i.MX Linux® Reference Manual

Document Number: IMXLXRM
Rev. 0, 12/2015

freescale™

Contents

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

Chapter 1
About this Book

1.1 Audience 29

1.1.1 Conventions 29

1.1.2 Definitions, Acronyms, and Abbreviations 29

Chapter 2
Introduction

2.1 Overview 33

2.1.1 Software Base 33

2.1.2 Features 33

Chapter 3
Machine Specific Layer (MSL)

3.1 Introduction 39

3.2 Interrupts (Operation) 39

3.2.1 Interrupt Hardware Operation 40

3.2.2 Interrupt Software Operation 40

3.2.3 Interrupt Features 40

3.2.4 Interrupt Source Code Structure 41

3.2.5 Interrupt Programming Interface 41

3.3 Timer 42

3.3.1 Timer Software Operation 42

3.3.2 Timer Features 42

3.3.3 Timer Source Code Structure 43

3.3.4 Timer Programming Interface 43

3.4 Memory Map 43

3.4.1 Memory Map Hardware Operation 43

3.4.2 Memory Map Software Operation 43

3.4.3 Memory Map Features 43

3.4.4 Memory Map Source Code Structure 44
3.5 IOMUX 44
3.5.1 IOMUX Hardware Operation 45
3.5.2 IOMUX Software Operation 45
3.5.3 IOMUX Features 45
3.5.4 IOMUX Source Code Structure 46
3.5.5 IOMUX Programming Interface 46
3.5.6 IOMUX Control Through GPIO Module 46
3.5.6.1 GPIO Hardware Operation 47
3.5.6.1.1 Muxing Control 47
3.5.6.1.2 PULLUP Control 47
3.5.6.2 GPIO Software Operation (general) 47
3.5.6.3 GPIO Implementation 47
3.6 General Purpose Input/Output(GPIO) 48
3.6.1 GPIO Software Operation 48
3.6.1.1 API for GPIO 48
3.6.2 GPIO Features 49
3.6.3 GPIO Module Source Code Structure 49
3.6.4 GPIO Programming Interface 2 49

Chapter 4
Smart Direct Memory Access (SDMA) API

4.1 Overview 51
4.1.1 Hardware Operation 51
4.1.2 Software Operation 51
4.1.3 Source Code Structure 52
4.1.4 Programming Interface 53
4.1.5 Usage Example 53

Chapter 5
AHB-to-APBH Bridge with DMA (APBH-Bridge-DMA)

5.1 Overview....55
5.1.1 Hardware Operation....55
5.1.2 Software Operation....56
5.1.3 Source Code Structure....56
5.1.4 Menu Configuration Options....57
5.1.5 Programming Interface....57
5.1.6 Usage Example....57

Chapter 6
Image Processing Unit (IPU) Drivers

6.1 Introduction....59
6.1.1 Hardware Operation....60
6.2 Software Operation....61
6.2.1 IPU Frame Buffer Drivers Overview....62
6.2.1.1 IPU Frame Buffer Hardware Operation....63
6.2.1.2 IPU Frame Buffer Software Operation....63
6.2.1.3 Synchronous Frame Buffer Driver....64
6.2.2 IPU Backlight Driver....65
6.2.3 IPU Device Driver....65
6.3 Source Code Structure....66
6.3.1 Menu Configuration Options....67
6.4 Unit Test....71
6.4.1 Framebuffer Tests....71
6.4.2 Video4Linux API test....71
6.4.3 IPU Device Unit test....73

Chapter 7
MIPI DSI Driver

7.1 Introduction....77
7.1.1 MIPI DSI IP Driver Overview....77

7.1.2 MIPI DSI Display Panel Driver Overview 78
7.1.3 Hardware Operation 78
7.2 Software Operation 78
7.2.1 MIPI DSI IP Driver Software Operation 78
7.2.2 MIPI DSI Display Panel Driver Software Operation 79
7.3 Driver Features 79
7.3.1 Source Code Structure 80
7.3.2 Menu Configuration Options 80
7.3.3 Programming Interface 80

Chapter 8
LVDS Display Bridge(LDB) Driver

8.1 Introduction 81
8.1.1 Hardware Operation 81
8.1.2 Software Operation 81
8.1.3 Source Code Structure 82
8.1.4 Menu Configuration Options 82

Chapter 9
Video for Linux Two (V4L2) Driver

9.1 Introduction 83
9.2 V4L2 Capture Device 84
9.2.1 V4L2 Capture IOCTLs 84
9.2.2 Use of the V4L2 Capture APIs 86
9.3 V4L2 Output Device 87
9.3.1 V4L2 Output IOCTLs 87
9.3.2 Use of the V4L2 Output APIs 88
9.4 Source Code Structure 88
9.4.1 Menu Configuration Options 89
9.4.2 V4L2 Programming Interface 89

Chapter 10
Electrophoretic Display Controller (EPDC) Frame Buffer Driver

10.1 Introduction....91
10.2 Hardware Operation....92
10.3 Software Operation....92
10.3.1 EPDC Frame Buffer Driver Overview....92
10.3.2 EPDC Frame Buffer Driver Extensions....93
10.3.3 EPDC Panel Configuration....93
10.3.3.1 Boot Command Line Parameters....94
10.3.4 EPDC Waveform Loading....94
10.3.4.1 Using a Default Waveform File....95
10.3.4.2 Using a Custom Waveform File....95
10.3.5 EPDC Panel Initialization....96
10.3.6 Grayscale Framebuffer Selection....97
10.3.7 Enabling an EPDC Splash Screen....97
10.4 Source Code Structure....98
10.5 Menu Configuration Options....98
10.6 Programming Interface....99
10.6.1 IOCTLs/Functions....99
10.6.2 Structures and Defines....102

Chapter 11
Pixel Pipeline (PxP) DMA-ENGINE Driver

11.1 Introduction....105
11.2 Hardware Operation....105
11.3 Software Operation....105
11.3.1 Key Data Structs....105
11.3.2 Channel Management....106
11.3.3 Descriptor Management....107
11.3.4 Completion Notification....107

11.3.5 Limitations..... 107
11.4 Menu Configuration Options..... 107
11.5 Source Code Structure..... 108

Chapter 12
ELCDIF Frame Buffer Driver

12.1 Introduction..... 109
12.2 Hardware Operation..... 109
12.3 Software Operation..... 109
12.4 Menu Configuration Options..... 110
12.5 Source Code Structure..... 110

Chapter 13
Graphics Processing Unit (GPU)

13.1 Introduction..... 111
13.1.1 Driver Features..... 111
13.1.1.1 Hardware Operation..... 112
13.1.1.2 Software Operation..... 112
13.1.1.3 Source Code Structure..... 113
13.1.1.4 Library Structure..... 113
13.1.1.5 API References..... 114
13.1.1.6 Menu Configuration Options..... 115

Chapter 14
Wayland

14.1 Introduction..... 117
14.2 Hardware Operation..... 117
14.3 Software Operation..... 117
14.4 Yocto Build Instructions..... 117
14.5 Customizing Weston..... 118
14.5.1 Multi display supported in Weston..... 118
14.5.2 Multi buffer supported in Weston..... 118
14.6 Running Weston..... 119

Chapter 15
On-Chip High Definition Multimedia Interface (HDMI) Driver

15.1 Introduction....121
15.1.1 Hardware Operation....121
15.2 Software Operation....123
15.2.1 Core....123
15.2.2 Video....124
15.2.3 Display Device Registration and Initialization....125
15.2.4 Hotplug Handling and Video Mode Changes....126
15.2.5 Audio....126
15.2.6 CEC....128
15.3 Source Code Structure....128
15.3.1 Linux Menu Configuration Options....130
15.4 Unit Test....131
15.4.1 Video....131
15.4.2 Audio....132
15.4.3 CEC....132
15.4.4 HDCP....132

Chapter 16
External High-Definition Multimedia Interface (HDMI) for i.MX 6SoloLite

16.1 Introduction....135
16.2 Software Operation....135
16.2.1 Hotplug Handling and Video Mode Changes....135
16.3 Source Code Structure....136
16.3.1 Linux Menu Configuration Options....137
16.4 Unit Test....137
16.4.1 Video....137
16.4.2 Audio....138

Chapter 17
X Windows Acceleration

17.1 Introduction....139
17.2 Hardware Operation....139
17.3 Software Operation....139
17.3.1 X-Windows Acceleration Architecture....140
17.3.2 i.MX 6 Driver for X-Windows System....141
17.3.3 i.MX 6 Direct Rendering Infrastructure (DRI) for X-Windows System....143
17.3.4 EGL- X Library....144
17.3.5 xorg.conf for i.MX 6....145
17.3.6 Setup X-Windows System Acceleration on Yocto....147
17.3.7 Setup X Window System Acceleration....148
17.3.8 Troubleshooting....148

Chapter 18
Video Processing Unit (VPU) Driver

18.1 Hardware Operation....151
18.1.1 Software Operation....152
18.1.2 Source Code Structure....153
18.1.3 Menu Configuration Options....154
18.1.4 Programming Interface....155
18.1.5 Defining an Application....156

Chapter 19
OmniVision Camera Driver

19.1 OV5640 Using MIPI CSI-2 interface....157
19.1.1 Hardware Operation....157
19.1.2 Software Operation....158
19.1.3 Source Code Structure....158
19.1.4 Linux Menu Configuration Options....158
19.2 OV5642 Using parallel interface....159
19.2.1 Hardware Operation....159

19.2.2 Software Operation..... 159
19.2.3 Source Code Structure..... 160
19.2.4 Linux Menu Configuration Options.....160

Chapter 20
MIPI CSI2 Driver

20.1 Introduction.....163
20.1.1 MIPI CSI2 Driver Overview.....163
20.1.2 Hardware Operation.....164
20.2 Software Operation..... 164
20.2.1 MIPI CSI2 Driver Initialize Operation..... 164
20.2.2 MIPI CSI2 Common API Operation.....165
20.3 Driver Features.....165
20.3.1 Source Code Structure..... 166
20.3.2 Menu Configuration Options..... 166
20.3.3 Programming Interface..... 166
20.3.4 Interrupt Requirements..... 167

Chapter 21
Low-level Power Management (PM) Driver

21.1 Hardware Operation.....169
21.1.1 Software Operation..... 169
21.1.2 Source Code Structure..... 170
21.1.3 Menu Configuration Options..... 171
21.1.4 Programming Interface..... 171
21.1.5 Unit Test.....171

Chapter 22
PF100 Regulator Driver

22.1 Introduction.....173
22.2 Hardware Operation.....173
22.2.1 Driver Features.....174

22.3 Software Operation 174
22.3.1 Regulator APIs 174
22.4 Driver Architecture 175
22.4.1 Driver Interface Details 177
22.4.2 Source Code Structure 177
22.4.3 Menu Configuration Options 178

Chapter 23
CPU Frequency Scaling (CPUFREQ) Driver

23.1 Introduction 179
23.1.1 Software Operation 179
23.1.2 Source Code Structure 180
23.2 Menu Configuration Options 181
23.2.1 Board Configuration Options 181

Chapter 24
Dynamic Bus Frequency Driver

24.1 Introduction 183
24.1.1 Operation 183
24.1.2 Software Operation 183
24.1.3 Source Code Structure 184
24.2 Menu Configuration Options 184
24.2.1 Board Configuration Options 184

Chapter 25
Thermal Driver

25.1 Introduction 187
25.1.1 Thermal Driver Overview 187
25.2 Hardware Operation 187
25.2.1 Thermal Driver Software Operation 188
25.3 Driver Features 188
25.3.1 Source Code Structure 188
25.3.2 Menu Configuration Options 188

25.3.3 Programming Interface 189
25.4 Unit Test 189

Chapter 26
Anatop Regulator Driver

26.1 Introduction 191
26.1.1 Hardware Operation 191
26.2 Driver Features 191
26.2.1 Software Operation 192
26.2.2 Regulator APIs 192
26.2.3 Driver Interface Details 193
26.2.4 Source Code Structure 193
26.2.5 Menu Configuration Options 193

Chapter 27
SNVS Real Time Clock (SRTC) Driver

27.1 Introduction 195
27.1.1 Hardware Operation 195
27.2 Software Operation 195
27.2.1 IOCTL 195
27.2.2 Keep Alive in the Power Off State 196
27.3 Driver Features 196
27.3.1 Source Code Structure 197
27.3.2 Menu Configuration Options 197

Chapter 28
Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver

28.1 ALSA Sound Driver Introduction 199
28.2 SoC Sound Card 202
28.2.1 Stereo CODEC Features 202
28.2.2 7.1 Audio Codec Features 203
28.2.3 AM/FM Codec Features 203
28.2.4 Sound Card Information 203

28.3 Hardware Operation....204
28.3.1 Stereo Audio CODEC....204
28.3.2 7.1 Audio Codec....205
28.3.3 AM/FM Codec....205
28.4 Software Operation....205
28.4.1 ASoC Driver Source Architecture....206
28.4.2 Sound Card Registration....207
28.4.3 Device Open....208
28.4.4 Devicetree Binding....208
28.4.5 Menu Configuration Options....208
28.5 Unit Test....209
28.5.1 Stereo CODEC Unit Test....209
28.5.2 7.1 Audio Codec Unit Test....210
28.5.3 AM/FM Codec Unit Test....211

Chapter 29
Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver for i.MX 6SoloLite

29.1 ALSA Sound Driver Introduction....213
29.2 SoC Sound Card....216
29.2.1 Stereo CODEC Features....216
29.2.2 AM/FM Codec Features....216
29.2.3 Sound Card Information....217
29.3 Hardware Operation....217
29.3.1 Stereo Audio CODEC....217
29.3.2 7.1 Audio Codec....218
29.3.3 AM/FM Codec....218
29.4 Software Operation....219
29.4.1 ASoC Driver Source Architecture....219
29.4.2 Sound Card Registration....219
29.4.3 Device Open....220

29.4.4 Platform Data 220
29.4.5 Menu Configuration Options 220

Chapter 30
Asynchronous Sample Rate Converter (ASRC) Driver

30.1 Introduction 223
30.1.1 Hardware Operation 223
30.2 Software Operation 224
30.2.1 Sequence for Memory to ASRC to Memory 225
30.2.2 Sequence for Memory to ASRC to Peripheral 225
30.3 Source Code Structure 226
30.3.1 Linux Menu Configuration Options 226
30.4 Devicetree Binding 226
30.4.1 Programming Interface (Exported API and IOCTLs) 227

Chapter 31
The Sony/Philips Digital Interface (S/PDIF) Driver

31.1 Introduction 229
31.1.1 S/PDIF Overview 229
31.1.2 Hardware Overview 230
31.1.3 Software Overview 231
31.1.4 The ASoC layer 231
31.2 S/PDIF Tx Driver 231
31.2.1 Driver Design 232
31.2.2 Provided User Interface 232
31.3 S/PDIF Rx Driver 233
31.3.1 Driver Design 234
31.3.2 Provided User Interface 234
31.4 Source Code Structure 236
31.5 Menu Configuration Options 237
31.6 Device Tree Bindings 237

31.7 Interrupts and Exceptions 237
31.8 Unit Test Preparation 238
31.8.1 Tx test step 238
31.8.2 Rx test step 238

Chapter 32
SPI NOR Flash Memory Technology Device (MTD) Driver

32.1 Introduction 239
32.1.1 Hardware Operation 239
32.1.2 Software Operation 240
32.1.3 Driver Features 240
32.1.4 Source Code Structure 240
32.1.5 Menu Configuration Options 241

Chapter 33
MMC/SD/SDIO Host Driver

33.1 Introduction 243
33.1.1 Hardware Operation 243
33.1.2 Software Operation 244
33.2 Driver Features 246
33.2.1 Source Code Structure 246
33.2.2 Menu Configuration Options 246
33.2.3 Devicetree Binding 247
33.2.4 Programming Interface 248
33.2.5 Loadable Module Operations 248

Chapter 34
NAND GPMI Flash Driver

34.1 Introduction 251
34.1.1 Hardware Operation 251
34.2 Software Operation 251
34.2.1 Basic Operations: Read/Write 252
34.2.2 Error Correction 252

34.2.3 Boot Control Block Management 252
34.2.4 Bad Block Handling 253
34.3 Source Code Structure 253
34.3.1 Menu Configuration Options 253

Chapter 35
SATA Driver

35.1 Hardware Operation 255
35.1.1 Software Operation 255
35.1.2 Source Code Structure Configuration 255
35.1.3 Linux Menu Configuration Options 256
35.1.4 Board Configuration Options 256
35.2 Programming Interface 256
35.2.1 Usage Example2 256
35.2.2 Usage Example 257

Chapter 36
Inter-IC (I2C) Driver

36.1 Introduction 259
36.1.1 I2C Bus Driver Overview 259
36.1.2 I2C Device Driver Overview 260
36.1.3 Hardware Operation 260
36.2 Software Operation 260
36.2.1 I2C Bus Driver Software Operation 260
36.2.2 I2C Device Driver Software Operation 261
36.3 Driver Features 261
36.3.1 Source Code Structure 261
36.3.2 Menu Configuration Options 262
36.3.3 Programming Interface 262
36.3.4 Interrupt Requirements 262

Chapter 37
Enhanced Configurable Serial Peripheral Interface (ECSPI) Driver

37.1 Introduction....263
37.1.1 Hardware Operation....263
37.2 Software Operation....263
37.2.1 SPI Sub-System in Linux OS....264
37.2.2 Software Limitations....265
37.2.3 Standard Operations....265
37.2.4 ECSPI Synchronous Operation....266
37.3 Driver Features....268
37.3.1 Source Code Structure....268
37.3.2 Menu Configuration Options....268
37.3.3 Programming Interface....269
37.3.4 Interrupt Requirements....269

Chapter 38
FlexCAN Driver

38.1 Driver Overview....271
38.1.1 Hardware Operation....271
38.1.2 Software Operation....271
38.1.3 Source Code Structure....272
38.1.4 Linux Menu Configuration Options....272

Chapter 39
Media Local Bus Driver

39.1 Introduction....275
39.1.1 MLB Device Module....275
39.1.2 Supported Features....276
39.1.3 MLB Driver Overview....277
39.2 MLB Driver....277
39.2.1 MLB Driver Architecture....277
39.2.2 Software Operation....279

39.3 Driver Files..... 280
39.4 Menu Configuration Options..... 280

Chapter 40
CHIPIDEA USB Driver

40.1 Introduction.....281
40.1.1 Architectural Overview.....281
40.2 Hardware Operation.....282
40.2.1 Software Operation..... 282
40.2.2 Source Code Structure..... 283
40.2.3 Menu Configuration Options..... 283
40.2.4 USB Wakeup Usage.....284
40.2.5 How to Close the USB Child Device Power.....284
40.2.6 Changing the Controller Operation Mode..... 284
40.2.7 Loadable Module Support.....284
40.2.8 USB Charger Detection..... 285
40.2.9 USB OTG HNP and SRP Support..... 285
40.2.10 Embeded Host Certification.....287
40.2.10.1 Adding TPL-Support Property.....287
40.2.10.2 VBUS Control.....287

Chapter 41
i.MX 6 PCI Express Root Complex Driver

41.1 Introduction.....289
41.1.1 PCIe.....289
41.1.2 Terminology and Conventions.....289
41.1.3 PCIe Topology on i.MX.....291
41.1.4 Features..... 293
41.2 Linux OS PCI Subsystem and RC driver.....293
41.2.1 RC Driver Source Files..... 294
41.2.2 Kernel Configurations.....294

41.3 System Resource: Memory Layout....294
41.3.1 System Resource: Interrupt lines....296
41.4 Using PCIe Endpoint and Running Tests....296
41.4.1 Ensuring PCIe System Initialization....298
41.4.2 Tests....298
41.4.3 Known issues....299

Chapter 42
EIM NOR Driver

42.1 Introduction....301
42.2 Hardware Operation....301
42.3 Software Operation....301
42.4 Source Code....301
42.5 Enabling the WEIM NOR....301

Chapter 43
Quad Serial Peripheral Interface (QuadSPI) Driver

43.1 Introduction....303
43.2 Hardware Operation....303
43.3 Software Operation....304
43.4 Driver Features....305
43.5 Source Code Structure....305
43.6 Menu Configuration Options....305

Chapter 44
Fast Ethernet Controller (FEC) Driver

44.1 Introduction....307
44.2 Hardware Operation....307
44.2.1 Software Operation....310
44.2.2 Source Code Structure....310
44.2.3 Menu Configuration Options....310
44.3 Programming Interface....311
44.3.1 Device-Specific Defines....311

44.3.2 Getting a MAC Address....312

Chapter 45
ENET IEEE-1588 Driver

45.1 Hardware Operation....313
45.1.1 Transmit Timestamping....314
45.1.2 Receive Timestamping....314
45.2 Software Operation....314
45.2.1 Source Code Structure....315
45.2.2 Linux Menu Configuration Options....315
45.3 Programming Interface....315
45.4 1588 Stack Support....315
45.4.1 1588 Stack Introduction....315
45.4.2 Linuxptp Stack Features....316
45.4.3 How to Use the Stacks in Linux OS....316

Chapter 46
Universal Asynchronous Receiver/Transmitter (UART) Driver

46.1 Introduction....317
46.2 Hardware Operation....318
46.2.1 Software Operation....318
46.2.2 Driver Features....319
46.2.3 Source Code Structure....319
46.3 Configuration....319
46.3.1 Configuration Options....320
46.3.2 Source Code Configuration Options....320
46.3.3 Chip Configuration Options....320
46.3.4 Board Configuration Options....320
46.4 Programming Interface....320
46.4.1 Interrupt Requirements....320

Chapter 47
Wi-Fi BCM4339 Driver

47.1 Hardware Operation....321
47.1.1 Software Operation....321
47.1.2 Driver features....321
47.1.3 Source Code Structure....321
47.1.4 Linux Menu Configuration Options....322
47.2 How to Install the Driver Module....322
47.3 Device Tree Binding....322
47.4 Murata Module Support Status....323

Chapter 48
Pulse-Width Modulator (PWM) Driver

48.1 Introduction....325
48.1.1 Hardware Operation....325
48.1.2 Clocks....326
48.1.3 Software Operation....327
48.1.4 Driver Features....327
48.1.5 Source Code Structure....327
48.1.6 Menu Configuration Options....328

Chapter 49
Watchdog (WDOG) Driver

49.1 Introduction....329
49.1.1 Hardware Operation....329
49.1.2 Software Operation....329
49.2 Generic WDOG Driver....329
49.2.1 Driver Features....330
49.2.2 Menu Configuration Options....330
49.2.3 Source Code Structure....330
49.2.4 Programming Interface....331

Chapter 50
OProfile

50.1 Introduction....333
50.1.1 Overview....333
50.1.2 Features....333
50.1.3 Hardware Operation....334
50.2 Software Operation....334
50.2.1 Architecture-specific Components....335
50.2.2 oprofilefs Pseudo Filesystem....335
50.2.3 Generic Kernel Driver....335
50.2.4 OProfile Daemon....335
50.2.5 Post Profiling Tools....336
50.3 Requirements....336
50.3.1 Source Code Structure....336
50.3.2 Menu Configuration Options....336
50.3.3 Programming Interface....337
50.3.4 Interrupt Requirements....337
50.3.5 Example Software Configuration....337

Chapter 51
CAAM (Cryptographic Acceleration and Assurance Module)

51.1 CAAM Device Driver Overview....339
51.2 Configuration and Job Execution Level....339
51.3 Control/Configuration Driver....340
51.4 Job Ring Driver....340
51.5 API Interface Level....341
51.6 Driver Configuration....344
51.7 Limitations....345
51.8 Limitations in the Existing Implementation Overview....346
51.9 Initialize Keystore Management Interface....346

51.10 Detect Available Secure Memory Storage Units....347
51.11 Establish Keystore in Detected Unit....347
51.12 Release Keystore....348
51.13 Allocate a Slot from the Keystore....348
51.14 Load Data into a Keystore Slot....348
51.15 Demo Image Update....349
51.16 Decapsulate Data in the Keystore....350
51.17 Read Data From a Keystore Slot....350
51.18 Release a Slot back to the Keystore....351
51.19 CAAM/SNVS - Security Violation Handling Interface Overview....353
51.20 Operation....353
51.21 Configuration Interface....354
51.22 Install a Handler....354
51.23 Remove an Installed Driver....354
51.24 Driver Configuration CAAM/SNVS....355

Chapter 52
Remote Processor Messaging (RPMsg)

52.1 Introduction....357
52.2 Features....358
52.3 Source Code....359
52.4 Kernel Configurations....359
52.5 Running i.MX RPMsg Test Programs....359

Chapter 53
Sipix Display Controller (SPDC) Frame Buffer Driver

53.1 Introduction....363
53.2 Hardware Operation....364
53.3 Software Operation....364
53.3.1 SPDC Frame Buffer Driver Overview....364
53.3.2 SPDC Frame Buffer Driver Extensions....365

53.3.3 SPDC Panel Configuration ... 365
53.3.3.1 Boot Command Line Parameters ... 366
53.3.4 SPDC Waveform Loading ... 366
53.3.4.1 Using a Default Waveform File ... 367
53.3.4.2 Using a Custom Waveform File ... 367
53.3.5 SPDC Panel Initialization ... 368
53.3.6 Grayscale Framebuffer Selection ... 368
53.4 Source Code Structure ... 369
53.5 Menu Configuration Options ... 370
53.6 Programming Interface ... 370
53.6.1 IOCTLs/Functions ... 370
53.6.2 Structures and Defines ... 374

Chapter 54
Display Content Integrity Checker (DCIC)

54.1 Introduction ... 375
54.2 Hardware Operation ... 375
54.3 Software Operation ... 375
54.3.1 Source Code Structure ... 375
54.3.2 Menu Configuration Options ... 376
54.3.3 DTS Configuration ... 376
54.4 Programming Interface ... 376
54.4.1 IOCTLs Functions ... 376
54.4.2 Structures ... 376
54.5 Unit Test ... 377
54.5.1 Source Code ... 377
54.5.2 DCIC CRC Calculation Functions ... 377
54.5.3 sample ... 377

Chapter 55
ADC Driver

55.1 ADC Introduction..... 379
55.2 ADC External Signals..... 379
55.3 ADC Driver Overview..... 380
55.3.1 ADC Driver File..... 380
55.3.2 Menu Configuration Options..... 380
55.3.3 Programming Interface..... 380

Chapter 56
Video Analog-to-Digital Converter (VADC)

56.1 Introduction..... 383
56.2 Hardware Operation..... 383
56.3 Software Operation..... 384
56.3.1 Source Code Structure..... 384
56.3.2 Menu Configuration Options..... 384
56.3.3 DTS Configuration..... 384
56.4 Unit Test..... 385

Chapter 57
Bluetooth® BCM4339 Driver

57.1 Bluetooth Introduction..... 387
57.2 Hardware Operation..... 387
57.3 Software Operation..... 387
57.3.1 Bluetooth Driver Overview..... 387
57.3.2 Bluetooth Driver Files..... 388
57.3.3 Bluetooth Stack..... 388
57.3.4 Menu Configuration Options..... 388

Chapter 58
Samsung MIPI DSI Driver

58.1 Introduction..... 391
58.1.1 MIPI DSI IP Driver Overview..... 391

58.1.2 MIPI DSI Display Panel Driver Overview 392
58.1.3 Hardware Operation 392
58.2 Software Operation 392
58.2.1 MIPI DSI IP Driver Software Operation 392
58.2.2 MIPI DSI Display Panel Driver Software Operation 393
58.3 Driver Features 393
58.3.1 Source Code Structure 393
58.3.2 Menu Configuration Options 394
58.3.3 Programming Interface 394

Chapter 1
About this Book

1.1 Audience

This document is targeted to individuals who will port the i.MX Linux® OS Board Support Package (BSP) to customer-specific products.

The audience is expected to have a working knowledge of the Linux OS 3.0 kernel internals, driver models, and i.MX processors.

1.1.1 Conventions

This document uses the following notational conventions:

- Courier monospaced type indicate commands, command parameters, code examples, and file and directory names.
- *Italic* type indicates replaceable command or function parameters.
- **Bold** type indicates function names.
- `<Yocto_BuildDir>` stands for `<Yocto build directory>/tmp/work/<machine-poky-linux-gnueabi>`

1.1.2 Definitions, Acronyms, and Abbreviations

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

Definitions and Acronyms

Term	Definition
ADC	Asynchronous Display Controller
address translation	Address conversion from virtual domain to physical domain
API	Application Programming Interface

Table continues on the next page...

Term	Definition
ARM®	Advanced RISC Machines processor architecture
AUDMUX	Digital audio MUX-provides a programmable interconnection for voice, audio, and synchronous data routing between host serial interfaces and peripheral serial interfaces
BCD	Binary Coded Decimal
bus	A path between several devices through data lines
bus load	The percentage of time a bus is busy
CODEC	Coder/decoder or compression/decompression algorithm-used to encode and decode (or compress and decompress) various types of data
CPU	Central Processing Unit-generic term used to describe a processing core
CRC	Cyclic Redundancy Check-Bit error protection method for data communication
CSI	Camera Sensor Interface
DFS	Dynamic Frequency Scaling
DMA	Direct Memory Access-an independent block that can initiate memory-to-memory data transfers
DPM	Dynamic Power Management
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage Frequency Scaling
EMI	External Memory Interface-controls all IC external memory accesses (read/write/erase/program) from all the masters in the system
Endian	Refers to byte ordering of data in memory. Little endian means that the least significant byte of the data is stored in a lower address than the most significant byte. In big endian, the order of the bytes is reversed
EPIT	Enhanced Periodic Interrupt Timer-a 32-bit set and forget timer capable of providing precise interrupts at regular intervals with minimal processor intervention
FCS	Frame Checker Sequence
FIFO	First In First Out
FIPS	Federal Information Processing Standards-United States Government technical standards published by the National Institute of Standards and Technology (NIST). NIST develops FIPS when there are compelling Federal government requirements such as for security and interoperability but no acceptable industry standards
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, where erasing can be done only in blocks or the entire chip.
Flash path	Path within ROM bootstrap pointing to an executable Flash application
Flush	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 produces the same hash value.
I/O	Input/Output
ICE	In-Circuit Emulation
IP	Intellectual Property

Table continues on the next page...

Term	Definition
IPU	Image Processing Unit -supports video and graphics processing functions and provides an interface to video/ still image sensors and displays
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-16-bit peripheral used as a keypad matrix interface or as general purpose input/output (I/O)
line	Refers to a unit of information in the cache that is associated with a tag
LRU	Least Recently Used-a policy for line replacement in the cache
MMU	Memory Management Unit-a component responsible for memory protection and address translation
MPEG	Moving Picture Experts Group-an ISO committee that generates standards for digital video compression and audio. It is also the name of the algorithms used to compress moving pictures and video
MPEG standards	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 video in applications such as digital television set-top boxes and DVD • MPEG-3 was merged into MPEG-2 • MPEG-4 is a standard for low-bandwidth video telephony and multimedia on the World-Wide Web
MQSPI	Multiple Queue Serial Peripheral Interface-used to perform serial programming operations necessary to configure radio subsystems and selected peripherals
MSHC	Memory Stick Host Controller
NAND Flash	Flash ROM technology-NAND Flash architecture is one of two flash technologies (the other being NOR) used in memory cards such as the Compact Flash cards. NAND is best suited to flash devices requiring high capacity data storage. NAND flash devices offer storage space up to 512-Mbyte and offers faster erase, write, and read capabilities over NOR architecture
NOR Flash	See NAND Flash
PCMCIA	Personal Computer Memory Card International Association-a multi-company organization that has developed a standard for small, credit card-sized devices, called PC Cards. There are three types of PCMCIA cards that have the same rectangular size (85.6 by 54 millimeters), but different widths
physical address	The address by which the memory in the system is physically accessed
PLL	Phase Locked Loop-an electronic circuit controlling an oscillator so that it maintains a constant phase angle (a lock) on the frequency of an input, or reference, signal
RAM	Random Access Memory
RAM path	Path within ROM bootstrap leading to the downloading and the execution of a RAM application
RGB	The RGB color model is based on the additive model in which Red, Green, and Blue light are combined to create other colors. The abbreviation RGB comes from the three primary colors in additive light models
RGBA	RGBA color space stands for Red Green Blue Alpha. The alpha channel is the transparency channel, and is unique to this color space. RGBA, like RGB, is an additive color space, so the more of a color placed, the lighter the picture gets. PNG is the best known image format that uses the RGBA color space
RNGA	Random Number Generator Accelerator-a security hardware module that produces 32-bit pseudo random numbers as part of the security module
ROM	Read Only Memory
ROM bootstrap	Internal boot code encompassing the main boot flow as well as exception vectors

Table continues on the next page...

Term	Definition
RTIC	Real-Time Integrity Checker-a security hardware module
SCC	SeCurity Controller-a security hardware module
SDMA	Smart Direct Memory Access
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on a Chip
SPBA	Shared Peripheral Bus Arbiter-a three-to-one IP-Bus arbiter, with a resource-locking mechanism
SPI	Serial Peripheral Interface-a full-duplex synchronous serial interface for connecting low-/medium-bandwidth external devices using four wires. SPI devices communicate using a master/slave relationship over two data lines and two control lines: *Also see SS, SCLK, MISO, and MOSI*
SRAM	Static Random Access Memory
SSI	Synchronous-Serial Interface-standardized interface for serial data transfer
TBD	To Be Determined
UART	Universal Asynchronous Receiver/Transmitter-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 themselves depending on how the cables are connected to the devices, and they also can connect to a host PC
word	A group of bits comprising 32-bits

Chapter 2
Introduction

2.1 Overview

The i.MX family Linux Board Support Package (BSP) supports the Linux Operating System (OS) on the following processors:

i.MX 6Dual/6Quad/6Solo/6DualLite/6SoloLite/6SoloX/6UltaLite/7Dual applications processor

The purpose of this software package is to support Linux OS on the i.MX 6Dual/6Quad/6Solo/6DualLite/6SoloLite/6UltaLite/7Dual family of Integrated Circuits (ICs) and their associated platforms. It provides the necessary software to interface the standard open-source Linux kernel to the i.MX hardware. The goal is to enable Freescale customers to rapidly build products based on i.MX devices that use the Linux OS.

The BSP is not a platform or product reference implementation. It does not contain all of the product-specific drivers, hardware-independent software stacks, Graphical User Interface (GUI) components, Java Virtual Machine (JVM), and applications required for a product. Some of these are made available in their original open-source form as part of the base kernel.

The BSP is not intended to be used for silicon verification. While it can play a role in this, the BSP functionality and the tests run on the BSP do not have sufficient coverage to replace traditional silicon verification test suites.

2.1.1 Software Base

The i.MX BSP is based on version 3.14.52 of the Linux kernel from the official Linux kernel website (www.kernel.org). It is enhanced with the features provided by Freescale.

2.1.2 Features

Table below describes the features supported by the Linux BSP for specific platforms.

Table 2-1. Linux BSP Supported Features

Feature	Description	Chapter Source	Applicable Platform
Machine-Specific Layer			
MSL	Machine-Specific Layer (MSL) supports interrupts, Timer, Memory Map, GPIO/IOMUX, SPBA, SDMA. • Interrupts GIC: The Linux kernel contains common ARM GIC interrupts handling code. • Timer (GPT): The General Purpose Timer (GPT) is set up to generate an interrupt as programmed to provide OS ticks. Linux OS facilitates timer use through various functions for timing delays, measurement, events, alarms, high resolution timer features, and so on. Linux OS defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation. • GPIO/EDIO/IOMUX: The GPIO and EDIO components in the MSL provide an abstraction layer between the various drivers and the configuration and utilization of the system, including GPIO, IOMUX, and external board I/O. The IO software module is board-specific, and resides in the MSL layer as a self-contained set of files. I/O configuration changes are centralized in the GPIO module so that changes are not required in the various drivers. • SPBA: The Shared Peripheral Bus Arbiter (SPBA) provides an arbitration mechanism among multiple masters to allow access to the shared peripherals. The SPBA implementation under MSL defines the API to allow different masters to take or release ownership of a shared peripheral.	Machine-Specific Layer (MSL)	All
SDMA API	The Smart Direct Memory Access (SDMA) API driver controls the SDMA hardware. It provides an API to other drivers for transferring data between MCU, DSP and peripherals. . The SDMA controller is responsible for transferring data between the MCU memory space, peripherals, and the DSP memory space. The SDMA API allows other drivers to initialize the scripts, pass parameters and control their execution. SDMA is based on a microRISC engine that runs channel-specific scripts.	Smart Direct Memory Access (SDMA) API	All
DMAC	Both AHB-to-APBH and AHB-to-APBX DMA support configurable DMA descript chain.	AHB-to-APBH Bridge with DMA (APBH-Bridge-DMA)	All
Low-level PM Drivers	The low-level power management driver is responsible for implementing hardware-specific operations to meet power requirements and also to conserve power on the	Low-level Power Management (PM) Driver	All

Table continues on the next page...

Table 2-1. Linux BSP Supported Features (continued)

Feature	Description	Chapter Source	Applicable Platform
	development platforms. Driver implementations are often different for different platforms. It is used by the DPM layer.		
CPU Frequency Scaling	The CPU frequency scaling device driver allows the clock speed of the CPUs to be changed on the fly.	CPU Frequency Scaling (CPUFREQ) Driver	All
Dynamic Bus Frequency Driver	In order to improve power consumption, the Bus Frequency driver dynamically manages the various system frequencies.	Dynamic Bus Frequency Driver	All
Multimedia Drivers			
LCD	The LCD interface driver supports the Samsung LMS430xx 4.3" WQVGA LCD panel.	ELCDIF Frame Buffer Driver	i.MX 6SoloLite, i.MX 6UltraLite, i.MX 7Dual
EPDC	The Electrophoretic Display Controller (EPDC) is a direct-drive active matrix EPD controller designed to drive E Ink EPD panels supporting a wide variety of TFT backplanes.	Electrophoretic Display Controller (EPDC) Frame Buffer	i.MX 6DualLite, i.MX 6Solo, i.MX 6SoloLite, i.MX 7Dual
PxP	The Pixel Pipeline (PxP) DMA-ENGINE driver provides a unique API, which are implemented as a DMA engine client that smooths over the details of different hardware offload engine implementations.	PXP DMA-ENGINE Driver	i.MX 6DualLite, i.MX 6Solo, i.MX 6SoloLite, i.MX 6UltraLite, i.MX 7Dual
IPU	The Image Processing Unit (IPU) is designed to support video and graphics processing functions and to interface with video/still image sensors and displays. The IPU driver is a self-contained driver module in the Linux kernel. It contains a custom kernel-level API to manipulate logical channels. A logical channel represents a complete IPU processing flow. The IPU driver includes a frame buffer driver, a V4L2 device driver, and low-level IPU drivers.	Image Processing Unit (IPU) Drivers	i.MX 6Quad, i.MX 6Dual, i.MX 6DualLite, i.MX 6Solo, i.MX 6UltraLite, i.MX 7Dual
HDMI	This driver provides the support HDMI module	HDMI Driver	All
V4L2 Output	The Video for Linux 2 (V4L2) output driver uses the IPU post-processing functions for video output. The driver implements the standard V4L2 API for output devices.	Video for Linux Two (V4L2) Driver	All
V4L2 Capture	The Video for Linux 2 (V4L2) capture device includes two interfaces: the capture interface and the overlay interface. The capture interface records the video stream. The overlay interface displays the preview video.	Video for Linux Two (V4L2) Driver	All

Table continues on the next page...

Table 2-1. Linux BSP Supported Features (continued)

Feature	Description	Chapter Source	Applicable Platform
VPU	The Video Processing Unit (VPU) is a multi-standard video decoder and encoder that can perform decoding and encoding of various video formats.	Video Processing Unit (VPU) Driver	i.MX 6Quad, i.MX 6Dual, i.MX 6DualLite, i.MX 6Solo
Sound Drivers			
ALSA Sound	The Advanced Linux Sound Architecture (ALSA) is a sound driver that provides ALSA and OSS compatible applications with the means to perform audio playback and recording functions. ALSA has a user-space component called ALSAlib that can extend the features of audio hardware by emulating the same in software (user space), such as resampling, software mixing, snooping, and so on. The ASoC Sound driver supports stereo CODEC playback and capture through SSI.	ALSA Sound Driver	All
S/PDIF	The S/PDIF driver is designed under the Linux ALSA subsystem. It implements one playback device for Tx and one capture device for Rx.	The Sony/Philips Digital Interface (S/PDIF) Driver	All
Memory Drivers			
SPI NOR MTD	The SPI NOR MTD driver provides the support to the Atmel data Flash using the SPI interface.	SPI NOR Flash Memory Technology Device (MTD) Driver	All
NAND MTD	The NAND MTD driver interfaces with the integrated NAND controller. It can support various file systems, such as UBIFS, CRAMFS and JFFS2UBI and UBIFSCRAMFS and JFFS2. The driver implementation supports the lowest level operations on the external NAND Flash chip, such as block read, block write and block erase as the NAND Flash technology only supports block access. Because blocks in a NAND Flash are not guaranteed to be good, the NAND MTD driver is also able to detect bad blocks and feed that information to the upper layer to handle bad block management.	NAND GPMI Flash Driver	i.MX 6Quad, i.MX 6Dual, i.MX 6DualLite, i.MX 6Solo, i.MX 6UltraLite, i.MX 7Dual
SATA	The SATA AHCI driver is based on the LIBATA layer of the block device infrastructure of the Linux kernel	SATA Driver	i.MX 6Quad, i.MX 6Dual
Input Device Drivers			
Networking Drivers			
ENET	The ENET Driver performs the full set of IEEE 802.3/ Ethernet CSMA/CD media access control and channel interface functions. The FEC requires an external interface adaptor and transceiver function to complete the interface to the Ethernet media. It supports half or full-duplex operation on 10M\100M\1G related Ethernet networks.	Fast Ethernet Controller (FEC) Driver	All
Bus Drivers			
I^2C	The I2C bus driver is a low-level interface that is used to interface with the I2C bus. This driver is invoked by the I2C chip driver; it is not exposed to the user space. The standard Linux kernel contains a core I2C module	Inter-IC (I2C) Driver	All

Table continues on the next page...

Table 2-1. Linux BSP Supported Features (continued)

Feature	Description	Chapter Source	Applicable Platform
	that is used by the chip driver to access the bus driver to transfer data over the I2C bus. This bus driver supports: • Compatibility with the I2C bus standard • Bit rates up to 400 Kbps • Standard I2C master mode • Power management features by suspending and resuming I2C.		
CSPI	The low-level Enhanced Configurable Serial Peripheral Interface (ECSPI) driver interfaces a custom, kernel-space API to both ECSPI modules. It supports the following features: • Interrupt-driven transmit/receive of SPI frames • Multi-client management • Priority management between clients • SPI device configuration per client	Enhanced Configurable Serial Peripheral Interface (ECSPI) Driver	All
MMC/SD/SDIO - uSDHC	The MMC/SD/SDIO Host driver implements the standard Linux driver interface to eSDHC.	MMC/SD/SDIO Host Driver	All
UART Drivers			
MXC UART	The Universal Asynchronous Receiver/Transmitter (UART) driver interfaces the Linux serial driver API to all of the UART ports. A kernel configuration parameter gives the user the ability to choose the UART driver and also to choose whether the UART should be used as the system console.	Universal Asynchronous Receiver/ Transmitter (UART) Driver	All
General Drivers			
USB	The USB driver implements a standard Linux driver interface to the ARC USB-OTG controller.	CHIPIDEA USB Driver	All
FlexCAN	The FlexCAN driver is designed as a network device driver. It provides the interfaces to send and receive CAN messages. The CAN protocol was primarily designed to be used as a vehicle serial data bus, meeting the specific requirements of this field: real-time processing, reliable operation in the EMI environment of a vehicle, cost-effectiveness and required bandwidth.	FlexCAN Driver	i.MX 6Quad, i.MX 6Dual, i.MX 6DualLite, i.MX 6Solo
ASRC	The Asynchronous Sample Rate Converter (ASRC) driver provides the interfaces to access the asynchronous sample rate converter module.	Asynchronous Sample Rate Converter (ASRC) Driver	i.MX 6Quad, i.MX 6Dual, i.MX 6DualLite, i.MX 6Solo
WatchDog	The Watchdog Timer module protects against system failures by providing an escape from unexpected hang or infinite loop situations or programming errors. This WDOG implements the following features: • Generates a reset signal if it is enabled but not serviced within a predefined time-out value • Does not generate a reset signal if it is serviced within a predefined time-out value	Watchdog (WDOG) Driver	All

Table continues on the next page...

Table 2-1. Linux BSP Supported Features (continued)

Feature	Description	Chapter Source	Applicable Platform
MXC PWM driver	The MXC PWM driver provides the interfaces to access MXC PWM signals	Pulse-Width Modulator (PWM) Driver	All
Thermal Driver	Thermal driver is a necessary driver for monitoring and protecting the SoC. The thermal driver will monitor the SoC's temperature in a certain frequency. It defines three trip points: critical, hot, and active.	Thermal Driver	All
OProfile	OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.	OProfile	All

Chapter 3
Machine Specific Layer (MSL)

3.1 Introduction

The Machine Specific Layer (MSL) provides the Linux kernel with the machine-dependent components found here.

- Interrupts including GPIO and EDIO (only on certain platforms)
- Timer
- Memory map
- General Purpose Input/Output (GPIO) including IOMUX on certain platforms
- Shared Peripheral Bus Arbiter (SPBA)
- Smart Direct Memory Access (SDMA)

These modules are normally available in the following directory:

```
<Yocto_BuildDir>/linux/arch/arm/mach-imx for the i.MX 6 and i.MX 7 platforms
```

The MSL layer contains not only the modules common to all the boards using the same processor, such as the interrupts and timer, but it also contains modules specific to each board, such as the memory map. The following sections describe the basic hardware and software operation and the software interfaces for MSL modules. First, the common modules, such as Interrupts and Timer are discussed. Next, the board-specific modules, such as Memory Map and General Purpose Input/Output (GPIO) (including IOMUX on some platforms) are detailed. Because of the complexity of the SDMA module, its design is explained in SDMA relevant chapter.

Each of the following sections contains an overview of the hardware operation. For more information, see the corresponding device documentation.

3.2 Interrupts (Operation)

This section describes the hardware and software operation of interrupts on the device.

3.2.1 Interrupt Hardware Operation

The Interrupt Controller controls and prioritizes a maximum of 128 internal and external interrupt sources.

Each source can be enabled or disabled by configuring the Interrupt Enable Register or using the Interrupt Enable/Disable Number Registers. When an interrupt source is enabled and the corresponding interrupt source is asserted, the Interrupt Controller asserts a normal or a fast interrupt request depending on the associated Interrupt Type Register setting.

Interrupt Controller registers can only be accessed in supervisor mode. The Interrupt Controller interrupt requests are prioritized in the following order: fast interrupts and normal interrupts in order of highest priority level, then highest source number with the same priority. There are sixteen normal interrupt levels for all interrupt sources, with level zero being the lowest priority. The interrupt levels are configurable through eight normal interrupt priority level registers. Those registers, along with the Normal Interrupt Mask Register, support software-controlled priority levels for normal interrupts and priority masking.

3.2.2 Interrupt Software Operation

For ARM architecture-based processors, normal interrupt and fast interrupt are two different exception types. The exception vector addresses can be configured to start at low address (0x0) or high address (0xFFFF0000).

The Linux OS implementation running on ARM architecture chooses the high vector address model.

The following file has a description of the ARM interrupt architecture.

```
<Yocto_BuildDir>/linux/Documentation/arm/Interrupts
```

The software provides a processor-specific interrupt structure with callback functions defined in the irqchip structure and exports one initialization function, which is called during system startup.

3.2.3 Interrupt Features

The interrupt implementation supports the following features:

- Interrupt Controller interrupt disable and enable
- Functions required by the Linux interrupt architecture as defined in the standard ARM interrupt source code (mainly the <Yocto_BuildDir>/linux/arch/arm/kernel/irq.c file)

3.2.4 Interrupt Source Code Structure

The interrupt module is implemented in the following file (located in the directory <Yocto_BuildDir>/linux/arch/arm/plat-mxc):

```
irq.c (If CONFIG_MXC_TZIC is not selected)
tzic.c (If CONFIG_MXC_TZIC is selected)
gic.c (If CONFIG_ARM_GIC is selected)
gpc.c (If CONFIG_MXC is selected)
```

There are also two header files (located in the include directory specified at the beginning of this chapter):

```
hardware.h
irqs.h
```

Table below lists the source files for interrupts.

Table 3-1. Interrupt Files

File	Description
hardware.h	Register descriptions
irqs.h	Declarations for number of interrupts supported
gic.c	Actual interrupt functions for GIC modules

3.2.5 Interrupt Programming Interface

The machine-specific interrupt implementation exports a single function.

This function initializes the Interrupt Controller hardware and registers functions for interrupt enable and disable from each interrupt source.

This is done with the global structure irq_desc of type struct irqdesc. After the initialization, the interrupt can be used by the drivers through the request_irq() function to register device-specific interrupt handlers.

In addition to the native interrupt lines supported from the Interrupt Controller, the number of interrupts is also expanded to support GPIO interrupt and (on some platforms) EDIO interrupts. This allows drivers to use the standard interrupt interface supported by ARM device running Linux OS, such as the request_irq() and free_irq() functions.

3.3 Timer

The Linux kernel relies on the underlying hardware to provide support for both the system timer (which generates periodic interrupts) and the dynamic timers (to schedule events).

After the system timer interrupt occurs, it does the following:

- Updates the system uptime
- Updates the time of day
- Reschedules a new process if the current process has exhausted its time slice
- Runs any dynamic timers that have expired
- Updates resource usage and processor time statistics

The timer hardware on most i.MX platforms consists of either Enhanced Periodic Interrupt Timer (EPIT) or general purpose timer (GPT) or both. GPT is configured to generate a periodic interrupt at a certain interval (every 10 ms) and is used by the Linux kernel.

3.3.1 Timer Software Operation

The timer software implementation provides an initialization function that initializes the GPT with the proper clock source, interrupt mode and interrupt interval.

The timer then registers its interrupt service routine and starts timing. The interrupt service routine is required to service the OS for the purposes mentioned in Timer. Another function provides the time elapsed as the last timer interrupt.

3.3.2 Timer Features

The timer implementation supports the following features:

- Functions required by Linux OS to provide the system timer and dynamic timers.
- Generates an interrupt every 10 ms.

3.3.3 Timer Source Code Structure

The timer module is implemented in the arch/arm/mach-imx/time.c file.

3.3.4 Timer Programming Interface

The timer module utilizes four hardware timers, to implement clock source and clock event objects.

This is done with the clocksource_mxc structure of struct clocksource type and clockevent_mxc structure of struct clockevent_device type. Both structures provide routines required for reading current timer values and scheduling the next timer event. The module implements a timer interrupt routine that services the Linux OS with timer events for the purposes mentioned in the beginning of this chapter.

3.4 Memory Map

A predefined virtual-to-physical memory map table is required for the device drivers to access to the device registers since the Linux kernel is running under the virtual address space with the Memory Management Unit (MMU) enabled.

3.4.1 Memory Map Hardware Operation

The MMU, as part of the ARM core, provides the virtual to physical address mapping defined by the page table. For more information, see the *ARM Technical Reference Manual* (TRM) from ARM Limited.

3.4.2 Memory Map Software Operation

A table mapping the virtual memory to physical memory is implemented for i.MX platforms as defined in the <Yocto_BuildDir>/arch/arm/mach-imx/pm-imx*.cfile.

3.4.3 Memory Map Features

The Memory Map implementation programs the Memory Map module to creates the physical to virtual memory map for all the I/O modules.

3.4.4 Memory Map Source Code Structure

The Memory Map module implementation is in pm-imx*.c under the platform-specific MSL directory. The hardware.h header file is used to provide macros for all the I/O module physical and virtual base addresses and physical to virtual mapping macros. All of the memory map source code is in the in the following file:

```
<Yocto_BuildDir>/arch/arm/mach-imx/pm-imx*.c
```

Table below lists the source file for the memory map.

Table 3-2. Memory Map Files

File	Description
mx6.h, mx7.h	Header files for the I/O module physical addresses
hardware.h	Memory map definition file

3.5 IOMUX

The limited number of pins of highly integrated processors can have multiple purposes.

The IOMUX module controls a pin usage so that the same pin can be configured for different purposes and can be used by different modules.

This is a common way to reduce the pin count while meeting the requirements from various customers. Platforms that do not have the IOMUX hardware module can do pin muxing through the GPIO module.

The IOMUX module provides the multiplexing control so that each pin may be configured either as a functional pin or as a GPIO pin. A functional pin can be subdivided into either a primary function or alternate functions. The pin operation is controlled by a specific hardware module. A GPIO pin, is controlled by the user through software with further configuration through the GPIO module. For example, the TXD1 pin might have the following functions:

- TXD1-internal UART1 Transmit Data. This is the primary function of this pin.
- UART2 DTR-alternate mode 3

- LCDC_CLS-alternate mode 4
- GPIO4[22]-alternate mode 5
- SLCDC_DATA[8]-alternate mode 6

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose and cannot be changed by software. Otherwise, the IOMUX module needs to be configured to serve a particular purpose that is dictated by the system (board) design. If the pin is connected to an external UART transceiver and therefore to be used as the UART data transmit signal, it should be configured as the primary function. If the pin is connected to an external Ethernet controller for interrupting the ARM core, then it should be configured as GPIO input pin with interrupt enabled. Again, be aware that the software does not have control over what function a pin should have. The software only configures pin usage according to the system design.

3.5.1 IOMUX Hardware Operation

The following discussion applies only to those processors that have an IOMUX hardware module.

The IOMUX controller registers are briefly described in this section.

For detailed information, see the pin multiplexing section of the IC Reference Manual.

- SW_MUX_CTL-Selects the primary or alternate function of a pin. Also enables loopback mode when applicable.
- SW_SELECT_INPUT-Controls pin input path. This register is only required when multiple pads drive the same internal port.
- SW_PAD_CTL-Control pad slew rate, driver strength, pull-up/down resistance, and so on.

3.5.2 IOMUX Software Operation

The IOMUX software implementation provides an API to set up pin functionality and pad features.

3.5.3 IOMUX Features

The IOMUX implementation programs the IOMUX module to configure the pins that are supported by the hardware.

3.5.4 IOMUX Source Code Structure

Table below lists the source files for the IOMUX module. The files are in the following directories:

- <Yocto_BuildDir>/drivers/pinctrl/pinctrl-imx.c
- <Yocto_BuildDir>/drivers/pinctrl/pinctrl-imx6sl.c
- <Yocto_BuildDir>/drivers/pinctrl/pinctrl-imx6q.c
- <Yocto_BuildDir>/drivers/pinctrl/pinctrl-imx6sx.c
- <Yocto_BuildDir>/drivers/pinctrl/pinctrl-imx6ul.c
- <Yocto_BuildDir>/drivers/pinctrl/pinctrl-imx7d.c

Table 3-3. IOMUX Files

File	Description
pinctrl-imx.c	i.MX pinctrl core driver
pinctrl-im6sl.c	i.MX 6SoloLite pinctrl driver
pinctrl-imx6q.c	i.MX 6Quad/DualLite pinctrl driver
pinctrl-imx6sx.c	i.MX 6SoloX pinctrl driver
pinctrl-imx6ul.c	i.MX 6UltraLite pinctrl driver
pinctrl-imx7d.c	i.MX 7Dual pinctrl driver

3.5.5 IOMUX Programming Interface

See pinctrl binding documents:

- imx-pinctrl.txt in Documentation/devicetree/bindings/pinctrl/fsl
- imx6sl-pinctrl.txt in Documentation/devicetree/bindings/pinctrl/fsl

3.5.6 IOMUX Control Through GPIO Module

For a multi-purpose pin, the GPIO controller provides the multiplexing control so that each pin may be configured either as a functional pin, or a GPIO pin.

The operation of the functional pin, which can be subdivided into either major function or one alternate function, is controlled by a specific hardware module. If it is configured as a GPIO pin, the pin is controlled by the user through software with further configuration through the GPIO module. In addition, there are some special configurations for a GPIO pin (such as output based A_IN, B_IN, C_IN or DATA register, but input based A_OUT or B_OUT).

The following discussion applies to those platforms that control the muxing of a pin through the general purpose input/output (GPIO) module.

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose which cannot be changed by software. Otherwise, the GPIO module needs to be configured properly to serve a particular purpose that is dictated with the system (board) design. If this pin is connected to an external UART transceiver, it should be configured as the primary function or if this pin is connected to an external Ethernet controller for interrupting the core, then it should be configured as GPIO input pin with interrupt enabled. The software does not have control over what function a pin should have. The software only configures a pin for that usage according to the system design.

3.5.6.1 GPIO Hardware Operation

The GPIO controller module is divided into MUX control and PULLUP control sub modules. The following sections briefly describe the hardware operation. For detailed information, refer to the relevant device documentation.

3.5.6.1.1 Muxing Control

The GPIO In Use Registers control a multiplexer in the GPIO module.

The settings in these registers choose if a pin is utilized for a peripheral function or for its GPIO function. One 32-bit general purpose register is dedicated to each GPIO port. These registers may be used for software control of IOMUX block of the GPIO.

3.5.6.1.2 PULLUP Control

The GPIO module has a PULLUP control register (PUEN) for each GPIO port to control every pin of that port.

3.5.6.2 GPIO Software Operation (general)

The GPIO software implementation provides an API to setup pin functionality and pad features.

3.5.6.3 GPIO Implementation

The GPIO implementation programs the GPIO module to configure the pins that are supported by the hardware.

3.6 General Purpose Input/Output(GPIO)

The GPIO module provides general-purpose pins that can be configured as either inputs or outputs.

When configured as an output, the pin state (high or low) can be controlled by writing to an internal register. When configured as an input, the pin input state can be read from an internal register.

3.6.1 GPIO Software Operation

The general purpose input/output (GPIO) module provides an API to configure the i.MX processor external pins and a central place to control the GPIO interrupts.

The GPIO utility functions should be called to configure a pin instead of directly accessing the GPIO registers. The GPIO interrupt implementation contains functions, such as the interrupt service routine (ISR) registration/un-registration and ISR dispatching once an interrupt occurs. All driver-specific GPIO setup functions should be made during device initialization in the MSL layer to provide better portability and maintainability. This GPIO interrupt is initialized automatically during the system startup.

If a pin is configured as GPIO by the IOMUX, the state of the pin should also be set since it is not initialized by a dedicated hardware module. Setting the pad pull-up, pull-down, slew rate and so on, with the pad control function may be required as well.

3.6.1.1 API for GPIO

API for GPIO lists the features supported by the GPIO implementation.

The GPIO implementation supports the following features:

- An API for registering an interrupt service routine to a GPIO interrupt. This is made possible as the number of interrupts defined by NR_IRQS is expanded to accommodate all the possible GPIO pins that are capable of generating interrupts.
- Functions to request and free an IOMUX pin. If a pin is used as GPIO, another set of request/free function calls are provided. The user should check the return value of the request calls to see if the pin has already been reserved before modifying the pin state. The free function calls should be made when the pin is not needed. See the API document for more details.

- Aligned parameter passing for both IOMUX and GPIO function calls. In this implementation the same enumeration for iomux_pins is used for both IOMUX and GPIO calls and the user does not have to figure out in which bit position a pin is located in the GPIO module.
- Minimal changes required for the public drivers such as Ethernet and UART drivers as no special GPIO function call is needed for registering an interrupt.

3.6.2 GPIO Features

This GPIO implementation supports the following features:

- Implements the functions for accessing the GPIO hardware modules
- Provides a way to control GPIO signal direction and GPIO interrupts

3.6.3 GPIO Module Source Code Structure

All of the GPIO module source code is in the GPIO framework, in the following files, located in the directories indicated at the beginning of this chapter:

Table 3-4. GPIO Files

File	Description
drivers/gpio/gpio-mxc.c	Function implementation

3.6.4 GPIO Programming Interface 2

For more information, see the Documentation/gpio.txt under Linux source code directory for the programming interface.

Chapter 4
Smart Direct Memory Access (SDMA) API

4.1 Overview

The Smart Direct Memory Access (SDMA) API driver controls the SDMA hardware.

It provides an API to other drivers for transferring data between MCU memory space and the peripherals. It supports the following features:

- Loading channel scripts from the MCU memory space into SDMA internal RAM
- Loading context parameters of the scripts
- Loading buffer descriptor parameters of the scripts
- Controlling execution of the scripts
- Callback mechanism at the end of script execution

4.1.1 Hardware Operation

The SDMA controller is responsible for transferring data between the MCU memory space and peripherals and includes the following features:

- Multi-channel DMA supporting up to 32 time-division multiplexed DMA channels.
- Powered by a 16-bit Instruction-Set micro-RISC engine.
- Each channel executes specific script.
- Very fast context-switching with two-level priority based preemptive multi-tasking.
- 4 Kbytes ROM containing startup scripts (that is, boot code) and other common utilities that can be referenced by RAM-located scripts.
- 8 Kbyte RAM area is divided into a processor context area and a code space area used to store channel scripts that are downloaded from the system memory.

4.1.2 Software Operation

The driver provides an API for other drivers to control SDMA channels. SDMA channels run dedicated scripts according to peripheral and transfer types. The SDMA API driver is responsible for loading the scripts into SDMA memory, initializing the channel descriptors, and controlling the buffer descriptors and SDMA registers.

The table below provides a list of drivers that use SDMA and the number of SDMA physical channels used by each driver. A driver can specify the SDMA channel number that it wishes to use, static channel allocation, or can have the SDMA driver provide a free SDMA channel for the driver to use, dynamic channel allocation. For dynamic channel allocation, the list of SDMA channels is scanned from channel 32 to channel 1. Upon finding a free channel, that channel is allocated for the requested DMA transfers.

Table 4-1. SDMA Channel Usage

Driver Name	Number of SDMA Channels	SDMA Channel Used
SDMA CMD	1	Static Channel allocation-uses SDMA channels 0
SSI	2 per device	Dynamic channel allocation
UART	2 per device	Dynamic channel allocation
SPDIF	2 per device	Dynamic channel allocation
ESAI	2 per device	Dynamic channel allocation

4.1.3 Source Code Structure

The dmaengine.h (header file for SDMA API) is available in the directory linux/include/linux

The table below shows the source files available in the directory / <Yocto_BuildDir>/linux/drivers/dma

Table 4-2. SDMA API Source Files

File	Description
dmaengine.c	SDMA management routine
imx-sdma.c	SDMA implement driver

The table below shows the image files available in the directory / <Yocto_BuildDir>/linux/firmware/imx/sdma

Table 4-3. SDMA Script Files

File	Description
sdma-mx6q-to1.bin.ihex	SDMA RAM scripts

4.1.4 Programming Interface

The module implements standard DMA API. Refer to the API documents, which are included in the Linux documentation package, for more information on the functions implemented in the driver. For additional information, you can refer to the ESAI driver.

4.1.5 Usage Example

Refer to one of the drivers, such as SPDIF driver, UART driver or SSI driver, that uses the SDMA API driver for a usage example.

Chapter 5
AHB-to-APBH Bridge with DMA (APBH-Bridge-DMA)

5.1 Overview

The AHB-to-APBH bridge provides the processor with an inexpensive peripheral attachment bus running on the AHB's HCLK.

(The H in APBH denotes that the APBH is synchronous to HCLK.)

The AHB-to-APBH bridge includes the AHB-to-APB PIO bridge for a memory-mapped I/O to the APB devices, as well as a central DMA facility for devices on this bus and a vectored interrupt controller for the ARM core. Each one of the APB peripherals, including the vectored interrupt controller, is documented in their own chapters elsewhere in this document.

There is no separate DMA bus for these devices. Contention between the DMA's use of the APBH bus and the AHB-to-APB bridge functions' use of the APBH is mediated by an internal arbitration logic. For contention between these two units, the DMA is favored and the AHB slave will report "not ready" through its HREADY output until the bridge transfer can complete. The arbiter tracks repeated lockouts and inverts the priority, guaranteeing the ARM platform every fourth transfer on the APB

5.1.1 Hardware Operation

The SDMA controller is responsible for transferring data between the MCU memory space and peripherals and includes the following features.

- Multi-channel DMA supporting up to 32 time-division multiplexed DMA channels
- Powered by a 16-bit Instruction-Set micro-RISC engine
- Each channel executes specific script
- Very fast context-switching with two-level priority based preemptive multi-tasking

- 4 Kbytes ROM containing startup scripts (that is, boot code) and other common utilities that can be referenced by RAM-located scripts
- 8 Kbyte RAM area is divided into a processor context area and a code space area used to store channel scripts that are downloaded from the system memory.

5.1.2 Software Operation

The DMA supports sixteen channels of DMA services, as shown in the following table. The shared DMA resource allows each independent channel to follow a simple chained command list. Command chains are built up using the general structure.

Table 5-1. APBH DMA Channel Assignments

APBH DMA CHANNEL #	USAGE
0	GPMI0
1	GPMI1
2	GPMI2
3	GPMI3
4	GPMI4
5	GPMI5
6	GPMI6
7	GPMI7
8	EMPTY
9	EMPTY
10	EMPTY
11	EMPTY
12	EMPTY
13	EMPTY
14	EMPTY
15	EMPTY

5.1.3 Source Code Structure

The table below shows the source files available in the directory, drivers/dma/

Table 5-2. APBH DMA Source Files

File	Description
mxs-dma.c	APBH DMA implement driver

5.1.4 Menu Configuration Options

The following Linux kernel configuration option is provided for this module:

- MXS_DMA -This is the configuration option for the APBH DMA driver. In menuconfig, this option is available under:
- Device Drivers > DMA Engine support > MXS DMA support.

5.1.5 Programming Interface

The module implements standard DMA API. Refer to the API documents, which are located in the Linux documentation package, for more information on the functions implemented in the driver such as GPMI NAND driver.

5.1.6 Usage Example

Refer to one of the drivers, such as GPMI NAND driver, that uses the APBH DMA driver for a usage example.

Chapter 6
Image Processing Unit (IPU) Drivers

6.1 Introduction

The image processing unit (IPU) is designed to support video and graphics processing functions and to interface with video and still image sensors and displays. The IPU driver provides a kernel-level API to manipulate logical channels. A logical channel represents a complete IPU processing flow. For example, a complete IPU processing flow (logical channel) might consist of reading a YUV buffer from memory, performing post-processing, and writing an RGB buffer to memory. A logical channel maps one to three IDMA channels and maps to either zero or one IC tasks. A logical channel can have one input, one output, and one secondary input IDMA channel. The IPU API consists of a set of common functions for all channels. Its functions are to initialize channels, set up buffers, enable and disable channels, link channels for auto frame synchronization, and set up interrupts.

Typical logical channels include:

- CSI direct to memory
- CSI to viewfinder pre-processing to memory
- Memory to viewfinder pre-processing to memory
- Memory to viewfinder rotation to memory
- Previous field channel of memory to video deinterlacing and viewfinder pre-processing to memory
- Current field channel of memory to video deinterlacing and viewfinder pre-processing to memory
- Next field channel of memory to video deinterlacing and viewfinder pre-processing to memory
- CSI to encoder pre-processing to memory
- Memory to encoder pre-processing to memory
- Memory to encoder rotation to memory
- Memory to post-processing rotation to memory
- Memory to synchronous frame buffer background

- Memory to synchronous frame buffer foreground
- Memory to synchronous frame buffer DC
- Memory to synchronous frame buffer mask

The IPU API has some additional functions that are not common across all channels, and are specific to an IPU sub-module. The types of functions for the IPU sub-modules are as follows:

- Synchronous frame buffer functions
- Panel interface initialization
- Set foreground positions
- Set local/global alpha and color key
- Set gamma
- CSI functions
- Sensor interface initialization
- Set sensor clock
- Set capture size
- Enable or disable prefetching linear frames by using PRE/PRG
- Enable or disable resolving tiled frames by using PRE/PRG

The higher level drivers are responsible for memory allocation, chaining of channels, and providing user-level API.

6.1.1 Hardware Operation

The detailed hardware operation of the IPU is discussed in the Applications Processor Reference Manual. The following figure shows the IPU hardware modules.

