through the 0 state is not a problem. Nonetheless, this behavior is something to consider when using a CS operation.

Also, a CS operation is not required for fields that are one bit wide. While the CS operation works in this case, it is more efficient to simply set or clear the target bit (that is, one write instead of two). A simple set or clear operation is also atomic, while a CS operation is not.

2.4 Naming conventions for include files and macros

The generated include files and macros follow a consistent naming convention. This prevents namespace collisions and makes the macros easier to remember. Macro names are built from patterns based on the names listed in the reference manual. Thus, it is possible to construct a macro name solely by looking at the reference manual contents.

2.4.1 Include file naming conventions

- The include file for a specific hardware module is named `regs<module>.h`. The module name is in all lower case and has no spaces.
- The `regs.h` header file is included by each of the hardware module header files. It provides several shared typedefs and macros.
- The `regs.h` file also provides a number of generic macros that can be used as an alternate syntax for the various register operations. Because many operations involve using two or more of the defined macros, the module, register, and bitfield names are

often repeated in a C expression. The generic macros provide shorthand to avoid the repetition. Refer to the examples in this chapter for the alternate syntax.

The include files are safe to use with assembly code as well as C or C++ code. Not all defines make sense in the assembly context, but many are useful. Those declarations that are only applicable to C/C++ are excluded when the headers are included in assembly code.

CAUTION

The preprocessor macro `__LANGUAGE_ASM__` must be defined prior to including the register definition header files in assembly code.

2.4.2 Register macro name conventions

The following tables show the register macro name conventions for single- and multi-instance modules.

Table 2-1. Single-instance modules

Format	Purpose	Example
hw_*<module>*_*<register>*_t	register struct	hw_gpmi_ctrl0_t
HW_*<module>*_*<register>*_ADDR	register address	HW_GPMI_CTRL0_ADDR
HW_*<module>*_*<register>*	access register struct	HW_GPMI_CTRL0
HW_*<module>*_*<register>*_RD()	read register	HW_GPMI_CTRL0_RD()
HW_*<module>*_*<register>*_WR(v)	write register	HW_GPMI_CTRL0_WR(0xc0000000)
HW_*<module>*_*<register>*_SET(v)	set register bits	HW_GPMI_CTRL0_SET(0x1000)
HW_*<module>*_*<register>*_CLR(v)	clear register bits	HW_GPMI_CTRL0_CLR(0x1000)
HW_*<module>*_*<register>*_TOG(v)	toggle register bits	HW_GPMI_CTRL0_TOG(0x1000)

Macros for multi-instance modules take the instance number as an additional first argument. In the definitions below, the parameter *x* is the instance number. The instance numbers accepted as arguments match the numbers shown in the register memory map in the reference manual. In almost all cases, instance numbers start at 1.

Table 2-2. Multi-instance modules

Format	Purpose	Example
hw_*<module>*_*<register>*_t	register struct	hw_ecspi_conreg_t
HW_*<module>*_*<register>*_ADDR(x)	register address	HW_ECSPI_CONREG_ADDR(2)
HW_*<module>*_*<register>(x)*	access register struct	HW_ECSPI_CONREG(2)
HW_*<module>*_*<register>*_RD(x)	read register	HW_ECSPI_CONREG_RD(2)

Table continues on the next page...

Table 2-2. Multi-instance modules (continued)

Format	Purpose	Example
HW_<module>_<register>_WR(x, v)	write register	HW_ECSPI_CONREG_WR(2, 0xc0000000)
HW_<module>_<register>_SET(x, v)	set register bits	HW_ECSPI_CONREG_SET(2, 0x1000)
HW_<module>_<register>_CLR(x, v)	clear register bits	HW_ECSPI_CONREG_CLR(2, 0x1000)
HW_<module>_<register>_TOG(x, v)	toggle register bits	HW_ECSPI_CONREG_TOG(2, 0x1000)

2.4.3 Bitfield macro name conventions

The following tables explain the bitfield macro naming conventions for single- and multi-instance modules.

Table 2-3. Single-instance modules

Format	Purpose	Example
BP_<module>_<register>_<field>	bit position	BP_GPMI_CTRL0_CLKGATE
BM_<module>_<register>_<field>	bit mask, pre-shifted	BM_GPMI_CTRL0_CLKGATE
BF_<module>_<register>_<field>(v)	shift and mask bitfield value	BF_CCM_ANALOG_PLL_ARM_DIV_SELECT(7)
BG_<module>_<register>_<field>(r)	get bitfield value from register value	BG_CCM_ANALOG_PLL_ARM_DIV_SELECT(HW_CCM_ANALOG_PLL_ARM_RD())
BW_<module>_<register>_<field>(v)	write bitfield using SCT or RMW	BW_CCM_ANALOG_PLL_ARM_DIV_SELECT(12)
BV_<module>_<register>_<field>__<value>	bitfield named value constant	BV_CCM_ANALOG_PLL_ARM_BYPASS_CLK_SRC__OSC_24M

Only the BW_ bitfield macro differs for multi-instance modules.

Table 2-4. Multi-instance modules

Format	Purpose	Example
BW_<module>_<register>_<field>(x, v)	write bitfield using SCT or RMW	BW_ECSPI_CONREG_CHANNEL_SELECT(1, 2)

2.4.4 Register struct naming conventions

For each register present in a module, the generated include files declare a struct (actually, a union) with a name similar to hw_ecspi_conreg_t. These struct declarations have members for each of the register's bitfields, as well as a member to access the register value as a whole.

The following code shows an example register struct declaration, using the EPITSR register of the EPIT mode:

```
typedef union _hw_epit_epitsr
{
    reg32_t U;
    struct _hw_epit_epitsr_bitfields
    {
        unsigned OCIF : 1; //!< [0] Output compare interrupt flag.
        unsigned RESERVED0 : 31; //!< [31:1] Reserved.
    } B;
} hw_epit_epitsr_t;
```

This example demonstrates the key features of register structs:

- The outer union declaration always includes a `U` field that represents the entire register value.
- A `B` member struct holds the bitfield member declarations.

Note

The `B` member struct is used to ensure that all compilers can understand the declarations. Some compilers do not allow anonymous union members.

2.4.5 Register struct usage

The include files also contain a macro to access each register as a reference to the corresponding register struct.

As shown in Table 2-2 , these macros look like `HW_EPIT_EPITSR`. Because the macro evaluates to a reference, not a pointer, members are accessed with the dot ('.') operator.

The following shows example uses of register structs:

```
// Integer value of the register
HW_GPMI_CTRL0.U       // single-instance
HW_EPIT_EPITSR(1).U  // multi-instance

// Bitfield access
HW_GPMI_CTRL0.B.CLKGATE // single-instance
HW_EPIT_EPITSR(2).B.OCIF // multi-instance
```

NOTE

Modifying bitfield values through the register struct causes the compiler to generate read-modify-write code, which is inherently non-atomic. In cases where registers have SCT instances, it is possible to update single-bit bitfields atomically using set or clear operations.

2.5 Examples

The following examples show how to code common register operations using the predefined include files. Each example shows preferred and alternate syntax and also shows constructs to avoid. Summaries are provided toward the end.

The examples are valid C and will compile without errors. The reader is encouraged to compile these examples and examine the resulting assembly code.

2.5.1 Setting 1-Bit Wide Field

```
// Preferred (one atomic write to SET register)
HW_GPMI_CTRL0_SET(BM_GPMI_CTRL0_UDMA);

// Alternate (same as above, just different syntax)
BF_SET(GPMI_CTRL0, UDMA);

// Avoid
BW_GPMI_CTRL0_UDMA(1);              // writes 1 to _CLR then 1 to _SET register
BF_WR(GPMI_CTRL0, UDMA, 1);         // same as above, just different syntax
HW_GPMI_CTRL0.B.UDMA = 1;           // RMW
```

2.5.2 Clearing 1-Bit Wide Field

```
// Preferred (one atomic write to _CLR register)
HW_GPMI_CTRL0_CLR(BM_GPMI_CTRL0_DEV_IRQ_EN);

// Alternate (same as above, just different syntax)
BF_CLR(GPMI_CTRL0, DEV_IRQ_EN);

// Avoid
BW_GPMI_CTRL0_DEV_IRQ_EN(0);        // writes 1 to _CLR then 0 to _SET register
BF_WR(GPMI_CTRL0, DEV_IRQ_EN, 0);     // same as above, just different syntax
HW_GPMI_CTRL0.B.DEV_IRQ_EN = 0;       // RMW
```

2.5.3 Toggling 1-Bit Wide Field

```
// Preferred (one atomic write to _TOG register)
HW_GPMI_CTRL0_TOG(BM_GPMI_CTRL0_RUN);

// Alternate (same as above, just different syntax)
BF_TOG(GPMI_CTRL0, RUN);

// Avoid
HW_GPMI_CTRL0.B.RUN ^= 1;             // RMW
```

2.5.4 Modifying n-Bit Wide Field

```
// Preferred (does CS operation or byte/halfword write if the field is
//   8 or 16 bits wide and properly aligned)
BW_GPMI_CTRL0_COMMAND_MODE(BV_GPMI_CTRL0_COMMAND_MODE_READ_AND_COMPARE);
BW_GPMI_CTRL0_COMMAND_MODE(iMode);
BW_GPMI_CTRL0_XFER_COUNT(2);        // this does a halfword write

// Alternate (same as above, just different syntax)
BF_WR(GPMI_CTRL0, COMMAND_MODE, BV_GPMI_CTRL0_COMMAND_MODE_READ_AND_COMPARE);
BF_WR(GPMI_CTRL0, COMMAND_MODE, iMode);
BF_WR(GPMI_CTRL0, XFER_COUNT, 2);   // this does a halfword write

// Avoid (RMW)
HW_GPMI_CTRL0.B.COMMAND_MODE = BV_GPMI_CTRL0_COMMAND_MODE_READ_AND_COMPARE;
HW_GPMI_CTRL0.B.COMMAND_MODE = iMode;
```

2.5.5 Modifying Multiple Fields

```
// Preferred (explicit CS operation)
HW_GPMI_CTRL0_CLR( OR3(BM_GPMI_CTRL0, RUN, DEV_IRQ_EN, COMMAND_MODE) );
HW_GPMI_CTRL0_SET( OR3(BF_GPMI_CTRL0, RUN(iRun), DEV_IRQ_EN(1),
                                                 COMMAND_MODE_V(READ_AND_COMPARE)) );
// Alternate (same as above, just different syntax)
BF_CS3(GPMI_CTRL0, RUN, iRun, DEV_IRQ_EN, 1, COMMAND_MODE,
                                       BV_GPMI_CTRL0_COMMAND_MODE_READ_AND_COMPARE);
// Avoid (multiple RMW - the C compiler does NOT merge into one RMW)
HW_GPMI_CTRL0.B.RUN          = iRun;
HW_GPMI_CTRL0.B.DEV_IRQ_EN   = 1;
HW_GPMI_CTRL0.B.COMMAND_MODE = BV_GPMI_CTRL0_COMMAND_MODE_READ_AND_COMPARE;
```

2.5.6 Writing Entire Register

These operations update all fields at once.

```
// Preferred
HW_GPMI_CTRL0_WR(BM_GPMI_CTRL0_SFTRST); // all other fields are set to 0

// Alternate (same as above, just different syntax)
HW_GPMI_CTRL0.U = BM_GPMI_CTRL0_SFTRST;
```

2.5.7 Reading a Bit Field

```
// Preferred
iRun = HW_GPMI_CTRL0.B.RUN;

// Alternate (same as above, just different syntax)
iRun = BF_RD(GPMI_CTRL0, RUN);

// Verbose Alternate (example of using bit position (BP_) define)
iRun = (HW_GPMI_CTRL0_RD() & BM_GPMI_CTRL0_RUN) << BP_GPMI_CTRL0_RUN;
```

2.5.8 Reading Entire Register

```
// Preferred
i = HW_GPMI_CTRL0_RD();

// Alternate (same as above, just different syntax)
i = HW_GPMI_CTRL0.U;
```

2.5.9 Assembly example

```
// 6.1 Take GPMI block out of reset and remove clock gate.

// 6.2 Write a value to GPMI CTRL0 register. All other fields are set to 0.
    ldr    r0, =HW_GPMI_CTRL0_CLR_ADDR
    ldr    r1, =BM_GPMI_CTRL0_SFTRST | BM_GPMI_CTRL0_CLKGATE
    str    r1, [r0]
    ldr    r0, =HW_GPMI_CTRL0_ADDR
    ldr    r1, =BF_GPMI_CTRL0_COMMAND_MODE(BV_GPMI_CTRL0_COMMAND_MODE__READ_AND_COMPARE)
    str    r1, [r0]
```

2.6 Summary examples

2.6.1 Preferred syntax

```
// Setting, clearing, toggling 1-bit wide field
HW_GPMI_CTRL0_SET(BM_GPMI_CTRL0_UDMA);
HW_GPMI_CTRL0_CLR(BM_GPMI_CTRL0_DEV_IRQ_EN);
HW_GPMI_CTRL0_TOG(BM_GPMI_CTRL0_RUN);

// Modifying n-bit wide field
BW_GPMI_CTRL0_XFER_COUNT(2);

// Modifying multiple fields
HW_GPMI_CTRL0_CLR( OR3(BM_GPMI_CTRL0, RUN, DEV_IRQ_EN, COMMAND_MODE) );
HW_GPMI_CTRL0_SET( OR3(BF_GPMI_CTRL0, RUN(iRun), DEV_IRQ_EN(1),
                                             COMMAND_MODE_V(READ_AND_COMPARE)) );

// Reading a bit field
iRun = HW_GPMI_CTRL0.B.RUN;

// Writing or reading entire register (all fields updated at once)
HW_GPMI_CTRL0_WR(BM_GPMI_CTRL0_SFTRST);
i = HW_GPMI_CTRL0_RD();
```

2.6.2 Alternate syntax

```
// Setting, clearing, toggling 1-bit wide field
BF_SET(GPMI_CTRL0, UDMA);
```

```
BF_CLR(GPMI_CTRL0, DEV_IRQ_EN);
BF_TOG(GPMI_CTRL0, RUN);

// Modifying n-bit wide field
BF_WR(GPMI_CTRL0, XFER_COUNT, 2);

// Modifying multiple fields
BF_CS3(GPMI_CTRL0, RUN, iRun, DEV_IRQ_EN, 1, COMMAND_MODE,
                                   BV_GPMI_CTRL0_COMMAND_MODE_READ_AND_COMPARE);

// Reading a bit field
iRun = BF_RD(GPMI_CTRL0, RUN);

// Writing or reading entire register (all fields updated at once)
HW_GPMI_CTRL0.U = BM_GPMI_CTRL0_SFTRST;
i = HW_GPMI_CTRL0.U;
```

Chapter 3
Multicore Startup

3.1 Overview

This chip includes multiple 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.

3.2 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 chip 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 which chip 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 chip 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 the following table for the list of registers.

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

Table 3-1. 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 i.MX 6 series Platform 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.

3.3 Activating the secondary cores

Although multiple cores are available on this processor, only core0 automatically activates during the initial boot process. SRC, the system reset controller module, handles the reset signal for each core. By default, 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 ROM 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.

3.4 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      enable_mmu
   ; 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.

3.4.1 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.

3.4.2 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 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 4
Configuring the GIC Driver

4.1 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:

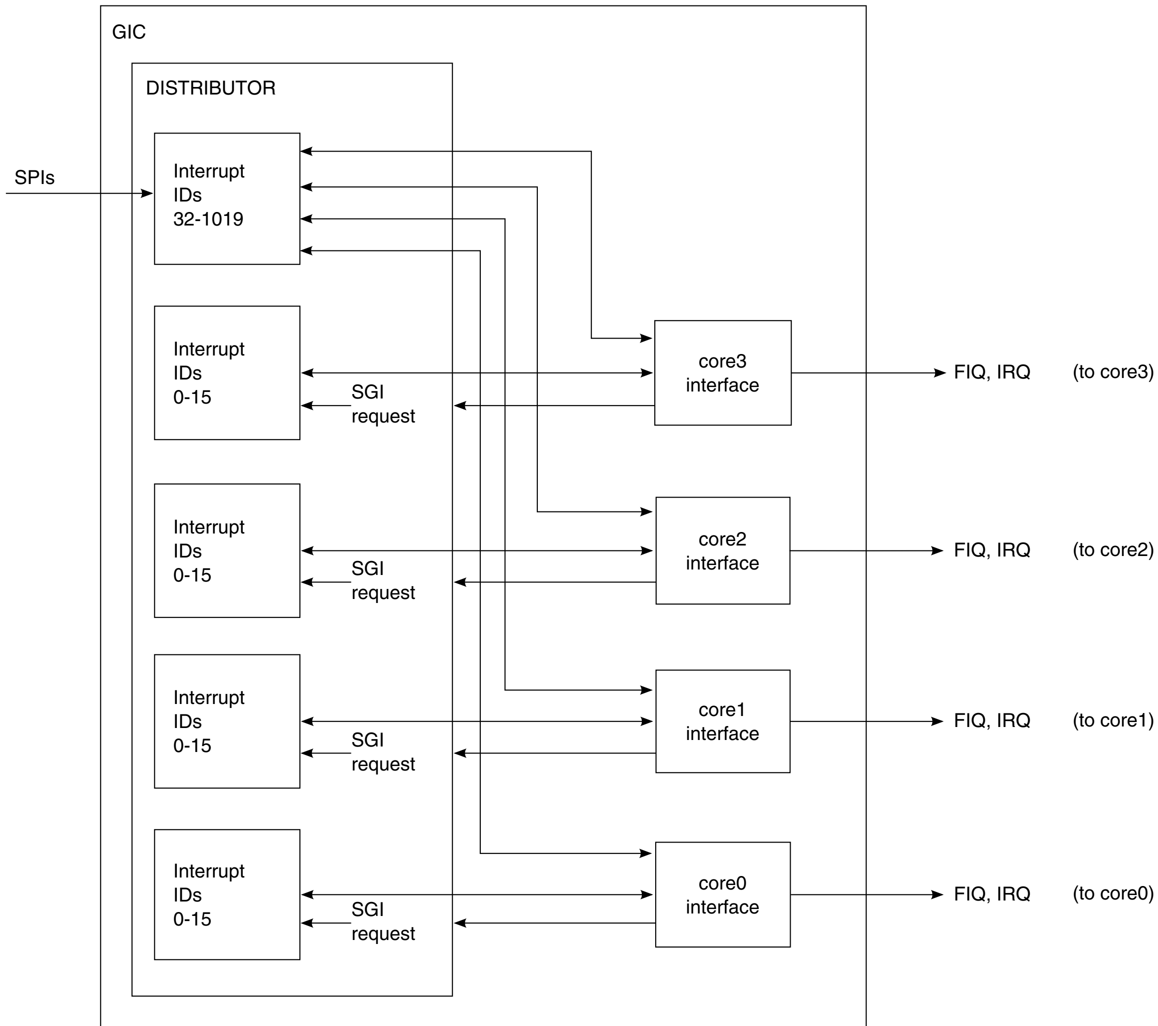


Figure 4-1. GIC simplified block diagram

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 4-1. 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

Table continues on the next page...

Table 4-1. Cortex A-9 general MPCore memory map (continued)

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

4.2 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

4.3 ARM interrupts and exceptions

All 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 4-2. 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 chip, 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.

4.3.1 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 chip's 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 4-3. 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)

4.3.2 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 4-4. 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.

4.4 Sample code

4.4.1 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
      }
}
```

4.4.2 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
```

4.4.3 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
```

4.4.4 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
```

4.4.5 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.MX 6Quad/6Dual 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
```

4.4.6 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
```

4.4.7 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
```

4.4.8 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
```

4.4.9 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
```

4.4.10 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
```

4.4.11 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 following example, these are the expected results:

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

4.4.12 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
    }
}
```

4.5 Initializing and using the GIC driver

Typically, the startup routine (for the i.MX 6 series Platform SDK, `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 follows:

```
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
```

Chapter 5
Configuring the AUDMUX Driver

5.1 Overview

AUDMUX, which is a digital audio multiplexer, provides a programmable interconnect device for voice, audio, and synchronous data routing between host serial interfaces (such as SSI, the synchronous serial interface controller) 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.
- 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 Port7) and four external ports (Port3, Port4, Port5, and Port6). Each port can be programmed to one of the following:

- A full 6-wire SSI interface for asynchronous receive and transmit
- 4-wire (synchronous) peripheral interfaces
- 6-wire (asynchronous) peripheral interfaces

The following figure shows the structure of the AUDMUX.

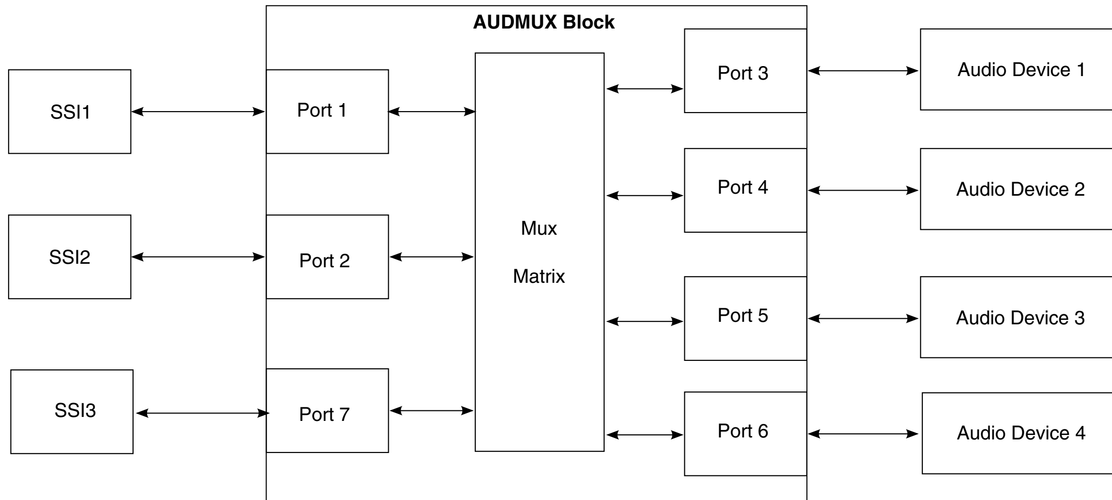


Figure 5-1. AUDMUX structure

The only instance of AUDMUX 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 chip reference manual.

5.2 Feature 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

5.3 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 chip reference manual for details.

5.4 IOMUX pin mapping

The IOMUX pin configuration shown in the following table is based on the connections in an engineering sample board in which PORT5 was connected with the SSI codec sgtl5000 in SYNC mode. Check your board's schematic for your board's specific pin assignments.

Table 5-1. 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

5.5 Modes of operation

The following table explains the AUDMUX modes of operation:

Table 5-2. 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
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

Table continues on the next page...

Table 5-2. Modes of operation (continued)

Mode	Description	Configuration
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

5.5.1 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.

5.5.2 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.

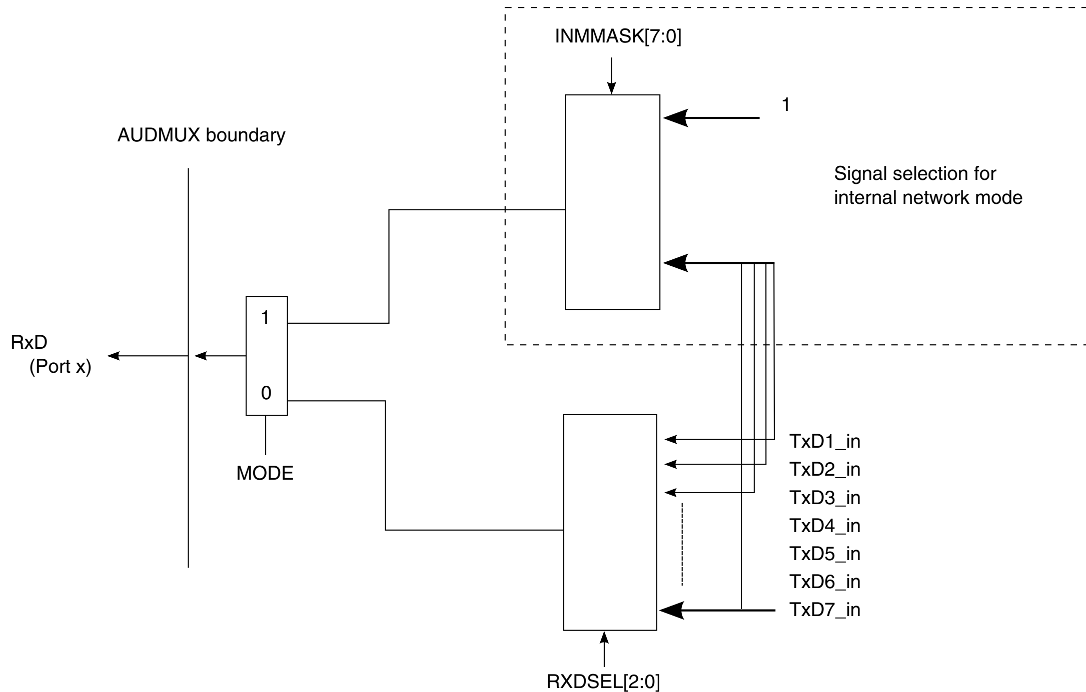


Figure 5-2. Port receive mode

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.

5.6 Port configuration

5.6.1 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.

5.6.2 AUDMUX default setting

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

- Port1 to Port6— Port6 provides the clock and frame sync.
- Port2 to Port5—Port5 provides the clock and frame sync.
- Port3 to Port4—Port4 provides the clock and frame sync.
- Port7 to Port7—in data loopback mode

5.6.2.1 Example: Port2 to Port5

Assume that the SSI audio codec is connected to the external Port5. Using the default setting, SSI controller 2 (connected to the internal Port2) and the codec are connected together. SSI controller 2 is the slave of the SSI bus and the codec is the master.

The registers related to port2 and port5 are listed in the following table along with their reset values:

Table 5-3. 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.

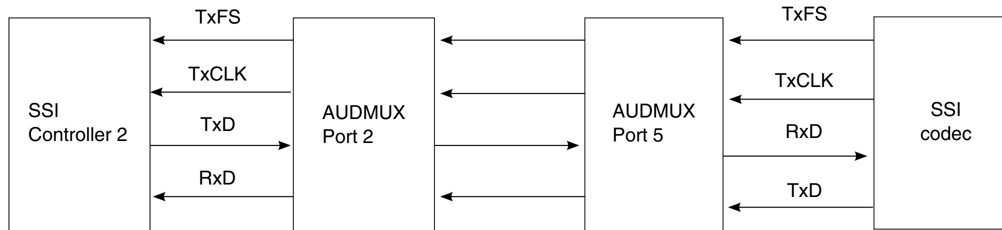


Figure 5-3. Signal muxing and direction of Port2 and Port5 under default setting

5.7 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.

5.8 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;
}
```

5.9 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;
}
```

Chapter 6
Configuring the eCSPI Driver

6.1 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 an engineering sample board's schematics for pin assignments. Refer to your board's schematics for your board's specific information.

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

6.2 Feature summary

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

6.3 I/O signals

The eCSPI block has below I/O signals:

Table 6-1. 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:

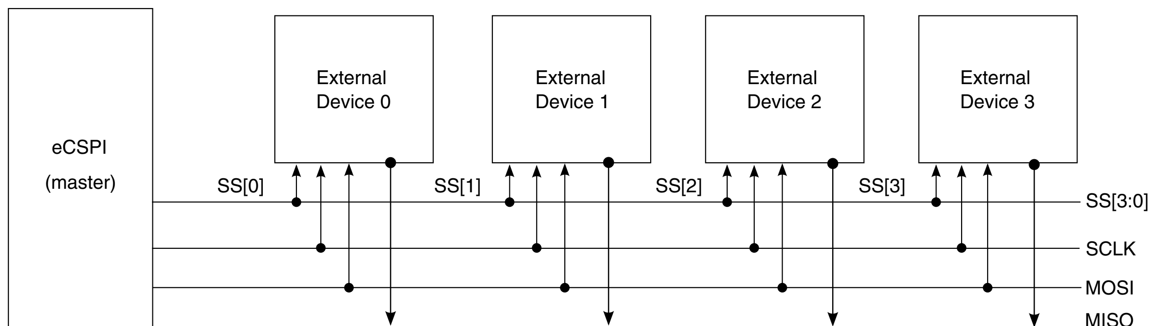


Figure 6-1. eCSPI as SPI master

6.4 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.

6.5 eCSPI IOMUX pin mapping

Refer to board schematics for correct pin assignments to configure the pin signals. The following table is based on an engineering sample board and is used as an example:

Table 6-2. eCSPI1 options

Signals	Option 1	
	PAD	MUX
MISO	CSI0_DAT6	ALT2
	DISP0_DAT22	ALT2
	EIM_D17	ALT1
	KEY_COL1	ALT0
MOSI	CSI0_DAT5	ALT2
	DISP0_DAT21	ALT2
	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.

6.6 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.

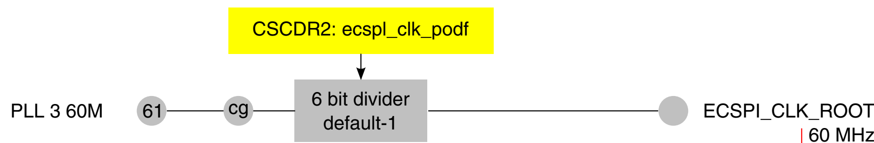


Figure 6-2. eCSPI clock source

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.

6.7 Controller initialization

To initialize the controller, configure the control and configuration registers. For a full listing of eCSPI controller registers, see the eCSPI memory map in the eCSPI chapter of your chip reference manual.

The control register's channel select field's value determines which chip select is used. The channel mode field's value determines the mode (master or slave) for each chip select. See the "Control Register (ECSPIx_CONREG)" section of the chip reference manual's eCSPI chapter for the complete description of these fields.

Note that you must select master mode because this 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. See the "Config Register (ECSPIx_CONFIGREG)" section of the chip reference manual's eCSPI chapter for the complete description of this register.

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 an engineering sample board's 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.

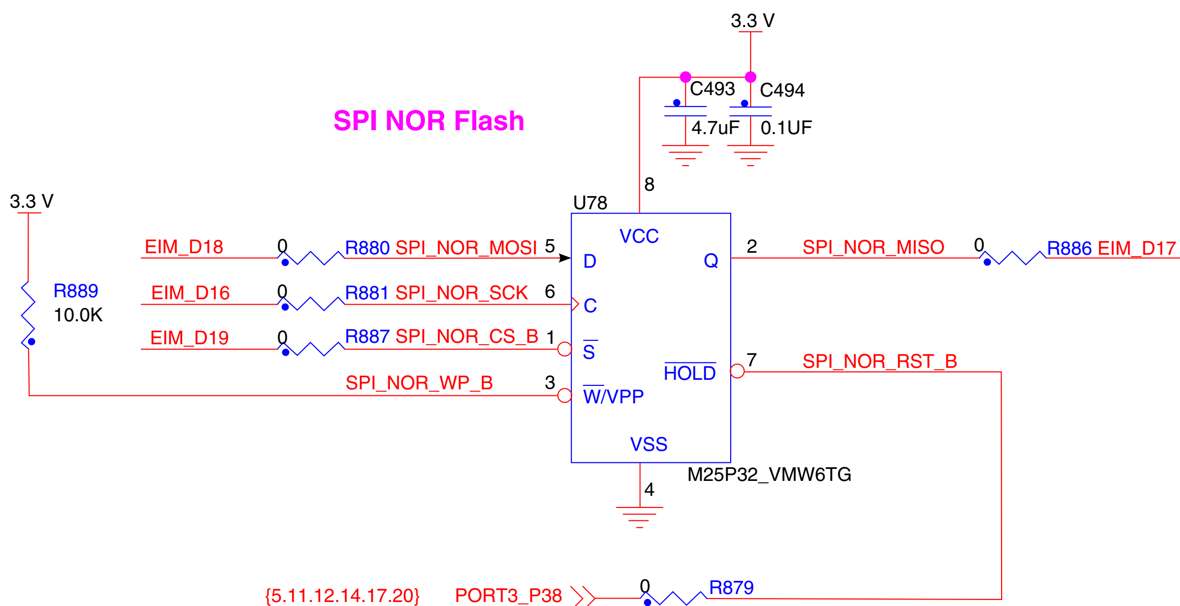


Figure 6-3. eCSPI example configuration

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

6.8 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.

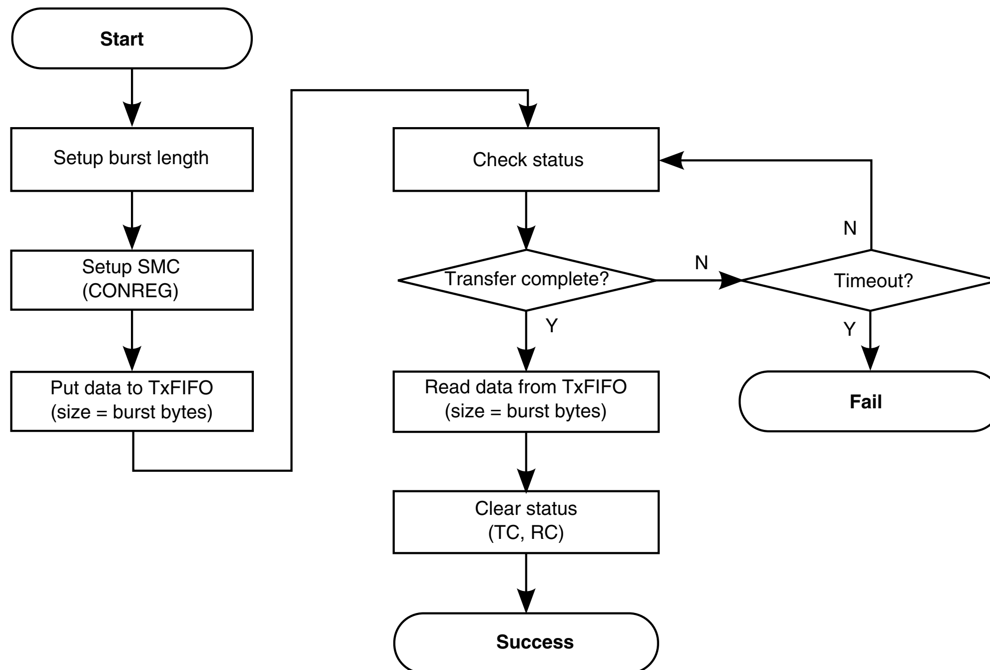


Figure 6-4. Master mode data transfer flow chart

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.

6.9 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 6-3. 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.
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

Chapter 7
Configuring the EIM Driver

7.1 EIM overview

This chapter explains how to configure the driver for the wireless external interface module (EIM). The EIM handles the interface to devices that are external to the chip, including the generation of chip selects (CS), clock, and control for external peripherals and memory.

The EIM provides access as follows:

- Asynchronous access to devices with SRAM-like interface
- Synchronous access to devices with NOR-flash-like or PSRAM-like interface.

Locations on the chip are:

- EIM controller register base = 021B 8000h
- External memory (EIM NOR/RAM CS0) = 0800 0000h.
- CS1 = 256 Mbytes after CS0
- CS2 = 256 Mbytes after CS1
- CS*n+1* = 256 Mbytes after CS*n*

The EIM controller supports the following programmable data port sizes: 8 bit, 16 bit, and 32 bit. It also supports the multiplexed address/data operation, which means A0-A15 can be multiplexed as data pins.

7.2 Feature summary

The EIM has the following features:

Table 7-1. EIM feature summary

Feature	Details
Chip selects	• Up to six chip selects for external devices • Flexible address decoding; each chip select can be configured separately with the configuration register • Individual select signal for each memory space defined • Selectable write protection for each chip select • Support for multiplexed address/data bus operation on 16-bit or 32-bit port size • Programmable data port size for each chip select (8 bit, 16 bit, or 32 bit) • Programmable wait-state generator for each chip select for write and read accesses separately
Accesses	• Asynchronous accesses with programmable setup and hold times for control signals • Support for asynchronous page mode accesses (16-bit and 32-bit port size)
Mode support	• Independent synchronous memory burst read mode support for NOR Flash and PSRAM memories (16-bit and 32-bit port size) • Independent synchronous memory burst write mode support for PSRAM- and NOR-Flash-like memories • Independent programable variable/fix latency support for read and write synchronous (burst) mode • Support for big endian and little endian operation modes per access
General feature support	• Support for NAND Flash devices with NOR Flash like interface-MDOC, OneNand • Support for one ID at a time ARM AXI slave interface • External interrupt support, RDY_INT signal function as external interrupt
Boot from external device support according to boot signals, using RDY_INT signal	• RDY signal support assertion after reset for MDOC device • INT signal support assertion after reset for OneNand device

7.3 Modes of operation

Table 7-2. EIM modes of operation

Mode	Description
Asynchronous mode	Non-burst mode used for SDRAM access
Asynchronous page mode	Allows burst accesses by emulating page mode operation
Multiplexed address/data mode	Address pins DA[15:0] can be multiplexed as data pins
Burst clock mode	Supports burst synchronous operations in various frequencies
Low power mode	Gates input clocks by ACT_CS bits
Boot mode	Boots from external device located on CS0

7.4 Clocks

The following table shows the EIM clock source.

Table 7-3. Reference clocks

Clock	Name	Description
EIM clock root	ACLK_EMI_SLOW_CLK_ROOT	132 MHz source by default. Can be changed through: • CSCMR1[aclk_emi_slow_sel] • CSCMR1[aclk_emi_slow_podf]

7.5 IOMUX pin mapping

The following table shows the pins that are used on the engineering sample board. Because some pins can be MUXed to multiple pads, refer to the board schematic to choose the proper pad.

Table 7-4. IOMUX pin mapping

Signal	PAD	MUX	SION
WEIM_DA_A[0]	EIM_DA0	ALT0	0
WEIM_DA_A[1]	EIM_DA1	ALT0	0
WEIM_DA_A[2]	EIM_DA2	ALT0	0
WEIM_DA_A[3]	EIM_DA3	ALT0	0
WEIM_DA_A[4]	EIM_DA4	ALT0	0
WEIM_DA_A[5]	EIM_DA5	ALT0	0
WEIM_DA_A[6]	EIM_DA6	ALT0	0
WEIM_DA_A[7]	EIM_DA7	ALT0	0
WEIM_DA_A[8]	EIM_DA8	ALT0	0
WEIM_DA_A[9]	EIM_DA9	ALT0	0
WEIM_DA_A[10]	EIM_DA10	ALT0	0
WEIM_DA_A[11]	EIM_DA11	ALT0	0
WEIM_DA_A[12]	EIM_DA12	ALT0	0
WEIM_DA_A[13]	EIM_DA13	ALT0	0
WEIM_DA_A[14]	EIM_DA14	ALT0	0
WEIM_DA_A[15]	EIM_DA15	ALT0	0
WEIM_D[16]	EIM_D16	ALT0	0
WEIM_D[17]	EIM_D17	ALT0	0
WEIM_D[18]	EIM_D18	ALT0	0
WEIM_D[19]	EIM_D19	ALT0	0
WEIM_D[20]	EIM_D20	ALT0	0
WEIM_D[21]	EIM_D21	ALT0	0
WEIM_D[22]	EIM_D22	ALT0	0
WEIM_D[23]	EIM_D23	ALT0	0
WEIM_D[24]	EIM_D24	ALT0	0

Table continues on the next page...

Table 7-4. IOMUX pin mapping (continued)

Signal	PAD	MUX	SION
WEIM_D[25]	EIM_D25	ALT0	0
WEIM_D[26]	EIM_D26	ALT0	0
WEIM_D[27]	EIM_D27	ALT0	0
WEIM_D[28]	EIM_D28	ALT0	0
WEIM_D[29]	EIM_D29	ALT0	0
WEIM_D[30]	EIM_D30	ALT0	0
WEIM_D[31]	EIM_D31	ALT0	0
WEIM_A[16]	EIM_A16	ALT0	0
WEIM_A[17]	EIM_A17	ALT0	0
WEIM_A[18]	EIM_A18	ALT0	0
WEIM_A[19]	EIM_A19	ALT0	0
WEIM_A[20]	EIM_A20	ALT0	0
WEIM_A[21]	EIM_A21	ALT0	0
WEIM_A[22]	EIM_A22	ALT0	0
WEIM_A[23]	EIM_A23	ALT0	0
WEIM_A[24]	EIM_A24	ALT0	0
WEIM_A[25]	EIM_A25	ALT0	0
WEIM_LBA	EIM_LBA	ALT0	0
WEIM_OE	EIM_OE	ALT0	0
WEIM_RW	EIM_RW	ALT0	0
WEIM_CS[0]	EIM_CS0	ALT0	0
WEIM_CS[1]	EIM_CS1	ALT0	0
WEIM_EB[0]	EIM_EB0	ALT0	0
WEIM_EB[1]	EIM_EB1	ALT0	0
WEIM_EB[2]	EIM_EB2	ALT0	0
WEIM_EB[3]	EIM_EB3	ALT0	0
WEIM_WAIT	EIM_WAIT	ALT0	0
WEIM_BCLK	EIM_BCLK	ALT0	0

7.6 Resets and interrupts

This driver does not implement an interrupt mode.

7.7 Initializing the driver

Use the following code to initialize the driver:

```
uint32_t eim_init(…)
{
Configure the DSZ field of GCR1
Configure the MUM field of GCR1
Configure the SHFT field of GCR1
Enable chip select
Set other fields according to external device and HW connection
}
```

7.8 Testing the driver

Build the SDK with the following command:

```
./tools/build_sdk -target mx6dq -board sabre_ai -board_rev a -test eim
```

This generates an ELF and binary file into the following locations:

- `output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-eim-sdk.elf`
- `output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-eim-sdk.bin`

Download `mx6dq_sabre_ai_rev_a` using RV-ICE or Lauterbach Trace32. Alternately, burn `mx6dq_sabre_ai_rev_a` to an SD card with the following command (entered in Windows's "Command Prompt" window):

```
cfimager-imx -o 0 -f mx6dq_sabre_ai_rev_a-eim-sdk.bin -d g:(SD drive name in your PC)
```

Finally, power-up the board to run the test.

The general test routine is as follows:

```
int main(void)
{
          Initialize test buffer
Initialize EIM controller
          Initialize NOR-flash on CS0
          Write data to flash
          Read data back from flash
          Compare data
}
```

Chapter 8
Configuring the EPIT Driver

8.1 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. They are located in the memory map at:

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

8.2 Feature 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

8.3 Modes of operation

The following table explains the EPIT modes of operation:

Table 8-1. 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.

8.4 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()

8.5 Clocks

EPIT receives three clock signals.

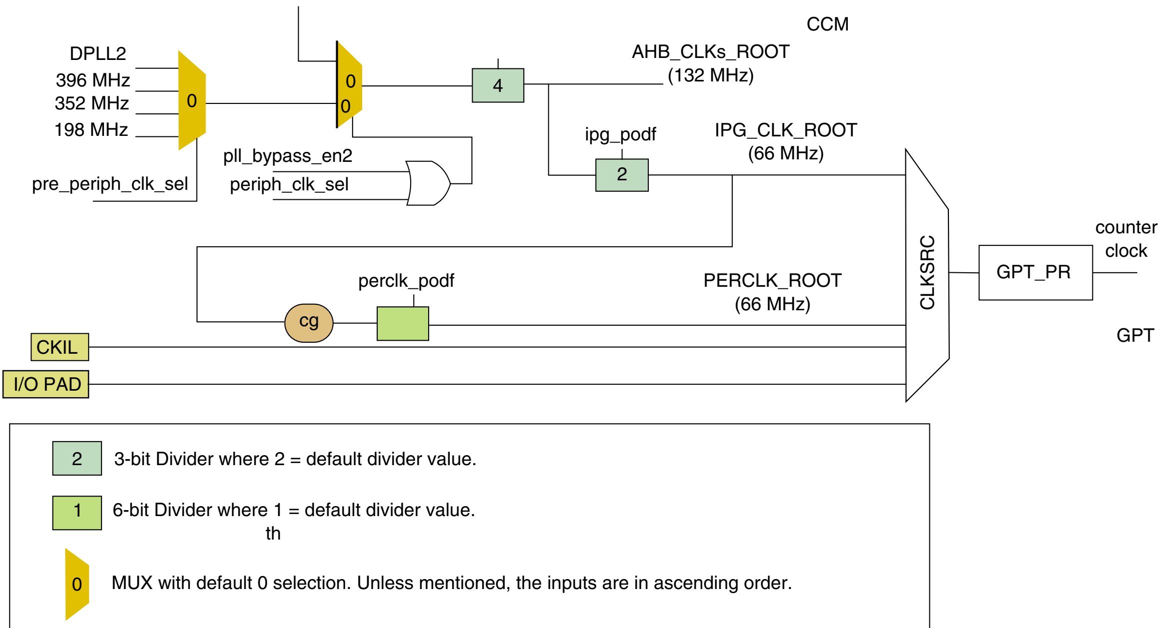


Figure 8-1. Reference clocks

The following table explains the EPIT reference clocks:

Table 8-2. 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.

8.6 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 8-3. EPIT IOMUX pin assignments

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	-	-	-

8.7 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 manual. In the SDK, the list is provided in ./src/include/mx6dq/soc_memory_map.h.

8.8 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 by using the hw_module structure, which is defined in ./src/include/io.h.

The following pseudocode provides an example of 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 following functions typically use the address of this structure.

