



# Intel® Agilex™ Hard Processor System Remote System Update User Guide

Updated for Intel® Quartus® Prime Design Suite: **21.2**



**Subscribe**

**Send Feedback**

**UG-20287 | 2021.09.03**

Latest document on the web: [PDF](#) | [HTML](#)

## Contents

---

<b>1. Overview.....</b>	<b>8</b>
1.1. Features .....	9
1.2. System Components.....	10
1.3. Glossary.....	11
<b>2. Use Cases.....</b>	<b>13</b>
2.1. Manufacturing.....	13
2.2. Application Image Boot.....	13
2.3. Factory Image Boot.....	13
2.4. Modifying the List of Application Images.....	14
2.5. Querying RSU Status.....	15
2.6. Loading a Specific Image.....	16
2.7. Protected Access to Flash.....	17
2.8. Remote System Update Watchdog.....	17
2.9. RSU Notify.....	17
2.10. Updating the Factory Image.....	18
2.11. Updating the Decision Firmware.....	19
2.12. Retrying when Configuration Fails.....	20
2.13. Querying the Decision Firmware Version.....	21
<b>3. Quad SPI Flash Layout.....</b>	<b>22</b>
3.1. High Level Flash Layout.....	22
3.1.1. Standard (non-RSU) Image Layout in Flash.....	22
3.1.2. RSU Image Layout in Flash – SDM Perspective.....	22
3.1.3. RSU Image Layout – Your Perspective.....	24
3.2. Detailed Quad SPI Flash Layout.....	25
3.2.1. RSU Image Sub-Partitions Layout.....	25
3.2.2. Sub-Partition Table Layout.....	26
3.2.3. Configuration Pointer Block Layout.....	28
3.2.4. Application Image Layout.....	29
3.3. Decision Firmware Data Max Retry Information.....	30
3.4. Firmware Version Information.....	30
<b>4. Intel Quartus Prime Software and Tool Support.....</b>	<b>32</b>
4.1. Intel Quartus Prime Pro Edition Software.....	32
4.1.1. Selecting Factory Load Pin.....	32
4.1.2. Enabling HPS Watchdog to Trigger RSU.....	34
4.1.3. Setting the Max Retry Parameter.....	34
4.2. Programming File Generator.....	35
4.2.1. Programming File Generator File Types.....	36
4.2.2. Bitswap Option.....	36
4.3. Intel Quartus Prime Programmer.....	36
<b>5. Software Support.....</b>	<b>38</b>
5.1. SDM RSU Support.....	38
5.2. Arm Trusted Firmware Support.....	38
5.3. U-Boot RSU Support.....	38
5.3.1. U-Boot RSU APIs.....	39

5.3.2. U-Boot RSU Commands.....	39
5.4. Linux RSU Support.....	41
5.4.1. Intel Service Driver.....	41
5.4.2. Intel RSU Driver.....	42
5.4.3. LIBRSU Library.....	43
5.4.4. RSU Client.....	43
<b>6. Flash Corruption - Detection and Recovery.....</b>	<b>44</b>
6.1. Decision Firmware.....	44
6.2. Decision Firmware Data.....	44
6.3. Configuration Pointer Block.....	45
6.4. Sub-Partition Table.....	45
6.5. Application Image.....	46
6.6. Factory Image.....	46
6.7. Example Maintenance Procedure.....	47
6.7.1. Maintenance Procedure.....	47
<b>7. Remote System Update Example.....</b>	<b>49</b>
7.1. Prerequisites.....	49
7.2. Building RSU Example Binaries.....	50
7.2.1. Setting up the Environment.....	52
7.2.2. Building the Hardware Projects.....	52
7.2.3. Building Arm Trusted Firmware.....	53
7.2.4. Building U-Boot.....	53
7.2.5. Creating the Initial Flash Image.....	54
7.2.6. Creating the Application Image.....	60
7.2.7. Creating the Factory Update Image.....	61
7.2.8. Creating the Decision Firmware Update Image.....	61
7.2.9. Building Linux.....	61
7.2.10. Building ZLIB.....	62
7.2.11. Building LIBRSU and RSU Client.....	62
7.2.12. Building the Root File System.....	63
7.2.13. Building the SD Card.....	63
7.3. Flashing the Binaries.....	65
7.3.1. Flashing the Initial RSU Image to QSPI.....	65
7.3.2. Writing the SD Card Image.....	65
7.4. Exercising U-Boot RSU Commands.....	65
7.4.1. Using U-Boot to Perform Basic Operations.....	65
7.4.2. Watchdog and Max Retry Operation.....	69
7.4.3. Updating the Factory Image Using U-Boot.....	71
7.4.4. Fallback on Flash Corruption of Application Images.....	73
7.4.5. Additional Flash Corruption Detection and Recovery.....	74
7.5. Exercising the RSU Client.....	81
7.5.1. Using the RSU Client to Perform Basic Operations.....	81
7.5.2. Watchdog Timeout and max retry Operation.....	83
7.5.3. Updating the Factory Image Using the RSU Client.....	86
7.5.4. Fallback on Flash Corruption of Application Images.....	87
7.5.5. Additional Flash Corruption Detection and Recovery.....	89
<b>8. Version Compatibility Considerations.....</b>	<b>98</b>
8.1. Component Versions.....	98
8.2. Component Interfaces.....	98

8.3. Component Version Compatibility.....	100
8.4. Using Multiple Intel Quartus Prime Software Versions for Bitstreams.....	101
8.5. Updating U-Boot to Support Different SSBL per Bitstream .....	101
8.5.1. Using Multiple SSBLs with SD/MMC.....	102
8.5.2. Using Multiple SSBLs with QSPI.....	102
8.5.3. U-Boot Source Code Details.....	104
<b>9. Using RSU with HPS First.....</b>	<b>106</b>
9.1. Update Hardware Designs to use HPS First.....	106
9.2. Creating Initial Flash Image for HPS First.....	107
9.3. Creating Application and Update Images for HPS First.....	108
<b>A. Configuration Flow Diagrams.....</b>	<b>109</b>
A.1. Load Image on Power Up or nConfig Flow.....	109
A.2. Try Loading Image Process.....	110
A.3. Request Specific Image Flow.....	111
A.4. Configure after Image Failure Process.....	112
A.5. Watchdog Timeout Flow.....	113
<b>B. RSU Status and Error Codes.....</b>	<b>114</b>
<b>C. U-Boot RSU Reference Information.....</b>	<b>116</b>
C.1. Configuration Parameters.....	116
C.1.1. rsu_protected_slot.....	116
C.1.2. rsu_log_level.....	116
C.1.3. rsu_spt_checksum.....	117
C.2. Error Codes.....	117
C.3. Using U-Boot RSU Without a Valid SPT or CPB.....	117
C.4. Macros.....	119
C.5. Data Types.....	119
C.5.1. rsu_slot_info.....	119
C.5.2. rsu_status_info.....	119
C.6. Functions.....	120
C.6.1. rsu_init.....	120
C.6.2. rsu_exit.....	120
C.6.3. rsu_slot_count.....	120
C.6.4. rsu_slot_by_name.....	120
C.6.5. rsu_slot_get_info.....	120
C.6.6. rsu_slot_size.....	121
C.6.7. rsu_slot_priority.....	121
C.6.8. rsu_slot_erase.....	121
C.6.9. rsu_slot_program_buf.....	121
C.6.10. rsu_slot_program_factory_update_buf.....	121
C.6.11. rsu_slot_program_buf_raw.....	122
C.6.12. rsu_slot_verify_buf.....	122
C.6.13. rsu_slot_verify_buf_raw.....	122
C.6.14. rsu_slot_enable.....	122
C.6.15. rsu_slot_disable.....	122
C.6.16. rsu_slot_load.....	123
C.6.17. rsu_slot_load_factory.....	123

C.6.18. rsu_slot_rename.....	123
C.6.19. rsu_slot_delete.....	123
C.6.20. rsu_slot_create.....	123
C.6.21. rsu_status_log.....	124
C.6.22. rsu_notify.....	124
C.6.23. rsu_clear_error_status.....	124
C.6.24. rsu_reset_retry_counter.....	124
C.6.25. rsu_dcmf_version.....	124
C.6.26. rsu_max_retry.....	125
C.6.27. rsu_dcmf_status.....	125
C.6.28. rsu_create_empty_cpb.....	125
C.6.29. rsu_restore_cpb.....	125
C.6.30. rsu_save_cpb.....	125
C.6.31. rsu_restore_spt.....	126
C.6.32. rsu_save_spt.....	126
C.6.33. rsu_running_factory.....	126
C.7. RSU U-Boot Commands.....	126
C.7.1. dtb.....	126
C.7.2. list.....	127
C.7.3. slot_by_name.....	127
C.7.4. slot_count.....	127
C.7.5. slot_disable.....	127
C.7.6. slot_enable.....	127
C.7.7. slot_erase.....	127
C.7.8. slot_get_info.....	128
C.7.9. slot_load.....	128
C.7.10. slot_load_factory.....	128
C.7.11. slot_priority.....	128
C.7.12. slot_program_buf.....	128
C.7.13. slot_program_buf_raw.....	129
C.7.14. slot_program_factory_update_buf.....	129
C.7.15. slot_rename.....	129
C.7.16. slot_delete.....	129
C.7.17. slot_create.....	129
C.7.18. slot_size.....	130
C.7.19. slot_verify_buf.....	130
C.7.20. slot_verify_buf_raw.....	130
C.7.21. status_log.....	130
C.7.22. update.....	130
C.7.23. notify.....	131
C.7.24. clear_error_status.....	131
C.7.25. reset_retry_counter.....	131
C.7.26. display_dcmf_version.....	131
C.7.27. display_dcmf_status.....	131
C.7.28. display_max_retry.....	132
C.7.29. restore_spt.....	132
C.7.30. save_spt.....	132

C.7.31. create_empty_cpb.....	132
C.7.32. restore_cpb.....	132
C.7.33. save_cpb.....	133
C.7.34. check_running_factory.....	133
<b>D. LIBRSU Reference Information.....</b>	<b>134</b>
D.1. Configuration File.....	134
D.2. Error Codes.....	135
D.3. Using LIBRSU Without a Valid SPT or CPB.....	135
D.4. Macros.....	137
D.5. Data Types.....	137
D.5.1. rsu_slot_info.....	137
D.5.2. rsu_status_info.....	138
D.5.3. rsu_data_callback.....	138
D.6. Functions.....	138
D.6.1. librsu_init.....	138
D.6.2. librsu_exit.....	138
D.6.3. rsu_slot_count.....	138
D.6.4. rsu_slot_by_name.....	139
D.6.5. rsu_slot_get_info.....	139
D.6.6. rsu_slot_size.....	139
D.6.7. rsu_slot_priority.....	139
D.6.8. rsu_slot_erase.....	139
D.6.9. rsu_slot_program_buf.....	140
D.6.10. rsu_slot_program_factory_update_buf.....	140
D.6.11. rsu_slot_program_file.....	140
D.6.12. rsu_slot_program_factory_update_file.....	140
D.6.13. rsu_slot_program_buf_raw.....	141
D.6.14. rsu_slot_program_file_raw.....	141
D.6.15. rsu_slot_verify_buf.....	141
D.6.16. rsu_slot_verify_file.....	141
D.6.17. rsu_slot_verify_buf_raw.....	141
D.6.18. rsu_slot_verify_file_raw.....	142
D.6.19. rsu_slot_program_callback.....	142
D.6.20. rsu_slot_program_callback_raw.....	142
D.6.21. rsu_slot_verify_callback.....	142
D.6.22. rsu_slot_verify_callback_raw.....	142
D.6.23. rsu_slot_copy_to_file.....	143
D.6.24. rsu_slot_enable.....	143
D.6.25. rsu_slot_disable.....	143
D.6.26. rsu_slot_load_after_reboot.....	143
D.6.27. rsu_slot_load_factory_after_reboot.....	143
D.6.28. rsu_slot_rename.....	144
D.6.29. rsu_slot_delete .....	144
D.6.30. rsu_slot_create .....	144
D.6.31. rsu_status_log.....	144
D.6.32. rsu_notify.....	144
D.6.33. rsu_clear_error_status.....	145

D.6.34. rsu_reset_retry_counter.....	145
D.6.35. rsu_dcmf_version.....	145
D.6.36. rsu_max_retry .....	145
D.6.37. rsu_dcmf_status .....	146
D.6.38. rsu_save_spt .....	146
D.6.39. rsu_restore_spt .....	146
D.6.40. rsu_save_cpb .....	146
D.6.41. rsu_create_empty_cpb .....	146
D.6.42. rsu_restore_cpb .....	147
D.6.43. rsu_running_factory .....	147
D.7. RSU Client Commands.....	147
D.7.1. count.....	147
D.7.2. list.....	147
D.7.3. size.....	147
D.7.4. priority.....	148
D.7.5. enable.....	148
D.7.6. disable.....	148
D.7.7. request.....	148
D.7.8. request-factory.....	148
D.7.9. erase.....	149
D.7.10. add.....	149
D.7.11. add-factory-update.....	149
D.7.12. add-raw.....	149
D.7.13. verify.....	149
D.7.14. verify-raw.....	150
D.7.15. copy.....	150
D.7.16. log.....	150
D.7.17. notify.....	150
D.7.18. clear-error-status.....	150
D.7.19. reset-retry-counter.....	151
D.7.20. display-dcmf-version.....	151
D.7.21. display-dcmf-status .....	151
D.7.22. display-max-retry .....	151
D.7.23. create-slot .....	152
D.7.24. delete-slot .....	152
D.7.25. restore-spt .....	152
D.7.26. save-spt .....	152
D.7.27. create-empty-cpb .....	152
D.7.28. restore-cpb .....	153
D.7.29. save-cpb .....	153
D.7.30. check-running-factory .....	153
D.7.31. help.....	153
<b>E. Combined Application Images.....</b>	<b>154</b>
E.1. Creating the Combined Application Image.....	155
E.2. Using the Combined Application Image.....	155
<b>15. Document Revision History for the SoC Remote System Update User Guide.....</b>	<b>157</b>

## 1. Overview

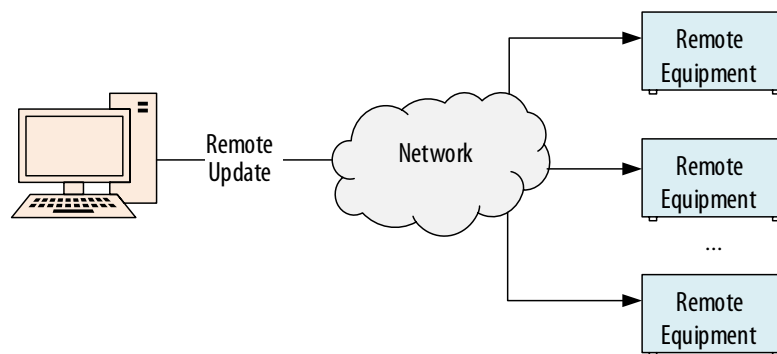
In the remote system update (RSU) implementation for active configuration schemes for all Intel® Agilex™ SoC devices, either the HPS drives the RSU or the FPGA drives the RSU. When the HPS drives the RSU, the RSU allows you to reconfigure the QSPI configuration bitstream for an Intel Agilex SoC device once a new version is available on the target equipment.

**Note:** It is your responsibility to deploy the image over the network to the target remote equipment.

The RSU performs configuration error detection during and after the reconfiguration process. If errors in the application images prevent reconfiguration, then the configuration reverts to the factory image and provides error status information.

If the configuration bitstream in the QSPI flash is corrupted rendering the device non-functional, the only method to recover the device is to connect to the device over JTAG and re-program the QSPI flash. However, this method may not be available if the system does not have a JTAG connector or if the target equipment is in a remote or hard to access location.

**Figure 1. Typical Remote System Update Usage**



This document details the Intel Agilex HPS remote system update solution and provides an update example using the *Intel Agilex F-Series Transceiver-SoC Development Kit User Guide*.

For more information about when the FPGA drives the RSU, refer to the *Intel Agilex Configuration User Guide*.

**Note:** There are references to "Stratix 10 drivers" in this document, because these drivers are compatible for both Intel Stratix® 10 and Intel Agilex SoC devices.<sup>(1)</sup>



### Related Information

- [Intel Agilex Configuration User Guide](#)
- [Intel Agilex F-Series Transceiver-SoC Development Kit User Guide](#)

## 1.1. Features

The remote system update solution:

- Provides support for creating the initial flash image for a system to support RSU. The image is created offline, before the device is deployed in the field.
- Provides you with the ability to load a specific application image or the factory image.<sup>(2)</sup>
- Allows a set of application images to be tried in a specific order until one is successful.<sup>(3)</sup> <sup>(4)</sup>
- Loads a factory image if no application image is available, or all application images failed.<sup>(5)</sup>
- Provides a pin to force load the factory image instead of the highest priority application image if asserted during a configuration caused by a POR or `nCONFIG` event.
- Provides you with the ability to add and remove application images.
- Provides you with the ability to change the order in which the application images are initially loaded.
- Provides you with information about the image that is currently running, and errors that the RSU has encountered.
- Provides the RSU Notify feature, which allows a 16-bit value reported by HPS software to survive when the current image fails due to a watchdog timeout. The value can then be queried from HPS software after the next image is loaded.

---

<sup>(1)</sup> The Linux\* Community has not approved creating duplicate drivers for Intel Agilex, at this time. Since Intel Stratix 10 and Intel Agilex SoC devices are compatible, it is permissible to use the Stratix 10 drivers.

<sup>(2)</sup> The factory and application images are also called bitstreams and typically contain the FPGA fabric configuration, the SDM firmware, and the HPS First Stage Bootloader. **Note:** For HPS first mode, the FPGA fabric configuration is omitted.

<sup>(3)</sup> When configuration succeeds, the HPS First Stage Bootloader is loaded and the HPS is taken out of reset (or just the HPS and HPS EMIF I/Os when in HPS first mode). Optionally, an HPS watchdog timeout can be treated as a RSU failure too.

<sup>(4)</sup> A maximum of seven application images can be specified at image creation time, but more can be added later. The maximum number of application images is 126; but typically only a few are used because flash memory size is limited.

<sup>(5)</sup> If the factory image also fails to configure, the SDM clears the FPGA and HPS and the device remains not configured.

- Provides the `max_retry` option, which allows each application image and the factory image the option to be tried up to three times. For more information, refer to the *Retrying when Configuration Fails* section.
- Provides a flow to update the factory image.
- Provides a flow for updating the decision firmware.

For a description of the image selection flow that occurs when the RSU enabled device is configured as a result of a power-up or `nCONFIG` event, refer to the *Configuration Flow Diagrams* appendix.

For more information about the watchdog timer, refer to the *Remote System Update Watchdog* and *Enabling HPS Watchdog to Trigger RSU* sections.

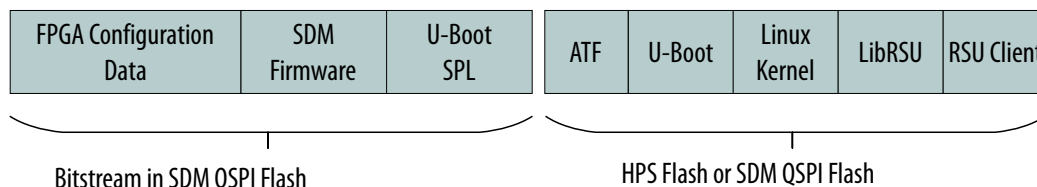
### Related Information

- [Intel Agilex Configuration User Guide](#)
- [Remote System Update Watchdog](#) on page 17
- [Enabling HPS Watchdog to Trigger RSU](#) on page 34
- [Retrying when Configuration Fails](#) on page 20

## 1.2. System Components

Figure 2 on page 10 describes the typical components of an Intel Agilex SoC based system using RSU.

**Figure 2. System Components**



The bitstream is stored in the configuration flash device (QSPI) connected to the SDM pins. The HPS software is typically stored in a mass storage flash device (SD/eMMC/NAND) connected to the HPS pins but can also be stored in the flash device connected to the SDM pins.

**Table 1. System Components**

Component	Description
FPGA Configuration Data	When using FPGA first mode, this component of the bitstream contains the full FPGA and I/O configuration data. When using HPS first mode, it contains just the HPS and HPS EMIF I/Os .
SDM Firmware	Firmware for the Secure Device Manager Provides commands for managing RSU. These commands are not directly accessible to you. Instead both U-Boot and Linux offer services for accessing the functionality offered by the SDM commands indirectly.
U-Boot SPL	The HPS First Stage Bootloader Initializes hardware and loads the HPS Second Stage Bootloader.
ATF	Arm Trusted Firmware Provides secure services to U-Boot and Linux through SMC (Secure Monitor Call) <sup>(6)</sup>
continued...	

Component	Description
U-Boot	HPS Second Stage Bootloader: <ul style="list-style-type: none"> <li>• Loads and boots the operating system</li> <li>• Provides RSU APIs and command line capabilities.</li> <li>• Uses ATF SMC calls to indirectly access the SDM services.</li> </ul>
Linux Drivers	Intel RSU driver uses the functionality provided by U-Boot SMC and makes services available to applications running on Linux.
LIBRSU	Linux user space library providing APIs for managing RSU.
RSU Client	Linux sample application which uses LIBRSU for managing RSU.

Intel provides an RSU solution that focuses on the reliable update of the components that are part of the configuration bitstream, and are located in SDM flash. It is your responsibility to devise a scheme for the reliable update of the rest of the system components.

For information about version compatibility requirements of various system components, refer to the *Version Compatibility Considerations* section.