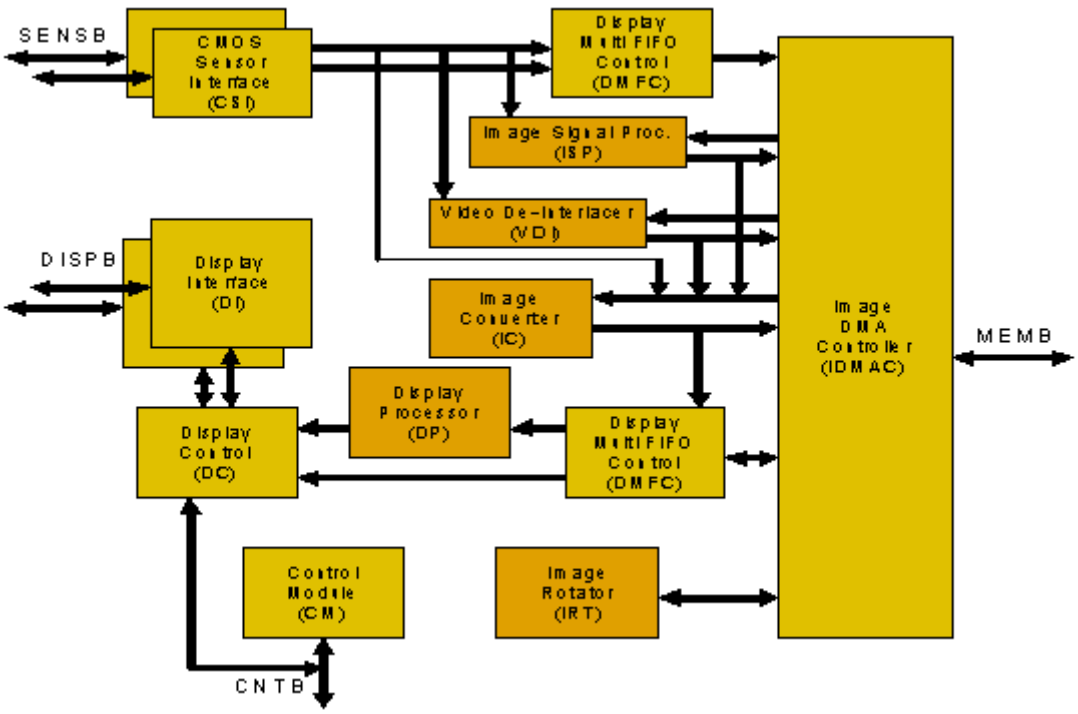


Figure 6-1. IPUv3EX/IPUv3H IPU Module Overview

6.2 Software Operation

The IPU driver is a self-contained driver module in the Linux kernel.

It consists of a custom kernel-level API for the following blocks:

- Synchronous frame buffer driver
- Display Interface (DI)
- Display Processor (DP)
- Image DMA Controller (IDMAC)
- CMOS Sensor Interface (CSI)
- Image Converter (IC)
- Prefetch/Resolve Engine/Gasket (PRE/PRG)

Figure below shows the interaction between the different graphics/video drivers and the IPU.

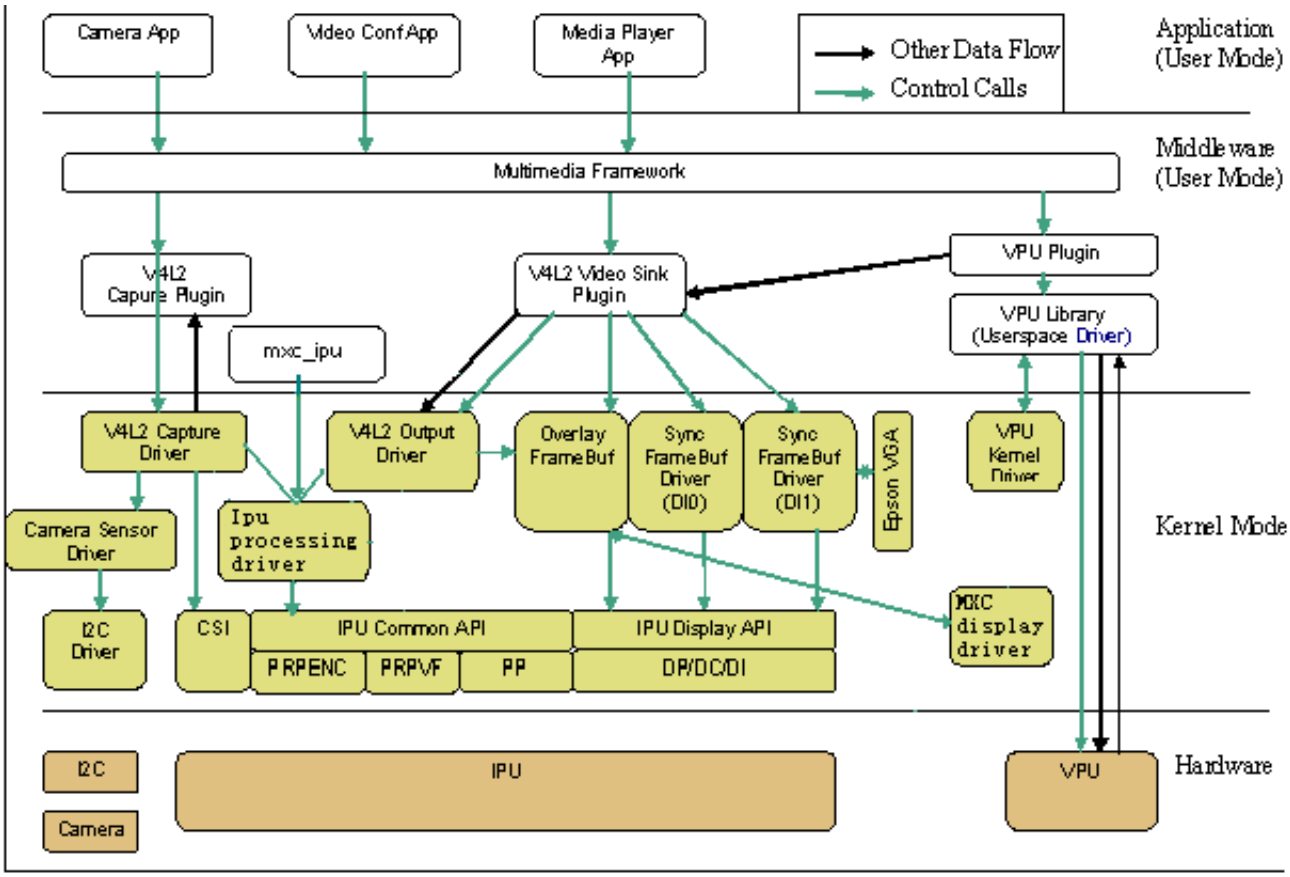


Figure 6-2. Graphics/Video Drivers Software Interaction for IPUv3

The IPU drivers are sub-divided as follows:

- Device drivers-include the frame buffer driver for the synchronous frame buffer, the frame buffer driver for the displays, V4L2 capture drivers for IPU pre-processing, the V4L2 output driver for IPU post-processing, and the ipu processing driver which provide system interface to user space or V4L2 drivers. The frame buffer device drivers are available in the <Yocto_BuildDir>/linux/drivers/video/mxc directory of the Linux kernel. The V4L2 device drivers are available in the <Yocto_BuildDir>/ linux/drivers/media/video directory of the Linux kernel.
- MXC display driver is introduced as a simple framework to manage interaction between IPU and display device drivers (e.g., LCD, LVDS, HDMI, MIPI, etc.)
- Low-level library routines-interface to the IPU hardware registers. They take input from the high-level device drivers and communicate with the IPU hardware. The low-level libraries are available in the directory of the Linux kernel.

6.2.1 IPU Frame Buffer Drivers Overview

The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer video hardware, and allows application software to access the graphics hardware through a well-defined interface, so that the software is not required to know anything about the low-level hardware registers.

The driver is enabled by selecting the frame buffer option under the graphics parameters in the kernel configuration. To supplement the frame buffer driver, the kernel builder may also include support for fonts and a startup logo. This device depends on the virtual terminal (VT) console to switch from serial to graphics mode. The device is accessed through special device nodes, located in the /dev directory, as /dev/fb*. fb0 is generally the primary frame buffer.

Other than the physical memory allocation and LCD panel configuration, the common kernel video API is utilized for setting colors, palette registration, image blitting, and memory mapping. The IPU reads the raw pixel data from the frame buffer memory and sends it to the panel for display.

6.2.1.1 IPU Frame Buffer Hardware Operation

The frame buffer interacts with the IPU hardware driver module.

6.2.1.2 IPU Frame Buffer Software Operation

A frame buffer device is a memory device, such as /dev/mem, and it has features similar to a memory device. Users can read it, write to it, seek to some location in it, and mmap() it (the main use). The difference is that the memory that appears in the special file is not the whole memory, but the frame buffer of some video hardware.

/dev/fb* also interacts with several IOCTLs, which allows users to query and set information about the hardware. The color map is also handled through IOCTLs. For more information on what IOCTLs exist and which data structures they use, see <Yocto_BuildDir>/linux/include/linux/fb.h. The following are a few of the IOCTLs functions:

- Request general information about the hardware, such as name, organization of the screen memory (planes, packed pixels, and so on), and address and length of the screen memory.
- Request and change variable information about the hardware, such as visible and virtual geometry, depth, color map format, timing, and so on. The driver suggests

values to meet the hardware capabilities (the hardware returns EINVAL if that is not possible) if this information is changed.
- Get and set parts of the color map. Communication is 16 bits-per-pixel (values for red, green, blue, transparency) to support all existing hardware. The driver does all the calculations required to apply the options to the hardware (round to fewer bits, possibly discard transparency value).

The hardware abstraction makes the implementation of application programs easier and more portable. The only thing that must be built into the application programs is the screen organization (bitplanes or chunky pixels, and so on), because it works on the frame buffer image data directly.

The MXC frame buffer driver ( ) interacts closely with the generic Linux frame buffer driver (<Yocto_BuildDir>/linux/drivers/video/fbmem.c).

6.2.1.3 Synchronous Frame Buffer Driver

The synchronous frame buffer screen driver implements a Linux standard frame buffer driver API for synchronous LCD panels or those without memory. The synchronous frame buffer screen driver is the top level kernel video driver that interacts with kernel and user level applications. This is enabled by selecting the Synchronous Panel Frame buffer option under the graphics support device drivers in the kernel configuration. To supplement the frame buffer driver, the kernel builder may also include support for fonts and a startup logo. This depends on the VT console for switching from serial to graphics mode.

Except for physical memory allocation and LCD panel configuration, the common kernel video API is utilized for setting colors, palette registration, image blitting and memory mapping. The IPU reads the raw pixel data from the frame buffer memory and sends it to the panel for display.

The frame buffer driver supports different panels as a kernel configuration option. Support for new panels can be added by defining new values for a structure of panel settings.

The frame buffer interacts with the IPU driver using custom APIs that allow:

- Initialization of panel interface settings
- Initialization of IPU channel settings for LCD refresh
- Changing the frame buffer address for double buffering support

The following features are supported:

- Configurable screen resolution

- Configurable RGB 16, 24 or 32 bits per pixel frame buffer
- Configurable panel interface signal timings and polarities
- Palette/color conversion management
- Power management
- LCD power off/on
- Enable/disable PRE/PRG features

User applications utilize the generic video API (the standard Linux frame buffer driver API) to perform functions with the frame buffer. These include the following:

- Obtaining screen information, such as the resolution or scan length
- Allocating user space memory using mmap for performing direct blitting operations

A second frame buffer driver supports a second video/graphics plane.

6.2.2 IPU Backlight Driver

The IPU backlight driver implements IPU PWM backlight control for panels. It exports a sys control file under /sys/class/backlight/pwm-backlight.0/brightness to user space. The default backlight intensity value is 128.

6.2.3 IPU Device Driver

IPU (processing) device driver provide image processing features: resizing/rotation/CSC/ combination/deinterlacing based on IC/IRT modules in IPUv3.

The IPU device driver is task based, user just need prepare task setting, queue task, then block wait task finish. The driver now support blocking method only, non-block method will be added in the future. The task structures are like below:

```
struct ipu_task {
        struct ipu_input input;
        struct ipu_output output;

        bool overlay_en;
        struct ipu_overlay overlay;

#define IPU_TASK_PRIORITY_NORMAL 0
#define IPU_TASK_PRIORITY_HIGH  1
        u8      priority;

#define IPU_TASK_ID_ANY 0
#define IPU_TASK_ID_VF  1
#define IPU_TASK_ID_PP  2
#define IPU_TASK_ID_MAX 3
        u8      task_id;

        int     timeout;
};
```

```
struct ipu_input {
        u32 width;
        u32 height;
        u32 format;
        struct ipu_crop crop;
        dma_addr_t paddr;

        struct ipu_deinterlace deinterlace;
        dma_addr_t paddr_n; /*valid when deinterlace enable*/
};

struct ipu_overlay {
        u32 width;
        u32 height;
        u32 format;
        struct ipu_crop crop;
        struct ipu_alpha alpha;
        struct ipu_colorkey colorkey;
        dma_addr_t
paddr;

};




struct ipu_output
{


        u32 width;
        u32 height;
        u32 format;
        u8 rotate;
        struct ipu_crop crop;
        dma_addr_t paddr;
};
```

To prepare task, user just needs to fill task.input, task.overlay(if need combine) and task.output parameters, then queue task either by:

```
 int ipu_queue_task(struct ipu_task *task);
```

if from kernel level (V4L2 driver for example), or by IPU_QUEUE_TASK ioctl under /dev/mxc_ipu if from application level.

6.3 Source Code Structure

Table 6-1 lists the source files associated with the IPU, Sensor, V4L2, and Panel drivers. These files are available in the following directories:

```
Yocto_BuildDir/linux/drivers/mxc/ipu3
Yocto_BuildDir/linux/drivers/video/mxc
Yocto_BuildDir/linux/drivers/media/platform/mxc
Yocto_BuildDir/linux/drivers/video/backlight
```

Table 6-1. IPU Driver Files

File	Description
ipu_common.c	IPU common library functions
ipu_ic.c	IPU IC base driver
ipu_device.c	IPU driver device interface and fops functions
ipu_capture.c	IPU CSI capture base driver
ipu_disp.c	IPU display functions
ipu_calc_stripes_sizes.c	Multi-stripes method functions for ipu_device.c
pre.c	Prefetch/Resolve the engine driver
prg.c	Prefetch/Resolve the Gasket driver
mxc_ipuv3_fb.c	Driver for synchronous frame buffer
mxc_lcdif.c	Display Driver for CLAA-WVGA and SEIKO-WVGA LCD support
mxc_hdmi.c	Display Driver for HDMI interface
ldb.c	Driver for synchronous frame buffer for on chip LVDS
mxc_dispdrv.c	Display Driver framework for synchronous frame buffer
mxc_edid.c	Driver for EDID
vdoa.c	VDOA post-processing driver, used by ipu_device.c

Table 6-2 lists the global header files associated with the IPU and Panel drivers. These files are available in the following directories:

```
Yocto_BuildDir/linux/drivers/mxc/ipu3/
Yocto_BuildDir/linux/include/linux/
Yocto_BuildDir/linux/drivers/media/platform/mxc/
```

Table 6-2. IPU Global Header Files

File	Description
ipu_param_mem.h	Helper functions for IPU parameter memory access
ipu_prv.h	Header file for Pre-processing drivers
ipu_regs.h	IPU register definitions
pre-regs.h	Prefetch/Resolve Engine register definitions
prg-regs.h	Prefetch/Resolve Gasket register definitions
vdoa.h	Header file for VDOA drivers
mxc_dispdrv.h	Header file for display driver
mxcfb.h	Header file for the synchronous framebuffer driver
ipu.h	Header file for IPU basic driver

6.3.1 Menu Configuration Options

The following Linux kernel configuration options are provided for the IPU module.

To get to these options use the command `bitbake linux-imx -c menuconfig` in the Yocto build directory. On the screen displayed, select Configure the kernel and exit. When the next screen appears select the options to configure.

- CONFIG_MXC_IPU_V3 - Includes support for the Image Processing Unit. In menuconfig, this option is available under:

  Device Drivers > MXC support drivers > Image Processing Unit Driver

  By default, this option is Y for all architectures.

  If ARCH_MXC is true, CONFIG_MXC_IPU_V3 will be set.

- CONFIG_MXC_IPU_V3_PRG - This enables support for the IPUv3 prefetch gasket engine to support double buffer handshake control bewteen IPUv3 and prefetch engine (PRE), snoop the AXI interface for display refresh requests to memory, and modify the request address to fetch the double buffered row of blocks in OCRAM.

  Device Drivers > MXC support drivers > i.MX IPUv3 prefetch gasket engine

  This option depends on CONFIG_MXC_IPU_V3 and CONFIG_MXC_IPU_V3_PRE.

- CONFIG_MXC_IPU_V3_PRE - This enables support for the IPUv3 prefetch engine to improve the system memory performance. The engine has the capability to resolve framebuffers in tile pixel format to linear.

  Device Drivers > MXC support drivers > i.MX IPUv3 prefetch engine

  This option depends on CONFIG_MXC_IPU_V3. Enabling this option selects CONFIG_MXC_IPU_V3_PRG.

- CONFIG_MXC_CAMERA_OV5640_MIPI - Option for both the OV 5640 mipi sensor driver and the use case driver. This option is dependent on the VIDEO_MXC_CAPTURE option. In menuconfig, this option is available under:

  Device Drivers > Multimedia support > V4L platform devices > MXC Video For Linux Video Capture > MXC Camera/V4L2 PRP Features support > OmniVision 5640 Camera support using mipi

- CONFIG_MXC_CAMERA_OV5640 - Option for both the OV5640 sensor driver and the use case driver. This option is dependent on the VIDEO_MXC_CAPTURE option. In menuconfig, this option is available under:

  Device Drivers > Multimedia platform > V4L platform devices > MXC Video For Linux Video Capture > MXC Camera/V4L2 PRP Features support > OmniVision ov5640 camera support

Only one sensor should be installed at a time.

- CONFIG_MXC_IPU_PRP_VF_SDC - Option for the IPU (here the > symbols illustrates data flow direction between HW blocks):

  CSI > IC > MEM MEM > IC (PRP VF) > MEM

  Use case driver for dumb sensor or

  CSI > IC(PRP VF) > MEM

  for smart sensors. In menuconfig, this option is available under:

  Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > Pre-Processor VF SDC library

  By default, this option is M for all.

- CONFIG_MXC_IPU_PRP_ENC - Option for the IPU:

  Use case driver for dumb sensors

  CSI > IC > MEM MEM > IC (PRP ENC) > MEM

  or for smart sensors

  CSI > IC(PRP ENC) > MEM.

  In menuconfig, this option is available under:

  Device Drivers > Multimedia Devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > Pre-processor Encoder library

  By default, this option is set to M for all.

- CONFIG_VIDEO_MXC_CAMERA - This is configuration option for V4L2 capture Driver. This option is dependent on the following expression:

  VIDEO_DEV && MXC_IPU && MXC_IPU_PRP_VF_SDC && MXC_IPU_PRP_ENC

  In menuconfig, this option is available under:

  Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera

  By default, this option is M for all.

- CONFIG_VIDEO_MXC_OUTPUT - This is configuration option for V4L2 output Driver. This option is dependent on VIDEO_DEV && MXC_IPU option. In menuconfig, this option is available under:

  Device Drivers > Multimedia devices > Video capture adapters > MXC Video for Linux Video Output

  By default, this option is Y for all.

- CONFIG_FB - This is the configuration option to include frame buffer support in the Linux kernel. In menuconfig, this option is available under:

  Device Drivers > Graphics support > Support for frame buffer devices

  By default, this option is Y for all architectures.

- CONFIG_FB_MXC - This is the configuration option for the MXC Frame buffer driver. This option is dependent on the CONFIG_FB option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support

  By default, this option is Y for all architectures.

- CONFIG_FB_MXC_SYNC_PANEL - This is the configuration option that chooses the synchronous panel framebuffer. This option is dependent on the CONFIG_FB_MXC option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer

  By default this option is Y for all architectures.

- CONFIG_FB_MXC_LDB - This configuration option selects the LVDS module on i.MX 6 chip. This option is dependent on CONFIG_FB_MXC_SYNC_PANEL and CONFIG_MXC_IPUV3 || FB_MXS options. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > MXC LDB

- CONFIG_FB_MXC_SII9022 - This configuration option selects the SII9022 HDMI chip. This option is dependent on CONFIG_FB_MXC_SYNC_PANEL option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > Si Image SII9022 DVI/HDMI Interface Chip

6.4 Unit Test

NOTE

In order to execute the tests properly, make sure you have the util-linux package selected and load the following modules:

```
insmod ipu_prp_enc.ko
insmod ipu_bg_overlay_sdc.ko
insmod ipu_fg_overlay_sdc.ko
insmod ipu_csi_enc.ko
insmod ov5640_camera.ko
insmod mxc_v4l2_capture.ko
```

6.4.1 Framebuffer Tests

There is a test application named mxc_fb_test.c under the <Yocto_BuildDir>/imx-test-"version"/test/mxc_fb_test directory.

Execute the fb test as follows:

./mxc_fb_test.out

The result should be Exiting PASS. The test includes fb0(background) and fb1(foreground) devices open, framebuffer parameters configure, global alpha blending, fb pan display test and gamma test.

Redirect an image directly to the framebuffer device as follows:

cat image.bin > /dev/fb0

6.4.2 Video4Linux API test

There are test applications named mxc_v4l2_test.c and mxc_v4l2_output.c under the <Yocto_BuildDir>/imx-test-"version"/test/mxc_v4l2_test directory.

Before running the v4l2 capture test application, you should be able see that the /dev/v4l/video0 has been created.

Test ID: FSL-UT-V4L2-capture-0010

```
        # mxc_v4l2_capture.out -iw 640 -ih 480 -m 0 -r 0 -c 50 -fr 30 test.yuv

        Capture the camera and store the 50 frames of YUV420 (VGA size)to a file called
test.yuv and set the frame rate to 30 fps. Look at mxc_v4l2_capture.out -help to see
usage.
```

Test ID: FSL-UT-V4L2-overlay-sdc-0010

```
        # mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 160 -oh 160 -ot 20 -ol 20 -r
0 -t 50 -d 0 -fg -fr 30

        Direct preview the camera to SDC foreground, and set frame rate to 30 fps, window
of
interest is 640 X 480 with starting offset(0,0), the preview size is 160 X 160 with
starting offset (20,20). mxc_v4l2_overlay.out -help to see the usage.
```

Test ID: FSL-UT-V4L2-overlay-sdc-0020

```
        # mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 160 -oh 160 -ot 20 -ol 20 -r
4 -t 50 -d 0 -fr 30

        Direct preview(90 degree rotation) the camera to SDC background, and set frame rate
to 30 fps.
```

Test ID: FSL-UT-V4L2-overlay-adc-0010

```
        # mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 120 -oh 120 -ot 40 -ol 40 -r
0 -t 50 -d 1 -fg -fr 30

        Direct preview the camera to foreground, and set frame rate to 30 fps.
```

Test ID: FSL-UT-V4L2-overlay-adc-0020

```
        # mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 120 -oh 120 -ot 40 -ol 40 -r
4 -t 50 -d 1 -fg -fr 30

        Direct preview(90 degree rotation) the camera to foreground, and set frame rate to
30
fps.
```

Test ID: FSL-UT-V4L2-output-0010

```
        # mxc_v4l2_output.out -iw 640 -ih 480 -ow 1024 -oh 768 -r 0 -fr 60 test.yuv

        Read the YUV420 stream file on test.yuv created by the mxc_v4l2_capture test as run
in test FSL-UT-V4L2-capture-0010. Apply color space conversion and resize, then
display on the framebuffer.
```

NOTE

The PRP channels require the stride line to be a multiple of 8, for example with no rotation, the width needs to be 8 bit aligned; and with 90 degree rotation, the height needs to be 8

bit aligned. Downsizing cannot exceed 8:1. For example, for a VGA sensor, the smallest downsize is 80 X 60.

6.4.3 IPU Device Unit test

There is a test application named mxc_ipudev_test.c under the <Yocto_BuildDir>/imx-test-"version"/test/mxc_ipudev_test directory.

Before running the IPU device test application, you should be able see that the /dev/mxc_ipu has been created.

Run test like:

```
./mxc_ipudev_test.out -C config_file raw_data_file

./mxc_ipudev_test.out -command_line_options raw_data_file
```

See <Yocto_BuildDir>/imx-test-"version"/test/ipudev_config_file for configure file instruction.

Below is a simple test source code of IPU device overlay which use alpha(global/local) blending to combine two layers:

NOTE: the overlay width and height must be same as output's. For example, the input is 240x320, output is 1024x768 which using rotation 90 degree, the overlay must be same as output, said, 1024x768.

```
static unsigned int fmt_to_bpp(unsigned int pixelformat)
{
        unsigned int bpp;

        switch (pixelformat) {
                case IPU_PIX_FMT_RGB565:
                /*interleaved 422*/
                case IPU_PIX_FMT_YUYV:
                case IPU_PIX_FMT_UYVY:
                /*non-interleaved 422*/
                case IPU_PIX_FMT_YUV422P:
                case IPU_PIX_FMT_YVU422P:
                        bpp = 16;
                        break;
                case IPU_PIX_FMT_BGR24:
                case IPU_PIX_FMT_RGB24:
                case IPU_PIX_FMT_YUV444:
                        bpp = 24;
                        break;
                case IPU_PIX_FMT_BGR32:
                case IPU_PIX_FMT_BGRA32:
                case IPU_PIX_FMT_RGB32:
                case IPU_PIX_FMT_RGBA32:
                case IPU_PIX_FMT_ABGR32:
                        bpp = 32;
                        break;
                /*non-interleaved 420*/
                case IPU_PIX_FMT_YUV420P:
                case IPU_PIX_FMT_YVU420P:
```

```
                case IPU_PIX_FMT_YUV420P2:
                case IPU_PIX_FMT_NV12:
                        bpp = 12;
                        break;
                default:
                        bpp = 8;
                        break;
        }
        return bpp;

}

static void dump_ipu_task(struct ipu_task *t)
{
        printf("====== ipu task ======\n");
        printf("input:\n");
        printf("\twidth: %d\n", t->input.width);
        printf("\theight: %d\n", t->input.height);
        printf("\tcrop.w = %d\n", t->input.crop.w);
        printf("\tcrop.h = %d\n", t->input.crop.h);
        printf("\tcrop.pos.x = %d\n", t->input.crop.pos.x);
        printf("\tcrop.pos.y = %d\n", t->input.crop.pos.y);
        printf("output:\n");
        printf("\twidth: %d\n", t->output.width);
        printf("\theight: %d\n", t->output.height);
        printf("\tcrop.w = %d\n", t->output.crop.w);
        printf("\tcrop.h = %d\n", t->output.crop.h);
        printf("\tcrop.pos.x = %d\n", t->output.crop.pos.x);
        printf("\tcrop.pos.y = %d\n", t->output.crop.pos.y);

       if (t->overlay_en) {
                printf("overlay:\n");
                printf("\twidth: %d\n", t->overlay.width);
                printf("\theight: %d\n", t->overlay.height);
                printf("\tcrop.w = %d\n", t->overlay.crop.w);
                printf("\tcrop.h = %d\n", t->overlay.crop.h);
                printf("\tcrop.pos.x = %d\n", t->overlay.crop.pos.x);
                printf("\tcrop.pos.y = %d\n", t->overlay.crop.pos.y);
        }

}

int main(int argc, char *argv[])
{
        int fd, fd_fb, isize, ovsize, alpsize, cnt = 50;
        int blank, ret;
        FILE * file_in = NULL;
        struct ipu_task task;
        struct fb_var_screeninfo fb_var;
        struct fb_fix_screeninfo fb_fix;
        void *inbuf, *ovbuf, *alpbuf, *vdibuf;

        fd = open("/dev/mxc_ipu", O_RDWR, 0);
        fd_fb = open("/dev/fb1", O_RDWR, 0);
        file_in = fopen(argv[argc-1], "rb");

        memset(&task, 0, sizeof(task));

        /* input setting */
        task.input.width = 320;
        task.input.height = 240;
        task.input.crop.pos.x = 0;
        task.input.crop.pos.y = 0;
        task.input.crop.w = 0;
        task.input.crop.h = 0;
        task.input.format = IPU_PIX_FMT_YUV420P;

        isize = task.input.paddr =
                task.input.width * task.input.height
                * fmt_to_bpp(task.input.format)/8;
```

```
        ioctl(fd, IPU_ALLOC, &task.input.paddr);
        inbuf = mmap(0, isize, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, task.input.paddr);

       /*overlay setting */
        task.overlay_en = 1;
        task.overlay.width = 1024;
        task.overlay.height = 768;
        task.overlay.crop.pos.x = 0;
        task.overlay.crop.pos.y = 0;
        task.overlay.crop.w = 0;
        task.overlay.crop.h = 0;
        task.overlay.format = IPU_PIX_FMT_RGB24;
#ifdef GLOBAL_ALP
        task.overlay.alpha.mode = IPU_ALPHA_MODE_GLOBAL;
        task.overlay.alpha.gvalue = 255;
        task.overlay.colorkey.enable = 1;
        task.overlay.colorkey.value = 0x555555;
#else
        task.overlay.alpha.mode = IPU_ALPHA_MODE_LOCAL;
        alpsize = task.overlay.alpha.loc_alp_paddr =
                task.overlay.width * task.overlay.height;
        ioctl(fd, IPU_ALLOC, &task.overlay.alpha.loc_alp_paddr);
        alpbuf = mmap(0, alpsize, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, task.overlay.alpha.loc_alp_paddr);
        memset(alpbuf, 0x00, alpsize/4);
        memset(alpbuf+alpsize/4, 0x55, alpsize/4);
        memset(alpbuf+alpsize/2, 0x80, alpsize/4);
        memset(alpbuf+alpsize*3/4, 0xff, alpsize/4);
#endif

        ovsize = task.overlay.paddr =
                task.overlay.width * task.overlay.height
                * fmt_to_bpp(task.overlay.format)/8;
        ioctl(fd, IPU_ALLOC, &task.overlay.paddr);
        ovbuf = mmap(0, ovsize, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, task.overlay.paddr);
#ifdef GLOBAL_ALP
        memset(ovbuf, 0x55, ovsize/4);
        memset(ovbuf+ovsize/4, 0xff, ovsize/4);
        memset(ovbuf+ovsize/2, 0x55, ovsize/4);
        memset(ovbuf+ovsize*3/4, 0x00, ovsize/4);
#else
        memset(ovbuf, 0x55, ovsize);
#endif
#endif

        /* output setting*/
        task.output.width = 1024;
        task.output.height = 768;
        task.output.crop.pos.x = 0;
        task.output.crop.pos.y = 0;
        task.output.crop.w = 0;
        task.output.crop.h = 0;
        task.output.format = IPU_PIX_FMT_RGB565;
        task.output.rotate = IPU_ROTATE_NONE;

        ioctl(fd_fb, FBIOGET_VSCREENINFO, &fb_var);
        fb_var.xres = task.output.width;
        fb_var.xres_virtual = fb_var.xres;
        fb_var.yres = task.output.height;
        fb_var.yres_virtual = fb_var.yres * 3;
        fb_var.activate |= FB_ACTIVATE_FORCE;
        fb_var.nonstd = task.output.format;
        fb_var.bits_per_pixel = fmt_to_bpp(task.output.format);
        ioctl(fd_fb, FBIOPUT_VSCREENINFO, &fb_var);
        ioctl(fd_fb, FBIOGET_VSCREENINFO, &fb_var);
        ioctl(fd_fb, FBIOGET_FSCREENINFO, &fb_fix);
        task.output.paddr = fb_fix.smem_start;
        blank = FB_BLANK_UNBLANK;
```

```
        ioctl(fd_fb, FBIOBLANK, blank);

        task.priority = IPU_TASK_PRIORITY_NORMAL;
        task.task_id = IPU_TASK_ID_ANY;
        task.timeout = 1000;

again:
        ret = ioctl(fd, IPU_CHECK_TASK, &task);

        if (ret != IPU_CHECK_OK)  {
                if (ret > IPU_CHECK_ERR_MIN) {
                        if (ret == IPU_CHECK_ERR_SPLIT_INPUTW_OVER)  {
                                task.input.crop.w -= 8;
                                goto again;
                        }
                        if (ret == IPU_CHECK_ERR_SPLIT_INPUTH_OVER) {
                                task.input.crop.h -= 8;
                                goto again;
                        }
                        if (ret == IPU_CHECK_ERR_SPLIT_OUTPUTW_OVER) {
                                task.output.crop.w -= 8;
                                goto again;
                        }
                        if (ret == IPU_CHECK_ERR_SPLIT_OUTPUTH_OVER) {
                                task.output.crop.h -= 8;
                                goto again;
                        }
                        ret = -1;
                        return ret;
                }
        }


        dump_ipu_task(&task);

        while (--cnt > 0) {
                fread(inbuf, 1, isize, file_in);
                ioctl(fd, IPU_QUEUE_TASK, &task);
        }

        munmap(ovbuf, ovsize);

        ioctl(fd, IPU_FREE, task.input.paddr);
        ioctl(fd, IPU_FREE, task.overlay.paddr);

        close(fd);
        close(fd_fb);
        fclose(file_in);
}
```

Chapter 7
MIPI DSI Driver

7.1 Introduction

The MIPI DSI driver for Linux OS is based on the IPU framebuffer driver.

This driver has two parts:

- MIPI DSI IP driver-low level interface used to communicate with MIPI device controller on the display panel
- MIPI DSI display panel driver provides an interface to configure the display panel through MIPI DSI

7.1.1 MIPI DSI IP Driver Overview

The MIPI DSI IP driver is registered through IPU framebuffer driver interface and it is not exposed to the user space.

The driver enables the platform-related regulators and clocks. It requests OS related system resources and registers framebuffer event notifier for blank/unblank operation. Next, the driver initializes MIPI D-PHY and configures the MIPI DSI IP according to the MIPI DSI display panel. MIPI DSI driver supports the following features:

- Compatibility with MIPI Alliance Specification for DSI, Version1.01.00
- Compatibility with MIPI Alliance Specification for D-PHY, Version 1.00.00
- Supports up to 2 D-PHY data lanes
- Bidirectional Communication and Escape Mode Support through Data Lane 0
- Programmable display resolutions, from 160x120(QQVGA) to 1024x768(XVGA)
- Video Mode Pixel Formats, 16bpp(565RGB),18bpp(666RGB)packed, 18bpp(666RGB)loosely, 24bpp(888RGB).
- Supports the transmission of all generic commands
- Supports ECC and checksum capabilities

- End-of-Transmission Packet(EoTp) support
- Supports ultra low power mode

7.1.2 MIPI DSI Display Panel Driver Overview

The MIPI DSI display panel driver implements MIPI DSI display panel related configuration.

It uses the APIs provided by the MIPI DSI IP driver to read/write the display module registers. Usually, there is a MIPI DSI slave controller integrated on the display panel. After power on reset, the MIPI DSI display panel needs to be configured through standard MIPI DCS command or MIPI DSI Generic command according to the manufacturer's specification.

7.1.3 Hardware Operation

The MIPI DSI module provides a high-speed serial interface between a host processor and a display module.

It has higher performance, lower power, less EMI and fewer pins compared with legacy parallel bus. It is designed to be compatible with the standard MIPI DSI protocol. MIPI DSI is built on exisiting MIPI DPI-2, MIPI DBI-2 and MIPI DCS standards. It sends pixels or commands to the peripheral and reads back status or pixel information from the peripheral. MIPI DSI serializes all pixels data, commands and events, and contains two basic modes: command mode and video mode. It uses command mode to read/write register and memory to the display controller while reading display module status information. On the other hand, it uses video mode to transmit a real-time pixel streams from host to peripheral in high-speed mode. It also generates an interrupt when error occurs.

7.2 Software Operation

The MIPI DSI driver for Linux OS has two parts: MIPI DSI IP driver and MIPI DSI display panel driver.

7.2.1 MIPI DSI IP Driver Software Operation

The MIPI DSI IP driver has a private structure called mipi_dsi_info. The IPU instance to which the MIPI DSI IP is attached is described in field int ipu_id while the DI instance inside IPU is described in the field int disp_id.

During startup, the MIPI DSI IP driver is registered with the IPU framebuffer driver through the field struct mxc_dispdrv_entry when the driver is loaded. It also registers a framebuffer event notifier with framebuffer core to perform the display panel blank/unblank operation. The field struct fb_videomode *mode and struct mipi_lcd_config *lcd_config are received from the display panel callback. The MIPI DSI IP needs this infomation to configure the MIPI DSI hardware registers.

After initializing the MIPI DSI IP controller and the display module, the MIPI DSI IP gets the pixel streams from IPU through DPI-2 interface and serializes pixel data and video event through high-speed data links for display. When there is an framebuffer blank/unblank event, the registered notifier will be called to enter/leave low power mode.

The MIPI DSI IP driver provides 3 APIs for MIPI DSI display panel driver to configure display module.

7.2.2 MIPI DSI Display Panel Driver Software Operation

The MIPI DSI Display Panel driver enables a particular display panel through MIPI DSI interface. The driver should provide struct fb_videomode configuration and struct mipi_lcd_config data: some MIPI DSI parameters for the display panel such as maximum D-PHY clock, numbers of data lanes and DPI-2 pixel format. Finally, the display driver needs to setup display panel initialize routine by calling the APIs provided by MIPI DSI IP drivers.

7.3 Driver Features

The MIPI DSI driver supports the following features:

- MIPI DSI communication protocol
- MIPI DSI command mode and video mode
- MIPI DCS command operation

NOTE

The MIPI DSI driver does not support the DBI-2 mode, since the DBI-2 and DPI-2 cannot be enabled at the same time on this controller.

7.3.1 Source Code Structure

Table below shows the MIPI DSI driver source files available in the directory:

<Yocto_BuildDir>/linux/drivers/video/mxc.

Table 7-1. MIPI DSI Driver Files

File	Description
mipi_dsi.c	MIPI DSI IP driver source file
mipi_dsi.h	MIPI DSI IP driver header file
mxcfb_hx8369_wvga.c	MIPI DSI Display Panel driver source file

7.3.2 Menu Configuration Options

The following Linux kernel configuration option is provided for this module. To get to this option, use the bitbake linux-imx -c menuconfigcommand. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > MXC MIPI_DSI

7.3.3 Programming Interface

The MIPI DSI Display Panel driver can use the API interface to read and write the registers of the display panel device connected to MIPI DSI link.

For more information, see <Yocto_BuildDir>/linux/driver/video/mxc/mipi_dsi.h.

Chapter 8
LVDS Display Bridge(LDB) Driver

8.1 Introduction

This section describes the LVDS Display Bridge(LDB) driver which controls LDB module to connect with external display devices with LVDS interface.

8.1.1 Hardware Operation

The purpose of the LDB is to support flow of synchronous RGB data from IPU or LCDIF to external display devices through LVDS interface.

This support covers all aspects of these activities:

1. Connectivity to relevant devices - Displays with LVDS receivers.
2. Arranging data as required by the external display receiver and by LVDS display standards.
3. Synchronization and control capabilities.

For detailed information about LDB, see the LDB chapter of the following documents:

- *i.MX 6Dual/6Quad Applications Processor Reference Manual* (IMX6DQRM)
- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)
- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)
- *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM)

8.1.2 Software Operation

LDB driver is functional if the driver is built-in.

When LDB device is probed properly, the driver will configure LDB reference resistor mode and LDB regulator by using platform data information. LDB driver probe function will also try to match video modes for external display devices to LVDS interface. The display signal polarities control bits of LDB are set according to the matched video modes. LVDS channel mapping mode and bit mapping mode of LDB are set according to the LDB device tree node set by the user. LDB is fully enabled in probe function if the driver identifies a display device with LVDS interface as the primary display device.

The steps the driver takes to enable a LVDS channel are:

1. Set ldb_di_clk's parent clk and the parent clk's rate.
2. Set ldb_di_clk's rate.
3. Enable both ldb_di_clk and its parent clk.
4. Set the LDB in a proper mode including display signals' polarities, LVDS channel mapping mode, bit mapping mode, and reference resistor mode.
5. Enable related LVDS channels.

See `<Yocto_BuildDir>/linux/drivers/video/mxc/ldb.c` for more information.

8.1.3 Source Code Structure

The source code is available in the following location:

```
<Yocto_BuildDir>/linux/drivers/video/mxc/ldb.c
```

8.1.4 Menu Configuration Options

The following Linux kernel configuration options are provided for this module.

To get to these options, use the `bitbake linux-imx -c menuconfig` command. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options as build-in status to enable this module:

```
Device Drivers -> Graphics support -> MXC Framebufer support ->
Synchronous Panel Framebuffer -> MXC LDB
```

Chapter 9
Video for Linux Two (V4L2) Driver

9.1 Introduction

The Video for Linux Two (V4L2) drivers are plug-ins to the V4L2 framework that enable support for camera and preprocessing functions, as well as video and post-processing functions.

The V4L2 camera driver implements support for all camera related functions. The V4l2 capture device takes incoming video images, either from a camera or a stream, and manipulates them. The output device takes video and manipulates it, then sends it to a display or similar device. The V4L2 Linux standard API specification is available at v4l2spec.bytesex.org/spec

The features supported by the V4L2 driver are as follows:

- Direct preview and output to SDC foreground overlay plane (with synchronized to LCD refresh)
- Direct preview to graphics frame buffer (without synchronized to LCD refresh)
- Color keying or alpha blending of frame buffer and overlay planes
- Streaming (queued) capture from IPU encoding channel
- Direct (raw Bayer) still capture (sensor dependent)
- Programmable pixel format, size, frame rate for preview and capture
- Programmable rotation and flipping using custom API
- RGB 16-bit, 24-bit, and 32-bit preview formats
- Raw Bayer (still only, sensor dependent), RGB 16, 24, and 32-bit, YUV 4:2:0 and 4:2:2 planar, YUV 4:2:2 interleaved, and JPEG formats for capture
- Control of sensor properties including exposure, white-balance, brightness, contrast, and so on
- Plug-in of different sensor drivers
- Link post-processing resize and CSC, rotation, and display IPU channels
- Streaming (queued) input buffer
- Double buffering of overlay and intermediate (rotation) buffers

- Configurable 3+ buffering of input buffers
- Programmable input and output pixel format and size
- Programmable scaling and frame rate
- RGB 16, 24, and 32-bit, YUV 4:2:0 and 4:2:2 planar, and YUV 4:2:2 interleaved input formats
- TV output

The driver implements the standard V4L2 API for capture, output, and overlay devices. The command modprobe mxc_v4l2_capture must be run before using these functions.

9.2 V4L2 Capture Device

The V4L2 capture device includes two interfaces:

- Capture interface-uses IPU pre-processing ENC channels to record the YCrCb video stream
- Overlay interface-uses the IPU device driver to display the preview video to the SDC foreground and background panel.

V4L2 capture support can be selected during kernel configuration. The driver includes two layers. The top layer is the common Video for Linux driver, which contains chain buffer management, stream API and other ioctl interfaces. The files for this device are located in <Yocto_BuildDir>/linux/drivers/media/video/mxc/capture/.

The V4L2 capture device driver is in the mxc_v4l2_capture.c file. The low level overlay driver is in the ipu_fg_overlay_sdc.c, ipu_bg_overlay_sdc.c

This code (ipu_prp_enc.c) interfaces with the IPU ENC hardware, and ipu_still.c interfaces with the IPU CSI hardware. Sensor frame rate control is handled by VIDIOC_S_PARM ioctl. Before the frame rate is set, the sensor turns on the AE and AWB turn on. The frame rate may change depending on light sensor samples.

Drivers for specific cameras can be found in <Yocto_BuildDir>/linux/drivers/media/video/mxc/capture/

9.2.1 V4L2 Capture IOCTLs

Currently, the memory map stream API is supported. Supported V4L2 IOCTLs include the following:

- VIDIOC_QUERYCAP
- VIDIOC_G_FMT

- VIDIOC_S_FMT
- VIDIOC_REQBUFS
- VIDIOC_QUERYBUF
- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_OVERLAY
- VIDIOC_G_FBUF
- VIDIOC_S_FBUF
- VIDIOC_G_CTRL
- VIDIOC_S_CTRL
- VIDIOC_CROPCAP
- VIDIOC_G_CROP
- VIDIOC_S_CROP
- VIDIOC_S_PARM
- VIDIOC_G_PARM
- VIDIOC_ENUMSTD
- VIDIOC_G_STD
- VIDIOC_S_STD
- VIDIOC_ENUMOUTPUT
- VIDIOC_G_OUTPUT
- VIDIOC_S_OUTPUT

V4L2 control code has been extended to provide support for rotation. The ID is V4L2_CID_PRIVATE_BASE. Supported values include:

- 0-Normal operation
- 1-Vertical flip
- 2-Horizontal flip
- 3-180° rotation
- 4-90° rotation clockwise
- 5-90° rotation clockwise and vertical flip
- 6-90° rotation clockwise and horizontal flip
- 7-90° rotation counter-clockwise

Figure below shows a block diagram of V4L2 Capture API interaction.

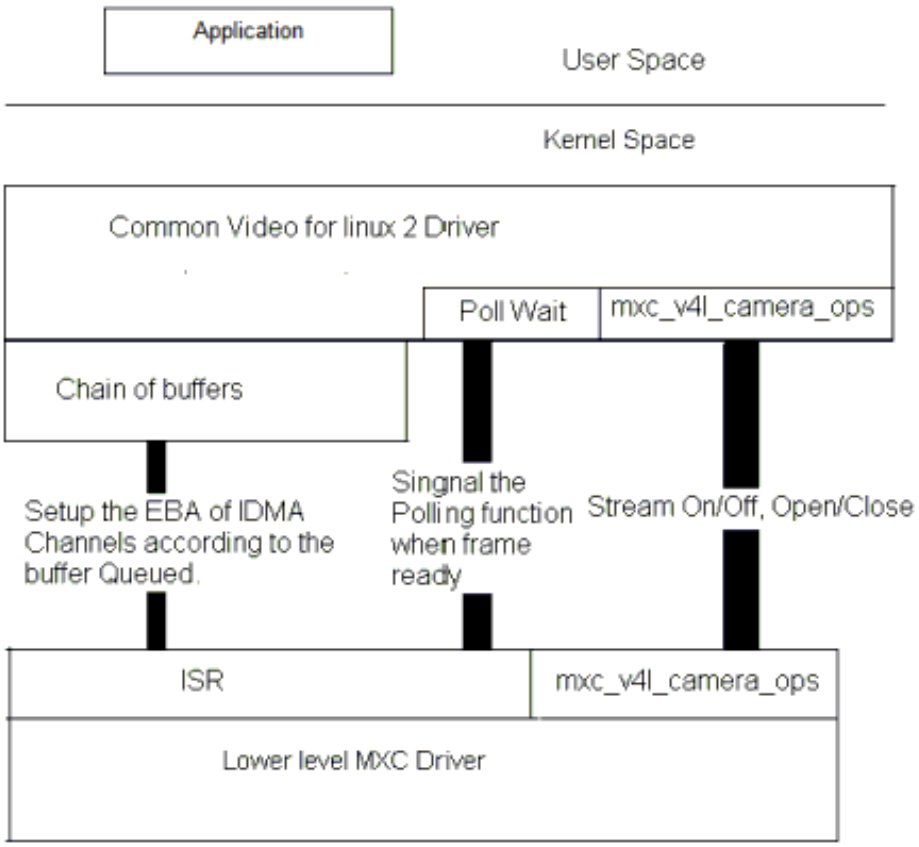


Figure 9-1. Video4Linux2 Capture API Interaction

9.2.2 Use of the V4L2 Capture APIs

This section describes a sample V4L2 capture process. The application completes the following steps:

1. Sets the capture pixel format and size by IOCTL VIDIOC_S_FMT.
2. Sets the control information by IOCTL VIDIOC_S_CTRL for rotation usage.
3. Requests a buffer using IOCTL VIDIOC_REQBUFS. The common V4L2 driver creates a chain of buffers (currently the maximum number of frames is 3).
4. Memory maps the buffer to its user space.
5. Queues buffers using the IOCTL command VIDIOC_QBUF.
6. Starts the stream using the IOCTL VIDIOC_STREAMON. This IOCTL enables the IPU tasks and the IDMA channels. When the processing is completed for a frame, the driver switches to the buffer that is queued for the next frame. The driver also signals the semaphore to indicate that a buffer is ready.
7. Takes the buffer from the queue using the IOCTL VIDIOC_DQBUF. This IOCTL blocks until it has been signaled by the ISR driver.

8. Stores the buffer to a YCrCb file.
9. Replaces the buffer in the queue of the V4L2 driver by executing VIDIOC_QBUF again.

For the V4L2 still image capture process, the application completes the following steps:

1. Sets the capture pixel format and size by executing the IOCTL VIDIOC_S_FMT.
2. Reads one frame still image with YUV422.

FOr the V4L2 overlay support use case, the application completes the following steps:

1. Sets the overlay window by IOCTL VIDIOC_S_FMT.
2. Turns on overlay task by IOCTL VIDIOC_OVERLAY.
3. Turns off overlay task by IOCTL VIDIOC_OVERLAY.

9.3 V4L2 Output Device

The V4L2 output driver uses the IPU post-processing functions for video output.

The driver implements the standard V4L2 API for output devices. V4L2 output device support can be selected during kernel configuration. The driver is available at <Yocto_BuildDir>/linux/drivers/media/video/mxc/output/mxc_vout.c.

9.3.1 V4L2 Output IOCTLs

Currently, the memory map stream API is supported. Supported V4L2 IOCTLs include the following:

- VIDIOC_QUERYCAP
- VIDIOC_REQBUFS
- VIDIOC_G_FMT
- VIDIOC_S_FMT
- VIDIOC_QUERYBUF
- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_G_CTRL
- VIDIOC_S_CTRL
- VIDIOC_CROPCAP

- VIDIOC_G_CROP
- VIDIOC_S_CROP
- VIDIOC_ENUM_FMT

The V4L2 control code has been extended to provide support for de-interlace motion. For this use, the ID is V4L2_CID_MXC_MOTION. Supported values include the following:

- 0-Medium motion
- 1-Low motion
- 2-High motion

9.3.2 Use of the V4L2 Output APIs

This section describes a sample V4L2 output process that uses the V4L2 output APIs. The application completes the following steps:

1. Sets the input pixel format and size using IOCTL VIDIOC_S_FMT.
2. Sets the control information using IOCTL VIDIOC_S_CTRL, for rotation, de-interlace motion(if need).
3. Sets the output information using IOCTL VIDIOC_S_CROP.
4. Requests a buffer using IOCTL VIDIOC_REQBUFS. The common V4L2 driver creates a chain of buffers (not allocated yet)
5. Memory maps the buffer to its user space.
6. Executes the IOCTL VIDIOC_QUERYBUF to query buffers.
7. Passes the data that requires post-processing to the buffer.
8. Queues the buffer using the IOCTL command VIDIOC_QBUF.
9. Executes the IOCTL VIDIOC_DQBUF to dequeue buffers.
10. Starts the stream by executing IOCTL VIDIOC_STREAMON.
11. Stop the stream by excuting IOCTL VIDIOC_STREAMOFF

9.4 Source Code Structure

Table below lists the source and header files associated with the V4L2 drivers.

These files are available in the following directory:

```
<Yocto_BuildDir>/linux/drivers/media/video/mxc
```

Table 9-1. V2L2 Driver Files

File	Description
capture/mxc_v4l2_capture.c	V4L2 capture device driver
output/mxc_vout.c	V4L2 output device driver
capture/mxc_v4l2_capture.h	Header file for V4L2 capture device driver
capture/ipu_prp_enc.c	Pre-processing encoder driver
capture/ipu_prp_vf_adc.c	Pre-processing view finder (asynchronous) driver
capture/ipu_prp_vf_sdc.c	Pre-processing view finder (synchronous foreground) driver
capture/ipu_prp_vf_sdc_bg.c	Pre-processing view finder (synchronous background) driver
capture/ipu_fg_overlay_sdc.c	synchronous forground driver
capture/ipu_bg_overlay_sdc.c	synchronous background driver
capture/ipu_still.c	Pre-processing still image capture driver

Drivers for specific cameras can be found in <Yocto_BuildDir>/linux/drivers/media/video/mxc/capture/

Drivers for specific output can be found in <Yocto_BuildDir>/linux/drivers/media/video/mxc/output/

9.4.1 Menu Configuration Options

The Linux kernel configuration options are provided in the chapter on the IPU module.

See Menu Configuration Options.

9.4.2 V4L2 Programming Interface

For more information, see the *V4L2 Specification* and the *API Documents* for the programming interface.

The API Specification is available at LINUX MEDIA INFRASTRUCTURE API.

Chapter 10
Electrophoretic Display Controller (EPDC) Frame Buffer Driver

10.1 Introduction

The Electrophoretic Display Controller (EPDC) is a direct-drive active matrix EPD controller designed to drive E Ink EPD panels supporting a wide variety of TFT backplanes. The EPDC framebuffer driver acts as a standard Linux frame buffer device while also supporting a set of custom API extensions, accessible from user space (via IOCTL) or another kernel module (via direct function call) in order to provide the user with access to EPD-specific functionality. The EPDC driver is abstracted from any specific E Ink® panel type, providing flexibility to work with a range of E Ink panel types and specifications.

The EPDC driver supports the following features:

- Support for EPDC driver as a loadable or built-in module.
- Support for RGB565 and Y8 frame buffer formats.
- Support for full and partial EPD screen updates.
- Support for up to 256 panel-specific waveform modes.
- Support for automatic optimal waveform selection for a given update.
- Support for synchronization by waiting for a specific update request to complete.
- Support for screen updates from an alternate (overlay) buffer.
- Support for automated collision handling.
- Support for 64 simultaneous update regions.
- Support for pixel inversion in a Y8 frame buffer format.
- Support for 90, 180, and 270 degree HW-accelerated frame buffer rotation.
- Support for panning (y-direction only).
- Support for automated full and partial screen updates through the Linux fb_deferred_io mechanism.
- Support for three EPDC driver display update schemes: Snapshot, Queue, and Queue and Merge.

- Support for setting the ambient temperature through either a one-time designated API call or on a per-update basis.
- Support for user control of the delay between completing all updates and powering down the EPDC.

10.2 Hardware Operation

For the detailed hardware operation of the EPDC, see the following documents:

- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)

10.3 Software Operation

The EPDC frame buffer driver is a self-contained driver module in the Linux kernel. It consists of a standard frame buffer device API coupled with a custom EPD-specific API extension, accessible through the IOCTL interface. This combined functionality provides the user with a robust and familiar display interface while offering full control over the contents and update mode of the E Ink display.

This section covers the software operation of the EPDC driver, both through the standard frame buffer device architecture, and through the custom E Ink API extensions. Additionally, panel intialization and framebuffer formats are discussed.

10.3.1 EPDC Frame Buffer Driver Overview

The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer video hardware and allows application software to access the graphics hardware through a well-defined interface, so that the software is not required to know anything about the low-level hardware registers. The EPDC driver supports this model with one key caveat: the contents of the frame buffer are not automatically updated to the E Ink display. Instead, a custom API function call is required to trigger an update to the E Ink display. The details of this process are explained in the EPDC Frame Buffer Driver Extensions.

The frame buffer driver is enabled by selecting the frame buffer option under the graphics parameters in the kernel configuration. To supplement the frame buffer driver, the kernel builder may also include support for fonts and a startup logo. The frame buffer device

depends on the virtual terminal (VT) console to switch from serial to graphics mode. The device is accessed through special device nodes, located in the /dev directory, as /dev/fb*. fb0 is generally the primary frame buffer.

A frame buffer device is a memory device, such as /dev/mem, and it has features similar to a memory device. Users can read it, write to it, seek to some location in it, and mmap() it (the main use). The difference is that the memory that appears in the special file is not the whole memory, but the frame buffer of some video hardware.

The EPDC frame buffer driver (drivers/video/mxc/mxc_epdc_fb.c) interacts closely with the generic Linux frame buffer driver (drivers/video/fbmem.c).

For additional details on the frame buffer device, see documentation in the Linux kernel found in Documentation/fb/framebuffer.txt.

10.3.2 EPDC Frame Buffer Driver Extensions

E Ink display technology, in conjunction with the EPDC, has several features that distinguish it from standard LCD-based frame buffer devices. These differences introduce the need for API extensions to the frame buffer interface. The EPDC refreshes the E Ink display asynchronously and supports partial screen updates. Therefore, the EPDC requires notification from the user when the frame buffer contents have been modified and which region needs updating. Another unique characteristic of EPDC updates to the E Ink display is the long screen update latencies (between 300-980ms), which introduces the need for a mechanism to allow the user to wait for a given screen update to complete.

The custom API extensions to the frame buffer device are accessible both from user space applications and from within kernel space. The standard device IOCTL interface provides access to the custom API for user space applications. The IOCTL extensions, along with relevant data structures and definitions, can be found in include/linux/mxcfb.h. A full description of these IOCTLs can be found in the Programming Interface section Programming Interface.

For kernel mode access to the custom API extensions, the IOCTL interface should be bypassed in favor of direct access to the underlying functions. These functions are included in include/linux/mxcfb_epdc_kernel.h, and are documented in the Programming Interface section Programming Interface.

10.3.3 EPDC Panel Configuration

The EPDC driver is designed to flexibly support E Ink panels with a variety of panel resolutions, timing parameters, and waveform modes. The EPDC driver is kept panel-agnostic through the use of an EPDC panel mode structure, imx_epdc_fb_mode, which can be found in include/linux/mxcfb_epdc.h.

```
struct imx_epdc_fb_mode {
                    struct fb_videomode *vmode;
                    int vscan_holdoff;
                    int sdoed_width;
                    int sdoed_delay;
                    int sdoez_width;
                    int sdoez_delay;
                    int gdclk_hp_offs;
                    int gdsp_offs;
                    int gdoe_offs;
                    int gdclk_offs;
                    int num_ce;
};
```

The imx_epdc_fb_mode structure consists of an fb_videomode structure and a set of EPD timing parameters. The fb_videomode structure defines the panel resolution and the basic timing parameters (pixel clock frequency, hsync and vsync margins) and the additional timing parameters in imx_epdc_fb_mode define EPD-specific timing parameters, such as the source and gate driver timings. For details on how to configure E Ink panel timing parameters, see the EPDC programming model section in the *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM).

In addition to the EPDC panel mode data, functions may be passed to the EPDC driver to define how to handle the EPDC pins when the EPDC driver is enabled or disabled. These functions should disable the EPDC pins for purposes of power savings.

10.3.3.1 Boot Command Line Parameters

Additional configuration for the EPDC driver is provided through boot command line parameters. The format of the command line option is as follows:

```
epdc video=mxcepdcfb:[panel_name],bpp=16
```

The EPDC driver parses these options and tries to match panel_name to the name of video mode specified in the imx_epdc_fb_mode panel mode structure. If no match is found, then the first panel mode provided in the platform data is used by the EPDC driver. The bpp setting from this command line sets the initial bits per pixel setting for the frame buffer. A setting of 16 selects RGB565 pixel format, while a setting of 8 selects 8-bit grayscale (Y8) format.

10.3.4 EPDC Waveform Loading

The EPDC driver requires a waveform file for proper operation. This waveform file contains the waveform information needed to generate the waveforms that drive updates to the E Ink panel. A pointer to the waveform file data is programmed into the EPDC before the first update is performed.

There are two options for selecting a waveform file:

1. Select one of the default waveform files included in this BSP and built into the kernel.
2. Use a new waveform file that is specific to the E Ink panel being used.

The waveform file is loaded by the EPDC driver using the Linux firmware APIs.

10.3.4.1 Using a Default Waveform File

The quickest and easiest way to get started using an E Ink panel and the EPDC driver is to use one of the default waveform files provided in the Linux BSP. This should enable updates to several different types of E Ink panel without a panel-specific waveform file. The drawback is that optimal quality should not be expected. Typically, using a non-panel-specific waveform file for an E Ink panel results in more ghosting artifacts and overall poorer color quality.

The following default waveform files included in the BSP reside in firmware/imx/:

- epdc_E60_V110.fw - Default waveform for the 6.0 inch V110 E Ink panel.
- epdc_E60_V220.fw - Default waveform for the 6.0 inch V220 E Ink panel (supports animation mode updates).
- epdc_E97_V110.fw - Default waveform for the 9.7 inch V110 E Ink panel.
- epdc_E060SCM.fw - Default waveform for the 6.0 inch Pearl E Ink panel (supports animation mode updates).

The EPDC driver attempts to load a waveform file with the name "imx/epdc_[panel_name].fw", where panel_name refers to the string specified in the fb_videomode `name` field. This panel_name information should be provided to the EPDC driver through the kernel command line parameters described in the preceding chapter. For example, to load the epdc_E060SCM.fw default firmware file for a Pearl panel, set the EPDC kernel command line paratmeter to the following:

```
video=mxcepdcfb:E060SCM,bpp=16
```

10.3.4.2 Using a Custom Waveform File

To ensure the optimal E Ink display quality, use a waveform file specific to E Ink panel being used. The raw waveform file type (.wbf) requires conversion to a format that can be understood and read by the EPDC. This conversion script is not included as part of the BSP. Therefore, contact Freescale to acquire this conversion script.

Once the waveform conversion script has been run on the raw waveform file, the converted waveform file should be renamed so that the EPDC driver can find it and load it. The driver is going to search for a waveform file with the name "imx/epdc_[panel_name].fw", where panel_name refers to the string specified in the fb_videomode `name` field. For example, if the panel is named "E60_ABCD", then the converted waveform file should be named epdc_E60_ABCD.fw.

The firmware script firmware.sh (lib/udev/firmware in the Linux root file system) contains the search path used to locate the firmware file. The default search path for firmware files is /lib/firmware;/usr/local/lib/firmware. A custom search path can be specified by modifying firmware.sh. Create an imx directory in one of these paths and add your new epdc_[panel_name].fw file there.

NOTE

If the EPDC driver is searching for a firmware waveform file that matches the names of one of the default waveform files (see preceding chapter), it will choose the default firmware files that are built into the BSP over any firmware file that has been added in the firmware search path. Thus, if you leave the BSP so that it builds those default firmware files into the OS image, be sure to use a panel name other than those associated with the default firmware files, since those default waveform files will be preferred and selected over a new waveform file placed in the firmware search path.

10.3.5 EPDC Panel Initialization

The framebuffer driver will not typically (see note below for exceptions) go through any hardware initialization steps when the framebuffer driver module is loaded. Instead, a subsequent user mode call must be made to request that the driver initialize itself for a specific EPD panel. To initialize the EPDC hardware and E Ink panel, an FBIOPUT_VSCREENINFO ioctl call must be made, with the xres and yres fields of the fb_var_screeninfo parameter set to match the X and Y resolution of a supported E Ink panel type. To ensure that the EPDC driver receives the initialization request, the activate field of the fb_var_screeninfo parameter should be set to FB_ACTIVATE_FORCE.

NOTE

The exception is when the FB Console driver is included in the kernel. When the EPDC driver registers the framebuffer device, the FB Console driver will subsequently make an FBIOPUT_VSCREENINFO ioctl call. This will in turn initialize the EPDC panel.

10.3.6 Grayscale Framebuffer Selection

The EPDC framebuffer driver supports the use of 8-bit grayscale (Y8) and 8-bit inverted grayscale (Y8 inverted) pixel formats for the framebuffer (in addition to the more common RGB565 pixel format). In order to configure the framebuffer format as 8-bit grayscale, the application would call the FBIOPUT_VSCREENINFO framebuffer ioctl. This ioctl takes an fb_var_screeninfo pointer as a parameter. This parameter specifies the attributes of the framebuffer and allows the application to request changes to the framebuffer format. There are two key members of the fb_var_screeninfo parameter that must be set in order to request a change to 8-bit grayscale format: bits_per_pixel and grayscale. bits_per_pixel must be set to 8 and grayscale must be set to one of the 2 valid grayscale format values: GRAYSCALE_8BIT or GRAYSCALE_8BIT_INVERTED.

The following code snippet demonstrates a request to change the framebuffer to use the Y8 pixel format:

```
fb_screen_info screen_info;
screen_info.bits_per_pixel = 8;
screen_info.grayscale = GRAYSCALE_8BIT;
retval = ioctl(fd_fb0, FBIOPUT_VSCREENINFO, &screen_info);
```

10.3.7 Enabling an EPDC Splash Screen

By default, the EPDC support in U-Boot is disabled, and therefore splash screen support is also disabled. To enable splash screen support, edit the configuration file /include/configs/mx6sl_evk.h/include/configs/mx6dl_arm2.h and enable the following defines:

```
#define CONFIG_SPLASH_SCREEN
#define CONFIG_MXC_EPDC
```

Once this change has been made, rebuild the U-Boot image and flash it to your SD card. Then perform the following steps to flash a waveform file to an SD card where U-Boot can find it:

1. Identify the EPDC waveform file from the Linux kernel firmware directory that is the best match for the panel you are using. For the DC2/DC3 boards, that would be the waveform file /firmware/imx/epdc_E060SCM.fw.ihex.

2. Convert the ihex firmware file to a stripped-down binary using the script ihex2bin.py. Contact Freescale to acquire this script.

```
python ihex2bin.py -i epdc_E060SCM.fw.ihex -o epdc_E060SCM_splash.bin
```

3. Write the firmware file to the SD card at the FAT partition.

```
cp epdc_E060SCM.bin [FAT partition on SD card]
```

10.4 Source Code Structure

Table below lists the source files associated with the EPDC driver. These files are available in the following directory:

```
drivers/video/mxc
```

Table 10-1. EPDC Driver Files

File	Description
mxc_epdc_v2_fb.c	The EPDC V2 frame buffer driver.
epdc_v2_regs.h	Register definitions for the EPDC V2 module.

Table below lists the global header files associated with the EPDC driver. These files are available in the following directory:

```
include/linux/
```

Table 10-2. EPDC Global Header Files

File	Description
mxcfb.h	Header file for the MXC framebuffer drivers
mxcfb_epdc.h	Header file for direct kernel access to the EPDC API extension

10.5 Menu Configuration Options

The following Linux kernel configuration options are provided for the EPDC module:

- CONFIG_MXC_EINK_PANEL includes support for the Electrophoretic Display Controller. In menuconfig, this option is available under:
  - Device Drivers > Graphics Support > E Ink Panel Framebuffer
- CONFIG_MXC_EINK_AUTO_UPDATE_MODE enables support for auto-update mode, which provides automated EPD updates through the deferred I/O framebuffer

driver. This option is dependent on the MXC_EINK_PANEL option. In menuconfig, this option is available under:

- Device Drivers > Graphics Support > E Ink Auto-update Mode Support

NOTE

This option only enables the use of auto-update mode. Turning on auto-update mode requires an additional IOCTL call using the MXCFB_SET_AUTO_UPDATE_MODE IOCTL.

- CONFIG_FB to include frame buffer support in the Linux kernel. In menuconfig, this option is available under:
  - Device Drivers > Graphics support > Support for frame buffer devices
  - By default, this option is Y for all architectures.
- CONFIG_FB_MXC is a configuration option for the MXC Frame buffer driver. This option is dependent on the CONFIG_FB option. In menuconfig, this option is available under:
  - Device Drivers > Graphics support > MXC Framebuffer support
  - By default, this option is Y for all architectures.
- CONFIG_MXC_PXP_V2 enables support for the PxP. The PxP is required by the EPDC driver for processing (color space conversion, rotation, auto-waveform selection) framebuffer update regions. This option must be selected for the EPDC framebuffer driver to operate correctly. In menuconfig, this option is available under:
  - Device Drivers > DMA Engine support > MXC PxP support

10.6 Programming Interface

10.6.1 IOCTLs/Functions

The EPDC Frame Buffer is accessible from user space and from kernel space. A single set of functions describes the EPDC Frame Buffer driver extension. There are, however, two modes for accessing these functions. For user space access the IOCTL interface should be used. For kernel space access the functions should be called directly. For each function below both the IOCTL code and the corresponding kernel function is listed.