```
/*!
 * 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 and clear if set.
 * 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)
```

8.9 Testing the EPIT driver

The EPIT driver runs the following tests:

- Delay test
- Tick test

8.9.1 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.

8.9.2 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.

Chapter 9
Configuring the ESAI Driver

9.1 ESAI overview

This chapter describes the module-level operation and programming for the enhanced serial audio interface (ESAI). The pseudocode supplied in the document is based on the source code for the ESAI driver, which is delivered with the i.MX 6 series Platform SDK.

The ESAI provides a full-duplex serial port for serial communication with serial devices, including industry-standard codecs and other DSPs. It consists of independent transmitter and receiver sections and can contain up to six transmitters and up to four receivers.

- Transmitters 2-5 and receivers 0-3 share the following pins:
  - SDO2/SDI3
  - SDO3/SDI2
  - SDO4/SDI1
  - SDO5/SDI0
- Transmitters 0 and 1 share the following pins, which they use exclusively:
  - SDO0
  - SDO1

Each independent transmitter and receiver section has its own clock generator, but all transmitters share the 128-word transmit FIFO and all receivers share the 128-word receive FIFO.

There is only one instance of the ESAI, which is located at base address = 0202 4000h.

9.2 Feature summary

The ESAI driver in the SDK has the following features:

- A simple framework for audio

- ESAI driver
- CS42888 driver for the external audio codec

9.3 Clocks

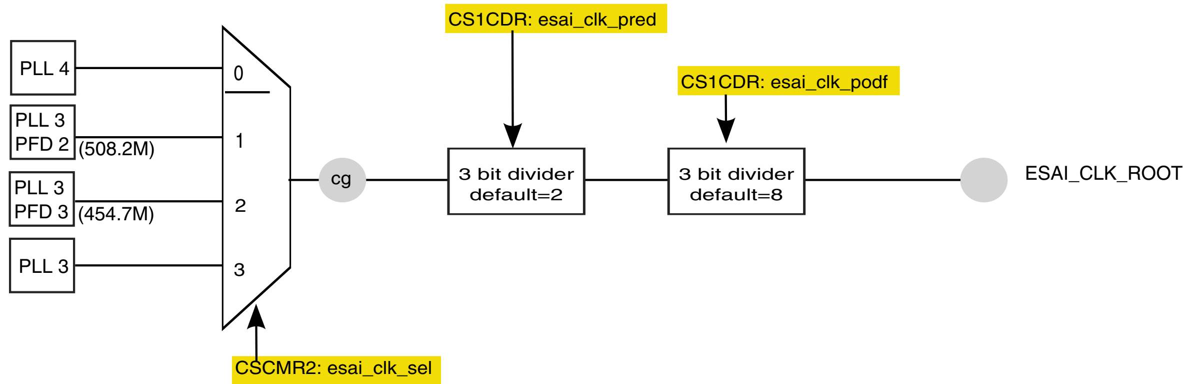


Figure 9-1. ESAI clock tree

Before the ESAI is used, CCM_CCGR1[CG8] must be set to gate on esai_clks. See the "Clock Controller Module" chapter in the chip reference manual for details.

By default, ESAI_CLK_ROOT is sourced from PLL3, which is 480 MHz in this chip. Taking the default CS1CDR esai_clk_pred value of 2 and the default CS1CDR esai_clk_podf value of 8, esai_clk_root is divided to 30 MHz.

9.4 IOMUX pin mapping

The IOMUX pin mapping in the following table is based on an engineering sample board. For other boards, see the board schematics for the specific pin assignments.

Table 9-1. Example IOMUX pin mapping

Signal name	Pin name	ALT	Daisy chain involved
ESAI_FSR	ENET_REF_CLK	ALT2	YES
ESAI1_FST	ENET_RXD1	ALT2	YES
ESAI_HCKT	ENET_RXD0	ALT2	YES
ESAI_SCKR	ENET_MDIO	ALT2	YES
ESAI_SCKT	ENET_CRS_DV	ALT2	YES

Table continues on the next page...

Table 9-1. Example IOMUX pin mapping (continued)

Signal name	Pin name	ALT	Daisy chain involved
ESAI_SDO0	NANDF_CS2	ALT2	YES
ESAI_ SDO1	NANDF_CS3	ALT2	YES
ESAI_SDO2/SDI3	ENET_TXD1	ALT2	YES
ESAI_ SDO3/SDI2	ENET_TX_EN	ALT2	YES
ESAI_ SDO4/SDI1	ENET_TXD0	ALT2	YES
ESAI_ SDO5/SDI0	ENET_MDC	ALT2	YES

9.5 External ESAI signal description

Table 9-2. ESAI signal descriptions (external)

Signal name	Description	I/O
SDO0	Used for transmitting data from the ESAI_TX0 serial transmit shift register.	O
SDO1	Used for transmitting data from the ESAI_TX1 serial transmit shift register.	O
SDO2/SDI3	• Used as SDO2 for transmitting data from the ESAI_TX2 serial transmit shift register when programmed as a transmitter pin • Used as the SDI3 signal for receiving serial data to the ESAI_RX3 serial receive shift register when programmed as a receiver pin	I/O
SDO3/SDI2	• Used as the SDO3 signal for transmitting data from the ESAI_TX3 serial transmit shift register when programmed as a transmitter pin • Used as the SDI2 signal for receiving serial data to the ESAI_RX2 serial receive shift register when programmed as a receiver pin	I/O
SDO4/SDI1	• Used as SDO4 for transmitting data from the ESAI_TX4 serial transmit shift register when programmed as transmitter pin • Used as SDI1 for receiving serial data to the RX1 serial receive shift register when programmed as a receiver pin	I/O
SDO5/SDI0	• Used as SDO5 for transmitting data from the ESAI_TX5 serial transmit shift register when programmed as transmitter pin • Used as SDI0 for receiving serial data to the ESAI_RX0 serial shift register when programmed as a receiver pin	I/O
SCKR	SCKR is a bidirectional pin providing the receivers' serial bit clock for the ESAI interface. The direction can be programmed.	I/O
SCKT	SCKT is a bidirectional pin providing the transmitters' serial bit clock for the ESAI interface. The direction can be programmed.	I/O
FSR	FSR is a bidirectional pin providing the receivers' frame sync signal for the ESAI interface.	I/O
FST	FST is a bidirectional pin providing the frame sync for both the transmitters and receivers in the synchronous mode (SYN=1) and for the transmitters only in asynchronous mode	I/O
HCKT	HCKT is a bidirectional pin providing the transmitters high frequency clock for the ESAI interface.	I/O
HCKR	HCKR is a bidirectional pin providing the receivers high frequency clock for the ESAI interface.	I/O

9.6 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 CS42888)

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.

9.6.1 audio_card_t data structure

The data structure `audio_card_t`, which describes the audio card, is:

```
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;
```

9.6.2 audio_ctrl_t data structure

The data structure `audio_ctrl_t`, which describes the audio controller, is:

```
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;
```

9.6.3 audio_codec_t data structure

The data structureaudio_codec_t, which describes the audio codec, is:

```
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;
```

9.6.4 audio_dev_ops_t data structure

The data structure audio_dev_ops_t, which describes the APIs of audio devices (codec, controller, and card), is:

```
 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;
```

9.6.5 audio_dev_para_t data structure

The data structure audio_dev_para_t, which describes the audio parameter that is passed to the configuration function is:

```
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;
```

9.7 ESAI driver functions

The ESAI driver has both local functions and public APIs.

The local functions are used to:

- Reset the ESAI
- Obtain the ESAI setting and status values

- Set ESAI parameters
- Enable ESAI sub-modules

The public APIs are used to:

- Initialize and deintialize the ESAI driver
- Configure the ESAI
- Playback through the ESAI

9.7.1 Resetting the ESAI

The ESAI and its submodules (transmitters, receivers, Tx FIFO, and Rx FIFO) can be reset, using the following function:

```
static int32_t esai_reset(audio_ctrl_p ctrl)
```

9.7.2 Obtaining ESAI parameters

The function `uint32_t esai_get_hw_para(audio_ctrl_p ctrl, uint32_t type)` returns the ESAI parameter values according to the parameter type, as follows:

```
typedef enum {
    ESAI_HW_PARA_ECR,
    ESAI_HW_PARA_TCR,
    ESAI_HW_PARA_RCR,
    ESAI_HW_PARA_TCCR,
    ESAI_HW_PARA_RCCR,
    ESAI_HW_PARA_TFCR,
    ESAI_HW_PARA_RFCR,
    ESAI_HW_PARA_TSR,
    ESAI_HW_PARA_SAICR,
    ESAI_HW_PARA_TSM,           //time slot mask
    ESAI_HW_PARA_RSM,           //time slot mask
    ESAI_HW_PARA_TX_WL,         //word len in bits
    ESAI_HW_PARA_RX_WL,         //word len in bits
} esai_hw_para_type_e;
```

The function can be called once ESAI has been initialized.

9.7.3 Setting ESAI parameters

The function static `uint32_t esai_set_hw_para(audio_ctrl_p ctrl, uint32_t type, uint32_t val)` sets ESAI parameters according to the parameter type. The supported parameter types are listed in the enumeration esai_hw_para_type_e.

9.7.4 Obtaining ESAI status

The function static uint32_t esai_get_status(audio_ctrl_p ctrl, uint32_t type) obtains the ESAI status according to status type. The status types are supported as follows:

```
typedef enum {
    ESAI_STATUS_ESR,
    ESAI_STATUS_TFSR,
    ESAI_STATUS_RFSR,
    ESAI_STATUS_SAISR,
} esai_status_e;
```

9.7.5 Enabling ESAI submodules

ESAI and its sub-modules can be enabled or disabled individually. The function `static uint32_t esai_sub_enable(audio_ctrl_p ctrl, uint32_t type, uint32_t val)` can enabled or disabled ESAI or its sub-modules according enabling types as follows:

```
typedef enum {
    ESAI_SUB_ENABLE_TYPE_ESAI,
    ESAI_SUB_ENABLE_TYPE_TX,
    ESAI_SUB_ENABLE_TYPE_RX,
    ESAI_SUB_ENABLE_TYPE_TXFIFO,
    ESAI_SUB_ENABLE_TYPE_RXFIFO,
} esai_sub_enable_type_e;
```

9.7.6 Initializing the ESAI

Before the ESAI driver can be used to play audio, the ESAI module must be initialized using the function `int esai_init(void *priv)`. Initialization includes the following:

- Configuring the IOMUX for external ESAI signals.
- Setting the clock, such as selecting the clock source and gating on clocks for ESAI.
- Resetting the ESAI module and puting all registers into their reset value.

9.7.7 Configuring the ESAI

The function `int esai_config(void *priv, audio_dev_para_p para)` configures the ESAI parameters according to the `audio_dev_para` data structure that is passed by the audio card driver. This function:

- Sets the direction of bit clock and frame sync clock
- Sets the attribute of bit clock and frame sync clock, such as polarity and frame sync length
- Sets bit clock dividers if the internal bit clock was used
- Sets frame length

- Sets word length and slot length
- Sets FIFO's watermarks.
- Fills zeros to Tx FIFO if Tx FIFO is used.
- Enables the ESAI
- Enables Tx FIFO, Rx FIFO, the transmitters, and the receivers

9.7.8 Playback through ESAI

After initialization and configuration, data can be written to ESAI Tx FIFO to play back music. ESAI_ESR[TFE] polls to determine whether the Tx FIFO is empty or not. If Tx FIFO is empty (the data count in Tx FIFO is less than the watermark), data can be written to it according to the word length (TFCR[TWA]).

9.7.9 ESAI de-initialization

This function de-initializes the ESAI and frees the resources that the ESAI had been using.

9.8 CS42888 driver

CS42888 is one of many codecs that have an ESAI interface and can be used as an external audio codec. This chapter discusses the ESAI controller itself, and therefore does not provide details about the CS42888 driver. See the CS42888 driver for details.

9.9 Testing the unit

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

```
tools/build_sdk -target mx6dq -board evb -test audio -clean
```

This creates the following ELF and binary files:

```
output/mx6dq/evb_rev_a/bin/mx6dqevb-audio-sdk.elf
output/mx6dq/evb_rev_a/bin/mx6dqevb-audio-sdk.bin
```

Use RV-ICE or Lauterbach to download mx6dqevb-audio-sdk.elf or burn mx6dqevb-audio-sdk.bin to an SD card by entering the following command in Windows's command prompt window:

```
cfimager-imx -o 0 -f mx6dqevb-audio-sdk.bin -d g:(SD drive name in your PC)
```

Then, power-up the board and select ESAI playback according to the prompt in the terminal. This runs the ESAI test unit.

The ESAI test unit demonstrates how to use the audio framework to play back music. The test unit works as follows:

1. Initialize the snd_card_esai which includes ESAI and CS42888.
2. Configure the snd_card_esai.
3. Write the music file to the snd_card_esai, that is, playback music.
4. If "exit" selected by the user, de-initialize the snd_card_esai and return.

If the test is successful, you will hear a voice in the headphone.

Chapter 10
Configuring the Ethernet Driver

10.1 Overview

This chapter explains how to use the driver for the MAC-NET core, which implements a triple speed 10/100/1000 Mbps Ethernet MAC compliant with the IEEE 802.3-2002 standard. The MAC layer provides compatibility with half- or full-duplex 10/100 Mbps Ethernet LANs and full-duplex gigabit Ethernet LANs.

The core also implements a hardware acceleration block that optimizes the performance of network controllers providing IP and TCP, UDP, ICMP protocol services. The acceleration block performs critical functions in hardware, which are typically implemented with large software overhead.

The programmable 10/100/1000 Ethernet MAC with IEEE 1588 integrates a standard IEEE 802.3 Ethernet MAC with a time-stamping module.

10.2 Feature summary

The following table summarizes the MAC-NET core features.

Table 10-1. Feature summary

Protocol	Feature list

Ethernet MAC	• Implements the full 802.3 specification with preamble/SFD generation, frame padding generation, CRC generation, and checking
	• Dynamically configurable to support 10/100 Mbps and Gigabit operation
	• Supports 10/100 Mbps full duplex and configurable half duplex operation
	• Supports gigabit full duplex operation
	• Supports AMD magic packet detection with interrupt for node remote power management
	• Seamless interface to commercial Ethernet PHY device via one of the following:
	    • A 4-bit MII (medium independent interface) operating at 25 MHz
	    • A 2-bit RMII (reduced medium independent interface) operating at 50 MHz
	    • A double data rate 4-bit RGMII (reduced gigabit media independent interface) operating at 125 MHz
	• Simple 64-Bit FIFO interface to user application
	• CRC-32 checking at full speed with optional forwarding of the frame check sequence
	• (FCS) field to the client
	• CRC-32 generation and append on transmit or forwarding of user application provided FCS selectable on a per-frame basis
	• When operating in full duplex mode, the MAC-NET core includes the following:
	    • Automated pause frame (802.3 x31A) generation and termination providing flow control without user application intervention
	    • Pause quanta used to form pause frames, dynamically programmable
	    • Pause frame generation additionally controllable by user application offering flexible traffic flow control
	    • Optional forwarding of received pause frames to the user application
	• In half-duplex mode, provides full collision support, including jamming, back off, and automatic retransmission
	• Support for VLAN-tagged frames according to IEEE 802.1Q
	• Programmable MAC address: insertion on transmit and discards frames with mismatching destination address on receive (except broadcast and pause frames)
	• Programmable promiscuous mode support to omit MAC destination address checking on receive
	• Multicast and unicast address filtering on receive based on 64 entries hash table reducing higher layer processing load
	• Programmable frame maximum length providing support for any standard or proprietary frame length
	• Statistics indicators for frame traffic and errors (alignment, CRC, length) and pause frames providing for IEEE 802.3 basic and mandatory management information database (MIB) package and remote network monitoring (RFC 2819)
	• Simple handshake user application FIFO interface with fully programmable depth and threshold levels
	• Separate status word available for each received frame on the user interface, providing information such as frame length, frame type, VLAN tag, and error information
	• Multiple internal loopback options
	• MDIO master interface for PHY device configuration and management with two programmable MDIO base addresses
	• Supports legacy FEC buffer descriptors

Table continues on the next page...

Table 10-1. Feature summary (continued)

Protocol	Feature list
IP Protocol Performance Optimization	• Operates on TCP/IP and UDP/IP and ICMP/IP protocol data or IP header only • Enables wire-speed processing • IPv4 and IPv6 support • Transparent passing of frames of other types and protocols • Support for VLAN tagged frames according to IEEE 802.1q with transparent forwarding of VLAN tag and control field • Automatic IP-header and payload (protocol specific) checksum calculation and verification on receive • Automatic IP-header and payload (protocol specific) checksum generation and automatic insertion on transmit configurable on a per-frame basis • Support for IP and TCP, UDP, ICMP data for checksum generation and checking • Full header options support for IPv4 and TCP protocol headers • IPv6 support limited to datagrams with base header only. Datagrams with extension headers are passed transparently unmodified/unchecked. • Statistics information for received IP and protocol errors • Configurable automatic discard of erroneous frames • Configurable automatic host-to-network (Rx) and network-to-host (Tx) byte order conversion for IP and TCP/UDP/ICMP headers within the frame • Configurable padding remove for short IP datagrams on receive • Configurable Ethernet payload alignment to allow for 32-bit word aligned header and payload processing • Programmable store-and-forward operation with clock and rate decoupling FIFOs
IEEE 1588	• Support for all IEEE 1588 frames • Reference clock can be chosen independently of the network speed • Software-programmable precise time-stamping of ingress and egress frames • Timer monitoring capabilities for system calibration and timing accuracy management • Precise time-stamping of external events with programmable interrupt generation • Programmable event and interrupt generation for external system control • Hardware- and software-controllable timer synchronization • 4 channel IEEE 1588 timer, each with support for input capture and output compare using the 1588 counter

10.3 Modes of operation

Table 10-2. Ethernet modes of operation

Mode	What it does
10M	10 Mbps
100M	100 Mbps
1000M	1000 Mbps

10.4 Clocks

Table 10-3. Reference clock

Clock	Name	Description
Ethernet PLL	Ethernet PLL	The PLL outputs a 500 MHz clock.

10.5 IOMUX pin mapping

Table 10-4. IOMUX pin mapping for the Ethernet driver

Signals	Driver			Description
	PAD	MUX	SION	
MDC	KEY_COL2	IOMUXC_SW_MUX_CTL_PAD_KEY_COL2	-	MDC
-	-	IOMUXC_SW_PAD_CTL_PAD_KEY_COL2	-	Pad control
MDIO	KEY_COL1	IOMUXC_SW_MUX_CTL_PAD_KEY_COL1	-	MDIO
-	-	IOMUXC_ENET_IPP_IND_MAC0_MDIO_SELECT_INPUT	-	Select KEY_COL1 Involved in Daisy Chain
-	-	IOMUXC_SW_PAD_CTL_PAD_KEY_COL1	-	Pad control
RGMII_RXD0	RGMII_RD0	IOMUXC_SW_MUX_CTL_PAD_RGMII_RD0	-	RXD0
-	-	IOMUXC_ENET_IPP_IND_MAC0_RXDATA_0_SELECT_INPUT	-	Select RGMII_RD0 Involved in Daisy Chain
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_RD0	-	Pad control
-	-	IOMUXC_SW_PAD_CTL_GRP_DDR_TYPE_RGMII	-	DDR Select Field
-	-	IOMUXC_SW_PAD_CTL_GRP_RGMII_TERM	-	On Die Termination Field
RGMII_RXD1	RGMII_RD1	IOMUXC_SW_MUX_CTL_PAD_RGMII_RD1	-	RXD1
-	-	IOMUXC_ENET_IPP_IND_MAC0_RXDATA_1_SELECT_INPUT	-	Select RGMII_RD1 Involved in Daisy Chain
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_RD1	-	Pad control
RGMII_RXD2	RGMII_RD2	IOMUXC_SW_MUX_CTL_PAD_RGMII_RD2	-	RXD2
-	-	IOMUXC_ENET_IPP_IND_MAC0_RXDATA_2_SELECT_INPUT	-	Select RGMII_RD2 involved in Daisy Chain
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_RD2	-	Pad control
RGMII_RXD3	RGMII_RD3	IOMUXC_SW_MUX_CTL_PAD_RGMII_RD3	-	RXD3

Table continues on the next page...

Table 10-4. IOMUX pin mapping for the Ethernet driver (continued)

Signals	Driver			Description
	PAD	MUX	SION	
-	-	IOMUXC_ENET_IPP_IND_MAC0_RXDATA_3_SELECT_INPUT	-	Select RGMII_RD3 involved in Daisy Chain
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_RD3	-	Pad control
RGMII_RX_CTL	RGMII_RX_CTL	IOMUXC_SW_MUX_CTL_PAD_RGMII_RX_CTL	-	RX control
-	-	IOMUXC_ENET_IPP_IND_MAC0_RXEN_SELECT_INPUT	-	Select RGMII_RX_CTL involved in Daisy Chain
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_RX_CTL	-	Pad control
RGMII_RXC	RGMII_RXC	IOMUXC_SW_MUX_CTL_PAD_RGMII_RXC	-	RX clock
-	-	IOMUXC_ENET_IPP_IND_MAC0_RXCLK_SELECT_INPUT	-	Select RGMII_RXC involved in Daisy Chain
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_RXC	-	Pad control
RGMII_TD0	RGMII_TD0	IOMUXC_SW_MUX_CTL_PAD_RGMII_TD0	-	TXD0
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_TD0	-	Pad control
RGMII_TD1	RGMII_TD1	IOMUXC_SW_MUX_CTL_PAD_RGMII_TD1	-	TXD1
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_TD1	-	Pad control
RGMII_TD2	RGMII_TD2	IOMUXC_SW_MUX_CTL_PAD_RGMII_TD2	-	TXD2
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_TD2	-	Pad contorl
RGMII_TD3	RGMII_TD3	IOMUXC_SW_MUX_CTL_PAD_RGMII_TD3	-	TXD3
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_TD3	-	Pad control
RGMII_TX_CTL	RGMII_TX_CTL	IOMUXC_SW_MUX_CTL_PAD_RGMII_TX_CTL	-	TX control
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_TX_CTL	-	Pad control
RGMII_TXC	RGMII_TXC	IOMUXC_SW_MUX_CTL_PAD_RGMII_TXC	-	TX clock
-	-	IOMUXC_SW_PAD_CTL_PAD_RGMII_TXC	-	Pad control
ENET_REF_CLK	ENET_REF_CLK	IOMUXC_SW_MUX_CTL_PAD_ENET_REF_CLK	-	Enet refernece clock
-	-	IOMUXC_SW_PAD_CTL_PAD_ENET_REF_CLK	-	Pad control

10.6 Resets and interrupts

The Ethernet IRQ number is 149. This SDK does not implement an interrupt mode.

10.7 Initializing the driver

To initialize the driver, use the following pseudocode:

```
void imx_ar8031_iomux(void)
{
          /*intialize IOMUX of ethernet*/
}
int imx_enet_init(imx_enet_priv_t * dev, unsigned long reg_base, int phy_addr)
{
          /*1.initialize BD buffers*/
          /*2.initialize ethernet chip*/
}
int imx_enet_mii_type(imx_enet_priv_t * dev, enum imx_mii_type mii_type)
{
          /*Set MMI type: RMII or RGMII*/
}
void imx_enet_phy_init(imx_enet_priv_t * dev)
{
          /*initialize ethernet PHY*/
}
```

10.8 Testing the driver

To test the driver, run the following code. Note that a loop cable is required on the RJ45 interface.

```
int main(void)
{
    imx_ar8031_iomux();
    //init enet0
    imx_enet_init(dev0, ENET_BASE_ADDR, 0);
    imx_enet_mii_type(dev0, RGMII);
    //init phy0.
imx_enet_phy_init(dev0);
/*send data*/
/*receive data*/
/*compare data*/
}
```

Chapter 11
Configuring the FlexCAN Modules

11.1 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.

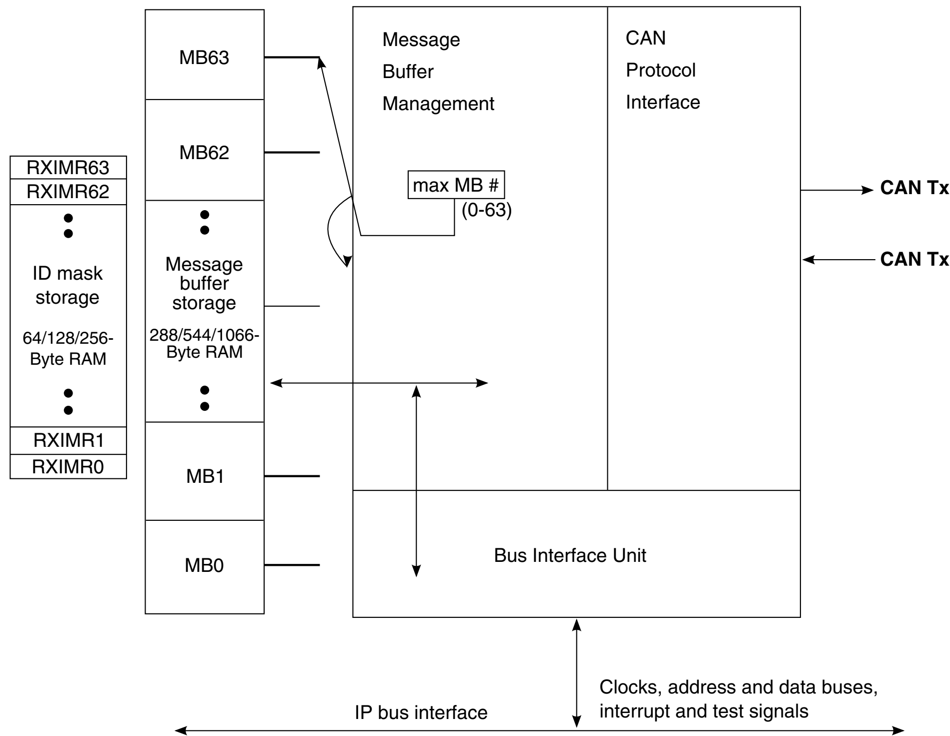


Figure 11-1. FlexCAN block diagram

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

11.2 Feature 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

11.3 Modes of operation

The following table summarizes the FlexCAN modes of operation.

Table 11-1. 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.

11.4 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.

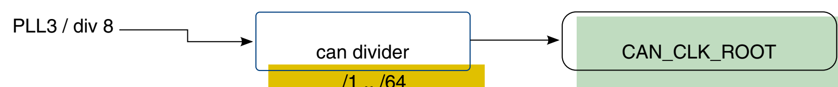


Figure 11-2. Clocking figure

Table 11-2. 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.

11.5 Module timing

The FlexCAN bit rate is derived from the serial clock, which is generated by dividing the PE clock by the programmed PRESDIV value. Each serial clock (Sclock, or Ftq. time quantum frequency) period is also referred to as a time quantum. The FlexCAN bitrate is defined as the Sclock divided by the number of time quanta, where time quanta are further broken down segments within the bit time (time to transmit and sample a bit).

The following list shows the CAN bitrates that are supported in the CAN module driver. It is possible to support other bit rates within the range of the default supported bit rates by updating the clocks and required time quanta in the **can_update_bitrate(struct imx_flexcan *can_module)** function to support the new bit rates.

- 1 Mbytes/s
- 800 Kbytes/s
- 500 Kbytes/s
- 250 Kbytes/s
- 125 Kbytes/s
- 62.5 Kbytes/s
- 20 Kbytes/s
- 10 Kbytes/s

11.6 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 11-3. 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 is 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.

11.7 Resets and interrupts

11.7.1 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
}
```

11.7.2 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.

11.8 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.

11.9 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 ----
```

Chapter 12
Configuring the GPU3D Driver

12.1 Overview

This chapter explains GPU3D operation and programming at the block level. All supplied pseudocode is based on the source code of GPU3D driver, which is delivered with the i.MX 6Dual/6Quad and i.MX 6Solo/6DualLite Platform SDKs.

The GPU3D is a high-performance core that delivers hardware acceleration for 3D graphics display for screen sizes ranging from the smallest cell phones to HD 1080p displays. The core accelerates numerous 3D graphics applications, including graphical user interfaces (GUI), menu displays, flash animation, and gaming.

There is only one instance of the GPU3D located in the memory map at GPU3D base address = 0013 0000h

12.2 Feature summary

The GPU3D has the following hardware features:

- OpenGL ES 2.0 compliance, including the following extensions
    - OpenGL ES 1.1
    - OpenVG 1.1
- IEEE 32-bit floating-point pipeline
- Ultra-threaded, unified vertex, and fragment shaders
- Low bandwidth at both high and low data rates
- Low CPU loading
- Dependent texture operation with high-performance
- Alpha blending
- Depth and stencil compare
- Support for the following textures:
    - Fragment shader simultaneous textures

- Vertex shader simultaneous textures
- Point sampling, bi-linear sampling, tri-linear filtering, and cubic textures
- Resolve and fast clear
- 8 x 8 kpixel texture size and 8 x 8 kpixel rendering target
- Vertex DMA streams
- 2 texture units and 2 pixel units for higher pixel processing rate
- Supports YUV format in display output (YUYV 4:2:0)

12.3 Modes of operation

The GPU3D supports four operation modes, as described in the following table. This driver configures the GPU3D controller for working in active mode.

Table 12-1. GPU3D modes of operation

Mode	What it does
Active mode	GPU is actively processing commands. One or more blocks are not in idle mode.
Idle mode	GPU is not processing any commands. All modules in the pipeline are in idle state.
Standby mode	All input clocks to the GPU are shut off.miriamgmir
Sleep mode	The entire GPU is powered off through power gating.

12.4 Clocks

Table 12-2. GPU3D reference clocks

Clock	Name	Description
GPU3D core clock	GPU3D_CORE_CLK_ROOT	Clock for GPU3D core
GPU3D shader clock	GPU3D_SHADER_CLK_ROOT	Clock for GPU3D shader

12.5 IOMUX pin mapping

There is no need to configure pins for GPU3D.

12.6 Resets and interrupts

The driver does not implement interrupt mode.

12.7 Initializing the GPU3D driver

The initialization procedure driver requires the following procedure:

1. Initialize the IPU, including IOMUX, power, etc.
2. Configure the LVDS interface and the LVDS panel.
3. Configure the IPU to support XGA.
4. Initialize GPU3D and run the command buffer.

12.8 Testing the GPU3D driver

The GPU3D testing demo is based on an engineering sample board and can be easily ported to other boards.

Build the SDK with the following command to generate ELF and binary files:

```
tools/build_sdk -target=mx6dq -board=evb -test=gpu -clean
```

Download `mx6dq/evb_rev_a/bin/mx6dq_evb_rev_a-gpu-sdk.elf` using RV-ICE or Lauterbach, or burn `mx6dq/evb_rev_a/bin/mx6dq_evb_rev_a-gpu-sdk.elf` to an SD card by entering the following command in Windows's command prompt window:

```
cfimager-imx -o 0 -f mx6dq_evb_rev_a-gpu-sdk.bin -d g:(SD drive name in your PC)
```

Then power up the board to run the test.

To test the driver, connect an LVDS display panel to LVDS0 on the board. The test shows a 3-D scene in the LVDS display panel.

Chapter 13
Configuring the GPMI Controller

13.1 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, including source synchronous DDR mode and the Samsung/Toshiba Toggle Mode DDR protocol.

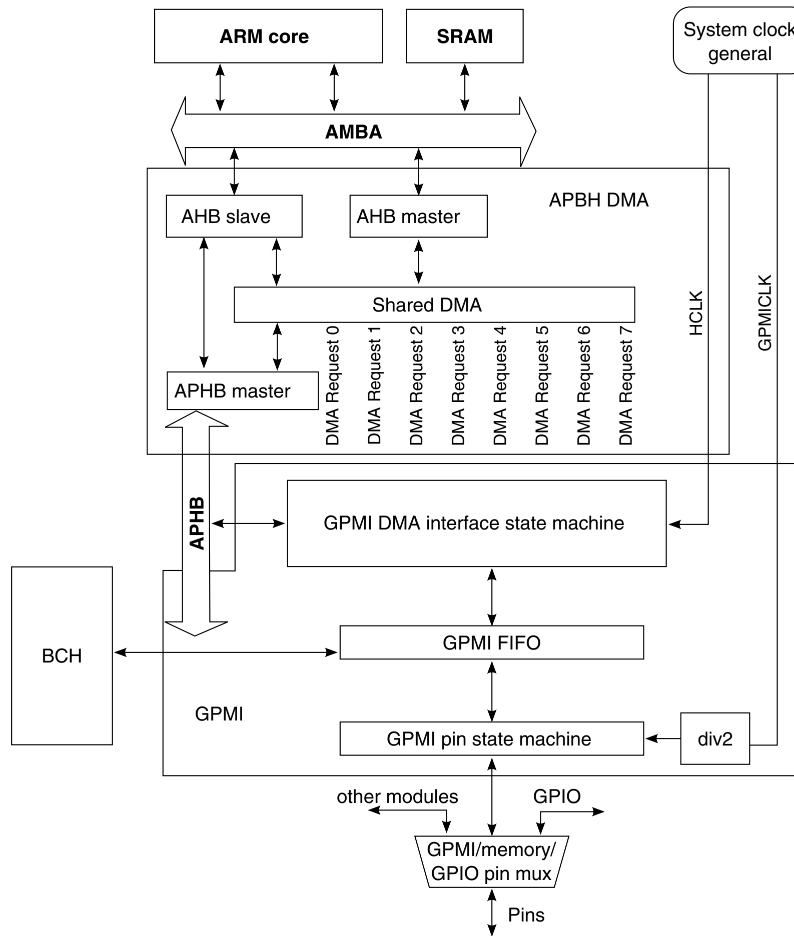


Figure 13-1. GPMI block diagram

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 both the GPMI and BCH modules. There is also a single associated APBH DMA module. These modules are memory mapped to the following locations:

- APBH DMA base address = 0011 0000h
- GPMI base address = 0011 2000h
- BCH base address = 0011 4000h

13.2 Feature summary

The key features are:

- Individual chip select and ready/busy pins for up to eight NAND devices
- Option to use ganged ready/busy mode to reduce the number of ready/busy pins to one
- Individual state machine and DMA channel for each chip select
- Special command modes that work with the DMA controller, using chained descriptors to perform arbitrarily complex NAND functions without CPU intervention
- Configurable timing based on a dedicated clock allows the optimal balance of high NAND performance and low system power
- Direct connection to a BCH ECC engine with support for up to 40-bit correction per 512 or 1024 bytes.

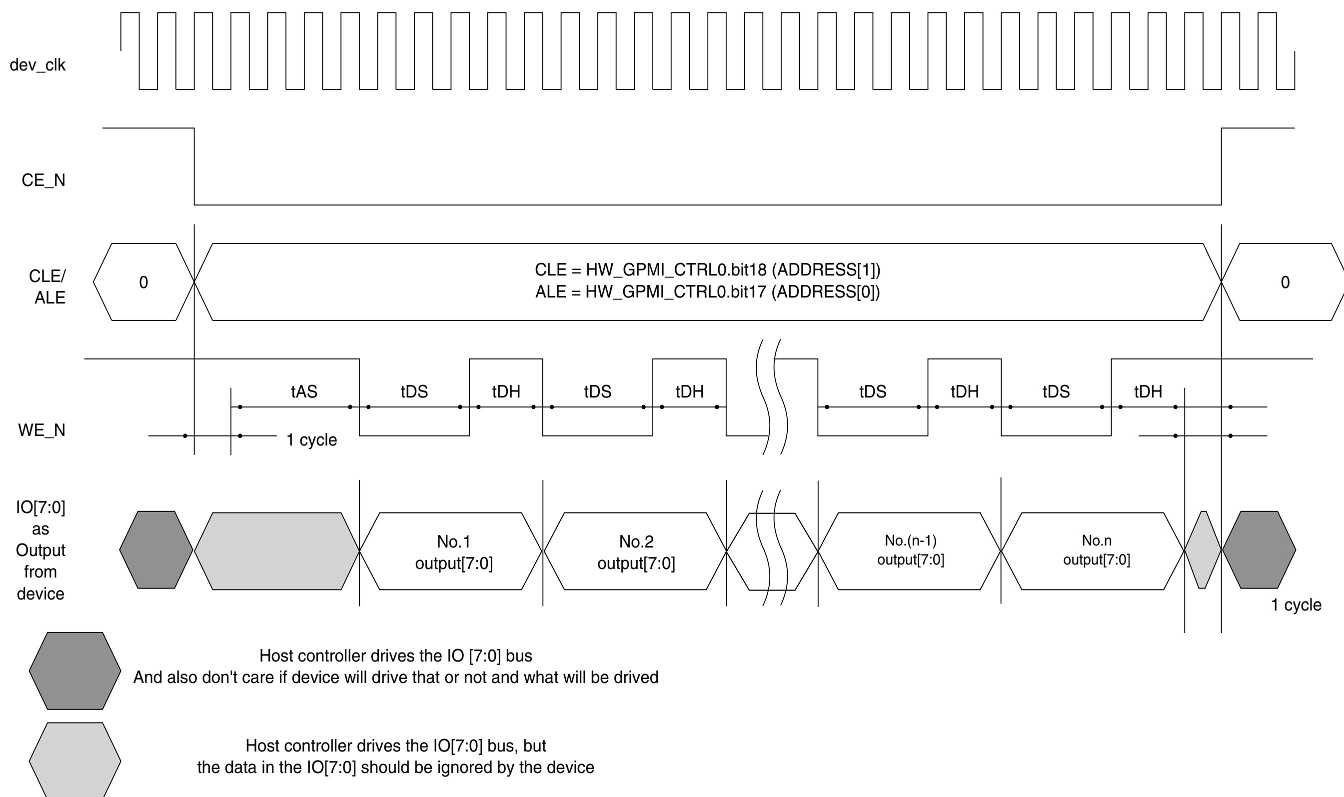
The GPMI and the DMA have been designed to handle complex multi-page operations without CPU intervention. The DMA uses a linked descriptor function with branching capability to automatically handle all of the operations needed to read/write multiple pages.

13.3 Modes of operation

Table 13-1. 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.

13.4 Basic NAND timing



- 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 the chip reference manual.

13.5 Clocks

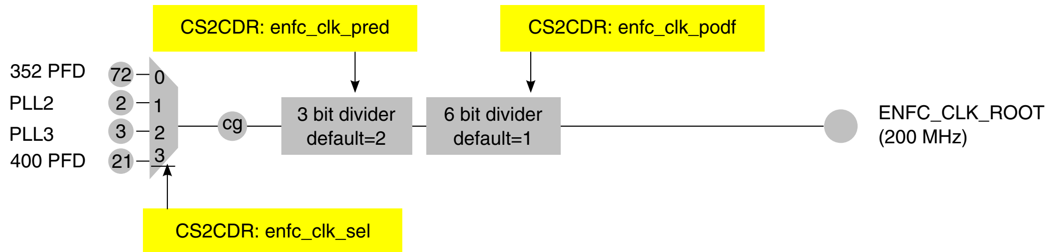


Figure 13-3. GPMI clock tree

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.

13.6 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 13-2. 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
CE3N	NANDF_CS3	ALT0	0

Table continues on the next page...

Table 13-2. GPMI IOMUX pin mapping options (continued)

Signal	GPMI		
	PAD	MUX	SION
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

The NANDF_RB*x* and NANDF_CS*x* signals need to be pulled up, as the NAND Flash device only drives them low. This can be accomplished either with external pull-up resistors, or by using the internal, on-chip pull-ups. To enable the internal pull-ups for NANDF_RB0, use the IOMUXC_IOMUXC_SW_PAD_CTL_PAD_NANDF_RB0 register. Set IOMUXC_IOMUXC_SW_PAD_CTL_PAD_NANDF_RB0[PUS] to 3, to select the 22kΩ pull-up. Also, be sure to set IOMUXC_IOMUXC_SW_PAD_CTL_PAD_NANDF_RB0[PUE] to 1 to select pull mode, and set IOMUXC_IOMUXC_SW_PAD_CTL_PAD_NANDF_RB0[PKE] to 1 to enable the pull-up. Other NANDF_RB*x* or NAND_CS*x* pads are configured similarly. If using an external pull-up resistor, you may wish to use a stronger pull-up such as 10kΩ.

The GPMI module has the option of using a wired-OR configuration for multiple ready/busy signals, so that only one ready/busy input pin (NANDF_RB0) is required. Most boards will use this configuration. To enable ganged ready/busy mode, set HW_GPMI_CTRL1[GANGED_RDYBUSY].

13.7 APBH DMA

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

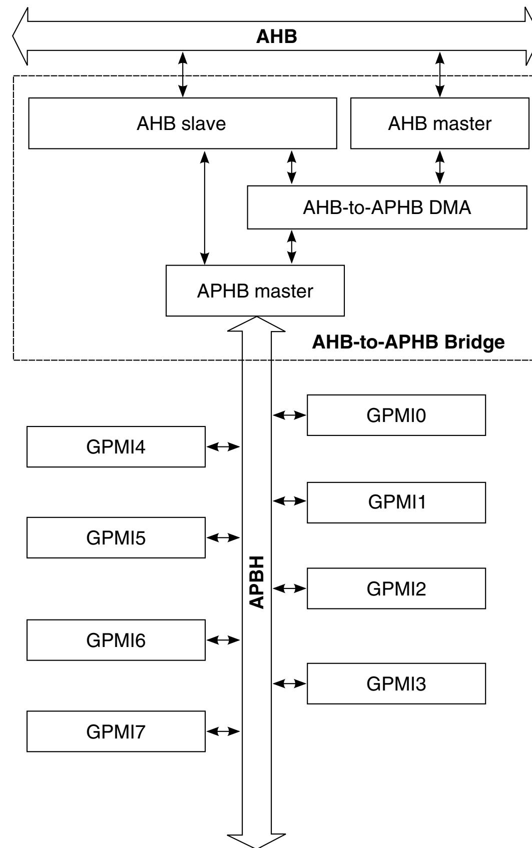


Figure 13-4. APBH DMA block diagram

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).

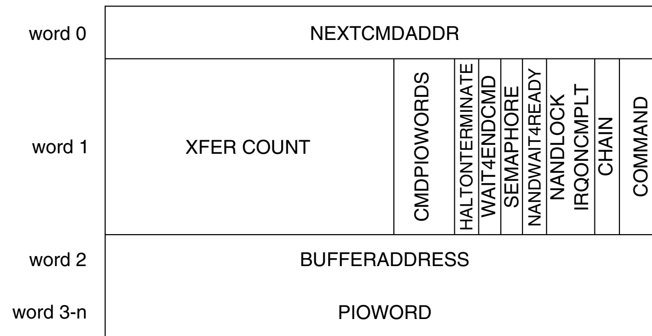


Figure 13-5. Channel-command word structure

13.8 BCH ECC

The hardware BCH (Bose, Ray-Chaudhuri, Hocquenghem) ECC accelerator provides a forward error-correction function for improving the reliability of NAND Flash media. It is capable of correcting from 2 to 40 single bit errors within a block of data no larger than about 1900 bytes, with either 512 bytes or 1024 bytes being typical.

The important things to remember when using the BCH engine are:

- Data block sizes must be a multiple of 4 bytes and be aligned in system memory.
- Metadata is always written at the beginning of the flash page to facilitate fast access for file-system operations.
- The BCH does not directly support a partial page write. Because most NAND Flash devices do not support partial page write, this is rarely a limitation. However, if a partial page write is desired, it can be accomplished by programming the BCH layout registers such that the BCH only sees a portion of the page (but see the note below about byte alignment).
- Flash read operations through the BCH can read either the entire page or the first ECC block on the page. The first ECC block is either only metadata or metadata plus the first data block, depending on the BCH flash layout register settings.
- Be sure to set HW_GPMI_CTRL1[BCH_MODE].
- When using GPMI with BCH, the driver must wait on both the APBH DMA completion IRQ (HW_APBH_CTRL1[CH*n*_CMDCMPLT_IRQ]) as well as the BCH completion IRQ (HW_BCH_CTRL[COMPLETE_IRQ]).

Note the following suggestions for BCH flash layout arrangement:

- The number of metadata bytes should usually be set to 10. Very few NAND drivers or file systems require more metadata.
- Combining the metadata with the first data block in the flash layout is more efficient in terms of NAND page usage.
- The BCH supports four flash layout configurations, and each of the eight chip enables can be set to use any of the flash layouts. However, it is uncommon to use a combination of different NAND Flash devices is one system. Thus, it is often easiest to only fill in the HW_BCH_FLASH0LAYOUT*n* and clear HW_BCH_LAYOUTSELECT. Be aware that the reset values of HW_BCH_LAYOUTSELECT are such that each chip enable uses a different flash layout.

NOTE

Depending on the selected level of error correction, the number of parity bits per encoded data block are not always equal to an even number of bytes. In these cases, the BCH does not insert pad bits between the parity for one block and the beginning of the next block. This means that the beginning of data blocks other than the first may not be aligned to byte boundaries. Not inserting pad bits allows the use of higher error correction levels in some cases, at the expense of not being able to directly read data blocks other than the first (for instance, by setting the column address in the NAND page read command).

See the "BCH Encoding for NAND Writes" section of the chip reference manual for instructions for constructing a DMA descriptor chain to write a NAND page using BCH ECC.

See the "BCH Decoding for NAND Reads" section of the chip reference manual for instructions for building a DMA chain to read a NAND page encoded with BCH ECC.

13.9 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;
```