### Related Information

[Version Compatibility Considerations](#) on page 98

## 1.3. Glossary

Terms	Description
Secure Device Manager (SDM)	Controller which manages device configuration and security.
Firmware	Firmware that runs on SDM. Implements various functions including: <ul style="list-style-type: none"> <li>• FPGA configuration</li> <li>• Voltage regulator configuration</li> <li>• Temperature measurement</li> <li>• HPS software load</li> <li>• HPS Reset</li> <li>• RSU</li> <li>• Device security</li> </ul> Each application image and the factory image have their firmware at offset zero in the image.
Decision firmware	SDM firmware used to implement the RSU feature, including identifying and loading the highest priority image. Previous versions of this user guide refer to decision firmware as DCMF. Four identical copies reside in flash, starting at address zero.
Decision firmware data	Data structure stored in flash containing the following information used by the decision firmware: <ul style="list-style-type: none"> <li>• Factory load pin location assignment</li> <li>• PLL settings for external clock source</li> <li>• Quad SPI pin assignments</li> <li>• Number of times the SDM attempts to load the same image.</li> </ul>

*continued...*

- <sup>(6)</sup> On Cortex-A53, there are four execution levels: EL0-Application, EL1-OS, EL2-Hypervisor, and EL3-Secure Monitor. Interacting with SDM is only allowed for software running at EL3. ATF runs at EL3, U-Boot and Linux run at EL1. For U-Boot and Linux to communicate with the SDM, they have to issue an SMC trap to a handler left resident by ATF. Then that handler runs at EL3 and is able to communicate with the SDM.

Terms	Description
	This data structure is populated by the Intel Quartus® Prime Programming File Generator with the information retrieved from the factory image SOF file.
Decision firmware update image	An image that updates the following RSU items in flash: <ul style="list-style-type: none"> <li>Decision firmware</li> <li>Decision firmware data</li> </ul>
Configuration pointer block (CPB)	List of application image addresses, in order of priority.
Sub-partition table (SPT)	Data structure to facilitate the management of the flash storage.
Slot/Partition	Area of flash, defined at image creation time, which can contain an application image or user data.
Application image	Configuration bitstream that implements your design.
Factory image	The fallback configuration bitstream that the RSU loads when all attempts to load an application image fail.
Initial RSU flash image	Contains the factory image, the application images, the decision firmware, and the associated RSU data structures.
Factory update image	An image that updates the following RSU-related items in flash: <ul style="list-style-type: none"> <li>Factory image</li> <li>Decision firmware</li> <li>Decision firmware data</li> </ul>

## 2. Use Cases

---

This section describes the main use cases for the Remote System Update.

### 2.1. Manufacturing

At manufacturing time, you must provide the flash partitioning information, a factory image and one or more application images. A tool called "Programming File Generator" uses this input to create an initial RSU flash image file. You must program the flash with this image file to prepare the device for remote system update. Both FPGA first and HPS first image types are supported.

### 2.2. Application Image Boot

The application image performs your application function. Remote system update allows you to safely switch a system from one application image to the next without restarting and risk of failure if one of the application images is corrupted or contains serious bugs.

The device maintains, in flash, a list of the addresses where application images are present in flash. This list is known as the configuration pointer block.

When attempting to load an application image the SDM traverses the configuration pointer block in reverse order. It attempts to load the first image, and if this load is successful then the image is now in control of the device.

If loading the first image is unsuccessful, then the SDM attempts to load the second image. If this image also fails, the SDM continues to advance to the next image until it reaches a successful image or all images fail to load. If no image is successful then the SDM loads the factory image.

### 2.3. Factory Image Boot

The purpose of the factory image is to provide enough functionality to allow a device whose application images have all been corrupted (or replaced with broken application images) to obtain a new application image and program that image into QSPI.

Once the factory image is loaded, new application images could be programmed in QSPI to replace the failed ones, from either U-Boot or Linux.

The SDM loads the factory image in the following situations:

- You assign the function LOADFACTORY to an SDM pin and assert the pin soon after a POR or nCONFIG release.
- All SDM attempts to load the application images fail.
- You request the factory image to be loaded.
- Decision firmware detects that the decision firmware data is corrupted.
- Decision firmware detects that both CPBs are corrupted.

When loading the factory image, the configuration system treats it in the same way as it does an application image.

## 2.4. Modifying the List of Application Images

The SDM uses the configuration pointer block to determine priority of application images.

The pointer block operates by taking into account the following characteristics of quad SPI flash memory:

- On a sector erase, all the sector flash bits become 1's.
- A program operation can only turn 1's into 0's.

The pointer block contains an array of values which have the following meaning:

- All 1's – the entry is unused. The client can write a pointer to this entry. This is the state after a quad SPI erase operation occurs on the pointer block.
- All 0's – the entry has been previously used and then canceled.
- A combination of 1's and 0's – a valid pointer to an application image.

When the configuration pointer block is erased, all entries are marked as unused. To add an application image to the list, the client finds the first unused location and writes the application image address to this location. To remove an application image from the list, the client finds the application image address in the pointer block list and writes all 0's to this address.

If the configuration pointer block runs out of space for new application images, the client compresses the pointer block by completing the following actions:

1. Read all the valid entries from the configuration
2. Erase the pointer block
3. Add all previously valid entries
4. Add the new image

When using HPS to manage RSU, both the U-Boot and LIBRSU clients implement the block compression. For designs that drive RSU from FPGA logic, you can implement pointer block compression many different ways, including Nios® II code, a scripting language, or a state machine.

Pointer block compression does not occur frequently because the pointer block has up to 508 available entries.

There are two configuration pointer blocks: a primary (CPB0) and a backup (CPB1). Two blocks enable the list of application images to be protected if a power failure occurs just after erasing one of them. When a CPB is erased and re-created, the

header is written last. The CPB header is checked prior to use to prevent accidental use if a power failure occurred. For more information, refer to the *Configuration Pointer Block Layout* topic. When compressing, the client compresses (erases and rewrites) the primary CPB completely. Once the primary CPB is valid, it is safe to modify the secondary CPB. When rewriting, the magic number at the start of a CPB is the last word written in the CPB. (After this number is written only image pointer slot values can be changed.)

When the client writes the application image to flash, it ensures that the pointers within the main image pointer of its first signature block are updated to point to the correct locations in flash. When using HPS to manage RSU, both the U-Boot and LIBRSU clients implement the required pointer updates. For more information, refer to the *Application Image Layout* section.

**Note:** In order to successfully perform CPB compression, the HPS software (U-Boot or Linux) must be configured to have a QSPI erase granularity of 32 KB or less. When configured with a coarser erase granularity (like 64 KB for example), the operation fails. All supported flash devices offer erase granularities of 4 KB, 32 KB, and 64 KB, and the default for the current HPS software is 4 KB.

#### Related Information

- [Configuration Pointer Block Layout](#) on page 28
- [Application Image Layout](#) on page 29

## 2.5. Querying RSU Status

The SDM firmware offers a command for querying the RSU status, providing the following information:

- RSU status code
- Address of the currently running image
- Address of the last failing image (sticky)
- Error code for last failing image (sticky)
- Error location for last failing image (sticky)
- Decision firmware RSU interface version
- Current image (application or factory) firmware RSU interface version
- retry counter value
- Index of last used decision firmware copy

The SDM firmware also offers commands for the following:

- Clearing the sticky fields related to last failing image
- Resetting the retry counter value to zero

The functionality offered by all of the above SDM commands is available from both U-Boot and LIBRSU.

For more information, refer to the [RSU Status and Error Codes](#) on page 114.

#### Related Information

[RSU Status and Error Codes](#) on page 114

## 2.6. Loading a Specific Image

The SDM firmware provides a command to load a specific image from flash. The image can be the factory image or one of the application images. You have access to this functionality from both U-Boot and LIBRSU.

If the requested image fails to load, the SDM tries to reload the image that was running at the time the command is issued. If this image also fails to load, the SDM then tries to load the highest priority image in the CPB. If the highest priority image in the CPB was the failing image, then it tries to load the second image in the CPB and continues to traverse the pointer table in priority order. If there are no more images left to try in the CPB, then the SDM tries to load the factory image.

For a description of the flow for requesting a specific image, refer to the *Configuration Flow Diagrams* appendix.

Enabling the `max_retry` feature allows each image to be tried multiple times. For more information, refer to the *Retrying when Configuration Fails* section.

**Note:** The effect of loading a new image on HPS is similar with a cold reset. However, the SDM HPS\_COLD\_nRESET pin (if configured) is not asserted by SDM as it would be on a regular cold reset.

Calling the SDM command to load a specific image causes the following sticky fields from the state reported by SDM to be cleared:

- `failed_image`
- `error_details`
- `error_location`
- `state`

When U-Boot issues this SDM command, the SDM immediately resets and wipes both the FPGA and HPS, then it proceeds to load the specified image.

When LIBRSU APIs `rsu_slot_load_after_reboot` or `rsu_slot_load_factory_after_reboot` are called, the command is not immediately sent to SDM as this would cause Linux to abruptly stop. Instead, these LIBRSU APIs cause the ATF SMC handler to make a note to call the SDM command with the specified address next time the `reboot` Linux command is executed. This enables the kernel to shut down gracefully before the requested image is loaded.

**Note:** Unless the Linux `reboot` command is issued, calling these LIBRSU APIs has no effect. If the Linux `reboot` command is configured to trigger a warm reset by passing the `reboot=warm` parameter to the kernel, calling these LIBRSU APIs has also no effect.  
(7)

For more information, refer to [RSU Status and Error Codes](#) on page 114.

### Related Information

- [RSU Status and Error Codes](#) on page 114

---

(7) This depends on the reset implementation in `linux-socfpga`.



- [Retrying when Configuration Fails](#) on page 20

## 2.7. Protected Access to Flash

After the HPS acquires QSPI flash ownership in the bootloader, the HPS has full access to QSPI and can potentially corrupt the flash.

In order to minimize the risk of rendering the system non-operational, decision firmware, decision firmware data, and factory image areas are not exposed as Linux Memory technology device (MTD) devices.

Another option to protect the flash may be to use QSPI vendor specific commands to mark the decision firmware, decision firmware data, and factory image areas as read-only. However, that may result in a more complex procedure for updating the decision firmware, decision firmware data, and factory image areas.

## 2.8. Remote System Update Watchdog

The HPS watchdog timer can be used to trigger a reset if it is not serviced periodically. The desired behavior with respect to the RSU flow on such a reset can be selected from Intel Quartus Prime tools to be one of the following:

- Trigger a cold reset – the SDM loads the HPS First Stage Bootloader (FSBL) from the current image and starts it.
- Trigger a warm reset – the SDM causes the HPS to re-start the HPS FSBL, without reloading it.
- Trigger a remote system update reconfiguration event – the SDM considers the last loaded application image a failure and then loads the next application image in the CPB, or loads the factory image, if the list is exhausted.

For more information about how to select the HPS watchdog behavior, refer to the *Intel Quartus Prime Pro Edition* section.

By default, the watchdog is enabled and serviced periodically in both U-Boot SPL and U-Boot.

It is your responsibility to enable and service the watchdog in Linux, if desired.

**Note:** The current application image is not removed from the CPB when a watchdog timeout occurs. This means that the image is not permanently marked as unusable and can be tried again (for example after a POR or `nCONFIG` even happens).

### Related Information

[Intel Quartus Prime Pro Edition Software](#) on page 32

## 2.9. RSU Notify

The SDM provides a command called `RSU Notify` that the HPS software can use to let the SDM know the current HPS software state as a 16-bit numerical value.

If the HPS watchdog timeout triggers an RSU failure, the next application or factory image is loaded. When the HPS queries the RSU state, after the next load, the top 16 bits are `0xf006` and the bottom 16 bits contain the last `RSU Notify` value reported to the SDM before the watchdog timeout.

The `RSU Notify` command is automatically called with a value of:

- 1—from FSBL just before SSBL is entered
- 2—from SSBL just before the control is passed to the operating system

You can also call `RSU Notify` with a custom value from either U-Boot or LIBRSU. Avoid using values of 0, 1, or 2, to not conflict with the automatically reported values.

Decoding the reported `RSU Notify` value:

Value	Description
0	FSBL either did not complete or did not reach the point of launching SSBL.
1	FSBL did complete, but SSBL either did not complete or did not reach the point of launching the operating system.
2	Both FSBL and SSBL are complete and attempting to launch the operating system.
Custom	You reported this custom value from either U-Boot or LIBRSU.

**Note:** Only the last reported data before the watchdog timeout is recorded.

**Note:** This data is lost on `nConfig` or POR. It only survives a reconfiguration caused by a watchdog timeout, when the watchdog is configured to trigger an RSU reconfiguration.

## 2.10. Updating the Factory Image

The factory image can be updated by running a factory update image.

**Note:** In addition to the factory image, the factory update image also updates the decision firmware and decision firmware data.

In order to make your system ready to support updating the factory image you need to have an available slot in flash that is the size of the maximum factory image you anticipate using plus 512 KB.

You may temporarily use an application image slot for the update procedure, because application images are typically larger than factory images. However, that means one less application image slot is available during the factory image update procedure.

The procedure for updating the factory image and decision firmware is as follows:

1. Create a factory update image with the Programming File Generator, using the new factory image SOF file as input. The factory update image contains the new factory image, new decision firmware, new decision firmware data, and special firmware to perform the actual update.
2. Deploy the factory update image on the remote system.
3. Write the factory update image to the flash and make it the highest priority, with either U-Boot or LIBRSU. Because the factory update image has a different format than application images, use the following LIBRSU APIs:  
`rsu_slot_program_factory_update_buf` and `rsu_slot_program_factory_update_file`. There are also U-Boot equivalents.

4. Pass control to the factory update image by requesting it to be loaded or by toggling nCONFIG.
5. The factory update image proceeds and replaces the factory image, decision firmware and decision firmware data with the new copies, then erases itself from the CPB.
6. At the end, the factory update image loads the new highest priority image in the CPB, or the factory image if the CPB is empty.

The factory update flow is resilient to power loss. If the power is lost during the update, the next time the power is back up, the factory update image resumes the update process from where it stopped.

For examples on how to perform the factory image update from both U-Boot and Linux, refer to the *Remote System Update Example* section.

For API reference information, refer to the *LIBRSU Reference Information* section, located in the *LIBRSU Reference Information* appendix.

#### Related Information

- [Remote System Update Example](#) on page 49
- [LIBRSU Reference Information](#) on page 134

## 2.11. Updating the Decision Firmware

The decision firmware can be updated by running a decision firmware update image.

**Note:** In addition to the decision firmware, the decision firmware update image also updates the decision firmware data.

In order to make your system ready to support updating the decision firmware you need to have an available slot in flash that is 512 KB or larger.

You may temporarily use an application image slot for the update procedure. However, that means one less application image slot is available during the decision firmware update procedure.

The procedure for updating the decision firmware is as follows:

1. Create a decision firmware update image with the Programming File Generator. Use a factory SOF file as input, from which the parameters for the decision firmware data are taken. The firmware update image contains the new decision firmware, new decision firmware data, and special firmware to perform the actual update.
2. Deploy the decision firmware update image on the remote system.
3. Write the decision firmware update image to the flash and make it the highest priority, with either U-Boot or LIBRSU. Because the decision firmware update image has a different format than application images, use the following LIBRSU APIs: `rsu_slot_program_factory_update_buf` and `rsu_slot_program_factory_update_file`. There are also U-Boot equivalents.

4. Pass control to the decision firmware update image by requesting it to be loaded or by toggling `nCONFIG`.
5. The decision firmware update image proceeds and replaces the decision firmware and decision firmware data with the new copies, then erases itself from the CPB.
6. At the end, the decision firmware update image loads the new highest priority image in the CPB, or the factory image if the CPB is empty.

The decision firmware update flow is resilient to power loss. If the power is lost during the update, the next time the power is back up, the decision firmware update image resumes the update process from where it stopped.

For examples on how to perform the decision firmware update from both U-Boot and Linux, refer to the *Remote System Update Example* section.

For API reference information, refer to the *LIBRSU Reference Information* section, located in the *LIBRSU Reference Information* appendix.

## 2.12. Retrying when Configuration Fails

The `max retry` option allows you the ability to configure the number of times both the application images and the factory image are tried again, if any configuration failure occurs, such as:

- Flash corruptions
- Authentication and encryption errors
- RSU watchdog timeouts

The `max retry` option is selected in Intel Quartus Prime for the factory image project and is stored in the decision firmware data structure in flash. The default value for `max retry` is 1, which means each image is tried only once. The maximum value for `max retry` is 3, which means each image is tried up to three times.

**Note:** Although the parameter is called `max retry`, its value actually denotes the total number of times each image is tried.

You can query the value of the `max retry` parameter from both U-Boot and LIBRSU. You must first query it in U-Boot for the value to also be available in Linux for LIBRSU.

The SDM maintains the `retry counter`, which counts the number of times the current image has failed to configure. Once that counter reaches `max retry`, the next image is loaded. You can query the value of the `retry counter` and also request it to be reset to zero from both U-Boot and LIBRSU.

The retry behavior applies to all configuration failures, including:

- Failure to initially configure images after a power up or `nCONFIG` event.
- Failure of an image after it was configured, due to an HPS watchdog timeout, when the watchdog was configured to trigger an RSU failure.
- Failure to load a specific image, applying to all possible failures: requested image failure, or failure of next images that are tried in case the requested image failed.

For examples of using the `max retry` option from both U-Boot and Linux, refer to the *Remote System Update Example* section.

#### Related Information

- [RSU Status and Error Codes](#) on page 114
- [Loading a Specific Image](#) on page 16
- [Remote System Update Example](#) on page 49

## 2.13. Querying the Decision Firmware Version

Each of the four decision firmware copies contains a field indicating the Intel Quartus Prime version that was used to create it. Both U-Boot and Linux offer the capability of querying the decision firmware version.

For more information about the firmware version, refer to the *Firmware Version Information* section.

For examples on how querying the decision firmware version works for both U-Boot and Linux, refer to the *Remote System Update Example* section.

#### Related Information

- [Firmware Version Information](#) on page 30
- [Remote System Update Example](#) on page 49



## 3. Quad SPI Flash Layout

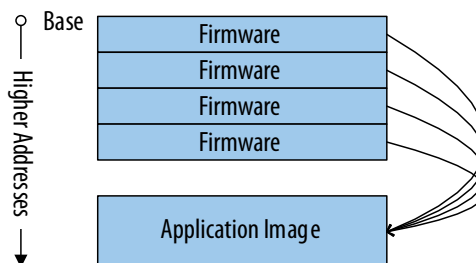
### 3.1. High Level Flash Layout

#### 3.1.1. Standard (non-RSU) Image Layout in Flash

In the standard (non-RSU) case, the flash contains four firmware images and the application image. To guard against possible corruption, there are four redundant copies of the firmware. The firmware contains a pointer to the location of the application image in flash.

**Figure 3. Image Layout in Flash: Non-RSU**

In this figure, each of the four firmware copies points to the Application Image.



#### 3.1.2. RSU Image Layout in Flash – SDM Perspective

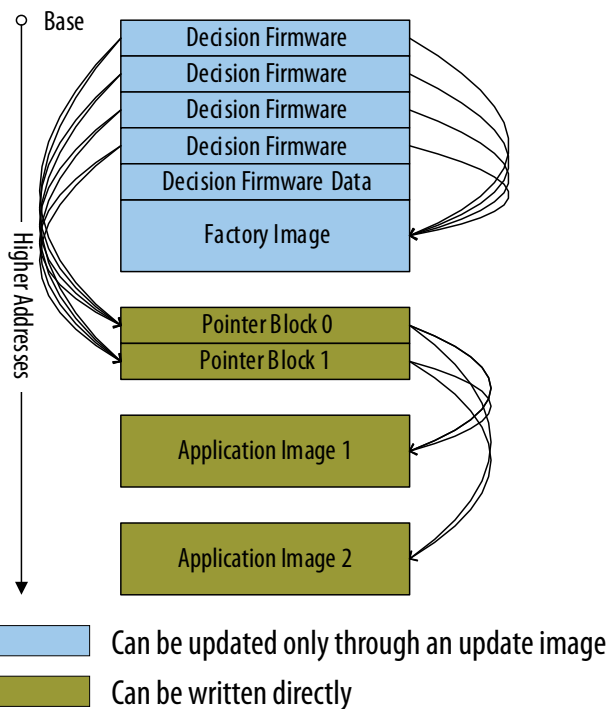
In the RSU case, decision firmware replaces the standard firmware. The decision firmware copies have pointers to the following structures in flash:

- Decision firmware data
- One factory image
- Two configuration pointer blocks (CPBs)

**Figure 4. RSU Image Layout in Flash : SDM Perspective**

In this figure:

- Each of the Decision Firmware copies point to the Factory Image and both Pointer Block 0 and Pointer Block 1.
- Both Pointer Block 0 and Pointer Block 1 point to Application Images.



The decision firmware data stores basic settings, including the following:

- The clock and pins that connect to quad SPI flash memory
- The **Direct to Factory Image** pin that forces the SDM to load the factory image

*Note:* You can set this pin on the following menu in the factory image project:

**Assignments > Device > Device and Pin Options > Configuration > Configuration Pin Options**

- The `max_retry` parameter value.

The pointer blocks contain a list of application images to try until one of them is successful. If none is successful, the SDM loads the factory image. To ensure reliability, the pointer block includes a main and a backup copy in case an update operation fails.

Both the factory image and the application images start with firmware. First, the decision firmware loads the firmware. Then, that firmware loads the rest of the image. These implementation details are not shown in the figure above. For more information, refer to the *Application Image Layout* section.

### Related Information

[Application Image Layout](#) on page 29

### 3.1.3. RSU Image Layout – Your Perspective

The sub-partition table (SPT) is used for managing the allocation of the quad SPI flash.