MXCFB_SET_WAVEFORM_MODES / mxc_epdc_fb_set_waveform_modes()

Description:

Defines a mapping for common waveform modes.

Parameters:

mxcfb_waveform_modes *_modes_

Pointer to a structure containing the waveform mode values for common waveform modes. These values must be configured in order for automatic waveform mode selection to function properly.

MXCFB_SET_TEMPERATURE / mxc_epdc_fb_set_temperature

Description:

Set the temperature to be used by the EPDC driver in subsequent panel updates.

Parameters:

int*32_t temperature*

Temperature value, in degrees Celsius. Note that this temperature setting may be overridden by setting the temperature value parameter to anything other than TEMP_USE_AMBIENT when using the MXCFB_SEND_UPDATE ioctl.

MXCFB_SET_AUTO_UPDATE_MODE / mxc_epdc_fb_set_auto_update

Description:

Select between automatic and region update mode.

Parameters:

__u32 mode

In region update mode, updates must be submitted via the MXCFB_SEND_UPDATE IOCTL.

In automatic mode, updates are generated automatically by the driver by detecting pages in frame buffer memory region that have been modified.

MXCFB_SET_UPDATE_SCHEME / mxc_epdc_fb_set_upd_scheme

Description:

Select a scheme that dictates how the flow of updates within the driver.

Parameters:

__u32 scheme

Select of the following updates schemes:

UPDATE_SCHEME_SNAPSHOT - In the Snapshot update scheme, the contents of the framebuffer are immediately processed and stored in a driver-internal memory buffer. By the time the call to MXCFB_SEND_UPDATE has completed, the framebuffer region is

free and can be modified without affecting the integrity of the last update. If the update frame submission is delayed due to other pending updates, the original buffer contents will be displayed when the update is finally submitted to the EPDC hardware. If the update results in a collision, the original update contents will be resubmitted when the collision has cleared.

UPDATE_SCHEME_QUEUE - The Queue update scheme uses a work queue to aynchronously handle the processing and submission of all updates. When an update is submitted via MXCFB_SEND_UPDATE, the update is added to the queue and then processed in order as EPDC hardware resources become available. As a result, the framebuffer contents processed and updated are not guaranteed to reflect what was present in the framebuffer when the update was sent to the driver.

UPDATE_SCHEME_QUEUE_AND_MERGE - The Queue and Merge scheme uses the queueing concept from the Queue scheme, but adds a merging step. This means that, before an update is processed in the work queue, it is first compared with other pending updates. If any update matches the mode and flags of the current update and also overlaps the update region of the current update, then that update will be merged with the current update. After attempting to merge all pending updates, the final merged update will be processed and submitted.

MXCFB_SEND_UPDATE / mxc_epdc_fb_send_update

Description:

Request a region of the frame buffer be updated to the display.

Parameters:

mxcfb_update_data **upd_data*

Pointer to a structure defining the region of the frame buffer, waveform mode, and collision mode for the current update. This structure also includes a flags field to select from one of the following update options:

EPDC_FLAG_ENABLE_INVERSION - Enables inversion of all pixels in the update region.

EPDC_FLAG_FORCE_MONOCHROME - Enables full black/white posterization of all pixels in the update region.

EPDC_FLAG_USE_ALT_BUFFER - Enables updating from an alternate (non-framebuffer) memory buffer.

If enabled, the final *upd_data* parameter includes detailed configuration information for the alternate memory buffer.

MXCFB_WAIT_FOR_UPDATE_COMPLETE / mxc_epdc_fb_wait_update_complete

Description:

Block and wait for a previous update request to complete.

Parameters:

mxfb_update_marker_data *marker_data*

The update_marker value used to identify a particular update (passed as a parameter in MXCFB_SEND_UPDATE IOCTL call) should be re-used here to wait for the update to complete. If the update was a collision test update, the collision_test variable will return the result indicating whether a collision occurred.

MXCFB_SET_PWRDOWN_DELAY / mxc_epdc_fb_set_pwrdown_delay

Description:

Set the delay between the completion of all updates in the driver and when the driver should power down the EPDC and the E Ink display power supplies.

Parameters:

int32_t *delay*

Input delay value in milliseconds. To disable EPDC power down altogether, use FB_POWERDOWN_DISABLE (defined below).

MXCFB_GET_PWRDOWN_DELAY / mxc_epdc_fb_get_pwrdown_delay

Description:

Retrieve the driver's current power down delay value.

Parameters:

int32_*t delay*

Output delay value in milliseconds.

10.6.2 Structures and Defines

```
#define GRAYSCALE_8BIT                                          0x1
#define GRAYSCALE_8BIT_INVERTED                                 0x2


#define AUTO_UPDATE_MODE_REGION_MODE                            0

#define AUTO_UPDATE_MODE_AUTOMATIC_MODE                         1
```

```
#define UPDATE_SCHEME_SNAPSHOT                                                  0
#define UPDATE_SCHEME_QUEUE                                                     1
#define UPDATE_SCHEME_QUEUE_AND_MERGE                                           2

#define UPDATE_MODE_PARTIAL                                                     0x0
#define UPDATE_MODE_FULL                                                        0x1

#define WAVEFORM_MODE_AUTO                                                      257

#define TEMP_USE_AMBIENT                                                        0x1000

#define EPDC_FLAG_ENABLE_INVERSION                                              0x01
#define EPDC_FLAG_FORCE_MONOCHROME                                              0x02
#define EPDC_FLAG_USE_ALT_BUFFER                                                0x100
#define EPDC_FLAG_TEST_COLLISION 0x200

#define FB_POWERDOWN_DISABLE                                                    -1

struct mxcfb_rect {
    __u32 left; /* Starting X coordinate for update region */
    __u32 top; /* Starting Y coordinate for update region */
    __u32 width; /* Width of update region */
    __u32 height; /* Height of update region */
};

struct mxcfb_waveform_modes {
    int mode_init; /* INIT waveform mode */
    int mode_du; /* DU waveform mode */
    int mode_gc4; /* GC4 waveform mode */
    int mode_gc8; /* GC8 waveform mode */
    int mode_gc16; /* GC16 waveform mode */
    int mode_gc32; /* GC32 waveform mode */
};

struct mxcfb_alt_buffer_data {
    __u32 phys_addr; /* physical address of alternate image buffer */
    __u32 width; /* width of entire buffer */
    __u32 height; /* height of entire buffer */
    struct mxcfb_rect alt_update_region; /* region within buffer to update */
};

struct mxcfb_update_data {
    struct mxcfb_rect update_region; /* Rectangular update region bounds */
    __u32 waveform_mode; /* Waveform mode for update */
    __u32 update_mode; /* Update mode selection (partial/full) */
    __u32 update_marker; /* Marker used when waiting for completion */
    int temp; /* Temperature in Celsius */
    uint flags; /* Select options for the current update */
    struct mxcfb_alt_buffer_data alt_buffer_data; /* Alternate buffer data */
};
 struct mxcfb_update_marker_data { __u32 update_marker; __u32 collision_test; };
```

Chapter 11
Pixel Pipeline (PxP) DMA-ENGINE Driver

11.1 Introduction

The Pixel Pipeline (PxP) DMA-ENGINE driver provides a unique API, which are implemented as a dmaengine client that smooths over the details of different hardware offload engine implementations. Typically, the users of PxP DMA-ENGINE driver include EPDC driver, V4L2 Output driver, and the PxP user-space library.

11.2 Hardware Operation

The PxP driver uses PxP registers to interact with the hardware. For detailed hardware operations, see the following documents:

- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)
- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)
- *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM)
- *i.MX 6UltraLite Applications Processor Reference Manual* (IMX6ULRM)

11.3 Software Operation

11.3.1 Key Data Structs

The PxP DMA Engine driver implementation depends on the DMA Engine Framework. There are three important structs in the DMA Engine Framework which are extended by the PxP driver: struct dma_device, struct dma_chan, struct dma_async_tx_descriptor. The PxP driver implements several callback functions which are called by the DMA Engine Framework (or DMA slave) when a DMA slave (client) interacts with the DMA Engine.

The PxP driver implements the following callback functions in struct dma_device:

device_alloc_chan_resources / allocate resources and descriptors */*

device_free_chan_resources / release DMA channel's resources */*

device_tx_status / poll for transaction completion */*

device_issue_pending / push pending transactions to hardware */*

and,

device_prep_slave_sg / prepares a slave DMA operation */*

device_control / manipulate all pending operations on a channel, returns zero or error code */*

The first four functions are used by the DMA Engine Framework, the last two are used by the DMA slave (DMA client). Notably, *device_issue_pending* is used to trigger the start of a PxP operation.

The PxP DMA driver also implements the interface *tx_submit in struct dma_async_tx_descriptor*, which is used to prepare the descriptor(s) which will be executed by the engine. When tasks are received in pxp_tx_submit, they are not configured and executed immediately. Rather, they are added to a task queue and the function call is allowed to return immediately.

11.3.2 Channel Management

Although ePxP does not have multiple channels in hardware, the virtual channels are supported in the driver; this provides flexibility in the multiple instance/client design. At any time, a user can call *dma_request_channel*() to get a free channel, and then configure this channel with several descriptors (a descriptor is required for each input plane and for the output plane). When the PxP is no longer being used, the channel should be released by calling *dma_release_channel*(). Detailed elements of channel management are handled by the driver and are transparent to the client.

11.3.3 Descriptor Management

The DMA Engine processes the task based on the descriptor. One DMA channel is usually associated with several descriptors. Descriptors are recycled resources, under control of the offload engine driver, to be reused as operations complete. The extended TX descriptor packet (pxp_tx_desc), allows the user to pass PxP configuration information to the driver. This includes everything that the PxP needs to execute a processing task.

11.3.4 Completion Notification

There are two ways for an application to receive notification that a PxP operation has completed.

- Call dma_wait_for_async_tx(). This call causes the CPU to spin while it polls for the completion of the operation.
- Specify a completion callback.

The latter method is recommended. After the PxP operation completes, the PxP output buffer data can be retrieved.

For general information for DMA Engine Framework, see *Documentation/dmaengine.txt* in the Linux kernel source tree.

11.3.5 Limitations

- The driver currently does not support scatterlist objects in the way they are traditionally used. Instead of using the scatterlist parameter object to provide a chain of memory sources and destinations, the driver currently uses it to provide the input and output buffers (and overlay buffers, if needed) for one transfer.
- The PxP driver may not properly execute a series of transfers that is queued in rapid sequence. It is recommended to wait for each transfer to complete before submitting a new one.

11.4 Menu Configuration Options

The following Linux kernel configuration option is provided for this module:

Device Drivers --->

DMA Engine support --->

[*] MXC PxP support

[*] MXC PxP Client Device

11.5 Source Code Structure

The PxP driver source code is located in drivers/dma/ and include/linux/.

Chapter 12
ELCDIF Frame Buffer Driver

12.1 Introduction

The ELCDIF frame buffer driver is designed using the Linux kernel frame buffer driver framework. It implements the platform driver for a frame buffer device. The implementation uses the ELCDIF API for generic LCD low-level operations. The ELCDIF API is also defined in the ELCDIF frame buffer driver to realize low level hardware control. Only DOTCLK mode of the ELCDIF API is tested, so theoretically the ELCDIF frame buffer driver can work with a sync LCD panel driver to support a frame buffer device. The sync LCD driver is organized in a flexible and extensible manner and is abstracted from any specific sync LCD panel support. To support another sync LCD panel, the user can write a sync LCD driver by referring to the existing one.

12.2 Hardware Operation

For detailed hardware operations, see the following documents:

- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)
- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)
- *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM)

12.3 Software Operation

A frame buffer device is a memory device similar to /dev/mem and it has the same features. It can be read from, written to, or some location in it can be sought and maped using mmap(). The difference is that the memory that appears is not the whole memory, but only the frame buffer of the video hardware. The device is accessed through special

device nodes, usually located in the /dev directory, /dev/fb*. /dev/fb* also has several IOCTLs which act on it and through which information about the hardware can be queried and set. The color map handling operates through IOCTLs as well. See linux/fb.h for more information on which IOCTLs there are and which data structures they use.

The frame buffer driver implementation for i.MX 6 is abstracted from the actual hardware. The default panel driver is picked up by video mode defined in platform data or passed in with 'video=mxc_elcdif_fb:resolution, bpp=bits_per_pixel' kernel bootup command during probing, where resolution should be in the common frame buffer video mode pattern and bits_per_pixel should be the frame buffer's color depth.

12.4 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_FB_MXS [=Y|N|M] Configuration option to compile support for the MXC ELCDIF frame buffer driver into the kernel.

12.5 Source Code Structure

The frame buffer driver source code is in drivers/video/mxc/mxsfb.c.

Chapter 13
Graphics Processing Unit (GPU)

13.1 Introduction

The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D/3D graphics applications.

The 3D graphics processing unit (GPU3D) is an embedded engine that accelerates user level graphics Application Programming Interface (APIs) such as OpenGL ES 1.1, OpenGL ES 2.0, and OpenGL ES 3.0 and OpenCL 1.1EP. The 2D graphics processing unit (GPU2D) is an embedded 2D graphics accelerator targeting graphical user interfaces (GUI) rendering boost. The VG graphics processing unit (GPUVG) is an embedded vector graphic accelerator for supporting the OpenVG 1.1 graphics API and feature set. The GPU driver kernel module source is in the kernel source tree, but the libs are delivered as binary only.

Graphics Processing Unit	Hardware	Applicable Platform
3D	Vivante GC2000	6Quad/6Dual
3D	Vivante GC880	6DualLite/6Solo
3D/2D	Vivante GC400T	6SoloX
2D	Vivante GC320	6Quad/6Dual/6DualLite/6Solo/6SoloLite
Vector	Vivante GC355	6Quad/6Dual/6SoloLite

NOTE

GC400T does not support OpenGL ES 3.0.

GC880/GC400T does not support OpenCL 1.1EP, and only GC2000 supports it.

13.1.1 Driver Features

The GPU driver enables this board to provide the following software and hardware support:

- EGL (EGL is an interface between Khronos rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system) 1.4 API defined by Khronos Group.
- OpenGL ES (OpenGL® ES is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems) 1.1 API defined by Khronos Group.
- OpenGL ES 2.0 API defined by Khronos Group.
- OpenGL ES 3.0 API defined by Khronos Group.
- OpenVG (OpenVG is a royalty-free, cross-platform API that provides a low-level hardware acceleration interface for vector graphics libraries such as Flash and SVG) 1.1 API defined by Khronos Group.
- OpenCL (OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors.) 1.1 EP API defined by Khronos Group.
- OpenGL 2.1 API defined by Khronos Group.
- Automatic 3D core slowing down, when hot notification from thermal driver is active, 3D core will run at 1/64 clock.

13.1.1.1 Hardware Operation

For detailed hardware operations, seee the GPU chapters in the following documents:

- *i.MX 6Dual/6Quad Applications Processor Reference Manual* (IMX6DQRM)
- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)
- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)

13.1.1.2 Software Operation

The GPU driver is divided into two layers. The first layer is running in kernel mode and acts as the base driver for the whole stack . This layer provides the essential hardware access, device management, memory management, command queue management, context management and power management. The second layer is running in user mode, implementing the stack logic and providing the following APIs to the upper layer applications:

- OpenGL ES 1.1, 2.0, and 3.0 API

- EGL 1.4 API
- OpenVG 1.1 API
- OpenCL 1.1 EP API

13.1.1.3 Source Code Structure

Table below lists GPU driver kernel module source structure:

<Yocto_BuildDir>/linux/drivers/mxc/gpu-viv

Table 13-1. GPU Driver Files

File	Description
Kconfig Kbuild config	Kernel configure file and makefile
hal/kernel/arch	Hardware-specific driver code for GC2000, GC880, GC400T, and GC320
hal/kernel/archvg	Hardware-specific driver code for GC355
hal/kernel	Kernel mode HAL driver
hal/os/linux/kernel	OS layer HAL driver

NOTE

If you replace the whole content in this directory, the GPU kernel driver can be upgraded.

13.1.1.4 Library Structure

Table below lists GPU driver user mode library structure:

<ROOTFS>/usr/lib

Table 13-2. GPU Library Files

File	Description
libCLC.so	OpenCL frontend compiler library
libEGL.so**	EGL1.4 library
libGAL.so	GAL user mode driver
libGLES_CL.so	OpenGL ES 1.1 common lite library (without EGL API, no float point support API)
libGL.so**	OpenGL 2.1 common library
libGLES_CM.so	OpenGL ES 1.1 common library (without EGL API, include float point support API)

Table continues on the next page...

Table 13-2. GPU Library Files (continued)

File	Description
libGLESv1_CL.so**	OpenGL ES 1.1 common lite library (with EGL API, no float point support API)
libGLESv1_CM.so**	OpenGL ES 1.1 common library (with EGL API, include float point support API)
libGLESv2.so**	OpenGL ES 2.0/3.0 library
libGLSLC.so	OpenGL ES shader language compiler library
libVSC.so	OpenGL front-end compiler library
libVivanteOpenCL.so	Vivante
libOpenCL.so	OpenCL ICD wrapper library
libOpenVG.so*	OpenVG 1.1 library
libVDK.so	VDK wrapper library.
libVIVANTE.so	Vivante user mode driver.
directfb-1.6-0/gfxdrivers/libdirectfb_gal.so	DirectFB 2D acceleration library.
dri/vivante_dri.so	DRI library for OpenGL2.1.
xorg/modules/drivers/vivante_drv.so	EXA library for X11 acceleration.
libwayland-viv.so	Wayland server-side library for Vivante's EGL driver
libgc_wayland_protocol.so	Vivante Wayland Protocol Extension Library

**SONAME is used for libEGL.so, libGLESv2.so, libGLESv1_CM.so, libGLESv1_CL.so, libGL.so.

*For libOpenVG.so, there are two libraries for the OpenVG feature. libOpenVG.3d.so is the gc2000/gc880/gc400t based OpenVG library. libOpenVG.2d.so is the gc355 based OpenVG library.

- For i.MX 6Dual/Quad, both libOpenVG.3d.so and libOpenVG.2d.so can be used.
- For i.MX 6DualLite and i.MX 6SoloX, only libOpenVG.3d.so can be used.
- For i.MX 6SoloLite, only libOpenVG.2d.so can be used.
- If no SOC limitation, for the x11 backend, libOpenVG.3d.so is linked by default.
- If no SOC limitation, for framebuffer, directFB, and Wayland backends, the default openVG library is linked to libOpenVG.2d.so.

This can be done by using the following sequence of commands:

```
cd <ROOTFS>/usr/lib
sudo ln -s libOpenVG_355.so libOpenVG.so
```

13.1.1.5 API References

Refer to the following web sites for detailed specifications:

- OpenGL ES 1.1, 2.0, and 3.0 API: www.khronos.org/opengles/
- OpenCL 1.1 EP www.khronos.org/opencl/
- EGL 1.4 API: www.khronos.org/egl/
- OpenVG 1.1 API: www.khronos.org/openvg/

13.1.1.6 Menu Configuration Options

The following Linux kernel configurations are provided for GPU driver:

- CONFIG_MXC_GPU_VIV is a configuration option for GPU driver. In the menuconfig this option is available under Device Drivers > MXC support drivers > MXC Vivante GPU support > MXC Vivante GPU support.

To get to the GPU library package in Yocto, use the command `bitbake linux-imx -c menuconfig`. On the screen displayed, select **Configure the kernel** and select "Device Drivers" > "MXC support drivers" > "MXC Vivante GPU support" > "MXC Vivante GPU support"and exit. When the next screen appears select the following options to enable the GPU driver:

- Package list > gpu-viv-bin-mx6q
- This package provides proprietary binary libraries, and test code built from the GPU for framebuffer

Chapter 14
Wayland

14.1 Introduction

Wayland is a protocol for a compositor to talk to its clients as well as a C library implementation of that protocol. The compositor can be a standalone display server running on Linux kernel modesetting and evdev input devices, an X application, or a Wayland client itself. The clients can be traditional applications, X servers or other display servers.

Part of the Wayland project is also the Weston reference implementation of a Wayland compositor. The Weston compositor is a minimal and fast compositor and is suitable for many embedded and mobile use cases.

This chapter describes how to enable Wayland/Weston support on an i.MX 6 series device.

14.2 Hardware Operation

i.MX 6SoloLite only supports GAL2D acceleration, and other SOCs in i.MX 6 series support EGL3D and GAL2D acceleration.

14.3 Software Operation

This release is based on the Wayland 1.6.0 version and Weston 1.6.0 version.

14.4 Yocto Build Instructions

The instructions for Yocto build are as follows:

1. Prepare a Yocto build directory and follow the setup instructions in the *Freescale Yocto Project User's Guide* (IMXLXYOCTOUG) for Wayland.
2. Set up Yocto for Wayland in the build directory:

```
$ source fsl-setup-release.sh -b build-wayland -e wayland
```

3. Build an image.

```
$ bitbake fsl-image-weston
```

14.5 Customizing Weston

The FSL-Weston includes two compositors. One is the EGL3D compositor, which is accelerated by the GC2000 3D core. The other is GAL2D compositor accelerated by the GC320 2D core.

Weston options can be updated in the file "/etc/init.d/weston".

Table 14-1. Common options for Weston

Weston option	Description
tty	default to current tty.
device	"/dev/fb0", default frame buffer , Multi display supported in Gal2D compositor.
use-gl	EGL accelerated, defaults to be "1".
use-gal2d	GAL2D accelerated, defaults to be "0".
idle-time	Idle time in seconds.

14.5.1 Multi display supported in Weston

Multi display was supported in Gal2D compositor only. Add these options to start Weston:

```
weston --tty=1 --device=/dev/fb0,/dev/fb2 --use-gal2d=1 &
```

14.5.2 Multi buffer supported in Weston

The Weston server supports both single buffering and multi buffering. In single buffering, the damage area is rendered to the offscreen surface and blits to front buffer.The offscreen surface is used to avoid flickering. By default, the Weston server starts with single buffering.

In multi buffering, instead of rendering to offscreen, the damage area is rendered to back buffer and does the flip, but the frame rate will be restricted to the display rate. A maximum of three buffers are supported.

Before starting the Weston server, export FB_MULTI_BUFFER to control the number of buffers to be used.

Environment variables for single buffering:

```
export FB_MULTI_BUFFER=1
```

Environment variables for double buffering:

```
export FB_MULTI_BUFFER=2
```

14.6 Running Weston

Perform the following operations to run Weston:

1. Boot the i.MX 6 series device.
2. To run clients, the second button in the top bar will run weston-terminal, from which you can run clients. There are a few demo clients available in the Weston build directory, but they are all pretty simple and mostly for testing specific features in the Wayland protocol:
   - 'weston-terminal' is a simple terminal emulator, not very compliant, but works well enough for bash.
   - 'weston-flower' draws a flower on the screen, testing the frame protocol.
   - 'weston-smoke' tests SHM buffer sharing.
   - 'weston-image' loads the image files passed on the command line and shows them.

Chapter 15
On-Chip High Definition Multimedia Interface (HDMI) Driver

15.1 Introduction

The High Definition Multimedia Interface (HDMI) driver supports the on-chip DesignWare HDMI hardware module, which provides the capability to transfer uncompressed video, audio, and data using a single cable.

The HDMI driver is divided into four sub-components: A video display device driver that integrates with the Linux Frame Buffer API, an audio driver that integrates with the ALSA/SoC sub-system, a CEC driver, and a multi-function device (MFD) driver which manages the shared software and hardware resources of the HDMI driver.

The HDMI driver supports the following features:

- Integration with the MXC Display Device framework (for managing display device connections with the IPU(s))
- HDMI video output up to 1080p60 resolution
- Support for reading EDID information from an HDMI sink device
- Hotplug detection
- Support CEC
- Automated clock management to minimize power consumption
- Support for system suspend/resume
- HDMI audio playback (2, 4, 6, or 8 channels, 16bit, for sample rates 32KHz to 192KHz)
- IEC audio header information exposed through ALSA using ‘iecset’ utility

15.1.1 Hardware Operation

The HDMI module receives video data from the Image Processing Unit (IPU), audio data from the external memory interface, and control data from the CPU, as shown in the figure below.

Output data is transmitted via three Transition-Minimized Differential Signaling (TMDS) channels to an HDMI sink device external to the SoC. Additionally, the HDMI carries a VESA Data Display Channel (DDC). The DDC is an I2C interface which allows the HDMI source to query the HDMI sink for Extended Display Identification Data (EDID). A CEC channel provides optional high-level control functions between the source and sink device.

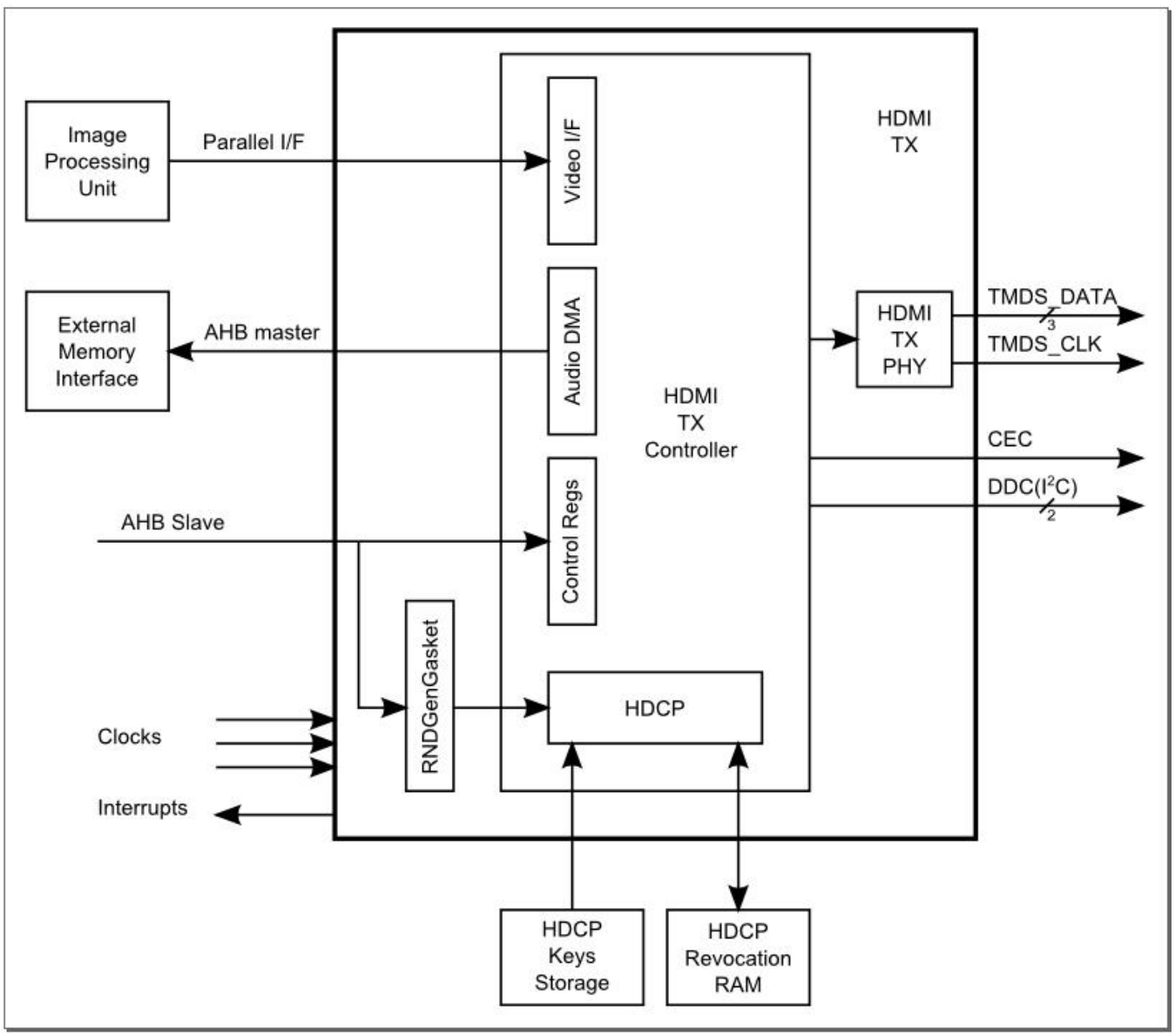


Figure 15-1. HDMI HW Integration

The video input to the HDMI is configurable and may come from either of the two IPU modules in the i.MX 6 serials and from either of the two Display Interface (DI) ports of the IPU, DI0 or DI1. This configuration is controlled through the IOMUX module using the HDMI_MUX_CTRL register field. See the figure below for an illustration of this interconnection.

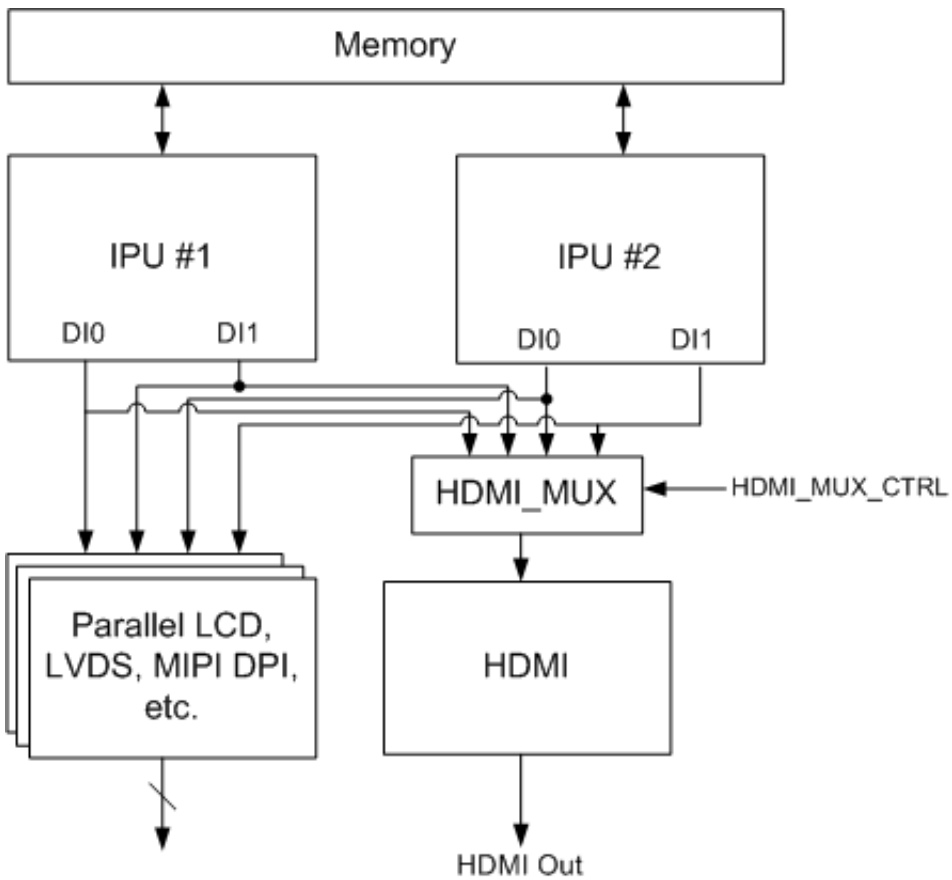


Figure 15-2. IPU-HDMI Hardware Interconnection

15.2 Software Operation

The HDMI driver is divided into sub-components based on its two primary purposes: providing video and audio to an HDMI sink device.

The video display driver component and audio driver component require an additional core driver component to manage common HDMI resources, including the HDMI registers, clocks, and IRQ.

15.2.1 Core

The HDMI core driver manages resources that must be shared between the HDMI audio and video drivers. The HDMI audio and video drivers depend on the HDMI core driver, and the HDMI core driver should always be loaded and initialized before audio and video. The core driver serves the following functions:

- Map the HDMI register region and provide APIs for reading and writing to HDMI registers
- Perform one-time initialization of key HDMI registers
- Initialize the HDMI IRQ and provide shared APIs for enabling and disabling the IRQ
- Provide a means for sharing information between the audio and video drivers (e.g., the HDMI pixel clock)
- Provide a means for synchronization between HDMI video and HDMI audio while blank/unbalnk, plug in/plug out events happen. HDMI audio can't start work while HDMI cable is in the state of plug out or HDMI is in state of blank. Every time HDMI audio starts a playback, HDMI audio driver should register its PCM into core driver and unregister PCM when the playback is finished. Once HDMI video blank or cable plug out event happens, core driver would pause HDMI audio DMA controller if its PCM is registered. When HDMI is unblanked or cable plug in event happens, core driver would firstly check if the cable is in the state of plug in, the video state is unblank and the PCM is registered. If items listed above are all yes, core driver would restart HDMI audio DMA.

15.2.2 Video

The following diagram illustrates both the interconnection between the various HDMI sub-drivers and the interconnection between the HDMI video driver and the Linux Frame Buffer subsystem.

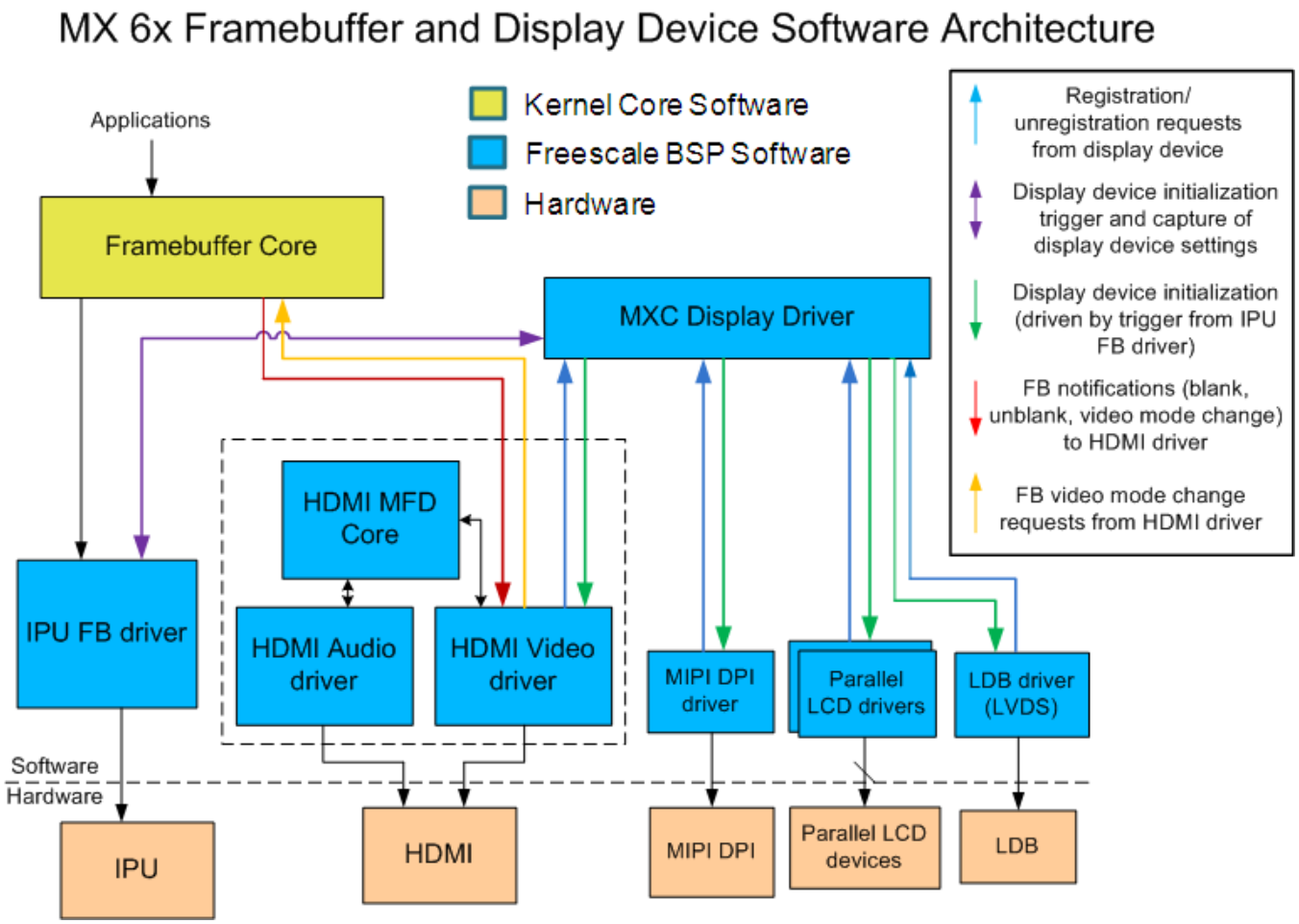


Figure 15-3. HDMI Video SW Architecture

The i.MX 6Dual/6Quad/6Solo/6DualLite/6SoloLite supports many different types of display output devices (e.g., LVDS, LCD, HDMI, and MIPI displays) connected to and driven by the IPU modules. The MXC Display Driver API provides a system for registering display devices and configuring how they should be connected to each of the IPU DIs. The HDMI driver registers itself as a display device using this API in order to receive the correct video input from the IPU.

15.2.3 Display Device Registration and Initialization

The following sequence of software activities occurs in the OS boot flow to connect the HDMI display device to the IPU FB driver through the MXC Display Driver system:

1. During the HDMI video driver initialization, `mxc_dispdrv_register()` is called to register the HDMI module as a display device and to set the `mxc_hdmi_disp_init()` function as the display device init callback.

2. When the IPU FB driver is initialized, `mxc_dispdrv_init()` is called. This results in an init call to all registered display devices.
3. The `mxc_hdmi_disp_init()` callback is executed. The HDMI driver receives a structure from the IPU FB driver containing frame buffer information (fbi). The HDMI driver also provides return information about which IPU and DI to select and the preferred output format for video data from the IPU. The HDMI driver registers itself to receive notifications of FB driver events. Finally, the HDMI driver can complete its initialization by configuring the HDMI to receive a hotplug interrupt.

NOTE

All display device drivers must be initialized before the IPU FB driver, in order for all display devices to be registered as MXC Display Driver devices before the IPU FB driver can initialize them.

15.2.4 Hotplug Handling and Video Mode Changes

Once the connection between the IPU and the HDMI has been established through the MXC Display Driver interface, the HDMI video driver waits for a hotplug interrupt, indicating that a valid HDMI sink device is connected and ready to receive HDMI video data. Subsequent communications between the HDMI and IPU FB are conducted through the Linux Frame Buffer APIs. The following list demonstrates the software flow to recognize an HDMI sink device and configure the IPU FB driver to drive video output to it:

1. The HDMI video driver receives a hotplug interrupt and reads the EDID from the HDMI sink device, constructing a list of video modes from the retrieved EDID information. Using either the video mode string from the Linux kernel command line (for the initial connection) or the most recent video mode (for a later HDMI cable connection), the HDMI driver selects a video mode from the mode list that is the closest match.
2. The HDMI video driver calls `fb_set_var()` to change the video mode in the IPU FB driver. The IPU FB driver completes its reconfiguration for the new mode.
3. As a result of calling fb_set_var(), an FB notification is sent back to the HDMI driver indicating that an FB_EVENT_MODE_CHANGE has occurred. The HDMI driver configures the HDMI hardware for the new video mode..
4. In the final step, the HDMI module is enabled to generate output to the HDMI sink device.

15.2.5 Audio

The HDMI Tx audio driver uses the ALSA SoC framework, so it is broken into several files, as is listed in Table 15-4. Most of the code is in the platform DMA driver (sound/soc/imx/imx-hdmi-dma.c). The machine driver (sound/soc/fsl/imx-hdmi.c) exists to allocate the SoC audio device and link all the SoC components together. The DAI driver (sound/soc/fsl/fsl-hdmi-dai.c) mostly exists because SoC wants there to be a DAI driver; it gets the platform data, but doesn't do anything else.

The HDMI codec driver does most of the initialization of the HDMI audio sampler. Note that the HDMI Tx block only implements the AHB DMA audio and not the other audio interfaces (SSI, S/PDIF, etc.). The other main function of the HDMI codec driver is to set up a struct of the IEC header information which needs to go into the audio stream. This struct is hooked into the ALSA layer, so the IEC settings will be accessible in userspace using the 'iecset' utility.

The platform DMA driver handles the HDMI Tx block's DMA engine. Note that HDMI audio uses the HDMI block's DMA as well as SDMA. SDMA is used to help implement the multi-buffer mechanism. The HDMI Tx block does not automatically merge the IEC audio header information into the audio stream, so the platform DMA driver does this in its hdmi_dma_copy()(for no memory map use) or hdmi_dma_mmap_copy()(for memory map mode use) function before the DMA sends the buffers out. Also note that, due to IEC audio header adding operation, it is possible that user space application is not able to get enough CPU periods to feed data into HDMI audio driver in time, especially when system loading is high. In this situation, some spark noise would be heard. In different audio framework(ALSA LIB, or PULSE AUDIO), different log about this noise may be printed. For example, in ALSA LIB, logs like "underrung!!! at least * ms is lost" are printed.

HDMI audio playback depends on HDMI pixel clock. So while in the state of HDMI blank and cable plug out, HDMI audio would be stopped or can't be played. See detailed information in software_operation_core.

Also note that, because HDMI audio driver need to add IEC header, driver need to know how many data has application already write into HDMI audio driver. If application is not able to tell how many data is wrote (for example, DMIX plugin in ALSA LIB), HDMI audio driver is not able to work properly. There would be no sound heard.

The HDMI audio support features below:

- Playback sample rate
  - 32k, 44.1k, 48k, 88.2k, 96k, 176.4k, 192k
  - capability of HDMI sink
- Playback Channels:

- 2, 4, 6, 8
- capability of HDMI sink

- Playback audio formats:
  - SNDRV_PCM_FMTBIT_S16_LE

15.2.6 CEC

HDMI CEC is a protocol that provides high-level control functions between all of the various audiovisual products is a user's environment. The HDMI CEC driver implements software part of HDMI CEC low Level protocol. It includes getting Logical address, CEC message sending and receiving, error handle, message re-transmitting, and etc.

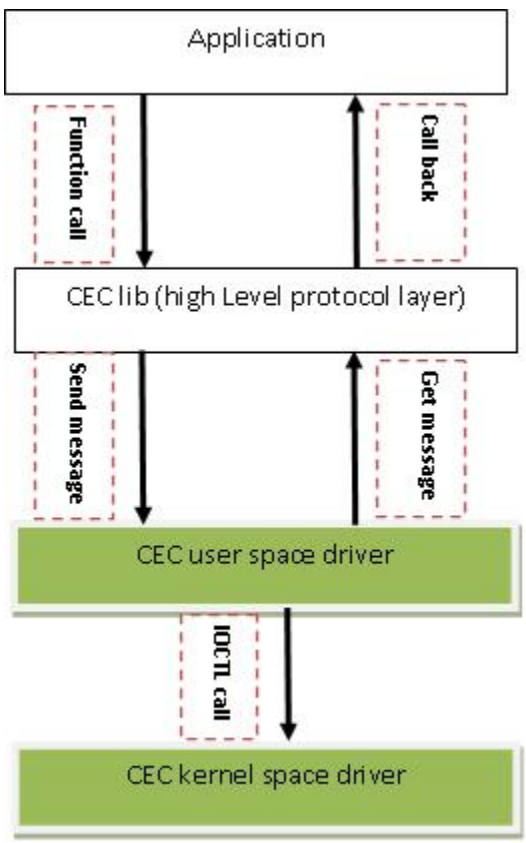


Figure 15-4. HDMI CEC SW Architecture

15.3 Source Code Structure

The bulk of the source code for the HDMI driver is divided amongst the three software components that comprise the driver: the HDMI core driver, the HDMI display driver, and the HDMI audio driver.

Additional platform-specific source code files provide the code for declaring and registering these HDMI drivers.

The source code for the HDMI core driver is available in the <Yocto_BuildDir>/linux/drivers/mfd/ directory.

Table 15-1. HDMI Core Driver File List

File	Description
mxc-hdmi-core.c	HDMI core driver implemention

A public header for the HDMI core driver is available in the <Yocto_BuildDir>/linux/include/linux/mfd/ directory.

Table 15-2. HDMI Core Display Driver Public Header File List

File	Description
mxc-hdmi-core.h	HDMI core driver header file

The source code for the HDMI display driver is available in the driver/video/mxc directory.

Table 15-3. HDMI Display Driver File List

File	Description
mxc_hdmi.c	HDMI display driver implemention

The source code for the HDMI audio driver is available in the <Yocto_BuildDir>/linux/drivers and sound/soc/ directory. Although the HDMI is one hardware block, the audio driver is divided into four c files corresponding to the ALSA SoC layers:

Table 15-4. HDMI Audio Driver File List

File	Description
fsl/fsl_hdmi.c	HDMI Audio SoC DAI driver implementation
fsl/imx-hdmi-dma.c	HDMI Audio SoC platform DMA driver implementation
fsl/imx-hdmi.c	HDMI Audio SoC machine driver implementation

The source code for the HDMI CEC driver is available in the <Yocto_BuildDir>/linux/drivers/mxc/ directory.

Table 15-5. HDMI CEC Driver File List

File	Description
drivers/mxc/hdmi-cec.c	HDMI CEC driver implemention

The source code for the HDMI lib is available in the <Yocto_BuildDir>/imx-lib/hdmi-cec/ directory.

Table 15-6. HDMI CEC lib File List

File	Description
hdmi-cec/mxc_hdmi-cec.c	HDMI CEC lib implemention
hdmi-cec/hdmi-cec.h	HDMI CEC lib header file
hdmi-cec/android.mk	HDMI CEC lib make file

The following platform-level source code files provide structures and functions for registering the HDMI drivers. These files can be found in the <Yocto_BuildDir>/linux/arch/arm/plat-mxc/ directory.

Table 15-7. HDMI Platform File List

File	Description
devices/platform-mxc-hdmi-core.c	HDMI core driver platform device code
devices/platform-mxc_hdmi.c	HDMI display driver platform device code
devices/platform-imx-hdmi-soc.c	HDMI audio driver platform device code
devices/platform-imx-hdmi-soc-dai.c	HDMI audio driver platform device code
include/mach/mxc_hdmi.h	HDMI register defines

15.3.1 Linux Menu Configuration Options

There are three main Linux kernel configuration options used to select and include HDMI driver functionality in the Linux OS image.

The CONFIG_FB_MXC_HDMI option provides support for the HDMI video driver, and can be selected in menuconfig at the following menu location:

Device Drivers > Graphics support > MXC HDMI driver support

HDMI video support is dependent on support for the Synchronous Panel Framebuffer and also on the inclusion of IPUv3 support.

The CONFIG_SND_SOC_IMX_HDMI option provides support for HDMI audio through the ALSA/SoC subsystem, and can be found in menuconfig at the following location:

Device Drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio support for IMX - HDMI

Selecting either of the previous two configuration options will cause the MXC HDMI Core configuration option, CONFIG_MFD_MXC_HDMI, to be selected. This option can also be found in the menuconfig here:

Device Drivers > Multifunction device drivers > MXC HDMI Core

The CONFIG_MXC_HDMI_CEC option provides support for the HDMI CEC driver, and can be selected in menuconfig at the following menu location:

Device Drivers > MXC support drivers > MXC HDMI CEC (Consumer Electronic Control) support

15.4 Unit Test

The HDMI video and audio drivers each have their own set of tests.

The HDMI video driver does not lend itself well to automated testing, so a number of manual tests are required to verify the correct functionality. For audio driver testing, the `aplay` audio file player and `iecset` utility provide confirmation of the the driver's proper integration into the ALSA framework. The following two section look at unit testing for both the HDMI audio and video drivers.

15.4.1 Video

The following set of manual tests can be used to verify the proper operation of the HDMI video driver:

1. Linux kernel command line-based tests: The initial mode used to display HDMI video can be specified through the Linux kernel command line boot parameters. Try several different valid display resolutions through the kernel parameters, re-booting the system each time and verifying that the desired resolution is displayed on the connected HDMI display.
2. Hotplug testing: Connect and disconnect the HDMI cable several times, from either the end attached to the i.MX board, or the end attached to the HDMI sink device. Each time the cable is reconnected, the driver should re-determine the appropriate video mode, based on the modes read via EDID from the HDMI sink, and display that mode on the sink device.
3. HDMI output device testing: Test by dynamically switching the HDMI sink device. The HDMI driver should be able to detect the valid video modes for each different HDMI sink device and provide video to that display that is closest to the most recent video mode configured in the HDMI driver.

15.4.2 Audio

The following sequence of tests can verify the correct operation of the HDMI audio driver:

1. Ensure that an HDMI cable is connected between the i.MX board and the HDMI sink device, and that the HDMI video image is being properly displayed on the device.
2. Use 'aplay -l' (that's a single dash and a lower-case L) to list out the audio playback cards and determine which the card number is. This is different on our various boards.
3. For example, if the HDMI ends up being card 2, use this command line to play out a pcm audio file "file.wav":

   ```
   $ aplay -Dplughw:2,0 file.wav
   ```
4. Use 'iecset' to list out the IEC information about the device. You will need to specify card number like:

   ```
   $ iecset -c2
   ```

NOTE

Note that HDMI audio is dependent on a reasonable pixel clock rate being established. If this is not the case, error messages indicating “pixel clock not supported” will appear. This is because there is no clock regenerator cts value that could be calculated for the current pixel clock.

15.4.3 CEC

The following test can be used to simple verify HDMI CEC function:

```
$ /unit_test/mxc_cec_test
```

Bootup device and connect HDMI sink to board, then run the above command, the HDMI CEC will send Poweroff command to HDMI sink.

15.4.4 HDCP

The following test can be used to verify the HDMI HDCP function. You need to make sure that the HDMI HDCP function is supported by the i.MX 6 part.

Use HDCP, specifically DTB imx6q-sabresd-hdcp.dtb, and boot up the SABRE-SD board.

Run the following commands:

```
$ /unit_tests/mxc_hdcp_app.out &
$ echo 1 > /sys/devices/soc0/soc.X/20e0000.hdmi_video/hdcp_enable
```

If the HDCP function is not support by the i.MX 6 part or TV, the screen displays the RED picture.

Chapter 16
External High-Definition Multimedia Interface (HDMI) for i.MX 6SoloLite

16.1 Introduction

The High Definition Multimedia Interface (HDMI) driver supports the external SiI9022 HDMI hardware module, which provides the capability to transfer uncompressed video, audio, and data using a single cable.

The HDMI driver is divided into two sub-components: a video display device driver that integrates with the Linux Frame Buffer API and an S/PDIF audio driver that transfers S/PDIF audio data to SiI9022 HDMI hardware module.

The HDMI driver is only for demo application and supports the following features:

- HDMI video output supports 1080p60 and 720p60 resolutions.
- Support for reading EDID information from an HDMI sink device for video.
- Hotplug detection
- HDMI audio playback (2 channels, 16/24 bit, 44.1 KHz sample rate)

16.2 Software Operation

The HDMI driver is divided into sub-components based on its two primary purposes: providing video and audio to an HDMI sink device.

The audio output depends on video display.

16.2.1 Hotplug Handling and Video Mode Changes

Once the connection between the ELCDIF and the HDMI has been established through the MXC Display Driver interface, the HDMI video driver waits for a hotplug interrupt indicating that a valid HDMI sink device is connected and ready to receive HDMI video data. Subsequent communications between the HDMI and LECDIF FB are conducted through the Linux Frame Buffer APIs. The following list demonstrates the software flow to recognize a HDMI sink device and configure the ELCDIF FB driver to drive video output:

1. The HDMI video driver receives a hotplug interrupt and reads the EDID from the HDMI sink device constructing a list of video modes from the retrieved EDID information. Using either the video mode string from the Linux kernel command line (for the initial connection) or the most recent video mode (for a later HDMI cable connection), the HDMI driver selects a video mode from the mode list that is the closest match.
2. The HDMI video driver calls `fb_set_var()` to change the video mode in the ELCDIF FB driver. The ELCDIF FB driver completes its reconfiguration for the new mode.
3. As a result of calling fb_set_var(), a FB notification is sent back to the HDMI driver indicating that an FB_EVENT_MODE_CHANGE has occurred. The HDMI driver configures the HDMI hardware for the new video mode.
4. Finally, the HDMI module is enabled to generate output to the HDMI sink device.

16.3 Source Code Structure

The bulk of the source code for the HDMI driver is divided amongst the three software components that comprise the driver: the HDMI display driver, and the HDMI audio driver.

The source code for the HDMI display driver is available in the <Yocto_BuildDir>/rpm/BUILD/linux/drivers/video/mxc directory.

Table 16-1. HDMI Display Driver File List

File	Description
mxcfb_sii902x_elcdif.c	HDMI display driver implementation.

The source code for the HDMI audio driver is available in the <Yocto_BuildDir>/rpm/BUILD/linux/drivers and sound/soc/ director. HDMI Audio data source comes from S/PDIF TX.

Table 16-2. HDMI Audio Driver File List

File	Description
sound/codecs/mxc_spdif.c	S/PDIF Audio SoC CODEC driver implementation.
sound/soc/imx/imx-spdif.c	S/PDIF Audio SoC Machine driver implementation.
sound/soc/imx/imx-spdif-dai.c	S/PDIF Audio SoC DAI driver implementation.
sound/soc/imx/imx-pcm-dma-mx2.c	S/PDIF Audio SoC platform layer driver implementation.

16.3.1 Linux Menu Configuration Options

There are two main Linux kernel configuration options used to select and include HDMI driver functionality in the Linux OS image.

The CONFIG_FB_MXC_SII902X_ELCDIFI option provides support for the Sii902x HDMI video driver and can be selected in menuconfig at the following menu location:

- Device Drivers > Graphics support > MXC Framebuffer support.

HDMI video support is dependent on MXC ELCDIF Framebuffer.

The CONFIG_SND_MXC_SPDIF option provides support for the HDMI Audio driver and can be selected in menuconfig at the following menu location:

- Device Drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio for Freescale i.MX CPUs > SoC Audio support for IMX - S/PDIF

16.4 Unit Test

The HDMI video and audio drivers each have their own set of tests.

The preparation for HDMI test:

- Insert the HDMI daughter card into J13 on the i.MX 6SoloLite EVK board.
- Insert the HDMI cable into the HDMI slots of both HDMI daughter board and the HDMI sink device.
- Power on the HDMI sink device.

16.4.1 Video

The following set of manual tests can be used to verify the proper operation of the HDMI video driver:

1. Hotplug testing: Connect and disconnect the HDMI cable several times, from either the end attached to the i.MX board, or the end attached to the HDMI sink device. Each time the cable is reconnected, the driver should re-determine the appropriate video mode based on the modes read via EDID from the HDMI sink and display that mode on the sink device.
2. HDMI output device testing: Test by dynamically switching the HDMI sink device. The HDMI driver should be able to detect the valid video modes for each different HDMI sink device and provide video to that display that is closest to the most recent video mode configured in the HDMI driver.

16.4.2 Audio

The following sequence of tests verifies the correct operation of the HDMI audio driver:

1. Ensure that an HDMI cable is connected between the HDMI daughter board and the HDMI sink device, and that the HDMI video image is being properly displayed on the device.
2. Use this command line to play out a pcm audio file "file.wav" to HDMI sink device:

```
$ aplay -Dplughw:1,0 file.wav
```

Chapter 17
X Windows Acceleration

17.1 Introduction

X-Windows System (aka X11 or X) is a portable, client-server based, graphics display system.

X-Windows system can run with a default frame buffer driver which handles all drawing operations to the main display. Since there is a 2D GPU (graphics processing unit) available, then some drawing operations can be accelerated. High level X operations may get decomposed into low level drawing operations which are accelerated for X-Windows System.

17.2 Hardware Operation

X-Windows System acceleration on i.MX 6 utilizes the Vivante GC320 2D GPU.

Acceleration is also dependent on the frame buffer memory.

17.3 Software Operation

X-Windows acceleration is supported for X.org X Server version 1.11.x and later versions supporting the EXA interface version 2.5.

The following list summarizes the types of operations that are accelerated for X11. All operations involve frame buffer memory which may be on screen or off screen:

- Solid fill of a rectangle.
- Upload image in system memory into video memory.

- Copy of a rectangle with same pixel format with possible source-target rectangle overlap.
- Copy of a rectangle supporting most XRender compositing operations with these options:
  - Pixel format conversion.
  - Repeating pattern source.
  - Porter-Duff blending of source with target.
  - Source alpha masking.

The following list includes additional features supported as part of the X-Windows acceleration:

- Allocation of X pixmaps directly in frame buffer memory.
- EGL swap buffers where the EGL window surface is an X-window.
- X-window can be composited into an X pixmap which can be used directly as any EGL surface.

17.3.1 X-Windows Acceleration Architecture

The following block diagram shows the components that are involved in the acceleration of X-Windows System:

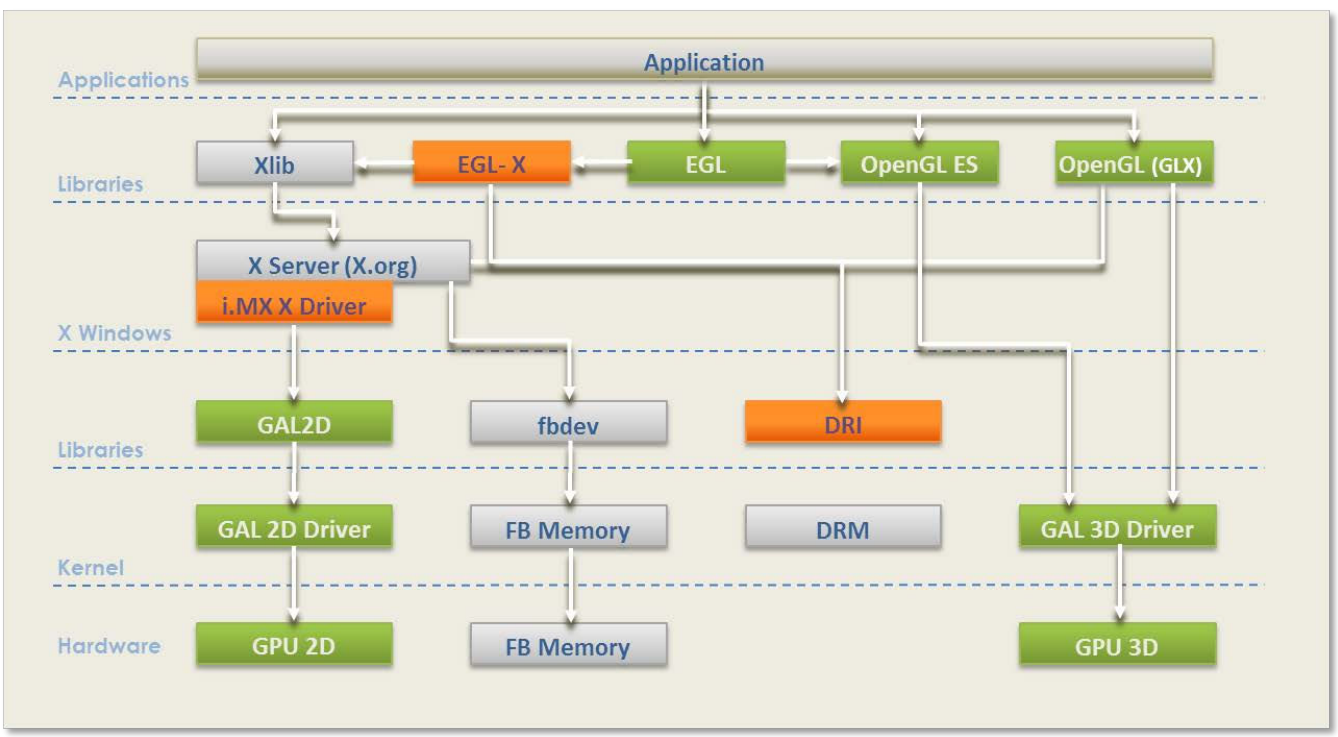


Figure 17-1. X Driver Architecture

The components shown in green are those provided as part of the Vivante 2D/3D GPU driver support which includes OpenGL/ES and EGL, though some i.MX 6 processors, such as i.MX 6SoloLite do not contain 3D HW module. The components shown in light gray are the standard components in the X-Windows System without acceleration. The components shown in orange are those added to support X-Windows System acceleration and briefly described here.

The **i.MX X Driver** library module (`vivante-drv.so`) is loaded by the X server and contains the high level implementation of the X-Windows acceleration interface for i.MX platforms containing the GC320 2D GPU core. The entire linearly contiguous frame buffer memory in `/dev/fb0` is used for allocating pixmaps for X both on screen and off screen. The driver supports a custom X extension which allows X clients to query the GPU address of any X pixmap stored in frame buffer memory.

The **libGAL.so** library module (`libGAL.so`) contains the register level programming interface to the GC320 GPU module. This includes the storing of register programming commands into packets which can be streamed to the device. The functions in the libGAL.so library are called by the i.MX X Driver code.

The **EGL-X** library module (`libEGL.so`) contains the X-Windows implementation of the low level EGL platform-specific support functions. This allows X-window and X pixmap objects to be used as EGL window and pixmap surfaces. The EGL-X library uses Xlib function calls in its implementation along with the i.MX X Driver module's X extension for querying the GPU address of X pixmaps stored in frame buffer memory.

17.3.2 i.MX 6 Driver for X-Windows System

The i.MX X Driver, referred to as vivante-drv.so, implements the EXA interface of the X server in providing acceleration.

The Vivante X Driver, referred to as vivante-drv.so, implements the EXA interface of the X server to provide acceleration.

The following list describes details particular to this implementation:

- The implementation builds upon the source from the fbdev frame buffer driver for X so that it can be the fallback when the acceleration is disabled.
- The implementation is based on X server EXA version 2.5.0.
- The EXA solid fill operation is accelerated, except for source/target drawables containing less than 300x300 pixels in which case fallback is to software rendering.
- The EXA copy operation is accelerated, except for source/target drawables containing less than 400x120 pixels in which case fallback is to software rendering.

- EXA putimage (upload into video memory) is accelerated, except for source drawables containing less than 400x400 pixels in which case fallback is to software rendering.For EXA solid fill and copy operations, only solid plane masks and only GXcopy raster-op operations are accelerated.
- For EXA copy operation, the raster-op operations (GXandInverted, GXnor, GXorReverse, GXorInverted, and GXnand) are not accelerated.
- EXA composite allows for many options and combinations of source/mask/target for rendering.
- Most of the (commonly used) EXA composite operations are accelerated.

The following types of EXA composite operations are accelerated:

- Composite operations for source/target drawables containing at least 640 pixels. If less than 640 pixels, the composite path falls to software.
- Simple source composite operations are used when source/target drawables contain more than 200x200 pixels (operations with mask not supported).
- Constant source (with or without alpha mask) composite with target.
- Repeating pattern source (with or without alpha mask) composite with target.
- Only these blending functions: SOURCE, OVER, IN, IN-REVERSE, OUT-REVERSE, and ADD (some of these are needed to support component-alpha blending which is accelerate).
- In general, the following types of (less commonly used) EXA composite operations are not accelerated:
    - Transformed (that is, scaled, rotated) sources and masks
    - Gradient sources
    - Alpha masks with repeating patterns

The implementation handles all pixmap allocation for X through the EXA callback interface. A first attempt is made to allocate the memory where it can be accessed by a physical GPU address. This attempt can fail if there is insufficient GPU accessible memory remaining, but it can also fail when the bits per pixel being requested for the pixmap is less than eight (8). If the attempt to allocate from the GPU accessible memory fails, then the memory is allocated from the system. If the pixmap memory is allocated from the system, then this pixmap cannot be involved in a GPU accelerated option. The number of pitch bytes used to access the pixmap memory may be different depending on whether it was allocated from GPU accessible memory or from the system. Once the memory for an X pixmap has been allocated, whether it is from GPU accessible memory or from the system, the pixmap is locked and can never migrate to the other type of memory. Pixmap migration from GPU accessible memory to system memory is not necessary since a system virtual address is always available for GPU accessible memory. Pixmap migration from system memory to GPU accessible memory is not currently implemented, but would only help in situations where there was insufficient GPU accessible memory at initial allocation but more memory becomes available (through de-

allocation) at a later time. The GPU accessible memory pitch (horizontal) alignment for Vivante 2D GPUs is 8 pixels. Because the memory can be allocated from GPU accessible memory, these pixels could be used in EGL for OpenGL/ES drawing operations. All of the memory allocated for /dev/fb0 is made available to an internal linear offscreen memory manager based on the one used in EXA. The portion of this memory beyond the screen memory is available for allocation of X pixmap, where this memory area is GPU accessible. The amount of memory allocated to /dev/fb0 needs to be several MB more than the amount needed for the screen. The actual amount needed depends on the number of X-Windows and pixmaps used, the possible usage of X pixmaps as textures, and whether X-Windows are using the XComposite extension. An X extension, i.e., VIVEXT shown in Fig. 1, is provided so that X clients can query the physical GPU address associated with an X pixmap, if that X pixmap was allocated in the GPU accessible memory.

17.3.3 i.MX 6 Direct Rendering Infrastructure (DRI) for X-Windows System

The Direct Rendering Infrastructure, also known as the DRI, is a framework for allowing direct access to graphics hardware under the X Window System in a safe and efficient manner. It includes changes to the X server, to several client libraries, and to the kernel (DRM, Direct Rendering Manager). The most important activity for the DRI is to create fast OpenGL and OpenGL ES implementations that render to framebuffer memory directly. Without DRI, the OpenGL driver has to depend on X server for final rendering (indirect rendering), which degrades the overall performance significantly.

The components of Vivante's DRI OpenGL implementation include:

- The Direct Rendering Manager (DRM) is a kernel module that provides APIs to userland to synchronize access to hardware and to manage different classes of video memory buffers. Vivante's DRI implementation uses selected DRM APIs for opening/closing DRI device, and locking/unlocking FB. Most other buffer management and DMA management functions are handled by Vivante's specific kernel module: galcore.ko.
- The EXA driver is a DRI-enabled DDX 2D driver which initializes the DRM when X server starts. As all X Window pixmap buffers are allocated by the EXA driver from GPU memory, the GPU can render directly into these buffers if the buffer information is passed from the X server process to the X client processes (GL or GLES applications) properly.
- The Vivante-specific X extension "vivext" passes buffer information from X server to X clients. This Vivante X extension includes the following three interfaces:

- DrawableFlush, which enables X clients to notify X server to flush the GPU cache for a drawable surface.
- DrawableInfo, which enables X clients to query the drawable information (position, size, physical address, stride, cliplist, etc.) from the X server.
- PixmapPhysAddr, which enables X clients to query the physical address and stride of a pixmap buffer from X server.

The integration of GL/GLES application windows with Ubuntu Unity2D desktop is achieved by following steps:

- GL/GLES applications render a frame into the pixmap buffers that are allocated in the EXA driver.
- In the SwapBuffers implementation, the driver notifies X server that the pixmap buffer region is damaged through Xdamage and Xfixes APIs.
- Then the X server will present the latest pixmap buffer to the Unity2D desktop while maintaining the proper window overlap characteristics relative to the other windows on the desktop.

On a compositing X desktop, such as Ubuntu Unity 2D, GLES/GL applications always render into the full rectangular back buffer of a window. There is no window clipping required. So the Vivante DRI implementation can take advantage of the GPU's resolve function and render into the window back buffer directly.

On a legacy X window desktop, such as Gnome, Xwin, etc., GLES/GL applications have to render onto the frame buffer surface directly. Thus, the DRI driver uses the DrawableInfo interface in the VIVEXT extension to obtain the cliplist of the window, then copies the sub-regions of the render target to the frame buffer according to the cliplist. This will ensure that the GLES/GL windows overlap with other windows on the desktop properly. However, the copying of the render target sub-regions to the frame buffer has to be done by the CPU as the sub-regions' starting address and alignment may not meet GPU copy requirements.

The Vivante DRI implementation can detect the type of X window manager (compositing desktop manager or legacy desktop manager) at run-time, and use appropriate DRI rendering paths for GLES/GL applications.