13.10 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;
```

13.11 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 14
Configuring the GPT Driver

14.1 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 the SABRE for Automotive Infotainment based on the i.MX 6 Series board schematics for pin assignments. For other board types, please refer to respective schematics.

There is one instance of GPT, which is located in the memory map at the GPT base address, 0209 8000h.

14.2 Feature 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

14.3 Modes of operation

The following table explains the GPT modes of operation:

Table 14-1. 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.

14.4 Events

14.4.1 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()`

14.4.2 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()
```

14.4.3 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()
```

14.5 Clocks

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

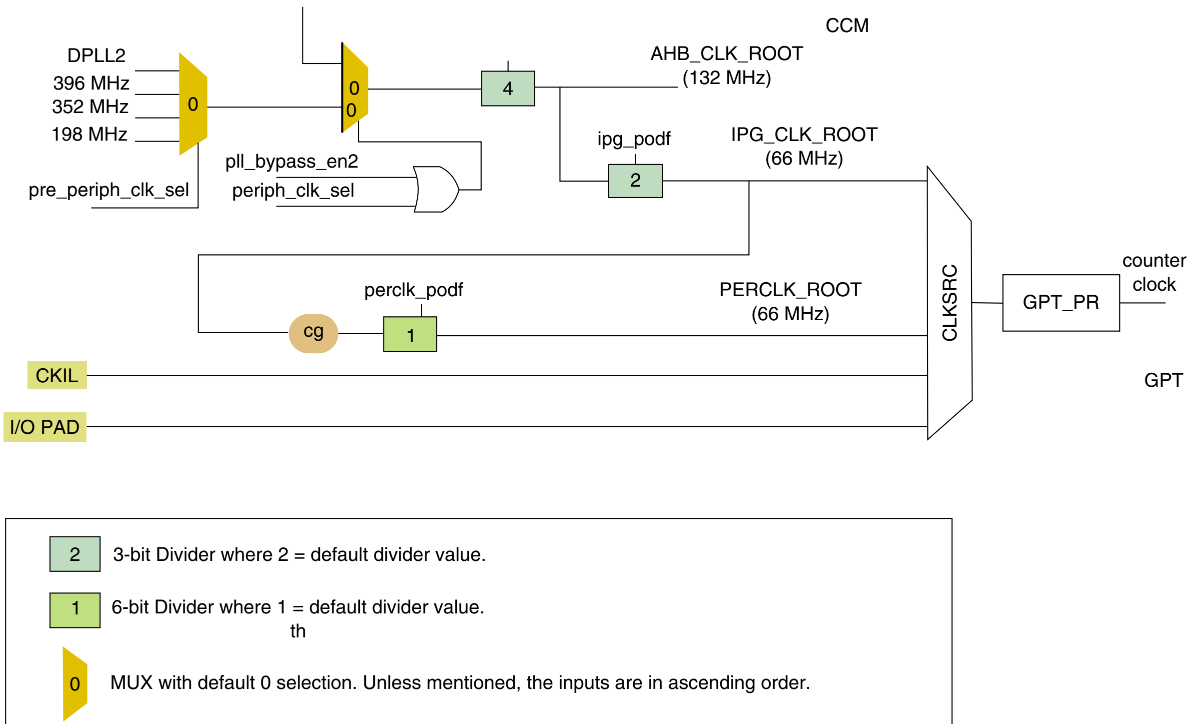


Figure 14-1. Reference clocks

The following table explains the GPT reference clocks:

Table 14-2. 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.

14.6 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 14-3. GPT IOMUX pin assignments

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	-

14.7 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/mx6dq/soc_memory_map.h`.

14.8 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 with the `hw_module` structure, which is defined in `./src/include/io.h`.

See the following example:

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

The following function's typically use this structure's address.

```
/*!
 * 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)
```

14.9 Testing the GPT driver

GPT can run the following tests:

- Output compare test
- Input capture test

14.9.1 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.

14.9.2 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.

Chapter 15
Configuring the HDMI Tx Module

15.1 Overview

This chapter explains how to configure and use the high-definition multimedia interface transmitter (HDMI Tx) module in the i.MX 6Dual/6Quad and i.MX 6Solo/6DualLite products. The HDMI Tx module is the source device for an HDMI system, which transmits uncompressed digital video data and uncompressed or compressed digital audio data. HDMI, the high definition multimedia interface, is a wired digital interconnect that replaces the analog TV out or VGA out. HDMI Tx consists of two parts:

- HDMI Tx controller
- HDMI Tx PHY

The following figure shows a basic HDMI system architecture.

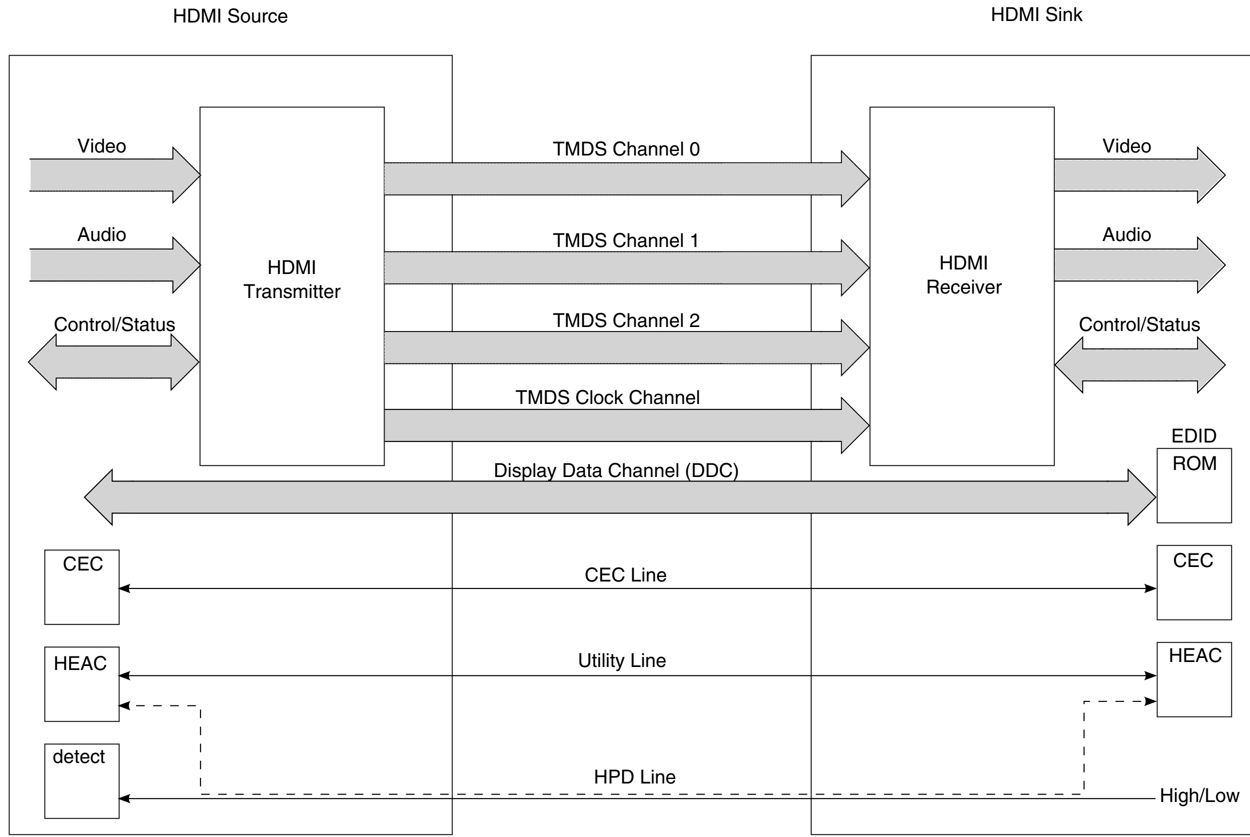


Figure 15-1. HDMI block diagram

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

- HDMI base address = 0012_0000h

15.2 Feature summary

Table 15-1. HDMI feature summary

Feature	Details
Standard Compliance	• HDMI 1.4a • HDMI CTS 1.4a • DVI 1.0
Video Standard Compliance	EIA/CEA-861D
Monitor Detection	Hot plug/unplug detection

Table continues on the next page...

Table 15-1. HDMI feature summary (continued)

Feature	Details
Supported Video Resolutions	• Up to 1080p at 120 Hz HDTV display • Up to QXGA graphics display • HDMI 1.4a 4K x 2K video formats • HDMI 1.4a 3D video modes with up to 340MHz TMDS clock
Pixel Clock Frequency	from 25 MHz to 340 MHz
Input Data Formats	• Parallel YCbCr 4:4:4 and parallel RGB 4:4:4 • Parallel YCbCr 4:2:2
Input Color Depth	24/30/36/48 bits/pixel
Input Syncs Format	Separate HSYNC and VSYNC plus data enable control
Internal Video Processing	Interpolation YCbCr 4:2:2 to 4:4:4, Color space conversion YCbCr to RGB and vice versa

15.3 Modes of operation

Table 15-2. HDMI modes of operation

Mode	What it does
HDMI 1.4	HDMI Tx includes the HDMI 1.4a specification features.
Color Space Converter	Parameter enables CSC support.
Consumer Electronics Control	Parameter enables CEC interface.
Internal Pixel Repetition	Parameter enables the internal pixel repetition support.

15.4 Events

HDMI_audio_done is mapped with SDMA souce 2, which is muxed with IPU-1 DMA Event and controlled by GPR0[0].

15.5 Clocks

The following table shows the clock frequency requirements for the HDMI Tx core.

Table 15-3. HDMI Tx clocks

Clock	Frequency	Description
ipixelclk	13.5 ~ 340 MHz	Pixel clock from the IPU display interface; becomes the input to the HDMI_TX module
isfrclk	18 ~ 27 MHz	Derived from video_27M_clk_root clock from CCM clock tree.

Table continues on the next page...

Table 15-3. HDMI Tx clocks (continued)

Clock	Frequency	Description
iahbclk	27 MHz	Derived from ahb_clk from the CCM clock tree
icecclk	32.768 kHz	Derived from external 32.768 kHz reference clock
ihclk	27 MHz	Derived from ahb_clk from the CCM clock tree

15.5.1 Video input interface clock

The following figure shows the video input interface clock signal.

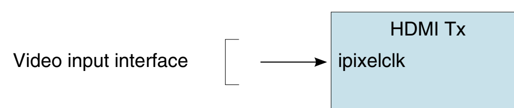


Figure 15-2. Video input interface clock signal

These signals require the input clock ipixelclk, which is the data pixel clock that is input from the IPU display interface clock (DI_CLKn). DI_CLKn is determined by the IPU and the DI port that the user chooses. The SDK example **hdmi_clock_set()** uses IPU1_DI0. See Display interface clocks (DI_CLK*n*) for detailed information about how to generate DI_CLKn.

15.5.2 System and slave register interface clocks

HDMI Tx supports the following interfaces:

- I2C
- SFR
- AMBA AHB slave
- OCP slave

The following figure shows which clocks are used by the slave interfaces. Table 15-4 provides a description of each clock

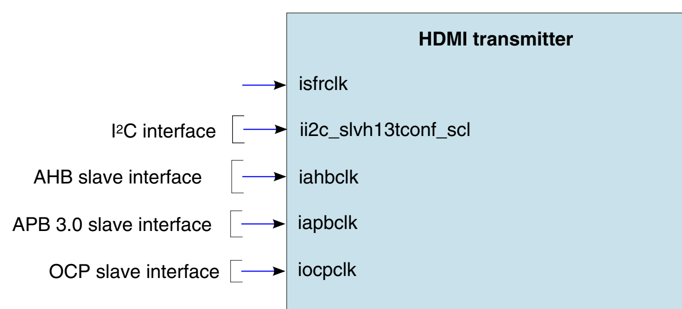


Figure 15-3. System and slave register interface clock signals

Table 15-4. System and slave register interface clock signal descriptions

Clock	Description
isfrclk	Internal register configuration clock
ii2c_slvh13tconf_scl	I2C slave clock for register configuration
iahbclk	AHB bus clock.
iapbclk	APB bus clock
iocpclk	OCP Slave Bus clock. This clock times all bus transfers. All signal timings are related to the rising edge.

15.5.3 EDID I^{2}C E-DDC interface clock

The E-DDC channel is a dedicated I^{2}C master interface that permits reading the E-EDID sink according to system needs.

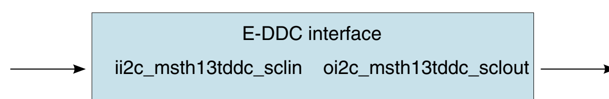


Figure 15-4. E-DDC interface clock signals

The E-DDC signals are defined as follows:

- **ii2c_msth13tddc_sclin** is the HDMI DDC I^{2}C slave clock input.
- **oi2c_msth13tddc_sclout** is the HDMI DDC I^{2}C slave clock input for E-EDID communication with the transmitter.

15.5.4 CEC interface clock

Consumer Electronics Control (CEC) is a protocol that provides high-level control functions between all of the various audiovisual products in a user's environment. It is an optional feature in the HDMI 1.3a specification. It uses only one bidirectional line for transmission and reception.

The following figure shows the interface signal of the CEC interface. icecclk (the CEC controller main clock input) should be a fixed frequency at 32.768 kHz.

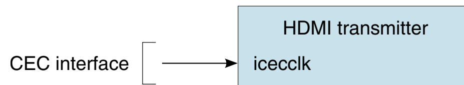


Figure 15-5. CEC interface clock signal

15.5.5 HDMI Tx PHY interface

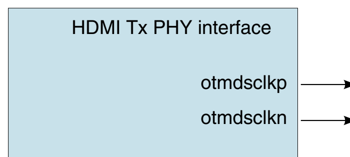


Figure 15-6. HDMI Tx PHY interface clock signal

otmdsclkp/otmdsclkn are the TMDS differential line driver clock output.

15.6 IOMUX pin mapping

Table 15-5. HDMI Tx IOMUX pin mapping

Signals	Driver		
	PAD	MUX	SION
CEC			
CEC_LINE (icecin)	EIM_A25	ALT6	0
DDC			
DDC_SCL (ii2c_msth13tddc_sclin)	EIM_EB2	ALT4	1
DDC_SDA (ii2c_msth13tddc_sdain)	EIM_D16	ALT4	1
HDMI 3D Tx PHY			
OPHYDTB[0] (ophytbd)	SD1_DAT1	ALT6	0
OPHYDTB[1] (ophytbd)	SD1_DAT0	ALT6	0

NOTE

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

15.7 Resets and interrupts

The chips GIC (global interrupt controller) has two ARM domain HDMI interrupts, which are described in the following table.

Table 15-6. ARM domain interrupts

IRQ	Description
147	HDMI master interrupt request
148	HDMI CEC engine dedicated interrupt signal raised by a wake-up event

15.8 Initializing the driver

This section explains how to initialize the driver according to the work flow, which is shown in the following figure.

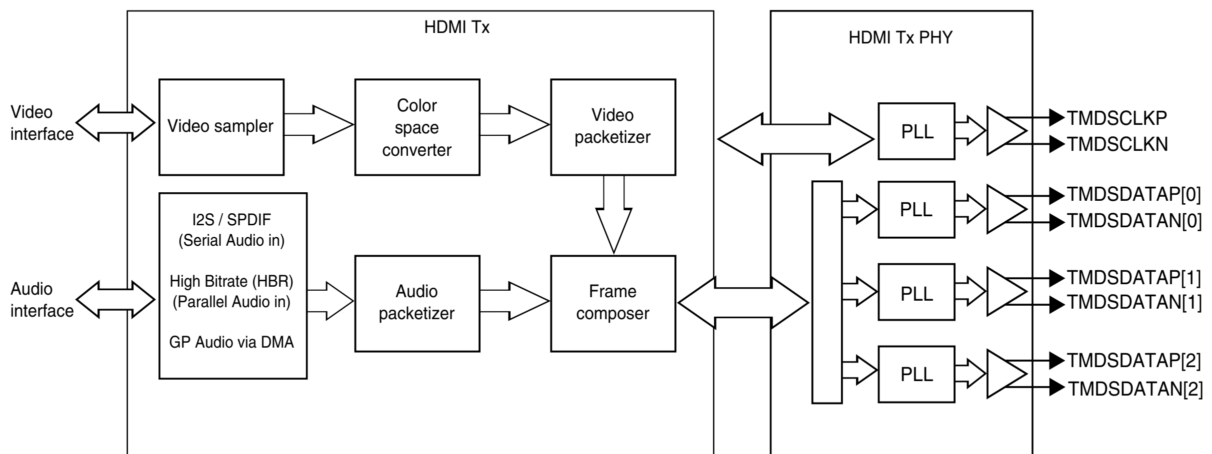


Figure 15-7. HDMI driver initialization

15.8.1 Setting up the video input

The video input source to the HDMI Tx module can be any output stream from the IPU module.

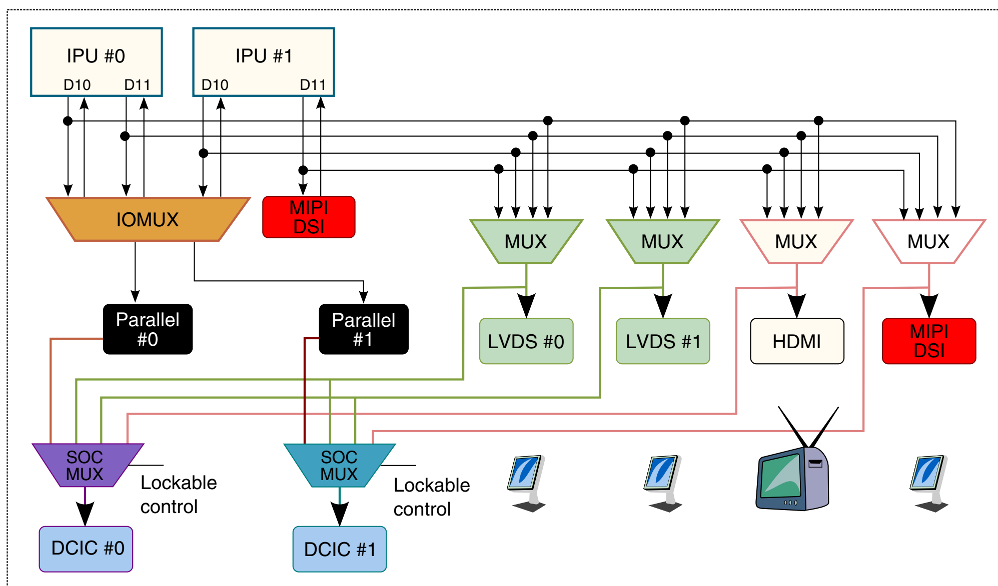


Figure 15-8. Available mux configurations

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 display interface 0 (IPU1-DI0)
- 01b - IPU1 display interface 1 (IPU1-DI1)
- 10b - IPU2 display interface 0 (IPU2-DI0)
- 11b - IPU2 display interface 1 (IPU2-DI1)

15.8.2 Setting up the video sampler

The video pixel sampler is responsible for the video data synchronization, according to the video data input mapping defined by the Color Depth (Deep Color) and format configuration. The following table describes the input video mapping.

Table 15-7. Input video mapping code

Color space	Color depth	Video mapping (hex)
RGB 4:4:4	8-bit	01
	10-bit	03
	12-bit	05
	16-bit	07

Table continues on the next page...

Table 15-7. Input video mapping code (continued)

Color space	Color depth	Video mapping (hex)
YCbCr 4:4:4	8-bit	09
	10-bit	0B
	12-bit	0D
	16-bit	0F
YCbCr 4:2:2	8-bit	16
	10-bit	14
	12-bit	12

15.8.3 Setting up the CSC (color space converter)

CSC is responsible for carrying out the following video color space conversion functions:

- RGB to/from YCbCr
- 4:2:2 to/from 4:4:4 up (pixel repetition or linear interpolation) / down-converter
- Limited to/from full quantization range conversion

The following figure shows the CSC block diagram.

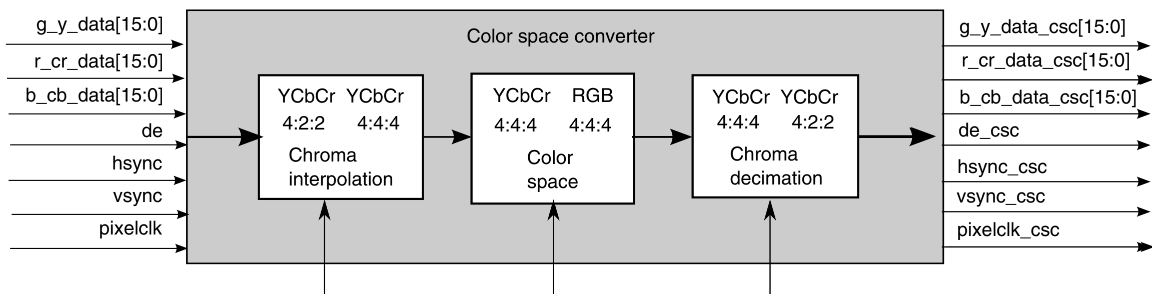


Figure 15-9. CSC block diagram

The CSC conversion function is:

$$\begin{bmatrix} out1 \\ out2 \\ out3 \end{bmatrix} = 2^{scale-12} * \begin{bmatrix} A_1 A_2 A_3 \\ B_1 B_2 B_3 \\ C_1 C_2 C_3 \end{bmatrix} \begin{bmatrix} in_1 \\ in_2 \\ in_3 \end{bmatrix} + 2^{scale-12} * \begin{bmatrix} A_4 \\ B_4 \\ C_4 \end{bmatrix}$$

15.8.4 Setting up the video packetizer

The following figure shows the video packetizer functional diagram.

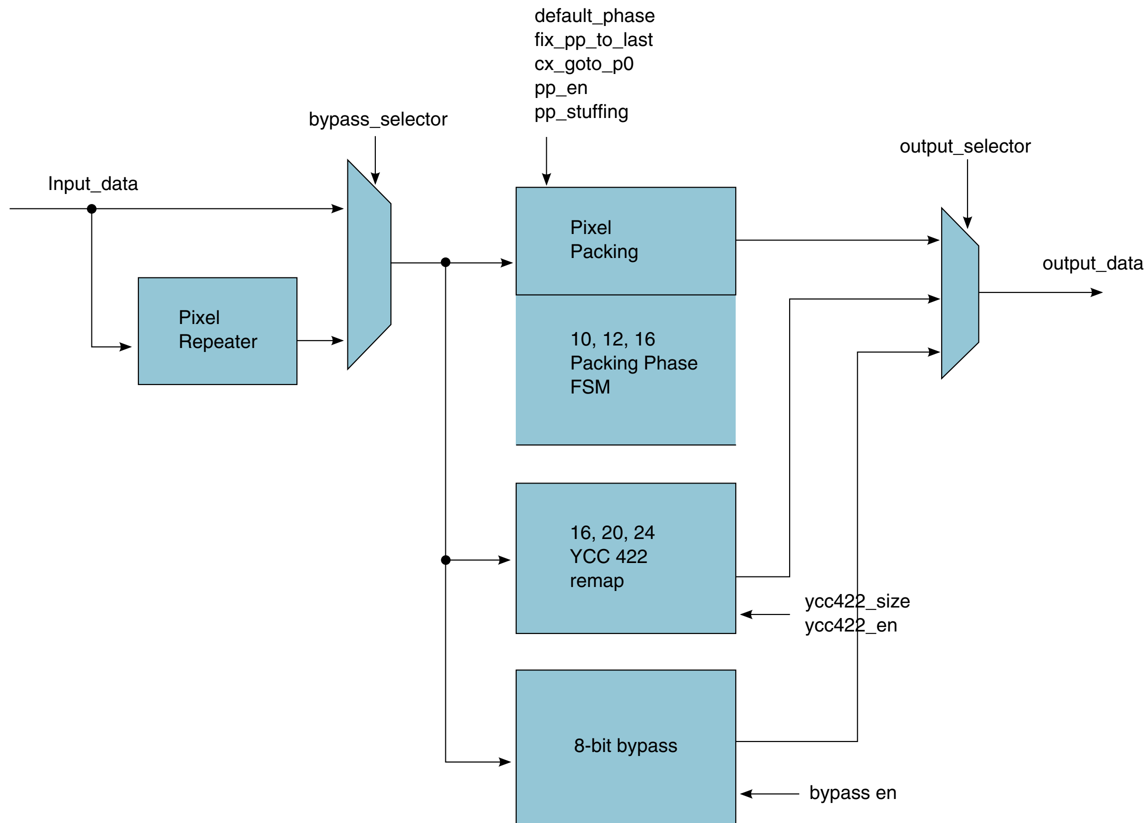


Figure 15-10. Video packetizer functional diagram

To set up the video packetizer:

1. Configure input data path -- pixel repeater or bypass selector
2. Configure pixel repetition, pixel packing, YCC422 stuffing, remap
3. Configure output selector

15.8.5 Setting up the frame composer

The frame composer is responsible for assembling video, audio, and data packets into a consistent frame that is streamed to the HDMI TX PHY.

To set up the video component:

1. Configure video synchronism for video signal: interlaced/progressive, vblank variation and polarity.

2. Configure video timing: hactive/vactive, hblank/vblank, hsyncoffset/vsyncoffse, hsyncwidth/vsyncwidth.
3. Configure pixel repetition ratio factor of the input and output video signal.

15.8.6 Setting up HDMI Tx PHY

The following HDMI PHY types can be configured by HDMI_PHY_SEL:

- 0 - chrt_065lp
- 1 - ibm_065lp
- 2 - tsmc_065gp
- 3 - tsmc_065lp
- 4 - phy_gen2

NOTE

Only phy_gen2 is supported in the i.MX 6 Series HDMI Tx PHY module.

When the phy_gen2 option is selected, a new I2C master interface is added to the HDMI TX. This then enables the I2C Master PHY registers to be programmed.

The process for setup HDMI Tx PHY is as follows

1. Setup the physical interfaces: power down, data enable polarity and interface control of the HDMI Source PHY control.
2. Configure Tx PHY type, behavior model
3. Scan test interface signals.
4. Software reset physcial and TMDS drivers

15.9 Testing the driver

The HDMI Tx unit test demonstrates how to output both audio and video from 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 and outputs it from the HDMI signals along with the IPU output.

Chapter 16
Configuring the I2C Controller as a Master Device

16.1 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 an engineering sample board's schematics as its reference for pin assignments. For other board types, refer to the appropriate schematics.

16.2 Initializing the I2C controller

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

16.2.1 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 I^2C blocks use:

Table 16-1. I^2C 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 chip reference manual.

16.2.2 Clocks

The following figure shows the clock control signals.

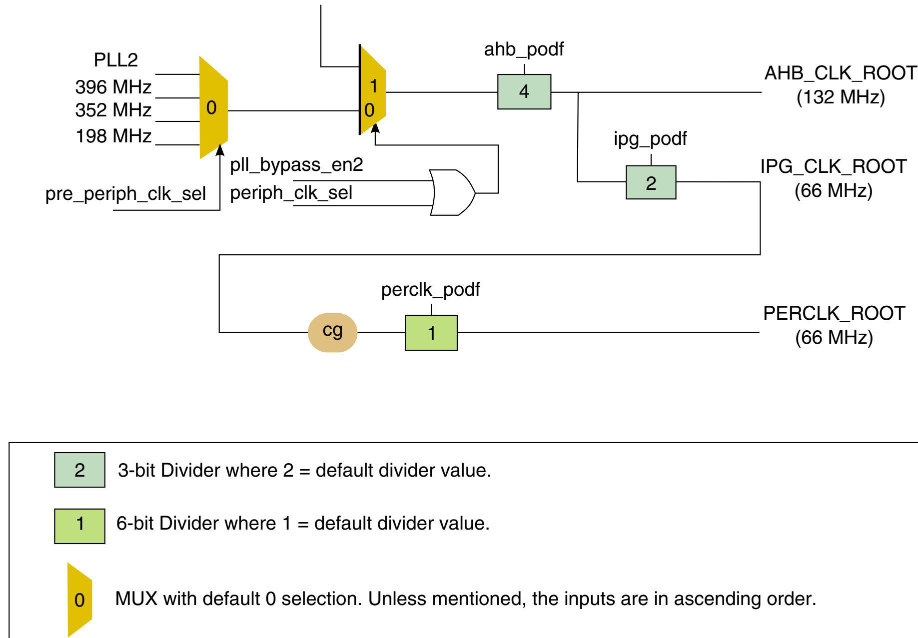


Figure 16-1. Clock control signals for I2C blocks

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 chip reference manual for more information about programming clocks.

16.2.3 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 16-2. 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
0Eh	192	1Eh	3072	2Eh	112	3Eh	1792
0Fh	240	1Fh	3840	2Fh	128	3Fh	2048

16.3 I2C protocol

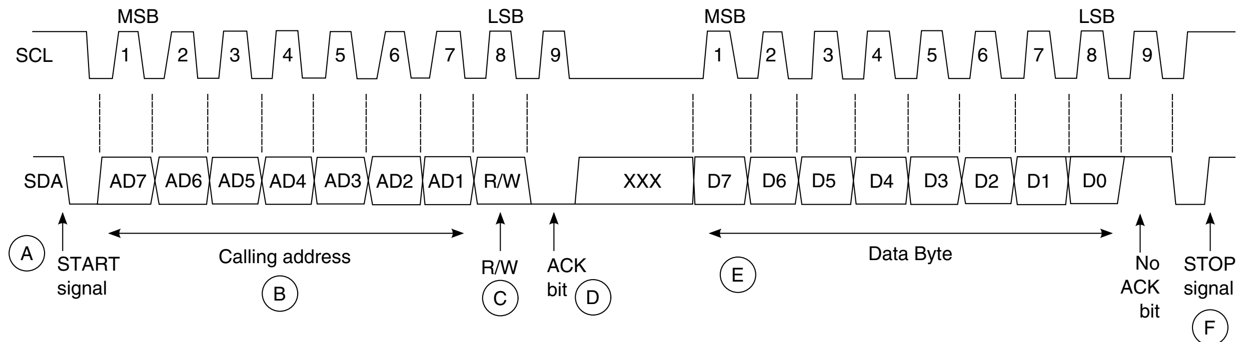


Figure 16-2. I2C standard communication protocol

The I2C communication protocol consists of the following six components:

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

16.3.1 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.

16.3.2 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.

16.3.3 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.

16.3.4 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.

16.3.5 Repeat start

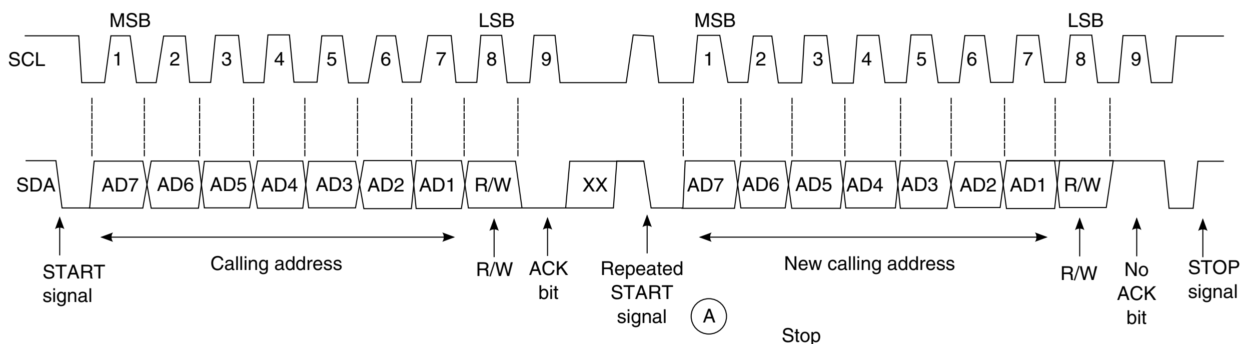


Figure 16-3. Repeated START

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.

16.4 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.

16.4.1 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)
```

16.4.2 Programming the I2C controller for I2C Read

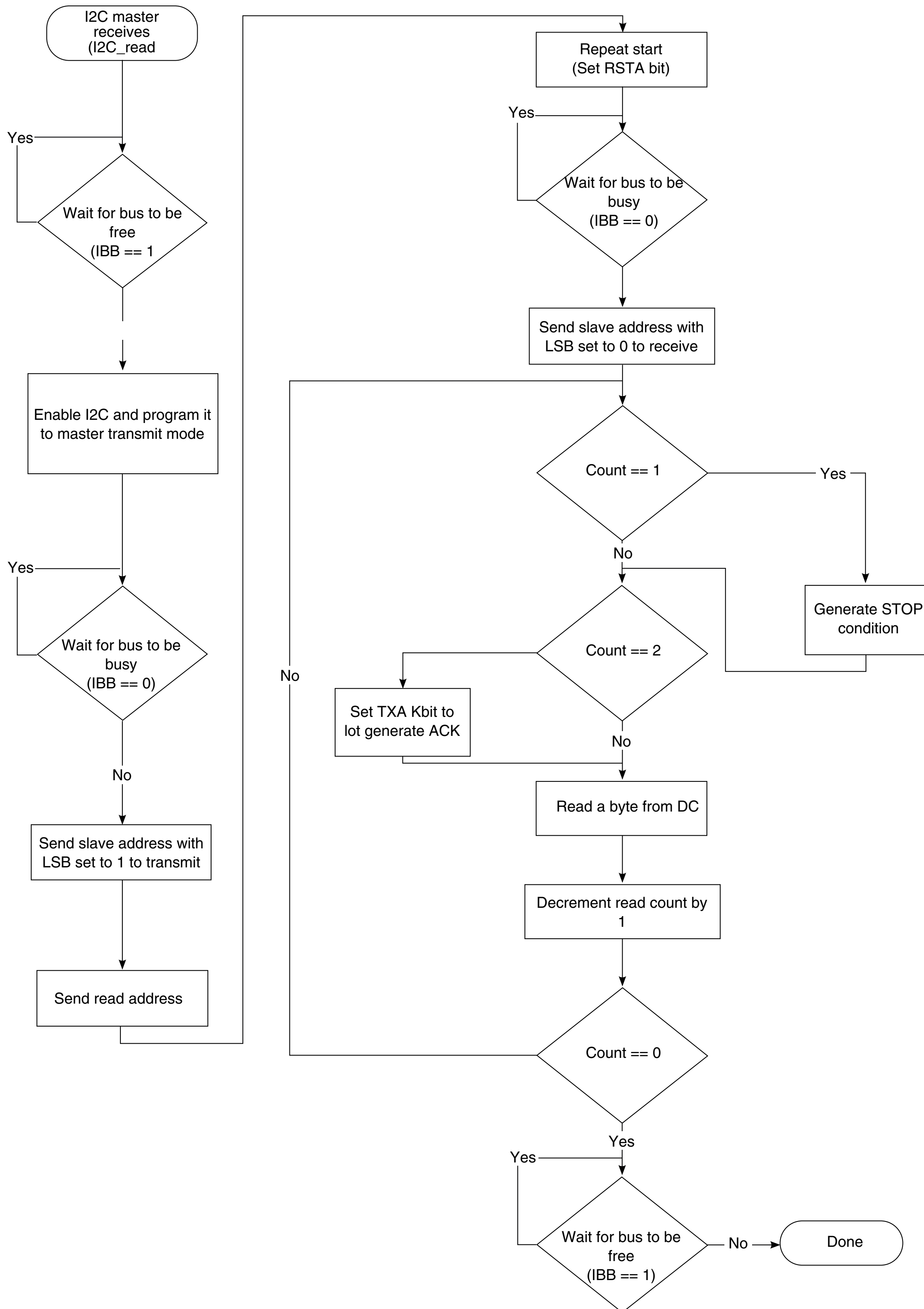


Figure 16-4. Flow chart for I2C read

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

16.4.3 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)
```

16.4.4 Programming the I2C controller for I2C Write

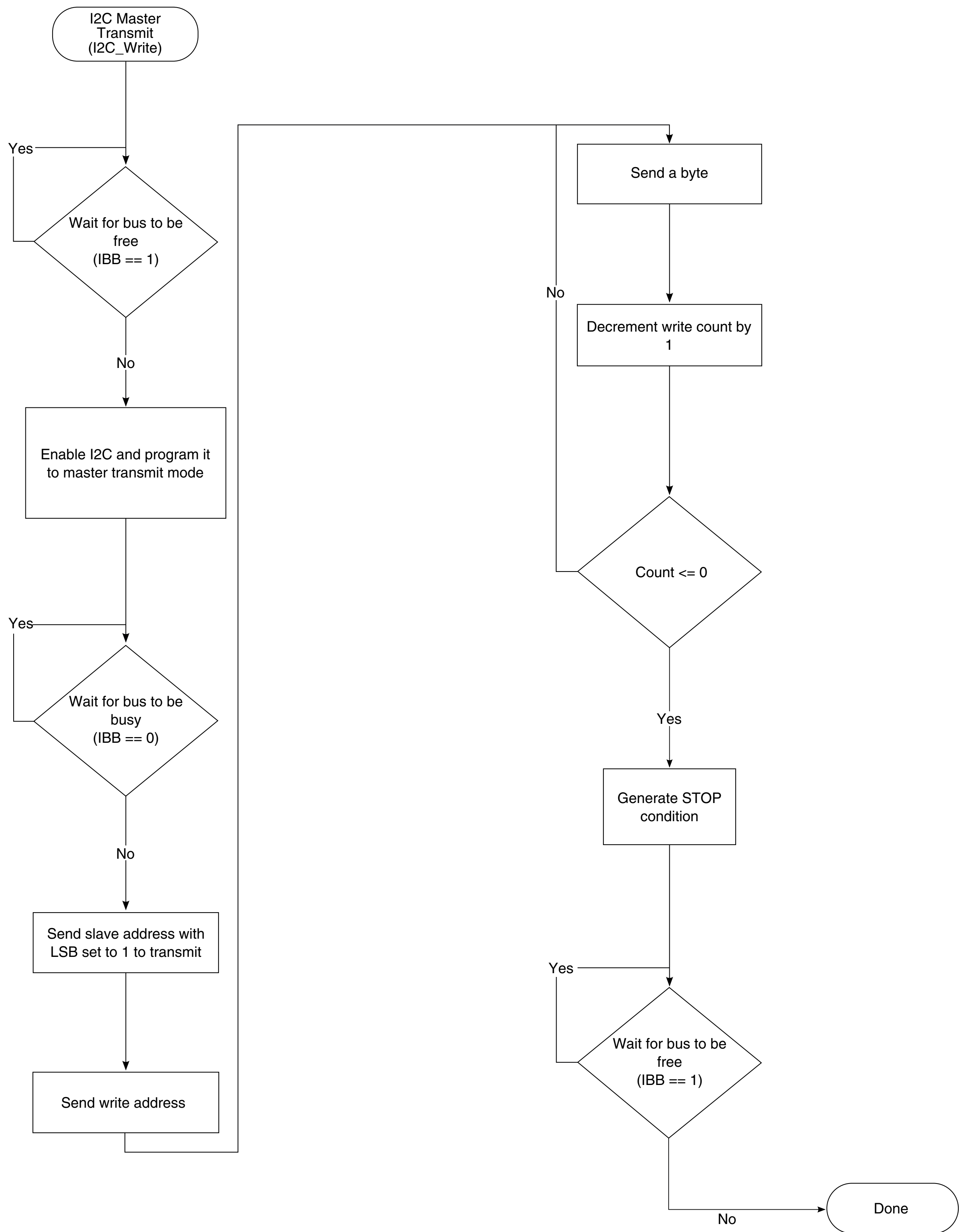


Figure 16-5. Flow chart for I2C write

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

16.4.5 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 17
Configuring the I2C Controller as a Slave Device

17.1 Overview

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

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

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

17.2 Feature summary

This low-level driver supports:

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

17.3 Modes of operation

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

Table 17-1. I^2C 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.

17.4 Clocks

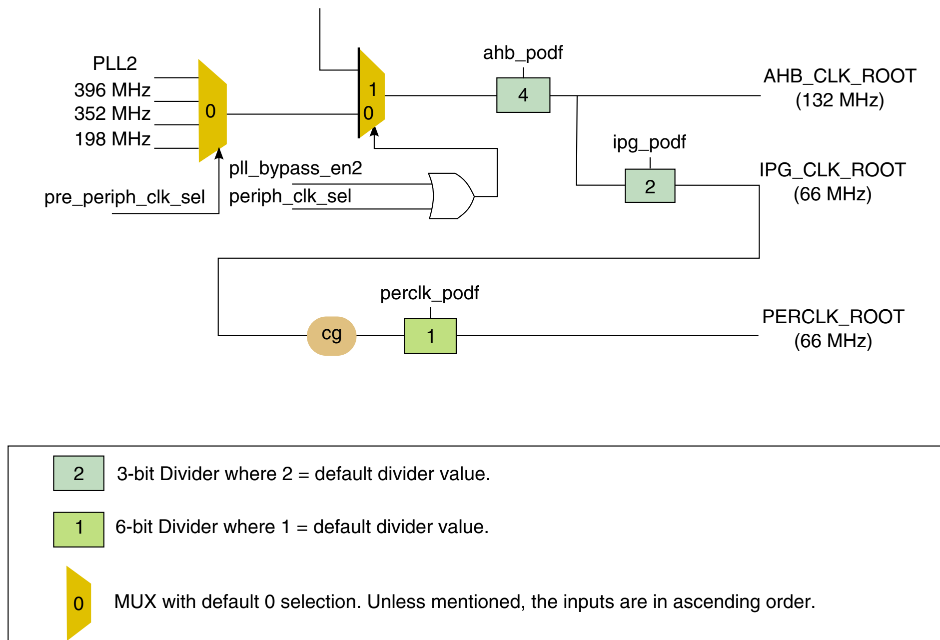


Figure 17-1. Clock control signals for I2C blocks

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 I²C chapter in the chip reference manual)

Table 17-2. I²C 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.

17.5 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/mx6dq/soc_memory_map.h`.

17.6 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^2C 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^2C 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)
```

17.7 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.