The Intel Quartus Prime Programming File Generator creates the SPT when creating the initial manufacturing image. To ensure reliable operation, the Programming File Generator creates two copies of the sub-partition table and the configuration pointer block, SPT0 and SPT1 and CPB0 and CPB1

The initial RSU image stored in flash typically contains the following partitions:

**Table 2. Typical Sub-Partitions of the RSU Image**

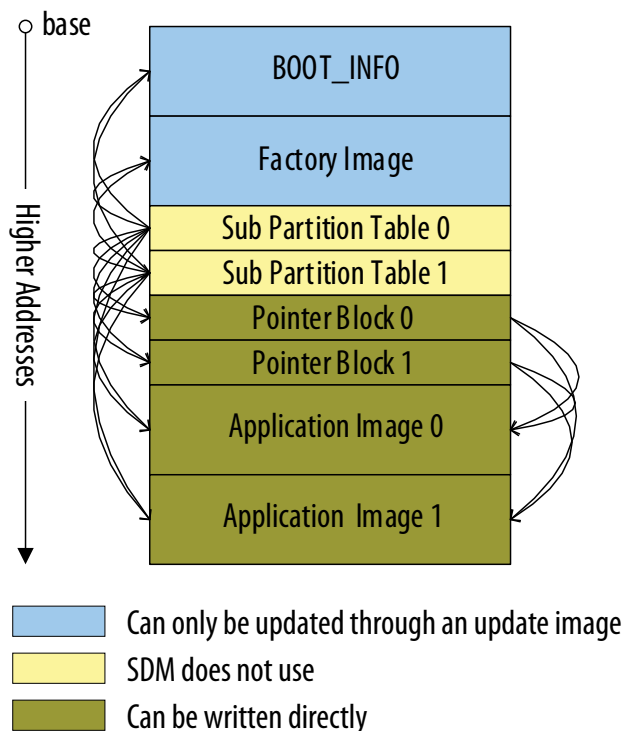
Sub-partition Name	Contents
BOOT_INFO	Decision firmware and decision firmware data
FACTORY_IMAGE	Factory Image
SPT0	Sub-partition table 0
SPT1	Sub-partition table 1
CPB0	Pointer block 0
CPB1	Pointer block 1
P1	Application image 1
P2	Application image 2



**Figure 5. RSU Image Layout - Your Perspective**

In this figure:

- SPT0 and SPT1 point to everything:
  - BOOT\_INFO
  - Factory Image
  - Pointer Block 0 and Pointer Block 1
  - All Application Images
- Pointer Block 0 and Pointer Block 1 point to all Application Images



To summarize, your view of flash memory is different from SDM view in two ways:

- You do not need to know the addresses of the decision firmware, decision firmware data, and factory image.
- You have access to the sub-partition tables. The sub-partition tables provide access to the data structures required for remote system update.

## 3.2. Detailed Quad SPI Flash Layout

### 3.2.1. RSU Image Sub-Partitions Layout

The *RSU Image Sub-Partitions Layout* table shows the layout of RSU image stored in QSPI flash.

If you anticipate changes to the factory or application images, you may consider reserving additional memory space. By default, the Intel Quartus Prime Programming File Generator reserves additional 256 kB of memory space for a factory image. To increase the partition size, update the **End Address** value in the **Edit Partition** dialog box window as described in the *Generating the Initial RSU Image*.

**Table 3. RSU Image Sub-Partitions Layout**

Flash Offset	Size (in bytes)	Contents	Sub-Partition Name
0 K	512 K	Decision firmware	BOOT_INFO
512 K	512 K	Decision firmware	
1 M	512 K	Decision firmware	
1.5 M	512 K	Decision firmware	
2 M	8 K + 24 K pad	Decision firmware data	
2 M+32 K	32 K	Reserved for SDM	
2 M+64 K	Varies	Factory image	FACTORY_IMAGE
Next	4 K + 28 K pad	Sub-partition table (copy 0)	SPT0
Next	4 K + 28 K pad	Sub-partition table (copy 1)	SPT1
Next	4 K + 28 K pad	Pointer block (copy 0)	CPB0
Next	4 K + 28 K pad	Pointer block (copy 1)	CPB1
Next	Varies	Application image 1	You assign
Next	Varies	Application image 2	You assign

The Intel Quartus Prime Programming File Generator allows you to create many user partitions. These partitions can contain application images and other items such as the Second Stage Boot Loader (SSBL), Linux kernel, or Linux root file system.

When you create the initial flash image, you can create up to seven partitions for application images. There are no limitations on creating empty partitions.

### Related Information

[Generating the Initial RSU Image](#)

## 3.2.2. Sub-Partition Table Layout

The following table shows the structure of the sub-partition table. The Intel Quartus Prime Programming File Generator software supports up to 126 partitions. Each sub-partition descriptor is 32 bytes.

**Note:** The firmware never updates the SPT.

**Table 4. Sub-partition Table Layout**

Offset	Size (in bytes)	Description
0x000	4	Magic number 0x57713427
0x004	4	Version number:
continued...		

Offset	Size (in bytes)	Description
		<ul style="list-style-type: none"> <li>0 - before Intel Quartus Prime Pro Edition software version 20.4</li> <li>1 - starting with Intel Quartus Prime Pro Edition software version 20.4</li> </ul>
0x008	4	Number of entries
0x00C	4	Checksum: <ul style="list-style-type: none"> <li>0 - before Intel Quartus Prime Pro Edition software version 20.4</li> <li>CRC32 checksum - starting with Intel Quartus Prime Pro Edition software version 20.4</li> </ul>
0x010	16	Reserved
0x020	32	Sub-partition Descriptor 1
0x040	32	Sub-partition Descriptor 2
0xFE0	32	Sub-partition Descriptor 126

Starting with Intel Quartus Prime Pro Edition software version 20.4, the SPT header contains a CRC32 checksum that is computed over the whole SPT. The value of the CRC32 checksum filed itself is assumed as zero when the checksum is computed. Refer to [Application Image Layout](#) on page 29 for the algorithm used to compute the CRC32 checksum. The checksum is provided as a convenience so that SPT corruptions can better be detected by HPS software. By default the feature is turned off.

Each 32-byte sub-partition descriptor contains the following information:

**Table 5. Sub-partition Descriptor Layout**

Offset	Size	Description
0x00	16	Sub-partition name, including a null string terminator
0x10	8	Sub-partition start offset
0x18	4	Sub-partition length
0x1C	4	Sub-partition flags

Two flags are currently defined:

- System set to 1: Reserved for RSU system. For partition offset value, refer to [Table 3](#) on page 26.
- Read only set to 1: The system protects partition against direct writes.

The Intel Quartus Programming File Generator sets these flags as follows at image creation time, then they are not changed afterward:

**Table 6. Flags Specifying Contents and Access**

Partition	System	Read Only
BOOT_INFO	1	1
FACTORY_IMAGE	1	1
SPT0	1	0
SPT1	1	0
CPB0	1	0
<i>continued...</i>		

Partition	System	Read Only
CPB1	1	0
P1	0	0
P2	0	0

**Note:** In order to successfully update SPTs, the HPS software (U-Boot or Linux) must be configured to have a QSPI erase granularity of 32 KB or less. When configured with a coarser erase granularity (like 64 KB for example), the operation fails. All supported flash devices offer erase granularities of 4 KB, 32 KB, and 64 KB, and the default for the current HPS software is 4 KB.

### 3.2.3. Configuration Pointer Block Layout

The configuration pointer block contains a list of application image addresses. The SDM tries the images in sequence until one of them is successful or all fail. The structure contains the following information:

**Table 7. Pointer Block Layout**

Offset	Size (in bytes)	Description
0x00	4	Magic number 0x57789609
0x04	4	Size of pointer block header (0x18 for this document)
0x08	4	Size of pointer block (4096 for this document)
0x0C	4	Reserved
0x10	4	Offset to image pointers (IPTAB)
0x14	4	Number of image pointer slots (NSLOTS)
0x18	8	Reserved
0x20 (IPTAB) <sup>(8)</sup>	8	First (lowest priority) image pointer slot (IPTAB)
	8	Second (2nd lowest priority) image pointer slot
	8	...
	8	Last (highest priority) image pointer

The configuration pointer block can contain up to 508 application image pointers. The actual number is listed as `NSLOTS`. A typical configuration pointer block update procedure consists of adding a new pointer and potentially clearing an older pointer. Typically, the pointer block update uses one additional entry. Consequently, you can make 508 updates before the pointer block must be erased. The erase procedure is called *pointer block compression*. This procedure is power failure safe, as there are two copies of the pointer block. The copies are in different flash erase sectors. While one copy is being updated the other copy is still valid.

<sup>(8)</sup> The offset may vary in future firmware updates.

**Note:** In order to successfully update CPBs, the HPS software (U-Boot or Linux) must be configured to have a QSPI erase granularity of 32 KB or less. When configured with a coarser erase granularity (like 64 KB for example), the operation fails. All supported flash devices offer erase granularities of 4 KB, 32 KB, and 64 KB, and the default for the current HPS software is 4 KB.

### 3.2.4. Application Image Layout

The application image comprises SDM firmware and the configuration data. The configuration data includes up to four sections. The SDM firmware contains pointers to those sections. The table below shows the location of the number of sections and the section pointers in a application image.

**Table 8. Application Image Section Pointers**

Offset	Size (in bytes)	Description
0x1F00	4	Number of sections
	...	
0x1F08	8	Address of 1st section
0x1F10	8	Address of 2nd section
0x1F18	8	Address of 3rd section
0x1F20	8	Address of 4th section
	...	
0x1FFC	4	CRC32 of 0x1000 to 0x1FFB

The section pointers must match the actual location of the FPGA image in flash. Two options are available to meet this requirement:

- You can generate the application image to match the actual location in quad SPI flash memory. This option may not be practical as different systems may have a different set of updates applied, which may result in different slots being suitable to store the new application image.
- You can generate the application image as if it is located at address zero, then update the pointers to match the actual location.
  - In Intel Quartus Prime software version 21.1 or later, a **Use relative address** option is the default option to generate a single application bitstream. You do not have to specify the start address since you can program the image to any available location in the QSPI flash memory. To load the image, you must correctly point to the starting address of the stored image in the flash memory.

If you choose to disable this option, unselect the **Use relative address** checkbox and provide the **Start address** for the image in flash memory. For more information, refer to *Generating an Application Image*.

When using the HPS to manage RSU, both U-Boot and LIBRSU clients implement the below procedure to relocate application images targeting address zero in the actual destination slot address.

The procedure to update the pointers from an application image created for INITIAL\_ADDRESS to NEW\_ADDRESS is:

1. Create the application image, targeting the `INITIAL_ADDRESS`.
2. Read the 32-bit value from offset 0x1F00 of the application image to determine the number of sections.
3. For `<s>= 1` to `number_of_sections`:
  - a. `section_pointer` = read the 64-bit section pointer from `0x1F00 + (s * 8)`
  - b. Subtract `INITIAL_ADDRESS` from `section_pointer`
  - c. Add `NEW_ADDRESS` to `section_pointer`
  - d. Store updated `section_pointer`
4. Recompute the CRC32 for addresses 0x1000 to 0x1FFB. Store the new value at offset 0x1FFC. The CRC32 value must be computed on a copy of the data using the following procedure:
  - a. Swap the bits of each byte so that the bits occur in reverse order and compute the CRC.
  - b. Swap the bytes of the computed CRC32 value to appear in reverse order.
  - c. Swap the bits in each byte of the CRC32 value.
  - d. Write the CRC32 value to flash.

**Note:** The factory update image has a different format, which requires a different procedure for the pointer updates. When using the HPS to manage RSU, both U-Boot and LIBRSU clients implement this procedure for relocating the factory update image.

**Note:** The combined application image has a different format, with no pointers that need to be relocated. The image can be placed unmodified in flash at any address.

#### Related Information

[Generating an Application Image](#)

### 3.3. Decision Firmware Data Max Retry Information

The only user accessible field from the decision firmware data structure is the `max_retry` value, stored as an 8-bit value at address 0x10018C in flash. The value in flash is actually `max_retry - 1`. That is, a value of zero means each image is tried just once.

Both U-Boot and LibRSU enable the `max_retry` option to be queried. For examples on querying the `max_retry` option for both U-Boot and Linux, refer to the *Remote System Update Example* section. The `max_retry` option must first be queried in U-Boot for the value to be available in Linux.

### 3.4. Firmware Version Information

The Intel Quartus Prime firmware version is stored as a 32bit value at offset 0x420 in the following:

- Decision firmware (each of the four copies),
- Application images,
- Factory update images,
- Decision firmware update images,
- Combined application images.

The format for the version field is described below:

**Table 9. Firmware Version Fields**

Bit Field	Bits	Description
version_major	31:24	Major release number for Intel Quartus Prime. Example: set to 20 for release 20.1.
version_minor	23:16	Minor release number for Intel Quartus Prime. Example: set to 1 for release 20.1.
version_update	15:8	Update release number for Intel Quartus Prime. Example: set to 2 for release 20.1.2
reserved	7:0	Set to zero

**Note:** Intel Quartus Prime patches do not change the firmware version field. All patches have the same firmware version field as the base release on top of which the patches are applied to.

Both U-Boot and LibRSU enable the decision firmware versions to be queried. For examples on querying the decision firmware version for both U-Boot and Linux, refer to the *Remote System Update Example* section. The decision firmware version must first be queried in U-Boot for the value to be available in Linux.

#### Related Information

[Remote System Update Example](#) on page 49



## 4. Intel Quartus Prime Software and Tool Support

---

This section lists the Intel Quartus Prime tools you can use for the RSU scenarios. For more information about each tool, refer to the corresponding documentation.

### 4.1. Intel Quartus Prime Pro Edition Software

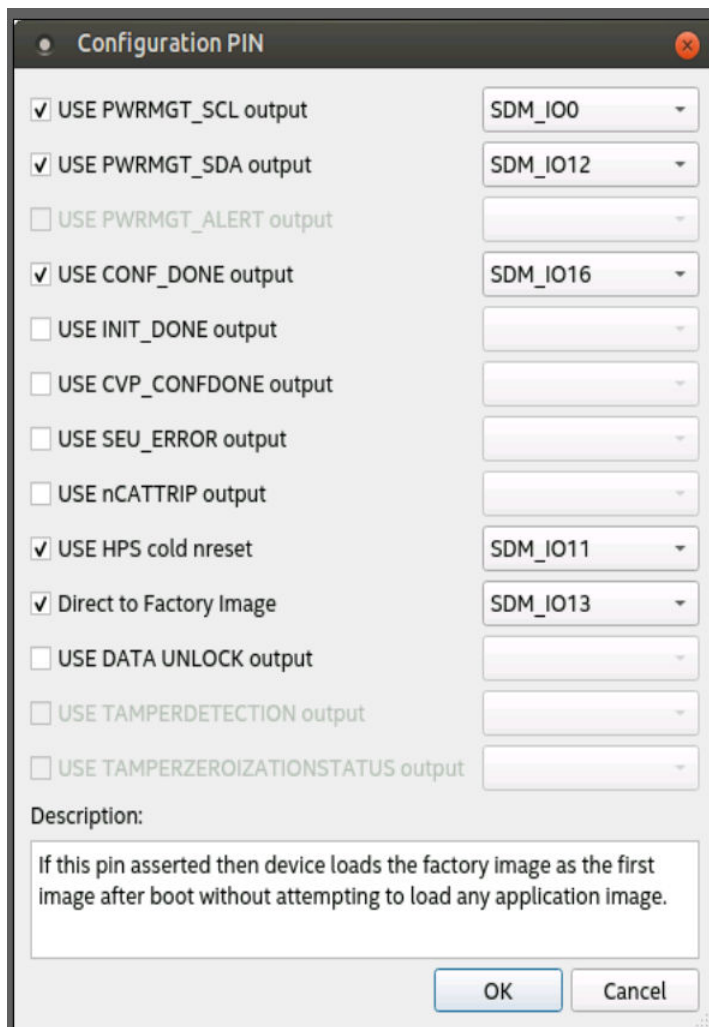
You must use the Intel Quartus Prime Pro Edition software to compile the hardware projects you use for remote system update.

#### 4.1.1. Selecting Factory Load Pin

Intel Quartus Prime offers the option to select the pin to use to force the factory application to load on a reset.

1. Navigate to **Assignments > Device > Device and Pin Options > Configuration > Configuration Pin Options**
2. Check the **Direct to Factory Image** check box.
3. Select the desired pin from the drop box.



**Figure 6. Configuration PIN GUI**

The Configuration PIN GUI is a dialog box with a title bar containing a close button. It contains a list of configuration options, each with a checkbox and a corresponding dropdown menu. The options are:

Option	Selected Value
<input checked="" type="checkbox"/> USE PWRMGT_SCL output	SDM_IO0
<input checked="" type="checkbox"/> USE PWRMGT_SDA output	SDM_IO12
<input type="checkbox"/> USE PWRMGT_ALERT output	
<input checked="" type="checkbox"/> USE CONF_DONE output	SDM_IO16
<input type="checkbox"/> USE INIT_DONE output	
<input type="checkbox"/> USE CVP_CONFDONE output	
<input type="checkbox"/> USE SEU_ERROR output	
<input type="checkbox"/> USE nCATTRIP output	
<input checked="" type="checkbox"/> USE HPS cold nreset	SDM_IO11
<input checked="" type="checkbox"/> Direct to Factory Image	SDM_IO13
<input type="checkbox"/> USE DATA_UNLOCK output	
<input type="checkbox"/> USE TAMPERDETECTION output	
<input type="checkbox"/> USE TAMPERZEROIZATIONSTATUS output	

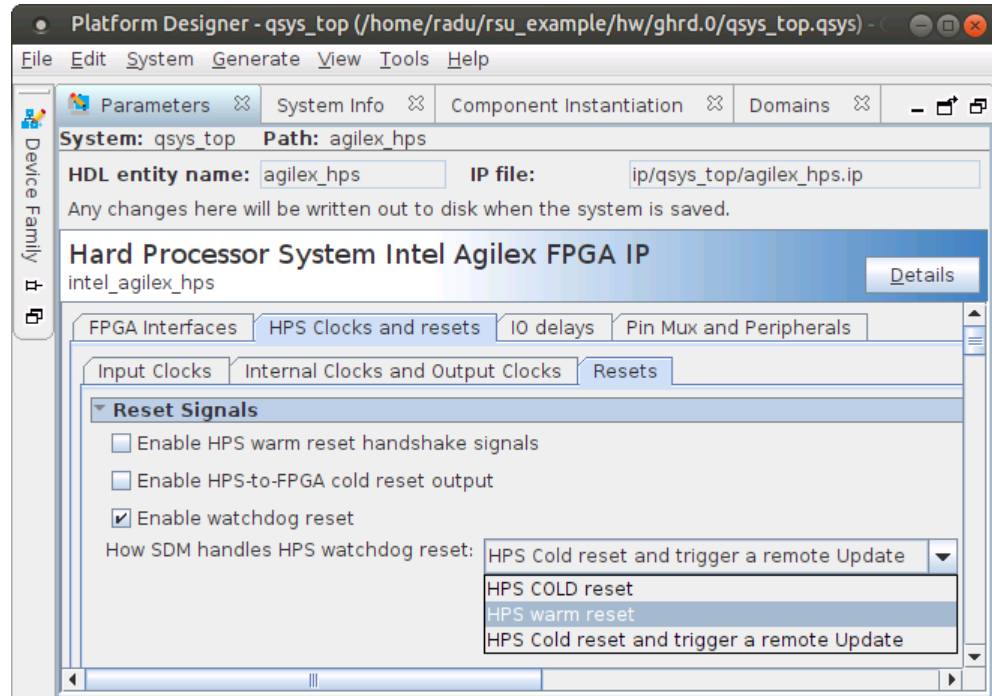
Description:

If this pin asserted then device loads the factory image as the first image after boot without attempting to load any application image.

OK Cancel

### 4.1.2. Enabling HPS Watchdog to Trigger RSU

The tool offers the option of selecting what happens when you enable an HPS watchdog, but do not service it, and produces an HPS reset. The option is available as an HPS component property in Platform Designer, as shown below:



When the highlighted option is selected and an HPS watchdog is enabled and times out because it is not serviced, the SDM considers the current application image as a failure. The SDM then tries to load the next application image in the configuration pointer block, or the factory image if all application images fail. When the factory image also fails to configure, the SDM clears the FPGA and HPS and the device remains not configured.

Enabling the HPS watchdog flow is:

1. You select the desired watchdog behavior in the Intel Quartus Prime project used to create the factory image SOF.
2. Then the Intel Quartus Prime Programming File Generator takes the setting from the factory SOF file, and stores it in the decision firmware data section.
3. The decision firmware then uses the data from that section to implement the selected behavior.