17.3.4 EGL- X Library

The EGL-X library implements the low level EGL interface when used in X Window System. The following list describes details particular to this implementation:

- The eglDisplay native display type is "Display*" in X.

- The eglWindowSurfacenative window surface type is “Window” in X.
- The eglPixmapSurface native pixmap surface type is “Pixmap” in X.

When an eglWindowSurface is created, the back buffers used for double-buffering can have different representations from the window surface (based on the selected eglConfig). An attempt is made to create each back buffer using the representation which provides the most efficient blit of the back buffer contents to the window surface when eglSwapBuffers is called.

The back buffer is allocated by creating an X pixmap of the necessary size. Use the X extension for the Vivante X Driver module to query the physical frame buffer address for this X pixmap if it was allocated in the offscreen frame buffer memory.

17.3.5 xorg.conf for i.MX 6

The `/etc/X11/xorg.conf` file must be properly configured to use the i.MX 6 X Driver.

The /etc/X11/xorg.conf file must be properly configured to use the Vivante X Driver. This configuration appears in a “Device” section of the file which contains some required entries and some entries that are optional. The following example shows a preferred configuration for using the Vivante X Driver:

```
Section "ServerLayout"
        Identifier      "Default Layout"
        Screen          "Default Screen"
EndSection

Section "Module"
        Load            "dbe"
        Load            "extmod"
        Load            "freetype"
        Load            "glx"
        Load            "dri"
EndSection

Section "InputDevice"
        Identifier      "Generic Keyboard"
        Driver          "kbd"
        Option          "XkbLayout" "us"
        Option          "XkbModel" "pc105"
        Option          "XkbRules" "xorg"
EndSection

Section "InputDevice"
        Identifier      "Configured Mouse"
        Driver          "mouse"
        Option          "CorePointer"
EndSection

Section "Device"
        Identifier      "Your Accelerated Framebuffer Device"
        Driver          "vivante"
        Option          "fbdev"         "/dev/fb0"
        Option          "vivante_fbdev" "/dev/fb0"
```

```
        Option          "HWcursor"      "false"
EndSection

Section "Monitor"
        Identifier      "Configured Monitor"
EndSection

Section "Screen"
        Identifier      "Default Screen"
        Monitor         "Configured Monitor"
        Device          "Your Accelerated Framebuffer Device"
        DefaultDepth    24
EndSection

Section "DRI"
        Mode 0666
EndSection
```

Mandatory Strings

Some important entries recognized by the Vivante X Driver are described as follows.

Device Identifier and Screen Device String

The mandatory Identifier entry in the Device section specifies the unique name to associate with this graphics device.

```
Section "Device"
        Identifier      "Your Accelerated Framebuffer Device"
```

The following entry ties a specific graphics device to a screen. The Device Identifier string must match the Device string in a Screensection of the xorg.conf file. For example:

```
Section "Screen"
        Identifier      "Default Screen"
        <other entries>
        Device          "Your Accelerated Framebuffer Device"
        <other entries>
EndSection
```

Device Driver String

The mandatory Driver entry specifies the name of the loadable Vivante X driver.

Driver "vivante"

Device fbdevPath Strings

The mandatory entries fbdev and vivante_dev specify the path for the frame buffer device to use.

```
Section "Device"
        Identifier      "Your Accelerated Framebuffer Device"
        Driver          "vivante"
        Option          "fbdev"       "/dev/fb0"
        Option          "vivante_fbdev" "/dev/fb0"
        <other entries>
EndSection
```

17.3.6 Setup X-Windows System Acceleration on Yocto

Prerequisites:

- xserver-xorg-video-imx-viv-<BSP Version>.tar.gz, which is Vivante EXA plugin source code based on GPU driver 4.6.9p12
- xserver-xorg, which should be the Xorg 1.11.x or above
- drm-update-arm.patch, which is a patch with adding the ARM lock implementation for libdrm xf86drm.h. Note that the original xh86drm.h header file from libdrm does not have lock for supporting ARM architecture. This patch is located in $YOCTO_BUILDER/sources/meta-fsl-bsp-release/imx/meta-fsl-arm/recipes-graphics/drm/libdrm/mx6, and shown below: drm-update-arm.patch:

```
+#elif defined(__arm__)
+       #undef DRM_DEV_MODE
+       #define DRM_DEV_MODE     (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)
+
+       #define DRM_CAS(lock,old,new,__ret)             \
+       do {                                            \
+               __asm__ __volatile__ (                  \
+                       "1: ldrex %0, [%1]\n"           \
+                       "   teq %0, %2\n"               \
+                       "   strexeq %0, %3, [%1]\n"     \
+               : "r" (__ret)                           \
+               : "r" (lock), "r" (old), "r" (new)      \
+               : "cc","memory");                       \
+       } while (0)
+
 #endif /* architecture */
 #endif /* __GNUC__ >= 2 */
```

Build and install instructions:

- Install the prerequisites modules or patches in the appropriate locations and with right recipes in Yocto environment.
- Build XServer with correct drm header file (xf86drm.h). The purpose is to create correct dri module
- Build GPU EXA module with the command ‘bitbake xf86-video-imxfb-vivante’. vivante_drv.so will be generated with successful build, and then install it together with xorg and libdri library in target board rootfs in /usr/lib/xorg/modules/
- Install the pre-Yocto-built gpu-viv binary which is built based on gpu-viv version 4.6.9p12 in target board rootfs. For accelerating X11, the X11 backend is required
- Now ready to run the X11 applications in target board.

NOTE

x11 applications hangs if the ARM core version xf86drm.h is not used

17.3.7 Setup X Window System Acceleration

- Install any packages appropriate for your platform.
- Verify that the device file /dev/galcore is present.
- Verify that the file /etc/X11/xorg.conf contains the correct entries as described in the previous section.
- Assuming the above steps have been performed, do the following to verify that X Window System acceleration is indeed operating.
- Examine the log file /var/log/Xorg.0.log and confirm that the following lines are present.

```
 [    41.752] (II) Loading /usr/lib/xorg/modules/drivers/vivante_drv.so
        [    41.752] (II) VIVANTE(0): using default device
        [    41.752] (II) VIVANTE(0): Creating default Display subsection in Screen
section "Default Screen" for depth/fbbpp 24/32
        [    41.752] (**) VIVANTE(0): Depth 24, (--) framebufferbpp 32
        [    41.752] (==) VIVANTE(0): RGB weight 888
        [    41.752] (==) VIVANTE(0): Default visual is TrueColor
        [    41.753] (==) VIVANTE(0): Using gamma correction (1.0, 1.0, 1.0)
        [    41.753] (II) VIVANTE(0): hardware: DISP3 BG (video memory: 8100kB)
        [    41.753] (II) VIVANTE(0): checking modes against framebuffer device...
        [    41.753] (II) VIVANTE(0): checking modes against monitor...
        [    41.753] (--) VIVANTE(0): Virtual size is 1920x1080 (pitch 1920)
        [    41.753] (**) VIVANTE(0):  Built-in mode "current": 148.5 MHz, 67.5 kHz,
60.0 Hz
        [    41.753] (II) VIVANTE(0): Modeline "current"x0.0  148.50  1920 2008 2052
2200  1080 1084 1089 1125 +hsync +
       vsync -csync (67.5 kHz)
        [    41.753] (==) VIVANTE(0): DPI set to (96, 96)
        [    41.753] (II) Loading sub module "fb"
        [    41.753] (II) LoadModule: "fb"
        [    41.754] (II) Loading /usr/lib/xorg/modules/libfb.so
        [    41.755] (II) Module fb: vendor="X.Org Foundation"
        [    41.755] compiled for 1.10.4, module version = 1.0.0
        [    41.755] ABI class: X.Org ANSI C Emulation, version 0.4
        [    41.755] (II) Loading sub module "exa"
        [    41.755] (II) LoadModule: "exa"
        [    41.756] (II) Loading /usr/lib/xorg/modules/libexa.so
        [    41.756] (II) Module exa: vendor="X.Org Foundation"
        [    41.756] compiled for 1.10.4, module version = 2.5.0
        [    41.756] ABI class: X.Org Video Driver, version 10.0
        [    41.756] (--) Depth 24 pixmap format is 32 bpp
        [    41.797] (II) VIVANTE(0): FB Start = 0x33142000  FB Base = 0x33142000  FB
Offset = (nil)
        [    41.797] (II) VIVANTE(0): test Initializing EXA
        [    41.798] (II) EXA(0): Driver allocated offscreenpixmaps
        [    41.798] (II) EXA(0): Driver registered support for the following
operations:
        [    41.798] (II)         Solid
        [    41.798] (II)         Copy
        [    41.798] (II)         Composite (RENDER acceleration)
        [    41.798] (II)         UploadToScreen
        [    42.075] (==) VIVANTE(0): Backing store disabled
        [    42.084] (==) VIVANTE(0): DPMS enabled
```

17.3.8 Troubleshooting

1. Framebuffer devices can be specified by environment variable. This is especially useful when there are multiple framebuffer devices.

```
export FB_FRAMEBUFFER_0=/dev/fb2
```

2. If the above does not resolve the issue:
   - If DRM booted up properly, check the /var/log/X11.n log file (n will represent instance number) for more information.
   - If DRM did not boot properly, check your kernel mode driver installation. (See sections 6.4.2 and 6.4.3 above).
3. Window is created, but nothing is drawn
   - If you run an OpenGL application and find a window was created, but nothing was drawn, try to export the ${__GL_DEV_FB} environment variable:

```
export __GL_DEV_FB=$FB_FRAMEBUFFER_0.
```

4. Cannot open Display message
   - If you have a message similar to “Cannot open Display,” use the following command to check whether X is running at :0 or at :1 instance, use:

```
$ ps -ef|grep X
```

   - Then depending on the returned instance number, add the following environment variable

```
export DISPLAY=:n
```

   - then run again.
5. UART terminal cannot run GPU application with lightdm
   - Use ssh terminal instead.
6. EXA build script failure
   - Check the log file and make sure your system time is set correctly.
7. Invalid MIT-MAGIC-COOKIE-1 Key error message
   - Some GPU applications are not permitted to run using root. Use an alternate account instead.
8. Segment fault occurs while running GPU application
   - Check the attribute for dev/galcore should be updated to 666.
   - To update this attribute automatically on system boot,
   - Locate and edit file /etc/udev/rules.d/<bsp-specific.rules>.
   - Add: “KERNEL==”galcore”,MODE=”0666””
   - Lastly, make sure your kernel and GPU drivers are matched.
9. Check whether Compiz is running
   - If your host or target has issues after installing the OpenGL Development Packages in Table 6, above, check whether or not compiz is running with the following command:

```
$ ps -ef|grep compiz
```

- If compiz is running, then Ubuntu is using Unity3D by default. To set the default window manager to Unity2D:
- Locate and edit file /var/lib/AccountsService/users/<username>.
- Change ubuntu to ubunto-2d.

Chapter 18
Video Processing Unit (VPU) Driver

18.1 Hardware Operation

The VPU hardware performs all of the codec computation and most of the bitstream parsing/packeting.

Therefore, the software takes advantage of less control and effort to implement a complex and efficient multimedia codec system.

The VPU hardware data flow is shown in the MPEG4 decoder example in Figure below.

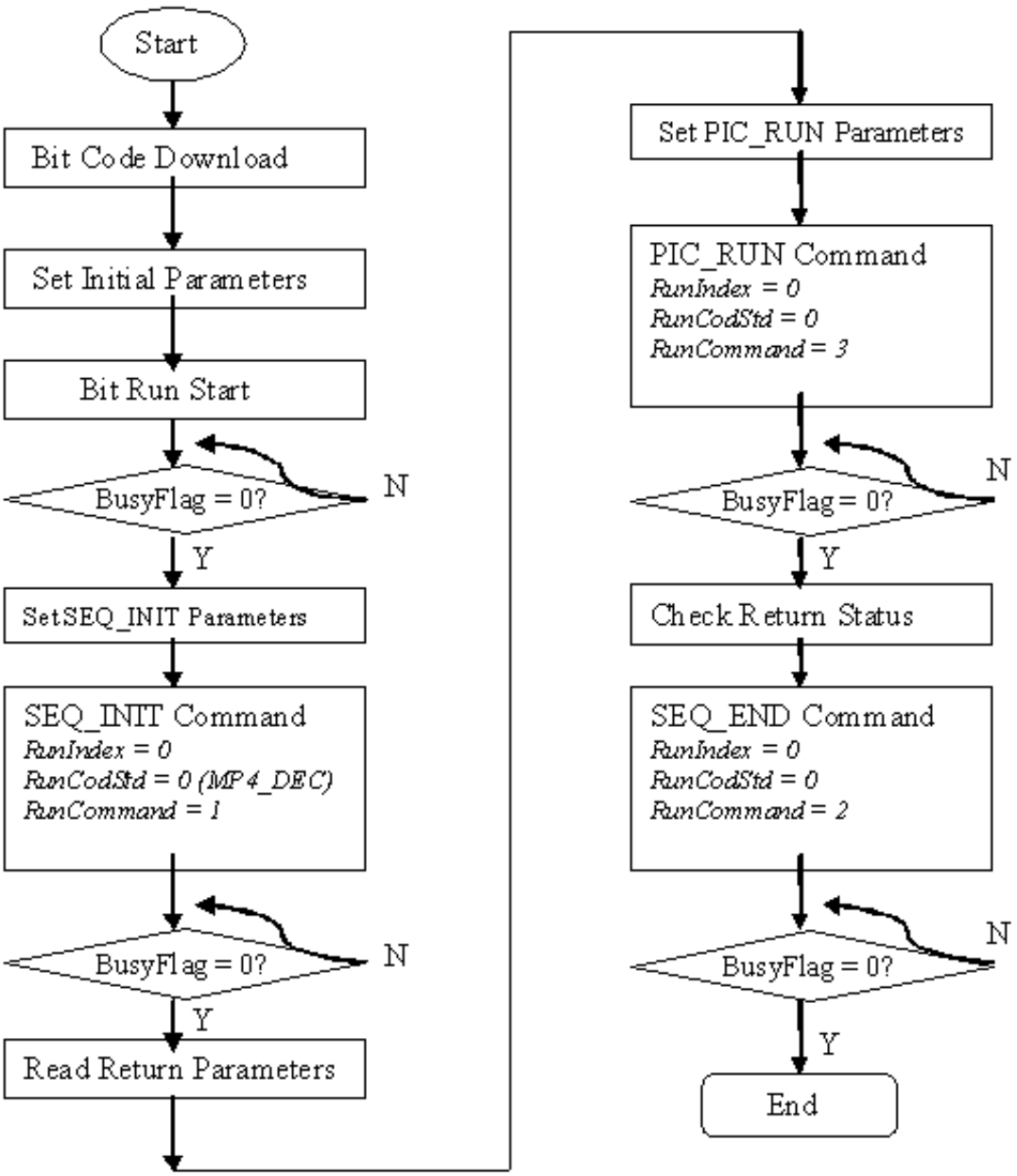


Figure 18-1. VPU Hardware Data Flow

18.1.1 Software Operation

The VPU software can be divided into two parts: the kernel driver and the user-space library as well as the application in user space. The kernel driver takes responsibility for system control and reserving resources (memory/IRQ). It provides an IOCTL interface

for the application layer in user-space as a path to access system resources. The application in user-space calls related IOCTLs and codec library functions to implement a complex codec system.

The VPU kernel driver includes the following functions:

- Module initialization which initializes the module with the device-specific structure
- Device initialization which initializes the VPU clock and hardware and request the IRQ
- Interrupt servicing routine which supports events that one frame has been finished
- File operation routine which provides the following interfaces to user space:
- File open
- File release
- File synchronization
- File IOCTL to provide interface for memory allocating and releasing
- Memory map for register and memory accessing in user space
- Device Shutdown-Shutdowns the VPU clock and hardware, and release the IRQ

The VPU user space driver has the following functions:

- Codec lib
- Downloads executable bitcode for hardware
- Initializes codec system
- Sets codec system configuration
- Controls codec system by command
- Reports codec status and result
- System I/O operation
- Requests and frees memory
- Maps and unmaps memory/register to user space
- Device management

18.1.2 Source Code Structure

Table below lists the kernel space source files available in the following directories:

```
<Yocto_BuildDir>/linux/arch/arm/plat-mxc/include/mach/

<Yocto_BuildDir>/linux/drivers/mxc/vpu/
```

Table 18-1. VPU Driver Files

File	Description
mxc_vpu.h	Header file defining IOCTLs and memory structures
mxc_vpu.c	Device management and file operation interface implementation

Table below lists the user-space library source files available in the <Yocto_BuildDir>/ imx-lib-11.11.00/vpu directory:

Table 18-2. VPU Library Files

File	Description
vpu_io.c	Interfaces with the kernel driver for opening the VPU device and allocating memory
vpu_io.h	Header file for IOCTLs
vpu_lib.c	Core codec implementation in user space
vpu_lib.h	Header file of the codec
vpu_reg.h	Register definition of VPU
vpu_util.c	File implementing common utilities used by the codec
vpu_util.h	Header file

Table below lists the firmware files available in the following directories:

`<Yocto_BuildDir>/firmware-imx-11.11.00/lib/firmware/vpu/ directory`

Table 18-3. VPU firmware Files

File	Description
vpu_fw_xxx.bin	VPU firmware

NOTE

To get the to files in Table 18-2, run the command: bitbake linux-imx -c menuconfig prep -p imx-lib in the console

18.1.3 Menu Configuration Options

To get to the VPU driver, use the command bitbake linux-imx -c menuconfig. On the screen displayed, select **Configure the kernel** and exit. When the next screen appears select the following options to enable the VPU driver:

- CONFIG_MXC_VPU-Provided for the VPU driver. In menuconfig, this option is available under
- Device Drivers > MXC support drivers > MXC VPU (Video Processing Unit) support

18.1.4 Programming Interface

There is only a user-space programming interface for the VPU module. A user in the application layer cannot access the kernel driver interface directly. The VPU library accesses the kernel driver interface for users.

The codec library APIs are listed below:

```
RetCode vpu_Init(void *);
void vpu_UnInit(void);
RetCode vpu_GetVersionInfo(vpu_versioninfo * verinfo);

RetCode vpu_EncOpen(EncHandle* pHandle, EncOpenParam* pop);
RetCode vpu_EncClose(EncHandle encHandle);
RetCode vpu_EncGetInitialInfo(EncHandle encHandle, EncInitialInfo* initialInfo);
RetCode vpu_EncRegisterFrameBuffer(EncHandle handle, FrameBuffer * bufArray,
                                                        int num, int frameBufStride, int
sourceBufStride,
                                                        PhysicalAddress subSampBaseA,
PhysicalAddress subSampBaseB,
                                                        ExtBufCfg *scratchBuf);
RetCode vpu_EncGetBitstreamBuffer(EncHandle handle, PhysicalAddress* prdPrt,
                                                       PhysicalAddress* pwrPtr, Uint32*
size);
RetCode vpu_EncUpdateBitstreamBuffer(EncHandle handle, Uint32 size);
RetCode vpu_EncStartOneFrame(EncHandle encHandle, EncParam* pParam);
RetCode vpu_EncGetOutputInfo(EncHandle encHandle, EncOutputInfo* info);
RetCode vpu_EncGiveCommand (EncHandle pHandle, CodecCommand cmd, void* pParam);
RetCode vpu_DecOpen(DecHandle* pHandle, DecOpenParam* pop);
RetCode vpu_DecClose(DecHandle decHandle);
RetCode vpu_DecGetBitstreamBuffer(DecHandle pHandle, PhysicalAddress* pRdptr,
                                        PhysicalAddress* pWrptr, Uint32* size);
RetCode vpu_DecUpdateBitstreamBuffer(DecHandle decHandle, Uint32 size);
RetCode vpu_DecSetEscSeqInit(DecHandle pHandle, int escape);
RetCode vpu_DecGetInitialInfo(DecHandle decHandle, DecInitialInfo* info);
RetCode vpu_DecRegisterFrameBuffer(DecHandle decHandle, FrameBuffer* pBuffer, int num,
                                        int stride, DecBufInfo* pBufInfo);
RetCode vpu_DecStartOneFrame(DecHandle handle, DecParam* param);
RetCode vpu_DecGetOutputInfo(DecHandle decHandle, DecOutputInfo* info);
RetCode vpu_DecBitBufferFlush(DecHandle handle);
RetCode vpu_DecClrDispFlag(DecHandle handle, int index);
RetCode vpu_DecGiveCommand(DecHandle pHandle, CodecCommand cmd, void* pParam);
int vpu_IsBusy(void);
int vpu_WaitForInt(int timeout_in_ms);
RetCode vpu_SWReset(DecHandle handle, int index);
```

System I/O operations are listed below:

```
int IOGetPhyMem(vpu_mem_desc* buff);
int IOFreePhyMem(vpu_mem_desc* buff);
int IOGetVirtMem (vpu_mem_desc* buff);
int IOFreeVirtMem(vpu_mem_desc* buff);
```

18.1.5 Defining an Application

The most important definition for an application is the codec memory descriptor. It is used for request, free, mmap and munmap memory as follows:

```
typedef struct vpu_mem_desc
{
          int size;                             /*request memory size*/
          unsigned long phy_addr;               /*physical memory get from system*/
          unsigned long cpu_addr;               /*address for system usage while freeing,
user doesn't need
                                                     to handle or use it*/
          unsigned long virt_uaddr; /*virtual user space address*/
} vpu_mem_desc;
```

See the *i.MX 6 VPU Application Programming Interface Linux® Reference Manual* for how to use API in the application (document IMXVPUAPI).

Chapter 19
OmniVision Camera Driver

19.1 OV5640 Using MIPI CSI-2 interface

This is an introduction for ov5640 camera driver which using MIPI CSI-2 interface.

19.1.1 Hardware Operation

The OV5640 is a small camera sensor and lens module with low power consumption. The camera driver is located under the Linux V4L2 architecture. and it implements the V4L2 capture interfaces. Applications cannot use the camera driver directly. Instead, the applications use the V4L2 capture driver to open and close the camera for preview and image capture, controlling the camera, getting images from camera, and starting the camera preview.

The OV5640 uses the serial camera control bus (SCCB) interface to control the sensor operation. It works as an I^2C client, V4L2 driver uses I^2C bus to control camera operation.

OV5640 supports two transfer mode: parallel interface and MIPI interface.

When using MIPI mode, OV5640 connects to i.MX AP chip by MIPI CSI-2 interface. MIPI receives the sensor data and transfers them to IPU CSI.

See the OV5640 datasheet to get more information on the sensor.

For more information on MIPI CSI-2 and IPU CSI, see the following documents:

- *i.MX 6Dual/6Quad Applications Processor Reference Manual* (IMX6DQRM)
- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)

- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)
- *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM)

19.1.2 Software Operation

The camera driver implements the V4L2 capture interface and applications and uses the V4L2 capture interface to operate the camera.

The supported operations of V4L2 capture are:

- Capture stream mode

The supported picture formats are:

- YUV422P
- UYVY
- YUV420

The supported picture sizes are:

- QVGA
- VGA
- 720P
- 1080P

19.1.3 Source Code Structure

Table below shows the camera driver source files available in the directory.

`<Yocto_BuildDir>/linux/drivers/media/video/mxc/capture.`

Table 19-1. Camera Driver Files

File	Description
ov5640_mipi.c	Camera driver implementation for ov5640 using MIPI CSI-2 interface

19.1.4 Linux Menu Configuration Options

The following Linux kernel configuration option is provided for this module.

To get to this option, use the bitbake linux-imx -c menuconfigcommand. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following option to enable this module:

- Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > OmniVision ov5640 camera support using mipi.

19.2 OV5642 Using parallel interface

This is an introduction for ov5642 camera driver which using parallel interface.

19.2.1 Hardware Operation

The OV5642 is a small camera sensor and lens module with low power consumption. The camera driver is located under the Linux V4L2 architecture. and it implements the V4L2 capture interfaces. Applications cannot use the camera driver directly. Instead, the applications use the V4L2 capture driver to open and close the camera for preview and image capture, controlling the camera, getting images from camera, and starting the camera preview.

The OV5642 uses the serial camera control bus (SCCB) interface to control the sensor operation. It works as an I^2C client, V4L2 driver uses I^2C bus to control camera operation.

OV5642 supports only parallel interface.

See the OV5642 datasheet to get more information on the sensor.

For more information on IPU CSI, see the following documents:

- *i.MX 6Dual/6Quad Applications Processor Reference Manual* (IMX6DQRM)
- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)
- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)
- *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM)

19.2.2 Software Operation

The camera driver implements the V4L2 capture interface and applications and uses the V4L2 capture interface to operate the camera.

The supported operations of V4L2 capture are:

- Capture stream mode
- Capture still mode

The supported picture formats are:

- YUV422P
- UYVY
- YUV420

The supported picture sizes are:

- QVGA
- VGA
- 720P
- 1080P
- QSXGA

19.2.3 Source Code Structure

Table below shows the camera driver source files available in the directory.

```
<Yocto_BuildDir>/linux/drivers/media/video/mxc/capture.
```

Table 19-2. Camera Driver Files

File	Description
ov5642.c	Camera driver implementation for ov5642 using parallel interface

19.2.4 Linux Menu Configuration Options

The following Linux kernel configuration option is provided for this module.

To get to this option, use the bitbake linux-imx -c menuconfigcommand. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following option to enable this module:

- Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > OmniVision ov5642 camera support.

Chapter 20
MIPI CSI2 Driver

20.1 Introduction

MIPI CSI-2 for i.MX 6 is MIPI-Camera Serial Interface Host Controller. It is a high performance serial interconnect bus for mobile application which connects camera sensors to the host system. The CSI-2 Host Controller is a digital core that implements all protocol functions defined in the MIPI CSI-2 Specification. In doing so, it provides an interface between the system and the MIPI D-PHY and allows communication with MIPI CSI-2-compliant Camera Sensor.

The MIPI CSI2 driver is used to manage the MIPI D-PHY and lets it co-work with MIPI sensor and IPU CSI. MIPI CSI2 driver implements functions as follows:

- MIPI CSI-2 low-level interface for managing the mipi D-PHY register and clock
- MIPI CSI-2 common API for communication between MIPI sensor and MIPI D-PHY

By calling MIPI common APIs, MIPI sensor can set certain information about sensor (such as datatype, lanes number, etc.) to MIPI CSI2 driver to configure D-PHY. In order for the IPU CSI module driver to have the correct configuration, receive appropriate data, and process it correctly, it is necessary for it to receive information about sensor (such as datatype, virtual channel, IPU ID, CSI ID, etc.) from the MIPI CSI2 driver.

20.1.1 MIPI CSI2 Driver Overview

MIPI CSI2 driver is invoked only by the MIPI sensor driver and IPU CSI module and is not exposed to the user space.

MIPI CSI2 driver supports the following features:

- Support 1~4 lanes
- Support IPU(0,1) and CSI(0,1) select

- Support virtual channel select(0~3)
- Support date type includes:
  - RGB formats: RGB888, RGB666, RGB565, RGB555, RGB444
  - YUV formats: YUV422 8bit, YUV422 10bit, YUV420 8bit, YUV420 10bit
  - RAW data: RAW6, RAW7, RAW8, RAW10, RAW12, RAW14

20.1.2 Hardware Operation

There are four blocks in the MIPI CSI-2 D-PHY: PHY adaptation layer, packet analyzer, image date interface, and register bank.

Functions and operations are listed as follows:

- PHY Adaptation Layer is responsible for managing the D-PHY interface including PHY error handling;
- Packet Analyzer is responsible for data lane merging if required, together with header decoding, error detection and correction, frame size error detection and CRC error detection;
- Image Date Interface separates CSI-2 packet header information and reorders data according to memory storage format. It also generates timing accurate video synchronization signals. Several error detections are also performed at frame-level and line-level;
- Register Bank is accessible through a standard AMBA-APB slave interface and provides access to the CSI-2 Host Controller register for configuration and control. There is also a fully programmable interrupt generator to inform the system upon certain events;

20.2 Software Operation

MIPI CSI2 driver for Linux OS has two parts: MIPI CSI2 driver initialize operation which initializes mipi_csi2_info struct, and MIPI CSI2 common APIs which exports APIs for CSI module driver and MIPI sensor driver.

20.2.1 MIPI CSI2 Driver Initialize Operation

MIPI CSI driver first initializes mipi_csi2_info struct, some key information about mipi sensor will be initialized, such as connected IPU ID, CSI ID, the virtual channel and date type. Then, the driver initializes D-PHY clock and pixel clock (pixel clock is used for MIPI D-PHY to transfer data to IPU CSI). After these operations, MIPI CSI csi2 driver waits for sensor connection.

20.2.2 MIPI CSI2 Common API Operation

MIPI CSI2 driver exports many APIs to manage MIPI D-PHY.

The following is the introduction for all APIs:

- mipi_csi2_get_info: get the mipi_csi_info
- mipi_csi2_enable: enable MIPI CSI interface
- mipi_csi2_disable: disable MIPI CSI interface
- mipi_csi2_get_status: get MIPI CSI interface disable/enable status
- mipi_csi2_get_bind_ipu: get the IPU ID which MIPI CSI will connect
- mipi_csi2_get_bind_csi: get the CSI ID which MIPI CSI will connect
- mipi_csi2_get_virtual_channel: get the virtual channel number by which MIPI sensor transfers data to MIPI D-PHY
- mipi_csi2_set_lanes: set the lanes number by which MIPI sensor transfers data to MIPI D-PHY
- mipi_csi2_set datatype: set the MIPI sensor data type
- mipi_csi2_get_datatype: get the MIPI sensor data type; This function is called by CSI module to set the CSI register
- mipi_csi2_dphy_status: get the MIPI D-PHY status
- mipi_csi2_get_error1: get the MIPI error1 register information
- mipi_csi2_get_error2: get the MIPI error2 register informaiton
- mipi_csi2_pixelclk_enable: enable the pixel clock
- mipi_csi2_pixelclk_disable: disable the pixel clock
- mipi_csi2_reset: reset the MIPI D-PHY for data receiving and transferring

20.3 Driver Features

MIPI CSI2 driver supports the following features:

- Support 1~4 lanes
- Support IPU(0,1) and CSI(0,1) select

- Support virtual channel select(0~3)
- Support date type includes:
  - RGB formats: RGB888, RGB666, RGB565, RGB555, RGB444
  - YUV formats: YUV422 8bit, YUV422 10bit, YUV420 8bit, YUV420 10bit
  - RAW data: RAW6, RAW7, RAW8, RAW10, RAW12, RAW14

20.3.1 Source Code Structure

Table below shows the MIPI CSI2 driver source files available in the directory.

<Yocto_BuildDir>/linux/drivers/mxc/mipi.

Table 20-1. MIPI CSI2 Driver Files

File	Description
mxc_mipi_csi2.c	MIPI CSI driver source file

20.3.2 Menu Configuration Options

The following Linux kernel configuration option is provided for this module.

To get to this option, use the bitbake linux-imx -c menuconfigcommand. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

Device Drivers > MXC support drivers > MXC MIPI Support > MIPI CSI2 support.

20.3.3 Programming Interface

MIPI CSI2 Common APIs can only be called by mipi sensor driver and IPU CSI module driver.

Before calling the API, in system initialization stage, use mipi_csi2_platform_data struct and imx6q_add_mipi_csi2 function to add a MIPI CSI2 driver.

For mipi sensor driver, the initialization steps are:
- get MIPI info by calling mipi_csi2_get_info()
- enable MIPI CSI interface by calling mipi_csi2_enable()
- set the lanes by calling mipi_csi2_set_lanes()
- reset the MIPI D-PHY by calling mipi_csi2_reset()
- configure MIPI sensor

- wait for MIPI D-PHY to receive the sensor clock and data until clock and data are stable by calling mipi_csi2_dphy_status() and mipi_csi2_get_error1()
- when uninstall the sensor driver, disable MIPI CSI interface by calling mipi_csi2_disable()

For sample code which explains how mipi sensor uses mipi APIs, reference ov5640_mipi driver source code.

For IPU CSI module driver, the call steps are:

- get MIPI info by calling mipi_csi2_get_info()
- get IPU id and CSI id to assure configuration of the correct CSI module by calling mipi_csi2_get_bind_ipu() and mipi_csi2_get_bind_csi()
- get datatype and virtual channel from MIPI CSI driver and configure the CSI module by calling mipi_csi2_get_datatype() and mipi_csi2_get_virtual_channel()
- perform other configure operation for CSI module and enable CSI
- enable the pixel clock to transfer data from MIPI D-PHY to IPU CSI by calling mipi_csi2_pixelclk_enable()
- when all tasks are done, disable CSI module first, then disable mipi pixel clock by calling mipi_csi2_pixelclk_disable()

For sample code which explains how the CSI module driver uses MIPI APIs, reference IPU CSI module driver source code.

20.3.4 Interrupt Requirements

No interrupt is needed for MIPI CSI driver.

Chapter 21
Low-level Power Management (PM) Driver

21.1 Hardware Operation

Information found here describes the low-level Power Management (PM) driver which controls the low-power modes.

The i.MX 6 supports four low power modes: RUN, WAIT, STOP, and DORMANT.

Table below lists the detailed clock information for the different low power modes.

Table 21-1. Low Power Modes

Mode	Core	Modules	PLL	CKIH/FPM	CKIL
RUN	Active	Active, Idle or Disable	On	On	On
WAIT	Disable	Active, Idle or Disable	On	On	On
STOP	Disable	Disable	Off	On	On
DORMANT	Power off	Disable	Off	Off	On

For the detailed information about lower power modes, see the following documents:

- *i.MX 6Dual/6Quad Applications Processor Reference Manual* (IMX6DQRM)
- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)
- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)
- *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM)
- *i.MX 6UltraLite Applications Processor Reference Manual* (IMX6ULRM)

21.1.1 Software Operation

The i.MX 6 PM driver maps the low-power modes to the kernel power management states as listed below:

- Standby-maps to STOP mode which offers significant power saving, as all blocks in the system are put into a low-power state, except for ARM core, which is still powered on, and memory is placed in self-refresh mode to retain its contents.
- Mem (suspend to RAM)which maps to DORMANT mode which offers most significant power saving as all blocks in the system are put into a low-power state, except for memory, which is placed in self-refresh mode to retain its contents
- System idle which maps to WAIT mode
- If ARM® Cortex®-M4 processor is alive together with ARM® Cortex®-A9 processor before the kernel enters standby/mem mode, and if ARM Cortex-M4 processor is not in its low power idle mode, ARM Cortex-A9 processor triggers the SOC to enter WAIT mode instead of STOP mode to make sure that ARM Cortex-M4 processor can continue running.

The i.MX 6 PM driver performs the following steps to enter and exit low power mode:

1. Allow the Cortex-A9 platform to issue a deep sleep mode request.
2. If STOP or DORMANT mode:
    - Program CCM CLPCR register to set low power control register.
    - If DORMANT mode, request switching off CPU power when pdn_req is asserted.
    - Request switching off embedded memory peripheral power when pdn_req is asserted.
    - Program GPC mask register to unmask wakeup interrupts.
3. Call cpu_do_idle to execute WFI pending instructions for wait mode.
4. Execute imx6_suspend in IRAM.
5. If in DORMANT mode, save ARM context, change the drive strength of MMDC PADs as "low" to minimize the power leakage in DDR PADs. Execute WFI pending instructions for stop mode.
6. Generate a wakeup interrupt and exit low power mode. If DORMANT mode, restore ARM core and DDR drive strength.

In DORMANT mode, the i.MX 6 can assert the VSTBY signal to the PMIC and request a voltage change. The U-Boot or Machine Specific Layer (MSL) usually sets the standby voltage in STOP mode according to i.MX 6 data sheet.

21.1.2 Source Code Structure

Table below shows the PM driver source files. These files are available in:

```
<Yocto_BuildDir>/arch/arm/mach-imx/
```

Table 21-2. PM Driver Files

File	Description
pm-imx6.c	Supports suspend operation
suspend-imx6.S	Assembly file for CPU suspend

21.1.3 Menu Configuration Options

The following Linux kernel configuration options are provided for this module. To get to these options, use the bitbake linux-imx -c menuconfigcommand. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_PM builds support for power management. In menuconfig, this option is available under:
  - Power management options > Power Management support
  - By default, this option is Y.
- CONFIG_SUSPEND builds support for suspend. In menuconfig, this option is available under:
  - Power management options > Suspend to RAM and standby

21.1.4 Programming Interface

The imx6_set_lpm API in the system.c function is provided for low-power modes. This implements all the steps required to put the system into WAIT and STOP modes.

21.1.5 Unit Test

To enter different system level low power modes:

```
echo mem > /sys/power/state
echo standby > /sys/power/state
```

To wake up system from low power modes, enable the wakeup source first, such as USB device, debug UART, or RTC, which can be used as a wakeup source. Below is the example of UART wakeup:

```
echo enabled > /sys/bus/platform/drivers/imx-uart/'xxxxxxx'.serial/tty/ttymxc'y'/power/
```

```
wakeup;
```

Here 'xxxxxxx' is the physical base address of your debugging UART. For example, for UART1, it is 2020000. 'y' is your debugging UART index.

To test this mode automatically, refer to our script in /unit_tests/suspend_random_auto.sh or /unit_tests/suspend_quick_auto.sh.

For FreeRTOS running with Linux OS together, press "s" on the FreeRTOS console to start the test. FreeRTOS will enter or exit its low power idle mode in a random period.

Chapter 22
PF100 Regulator Driver

22.1 Introduction

PF100 is a PMIC chip which is specified by i.MX 6.

PF200/PF3000 is baed on PF100 with little change, since they share the same PF100 driver. PF100 regulator driver provides the low-level control of the power supply regulators, selection of voltage levels, and enabling/disabling of regulators. This device driver makes use of the PF100 regulator driver to access the PF100 hardware control registers. PF100 regulator driver is based on regulator core driver and it is attached to kernel I2C bus.

22.2 Hardware Operation

PF100 provides reference and supply voltages for the application processor and peripheral devices.

Four buck (step down) converters (up to 6 independent output) and one boost (step up) converter are included. The buck converters provide the power supply to processor cores and to other low voltage circuits such as memory. Dynamic voltage scaling is provided to allow controlled supply rail adjustments for the processor cores and/or other circuitry.

Linear regulators are directly supplied from the battery or from the switchers and include supplies for I/O and peripherals, audio, camera, BT, WLAN, and so on. Naming conventions are suggestive of typical or possible use case applications, but the switchers and regulators may be utilized for other system power requirements within the guidelines of specified capabilities.

The only power on event of PF100 is PWRON is high, and the only power off event of PF100 is PWRON is low. PMIC_ON_REQ pin of i.MX 6, which is controlled by SNVS block of i.MX 6, will connect with PWRON pin of PF100 to control PF100 on/off, so that system can power off.

22.2.1 Driver Features

PF100 regulator driver is based on regulator core driver. It provides the following services for regulator control of the PMIC component:

- Switch ON/OFF all voltage regulators.
- Set the value for all voltage regulators.
- Get the current value for all voltage regulators.

22.3 Software Operation

PF100 regulator client driver performs operations by reconfiguring the PMIC hardware control registers.

Some of the PMIC power management operations depend on the system design and configuration. For example, if the system is powered by a power source other than the PMIC, then turning off or adjusting the PMIC voltage regulators has no effect. Conversely, if the system is powered by the PMIC, then any changes that use the power management driver and the regulator client driver can affect the operation or stability of the entire system.

22.3.1 Regulator APIs

The regulator power architecture is designed to provide a generic interface to voltage and current regulators within the Linux kernel.

It is intended to provide voltage and current control to client or consumer drivers and to provide status information to user space applications through a sysfs interface. The intention is to allow systems to dynamically control regulator output to save power and prolong battery life. This applies to both voltage regulators (where voltage output is controllable) and current sinks (where current output is controllable).

For more details, visit opensource.wolfsonmicro.com/node/15

Under this framework, most power operations can be done by the following unified API calls:

- `regulator_get` is an unified API call to lookup and obtain a reference to a regulator:

```
struct regulator *regulator_get(struct device *dev, const char *id);
```

- `regulator_put` is an unified API call to free the regulator source:

```
void regulator_put(struct regulator *regulator, struct device *dev);
```

- `regulator_enable` is an unified API call to enable regulator output:

```
int regulator_enable(struct regulator *regulator);
```

- `regulator_disable` is an unified API call to disable regulator output:

```
int regulator_disable(struct regulator *regulator);
```

- `regulator_is_enabled` is the regulator output enabled:

```
int regulator_is_enabled(struct regulator *regulator);
```

- `regulator_set_voltage` is an unified API call to set regulator output voltage:

```
int regulator_set_voltage(struct regulator *regulator, int uV);
```

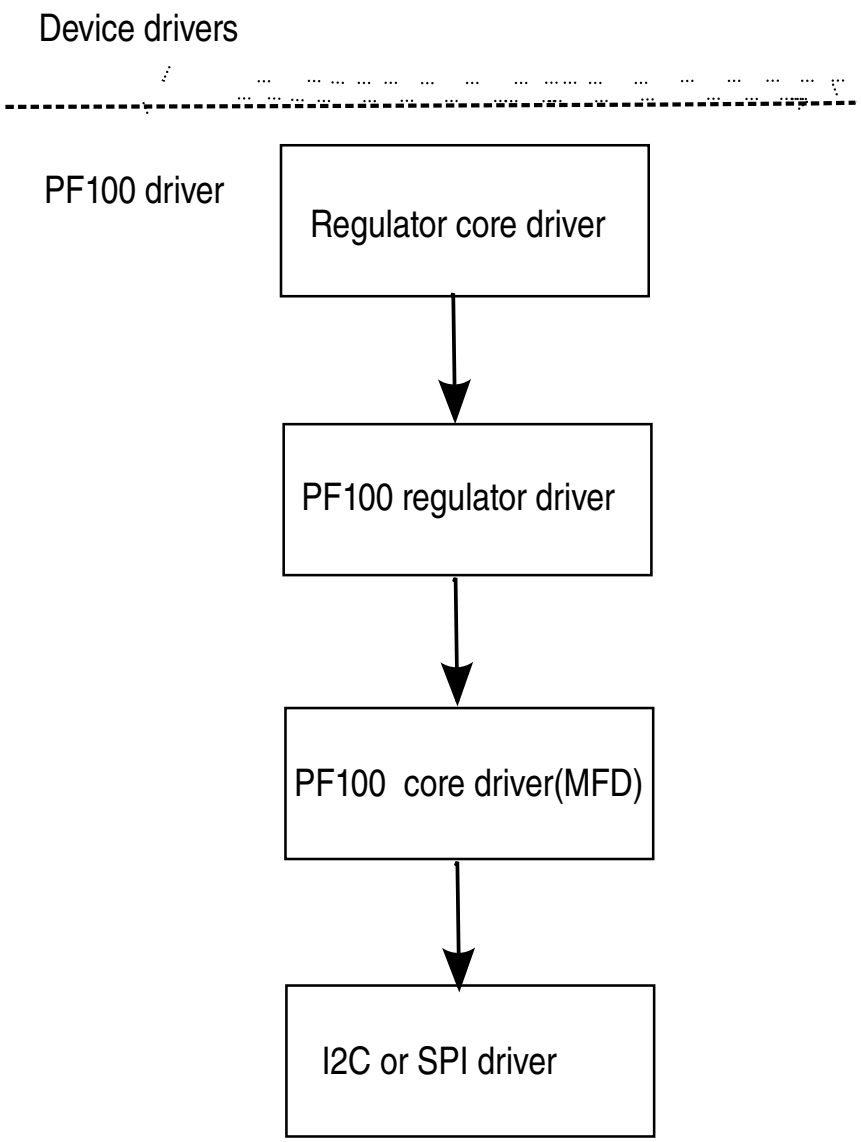
- `regulator_get_voltage` is an unified API call to get regulator output voltage:

```
int regulator_get_voltage(struct regulator *regulator);
```

You can find more APIs and details in the regulator core source code inside the Linux kernel at: `<Yocto_BuildDir>/linux/drivers/regulator/core.c`.

22.4 Driver Architecture

Figure below shows the basic architecture of the PF100 regulator driver.



22.4.1 Driver Interface Details

Access to PFUZE100 regulator is provided through the API of the regulator core driver.

PFUZE100 regulator driver provides the following regulator controls:

- 4 buck switch regulators on normal mode (up to 6 independent rails): SW1AB, SW1C, SW2, SW3A, SW3B, and SW4.
- Buck switch can be programmed to a state of standby with specific register (PFUZE100_SWxSTANDBY) in advance.
- 6 Linear Regulators: VGEN1, VGEN2, VGEN3, VGEN4, VGEN5, and VGEN6.
- 1 LDO/Switch supply for VSNVS support on i.MX processors.
- 1 Low current, high accuracy, voltage reference for DDR Memory reference voltage.
- 1 Boost regulator with USB OTG support.
- Most power rails from PFUZE100 have been programmed properly according to the hardware design. Therefore, you can't find the kernel using PFUZE100 regulators. PFUZE100 regulator driver has implemented these regulators so that customers can use it freely if default PFUZE100 value can't meet their hardware design.

22.4.2 Source Code Structure

The PFUZE100 regulator driver is located in the regulator device driver directory:

```
<Yocto_BuildDir>/linux/drivers/regulator
```

Table 22-1. PFUZE100 core Driver Files

File	Description
drivers/regulator/ pfuze100-regulator.c	Implementation of the PFUZE100 regulator client driver.

There is no board file related to pmic. Some code moves to U-Boot, such as standby voltage setting. Some code is implemented by DTS file. See pfuze100 device node in arch/arm/boot/dts/imx6qdl-sabresd.dtsi and arch/arm/boot/dts/imx6qdl-sabreauto.dtsi

There is no board file related to pmic. Some code moves to U-Boot, such as standby voltage setting. Some code is implemented by DTS file. See pfuze100 device node in arch/arm/boot/dts/imx6qdl-sabresd.dtsi and arch/arm/boot/dts/imx6qdl-sabreauto.dtsi

There is no board file related to pmic. Some code moves to U-Boot, such as standby voltage setting. Some code is implemented by DTS file. See pfuze100 device node in arch/arm/boot/dts/imx6qdl-sabresd.dtsi and arch/arm/boot/dts/imx6qdl-sabreauto.dtsi

22.4.3 Menu Configuration Options

The following are menu configuration options:

1. To get to the PMIC power configuration, use the command:

   ```
   bitbake linux-imx -c menuconfig
   ```
2. On the configuration screen select Configure Kernel, exit, and when the next screen appears, choose the following:
3. Device Drivers > Voltage and Current regulator support > Support regulators on Freescale PF100 PMIC.

Chapter 23
CPU Frequency Scaling (CPUFREQ) Driver

23.1 Introduction

The CPU frequency scaling device driver allows the clock speed of the CPU to be changed on the fly. Once the CPU frequency is changed, the voltage VDDCORE, VDDSOC and VDDPU are changed to the voltage value defined in device tree scripts（DTS）. This method can reduce power consumption (thus saving battery power), because the CPU uses less power as the clock speed is reduced.

23.1.1 Software Operation

The CPUFREQ device driver is designed to change the CPU frequency and voltage on the fly.

If the frequency is not defined in DTS, the CPUFREQ driver changes the CPU frequency to the nearest higher frequency in the array. The frequencies are manipulated using the clock framework API, while the voltage is set using the regulators API. The CPU frequencies in the array are based on the boot CPU frequency. Interactive CPU frequency governor is used which cannot be changed manually. To change CPU frequency manually, the userspace CPU frequency governor can be used.By default, the conservative CPU frequency governor is used.

Refer to the API document for more information on the functions implemented in the driver.

To view what values the CPU frequency can be changed to in KHz (The values in the first column are the frequency values) use this command:

```
cat /sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state
```

To change the CPU frequency to a value that is given by using the command above (for example, to 792 MHz) use this command:

```
echo 792000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

The frequency 792000 is in KHz, which is 792 MHz.

The maximum frequency can be checked using this command:

```
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq
```

Use the following command to view the current CPU frequency in KHz:

```
cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_cur_freq
```

Use the following command to view available governors:

```
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors
```

Use the following command to change to interactive CPU frequency governor:

```
echo interactive > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

23.1.2 Source Code Structure

Table below shows the source files and headers available in the following directory:

```
drivers/cpufreq/
```

Table 23-1. CPUFREQ Driver Files

File	Description
imx6q-cpufreq.c/ imx7-cpufreq.c	CPUFREQ functions

For CPU frequency working point settings, see:

- arch/arm/boot/dts/imx6q.dtsi for i.MX 6Quad and i.MX 6QuadPlus
- arch/arm/boot/dts/imx6dl.dtsi for i.MX 6DualLite
- arch/arm/boot/dts/imx6sl.dtsi for i.MX 6SoloLite
- arch/arm/boot/dts/imx6sx.dtsi for i.MX 6SoloX
- arch/arm/boot/dts/imx6ul.dtsi for i.MX 6UltraLite
- arch/arm/boot/dts/imx7d.dtsi for i.MX 7Dual

23.2 Menu Configuration Options

The following Linux kernel configuration is provided for this module:

- CONFIG_CPU_FREQ; In menuconfig, this option is located under:
  - CPU Power Management > CPU Frequency scaling
- The following options can be selected:
  - CPU Frequency scaling
  - CPU frequency translation statistics
  - Default CPU frequency governor (conservative)(interactive)
  - Performance governor
  - Powersave governor
  - Userspace governor for userspace frequency scaling
  - Interactive CPU frequency policy governor
  - Conservative CPU frequency governor
  - CPU frequency driver for i.MX CPUs

23.2.1 Board Configuration Options

There are no board configuration options for the CPUFREQ device driver.

Chapter 24
Dynamic Bus Frequency Driver

24.1 Introduction

In order to improve power consumption, the Bus Frequency driver dynamically manages the various system frequencies.

The frequency changes are transparent to the higher layers and require no intervention from the drivers or middleware. Depending on activity of the peripheral devices and CPU loading, the bus frequency driver varies the DDR frequency between 24 MHz and its maximum frequency. Similarly the AHB frequency is varied between 24 MHz and 132 MHz.

24.1.1 Operation

The Bus Frequency driver is part of the power management module in the Linux BSP. The main purpose of this driver is to scale the various operating frequency of the system clocks (like AHB, DDR, AXI etc.) based on peripheral activity and CPU loading.

24.1.2 Software Operation

The bus frequency depends on the request and release of device drivers for its operation. Drivers will call bus frequency APIs to request or release the bus setpoint they want. The bus frequency will set the system frequency to highest frequency setpoint based on the peripherals that are currently requesting.

If ARM Cortex-M4 processor is alive with ARM Cortex-A9 processor together, ARM Cortex-M4 processor will also request/release bus frequency high setpoint for its operation. This means that ARM Cortex-A9 processor treats ARM Cortex-M4 processor as one of its high-speed devices.

The following setpoints are defined for all i.MX 6 platforms:

1. High Frequency Setpoint: AHB is at 132 MHz, AXI is at 264 Mhz and DDR is at the maximum frequency. This mode is used when most peripehrals that need higher frequency for good performance are active. For example, video playback and graphics processing.
2. Audio Playback setpoints: AHB is at 25 MHz, AXI is at 50 MHz and DDR is at 50 MHz for i.MX 6Quad/6DualLite/6SoloX and 100 MHz for i.MX 6SoloLite. This mode is used in audio playback mode.
3. Low Frequency setpoint: AHB is at 24 MHz, AXI is at 24 MHz and DDR is at 24 MHz. This mode is used when the system is idle waiting for user input (display is off).

To Enable the bus frequency driver use the following command:

```
echo 1 > /sys/bus/platform/drivers/imx6_busfreq/busfreq.13/enable
```

To Disable the bus frequency driver use the following command:

```
echo 0 > /sys/bus/platform/drivers/imx6_busfreq/busfreq.13/enable
```

24.1.3 Source Code Structure

Table below lists the source files and headers available in the following directory:

```
arch/arm/mach-imx
```

Table 24-1. BusFrequency Driver Files

File	Description
busfreq-imx.c	Bus Frequency functions
busfreq_ddr3.c, busfreq_lpddr2.c, ddr3_freq_imx6.S, lpddr2_freq_imx6.S, ddr3_freq_imx6sx.S, ddr3_freq_imx6sx.S, ddr3_freq_imx7d.S	DDR frequency change functions

24.2 Menu Configuration Options

There are no menu configuration options for this driver. The Bus Frequency drivers is included and enabled by default.

24.2.1 Board Configuration Options

There are no board configuration options for the Linux BusFreq device driver.

Chapter 25
Thermal Driver

25.1 Introduction

Thermal driver is a necessary driver for monitoring and protecting the SoC. The thermal driver will monitor the SoC temperature in a certain frequency.

It defines two trip points: critical and passive. Cooling device will take actions to protect the SoC according to the different trip points that SoC has reached:

- When reaching critical point, cooling device will shut down the system.
- When reaching passive point, cooling device will lower CPU frequency and notify GPU to run at a lower frequency.
- When the temperature drops to 10 °C below passive point, cooling device will release all the cooling actions.

Thermal driver has two parts:

- Thermal zone defines trip points and monitors the SoC's temperature.
- Cooling device takes the actions according to the different trip points.

25.1.1 Thermal Driver Overview

The thermal driver implements the SoC temperature monitor function and protection. It creates a sys file interface of /sys/class/thermal/thermal_zone0/ for user. Internally, the thermal driver will monitor the SoC temperature and do necessary protection according to the different trip points that SoC's temperature reaches.

25.2 Hardware Operation

The thermal driver uses internal thermal sensor to monitor the SoC temperature. The cooling device uses the CPU frequency to protect the SoC.

All the related modules are in SoC.

25.2.1 Thermal Driver Software Operation

The thermal driver registers a thermal zone and a cooling device. A structure,`thermal_zone_device_ops`, describes the necessary interface that the thermal framework needs. The framework will call the related thermal zone interface to monitor the SoC temperature and do the cooling protection.

25.3 Driver Features

The thermal driver supports the features found here.

- Thermal monitors the SoC temperature.
- Cooling device protects the SoC when the temperature reaches passive or critical points.

25.3.1 Source Code Structure

Table below shows the driver source files available in the directory:

<Yocto_BuildDir>/linux/drivers/thermal

Table 25-1. Thermal Driver Files

File	Description
imx_thermal.c, device_cooling.c	thermal zone driver source file

25.3.2 Menu Configuration Options

The following Linux kernel configuration option is provided for this module. To get to this option, use the bitbake linux-imx -c menuconfigcommand. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

Device Drivers Generic Thermal sysfs driver > Temperature sensor driver for Freescale i.MX SoCs.

25.3.3 Programming Interface

The thermal driver can be accessed via /sys/bus/platform/drivers/imx_thermal/.

25.4 Unit Test

Modify the trip point's temperature through /sys/class/thermal/thermal_zone0/trip_point_x_temp. Here 'x' can be 0 and 1, indicating critical and passive trip point, the value of trip points should be critical > passive. Then run some program to make SoC in heavy loading, when the SoC temperature reach the trip points, the thermal driver will take action to do some protections according to each trip point's mechanism. Restore the trip point's temperature, when SoC temperature drop to 10 °C below passive, thermal driver will remove all the protections.

Chapter 26
Anatop Regulator Driver

26.1 Introduction

The Anatop regulator driver provides the low-level control of the power supply regulators, and selection of voltage levels.

This device driver makes use of the regulator core driver to access the Anatop hardware control registers.

26.1.1 Hardware Operation

The Power Management Unit on the die is built to simplify the external power interface and allow the die to be configured in a power appropriate manner. The power system consists of the input power sources and their characteristics, the integrated power transforming and controlling elements, and the final load interconnection and requirements.

Utilizing 7 LDO regulators, the number of external supplies is greatly reduced. If the backup coin and USB inputs are neglected, then the number of external supplies is reduced to two. Missing from this external supply total are the necessary external supplies to power the desired memory interface. This will change depending on the type of external memory selected. Other supplies might also be necessary to supply the voltage to the different I/O power segments if their I/O voltage needs to be different than what is provided above.

Some internal regulator can be bypassed , so that external pmic can supply these power directly to decrease power numer. such as VDD_SOC, VDD_ARM

26.2 Driver Features

The Anatop regulator driver is based on regulator core driver. A list of services provided for regulator control can be found here.

- Switch ON/OFF all voltage regulators.
- Set the value for all voltage regulators.
- Get the current value for all voltage regulators.

26.2.1 Software Operation

The Anatop regulator client driver performs operations by reconfiguring the Anatop hardware control registers. This is done by calling regulator core APIs with the required register settings.

26.2.2 Regulator APIs

The regulator power architecture is designed to provide a generic interface to voltage and current regulators within the Linux kernel. It is intended to provide voltage and current control to client or consumer drivers and also provide status information to user space applications through a sysfs interface. The intention is to allow systems to dynamically control regulator output to save power and prolong battery life. This applies to both voltage regulators (where voltage output is controllable) and current sinks (where current output is controllable).

For more details visit opensource.wolfsonmicro.com/node/15

Under this framework, most power operations can be done by the following unified API calls:

- `regulator_get` used to lookup and obtain a reference to a regulator:
  - `struct regulator *regulator_get(struct device *dev, const char *id);`
- `regulator_put` used to free the regulator source:
  - `void regulator_put(struct regulator *regulator, struct device *dev);`
- `regulator_enable` used to enable regulator output:
  - `int regulator_enable(struct regulator *regulator);`
- `regulator_disable` used to disable regulator output:
  - `int regulator_disable(struct regulator *regulator);`
- `regulator_is_enabled` is the regulator output enabled:
  - `int regulator_is_enabled(struct regulator *regulator);`
- `regulator_set_voltage` used to set regulator output voltage:

    - `int regulator_set_voltage(struct regulator *regulator, int uV);`

- `regulator_get_voltage` used to get regulator output voltage:
    - `int regulator_get_voltage(struct regulator *regulator);`

For more APIs and details in the regulator core source code inside the Linux kernel see: <Yocto_BuildDir>/linux/drivers/regulator/core.c.

26.2.3 Driver Interface Details

Access to the Anatop regulator is provided through the API of the regulator core driver. The Anatop regulator driver provides the following regulator controls:

- Seven LDO regulators
- All of the regulator functions are handled by setting the appropriate Anatop hardware register values. This is done by calling the regulator core APIs to access the Anatop hardware registers.

26.2.4 Source Code Structure

The Anatop regulator driver is located in the regulator device driver directory:

```
<Yocto_BuildDir>/linux/drivers/regulator
```

Table 26-1. Anatop Power Management Driver Files

File	Description
`core.c`	Linux kernel interface for regulators.
`anatop-regulator.c`	Implementation of the Anatop regulator client driver

The Anatop regulators are registered in each SoC-specific dts file. For example, on the i.MX 6Quad/6DualLite/6Solo, the DTS file is arch/arm/boot/dts/imx6qdl.dtsi.

26.2.5 Menu Configuration Options

To get to the Anatop regulator configuration, use the commandbitbake linux-imx -c menuconfig. On the configuration screen select Configure Kernel, exit, and when the next screen appears, choose. The following Linux kernel configurations are provided for the Anatop Regulator driver:

- Device Drivers > Voltage and Current regulator support > Anatop Regulator Support.
- System Type > Freescale MXC Implementations > Internal LDO in i.MX 6Quad/ i.MX 6DualLite bypass.

Chapter 27
SNVS Real Time Clock (SRTC) Driver

27.1 Introduction

The SNVS Real Time Clock (SRTC) module is used to keep the time and date. It provides a certifiable time to the user and can raise an alarm if tampering with counters is detected. The SRTC is composed of two sub-modules: Low power domain (LP) and High power domain (HP). The SRTC driver only supports the LP domain with low security mode.

27.1.1 Hardware Operation

The SRTC is a real time clock with enhanced security capabilities.

It provides an accurate, constant time, regardless of the main system power state and without the need to use an external on-board time source, such as an external RTC. The SRTC can wake up the system when a pre-set alarm is reached.

27.2 Software Operation

The following sections describe the software operation of the SRTC driver.

27.2.1 IOCTL

The SRTC driver complies with the Linux RTC driver model. See the Linux documentation in <Yocto_BuildDir>/linux/Documentation/rtc.txt for information on the RTC API.

Besides the initialization function, the SRTC driver provides IOCTL functions to set up the RTC timers and alarm functions. The following RTC IOCTLs are implemented by the SRTC driver:

- RTC_RD_TIME
- RTC_SET_TIME
- RTC_AIE_ON
- RTC_AIE_OFF
- RTC_ALM_READ
- RTC_ALM_SET

The driver information can be access by the proc file system. For example:

```
root@freescale /unit_tests$ cat /proc/driver/rtc
rtc_time        : 12:48:29
rtc_date        : 2009-08-07
alrm_time       : 14:41:16
alrm_date       : 1970-01-13
alarm_IRQ       : no
alrm_pending    : no
24hr            : yes
```

27.2.2 Keep Alive in the Power Off State

To preserve the time when the device is in the power off state, the SRTC clock source should be set to CKIL and the voltage input, NVCC_SRTC_POW, should remain active. Usually these signals are connected to the PMIC and software can configure the PMIC registers to enable the SRTC clock source and power supply.

Ordinarily, when the main battery is removed and the device is in power off state, a coin-cell battery is used as a backup power supply. To avoid SRTC time loss, the voltage of the coin-cell battery should be sufficient to power the SRTC. If the coin-cell battery is chargeable, it is recommend to automatically enable the coin-cell charger so that the SRTC is properly powered.

27.3 Driver Features

The SRTC driver includes the following features:

- Implements all the functions required by Linux OS to provide the real time clock and alarm interrupt
- Reserves time in power off state
- Alarm wakes up the system from low power modes

27.3.1 Source Code Structure

The RTC module is implemented in the following directory:

```
<Yocto_BuildDir>/linux/drivers/rtc
```

Table below shows the RTC module files.

Table 27-1. RTC Driver Files

File	Description
rtc-snvs.c	SNVS RTC driver implementation file

The source file for the SRTC specifies the SRTC function implementations.

27.3.2 Menu Configuration Options

To get to the SRTC driver, use the command bitbake linux-imx -c menuconfig. On the screen displayed, select **Configure the kernel** and exit. When the next screen appears select the following options to enable the SRTC driver:

- Device Drivers > Real Time Clock > Freescale SNVS Real Time Clock

Chapter 28
Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver

28.1 ALSA Sound Driver Introduction

The Advanced Linux Sound Architecture (ALSA), now the most popular architecture in Linux system, provides audio and MIDI functionality to the Linux operating system.

ALSA has the following significant features:

- Efficient support for all types of audio interfaces, from consumer sound cards to professional multichannel audio interfaces.
- Fully modularized sound drivers.
- SMP and thread-safe design.
- User space library (alsa-lib) to simplify application programming and provide higher level functionality.
- Support for the older Open Sound System (OSS) API, providing binary compatibility for most OSS programs

ALSA System on Chip (ASoC) layer is designed for SoC audio. The overall project goal of the ASoC layer provides better ALSA support for embedded system on chip processors and portable audio CODECs.

The ASoC layer also provides the following features:

- CODEC independence. Allows reuse of CODEC drivers on other platforms and machines.
- Easy I2S/PCM audio interface setup between CODEC and SoC. Each SoC interface and CODEC registers its audio interface capabilities with the core.
- Dynamic Audio Power Management (DAPM). DAPM is an ASoC technology designed to minimize audio subsystem power consumption no matter what audio-use case is active. DAPM guarantees the lowest audio power state at all times and is completely transparent to user space audio components. DAPM is ideal for mobile devices or devices with complex audio requirements.

- Pop and click reduction. Pops and clicks can be reduced by powering the CODEC up/down in the correct sequence (including using digital mute). ASoC signals the CODEC when to change power states.
- Machine-specific controls. Allow machines to add controls to the sound card, for example, volume control for speaker amp.

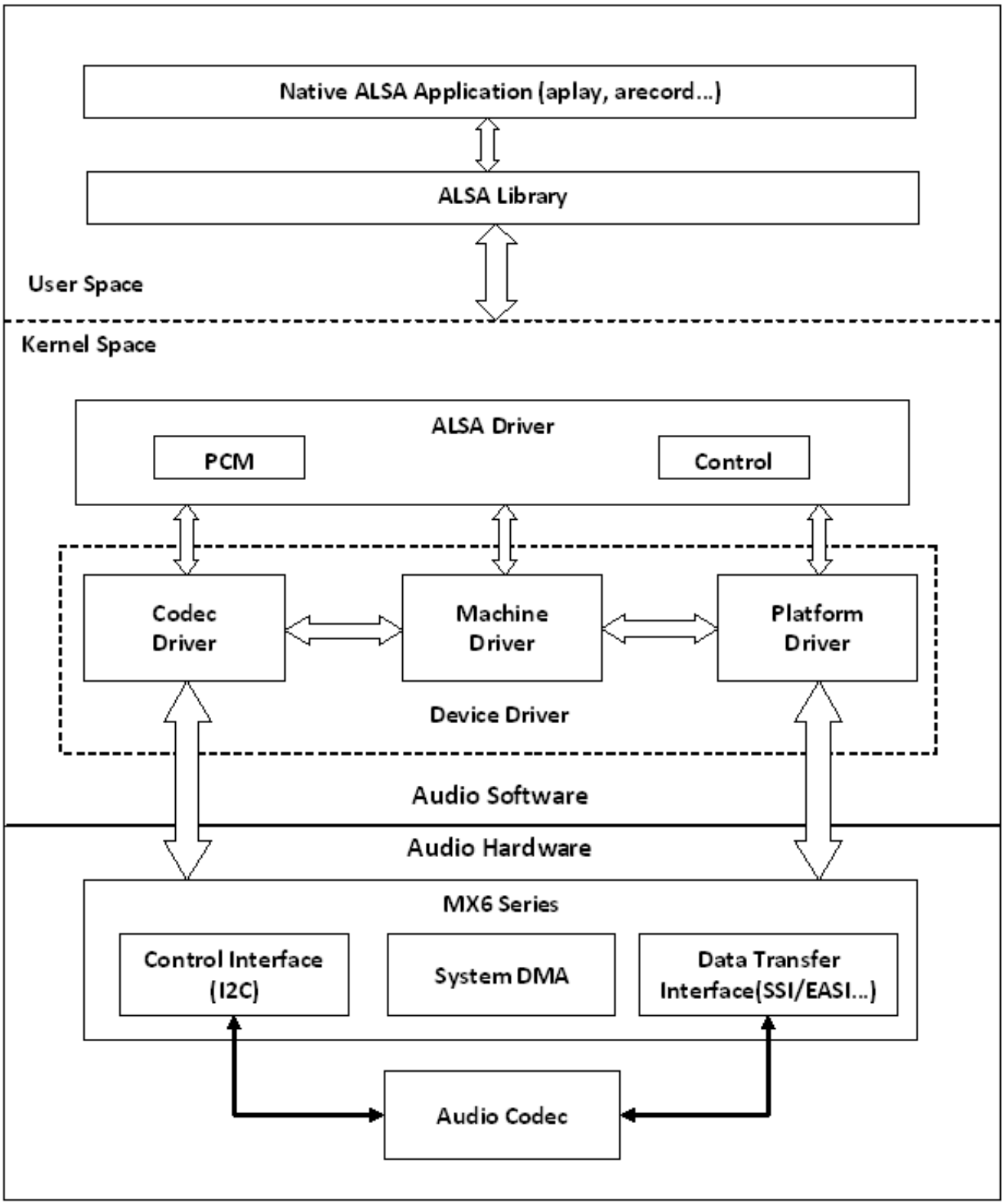


Figure 28-1. ALSA SoC Software Architecture

ASoC basically splits an embedded audio system into 3 components:

- Machine driver-handles any machine-specific controls and audio events, such as turning on an external amp at the beginning of playback.
- Platform driver-contains the audio DMA engine and audio interface drivers (for example, I^2S, AC97, PCM) for that platform.
- CODEC driver-platform independent and contains audio controls, audio interface capabilities, the CODEC DAPM definition, and CODEC I/O functions.

More detailed information about ASoC can be found in the Linux kernel documentation in the Linux OS source tree at linux/Documentation/sound/alsa/soc and at www.alsa-project.org/main/index.php/ASoC.