17.7.1 Running the test

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

```
./tools/build_sdk -target mx6dq -board sabre_ai -board_rev a -test i2c
```

This generates the following ELF and binary files:

- `./output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-i2c-sdk.elf`
- `./output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-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.

Chapter 18
Configuring the IPU Driver

18.1 Overview

The IPU is a part of the video and graphics subsystem in the i.MX 6Dual/6Quad and i.MX 6Solo/6DualLite application processors. 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, which are 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:

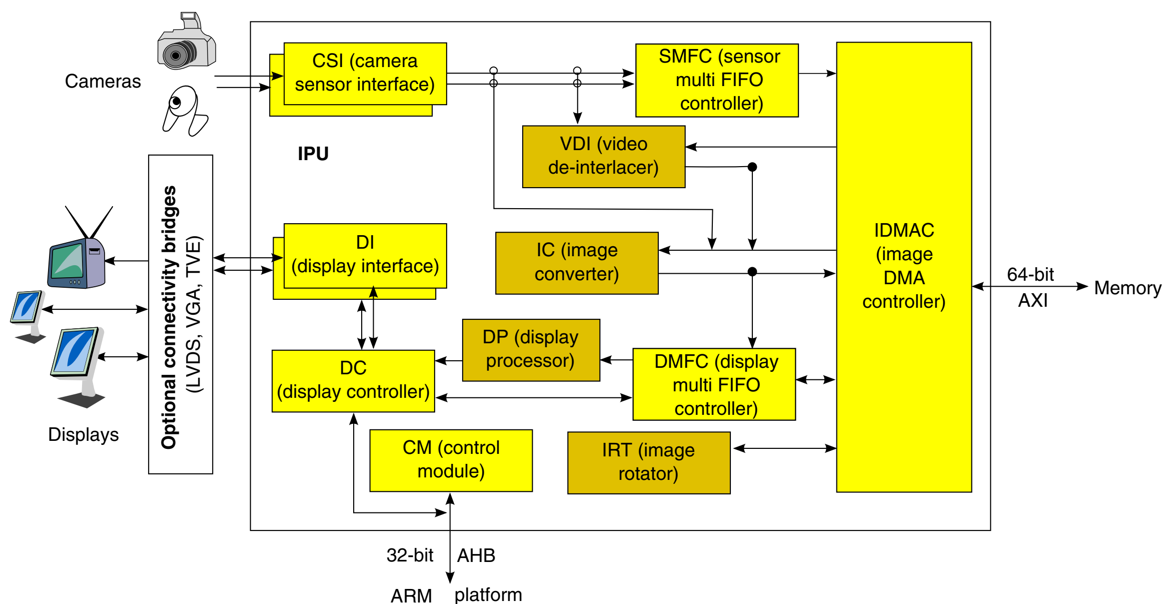


Figure 18-1. IPU block diagram

The following table describes the role of each block.

Table 18-1. 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
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

Table continues on the next page...

Table 18-1. IPU block descriptions (continued)

Block	Description
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.

18.2 IPU task management

The detailed IPU diagram is shown in the following figure:

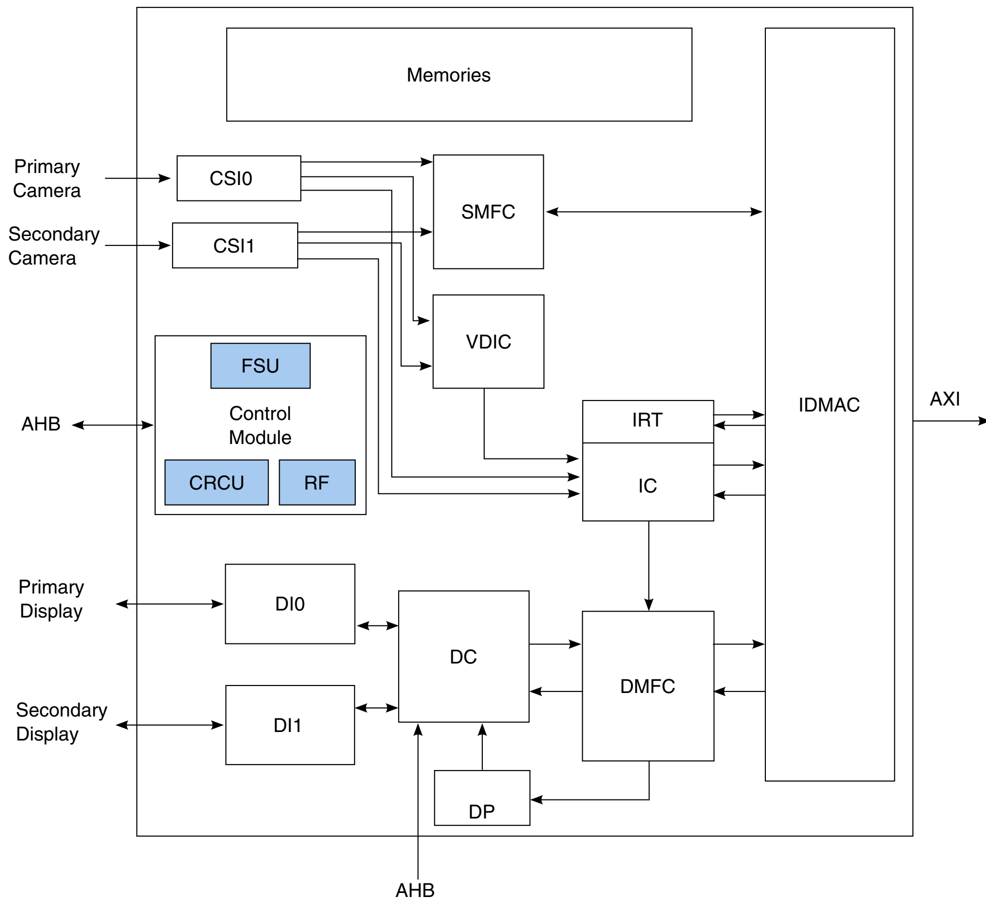


Figure 18-2. Detailed IPU block diagram

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

Table 18-2. 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.
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.	-

18.3 Image rendering

Image rendering is the process by which the image data that is 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 (the chip provides the HDMI/DVI transmitter, and the PHY converts the data into serialized differentiated data lanes)
  - LVDS panels (the chip provides LDB as a bridge to LVDS display)
  - MIPI DSI

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

18.3.1 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.

18.3.2 DMFC

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

Table 18-3. 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.

18.3.3 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:

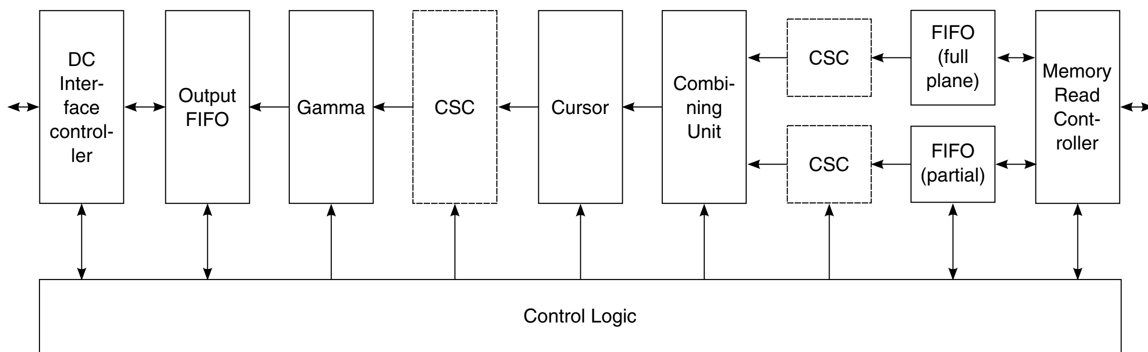


Figure 18-3. Display processor architecture diagram

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.

18.3.4 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.

18.3.5 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.

18.4 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.

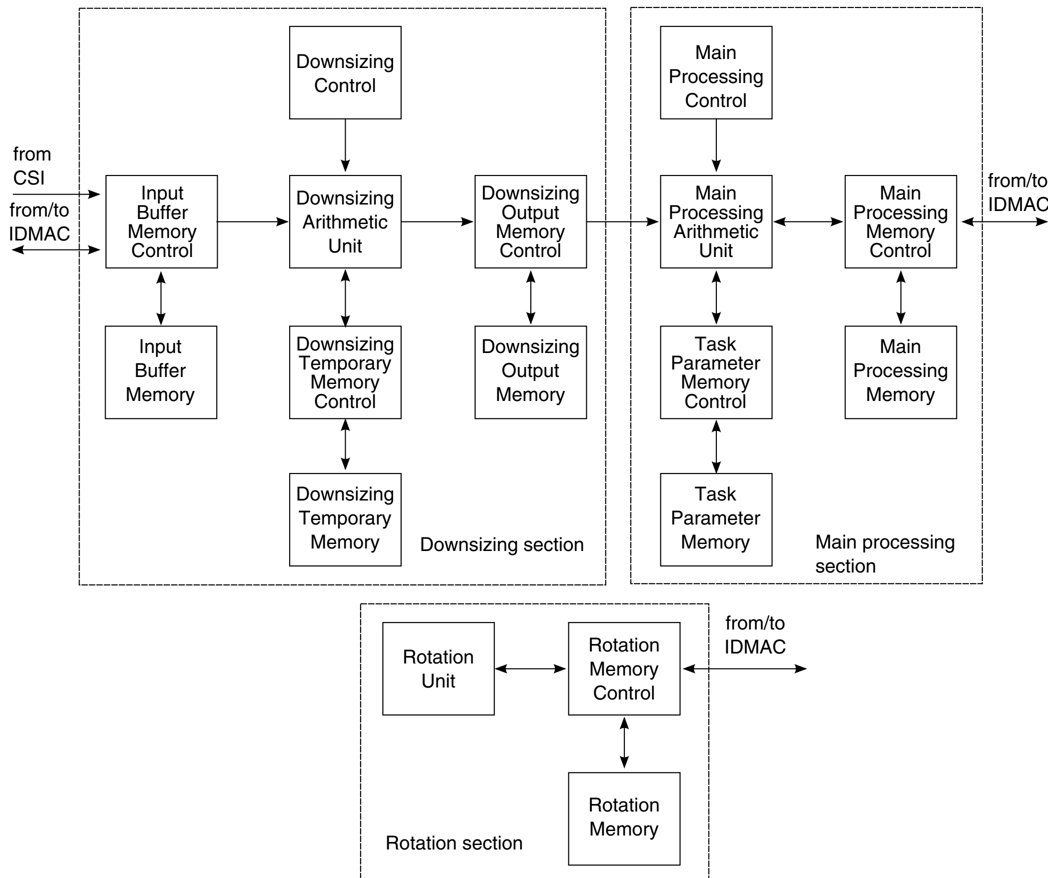


Figure 18-4. Image converter diagram

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.

18.4.1 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.

18.4.2 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.

18.4.3 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.

18.5 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.

18.5.1 CSI interfaces

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

18.5.1.1 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.

18.5.1.2 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.

18.5.2 CSI modes

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

18.5.2.1 Gated mode

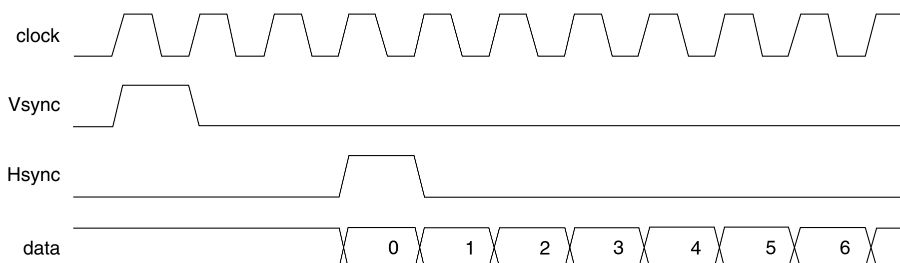


Figure 18-5. CSI gated mode

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.

18.5.2.2 Non-gated mode

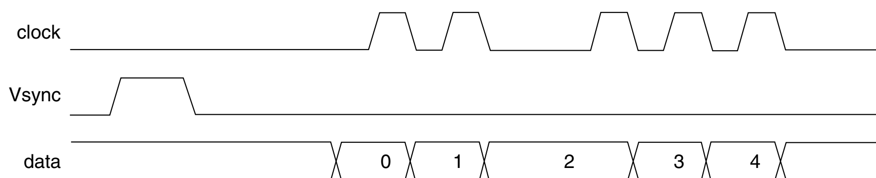


Figure 18-6. CSI non-gated mode

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.

18.5.2.3 BT656 mode

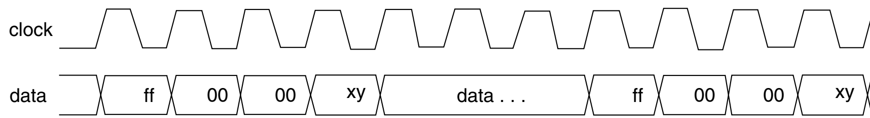


Figure 18-7. BT656 mode

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.

18.5.2.4 BT1120 mode

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

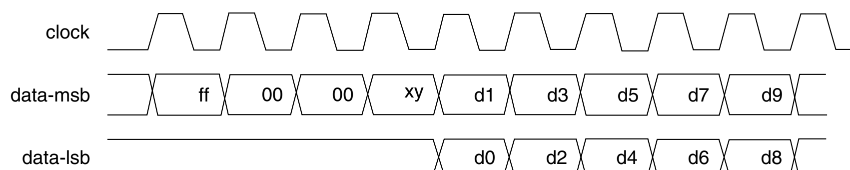


Figure 18-8. BT1120 SDR mode

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.

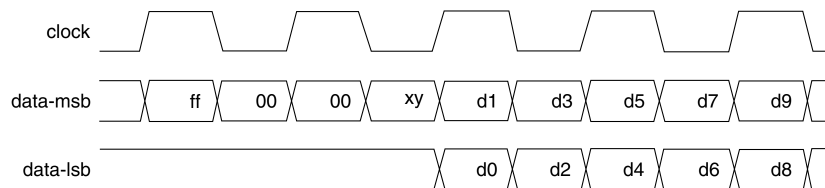


Figure 18-9. BT1120 DDR Mode

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

18.6 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.

18.7 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.

18.8 Clocks

The following table lists the IPU clock sources:

Table 18-4. 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.

18.8.1 High-speed processing clock (HSP_CLK)

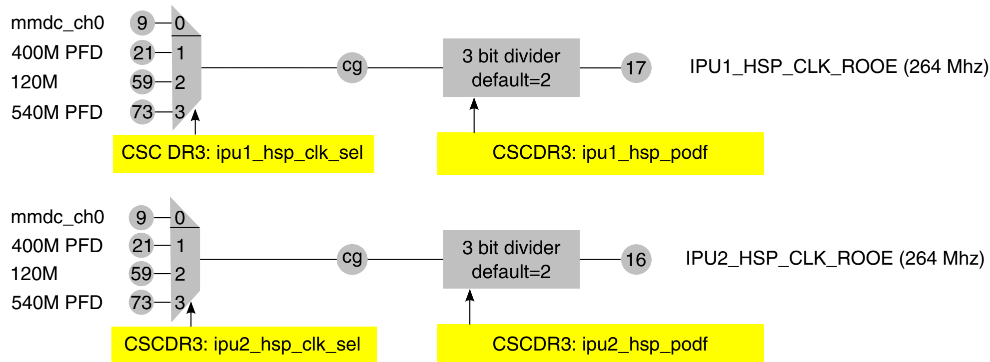


Figure 18-10. IPU HSP_CLK clock tree

The IPU main clock (HSP_CLK) is generated by the internal clock control module (CCM) (see Figure 18-10). The default HSP_CLK is divided from mmdc_ch0 by 2.

18.8.2 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.

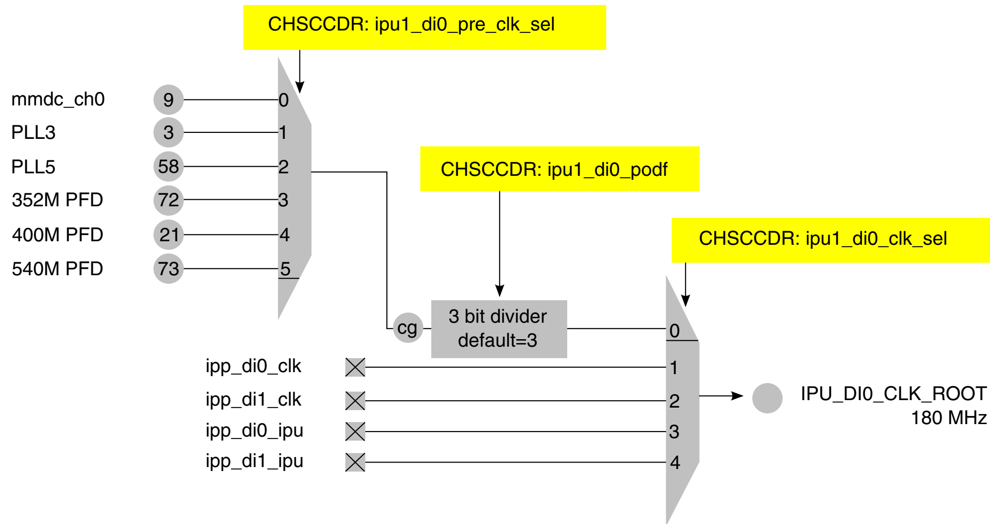


Figure 18-11. IPU DI0 clock tree

18.9 IOMUX pin mapping

IPU 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 18-5. 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

18.10 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.

18.10.1 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.

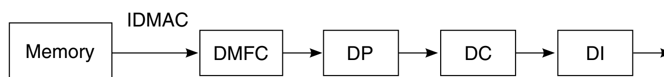


Figure 18-12. IPU process for single image rendering

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:

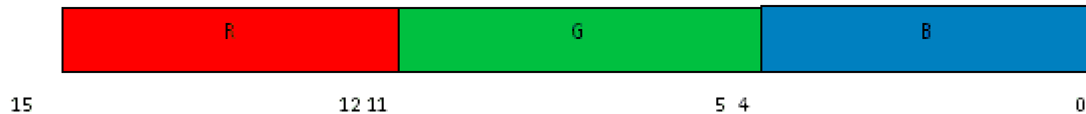
1. Configuring the IPU DMA channel (single image rendering)
2. Allocating the DMFC block
3. Configuring the DP block
4. Configuring the DC block
5. Configuring the DI block
6. Enabling the blocks involved in the display flow

18.10.1.1 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.

18.10.1.2 Allocating the DMFC block

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

18.10.1.3 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.

18.10.1.4 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.

18.10.1.5 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 18-11.

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

Table 18-6. 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 18-13. 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.

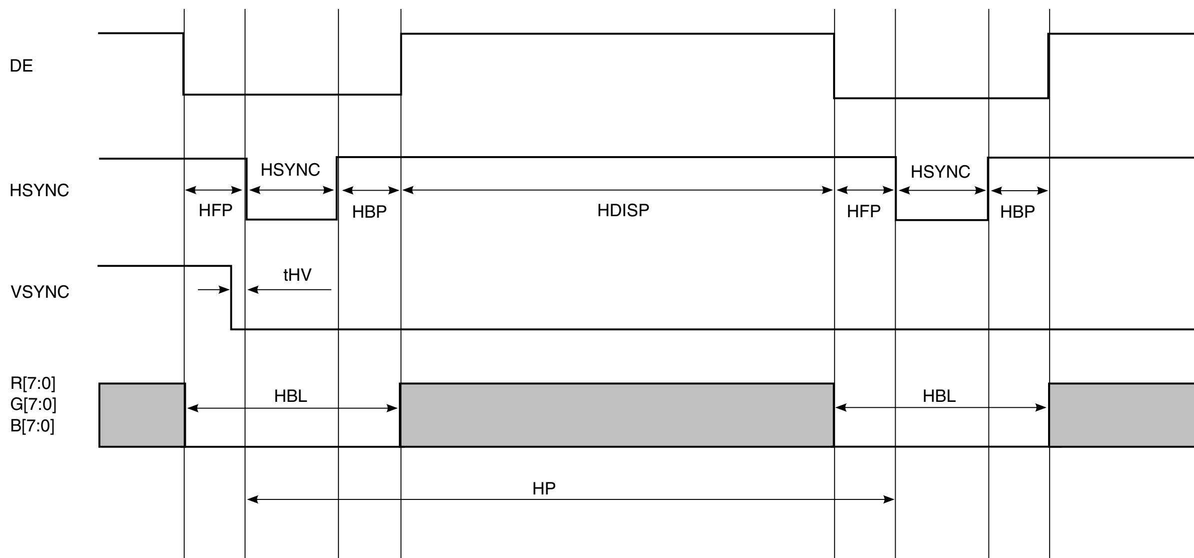


Figure 18-14. Horizontal display timing

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().

18.10.1.6 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.

18.10.2 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.

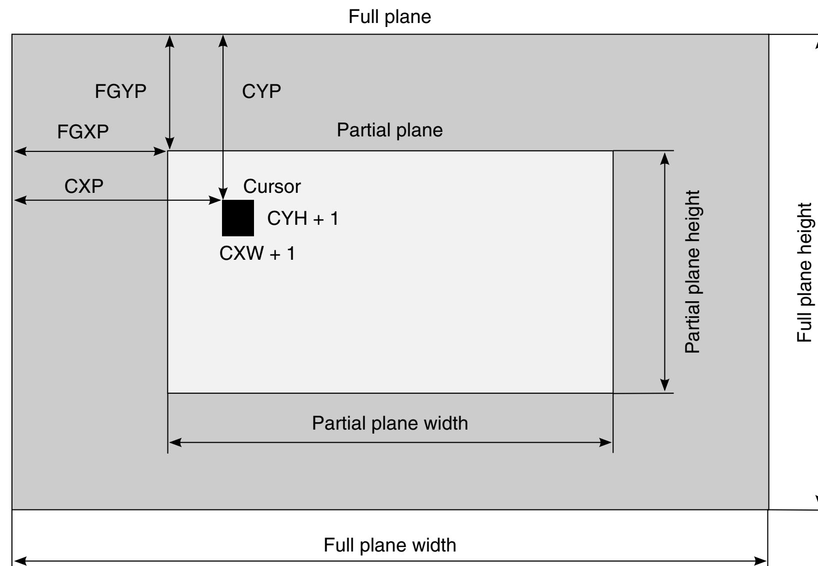


Figure 18-15. Display planes

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.

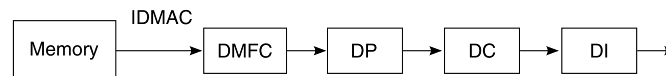


Figure 18-16. IPU process for image combining

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 18-12), IPU hardware configuration is different in the IDMAC, DMFC and DP modules.

18.10.2.1 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 18-7. 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.

18.10.2.2 Allocating the DMFC

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

18.10.2.3 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).

18.10.2.4 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.

18.10.3 Image rotate example

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

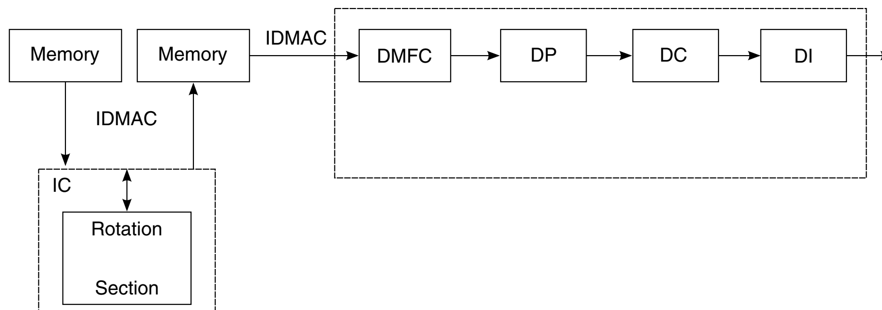


Figure 18-17. IPU process for image rotation

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)

18.10.3.1 Configuring IDMAC channels for IC tasks (IC rotate)

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

Table 18-8. 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.

18.10.3.2 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.

18.10.3.3 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.

18.10.3.4 Image rendering process (IDMAC)

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

18.10.4 Image resizing example

18.10.4.1 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.

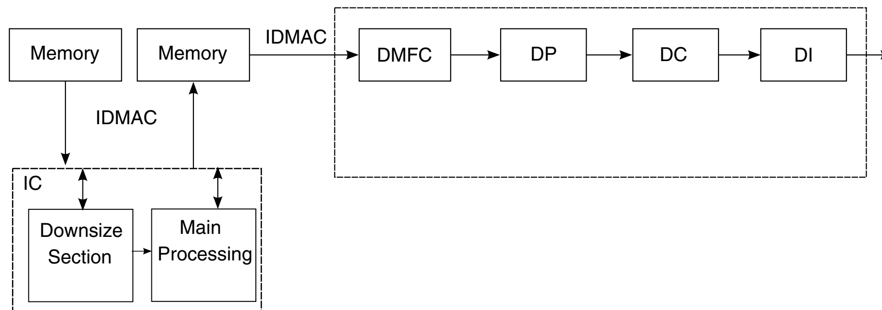


Figure 18-18. IPU process for image rotation

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

18.10.4.2 Configuring IDMAC channels for IC resize tasks

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

Table 18-9. 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
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 for input channel #11and output channel #22 is NPB (number of pixels per burst access), which is 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.

18.10.4.3 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 18-10. 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.
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.

Table continues on the next page...

Table 18-10. Resize task settings (continued)

Setting	What it does
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: $RS_C_H = \left(\sum_{k=0}^{R-1} RS_R_H\right) mod(8196)$
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 $RS_C_V = \left(\sum_{k=0}^{C-1} RS_R_V\right) mod(8196)$
PP_EN	Enables the postproduction task

18.10.4.4 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.

18.10.4.5 Image rendering process

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

18.10.5 Color space conversion example

18.10.5.1 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.

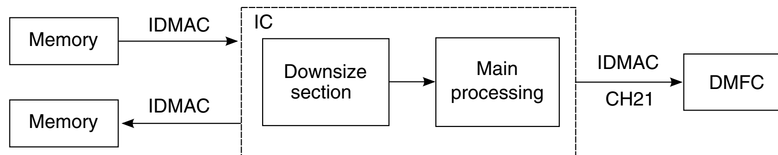


Figure 18-19. IPU process for color space conversion (IC module)

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.

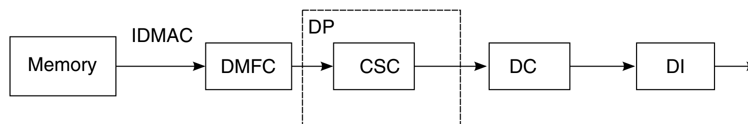


Figure 18-20. IPU process for CSC (DP module)

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

18.10.5.2 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.

18.10.5.3 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:

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

- 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 following table shows the resizing related bit fields' setting for the postprocessing CSC1 task lists. For all parameters, use the following:

- Address at 6060h
- word 0 and word 1

Table 18-11. Postprocessing task IC parameters for color space conversion

Parameter	Field	Description
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	26:13	
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 viewfinder 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 viewfinder task: • 0 --> (min, max) = (0, 255) • 1 --> (min, max) = (16, 240) for Z1,Z2 • 1 --> (min, max) = (16, 235) for Z0

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.

18.10.5.4 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:

$x \rightarrow$ Clip(Round(S×2^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 = [CSC_B0 CSC_B1 CSC_B2]

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

  E = [CSC_S0 CSC_S1 CSC_S2]

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 19
Configuring the Keypad Controller

19.1 Overview

This chapter explains how to configure the keypad controller to manage a key matrix of up to 8 x 8 keys.

This chip has one instance of the keypad, which is located in the memory map at 020B 8000h.

19.2 Feature summary

This low-level driver supports:

- Single and multiple key press on interrupt from up to an 8 x 8 matrix.
- Release detection of all keys.

19.3 Modes of operation

The following table explains the keypad driver modes of operation:

Table 19-1. Keypad modes of operation

Mode	What it does
Multiple key press	A scanning routine returns the detected pressed key(s) when an interrupt is triggered by a press event.
Release detection	When a key or multiple keys were pressed, an interrupt is generated when all keys are released.

19.4 Clocks

This module has no clock to configure.

19.5 IOMUX pin mapping

The following table shows the IOMUX pin map for the keypad controller. Shading indicates the version that was tested on an engineering sample board.

Table 19-2. IOMUX pin map

Signals	Driver			
	PAD	MUX	SION	DAISY CHAIN
KEY_COL0	KEY_COL0	ALT3	1	N/A
KEY_COL1	KEY_COL1	ALT3	1	N/A
KEY_COL2	KEY_COL2	ALT3	1	N/A
KEY_COL3	KEY_COL3	ALT3	1	N/A
KEY_COL4	KEY_COL4	ALT3	1	N/A
KEY_COL5	GPIO_0	ALT2	1	0
	GPIO_19	ALT0		1
	CSI0_DAT4	ALT3		2
	SD2_CLK	ALT2		3
KEY_COL6	GPIO_9	ALT2	1	0
	CSI0_DAT6	ALT3		1
	SD2_DAT3	ALT2		2
KEY_COL7	SD2_DAT1	ALT4	1	0
	GPIO_4	ALT2		1
	CSI0_DAT8	ALT3		2
KEY_ROW0	KEY_ROW0	ALT3	1	N/A
KEY_ROW1	KEY_ROW1	ALT3	1	N/A
KEY_ROW2	KEY_ROW2	ALT3	1	N/A
KEY_ROW3	KEY_ROW3	ALT3	1	N/A
KEY_ROW4	KEY_ROW4	ALT3	1	N/A
KEY_ROW5	GPIO_1	ALT2	1	0
	CSI0_DAT5	ALT3		1
	SD2_CMD	ALT3		2

Table continues on the next page...

Table 19-2. IOMUX pin map (continued)

Signals	Driver			
	PAD	MUX	SION	DAISY CHAIN
KEY_ROW6	SD2_DAT2	ALT4	1	0
	GPIO_2	ALT2		1
	CSI0_DAT7	ALT3		2
KEY_ROW7	SD2_DAT0	ALT4	1	0
	GPIO_5	ALT2		1
	CSI0_DAT9	ALT3		2

19.6 Resets and interrupts

This module resets along with the chip on both warm and cold resets. This is reset with the chip’s global reset signal and is not a software controllable reset.

All the interrupt sources are listed in the reference manual in the "Interrupts and DMA Events" chapter. The SDK provides this list at `./src/include/mx6dq/soc_memory_map.h`.

The interrupt source for the keypad is: IMX_INT_KPP.

The driver provides an interrupt routine (kpp_interrupt_routine). This routine disables the key press and release interrupts as well as clears a flag used as wait for interrupt into some of driver's functions.

19.7 Initializing the driver

The application initializes the keypad controller by calling the kpp_open function (see below), which is available in the keypad port driver at ./src/sdk/keypad/drv/keypad_port.c.

```
/*!
 * Initialize the keypad controller.
 *
 * @param   kpp_col - active columns in the keypad.
 * @param   kpp_row - active rows in the keypad.
 */
void kpp_open(uint8_t kpp_col, uint8_t kpp_row)
The value for kpp_col and kpp_row are those used to fill the KPP_KPCR register.
In the test example, the keypad matrix only uses the rows and columns from 5 to 7. Set these
variables to:
kpp_col = kpp_row = 0xE0.
This sets up a 3 x 3 matrix.
```

19.7.1 Closing the keypad port

```
The application calls the following function to close the keypad port properly.
/*!
 * Leave the keypad controller in a known state.
 *
 */
void kpp_close(void)
```

19.7.2 Waiting for or obtaining a key press event

```
When the application needs to wait for or obtain a key press event, it calls the following
function:
/*!
 * Keypad port function to return the read key.
 *
 * @param   rd_keys - active columns in the keypad.
 * @param   condition - keypad state is read immediately (IMMEDIATE)
 *                      or it waits for key pressed interrupt (WF_INTERRUPT).
 */
void kpp_get_keypad_state(uint16_t *rd_keys, uint8_t condition)
```

The scan routine can be processed immediately to return the current keypad state, or it can be executed once a key press event has been detected. The columns configured in output mode are consecutively placed into a known state, and the state of the rows configured in input mode is logged each time.

rd_keys is a pointer to an array of 8 unsigned short elements. This array can be parsed like in the example test and compared to a key map to determine the name of the pressed key(s). It logs the sampled status of the 8 rows, and the value of the register KPP_KPDR provides the coordinates of the pressed key(s).

See the keypad port chapter of the chip reference manual for a detailed description of the scanning routine.

19.7.3 Waiting for all keys to release

The application uses the following function once at least one key has been pressed to wait for all keys to release. Note that the keypad controller cannot detect that a particular key has been released; it can only return the information that none of the keys are pressed.

```
/*!
 * Keypad port function that waits for all keys to release.
 * The hardware can only detect this condition, and couldn't
 * detect the release of a single key but by doing it
 * by software.
 */
void kpp_wait_for_release_state(void)
```

19.8 Testing the driver

A test is available that uses the keypad driver to retrieve the names of the pressed keys.

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

```
./tools/build_sdk -target mx6dq -board evb -board_rev a -test keypad
```

This generates the following binary and ELF files:

- `./output/mx6dq/evb_rev_a/bin/mx6dq_evb_rev_a-keypad-sdk.elf`
- `./output/mx6dq/evb_rev_a/bin/mx6dq_evb_rev_a-keypad-sdk.bin`

The multiple keys pressed test works as follows:

1. Waits for a key press event.
2. Upon a key press event's occurence, scans the matrix.
3. Waits for 50 ms
4. Re-scans the matrix in case of multiple pressed keys.
5. Waits for all keys to release.

When the test is completed, the console displays the name of the pressed keys based on the keys map (located in `keypad_test.h)`.

Chapter 20
Configuring the LDB Driver

20.1 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 in the i.MX 6Dual/6Quad and i.MX 6Solo/6DualLite products. 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.

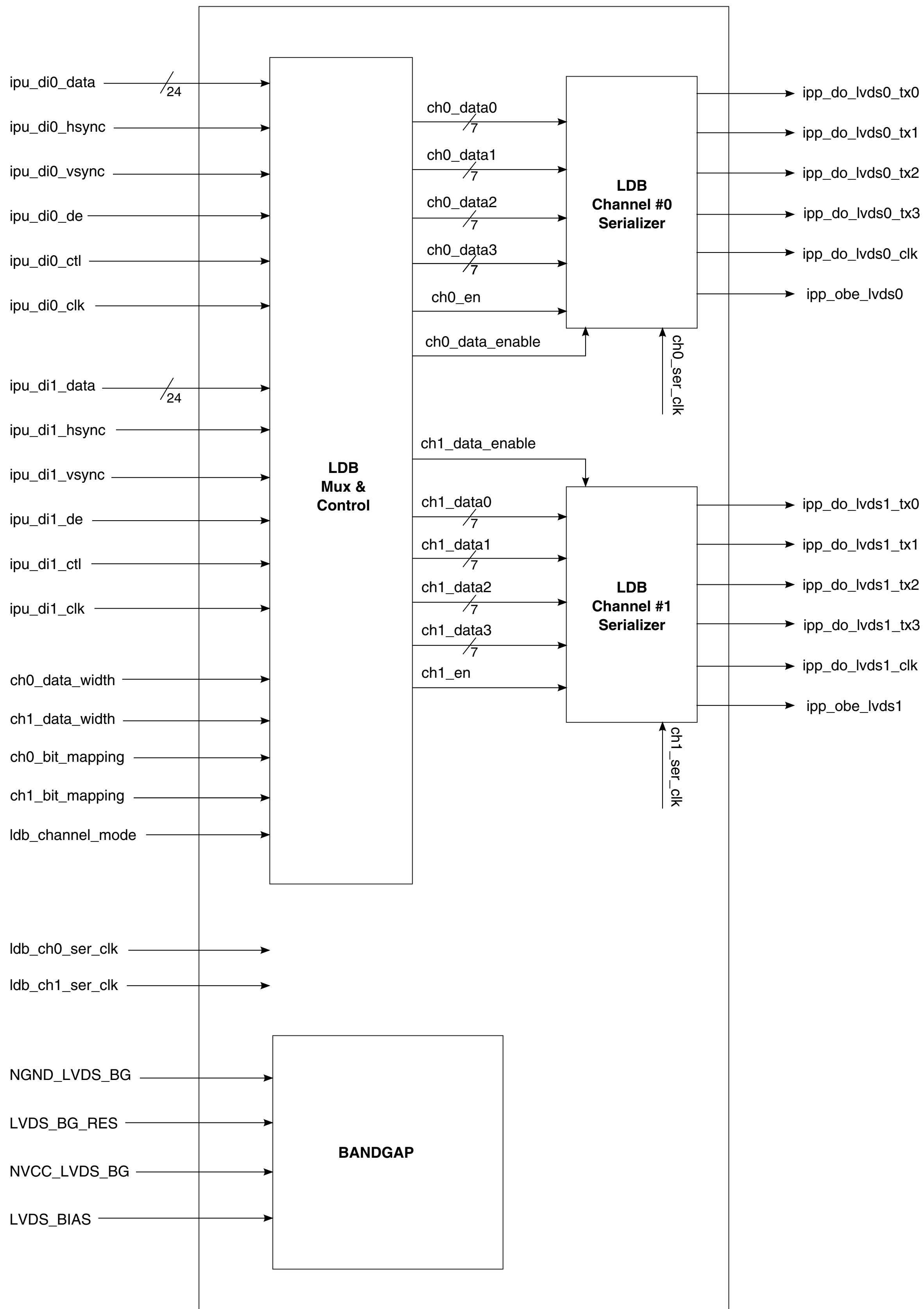


Figure 20-1. LDB block diagram

20.2 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.

20.3 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.

20.4 Modes of operation

LDB supports the following modes of operation:

- Single display mode
- Dual display mode
- Separate display mode
- Split mode

The following table summarizes the channel mapping for each mode:

Table 20-1. Channel mapping

Use Case	LVDS channel 0	LVDS channel 1
Single display mode		
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)

20.4.1 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.

20.4.2 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).

20.4.3 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.

20.4.4 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.

20.5 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.

20.5.1 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 20-2. 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

20.5.2 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 20-3. 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

20.6 Clocks

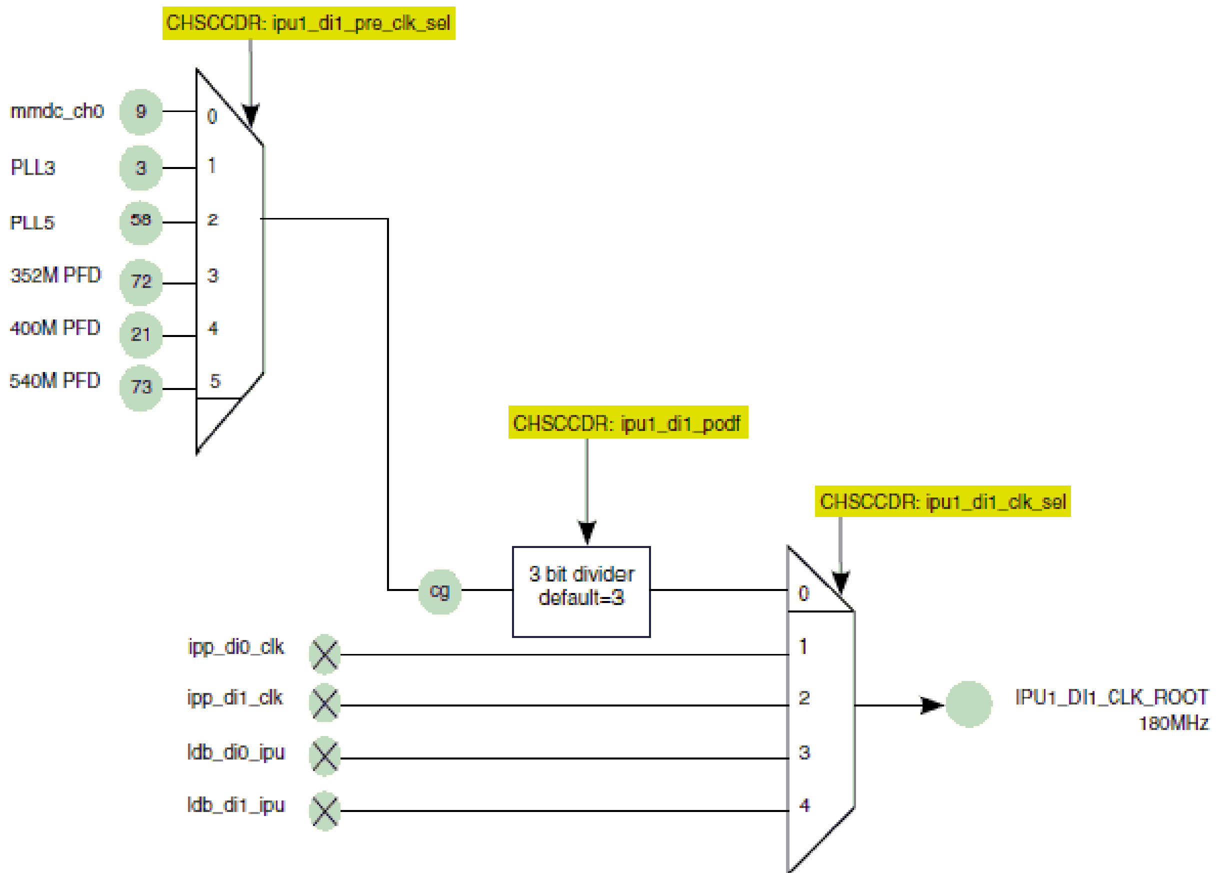


Figure 20-2. LDB clock tree

Route IPU_DI_CLK_ROOT to ipp_di_clk.