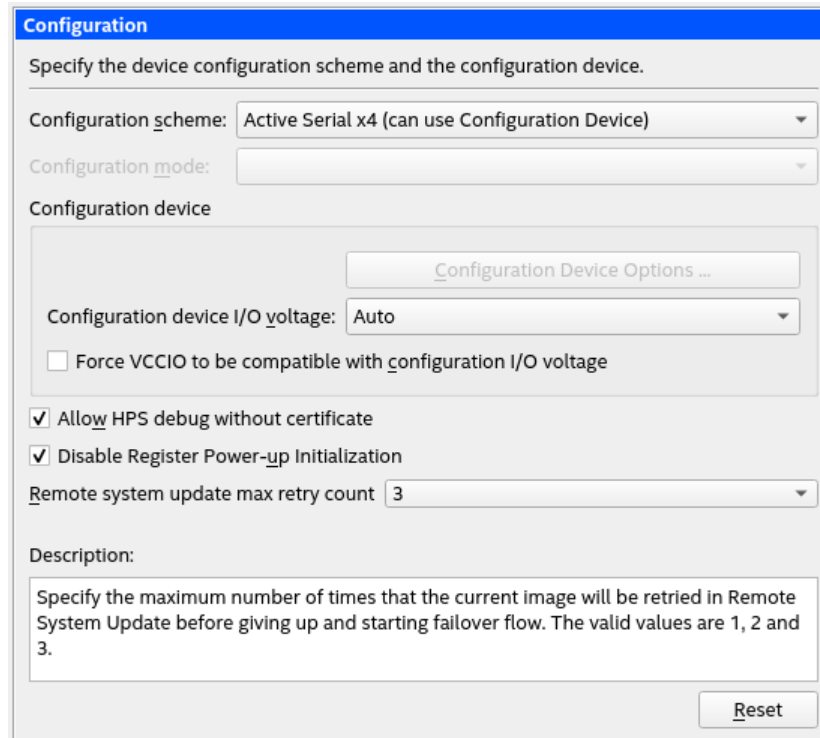
*Note:* The behavior can be changed through RSU by updating the factory image using a SOF with a different setting.

### 4.1.3. Setting the Max Retry Parameter

The `max_retry` parameter specifies how many times the application and factory images are tried when configuration failures occur. The default value is one, which means each image is tried only once. The maximum possible value is three, which means each image can be tried up to three times.

The `max_retry` parameter is stored in the decision firmware data area as defined in the *Glossary* section, and is programmed when the decision firmware and decision firmware data are programmed as part of the Initial Flash Layout or the *Updating the Factory Image* sections. The decision firmware data can also be updated by a decision firmware update image, or by a combined application image.

The `max_retry` parameter is specified for the hardware project used to create the factory image, from the Intel Quartus Prime GUI by navigating to **Assignments > Device > Device and Pin Options > Configuration** and selecting the value for the "Remote system update max retry count" field.



The parameter can also be specified by directly editing the project Intel Quartus Prime settings file (.qsf) and adding the following line or changing the value if it is already there:

```
set_global_assignment -name RSU_MAX_RETRY_COUNT 3
```

## 4.2. Programming File Generator

Intel Quartus Prime Programming File Generator, is part of Intel Quartus Prime Pro Edition software. The tool creates programming files for all RSU scenarios: initial flash images, application images, factory update images, decision firmware update images and combined application images.

For usage examples, refer to the *Creating the Initial Flash Image*, *Creating the Initial Flash Image*, *Creating the Factory Update Image*, *Creating the Firmware Update Image*, and *Creating the Combined Application Image* sections.

For more information about the tool, refer to *Intel Quartus Prime Pro Edition User Guide: Programmer*.

#### Related Information

- [Creating the Initial Flash Image](#) on page 54
- [Creating the Application Image](#) on page 60
- [Creating the Factory Update Image](#) on page 61
- [Intel Quartus Prime Pro Edition User Guide: Programmer](#)
- [Creating the Decision Firmware Update Image](#) on page 61

### 4.2.1. Programming File Generator File Types

The most important file types created by the Programming File Generator for RSU are listed in the following table:

File Extension	File Type	Description
.jic	JTAG Indirect Configuration File	These files are intended to be written to the flash by using the Intel Quartus Prime Programmer tool. They contain the actual flash data, and also a flash loader, which is a small FPGA design used by the Intel Quartus Prime Programmer to write the data.
.rpd	Raw Programming Data File	These files contain actual binary content for the flash and no additional metadata. They can contain the full content of the flash, similar with the .jic file—this is typically used in the case where an external tool is used to program the initial flash image. They can also contain an application image, or a factory update image.
.map	Memory Map File	These files contain details about where the input data was placed in the output file. This file is human readable.
.rbf	Raw Binary File	These files are binary files which can be used typically to configure the FPGA fabric for HPS first use cases. They can also be used for passively configuring the FPGA device through Avalon® streaming interface, but that is not supported with RSU.

### 4.2.2. Bitswap Option

The Intel Quartus Prime Programmer assumes by default that the binary files have the bits in the reversed order for each byte. Because of this, the "bitswap=on" option needs to be enabled as follows:

- For each input binary file (.bin and .hex files are supported).
- For each output RPD file (full flash images, application images, factory update images, decision firmware update images, and combined application images.).

The bitswap option is used accordingly in the examples presented in this document.

## 4.3. Intel Quartus Prime Programmer

You can use the Intel Quartus Prime Programmer to program the initial flash image. For an example, refer to the *Flashing the Initial Image to QSPI* section.



### **Related Information**

[Flashing the Initial RSU Image to QSPI](#) on page 65



## 5. Software Support

---

This section presents the HPS software support for RSU.

Refer to [Component Versions](#) on page 98 for the Intel Quartus Prime Pro Edition software version and the HPS software component versions covered by this document.

### 5.1. SDM RSU Support

Besides implementing the actual RSU configuration flow, the SDM offers commands to interact with RSU:

- Get the flash address of the currently running image.
- Get details on the errors that occurred when trying to load an image which failed.
- Clear the sticky error information to be able to record new error information.
- Get the locations of SPTs.
- Load a specific image.
- Report the state of HPS software.
- Query the value of the current retry counter.
- Reset the value of the current retry counter.

The SDM commands need to be called from EL3, the highest execution level on Cortex-A53.

The SDM commands are not publicly documented. Full support is offered for accessing the relevant services from both U-Boot and LIBRSU.

### 5.2. Arm Trusted Firmware Support

The Arm Trusted Firmware (ATF) runs at the EL3, the highest execution level on Cortex-A53. ATF is loaded as part of the boot process, and it remains resident.

ATF exports a SMC (Secure Monitor Call) handler which allows software running at lower execution levels to access services offered by the SDM such as FPGA configuration and RSU commands. Both U-Boot and Linux use the SMC handler.

### 5.3. U-Boot RSU Support

U-Boot provides the following RSU-related features:

- Enables you to access the RSU functionality from U-Boot source code, using an interface similar with LIBRSU.
- Enables you to access the RSU functionality from U-Boot command line

U-Boot runs at EL1, and it uses the SMC (Secure Monitor Call) handler provided by ATF (which runs at EL3) to access the SDM commands. U-Boot also accesses the QSPI flash to implement the RSU functionality.

**Note:** Prior to branch `socfpga_v2020.07`, U-Boot was running at EL3 and was the provider of the SMC handler for the rest of the HPS software. In branch `socfpga_v2020.07` there was the option to have either U-Boot or ATF to implement the SMC handler. Starting with branch `socfpga_v2020.10` only the ATF provides the SMC handler. Using the U-Boot as the SMC handler is now only supported for experimental purposes such as bringing up a new board, as it allows convenient access to all HPS registers from U-Boot command line, at the expense of reduced security and functionality. All new features are implemented in ATF only.

#### Related Information

[Remote System Update Example](#) on page 49

### 5.3.1. U-Boot RSU APIs

U-Boot offers a full set of APIs to manage RSU, similar with the LIBRSU APIs.

Refer to the source code file `arch/arm/mach-socfpga/include/mach/rsu.h` for details on the APIs. Refer to source code `arch/arm/mach-socfpga/rsu_sl0.c` for the U-Boot `rsu` command implementation, which are implemented using the APIs.

Refer to [U-Boot RSU Reference Information](#) on page 116 for more details about U-Boot RSU APIs.

#### Related Information

[LIBRSU Reference Information](#) on page 134

### 5.3.2. U-Boot RSU Commands

U-Boot offers the `rsu` command, with a full set of options for managing the RSU, similar with the functionality offered by the RSU client.

The following commands do not have an RSU client equivalent:

- `list`
- `update`
- `dtb`

The `rsu list` command:

- Queries the SDM about the location of the SPT in flash, reads and displays it.
- Reads the CMF pointer block from flash and displays the relevant information.
- Queries SDM about the currently running image, RSU state and the encountered errors and displays the information.

The following is an example of the `rsu list` command being used:

```
SOCFPGA # rsu list
RSU: Remote System Update Status
Current Image   : 0x01000000
Last Fail Image : 0x00000000
```

```

State          : 0x00000000
Version        : 0x00000202
Error location  : 0x00000000
Error details   : 0x00000000
Retry counter   : 0x00000000
RSU: Sub-partition table 0 offset 0x00910000
RSU: Sub-partition table 1 offset 0x00918000
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total 256 MiB
RSU: Sub-partition table content
      BOOT_INFO      Offset: 0x0000000000000000      Length: 0x00210000      Flag :
0x000000003
      FACTORY_IMAGE   Offset: 0x0000000000210000      Length: 0x00700000      Flag :
0x000000003
      P1              Offset: 0x0000000001000000      Length: 0x01000000      Flag :
0x000000000
      SPT0            Offset: 0x0000000000910000      Length: 0x00008000      Flag :
0x000000001
      SPT1            Offset: 0x0000000000918000      Length: 0x00008000      Flag :
0x000000001
      CPB0            Offset: 0x0000000000920000      Length: 0x00008000      Flag :
0x000000001
      CPB1            Offset: 0x0000000000928000      Length: 0x00008000      Flag :
0x000000001
      P2              Offset: 0x0000000002000000      Length: 0x01000000      Flag :
0x000000000
      P3              Offset: 0x0000000003000000      Length: 0x01000000      Flag :
0x000000000
RSU: CMF pointer block offset 0x00920000
RSU: CMF pointer block's image pointer list
Priority 1 Offset: 0x0000000001000000 nslot: 0

```

The `rsu update` command is used to tell the SDM to load an image from a specific address. The following is an example of the `rsu update` command:

```

SOCFPGA # rsu update 0x03000000
RSU: RSU update to 0x0000000003000000

```

The `rsu dtb` command is used to let U-Boot update the QSPI partition called "qspi\_boot" in the Linux DTB so that it starts immediately after the `BOOT_INFO` partition. This way the decision firmware, decision firmware data, and the factory image are not accessible from Linux, as this reduces the risk of accidental corruption for them. The size of the partition is also reduced accordingly.

For more information, refer to the *Protected Access to Flash* section.

The `rsu dtb` operates on the DTB loaded in memory by U-Boot, before passing it to Linux. The sequence to use is:

1. Load DTB.
2. Run `rsu dtb` command.
3. Boot Linux

Refer to the *Exercising U-Boot RSU Commands* section for examples of how to use the U-Boot `rsu` command.

Refer to [U-Boot RSU Reference Information](#) on page 116 for more details about U-Boot commands.

### Related Information

- [Protected Access to Flash](#) on page 17
- [Exercising U-Boot RSU Commands](#) on page 65

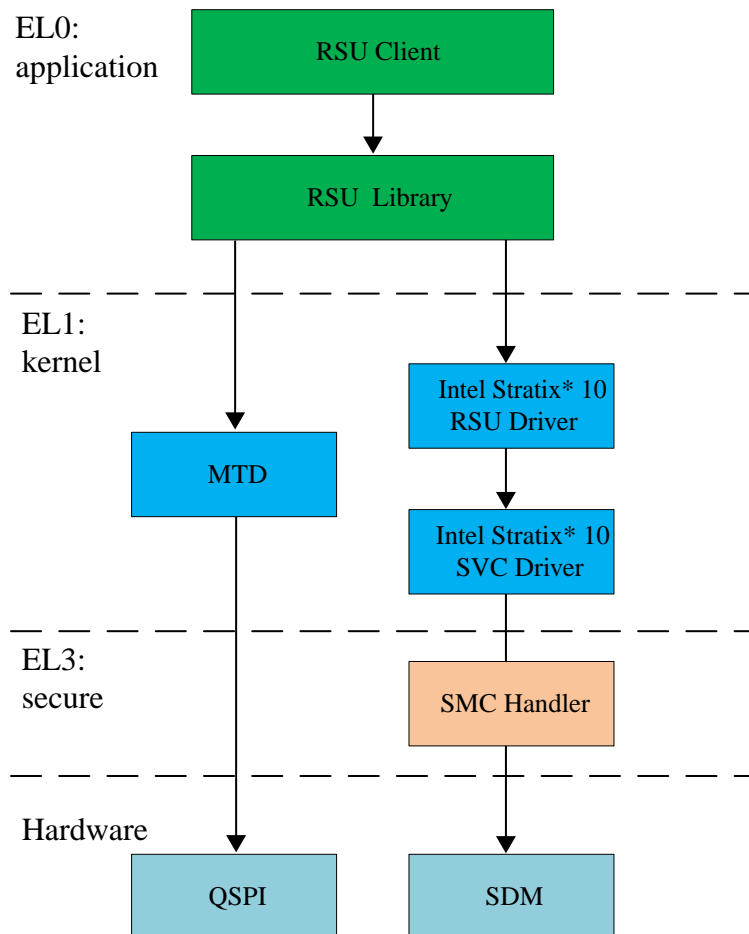


## 5.4. Linux RSU Support

The following RSU facilities are offered on Linux:

- LIBRSU user-space library which allows you to perform a complete set of RSU operations.
- RSU client example application which exercises the LIBRSU APIs.

**Figure 7. Linux RSU Overview**



\* Intel Stratix® 10 drivers are compatible with Intel® Agilex® SoC devices.

### 5.4.1. Intel Service Driver

The SMC handler is left resident by ATF and is running at EL3, the highest execution level, therefore it can access the SDM commands. The Intel Service Driver allows the kernel to execute SMCs in order to access the services offered by the SMC handler.

The default kernel configuration defines `CONFIG_INTEL_STRATIX10_SERVICE=y`, which means that this driver is part of the kernel. Besides RSU, the Intel Service Driver also provides other services, such as the FPGA configuration services offered by the ATF SMC handler.

The driver does not need to be used directly, so its interface is not documented here. For details, refer to the source code in the `drivers/firmware/stratix10-svc.c`.

**Note:** Linux is a rapidly changing community project. The information from this section is valid for Linux kernel version 5.4.114-lts, and may change again in the future.

### 5.4.2. Intel RSU Driver

The Intel RSU driver exports the RSU services using the **sysfs** interface. The source code is located in the `drivers/firmware/stratix10-rsu.c` file.

The default kernel configuration defines `CONFIG_INTEL_STRATIX10_RSU=m` which means it is configured as a loadable module which needs to be loaded with `insmod`. For information about how to use this driver, refer to the *Exercising the RSU Client* section, located in the *LIBRSU Reference Information* appendix.

The driver offers its services through the following `sysfs` files located in the `/sys/devices/platform/stratix10-rsu.0` folder. This path is a configurable parameter for LIBRSU. For more information, refer to the *Configuration File* section, located in the *LIBRSU Reference Information* appendix.

**Table 10. SysFS Entries for Intel RSU Driver**

File	R/W	Description
<code>current_image</code>	RO	Location of currently running image in SDM QSPI flash.
<code>fail_image</code>	RO	Location of failed image in SDM QSPI flash.
<code>error_details</code>	RO	Opaque error code, with no meaning to users.
<code>error_location</code>	RO	Location of error within the image that failed.
<code>state</code>	RO	State of the RSU system.
<code>version</code>	RO	Version of the RSU system.
<code>notify</code>	WO	Used to notify SDM of the HPS software execution stage, also for clearing the error status, and resetting the retry counter.
<code>retry_counter</code>	RO	Current value of the <code>retry_counter</code> , indicating how many times the current image retries to be loaded. A value of zero means this is the first time.
<code>reboot_image</code>	WO	Address of image to be loaded on next <code>reboot</code> command.
<code>dcmf0</code>	RO	Decision firmware copy 0 version information
<code>dcmf1</code>	RO	Decision firmware copy 1 version information
<code>dcmf2</code>	RO	Decision firmware copy 2 version information
<code>dcmf3</code>	RO	Decision firmware copy 3 version information
<code>dcmf0_status</code>	RO	Decision firmware copy 0 status. A value of zero means the copy is fine, anything else means the copy is corrupted.
<code>dcmf1_status</code>	RO	Decision firmware copy 1 status. A value of zero means the copy is fine, anything else means the copy is corrupted.
continued...		

File	R/W	Description
dcmf2_status	RO	Decision firmware copy 2 status. A value of zero means the copy is fine, anything else means the copy is corrupted.
dcmf3_status	RO	Decision firmware copy 3 status. A value of zero means the copy is fine, anything else means the copy is corrupted.
max_retry	RO	Value of max_retry option, read from decision firmware data section in flash.

For more information, refer to the *RSU Status and Error Codes* and *Decision Firmware Version Information* sections.

**Note:** This information is provided as reference only, as using the RSU driver directly is not recommended or supported. Instead, use the LIBRSU or RSU client directly to perform the RSU tasks.

**Note:** Linux is a rapidly changing community project. The information from this section is valid for Linux kernel version 5.4.114-lts, and may change again in the future.

#### Related Information

- [RSU Status and Error Codes](#) on page 114
- [Exercising the RSU Client](#) on page 81
- [Configuration File](#) on page 134
- [Firmware Version Information](#) on page 30

### 5.4.3. LIBRSU Library

The RSU library offers a complete set of RSU APIs which are callable from your applications. The library is built on top of the Intel RSU driver and it uses the Linux MTD framework to access the QSPI flash.

The LIBRSU Library uses the term “slot” to refer to a sub-partition which is intended to contain an application image. It also uses the term “priority” to refer to the fact that the images are loaded by SDM in the order defined by the configuration pointer block.

For information about LIBRSU configuration file, data types, and APIs, refer to the *LIBRSU Reference Information* section.

#### Related Information

[LIBRSU Reference Information](#) on page 134

### 5.4.4. RSU Client

The RSU Client is an example Linux application which is built on top of the LIBRSU and exercises the APIs offered by the library. For more information about the RSU client command option and also usage examples, refer to the *Exercising the RSU Client* section, located in the *LIBRSU Reference Information* appendix.

The RSU Client can be used as-is, but Intel recommends that you write your own application to manage RSU according to your specific requirements.

#### Related Information

[Exercising the RSU Client](#) on page 81



## 6. Flash Corruption - Detection and Recovery

---

The Remote System Update enables robust system operation, with resilience to configuration failures, runtime failures and flash corruption. This section presents an overview of how each component in the system is protected against flash corruptions, and how it can be recovered in case such corruptions occur. This section also includes an example maintenance and recovery procedure.

For examples on how to use these features from both U-Boot and Linux, refer to the *Remote System Update Example* section.

### 6.1. Decision Firmware

The decision firmware is extremely important, as it is executed first when the device powers up, to select and load the highest priority application image. It is also executed each time another application image or the factory image needs to be loaded.

There are four copies of the decision firmware, stored at address zero in QSPI flash. The decision firmware is protected by SHA checksums. If the SDM BootROM detects that the first copy is corrupted, it tries the second copy, then the third and fourth copies.

The index of the currently used decision firmware copy is reported to the HPS through the `version` field of the `RSU_STATUS`. U-Boot uses this index to compare the currently used decision firmware copy with the other copies in flash, and reports the ones which do not match as corrupted. The information is also reported to Linux, to be used by LibRSU.

Once you see that one or more copies of the decision firmware are corrupted, you can run either a factory update image or a decision firmware update image to recover. Both options update the decision firmware and the decision firmware data, while the factory update image also updates the factory image.

### 6.2. Decision Firmware Data

The decision firmware data contains parameters taken from the SOF file used for the factory image, which includes the QSPI clock configuration, the force factory load pin, the value of `max_retry` parameter and the selected behavior for the HPS watchdog.

There is one copy of the decision firmware data, and it is protected by a checksum. If the decision firmware detects that the decision firmware data has been corrupted, it loads the factory image, and reports the error through `RSU_STATUS` with a special error code.

Once you see that the decision firmware data is corrupted, you can recover it by running either a factory update image or a decision firmware update image. Both options update the decision firmware and the decision firmware data, while the factory update image also updates the factory image.

### 6.3. Configuration Pointer Block

The configuration pointer block contains the list of application images to be tried, in priority order.

The decision firmware uses the configuration pointer block to determine which image to load on power up, nCONFIG or if the previous image has failed.

The decision firmware never writes to the CPB, except to remove a factory update image, or a decision firmware update image after they have successfully completed, by clearing the value. The decision firmware never erases a CPB.

The HPS software (both U-Boot and LibRSU) use the CPB to add or remove application images, and to change their priorities. Typically this is done by writing new entries in unused locations (all bits set to '1') after erasing flash or cancelling images by writing zero to the corresponding locations. Occasionally HPS software needs to erase CPBs as they become full of cancelled entries, a process called "CPB compression".

There are two copies of the configuration pointer block. Whenever one CPB is erased in order to write new content to it, the other CPB is left untouched so that if a power failure happens before the first CPB is fully written to flash, the second one remains valid. There is a `magic` field in the CPB header, which is written last, after a CPB is erased and re-populated. An incorrect `magic` field indicates that a power failure has occurred and that CPB copy is invalid.

If the decision firmware detects that CPB0 is invalid, it tries to use CPB1 instead. If both CPBs are invalid, the decision firmware loads the factory image. The decision firmware reports to the user whether CPB0 or both CPB0 and CPB1 are corrupted.

When only one CPB is corrupted, the HPS software automatically detects the corruption and recovers the corrupt CPB from the other good copy. This happens at initialization time, both in U-Boot and LibRSU.

In addition to the `magic` field, the HPS software checks CPB integrity by validating that all entries correspond to valid SPT partitions. It also compares the two CPBs and if not identical it considers both corrupted.

When both CPBs are corrupted, you are notified by both U-Boot and LibRSU, and you have reduced functionality available. You can recover the CPBs by either restoring a saved copy, or creating an empty CPB, then adding images to it, from both U-Boot and Linux.