28.2 SoC Sound Card

Currently, the stereo CODEC (wm8962), 7.1 CODEC (cs42888), and AM/FM CODEC (si4763) drivers are implemented using ASoC architecture.

These sound card drivers are built in independently. The stereo sound card supports stereo playback and capture. The 7.1 sound card supports up to eight channels of audio playback. While enabling ASRC, 7.1 sound card only supports 2 or 6 channels audio playback. The AM/FM sound card supports radio PCM capture.

NOTE

The 7.1 CODEC is only supported on the i.MX 6Quad and i.MX 6Solo SABRE Auto platform.

The AM/FM CODEC is only supported on the i.MX 6Quad and i.MX 6Solo SABRE Auto platform.

28.2.1 Stereo CODEC Features

The stereo CODEC supports the following features:

- Sample rates for playback and capture are 8 KHz, 32 KHz, 44.1 KHz, 48 KHz, and 96 KHz
- Channels:
  - Playback: supports two channels.
  - Capture: supports two channels.
- Audio formats:
  - Playback:
    - SNDRV_PCM_FMTBIT_S16_LE

        - SNDRV_PCM_FMTBIT_S20_3LE
        - SNDRV_PCM_FMTBIT_S24_LE
    - Capture:
        - SNDRV_PCM_FMTBIT_S16_LE
        - SNDRV_PCM_FMTBIT_S20_3LE
        - SNDRV_PCM_FMTBIT_S24_LE

28.2.2 7.1 Audio Codec Features

- Sample rates for playback and record:
    - 48 KHz, 96 KHz, 192 KHz
    - Playback: 5.512 k, 8 k, 11.025 k, 16 k, 22 k, 32 k, 44.1 k, 48 k, 64 k, 88.2 k, 96 k, 176.4 k, 192 k (ASRC enabled)
- Channels:
    - Playback: 2, 4, 6, 8 channels
    - Playback(ASRC enabled): 2, 6 channels
    - Capture: 2, 4 channels
- Audio formats:
    - Playback:
        - SNDRV_PCM_FMTBIT_S16_LE
        - SNDRV_PCM_FMTBIT_S20_3LE
        - SNDRV_PCM_FMTBIT_S24_LE
    - Playback(ASRC enabled):
        - SNDRV_PCM_FMTBIT_S16_LE
        - SNDRV_PCM_FMTBIT_S24_LE
    - Capture:
        - SNDRV_PCM_FMTBIT_S16_LE
        - SNDRV_PCM_FMTBIT_S20_3LE
        - SNDRV_PCM_FMTBIT_S24_LE

28.2.3 AM/FM Codec Features

- Supported sample rate for Capture: 48 KHz
- Supported channels:
    - Capture: supports two channels.
- Supported audio formats:
    - Capture: SNDRV_PCM_FMTBIT_S16_LE

28.2.4 Sound Card Information

The registered sound card information can be listed as follows using the commands aplay -l and arecord -l. For example, the stereo sound card is registered as card 0.

```
root@freescale /$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: wm8962audio [wm8962-audio], device 0: HiFi wm8962-0 []
 Subdevices: 1/1
 Subdevice #0: subdevice #0
```

28.3 Hardware Operation

The following sections describe the hardware operation of the ASoC driver.

28.3.1 Stereo Audio CODEC

The stereo audio CODEC is controlled by the I^2C interface. The audio data is transferred from the user data buffer to/from the SSI FIFO through the DMA channel. The DMA channel is selected according to the audio sample bits. AUDMUX is used to set up the path between the SSI port and the output port which connects with the CODEC. The CODEC works in master mode and provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate.

The WM8962 ASoC CODEC driver exports the audio record/playback/mixer APIs according to the ASoC architecture.

The CODEC driver is generic and hardware independent code that configures the CODEC to provide audio capture and playback. It does not contain code that is specific to the target platform or machine. The CODEC driver handles:

- CODEC DAI and PCM configuration
- CODEC control I/O-using I^2C
- Mixers and audio controls
- CODEC audio operations
- DAC Digital mute control

The WM8962 CODEC is registered as an I^2C client when the module initializes. The APIs are exported to the upper layer by the structure snd_soc_dai_ops .

Headphone insertion/removal can be detected through a GPIO interrupt signal.

SSI dual FIFO features are enabled by default.

28.3.2 7.1 Audio Codec

The 7.1 audio codec includes 8-channel DAC and 4-channel ADC, which are controlled by the I2C interface. The audio data is transferred from the user data buffer to the ESAI fifo, through a DMA channel. The DMA channel is selected according to audio sample bits. The codec works in slave mode as the esai provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate. The ESAI supports up to eight audio output ports. While enabling ASRC, 7.1 audio codec supports 2 or 6 channel playback through ASRC. On the i.MX 6 Sabre ARD board, a cs42888 codec with 4 audio in port is used, each port receive two channels of data in the I2S format(network mode), providing 8-channel of playback functionality. This codec also has 2 audio output port connected with ESAI, providing 4-channel of recording functionality.

The codec driver is generic and hardware independent code that configures the codec to provide audio capture and playback. It does not contain code that is specific to the target platform or machine. The codec driver handles:

- Codec DAI and PCM configuration
- Codec control I/O-using I2C
- Mixers and audio controls
- Codec audio operations
- DAI Digital mute control

The CS42888 codec is registered as an I2C client when the module initializes. The APIs are exported to the upper layer by the structure snd_soc_dai_ops.

28.3.3 AM/FM Codec

The AM/FM codec is a virtual codec, it only has a PCM interface connected to the Tuner device. The audio data is transferred from the user data buffer to or from the SSI FIFO through the DMA channel. The DMA channel is selected according to the audio sample bits. AUDMUX is used to set up the path between the SSI port and the output port which connects with the codec. The codec works in master mode as it provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate.

28.4 Software Operation

The following sections describe the software operation of the ASoC driver.

28.4.1 ASoC Driver Source Architecture

File imx-pcm-dma.c is shared by the stereo ALSA SoC driver, the 7.1 ALSA SoC driver and other CODEC driver. This file is responsible for preallocating DMA buffers and managing DMA channels.

The stereo CODEC is connected to the CPU through the SSI interface. fsl_ssi.c registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip SSI interface. wm8962.c registers the stereo CODEC and hifi DAI drivers. The direct hardware operations on the stereo codec are in wm8962.c. imx-wm8962.c is the machine layer code which creates the driver device and registers the stereo sound card.

The multi-channel codec is connected to the CPU through the ESAI interface. fsl_esai.c registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip ESAI interface. cs42888.c registers the multi-channel CODEC and hifi DAI drivers. The direct hardware operations on the multi-channel CODEC are in cs42888.c. imx-cs42888.c is the machine layer code which creates the driver device and registers the stereo sound card.

The AM/FM CODEC is connected to the CPU through the SSI interface. fsl_ssi.c registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip SSI interface. si476x.c registers the Tuner CODEC and Tuner DAI drivers. The direct hardware operations on the CODEC are in si476x.c. imx-si476x.c is the machine layer code which creates the driver device and registers the sound card.

Table below shows the stereo codec SoC driver source files. These files are under the <Yocto_BuildDir>/linux/sound/soc directory.

Table 28-1. Stereo Codec SoC Driver Files

File	Description
fsl/imx-wm8962.c	Machine layer for stereo CODEC ALSA SoC (CODEC as I2S Master)
fsl/imx-pcm-dma.c	Platform layer for stereo CODEC ALSA SoC
fsl/imx-pcm.h	Header file for PCM driver and AUDMUX register definitions
fsl/fsl_ssi.c	SSI CPU DAI driver for stereo CODEC ALSA SoC
fsl/fsl_ssi.h	Header file for SSI CPU DAI driver and SSI register definitions
codecs/wm8962.c	CODEC layer for stereo CODEC ALSA SoC
codecs/wm8962.h	Header file for stereo CODEC driver

Table below lists the AM/FM codec SoC driver source files. These files are under the <Yocto_BuildDir>/linux/sound/soc directory.

Table 28-2. AM/FM Codec SoC Driver Source Files

File	Description
fsl/imx-si476x.c	Machine layer for stereo CODEC ALSA SoC (CODEC as I2S Slave)
fsl/imx-pcm-dma.c	Platform layer for stereo CODEC ALSA SoC
fsl/imx-pcm.h	Header file for pcm driver and AUDMUX register definitions
fsl/fsl_ssi.c	SSI CPU DAI driver for stereo CODEC ALSA SoC
fsl/fsl_ssi.h	Header file for SSI CPU DAI driver and SSI register definitions
codecs/si476x.c	Codec layer for stereo CODEC ALSA SoC
fsl/fsl_sai.c	SAI CPU DAI driver for stereo CODEC ALSA SoC
fsl/fsl_ssi.h	Header file for the SAI CPU DAI driver and SAI register definitions

Table below shows the multiple-channel ADC SoC driver source files. These files are also under the <Yocto_BuildDir>/linux/sound/soc directory.

Table 28-3. CS42888 ASoC Driver Source File

File	Description
fsl/imx-cs42888.c	Machine layer for mutliple-channel CODEC ALSA SoC
fsl/imx-pcm-dma.c	Platform layer for mutliple-channel CODEC ALSA SoC
fsl/imx-pcm.h	Header file for pcm driver
fsl/fsl_esai.c	ESAI CPU DAI driver for mutliple-channel CODEC ALSA SoC
fsl/fsl_esai.h	Header file for ESAI CPU DAI driver
codecs/cs42xx8.c	CODEC layer for mutliple-channel codec ALSA SoC
codecs/cs42xx8.h	Header file for mutliple-channel CODEC driver
fsl/fsl_asrc.c	CPU DAI driver of ASRC P2P
fsl/fsl_asrc.h	Header file for CPU DAI driver of ASRC P2P
fsl/fsl_asrc_pcm.c	Platform layer for ASRC P2P

28.4.2 Sound Card Registration

The codecs have the same registration sequence:

1. The codec driver registers the codec driver, DAI driver, and their operation functions.
2. The platform driver registers the PCM driver, CPU DAI driver and their operation functions, pre allocates buffers for PCM components and sets playback and capture operations as applicable.
3. The machine layer creates the DAI link between codec and CPU registers the sound card and PCM devices.

28.4.3 Device Open

The ALSA driver performs the following functions:

- Allocates a free substream for the operation to be performed.
- Opens the low level hardware device.
- Assigns the hardware capabilities to ALSA runtime information (the runtime structure contains all the hardware, DMA, and software capabilities of an opened substream).
- Configures DMA read or write channel for operation.
- Configures CPU DAI and CODEC DAI interface.
- Configures CODEC hardware.
- Triggers the transfer.

After triggering for the first time, the subsequent DMA read/write operations are configured by the DMA callback.

28.4.4 Devicetree Binding

See the following documents:

- Documentation/devicetree/bindings/powerpc/fsl/ssi.txt
- Documentation/devicetree/bindings/sound/fsl-sai.txt
- Documentation/devicetree/bindings/sound/fsl-easi.txt
- Documentation/devicetree/bindings/sound/fsl-asrc-p2p.txt
- Documentation/devicetree/bindings/sound/wm8962.txt
- Documentation/devicetree/bindings/sound/cs42xx8.txt
- Documentation/devicetree/bindings/sound/imx-audmux.txt
- Documentation/devicetree/bindings/sound/imx-audio-wm8962.txt
- Documentation/devicetree/bindings/sound/imx-audio-cs42888.txt
- Documentation/devicetree/bindings/sound/imx-audio-si476x.txt

28.4.5 Menu Configuration Options

The following Linux kernel configuration options are provided for this module.

- SoC Audio supports for wm8962 CODEC. In menuconfig, this option is available:

```
-> Device Drivers
     -> Sound card support
          -> Advanced Linux Sound Architecture
```

```
                -> ALSA for SoC audio support
                     -> SoC Audio for Freescale i.MX CPUs
                          -> SoC Audio support for i.MX boards with wm8962
```

- SoC Audio supports for i.MX cs42888. In menuconfig, this option is available:

```
-> Device Drivers
     -> Sound card support
          -> Advanced Linux Sound Architecture
               -> ALSA for SoC audio support
                  -> SoC Audio for Freescale i.MX CPUs
                         -> SoC Audio support for i.MX boards with cs42888
```

- SoC Audio supports for AM/FM. In menuconfig, this option is available:

```
-> Device Drivers
     -> Sound card support
          -> Advanced Linux Sound Architecture
               -> ALSA for SoC audio support
                    -> SoC Audio for Freescale i.MX CPUs
                       -> SoC Audio support for i.MX boards with si476x
```

28.5 Unit Test

This section descrbes how to use the ALSA driver.

28.5.1 Stereo CODEC Unit Test

Stereo CODEC driver supports playback and record features. There are a default volume, and you may adjust volume by alsamixer command.

Playback feature may be tested by the following command:

- aplay [-Dplughw:0,0] audio.wav

Record feature supports analog micphone and digital micphone. The default is digital micphone if analog micphone isn't plug-in.

Because analog micphone is connected to IN3R port of WM8962 CODEC, the following amixer commands are needed to input into command line for enabling analog micphone.

- amixer sset 'MIXINR IN3R' on
- amixer sset 'INPGAR IN3R' on

Record feature may be tested by the following command:

- arecord [-Dplughw:0,0] -r 44100 -f S16_LE -c 2 -d 5 record.wav

More usage for aplay/arecord/amixer may be gotten by the following commands.

- aplay --h
- arecord --h
- amixer --h

28.5.2 7.1 Audio Codec Unit Test

The 7.1 Audio codec driver support multi-channel playback and record feature. The codec has a default volume, and you can adjust volume by alsamixer command.

Playback feature can be tested by following command:

- aplay [-Dplughw:0,0] audio.wav

While enabling ASRC, the 7.1 audio codec should use the device 1 for playback. The codec has a default volume, and you can adjust volume by alsamixer command.

- aplay [-Dplughw:0,1] audio.wav

Record feature supports line in and mic in simultaneously. While on i.MX 6 Sabre ARD board, LINE-IN (L/R) use AIN1/AIN2, MICS1/MICS2 use AIN3/AIN4. By default, 2-ch record uses AIN1/AIN2, 4-ch record uses AIN1/AIN2/AIN3/AIN4 together.

Record feature can be tested by following command:

- arecord [-Dplughw:0,0] -r 48000 -f S16_LE -c 2 -d 5 record.wav

Note:The default ALSA config file, asound.conf located under /etc/, only supports stereo playback and record, which means, if you want to test 4,6,8-ch playback or 4-ch recording, using aplay audio.wav or arecord -c 4 audio.wav(without -Dplughw), you will have to make slight changes to the configure file as following:

- a) make sure playback pcm use dmix_48000 and capture pcm use dsnoop_48000 under pcm.asymed{};
- b) add "channels x" to the end of struct pcm.dmix_48000{} if you want to playback x-ch wav file(x is greater than 2);
- c) add "channels x" to the end of struct pcm.!dsnoop_48000{} if you want to record to x-ch wav(x is greater than 2);

If plug plughw is used to make a playback or record, examples as below,

- aplay -Dplughw:0,0 audio.wav or
- arecord -Dplughw:0,0 -c 4 -r 48000 -f S16_LE record.wav

You are not required to change asound.conf because this configure file is not used here.

More usage for aplay/arecord/amixer may be gotten by the following commands.

- aplay --h
- arecord --h
- amixer --h

28.5.3 AM/FM Codec Unit Test

This test turns on the AM/FM radio tuner (SI476x). It also sets and gets the current station.

NOTE: An underrun error may occur sometimes.

This underrun behaviour is normal, since the test connects the AM/FM output to the audio codec by a simple pipe.

There is not sync method between them. Upper layers (such as gstreamer plugins) should take care of this sync.

Input the following command in command line to start unit test:

- ./mxc_tuner_test.sh

The following infomation will be output to console window

Welcome to radio menu.

1. Turn on the radio

2. Get current frequency

3. Set current frequency

4. Turn off the radio

9. Exit.

- To turn on the radio select option 1
- To get the current frequency select option 2
- To set the desire frecuency select option 3 <enter> set the frequency <9740>
- To turn off the radio select option 4
- To Exit select option 9

Chapter 29
Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver for i.MX 6SoloLite

29.1 ALSA Sound Driver Introduction

The Advanced Linux Sound Architecture (ALSA), now the most popular architecture in Linux system, provides audio and MIDI functionality to the Linux operating system.

ALSA has the following significant features:

- Efficient support for all types of audio interfaces, from consumer sound cards to professional multichannel audio interfaces.
- Fully modularized sound drivers.
- SMP and thread-safe design.
- User space library (alsa-lib) to simplify application programming and provide higher level functionality.
- Support for the older Open Sound System (OSS) API, providing binary compatibility for most OSS programs

ALSA System on Chip (ASoC) layer is designed for SoC audio. The overall project goal of the ASoC layer provides better ALSA support for embedded system on chip processors and portable audio CODECs.

The ASoC layer also provides the following features:

- CODEC independence. Allows reuse of CODEC drivers on other platforms and machines.
- Easy I2S/PCM audio interface setup between CODEC and SoC. Each SoC interface and CODEC registers its audio interface capabilities with the core.
- Dynamic Audio Power Management (DAPM). DAPM is an ASoC technology designed to minimize audio subsystem power consumption no matter what audio-use case is active. DAPM guarantees the lowest audio power state at all times and is completely transparent to user space audio components. DAPM is ideal for mobile devices or devices with complex audio requirements.

- Pop and click reduction. Pops and clicks can be reduced by powering the CODEC up/down in the correct sequence (including using digital mute). ASoC signals the CODEC when to change power states.
- Machine-specific controls. Allow machines to add controls to the sound card, for example, volume control for speaker amp.

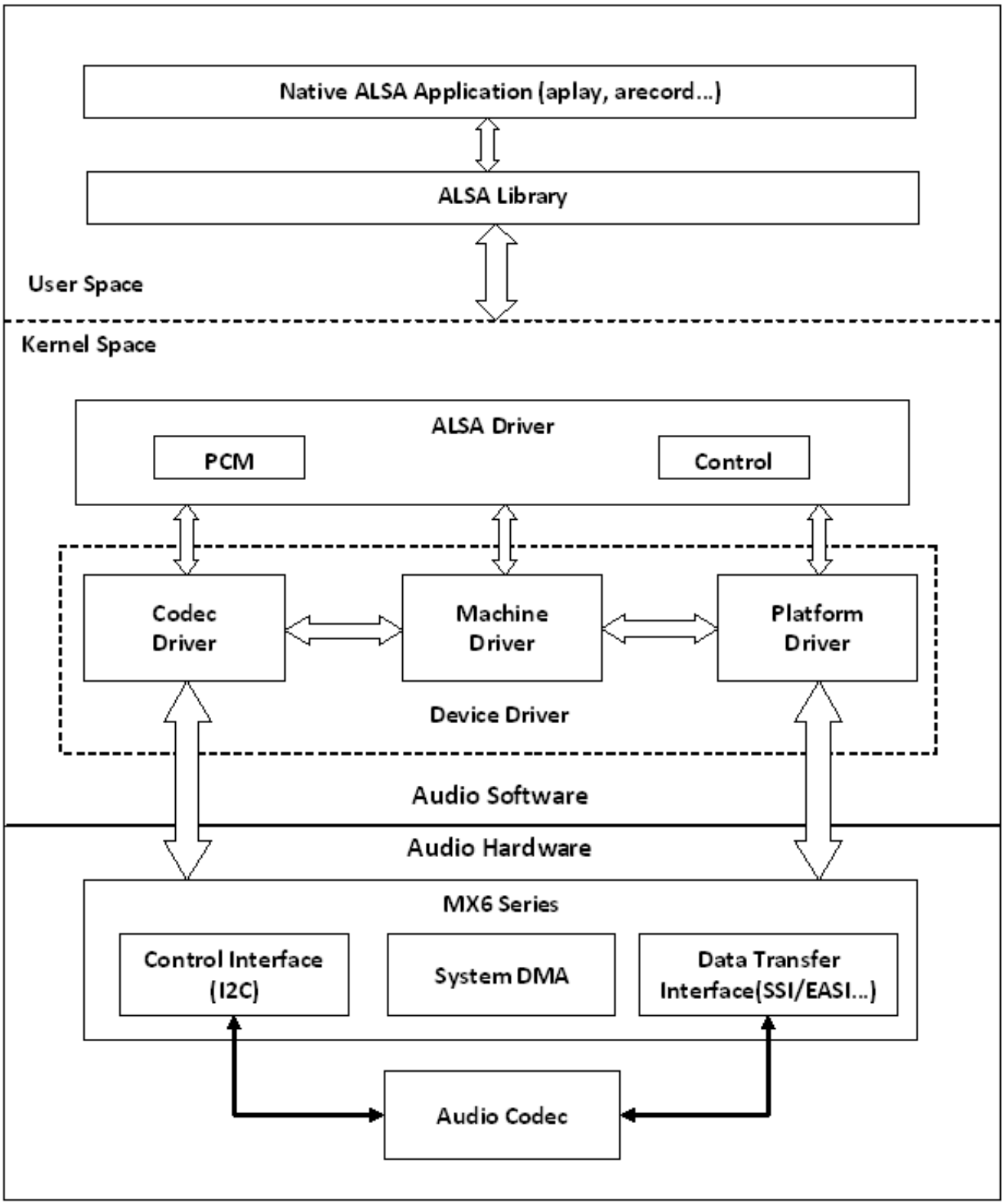


Figure 29-1. ALSA SoC Software Architecture

ASoC basically splits an embedded audio system into 3 components:

- Machine driver-handles any machine-specific controls and audio events, such as turning on an external amp at the beginning of playback.
- Platform driver-contains the audio DMA engine and audio interface drivers (for example, I^2S, AC97, PCM) for that platform.
- CODEC driver-platform independent and contains audio controls, audio interface capabilities, the CODEC DAPM definition, and CODEC I/O functions.

More detailed information about ASoC can be found in the Linux kernel documentation in the Linux OS source tree at linux/Documentation/sound/alsa/soc and at www.alsa-project.org/main/index.php/ASoC.

29.2 SoC Sound Card

Currently, the stereo CODEC (wm8962) is implemented by using SoC architecture on i.MX 6SoloLite.

29.2.1 Stereo CODEC Features

The stereo CODEC supports the following features:

- Sample rates for playback and capture are 8KHz, 32 KHz, 44.1 KHz, 48 KHz, and 96 KHz
- Channels:
  - Playback: supports two channels.
  - Capture: supports two channels.
- Audio formats:
  - Playback:
    - SNDRV_PCM_FMTBIT_S16_LE
    - SNDRV_PCM_FMTBIT_S20_3LE
    - SNDRV_PCM_FMTBIT_S24_LE
  - Capture:
    - SNDRV_PCM_FMTBIT_S16_LE
    - SNDRV_PCM_FMTBIT_S20_3LE
    - SNDRV_PCM_FMTBIT_S24_LE

29.2.2 AM/FM Codec Features

- Supported sample rate for Capture: 48 KHz
- Supported channels:

    - Capture: supports two channels.
- Supported audio formats:
    - Capture: SNDRV_PCM_FMTBIT_S16_LE

29.2.3 Sound Card Information

The registered sound card information can be listed as follows using the commands aplay -l and arecord -l. For example, the stereo sound card is registered as card 0.

```
root@freescale /$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: wm8962audio [wm8962-audio], device 0: HiFi wm8962-0 []
 Subdevices: 1/1
 Subdevice #0: subdevice #0
```

29.3 Hardware Operation

The following sections describe the hardware operation of the ASoC driver.

29.3.1 Stereo Audio CODEC

The stereo audio CODEC is controlled by the I^2C interface. The audio data is transferred from the user data buffer to/from the SSI FIFO through the DMA channel. The DMA channel is selected according to the audio sample bits. AUDMUX is used to set up the path between the SSI port and the output port which connects with the CODEC. The CODEC works in master mode and provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate.

The WM8962 ASoC CODEC driver exports the audio record/playback/mixer APIs according to the ASoC architecture.

The CODEC driver is generic and hardware independent code that configures the CODEC to provide audio capture and playback. It does not contain code that is specific to the target platform or machine. The CODEC driver handles:

- CODEC DAI and PCM configuration
- CODEC control I/O-using I^2C
- Mixers and audio controls
- CODEC audio operations
- DAC Digital mute control

The WM8962 CODEC is registered as an I^2C client when the module initializes. The APIs are exported to the upper layer by the structure snd_soc_dai_ops .

Headphone insertion/removal can be detected through a GPIO interrupt signal.

SSI dual FIFO features are enabled by default.

29.3.2 7.1 Audio Codec

The 7.1 audio codec includes 8-channel DAC and 4-channel ADC, which are controlled by the I2C interface. The audio data is transferred from the user data buffer to the ESAI fifo, through a DMA channel. The DMA channel is selected according to audio sample bits. The codec works in slave mode as the esai provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate. The ESAI supports up to eight audio output ports. While enabling ASRC, 7.1 audio codec supports 2 or 6 channel playback through ASRC. On the i.MX 6 Sabre ARD board, a cs42888 codec with 4 audio in port is used, each port receive two channels of data in the I2S format(network mode), providing 8-channel of playback functionality. This codec also has 2 audio output port connected with ESAI, providing 4-channel of recording functionality.

The codec driver is generic and hardware independent code that configures the codec to provide audio capture and playback. It does not contain code that is specific to the target platform or machine. The codec driver handles:

- Codec DAI and PCM configuration
- Codec control I/O-using I2C
- Mixers and audio controls
- Codec audio operations
- DAI Digital mute control

The CS42888 codec is registered as an I2C client when the module initializes. The APIs are exported to the upper layer by the structure snd_soc_dai_ops.

29.3.3 AM/FM Codec

The AM/FM codec is a virtual codec, it only has a PCM interface connected to the Tuner device. The audio data is transferred from the user data buffer to or from the SSI FIFO through the DMA channel. The DMA channel is selected according to the audio sample bits. AUDMUX is used to set up the path between the SSI port and the output port which connects with the codec. The codec works in master mode as it provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate.

29.4 Software Operation

The following sections describe the software operation of the ASoC driver.

29.4.1 ASoC Driver Source Architecture

File imx-pcm-dma.c is shared by the stereo ALSA SoC driver, the 7.1 ALSA SoC driver and other CODEC driver. This file is responsible for preallocating DMA buffers and managing DMA channels.

The stereo CODEC is connected to the CPU through the SSI interface. fsl_ssi.c registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip SSI interface. wm8962.c registers the stereo CODEC and hifi DAI drivers. The direct hardware operations on the stereo codec are in wm8962.c. imx-wm8962.c is the machine layer code which creates the driver device and registers the stereo sound card.

Table below shows the stereo CODEC SoC driver source files. These files are under the <YoctoBuildDir>/linux/sound/soc directory.

Table 29-1. Stereo Codec SoC Driver Files

File	Description
fsl/imx-wm8962.c	Machine layer for stereo CODEC ALSA SoC
fsl/imx-pcm-dma.c	Platform layer for stereo CODEC ALSA SoC
fsl/imx-pcm.h	Header file for PCM driver and AUDMUX register definitions
fsl/fsl_ssi.c	SSI CPU DAI driver for stereo CODEC ALSA SoC
fsl/fsl_ssi.h	Header file for SSI CPU DAI driver and SSI register definitions
codecs/wm8962.c	CODEC layer for stereo CODEC ALSA SoC
codecs/wm8962.h	Header file for stereo CODEC driver

29.4.2 Sound Card Registration

The CODECs have the same registration sequence:

1. The CODEC driver registers the CODEC driver, DAI driver, and their operation functions.

2. The platform driver registers the PCM driver, CPU DAI driver and their operation functions, pre allocates buffers for PCM components and sets playback and capture operations as applicable.
3. The machine layer creates the DAI link between CODEC and CPU registers the sound card and PCM devices.

29.4.3 Device Open

The ALSA driver performs the following functions:

- Allocates a free substream for the operation to be performed.
- Opens the low level hardware device.
- Assigns the hardware capabilities to ALSA runtime information (the runtime structure contains all the hardware, DMA, and software capabilities of an opened substream).
- Configures DMA read or write channel for operation.
- Configures CPU DAI and CODEC DAI interface.
- Configures CODEC hardware.
- Triggers the transfer.

After triggering for the first time, the subsequent DMA read/write operations are configured by the DMA callback.

29.4.4 Platform Data

See the following documents:

- Documentation/devicetree/bindings/powerpc/fsl/ssi.txt
- Documentation/devicetree/bindings/sound/wm8962.txt
- Documentation/devicetree/bindings/sound/imx-audmux.txt
- Documentation/devicetree/bindings/sound/imx-audio-wm8962.txt

29.4.5 Menu Configuration Options

The following Linux kernel configuration options are provided for this module.

- SoC Audio supports for wm8962 CODEC. In menuconfig, this option is available:

```
-> Device Drivers
     -> Sound card support
          -> Advanced Linux Sound Architecture
               -> ALSA for SoC audio support
```

```
-> SoC Audio for Freescale i.MX CPUs
     -> SoC Audio support for i.MX boards with wm8962
```

Chapter 30
Asynchronous Sample Rate Converter (ASRC) Driver

30.1 Introduction

The Asynchronous Sample Rate Converter (ASRC) converts the sampling rate of a signal to a signal of different sampling rate. The ASRC supports concurrent sample rate conversion of up to 10 channels. The sample rate conversion of each channel is associated to a pair of incoming and outgoing sampling rates. The ASRC supports up to three sampling rate pairs simultaneously.

30.1.1 Hardware Operation

ASRC includes the following features:

- Supports ratio (Fsin/Fsout) range between 1/24 to 8.
- Designed for rate conversion between 44.1 KHz, 32 KHz, 48 KHz, and 96 KHz.
- Other input sampling rates in the range of 8 KHz to 100 KHz are also supported, but with less performance (see IC spec for more details).
- Other output sampling rates in the range of 30 KHz to 100 KHz are also supported, but with less performance.
- Automatic accommodation to slow variations in the incoming and outgoing sampling rates.
- Tolerant to sample clock jitter.
- Designed mainly for real-time streaming audio usage. Can be used for non-realtime streaming audio usage when the input sampling clocks are not available.
- In any usage case, the output sampling clocks must be activated.
- In case of real-time streaming audio, both input and output clocks need to be available and activated.
- In case of non-realtime streaming audio, the input sampling rate clocks can be avoided by setting ideal-ratio values into ASRC interface registers.

The ASRC supports polling, interrupt and DMA modes, but only DMA mode is used in the platform for better performance. The ASRC supports following DMA channels:

- Peripheral to peripheral, for example: ASRC to ESAI
- Memory to peripheral, for example: memory to ASRC
- Peripheral to memory, for example: ASRC to memory

For more information, see the chapter on ASRC in the Multimedia Applications Processor documentation.

30.2 Software Operation

As an assistant component in the audio system, the ASRC driver implementation depends on the use cases in the platform.

Currently ASRC is used in following two scenarios.

- Memory > ASRC > Memory, ASRC is controlled by user application or ALSA plug-in.
- Memory > ASRC > peripheral, ASRC is controlled directly by other ALSA driver.

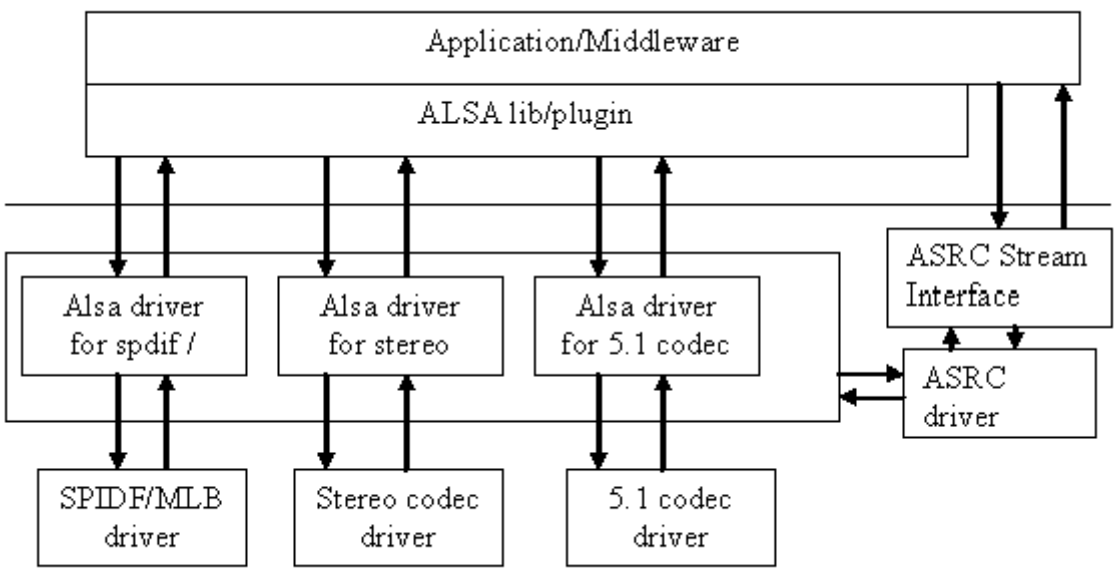


Figure 30-1. Audio Driver Interactions

As illustrated in figure above, the ASRC stream interface provides the interface for the user space. The ASRC registers itself under /dev/mxc_asrc and creates proc file /proc/driver/asrc when the module is inserted. proc is used to track the channel number for each

pair. If all the pairs are not used, users can adjust the channel number through the proc file. The total channels number should equal ten, or else the adjusted value cannot be saved properly.

Now 7.1 audio codec driver support calling ASRC driver for memroy > ASRC > perripheral(ESAI TX). All input audio file is convert into board defined sampling rate(for example, 48khz). This use case only supports 2 or 6 channel playback. To call this use case, user show follow steps below:

- call `aplay -l | grep ASRC` to get the card number and device number of playback PCM. The device name is CS42888_ASRC. For example, the card number is 0, device number is 1.
- play audio file with the card0device1 device. For example, aplay -Dplughw:0,1 $AUDIO_FILE.

30.2.1 Sequence for Memory to ASRC to Memory

- Open /dev/mxc_asrc device
- Request ASRC pair - ASRC_REQ_PAIR
- Configure ASRC pair - ASRC_CONIFG_PAIR
- Start ASRC - ASRC_START_CONV
- Write the raw audio data (to be converted) into the user maintained input buffer. Fill asrc_convert_buffer struct with input/output buffer length and address. Driver would copy output data to user maintained output buffer address according to the output buffer size. Repeat this step until all data is converted. -ASRC_CONVERT
- Stop ASRC conversion - ASRC_STOP_CONV
- Release ASRC pair - ASRC_RELEASE_PAIR
- Close /dev/mxc_asrc device

30.2.2 Sequence for Memory to ASRC to Peripheral

Memory to ASRC to peripheral audio path is involved in 7.1 audio codec driver. In 7.1 audio sound card, a new device with the name "cs42888audio [cs42888-audio], device 1: HiFi-ASRC-FE (*)" is specified for playback and capture with ASRC. The steps below show the flow of calling ASRC to memroy to peripheral:

- The sound device(PCM) has been registered and start to enable the DMA channel in ALSA driver
- Request ASRC pair - asrc_req_pair
- Configure ASRC pair - asrc_config_pair

- Enable the DMA channel from Memory to ASRC and from ASRC to Memory
- Start DMA channel and start ASRC conversion - asrc_start_conv
- When audio data playback complete, stop DMA channel and ASRC - asrc_stop_conv
- Release ASRC pair - asrc_release_pair

30.3 Source Code Structure

The table below lists the source files available in the devices directory.

```
<Yocto_BuildDir>/linux/drivers/mxc/asrc
<Yocto_BuildDir>/linux/include/linux/
<Yocto_BuildDir>/linux/sound/soc/fsl/
<Yocto_BuildDir>/linux/sound/soc/codec/
```

Table 30-1. ASRC Source File List

File	Description
mxc_asrc.c	ASRC driver implementation codes including stream interface
mxc_asrc.h	ASRC register definitions and export function declarations
imx-cs42888.c	memory to ASRC to ESAI TX implementation in 7.1 audio codec machine driver.
imx-pcm-dma.c	memroy to ASRC to ESAI TX implementation in 7.1 audio codec platform driver.
fsl_esai.c	memroy to ASRC to ESAI TX implementation in 7.1 audio codec cpu driver.
cs42xx8	memory to ASRC to ESAI TX implementation in 7.1 audio codec codec driver.
fsl_asrc.c	ALSA CPU DAI driver of ASRC P2P
fsl_asrc.h	Header file for ALSA CPU DAI driver of ASRC P2P
fsl_asrc_pcm.c	ALSA Platform layer for ASRC P2P

30.3.1 Linux Menu Configuration Options

The menu configuration options are as follows:

```
Device Drivers
    -> MXC support drivers
          -> MXC Asynchronous Sample Rate Converter support
                -> ASRC support
```

Now ASRC driver can only be configured build-in module.

30.4 Devicetree Binding

The functions of device tree bindings for ASRC M2M are as follows:

- compatible: Compatible list, must contain "fsl,imx6q-asrc".
- reg: Offset and length of the register set for the device.
- interrupts: Contains the asrc interrupt.
- clocks: Contains an entry for each entry in clock-names.
- clock-names: Must contain "mem", "ipg", "asrck", and "dma". (Generally, "dma" is used for SPBA clock.)
- dmas: Generic dma devicetree binding as described in Documentation/devicetree/bindings/dma/dma.txt.
- dma-names: Six dmas have to be defined, "txa", "rxa", "txb", "rxb", "txc", "rxc".
- fsl,clk-map-version: the mapping relationship in different SOC is different. This version number can be used to indicate clock map information.
- fsl,clk-channel-bits: indicates the channel bit information.

The functions of device tree bindings for ASRC P2P are as follows:

- compatible: Compatible list, must contain "fsl,imx6q-asrc-p2p".
- fsl,p2p-rate: A valid sample rate for Back-End (I2S) playback and record.
- fsl,p2p-width: A valid sample width for Back-End (I2S) playback and record.
- fsl,asrc-dma-rx-events: Contains three SDMA event numbers for ASRC Rx.
- fsl,asrc-dma-tx-events: Contains three SDMA event numbers for ASRC Tx.

30.4.1 Programming Interface (Exported API and IOCTLs)

The ASRC Exported API allows the ALSA driver to use ASRC services.

The ASRC IOCTLs below are used for user space applications:

ASRC_REQ_PAIR:

Apply a pair from ASRC driver. Once a pair is allocated, ASRC core clock is enabled.

ASRC_CONFIG_PAIR:

Configure ASRC pair allocated. User is responsible for providing parameters defined in struct asrc_config. Items in asrc_config is listed below:

- pair: ASRC pair allocated by the IOCTL(ASRC_REQ_PAIR).
- channel_num: channel number.
- buffer_num: buffer number need for input and output buffer use.The input/output buffers are allocated inside ASRC driver. User is responsible for remap it into user space.

- dma_buffer_size: buffer size for input and output buffers. The buffer size should be in the unit of page size. Usually, 4k bytes is used.
- input_sample_rate: input sampling rate. Input sample rate should be in 5.512k, 8k, 11.025k, 16k, 22k, 32k, 44.1k, 48k, 64k, 88.2k 96k, 176.4k, 192k.
- output_sample_rate: output sampling rate. Output sampling rate should be in 32k, 44.1k, 48k, 64k, 88.2k, 96k, 176.4k 192k.
- input_word_width: word width of input audio data. The input data word width can be 16 bit or 24 bit.
- output_word_width: word width of output audio data. The output data word width can be 16 bit or 24 bit.
- inclk: the input clock source can be ESAI RX clock, SSI1 RX clock, SSI2 RX clock, SPDIF RX clock, MLB_clock, ESAI TX clock, SSI1 TX clock, SSI2 TX clock, SPDIF TX clock, ASRCLK1 clock, NONE. If using clock except NONE, user should make sure that the clock is available.
- outclk: the output clock source is the same as the input clock source.

ASRC_CONVERT:

Convert the input data into output data according to the parameters set by ASRC_CONFIG_PAIR. Driver would copy input_buffer_length bytes data from the input_buffer_vaddr for convert. After convert, driver fill the output_buffer_length according to data number generated by ASRC and copy output_buffer_length to output_buffer_vaddr. However, before calling ASRC_CONVERT, User is responsible for filling the output_buffer_length according to the ratio of input sample rate and output sample rate. If the generated buffer size is larger than user filled output_buffer_size, driver would only copy user filled output_buffer_size to output_buffer_vaddr. If the generated buffer size is smaller than user filled output_buffer_size(the difference should be less than 64 bytes.), calling ASRC_CONVERT would fail.

- input_buffer_vaddr: virtual address of input buffer.
- output_buffer_vaddr: virtual address of output buffer.
- input_buffer_length: length of input buffer(bytes).
- output_buffer_length: length of output buffer(bytes).

ASRC_START_CONV:

Start ASRC pair convert.

ASRC_STOP_CONV:

Stop ASRC pair convert.

ASRC_STATUS:

Query ASRC pair status.

Chapter 31
The Sony/Philips Digital Interface (S/PDIF) Driver

31.1 Introduction

The Sony/Philips Digital Interface (S/PDIF) audio module is a stereo transceiver that allows the processor to receive and transmit digital audio. The S/PDIF transceiver allows the handling of both S/PDIF channel status (CS) and User (U) data. The frequency measurement block allows the S/PDIF RX section to derive the receive clock from the incoming S/PDIF stream.

31.1.1 S/PDIF Overview

Figure below shows the block diagram of the S/PDIF interface.

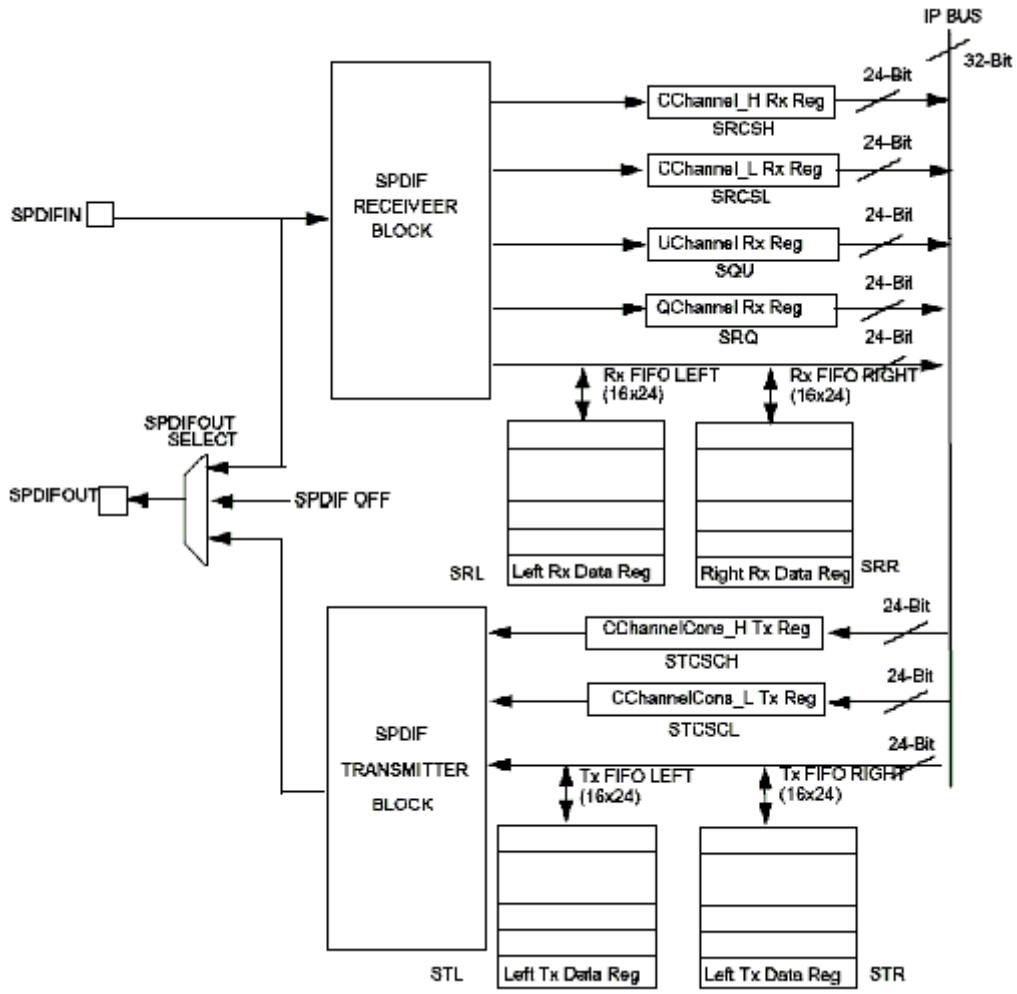


Figure 31-1. S/PDIF Transceiver Data Interface Block Diagram

31.1.2 Hardware Overview

The S/PDIF is composed of two parts:

- The S/PDIF receiver extracts the audio data from each S/PDIF frame and places the data in the S/PDIF Rx left and right FIFOs. The Channel Status and User Bits are also extracted from each frame and placed in the corresponding registers. The S/PDIF receiver provides a bypass option for direct transfer of the S/PDIF input signal to the S/PDIF transmitter.
- For the S/PDIF transmitter, the audio data is provided by the processor through the SPDIFTxLeft and SPDIFTxRight registers. The Channel Status bits are provided through the corresponding registers. The S/PDIF transmitter generates a S/PDIF output bitstream in the biphase mark format (IEC958), which consists of audio data, channel status and user bits.

In the S/PDIF transmitter, the IEC958 biphase bit stream is generated on both edges of the S/PDIF Transmit clock. The S/PDIF Transmit clock is generated by the S/PDIF internal clock dividers and the sources are from outside of the S/PDIF block. The S/PDIF receiver can recover the S/PDIF Rx clock from the S/PDIF stream. Figure 31-1 shows the clock structure of the S/PDIF transceiver.

31.1.3 Software Overview

The S/PDIF driver is designed under ALSA System on Chip (ASoC) layer. The ASoC driver for S/PDIF provides one playback device for Tx and one capture device for Rx. The playback output audio format can be linear PCM data or compressed data with 16-bit, 20-bit, and 24-bit audio. The allowed sampling bit rates are 44.1, 48 or 32 KHz. The capture input audio format can be linear PCM data or compressed 24-bit data and the allowed sampling bit rates are from 16 to 96 KHz. The driver provides the same interface for PCM and compressed data transmission.

31.1.4 The ASoC layer

The ASoC layer divides audio drivers for embedded platforms into separate layers that can be reused. ASoC divides an audio driver into a codec driver, a machine layer, a DAI (digital audio interface) layer, and a platform layer. The Linux kernel documentation has some concise description of these layers in linux/Documentation/sound/alsa/soc. In the case of the S/PDIF driver, we are able to reuse the platform layer (imx-pcm-dma-mx2.c) that is used by the ssi stereo codec driver.

31.2 S/PDIF Tx Driver

The S/PDIF Tx driver supports the following features.

- 32, 44.1 and 48 KHz sample rates.
- Signed 16 and 24-bit little Endian sample format. Due to S/PDIF SDMA feature, the 24-bit output sample file must have 32-bits in each channel per frame. Only the 24 LSBs are valid.
- In the ALSA subsystem, the supported format is defined as S16_LE and S24_LE.
- Stereo playback.
- Information query through iecset or amixer.

- The device ID can be determined by using the 'aplay -l' utility to list out the playback audio devices.

  For example:

```
root@freescale ~$ aplay -l

**** List of PLAYBACK Hardware Devices ****

card 0: imxspdif [imx-spdif], device 0: S/PDIF PCM Playback dit-hifi-0 []

  Subdevices: 1/1

  Subdevice #0: subdevice #0
```

- **NOTE**

  The number at the beginning of the IMX_SPDIF line is the card ID. The string in the square brackets is the card name.

- The ALSA utility provides a common method for user spaces to operate and use ALSA drivers

```
#aplay -Dplughw:0,0 audio.wav
```

NOTE

The -D parameter of aplay indicates the PCM device with card ID and PCM device ID: hw:[card id],[pcm device id]

The "iecset" utility provides a common method to set or dump the IEC958 status bits.

```
#iecset -c 0
```

31.2.1 Driver Design

Before S/PDIF playback, the configuration, interrupt, clock and channel registers are initialized. During S/PDIF playback, the channel status bits are fixed. The DMA and interrupts are enabled. S/PDIF has 16 TX sample FIFOs on Left and Right channel respectively. When both FIFOs are empty, an empty interrupt is generated if the empty interrupt is enabled. If no data are refilled in the 20.8 μs (1/48000), an underrun interrupt is generated. Overrun is avoided if only 16 sample FIFOs are filled for each channel every time. If auto re-synchronization is enabled, the hardware checks if the left and right FIFO are in sync, and if not, it sets the filling pointer of the right FIFO to be equal to the filling pointer of the left FIFO and an interrupt is generated.

31.2.2 Provided User Interface

The S/PDIF transmitter driver provides one ALSA mixer sound control interface to the user besides the common PCM operations interface. It provides the interface for the user to write S/PDIF channel status codes into the driver so they can be sent in the S/PDIF stream. The input parameter of this interface is the IEC958 digital audio structure shown below, and only status member is used:

```
struct snd_aes_iec958 {
                    unsigned char status[24];       /* AES/IEC958 channel status bits */
                    unsigned char subcode[147];     /* AES/IEC958 subcode bits */
                    unsigned char pad;              /* nothing */
                    unsigned char dig_subframe[4];  /* AES/IEC958 subframe bits */
};
```

31.3 S/PDIF Rx Driver

The S/PDIF Rx driver supports the following features:

- 16, 32, 44.1, 48, 64 and 96 KHz receiving sample rate
- Signed 24-bit little endian sample format. Due to S/PDIF SDMA feature, each channel bit length in PCM recorded frame is 32 bits, and only the 24 LSBs are valid

  In ALSA subsystem, the supported format is defined as S24_LE.

- Stereo record.
- The device ID can be determined by using the 'arecord -l' to list out record devices.

  For example:

  ```
  root@freescale ~$ arecord -l

  **** List of CAPTURE Hardware Devices ****

  card 0: cs42888audio [cs42888-audio], device 0: HiFi CS42888-0 []

    Subdevices: 1/1

    Subdevice #0: subdevice #0

  card 1: imxspdif [imx-spdif], device 0: S/PDIF PCM Capture dir-hifi-0 []

    Subdevices: 1/1

    Subdevice #0: subdevice #0
  ```

- The ALSA utility provides a common method for user spaces to operate and use ALSA drivers.

  ```
  #arecord -Dplughw:1,0" -c 2 -r 44100 -f S24_LE record.wav
  ```

NOTE

The -D parameter of the arecord indicates the PCM device with card ID and PCM device ID: hw:[card id],[pcm device id]

The "iecset" utility provides a common method to set or dump the IEC958 status bits.

```
#iecset -c 1
```

31.3.1 Driver Design

Before the driver can read a data frame from the S/PDIF receiver FIFO, it must wait for the internal DPLL to be locked. Using the high-speed system clock, the internal DPLL can extract the bit clock (advanced pulse) from the input bit stream. When this internal DPLL is locked, the LOCK bit of PhaseConfig Register is set and the driver configures the interrupt, clock and SDMA channel. After that, the driver can receive audio data, channel status, user bits and valid bits concurrently.

For channel status reception, a total of 48 channel status bits are received in two registers. The driver reads them out when a user application makes a request.

For user bits reception, there are two modes for User Channel reception: CD and non-CD. The mode is determined by the USyncMode (bit 1 of CDText_Control register). User can call the sound control interface to set the mode (see Table 31-1), but no matter what the mode is, the driver handles the user bits in the same way. For the S/PDIF Rx, the hardware block copies the Q bits from the user bits to the QChannel registers and puts the user bits in UChannel registers. The driver allocates two queue buffers for both U bits and Q bits. The U bits queue buffer is 96x2 bytes in size, the Q bits queue buffer is 12x2 bytes in size, and queue buffers are filled in the U/Q Full, Err and Sync interrupt handlers. This means that the user can get the previous ready U/Q bits while S/PDIF driver is reading new U/Q bits.

For valid bit reception, S/PDIF Rx hardware block triggers an interrupt and set interrupt status upon reception. A sound control interface is provided for the user to get the status of this valid bit.

31.3.2 Provided User Interface

The S/PDIF Rx driver provides interfaces for user application as shown in table below.

Table 31-1. S/PDIF Rx Driver Interfaces

Interface	Type	Mode[1]	Parameter	Comment
Common PCM	PCM	-	-	PCM open/close prepare/trigger hw_params/sw_params
Rx Sample Rate	Sound Control[2]	r	Integer Range: [16000, 96000]	Get sample rate. It is not accurate due to DPLL frequency measure module. So the user application must do a correction to the get value.
USyncMode	Sound Control	rw	Boolean Value: 0 or 1	Set 1 for CD mode Set 0 for non-CD mode
Channel Status	Sound Control	r	struct snd_aes_iec958 Only status [6] array member is used	-
User bit	Sound Control	r	Byte array 96 bytes for U bits 12 bytes for Q bits	-
No good V bit	Sound Control	r	Boolean Value: 0 or 1	An interrupt is associated with the valid flag. (interrupt 16 - SPDIFValNoGood). This interrupt is set every time a frame is seen on the SPDIF interface with the valid bit set to invalid.

1. The mode column shows the interface attribute: r (read) or w (write)
2. The sound control type of interface is called by the snd_ctl_xxx() alsa-lib function

The user application can follow the program flow from Figure 31-2 to use the S/PDIF Rx driver. First, the application opens the S/PDIF Rx PCM device, waits for the DPLL to lock the input bit stream, and gets the input sample rate. If the USyncMode needs to be set, set it before reading the U/Q bits. Next, set the hardware parameters, including channel number, format and capture sample rate which is obtained from the driver. Then, call prepare and trigger to startup S/PDIF Rx stream read. Finally, call the read function to get the data. During the reading process, applications can read the U/Q bits and channel status from the driver and valid the no good bit.

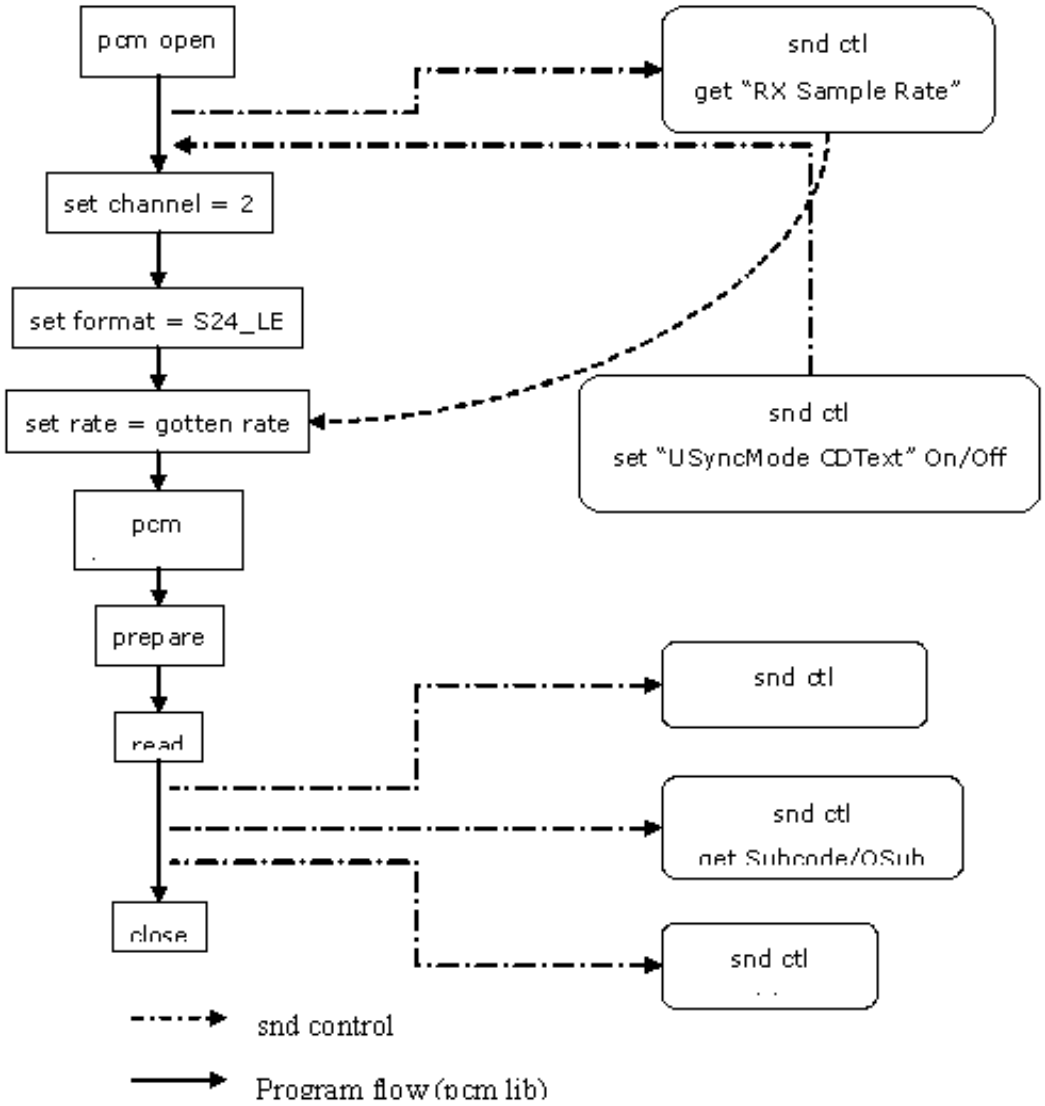


Figure 31-2. S/PDIF Rx Application Program Flow

31.4 Source Code Structure

Table below lists the source files for the driver.

These files are under the <Yocto_BuildDir>/linux/ directory.

Table 31-2. S/PDIF Driver Files

File	Description
sound/soc/codecs/spdif_transmitter.c	S/PDIF ALSA SOC playback codec driver
sound/soc/codecs/spdif_receiver.c	S/PDIF ALSA SOC record codec driver
sound/soc/fsl/imx-spdif.c	S/PDIF ALSA SOC machine layer

Table continues on the next page...

Table 31-2. S/PDIF Driver Files (continued)

File	Description
sound/soc/fsl/fsl_spdif.c	S/PDIF ALSA SOC DAI layer
sound/soc/fsl/imx-pcm-dma.c	ALSA SOC platform layer
sound/soc/fsl/imx-pcm.h	ALSA SOC platform layer header

31.5 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

To get to these options, use the bitbake linux-imx -c menuconfig command. Select **Configure the Kernel** on the screen displayed and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_SND_IMX_SPDIF - Configuration option for the S/PDIF driver. In the menuconfig, this option is available:

```
-> Device Drivers
     -> Sound card support
          -> Advanced Linux Sound Architecture
               -> ALSA for SoC audio support
                    -> SoC Audio for Freescale i.MX CPUs
                         -> SoC Audio support for i.MX boards with S/PDIF
```

31.6 Device Tree Bindings

Please refer to the following documents:

- Documentation/devicetree/bindings/sound/fsl,spdif.txt
- Documentation/devicetree/bindings/sound/imx-audio-spdif.txt

31.7 Interrupts and Exceptions

S/PDIF Tx/Rx hardware block has many interrupts to indicate the success, exception and event.

The driver handles the following interrupts:

- DPLL Lock and Loss Lock-Saves the DPLL lock status; this is used when getting the Rx sample rate

- U/Q Channel Full and overrun/underrun-Puts the U/Q channel register data into queue buffer, and update the queue buffer write pointer
- U/Q Channel Sync-Saves the ID of the buffer whose U/Q data is ready for read out
- U/Q Channel Error-Resets the U/Q queue buffer

31.8 Unit Test Preparation

In order to prepare to run a unit test, perform the following actions:

- Setup M-Audio Transit USB sound card by installing M-Audio Transit driver on your PC.
- Install WaveLab tools on your PC.

31.8.1 Tx test step

- Plug optical line into [line\optical] port of M-Audio transit.

NOTE

Make sure the [optical out] port of M-Audio transit has no output (red light off) after plugging the optical line.

- Startup WaveLab, press record button on toolbar, setup the record file name, sample rate, channel number, then do record.
- Meanwhile, on board use following command to play one wave file:

```
#aplay -D hw:[card id],[pcm id] audioXXkYYS.wav
```

- After aplay finishing, stop recording in WaveLab.
- Play the recorded wav file in wavelab to check.

31.8.2 Rx test step

- Plug optical line into [optical port] of M-Audio transit
- Startup WaveLab, open a test wav file: audioXXkYYS.wav to play in loop
- Meanwhile, on board use following command to record one wave file. After finish recording, you may playback the record wav file on other audio card on board or PC

```
#arecord -D hw:[card id],[pcm id] -c 2 -d 20 -r [sample rate in Hz] -f S24_LE record.wav
```

NOTE

The sample rate argument in the arecord command must be consistent with wav file playing on WaveLab.

Chapter 32
SPI NOR Flash Memory Technology Device (MTD) Driver

32.1 Introduction

The SPI NOR Flash Memory Technology Device (MTD) driver provides the support to the data Flash though the SPI interface.

By default, the SPI NOR Flash MTD driver creates static MTD partitions to support data Flash.

32.1.1 Hardware Operation

On some boards, the SPI NOR - AT45DB321D is equipped, while on some boards M25P32 is equipped. Check the SPI NOR type on the boards and then configure it properly.

The AT45DB321D is a 2.7 V, serial-interface sequential access Flash memory. The AT45DB321D serial interface is SPI compatible for frequencies up to 66 MHz. The memory is organized as 8,192 pages of 512 bytes or 528 bytes. The AT45DB321D also contains two SRAM buffers of 512/528 bytes each which allow receiving of data while a page in the main memory is being reprogrammed, as well as writing a continuous data stream.

The M25P32 is a 32 Mbit (4M x 8) Serial Flash memory, with advanced write protection mechanisms, accessed by a high-speed SPI-compatible bus up to 75 MHz. The memory is organized as 64 sectors, each containing 256 pages. Each page is 256 bytes wide. Thus, the whole memory can be viewed as consisting of 16384 pages, or 4,194,304 bytes. The memory can be programmed 1 to 256 bytes at a time using the Page Program instruction. The whole memory can be erased using the Bulk Erase instruction, or a sector at a time, using the Sector Erase instruction.

Unlike conventional Flash memories that are accessed randomly, these two SPI NOR access data sequentially. They operate from a single 2.7-3.6 V power supply for program and read operations. They are enabled through a chip select pin and accessed through a three-wire interface: Serial Input, Serial Output, and Serial Clock.

32.1.2 Software Operation

In a Flash-based embedded Linux system, a number of Linux technologies work together to implement a file system. Figure below illustrates the relationships between some of the standard components.

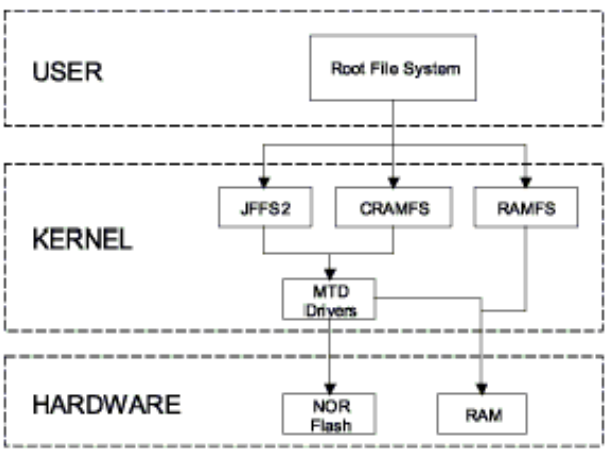


Figure 32-1. Components of a Flash-Based File System

The MTD subsystem for Linux OS is a generic interface to memory devices, such as Flash and RAM, providing simple read, write, and erase access to physical memory devices. Devices called mtdblock devices can be mounted by JFFS, JFFS2 and CRAMFS file systems. The SPI NOR MTD driver is based on the MTD data Flash driver in the kernel by adding SPI access. In the initialization phase, the SPI NOR MTD driver detects a data Flash by reading the JEDEC ID. Then the driver adds the MTD device. The SPI NOR MTD driver also provides the interfaces to read, write, and erase NOR Flash.

32.1.3 Driver Features

This NOR MTD implementation supports the following features:

- Provides necessary information for the upper layer MTD driver

32.1.4 Source Code Structure

The SPI NOR MTD driver is implemented in the following directory:

drivers/mtd/devices/

Table below shows the driver files:

Table 32-1. SPI NOR MTD Driver Files

File	Description
m25p80.c	Source file

32.1.5 Menu Configuration Options

To get to the SPI NOR MTD driver, use the command bitbake linux-imx -c menuconfig. On the screen displayed, select Configure the kernel and exit. When the next screen appears select the following options to enable the SPI NOR MTD driver accordingly:

- CONFIG_MTD_M25P80: This config enables access to most modern SPI flash chips, used for program and data storage.
- Device Drivers > Memory Technology Device (MTD) support >Self-contained MTD device drivers > Support most SPI Flash chips (AT26DF, M25P, W25X, ...)

Chapter 33
MMC/SD/SDIO Host Driver

33.1 Introduction

The MultiMediaCard (MMC)/ Secure Digital (SD)/ Secure Digital Input Output (SDIO) Host driver implements a standard Linux driver interface to the ultra MMC/SD host controller (uSDHC) .

The host driver is part of the Linux kernel MMC framework.

The MMC driver has the following features:

- 1-bit or 4-bit operation for SD3.0 and SDIO 2.0 cards (so far we support SDIO v2.0 (AR6003 is verified)).
- Supports card insertion and removal detections.
- Supports the standard MMC commands.
- PIO and DMA data transfers.
- Supports power management.
- Supports 1/4/8-bit operations for MMC cards.
- For i.MX 6, USDHC supports eMMC4.4 SDR and DDR modes.
- For i.MX 7Dual, USDHC supports eMMC5.0, which includes HS400 and HS200.
- Supports SD3.0 SDR50 and SDR104 modes.

33.1.1 Hardware Operation

The MMC communication is based on an advanced 11-pin serial bus designed to operate in a low voltage range. The uSDHC module supports MMC along with SD memory and I/O functions. The uSDHC controls the MMC, SD memory, and I/O cards by sending commands to cards and performing data accesses to and from the cards. The SD memory card system defines two alternative communication protocols: SD and SPI. The uSDHC only supports the SD bus protocol.

The uSDHC command transfer type and uSDHC command argument registers allow a command to be issued to the card. The uSDHC command, system control, and protocol control registers allow the users to specify the format of the data and response and to control the read wait cycle.

There are four 32-bit registers used to store the response from the card in the uSDHC. The uSDHC reads these four registers to get the command response directly. The uSDHC uses a fully configurable 128x32-bit FIFO for read and write. The buffer is used as temporary storage for data being transferred between the host system and the card, and vice versa. The uSDHC data buffer access register bits hold 32-bit data upon a read or write transfer.

For receiving data, the steps are as follows:

1. The uSDHC controller generates a DMA request when there are more words received in the buffer than the amount set in the RD_WML register
2. Upon receiving this request, DMA engine starts transferring data from the uSDHC FIFO to system memory by reading the data buffer access register.

For transmitting data, the steps are as follows:

1. The uSDHC controller generates a DMA request whenever the amount of the buffer space exceeds the value set in the WR_WML register.
2. Upon receiving this request, the DMA engine starts moving data from the system memory to the uSDHC FIFO by writing to the Data Buffer Access Register for a number of pre-defined bytes.

The read-only uSDHC Present State and Interrupt Status Registers provide uSDHC operations status, application FIFO status, error conditions, and interrupt status.

When certain events occur, the module has the ability to generate interrupts as well as set the corresponding Status Register bits. The uSDHC interrupt status enable and signal-enable registers allow the user to control if these interrupts occur.

33.1.2 Software Operation

The Linux OS contains an MMC bus driver which implements the MMC bus protocols. The MMC block driver handles the file system read/write calls and uses the low level MMC host controller interface driver to send the commands to the uSDHC.