20.6.1 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.

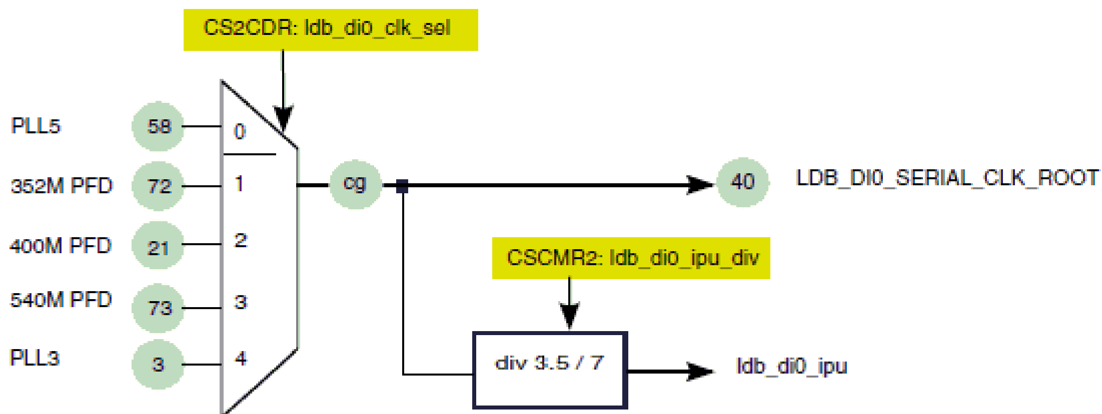


Figure 20-3. LDB serialization clock

The pixel clock is generated by dividing the clock selected by ldb_di_clk_sel (as shown in Figure 20-3, there are five clock sources available) by 3.5 if in split mode or 7 if in non-split mode.

20.7 Configuring the LDB_CTRL register

The following table shows how to configure the parameters in the LDB_CTRL register for your use case:

Table 20-4. LDB_CTRL register parameter configurations

Parameter	Configuration
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

20.8 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 20-5. 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 20-6. 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 21
Configuring the Camera Preview Driver

21.1 Overview

This chapter describes the camera preview driver design for the camera sensor interface (CSI) in the i.MX 6Dual/6Quad and i.MX 6Solo/6DualLite products. It includes a description of the connectivity between the camera sensor and chip.

The following figure shows the task flow between the camera sensor and the display device.

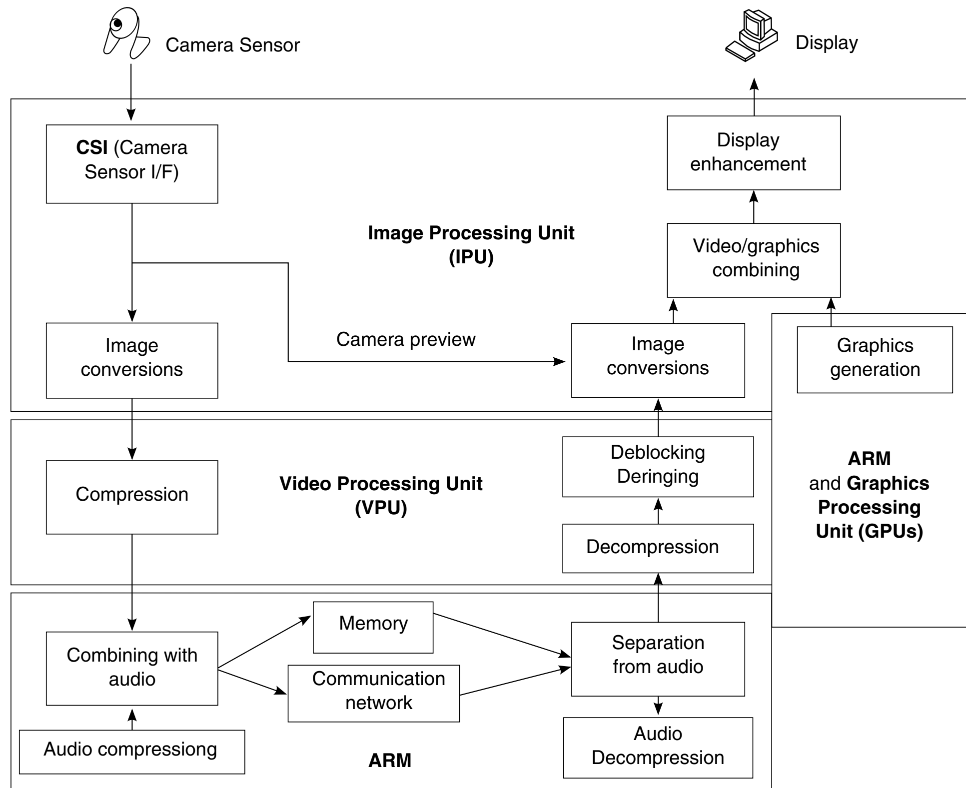


Figure 21-1. Task flow between camera sensor and display

The camera ports receive input from camera sensors and provide support for time-sensitive control signals to the camera. (The ARM main control unit (MCU) performs non-time-sensitive controls, such as configuration or reset, through an I^2C I/F or GPIO signals.) The IPU sends the camera preview image directly to the display. The VPU and ARM performs high performance video processing.

This chip supports four parallel or MIPI camera interfaces. Up to three interfaces may be active at once:

- Two parallel camera ports (up to 20-bit, up to 240 MHz peak2 each).
- MIPI CSI-2 port, supporting from 80 Mbps up to 1 Gbps speed per data lane.

The CSI block provides an interface to an image sensor or a related device.

- Data coming from two parallel camera sensor is received by CSI directly.
- Serial data from MIPI CSI-2 camera sensor is unpacked by MIPI CSI-2 host controller (for further information, see MIPI CSI-2 (Camera Serial Interface 2) driver design chapter. And then the unpacked data is received by CSI.

This chip has two instances of an IPU (IPU1 and IPU2), and each IPU has two CSI blocks (CSI0 and CSI1). The CSI blocks are located in the memory map at the following base addresses:

- CSI0 in IPU1 base address = 0263 0000h
- CSI1 in IPU1 base address = 0263 8000h
- CSI0 in IPU2 base address = 02A3 0000h
- CSI1 in IPU2 base address = 02A3 8000h

21.2 Feature summary

Each of the camera ports includes:

- Direct connectivity to the most relevant image sensors and to TV decoders.
- Parallel interface, up to 20-bit data bus
- Frame size: up to 8192 x 4096 pixels (including blanking intervals)
- Support for the following data formats
  - Raw (Bayer)
  - RGB
  - YUV 4:4:4
  - YUV 4:2:2
  - Grayscale, up to 16 bits per value (component).
- Two methods for synchronization (video mode and still image capture)

The camera ports include the following auxiliary features:

- Frame rate reduction, by periodic skipping of frames
- Downsizing x2, by skipping rows/columns
- Window-of-interest selection
- Pre-flash for red-eye reduction and for measurements such as focus in low-light conditions

21.2.1 Synchronization performance details

In video mode synchronization, the sensor is the master of the pixel clock (PIXCLK) and synchronization signals. Synchronization signals are received using either of the following methods:

- Dedicated control signals (VSYNC, HSYNC) with programmable pulse width and polarity
- Controls embedded in the data stream, loosely following the BT.656 protocol, with flexibility in code values and location

In still image capture synchronization, the image capture is triggered by the MCU or by an external signal, such as a mechanical shutter. Synchronized strobes are generated for up to six outputs: the sensor and five additional camera peripherals, such as the flash or mechanical shutter.

21.2.2 Simultaneous functionality support

Several sensors can be connected to each CSI. Simultaneous functionality (sending data) is supported as follows:

1. Two sensors send data independently, each through a different port.
2. One of the streams is transferred to the VDI or IC for on-the-fly processing while the other one is sent directly to system memory.

21.2.3 Data rate support

The input rate supported by the camera port is as follows:

- Peak: up to 240 MHz (values/sec)
- Average, assuming 35% blanking overhead, for YUV 4:2:2
- Pixel in one cycle (BT.1120): up to 180 MP/sec, for example 12 Mpixels at 15 fps
- Pixel on two cycles (BT.656): up to 90 MP/sec, for example. 6 Mpixels at 15 fps.
- On-the-fly processing may be restricted to a lower input rate.

21.3 Modes of operation

CSI supports the following types of modes of operation:

- Two types of interfaces
- Gated and non-gated mode
- Compliance with recommendation ITU-R BT.656 or ITU-R BT.1120

See Interface modes and Work modes for details about each mode.

21.3.1 Interface modes

The CSI supports using either a parallel interface or a high-speed serial interface (MIPI CSI-2). The DATA_SOURCE register controls the interface mode.

Table 21-1. Interface modes

Mode	What it does
Parallel	Up to 20 bit data inputs 5-6 clocks and controls
High-speed serial (MIPI CSI-2)	Up to 4 D-PHY data lanes Different clock.

21.3.2 Work modes

Table 21-2. Work modes

Mode	What it does
Gated mode	VSYNC is used to indicate beginning of a virtual frame HSYNC is used to indicate beginning of a raw frame, including active sensor frame and horizontal blanking intervals. Sensor clock ticks continuously.
Non-gated mode	VSYNC is used to indicate beginning of a frame. Sensor clock is ticking only when data is valid. HSYNC is not used. Sensor clock is ticking only when data is valid
BT.656 mode	CSI works in compliance with recommendation ITU-R BT.656
BT.1120 mode	CSI works in compliance with recommendation ITU-R BT.1120

21.4 Clocks

The following figure shows the clock interface between the external sensor and the chip. The external sensor and the camera interface share a single clock (sensor_clk) that must be synchronized.

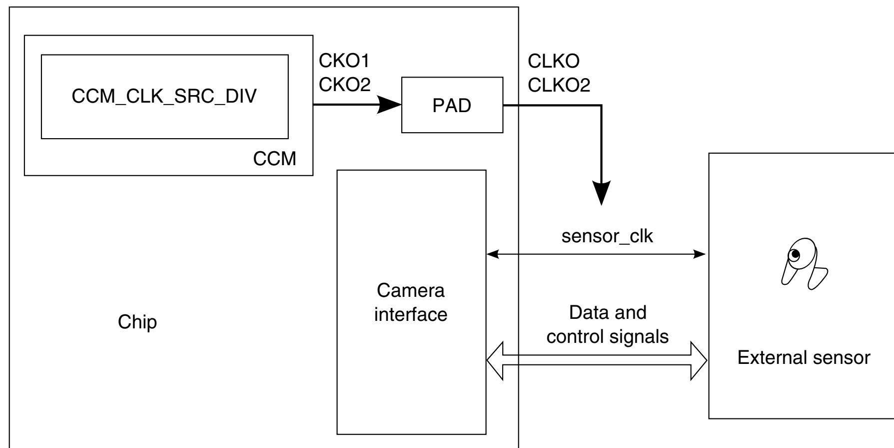


Figure 21-2. Clock interface between the external sensor and the chip

The chip generates sensor_clk and provides it to the external sensor as its input clock. CCM_CLK_SRC_DIV of the chip's internal clock control module (CCM) muxes different clocks to two output clocks: CKO1 and CKO2. CKO1 and CKO2 are connected to several pads. Choose a CLKO/CLKO2 instance of a pad connected CKO1/CKO2.

The following table shows the internal pad selection for CLKO/CLKO2.

Table 21-3. sensor_clk internal source selection

Clock	Pad	Mode
CLKO	CSI0_MCLK	ALT3
	GPIO_0	ALT0
	GPIO_19	ALT3
	GPIO_5	ALT3
CLKO2	GPIO_3	ALT4
	NANDF_CS2	ALT4

21.5 IOMUX pin mapping

Set bits 19-20 of the General Purpose Register (IOMUXC_GPR1) appropriately to enable either the CSI parallel interface or the MIPI CSI-2 interface based on the settings shown in table below. The IOMUX pin mapping varies depending on which interface is enabled.

Table 21-4. Camera interface mode setting

IOMUXC_GPR1	Value	Description
Bit 19	0 (default)	Enables MIPI interface to IPU1 CSI0; virtual channel is fixed to 0
	1	Enables parallel interface to IPU1 CSI0
Bit 20	0	Enables MIPI interface to IPU2 CSI1; virtual channel is fixed to 3
	1 (default)	Enables parallel interface to IPU2 CSI1

The CSI driver controls the camera data source as follows:

- IPU1 CSI0 can configure the data source; it connects to the MIPI CSI-2 interface by default.
- IPU1 CSI1 connects directly to the MIPI CSI-2 interface, and the virtual channel is fixed to 1.
- IPU2 CSI1 can configure the data source; it connects to the parallel interface by default.
- IPU2 CSI0 connects directly to MIPI CSI-2 interface, and the virtual channel is fixed to 2.

21.5.1 IOMUX pin mapping for CSI0/CIS1 parallel interface

Table 21-5. IOMUX pin mapping for CSI*n* parallel interface

Signals	Driver		
	PAD	MUX	SION
CSI*n*_PIXCLK	CSI*n*_PIXCLK	ALT0	1
CSI*n*_HSYNC	CSI*n*_MCLK	ALT0	1
CSI*n*_VSYNC	CSI*n*_VSYNC	ALT0	1
CSI*n*_DATA_EN	CSI*n*_DATA_EN	ALT0	1
CSI*n*_D[12]	CSI*n*_DAT12	ALT0	1
CSI*n*_D[13]	CSI*n*_DAT13	ALT0	1
CSI*n*_D[14]	CSI*n*_DAT14	ALT0	1
CSI*n*_D[15]	CSI*n*_DAT15	ALT0	1
CSI*n*_D[16]	CSI*n*_DAT16	ALT0	1
CSI*n*_D[17]	CSI*n*_DAT17	ALT0	1
CSI*n*_D[18]	CSI*n*_DAT18	ALT0	1
CSI*n*_D[19]	CSI*n*_DAT19	ALT0	1

21.5.2 IOMUX pin mapping for the MIPI CSI-2 interface

The data and control signals—DATA_EN, VSYNC, and HSYNC—are derived from the serial data packet. The MIPI CSI-2 interface does not require IOMUX pin mapping.

21.6 Resets and interrupts

21.6.1 Resets

The chip generates sensor_rst and provides it to the external sensor as its reset signal. The sensor_rst signal can be generated by any GPIO that is connected to the reset pad of the external sensor. The reset signal should follow the timing request of the external sensor.

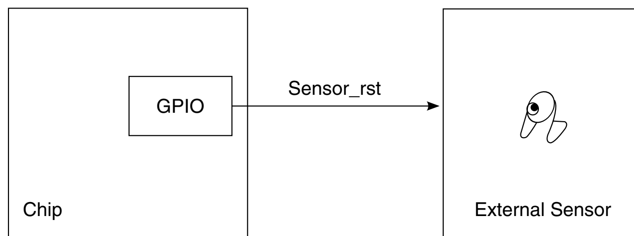


Figure 21-3. Reset signal for external sensor

21.6.2 Interrupts

The IPU's interrupt generator (IG) provides error interrupts to the main control unit (MCU) for monitoring CSI errors. All interrupts are maskable.

The following table describes the error interrupts of CSI.

Table 21-6. CSI error interrupts

Location	Submodule	Interrupt Status Name	Description
IPU_INT_STAT_9 [30]	CSI0	CSI0_PUPE	The error is generated in cases where a new frame arrived from the CSI0 before the SRM (shadow registers module) completed CSI0's parameter updates.
IPU_INT_STAT_9 [31]	CSI1	CSI1_PUPE	The error is generated in cases where a new frame arrived from the CSI1 before the SRM completed CSI1's parameter updates.

21.7 Initializing the driver

The following figure shows the procedure for intializing the camera preview driver:

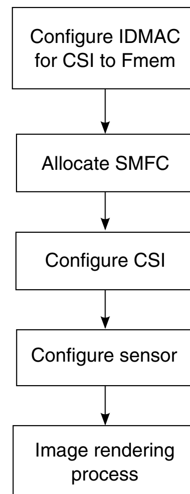


Figure 21-4. Flow for initializing the camera preview driver

21.7.1 Configuring the IDMAC channel for CSI

The following table describes the IDMAC channels. Choose any one of four IDMAC channels as the CSI input channel.

Table 21-7. IDMAC channels for CSI input

Channel #	Source	Destination	Purpose	Data type
0	CSI (via SMFC)	Fmem	VF2 - Bayer; BPP>8; JPEG; MIPI additional channels	Generic or pixel
1	CSI (via SMFC)	Fmem	VF2 - Bayer; BPP>8; JPEG; MIPI additional channels	Generic or pixel
2	CSI (via SMFC)	Fmem	VF2 - Bayer; BPP>8; JPEG; MIPI additional channels	Generic or pixel
3	CSI (via SMFC)	Fmem	VF2 - Bayer; BPP>8; JPEG; MIPI additional channels	Generic or pixel

21.7.2 Allocating SMFC

The SMFC (sensor multifile controller) provides a buffer between the CSI and the IDMAC (image DMA controller). Two masters (CSIs) can be connected to the SMFC. Both masters can be active simultaneously. Each master can send up to four frames,

distinguished by the csi_id bus. The frame can be mapped to one of four IDMAC channels by means of the SMFC mapping registers. Each DMA channel has a dedicated FIFO.

Allocate one SMFC channel for IPU*n* CSI*n*.

21.7.3 Configuring CSI

The CSI obtains data from the sensor, synchronizes the data and the control signals to the IPU clock (HSP_CLK), and, depending on the configuration of DATA_DEST register, transfers it to either the image controller (IC) block, the sensor multifile controller (SMFC), or both.

The following figure shows the data flow for the camera interface (CSI):

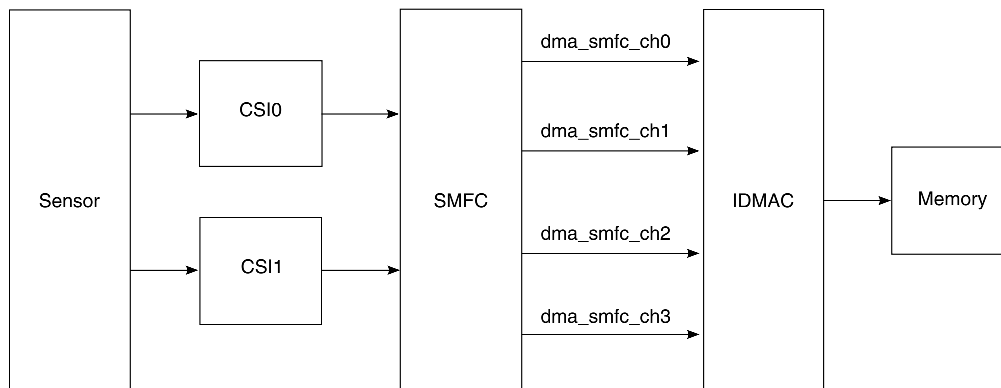


Figure 21-5. Data flow for the CSI

The relative bit fields are:

- CSI*n*_DATA_DEST sets the destination of the data coming from the CSI.
- CSI*n*_DATA_WIDTH sets the number of bits per color.
- CSI*n*_SENS_DATA_FORMAT sets the data format from the sensor.
- CSI*n*_SENS_PRTCL sets the sensor timing/data mode protocol.
- CSI*n*_SENS_FRM_HEIGHT/WIDTH sets the sensor frame height and width.
- CSI*n*_SEL sets whether CSI0 or CSI1 is selected.
- CSI*n*_DATA_SOURCE sets whether the data source for the CSI1 is parallel or MIPI.
- CSI*n*_EN sets whether CSI0 is enabled.

21.7.4 Configuring the sensor

Access the camera sensor through the I^2C interface to initialize and configure the camera sensor.

21.7.5 Image rendering

To send camera preview images to display, see the "Single image rendering example" section in the IPU chapter for the image rendering process settings.

21.8 Testing the driver

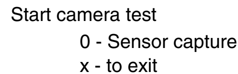


Figure 21-6. Camera test start image

Test the driver using the following procedure:

1. Plug sensor ov5640 into the parallel sensor interface on the development board.
2. Run the Platform SDK test suite.
3. Press 0 to run the parallel sensor capture test.

If the test is sucessful, the LVDS0 display shows the 640 x 480 camera preview.

Chapter 22
Configuring the MIPI CSI-2 Driver

22.1 Overview

This chapter describes the camera preview driver design for the MIPI CSI-2 (camera serial interface 2) host processor for the i.MX 6Dual/6Quad and i.MX 6Solo/6DualLite products. It includes a description of the connectivity between the MIPI CSI-2 host controller and the MIPI CSI-2 camera sensor.

The following figure shows the relationship between the MIPI CSI-2 camera sensor and the chip.

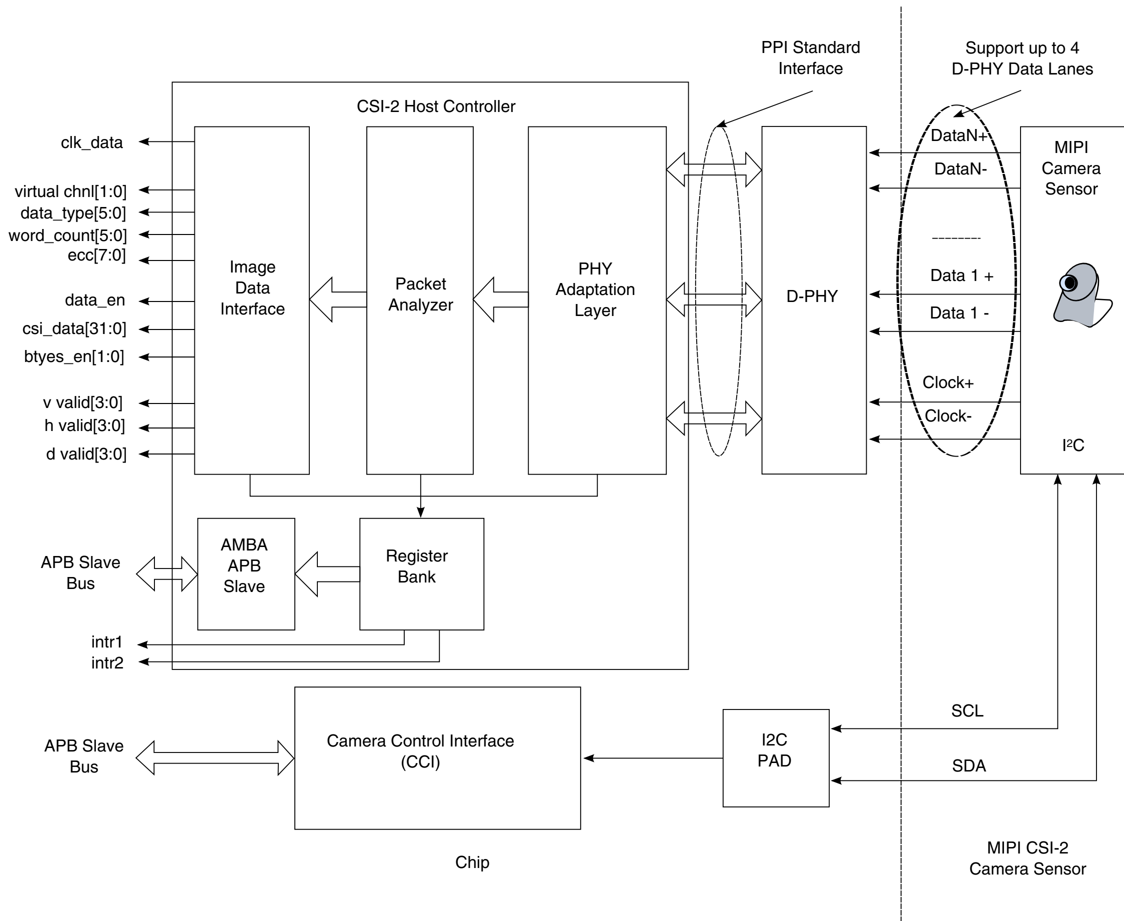


Figure 22-1. Relationship between MIPI CSI-2 camera sensor and the chip

The MIPI CSI-2 is a high speed serial interface, and the chip works as a MIPI CSI-2 receiver. It is controlled by a CSI-2 host controller and camera control interface (CCI).

- The CSI-2 host controller receives data from a CSI-2 compliant camera sensor.
  - It is a digital core that implements all protocol functions defined in the MIPI CSI-2 specification.
  - It provides an interface between the chip image data interface and the MIPI D-PHY, allowing communication with a MIPI CSI-2 compliant camera sensor.
- The camera control interface (CCI) controls the transmission through I^2C interface.
  - A CSI-2 receiver should be configured as a master on the CCI bus.
  - A CSI-2 transmitter should be configured as a slave on the CCI bus.

The MIPI CSI module is located in the memory map at 021D C000h.

22.2 Feature summary

The MIPI CSI-2 host controller supports the following features:

- Conformity with multiple standards
    - MIPI Alliance Standard for Camera Serial Interface 2 (CSI-2), Version 1.00
    - Interface with MIPI D-PHY following PHY Protocol Interface (PPI), as defined in MIPI Alliance Specification for D-PHY, Version 1.00.00
    - Optional support for camera control interface (CCI) through the I^2C interface
- Supports up to four D-PHY Rx data lanes
- Dynamically configurable multi-lane merging
- Long and short packet decoding
- Timing accurate signaling for frame and line synchronization packets
- Support for several frame formats such as:
    - General Frame or Digital Interlaced Video with or without accurate sync timing
    - Data type (packet or frame level) and virtual channel interleaving
- 32-bit image data interface, delivering data formatted as recommended in the CSI-2 specification
- Supports all primary and secondary data formats
    - RGB, YUV, and RAW color space definitions
    - From 24-bit down to 6-bit per pixel
    - Generic or user-defined byte-based data types
- Error detection and correction
    - PHY level
    - Packet level
    - Line level
    - Frame level

22.3 Modes of operation

Table 22-1. MIPI CSI-2 modes of operation

Mode	What it does
Continuous clock	Clock lane remains in high-speed mode generating active clock signals between the transmission of data packets
Non-continuous clock	Clock lane enters the LP-11 state between the transmission of data packets.

22.4 Clocks

22.4.1 Output clock

The following figure shows the clock interface between the external sensor and the chip. The external sensor and the camera interface share a single clock (**sensor_clk**) that must be synchronized.

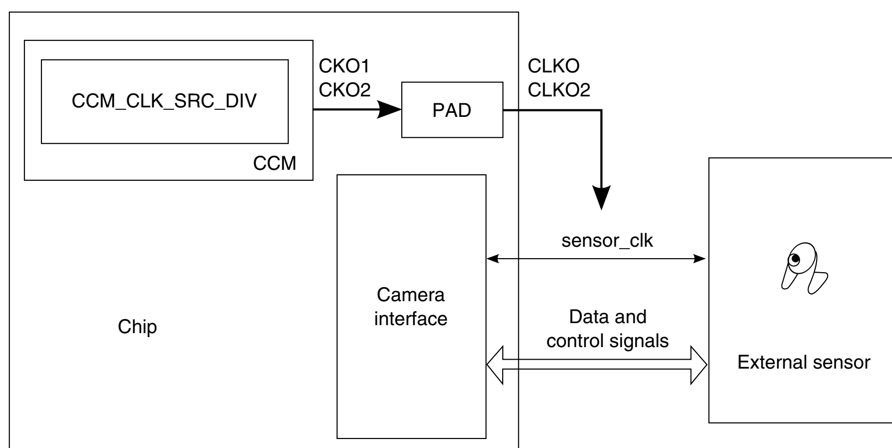


Figure 22-2. Clock interface between the external sensor and the chip

The chip generates **sensor_clk** and provides it to the external sensor as its input clock. CCM_CLK_SRC_DIV of the chip's internal clock control module (CCM) muxes different clocks to two output clocks: CKO1 and CKO2. These output clocks are connected to several pads. Choose a CLKO/CLKO2 instance of a pad connected to CKO1/CKO2.

22.4.2 Input clock

The MIPI CSI-2 host controller typically works with the high speed byte clock provided by RXBYTECLKHS. RXBYTECLKHS is, by specification, 1/4 of the DDR clock on the D-PHY clock.

The DDR clock (RxDDRClkHS) is received from the MIPI camera sensor. The MIPI CSI-2 clock lane receives the clock± signals and obtains the high-speed receive DDR clock. The following figure shows the MIPI CSI-2 clock lane receiver.

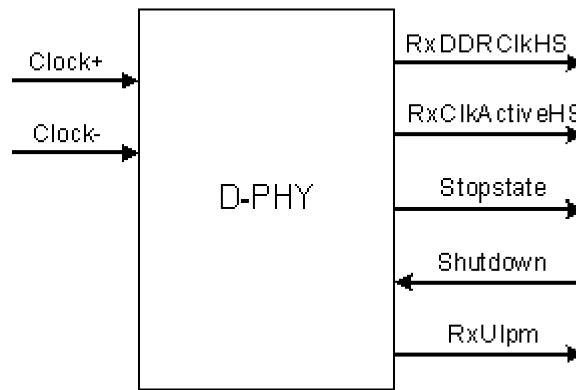


Figure 22-3. MIPI CSI-2 clock lane receiver

The following table describes the clock lane PPI interface signals to the MIPI CSI-2 host controller.

Table 22-2. PPI interface signals

Interface signal name	Input/output	Description
RxDDRClkHS	Output	High-speed receive DDR clock; samples the data in all data lanes
RxClkActiveHS	Output	High-speed reception active. This active high signal indicates that the clock lane is receiving valid clock. This signal is asynchronous.
Stopstate	Output	Lane is in stop state. This active high signal indicates that the lane module is currently in stop state. This signal is asynchronous.
Shutdown	Input	Shutdown lane module. This active high signal forces the lane module into shutdown, disabling all activity. All line drivers, including terminators, are turned off when shutdown is asserted. When shutdown is high, all PPI outputs are driven to the default inactive state. Shutdown is a level sensitive signal and does not depend on any clock.
RxUlpmEsc	Output	Escape ultra low power (receive) mode. This active high signal is asserted to indicate that the lane module has entered the ultra low power mode. The lane module remains in this mode with RxUlpmEsc asserted until a stop state is detected on the lane interconnect.

The following table summarizes the MIPI CSI-2 clocks.

Table 22-3. Reference clocks

Clock	Name	Description
Sensor clock	CLKO	Select osc_clk 24 MHz to generate CLKO
DDR clock	RxDDRClkHS	Receive from MIPI CSI-2 camera sensor

22.5 IOMUX pin mapping

The MIPI CSI-2 does not require IOMUX pin mapping. As shown in Figure 22-1, the CSI-2 host controller derives its image data and frame control signal from the serial data packet.

22.6 Resets and interrupts

The CSI-2 host controller provides an interrupt mechanism for monitoring errors and debugging. The interrupt mechanism uses two interrupt signals: intr1 and intr2. These signals are synchronous with the AMBA APB clock signal.

Registers MASK1 and MASK2 respectively assert intr1 and intr2 to select which bits of registers ERR1 and ERR2 can generate interrupts. Both ERR1 and ERR2 always contain the information about events, regardless of the state of MASK1 and MASK2. ERR1 and ERR2 self-clear after a read access. Interrupt signals intr1 and intr2 are de-asserted upon read access of ERR1 and ERR2, respectively. For more information, see the "Error state register 1 (MIPI_CSI_ERR1)" and "Error state register 2 (MIPI_CSI_ERR2)" sections of the MIPI-CSI chapter in the reference manual.

22.7 Initializing the driver

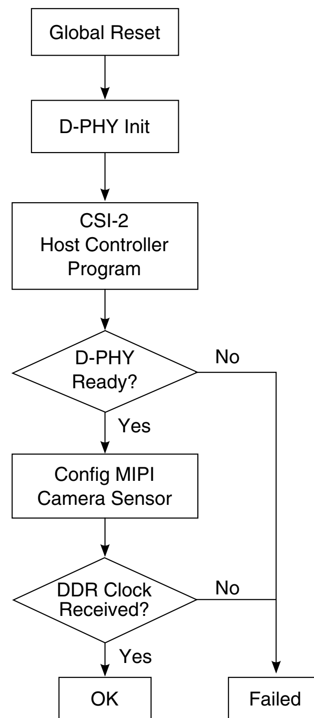


Figure 22-4. Flow for initializing the driver

1. Deassert the CSI2 presetn signal (global reset).
2. Configure the MIPI camera sensor to have all Tx lanes in the LP-11 state (STOPSTATE) if required.
3. The D-PHY specification states that the D-PHY master should be initialized at LP-11 state (STOPSTATE); however, a CCI command may be required to switch-on the MIPI interface.
4. Access the D-PHY programming interface to initialize and program the D-PHY according to the selected operating mode. This step is D-PHY dependent; use the D-PHY data book to identify the correct programming.
5. Program the CSI2 Host controller registers according to the operating mode's requirements:
    - Number of Lanes (register N_LANES)
    - Deassert PHY shutdown (register PHY_SHUTDOWNZ)
    - Deassert PHY reset (register PHY_RSTZ)
    - Deassert CSI reset (register CSI2_RESETN)
    - (Optional) Program Data IDs for matching error reporting (registers DATA_IDS_1 and DATA_IDS_2)
    - (Optional) Program the interrupt masks (registers MASK1 and MASK2)

6. Read the PHY status register (PHY_STATE) to confirm that all data and clock lanes of the D-PHY are in stop state, which means they are ready to receive data.
7. Access the camera sensor using the CCI interface to initialize and configure the camera sensor to transmit a clock on the D-PHY clock lane.
8. Read the PHY status register (PHY_STATE) to confirm that the D-PHY is receiving a clock on the D-PHY clock lane.

22.8 Testing the driver

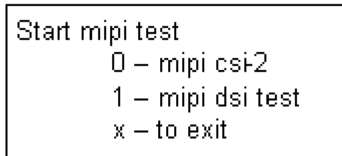


Figure 22-5. MIPI test screen image

Test the driver using the following procedure

1. Plug sensor ov5640 into the MIPI interface on the development board.
2. Run the platform SDK test suite.
3. Press 0 to run the mipi csi-2 test.

If the test is sucessful, the LVDS0 display shows the 640 x 480 camera preview from the MIPI CSI-2.

Chapter 23
Configuring the MIPI DSI driver

23.1 Overview

This chapter explains how to configure the MIPI DSI driver for the i.MX 6Dual/6Quad and i.MX 6Solo/6DualLite products. The DSI (display serial interface) host controller is a digital core that implements all protocol functions defined in the MIPI DSI specification. It provides an interface between the system and the MIPI D-PHY, which allows communication with a MIPI DSI compliant display.

The following figure shows the overall architecture of the DSI host controller. See the "Architecture" section of the MIPI DSI chapter in the reference manual for a description of the component blocks.

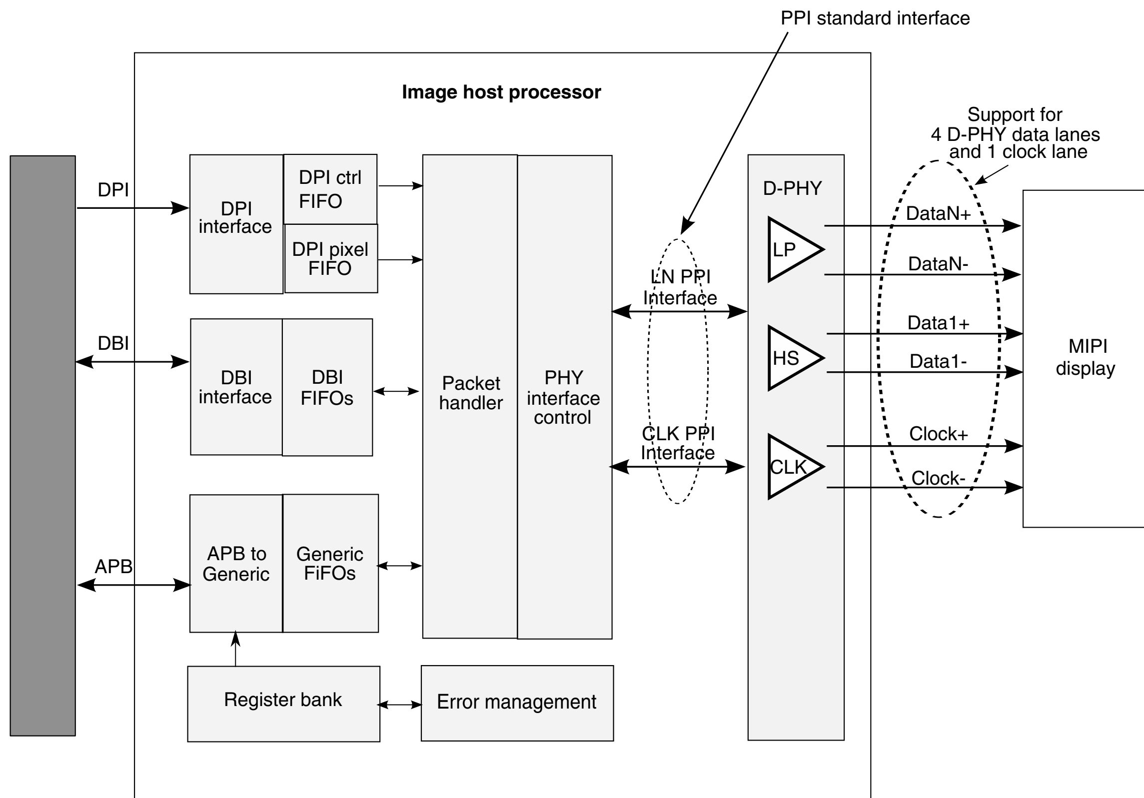


Figure 23-1. MIPI DSI block guide

The MIPI DSI module is located in the memory map at the MIPI DSI base address: 021E 0000h

23.2 Feature summary

The MIPI DSI host controller supports the following features:

- Conformity to standards and specifications as follows:
  - Conforms to the MIPI alliance standard for display pixel interface (DPI-2), version 2.00, with pixel data bus width up to 24 bits
  - Conforms to the MIPI alliance standard for display bus interface (DBI-2), version 2.00, for the following DBI types:
    - Type A Fixed E mode
    - Type A Clocked E mode
    - Type B
    - 16-bit, 9-bit and 8-bit data bus width
  - Support all commands defined in MIPI Alliance specification for display command set (DCS), Version 1.02.00
  - Interface with MIPI D-PHY following PHY protocol interface (PPI), as defined in MIPI alliance specification for D-PHY, Version 1.00.00

- Supports the following general features:
  - Up to 4 D-PHY data lanes
  - Bidirectional communication and escape mode support through data lane 0
  - Programmable display resolutions, from 160 x 120 (QQVGA) to 1024 x 768 (XVGA)
  - Multiple peripheral support capability and configurable virtual channels
  - Transmission of all generic commands
  - ECC and checksum
  - End of transmission packet (EoTp)
  - Ultra low power mode
  - Schemes for fault recovery
- Supports the following video mode pixel formats
  - 16 bpp (5,6,5 RGB)
  - 18 bpp (6,6,6,RGB) packed
  - 18 bpp (6,6,6,RGB) loosely
  - 24 bpp (8,8,8,RGB).

23.3 Modes of operation

The following two tables show the MIPI DSI interface modes and the MIPI DSI operating modes. Note that the display bus interface and display pixel interface can coexist, but only one can be operational.

Table 23-1. MIPI DSI interface modes

Data interface	What it does
DPI mode	In DPI mode, the DPI interface captures the data and control signals and conveys them to the FIFO interfaces that transmit them to the DSI link. Two different streams of data are present at the interface: video control signals and pixel data. Depending on the interface color coding, the pixel data is handled differently throughout the dpipixdata bus.
DBI mode	In DBI mode, the DBI-2 interface encapsulates DCS commands in DSI packets to be transmitted through the D-PHY link

Table 23-2. MIPI DSI operating modes

Operating mode	What it does
High-speed mode	Data transmission at higher speeds than the other operating modes
Escape mode	Escape mode is an optional mode of operation for data lanes. This mode allows low bit-rate commands and data to be transferred in a very low power state
Control mode	Provides an alternative to escape mode for a low-power signaling state.

23.4 Clocks

To start the internal PLL, sett the PLL clock by calling **dphy_write_control()**.

The working clock for DSI controller and DPHY are all derived from an on-chip clock source. The reference clock is 27 MHz. The following table shows the configuration options for the work frequency of the DSI controller and PHY.

Table 23-3. Configuration settings for the DSI working clock

Ranges (Mbps)	Settings (binary)	Default bit rate (Mbps)
80-90 (default)	000000	90
90-100	010000	99
100-110	100000	108
110-125	000001	123
125-140	010001	135
140-150	100001	150
150-160	000010	159
160-180	010010	180
180-200	100010	198
200-210	000011	210
210-240	010011	240
240-250	100011	249
250-270	000100	270
270-300	010100	300
300-330	000100	330
330-360	010101	360
360-400	100101	399
400-450	000110	450
450-500	010110	486
500-550	000111	549
550-600	010111	600
600-650	001000	648
650-700	011000	699
700-750	001001	750
750-800	011001	783
800-850	001010	849
850-900	011010	900
900-950	101010	972
950-1000	111010	999

23.5 IOMUX pin mapping

The MIPI DSI interface does not need IOMUX pin mapping.

23.6 Resets and Interrupts

The DSI supports multiple methods for reset.

- For soft reset, the DSI host sends soft_reset (command ID 01h) through DCS.
- For hard reset of DSI controller, set register PWR_UP.
- For a D-PHY reset, set register PHY_RSTZ.

23.7 Initializing the driver

This driver is initialized in two parts:

- Initializing the DSI controller
- Initializing the D-PHY

23.7.1 Initializing the DSI controller

To initialize the DSI to work in DPI mode, use the following procedure.

1. Global configuration.
2. Configure the DPI interface.
3. Select the video transmission mode.
4. Define the DPI horizontal timing configuration.
5. Define the vertical line configuration.

See the following subsections for detailed information about each step.

23.7.1.1 Global configuration

Use DSI_PHY_IF_CFG[1:0] to configure the number of lanes available to the controller for performing high speed transmissions. The settings are as follows:

- Use 00b for a single data lane (lane 0).
- Use 01b for two data lanes (lane 0 and 1).