### 6.4. Sub-Partition Table

The Sub-partition table lists the flash partitions. These partitions include the `BOOT_INFO`, `FACTORY_IMAGE`, `SPTs`, `CPBs` and application image partitions. They also include any custom partitions defined at initial image creation time, or created at run time using U-Boot or LibRSU.

There are two SPT copies. Whenever one SPT is erased in order to write new content to it, the other SPT is left untouched so that if a power failure happens before the first SPT is fully written to flash, the second one remains valid. There is a `magic` field in the SPT header, which is written last, after a SPT is erased and re-populated correctly. An incorrect `magic` field indicates that a power failure has occurred and that SPT copy is invalid.

The decision firmware does not use the SPTs, never reads, erases or writes to them. It only reports their location to HPS software. The decision firmware is not impacted by SPT corruptions.

When only one SPT is corrupted, the HPS software automatically detects the corruption and recovers the corrupt SPT from the other good copy. This happens at initialization time, both in U-Boot and LibRSU.

In addition to the `magic` field, the HPS software checks SPT integrity by validating that partitions do not overlap. It also compares the two SPTs and if not identical it considers both corrupted.

Starting with Intel Quartus Prime Pro Edition software version 20.4, the SPTs are protected by a checksum in their header. The checksum is filled in at image creation time by the Programming File Generator. The HPS software can be enabled by a run-time configuration option to check and maintain the SPT checksums. The option is turned off by default, as it requires all the software components in the system to know about the checksum and maintain it appropriately, which may not be the case in all situations.

When both SPTs are corrupted, you are notified by both U-Boot and LibRSU, and you have access to reduced functionality. You can recover the SPTs by restoring a saved copy, from both U-Boot and Linux.

**Note:** If both SPTs are corrupted, the HPS software cannot locate the CPBs, and they are also considered corrupted. The available functionality is reduced even further in this case, until SPTs are restored.

## 6.5. Application Image

Application images are protected against corruption by internal SHA and CRC32 checksums. A system typically contains several application images, so in case one fails, another application image is also tried.

Both U-Boot and LibRSU offer services for maintaining and loading application images, as this is the main functionality offered by the Remote System Update.

## 6.6. Factory Image

Similar with application images, the factory image is protected against corruption by internal SHA and CRC32 checksums. A system contains only one factory image, which is tried when all the application images fail, or the decision firmware data is corrupted, or when both CPBs are corrupted or empty.

It is important to have a very reliable factory image, as it is the last one that is tried. Typically it is a small image, which does not need to perform the actual system function, just provide enough functionality for the system to be restored back to normal. This could include the following actions performed by HPS software:

- Running a decision firmware update image, if decision firmware or decision firmware data are corrupted,
- Restoring CPBs if both corrupted,
- Restoring SPTs if both corrupted,
- Adding or replacing application images if they are failing

The factory image can be updated by running a factory update image.

## 6.7. Example Maintenance Procedure

This section presents an example maintenance procedure, which can be followed each time a new image is loaded. This procedure can be implemented in either U-Boot or Linux.

### 6.7.1. Maintenance Procedure

1. Initialize RSU by calling `librsu_init()` in Linux or `rsu_init()` in U-Boot. This allows you to:
  - Check SPT integrity, recover a bad SPT if the other one is valid.
  - Check CPB integrity, recover a bad CPB if the other one is valid.
2. Query the RSU status by calling `rsu_status_log()`.
3. If the State field from RSU status is `STATE_DCIO_CORRUPTED (0xF004D00F)`, it means the decision firmware data got corrupted, go to the *Recover Decision Firmware Procedure*.
4. Query the decision firmware status by calling `rsu_dcmf_status()`.
5. If any DCMF shows as corrupted, go to the *to Recover Decision Firmware Procedure*.
6. Call `rsu_slot_count()`.
7. If the above returns `-ECORRUPTED_SPT`, it means both SPTs are corrupted, and you must restore SPT from a backup copy.
8. Call `rsu_slot_get_info()`.
9. If the above returns `-ECORRUPTED_CPB`, it means both CPBs are corrupted, and you must restore CPB from a backup copy, or create a new empty CPB.

Additional optional steps:

10. Save RSU status reported by firmware.
11. Clear RSU error reported by firmware by calling `rsu_clear_error_status()`.
12. Look at the State field from RSU status for more information on what happened:

- STATE\_CPB0\_CORRUPTED (0xF004D010): Corrupt CPB0 - already recovered at this point
  - STATE\_CPB0\_CPB1\_CORRUPTED (0xF004D011): Corrupt CPB0 & CPB1 - already recovered at this point
  - 0xF006YYYY: Watchdog timeout, with YYYY being the last Notify value sent by HPS software
13. Check if running a factory image by calling `rsu_running_factory()`.
  14. Look at the Failed image from saved RSU status. Use custom logic to determine if one or all application images are corrupted and something needs to be done.

#### 6.7.1.1. Recover Decision Firmware Procedure

1. Find a slot which can be used to store the recovery image for the procedure, and erase it.
2. Add a factory update image, or a decision firmware update image to the slot, using the U-Boot or LIBRSU APIs. This also makes it highest priority.  
*Note:* The factory update image and the decision firmware update image need to be written to flash each time they are to be executed. This is because, depending on the firmware version, they can also erase themselves from flash after a successful execution.
3. Pass control to the above slot by loading the specific image directly with the U-Boot or LIBRSU APIs, or pulsing the nCONFIG signal.
4. The image updates the decision firmware, decision firmware data, and factory image if that type of image is used.
5. The image then removes itself from CPB.
6. The image then loads the highest priority image in the CPB.





## 7. Remote System Update Example

---

This chapter presents a complete Remote System Update example, including the following:

- Creating the initial flash image containing:
  - Bitstreams for a factory image
  - One application image
  - Two empty slots to contain additional application images
- Creating an SD card with:
  - U-Boot
  - Arm\* Trusted Firmware
  - Linux
  - LIBRSU
  - RSU client
  - Application image
  - Factory update image
  - Decision firmware update image
- Exercising the Linux RSU client application.
- Exercising the U-Boot RSU commands.

**Note:** This example uses FPGA configuration first mode. For instructions on how to modify the example to support HPS boot first mode, refer to *Using RSU with HPS First*.

For the Intel Quartus Prime Pro Edition software version and the HPS software component versions discussed in this document, refer to the *Component Versions* section.

### Related Information

[Component Versions](#) on page 98

### 7.1. Prerequisites

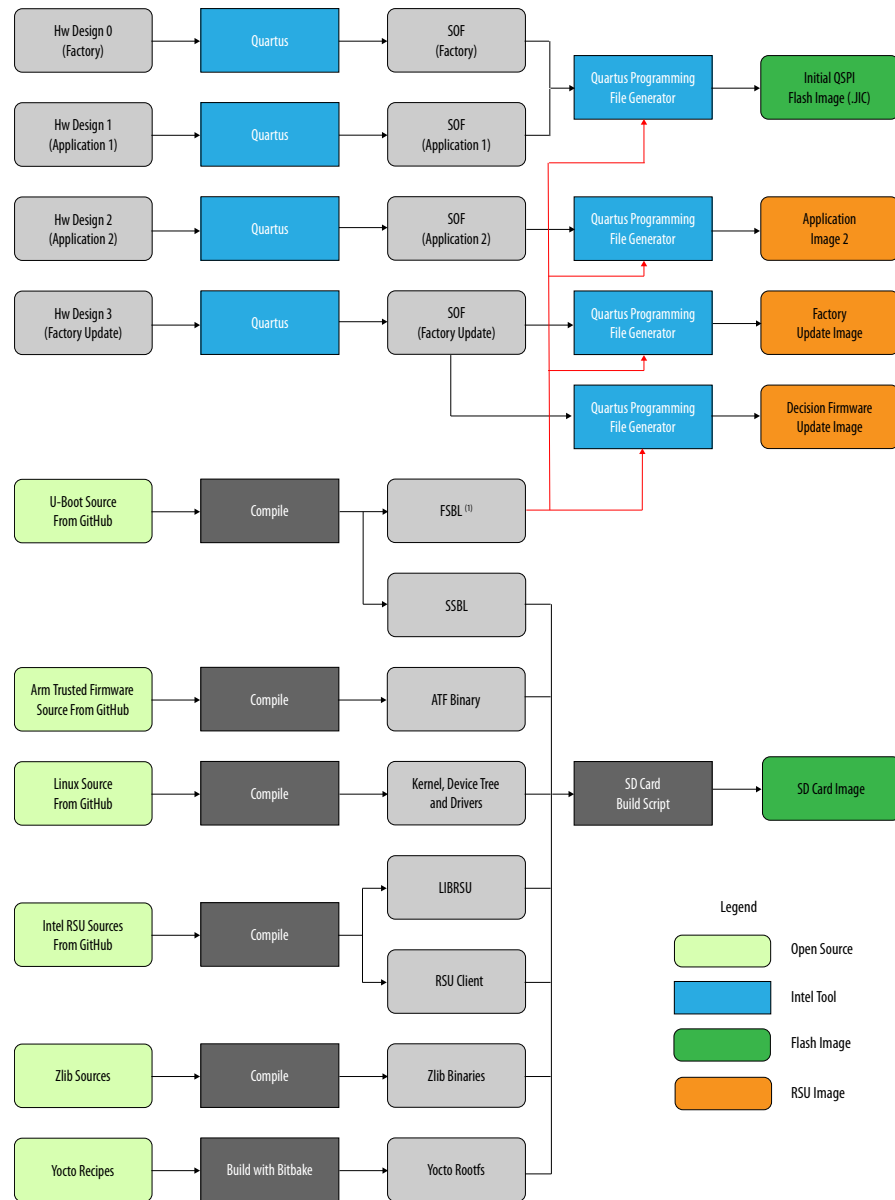
The following items are required to run the RSU example:

- Host PC running Ubuntu 18.04 LTS (other Linux versions may work too)
- Minimum 48 GB of RAM, required for compiling the hardware designs
- Intel Quartus Prime Pro Edition software version 21.2 – for compiling the hardware projects, generating the flash images and writing to flash
- Access to Internet – to download the hardware project archive, clone the git trees for U-Boot, Arm Trusted Firmware, Linux, zlib and LIBRSU and to build the Linux rootfs using Yocto.
- Intel Agilex SoC Development kit, production version – for running the example.

## 7.2. Building RSU Example Binaries

The diagram below illustrates the build flow used for this example.

Figure 8. RSU Example Build Flow



(1) It is not mandatory to use the same FSBL for each image. For example, if you have some low level recovery code in the factory image, then the actual binary may differ.

The end results of the build flow are:

- Initial flash image: contains the factory image, an application image and two empty application image partitions aka slots.
- SD card image: contains SSBL (U-Boot), ATF (Arm Trusted Firmware), Linux device tree, Linux kernel, Linux rootfs with the Intel RSU driver, LIBRSU, RSU Client, an application image, a factory update image and a decision firmware update image.

## 7.2.1. Setting up the Environment

Follow the instructions below to set up the required environment:

```
# create the top folder used to store all the example files
sudo rm -rf rsu_example && mkdir rsu_example && cd rsu_example
export TOP_FOLDER=`pwd`

# retrieve and setup the toolchain
cd $TOP_FOLDER
wget https://developer.arm.com/-/media/Files/downloads/gnu-a/10.2-2020.11/\
binrel/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu.tar.xz
tar xf gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu.tar.xz
export PATH=`pwd`/gcc-arm-10.2-2020.11-x86_64-aarch64-none-linux-gnu/bin:$PATH
export ARCH=arm64
export CROSS_COMPILE=aarch64-none-linux-gnu-
```

## 7.2.2. Building the Hardware Projects

Create four different hardware projects, based on the GHRD from GitHub with a few changes:

- Change the boot mode to FPGA first
- Use a different ID in the SystemID component, to make the binaries for each project slightly different.
- Change the behavior of watchdog timeout, to trigger an RSU event.
- Set the max\_retry parameter to 3, so that each application image and the factory image are tried up to three time when configuration failures occur.

The commands to create and compile the projects are listed below:

```
cd $TOP_FOLDER
# compile hardware designs: 0-factory, 1,2-applications, 3-factory update
rm -rf hw && mkdir hw && cd hw
wget https://github.com/altera-opensource/ghrd-socfpga/archive/\
ACDS-21.1pro-20.1std.zip
unzip ACDS-21.1pro-20.1std.zip
mv ghrd-socfpga-ACDS-21.1pro-20.1std/agilex_soc_devkit_ghrd .
rm -rf ghrd-socfpga-ACDS-21.1pro-20.1std ACDS-21.1pro-20.1std.zip
for version in {0..3}
do
rm -rf ghrd.$version
cp -r agilex_soc_devkit_ghrd ghrd.$version
cd ghrd.$version
make clean
make scrub_clean
rm -rf *.qpf *.qsf *.txt *.bin *.qsys ip/qsys_top/ ip/subsys_jtg_mst/ ip\
/subsys_periph/
sed -i 's/BOOTS_FIRST.*= */BOOTS_FIRST := fpga/g' Makefile
sed -i 's/ENABLE_WATCHDOG_RST.*= */ENABLE_WATCHDOG_RST := 1/g' Makefile
sed -i 's/WATCHDOG_RST_ACTION.*= */WATCHDOG_RST_ACTION := remote_update/g'
Makefile
sed -i 's/0xACD5CAFE/0xABAB000'$version'/g' create_ghrd_qsys.tcl
export IP_ROOTDIR=~/.intelFPGA_pro/21.2/ip
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
make generate_from_tcl
echo "set_global_assignment -name RSU_MAX_RETRY_COUNT 3" \
>> ghrd_agfb014r24a3e3vr0.qsf
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
make sof
cd ..
done
rm -rf agilex_soc_devkit_ghrd
cd ..
```

After completing the above steps, the following SOF files are created:

- hw/ghrd.0/output\_files/ghrd\_agfb014r24a3e3vr0.sof
- hw/ghrd.1/output\_files/ghrd\_agfb014r24a3e3vr0.sof
- hw/ghrd.2/output\_files/ghrd\_agfb014r24a3e3vr0.sof
- hw/ghrd.3/output\_files/ghrd\_agfb014r24a3e3vr0.sof

### 7.2.3. Building Arm Trusted Firmware

The following commands are used to retrieve the Arm Trusted Firmware (ATF) and compile it:

```
cd $TOP_FOLDER
rm -rf arm-trusted-firmware
git clone https://github.com/altera-opensource/arm-trusted-firmware
cd arm-trusted-firmware
# checkout the branch used for this document, comment out to use default
# git checkout -b test -t origin/socfpga_v2.4.1
make bl31 PLAT=agilex DEPRECATED=1 HANDLE_EA_EL3_FIRST=1
cd ..
```

After completing the above steps, the Arm Trusted Firmware binary file is created and is located here: arm-trusted-firmware/build/agilex/release/bl31.bin

### 7.2.4. Building U-Boot

The following commands can be used to get the U-Boot source code and compile it:

```
cd $TOP_FOLDER
rm -rf u-boot-socfpga
git clone https://github.com/altera-opensource/u-boot-socfpga
cd u-boot-socfpga
# checkout the branch used for this document, comment out to use default
# git checkout -b test -t origin/socfpga_v2021.01
# change the prompt text
sed -i 's/SOCFPGA_AGILEX #/SOCFPGA #/g' configs/socfpga_agilex_defconfig
make clean && make mrproper
make socfpga_agilex_atf_defconfig
make -j 24
ln -s ../arm-trusted-firmware/build/agilex/release/bl31.bin .
make u-boot.itb
cd ..
```

After completing the above steps, the following files are created:

- u-boot-socfpga/spl/u-boot-spl-dtb.hex — FSBL (U-boot SPL) hex file
- u-boot-socfpga/u-boot.itb — FIT image file containing SSBL (U-Boot) and ATF (Arm Trusted Firmware) binaries

**Note:**

Intel policy specifies that only the current and immediately previous U-Boot branches are kept on GitHub. At some point, the current U-Boot branch is removed and the above tag does not work anymore. In such an event, you can move to the latest release of all the components, as this should work. You must keep a local copy of the sources used to build your binaries in case you need to reproduce the build or make changes in the future.

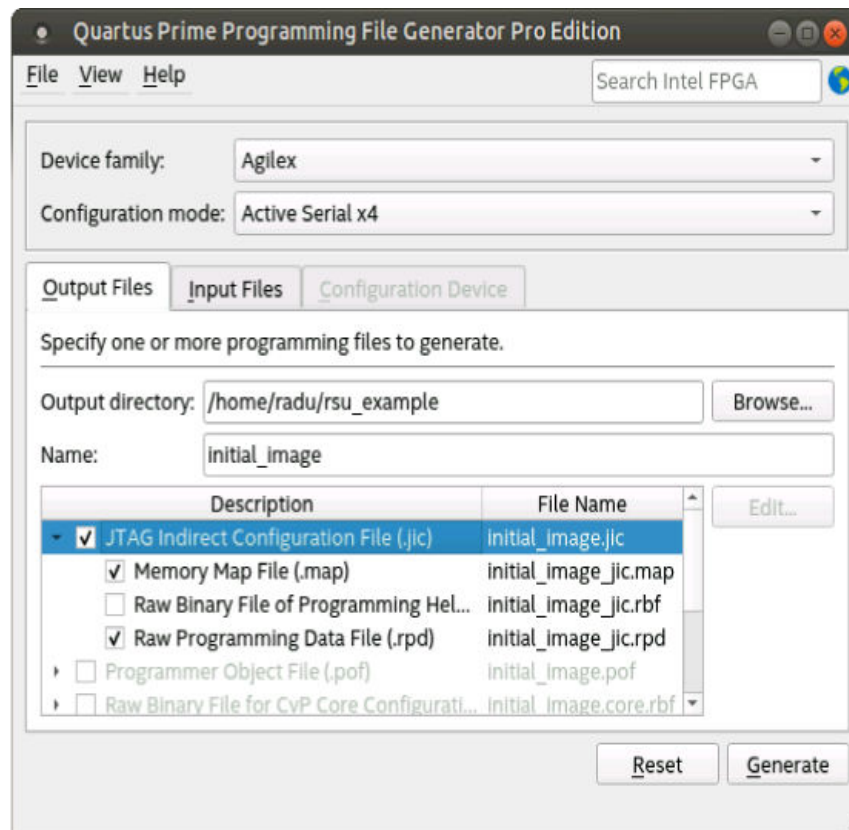
### 7.2.5. Creating the Initial Flash Image

Create an initial flash image containing the factory and application images and the associated RSU data structures.

1. Start the **Programming File Generator** tool by running the **qpfgw** command:

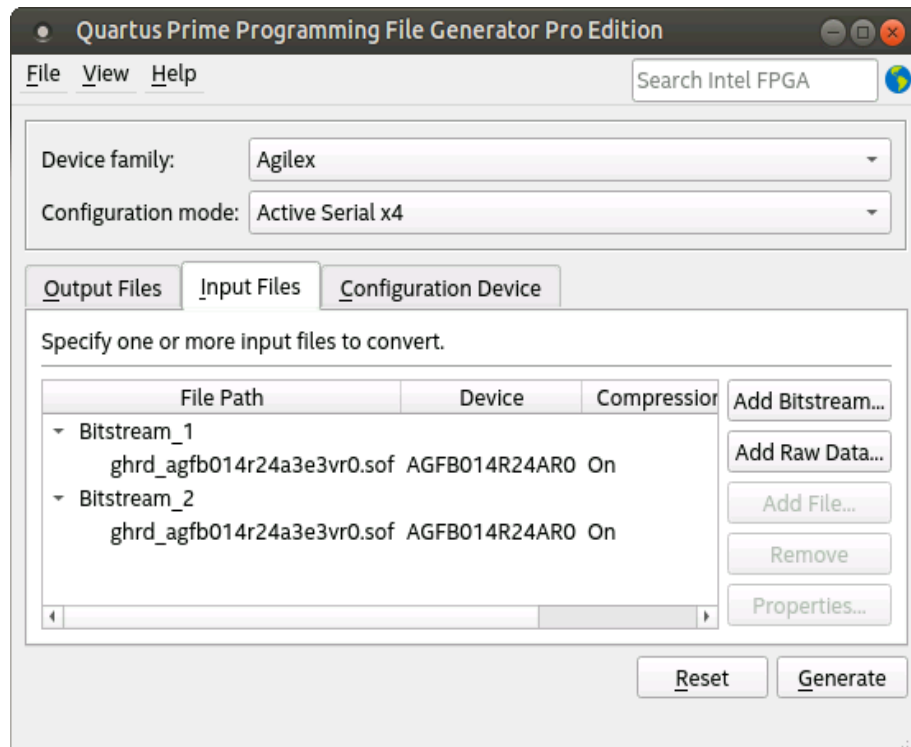
```
cd $TOP_FOLDER
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
qpfgw &
```

2. Select the **Device family** as **Intel Agilex**, and **Configuration mode** as **Active Serial x4**.
3. Change the **Name** to "initial\_image".
4. Select the output file type as **JTAG Indirect Configuration File (.jic)**, which is the format used by the Intel Quartus Prime Programmer tool for writing to the QSPI flash.
5. Select the optional **Memory Map File (.map)** file so that it is also generated. The .map file contains information about the resulted flash layout.
6. Select the optional **Raw Programming Data File (.rpd)** file so that it is also generated. This file contains the binary flash content, without anything else added. The window looks similar to this:

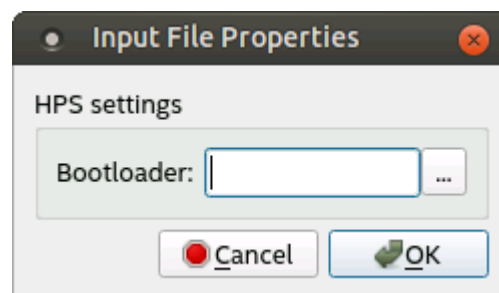


7. Click the **Raw Programming Data File (.rpd)** file to select it. Then click the **Edit ...** button and select the **Bitswap** option to be "On". This enables the RPD file to be usable by HPS software like U-Boot and Linux if needed.