The MMC driver is responsible for implementing standard entry points for `init`, `exit`, `request`, and `set_ios`. The driver implements the following functions:

- The `init` function `esdhc_pltfm_init()` initializes the platform hardware and set platform dependant flags or values to sdhci_host structure.
- The `exit` function `esdhc_pltfm_exit()` deinitializes the platform hardware and frees the memory allocated.
- The function `esdhc_pltfm_get_max_clock()` gets the maximum SD bus clock frequency supported by the platform.
- The function `esdhc_pltfm_get_min_clock()` gets the minimum SD bus clock frequency supported by the platform.
- `esdhc_pltfm_get_ro()` gets the card read only status.
- `esdhc_execute_tuning()` handles the preparation for tuning. It's only used for SD3.0 UHS-I mode.
- `esdhc_set_clock()` handles the clock change request.

Figure below shows how the MMC-related drivers are layered.

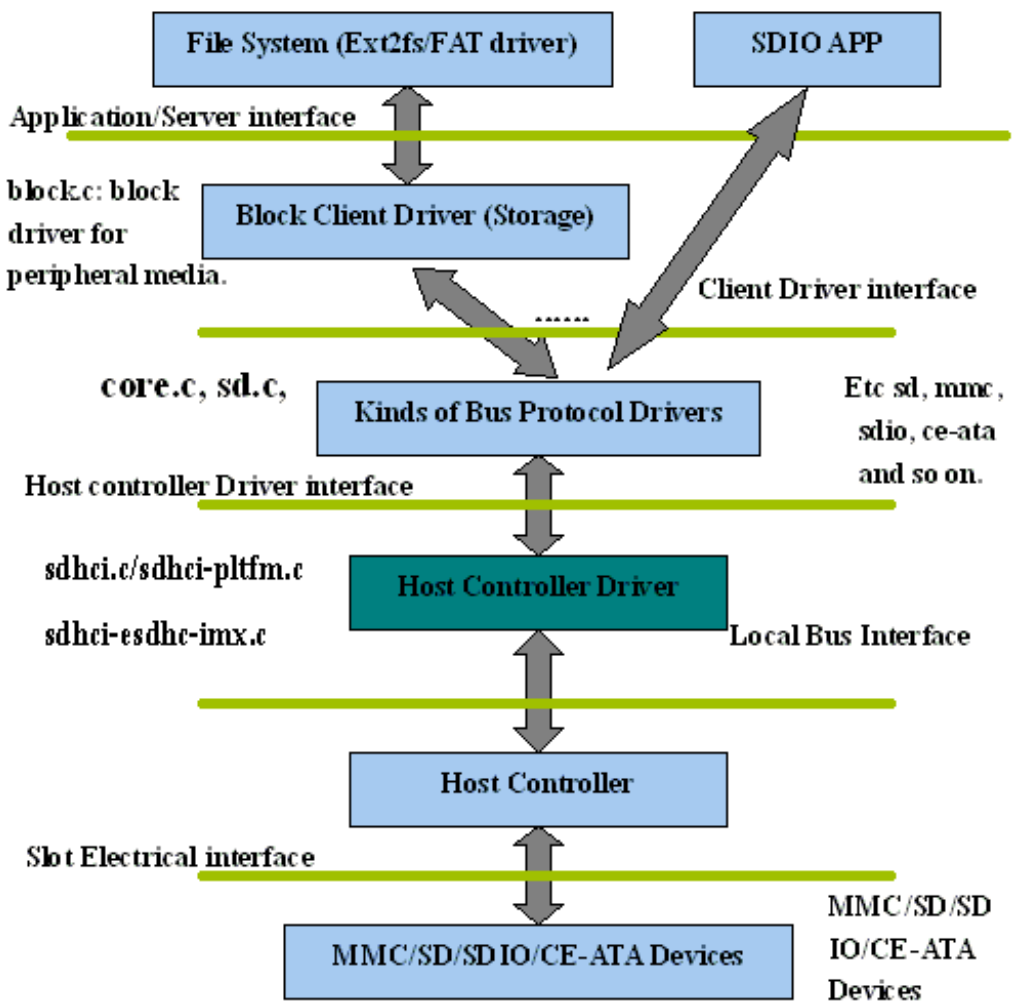


Figure 33-1. MMC Drivers Layering

33.2 Driver Features

The MMC driver supports the following features:

- Supports multiple uSDHC modules.
- Provides all the entry points to interface with the Linux MMC core driver.
- MMC and SD cards.
- SDIO cards.
- SD3.0 cards.
- Recognizes data transfer errors such as command time outs and CRC errors.
- Power management.
- It supports to be built as loadable or builtin module

33.2.1 Source Code Structure

Table below shows the uSDHC source files available in the kernel source directory: drivers/mmc/host/.

Table 33-1. uSDHC Driver Files MMC/SD Driver Files

File	Description
sdhci.c	sdhci standard stack code
sdhci-pltfm.c	sdhci platform layer
sdhci-esdhc-imx.c	uSDHC driver
sdhci-esdhc.h	uSDHC driver header file

33.2.2 Menu Configuration Options

The following Linux kernel configuration options are provided for this module.

- CONFIG_MMC builds support for the MMC bus protocol. In menuconfig, this option is available under:
  - Device Drivers > MMC/SD/SDIO Card support
  - By default, this option is Y.
- CONFIG_MMC_BLOCK builds support for MMC block device driver which can be used to mount the file system. In menuconfig, this option is available under:
  - Device Drivers > MMC/SD Card Support > MMC block device driver
  - By default, this option is Y.

- CONFIG_MMC_SDHCI_ESDHC_IMX is used for the i.MX USDHC ports. In menuconfig, this option is found under:
  - Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller

  To compile SDHCI driver as a loadable module, several options should be selected as indicated below:
  - CONFIG_MMC_SDHCI=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support
  - CONFIG_MMC_SDHCI_PLTFM=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus.
  - CONFIG_MMC_SDHCI_ESDHC_IMX=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller

  To compile SDHCI driver as a builttin module, several options should be selected as indicated below:
  - CONFIG_MMC_SDHCI=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support
  - CONFIG_MMC_SDHCI_PLTFM=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus.
  - CONFIG_MMC_SDHCI_ESDHC_IMX=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller
- CONFIG_MMC_UNSAFE_RESUME is used for embedded systems which use a MMC/SD/SDIO card for rootfs. In menuconfig, this option is found under:
  - Device drivers > MMC/SD/SDIO Card Support > Assume MMC/SD cards are non-removable.

33.2.3 Devicetree Binding

Required properties:

- compatible : Should be "fsl,<chip>-esdhc"
- reg : Should contain eSDHC registers location and
- interrupts : Should contain eSDHC interrupt

Optional properties:

- non-removable : Indicate the card is wired to host permanently
- fsl,cd-internal : Indicate to use controller internal card detection
- fsl,wp-internal : Indicate to use controller internal write protection
- cd-gpios : Specify GPIOs for card detection
- wp-gpios : Specify GPIOs for write protection
- fsl,delay-line : Specify delay line value for emmc ddr mode

```
Example:usdhc@02194000 { /* uSDHC2 */
        compatible = "fsl,imx6q-usdhc";
        reg = <0x02194000 0x4000>;
        interrupts = <0 23 0x04>;
        clocks = <&clks 164>, <&clks 164>, <&clks 164>;
        clock-names = "ipg", "ahb", "per";
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_usdhc2_1>;
        cd-gpios = <&gpio2 2 0>;
        wp-gpios = <&gpio2 3 0>;
        bus-width = <8>;
        no-1-8-v;
        keep-power-in-suspend;
        enable-sdio-wakeup;
        status = "okay";
};
```

Reference:

- Documentation/devicetree/bindings/mmc/fsl-imx-esdhc.txt
- arch/arm/boot/dts/imx6*.dtsi

33.2.4 Programming Interface

This driver implements the functions required by the MMC bus protocol to interface with the i.MX uSDHC module.

See the Linux document generated from build: make htmldocs.

33.2.5 Loadable Module Operations

The SDHCI driver can be built as loadable or builtin module.

1. How to build SDHCI driver as loadable module.
   - CONFIG_MMC_SDHCI=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support

- CONFIG_MMC_SDHCI_PLTFM=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus.
- CONFIG_MMC_SDHCI_ESDHC_IMX=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller

2. How to load and unload SDHCI module.

   Due to dependency, load or unload the module following the module sequence shown below.

   run the following commands to load module:

   - load modules via insmod command, assuming the files of sdhci.ko and sdhci-platform.ko exist in current directory.

     ```
     $> insmod sdhci.ko
     $> insmod sdhci-platform.ko
     ```

   - load modules via modprobe command, make sure the files of sdhci.ko and sdhci-platform.ko exist in corresponding kernel module lib directory.

     ```
     $> modprobe sdhci.ko
     $> modprobe sdhci-platform.ko
     ```

   run the following commands to unload module.:

   - unload modules via insmod command.

     ```
     $> rmsmod sdhci-platform
     $> rmsmod sdhci
     ```

   - unload modules via modprobe command.

     ```
     $> modprobe -r sdhci-platform
     $> modprobe -r sdhci
     ```

Chapter 34
NAND GPMI Flash Driver

34.1 Introduction

The NAND Flash Memory Technology Devices (MTD) driver is used in the Generic-Purpose Media Interface (GPMI) controller on the i.MX 6 serials.

Only the hardware-specific layer has to be implemented for the NAND MTD driver to operate.

The rest of the functionality such as Flash read/write/erase is automatically handled by the generic layer provided by the Linux MTD subsystem for NAND devices.

34.1.1 Hardware Operation

NAND Flash is a nonvolatile storage device used for embedded systems.

It does not support random accesses of memory as in the case of RAM or NOR Flash. Reading or writing to NAND Flash must be done through the GPMI. NAND Flash is a sequential access device appropriate for mass storage applications. Code stored on NAND Flash cannot be executed from there. Code must be loaded into RAM memory and executed from there. The i.MX 6 contains a hardware error-correcting block.

34.2 Software Operation

MTDs in Linux covers all memory devices such as RAM, ROM, and different kinds of NOR/NAND Flashes.

The MTD subsystem provides uniform access to all such devices. Above the MTD devices there could be either MTD block device emulation with a Flash file system (JFFS2) or a UBI layer. The UBI layer in turn, can have either UBIFS above the volumes or a Flash Translation Layer (FTL) with a regular file system (FAT, Ext2/3) above it. The

hardware-specific driver interfaces with the GPMI module on the i.MX 6. It implements the lowest level operations such as read, write and erase. If enabled, it also provides information about partitions on the NAND device-this information has to be provided by platform code.

The NAND driver is the point where read/write errors can be recovered if possible. Hardware error correction is performed by BCH blocks and is driven by NAND drivers code.

Detailed information about NAND driver interfaces can be found at www.linux-mtd.infradead.org

34.2.1 Basic Operations: Read/Write

The NAND driver exports the following callbacks:

```
gpmi_ecc_read_page (with ECC)
gpmi_ecc_write_page (with ECC)
gpmi_read_byte (without ECC)
gpmi_read_buf (without ECC)
gpmi_write_buf (without ECC)
gpmi_ecc_read_oob (with ECC)
gpmi_ecc_write_oob (with ECC)
```

These functions read the requested amount of data, with or without error correction. In the case of read, the gpmi_read_page() function is called, which creates the DMA chain, submits it to execute, and waits for completion. The write case is a bit more complex: the data to be written is mapped and flushed out by calling gpmi_send_page().

34.2.2 Error Correction

When reading or writing data to Flash, some bits can be flipped. This is normal behavior, and NAND drivers utilize various error correcting schemes to correct this. It could be resolved with software or hardware error correction. The GPMI driver uses only a hardware correction scheme with the help of an hardware accelerator-BCH.

For BCH, the page laylout of 2K page is (2k + 64), the page layout of 4K page is (4k + 218) the page layout of 8K page is (8K + 448).

34.2.3 Boot Control Block Management

During startup, the NAND driver scans the first block for the presence of a NAND Control Block (NCB). Its presence is detected by magic signatures. When a signature is found, the boot block candidate is checked for errors using Hamming code. If errors are found, they are fixed, if possible. If the NCB is found, it is parsed to retrieve timings for the NAND chip.

All boot control blocks are created when formatting the medium using the user space application kobs-ng .

34.2.4 Bad Block Handling

When the driver begins, by default, it builds the bad block table. It is possible to determine if a block is bad, dynamically, but to improve performance it is done at boot time. The badness of the erase block is determined by checking a pattern in the beginning of the spare area on each page of the block. However, if the chip uses hardware error correction, the bad marks falls into the ECC bytes area. Therefore, if hardware error correction is used, the bad block mark should be moved.

34.3 Source Code Structure

The NAND driver is located in the drivers/mtd/nand/ directory.

The following files are included in the NAND driver:

```
bch-regs.h
gpmi-lib.c
gpmi-nand.c
gpmi-nand.h
gpmi-regs.h
Makefile
```

34.3.1 Menu Configuration Options

To enable the NAND driver, the following options must be set:

- CONFIG_IMX_HAVE_PLATFORM_GPMI_NAND= [Y]
- CONFIG_MTD_NAND_GPMI_NAND= [Y | M]

In addition, these MTD options must be enabled:

- CONFIG_MTD_NAND = [y | m]

- CONFIG_MTD = y
- CONFIG_MTD_PARTITIONS = y
- CONFIG_MTD_CHAR = y
- CONFIG_MTD_BLOCK = y

In addition, these UBI options must be enabled:

- CONFIG_MTD_UBI=y
- CONFIG_MTD_UBI_WL_THRESHOLD=4096
- CONFIG_MTD_UBI_BEB_RESERVE=1
- CONFIG_UBIFS_FS=y
- CONFIG_UBIFS_FS_LZO=y
- CONFIG_UBIFS_FS_ZLIB=y

Chapter 35
SATA Driver

35.1 Hardware Operation

The detailed hardware operation of SATA is detailed in the Synopsys DesignWare Cores SATA AHCI documentation, named SATA_Data_Book.pdf.

35.1.1 Software Operation

The details about the libata APIs, see the libATA Developer's Guide named libata.pdf pulished by Jeff Gazik.

The SATA AHCI driver is based on the LIBATA layer of the block device infrastructure of the Linux kernel . Freescale-integrated AHCI linux driver combined the standard AHCI drivers handle the details of the integrated Freescale SATA AHCI controller, while the LIBATA layer understands and executes the SATA protocols. The SATA device, such as a hard disk, is exposed to the application in user space by the /dev/sda* interface. Filesystems are built upon the block device. The AHCI specified integrated DMA engine, which assists the SATA controller hardware in the DMA transfer modes.

35.1.2 Source Code Structure Configuration

The source code of Freescale's AHCI SATA driver is located in the following folder: <kernel_dir>/driver/ata/ahci_imx.c

The standard AHCI and AHCI platform drivers are used to do the actual SATA operations.

The source code of the standard AHCI and AHCI platform drivers are located in drivers/ata/ folder, named as ahci.c and ahci-platform.c.

35.1.3 Linux Menu Configuration Options

The following Linux kernel configurations are provided for SATA driver:

```
   Symbol: AHCI_IMX
[=y]

   Type  :
tristate

   Prompt: Freescale i.MX AHCI SATA
support


Location:

       -> Device
Drivers

         -> Serial ATA and Parallel ATA drivers (ATA
[=y])
           -> Platform AHCI SATA support (SATA_AHCI_PLATFORM
[=y])
```

In busybox, enable "fdisk" under "Linux System Utilities".

35.1.4 Board Configuration Options

With the power off, install the SATA cable and hard drive.

35.2 Programming Interface

The application interface to the SATA driver is the standard POSIX device interface (for example: open, close, read, write, and ioctl) on /dev/sda*.

35.2.1 Usage Example2

NOTE

There may be a known error message when few kinds of SATA disks are initialized, such as:

ata1.00: serial number mismatch '090311PB0300QKG3TB1A' ! = ''

ata1.00: revalidation failed (errno=-19)

pls ignore that.

1. After building the kernel and the SATA AHCI driver and deploying, boot the target, and log in as root.
2. Make sure that the AHCI and AHCI platform drivers are built in the kernel or loaded into the kernel.

You should see messages similar to the following:

```
ahci: SSS flag set, parallel bus scan disabled
ahci ahci: AHCI 0001.0300 32 slots 1 ports 3 Gbps 0x1 impl platform mode
ahci ahci: flags: ncq sntf stag pm led clo only pmp pio slum part ccc apst
scsi0 : ahci_platform
ata1: SATA max UDMA/133 mmio [mem 0x02200000-0x02203fff] port 0x100 irq 71
ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
ata1.00: ATA-8: SAMSUNG HM100UI, 2AM10001, max UDMA/133
ata1.00: 1953525168 sectors, multi 0: LBA48 NCQ (depth 31/32)
ata1.00: configured for UDMA/133
scsi 0:0:0:0: Direct-Access     ATA      SAMSUNG HM100UI  2AM1 PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 1953525168 512-byte logical blocks: (1.00 TB/931 GiB)
sd 0:0:0:0: [sda] 4096-byte physical blocks
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO or FUA
sda: sda1
sd 0:0:0:0: [sda] Attached SCSI disk
```

You may use standard Linux utilities to partition and create a file system on the drive (for example: fdisk and mke2fs) to be mounted and used by applications.

The device nodes for the drive and its partitions appears under /dev/sda*. For example, to check basic kernel settings for the drive, execute hdparm /dev/sda.

35.2.2 Usage Example

Create Partitons

The following command can be used to find out the capacities of the hard disk. If the hard disk is pre-formatted, this command shows the size of the hard disk, partitions, and filesystem type:

```
$fdisk -l /dev/sda
```

If the hard disk is not formatted, create the partitions on the hard disk using the following command:

```
$fdisk /dev/sda
```

After the partition, the created files resemble /dev/sda[1-4].

Block Read/Write Test: The command, dd, is used for for reading/writing blocks. Note this command can corrupt the partitions and filesystem on Hard disk.

To clear the first 5 KB of the card, do the following:

```
$dd if=/dev/zero of=/dev/sda1 bs=1024 count=5
```

The response should be as follows:

5+0 records in

5+0 records out

To write a file content to the card enter the following text, substituting the name of the file to be written for file_name, do the following:

```
$dd if=file_name of=/dev/sda1
```

To read 1KB of data from the card enter the following text, substituting the name of the file to be written for output_file, do the following:

```
$dd if=/dev/sda1 of=output_file bs=1024 count=1
```

Files System Tests

Format the hard disk partitons using mkfs.vfat or mkfs.ext2, depending on the filesystem:

```
$mkfs.ext2 /dev/sda1
$mkfs.vfat /dev/sda1
```

Mount the file system as follows:

```
$mkdir /mnt/sda1
$mount -t ext2 /dev/sda1 /mnt/sda1
```

After mounting, file/directory, operations can be performed in /mnt/sda1.

Unmount the filesystem as follows:

```
$umount /mnt/sda1
```

Chapter 36
Inter-IC (I2C) Driver

36.1 Introduction

I2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices.

The I2C driver for Linux OS has two parts:

- I2C bus driver-low level interface that is used to talk to the I2C bus
- I2C chip driver-acts as an interface between other device drivers and the I2C bus driver

36.1.1 I2C Bus Driver Overview

The I2C bus driver is invoked only by the I2C chip driver and is not exposed to the user space.

The standard Linux kernel contains a core I2C module that is used by the chip driver to access the I2C bus driver to transfer data over the I2C bus. The chip driver uses a standard kernel space API that is provided in the Linux kernel to access the core I2C module. The standard I2C kernel functions are documented in the files available under Documentation/i2c in the kernel source tree. This bus driver supports the following features:

- Compatible with the I2C bus standard
- Bit rates up to 400 Kbps
- Starts and stops signal generation/detection
- Acknowledge bit generation/detection
- Interrupt-driven, byte-by-byte data transfer
- Standard I2C master mode

36.1.2 I2C Device Driver Overview

The I2C device driver implements all the Linux I2C data structures that are required to communicate with the I2C bus driver. It exposes a custom kernel space API to the other device drivers to transfer data to the device that is connected to the I2C bus. Internally, these API functions use the standard I2C kernel space API to call the I2C core module. The I2C core module looks up the I2C bus driver and calls the appropriate function in the I2C bus driver to transfer data. This driver provides the following functions to other device drivers:

- Read function to read the device registers
- Write function to write to the device registers

The camera driver uses the APIs provided by this driver to interact with the camera.

36.1.3 Hardware Operation

The I2C module provides the functionality of a standard I2C master and slave.

It is designed to be compatible with the standard Philips I2C bus protocol. The module supports up to 64 different clock frequencies that can be programmed by setting a value to the Frequency Divider Register (IFDR). It also generates an interrupt when one of the following occurs:

- One byte transfer is completed
- Address is received that matches its own specific address in slave-receive mode
- Arbitration is lost

36.2 Software Operation

The I2C driver for Linux OS has two parts: an I2C bus driver and an I2C chip driver.

36.2.1 I2C Bus Driver Software Operation

The I2C bus driver is described by a structure called i2c_adapter. The most important field in this structure is struct i2c_algorithm *algo. This field is a pointer to the i2c_algorithm structure that describes how data is transferred over the I2C bus. The algorithm structure contains a pointer to a function that is called whenever the I2C chip driver wants to communicate with an I2C device.

During startup, the I2C bus adapter is registered with the I2C core when the driver is loaded. Certain architectures have more than one I2C module. If so, the driver registers separate i2c_adapter structures for each I2C module with the I2C core. These adapters are unregistered (removed) when the driver is unloaded.

After transmitting each packet, the I2C bus driver waits for an interrupt indicating the end of a data transmission before transmitting the next byte. It times out and returns an error if the transfer complete signal is not received. Because the I2C bus driver uses wait queues for its operation, other device drivers should be careful not to call the I2C API methods from an interrupt mode.

36.2.2 I2C Device Driver Software Operation

The I2C driver controls an individual I2C device on the I2C bus. A structure, i2c_driver, describes the I2C chip driver. The fields of interest in this structure are flags and attach_adapter. The flags field is set to a value I2C_DF_NOTIFY so that the chip driver can be notified of any new I2C devices, after the driver is loaded. When the I2C bus driver is loaded, this driver stores the i2c_adapter structure associated with this bus driver so that it can use the appropriate methods to transfer data.

36.3 Driver Features

The I2C driver supports the following features:

- I2C communication protocol
- I2C master mode of operation

NOTE

The I2C driver does not support the I2C slave mode of operation.

36.3.1 Source Code Structure

Table below shows the I2C bus driver source files available in the directory:

<Yocto_BuildDir>/drivers/i2c/busses.

Table 36-1. I2C Bus Driver Files

File	Description
i2c-imx.c	I2C bus driver source file

36.3.2 Menu Configuration Options

Configure the kernel option to enable the module by menuconfig:

Device Drivers > I2C support > I2C Hardware Bus support > IMX I2C interface.

36.3.3 Programming Interface

The I2C device driver can use the standard SMBus interface to read and write the registers of the device connected to the I2C bus.

For more information, see include/linux/i2c.h .

36.3.4 Interrupt Requirements

The I2C module generates many kinds of interrupts.

The highest interrupt rate is associated with the transfer complete interrupt as shown in table below.

Table 36-2. I2C Interrupt Requirements

Parameter	Equation	Typical	Best Case
Rate	Transfer Bit Rate/8	12,500/sec	50,000/sec
Latency	8/Transfer Bit Rate	80 us	20 us

The typical value of the transfer bit-rate is 100 Kbps. The best case values are based on a baud rate of 400 Kbps (the maximum supported by the I2C interface).

Chapter 37
Enhanced Configurable Serial Peripheral Interface (ECSPI) Driver

37.1 Introduction

The ECSPI driver implements a standard Linux driver interface to the ECSPI controllers.

It supports the following features:

- Interrupt-driven transmit/receive of bytes
- Multiple master controller interface
- Multiple slaves select
- Multi-client requests

37.1.1 Hardware Operation

ECSPI is used for fast data communication with fewer software interrupts than conventional serial communications.

Each ECSPI is equipped with a data FIFO and is a master/slave configurable serial peripheral interface module, allowing the processor to interface with external SPI master or slave devices.

The primary features of the ECSPI includes:

- Master/slave-configurable
- Four chip select signals to support multiple peripherals
- Up to 32-bit programmable data transfer
- 64 x 32-bit FIFO for both transmit and receive data
- Configurable polarity and phase of the Chip Select (SS) and SPI Clock (SCLK)

37.2 Software Operation

The following sections describe the ECSPI software operation.

37.2.1 SPI Sub-System in Linux OS

The ECSPI driver layer is located between the client layer (SPI-NOR Flash are examples of clients) and the hardware access layer. Figure below shows the block diagram for SPI subsystem in Linux OS.

The SPI requests go into I/O queues. Requests for a given SPI device are executed in FIFO order and they complete asynchronously through completion callbacks. There are also some simple synchronous wrappers for those calls including the ones for common transaction types such as writing a command and then reading its response.

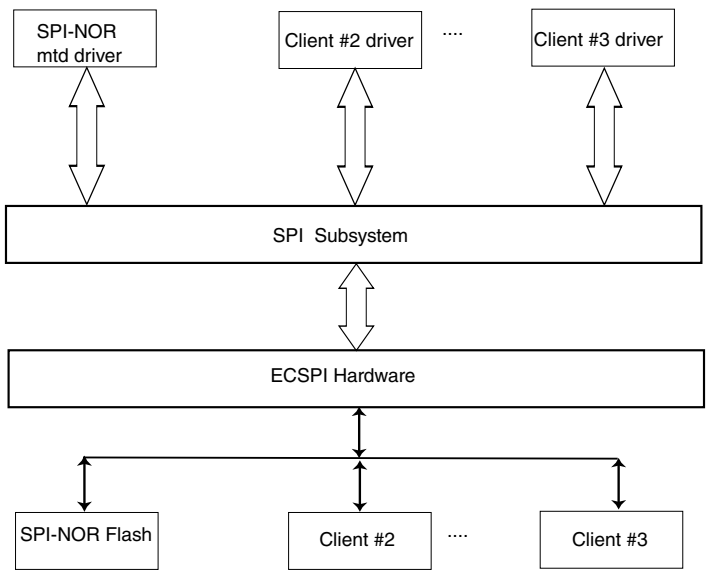


Figure 37-1. SPI Subsystem

All SPI clients must have a protocol driver associated with them and they all must be sharing the same controller driver. Only the controller driver can interact with the underlying SPI hardware module. Figure below shows how the different SPI drivers are layered in the SPI subsystem.

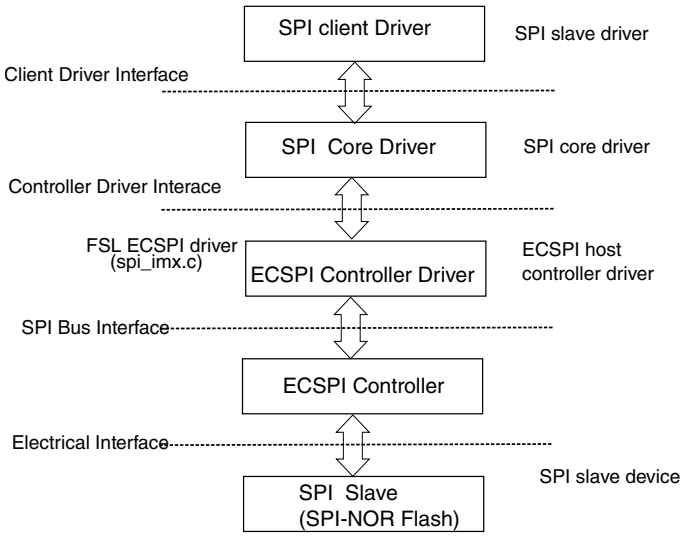


Figure 37-2. Layering of SPI Drivers in SPI Subsystem

37.2.2 Software Limitations

The ECSPI driver limitations are as follows:

- Does not currently have SPI slave logic implementation
- Does not support a single client connected to multiple masters
- Does not currently implement the user space interface with the help of the device node entry but supports sysfs interface

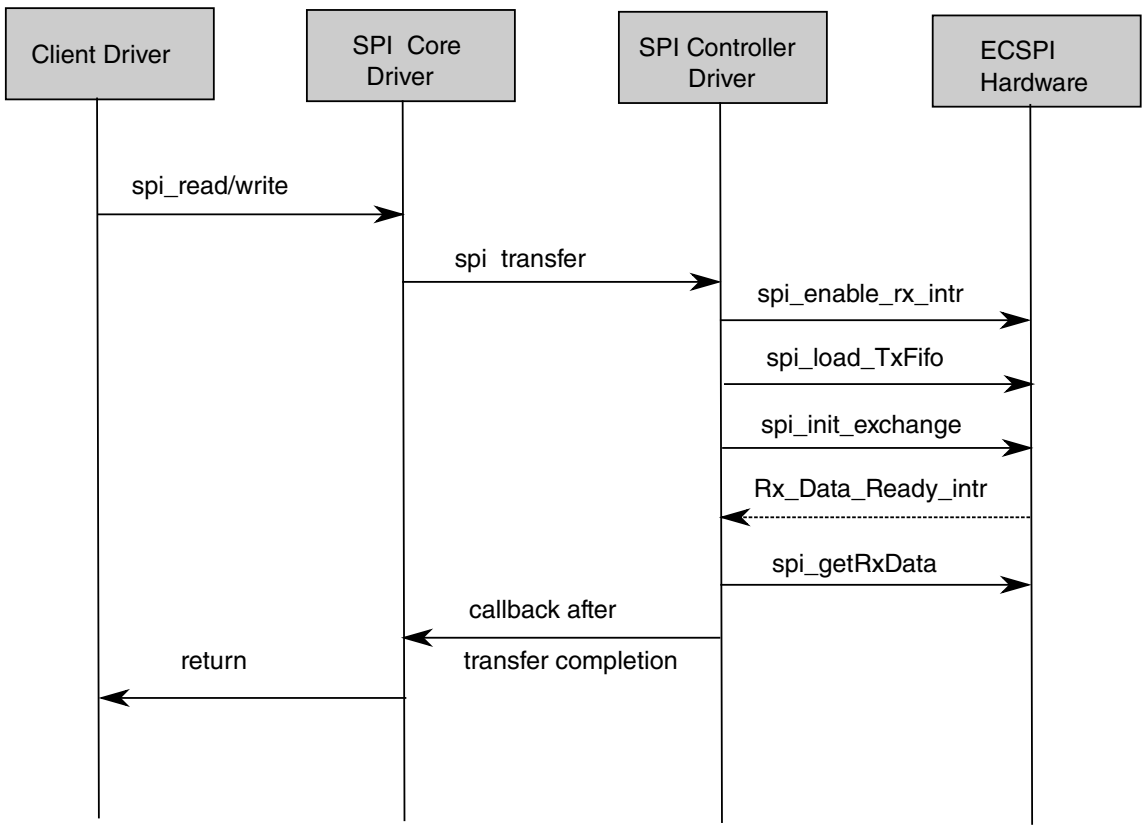
37.2.3 Standard Operations

The ECSPI driver is responsible for implementing standard entry points for init, exit, chip select, and transfer. The driver implements the following functions:

- Init function spi_imx_init() registers the device_driver structure.
- Probe function spi_imx_probe() performs initialization and registration of the SPI device-specific structure with SPI core driver. The driver probes for memory and IRQ resources. Configures the IOMUX to enable ECSPI I/O pins, requests for IRQ and resets the hardware.
- Chip select function spi_imx_chipselect() configures the hardware ECSPI for the current SPI device. Sets the word size, transfer mode, data rate for this device.
- SPI transfer function spi_imx_transfer() handles data transfers operations.
- SPI setup function spi_imx_setup() initializes the current SPI device.
- SPI driver ISR spi_imx_isr() is called when the data transfer operation is completed and an interrupt is generated.

37.2.4 ECSPI Synchronous Operation

Figure below shows how the ECSPI provides synchronous read/write operations.


Client Driver
SPI Core Driver
SPI Controller Driver
ECSPI Hardware
spi_read/write
spi transfer
spi_enable_rx_intr
spi_load_TxFifo
spi_init_exchange
Rx_Data_Ready_intr
spi_getRxData
callback after
transfer completion
return

37.3 Driver Features

The ECSPI module supports the following features:

- Implements each of the functions required by a ECSPI module to interface to Linux OS
- Multiple SPI master controllers
- Multi-client synchronous requests

37.3.1 Source Code Structure

Table below shows the source files available in the devices directory:

```
<Yocto_BuildDir>/linux/drivers/spi/
```

Table 37-1. CSPI Driver Files

File	Description
spi_imx.c	SPI Master Controller driver

37.3.2 Menu Configuration Options

To get to the Linux kernel configuration options provided for this module, use the bitbake linux-imx -c menuconfigcommand.

On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_SPI build support for the SPI core. In menuconfig, this option is available under:
  - Device Drivers > SPI Support.
- CONFIG_BITBANG is the Library code that is automatically selected by drivers that need it. SPI_IMX selects it. In menuconfig, this option is available under:
  - Device Drivers > SPI Support > Utilities for Bitbanging SPI masters.
- CONFIG_SPI_IMX implements the SPI master mode for ECSPI. In menuconfig, this option is available under:
  - Device Drivers > SPI Support > Freescale i.MX SPI controllers.

37.3.3 Programming Interface

This driver implements all the functions that are required by the SPI core to interface with the ECSPI hardware.

For more information, see the Linux document generated from build: make htmldocs.

37.3.4 Interrupt Requirements

The SPI interface generates interrupts.

ECSPI interrupt requirements are listed in table below.

Table 37-2. ECSPI Interrupt Requirements

Parameter	Equation	Typical	Worst Case
BaudRate/ Transfer Length	(BaudRate/(TransferLength)) * (1/Rxtl)	31250	1500000

The typical values are based on a baud rate of 1 Mbps with a receiver trigger level (Rxtl) of 1 and a 32-bit transfer length. The worst-case is based on a baud rate of 12 Mbps (max supported by the SPI interface) with a 8-bits transfer length.

Chapter 38
FlexCAN Driver

38.1 Driver Overview

FlexCAN is a communication controller implementing the CAN protocol according to the CAN 2.0B protocol specification.

The CAN protocol was primarily designed to be used as a vehicle serial data bus meeting the specific requirements of this field such as real-time processing, reliable operation in the EMI environment of a vehicle, cost-effectiveness, and required bandwidth. The standard and extended message frames are supported. The maximum message buffer is 64. The driver is a network device driver of PF_CAN protocol family.

For detailed information, see lwn.net/Articles/253425 or Documentation/networking/can.txt in Linux source directory.

38.1.1 Hardware Operation

For the information on hardware operations, see the following documents:

- *i.MX 6Dual/6Quad Applications Processor Reference Manual* (IMX6DQRM)
- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)
- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)
- *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM)

38.1.2 Software Operation

The CAN driver is a network device driver. For the common information on software operation, refer to the documents in the kernel source directory Documentation/ networking/can.txt.

The CAN network device driver interface.

The CAN network device driver interface provides a generic interface to setup, configure and monitor CAN network devices. The user can then configure the CAN device, like setting the bit-timing parameters, via the netlink interface using the program "ip" from the "IPROUTE2" utility suite.

Starting and stopping the CAN network device.

A CAN network device is started or stopped as usual with the command "ifconfig canX up/down" or "ip link set canX up/down". Be aware that you *must* define proper bit-timing parameters for real CAN devices before you can start it to avoid error-prone default settings:

- ip link set canX up type can bitrate 125000

The iproute2 tool also provides some other configuration capbilities for can bus such as bit-timing setting. For details, see kernel doc: Documentation/networking/can.txt

38.1.3 Source Code Structure

Table below shows the driver source file available in the directory, /linux/drivers/net/can/

Table 38-1. FlexCAN Driver Files

File	Description
flexcan.c	FlexCAN driver

38.1.4 Linux Menu Configuration Options

The following Linux kernel configuration options are provided for this module.

- CONFIG_CAN - Build support for PF_CAN protocol family. In menuconfig, this option is available under

  Networking > CAN bus subsystem support.

- CONFIG_CAN_RAW - Build support for Raw CAN protocol. In menuconfig, this option is available under

Networking > CAN bus subsystem support > Raw CAN Protocol (raw access with CAN-ID filtering).

- CONFIG_CAN_BCM - Build support for Broadcast Manager CAN protocol. In menuconfig, this option is available under

  Networking > CAN bus subsystem support > Broadcast Manager CAN Protocol (with content filtering).

- CONFIG_CAN_VCAN - Build support for Virtual Local CAN interface (also in Ethernet interface). In menuconfig, this option is available under

  Networking > CAN bus subsystem support > CAN Device Driver > Virtual Local CAN Interface (vcan).

- CONFIG_CAN_DEBUG_DEVICES - Build support to produce debug messages to the system log to the driver. In menuconfig, this option is available under

  Networking > CAN bus subsystem support > CAN Device Driver > CAN devices debugging messages.

- CONFIG_CAN_FLEXCAN - Build support for FlexCAN device driver. In menuconfig, this option is available under

  Networking > CAN bus subsystem support > CAN Device Driver > Freescale FlexCAN.

Chapter 39
Media Local Bus Driver

39.1 Introduction

MediaLB is an on-PCB or inter-chip communication bus specifically designed to standardize a common hardware interface and software API library.

This standardization allows an application or multiple applications to access the MOST Network data or to communicate with other applications with minimum effort. MediaLB supports all the *MOST Network data transport* methods: synchronous stream data, asynchronous packet data, and control message data. MediaLB also supports an isochronous data transport method. For detailed information about the MediaLB, see the Media Local Bus Specification.

39.1.1 MLB Device Module

The MediaLB module implements the Physical Layer and Link Layer of the MediaLB specification, interfacing the i.MX to the MediaLB controller.

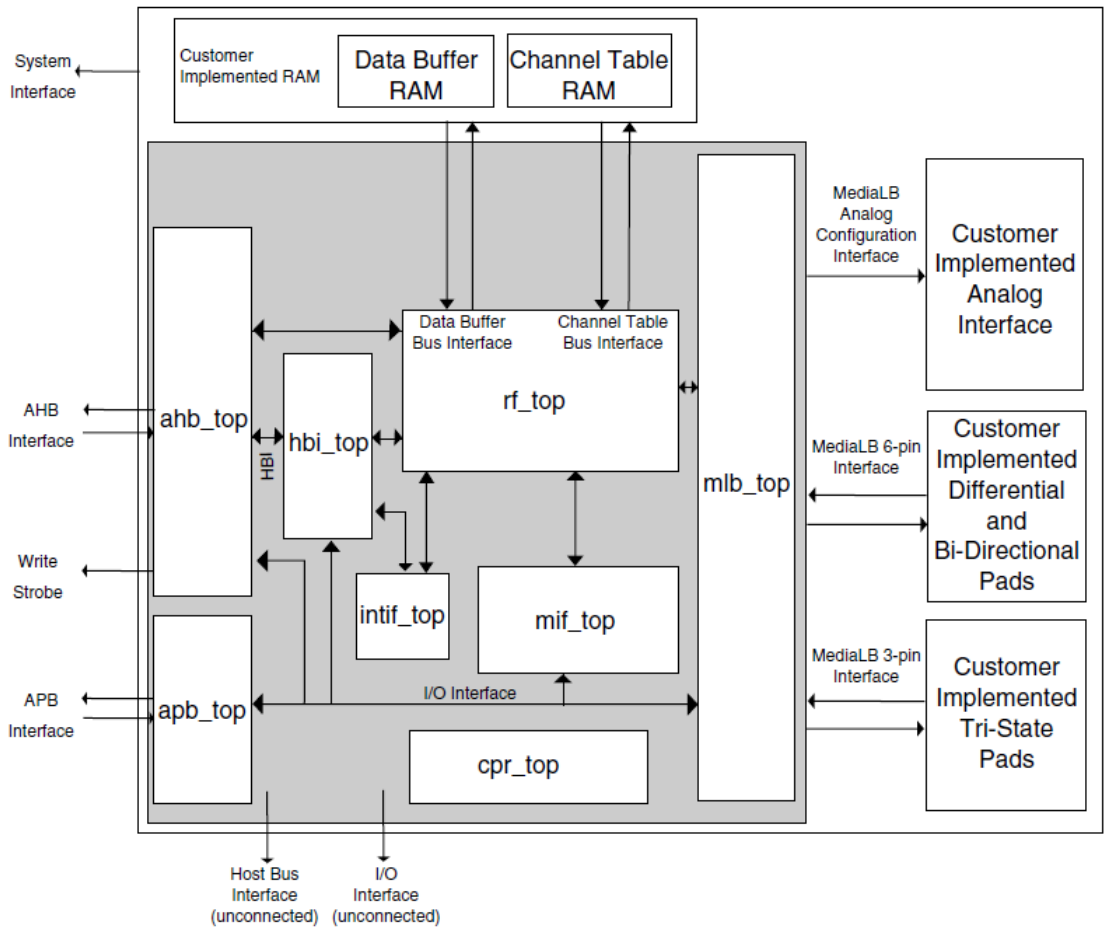


Figure 39-1. MLB Device Top-Level Block Diagram

The MLB implements the 3-pin MediaLB mode and can run at speeds up to 1024Fs. It does not implement MediaLB controller functionality. All MediaLB devices support a set of physical channels for sending data over the MediaLB. Each physical channel is 4 bytes in length (quadlet) and grouped into logical channels with one or more physical channels allocated to each logical channel. These logical channels can be any combination of channel type (synchronous, asynchronous, control, or isochronous) and direction (transmit or receive).

The MLB provides support for up to 64 logical channels and up to 64 physical channels. Each logical channel is referenced using an unique channel address and represents a unidirectional data path between a MediaLB device transmitting the data and the MediaLB device(s) receiving the data.

39.1.2 Supported Features

- Synchronous, asynchronous, control and isochronous channel.

- Up to 64 logical channels and 64 physical channels running at a maximum speed of 1024Fs
- Transmission of commands and data and reception of receive status when functioning as the transmitting device associated with a logical channel address
- Reception of commands and data and transmission as receive status responses when functioning as the receiving device associated with a logical channel address
- MediaLB lock detection
- System channel command handling
- 256Fs, 512Fs and 1024Fs frame rates.
- Asynchronous, control, synchronous, and isochronous channel types.
- The following configurations to MLB device module:
  - Frame rate
  - Device address
  - Channel address
- MLB channel exception get interface. All the channel exceptions are sent and handled by the application.

39.1.3 MLB Driver Overview

The MLB driver is designed as a common Linux OS character driver. It implements one asynchronous and one control channel device with Ping-Pong buffering operation mode. The supported frame rates are 256, 512, and 1024Fs. The MLB driver uses common read/write interfaces to receive/send packets and uses the ioctl interface to configure the MLB device module.

39.2 MLB Driver

Functionality of the MLB driver is described in supported features, MLB driver architecture, and software operation.

39.2.1 MLB Driver Architecture

The MLB driver is a common Linux character driver and the architecture is shown in figure below.

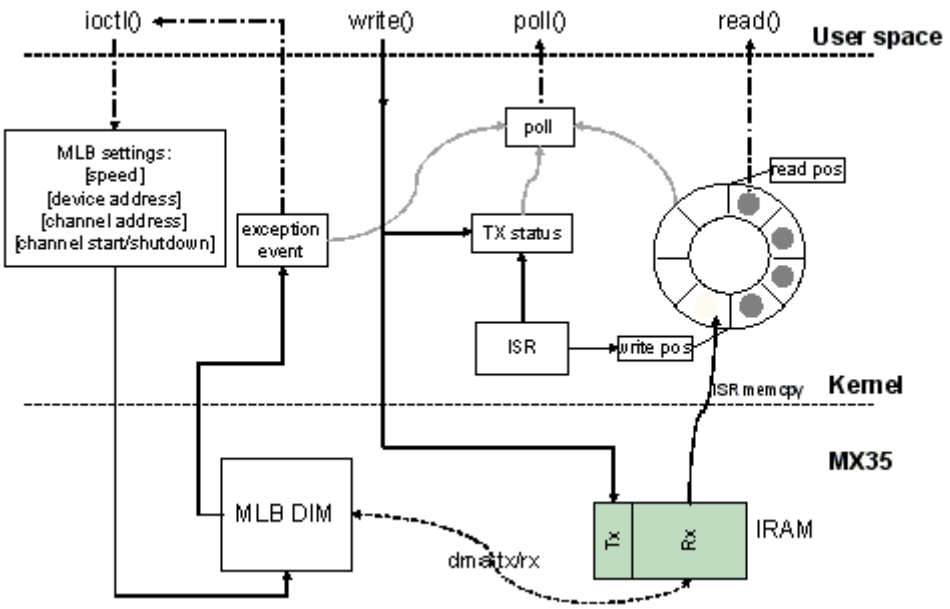


Figure 39-2. MLB Driver Architecture Diagram

The MLB driver creates four minor devices. These four devices support control Tx/Rx channel, asynchronous Tx/Rx channel, synchronous Tx/Rx channel, and isochronous Tx/Rx channel. Their device files are /dev/ctrl, /dev/async, /dev/sync, and /dev/isoc. Each minor device has the same interfaces, and handle both Tx and Rx operation. The following description is for both control and asynchronous device.

The driver uses IRAM as MLB device module Tx/Rx buffer. All the data transmission and reception between module and IRAM is handled by the MLB module DMA. The driver is responsible for configuring the buffer start and end pointer for the MLB module.

For reception, the driver uses a ring buffer to buffer the received packet for read. When a packet arrives, the MLB module puts the received packet into the IRAM Rx buffer, and notifies the driver by interrupt. The driver then copy the packet from the IRAM to one ring buffer node indicated by the write position, and updates the write position with the next empty node. Finally the packet reader application is notified, and it gets one packet from the node indicated by the read position of ring buffer. After the read completed, it updates the read position with the next available buffer node. There is no received packet in the ring buffer when the read and write position is the same.

For transmission, the driver writes the packet given by the writer application into the IRAM Tx buffer, updates the Tx status and sets MLB device module Tx buffer pointer to start transmission. After transmission completes, the driver is notified by interrupt and updates the Tx status to accept the next packet from the application.

The driver supports NON BLOCK I/O. User applications can poll to check if there are packets or exception events to read, and also they can check if a packet can be sent or not. If there are exception events, the application can call ioctl to get the event. The ioctl also provides the interface to configure the frame rate, device address and channel address.

39.2.2 Software Operation

The MLB driver provides a common interface to application.

- Packet read/write-BLOCK and NONBLOCK Packet I/O modes are supported. Only one packet can be read or written at once. The minimum read length must be greater or equal to the received packet length, meanwhile the write length must be shorter than 1024Bytes.
- Polling-The MLB driver provide polling interface which polls for three status, application can use select to get current I/O status:
  - Packet available for read (ready to read)
  - Driver is ready to send next packet (ready to write)
  - Exception event comes (ready to read)
- ioctl-MLB driver provides the following ioctl:

`MLB_SET_FPS`

Argument type: unsigned int

Set frame rate, the argument must be 256, 512 or 1024.

`MLB_GET_VER`

Argument type: unsigned long

Get MLB device module version, which is 0x02000202 by default on the i.MX35.

`MLB_SET_DEVADDR`

Argument type: unsigned char

Set MLB device address, which is used by the system channel MlbScan command.

`MLB_CHAN_SETADDR`

Argument type: unsigned int

Set the corresponding channel address [8:1] bits. This ioctl combines both tx and rx channel address, the argument format is: tx_ca[8:1] << 16 | rx_ca[8:1]

`MLB_CHAN_STARTUP`

Startup the corresponding type of channel for transmit and reception.

`MLB_CHAN_SHUTDOWN`

Shutdown the corresponding type of channel.

`MLB_CHAN_GETEVENT`

Argument type: unsigned long

Get exception event from MLB device module, the event is defined as a set of enumeration:

```
MLB_EVT_TX_PROTO_ERR_CUR
MLB_EVT_TX_BRK_DETECT_CUR
MLB_EVT_RX_PROTO_ERR_CUR
MLB_EVT_RX_BRK_DETECT_CUR
```

39.3 Driver Files

Table below lists the source file associated with the MLB driver that are found in the directory drivers/mxc/mlb/.

Table 39-1. MLB Driver Source File List

File	Description
mxc_mlb150.c	Source file for MLB driver
include/linux/mxc_mlb.h	Include file for MLB driver

39.4 Menu Configuration Options

To get to the MediaLB configuration, use the command bitbake linux-imx -c menuconfig. On the screen, select **Configure Kernel**, exit, and a new screen appears. This option is available under:

- Device Drivers > MXC support drivers > MXC Media Local Bus Driver > MLB support.

Chapter 40
CHIPIDEA USB Driver

40.1 Introduction

The universal serial bus (USB) driver implements a standard Linux driver interface to the CHIPIDEA USB-HS OTG controller.

The USB provides a universal link that can be used across a wide range of PC-to-peripheral interconnects. It supports plug-and-play, port expansion, and any new USB peripheral that uses the same type of port.

The CHIPIDEA USB controller is Enhanced Host Controller Interface (EHCI)-compliant. This USB driver has the following features:

- high-speed OTG core supported
- high-speed Host Only core (Host1), high-speed, full speed, and low devices are supported.
- high-speed Inter-Chip core (Host2 & Host3)
- high-speed Host Only core (OTG2), high-speed, full speed, and low devices are supported. A USB2Pci bridge is connected to OTG2 by default. Therefore, User may not be able to connect other USB devices on this port.
- high-speed Inter-Chip core (Host2)
- Host mode-Supports HID (Human Interface Devices), MSC (Mass Storage Class)
- Peripheral mode-Supports MSC, and CDC (Communication Devices Class) drivers which include ethernet and serial support
- Embedded DMA controller

40.1.1 Architectural Overview

The USB host system is composed of a number of hardware and software layers.

Figure below shows a conceptual block diagram of the building block layers in a host system that support USB 2.0.

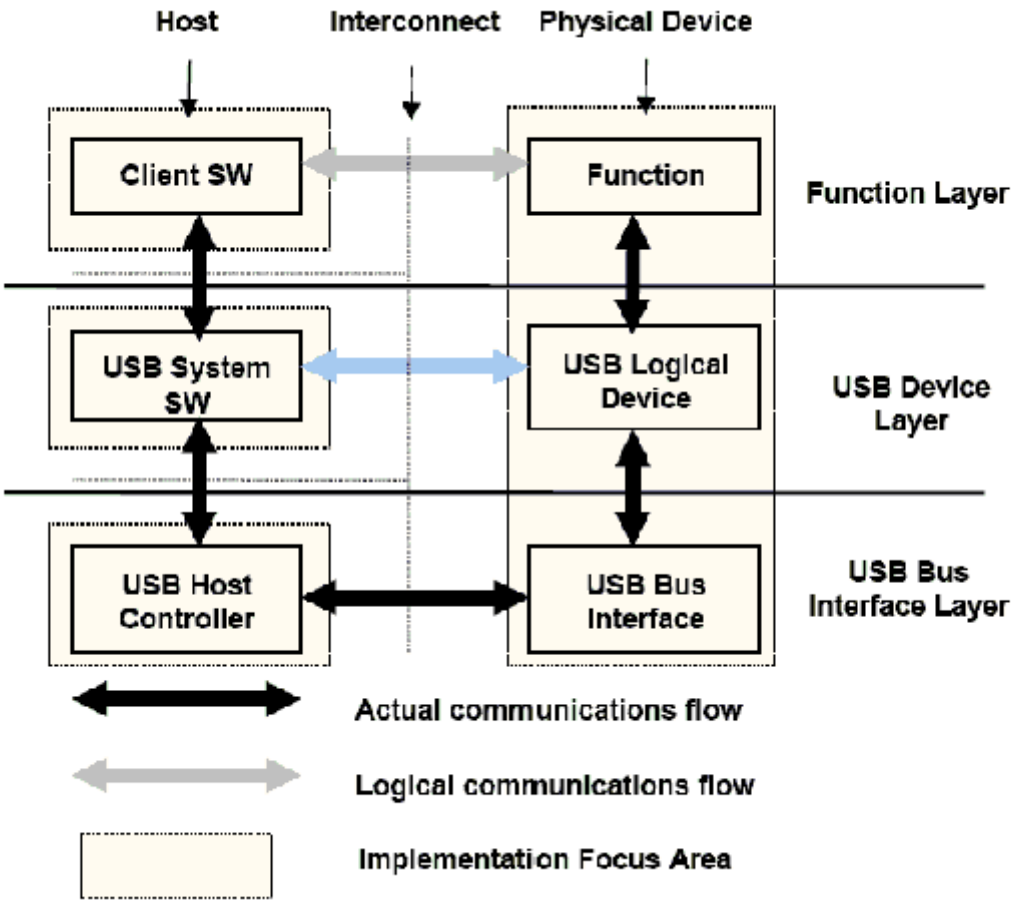


Figure 40-1. USB Block Diagram

40.2 Hardware Operation

For information on hardware operations, refer to the EHCI spec.ehci-r10.pdf.

The spec is available at Enhanced Host Controller Interface for USB 2.0: Specification

40.2.1 Software Operation

The Linux OS contains a USB driver, which implements the USB protocols. For the USB host, it only implements the hardware specified initialization functions. For the USB peripheral, it implements the gadget framework. For OTG, ID dynamic switch host/ device modes are supported. In addition, the OTG HNP and SRP functions are already supported. Currently, the runtime suspend for USB is supported, that is to say when the USB is not in use (both for host and peripheral mode), the USB will enter low power mode.

40.2.2 Source Code Structure

The table below shows the source files available in the source directory, $KERNEL/drivers/usb/

Table 40-1. USB Driver Files

File	Description
chipidea/core.c	Chipidea IP core driver
chipidea/udc.c	Chipidea peripheral driver
chipidea/host.c	Chipidea host driver
chipidea/otg.c	Chipidea OTG driver
chipidea/otg_fsm.c	Chipidea OTG HNP and SRP driver
chipidea/ci_hdrc_imx.c	i.MX glue layer
chipidea/usbmisc_imx.c	i.MX SoC abstract layer
phy/phy-mxs-usb.c	i.MX 6 USB physical driver

40.2.3 Menu Configuration Options

1. CONFIG_USB-Build Support for Host-side USB
2. CONFIG_USB_EHCI_HCD EHCI HCD (USB 2.0) support

   Default y

3. CONFIG_USB_CHIPIDEA- ChipIdea high-speed Dual Role Controller

   Default y

4. CONFIG_USB_CHIPIDEA_UDC - ChipIdea device controller

   Default y

5. CONFIG_USB_CHIPIDEA_HOST - ChipIdea host controller

   Default y

6. CONFIG_USB_GADGET - USB Gadget Support

   Default y

7. CONFIG_USB_MXS_PHY - Freescale MXS USB PHY support

   Default y

40.2.4 USB Wakeup Usage

The following example is for the OTG port and the first EHCI device.

Controller wakeup setting, after the following settings, the vbus and ID will be wakeup source.

```
echo enabled > /sys/bus/platform/devices/20c9000.usbphy/power/wakeup
echo enabled > /sys/bus/platform/devices/2184000.usb/power/wakeup
echo enabled > /sys/bus/platform/devices/ci_hdrc.0/power/wakeup
```

EHCI wakeup setting, after the following settings, the host will have wakeup ability, such as remote wakeup and connect/disconnect wakeup

```
echo enabled > /sys/bus/usb/devices/usb1/power/wakeup
echo enabled > /sys/bus/usb/devices/1-1/power/wakeup
```

NOTE

When the OTG mode switches from the host to the device, it will delete the EHCI wakeup, and the user needs to set it again before the system suspending.

40.2.5 How to Close the USB Child Device Power

The following code string outlines how to close the USB child device power:

```
echo auto > /sys/bus/usb/devices/1-1/power/control
echo auto > /sys/bus/usb/devices/1-1.1/power/control (If there is a hub at USB device)
```

40.2.6 Changing the Controller Operation Mode

To change the default settings, the use can modify the DTS file as follows:

```
dr_mode =  "host"  /* Set controller as gadget-only mode */
dr_mode =  "peripheral" /* Set controller as host-only mode */
dr_mode =  "otg" /* Set controller as otg mode */
```

40.2.7 Loadable Module Support

The kernel configuration is as follows:

```
   Device Drivers  --->
    [*] USB support  --->
        <M>     EHCI HCD (USB 2.0) support
        <M>   ChipIdea Highspeed Dual Role Controller
[*]   USB Physical Layer drivers  --->
<M>   Freescale MXS USB PHY support
<M>   USB Gadget Support  --->
```

The modprobe utility will automatically load the modules which have dependency among all modules.

The loading command is as follows:

```
modprobe phy_mxs_usb
modprobe ci_hdrc_imx
```

The unloading command is as follows:

```
modprobe -r ci_hdrc_imx
modprobe -r phy_mxs_usb
```

40.2.8 USB Charger Detection

i.MX SoC has USB charger detection ability, but it has no charging ability. The user can use the `/sys` entry to know the USB charger type, charging current, and whether the charger exists (see below three entries).

```
cat /sys/class/power_supply/imx6_usb_charger/type
cat /sys/class/power_supply/imx6_usb_charger/current_max
cat /sys/class/power_supply/imx6_usb_charger/present
```

Currently, the i.MX 6 Sabre-SD board does not support the USB charger detection function. i.MX 6 Sabre-Auto and i.MX 6SoloLite EVK support the function.

40.2.9 USB OTG HNP and SRP Support

i.MX SoC and the driver can support OTG HNP (Host Negotiation Protocol) and SRP (Session Request Protocol) according to "On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification July 27, 2012, Revision 2.0, Version 1.1a", which is not enabled by default. To enable this, add otg-rev property, remove hnp-disable and srp-disable in DTS as follows:

```
otg-rev = <0x0200>;
adp-disable;
```

All the required drivers should be built into the kernel, including the gadget driver. For example, if you want to enable mass storage as gadget driver:

Choose ‘Y’ at Kernel Configuration:

```
   Device Drivers  --->
    [*] USB support  --->
        <*>   USB Gadget Support  --->
            <*>   USB Gadget Drivers (Mass Storage Gadget)  --->
```

Add some module parameters for g_mass_storage at U-Boot bootargs.

```
g_mass_storage.removable=1
g_mass_storage.idVendor=0x15a2
g_mass_storage.idProduct=0x7b
g_mass_storage.iSerialNumber=123456abcdef
g_mass_storage.luns=1
```

Add the back file for g_mass_storage after the system boots up:

echo "/dev/mmcblk3p1" > /sys/bus/platform/devices/2184000.usb/ci_hdrc.0/gadget/lun0/file

NOTE

Yocto rootfs has some limitations. The back file assignment will not be effected if the mass storage gadget has already been recognized. The Windows® OS 7 and Ubuntu do not have this issue. To solve this limitation:

- Do not connect the USB cable before back file assignment.
- Disconnect and reconnect the USB cable between devices A and B.

The HNP and SRP have been verified with two i.MX 6 reference boards. For details on how to demo them, see the document in the Linux kernel source:

.Documentation/usb/chipidea.txt

NOTE

For all i.MX 6 series, if you want to support OTG SRP on one OTG port (e.g., usbotg1), the VBUS of another port with internal PHY (for usbotg2 or host1) should be provided at all times.

This can be achieved by keeping the second port vbus always on, but for the i.MX 6UltraLite EVK board, the vbus control GPIO is multiplexed with another module, so it cannot be done by this work around. OTG SRP cannot be supported, and you need to change the hardware design on this point to make one of the internal USB PHY power supplies active.

40.2.10 Embeded Host Certification

40.2.10.1 Adding TPL-Support Property

To pass embeded host USB certification, "tpl-support" should be added in DTS to enable Targeted Peripheral List (TPL). For example, to enable TPL on the Host port of i.MX 6UltraLite EVK board (imx6ul-14x14-evk.dts):

```
&usbotg2 {
        dr_mode = "host";
        disable-over-current;
        tpl-support;
        status = "okay";
};
```

40.2.10.2 VBUS Control

The VBUS should be kept off until the Linux USB host function is ready. For example, on the i.MX 6UltraLite EVK board, because the pin is multiplexed with the touch function, you need to rework the board to make the GPIO (GPIO1_IO02) selected for VBUScontrol.

Disable the touch function in its DTS file (imx6ul-14x14-evk.dts) as follows:

```
&tsc {
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_tsc>;
        xnur-gpio = <&gpio1 3 0>;
        measure_delay_time = <0xffff>;
        pre_charge_time = <0xfff>;
        status = "disabled";
};
```

Add VBUS GPIO pinctrl and its regulator node:

```
pinctrl_usb_otg2: usbotg2grp {
            fsl,pins = <
                MX6UL_PAD_GPIO1_IO02__GPIO1_IO02        0xb0
            >;
  };

reg_usb_otg2_vbus: regulator@2 {
            compatible = "regulator-fixed";
            reg = <2>;
            pinctrl-names = "default";
            pinctrl-0 = <&pinctrl_usb_otg2>;
            regulator-name = "usb_otg2_vbus";
            regulator-min-microvolt = <5000000>;
            regulator-max-microvolt = <5000000>;
            gpio = <&gpio1 2 GPIO_ACTIVE_HIGH>;
            enable-active-high;
};

&usbotg2 {
```

```
    vbus-supply = <&reg_usb_otg2_vbus>;
    dr_mode = "host";
    disable-over-current;
    tpl-support;
    status = "okay";
};
```

Chapter 41
i.MX 6 PCI Express Root Complex Driver

41.1 Introduction

PCI Express hardware module, contained in i.MX SoC, can either be configured to act as a Root Complex or a PCIe Endpoint.

This document is used to describe the PCI Express Root Complex implementation on i.MX SoC families.

It also describes the drivers needed to be configured and operated on i.MX PCI Express device as Root Complex.

41.1.1 PCIe

PCI Express (PCIe) is Third Generation I/O Interconnect, targeting low cost, high volume, multi-platform interconnection usages. It has the concepts with earlier PCI and PCI-X and offers backwards compatibility for existing PCI software with following differences:

- PCIe is a point-to-point interconnect
- Serial link between devices
- Packet based communication
- Scalable performance via aggregated Lanes from X1 to X16
- Need PCIe switch to have connection between more than two PCIe devices

41.1.2 Terminology and Conventions

Following terminologies and conventions are used in this document:

- Bridge

A Function that virtually or actually connects a PCI/PCI-X segment or PCI Express Port with an internal component interconnect or with another PCI/PCI-X bus segment or PCI Express Port.

- Downstream
  - 1. The relative position of an interconnect/System Element (Port/component) that is farther from the Root Complex. The Ports on a Switch that are not the Upstream Port are Downstream Ports. All Ports on a Root Complex are Downstream Ports. The Downstream component on a Link is the component farther from the Root Complex.
  - 2. A direction of information flow where the information is flowing away from the Root Complex.
- Endpoint

  One of several defined System Elements. A Function that has a Type 00h Configuration Space header.

- Host

  The entity comprising of one (or more) Central Processing Unit(s) (CPU) and resources, such as Memory (RAM) that can be shared across multiple PCIe nodes connected through a Root Complex.

- Lane

  A set of differential signal pairs, one pair for transmission and one pair for reception.

- Link

  The collection of two Ports and their interconnecting Lanes. A Link is a dual simplex communications path between two components.

- PCIe Fabric

  A topology comprised of various PCI Express nodes, also referred as devices. A device in the fabric can be Root Complex, Endpoint, PCIe-PCI/PCI-X Bridge or a Switch.

- Port
  - 1. Logically, an interface between a component and a PCI Express Link.
  - 2. Physically, a group of Transmitters and Receivers located on the same chip that define a Link.
- Root Complex

  RC A defined System Element that includes a Host Bridge, zero or more Root Complex Integrated Endpoints, zero or more Root Complex Event Collectors, and one or more Root Ports

- Root Port

  A PCI Express Port on a Root Complex that maps a portion of the Hierarchy through an associated virtual PCI-PCI Bridge.

- Upstream
  - 1. The relative position of an interconnect/System Element (Port/component) that is closer to the Root Complex. The Port on a Switch that is closest topologically to the Root Complex is the Upstream Port. The Port on a component that contains only Endpoint or Bridge Functions is an Upstream Port. The Upstream component on a Link is the component closer to the Root Complex.

Any element of the fabric which is relatively closer towards RC is treated as 'Upstream'. All PCIe Endpoint ports (including termination points for bridges) and Switch ports, which are closer to RC are called Upstream Ports on that device. A Upstream Flow is the communication moving towards RC.

41.1.3 PCIe Topology on i.MX

There is one PCIe port on the i.MX. Up to now, only the RC mode is enabled in the Linux BSP.

The following figure describes the diagram of the PCIe RC port on i.MX.

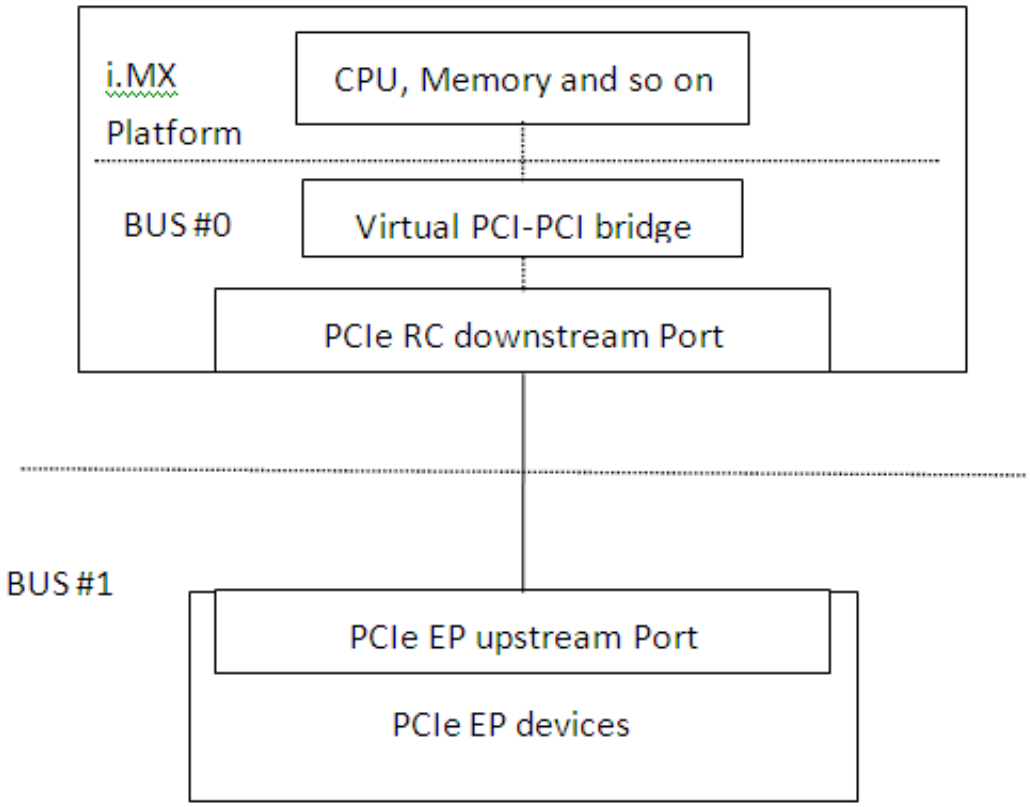


Figure 41-1. diagram of the PCIe RC port on i.MX

PCI Enumeration Mapping

Since PCI Express is point to point topology, to maintain compatibility with legacy PCI Bus - Device notion used for Software Enumeration, we introduce following concepts which allow identifying various nodes and their internals (e.g., PCIe Switches) in terms of PCI devices/functions:

- Host Bridge: A bridge, integrated into RC to have PCI compatible connection to Host. The PCI side of this bridge is Bus #0 always. This means, the device on this bus will be the host itself.
- Virtual PCI-PCI Bridge: Each PCI Express port which is part of RC or a Switch is treated as a virtual PCI-PCI bridge. This means each port has a primary and secondary PCI bus and the downstream is mapped into the remote configuration space.
- Root port associated virtual bridge has Bus #0 on the primary side with secondary bus on the downstream.
- Each PCIe Switch is viewed as collection of as many virtual PCI-PCI bridges as number of downstream ports, connected to a virtual PCI bus which is actually secondary bus of another PCI-PCI bridge forming the upstream port of the switch.
- The upstream port of each EP can either be part of the secondary bus segment of virtual PCI-PCI Bridge representing downstream port of a switch or of the root port.