- Use 10b for three data lanes (lane 0, 1, and 2).
- Use 11b for four data lanes (all)

See the "D-PHY interface configuration (MIPI_DSI_PHY_IF_CFG_)" section in the MIPI DSI chapter of the reference manual for additional information about this register.

23.7.1.2 Configure the DPI interface

Configure the DSI_DPI_CFG register to define how the DPI interface interacts with the controller. The fields are:

- Virtual Channel ID (DSI_DPI_CFG[1:0])-configures the virtual channel that the packet generated by this interface is indexed to.
- Dpi_color_coding (DSI_DPI_CFG [4:2])-configures the bits per pixels that the interface transmits and also the variant configuration of each bpp.

NOTE

When the 18 bpp is selected and Enable_18_loosely_packed is not active, the number or pixels per line must be a multiple of four.

- Data_active_low_enable (DSI_DPI_CFG [5])-configures the polarity of the DATAEN signal and enables it as active low.
- Vsync_active_low_enable (DSI_DPI_CFG [6])-configures the polarity of the VSYNC signal and enables it as active low.
- Hsync_active_low_enable (DSI_DPI_CFG [7])-configures the polarity of the HSYNC signal and enables it as active low.
- Shutd_active_low_enable (DSI_DPI_CFG [8])-configures the polarity of the SHUTD signal and enables it as active low.
- Colorm_active_low_enable (DSI_DPI_CFG [9])-configures the polarity of the COLORM signal and enables it as active low.
- Enable_18_loosely_packed (DSI_DPI_CFG [10])-configures whether pixel packing is loose or packed when dpi_color_coding selects 18 bpp. This bit enables loose packing.

See the "DPI interface configuration (MIPI_DSI_DPI_CFG)" section of the MIPI DSI chapter in the reference manual for additional information about this register.

23.7.1.3 Select the video transmission mode

Use the DSI_VID_MODE_CFG to define how the the video line will be transported through the DSI link. The fields are as follows:

- The enable low power fields (DSI_VID_MODE_CFG[8:3]) defines the video periods that are permitted to go to low power if there is available time to do so.
- frame_bta_ack (DSI_VID_MODE_CFG [11]) defines whether the controller should request the peripheral acknowledge message at the end of a frames.
- vid_mode_type (DSI_VID_MODE_CFG[2:1]) sets whether the mode is burst or non-burst.
    - In burst mode, the entire active pixel line is buffered into a FIFO and transmitted in a single packet with no interruptions. This transmission mode requires that the DPI pixel FIFO can store a full line of active pixel data inside it. This mode is best used when there is a large difference between the pixel required bandwidth and DSI link bandwidth because it enables the controller to dispatch the entire active video line in a single burst of data and then return to low power.
        - Configure video_mode_type (DSI_VID_MODE_CFG [2:1]) with value 01b.
        - Configure video_packet_size (DSI_VID_PKT_CFG [10:0]) with the size of the active line period.
        - The controller automatically ignores the following fields: enable_multiple_packets (DSI_VID_MODE_CFG [9]), enable_null_packets (DSI_VID_MODE_CFG [10]), number_of_chunks (DSI_VID_PKT_CFG 20:11]) and null_packet_size (DSI_VID_PKT_CFG [30:21]).
    - In non-burst mode, the processor uses the partitioning properties of the controller to divide the video line transmission into several DSI packets, which matches the pixel required bandwidth to the DSI link bandwidth. This mode allows the controller configuration not store only the content of one video packet inside the DPI pixel FIFO instead of a full line of pixel data.
        - Configure the register field video_mode_type (DSI_VID_MODE_CFG [2:1]) with the value 00b.
        - Configure video_mode_type (DSI_VID_MODE_CFG [2:1]) with 00b to enable the transmission of sync pulses.
        - Configure video_mode_type (DSI_VID_MODE_CFG [2:1]) with 01b to enable the transmission of sync events.
        - Configure the video_packet_size (DSI_VID_PKT_CFG[10:0]) with the number of pixels to be transmitted in a single packet.
        - The field enable_multiple_packets (DSI_VID_MODE_CFG [9]) enables the division of the active video transmission into more than one packet.
        - The field number_of_chunks (DSI_VID_PKT_CFG [20:11]) configures the number of video chunks that the active video transmission is divided into.
        - The field enable_null_packets (DSI_VID_MODE_CFG [10]) enables the insertion of null packets between video packets.
        - The field null_packet_size (DSI_VID_PKT_CFG [30:21]) configures the actual size of the inserted null packet.

See the "Video Mode Configuration (MIPI_DSI_VID_MODE_CFG)" section and the "Video packet configuration (MIPI_DSI_VID_PKT_CFG)" section of the MIPI DSI chapter in the reference manual for additional information about these fields.

23.7.1.4 Define the DPI horizontal timing configuration

Use the TMR_LINE_CFG register to define the DPI horizontal timing configuration. The fields are as follows:

- hline_time (TMR_LINE_CFG[31:18])-configures the time taken by a DPI video line.

NOTE

When the DPI clock and clock lane clock are not multiples, hline_time is the result of a rounded number. If the core is configured to go to low power a few times, an error induced in one line can be incremented with the next one. At the end of several lines, the controller may have enough error to cause the video transmission to malfunction.

- hsa_time (TMR_LINE_CFG [8:0])-configures the time taken by a DPI horizontal sync active period.
- hbp_time (TMR_LINE_CFG[17:9])-configures the time taken by the DPI horizontal back porch period. Pay close attention to the calculation of this parameter and all timing parameter settings. if the timing setting does not match the IPU output signals, there can be problems with the display, such as the screen flicking or lines displaying abnormally.

NOTE

All time is calculated in clock lane bytes clock cycles, which is normally a period of 8 ns.

See the "Line timer configuration (MIPI_DSI_TMR_LINE_CFG)" section in the MIPI DSI chapter of the reference manual for additional information about the register.

23.7.1.5 Define the vertical line configuration

Use the DSI_VTIMING_CFG register to define the vertical line configuration. The fields are as follows:

- vsa_lines (DSI_VTIMING_CFG[3:0])-configures the number of lines existing in the DPI vertical sync active period.

- vbp_line (DSI_VTIMING_CFG [9:4])-configures the number of lines existing in the DPI vertical back porch period.
- vfp_line (DSI_VTIMING_CFG [15:10])-configures the number of lines existing in the DPI vertical front porch period.
- vertical_active_lines (DSI_VTIMING_CFG [26:16])-configures the number of lines existing in the DPI vertical active period.

See the "Vertical timing configuration (MIPI_DSI_VTIMING_CFG)" section of the MIPI DSI chapter in the reference manual for additional information about the register.

23.7.2 Initializing the D-PHY

The initialization procedure is as below:

1. By default, the PHY_RSTZ register activates the PHY resets physhutdownz and phyrstz as well as disables enableclk.
2. Configure the PHY_IF_CFG register with the correct number of lanes to be used by the controller.
3. Configure the TX_ESC clock frequency to a frequency lower than 20 MHz, which is the maximum allowed frequency for D-PHY ESCAPE mode.
    - Write in CLKMGR_CFG[TX_ESC_CLK_DIVISION] (see the "Number of active data lanes (MIPI_DSI_CLKMGR_CFG)" section in the MIPI DSI chapter in the reference manual).
    - TX_ESC_CLK_DIVISION divides the byte clock and generates a TX_ESC clock for the D-PHY.
4. Configure the DPHY PLL clock frequency through the TEST interface to operate at 1 GHz, assuming that the REF_CLK is provided with a frequency of 27 MHz.

    This step is performed in dphy_write_control. The first parameter is the control command, and the second parameter is the clock selection number. Refer to Table 23-3 for different clock configurations.

    - Write at PHY_TST_CTRL0-0000 0000h disables the testclr pin, which enables the interface to write new values to the DPHY internal registers.
    - Write at PHY_TST_CTRL1-0001 0044h enables the testen pin bit 16 of this core register and configures the test datain to 44h. This operation initiates the configuration process of the test code number 44h.
    - Write at PHY_TEST_CTRL0-0000 0002h followed by a new write to PHY_TEST_CTRL0 0000 0000h. This operation toggles the testclk (bit 2). The testdin is sampled on the falling edge of testclk latching a new test code.

- Write at PHY_TEST_CTRL1-0000 0074h disables the testen pin and configures testdatain to 74h. This operation prepares the interface to load the 74h value into the test code 4h.
- Write at PHY_TEST_CTRL0-0000 0002h followed by a new write to PHY_TEST_CTRL0 0000 0000h. This operation toggles the testclk. The testdin is sampled on the rising edge of testclk, latching new content data to the configured test code.
- Write at PHY_RSTZ-0000 0007h asserts physhutdownz, phyrstz, and enableclk releasing the PHY from power down. The PHY initiates the PLL locking procedure to 1 GHz operation.
- Read at PHY_STATUS - *nnnn nnn*1, until bit 0, phylock, is detected at 1. This signals that the PLL is locked and that a stable byte clock is being provided to the DSI host controller.
- Read at PHY_STATUS - *nnnnnnn*3h, until bit 2, phystopstateclklane, is read at 1. This identifies that the clock lane is in stop state. The clock lane needs to be in stop state so that the D-PHY can switch to other operational states, such as high speed mode.

5. Write register PHY_IF_CTRL bit 0 to generate high speed clock (txrequestHSclk).
6. After the PLL is locked and the clock lane is in stop state, the PHY drives the correct LP sequence to configure the receiver end for high speed.
7. D-PHY transmits the high speed clock on the clock lane.
8. (Optional) Program the interrupt masks (registers MASK1 and MASK2).

23.8 Testing the driver

Test the drive according to the following procedure.

1. Connect the MIPI expansion board to the chip's CPU board
2. Run the MIPI test program.
3. Select the MIPI display test.

The screen should display an image.

Chapter 24
Configuring the Power Modes

24.1 Overview

This chapter explains how to use the low-power modes driver, which provides an example of how to use the processor's low power modes.

24.2 Feature summary

This low-level driver supports:

- Entering in wait or stop mode.
- Configuration of the interrupt sources that can wake up the processor.

24.3 Modes of operation

Table 24-1. Low-power modes of operation

Mode	What it does
Enter wait state	Places a core in a wait-for-interrupt state with the core clock disabled and well-biasing enabled.
Enter stop mode	Places a core in a wait-for-interrupt state with all clocks disabled and well-biasing enabled.

24.4 Clocks

All clocks are under the fully automated control of the clock controller module (CCM).

24.5 IOMUX pin mapping

The low-power functions do not implement the ability to control an external power management IC. Even if this is an option, the current driver does not need to configure any I/Os.

24.6 Resets and interrupts

The clock controller module resets along with the chip with a warm or cold reset. This is reset with the chip's global reset signal and is not a software controllable reset.

The low-power functions do not have a dedicated interrupt. However, all interrupt sources can be used to wake up the processor.

24.7 Using the driver

This driver provides an example implementation of the low-power modes. The processor's ability to lower the core voltage or switch that voltage off is not implemented.

To enter wait or stop mode, call the following function:

```
/*!
 * Prepare and enter in a low power mode.
 * @param   lp_mode - low power mode : WAIT_MODE or STOP_MODE.
 */
void ccm_enter_low_power(uint32_t lp_mode)
```

To enable the interrupt source(s) that can wake up the system once in a wait for interrupt state, call the following function:

```
/*!
 * Mask/Unmask an interrupt source that can wake up the processor when in a
 * low power mode.
 * @param   irq_id - ID of the interrupt to mask/unmask.
 * @param   state - masked/unmasked the source ID : ENABLE/DISABLE.
 */
void ccm_set_lpm_wakeup_source(uint32_t irq_id, uint32_t state)
```

All interrupt IDs are provided in the "Interrupt and DMA events" chapter of the reference manual.

24.8 Testing the driver

A test is available that excersises this set of functions. It uses the EPIT timer as a source for the interruption that wakes up the processor. The core is first placed in a wait state during 5 seconds; then it is placed in stop mode for approximately 5 seconds.

24.9 Running the test

To run the lower-power modes test, the SDK builds the test with the following command:

```
./tools/build_sdk -target mx6dq -board sabre_ai -board_rev a -test power_modes
```

This generates the following ELF and binary files:

- `./output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-power_modes-sdk.elf`
- `./output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-power_modes-sdk.bin`

Chapter 25
Configuring the OCOTP Driver

25.1 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.

25.2 Feature summary

This low-level driver supports:

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

25.3 Modes of operation

The following table explains the OCOTP modes of operation:

Table 25-1. 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

25.4 Clocks

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

Table 25-2. 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.

25.5 IOMUX pin mapping

This module has no off-chip connection.

25.6 Resets and interrupts

This block is reset with the chip's global reset signal and does not have a software controllable reset.

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/mx6dq/soc_memory_map.h.

25.7 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 the chip reference manual.

25.8 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.

25.9 Running the test

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

```
./tools/build_sdk -target mx6dq -board sabre_ai -board_rev a -test ocotp
```

This generates the following ELF and binary files:

- `./output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-ocotp-sdk.elf`
- `./output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-ocotp-sdk.bin`

Chapter 26
Configuring the PCI Express Driver

26.1 Overview

This chapter describes the PCI Express (PCIe controller and PHY) operation and programming at the module level. All supplied pseudocode is based on the source code of GPU3D driver, which is delivered with the Plat-SDK.

The PCIe driver:

- Provides APIs to operate the PCIe controller.
- Provides APIs to operate the PCIe PHY.
- Provides APIs for the PCIe protocol.

The chip's PCIe controller and the PCIe PHY are compatible with PCI Express Spec 2.0 and support one PCIE link.

There is only one instance of PCIe. It is located in the memory map at PCIE base address = 01FF C000h

The GPU3D testing demo is based on an engineering sample board and can be easily ported to other boards.

26.2 Feature summary

The PCIe controller includes the following features.

- Compatible with PCI Express Spec 2.0.
- Supports root complex and endpoint modes
- Internal memory address mapping supported (iATU)
- Supports one PCIE link
- Supports the debug feature

The PCIe PHY includes the following features:

- 5 Gbps data transmission rate
- PIPE3 compliant transceiver interface
- Configurable using soft PCS layer above hard macro PHY
- Supports the following interfaces:
    - 8-bit interface at 500 MHz operation
    - 16-bit interface at 250 MHz operation
    - 32-bit interface at 125 MHz operation
- Integrated PHY includes the following:
    - Transmitter
    - Receiver
    - PLL
    - Digital core
    - ESD
- Programmable Rx equalization
- Designed for excellent performance margin and receiver sensitivity
- Robust PHY architecture tolerates wide process, voltage and temperature variations
- Low-jitter PLL technology with excellent supply isolation
- IEEE 1149.6 (JTAG) boundary scan
- Built-in self-test (BIST) features for production at-speed testing on any digital tester
- 5Gb/s PCIe Gen 2 and 2.5Gb/s PCIe Gen 1.1 test modes supported
- Advanced built-in diagnostics including on-chip sampling scope for easy debug
- Visibility and controllability of hard macro functionality through programmable registers in the design
- Overrides on all ASIC side inputs for easy debug
- Access register space through simple 16-bit parallel interface or through JTAG

26.3 Modes of operation

This driver configures the PCIe controller working in root complex (RC) mode. Refer to the PCI Express Specification, version 2.0 for the definition of RC. The specification is located at http://www.pcisig.com.

Table 26-1. PCIe operation modes

Mode	What it does
RC	The PCIe works as root complex.

26.4 Clocks

Table 26-2. PCIe reference clocks

Clock	Name	Description
PCIe reference clock	Ref_PCIe	Reference Clock for the PCIe endpoint device, 125 MHz

26.5 IOMUX pin mapping

All PCIe pins (PCIE_RXM, PCIE_RXP, PCIE_TXM, PCIE_TXP) are dedicated to the PCIe PHY itself. There is no need to configure the IOMUX.

26.6 Resets and interrupts

This driver does not implement an interrupt mode.

26.7 Initializing the driver

Use the following code to initialize the PCIe driver.

```
int pcie_init(pcie_dm_mode_e dev_mode)
{
          Configure the GPR fields related with PCIE.
          Configure the PCIE reference clock.
          Reset the PCIE endpoint device.
          Start linkup the endpoint device.
          Wait until the linkup setup unless timeout.
}
```

26.8 Testing the driver

Build the SDK with the following command:

```
./tools/build_sdk -target mx6dq -board evb -board_rev a -test pcie
```

This generates an ELF and file into:

- `output/mx6dq/evb_rev_a/bin/mx6dq_evb_rev_a-pcie-sdk.elf`
- `output/mx6dq/evb_rev_a/bin/mx6dq_evb_rev_a-pcie-sdk.bin`

Download `mx6dq_evb_rev_a-pcie-sdk.elf` using RV-ICE or Lauterbach or burn `mx6dq_evb_rev_a-pcie-sdk.bin` to an SD card with the following command (entered in Windows's command prompt window):

```
cfimager-imx -o 0 -f mx6dq_evb_rev_a-pcie-sdk.bin -d g:(SD drive name in your PC)
```

Then power-up the board to run the test.

To test the driver, connect a PCIe device, such as a PCIe Wi-Fi card, to the PCIe connector.

The test routine is as follows:

```
int main(void)
{
          //Initialize the PCIE controller and phy and then link up the PCIE endpoint device.
          pcie_init(PCIE_DM_MODE_RC);
          //Verify the PCIE phy's JTAG ID.
          …
          //Remap the endpoint's configuration space to ARM's memory space
. uint32_t cfg_hdr_base = pcie_map_space(PCIE_IATU_VIEWPORT_0,
                                          TLP_TYPE_CfgRdWr0,
                                          PCIE_ARB_BASE_ADDR, 0, SZ_64K);
          //Enumerate endpoint's resource.
          pcie_enum_resources((uint32_t *) cfg_hdr_base, ep1_resources, &res_num);
          //Allocate resources for the endpoint.
pcie_cfg_ep_bar(cfg_hdr_base, bar, *ep_base, ~(size - 1));
          ep1_resources[i].base = pcie_map_space(viewport, tlp_type, base_cpu_side,
*ep_base,
size);
          …
          Write-then-read the endpoint's first space to verify the access to the endpoint
}.
```

Chapter 27
Configuring the PWM driver

27.1 Overview

The pulse width modulator (PWM) generates various modulated waveforms with a specified pulse width and duty cycle. This chapter provides an overview of how to write the device driver for the PWM controller.

This chip has four instances of the PWM controller, which are located in the memory map at:

- PWM1 base address - 0208 0000h
- PWM2 base address - 0208 4000h
- PWM3 base address - 0208 8000h
- PWM4 base address - 0208 C000h

The PWM has only one external signal: ipp_do_pwmo. This signal is a clock signal whose period and duty cycle vary based on the different PWM settings. For full details, see the external signals table in the "Pulse Width Modulation (PWM)" chapter in the reference manual.

27.2 Feature summary

The PWM has the following features:

- 16-bit up-counter with clock source selection
- 4 x 16 FIFO to minimize interrupt overhead
- 12-bit prescaler for division of clock
- Sound and melody generation
- Active-high or active-low configured output
- Programmeable for active in low power and debug modes
- Interrupts at compare and rollover

27.3 Clocks

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

- For PWM1, set CG8 (bits 16-17) of CCM_CCGR4.
- For PWM2, set CG9 (bits 18-19) of CCM_CCGR4.
- For PWM3, set CG10 (bits 20-21) of CCM_CCGR4.
- For PWM4, set CG11 (bits 22-23) of CCM_CCGR4.

According to the PWM controller configuration, the clock can be selected from one of three sources: CKIL, CKIH, and IPG_CLK.

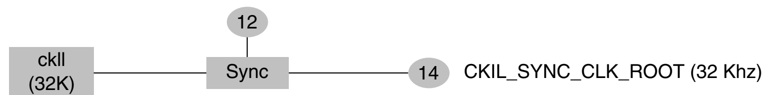


Figure 27-1. CKIL clock source

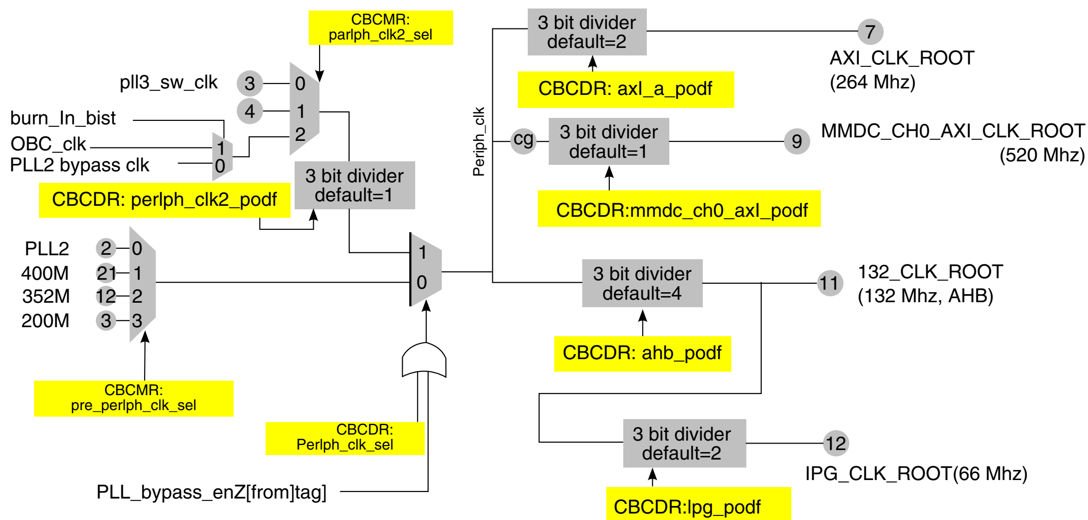


Figure 27-2. IPG_CLK clock source

The CKIH clock source is the external oscillator (22 MHz-27 MHz)

27.4 IOMUX pin mapping

The PWM driver only requires configuration of one external signal: PWM0. Because the PWMO is an output signal, it does not require an input source and daisy chain configuration is not necessary.

Table 27-1. PWM IOMUX pin mapping

Signals	PAD	MUX	SION
PWM1_PWMO	DISP0_DAT8	ALT2	0
	GPIO_9	ALT4	0
	SD1_DAT3	ALT3	0
PWM2_PWMO	DISP0_DAT9	ALT2	0
	GPIO_1	ALT4	0
	SD1_DAT2	ALT3	0
PWM3_PWMO	SD1_DAT1	ALT2	0
	SD4_DAT1	ALT2	0
PWM4_PWMO	SD1_CMD	ALT2	0
	SD4_DAT2	ALT2	0

27.5 Resets and interrupts

The general interrupt controller (GIC) supports the PWM interrupt. The IRQ IDs are as follows:

- ID115-PWM-1
- ID116-PWM-2
- ID117-PWM-3
- ID118-PWM-4

See the "ARM Domain Interrupts Summary" table in the "Interrupts and DMA Events" chapter in the reference manual for the full description of the PWM interrupt source.

The PWM interrupt supports three additional types of interrupts-CIE, RIE, and FIE-according to the value of the fields in PWMIR as below:

Table 27-2. Interrupt summary

Field Name	Description	Enable	Disable
CIE	Interrupt generated when compare event happens	1	0
RIE	Interrupt generated when rollover event happens	1	0
FIE	Interrupt generated when sample FIFO is empty	1	0

Clear the respective status bit (CMP, ROV, or FE) in PWMSR in the interrupt service routine to avoid the redundant interrupts. See the "PWM Status Register" status in the "Pulse Width Modulation (PWM)" chapter in the reference manual for the full description of the status bits.

27.6 Initializing the driver

The necessary initialization process can be summarized as:

1. Pin-mux configuration
2. Clock configuration and controller initialization
3. Controller ready to enable the output

To configure the PWM output, configure the following registers:

- PWMCR - configures clock source, pre-scale
- PWMSAR - configures samples into PWM FIFO
- PWMPR - configures the period of the output waveform

For details about the registers, see the "Programmable Registers" section in the "Pulse Width Modulation (PWM)" chapter in the reference manual.

27.6.1 Configuring the PWM output

The PWM controller generates modulated waveforms with different pulse width and duty cycle according to the values of configuration registers.

27.6.1.1 Generating the pulse width

The pulse width is generated according to the following equation:

Pulse width = (period + 1) ÷ PCLK = (period + 1) x prescale ÷ Fsrc

The following figure shows the source selection and division of the input clock.

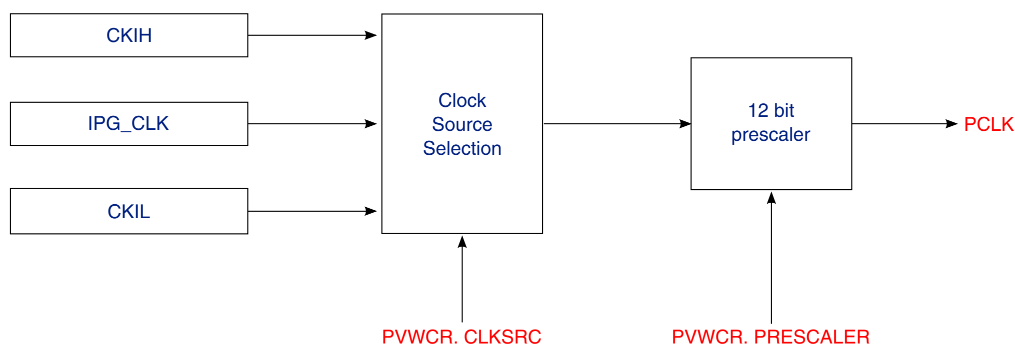


Figure 27-3. Input clock flow chart

27.6.1.2 Generating the duty cycle

PWMCR[POUTC] controls the PWM output voltage based on the following settings:

- 00, output pin is set at a rollover event and cleared at a comparison event.
- 01, output pin is cleared at a rollover event and set at a comparison event.
- 10 and 11, PWM output is disconnected.

A comparison event means that the incremented counter is equal to the sample value, and a rollover event means that the counter is equal to period + 1. Therefore, the following equations provide the duty cycle:

Duty cycle = sample ÷ (period + 1), if PWMCR.POUTC == 00b

Duty cycle = 1 - sample ÷ (period + 1), if PWMCR.POUTC == 01b

27.6.2 Enabling PWM output

Setingt PWMCR[EN] enables the PWM output and clearing PWMCR[EN] disables the PWM output.

27.7 Application program interface

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

Table 27-3. PWM APIs

API	Description	Parameters	Return
int **pwm_init**(struct hw_module *port, uint16_t freq, uint16_t prd, uint16_t *smp, uint16_t cnt);	Initializes the PWM controller that is specified by the device with the specified parameters	*port*: PWM instance *freq*: frequency pre-scale *prd*: period of pulse width *smp*: sample list *cnt*: number of samples in list	• TRUE on success • FALSE on fail
void **pwm_setup_interrupt** (struct hw_module *port, uint8_t enable, uint8_t mask);	Enables or disables the PWM interrupt	*port*: PWM instance *enable*: enable or disable *mask*: interrupt mask, could be FIE, CIE, RIE or ORed of them	-

Table continues on the next page...

Table 27-3. PWM APIs (continued)

API	Description	Parameters	Return
void **pwm_clear_int_status** (struct hw_module *port, uint32_t mask);	Clears the respective bits of the PWMSR register	*port*: PWM instance *mask*: status mask, could be FIE, CIE, RIE or ORed of them	-
void **pwm_enable**(struct hw_module *port);	Enables the PWM output	*port*: PWM instance	-
void pwm_disable(struct hw_module *port);	Disables the PWM output	*port*: PWM instance	-

Chapter 28
Using the SATA SDK

28.1 Overview

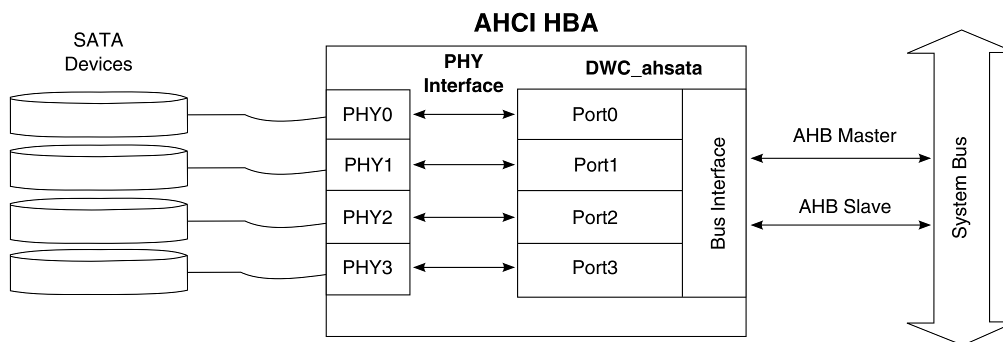


Figure 28-1. SATA system block diagram

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.

28.2 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

28.3 Modes of operation

Table 28-1. Modes of operation

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

28.4 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

28.5 IOMUX pin mapping

Table 28-2. SATA pin mapping

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

28.6 Resets and Interrupts

SATA IRQ number is 71.

The SDK did not implement an interrupt mode.

28.7 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*/
}
```

28.8 Testing the driver

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

Chapter 29
Configuring the SDMA Driver

29.1 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 an engineering sample board's schematics for pin assignments. For other board types refer to the respective schematics.

There is one instance of SDMA, which is located in the memory map at the SDMA base address of 020E C000h.

29.2 IOMUX pin mapping

Configure the IOMUX for SDMA into the **iomux_config()** function located in `./src/mx6dq/hardware.c`.

Table 29-1. SDMA IOMUX pin assignments

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
SDMA_EXT_EVENT[1]	GPIO_18	ALT3	1

Table continues on the next page...

Table 29-1. SDMA IOMUX pin assignments (continued)

Signal	IOMUXC Setting for SDMA		
	PAD	MUX	SION
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
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

Table continues on the next page...

Table 29-1. SDMA IOMUX pin assignments (continued)

Signal	IOMUXC Setting for SDMA		
	PAD	MUX	SION
DEBUG_EVENT_CHANNEL[3]	DISP0_DAT10	ALT4	1
DEBUG_EVENT_CHANNEL[4]	DISP0_DAT11	ALT4	1
DEBUG_EVENT_CHANNEL[5]	DISP0_DAT12	ALT4	1

29.3 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.

29.4 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.

29.5 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:

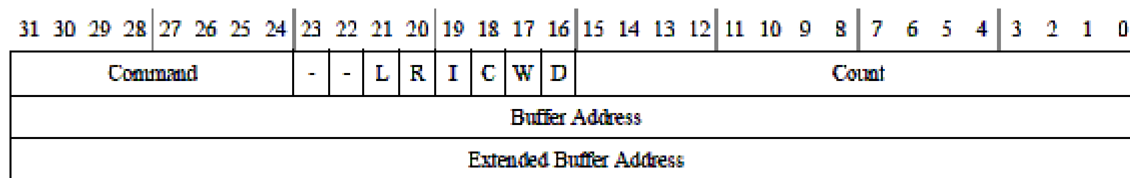


Figure 29-1. Buffer descriptor format

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.

29.6 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

29.7 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 s**dma_channel_stop**.
9. **sdma_deinit** provides a way to re-initialize the SDMA together with the **sdma_init**.

Chapter 30
Configuring the SPDIF Driver

30.1 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 an engineering sample board's schematics for pin assignments. For other board types, refer to the respective schematics.

30.2 Feature summary

The SPDIF driver supports:

- A simple framework for audio
- SPDIF APIs

30.3 Clocks

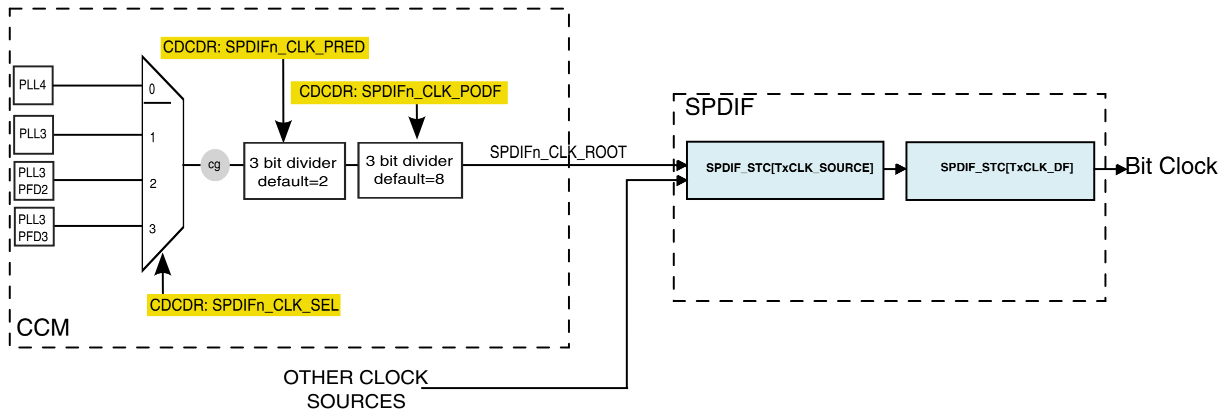


Figure 30-1. SPDIF clock tree (default)

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.

30.4 IOMUX pin mapping

The following table lists the IOMUX configurations based on an engineering sample board as an example. For other boards, refer to the appropriate board schematics for correct pin assignments.

Table 30-1. IOMUX pin mapping for SPDIF

Signals	PAD	MUX
SPDIFIN	KEY_COL3	ALT6
SPDIFOUT	GPIO_19	ALT2

30.5 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 CS42888)

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.

30.5.1 `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,
};
```

30.5.2 `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;
```

30.5.3 `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;
```

30.5.4 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;
```

30.5.5 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;
```

30.6 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

30.6.1 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
 */
```

30.6.2 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
 */
```

30.6.3 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

30.6.4 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)
```

30.6.5 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)
```

30.6.6 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)
```

30.6.7 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
 */
```

30.7 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 following command:

```
./tools/build_sdk -target mx6dq -board evb -board_rev a -test audio
```

This generates the following ELF and binary files:

- `output/mx6dq/evb_rev_a/bin/mx6dq_evb_rev_a-audio-sdk.elf`
- `output/mx6dq/evb_rev_a/bin/mx6dq_evb_rev_a-audio-sdk.bin`

To run the test:

1. Download `mx6dq_evb_rev_a.elf` using RV-ICE or Lauterbach or burn `mx6dq_evb_rev_a.bin` to an SD card with the following command (entered in the Windows's command prompt window):

   ```
   cfimager-imx -o 0 -f mx6dq_evb_rev_a-audio-sdk.bin -d g: (SD drive name in your PC)
   ```
2. Ensure the following:
   a. The board is mounted on the MX6QVPC board.
   b. A rework was done to connect TP6[SPDIF_OUT] with PORT2_P98 on the MX6QVPC board.
   c. 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.

Chapter 31
Using the SNVS RTC/SRTC Driver

31.1 Overview

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.

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 ensure 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)

The single SNVS module is located on the memory map at: 020C C000h.

Refer to the SNVS chapter of the device reference manual for the description of SNVS HP and LP registers and further documentation of the SNVS module.

31.2 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)

31.3 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.

31.4 Clocks

Table 31-1. 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.

31.5 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.

31.5.1 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.

31.5.1.1 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.

31.5.1.2 Non-Secured Real Periodic Interrupt

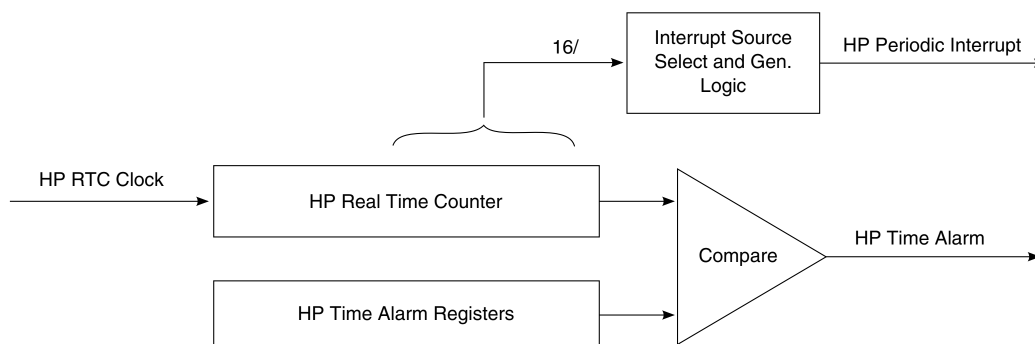


Figure 31-1. SNVS_HP real time counter, alarm, and interrupts

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 31-1 2.

31.5.2 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.

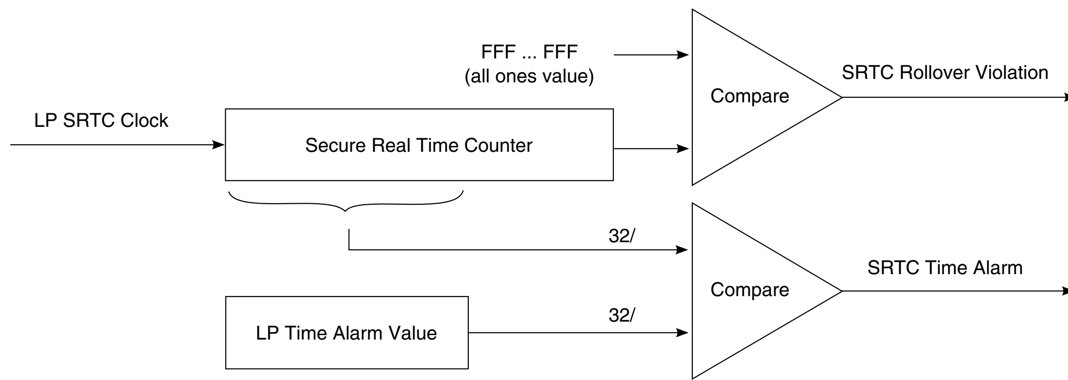


Figure 31-2. SNVS_LP secure real time counter

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.

31.6 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

31.6.1 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.

31.6.1.1 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);
    }
}
```

31.6.1.2 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);
    }
}
```

31.6.1.3 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);
    }
}
```

31.6.1.4 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);
}
```

31.6.1.5 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);
}
```

31.6.1.6 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);
    }
}
```

31.6.1.7 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);
    }
}
```

31.6.1.8 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);
}
```

31.6.1.9 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);
}
```

31.6.2 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.

31.6.2.1 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);
}
```

31.6.2.2 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);
}
```

31.6.2.3 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);
}
```

31.6.2.4 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);
}
```

31.6.2.5 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);
}
```

31.6.3 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.

31.6.4 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);
}
```

31.6.5 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);
}
```

31.6.6 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);
}
```

31.6.7 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;
}
```

Chapter 32
Configuring the SSI Driver

32.1 SSI 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.

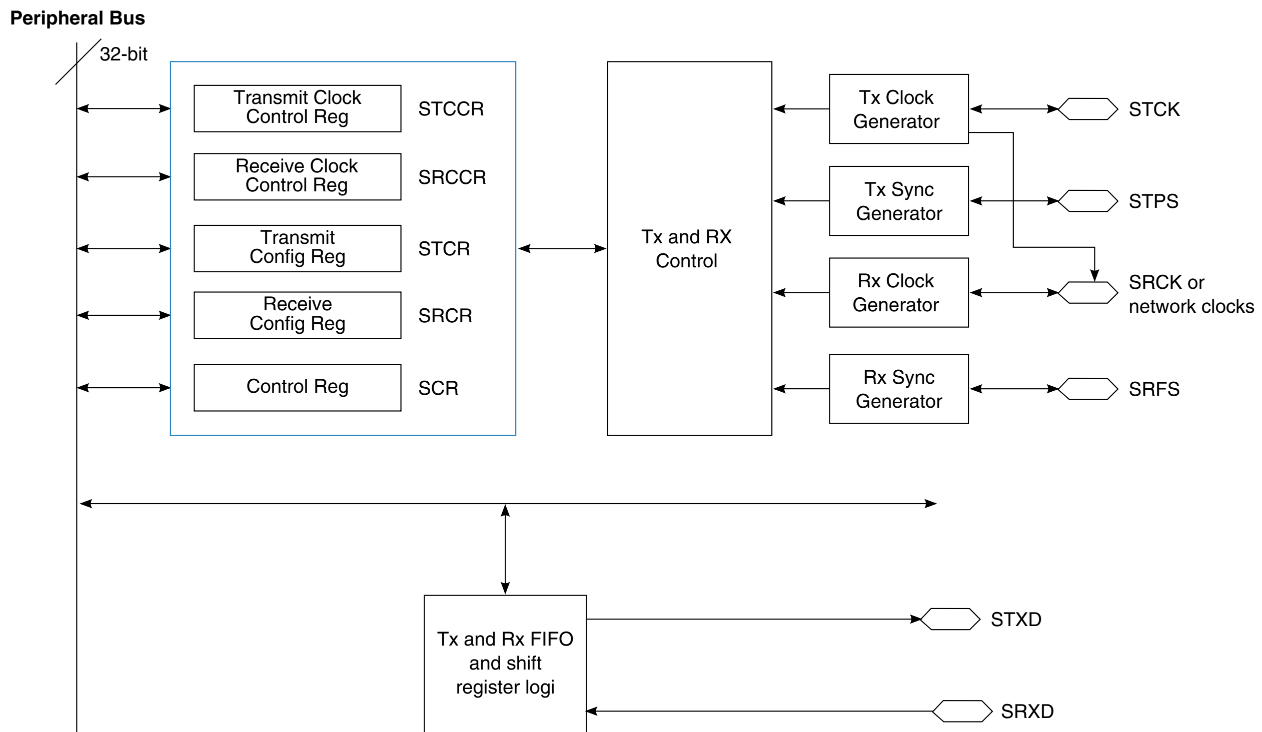


Figure 32-1. SSI block diagram

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 chip: SSI1, SSI2, and SSI3. The SSI signals connect to the AUDMUX, which is another module within the chip. For AUDMUX details, see AUDMUX.

32.2 Feature summary

The SSI driver supports the following features:

- A simple framework for audio
- SSI driver supporting I2S protocol
- sgtl5000 driver supporting I2S protocol

32.3 Clocks

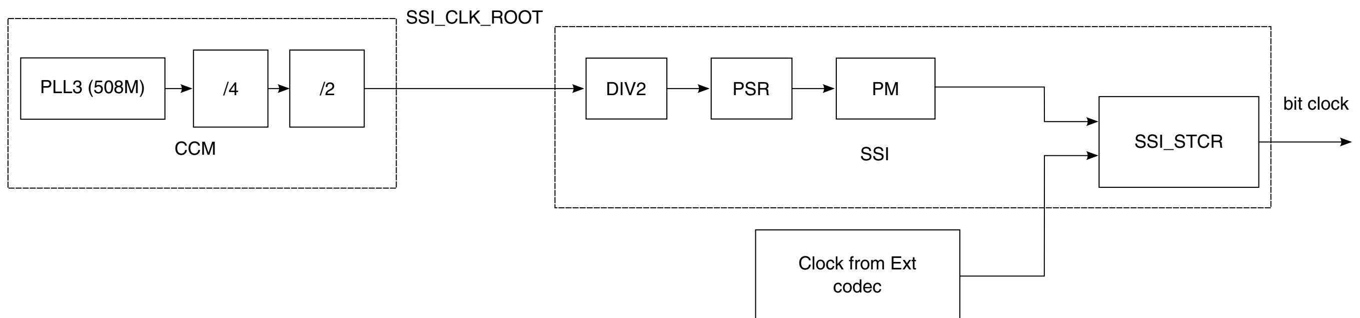


Figure 32-2. SSI clock tree

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 chip 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.