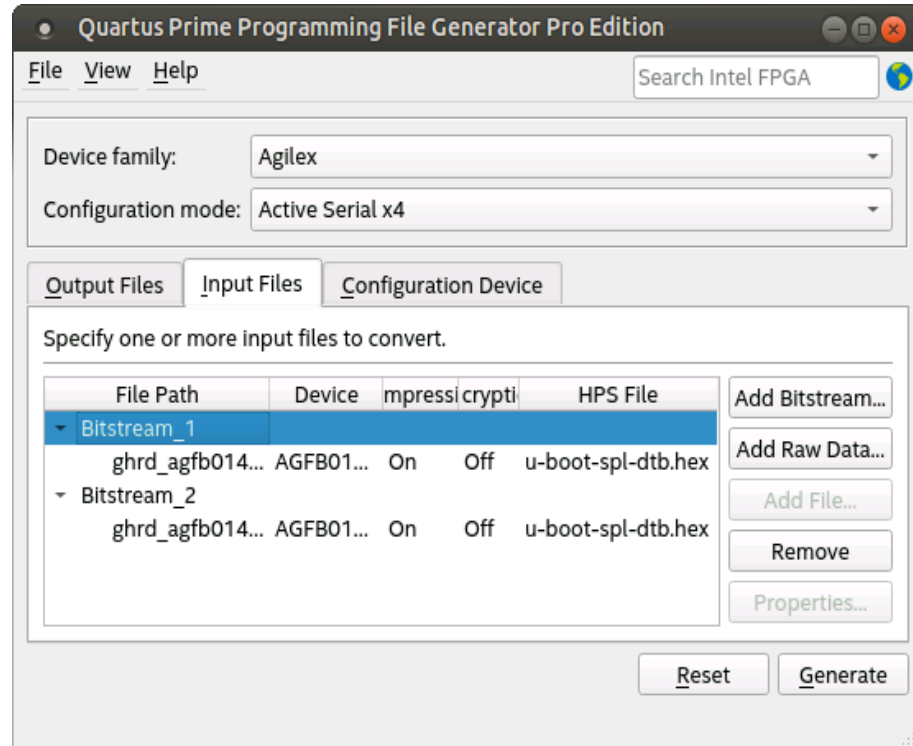
8. Once the output type was selected, click the **Input Files** tab.
9. In the **Input Files** tab click the **Add Bitstream** button, then browse to \$TOP\_FOLDER/hw/ghrd.0/output\_files, select the file ghrd\_agfb014r24a3e3vr0.sof, and then click **Open**. This is the initial factory image. Do the same for the \$TOP\_FOLDER/hw/ghrd.1/output\_files/ghrd\_agfb014r24a3e3vr0.sof image. This is the initial application image. The tab now looks like below:



10. Click the first .sof file, then click the **Properties** button on the right side. This opens the window to browse for the FSBL and select authentication and encryption settings.

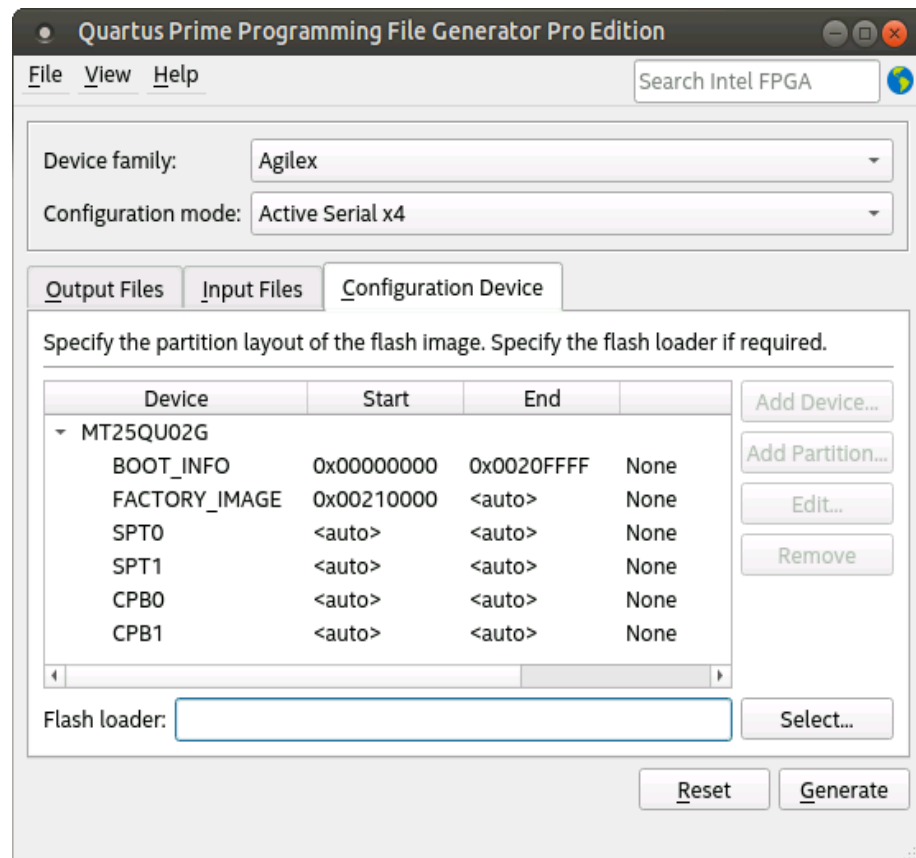


11. Click the **Bootloader ... (Browse)** button and select the file \$TOP\_FOLDER/u-boot-socfpga/spl/u-boot-spl-dtb.hex, then click OK.
12. Click the second .sof file and add the same FSBL file to it. The **Input Files** tab now looks like shown below:



13. Click the **Configuration Device** tab. Note that the tab is only enabled once at least one input file was added in the **Input Files** tab.
14. Because more than one input file was added in the **Input Files** tab, it displays the options for remote system update. Otherwise, it only enables the standard configuration flow.
15. In the **Configuration Device** tab, click **Add Device**, select the **MT25QU02G** in the dialog box window, then click **OK**. Once that is done, the window displays the default initial partitioning for RSU:





*Note:* You can also use another supported flash device, because this example only needs 512Mb of flash space. The Intel Quartus Prime Programmer displays some warnings in case another supported 512 Mb or larger flash is used, but the example works.

16. Select the **FACTORY\_IMAGE** entry, and click the **Edit...** button. The **Edit Partition** window pops up. Select the **Input file** as **Bitstream\_1 (ghrd\_agfb014r24a3e3vr0.sof)**. Change **Address Mode** to **Block** because you want to make sure you are leaving enough space for the biggest factory image you anticipate using. Set the **End Address** to **0x0090FFFF** in order to reserve 7MB for the factory image. This end address was calculated by adding 8MB to the end of the **BOOT\_INFO** partition. Click **OK**.

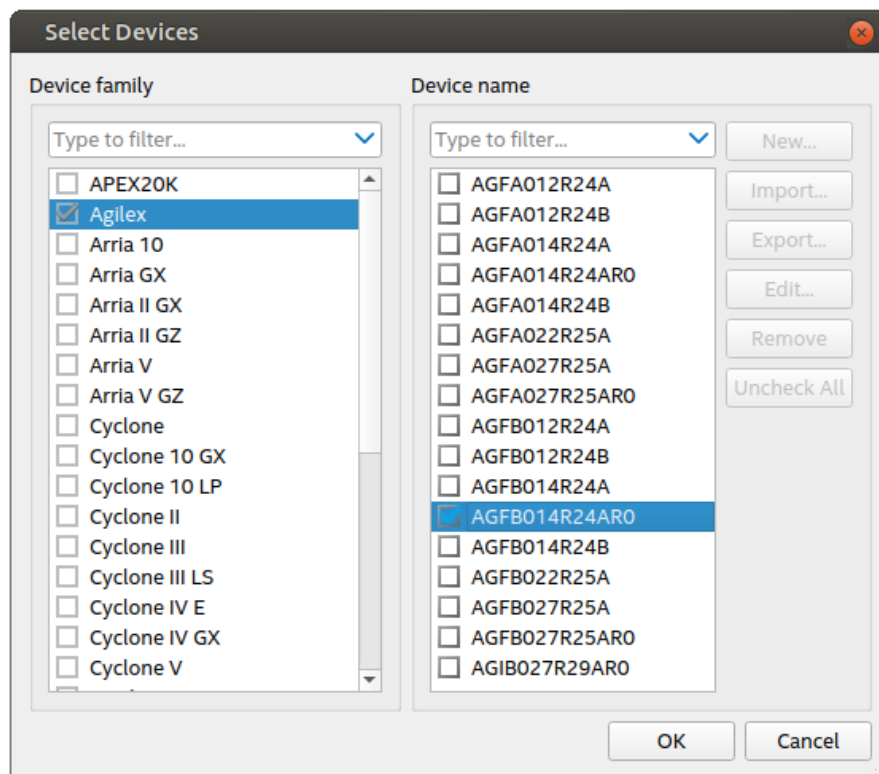
*Note:* The **Page** property for FACTORY\_IMAGE partition must always be set to 0. This means that the FACTORY\_IMAGE can retry only after all the application images failed.

17. Select the **MT25QU02G** flash device in the **Configuration Device** tab by clicking it, then click the **Add Partition...** button to open the **Add Partition** window. Leave the **Name** as **P1** and select the **Input file** as **Bitstream\_2(ghrd\_agfb014r24a3e3vr0.sof)**. This becomes the initial application image. Select the **Page** as **1** – this means it has the highest priority of all application images. Select the **Address Mode** as **Block** and allocate 16MB of data by setting **Start Address** = **0x01000000** and **End Address** = **0x01FFFFFF**.

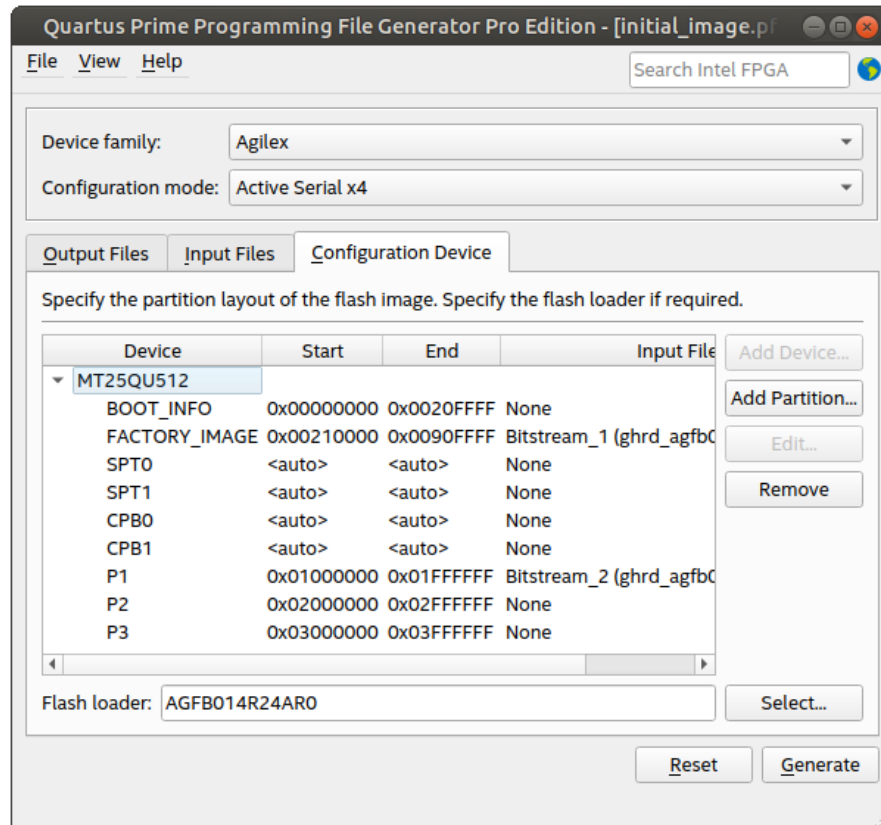
The Page property is used by the Programming File Generator to determine the order in which images appear initially in the configuration pointer block. The highest priority is the one with Page=1, then the one with Page=2, and so on. The Programming File Generator issues an error if there are multiple partitions with the same page number, or if there are any “gaps” as in having a Page=1 then a Page=3, without a Page=2 for example.

Only up to seven partitions can contain application images at initial flash image creation time. This limitation does not have adverse effects, as typically at creation time it is expected to have just a factory image and one application image.

18. Create two more partitions P2 and P3 using the same procedure as for the previous step, except set the **Input file** to **None**, leave **Page** unchanged (it does not matter for empty partitions) and set the start and end addresses as follows:
  - P2: **Start Address** = **0x02000000** and **End Address** = **0x02FFFFFF**.
  - P3: **Start Address** = **0x03000000** and **End Address** = **0x03FFFFFF**.
19. Click **Select ...** to select the Flash loader. The flash loader becomes part of the JIC file and is used by the Flash Programmer tool. Select the desired **Device family** and **Device name** as shown below:



The **Configuration Device** tab now looks like as shown below:



20. Click **File ► Save As ..** and save the file as \$TOP\_FOLDER/initial\_image.pfg. This file can be useful later, if you wanted to re-generate the initial image by using the command:

```
cd $TOP_FOLDER
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
quartus_pfg -c initial_image.pfg
```

*Note:* The created pfg file is actually an XML file which can be manually edited to replace the absolute file paths with relative file paths. You cannot directly edit the .pfg file for other purposes. The .pfg file can be opened from Programming File Generator, if changes are needed.

21. Click the **Generate** button to generate the initial flash image as \$TOP\_FOLDER/initial\_image.jic and the map file as \$TOP\_FOLDER/initial\_image\_jic.map. A dialog box opens indicating the files were generated successfully.

## 7.2.6. Creating the Application Image

The following commands are used to create the application image used in this example:

```
cd $TOP_FOLDER
mkdir -p images
rm -rf images/application2.rpd
```

```
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
quartus_pfg -c hw/ghrd.2/output_files/ghrd_agfb014r24a3e3vr0.sof \
  images/application2.rpd \
  -o hps_path=u-boot-socfpga/spl/u-boot-spl-dtb.hex \
  -o mode=ASX4 -o start_address=0x000000 -o bitswap=ON
```

The following application image is created: images/application2.rpd.

### 7.2.7. Creating the Factory Update Image

The following commands are used to create the factory update image used in this example:

```
cd $TOP_FOLDER
mkdir -p images
rm -f images/factory_update.rpd
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
quartus_pfg -c hw/ghrd.3/output_files/ghrd_agfb014r24a3e3vr0.sof \
  images/factory_update.rpd \
  -o hps_path=u-boot-socfpga/spl/u-boot-spl-dtb.hex \
  -o mode=ASX4 -o start_address=0x000000 -o bitswap=ON \
  -o rsu_upgrade=ON
```

The following factory update image is created: images/factory\_update.rpd.

### 7.2.8. Creating the Decision Firmware Update Image

The following commands are used to create the decision firmware update image used in this example:

```
cd $TOP_FOLDER
mkdir -p images
rm -f images/decision_firmware_update.rpd
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
quartus_pfg -c hw/ghrd.3/output_files/ghrd_agfb014r24a3e3vr0.sof \
  images/decision_firmware_update.rpd \
  -o hps_path=u-boot-socfpga/spl/u-boot-spl-dtb.hex \
  -o mode=ASX4 -o start_address=0x000000 -o bitswap=ON \
  -o rsu_upgrade=ON \
  -o firmware_only=1
```

The following decision firmware update image is created: images/decision\_firmware\_update.rpd.

**Note:**

The provided SOF file is used by the `quartus_pfg` to determine the parameters that are written to the decision firmware data structure. This includes QSPI clock and pin settings, the value of `max_retry` parameter, and the selected behavior of the HPS watchdog. The actual configuration data from the SOF file is not used.

### 7.2.9. Building Linux

The following commands can be used to obtain the Linux source code and build Linux:

```
cd $TOP_FOLDER
rm -rf linux-socfpga
git clone https://github.com/altera-opensource/linux-socfpga
cd linux-socfpga
# checkout the branch used for this document, comment out to use default
# git checkout -b test -t origin/socfpga-5.4.114-lts
# configure the RSU driver to be built into the kernel
sed -i 's/.*CONFIG_INTEL_STRATIX10_RSU=.*CONFIG_INTEL_STRATIX10_RSU=y/g' \
```

```
arch/arm64/configs/defconfig
make clean && make mrproper
make defconfig
make -j 24 Image dtbs
cd ..
```

After completing the above steps, the following files are created in the \$TOP\_FOLDER/linux-socfpga :

- arch/arm64/boot/Image — kernel image
- arch/arm64/boot/dts/intel/socfpga\_agilex\_socdk.dtb— kernel device tree

**Note:** The Intel SoC FPGA Linux releases on GitHub have a retention policy described at [Linux Git Guidelines](#). At some point, the current Linux branch is removed and the above tag does not work anymore. In such an event, you can move to the latest release of all the components, as this should work.

### 7.2.10. Building ZLIB

The ZLIB is required by LIBRSU. The following steps can be used to compile it:

```
cd $TOP_FOLDER
rm -rf zlib-1.2.11
wget http://zlib.net/zlib-1.2.11.tar.gz
tar xf zlib-1.2.11.tar.gz
rm zlib-1.2.11.tar.gz
cd zlib-1.2.11/
export LD=${CROSS_COMPILE}ld
export AS=${CROSS_COMPILE}as
export CC=${CROSS_COMPILE}gcc
./configure
make
export ZLIB_PATH=`pwd`
cd ..
```

**Note:** The version of zlib mentioned above is the one that was tested with this release. You may want to use the latest zlib version, as it may contain updates and bug fixes.

After the above steps are completed, the following items are available:

- \$TOP\_FOLDER/zlib-1.2.11/zlib.h — header file, used to compile files using zlib services
- \$TOP\_FOLDER/zlib-1.2.11/libz.so\* — shared objects, used to run executables linked against zlib APIs

### 7.2.11. Building LIBRSU and RSU Client

The following commands can be used to build the LIBRSU and the example client application:

```
cd $TOP_FOLDER
export ZLIB_PATH=`pwd`/zlib-1.2.11
rm -rf intel-rsu
git clone https://github.com/altera-opensource/intel-rsu
cd intel-rsu
# checkout the branch used for this document, comment out to use default
# git checkout -b test -t origin/master
cd lib
# add -I$(ZLIB_PATH) to CFLAGS
```

```
sed -i 's/\(CFLAGS := .*\)$/\1 -I\$(ZLIB_PATH)/g' makefile
make
cd ..
cd example
# add -L$(ZLIB_PATH) to LDFLAGS
sed -i 's/\(LDFLAGS := .*\)$/\1 -L\$(ZLIB_PATH)/g' makefile
make
cd ..
cd ..
```

The following files are created:

- `$TOP_FOLDER/intel-rsu/lib/librsu.so` — shared object required at runtime for running applications using `librsu`
- `$TOP_FOLDER/intel-rsu/etc/qspi.rc` — resource file for `librsu` configuration
- `$TOP_FOLDER/intel-rsu/example/rsu_client` — example client application using `librsu`

### 7.2.12. Building the Root File System

A root file system is required to boot Linux. There are a lot of ways to build a root file system, depending on your specific needs. This section shows how to build a small root file system using Yocto.

1. Various packages may be needed by the build system. On a Ubuntu 18.04 machine the following command was used to install the required packages:

```
sudo apt-get install gawk wget git-core diffstat unzip texinfo \
gcc-multilib build-essential chrpath socat cpio python python3 \
python3-pip python3-pexpect xz-utils debianutils iputils-ping \
python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm
```

2. Run the following commands to build the root file system:

```
cd $TOP_FOLDER
rm -rf yocto && mkdir yocto && cd yocto
git clone -b gatesgarth git://git.yoctoproject.org/poky.git
git clone -b master git://git.yoctoproject.org/meta-intel-fpga.git
source poky/oe-init-build-env ./build
echo 'MACHINE = "agilex"' >> conf/local.conf
echo 'BBLAYERS += " ${TOPDIR}/../meta-intel-fpga "' >> conf/bblayers.conf
echo 'IMAGE_FSTYPES = "tar.gz"' >> conf/local.conf
bitbake core-image-minimal
```

After the build completes, which can take a few hours depending on your host system processing power and Internet connection speed, the following root file system archive is created: `$TOP_FOLDER/yocto/build/tmp/deploy/images/agilex/core-image-minimal-agilex.tar.gz`

For more information about building Linux, including building the rootfs using Yocto, refer to the [Rocketboards Getting Started](#) web page.