41.1.4 Features

Listed below are the various features supported by i.MX as a PCI Express Root Complex driver.

- Express Base Specification Revision 2.0-compliant.
- Gen2 operation with x1 link supporting 5 GT/s raw transfer rate in single direction.
- Support Legacy Interrupts (INTx) and MSI.
- Max_Payload_Size size (128 bytes).
- It fits into Linux PCI Bus framework to provide PCI compatible software enumeration support
- In addition, it provides interface to Endpoint Drivers to access the respective devices detected downstream.
- The same interface can be used by the PCI Express Port Bus Driver framework in Linux OS to handle AER, ASP etc.
- Interrupt handling facility for EP drivers either as Legacy Interrupts (INTx).
- Access to EP I/O BARs through generic I/O accessories in Linux PCI subsystem.
- Seamless handling of PCIe errors.

41.2 Linux OS PCI Subsystem and RC driver

In Linux OS, the PCI implementation can roughly be divided into following main components: PCI BIOS architecture-specific Linux OS implementation, Host Controller (RC) Module, and Core.

- PCI BIOS Architecture-specific Linux OS implementation to kick off PCI bus initialization. It interfaces with PCI Host Controller code as well as the PCI Core to perform bus enumeration and allocation of resources such as memory and interrupts. The successful completion of BIOS execution assures that all the PCI devices in the system are assigned parts of available PCI resources and their respective drivers (referred as Slave Drivers). PCI can take control of them using the facilities provided by PCI Core. It is possible to skip resource allocation (if they were assigned before Linux OS was booted, for example PC scenario).
- Host Controller (RC) Module handles hardware (SoC + Board) specific initialization and configuration and it invokes PCI BIOS. It should provide callback functions for BIOS as well as PCI Core, which will be called during PCI system initialization and accessing PCI bus for configuration cycles. It provides resources information for available memory/IO space, INTx interrupt lines, MSI. It should also facilitate IO

space access (as supported) through in _x_ () out _x_ () You may need to provide indirect memory access (if supported by h/w) through read _x_ () write _x_ ()
- Core creates and initializes the data structure tree for bus devices as well as bridges in the system, handles bus/device numberings, creates device entries and proc/sysfs information, provides services for BIOS and slave drivers and provides hot plug support (optional/as supported by h/w). It targets (EP) driver interface query and initializes corresponding devices found during enumeration. It also provides MSI interrupt handling framework and PCI express port bus support. It provides Hot-Plug support (if supported), advanced error reporting support, power management event support, and virtual Channel support to run on PCI express ports (if supported).

41.2.1 RC Driver Source Files

The driver files are present at the following path relative to extracted kernel source directory.

drivers/pci/host/pci-imx6.c

41.2.2 Kernel Configurations

Root Complex is not supported by the default kernel configurations on i.MX boards.

To set the default configuration, execute the following command as follows:

```
make CROSS_COMPILE=arm-none-linux-gnueabi-ARCH=arm imx_v7_defconfig
```

Configure the Root Complex to be built in:

```
#
# Bus support
#
CONFIG_PCI=y
CONFIG_PCI_DOMAINS=y
CONFIG_PCI_SYSCALL=y
CONFIG_PCI_MSI=y

#
# PCI host controller drivers
#
CONFIG_PCIE_DW=y
CONFIG_PCI_IMX6=y
```

NOTE

PCI Express support can't be built as a module.

41.3 System Resource: Memory Layout

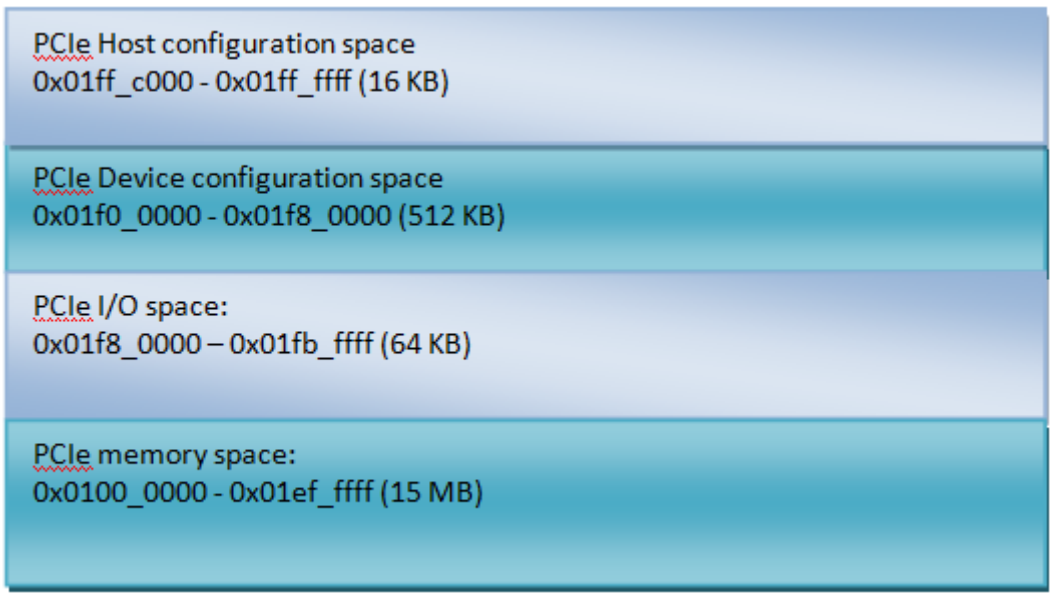


Figure 41-2. Memory Layout (i.MX 6Quad/6DualLite/6Solo)

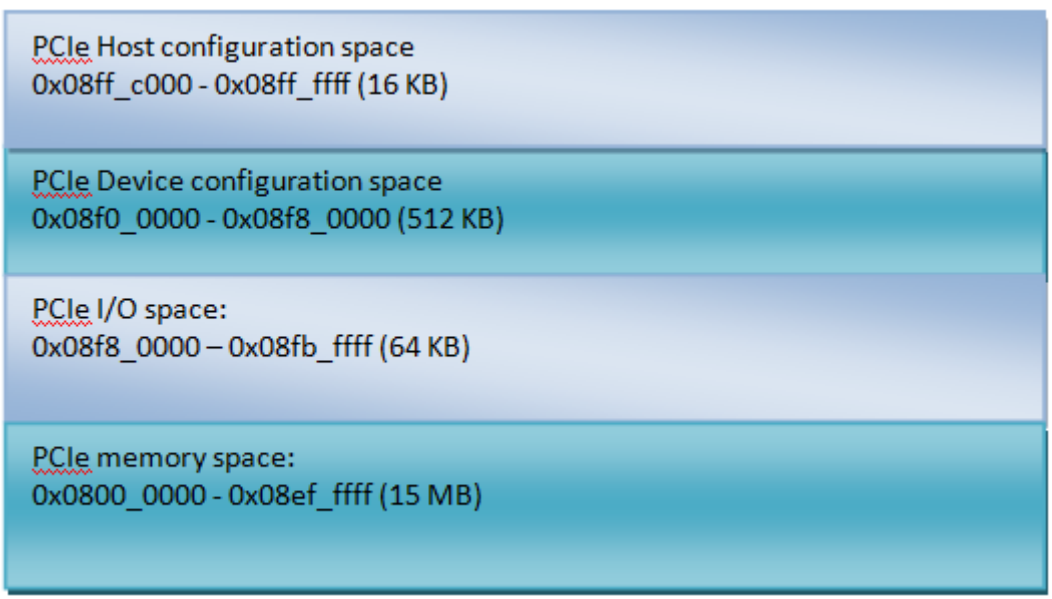


Figure 41-3. Memory Layout (i.MX 6SoloX)

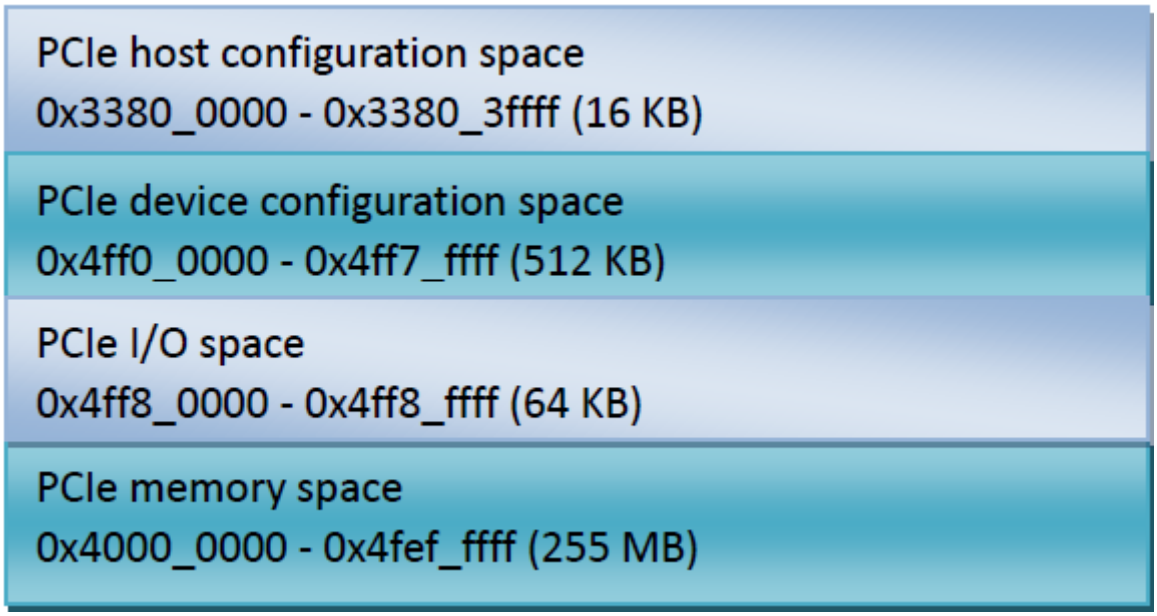


Figure 41-4. Memory Layout (i.MX 7Dual)

- IO and memory spaces are two address spaces used by the devices to communicate with their device driver running in the Linux kernel on CPU.
- The upper 16Kbytes PCIe host configuration space.
  - This memory segment is used to map the configuration space of PCIe RC. SW can access PCIe RC core configuration space through the DBI interface.
- PCIe device configuration space.
  - Used to map the configuration spaces of PCIe EP devices that are inserted to the RC downstream port.

41.3.1 System Resource: Interrupt lines

i.MX Root Complex driver uses interrupt line 152 for MSI INT on i.MX 6 platforms, and 154 for MSI INT on i.MX 7Dual platforms.

41.4 Using PCIe Endpoint and Running Tests

Perform the following steps to use PCIe endpoint and run tests:

Configure the driver according to PCIe Endpoint device.

Run "make menuconfig" after run "make ARCH=arm imx_v7_defconfig".

```
Freescale i.MX6 PCIe controller
```

```
          -> Bus support
              -> PCI host controller drivers
```

Implement the following configurations according to the PCIe EP devices:

- PCIe to USB card driver

```
    Symbol: USB_XHCI_HCD [=y]
    Type  : tristate
    Prompt: xHCI HCD (USB 3.0) support (EXPERIMENTAL)
    Defined at drivers/usb/host/Kconfig:20
    Depends on: USB_SUPPORT [=y] && USB [=y] && PCI [=y] && EXPERIMENTAL [=y]
      Location:
          -> Device Drivers
              -> USB support (USB_SUPPORT [=y])
```

- Intel CT gigabit network card driver

```
    Symbol: E1000E
[=y]

    Type  :
tristate

    Prompt: Intel(R) PRO/1000 PCI-Express Gigabit Ethernet
support

Location:

        -> Device
Drivers

          -> Network device support (NETDEVICES
[=y])
            -> Ethernet driver support (ETHERNET
[=y])
              -> Intel devices (NET_VENDOR_INTEL [=y])
```

- Intel iwl4965 or iwl6300 card driver

```
    Symbol: IWL4965
[=y]

    Type  :
tristate

    Prompt: Intel Wireless Wi-Fi 4965AGN
(iwl4965)


Location:

        -> Device
Drivers

          -> Network device support (NETDEVICES
[=y])
            -> Wireless LAN (WLAN [=y])
```

To enable the Wi-Fi driver, we need to enable one of the two options: IWL4965 or IWLAGN. Choose one, but not both.

```
CONFIG_IWLAGN:


Select to build the driver supporting the:
Intel Wireless WiFi Link Next-Gen AGN
```

This option enables support with the following hardware:

```
Intel Wireless WiFi Link 6250AGN Adapter
Intel 6000 Series Wi-Fi Adapters (6200AGN and 6300AGN)
Intel WiFi Link 1000BGN
Intel Wireless WiFi 5150AGN
Intel Wireless WiFi 5100AGN, 5300AGN, and 5350AGN
Intel 6005 Series Wi-Fi Adapters
Intel 6030 Series Wi-Fi Adapters
Intel Wireless WiFi Link 6150BGN 2 Adapter
Intel 100 Series Wi-Fi Adapters (100BGN and 130BGN)
Intel 2000 Series Wi-Fi Adapters
```

- Wi-Fi firmware configurations:

In order to install the mandatory required firmware by Intel IWL Wi-Fi devices, see the following link for guidance intellinuxwireless.org/?n=Info

41.4.1 Ensuring PCIe System Initialization

Run 'lspci' after login the consol. There should be the following similar message if the PCIe link is established.

root@freescale ~$ lspci

00:00.0 PCI bridge: Unknown device 16c3:abcd (rev 01)

01:00.0 Network controller: Intel Corporation Unknown device 4237

41.4.2 Tests

Run different tests according the different PCIe EP devices.

- Intel Iwl6300 mini-PCIe x1 WIFI card
  - Iperf, netperf
  - Overnight different packet ping
- Intel CT gigabit standard PCIe X1 network card
  - NFS mount/data IO through NFS
  - Iperf, netperf
  - Overnight different packet ping
- PCIe to USB3.0 standard PCIe X1 card
  - General tests
    - * Block storage device, recognization,

    - * Partition creation, format and so on.
    - * Hundreds MB data read/write by copy command
  - Stress tests
    - `./iozone -a -n 2000m -g 2000m -i 0 -i 1 -f /mnt/src/iozone.tmpfile -Rb ./iozone`

41.4.3 Known issues

- Connect an external Wi-Fi antenna to enlarge the Wi-Fi signal strength if the Wi-Fi card tests cannot work well.

Chapter 42
EIM NOR Driver

42.1 Introduction

The Wireless External Interface Module (WEIM) NOR driver supports the Parallel NOR flash.

42.2 Hardware Operation

By default, there is a parallel NOR in the i.MX 6Quad/6Dual SABRE-AI boards. The parallel NOR has more pins than the SPI NOR. On some boards, the M29W256GL7AN6E is equipped. Refer to the datasheet for details on the parallel NOR.

42.3 Software Operation

Similar to the SPI NOR, the parallel NOR uses the MTD subsystem. Because the parallel NOR is very small, you may only use the jffs2 but cannot use the UBIFS for it.

42.4 Source Code

To set the proper timing only for the parallel NOR, refer to `mx6q_setup_weimcs()` in arch/arm/mach-mx6/board-mx6q_sabreauto.c.

42.5 Enabling the WEIM NOR

Add `weim-nor` to the kernel command line to enable the WEIM NOR. The WEIM NOR has pin conflict with some other modules, such as the SPI.

Chapter 43
Quad Serial Peripheral Interface (QuadSPI) Driver

43.1 Introduction

The Quad Serial Peripheral Interface (QuadSPI) block acts as an interface to one single or two external serial flash devices, each with up to four bidirectional data lines.

It supports the following features:

- Flexible sequence engine to support various flash vendor devices.
- Single, dual, quad and octal mode of operation.
- DDR/DTR mode wherein the data is generated on every edge of the serial flash clock.
- Support for flash data strobe signal for data sampling in DDR and SDR mode.
- DMA support to read RX Buffer data via AMBA AHB bus (64-bit width interface) or IP registers space (32-bit access).

43.2 Hardware Operation

On some boards, the Quad SPI NOR - N25Q256A is equipped, while on some other boards S25FL128S is equipped. Check the Quad SPI NOR type on the boards and then configure it properly.

The N25Q256A is a high-performance multiple input/output serial Flash memory device. The innovative, high-performance, dual and quad input/output instructions enable double or quadruple the transfer bandwidth for READ and PROGRAM operations.The memory is organized as 512 (64KB) main sectors and can be erased 64KB sectors at a time. The device features 3-byte or 4-byte address modes to access memory beyond 128Mb. When 4-byte address mode is enabled, all commands requiring an address must be entered and exited with a 4-byte address mode command: ENTER 4-BYTE ADDRESS MODE command and EXIT 4-BYTE ADDRESS MODE command. The 4-byte address mode can also be enabled through the nonvolatile configuration register. The memory can be

operated with three different protocols:Extended SPI (standard SPI protocol upgraded with dual and quad operations), Dual I/O SPI and Quad I/O SPI. Each protocol contains unique commands to perform READ operations in DTR mode. This enables high data throughput while running at lower clock frequencies.

The S25FL128S device is flash non-volatile memory product. It connects to a host system via a Serial Peripheral Interface (SPI). Traditional SPI single bit serial input and output (SIngle I/O or SIO) is supported as well as optional two bit (Dual I/O or DIO) and four bit (Quad I/O or QIO) serial commands. It also adds support for Double Data Rate (DDR) read commands for SIO, DIO, and QIO that transfer address and read data on both edges of the clock.

43.3 Software Operation

In a Flash-based embedded Linux system, a number of Linux technologies work together to implement a file system. Figure below illustrates the relationships between some of the standard components.

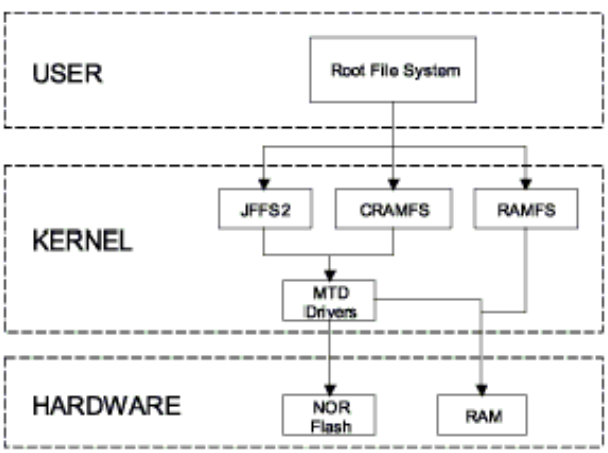


Figure 43-1. Components of a Flash-Based File System

The MTD subsystem for Linux is a generic interface to memory devices, such as Flash and RAM, providing simple read, write, and erase access to physical memory devices. Devices called mtdblock devices can be mounted by JFFS, JFFS2 and CRAMFS file systems. The Quad SPI NOR MTD driver is based on the MTD data Flash driver in the kernel by adding SPI access. In the initialization phase, the Quad SPI NOR MTD driver detects a data Flash by reading the JEDEC ID. Then the driver adds the MTD device. The SPI NOR MTD driver also provides the interfaces to read, write, and erase NOR Flash.

43.4 Driver Features

This Quad NOR driver implementation supports the following feature:

- Provides necessary information for the upper-layer MTD driver.

43.5 Source Code Structure

The Quad SPI NOR driver is implemented in the following directory:

drivers/mtd/spi-nor/

Table below shows the driver file:

Table 43-1. SPI NOR MTD Driver File

File	Description
spi-nor.c	Source file, spi-nor framework
fsl-quadspi.c	Source file, FSL Quad SPI Driver

43.6 Menu Configuration Options

To enable the Quad SPI driver, the following options must be set:

- CONFIG_MTD_SPI_NOR_BASE: This is the framework for the SPI NOR which can be used by the SPI device drivers and the SPI-NOR device driver.
- CONFIG_SPI_FSL_QUADSPI: This enables support for the Quad SPI controller in master mode.

Chapter 44
Fast Ethernet Controller (FEC) Driver

44.1 Introduction

The Fast Ethernet Controller (FEC) driver performs the full set of IEEE 802.3/Ethernet CSMA/CD media access control and channel interface functions.

The FEC requires an external interface adapter and transceiver function to complete the interface to the Ethernet media. It supports half or full-duplex operation on 10 Mbps, 100 Mbps or 1000 Mbps related Ethernet networks.

The FEC driver supports the following features:

- Full/Half duplex operation
- Link status change detect
- Auto-negotiation (determines the network speed and full or half-duplex operation)
- Transmits features such as automatic retransmission on collision and CRC generation
- Obtaining statistics from the device such as transmit collisions

The network adapter can be accessed through the ifconfig command with interface name ethx. The driver auto-probes the external adaptor (PHY device).

44.2 Hardware Operation

The FEC is an Ethernet controller that interfaces the system to the LAN network.

The FEC supports different standard MAC-PHY (physical) interfaces for connection to an external Ethernet transceiver. The FEC supports the 10/100 Mbps MII, and 10/100 Mbps RMII. In addition, the FEC supports 1000 Mbps RGMII, which uses 4-bit reduced GMII operating at 125 MHz.

A brief overview of the device functionality is provided here. For details, see the FEC chapter of the following documents:

- *i.MX 6Dual/6Quad Applications Processor Reference Manual* (IMX6DQRM)
- *i.MX 6Solo/6DualLite Applications Processor Reference Manual* (IMX6SDLRM)
- *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM)
- *i.MX 6SoloX Applications Processor Reference Manual* (IMX6SXRM)
- *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM)

In MII mode, there are 18 signals defined by the IEEE 802.3 standard and supported by the EMAC. MII, RMII and RGMII modes uses a subset of the 18 signals. These signals are listed in table below.

Table 44-1. Pin Usage in MII, RMII and RGMII Modes

Direction	EMAC Pin Name	MII Usage	RMII Usage	RGMII Usage (not supported by i.MX 6SoloLite or i.MX 6UltraLite)
In/Out	FEC_MDIO	Management Data Input/Output	Management Data Input/output	Management Data Input/Output
Out	FEC_MDC	Management Data Clock	General output	Management Data Clock
Out	FEC_TXD[0]	Data out, bit 0	Data out, bit 0	Data out, bit 0
Out	FEC_TXD[1]	Data out, bit 1	Data out, bit 1	Data out, bit 1
Out	FEC_TXD[2]	Data out, bit 2	Not Used	Data out, bit 2
Out	FEC_TXD[3]	Data out, bit 3	Not Used	Data out, bit 3
Out	FEC_TX_EN	Transmit Enable	Transmit Enable	Transmit Enable
Out	FEC_TX_ER	Transmit Error	Not Used	Not Used
In	FEC_CRS	Carrier Sense	Not Used	Not Used
In	FEC_COL	Collision	Not Used	Not Used
In	FEC_TX_CLK	Transmit Clock	Not Used	Synchronous clock reference (REF_CLK, can connect from PHY)
In	FEC_RX_ER	Receive Error	Receive Error	Not Used
In	FEC_RX_CLK	Receive Clock	Not Used	Synchronous clock reference (REF_CLK, can connect from PHY)
In	FEC_RX_DV	Receive Data Valid	Receive Data Valid and generate CRS	RXDV XOR RXERR on the falling edge of FEC_RX_CLK.
In	FEC_RXD[0]	Data in, bit 0	Data in, bit 0	Data in, bit 0
In	FEC_RXD[1]	Data in, bit 1	Data in, bit 1	Data in, bit 1
In	FEC_RXD[2]	Data in, bit 2	Not Used	Data in, bit 2
In	FEC_RXD[3]	Data in, bit 3	Not Used	Data in, bit 3

The MII management interface consists of two pins, FEC_MDIO, and FEC_MDC. The FEC hardware operation can be divided in the parts listed below. For details, see the Applications Processor Reference Manuals.

- Transmission-The Ethernet transmitter is designed to work with almost no intervention from software. Once ECR[ETHER_EN] is asserted and data appears in the transmit FIFO, the Ethernet MAC is able to transmit onto the network. When the transmit FIFO fills to the watermark (defined by the TFWR), the MAC transmit logic asserts FEC_TX_EN and starts transmitting the preamble (PA) sequence, the start frame delimiter (SFD), and then the frame information from the FIFO. However, the controller defers the transmission if the network is busy (FEC_CRS asserts).
- Before transmitting, the controller waits for carrier sense to become inactive, then determines if carrier sense stays inactive for 60 bit times. If the transmission begins after waiting an additional 36 bit times (96 bit times after carrier sense originally became inactive), both buffer (TXB) and frame (TXF) interrupts may be generated as determined by the settings in the EIMR.

- Reception-The FEC receiver is designed to work with almost no intervention from the host and can perform address recognition, CRC checking, short frame checking, and maximum frame length checking. When the driver enables the FEC receiver by asserting ECR[ETHER_EN], it immediately starts processing receive frames. When FEC_RX_DV asserts, the receiver checks for a valid PA/SFD header. If the PA/SFD is valid, it is stripped and the frame is processed by the receiver. If a valid PA/SFD is not found, the frame is ignored. In MII mode, the receiver checks for at least one byte matching the SFD. Zero or more PA bytes may occur, but if a 00 bit sequence is detected prior to the SFD byte, the frame is ignored.
- After the first six bytes of the frame have been received, the FEC performs address recognition on the frame. During reception, the Ethernet controller checks for various error conditions and once the entire frame is written into the FIFO, a 32-bit frame status word is written into the FIFO. This status word contains the M, BC, MC, LG, NO, CR, OV, and TR status bits, and the frame length. Receive Buffer (RXB) and Frame Interrupts (RXF) may be generated if enabled by the EIMR register. When the receive frame is complete, the FEC sets the L bit in the RxBD, writes the other frame status bits into the RxBD, and clears the E bit. The Ethernet controller next generates a maskable interrupt (RXF bit in EIR, maskable by RXF bit in EIMR), indicating that a frame has been received and is in memory. The Ethernet controller then waits for a new frame.

- Interrupt management-When an event occurs that sets a bit in the EIR, an interrupt is generated if the corresponding bit in the interrupt mask register (EIMR) is also set. The bit in the EIR is cleared if a one is written to that bit position; writing zero has no effect. This register is cleared upon hardware reset. These interrupts can be divided into operational interrupts, transceiver/network error interrupts, and internal error interrupts. Interrupts which may occur in normal operation are GRA, TXF, TXB, RXF, RXB. Interrupts resulting from errors/problems detected in the network or transceiver are HBERR, BABR, BABT, LC, and RL. Interrupts resulting from

internal errors are HBERR and UN. Some of the error interrupts are independently counted in the MIB block counters. Software may choose to mask off these interrupts as these errors are visible to network management through the MIB counters.

- PHY management-phylib was used to manage all the FEC phy related operation such as phy discovery, link status, and state machine.MDIO bus will be created in FEC driver and registered into the system. See Documentation/networking/phy.txt under the Linux OS source directory for more information.

44.2.1 Software Operation

The FEC driver supports the following functions:

- Module initialization-Initializes the module with the device-specific structure
- Rx/Tx transmition
- Interrupt servicing routine
- PHY management
- FEC management such init/start/stop
- i.MX 6 FEC module use little-endian format

44.2.2 Source Code Structure

Table below shows the source files.

They are available in the

`drivers/net/ethernet/freescale/ directory.`

Table 44-2. FEC Driver Files

File	Description
fec.h	Header file defining registers
fec_main.c	Linux driver for Ethernet LAN controller

For more information about the generic Linux driver, see the drivers/net/ethernet/freescale/fec_main.c source file.

44.2.3 Menu Configuration Options

Configure the kernel to provide for this module:

- CONFIG_FEC is provided for this module. This option is available under:

- Device Drivers > Network device support > Ethernet (10, 100 or 1000 Mbit) > FEC Ethernet controller.
- To mount NFS-rootfs through FEC, disable the other Network config in the menuconfig if need.

44.3 Programming Interface

Table 44-2 lists the source files for the FEC driver.

The following section shows the modifications that were required to the original Ethernet driver source for porting it to the i.MX device.

44.3.1 Device-Specific Defines

Device-specific defines are added to the header file (fec.h) and they provide common board configuration options.

fec.h defines the struct for the register access and the struct for the buffer descriptor. For example,

```
/*
 *      Define the buffer descriptor structure.
 */
struct bufdesc {
          unsigned short            cbd_datlen;       /* Data length */
          unsigned short            cbd_sc;           /* Control and status info */
          unsigned long             cbd_bufaddr;      /* Buffer address */

};
struct bufdesc_ex {
        struct bufdesc desc;
        unsigned long cbd_esc;
        unsigned long cbd_prot;
        unsigned long cbd_bdu;
        unsigned long ts;
        unsigned short res0[4];
};

/*
 *      Define the register access structure.
 */
#define FEC_IEVENT              0x004 /* Interrupt event reg */
#define FEC_IMASK               0x008 /* Interrupt mask reg */
#define FEC_R_DES_ACTIVE        0x010 /* Receive descriptor reg */
#define FEC_X_DES_ACTIVE        0x014 /* Transmit descriptor reg */
#define FEC_ECNTRL              0x024 /* Ethernet control reg */
#define FEC_MII_DATA            0x040 /* MII manage frame reg */
#define FEC_MII_SPEED           0x044 /* MII speed control reg */
#define FEC_MIB_CTRLSTAT        0x064 /* MIB control/status reg */
#define FEC_R_CNTRL             0x084 /* Receive control reg */
#define FEC_X_CNTRL             0x0c4 /* Transmit Control reg */
#define FEC_ADDR_LOW            0x0e4 /* Low 32bits MAC address */
#define FEC_ADDR_HIGH           0x0e8 /* High 16bits MAC address */
#define FEC_OPD                 0x0ec /* Opcode + Pause duration */
```

```
#define FEC_HASH_TABLE_HIGH     0x118 /* High 32bits hash table */
#define FEC_HASH_TABLE_LOW      0x11c /* Low 32bits hash table */
#define FEC_GRP_HASH_TABLE_HIGH 0x120 /* High 32bits hash table */
#define FEC_GRP_HASH_TABLE_LOW  0x124 /* Low 32bits hash table */
#define FEC_X_WMRK              0x144 /* FIFO transmit water mark */
#define FEC_R_BOUND             0x14c /* FIFO receive bound reg */
#define FEC_R_FSTART            0x150 /* FIFO receive start reg */
#define FEC_R_DES_START         0x180 /* Receive descriptor ring */
#define FEC_X_DES_START         0x184 /* Transmit descriptor ring */
#define FEC_R_BUFF_SIZE         0x188 /* Maximum receive buff size */
#define FEC_MIIGSK_CFGR         0x300 /* MIIGSK config register */
#define FEC_MIIGSK_ENR          0x308 /* MIIGSK enable register */
```

44.3.2 Getting a MAC Address

The MAC address can be set through the kernel command line, kernel device tree DTS file, OCOTP, or MAC registers set by bootloader, such as U-Boot. The FEC driver uses it to configure the MAC address for the network device. In general, use kernel command line in a form of fec.macaddr=0x00,0x04,0x9f,0x01,0x30,0xe0 to set the MAC address. Due to certain pin conflicts (FEC RMII mode needs to use GPIO_16 or RGMII_TX_CTL pin as reference clock input/output channel), the one of the both pins cannot connect to branch lines for other modules use because the branch lines have serious influence on clock.

Chapter 45
ENET IEEE-1588 Driver

45.1 Hardware Operation

ENET IEEE-1588 driver performs a set of functions that enabling precise synchronization of clocks in network communication.

The driver requires a protocol stack to complete IEEE-1588 full protocol. It complies with the IXXAT stack interfaces.

To allow for IEEE 1588 or similar time synchronization protocol implementations, the ENET MAC is combined with a time-stamping module to support precise time stamping of incoming and outgoing frames. 1588 Support is enabled when the register bit ENA_1588 is set to '1'.

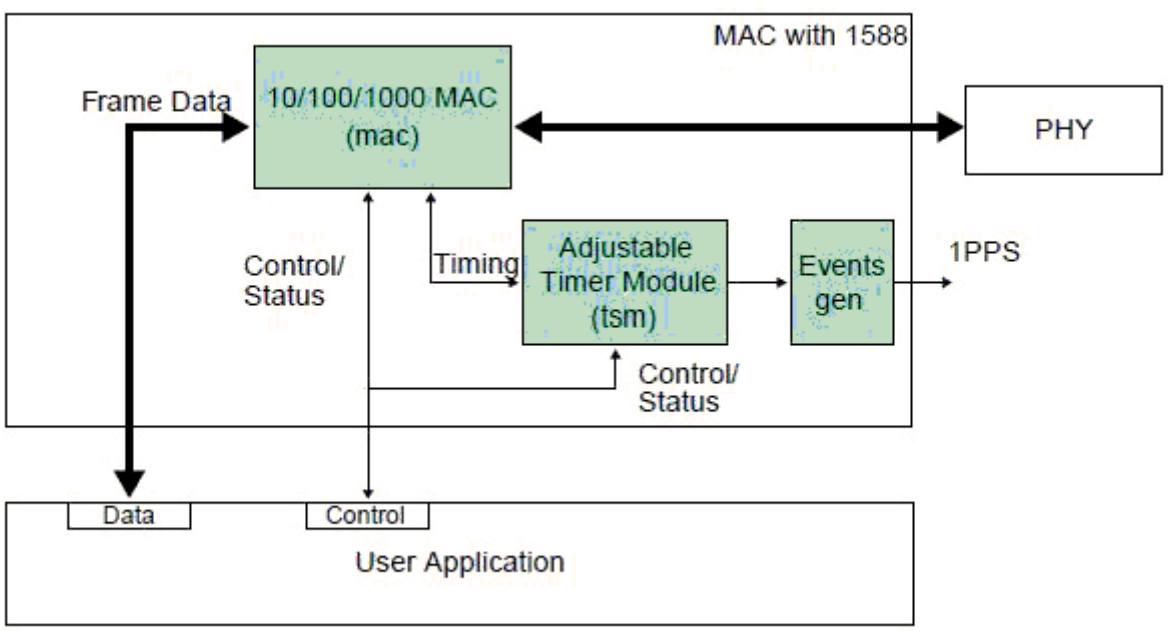


Figure 45-1. IEEE 1588 Functions Overview

45.1.1 Transmit Timestamping

On transmit, only 1588 event frames need to be time-stamped. The Client application (for example, the MAC driver) should detect 1588 event frames and set the signal ff_tx_ts_frm together with the frame.

For every transmitted frame, the MAC returns the captured timestamp on tx_ts (31:0) with the frame sequence number (tx_ts_id(3:0)) and the transmit status. The transmit status bit tx_ts_stat (5) indicates that the application had the ff_tx_ts_frm signal asserted for the frame.

If ff_tx_ts_frm is set to '1', the MAC additionally memorizes the timestamp for the frame in the register TS_TIMESTAMP. The interrupt bit EIR (TS_AVAIL) is set to indicate that a new timestamp is available.

Software would implement a handshaking procedure by setting the ff_tx_ts_frm signal when it transmits the frame it needs a timestamp for and then waits on the EIR (TS_AVAIL) interrupt bit to know when the timestamp is available. It then can read the timestamp from the TS_TIMESTAMP register. This is done for all event frames; other frames do not use the ff_tx_ts_frm indicator and hence do not interfere with the timestamp capture.

45.1.2 Receive Timestamping

When a frame is received, the MAC latches the value of the timer when the frame SFD field is detected and provides the captured timestamp on ff_rx_ts(31:0). This is done for all received frames.

The DMA controller has to ensure that it transfers the timestamp provided for the frame into the corresponding field within the receive descriptor for software access.

45.2 Software Operation

The 1588 Driver has the functions listed below:

- Module initialization-Initializes the module with the device-specific structure, and registers a character driver.
- Interrupt servicing routine-Supports events, such as TS_AVAIL, TS_TIMER. The driver shares interrupt servicing routine with FEC driver.

45.2.1 Source Code Structure

Table below lists the source files available in the drivers/net/ethernet/freescale/ directory.

Table 45-1. ENET 1588 File List

File	Description
fec.h	Header file defining registers
fec_ptp.c	Linux driver for ENET 1588 timer

For more information about the generic Linux driver, see the drivers/net/ethernet/freescale/fec_ptp.c source file.

45.2.2 Linux Menu Configuration Options

By default, ENET 1588 is enabled.

45.3 Programming Interface

The 1588 driver complies with the Linuxptp protocol stack interface.

Stack-specific defines are added to the header file (fec.h).

45.4 1588 Stack Support

The 1588 driver supports Linuxptp protocol stack.

45.4.1 1588 Stack Introduction

This release supports the following type of the 1588 Stack:

- Linuxptp stack

  This software is an implementation of the Precision Time Protocol (PTP) according to IEEE standard 1588 for Linux OS. The dual design goals are to provide a robust implementation of the standard and to use the most relevant and modern Application

Programming Interfaces (API) offered by the Linux OS kernel. Supporting legacy APIs and other platforms is not a goal. The software is copyrighted by the authors and is licensed under the GNU General Public License.

The software development is hosted at Source Forge: sourceforge.net/projects/linuxptp/

45.4.2 Linuxptp Stack Features

Linuxptp support the following features:

- Ordinary/Boundary Clock
- Best master clock algorithm
- Transport over UDP/IPv4, UDP/IPv6, and IEEE 802.3
- Transparent clock (E2E/P2P)
- Slave only
- Supporting IEEE 802.1AS-2011 in the role of end station

45.4.3 How to Use the Stacks in Linux OS

In Linux OS, run 1588 stack binary with the following commands.

Linuxptp:

```
Transport on UDP IPV4 with E2E delay mechanism: ptp4l -A -4 -H -m -i eth0
Transport on UDP IPV4 with P2P delay mechanism: ptp4l -P -A -4 -H -m -i eth0
Transport on UDP IPV6 with E2E delay mechanism: ptp4l -A -6 -H -m -i eth0
Transport on UDP IPV6 with P2P delay mechanism: ptp4l -P -A -6 -H -m -i eth0
Transport on IEEE 802.3 with E2E delay mechanism: ptp4l -A -2 -H -m -i eth0
Transport on IEEE 802.3 with P2P delay mechanism: ptp4l -P -A -2 -H -m -i eth0
```

Chapter 46
Universal Asynchronous Receiver/Transmitter (UART) Driver

46.1 Introduction

The low-level UART driver interfaces the Linux serial driver API to all the UART ports.

It has the following features:

- Interrupt-driven and SDMA-driven transmit/receive of characters
- Standard Linux baud rates up to 4 Mbps
- Transmit and receive characters with 7-bit and 8-bit character lengths
- Transmits one or two stop bits
- Supports TIOCMGET IOCTL to read the modem control lines. Only supports the constants TIOCM_CTS and TIOCM_CAR, plus TIOCM_RI in DTE mode only
- Supports TIOCMSET IOCTL to set the modem control lines. Supports the constants TIOCM_RTS and TIOCM_DTR only
- Odd and even parity
- XON/XOFF software flow control. Serial communication using software flow control is reliable when communication speeds are not too high and the probability of buffer overruns is minimal
- CTS/RTS hardware flow control-both interrupt-driven software-controlled hardware flow and hardware-driven hardware-controlled flow
- Send and receive break characters through the standard Linux serial API
- Recognizes frame and parity errors
- Ability to ignore characters with break, parity and frame errors
- Get and set UART port information through the TIOCGSSERIAL and TIOCSSERIAL TTY IOCTL. Some programs like setserial and dip use this feature to make sure that the baud rate was set properly and to get general information on the device. The UART type should be set to 52 as defined in the serial_core.h header file.
- Serial IrDA

- Power management feature by suspending and resuming the UART ports
- Standard TTY layer IOCTL calls

All the UART ports can be accessed from the device files /dev/ttymxc0 to /dev/ttymxc1. Autobaud detection is not supported.

46.2 Hardware Operation

See the *i.MX VPU Application Programming Interface Linux Reference Manual* (IMXVPUAPI) to determine the number of UART modules available in the device.

Each UART hardware port is capable of standard RS-232 serial communication and has support for IrDA 1.0.

Each UART contains a 32-byte transmitter FIFO and a 32-half-word deep receiver FIFO. Each UART also supports a variety of maskable interrupts when the data level in each FIFO reaches a programmed threshold level and when there is a change in state in the modem signals. Each UART can be programmed to be in DCE or DTE mode.

46.2.1 Software Operation

The Linux OS contains a core UART driver that manages many of the serial operations that are common across UART drivers for various platforms.

The low-level UART driver is responsible for supplying information such as the UART port information and a set of control functions to the core UART driver. These functions are implemented as a low-level interface between the Linux OS and the UART hardware. They cannot be called from other drivers or from a user application. The control functions used to control the hardware are passed to the core driver through a structure called uart_ops, and the port information is passed through a structure called uart_port. The low level driver is also responsible for handling the various interrupts for the UART ports, and providing console support if necessary.

Each UART can be configured to use DMA for the data transfer by enabling the DMA channel in the DTS file.

The driver requests two DMA channels for the UARTs that need DMA transfer. On a receive transaction, the driver copies the data from the DMA receive buffer to the TTY Flip Buffer.

While using DMA to transmit, the driver copies the data from the UART transmit buffer to the DMA transmit buffer and sends this buffer to the DMA system. For more information, see the Linux documentation on the serial driver in the kernel source tree.

The low-level driver supports both interrupt-driven software-controlled hardware flow control and hardware-driven hardware flow control. The hardware flow control method can be configured using the options provided in the header file. The user has the capability to de-assert the CTS line using the available IOCTL calls. If the user wishes to assert the CTS line, then control is transferred back to the receiver, as long as the driver has been configured to use hardware-driven hardware flow control.

46.2.2 Driver Features

The UART driver supports the following features:

- Baud rates up to 4 Mbps
- Recognizes frame and parity errors only in interrupt-driven mode; does not recognize these errors in DMA-driven mode
- Sends, receives, and appropriately handles break characters
- Recognizes the modem control signals
- Ignores characters with frame, parity, and break errors if requested to do so
- Implements support for software and hardware flow control (software-controlled and hardware-controlled)
- Get and set the UART port information; certain flow control count information is not available in hardware-driven hardware flow control mode
- Implements support for Serial IrDA
- Power management
- Interrupt-driven and DMA-driven data transfer

46.2.3 Source Code Structure

Table below shows the UART driver source files that are available in the directory:

`<Yocto_BuildDir>/linux/drivers/tty/serial.`

Table 46-1. UART Driver Files

File	Description
imx.c	Low level driver

46.3 Configuration

This section discusses configuration options associated with Linux OS, chip configuration options, and board configuration options.

46.3.1 Configuration Options

The UART driver is enabled by default.

46.3.2 Source Code Configuration Options

This section details the chip configuration options and board configuration options.

46.3.3 Chip Configuration Options

46.3.4 Board Configuration Options

For the i.MX 6Quad/6DualLite/6SoloLite/6SoloX, the board-specific configuration options for the driver are set in:

```
arch/arm/boot/dts/imx6*.dts
```


```
arch/arm/boot/dts/imx6*.dts
```


46.4 Programming Interface

The UART driver implements all the methods required by the Linux serial API to interface with the UART port.

The driver implements and provides a set of control methods to the Linux core UART driver. For more information about the methods implemented in the driver, see the API document.

46.4.1 Interrupt Requirements

The UART driver interface generates only one interrupt.

The status is used to determine which kinds of interrupt occurs, such as RX or TX.

Chapter 47
Wi-Fi BCM4339 Driver

47.1 Hardware Operation

The officially supported Wi-Fi chip with FSL BSP is Murata Type ZP module based on Broadcom BCM4339.

It is an IEEE802.11a/b/g/n/acW-LAN + Bluetooth 4.0+ FM Rx combo module.

47.1.1 Software Operation

FSL BSP uses the BCMDHD_1_141_72 Wi-Fi driver delivered from Broadcom.

47.1.2 Driver features

The bcmdhd is a cfg80211 driver, which supports both the station and AP mode of operation.

Station mode supports 802.11 a/b/g/n with HT20 on 2.4/5 GHz and HT40 only on 5GHz. Some of the other features include WPA/WPA2,WPS, WMM, WMM-PS, and Bluetooth wireless coexistence. AP mode can be operated only in b/g mode with support for a subset of features mentioned above.

The driver supports cfg80211 but comes with its own set of wext ioctls which have historically supported some of our customers with features, such as BT 3.0 and AP mode of operation.

The driver requires firmware that runs on the chip's network processor.

47.1.3 Source Code Structure

The BCMDHD driver source files are available in the kernel source directory: drivers/net/wireless/bcmdhd.

47.1.4 Linux Menu Configuration Options

The following Linux kernel configuration option is provided for this module:

- CONFIG_BCMDHD
- CONFIG_BCMDHD_FW_PATH
- CONFIG_BCMDHD_NVRAM_PATH
- CONFIG_BCMDHD_SDIO

47.2 How to Install the Driver Module

```
modprobe bcmdhd firmware_path=/lib/firmware/bcm/ZP_BCM4339/fw_bcmdhd.bin nvram_path=/lib/
firmware/bcm/ZP_BCM4339/bcmdhd.ZP.SDIO.cal
```

NOTE

firmware_path and nvram_path should be changed according to the environment.

The wlan0 link should become ready automatically.

47.3 Device Tree Binding

For device tree, the BCMDHD driver requires the following nodes to be defined in the device tree. For example,

```
regulators {
                compatible = "simple-bus";
                #address-cells = <1>;
                #size-cells = <0>;

                wlreg_on: fixedregulator@100 {
                        compatible = "regulator-fixed";
                        regulator-min-microvolt = <5000000>;
                        regulator-max-microvolt = <5000000>;
                        regulator-name = "wlreg_on";
                        gpio = <&gpio4 21 GPIO_ACTIVE_HIGH>;
                        enable-active-high;
                };
        };

        bcmdhd_wlan_0: bcmdhd_wlan@0 {
                compatible = "android,bcmdhd_wlan";
                wlreg_on-supply = <&wlreg_on>;
        };
```

The bcmdhd_wlan_0 is the basic device node for the BCMDHD driver to probe and wlreg_on is the standard regulator node for the driver to control the WL_REG_ON regulator.

47.4 Murata Module Support Status

Table 47-1. Murata Module Support status

	Murata Adapter	Module	HW Rework Notes	Wi-Fi Module Feature	Bluetooth Module Feature
i.MX 6Quad/ 6DualLite SABRE-SD	Ver 2.0	Type ZP	FSL board rework required. See the Murata HW rework Guide.	WL_REG_ON	-
i.MX 6SoloX SABRE-SD	Ver 1.0 + SD Card Ext	Type ZP	No HW rework.	WL_REG_ON	-
i.MX 6SoloLite EVK	Ver 1.0 + SD Card Ext	Type ZP	No HW rework.	WL_REG_ON	Unsupported
i.MX 6UltraLite EVK	Ver 2.0	Type ZP	No HW rework.	WL_REG_ON	-
i.MX 7Dual SDB	No adapter needed, integrated on the board	Type ZP	No HW rework.	WL_REG_ON	-

NOTE

Ver 1.0/Ver 2.0 represents the Murata adapter version.

The ARD board does not support the Murata module.

Chapter 48
Pulse-Width Modulator (PWM) Driver

48.1 Introduction

The pulse-width modulator (PWM) has a 16-bit counter and is optimized to generate sound from stored sample audio images and generate tones.

The PWM has 16-bit resolution and uses a 4x16 data FIFO to generate sound. The software module is composed of a Linux driver that allows privileged users to control the backlight by the appropriate duty cycle of the PWM Output (PWMO) signal.

48.1.1 Hardware Operation

Figure below shows the PWM block diagram.

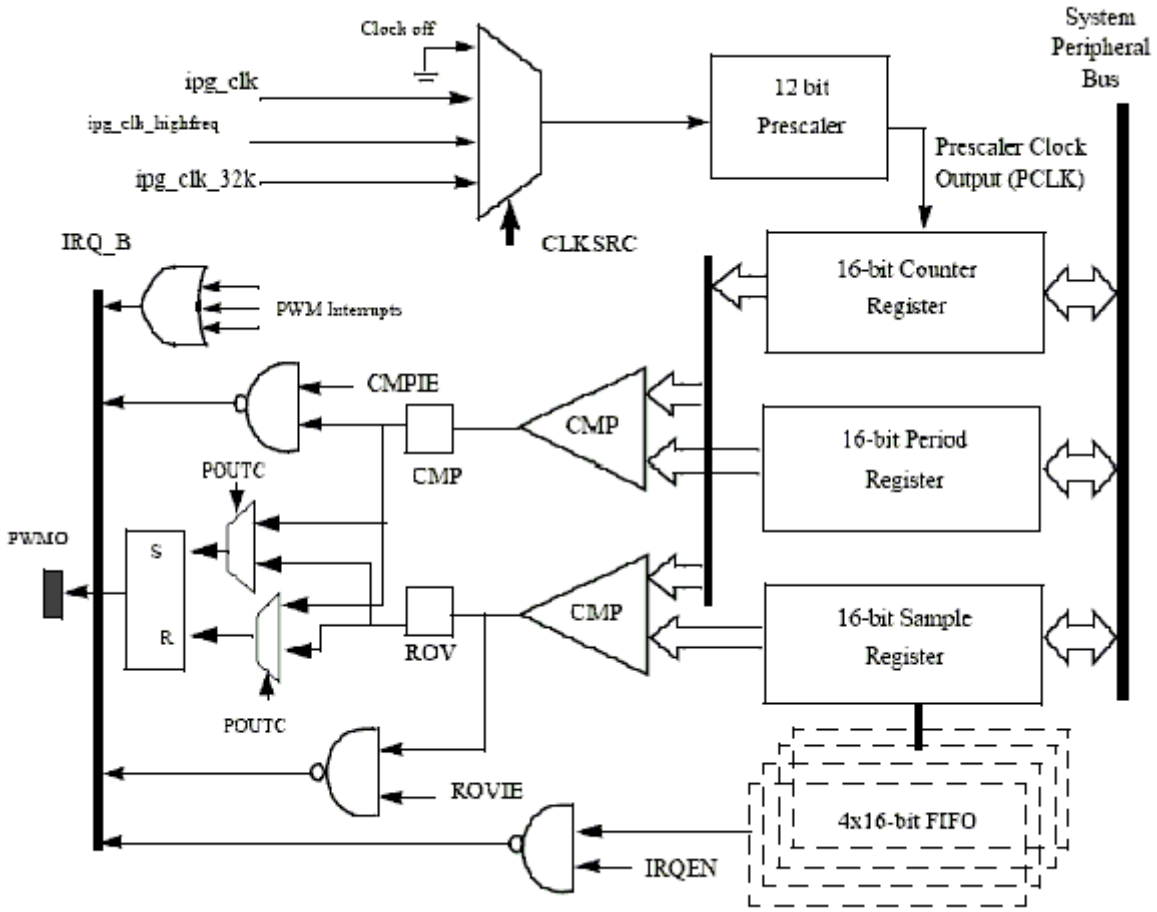


Figure 48-1. PWM Block Diagram

The PWM follows IP Bus protocol for interfacing with the processor core. It does not interface with any other modules inside the device except for the clock and reset inputs from the Clock Control Module (CCM) and interrupt signals to the processor interrupt handler. The PWM includes a single external output signal, PMWO. The PWM includes the following internal signals:

- Three clock inputs
- Four interrupt lines
- One hardware reset line
- Four low power and debug mode signals
- Four scan signals
- Standard IP slave bus signals

48.1.2 Clocks

The clock that feeds the prescaler can be selected from:

- High frequency clock-provided by the CCM. The PWM can be run from this clock in low power mode.
- Low reference clock-32 KHz low reference clock provided by the CCM. The PWM can be run from this clock in the low power mode.
- Global functional clock-for normal operations. In low power modes this clock can be switched off.

The clock input source is determined by the CLKSRC field of the PWM control register. The CLKSRC value should only be changed when the PWM is disabled.

48.1.3 Software Operation

The PWM device driver reduces the amount of power sent to a load by varying the width of a series of pulses to the power source. One common and effective use of the PWM is controlling the backlight of a QVGA panel with a variable duty cycle.

Table below provides a summary of the interface functions in source code.

Table 48-1. PWM Driver Summary

Function	Description
struct pwm_device *pwm_request(int pwm_id, const char *label)	Request a PWM device
void pwm_free(struct pwm_device *pwm)	Free a PWM device
int pwm_config(struct pwm_device *pwm, int duty_ns, int period_ns)	Change a PWM device configuration
int pwm_enable(struct pwm_device *pwm)	Start a PWM output toggling
int pwm_disable(struct pwm_device *pwm)	Stop a PWM output toggling

The function pwm_config() includes most of the configuration tasks for the PWM module, including the clock source option, and period and duty cycle of the PWM output signal. It is recommended to select the peripheral clock of the PWM module, rather than the local functional clock, as the local functional clock can change.

48.1.4 Driver Features

The PWM driver includes the following software and hardware support:

- Duty cycle modulation
- Varying output intervals
- Two power management modes-full on and full of

48.1.5 Source Code Structure

Table below lists the source files and headers available in the following directories:

```
<Yocto_BuildDir>/linux/arch/arm/plat-mxc/pwm.c
<Yocto_BuildDir>/linux/include/linux/pwm.h
```

Table 48-2. PWM Driver Files

File	Description
pwm.h	Functions declaration
pwm.c	Functions definition

48.1.6 Menu Configuration Options

To get to the PWM driver, use the command bitbake linux-imx -c menuconfig. On the screen displayed, select **Configure the kernel** and exit. When the next screen appears select the following option to enable the PWM driver:

- System Type > Enable PWM driver
- Select the following option to enable the Backlight driver:

  Device Drivers > Graphics support > Backlight & LCD device support > Generic PWM based Backlight Driver

Chapter 49
Watchdog (WDOG) Driver

49.1 Introduction

The Watchdog Timer module protects against system failures by providing an escape from unexpected hang or infinite loop situations or programming errors.

Some platforms may have two WDOG modules with one of them having interrupt capability.

49.1.1 Hardware Operation

Once the WDOG timer is activated, it must be serviced by software on a periodic basis.

If servicing does not take place in time, the WDOG times out. Upon a time-out, the WDOG either asserts the wdog_b signal or a wdog_rst_b system reset signal, depending on software configuration. The watchdog module cannot be deactivated once it is activated.

49.1.2 Software Operation

The Linux OS has a standard WDOG interface that allows support of a WDOG driver for a specific platform.

WDOG can be suspended/resumed in STOP/DOZE and WAIT modes independently. Since some bits of the WGOD registers are only one-time programmable after booting, ensure these registers are written correctly.

49.2 Generic WDOG Driver

The generic WGOD driver is implemented in the <Yocto_BuildDir>/linux/drivers/watchdog/imx2_wdt.c file.

It provides functions for various IOCTLs and read/write calls from the user level program to control the WDOG.

49.2.1 Driver Features

This WDOG implementation includes the following features:

- Generates the reset signal if it is enabled but not serviced within a predefined timeout value (defined in milliseconds in one of the WDOG source files)
- Does not generate the reset signal if it is serviced within a predefined timeout value
- Provides IOCTL/read/write required by the standard WDOG subsystem

49.2.2 Menu Configuration Options

To get to the Linux kernel configuration option provided for this module, use the bitbake linux-imx -c menuconfigcommand. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following option to enable this module:

- CONFIG_IMX2_WDT-Enables Watchdog timer module. This option is available under Device Drivers > Watchdog Timer Support > IMX2+ Watchdog.

49.2.3 Source Code Structure

Table below shows the source files for WDOG drivers that are in the following directory:

`<Yocto_BuildDir>/linux/drivers/watchdog.`

Table 49-1. WDOG Driver Files

File	Description
imx2_wdt.c	WDOG function implementations

Watchdog system reset function is located under <Yocto_BuildDir>/linux/arch/arm/plat-mxc/system.c

49.2.4 Programming Interface

The following IOCTLs are supported in the WDOG driver:

- WDIOC_GETSUPPORT
- WDIOC_GETSTATUS
- WDIOC_GETBOOTSTATUS
- WDIOC_KEEPALIVE
- WDIOC_SETTIMEOUT
- WDIOC_GETTIMEOUT

For detailed descriptions about these IOCTLs, see <Yocto_BuildDir>/linux/Documentation/watchdog.

Chapter 50
OProfile

50.1 Introduction

OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.

OProfile is released under the GNU GPL license. It consists of a kernel driver, a daemon for collecting sample data, and several post-profiling tools for turning data into information.

50.1.1 Overview

OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling.

All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

50.1.2 Features

OProfile has the following features.

- Unobtrusive-No special recompilations or wrapper libraries are necessary. Even debug symbols (-g option to gcc) are not necessary unless users want to produce annotated source. No kernel patch is needed; just insert the module.
- System-wide profiling-All code running on the system is profiled, enabling analysis of system performance.
- Performance counter support-Enables collection of various low-level data and association for particular sections of code.

- Call-graph support-With an 2.6 kernel, OProfile can provide gprof-style call-graph profiling data.
- Low overhead-OProfile has a typical overhead of 1-8% depending on the sampling frequency and workload.
- Post-profile analysis-Profile data can be produced on the function-level or instruction-level detail. Source trees, annotated with profile information, can be created. A hit list of applications and functions that utilize the most CPU time across the whole system can be produced.
- System support-Works with almost any 2.2, 2.4 and 2.6 kernels, and works on based platforms.

50.1.3 Hardware Operation

OProfile is a statistical continuous profiler.

In other words, profiles are generated by regularly sampling the current registers on each CPU (from an interrupt handler, the saved PC value at the time of interrupt is stored), and converting that runtime PC value into something meaningful to the programmer.

OProfile achieves this by taking the stream of sampled PC values, along with the detail of which task was running at the time of the interrupt, and converting the values into a file offset against a particular binary file. Each PC value is thus converted into a tuple (group or set) of binary-image offset. The userspace tools can use this data to reconstruct where the code came from, including the particular assembly instructions, symbol, and source line (through the binary debug information if present).

Regularly sampling the PC value like this approximates what actually was executed and how often and, more often than not, this statistical approximation is good enough to reflect reality. In common operation, the time between each sample interrupt is regulated by a fixed number of clock cycles. This implies that the results reflect where the CPU is spending the most time. This is a very useful information source for performance analysis.

The ARM CPU provides hardware performance counters capable of measuring these events at the hardware level. Typically, these counters increment once per each event and generate an interrupt on reaching some pre-defined number of events. OProfile can use these interrupts to generate samples and the profile results are a statistical approximation of which code caused how many instances of the given event.

50.2 Software Operation

50.2.1 Architecture-specific Components

OProfile supports the hardware performance counters available on a particular architecture. Code for managing the details of setting up and managing these counters can be located in the kernel source tree in the relevant <Yocto_BuildDir>/linux/arch/arm/oprofile directory. The architecture-specific implementation operates through filling in the oprofile_operations structure at initialization. This provides a set of operations, such as setup(), start(), stop(), and so on, that manage the hardware-specific details the performance counter registers.

The other important facility available to the architecture code is oprofile_add_sample(). This is where a particular sample taken at interrupt time is fed into the generic OProfile driver code.

50.2.2 oprofilefs Pseudo Filesystem

OProfile implements a pseudo-filesystem known as oprofilefs, which is mounted from userspace at /dev/oprofile. This consists of small files for reporting and receiving configuration from userspace, as well as the actual character device that the OProfile userspace receives samples from. At setup() time, the architecture-specific code may add further configuration files related to the details of the performance counters. The filesystem also contains a stats directory with a number of useful counters for various OProfile events.

50.2.3 Generic Kernel Driver

The generic kernel driver resides in <Yocto_BuildDir>/linux/drivers/oprofile/, and forms the core of how OProfile operates in the kernel. The generic kernel driver takes samples delivered from the architecture-specific code (through oprofile_add_sample()), and buffers this data (in a transformed configuration) until releasing the data to the userspace daemon through the /dev/oprofile/buffer character device.

50.2.4 OProfile Daemon

The OProfile userspace daemon takes the raw data provided by the kernel and writes it to the disk. It takes the single data stream from the kernel and logs sample data against a number of sample files (available in /var/lib/oprofile/samples/current/). For the benefit of

the separate functionality, the names and paths of these sample files are changed to reflect where the samples were from. This can include thread IDs, the binary file path, the event type used, and more.

After this final step from interrupt to disk file, the data is now persistent (that is, changes in the running of the system do not invalidate stored data). This enables the post-profiling tools to run on this data at any time (assuming the original binary files are still available and unchanged).

50.2.5 Post Profiling Tools

The collected data must be presented to the user in a useful form. This is the job of the post-profiling tools. In general, they collate a subset of the available sample files, load and process each one correlated against the relevant binary file, and produce user readable information.

50.3 Requirements

OProfile has the following requirements.

- Add Oprofile support with Cortex-A7 Event Monitor

50.3.1 Source Code Structure

Oprofile platform-specific source files are available in the directory:

```
<Yocto_BuildDir>/linux/arch/arm/oprofile/
```

Table 50-1. OProfile Source Files

File	Description
op_arm_model.h	Header File with the register and bit definitions
common.c	Source file with the implementation required for all platforms

The generic kernel driver for Oprofile is located under <Yocto_BuildDir>/linux/drivers/oprofile/

50.3.2 Menu Configuration Options

The following Linux kernel configurations are provided for this module.

To get to the Oprofile configuration, use the command bitbake linux-imx -c menuconfig. On the screen, first go to Package list and select Oprofile. Then return to the first screen and, select **Configure Kernel**, then exit, and a new screen appears.

- CONFIG_OPROFILE-configuration option for the oprofile driver. In the menuconfig this option is available under
- General Setup > Profiling support (EXPERIMENTAL) > OProfile system profiling (EXPERIMENTAL)

50.3.3 Programming Interface

This driver implements all the methods required to configure and control PMU and L2 cache EVTMON counters.

More information, see the Linux document generated from build: make htmldocs.

50.3.4 Interrupt Requirements

The number of interrupts generated with respect to the OProfile driver are numerous. The latency requirements are not needed.

The rate at which interrupts are generated depends on the event.

50.3.5 Example Software Configuration

The following steps show and example of how to configure the OProfile:

1. Use the command bitbake linux-imx -c menuconfig. On the screen, first, go to Package list and select Oprofile.
2. Then, return to the first screen and select Configure Kernel, follow the instruction from Menu Configuration Options, to enable Oprofile in the kernel space.
3. Save the configuration and start to build.
4. Copy Oprofile binaries to target rootfs. Copy vmlinux to /boot directory and run Oprofile

```
root@ubuntu:/boot# opcontrol --separate=kernel --vmlinux=/boot/vmlinux
root@ubuntu:/boot# opcontrol --reset
Signalling daemon... done
root@ubuntu:/boot# opcontrol --setup --event=CPU_CYCLES:100000
```

```
root@ubuntu:/boot# opcontrol --start
Profiler running.
root@ubuntu:/boot# opcontrol --dump
root@ubuntu:/boot# opreport
Overflow stats not available
CPU: ARM V7 PMNC, speed 0 MHz (estimated)
Counted CPU_CYCLES events (Number of CPU cycles) with a unit mask of 0x00 (No un
it mask) count 100000
CPU_CYCLES:100000|
  samples|      %|
------------------
        4 22.2222 grep
        CPU_CYCLES:100000|
          samples|      %|
        ------------------
                4 100.000 libc-2.9.so
        2 11.1111 cat
        CPU_CYCLES:100000|
          samples|      %|
        ------------------
                1 50.0000 ld-2.9.so
                1 50.0000 libc-2.9.so
...
root@ubuntu:/boot# opcontrol --stop
Stopping profiling.
```

Chapter 51
CAAM (Cryptographic Acceleration and Assurance Module)

51.1 CAAM Device Driver Overview

This section discusses implementation specifics of the kernel driver components supporting CAAM (Cryptographic Acceleration and Assurance Module) within the Linux kernel.

CAAM's base driver packaging can be categorized on two distinct levels:

- Configuration and Job Execution Level
- API Interface Level

Configuration and Job Execution Level consists of:

- a control and configuration module which maps the main register page and writes global or system required configuration information.
- a module that feeds jobs through job rings, and reports status.

API Interface Level consists of:

- An interface to the Sctterlist Crypto API supporting asynchronous single-pass authentication-encryption operations, and common blockciphers - `caamalg`.
- An interface to the Scatterlist Crypto API supporting asynchronous hashes - `caamhash`.
- An interface to the hwrng API supporting use of the Random Number Generator - `caamrng`.

51.2 Configuration and Job Execution Level

This section has two parts:

- Control/Configuration Driver
- Job Ring Driver

51.3 Control/Configuration Driver

The control and configuration driver is responsible for initializing and setting up the master register page, initializing early-on feature initialization, providing limited debug and monitoring capability, and generally ensuring that all other dependent driver subsystems can connect to a correctly-configured device.

Step by step, it performs the following actions at startup:

- Allocates a private storage block for this level.
- Maps a virtual address to the full CAAM register page.
- Maps a virtual address for the SNVS register page.
- Maps a virtual (cache coherent) address for Secure Memory.
- Registers the security violation interrupt.
- Selects the correct DMA address size for the platform, and sets DMA address masks to match.
- Identifies other pertinent interrupt connections
- Initializes all job ring instances
- If the system configuration includes a DPAA Queue Interface, that interface has frame-pop enabled.

NOTE

i.MX 6 configurations do not contain this logic.

- If the instance contains a TRNG, it's oscillator/entropy configuration is set and then "kickstarted".
- Configuration information is sent to the system console to indicate that the driver is alive, and what configuration it has assumed.
- If CONFIG_DEBUG_FS is selected in the kernel configuration, then entries are added to enable debugfs views to useful registers in the performance monitor. Register views are accessible under the caam/ctl directory at the debugfs root entry.

51.4 Job Ring Driver

The Job Ring driver is responsible for providing job execution service to higher-level drivers. It takes care of overall management of both input and output rings and interrupt service driving the output ring.

One driver call is available for higher layers to use for queueing jobs to a ring for execution:

```
int caam_jr_enqueue(struct device *dev, u32 *desc, void (*cbk)(struct device
*dev, u32 *desc, u32 status, void *areq), void *areq);
```

Arguments:

`dev` Pointer to the struct device associated with the job ring for use. In the current configuration, one or more struct device entries exist in the controller's private data block, one for each ring.

`desc` Pointer to a CAAM job descriptor to be executed. The driver will map the descriptor prior to execution, and unmap it upon completion. However, since the driver can't reasonably know anything about the data referenced by the descriptor, it is the caller's responsibility to map/flush any of this data prior to submission, and to unmap/invalidate data after the request completes.

`cbk` Pointer to a callback function that will be called when the job has completed processing.

`areq` Pointer to metadata or context data associated with this request. Often, this can contain referenced data mapping information that request postprocessing (via the callback) can use to clean up or release resources once complete.

Callback Function Arguments:

`dev` Pointer to the struct device associated with the job ring for use.

`desc` Pointer to the original descriptor submitted for execution.

`status` Completion status received back from the CAAM DECO that executed the request. Nonzero only if an error occurred. Strings describing each error are enumerated in error.c.

`areq` Metadata/context pointer passed to the original request.

Returns:

- Zero on successful job submission
- -EBUSY if the input ring was full
- -EIO if driver could not map the job descriptor

51.5 API Interface Level

CAAM module provides a connection through the Scatterlist Crypto API both for common symmetric blockciphers, and for single-pass authentication-encryption services. This table lists all installed authentication-encryption algorithms by their common name, driver name, and purpose. Note that certain platforms, such as i.MX 6, contain a low-power MDHA accelerator, which cannot support SHA384 or SHA512.