32.4 IOMUX pin mapping

The following figure shows the SSI signal routing.

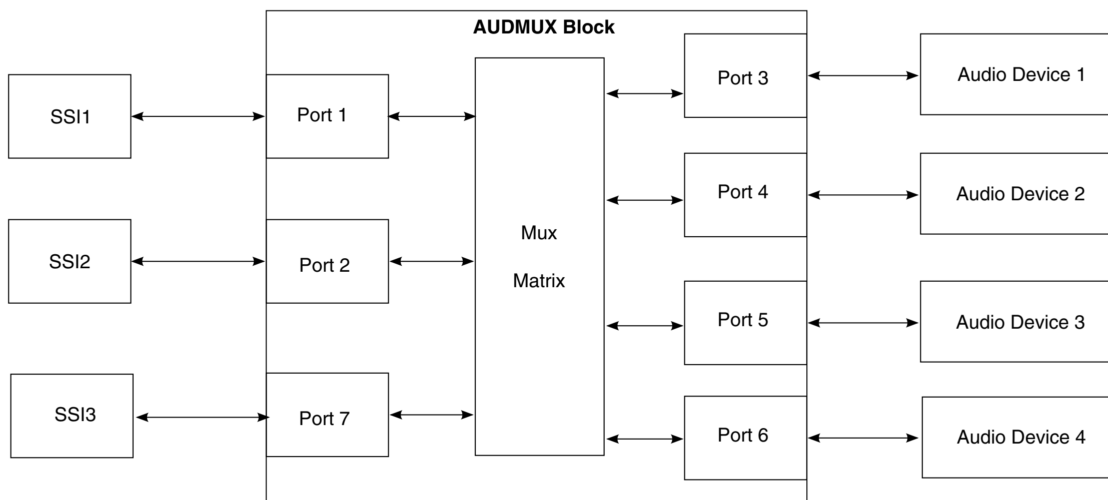


Figure 32-3. SSI signal routing

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.

The following table is based on an engineering sample board in which PORT5 was connected with the SSI codec sgtl5000 in SYN mode. When using another board, check the board schematic for the specific pin assignments.

Table 32-1. 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

32.5 Audio framework

The chip 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, or other)
- audio_codec_t—describes the audio codec (sgtl5000, CS428888, or other)

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

32.5.1 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;
```

32.5.2 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;
```

32.5.3 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;
```

32.5.4 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;
```

32.5.5 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;
```

The engineering sample board uses sgtl5000 and SSI2, 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

This section is based on an engineering sample board. Check your board schematic for the correct pin assignments.

32.6 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

32.6.1 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).

32.6.2 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.

32.6.3 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).

32.6.4 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;
```

32.6.5 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.

32.6.6 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

32.6.6.1 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).

32.7 sgtl5000 driver

The sgtl5000 is one of many codecs that have an SSI interface and thus can be used as an external audio codec. Discussion of the sgtl5000 is beyond the scope of this chapter. See the sgtl5000 driver for details.

32.8 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 following command:

```
./tools/build_sdk -target mx6dq -board smart_device -board_rev a -test audio
```

The command generates the following binary and ELF files:

- `output/mx6dq/smart_device_ai_rev_a/bin/mx6dq_smart_device_rev_a-audio-sdk.elf`
- `output/mx6dq/smart_device_ai_rev_a/bin/mx6dq_smart_device_rev_a-audio-sdk.bin`

After the files have been generated, perform the following steps:

1. Download `mx6dq_smart_device_rev_a-audio-sdk.elf` using RV-ICE or Lauterbach or burn `mx6dq_smart_device_rev_a-audio-sdk.bin` to SD card with the following command in Windows's Command Prompt window:

   cfimager-imx -o 0 -f mx6dq_smart_device_rev_a-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

This example test is based on an engineering sample board. Refer to your board's schematics for the correct pin assignments.

32.9 Functions

32.9.1 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)
```

32.9.2 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)
```

Chapter 33
Configuring the UART Driver

33.1 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.

33.2 Feature 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

33.3 Modes of operation

The following table explains the UART modes of operation:

Table 33-1. 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.

33.4 Clocks

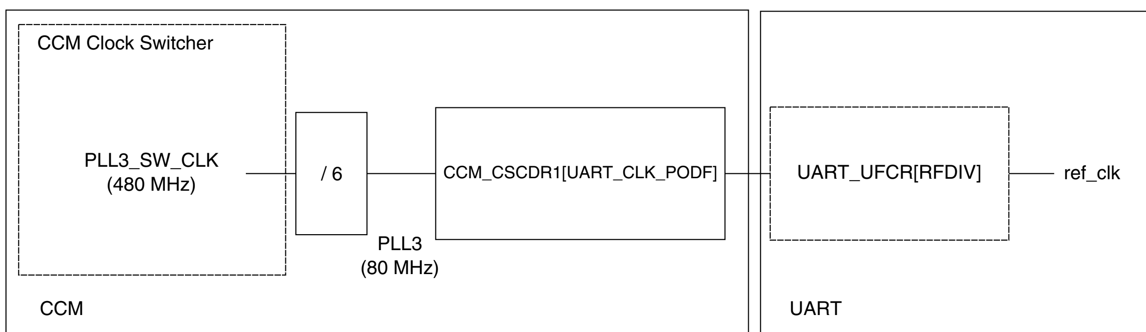


Figure 33-1. UART reference clock

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.

33.5 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 using the **iomux_config()** function located in `./src/mxdq/iomux/board_name/uart_iomux_config.c`.

33.6 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/mx6dq/soc_memory_map.h.

33.7 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)
```

33.8 Testing the UART driver

The UART driver runs the following tests:

- Echo test
- SDMA test

33.8.1 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.

33.8.2 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.

33.8.3 Running the UART test

To run the UART tests, the SDK uses the following command to build the test:

```
./tools/build_sdk -target mx6dq -board sabre_ai -board_rev a -test uart
```

This command generates the following ELF and binary files:

- `./output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-uart-sdk.elf`
- `./output/mx6dq/sabre_ai_rev_a/bin/mx6dq_sabre_ai_rev_a-uart-sdk.bin`

Chapter 34
Configuring the USB Host Controller Driver

34.1 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

34.2 Feature summary

This low-level driver demonstrates the configuration and basic functionality of the USB controller. It supports:

- Initialization of controllers and basic data structures
- Initialization of the PHY and clocks

- Host-side device enumeration
- Device-side device enumeration
- Control transfers
- Low-level bulk transfers
- Low-level interrupt transfers

34.3 Modes of operation

The OTG controller can operate as either 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).

34.4 Clocks

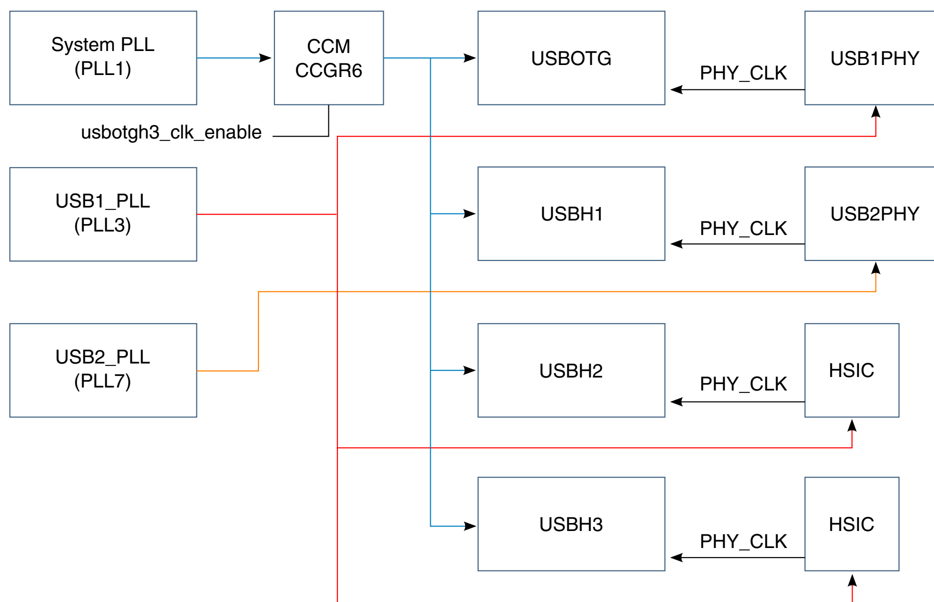


Figure 34-1. USB module clocks

The USB module uses three independent clocks: one shared clock for the control logic and DMA transfers and 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.

34.5 IOMUX pin mapping

The pin descriptions in this section apply to the SABRE for Automotive Infotainment based on the i.MX 6 series. 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 34-1. 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 for Automotive Infotainment based on the i.MX 6 Series design.

34.6 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.

Each controller has a single interrupt vector in the vector table. The UTMI transceivers each have an interrupt vector in the main vector table. Vector numbers are assigned as indicated in the following table.

Table 34-2. Vector numbers

ID number	Description
ID72	USB host1 interrupt
ID73	USB host2 interrupt
ID74	USB host3 interrupt
ID75	USB OTG interrupt
ID76	USBPHY (UTMI0) interrupt
ID77	USBPHY (UTMI1) interrupt

Each USB provides control over its interrupt sources through its USBINTR and USBSTS registers.

- Interrupt request flags are located in the USBSTS register.
- Each individual source can be enabled or disabled for interrupt generation in the USBINTR register.

The controller also allows adjustment of the maximum rate at which the controller can issue interrupts. The interrupt rate can be programmed in the ITC field of the USBCMD register. Values range from immediate through 1 interrupt per 64 micro frames (8 ms).

This driver does not use interrupts. Instead, it polls the interrupt flag where an interrupt is expected.

34.7 Initializing the host driver

The driver's API contains initialization 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 generates 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 synchronous 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.

34.8 Initializing the device driver

Like the host driver, the device driver has an init routine that enables the clock and PHY and also configures the OTG controller for device operation. The routine initializes the device endpoint list, creates the IN and OUT queue heads for endpoint 0, and starts the controller.

The driver's init routine performs the following steps:

1. Disable Vbus power – Devices are not allowed to drive Vbus.
2. Start clocks
3. Enable transceiver
4. Reset controller
5. Set controller's mode to device mode
6. Set up endpoint list
7. Configure endppoint 0
8. Start controller

The controller is now running and ready to detect the USB bus reset.

34.9 Testing the host mode

The host mode has two applications for testing: usbh0_host_test and usbh0_host_testmodes_test.

Test application usbh0_host_test waits for a device to connect and enumerates the device. If the device is a mouse, it polls the interrupt endpoint for mouse movement data. Clicking the right-mouse button ends the test application.

Test application usbh0_host_testmodes_test shows how to configure the EHCI test modes on a host controller. These test modes are used for electrical measurements in high-speed mode. The application initializes the clocks, controller, and PHY and then presents a menu on a terminal connected to the UART port where the user can select the test mode. Supported modes are:

- Test_J: forces a J state on the port t for DC measurements.
- Test_K: forces a K state on the port.
- Test_SE0/NAK: forces SE0 on the port.
- Test_packet: sends out the test packet for eye diagram measurements.
- Suspend: suspends the bus to measure suspend timing.
- Resume: resumes the bus to measure resume timing.
- Reset: sends reset on the USB bus to measure reset timing.

34.10 Testing the device mode

The device mode application emulates a mouse peripheral. When connected to a PC, it sends mouse movement data to make the cursor move in a circle. This application demonstrates the following:

- How to set up the device controller
- Provide enumeration responses
- How to add transfers on an active pipe

The application also supports setting test modes on the device controller for electrical measurements. It responds to commands sent by the USBHSET tool, which is available from the USB-IF web site (http://www.usb.org/developers/tools/).

34.11 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.

```
/*!
 * This function enables the clocks needed for USB operation.
 * @param port
 * @return
 */
int usbEnableClocks(usb_module_t *port)

/*!
 * Enable USB transceiver\n
 * This function enables the USB transceiver for the selected USB port.
 *
 * @param port      USB module to initialize
 */
int usbEnableTransceiver(usb_module_t *port)

/*!
 * This function enables Vbus for the given USB port\n
 * The procedure to enable Vbus depends on both the Chip and board hardware\n
 * This implementation is for the SABRE for Automotive Infotainment based on the i.MX 6
Series.\n
 *
 * @param port      USB module to initialize
 */
void usbEnableVbus(usb_module_t *port)

/*!
 * This function disables Vbus for the given USB port\n
 * The procedure to enable Vbus depends on both the Chip and board hardware\n
 * This implementation is for the SABRE for Automotive Infotainment  based on the i.MX 6
Series\n
 *
 * @param port      USB module to initialize
 */
void usbDisableVbus(usb_module_t *port)
```

34.12 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 schedule.
```

```
 *
 * @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)

/*!
 * Enable the asynchronous schedule\n
 * This function enables the Asynchronous schedule.\n
 * The application code must create descriptors and queue heads and\n
 * set the Asynchronous list address before calling this function.
 */
void usbh_enable_asynchronous_schedule(usb_module_t *port)

/*!
 * Disable the asynchronous schedule.
 *
 * @param port  USB module
 */
void usbh_disable_asynchronous_schedule(struct usb_module *port)
            uint32_t *queue head)

/*!
* Disable the periodic list.
*
* @param port  USB module
*/
void usbh_disable_Periodic_list (struct usb_module *port)


/*!
 * Initialize the QH.
 * 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       used to mark the QH as the first in the linked list (not used 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)
 */

usbhQueueHead_t * usbh_qh_init(uint32_t max_packet, uint32_t head, uint32_t eps, uint32_t
epnum, uint32_t dev_addr, uint32_t smask)

/*!
 * Issue a USB reset to the specified port.
 *
 * @param port      USB module to send reset
 */
void usbh_bus_reset(struct usb_module *port)

/*!
 * Initialize the qTD.
 * This function initializes a transfer descriptor.
 * the next qTD and alternate next qTD pointers are initialized with the terminate bit set.
 *
 * @param transferSize   number of bytes to be transferred
 * @param ioc            interrupt on complete flag
 * @param pid            PID code for the transfer
 * @param bufferPointer  pointer to the data buffer
 */
usbhTransferDescriptor_t * usbh_qtd_init(uint32_t transferSize, uint32_t ioc, uint32_t pid,
uint32_t *bufferPointer)
```

34.13 USB device API

```
//! Function to initialize the USB controller for device operation.
/*! This initialization performs basic configuration to prepare the device for connection to
a host.
 *
 * @param port           The USB module to use
 * @param endpointList   pointer to list with endpoint queue heads
 */
uint32_t usbd_device_init(usb_module_t *port, usbdEndpointPair_t *endpointList)

//! Function to initialize the controller after the USB bus reset
/*!
 * USB device response to a USB bus reset.
 *
 * @param portUSB controller to use
 * @returnreturns the operating speed of the port
 */
usbPortSpeed_t usbd_bus_reset(usb_module_t *port)

/*!
 * USB device function to return the data from a setup packet.
 * NOTE: We assume only endpoint 0 is a control endpoint
 *
 * @paramendpointListpointer to the device endpoint list address
 * @paramportpointer to controller info structure
 * @paramsetupPacketSetup data of the setup packet
 */
void usbd_get_setup_packet(usb_module_t *port, usbdEndpointPair_t *endpointList,
usbdSetupPacket_t *setupPacket)

//! Function to send an IN control packet to the host.
/*! NOTE: this function uses the default control endpoint (0).\n
 *       The endpoint number is hard-coded.
 *
 * @paramportController to use
 * @paramendpointListpointer to the device endpoint list
 * @parambufferData to be sent to host
 * @paramsizeAmount of data to be transferred in bytes
 */
void usbd_device_send_control_packet(usb_module_t *port, usbdEndpointPair_t *endpointList,
uint8_t* buffer, uint32_t size)

//! Function to send an zero length IN packet to the host.
 /*!
 * Zero Length packets are used as completion handshake in control transfers.\n
 * They can also be used to signal the end of a variable length transfer.\n
 *
 * @param portUsb controller to use
 * @param endpointListpointer to the device endpoint list
 * @param endpointNumberendpoint info data structure for the endpoint to use
 */
void usbd_device_send_zero_len_packet(usb_module_t *port, usbdEndpointPair_t *endpointList,
uint32_t endpointNumber)

//! Function to initialize an endpoint queue head
/*!
 * Initialize an endpoint queue head. The space for the endpoint queue heads was
 * allocated when the endpoint list was created, so this function does not
 * call malloc.
 *
 * @param endpointListlocation of the endpoint list
 * @param usbdEndpoint    Pointer to the endpoint characteristics
 * @param nextDtdpointer to the first transfer descriptor for the queue head
 */
```

```
void usbd_endpoint_qh_init(usbdEndpointPair_t *endpointList, usbdEndpointInfo_t
*usbdEndpoint, uint32_t nextDtd)

//! Function to create a new transfer descriptor
/*!
 * This functions allocate memory for a device transfer descriptor (dTD) and
 * initializes the dTD. This function assumes the dTD is the last in the list so
 * the next dTD pointer is marked as invalid.
 *
 * @param transferSizenumber of bytes to be transferred
 *@param interruptOnCompleteinterrupt on complete flag
 *@param multOverrideOverride the queue head multiplier setting (0 for default)
 *@param bufferPointerpointer to the data buffer
 *
 *@return pointer to the transfer descriptor
 *
 */
usbdEndpointDtd_t *usbd_dtd_init(uint32_t transferSize, uint32_t interruptOnComplete,
uint32_t multOverride, uint32_t *bufferPointer)

//! Function to add a transfer descriptor or a list of transfer descriptors to an active
endpoint
/*! This function places a new transfer on the linked list of transfer descriptors.\n
 * If the list was empty, the new transfer descriptor is placed on the queue head.
 *
 * @param portPointer to controller info structure.
 * @param usbdEndpointEndpoint
 *@param endpointListPointer to the endpoint list
 *@param new_dtdpointer to the descriptor to add
 *
 */
void usbd_add_dtd(usb_module_t *port, usbdEndpointPair_t *endpointList, usbdEndpointInfo_t
*usbdEndpoint, usbdEndpointDtd_t *new_dtd)

//! Function to reclaim used transfer descriptors.
/*! This function parses the list of transfer descriptors, starting\n
 * at the Head pointer and up to the currently active descriptor.\n
 * It removes retired descriptors from the list and returns memory used by the descriptor to
the heap.
 *
 * @param portPointer to controller info structure.
 * @param usbdEndpointEndpoint
 *@param endpointListPointer to the endpoint list
 *@param headpointer to the head of the list
 *
 *@returnPointer to the new list head.
 *
 */
usbdEndpointDtd_t *usbd_reclaim_dtd(usb_module_t *port, usbdEndpointPair_t *endpointList,
usbdEndpointInfo_t *usbdEndpoint, usbdEndpointDtd_t *head)
```

34.14 Source code and structure

Table 34-3. Source code file locations

Description	Location
Source files	
Host mode low-level driver	./src/sdk/usb/host/usbh_drvr.c
Device mode driver	./src/sdk/usb/drv/usbd_drv.c
Common routines	./src/sdk/usb/common/usb_common.c

Table continues on the next page...

Table 34-3. Source code file locations (continued)

Platform specific initialization	./src/sdk/usb/common/usb_mx61.c
Chip-specific USB registers	./src/include/mx61/usb_regs.h
Bit definitions	./src/include/usb_defines.h
USB controller registers	./src/include/usb_registers.h
Prototypes, structures, and enum	./ src/include/usb.h
Test programs	
USB test startup	./src/sdk/usb/test/usb_test.c
Host mode test	./src/sdk/usb/test/usbh_host_test.c
Host mode EHCI test modes	./src/sdk/usb/test/usbh_host_testmodes_test.c
Device mode test - mouse emulation	./src/sdk/usb/test/usbd_device_mouse_test.c

Chapter 35
Configuring the uSDHC Driver

35.1 Overview

This chapter provides a guide for firmware developers about how to write the device driver for the uSDHC controller. It uses an engineering sample board's 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 chip. They are located in the 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

35.2 Clocks

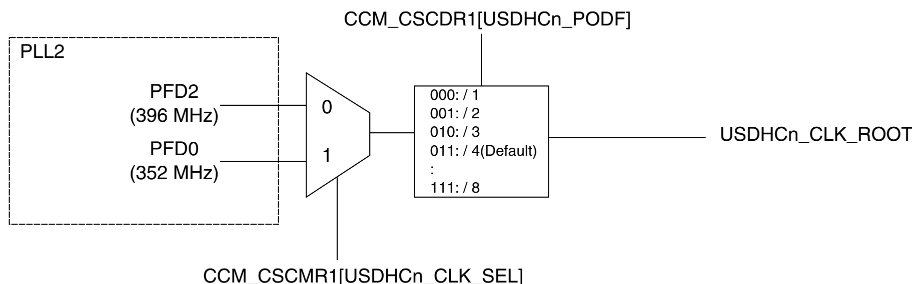


Figure 35-1. uSDHC clock tree

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 chip reference manual for more information about programming clocks.

35.3 IOMUX pin mapping

The following table is based on an engineering sample board and is shown as an example. Refer to your board's schematics for the board's specific pin assignments.

Table 35-1. 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 chip reference manual.

35.4 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.

35.4.1 Initializing the SD/MMC card

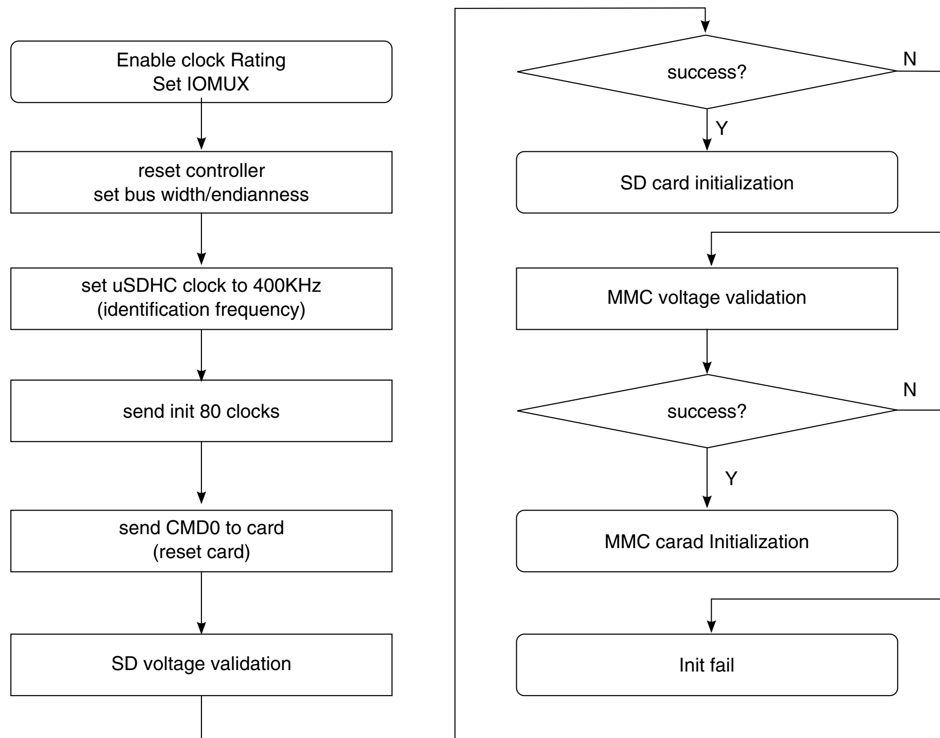


Figure 35-2. Initialization process flow chart

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

35.4.2 Frequency divider configuration

The following figure shows the flow chart for the frequency divider configuration process.

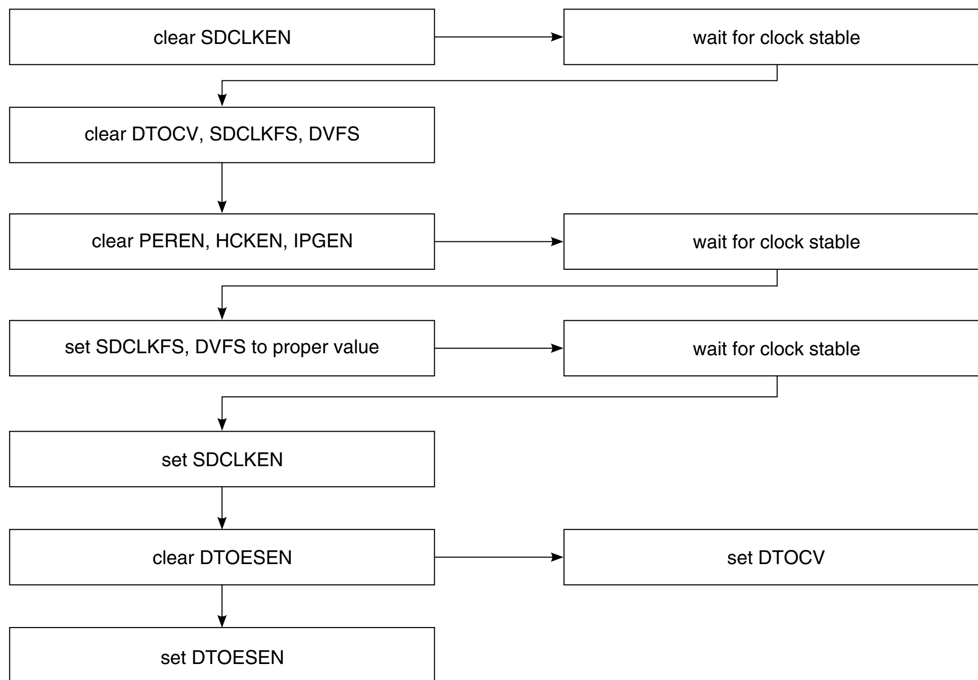


Figure 35-3. Frequency divider configuration process

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.

35.4.3 Send command to card flow chart

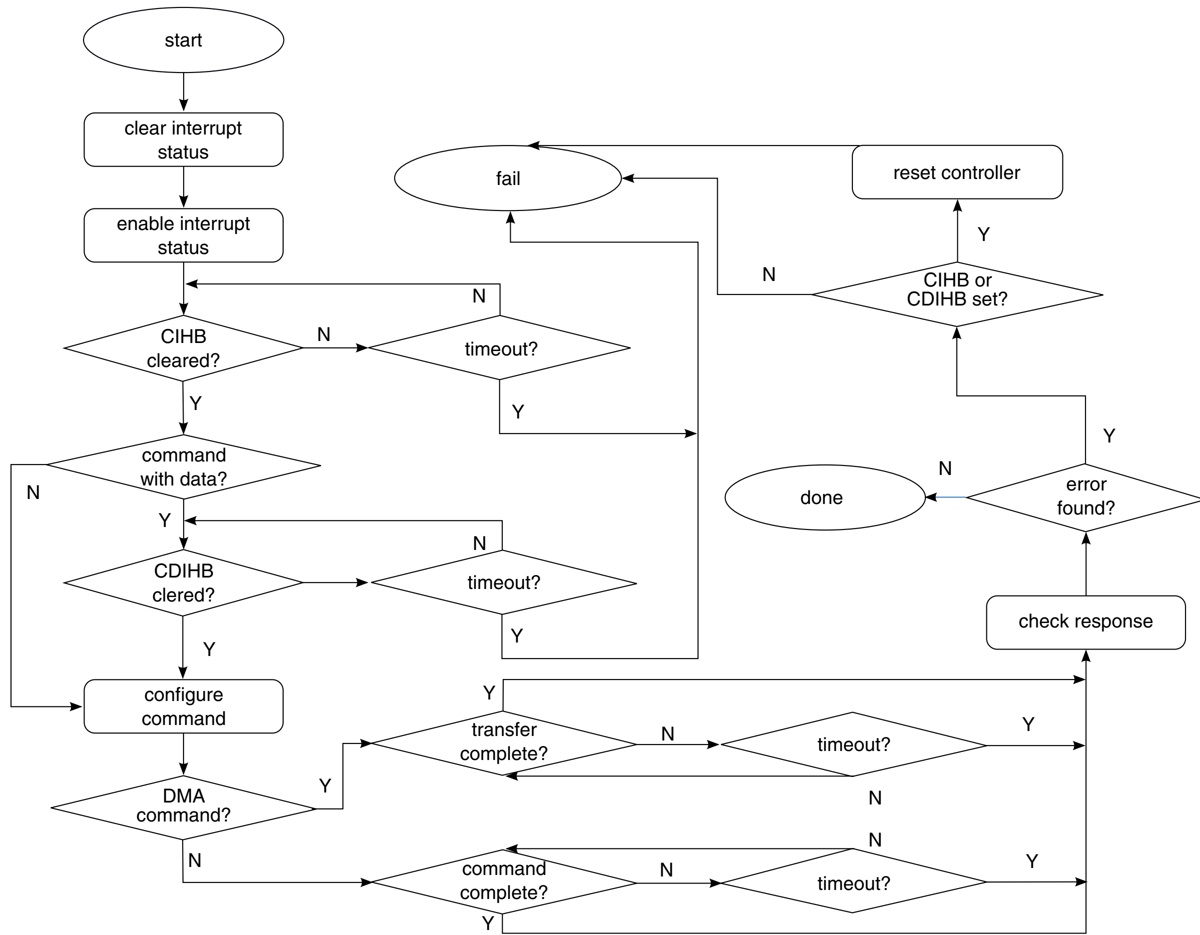


Figure 35-4. Send command to card flow chart

35.4.4 SD voltage validation flow chart

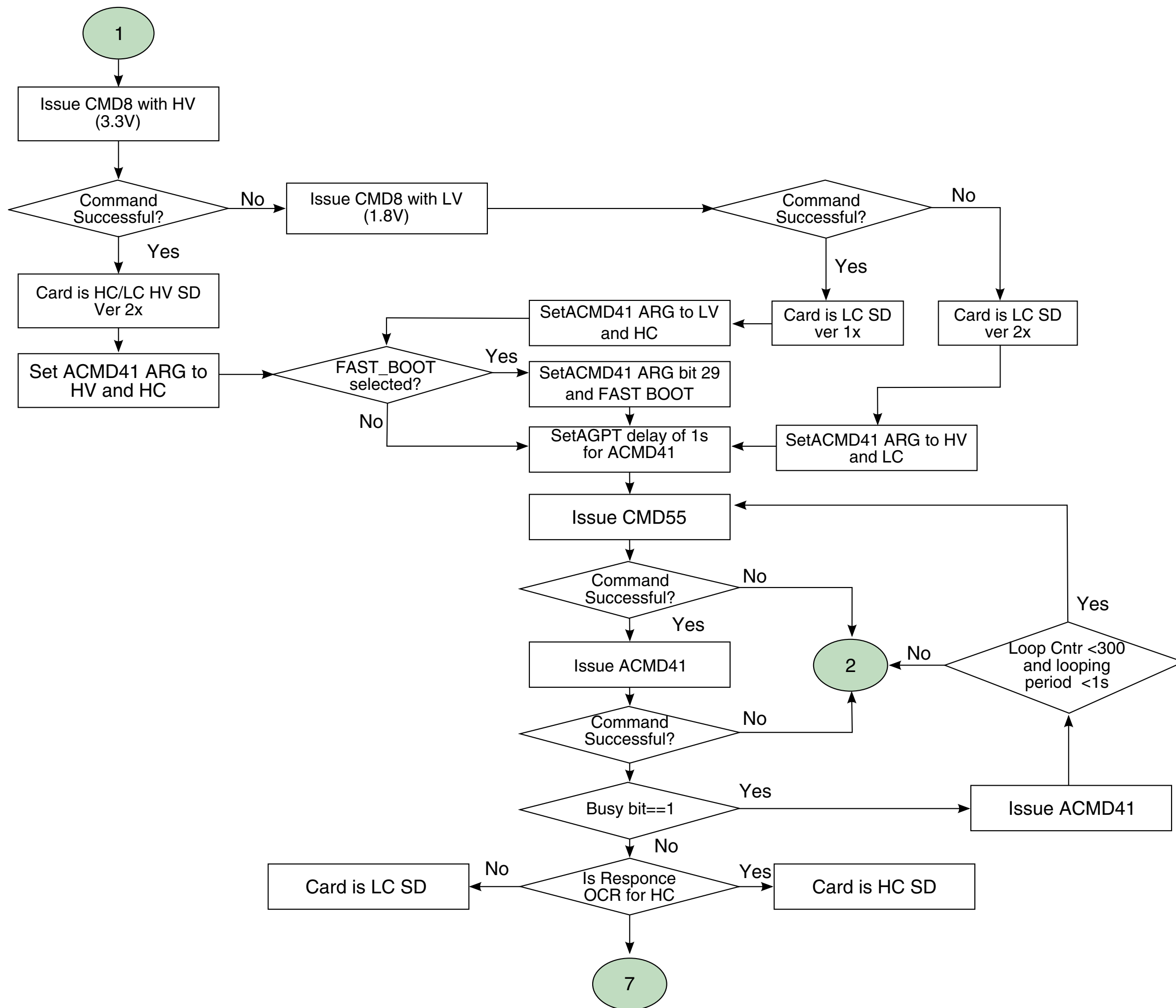


Figure 35-5. SD boot voltage validation flow chart

35.4.5 SD card initialization flow chart

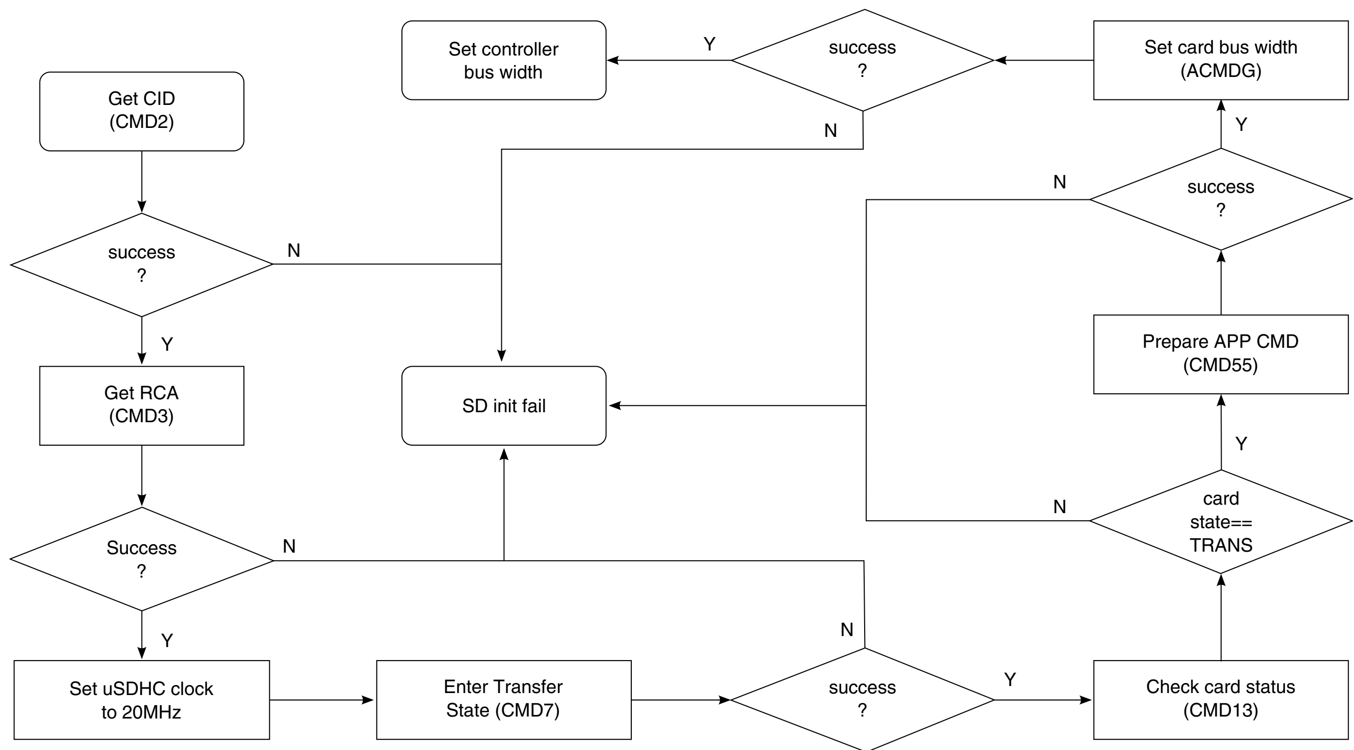


Figure 35-6. SD card initialization flow chart

35.4.6 MMC voltage validation flow chart

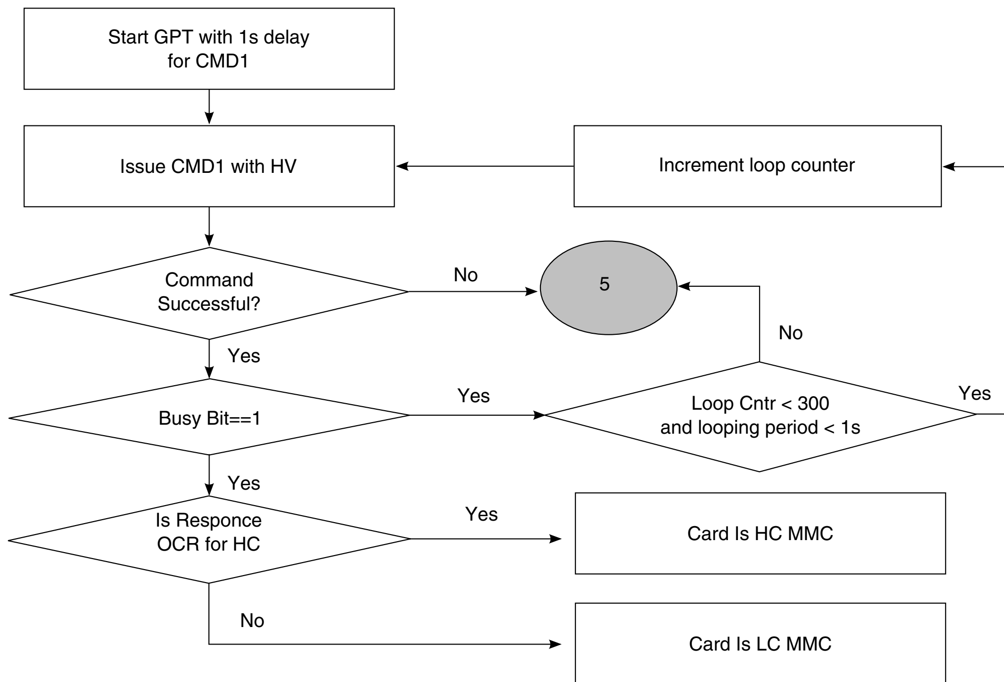


Figure 35-7. MMC voltage validation flow chart

35.4.7 MMC card initialization flow chart

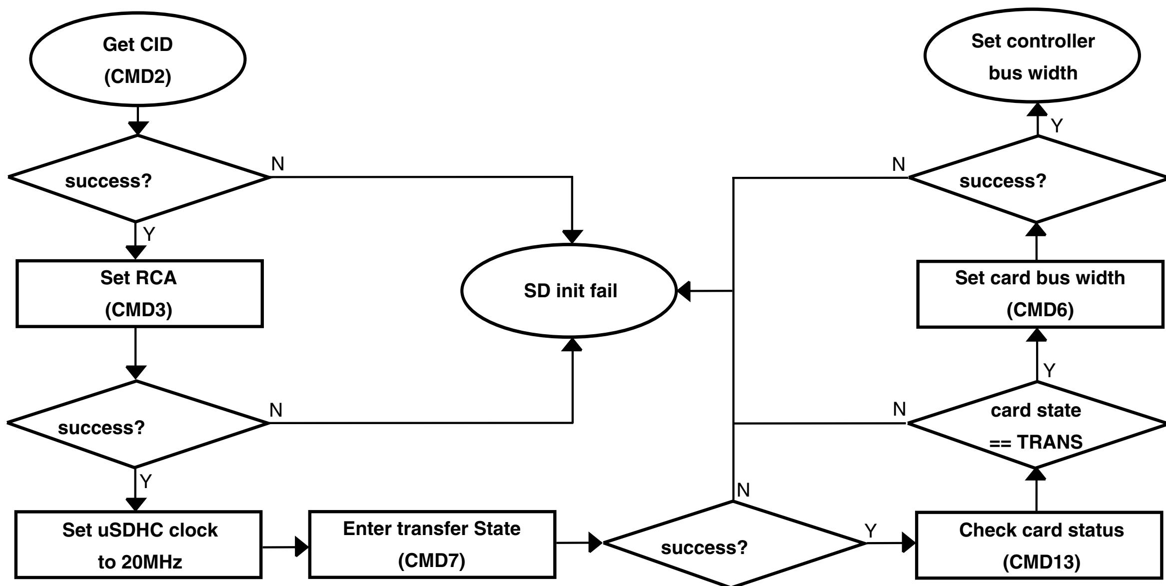


Figure 35-8. MMC card initialization flow chart

35.5 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.

35.5.1 Reading data from the card

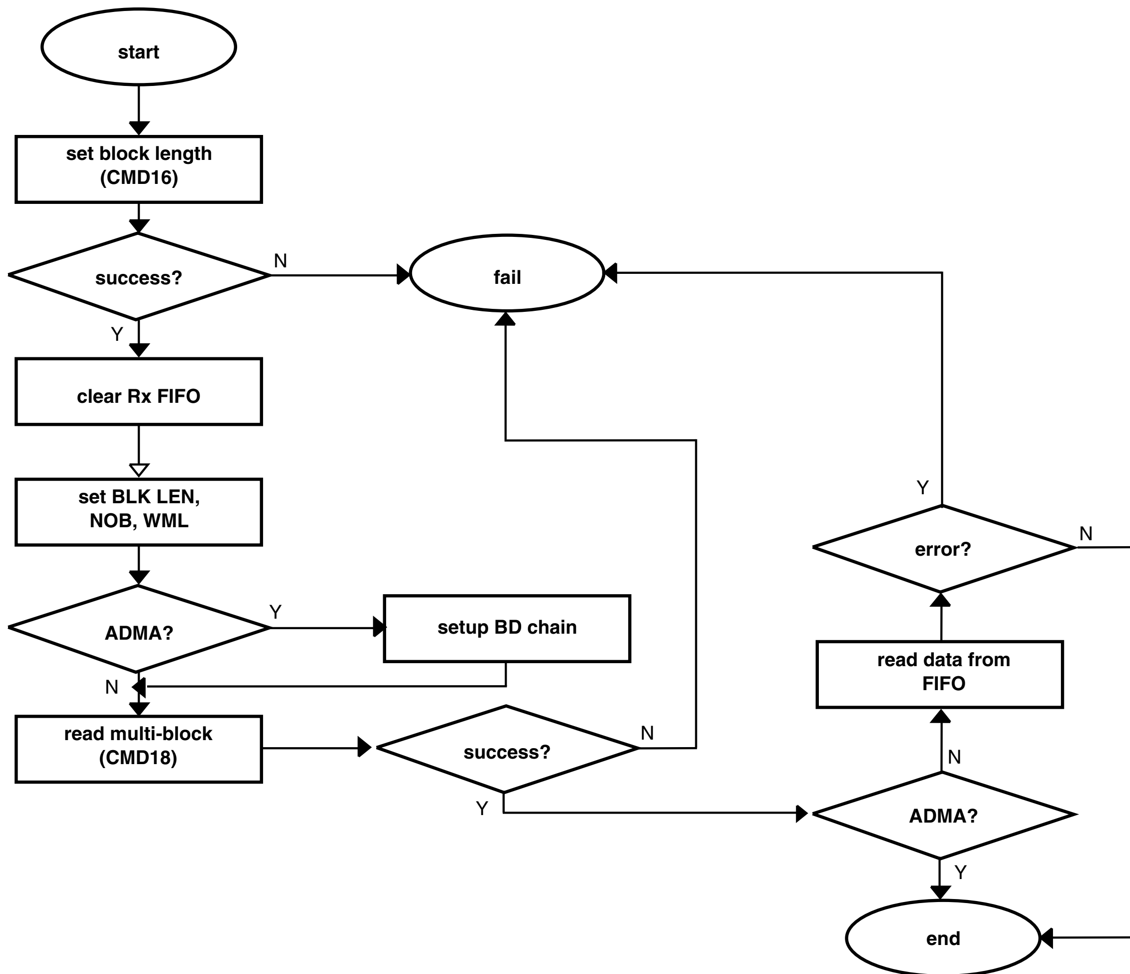


Figure 35-9. Reading data flow chart

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 chip reference manual.