### 7.2.13. Building the SD Card

The following commands can be used to create the SD card image used in this example:

```
cd $TOP_FOLDER
sudo rm -rf sd_card && mkdir sd_card && cd sd_card
wget https://releases.rocketboards.org/release/2021.04/gsrcd/
```

```
tools/make_sdimage_p3.py
chmod +x make_sdimage_p3.py
# prepare the fat partition contents
mkdir fat && cd fat
cp ../../u-boot-socfpga/u-boot.itb .
cp ../../linux-socfpga/arch/arm64/boot/Image .
cp ../../linux-socfpga/arch/arm64/boot/dts/intel/socfpga_agilex_socdk.dtb .
cp ../../images/*.rpd .
cd ..
# prepare the rootfs partition contents
mkdir rootfs && cd rootfs
sudo tar xf ../../yocto/build/tmp/deploy/images/agilex/\
core-image-minimal-agilex.tar.gz
sudo sed -i 's/agilex/linux/g' etc/hostname
sudo rm -rf lib/modules/*
sudo cp ../../images/*.rpd home/root
sudo cp ../../intel-rsu/example/rsu_client home/root/
sudo cp ../../intel-rsu/lib/librsu.so lib/
sudo cp ../../intel-rsu/etc/qspi.rc etc/librsu.rc
sudo cp ../../zlib-1.2.11/libz.so* lib/
cd ..
# create sd card image
sudo python3 ./make_sdimage_p3.py -f \
-P fat/*,num=1,format=vfat,size=100M \
-P rootfs/*,num=2,format=ext3,size=100M \
-s 256M \
-n sdcard_rsu.img
cd ..
```

This creates the SD card image as \$TOP\_FOLDER/sd\_card/sdcard\_rsu.img.

The following items are included in the rootfs on the SD card:

- U-Boot
- ATF
- Linux kernel, including RSU driver
- ZLIB shared objects
- LIBRSU shared objects and resource files
- RSU client application
- Application image
- Factory update image
- Decision firmware update image



## 7.3. Flashing the Binaries

### 7.3.1. Flashing the Initial RSU Image to QSPI

1. Make sure to install the QSPI SDM bootcard on the Intel Agilex SoC Development Kit
2. Power down the board if powered up
3. Configure the Intel Agilex SoC Development Kit switches to have MSEL set to JTAG.
4. Run the following command to write the image to SDM QSPI by using the command line version of the Intel Quartus Prime Programmer:

```
cd $TOP_FOLDER
quartus_pgm -c 1 -m jtag -o "pvi;./initial_image.jic"
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
```

5. Configure the Intel Agilex SoC Development Kit switches to have MSEL set to QSPI.

### 7.3.2. Writing the SD Card Image

1. Write the SD card image `$TOP_FOLDER/sd_card/sdcard_sl0_rsu.img` to a microSD card. You can use an USB micro SD card writer and the Linux `dd` command on your host PC to achieve this. Exercise caution when using the `dd` command, as incorrect usage can lead to your host Linux system becoming corrupted and non-bootable.
2. Insert the micro SD card in the slot on the Intel Agilex SoC Development kit HPS daughtercard.

## 7.4. Exercising U-Boot RSU Commands

### 7.4.1. Using U-Boot to Perform Basic Operations

This section demonstrates how to use U-Boot to perform the following basic operations:

- Querying the RSU status.
- Querying the number of slots and the information about them.
- Adding a new application image.
- Verifying that an application image was written correctly.
- Requesting a specific application image to be loaded.

**Note:**

This section assumes that the flash contains the initial RSU image. If that is not true, you need to re-flash the initial image, as shown in the *Flashing the Initial RSU Image to QSPI*.

1. Power cycle the board and press any key when prompted, to get to the U-Boot command prompt:

```
U-Boot 2021.01-12712-ge59d8e9eaa-dirty (Jun 30 2021 - 10:40:17
-0500)socfpga_agilex
```

```
CPU: Intel FPGA SoCFPGA Platform (ARMv8 64bit Cortex-A53)
Model: SoCFPGA Agilex SoCDK
DRAM: 8 GiB
WDT: Started with servicing (10s timeout)
MMC: dwmmc0@ff808000: 0
Loading Environment from MMC... *** Warning - bad CRC, using default
environment

In: serial0@ffc02000
Out: serial0@ffc02000
Err: serial0@ffc02000
Net:
Warning: ethernet@ff800000 (eth0) using random MAC address -
62:08:b9:b7:79:2d
eth0: ethernet@ff800000
Hit any key to stop autoboot: 5

0
SOCFPGA #
```

2. Run the `rsu` command without parameters, to display its help message, and usage options:

```
SOCFPGA # rsu
rsu - Agilex SoC Remote System Update

Usage:
rsu dtb - Update Linux DTB qspi-boot partition offset with spt0 value
list - List down the available bitstreams in flash
slot_by_name <name> - find slot by name and display the slot number
slot_count - display the slot count
slot_disable <slot> - remove slot from CPB
slot_enable <slot> - make slot the highest priority
slot_erase <slot> - erase slot
slot_get_info <slot> - display slot information
slot_load <slot> - load slot immediately
slot_load_factory - load factory immediately
slot_priority <slot> - display slot priority
slot_program_buf <slot> <buffer> <size> - program buffer into slot, and
make it highest priority
slot_program_buf_raw <slot> <buffer> <size> - program raw buffer into slot
slot_program_factory_update_buf <slot> <buffer> <size> - program factory
update buffer into slot, and make it highest priority
slot_rename <slot> <name> - rename slot
slot_size <slot> - display slot size
slot_verify_buf <slot> <buffer> <size> - verify slot contents against buffer
slot_verify_buf_raw <slot> <buffer> <size> - verify slot contents against
raw buffer
status_log - display RSU status
update <flash_offset> - Initiate firmware to load bitstream as specified by
flash_offset
notify <value> - Let SDM know the current state of HPS software
clear_error_status - clear the RSU error status
reset_retry_counter - reset the RSU retry counter
display_dcmf_version - display DCMF versions and store them for SMC handler
usage
display_dcmf_status - display DCMF status and store it for SMC handler usage
display_max_retry - display max_retry parameter, and store it for SMC
handler usage
restore_spt <address> - restore SPT from an address
save_spt <address> - save SPT to an address
create_empty_cpb - create a empty CPB
restore_cpb <address> - restore CPB from an address
save_cpb <address> - save CPB to an address
check_running_factory - check if currently running the factory image
```

3. Run the `rsu list` command to display the RSU partitions, CPBs, the currently running image and the status:

```
SOCFPGA # rsu list
RSU: Remote System Update Status
Current Image      : 0x01000000
Last Fail Image    : 0x00000000
State              : 0x00000000
Version            : 0x00000202
Error location     : 0x00000000
Error details      : 0x00000000
Retry counter      : 0x00000000
RSU: Sub-partition table 0 offset 0x00910000
RSU: Sub-partition table 1 offset 0x00918000
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
RSU: Sub-partition table content
      BOOT_INFO      Offset: 0x0000000000000000      Length: 0x00210000
Flag : 0x00000003
      FACTORY_IMAGE   Offset: 0x0000000000210000      Length: 0x00700000
Flag : 0x00000003
      P1              Offset: 0x0000000000100000      Length: 0x01000000
Flag : 0x00000000
      SPT0            Offset: 0x0000000000091000      Length: 0x00008000
Flag : 0x00000001
      SPT1            Offset: 0x0000000000091800      Length: 0x00008000
Flag : 0x00000001
      CPB0            Offset: 0x0000000000092000      Length: 0x00008000
Flag : 0x00000001
      CPB1            Offset: 0x0000000000092800      Length: 0x00008000
Flag : 0x00000001
      P2              Offset: 0x0000000000200000      Length: 0x01000000
Flag : 0x00000000
      P3              Offset: 0x0000000000300000      Length: 0x01000000
Flag : 0x00000000
RSU: CMF pointer block offset 0x00920000
RSU: CMF pointer block's image pointer list
Priority 1 Offset: 0x0000000000100000 nslot: 0
```

**Note:** The `rsu list` U-Boot command does not have a RSU client equivalent. Instead, the same information can be retrieved using other commands, as shown next.

4. Run the `rsu status_log` command to display the RSU status:

```
SOCFPGA # rsu status_log
Current Image      : 0x01000000
Last Fail Image    : 0x00000000
State              : 0x00000000
Version            : 0x00000202
Error location     : 0x00000000
Error details      : 0x00000000
Retry counter      : 0x00000000
```

Application image P1 is loaded, as it is the highest priority in the CPB. There are no errors.

5. Run the `rsu display_dcmf_version` to query and display the decision firmware versions:

```
SOCFPGA # rsu display_dcmf_version
DCMF0 version = 21.2.0
DCMF1 version = 21.2.0
DCMF2 version = 21.2.0
DCMF3 version = 21.2.0
```

## 6. Display information about the slots:

```
SOCFPGA # rsu slot_count
Number of slots = 3.
SOCFPGA # rsu slot_get_info 0
NAME: P1
OFFSET: 0x0000000001000000
SIZE: 0x01000000
PRIORITY: 1
SOCFPGA # rsu slot_get_info 1
NAME: P2
OFFSET: 0x0000000002000000
SIZE: 0x01000000
PRIORITY: [disabled]
SOCFPGA # rsu slot_get_info 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: [disabled]
SOCFPGA # rsu slot_size 0
Slot 0 size = 16777216.
SOCFPGA # rsu slot_size 1
Slot 1 size = 16777216.
SOCFPGA # rsu slot_size 2
Slot 2 size = 16777216.
```

## 7. Erase slot 1 and add the application2.rpd image to slot 1:

```
SOCFPGA # rsu slot_erase 1
Slot 1 erased
SOCFPGA # load mmc 0:1 $loadaddr application2.rpd
3358720 bytes read in 153 ms (20.9 MiB/s)
SOCFPGA # rsu slot_program_buf 1 $loadaddr $filesize
Slot 1 was programmed with buffer=0x0000000002000000 size=3358720.
```

## 8. Verify that the application image was written correctly:

```
SOCFPGA # rsu slot_verify_buf 1 $loadaddr $filesize
Slot 1 was verified with buffer=0x0000000002000000 size=3358720.
```

## 9. Confirm that slot 1 (partition P2) contains now the highest priority image:

```
SOCFPGA # rsu slot_get_info 1
NAME: P2
OFFSET: 0x0000000002000000
SIZE: 0x01000000
PRIORITY: 1
```

## 10. Power-cycle the board, stop U-Boot and check the RSU status log:

```
SOCFPGA # rsu status_log
Current Image : 0x02000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

The application image from slot 1 (partition P2) was loaded, because it is marked as the highest priority in the CPB.

## 11. Load the application image from slot 0 (partition P1) by running any of the following two commands:

```
SOCFPGA # rsu update 0x01000000
RSU: RSU update to 0x0000000001000000
```

or

```
SOCFPGA # rsu slot_load 0
```

12. Load the newly requested image. Stop at U-Boot prompt and check the status log to confirm it:

```
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

*Note:* In U-Boot, the effect of requesting a specific image is immediate. On Linux, it only takes effect on the next `reboot` command.

### Related Information

- [Flashing the Initial RSU Image to QSPI](#) on page 65
- [Watchdog and Max Retry Operation](#) on page 69
- [Updating the Factory Image Using U-Boot](#) on page 71
- [Fallback on Flash Corruption of Application Images](#) on page 73
- [Flashing the Initial RSU Image to QSPI](#) on page 65

## 7.4.2. Watchdog and Max Retry Operation

This section uses U-Boot to demonstrate the following:

- RSU handling of watchdog timeouts.
- `max retry` feature, which allows each image to be tried to be loaded up to three times.
- RSU notify, which allows the HPS software state to be reported before and retrieved after a watchdog timeout.
- Clearing the RSU status error fields.
- Resetting the current retry counter value.

*Note:* The commands listed in this section rely on the commands from the *Basic Operation* section running first, specifically adding an application image to the P2 flash partition.

1. Power-cycle the board, stop U-Boot and check the RSU status log:

```
SOCFPGA # rsu status_log
Current Image : 0x02000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

The application image from slot 1 (partition P2) was loaded, since it is marked as the highest priority in the CPB.

What do the fields mean:

- Retry counter is 0x00000000—first attempt to load this image.
- State is 0x00000000—No errors to report

2. Query and display the `max_retry` value:

```
SOCFPGA # rsu display_max_retry
max_retry = 3
```

3. Cause a watchdog timeout by setting the timeout value to lowest possible. This prevents U-Boot from being able to service it in time:

```
SOCFPGA # mw.l 0xffd00204 0
```

4. The watchdog immediately times out, and SDM reloads the same application image, since the `max_retry` parameter is set to three. Look at the U-Boot console and check the status log:

```
SOCFPGA # rsu status_log
Current Image : 0x02000000
Last Fail Image : 0x02000000
State : 0xf0060001
Version : 0x0acf0202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000001
```

The same P2 image is loaded, but the `retry` counter value is now one, which means this is the second retry for this image to be loaded. The `version` field shows the last failure was by an application image (0xACF). The `state` field shows the last error was a watchdog timeout (0xF006) and that the latest notify value from HPS software was from SPL loading U-Boot (0x0001).

5. Clear the error status so we can see the next errors. Query the status to show the errors were cleared:

```
SOCFPGA # rsu clear_error_status
SOCFPGA # rsu status_log
Current Image : 0x02000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000001
```

6. Cause another watchdog timeout. At the U-Boot prompt, query the RSU log and observe that the `retry` counter is now two:

```
SOCFPGA # rsu status_log
Current Image : 0x02000000
Last Fail Image : 0x02000000
State : 0xf0060001
Version : 0x0acf0202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000002
```

7. Clear the error status so we can see the next errors.

8. Use the `notify` command to let SDM know the state of HPS software as a 16bit value:

```
SOCFPGA0 # rsu notify 0x1234
```

9. Cause a watchdog timeout one more time and display the RSU status log after the restart:

```
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x02000000
State : 0xf0061234
Version : 0x0acf0202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

The SDM loaded the next application image in the CPB (P1), and it reports that the image P2 failed. The state indicates that a watchdog timeout occurred (upper 16 bits = 0xF006) and that the notify value reported by HPS software was 0x1234. The upper 16 bits of version are set to 0x0ACF which means the previous error was reported by the application image firmware. For more information, refer to [RSU Status and Error Codes](#) on page 114.

10. Clear the errors and display the status - it shows no errors:

```
SOCFPGA # rsu clear_error_status
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

11. Cause a watchdog timeout, boot to U-Boot, and display the status - it shows the retry counter is one:

```
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x01000000
State : 0xf0060001
Version : 0x0acf0202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000001
```

12. Reset the current retry counter value to zero and query the status again to confirm it:

```
SOCFPGA # rsu reset_retry_counter
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x01000000
State : 0xf0060001
Version : 0x0acf0202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

### 7.4.3. Updating the Factory Image Using U-Boot

This section demonstrates how to use U-Boot to update the factory image.

**Note:** The commands listed in this section rely on the commands from the *Basic Operation* section running first, specifically adding an application image to the P2 flash partition.

1. Power-cycle the board, stop U-Boot and check the RSU status log:

```
SOCFPGA # rsu status_log
Current Image : 0x02000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

The application image from slot 1 (partition P2) was loaded, because it is marked as the highest priority in the CPB.

2. Confirm that slot 2 is not used, erase slot 2, write the factory update image to it, and verify it was written correctly:

```
SOCFPGA # rsu slot_get_info 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: [disabled]
SOCFPGA # rsu slot_erase 2
CPBs are GOOD!!!
Slot 2 erased.
SOCFPGA # load mmc 0:1 $loadaddr factory_update.rpd
3485696 bytes read in 158 ms (21 MiB/s)
SOCFPGA # rsu slot_program_factory_update_buf 2 $loadaddr $filesize
CPBs are GOOD!!!
Slot 2 was programmed with buffer=0x0000000002000000 size=3485696.
SOCFPGA # rsu slot_verify_buf 2 $loadaddr $filesize
Slot 2 was verified with buffer=0x0000000002000000 size=3485696.
```

3. Confirm that slot 2 is now the highest priority in the CPB:

```
SOCFPGA # rsu slot_get_info 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: 1
```

4. Instruct the SDM to load the factory update image from slot 2:

```
SOCFPGA # rsu slot_load 2
Slot 2 loading.
```

5. The factory update image runs for a few seconds, and updates the decision firmware, decision firmware data and factory image in flash. Then it removes itself from the CPB and loads the now highest priority image in the CPB. At the U-Boot prompt, confirm that P2 is now loaded and P3 is disabled:

```
SOCFPGA # rsu status_log
Current Image : 0x02000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
SOCFPGA # rsu slot_get_info 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: [disabled]
```



#### 7.4.4. Fallback on Flash Corruption of Application Images

This section uses U-Boot to demonstrate falling back in case of configuration errors caused by flash corruption of application images.

**Note:** The commands listed in this section rely on the commands from the *Basic Operation* section running first, specifically adding an application image to the P2 flash partition.

1. Power-cycle the board, stop U-Boot and check the RSU status log:

```
SOCFPGA # rsu status_log
Current Image      : 0x02000000
Last Fail Image    : 0x00000000
State              : 0x00000000
Version            : 0x00000202
Error location     : 0x00000000
Error details      : 0x00000000
Retry counter      : 0x00000000
```

The application image from slot 1 (partition P2) was loaded, because it is marked as the highest priority in the CPB.

What do the fields mean:

- Retry counter is 0x00000000—first attempt to load this image.
- State is 0x00000000—No errors to report

2. Corrupt the image in the slot 1 by erasing some of it:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x02000000 0x4000
SF: 16384 bytes @ 0x2000000 Erased: OK
```

3. Power cycle the board, stop at U-Boot prompt, and query the RSU log:

```
SOCFPGA # rsu status_log
Current Image      : 0x01000000
Last Fail Image    : 0x02000000
State              : 0xf004d003
Version            : 0x0dcf0202
Error location     : 0x00000000
Error details      : 0x00000000
Retry counter      : 0x00000000
```

The current image is P1, and the P2 shows as a failure. Note that SDM tried to load the image three times from flash, as specified by the `max retry` option. The top 16 bits of the version field are set as 0x0DCF which means the error was caused reported by the decision firmware, as it was not able to load the image. The top 16 bits of the state field are set to 0xF004, which indicate an internal error. For more information, refer to [RSU Status and Error Codes on page 114](#).

4. Clear the error status and display the log again to confirm it was cleared:

```
SOCFPGA # rsu clear_error_status
SOCFPGA # rsu status_log
Current Image      : 0x01000000
Last Fail Image    : 0x00000000
State              : 0x00000000
Version            : 0x00000202
Error location     : 0x00000000
Error details      : 0x00000000
Retry counter      : 0x00000000
```

## 7.4.5. Additional Flash Corruption Detection and Recovery

This section presents examples of detecting and recovering corrupted decision firmware, decision firmware data, configuration pointer blocks and sub-partition tables with the aid of the U-Boot RSU commands.

### 7.4.5.1. Corrupted Decision Firmware

This example uses U-Boot commands to demonstrate detecting that a decision firmware copy is corrupted, and recovering it by running a decision firmware update image.

**Note:** The commands listed in this example assume the initial flash image (JIC) was written to flash, with no other changes.

1. Power cycle the board, boot to U-Boot prompt.
2. Display the RSU status:

```
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

The top four bits of the state field are 0x0 which means the currently used decision firmware index is zero. There are no errors.

3. Display the decision firmware status:

```
SOCFPGA # rsu display_dcmf_status
DCMF0: OK
DCMF1: OK
DCMF2: OK
DCMF3: OK
```

The command compares the currently used decision firmware copy 0 with the other copies, and displays that all decision firmware copies are fine.

4. Corrupt decision firmware copies 0 and 2:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total 256 MiB
SOCFPGA # sf erase 0 0x1000
SF: 4096 bytes @ 0x0 Erased: OK
SOCFPGA # sf erase 0x80000 0x1000
SF: 4096 bytes @ 0x80000 Erased: OK
```

5. Power cycle the board, boot to U-Boot prompt, display the RSU status and decision firmware status:

```
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x10000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

```
SOCFPGA # rsu display_dcmf_status
DCMF0: Corrupted
DCMF1: OK
DCMF2: Corrupted
DCMF3: OK
```

The currently used copy of the decision firmware is 1, as indicated by top four bits of the version field. The decision firmware copies 0 and 2 are detected as corrupted.

6. Erase an unused slot, add the decision firmware update image to the slot, verify it was written fine, and confirm it is now the highest priority slot:

```
SOCFPGA # rsu slot_erase 2
Slot 2 erased.
SOCFPGA # fatload mmc 0:1 ${loadaddr} decision_firmware_update.rpd
151552 bytes read in 9 ms (16.1 MiB/s)
SOCFPGA # rsu slot_program_factory_update_buf 2 ${loadaddr} ${filesize}
Slot 2 was programmed with buffer=0x0000000002000000 size=151552.
SOCFPGA # rsu slot_verify_buf 2 ${loadaddr} ${filesize}
Slot 2 was verified with buffer=0x0000000002000000 size=151552.
SOCFPGA # rsu slot_get_info 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: 1
```

7. Pass control to the decision firmware update image:

```
SOCFPGA # rsu slot_load 2
```

8. The decision firmware update image writes new decision firmware copies and new decision firmware data to flash, remove itself from CPB, then pass control to the highest priority image.
9. Stop at U-Boot prompt and confirm the decision firmware copies are all good, the decision firmware update image was removed from CPB, and the highest priority image is running:

```
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
SOCFPGA # rsu display_dcmf_status
DCMF0: OK
DCMF1: OK
DCMF2: OK
DCMF3: OK
SOCFPGA # rsu slot_get_info 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: [disabled]
```

#### 7.4.5.2. Corrupted Decision Firmware Data

This example uses U-Boot commands to demonstrate detecting that the decision firmware data is corrupted, and recovering it by running a decision firmware update image.

**Note:** The commands listed in this example assume the initial flash image (JIC) was written to flash, with no other changes.

1. Power cycle the board, boot to U-Boot prompt.
2. Display the RSU status:

```
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

There are no errors.