Name	Driver Name	Purpose
authenc(hmac(md5),cbc(aes))	authenc-hmac-md5-cbc-aes-caam	Single-pass authentication/encryption using MD5 and AES-CBC
authenc(hmac(sha1),cbc(aes))	authenc-hmac-sha1-cbc-aes-caam	Single-pass authentication/encryption using SHA1 and AES-CBC
authenc(hmac(sha224),cbc(aes))	authenc-hmac-sha224-cbc-aes-caam	Single-pass authentication/encryption using SHA224 and AES-CBC
authenc(hmac(sha256),cbc(aes))	authenc-hmac-sha256-cbc-aes-caam	Single-pass authentication/ encryptionusing SHA256 and AES-CBC
authenc(hmac(sha384),cbc(aes))	authenc-hmac-sha384-cbc-aes-caam	Single-pass authentication/encryption using SHA384 and AES-CBC
authenc(hmac(sha512),cbc(aes))	authenc-hmac-sha512-cbc-aes-caam	Single-pass authentication/encryption using SHA512 and AES-CBC
authenc(hmac(md5),cbc(des3_ede))	authenc-hmac-md5-cbcdes3_ede-caam	Single-pass authentication/encryption using MD5 and Triple-DES-CBC
authenc(hmac(sha1),cbc(des3_ede))	authenc-hmac-sha1-cbc-des3_ede-caam	Single-pass authentication/encryption using SHA1 and Triple-DES-CBC
authenc(hmac(sha224),cbc(des3_ede))	authenc-hmac-sha224-cbc-des3_ede-caam	Single-pass authentication/encryption using SHA224 and Triple-DES-CBC
authenc(hmac(sha256),cbc(des3_ede))	authenc-hmac-sha256-cbc-des3_ede-caam	Single-pass authentication/encryption using SHA256 and Triple-DES-CBC
authenc(hmac(sha384),cbc(des3_ede))	authenc-hmac-sha384-cbc-des3_ede-caam	Single-pass authentication/encryption using SHA384 and Triple-DES-CBC
authenc(hmac(sha512),cbc(des3_ede))	authenc-hmac-sha512-cbc-des3_ede-caam	Single-pass authentication/encryption using SHA512 and Triple-DES-CBC
authenc(hmac(md5),cbc(des))	authenc-hmac-md5-cbc-des-caam	Single-pass authentication/encryption using MD5 and Single-DES-CBC
authenc(hmac(sha1),cbc(des))	authenc-hmac-sha1-cbc-des-caam	Single-pass authentication/encryption using SHA1 and Single-DES-CBC
authenc(hmac(sha224),cbc(des))	authenc-hmac-sha224-cbc-des-caam	Single-pass authentication/encryption using SHA224 and Single-DES-CBC
authenc(hmac(sha256),cbc(des))	authenc-hmac-sha256-cbc-des-caam	Single-pass authentication/encryption using SHA256 and Single-DES-CBC
authenc(hmac(sha384),cbc(des))	authenc-hmac-sha384-cbc-des-caam	Single-pass authentication/encryption using SHA384 and Single-DES-CBC
authenc(hmac(sha512),cbc(des))	authenc-hmac-sha512-cbc-des-caam	Single-pass authentication/encryption using SHA512 and Single-DES-CBC

This table lists all installed symmetric key blockcipher algorithms by their common name, driver name, and purpose.

Name	Driver Name	Purpose
cbc(aes)	cbc-aes-caam	AES with a CBC mode wrapper
cbc(des3_ede)	cbc-3des-caam	Triple DES with a CBC mode wrapper
cbc(des)	cbc-des-caam	Single DES with a CBC mode wrapper
ecb(aes)	ecb-aes-caam	AES with a ECB mode wrapper
ecb(des3_ede)	ecb-3des-caam	Triples DES with a ECB mode wrapper
ecb(des)	ecb-des-caam	Single DES with a ECB mode wrapper
ecb(arc4)	ecb-arc4-caam	ARC4 with a ECB mode wrapper
ctr(aes)	ctr-aes-caam	AES with a CTR mode wrapper

Use of these services through the API is exemplified in the common conformance/ performance testing module in the kernel's crypto subsystem, known as tcrypt, visible in the kernel source tree at crypto/tcrypt.c.

The `caamhashmodule` provides a connection through the Scatterlist Crypto API both for common asynchronous hashes.

This table lists all installed asynchronous hashes by their common name, driver name, and purpose. Note that certain platforms, such as i.MX 6, contain a low-power MDHA accelerator, which cannot support SHA384 or SHA512.

Name	Driver Name	Purpose
sha1	sha1-caam	SHA1-160 Hash Computation
sha224	sha224-caam	SHA224 Hash Computation
sha256	sha256-caam	SHA256 Hash Computation
sha384	sha384-caam	SHA384 Hash Computation
sha512	sha512-caam	SHA512 Hash Computation
md5	md5-caam	MD5 Hash Computation
hmac(sha1)	hmac-sha1-caam	SHA1-160 Hash-based Message Authentication Code
hmac(sha224)	hmac-sha224-caam	SHA224 Hash-based Message Authentication Code
hmac(sha256)	hmac-sha256-caam	SHA256 Hash-based Message Authentication Code
hmac(sha384)	hmac-sha384-caam	SHA384 Hash-based Message Authentication Code
hmac(sha512)	hmac-sha512-caam	SHA512 Hash-based Message Authentication Code
hmac(md5)	hmac-md5-caam	MD5 Hash-based Message Authentication Code

Use of these services through the API is exemplified in the common conformance/ performance testing module in the kernel's crypto subsystem, known as tcrypt, visible in the kernel source tree at crypto/tcrypt.c.

The caamrng module installs a mechanism to use CAAM's random number generator to feed random data into a pair of buffers that can be accessed through /dev/hw_random.

/dev/hw_random is commonly used to feed the kernel's own entropy pool, which can be used internally, as an entropy source for other random data "devices".

For more information regarding support for this service, see rng-tools available in sourceforge.net/projects/gkernel/files/rng-tools.

51.6 Driver Configuration

Configuration of the driver is controlled by the following kernel confguration parameters (found under Cryptographic API -> Hardware Crypto Devices):

CRYPTO_DEV_FSL_CAAM

Enables building the base controller driver and the job ring backend.

CRYPTO_DEV_FSL_CAAM_RINGSIZE

Selects the size (e.g., the maximum number of entries) of job rings. This is selectable as a power of 2 in the range of 2-9, allowing selection of a ring depth ranging from 4 to 512 entries.

The default selection is 9, resulting in a ring depth of 512 job entries.

CRYPTO_DEV_FSL_CAAM_INTC

Enables the use of the hardware's interrupt coalescing feature, which can reduce the amount of interrupt overhead the system incurs during periods of high utilization. Leaving this disabled forces a single interrupt for each job completion, simplifying operation, but increasing overhead.

CRYPTO_DEV_FSL_CAAM_INTC_COUNT_THLD

If coalescing is enabled, selects the number of job completions allowed to queue before an interrupt is raised. This is selectable within the range of 1 to 255. Selecting 1 effectively defeats the coalescing feature. Any selection of a size greater than the job ring size forces a situation where the interrupt times out before ever raising an interrupt.

The default selection is 255.

CRYPTO_DEV_FSL_CAAM_INTC_TIME_THLD

If coalescing is enables, selects the count of bus clocks (divided by 64) before a coalescing timeout where, if the count threshold has not been met, an interrupt is raised at the end of the time period. The selection range is an integer from 1 to 65535.

The default selection is 2048.

`CRYPTO_DEV_FSL_CAAM_CRYPTO_API`

Enables Scatterlist Crypto API support for asynchronous blockciphers and for single-pass autentication-encryption operations through the API using CAAM hardware for acceleration.

`CRYPTO_DEV_FSL_CAAM_AHASH_API`

Enables Scatterlist Crypto API support for asynchronous hashing through the API using CAAM hardware for acceleration.

`CRYPTO_DEV_FSL_CAAM_RNG_API`

Enables use of the CAAM Random Number generator through the hwrng API. This can be used to generate random data to feed an entropy pool for the kernels pseudo-random number generator.

`CRYPTO_DEV_FSL_CAAM_RNG_TEST`

Enables a captive test to ensure that the CAAM RNG driver is operating and buffering random data.

51.7 Limitations

- Components of the driver do not currently build and run as modules. This may be rectified in a future version.
- Interdependencies exist between the controller and job ring backends, therefore they all must run in the same system partition. Future versions of the driver may separate out the job ring back-end as a standalone module that can run independently (and support independent API and SM instances) in its own system partition.
- The full CAAM register page is mapped by the controller driver, and derived pointers to selected subsystems are calculated and passed to higher-layer driver components. Partition-independent configurations will have to map their own subsystem pointers instead.
- Upstream variants of this driver support only Power architecture. This ARM architecture-specific port is not upstreamed at this time, although portions may be upstreamed at some point.

- TRNG kickstart may need to be moved to the bootloader in a future release, so that the RNG can be used earlier.
- The Job Ring driver has a registration and de-registration functions that are not currently necessary (and may be rewritten in future editions to provide for shutdown notifications to higher layers.
- The full CAAM function is exclusive with the Mega/Fast mix off feature in DSM. If CAAM is enabled, the Mega/Fast mix off feature needs to be disabled, and the user should "echo enabled > /sys/bus/platform/devices/2100000.aips-bus/2100000.caam/2101000.jr0/power/wakeup" after the kernel boots up, and then Mega/Fast mix will keep the power on in DSM.

51.8 Limitations in the Existing Implementation Overview

This chapter describes a prototype of a Keystore Management Interface intended to provide access to CAAM Secure Memory.

Secure memory provides a controlled and access-protected area where critical system security parameters can be stored and processed in a running system without bus-level exposure of clear secrets. Secrets can be imported into and exported from secure memory, but never exported from secure memory in their cleartext form. Instead, secrets may be exported from secure memory in a covered form, using keys never visible to the outside.

This driver, with it's kernel-level API, exposes a basic interface to allow kernel-level services access to secure memory functionality. It is split into two pieces:

- Keystore Initialization and Maintenance Interfaces
- Keystore Access Interface

The initialization and maintenance services exist to initialize and define the instance of a keystore interface. Likewise, the access interface allows kernel-level services to use the API for management of security parameters.

51.9 Initialize Keystore Management Interface

Installs a set of pointers to functions that implement an underlying physical interface to the keystore subsystem.

In the present release, a default (and hidden) suite of functions implement this interface. Future implementations of this API may provide for the installation of an alternate interface. If this occurs, an alternate to this call can be provided.

```
void sm_init_keystore(struct device *dev);
```

Arguments:

`dev` points to a `struct device` established to manage resources for the secure memory subsystem.

51.10 Detect Available Secure Memory Storage Units

Returns the number of available units ("pages") that can be accessed by the local instance of this driver. Intended for use as a resource probe.

```
u32 sm_detect_keystore_units(struct device *dev);
```

Arguments:

`dev` Points to a `struct device` established to manage resources for the secure memory subsystem.

Returns: Number of detected units available for use, 0 through n - 1 may be used with subsequent calls to all other API functions.

51.11 Establish Keystore in Detected Unit

Sets up an allocation table in a detected unit that can be used for the storage of keys (or other secrets). The unit will be divided into a series of fixed-size slots, each one of which is marked available in the allocation table. The size of each slot is a build-time selectable parameter.

No calls to the keystore access interface can occur until `sm_establish_keystore()` has been called.

`sm_establish_keystore()` should follow a call to `sm_detect_keystore_units()`.

```
int sm_establish_keystore(struct device *dev, u32 unit);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

Returns:

- Zero on successful return
- -EINVAL if the keystore subsystem was not initialized
- -ENOSPC if no memory was available for the allocation table and associated context data.

51.12 Release Keystore

Releases all resources used by this keystore unit. No further calls to the keystore access interface can be made.

```
void sm_release_keystore(struct device *dev, u32 unit);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

51.13 Allocate a Slot from the Keystore

Allocate a slot from the keystore for use in all other subsequent operations by the keystore access interface.

```
int sm_keystore_slot_alloc(struct device *dev, u32 unit, u32 size, u32*slot);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`size` Desired size of data for storage in the allocated slot.

`slot` Pointer to the variable to receive the allocated slot number, once known.

Returns:

- Zero for successful completion.
- -EKEYREJECTED if the requested size exceeds the selected slot size.

51.14 Load Data into a Keystore Slot

Load data into an allocated keystore slot so that other operations (such as encapsulation) can be carried out upon it.

```
int sm_keystore_slot_load(struct device *dev, u32 unit, u32 slot, constu8 *key_data, u32
key_length);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`key_length` Length (in bytes) of information to write to the slot.

`key_data` Pointer to buffer with the data to be loaded. Must be a contiguous buffer.

Returns:

- Zero for successful completion.
- -EFBIG if the requested size exceeds that which the slot can hold.

51.15 Demo Image Update

Encapsulate data written into a keystore slot as a Secure Memory Blob.

```
int sm_keystore_slot_encapsulate(struct device *dev, u32 unit, u32
inslot, u32 outslot, u16 secretlen, u8 *keymod, u16 keymodlen);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`inslot` Slot holding the input secret, loaded into that slot by sm_keystore_slot_load(). Note that the slot containing this secret should be overwritten or deallocated as soon as practical, since it contains cleartext at this point.

`outslot` Allocated slot to hold the encapsulated output as a Secure Memory Blob.

`secretlen` Length of the secret to be encapsulated, not including any blob storage overhead (blob key, MAC, etc.).

`keymod` Key modifier component to be used for encapsulation. The key modifier allows an extra secret to be used in the encapsulation process. The same modifier will also be required for decapsulation.

`keymodlen` Lenth of key modifier in bytes.

Returns:

- Zero on success
- CAAM job status if a failure occurs

51.16 Decapsulate Data in the Keystore

Decapsulate data in the keystore into a Black Key Blob for use in other cryptographic operations. A Black Key Blob allows a key to be used "covered" in main memory without exposing it as cleartext.

```
int sm_keystore_slot_decapsulate(struct device *dev, u32 unit, u32
inslot, u32 outslot, u16 secretlen, u8 *keymod, u16 keymodlen);
```

Arguments:

`dev` Points to a struct device established to manage resourcesfor the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`inslot` Slot holding the input data, processed by a prior call to `sm_keystore_slot_encapsulate()`, and containing a Secure Memory Blob.

`outslot` Allocated slot to hold the decapsulated output data in the form of a Black Key Blob.

`secretlen` Length of the secret to be decapsulated, without any blob storage overhead.

`keymod` Key modified component specified at the time of encapsulation.

`keymodlen` Lenth of key modifier in bytes.

Returns:

- Zero on success
- CAAM job status if a failure occurs

51.17 Read Data From a Keystore Slot

Extract data from a keystore slot back to a user buffer. Normally to be used after some other operation (e.g., decapsulation) occurs.

```
int sm_keystore_slot_read(struct device *dev, u32 unit, u32 slot, u32
key_length, u8 *key_data);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`slot` Allocated slot to read from.

`key_length` Length (in bytes) of information to read from the slot.

`key_data` Pointer to buffer to hold the extracted data. Must be a contiguous buffer.

Returns:

- Zero for successful completion.
- -EFBIG if the requested size exceeds that which the slot can hold.

51.18 Release a Slot back to the Keystore

Release a keystore slot back to the available pool. Information in the store is wiped clean before the deallocation occurs.

```
int sm_keystore_slot_dealloc(struct device *dev, u32 unit, u32 slot);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to `sm_detect_keystore_units()`.

`slot` Number of the allocated slot to be released back to the store.

Returns:

- Zero for successful completion.
- -EINVAL if an unallocated slot is specified.

Configuration of the Secure Memory Driver / Keystore API is dependent on the following kernel configuration parameters:

`CRYPTO_DEV_FSL_CAAM_SM`

Turns on the secure memory driver in the kernel build.

`CRYPTO_DEV_FSL_CAAM_SM_SLOTSIZE`

Configures the size of a secure memory "slot".

Each secure memory unit is block of internal memory, the size of which is implementation dependent. This block can be subdivided into a number of logical "slots" of a size which can be selected by this value. The size of these slots needs to be set to a value that can hold the largest secret size intended, plus the overhead of blob parameters (blob key and MAC, typically no more than 48 bytes).

The values are selectable as powers of 2, limited to a range of 32 to 512 bytes. The default value is 7, for a size of 128 bytes.

`CRYPTO_DEV_FSL_CAAM_SM_TEST`

Enables operation of a captive test / example module that shows how one might use the API, while verifying its functionality. The test module works along this flow:

- Creates a number of known clear keys (3 sizes).
- Allocated secure memory slots.
- Inserts those keys into secure memory slots and encapsulates.
- Decapsulates those keys into black keys.
- Enrcrypts DES, AES128, and AES256 plaintext with black keys. Since this uses symmetric ciphers, same-key encryption/decryption results will be equivalent.
- Decrypts enciphered buffers with equivalent clear keys.
- Compares decrypted results with original ciphertext and compares. If they match, the test reports OK for each key case tested.

Normal output is reported at the console as follows:

```
platform caam_sm.0: caam_sm_test: 8-byte key test match OK platform
caam_sm.0: caam_sm_test: 16-byte key test match OK platform caam_sm.0:
caam_sm_test: 32-byte key test match OK
```

- The secure memory driver is not implemented as a kernel module at this point in time.
- Implementation is presently limited to kernel-mode operations.

- One instance is possible at the present time. In the future, when job rings can run independently in different system partitions, a multiple instance secure memory driver should be considered.
- All storage requests are limited to the storage size of a single slot (which is of a build-time configurable length). It may be possible to allow a secret to span multiple slots so long as those slots can be allocated contiguously.
- Slot size is fixed across all pages/partitions.
- Encapsulation/Decapsulation interfaces could allow for authentication to be specified; the underlying interface does not request it.
- Encapsulation/Decapsulation interfaces return a job status; this status should be translated into a meaningful error from `errno.h`

51.19 CAAM/SNVS - Security Violation Handling Interface Overview

This chapter describes a prototype of a driver component and control interface for SNVS Security Violations. It provides a means of installing, managing, and executing application defined handlers meant to process security violation events as a response to their occurrence in a system.

SNVS allows for the continuous monitoring of a number of possible attack vectors in a running system. If the occurrence of one of these attach vectors is sensed, (e.g., a Security Violation has been detected), SNVS can, along with erasing critical security parameters and transitioning to a failure state. generate an interrupt indicating that the violation has occurred. This interrupt can dispatch an application-defined routine to take cleanup action as a consequence of the violation, such that an orderly shutdown of security services might occur.

Therefore, the purpose of this interface is to allow system-level services to install handlers for these types of events. This allows the system designer to select how he wants to respond to specific security violation causes using a simple function call written to his system-specific requirements.

51.20 Operation

For existing platforms, 6 security violation interrupt causes are possible within SNVS. 5 of these violation causes are normally wired for use, and these causes are defined as:

- SECVIO_CAUSE_CAAM_VIOLATION - Violation detected inside CAAM/SNVS
- SECVIO_CAUSE JTAG_ALARM - JTAG activity detected

- SECVIO_CAUSE_WATCHDOG - Watchdog expiration
- SECVIO_CAUSE_EXTERNAL_BOOT - External bootload activity
- SECVIO_CAUSE_TAMPER_DETECT - Tamper detection logic triggered

Each of these causes can be associated with an application-defined handler through the API provided with this driver. If no handler is specified, then a default handler will be called. This handler does no more than to identify the interrupt cause to the system console.

51.21 Configuration Interface

The following interface can be used to define or remove application-defined violation handlers from the driver's dispatch table.

51.22 Install a Handler

```
int caam_secvio_install_handler(struct device *dev, enum secvio_cause
cause, void (*handler)(struct device *dev, u32 cause, void *ext), u8
*cause_description, void *ext);
```

Arguments:

`dev` Points to SNVS-owning device.

`cause` Interrupt source cause from the above list of enumerated causes.

`handler` Application-defined handler, gets called with dev, source cause, and locally-defined handler argument

`cause_description` Points to a string to override the default cause name, this can be used as an alternate for error messages and such. If left NULL, the default description string is used. ext pointer to any extra data needed by the handler.

Returns:

- Zero on success.
- -EINVAL if an argument was invalid or unusable.

51.23 Remove an Installed Driver

```
int caam_secvio_remove_handler(struct device *dev, enum secvio_cause
cause);
```

Arguments:

`dev` Points to SNVS-owning device.

`cause` Interrupt source cause.

Returns:

- Zero on success.
- -EINVAL if an argument was invalid or unusable.

51.24 Driver Configuration CAAM/SNVS

`CRYPTO_DEV_FSL_CAAM_SECVIO`

Enables inclusion of Security Violation driver and configuration interface as part of the build configuration. Note that the driver is not buildable as a module in its present form.

Chapter 52
Remote Processor Messaging (RPMsg)

52.1 Introduction

With the newest multi-core architecture designed by using the ARM Cortex®-A series processors and the ARM Cortex®-M series processors, industrial applications can achieve greater power efficiency for a reduced carbon footprint. This reduces power consumption without performance deterioration.

A homogeneous SoC would traditionally run a single operating system (OS) that controls all the memory. The OS or a hypervisor would handle task management among available cores to maximize system utilization. Such a system is called Symmetric Multi-Processing (SMP).

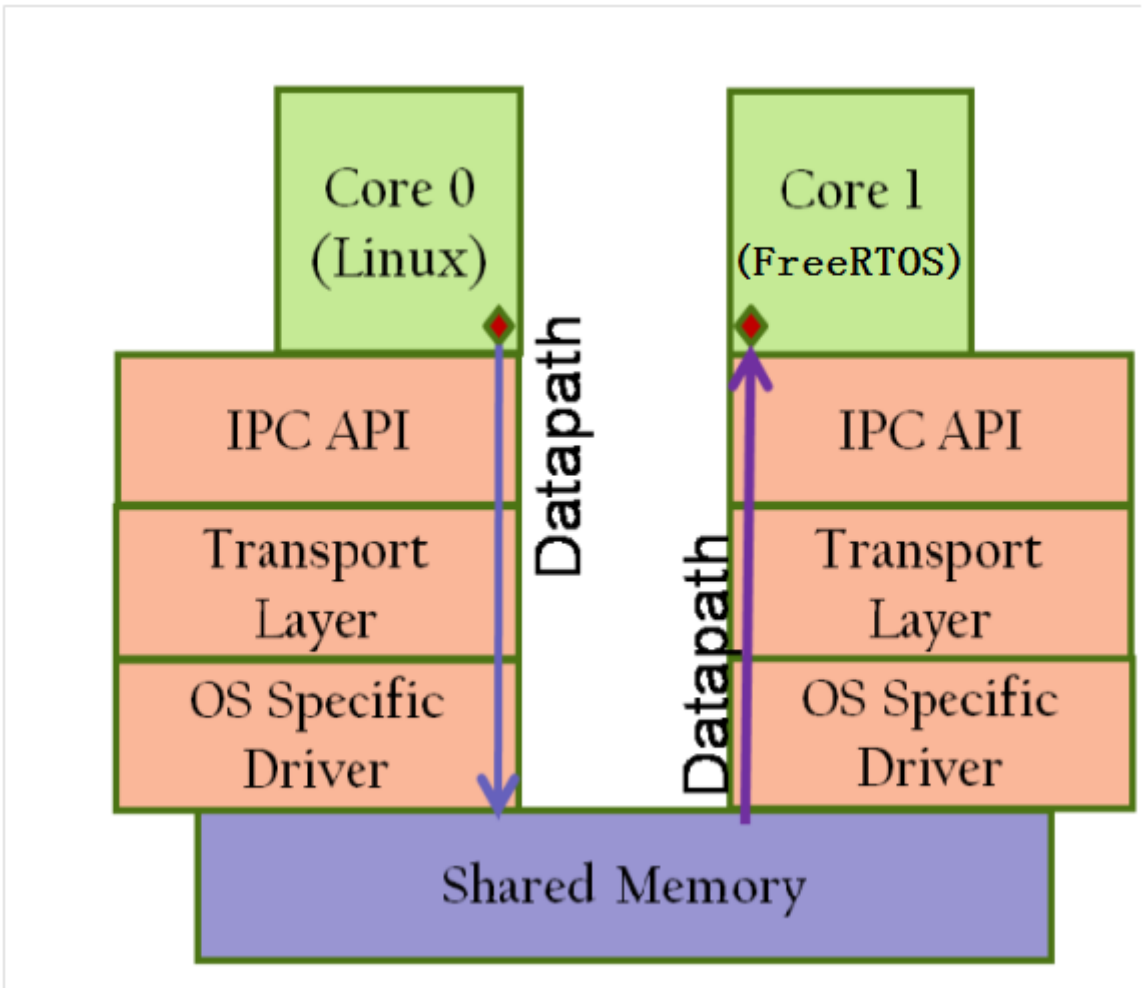
As explained previously, a heterogeneous multi-core chip where different processing cores running different instruction sets and different OSs. Each processing core handles a specific task as required. Such a system is called Asymmetric Multiprocessing (AMP). To understand the distinction between the SMP and AMP systems, it is possible for a homogeneous multi-core SoC to be an AMP system but a heterogeneous multi-core SoC cannot be an SMP system.

A multi-core architecture brings new challenges to the system design, because the software must be rewritten to distribute tasks across the available cores. In addition, all the peripheral resources need to be properly allocated to avoid resource contention and achieve efficient sharing of the data spaces between the cores. A multi-core SoC also needs mechanisms for reliable communication and synchronization among tasks running on different processing cores.

RPMsg is a virtio-based messaging bus, which allows kernel drivers to communicate with remote processors available on the system. In turn, drivers could then expose appropriate user space interfaces if needed. Every RPMsg device is a communication channel with a remote processor (so the RPMsg devices are called channels). Channels are identified by a textual name and have a local ("source") RPMsg address, and remote ("destination") RPMsg address.

As shown in the following figure, the messages pass between endpoints through bidirectional connection-less communication channels.

Figure 52-1. New multi-core, multi-OS architecture



52.2 Features

- Designed for low-latency and low overhead operation, and compliant with the Linux RPMsg framework.
- Optimized for embedded environments with constrained CPU and memory resources.
- Implementation by using shared memory without data translation or message headers.
- Application communication by using a client-server methodology.
- Dynamic allocation of the RPMsg channels.

52.3 Source Code

- Common code:

  drivers/rpmsg/virtio_rpmsg_bus.c

- i.MX platform-related code:

  arch/arm/mach-imx/imx_rpmsg.c

- i.MX RPMsg pingpong tests:

  drivers/rpmsg/imx_rpmsg_pingpong.c

- i.MX RPMsg TTY driver

  drivers/rpmsg/imx_rpmsg_tty.c

52.4 Kernel Configurations

```
For RPMSG pingpong test
Symbol: IMX_RPMSG_PINGPONG [=m]
Type  : tristate
Prompt: IMX RPMSG pingpong driver
  Location:
    -> Device Drivers
      -> Rpmsg drivers
      -> RPMSG bus driver (RPMSG [=y])

For RPMSG TTY driver
Symbol: IMX_RPMSG_TTY [=m]
Type  : tristate
Prompt: IMX RPMSG tty driver
  Location:
    -> Device Drivers
      -> Rpmsg drivers
        -> RPMSG bus driver (RPMSG [=y])
```

52.5 Running i.MX RPMsg Test Programs

To run the i.MX RPMsg test program, perform the following operations:

1. Make sure that the proper RTOS image is used for the Cortex-M4 processor.
   For example, on the i.MX 7Dual platform:
   - rpmsg_pingpong_sdk_7dsdb.bin -> pingpong test used on the i.MX 7Dual SDB board
   - rpmsg_str_echo_sdk_7dsdb.bin -> tty string echo test used on the i.MX 7Dual SDB board

- rpmsg_pingpong_sdk_7dval.bin -> pingpong test used on the i.MX 7Dual 12x12 LPDDR3 ARM2 board
- rpmsg_str_echo_sdk_7dval.bin -> tty string echo test used on the i.MX 7Dual 12x12 LPDDR3 ARM2 board

2. Load the Cortex-M4 processor RTOS image, and kick it off in U-Boot.

   Load the Cortex-M4 processor RTOS image by the TFTP server on the i.MX 7Dual platform.

   For example, on the i.MX 7Dual platform:

   ```
   dhcp 0x7f8000 10.192.242.53:rpmsg_pingpong_sdk_7dval.bin; bootaux 0x7f8000
   ```

   For i.MX 6SoloX platforms:

   a. Cortex-M4 processor RTOS runs on the QSPI NOR flash on the i.MX 6SoloX,so the Cortex-M4 processor RTOS image should be pre-programmed into the QSPI flash to kick off the Cortex-M4 processor RTOS.
   b. Create one vfat partition on the SD card when the SD card is used to boot up i.MX 6SoloX platforms.
   c. Copy the responded Cortex-M4 processor RTOS image of the i.MX 6SoloX to that partition and name it as "m4_qspi.bin".
   d. To program the Cortex-M4 processor RTOS image into the QSPI NOR flash, boot the i.MX 6SoloX platform, stop at the U-Boot environment, and run the `run update_m4_from_sd` command. Then, the responded Cortex-M4 processor RTOS image would be programmed into the QSPI NOR flash.
   e. Ensure that `uart_from_osc_clk` is contained in the Linux OS command line, when booting up AMP for the Cortex-A9 and Cortex-M4 RTOS platform. The following is an example of the Cortex-A9 processor Linux command line:

      ```
      setenv bootargs noinitrd console=ttymxc0,115200 root=/dev/nfs nfsroot=<your_nfs>,
      tcp ip=dhcp uart_from_osc_clk rw;bootz ${loadaddr} - ${fdt_addr}
      ```

3. Run the RPMsg test program.

   a. Make sure that `imx_rpmsg_pingpong.ko` and `imx_rpmsg_tty.ko` are built out.
   b. Use `insmod imx_rpmsg_pingpong.ko` or `insmod imx_rpmsg_tty.ko` to run the test program.

   NOTE

   DO NOT run different test programs at the same time.

   c. Run the following command and ensure that the RPMsg TTY receiving program is running at backend when starting RPMsg TTY tests.

      ```
      /unit_tests/mxc_mcc_tty_test.out /dev/ttyRPMSG 115200 R 100 1000 &
      ```

      Logs at the Linux OS side:

```
insmod imx_rpmsg_tty.ko
imx_rpmsg_tty rpmsg0: new channel: 0x400 -> 0x1!
Install rpmsg tty driver!
echo deadbeaf > /dev/ttyRPMSG
imx_rpmsg_tty rpmsg0: msg(<- src 0x1) deadbeaf len 8
```

Chapter 53
Sipix Display Controller (SPDC) Frame Buffer Driver

53.1 Introduction

Electrophoretic Display Timing Controller (SPDC) is a direct-drive active matrix EPD controller designed to drive Sipix panel for E-Book application. The SPDC provides control signals for the source driver and gate drivers. This IP provides a high performance, low cost solution for SiPix EPDs (Electronic Paper Display). Partial update and concurrent display updates resulting in high responsive screen changes are also implemented for these applications. The SPDC module co-works in conjunction with the ePXP IP module to form a complete display processing solution (such as rotation and flip function).

The SPDC driver supports the following features:

- Support for SPDC driver as a loadable or built-in module.
- Support for RGB565 and Y4 frame buffer formats.
- Support 800x600 resolution.
- Support for full and partial EPD screen updates.
- Support for automatic optimal waveform selection for a given update.
- Support for synchronization by waiting for a specific update request to complete.
- Support for screen updates from an alternate (overlay) buffer.
- Support for 90, 180, and 270 degree HW-accelerated frame buffer rotation.
- Support for panning (y-direction only).
- Support for automated full and partial screen updates through the Linux fb_deferred_io mechanism.
- Support for three SPDC driver display update schemes: Snapshot, Queue, and Queue and Merge.
- Support for setting the ambient temperature through either a one-time designated API call or on a per-update basis.
- Support for user control of the delay between completing all updates and powering down the SPDC.

53.2 Hardware Operation

For the detailed hardware operation of the SPDC, see the *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM).

53.3 Software Operation

The SPDC frame buffer driver is a self-contained driver module in the Linux kernel. It consists of a standard frame buffer device API coupled with a custom EPD-specific API extension which is accessible through the IOCTL interface. The combined functionality provides the user with a robust and familiar display interface while offering full control over the contents and update mode of the Sipix display.

This section covers the software operation of the SPDC driver, both through the standard frame buffer device architecture, and through the custom Sipix API extensions. Additionally, panel intialization and framebuffer formats are discussed.

53.3.1 SPDC Frame Buffer Driver Overview

The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer video hardware and allows application software to access the graphics hardware through a well-defined interface, so that the software is not required to know anything about the low-level hardware registers. The SPDC driver supports this model with one key caveat: the contents of the frame buffer are not automatically updated to the Sipix display. Instead, a custom API function call is required to trigger an update to the Sipix display. The details of this process are explained in SPDC Frame Buffer Driver Extensions.

The frame buffer driver is enabled by selecting the frame buffer option under the graphics parameters in the kernel configuration. To supplement the frame buffer driver, the kernel builder may also include support for fonts and a startup logo. The frame buffer device depends on the virtual terminal (VT) console to switch from serial to graphics mode. The device is accessed through special device nodes, located in the /dev directory, as /dev/fb*. fb0 is generally the primary frame buffer.

A frame buffer device is a memory device, such as /dev/mem, and it has features similar to a memory device. Users can read it, write to it, seek to some location in it, and mmap() it (the main use). The difference is that the memory that appears in the special file is not the whole memory, but the frame buffer of some video hardware.

The SPDC frame buffer driver (<Yocto_BuildDir>/rpm/BUILD/linux/drivers/video/mxc/mxc_spdc_fb.cdrivers/video/mxc/mxc_spdc_fb.c) interacts closely with the generic Linux frame buffer driver (drivers/video/fbmem.c).

For additional details on the frame buffer device, see documentation in the Linux kernel found in Documentation/fb/framebuffer.txt.

53.3.2 SPDC Frame Buffer Driver Extensions

Sipix display technology, in conjunction with the SPDC, has several features that distinguish it from standard LCD-based frame buffer devices. These differences introduce the need for API extensions to the frame buffer interface. The SPDC refreshes the Sipix display asynchronously and supports partial screen updates. Therefore, the SPDC requires notification from the user when the frame buffer contents have been modified and which region needs updating. Another unique characteristic of SPDC updates to the Sipix display is the long screen update latencies (between 300-980ms), which introduces the need for a mechanism to allow the user to wait for a given screen update to complete.

The custom API extensions to the frame buffer device are accessible both from user space applications and from within kernel space. The standard device IOCTL interface provides access to the custom API for user space applications. The IOCTL extensions, along with relevant data structures and definitions, can be found in include/linux/mxcfb.h. A full description of these IOCTLs can be found in the Programming Interface section Programming Interface.

For kernel mode access to the custom API extensions, the IOCTL interface should be bypassed in favor of direct access to the underlying functions. These functions are included in include/linux/mxcfb_epdc_kernel.h, and are documented in the Programming Interface section Programming Interface.

53.3.3 SPDC Panel Configuration

The SPDC driver is designed to flexibly support Sipix panels with a variety of panel resolutions, timing parameters, and waveform modes. The SPDC driver is kept panel-agnostic through the use of an SPDC panel mode structure, imx_spdc_fb_mode, which can be found in arch/arm/plat-mxc/include/mach/epdc.h.

```
struct imx_spdc_fb_mode {
            struct fb_videomode *vmode;
            struct imx_spdc_panel_init_set *init_set;
            const char
*wave_timing;
```

```
};
```

The imx_spdc_fb_mode structure consists of an fb_videomode structure and a set of EPD timing parameters. The fb_videomode structure defines the panel resolution and the basic timing parameters (pixel clock frequency, hsync and vsync margins) and the additional timing parameters in imx_spdc_fb_mode define EPD-specific timing parameters, such as the source and gate driver timings. For details on how to configure Sipix panel timing parameters, see the SPDC programming model section in the *i.MX 6SoloLite Applications Processor Reference Manual* (IMX6SLRM).

This SPDC panel mode is part of the mxc_spdc_fb_platform_data structure that is passed to the SPDC driver during driver registration.

```
struct imx_spdc_fb_platform_data {
            struct imx_spdc_fb_mode *spdc_mode;
            int num_modes;
            int (*get_pins) (void);
            void (*put_pins) (void);
            void (*enable_pins) (void);
            void (*disable_pins) (void);
};
```

In addition to the SPDC panel mode data, functions may be passed to the SPDC driver to define how to handle the SPDC pins when the SPDC driver is enabled or disabled. These functions should disable the SPDC pins for purposes of power savings.

53.3.3.1 Boot Command Line Parameters

Additional configuration for the SPDC driver is provided through boot command line parameters. The format of the command line option is as follows:

```
spdc video=mxcspdcfb:[panel_name],bpp=16
```

The SPDC driver parses these options and tries to match panel_name to the name of video mode specified in the mxc_spdc_fb_mode panel mode structure. If no match is found, then the first panel mode provided in the platform data is used by the SPDC driver. The bpp setting from this command line sets the initial bits per pixel setting for the frame buffer. A setting of 16 selects RGB565 pixel format, while a setting of 4 selects 4-bit grayscale (Y4) format.

53.3.4 SPDC Waveform Loading

The SPDC driver requires a waveform file for proper operation. This waveform file contains the waveform information needed to generate the waveforms that drive updates to the Sipix panel. A pointer to the waveform file data is programmed into the SPDC before the first update is performed.

There are two options for selecting a waveform file:

1. Select one of the default waveform files included in this BSP and built into the kernel.
2. Use a new waveform file that is specific to the Sipix panel being used.

The waveform file is loaded by the SPDC driver using the Linux firmware APIs.

53.3.4.1 Using a Default Waveform File

The quickest and easiest way to get started using an Sipix panel and the SPDC driver is to use one of the default waveform files provided in the Linux BSP. This should enable updates to several different types of Sipix panel without a panel-specific waveform file. The drawback is that optimal quality should not be expected. Typically, using a non-panel-specific waveform file for an Sipix panel results in more ghosting artifacts and overall poorer color quality.

The following default waveform files included in the BSP reside in firmware/imx/:

- spdc_pvi.fw - Default waveform for the AUO Sipix panel ERK_1_4_A01 version.

The SPDC driver attempts to load a waveform file with the name "imx/spdc_[wave_timing].fw", where wave_timing refers to the string specified in the imx_spdc_fb_mode `wave_timing` field.

53.3.4.2 Using a Custom Waveform File

To ensure the optimal Sipix display quality, use a waveform file specific to Sipix panel being used. The raw waveform file type requires conversion to a format that can be understood and read by the SPDC. This conversion script is not included as part of the BSP. Contact Freescale to acquire this conversion script.

Once the waveform conversion script has been run on the raw waveform file, the converted waveform file should be renamed so that the SPDC driver can find it and load it. The driver is going to search for a waveform file with the name "imx/spdc_[wave_timing].fw", where wave_timing refers to the string specified in the imx_spdc_fb_mode `wave_timing` field. For example, if the panel waveform is named "pvi", then the converted waveform file should be named spdc_pvi.fw.

The firmware script firmware.sh (lib/udev/firmware in the Linux root file system) contains the search path used to locate the firmware file. The default search path for firmware files is /lib/firmware;/usr/local/lib/firmware. A custom search path can be specified by modifying firmware.sh. Create an imx directory in one of these paths and add your new spdc_[wave_timing].fw file there.

NOTE

If the SPDC driver is searching for a firmware waveform file that matches the names of one of the default waveform files (see preceding chapter), it will choose the default firmware files that are built into the BSP over any firmware file that has been added in the firmware search path. Therefore, if you leave the BSP so that it builds those default firmware files into the OS image, be sure to use a panel name other than those associated with the default firmware files, since those default waveform files will be preferred and selected over a new waveform file placed in the firmware search path.

53.3.5 SPDC Panel Initialization

The framebuffer driver will not typically (see note below for exceptions) go through any hardware initialization steps when the framebuffer driver module is loaded. Instead, a subsequent user mode call must be made to request that the driver initialize itself for a specific EPD panel. To initialize the SPDC hardware and E Ink panel, an FBIOPUT_VSCREENINFO ioctl call must be made, with the xres and yres fields of the fb_var_screeninfo parameter set to match the X and Y resolution of a supported E Ink panel type. To ensure that the SPDC driver receives the initialization request, the activate field of the fb_var_screeninfo parameter should be set to FB_ACTIVATE_FORCE.

NOTE

The exception is when the FB Console driver is included in the kernel. When the SPDC driver registers the framebuffer device, the FB Console driver will subsequently make an FBIOPUT_VSCREENINFO ioctl call. This will in turn initialize the SPDC panel.

53.3.6 Grayscale Framebuffer Selection

The SPDC framebuffer driver supports the use of 4-bit grayscale (Y4) and 4-bit inverted grayscale (Y4 inverted) pixel formats for the framebuffer (in addition to the more common RGB565 pixel format). In order to configure the framebuffer format as 4-bit grayscale, the application would call the FBIOPUT_VSCREENINFO framebuffer ioctl. This ioctl takes an fb_var_screeninfo pointer as a parameter. This parameter specifies the attributes of the framebuffer and allows the application to request changes to the framebuffer format. There are two key members of the fb_var_screeninfo parameter that must be set in order to request a change to 4-bit grayscale format: bits_per_pixel and grayscale. bits_per_pixel must be set to 4, and grayscale must be set to one of the 2 valid grayscale format values: GRAYSCALE_4BIT or GRAYSCALE_4BIT_INVERTED.

The following code snippet demonstrates a request to change the framebuffer to use the Y4 pixel format:

```
fb_screen_info screen_info;
screen_info.bits_per_pixel = 4;
screen_info.grayscale = GRAYSCALE_4BIT;
retval = ioctl(fd_fb0, FBIOPUT_VSCREENINFO, &screen_info);
```

53.4 Source Code Structure

Table below lists the source files associated with the SPDC driver. These files are available in the following directory:

`drivers/video/mxc`

Table 53-1. SPDC Driver Files

File	Description
mxc_spdc_fb.c	The SPDC frame buffer driver.
mxc_spdc_fb.h	Register definitions for the SPDC module.

Table below lists the global header files associated with the SPDC driver. These files are available in the following directory:

`include/linux/`

Table 53-2. SPDC Global Header Files

File	Description
mxcfb.h	Header file for the MXC framebuffer drivers
mxcfb_epdc_kernel.h	Header file for direct kernel access to the SPDC API extension

53.5 Menu Configuration Options

The following Linux kernel configuration options are provided for the SPDC module.

- CONFIG_FB_MXC_SIPIX_PANEL includes support for the Electrophoretic Display Controller. In menuconfig, this option is available under:
  - Device Drivers > Graphics Support > Sipix Panel Framebuffer
- CONFIG_MXC_SIPIX_AUTO_UPDATE_MODE option enables support for auto-update mode which provides automated EPD updates through the deferred I/O framebuffer driver. This option is dependent on the MXC_SIPIX_PANEL option. In menuconfig, this option is available under:
  - Device Drivers > Graphics Support > Sipix Auto-update Mode Support

NOTE

This option only enables the use of auto-update mode. Turning on auto-update mode requires an additional IOCTL call using the MXCFB_SET_AUTO_UPDATE_MODE IOCTL.

- CONFIG_FB is the configuration option to include frame buffer support in the Linux kernel. In menuconfig, this option is available under:
  - Device Drivers > Graphics support > Support for frame buffer devices
  - By default, this option is Y for all architectures.
- CONFIG_FB_MXC is the configuration option for the MXC Frame buffer driver. This option is dependent on the CONFIG_FB option. In menuconfig, this option is available under:
  - Device Drivers > Graphics support > MXC Framebuffer support
  - By default, this option is Y for all architectures.
- CONFIG_MXC_PXP configuration option enables support for the PxP. The PxP is required by the SPDC driver for processing (color space conversion, rotation, auto-waveform selection) framebuffer update regions. This option must be selected for the SPDC framebuffer driver to operate correctly. In menuconfig, this option is available under:
  - Device Drivers > DMA Engine support > MXC PxP support

53.6 Programming Interface

53.6.1 IOCTLs/Functions

The SPDC Frame Buffer is accessible from user space and from kernel space. A single set of functions describes the SPDC Frame Buffer driver extension, but there are two modes for accessing these functions. For user space access, the IOCTL interface should be used, and for kernel space access, the functions should be called directly. For each function below, both the IOCTL code and the corresponding kernel function is listed.

MXCFB_SET_WAVEFORM_MODES / mxc_spdc_fb_set_waveform_modes()

Description:

Defines a mapping for common waveform modes.

Parameters:

mxcfb_waveform_modes **modes*

Pointer to a structure containing the waveform mode values for common waveform modes. These values must be configured in order for automatic waveform mode selection to function properly.

MXCFB_SET_TEMPERATURE / mxc_spdc_fb_set_temperature

Description:

Set the temperature to be used by the SPDC driver in subsequent panel updates.

Parameters:

int*32_t temperature*

Temperature value, in degrees Celsius. Note that this temperature setting may be overridden by setting the temperature value parameter to anything other than SPDC_DEFAULT_TEMP when using the MXCFB_SEND_UPDATE ioctl.

MXCFB_SET_AUTO_UPDATE_MODE / mxc_spdc_fb_set_auto_update

Description:

Select between automatic and region update mode.

Parameters:

__u32 mode

In region update mode, updates must be submitted via the MXCFB_SEND_UPDATE IOCTL.

In automatic mode, updates are generated automatically by the driver by detecting pages in frame buffer memory region that have been modified.

MXCFB_SET_UPDATE_SCHEME / mxc_spdc_fb_set_upd_scheme

Description:

Select a scheme that dictates how the flow of updates within the driver.

Parameters:

__u32 scheme

Select of the following updates schemes:

UPDATE_SCHEME_SNAPSHOT - In the Snapshot update scheme, the contents of the framebuffer are immediately processed and stored in a driver-internal memory buffer. By the time the call to MXCFB_SEND_UPDATE has completed, the framebuffer region is free and can be modified without affecting the integrity of the last update. If the update frame submission is delayed due to other pending updates, the original buffer contents will be displayed when the update is finally submitted to the SPDC hardware.

UPDATE_SCHEME_QUEUE - The Queue update scheme uses a work queue to aynchronously handle the processing and submission of all updates. When an update is submitted via MXCFB_SEND_UPDATE, the update is added to the queue, and then processed in order, as SPDC hardware resources become available. As a result, the framebuffer contents processed and updated are not guaranteed to reflect what was present in the framebuffer when the update was sent to the driver.

UPDATE_SCHEME_QUEUE_AND_MERGE - The Queue and Merge scheme uses the queueing concept from the Queue scheme, but adds a merging step. This means that before an update is processed in the work queue, it is first compared with other pending updates. If any update matches the mode and flags of the current update, and also overlaps the update region of the current update, then that update will be merged with the current update. After attempting to merge all pending updates, the final merged update will be processed and submitted.

MXCFB_SEND_UPDATE / mxc_spdc_fb_send_update

Description:

Request a region of the frame buffer be updated to the display.

Parameters:

mxcfb_update_data **upd_data*

Pointer to a structure defining the region of the frame buffer, waveform mode, and collision mode for the current update. This structure also includes a flags field to select from one of the following update options:

SPDC_FLAG_ENABLE_INVERSION - Enables inversion of all pixels in the update region.

SPDC_FLAG_FORCE_MONOCHROME - Enables full black/white posterization of all pixels in the update region.

SPDC_FLAG_USE_ALT_BUFFER - Enables updating from an alternate (non-framebuffer) memory buffer.

If enabled, the final *upd_data* parameter includes detailed configuration information for the alternate memory buffer.

MXCFB_WAIT_FOR_UPDATE_COMPLETE / mxc_spdc_fb_wait_update_complete

Description:

Block and wait for a previous update request to complete.

Parameters:

mxfb_update_marker_data *marker_data*

The update_marker value used to identify a particular update (passed as a parameter in MXCFB_SEND_UPDATE IOCTL call) should be re-used here to wait for the update to complete. If the update was a collision test update, the collision_test variable will return the result indicating whether a collision occurred.

MXCFB_SET_PWRDOWN_DELAY / mxc_spdc_fb_set_pwrdown_delay

Description:

Set the delay between the completion of all updates in the driver and when the driver should power down the SPDC and the Sipix display power supplies.

Parameters:

int32_t *delay*

Input delay value in milliseconds. To disable SPDC power down altogether, use FB_POWERDOWN_DISABLE (defined below).

MXCFB_GET_PWRDOWN_DELAY / mxc_spdc_fb_get_pwrdown_delay

Description:

Retrieve the driver's current power down delay value.

Parameters:

int32_*t delay*

Output delay value in milliseconds.

53.6.2 Structures and Defines

```
#define GRAYSCALE_4BIT                          0x3
#define GRAYSCALE_4BIT_INVERTED                 0x4

#define AUTO_UPDATE_MODE_REGION_MODE                                    0

#define AUTO_UPDATE_MODE_AUTOMATIC_MODE                                 1

#define UPDATE_SCHEME_SNAPSHOT                                          0
#define UPDATE_SCHEME_QUEUE                                             1
#define UPDATE_SCHEME_QUEUE_AND_MERGE                                   2

#define UPDATE_MODE_PARTIAL                                             0x0
#define UPDATE_MODE_FULL                                                0x1

#define WAVEFORM_MODE_AUTO                                              257

#define SPDC_DEFAULT_TEMP                                               30

#define EPDC_FLAG_ENABLE_INVERSION                                      0x01
#define EPDC_FLAG_FORCE_MONOCHROME                                      0x02
#define EPDC_FLAG_USE_ALT_BUFFER                                        0x100

#define FB_POWERDOWN_DISABLE                                            -1

struct mxcfb_rect {
    __u32 left; /* Starting X coordinate for update region */
    __u32 top; /* Starting Y coordinate for update region */
    __u32 width; /* Width of update region */
    __u32 height; /* Height of update region */
};

struct mxcfb_waveform_modes {
    int mode_init; /* INIT waveform mode */
    int mode_du; /* DU waveform mode */
    int mode_gc4; /* GC4 waveform mode */
    int mode_gc8; /* GC8 waveform mode */
    int mode_gc16; /* GC16 waveform mode */
    int mode_gc32; /* GC32 waveform mode */
};

struct mxcfb_alt_buffer_data {
    __u32 phys_addr; /* physical address of alternate image buffer */
    __u32 width; /* width of entire buffer */
    __u32 height; /* height of entire buffer */
    struct mxcfb_rect alt_update_region; /* region within buffer to update */
};

struct mxcfb_update_data {
    struct mxcfb_rect update_region; /* Rectangular update region bounds */
    __u32 waveform_mode; /* Waveform mode for update */
    __u32 update_mode; /* Update mode selection (partial/full) */
    __u32 update_marker; /* Marker used when waiting for completion */
    int temp; /* Temperature in Celsius */
    uint flags; /* Select options for the current update */
    struct mxcfb_alt_buffer_data alt_buffer_data; /* Alternate buffer data */
};
struct mxcfb_update_marker_data { __u32 update_marker; __u32 collision_test; };
```

Chapter 54
Display Content Integrity Checker (DCIC)

54.1 Introduction

The goal of the DCIC is to verify that a safety-critical information sent to a display is not corrupted.

54.2 Hardware Operation

The DCIC has the following features:

- Pixel clock up to 148.5 MHz
- Configurable polarity of Display Interface control signals
- 24-bit pixel data bus
- Up to 16 rectangular ROIs with a configurable location and size
- Independent CRC32 signature calculation for each ROI
- External controller mismatch indication signal

54.3 Software Operation

54.3.1 Source Code Structure

Table below shows the driver source files available in the directory: <Yocto_BuildDir>/linux/drivers/video/mxc/.

Table 54-1. DCIC Driver Files

File	Description
mxc_dcic.c	DCIC driver source code

54.3.2 Menu Configuration Options

The following Linux kernel configuration options are provided for this module. To get to these options, use the bitbake linux-imx -c menuconfig command. On the screen displayed, select Configure the Kernel and exit. When the next screen appears, select the following options as build-in status to enable this module:

Device Drivers -> Graphics support -> MXC DCIC

54.3.3 DTS Configuration

```
dcic_id = <0>;  /* DCIC device index 0-dcic1, i-dcic2 */
dcic_mux = "dcic-lcdif1"; /* DCIC input select */
```

Table 54-2. DCIC Input Select

Module	i.MX 6SoloX	i.MX 6Dual/6Quad
DCIC1	dcic_lvds dcic_lcdif1	dcic-ipu0-di1 dcic-lvds0 dcic-lvds1 dcic-hdmi
DCIC2	dcic_lvds dcic_lcdif2	dcic-ipu0-di0/dcic-ipu1-di0 dcic-lvds0 dcic-lvds1 dcic-mipi_dpi

54.4 Programming Interface

54.4.1 IOCTLs Functions

The DCIC driver supports the following IOCTLs:

- DCIC_IOC_CONFIG_DCIC: Configures the DCIC input CLK, VSYNC, HSYNC, and data signal polarity.
- DCIC_IOC_CONFIG_ROI: Configures the ROI block size and reference signature.
- DCIC_IOC_GET_RESULT: Gets the result of the ROI calculated signature.

54.4.2 Structures

```
struct roi_params {
     unsigned int roi_n;      /* ROI index */
     unsigned int ref_sig;   /* Reference CRC32 */
     unsigned int start_y;   /* start vertical lines of ROI */
     unsigned int start_x;  /* start horizon lines of ROI */
     unsigned int end_y;   /* end vertical lines of ROI */
     unsigned int end_x;   /* end horizon lines of ROI */
     char freeze;                /*  state of ROI */
 };
```

54.5 Unit Test

54.5.1 Source Code

The DCIC unit test is a sample for how to use DCIC to check the display content. The source located at:

<Yocto_BuildDir>/linux –test/test/mxc_dcic_test

In this unit test, there are three ROIs allocated.

NOTE

All ROIs block cannot overlay with each other.

54.5.2 DCIC CRC Calculation Functions

There are four functions in this unit test to calculate reference signature.

```
crc32_calc_18of24bit()    /* CRC calculate 18 bit of 24  */
crc32_calc_24bit()           /* CRC calculate 24  */
crc32_calc_24of16bit()  /* CRC calculate 24 bit of 16  */
crc32_calc_18of16bit() /* CRC calculate 18 bit of 16  */
```

DCIC calculates CRC according to the display bus width, but the display bus width does not always align with bytes per pixel (bpp), and the four functions above can cover different display bus widths and bpps.

54.5.3 sample

The pixel bpp in the frame buffer is 24, but the display bus width is 18. Therefore, the unit test should run with the parameter “–bw 18” as follows:

```
./mxc_dcic_test.out -bw 18 -dev 1
```

Chapter 55
ADC Driver

55.1 ADC Introduction

The features of the ADC-Digital are as follows:

- Two 12-bit ADCs
- Linear successive approximation algorithm with up to 12-bit resolution with 10/11 bit accuracy
- Up to 1MS/s sampling rate
- Up to 8 single-ended external analog inputs
- Single or continuous conversion (automatic return to idle after single conversion)
- Output Modes: (in right-justified unsigned format)
    - 12-bit
    - 10-bit
    - 8-bit
- Configurable sample time and conversion speed/power
- Conversion complete and hardware average complete flag and interrupt
- Input clock selectable from up to four sources
- Asynchronous clock source for lower noise operation with option to output the clock
- Selectable asynchronous hardware conversion trigger with hardware channel select
- Selectable voltage reference, Internal, External, or Alternate
- Operation in low power modes for lower noise operation
- Hardware average function
- Self-calibration mode

55.2 ADC External Signals

- ADC_VREFH: Voltage reference high
- ADC_VREHL: Voltage reference low
- ADC1_IN0: Analog channel 1 input 0
- ADC1_IN1: Analog channel 1 input 1

- ADC1_IN2: Analog channel 1 input 2
- ADC1_IN3: Analog channel 1 input 3
- ADC2_IN0: Analog channel 2 input 0
- ADC2_IN1: Analog channel 2 input 1
- ADC2_IN2: Analog channel 2 input 2
- ADC2_IN3: Analog channel 2 input 3

The ADC pin settings should be done in the ADCx_PCTL register. No other extra IOMUX settings are required.

55.3 ADC Driver Overview

The ADC driver is developed under the Linux IIO (Industrial I/O) driver frame. The ADC driver only provides the basic functions. The following features are supported:

- Four external inputs for each ADC controller channel
- 12 bit ADC
- Single conversion
- Hardware average
- Low power mode of ADC
- Sample rate changes in the available sample rate group

55.3.1 ADC Driver File

The ADC driver file is <Yocto_BuildDir>/linux/drivers/iio/adc/vf610_adc.c for i.MX 6UltraLite and i.MX 6SoloX, <Yocto_BuildDir>/linux/drivers/iio/adc/ad2802a.c for i.MX 7Dual.

55.3.2 Menu Configuration Options

Configure the kernel option to enable the module by menuconfig:

Device Drivers > Industrial I/O support > Analog to digital converters > 2802A ADC driver

Device Drivers > Industrial I/O support> Analog to digital converters > Freescale vf610 ADC driver

55.3.3 Programming Interface

Linux IIO provides some system interface to get the raw ADC data from the related input. Users can also set the sample rate in the available sample rate group. The ADC controllers system interface is located:

/sys/devices/soc0/soc.1/2200000.aips-bus/2280000.adc/iio:device0:

/sys/devices/soc0/soc.1/2200000.aips-bus/2284000.adc/iio:device1:

The following table lists the software interfaces.

Table 55-1. Software Interfaces

Software interface	Description
in_voltage0_raw~ in_voltage3_raw	**cat in_voltage0_raw** to get raw ADC data
sampling_frequency_available	**cat sampling_frequency_available** to get available sample rate group
in_voltage_sampling_frequency	**cat in_voltage_sampling_frequency** to show current sample rate **echo value** > **in_voltage_sampling_frequency** to set the sample rate

Chapter 56
Video Analog-to-Digital Converter (VADC)

56.1 Introduction

The video analog-to-digital converter (VADC) consists of an analog video front end (AFE), and a digital video decoder. The AFE accepts NTSC or PAL input from a device, such as an analog camera.

The two parts are configured in the VADC driver. The video decoder outputs the YUV444-formatted data.

56.2 Hardware Operation

The Video ADC has the following features:

- Internal voltage and current reference generator
- 10-bit resolution (9.5 bit ENOB at 66.5 Msps)
- 4 analog inputs, with all inputs available for CVBS
- Programmable anti-aliasing filter, gain, and clamp

The video decoder has the following features:

- NTSC/PAL decoder
- Direct data path (no complex resampling)
- Automatic standards detection
- 2D adaptive comb filter
- Datapath/clocking architecture encompasses a time base corrector for VCR signals
- Luma passband is flat to > 6MHz

56.3 Software Operation

The VADC driver is located under the Linux V4L2 architecture and it implements the V4L2 capture interfaces. Applications cannot use the camera driver directly. Instead, the applications use the V4L2 capture driver to open and close the camera for image capture.

The V4L2 capture supports the following operation:

- Capture stream mode

The following picture format is supported:

- YUV444

The following picture sizes are supported:

- PAL
- NTSC

56.3.1 Source Code Structure

Table below shows the driver source files available in the directory: <Yocto_BuildDir>/ linux/drivers/media/platform/mxc/capture

Table 56-1. VADC Driver Files

File	Description
mxc_vadc.c	VADC driver source code

56.3.2 Menu Configuration Options

The following Linux kernel configuration option is provided for this module. To get to this option, use the `bitbake linux-imx -c menuconfig` command. On the screen displayed, select Configure the Kernel and exit. When the next screen appears, select the following option to enable this module:

Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > MXC VADC support

56.3.3 DTS Configuration

VADC analog inputs can choose [0-3]. CSI1 or CSI2 can be used to capture the VADC data. They can be configured in the DTS file.

For example:

```
vadc_in = <0>;  /* VADC input select */
csi_id = <1>; /* CSI select */
```

The VADC input selected to vin1 and CSI2 is used to capture the VADC data.

56.4 Unit Test

Before running the unit test, make sure that the following modules are loaded:

- insmod fsl_csi.ko
- insmod mxc_vadc_tvin.ko
- insmod csi_v4l2_capture.ko

Run the unit test:

```
/unit_tests/csi_v4l2_tvin.out -d /dev/video<x>
```

Chapter 57
Bluetooth® BCM4339 Driver

57.1 Bluetooth Introduction

Bluetooth technology is low-cost, low-power, short-range wireless technology. It was designed as a replacement for cables and other short-range technologies like IrDA. Bluetooth wireless technology operates in personal area range that typically extends up to 10 meters. For more information about Bluetooth wireless technology, see www.bluetooth.com/.

57.2 Hardware Operation

The officially supported Bluetooth wireless chip with Freescale BSP is BCM4339 from Broadcom. The Broadcom® BCM4339 single-chip device provides the highest level of integration for a mobile or handheld wireless system with integrated single stream IEEE 802.11ac MAC/baseband/radio, Bluetooth 4.1, and FM radio receiver.

For the Bluetooth section, i.MX host interface is a high-speed 4-wire UART with flow control.

57.3 Software Operation

57.3.1 Bluetooth Driver Overview

FSL BSP uses the open source bluetooth driver from kernel 3.14.52 for BCM4339. The Bluetooth software is divided into four parts as follows:

- 4-wire UART and TTY driver: It is the communication interface with the Bluetooth module.

- Bluetooth HCI device driver: UART (H4) is a serial protocol for communication between the Bluetooth device and host. This protocol is required for most Bluetooth devices with the UART interface.
- Bluetooth kernel stack: Bluetooth framework and protocols implementation.
- Bluetooth user stack: Supplies several user-space utilities and integrate many profiles for use cases.

57.3.2 Bluetooth Driver Files

The Bluetooth driver source files are available in the kernel source directory.

- Bluetooth HCI device driver:
  - <Yocto_BuildDir>/linux/drivers/bluetooth/hci_h4.c
  - <Yocto_BuildDir>/linux/drivers/bluetooth/hci_ldisc.c
- Bluetooth kernel stack:
  - <Yocto_BuildDir>/linux/net/bluetooth/*

57.3.3 Bluetooth Stack

BlueZ is the official Linux standard Bluetooth protocol stack, it is the latest version of 5.x and it is a Bluetooth stack for Linux kernel-based family of operating systems. Its goal is to program an implementation of the Bluetooth wireless standards specifications for Linux. To use Linux Bluetooth subsystem, you need several user-space utilities like hciconfig and bluetoothd. These utilities and updates to Bluetooth kernel modules are provided in the BlueZ packages. For more information, see www.bluez.org/

BlueZ source code are available in the git: git://git.kernel.org/pub/scm/bluetooth/bluez.git. The current BSP package test pass with BlueZ 5.28.

57.3.4 Menu Configuration Options

The following Linux kernel configuration option is provided for this module:

- UART interface:
  - CONFIG_SERIAL_IMX
  - CONFIG_TTY
- HCI interface:
  - CONFIG_BT_HCIUART
  - CONFIG_BT_HCIUART_H4
- Bluetooth Stack:
  - CONFIG_BT

- CONFIG_BT_RFCOMM
- CONFIG_BT_RFCOMM_TTY
- CONFIG_BT_BNEP
- CONFIG_BT_BNEP_MC_FILTER
- CONFIG_BT_BNEP_PROTO_FILTER
- CONFIG_BT_HIDP

Chapter 58
Samsung MIPI DSI Driver

58.1 Introduction

On the i.MX 7Dual platform, the MIPI DSI module comes from Samsung. The MIPI DSI driver for the Linux OS is based on the LCDIF framebuffer driver.

This driver has two parts:

- MIPI DSI IP driver-low level interface, used to communicate with the MIPI device controller on the display panel.
- MIPI DSI display panel driver, provides an interface to configure the display panel through MIPI DSI.

58.1.1 MIPI DSI IP Driver Overview

The MIPI DSI IP driver is registered through the LCDIF framebuffer driver interface and it is not exposed to the user space.

The driver enables the platform-related regulators and clocks. It requests OS related system resources and the registers framebuffer event notifier for blank/unblank operation. Then, the driver initializes MIPI D-PHY and configures the MIPI DSI IP according to the MIPI DSI display panel. The MIPI DSI driver supports the following features:

- Compatibility with the MIPI Alliance Specification for DSI, V1.01r11
- Compatibility with the MIPI Alliance Specification for D-PHY, Version 1.00.00
- Supports up to two D-PHY data lanes
- Bidirectional Communication and Escape Mode Support through Data Lane 0
- Maximum resolution ranges up to SXGA+(1400 x 1050 @ 60 Hz, 24 bpp)
- Supports pixel format: 16 bpp, 18 bpp packed, 18 bpp loosely packed (3 byte format), and 24bpp
- End-of-Transmission Packet (EoTp) support
- Supports ultra low power mode

- Supports PMS control interface for PLL to configure byte clock frequency
- Supports Prescaler to generate escape clock from byte clock

58.1.2 MIPI DSI Display Panel Driver Overview

The MIPI DSI display panel driver implements the MIPI DSI display panel related configuration.

It uses the APIs provided by the MIPI DSI IP driver to read/write the display module registers. Usually, there is a MIPI DSI slave controller integrated on the display panel. After power-on reset, the MIPI DSI display panel needs to be configured through standard MIPI DCS command or MIPI DSI Generic command according to the manufacturer's specification.

58.1.3 Hardware Operation

The MIPI DSI module provides a high-speed serial interface between a host processor and a display module.

It has higher performance, lower power, less EMI, and fewer pins compared with legacy parallel bus. It is designed to be compatible with the standard MIPI DSI protocol. MIPI DSI is built on the existing MIPI DPI-2, MIPI DBI-2, and MIPI DCS standards. It sends pixels or commands to the peripheral and reads back status or pixel information from the peripheral. MIPI DSI serializes all pixels data, commands and events, and contains two basic modes: command mode and video mode. It uses command mode to read/write register and memory to the display controller while reading display module status information. On the other hand, it uses video mode to transmit a real-time pixel streams from the host to peripheral in high-speed mode. It also generates an interrupt when error occurs.

58.2 Software Operation

The MIPI DSI driver for the Linux OS has two parts: MIPI DSI IP driver and MIPI DSI display panel driver.

58.2.1 MIPI DSI IP Driver Software Operation

The MIPI DSI IP driver has a private structure called mipi_dsi_info. During startup, the MIPI DSI IP driver is registered with the LCDIF framebuffer driver through the field struct mxc_dispdrv_handle *dispdrv when the driver is loaded. It also registers a framebuffer event notifier with framebuffer core to perform the display panel blank/unblank operation. The field struct fb_videomode *mode and struct mipi_lcd_config *lcd_config are received from the display panel callback. The MIPI DSI IP needs this information to configure the MIPI DSI hardware registers.

After initializing the MIPI DSI IP controller and the display module, the MIPI DSI IP gets the pixel streams from LCDIF through DPI-2 interface and serializes pixel data and video event through high-speed data links for display. When there is a framebuffer blank/unblank event, the registered notifier is called to enter/leave low power mode. The MIPI DSI IP driver provides three APIs for MIPI DSI display panel driver to configure the display module.

58.2.2 MIPI DSI Display Panel Driver Software Operation

The MIPI DSI Display Panel driver enables a particular display panel through the MIPI DSI interface. The driver should provide struct fb_videomode configuration and struct mipi_lcd_config data: some MIPI DSI parameters for the display panel such as maximum D-PHY clock, numbers of data lanes and DPI-2 pixel format. Finally, the display driver needs to set up the display panel initialization routine by calling the APIs provided by MIPI DSI IP drivers.

58.3 Driver Features

The MIPI DSI driver supports the following features:

- MIPI DSI communication protocol
- MIPI DSI command mode and video mode
- MIPI DCS command operation

58.3.1 Source Code Structure

The table below shows the MIPI DSI driver source files available in the directory:

<Yocto_BuildDir>/linux/drivers/video/mxc

Table 58-1. MIPI DSI Driver Files

File	Description
mipi_dsi_samsung.c	MIPI DSI IP driver source file
mipi_dsi_samsung.h	MIPI DSI IP driver header file
mxcfb_hx8369_wvga.c	MIPI DSI Display Panel driver source file

58.3.2 Menu Configuration Options

The following Linux kernel configuration option is provided for this module. To get to this option, use the bitbake linux-imx -c menuconfigcommand. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > MXC MIPI_DSI_SAMSUNG

58.3.3 Programming Interface

The MIPI DSI Display Panel driver can use the API interface to read and write the registers of the display panel device connected to MIPI DSI link.

For more information, see <Yocto_BuildDir>/linux/driver/video/mxc/ mipi_dsi_samsung.h.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: IMXLXRM
Rev. L3.14.52_1.1.0-ga
12/2015