35.5.2 Writing data to the card

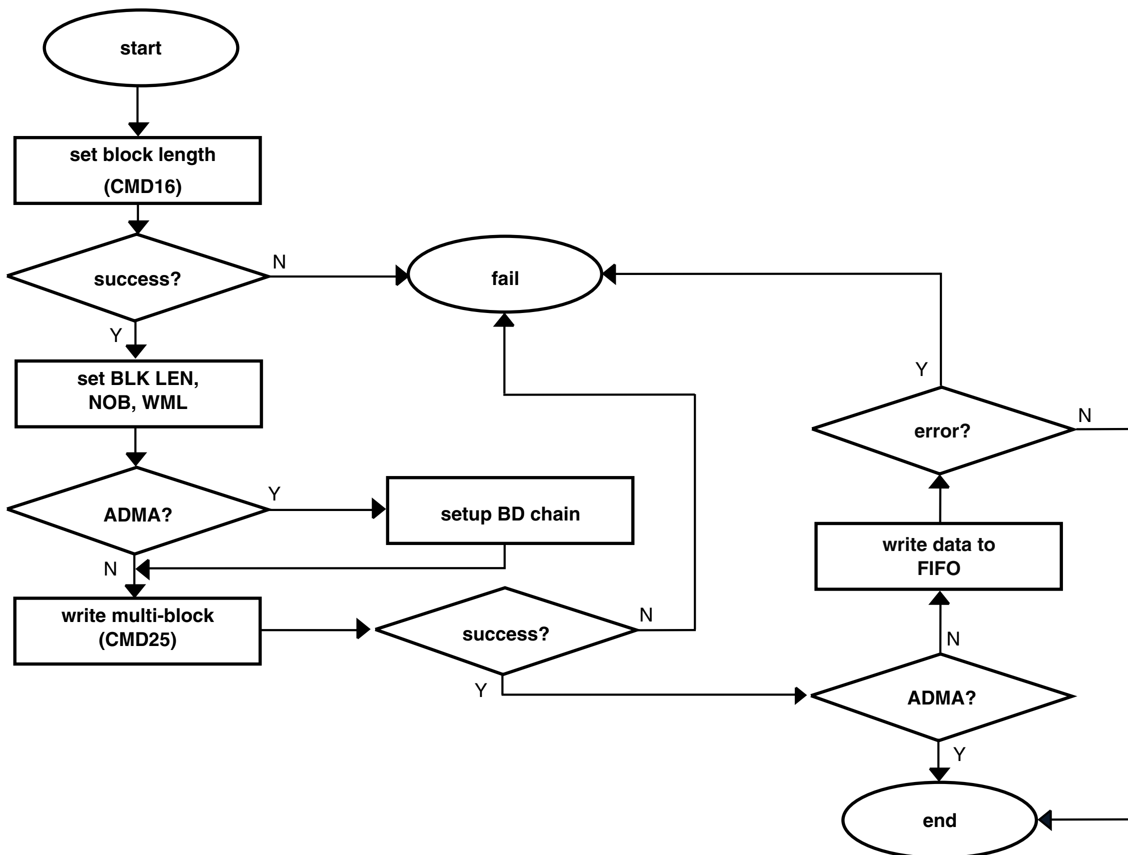


Figure 35-10. Writing data flow chart

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.

35.6 Application programming interfaces

All external function calls and variables are inside inc/usdhc_ifc.h:

35.6.1 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.

35.6.2 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.

35.6.3 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.

Chapter 36
Configuring the VDOA Driver

36.1 Overview

This chapter describes the video processing unit (VPU) for the i.MX 6Dual/6Quad and i.MX 6Solo/6DualLite products. The VPU module can output data in tiled mode to increase the decoding performance. However, the image processing unit (IPU) cannot post process the data layout in tiled mode. The video data order adapter (VDOA) is a DMA whose purpose is to convert the data from tiled mode to raster mode, which the IPU can accept.

The VDOA is located at 021E 4000h in the system memory map.

36.2 Feature summary

VDOA supports the following features:

- Data conversion from tiled to raster mode
- High resolution support for frame sizes of up to 8192 x 4096 pixels
- High speed
    - At 264 MHz, the burst size is 64 bits.
    - In YUV420 partial interleaved mode, the peak conversion speed is 3 pixels x 264 MHz, or 792 Mpixels.
- Support for data conversion of up to 3 buffers concurrently

36.3 Modes of operation

The VDOA supports two modes for conversion: sync or non-sync.

The VDOA uses sync mode to work with the IPU when it is in band mode, a mode in which the IPU divides a complete frame into bands for transfer. In sync mode, the VDOA handshakes with the IPU after a band of data has been transferred to notify the IPU to resume work. However, the IPU driver does not currently support band mode because the chip has enough memory that band mode provides no advantage over frame based mode.

In non-sync mode, the VDOA works on a frame basis. The VPU output is fixed to YUV420 partial interleaved mode, which then serves as the input for the VDOA. The VDOA output serves as the input for the IPU and can be either YUV422 interleaved mode or YUV420 partial interleaved mode.

36.4 Clocks

VDOA root clock is derived from the AXI clock. By default, its frequency is 264 MHz.

36.5 Resets and interrupts

VDOA has no reset mechanism.

VDOA use interrupt 50 to notify the CPU when a transfer is complete.

36.6 Initializing the driver

The **vdoa_setup()** function initializes the VDOA. This function requires setting the following parameters:

- Frame width and height
- Stride line of the VPU output, which is also the input of VDOA
- Stride line of the IPU input, which is also the output of VDOA
- Interlaced mode selection.
    - If the VPU output is interlaced, three buffers are used to transfer the sequential three fields of the stream for deinterlacing purposes.
    - If the VPU output is not interlaced, only one frame buffer is used.
- Band mode settings
    - If the VDOA is in sync mode, the driver needs to set the band size and which IPU to use for the handshake.
    - In non-sync mode, these two parameters are ignored.
- Pixel format selection

- This sets whether the VDOA outputs in YUV420 partial interleaved mode or in YUV422 interleaved mode.
- This parameter is used in the IPU post processing.

36.7 Testing the driver

The VDOA is tested inside the VPU decoding test.

When the map type of the VPU is set to TILED_FRAME_MB_RASTER_MAP, the VDOA must be enabled as follows:

1. Before starting a new VDOA tranfer, the VDOA must be in idle state. Ensure that the input and output address are all 3 LSB aligned.
2. When running the VPU decoding test, a prompt asks whether VDOA should be enabled.
3. Enter 'Y' or 'y' to confirm, and the VDOA is enabled.

Chapter 37
Configuring the VPU Driver

37.1 Overview

The video processing unit (VPU) is a high performance multi-standard video codec in the i.MX 6Dual/6Quad and i.MX 6Solo/6DualLite products. It is located in the memory map at 0204 0000h.

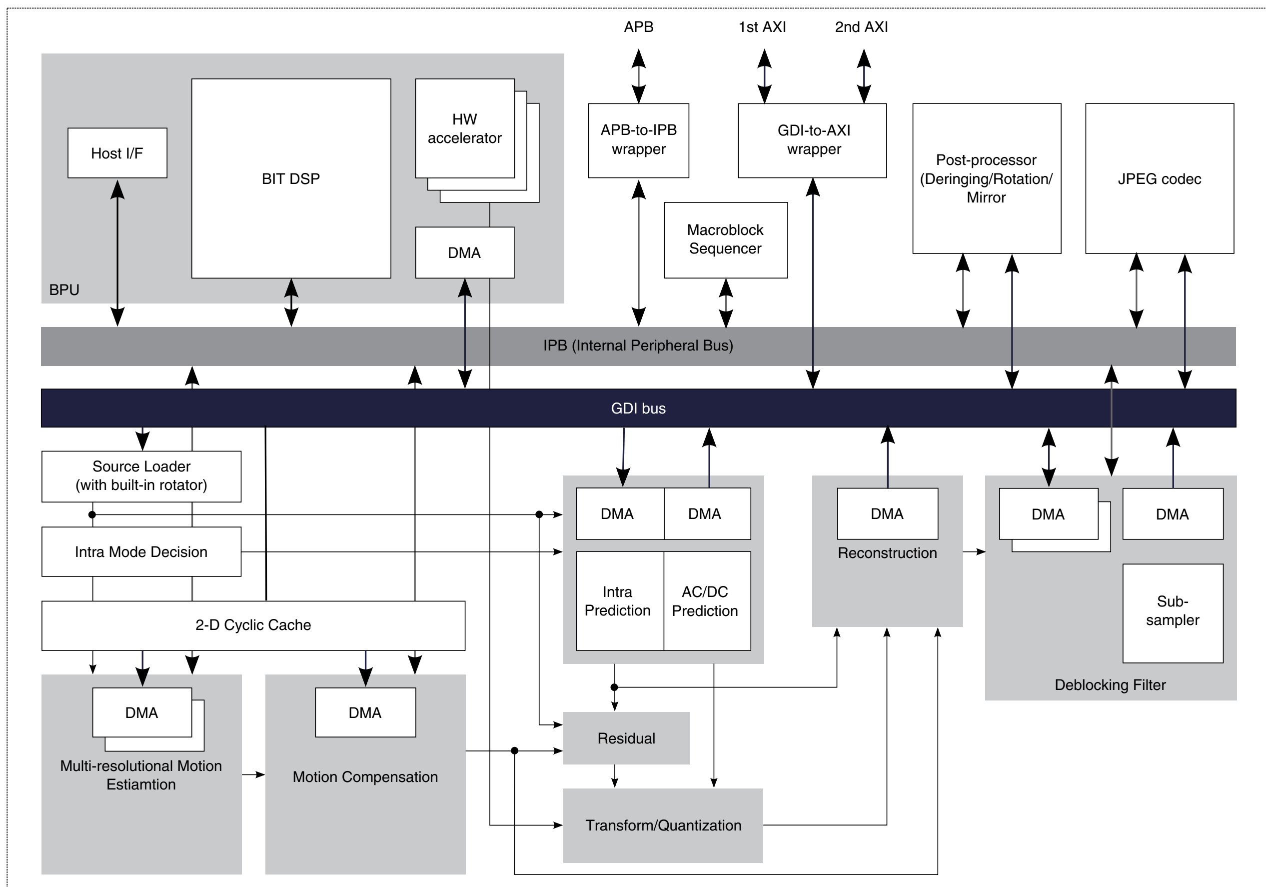


Figure 37-1. VPU block diagram

The VPU has an embedded BIT processor, which controls internal video processing subblocks and communicates with the host processor through the IP bus. The VPU can directly access the memory through the AXI bus for data throughput.

37.2 Feature summary

The VPU supports the following features:

- Video/image encode for the following
  - H.264 BP/CBP/MP/HP
  - VC-1 SP/MP/AP
  - MPEG-4 SP/ASP
  - H.263 P0/P3
  - MPEG-1/2 MP/HP
  - Divx (Xvid) HP/PP/HTP/HDP
  - RV8/9/10
  - Sorenson Spark
  - VP8(1280 x 720)
  - AVS
  - H.264-MVC (1280 x 720)
  - MJPEG BP
- Video/image decode for the following
  - H.264 BP/CBP
  - MPEG-4 SP
  - H.263 P0/P3
  - MJPEG BP encoding
- Multi-instance
  - VPU can support infinite instances of decoder plus encoder concurrently by switching the contexts of codec in frame based mode which are stored in the memory. It is very helpful for multi-channel decoder applications.
- Performance
  - full HD video decoder up to 1920 x 1088 at 30 fps plus D1 at 303 fps
  - full HD encoder up to 1920 x 1088 at 30 fps
  - MJPEG codec up to 8192 x 8192

The firmware driver supports H264/VC-1 decoding, H264 encoding, and dual video decoder plus display.

37.3 Modes of operation

As described in the following tables, the VPU input stream and output stream each have two modes of operation.

Table 37-1. VPU input modes of operation

Mode	What it does
Stream	Places the raw bitstream into the stream buffer. For full details, see Using the input stream modes.
File play	Processes one frame at a time. For full details, see Using the input stream modes.

Table 37-2. VPU output modes of operation

Mode	What it does
Linear	The video output is in frame mode, meaning a complete frame is produced and stored in the registered frame buffer. For full details, see Using the output stream modes.
Tiled	The video output is in 16 x 16 block format. For full details, see Using the output stream modes.

37.3.1 Using the input stream modes

In streaming mode, the raw bitstream enter the stream buffer as there is space. The VPU's read and write pointer records and indicates the current status. Tihe stream buffer is in ring-buffer mode, and so the write pointer returns to the start after it reaches the end. In the decoder, the VPU analyzes the bitstream and starts decoding by checking the start sequence. In the encoder, the size of a frame is fixed. The VPU obtains the YUV bitstream from the buffer and all encoder configuration parameters from user input.

In file play mode, one complete frame is placed into the frame buffer at a time. The next frame is not placed into the frame buffer until after the current frame is processed. Currently, the VPU firmware does not support file play mode.

37.3.2 Using the output stream modes

In linear mode, video is output in frame mode, meaning that a complete frame is produced and stored in the registered frame buffer. The output is put into a continuous frame buffer regardless of whether it is in progressive or interlaced mode. The output data appears as shown in the following figure.

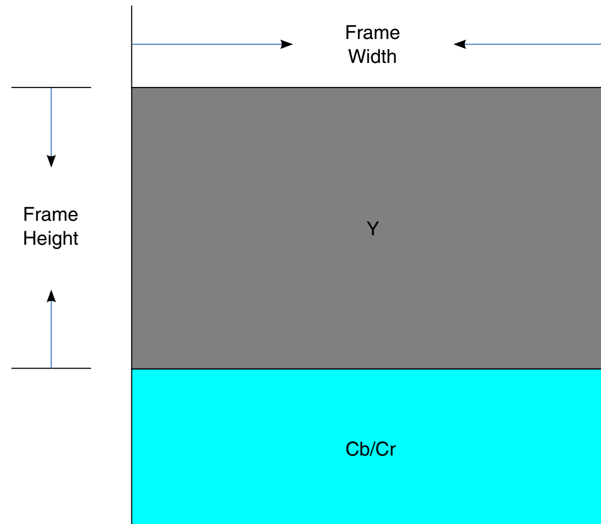


Figure 37-2. YUV420 partial interleaved data format layout

For YUV420 partial interleaved mode, which is also known as NV12, the data lies as:

- Y(0,0)Y(0,1)..Y(0,fw-1)Y(1,0)..Y(fh-1,fw-1)
- Cb(0,0)Cr(0,0)Cb(0,1)Cr(0,1)..Cb(0,fw/2-1)Cr(0,fw/2-1)Cb(1,0)Cr(1,0)..Cb(fh/2-1,fw/2-1)Cr(fh/2-1,fw/2-1)

In tiled mode, the video is output in a 16 x 16 block format, which permits faster loading than in frame mode. 16 x 16 is the size of macroblocks for some codec standards, such as AVC, and these macroblocks are frequently exchanged between the VPU internal ram/cache and the external memory. With tiled mode enabled, the decoding performance can increase by approximately 10%.

The following figure shows the data arrangement in the memory for progressive YCbCr4:2:0. The M-N-O-P format indicates the position of a pixel component:

- M means the component name (Y, Cb or Cr)
- N means the block number
- O means the line number inside the block
    - For Luma, it varies from 0~7
    - For Chroma, it varies from 0~3
- P means the pixel index inside the line.

base \ offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x00	Y0-0-0	Y0-0-1	Y0-0-2	Y0-0-3	Y0-0-4	Y0-0-5	Y0-0-6	Y0-0-7	Y0-1-0	Y0-1-1	Y0-1-2	Y0-1-3	Y0-1-4	Y0-1-5	Y0-1-6	Y0-1-7
0x10	Y0-2-0	Y0-2-1	Y0-2-2	Y0-2-3	Y0-2-4	Y0-2-5	Y0-2-6	Y0-2-7	Y0-3-0	Y0-3-1	Y0-3-2	Y0-3-3	Y0-3-4	Y0-3-5	Y0-3-6	Y0-3-7
0x20	Y0-4-0	Y0-4-1	Y0-4-2	Y0-4-3	Y0-4-4	Y0-4-5	Y0-4-6	Y0-4-7	Y0-5-0	Y0-5-1	Y0-5-2	Y0-5-3	Y0-5-4	Y0-5-5	Y0-5-6	Y0-5-7
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
0x70	Y0-14-0	Y0-14-1	Y0-14-2	Y0-14-3	Y0-14-4	Y0-14-5	Y0-14-6	Y0-14-7	Y0-15-0	Y0-15-1	Y0-15-2	Y0-15-3	Y0-15-4	Y0-15-5	Y0-15-6	Y0-15-7
0x80	Y0-0-8	Y0-0-9	Y0-0-10	Y0-0-11	Y0-0-12	Y0-0-13	Y0-0-14	Y0-0-15	Y0-1-8	Y0-1-9	Y0-1-10	Y0-1-11	Y0-1-12	Y0-1-13	Y0-1-14	Y0-1-15
0x90	Y0-2-8	Y0-2-9	Y0-2-10	Y0-2-11	Y0-2-12	Y0-2-13	Y0-2-14	Y0-2-15	Y0-3-8	Y0-3-9	Y0-3-10	Y0-3-11	Y0-3-12	Y0-3-13	Y0-3-14	Y0-3-15
0xA0	Y0-4-8	Y0-4-9	Y0-4-10	Y0-4-11	Y0-4-12	Y0-4-13	Y0-4-14	Y0-4-15	Y0-5-8	Y0-5-9	Y0-5-10	Y0-5-11	Y0-5-12	Y0-5-13	Y0-5-14	Y0-5-15
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
0xF0	Y0-14-8	Y0-14-9	Y0-14-10	Y0-14-11	Y0-14-12	Y0-14-13	Y0-14-14	Y0-14-15	Y0-15-8	Y0-15-9	Y0-15-10	Y0-15-11	Y0-15-12	Y0-15-13	Y0-15-14	Y0-15-15
0x100	Y1-0-0	Y1-0-1	Y1-0-2	Y1-0-3	Y1-0-4	Y1-0-5	Y1-0-6	Y1-0-7	Y1-1-0	Y1-1-1	Y1-1-2	Y1-1-3	Y1-1-4	Y1-1-5	Y1-1-6	Y1-1-7
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Figure 37-3. Luma data layout in VPU output tiled mode

base \ offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x00	CB0-0-0	CR0-0-0	CB0-0-1	CR0-0-1	CB0-0-2	CR0-0-2	CR0-0-3	CR0-0-3	CR0-1-0	CR0-1-0	CR0-1-1	CR0-1-1	CR0-1-2	CR0-1-2	CR0-1-3	CR0-1-3
0x10	CB0-2-0	CR0-2-0	CB0-2-1	CR0-2-1	CB0-2-2	CR0-2-2	CR0-2-3	CR0-2-3	CR0-3-0	CR0-3-0	CR0-3-1	CR0-3-1	CR0-3-2	CR0-3-2	CR0-3-3	CR0-3-3
0x20	CB0-4-0	CR0-4-0	CB0-4-1	CR0-4-1	CB0-4-2	CR0-4-2	CR0-4-3	CR0-4-3	CR0-5-0	CR0-5-0	CR0-5-1	CR0-5-1	CR0-5-2	CR0-5-2	CR0-5-3	CR0-5-3
0x30	CB0-6-0	CR0-6-0	CB0-6-1	CR0-6-1	CB0-6-2	CR0-6-2	CR0-6-3	CR0-6-3	CR0-7-0	CR0-7-0	CR0-7-1	CR0-7-1	CR0-7-2	CR0-7-2	CR0-7-3	CR0-7-3
0x40	CB0-0-4	CR0-0-4	CB0-0-5	CR0-0-5	CB0-0-6	CR0-0-6	CR0-0-7	CR0-0-7	CR0-1-4	CR0-1-4	CR0-1-5	CR0-1-5	CR0-1-6	CR0-1-6	CR0-1-7	CR0-1-7
...							...	...	...	...	...	...	...	...	...	...
0x70	CB0-6-4	CR0-6-4	CB0-6-5	CR0-6-5	CB0-6-6	CR0-6-6	CR0-6-7	CR0-6-7	CR0-7-4	CR0-7-4	CR0-7-5	CR0-7-5	CR0-7-6	CR0-7-6	CR0-7-7	CR0-7-7
...						...	...	...	...	...	...	...	...	...	...	...

Figure 37-4. Chroma data layout in VPU output tiled mode

37.4 Clocks

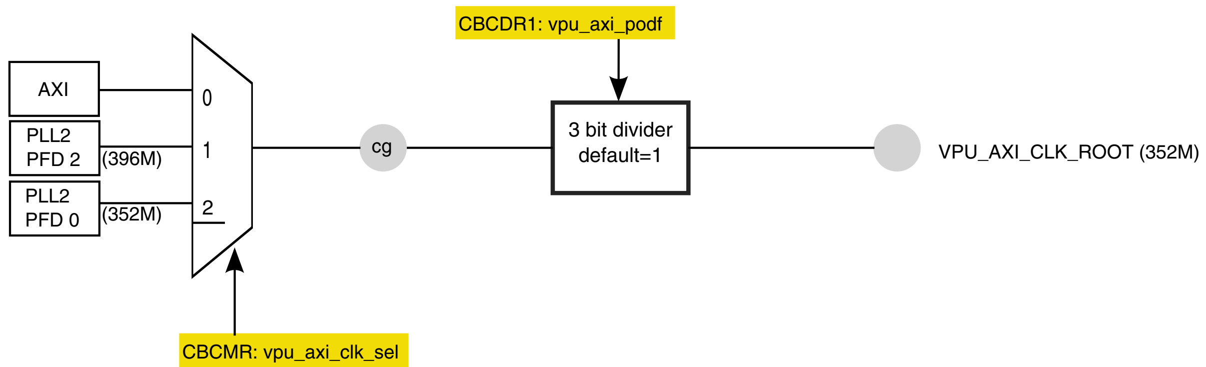


Figure 37-5. VPU working root clock path

The VPU works on 264 MHz. The clock source can be either the AXI clock or the PLL2 PFD0/PFD2.

37.5 Resets and interrupts

The VPU can be reset either by using the BIT_SW_RESET register to reset the internal VPU bus and modules or through the system reset controller. To reset the VPU with the system reset controller, set VPU_SW_RESET in the SRC register and wait for it to self-clear.

The VPU supports interrupts by setting the register BIT_INT_ENABLE. The most useful is DEC_PIC_RUN/ENC_PIC_RUN, which indicates that the current frame for decoding/ encoding has been finished.

37.6 Initializing the driver

37.6.1 Initializing the VPU for the first time

Before starting the codec tasks, VPU must be initialized for the first time. The host processor performs the following operations step by step in **VPU_Init()** automatically.

1. Set the IO system. Reserve enough chunks of memory for the VPU work buffer and DMA usage.
2. Initialize the codec instances.
3. Check whether the VPU has been initialized by checking the PC pointer of the BIT processor.
4. If the PC does not equal zero, the VPU has been initialized, and the initialization procedure is complete. If the PC equals zero, continue with the next steps.
5. Download the firmware to the VPU work buffer, which the VPU can access directly during runtime.
6. Download the first 4 Kbytes of the microcode (firmware) to the program memory in the BIT processor.
7. Set the BIT processor buffer pointers for the working buffer, the parameter buffer, and the code buffer.

   The working buffer stores the context of codec instances, so its size needs to be increased by as many instances. The common parts takes 210 Kbytes and for an extra instance, 47 Kbyte is needed. That means the working buffer size should be no less than:

   210 Kbytes + MAX_NUM_INSTANCES x 47 Kbytes.

The code buffer stores the firmware binary. It should be no smaller than the firmware size of the VPU, which varies according to the different versions of firmware or VPU products. Use the **sizeof()** function to obtain the size of the array which stores the firmware.

8. Set the control options (full/empty check) and endian mode of bitstream buffer. The VPU supports both little-endian and big-endian modes.
9. Set the frame buffer endian and chrominance option (CbCr Interleave or plannar). Note that if VDOA is enabled, the output data must be in CbCr interleaved mode.
10. Set the Interrupt Enable register.
11. Enable the BIT Processor by setting the register BIT_CODE_RUN to be 1.

37.6.2 Initializing the VPU decoder

Calling **decoder_setup()** creates a decoding instance as follows:

1. Call **VPU_DecOpen**, which allocates and configures a codec instance.
2. Set the following parameters: the codec standard (AVC, VC-1, MPEG4), instance index, and data map. Note that to support multi-instances, the context of the instance is saved for future task switches. The following registers should be backed up and restored before switching instances:
    - BIT_BIT_STREAM_CTRL
    - BIT_FRAME_MEM_CTRL
    - BIT_BIT_STREAM_PARAM
    - BIT_RD_PTR
    - BIT_WR_PTR
    - BIT_AXI_SRAM_USE
    - BIT_FRM_DIS_FLAG
3. Feed the bitstream into the bitstream buffer, and update BIT_WR_PTR.
4. Parse the bitstream, which performs sequence initialization as follows:
    - The VPU searches for the start code.
    - The VPU obtains all required configuration information from the stream, such as picture size and frame rate. Note that each encoder standard requires different configuration information.
5. Allocate the buffers and register them to the VPU. All registered buffers are used for the VPU output. The number of buffers can be greater than the minimum required, but must be no fewer.

Now we can start the video decoding by calling **VPU_DecStartOneFrame**.

37.6.3 VPU encoder initialization

Calling **encoder_setup()** creates an encoding instance as follows:

1. Call **VPU_EncOpen**, which allocates and configures a codec instance. The user must set the following parameters in this step: picture size (width and height), codec stardard, data format, GOP size and frame rate.
2. Feed the bitstream into the bitstream buffer, using either ring buffer mode or line-buffer mode.
   - In ring buffer mode, a single fixed-size buffer is used as if it were connected end to end. A read and a write pointer indicates the usage of the buffer. This mode is useful when the system memory is very limited.
   - In line-buffer mode, a whole frame is be put into the bitstream buffer, and VPU encodes from the start to the end of the frame without exchanging data with the host.
3. Perform sequence initialization. Applications should reserve a minimum number of frame buffers to VPU for proper encoding operation, using the returned parameter from **VPU_EncGetInitialInfo()** to identify the minimum number of frame buffers required.
4. Allocate the buffers and register them to the VPU. All registered buffers are used for the VPU output. The number of buffers can be greater than the minimum required, but must be no fewer.
5. When opening an encoder instance is completed by calling **VPU_EncGetInitialInfo()**, applications must use **VPU_EncGiveCommand()** to generate the high-level header syntaxes, such as VOS/VO/VOL headers in MPEG-4 and SPS/PPS in AVC.
   - The recommended way for obtaining header syntaxes is to use the ENC_PUT_AVC/MP4_HEADER command by means of the bitstream buffer. If applications use this set of commands, the header syntaxes are stored in the bitstream buffer according to the given endian setting.
   - The other way for generating header syntaxes is to use the PARA_BUF. Regardless of streaming mode, this command generates header syntaxes and writes them to PARA_BUF instead of the bitstream buffer. However, endian setting is always big endian, so endian conversion must be performed by the host processor for little-endian systems. Perform endian conversion as follows:
     - For MPEG-4, use ENC_GET_VOS_HEADER, ENC_GET_VO_HEADER, or ENC_GET_VOL_HEADER.
     - For H.264, use ENC_GET_SPS_RBSP or ENC_GET_PPS_RBSP.

Now we can call **VPU_EncStartOneFrame** to initiate the video encoding. After the frame encoding is finished, the host processor can obtain the output from either the ring or line stream buffer and store the output to the destination.

37.6.4 Using the multi-instance operation

To support the multi-instance operation, the BIT processor uses an internal context parameter set for each decoder instance. While creating a new instance and starting picture processing, the VPU automatically creates and updates a set of these context parameters. Because of this internal context management scheme, different decoder tasks running on the host processor can use their own instance numbers to control VPU operations independently.

When creating a new instance, the application task is given a new handle to specify an instance as long as a new handle is available on the VPU. The application task can then handle all of it subsequent operations separately on the VPU by using this task-specific handle. If no new handle is available, instance creation fails.

Because the VPU can only perform one picture processing task at a time, each application shares the unique hardware resources in time-division mode. As a result, each task should check whether the VPU is ready before starting a new picture operation.

By calling a function for closing a certain instance, the application can terminate a single video operation task on the VPU.

37.7 Testing the driver

The VPU has tests for the encoder and decoder as well as a multi-instance demo (dual video decoder + display).

37.7.1 Testing the decoder

The decoder test has two modes:

- Endless test
- Play the file to end

To run the test, the user enters either "1" for the endless test or "2" to play the file to end. If the user selects 1, the video plays repeatedly once it reaches the end. if the user selects 2, the video plays to the end and then exits the decoding processing, just like a normal movie view.

37.7.2 Testing the encoder

In the encoder test, the default encode stardard is AVC, and the default input size is 320 x 240. The output is stored in the memory.

Run the test as follows:

1. Load a yuv420 file onto the SD card with FAT32 formatted. Note that extension must be ".YUV" (case sensitive).
2. Set the user input parameters in **encode_test()**, the cmdl structure. Oay attention to the enc_width and enc_height, format, and fps.
3. The output is stored in the memory.

After the test has completed done, the user can choose to either start a decoding process to play the encoded file or to check the data in PC side by using debug tools to dumping the data to files.

37.7.3 Running the multi-instance demo

This section explains how to set up and run the dual video + dual display demo on an engineering sample board. See your board's schematics for your board's specific settings.

- The default video standard is AVC.
- The first display is the Hannstar LVDS panel.
- The second display is an embedded HDMI display.

First, use the following procedure to create the image on the SD card. In Linux, use `fidsk/mkfs.vfat/dd` to create a bootable image on the same SD card as the FAT32 file system.

1. Enter `sudo fdisk /dev/sdb` at the command line. sudo fidsk /dev/sdx , sdx is the device name of your SD card.
2. Delete the existing partition if there is one, as follows.

```
Command (m for help): d
Selected partition 1
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1023, default 1): 256
```

NOTE

The start address should be larger than 2 Mbytes so that there is space reserved to burn the test binary. For one cylinder, it is 4 Kbytes. This demo shows a 1-Gbyte space.

```
Last cylinder or +size or +sizeM or +sizeK (256-1023, default 1023): 1023
Command (m for help): w
The partition table has been altered!
Calling ioctl() to re-read partition table.
Syncing disks.
```

3. Now there is one partition on the SD card.
4. Enter cat /proc/partitions

```
cat /proc/partitions
major minor  #blocks  name
8     0   78125000 sda
   8     1     104391 sda1
   8     2   78019672 sda2
 253     0   75956224 dm-0
 253     1    2031616 dm-1
   8    16    3872256 sdb
   8    17    2904576 sdb1
```

5. Enter sudo mkfs.vfat /dev/sdb1 to format the partition.
6. Copy two video clips to the SD card.
    - Note that the filenames for the video clips should contain fewer than 8 characters and have a .264 extension.
    - The video should be RAW h264 encoded files with no container; the program finds the first two valid 264 files.
7. Enter sudo dd if=output/mx6dq/evb_rev_a/bin/mx6dq_evb_rev_a-vdec-sdk.bin of=/dev/sdb seek=2 skip=2 & sync to burn the image to the SD card.

NOTE

Seek=2 skip=2 is mandatory. Without them, the MBR of the file system will be overwritten.

Once the SD card is created, set up the demo according to the following sequence (see Figure 37-6):

1. Put the SD card into SLOT4.
2. Set the boot switch to boot from SD4.
3. Plug the Hannstar LVDS panel into LVDS0 connector.
4. Plug the HDMI cable to J5 for the secondary display.
5. Connect the serial cable and 5 V power supply, which powers on the board.

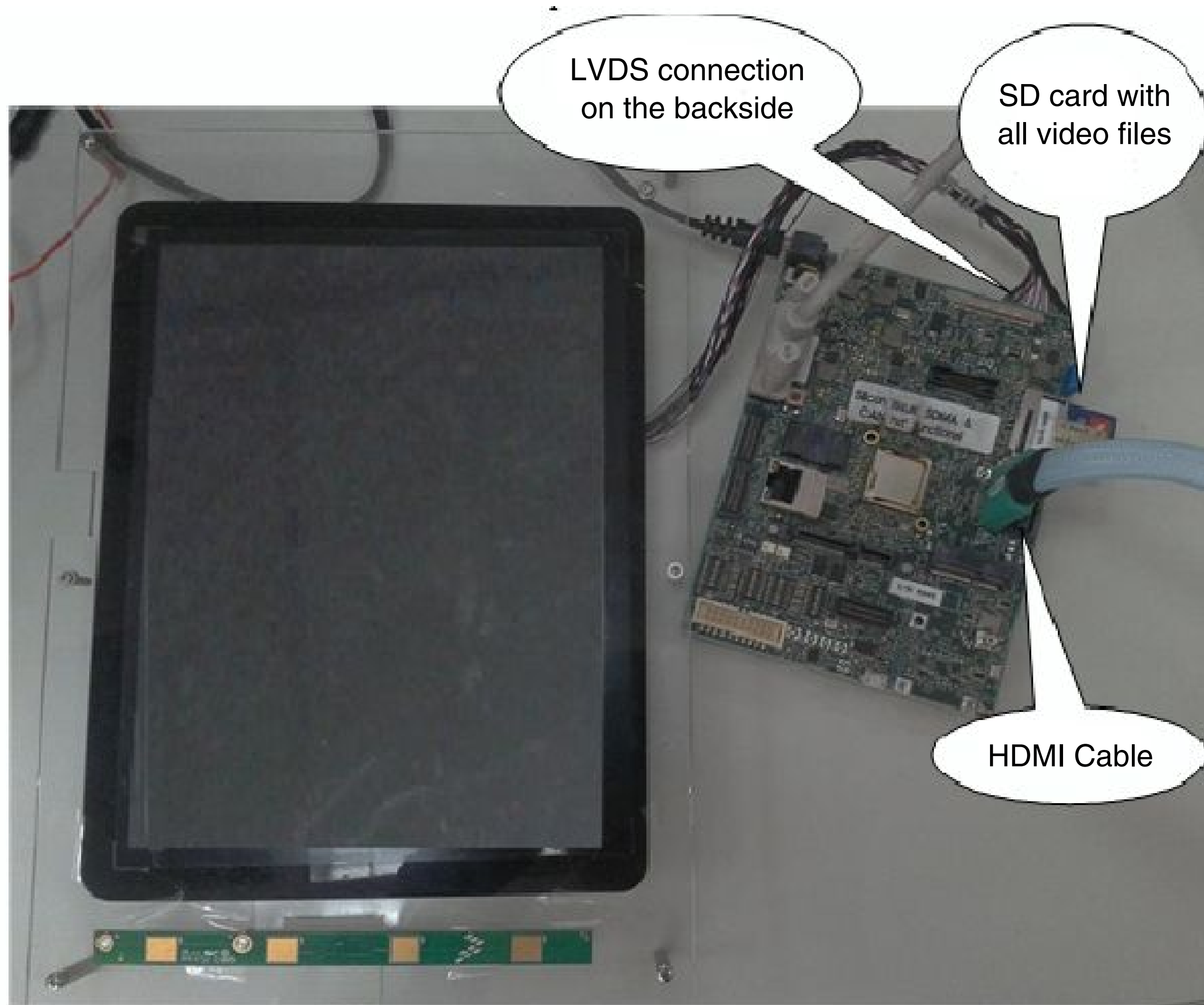


Figure 37-6. Demo connections

Chapter 38
Configuring the Watchdog Driver

38.1 Overview

This chapter explains how to configure the watchdog driver. The watchdog timer (WDOG-1) protects against system failures by providing a way to escape unexpected events or programming errors.

Once the WDOG-1 is activated, it must be serviced by the software on a periodic basis. If servicing does not take place, the timer times out. Upon a timeout, the WDOG-1 asserts the internal system reset signal, wdog_rst, which goes to the system reset controller.

The watchdog also has a provision for WDOG-1 signal assertion by timeout counter expiration and programmable interrupt generation before the counter actually times out.

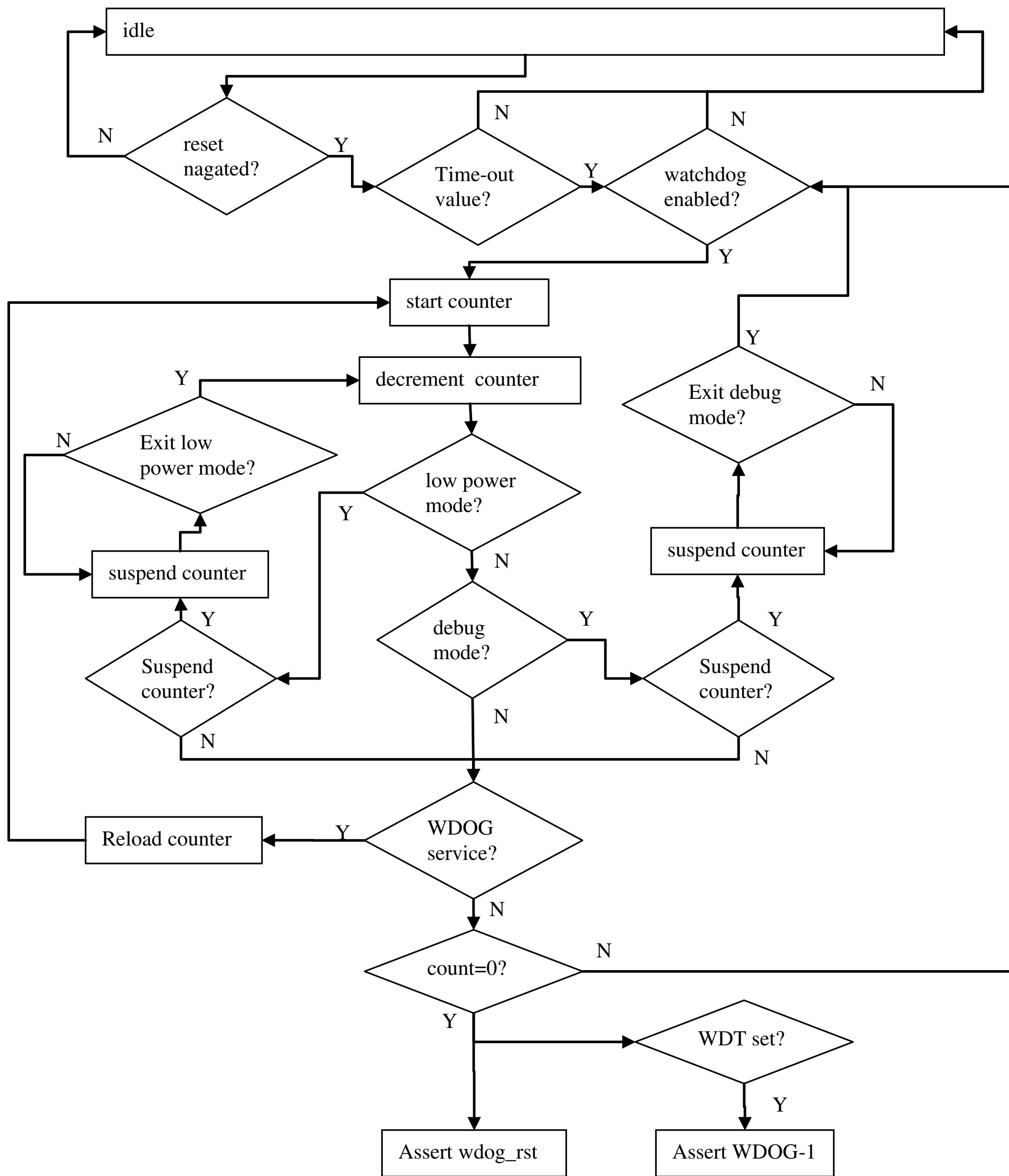


Figure 38-1. Watchdog flow diagram

38.2 Feature summary

The watchdog timer has the following features:

- Configurable timeout counter with timeout periods from 0.5 seconds up to 128 seconds; timeout expiration results in the assertion of of the wdog_rst reset signal.
- Time resolution of 0.5 seconds
- Configurable timeout counter that can be programmed to run or stop during low power modes
- Configurable timeout counter that can be programmed to run or stop during debug mode
- Programmable interrupt generation prior to timeout
- Programmable time duration between interrupt and timeout events, from 0 to 127.5 seconds in steps of 0.5 seconds.
- Power down counter with fixed time-out period of 16 seconds; if not disabled, asserts WDOG-1 signal low after reset

38.3 Modes of operation

Table 38-1. Watchdog modes of operation

Mode	What it does
Low-power modes	The WDOG-1 timer operation can be suspended in low power mode
Debug mode	The WDOG-1 timer operation can be suspended in debug mode
Normal mode	Normal operation

38.4 Signals

Table 38-2. Watchdog signals

signals	IO	Description
WDOG-1	O	This signal powers down the chip.
wdog_rst	O	This signal is a reset source for the chip.

38.5 Resets and interrupts

The SDK only implements the watchdog's software reset function and did not implement interrupt mode.

The Watchdog IRQ numbers are 112 and 113.

38.6 Initializing the driver

Initialize the driver as follows:

1. Set the counter to 3h, which means 2 seconds.
2. Disable software reset and power down.

38.7 Testing the driver

Test the driver as follows:

1. Run the test code to initialize and enable watchdog.
2. The test code feeds watchdog in an endless loop.
3. Enter "Y" to stop the feeding.
4. The system resets in 2 seconds.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, AltiVec, C-5, CodeTest, CodeWarrior, ColdFire, C-Ware, Energy Efficient Solutions logo, Kinetis, mobileGT, PowerQUICC, Processor Expert, QorIQ, Qorivva, StarCore, Symphony, and VortiQa are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Airfast, BeeKit, BeeStack, ColdFire+, CoreNet, Flexis, MagniV, MXC, Platform in a Package, QorIQ Qonverge, QUICC Engine, Ready Play, SafeAssure, SMARTMOS, TurboLink, Vybrid, and Xtrinsic are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.
© 2012 Freescale Semiconductor, Inc.

Document Number: IMX6FG
Rev. 0, 11/2012

freescale™