3. Corrupt decision firmware data:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total 256 MiB
SOCFPGA # sf erase 0x100000 0x1000
SF: 4096 bytes @ 0x100000 Erased: OK
```

4. Power cycle the board, boot to U-Boot prompt, display the RSU status:

```
SOCFPGA # rsu status_log
Current Image : 0x00110000
Last Fail Image : 0x00100000
State : 0xf004d00f
Version : 0x0dcf0202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

The State contains the special error code 0xf004d00f indicating that the decision firmware data was corrupted. The current image is listed as the factory image. The error source is listed as 0xdcf, meaning the error was reported by the decision firmware. The Last Fail Image is set to 0x00100000, which is a special value indicating an error was reported by the decision firmware.

5. Erase an unused slot, add the decision firmware update image to the slot, verify it was written fine, and confirm it is now the highest priority running slot:

```
SOCFPGA # rsu slot_erase 2
Slot 2 erased.
SOCFPGA # fatload mmc 0:1 ${loadaddr} decision_firmware_update.rpd
151552 bytes read in 9 ms (16.1 MiB/s)
SOCFPGA # rsu slot_program_factory_update_buf 2 ${loadaddr} ${filesize}
Slot 2 was programmed with buffer=0x0000000002000000 size=151552.
SOCFPGA # rsu slot_verify_buf 2 ${loadaddr} ${filesize}
Slot 2 was verified with buffer=0x0000000002000000 size=151552.
SOCFPGA # rsu slot_get_info 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: 1
```

6. Pass control to the decision firmware update image:

```
SOCFPGA # rsu slot_load 2
```

7. The decision firmware update image writes new decision firmware copies and new decision firmware data to flash, remove itself from CPB, then pass control to the highest priority image.
8. Stop at U-Boot prompt and confirm the decision firmware data is fine, the decision firmware update image was removed from CPB, and the highest priority image is running:

```
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
SOCFPGA # rsu slot_get_info 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: [disabled]
```

#### 7.4.5.3. Corrupted Configuration Pointer Block

This section uses U-Boot commands to demonstrate how configuration pointer block corruptions can be detected and recovered.

**Note:** The commands listed in this example assume the initial flash image (JIC) was written to flash, with no other changes.

1. Power up board, stop at U-Boot prompt.
2. Run the `rsu list` command to display a detailed status, including all the partitions, to determine the location of the SPTs:

```
SOCFPGA # rsu list
RSU: Remote System Update Status
Current Image : 0x01000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
RSU: Sub-partition table 0 offset 0x00910000
RSU: Sub-partition table 1 offset 0x00918000
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total 256 MiB
RSU: Sub-partition table content
      BOOT_INFO      Offset: 0x0000000000000000      Length: 0x00110000
Flag : 0x00000003
      FACTORY_IMAGE   Offset: 0x0000000000110000      Length: 0x00800000
Flag : 0x00000003
      P1              Offset: 0x0000000001000000      Length: 0x01000000
Flag : 0x00000000
      SPT0            Offset: 0x0000000000910000      Length: 0x00008000
Flag : 0x00000001
      SPT1            Offset: 0x0000000000918000      Length: 0x00008000
Flag : 0x00000001
      CPB0            Offset: 0x0000000000920000      Length: 0x00008000
Flag : 0x00000001
      CPB1            Offset: 0x0000000000928000      Length: 0x00008000
Flag : 0x00000001
      P2              Offset: 0x0000000002000000      Length: 0x01000000
Flag : 0x00000000
      P3              Offset: 0x0000000003000000      Length: 0x01000000
```

```
Flag : 0x00000000
RSU: CMF pointer block offset 0x00920000
RSU: CMF pointer block's image pointer list
Priority 1 Offset: 0x0000000001000000 nslot: 0
```

3. Corrupt CPB0 by erasing the corresponding flash area:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x0920000 0x1000
SF: 4096 bytes @ 0x920000 Erased: OK
```

4. Power cycle the board, the stop to U-Boot prompt and query the RSU status:

```
SOCFPGA # rsu status_log
FW detects corrupted CPB0 but CPB1 is fine
Restoring CPB0
Current Image : 0x01000000
Last Fail Image : 0x00100000
State : 0xf004d010
Version : 0x0dcf0202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

The State field has the special error code 0xf004d010 which indicates that CPB0 was corrupted. The Last Fail Image has the special value 0x00100000 which is used in this case. The Version field indicates that the error was reported by the decision firmware (0xDCf). The rsu\_init function which is called first time an U-Boot RSU command is executed detected the CPB corruption and recovered CPB0 from CPB1.

5. Save the CPB contents to a file, to be used later for recovery.

```
SOCFPGA # rsu save_cpb ${loadaddr}
4100 bytes CPB data saved
SOCFPGA # fatwrite mmc 0:1 ${loadaddr} cpb-backup.bin ${filesize}
4100 bytes written
```

6. Corrupt both CPBs by erasing the flash at their location:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x0920000 0x1000
SF: 4096 bytes @ 0x920000 Erased: OK
SOCFPGA # sf erase 0x0928000 0x1000
SF: 4096 bytes @ 0x928000 Erased: OK
```

7. Power cycle the board, boot to U-Boot prompt and query RSU status:

```
SOCFPGA # rsu status_log
FW detects both CPBs corrupted
Current Image : 0x00110000
Last Fail Image : 0x00100000
State : 0xf004d011
Version : 0x0dcf0202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

The Current Image is reported as being the factory image, as expected. The State field has the special error code 0xf004d011 which indicates that both CPBs were corrupted. The Last Fail Image has the special value 0x00100000

which is used in this case. The Version field indicates that the error was reported by the decision firmware (0xDCf). The `rsu_init` function which is called first time an U-Boot RSU command is executed detected and reported that both CPBs are corrupted.

8. Try to run a command which requires a valid CPB - it is rejected:

```
SOCFPGA # rsu slot_get_info 0
corrupted CPB --run rsu create_empty_cpb or rsu restore_cpb <address> first
```

9. Restore the saved CPB from the backup file that we created:

```
SOCFPGA # fatload mmc 0:1 ${loadaddr} cpb-backup.bin
4100 bytes read in 3 ms (1.3 MiB/s)
SOCFPGA # rsu restore_cpb ${loadaddr}
```

10. Clear the errors reported by firmware, as the CPB was restored.

```
SOCFPGA # rsu clear_error_status
```

Alternatively you can also power cycle or assert `nCONFIG` to clear the errors reported by firmware.

11. Try again to run a command which require a valid CPB - it succeeds:

```
SOCFPGA # rsu slot_get_info 1
NAME: P2
OFFSET: 0x0000000002000000
SIZE: 0x01000000
PRIORITY: 1
```

#### 7.4.5.4. Corrupted Sub-Partition Table

This example uses U-Boot commands to demonstrate how sub-partition table corruptions can be detected and recovered.

**Note:** The commands listed in this example assume the initial flash image (JIC) was written to flash, with no other changes.

1. Power up the board, boot up to the U-Boot prompt and query the RSU status:

```
SOCFPGA # rsu status_log
Current Image : 0x01000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
```

The highest priority image is running, and there are no errors.

2. Corrupt SPT0 file by erasing the flash at its location:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x0910000 0x1000
SF: 4096 bytes @ 0x910000 Erased: OK
```

3. Power cycle the board, stop to U-Boot prompt and query RSU status:

```
SOCFPGA # rsu status_log
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
```

```
Bad SPT0 magic number 0xFFFFFFFF
Restoring SPT0
Current Image      : 0x01000000
Last Fail Image    : 0x00000000
State              : 0x00000000
Version           : 0x00000202
Error location     : 0x00000000
Error details      : 0x00000000
Retry counter      : 0x00000000
```

The decision firmware loads the highest priority image, and it does not look at the SPTs. The `rsu_init` function is called when the first RSU U-Boot command is executed, it detects that the SPT0 is corrupted, and it recovers it from SPT1.

4. Save the currently used SPT to a file for backup purposes:

```
SOCFPGA # rsu save_spt ${loadaddr}
4100 bytes SPT data saved
SOCFPGA # fatwrite mmc 0:1 ${loadaddr} spt-backup.bin ${filesize}
4100 bytes written
```

5. Corrupt both SPTs by erasing the flash at their locations:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x910000 0x1000
SF: 4096 bytes @ 0x910000 Erased: OK
SOCFPGA # sf erase 0x918000 0x1000
SF: 4096 bytes @ 0x918000 Erased: OK
```

6. Power cycle the board, stop to U-Boot prompt and query RSU status:

```
SOCFPGA # rsu status_log
Bad SPT1 magic number 0xFFFFFFFF
Bad SPT0 magic number 0xFFFFFFFF
no valid SPT0 and SPT1 found
Current Image      : 0x01000000
Last Fail Image    : 0x00000000
State              : 0x00000000
Version           : 0x00000202
Error location     : 0x00000000
Error details      : 0x00000000
Retry counter      : 0x00000000
```

The decision firmware loads the highest priority image, and it does not look at the SPTs. The `rsu_init` function is called when the first RSU U-Boot command is executed and it detects that both SPTs are corrupted.

7. Try to run an RSU command which requires a valid SPT - it fails:

```
SOCFPGA # rsu slot_count
corrupted SPT --run rsu restore_spt <address> first
```

8. Restore the SPT from the backup copy that we have created:

```
SOCFPGA # fatload mmc 0:1 ${loadaddr} spt-backup.bin
4100 bytes read in 2 ms (2 MiB/s)
SOCFPGA # rsu restore_spt ${loadaddr}
```

9. Power cycle the board, the highest priority image loads, and all functionality is available. This power cycle is needed to cause the `rsu_init` function to be called in U-Boot, as it is only called once when the first RSU command is called.



## 7.5. Exercising the RSU Client

### 7.5.1. Using the RSU Client to Perform Basic Operations

This section demonstrates how to use the RSU client to perform the following basic operations:

- Querying the RSU status.
- Querying the number of slots and the information about them.
- Adding a new application image.
- Verifying that the application image was written correctly.
- Requesting a specific application image to be loaded.

**Note:** This section assumes that the flash contains the initial RSU image. If that is not true, you need to re-flash the initial image, as shown in the *Flashing the Initial RSU Image to QSPI* section.

1. Power cycle the board and let Linux boot.
2. Log in using 'root' as user name, no password is required.
3. Run the `rsu_client` without parameters, to display its help message:

```
root@linux:~# ./rsu_client
--- RSU app usage ---
-c|--count                get the number of slots
-l|--list slot_num        list the attribute info from the selected
slot
-z|--size slot_num        get the slot size in bytes
-p|--priority slot_num    get the priority of the selected slot
-E|--enable slot_num      set the selected slot as the highest
priority
-D|--disable slot_num     disable selected slot but to not erase it
-r|--request slot_num     request the selected slot to be loaded
after the next reboot
-R|--request-factory      request the factory image to be loaded
after the next reboot
-e|--erase slot_num       erase app image from the selected slot
-a|--add file_name -s|--slot slot_num add a new app image to the selected
slot
-u|--add-factory-update file_name -s|--slot slot_num add a new factory
update image to the selected slot
-A|--add-raw file_name -s|--slot slot_num add a new raw image to the
selected slot
-v|--verify file_name -s|--slot slot_num verify app image on the selected
slot
-V|--verify-raw file_name -s|--slot slot_num verify raw image on the
selected slot
-f|--copy file_name -s|--slot slot_num read the data in a selected slot
then write to a file
-g|--log                  print the status log
-n|--notify value         report software state
-C|--clear-error-status   clear errors from the log
-Z|--reset-retry-counter  reset current retry counter
-m|--display-dcmf-version print DCMF version
-y|--display-dcmf-status  print DCMF status
-x|--display-max-retry    print max_retry parameter
-t|--create-slot slot_name -S|--address slot_address -L|--length slot_size
create a new slot using unallocated space
-d|--delete-slot slot_num delete selected slot, freeing up
allocated space
-W|--restore-spt file_name restore spt from a file
-X|--save-spt file_name   save spt to a file
```

```
-b|--create-empty-cpb          create a empty cpb
-B|--restore-cpb file_name     restore cpb from a file
-P|--save-cpb file_name        save cpb to a file
-k|--check-running-factory     check if currently running the factory
image
-h|--help                      show usage message
```

4. Exercise the `rsu_client` command that displays the current status, it shows the application image from slot 0 (partition P1) is loaded with no errors:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

5. Run the RSU client commands that display information about the slots:

```
root@linux:~# ./rsu_client --count
number of slots is 3
Operation completed
root@linux:~# ./rsu_client --list 0
NAME: P1
OFFSET: 0x0000000001000000
SIZE: 0x01000000
PRIORITY: 1
Operation completed
root@linux:~# ./rsu_client --list 1
NAME: P2
OFFSET: 0x0000000002000000
SIZE: 0x01000000
PRIORITY: [disabled]
Operation completed
root@linux:~# ./rsu_client --list 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: [disabled]
Operation completed
```

6. Display the decision firmware version information:

```
root@linux:~# ./rsu_client --display-dcmf-version
DCMF0 version = 21.2.0
DCMF1 version = 21.2.0
DCMF2 version = 21.2.0
DCMF3 version = 21.2.0
Operation completed
```

7. Erase slot 1 and add the application2.rpd application image to slot 1 (partition P2):

```
root@linux:~# ./rsu_client --erase 1
Operation completed
root@linux:~# ./rsu_client --add application2.rpd --slot 1
Operation completed
```

8. Verify that the application image was written correctly to flash:

```
root@linux:~# ./rsu_client --verify application2.rpd --slot 1
Operation completed
```

9. List again the slots, it shows the most recently written partition P2 image having the highest priority (lowest priority number that is):

```
root@linux:~# ./rsu_client --list 0
NAME: P1
OFFSET: 0x0000000001000000
SIZE: 0x01000000
PRIORITY: 2
Operation completed
root@linux:~# ./rsu_client --list 1
NAME: P2
OFFSET: 0x0000000002000000
SIZE: 0x01000000
PRIORITY: 1
Operation completed
root@linux:~# ./rsu_client --list 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: [disabled]
Operation completed
```

10. Power cycle the board, boot Linux, and display the status – it shows the image from partition P2 running:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000002000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

11. Instruct the RSU client to request slot 0 (partition P1) from SDM on next reboot command:

```
root@linux:~# ./rsu_client --request 0
Operation completed
```

12. Restart Linux by running the `reboot` command:

```
root@linux:~# reboot
```

13. Log into Linux and display the RSU status:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

The status shows that the image from partition P1 was loaded, as requested.

### 7.5.2. Watchdog Timeout and `max retry` Operation

This section uses the RSU client to demonstrate the following:

- RSU handling of watchdog timeouts.
- `max_retry` feature, which allows each image up to three times to load.
- RSU notify, which allows the HPS software state to be reported and retrieved after a watchdog timeout.
- Clearing the RSU status error fields.
- Resetting the current `retry` counter value.

**Note:** The commands listed in this section rely on the commands from the *Basic Operation* section running first, specifically adding image P2 to the flash.

1. Power cycle the board, boot Linux, and display the status – it shows the P2 image running, as it is the highest priority:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000002000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

What do the version fields mean:

- Retry counter is 0x00000000—first attempt to load this image.
  - State is 0x00000000—No errors to report
2. Query and display the `max_retry` value:

```
root@linux:~# ./rsu_client --display-max-retry
max_retry = 3
Operation completed
```

**Note:** The `max_retry` option must be queried from U-Boot first, in order for it to be available on Linux.

3. Enable the watchdog but do not service it, as this produces a timeout, and restarts Linux:

```
root@linux:~# echo "something" > /dev/watchdog
[ 603.649746] watchdog: watchdog0: watchdog did not stop!
```

4. Wait for Linux to restart after the watchdog timeout, then display the log:

```
root@linux:~# ./rsu_client --log
VERSION: 0x0ACF0202
STATE: 0xF0060002
CURRENT IMAGE: 0x0000000002000000
FAIL IMAGE: 0x0000000002000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000001
Operation completed
```

The same P2 image is loaded, but the `retry` counter value is now one, which means this is the second retry for this image to be loaded. The `version` field shows the last failure was by an application image (0xACF). The `state` field shows the last error was a watchdog timeout (0xF006) and that the latest notify value from HPS software was from U-Boot loading Linux (0x0002).

5. Clear the error status so we can see the next errors. Query the status to show the errors were cleared:

```
root@linux:~# ./rsu_client --clear-error-status
Operation completed
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000002000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000001
Operation completed
```

6. Cause another watchdog timeout and wait for Linux to restart. After the restart, query the RSU log and observe that the `retry` counter is now two:

```
root@linux:~# ./rsu_client --log
VERSION: 0x0ACF0202
STATE: 0xF0060002
CURRENT IMAGE: 0x0000000002000000
FAIL IMAGE: 0x0000000002000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000002
Operation completed
```

7. Clear the error status so we can see the next errors.
8. Notify the SDM of the HPS execution stage as a 16bit number:

```
root@linux:~# ./rsu_client --notify 0x1234
Operation completed
```

9. Cause another watchdog timeout and watch for Linux to restart. After the restart, query the RSU log:

```
root@linux:~# ./rsu_client --log
VERSION: 0x0ACF0202
STATE: 0xF0061234
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000002000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

The SDM loaded the next application image in the CPB (P1), and it reports that the image P2 failed. The state indicates that a watchdog timeout occurred (upper 16 bits = 0xF006) and that the notify value reported by HPS software was 0x1234. The upper 16 bits of the version are set to 0x0ACF which means the previous error was reported by the application image firmware. For more information, refer to [RSU Status and Error Codes](#) on page 114.

10. Clear the error status so we can see the next errors.
11. Cause a watchdog timeout and display the status - it shows a `retry` counter value of one:

```
root@linux:~# ./rsu_client --log
VERSION: 0x0ACF0202
STATE: 0xF0060002
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000001000000
ERROR LOC: 0x00000000
```

```
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000001
Operation completed
```

12. Use the RSU client to reset the current retry counter value to zero, and query the status again to confirm it:

```
root@linux:~# ./rsu_client --reset-retry-counter
Operation completed
root@linux:~# ./rsu_client --log
VERSION: 0x0ACF0202
STATE: 0xF0060002
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000001000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

### Related Information

Using the RSU Client to Perform Basic Operations on page 81

## 7.5.3. Updating the Factory Image Using the RSU Client

This section demonstrates how to use the RSU client to update the factory image.

### Note:

The commands listed in this section rely on the commands from the *Basic Operation* section running first, specifically adding an application image to the P2 flash partition.

1. Power cycle the board, boot Linux and display the status – it shows the P2 image running, as it is the highest priority:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000002000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

2. Confirm that slot 2 (partition P3) is not used, erase slot 2, write the factory update image to it, and verify it was written correctly:

```
root@linux:~# ./rsu_client --list 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: [disabled]
Operation completed
root@linux:~# ./rsu_client --erase 2
Operation completed
root@linux:~# ./rsu_client --add-factory-update factory_update.rpd --slot 2
Operation completed
root@linux:~# ./rsu_client --verify factory_update.rpd --slot 2
Operation completed
```

3. Confirm that slot 2 is now the highest priority in the CPB:

```
root@linux:~# ./rsu_client --list 2
NAME: P3
OFFSET: 0x0000000003000000
```

```

        SIZE: 0x01000000
        PRIORITY: 1
    Operation completed

```

4. Instruct the RSU client to request slot 2 (partition P3) to be loaded on next reboot command:

```

root@linux:~# ./rsu_client --request 2
Operation completed

```

5. Restart Linux by running the `reboot` command:

```

root@linux:~# reboot

```

6. Linux shuts down, then the factory update image updates the decision firmware, decision firmware data and factory image in flash. This takes a few seconds, and there is no activity on the serial console during this time. Then it removes itself from the CPB and loads the now highest priority image in the CPB. Confirm that P2 is now loaded and P3 is disabled:

```

root@linux:~# ./rsu_client --log
    VERSION: 0x00000202
    STATE: 0x00000000
    CURRENT IMAGE: 0x0000000002000000
    FAIL IMAGE: 0x0000000000000000
    ERROR LOC: 0x00000000
    ERROR DETAILS: 0x00000000
    RETRY COUNTER: 0x00000000
    Operation completed
root@linux:~# ./rsu_client --list 2
    NAME: P3
    OFFSET: 0x0000000003000000
    SIZE: 0x01000000
    PRIORITY: [disabled]
    Operation completed

```

### Related Information

[Using the RSU Client to Perform Basic Operations](#) on page 81

## 7.5.4. Fallback on Flash Corruption of Application Images

This section uses the RSU client to demonstrate falling back in case of configuration errors caused by flash corruption of application images.

**Note:** The commands listed in this section rely on the commands from the *Basic Operation* section running first, specifically adding image P2 to the flash.

1. Power cycle the board, boot Linux and display the status – it shows the P2 image running, as it is the highest priority:

```

root@linux:~# ./rsu_client --log
    VERSION: 0x00000202
    STATE: 0x00000000
    CURRENT IMAGE: 0x0000000002000000
    FAIL IMAGE: 0x0000000000000000
    ERROR LOC: 0x00000000
    ERROR DETAILS: 0x00000000
    RETRY COUNTER: 0x00000000
    Operation completed

```

What do the version fields mean:

- Retry counter is 0x00000000—first attempt to load this image.
  - State is 0x00000000—No errors to report
2. Erase slot 1, which also takes it out of CPB:

```
root@linux:~# ./rsu_client --erase 1
Operation completed
```

3. Create a file with random data, and write it to the P2 slot:

```
root@linux:~# dd if=/dev/urandom of=corrupt.rpd bs=1M count=1
1+0 records in
1+0 records out
root@linux:~# ./rsu_client --add-raw corrupt.rpd --slot 1
Operation completed
```

4. Enable the P2 slot, which puts it as the highest priority in the CPB:

```
root@linux:~# ./rsu_client --enable 1
Operation completed
```

5. Confirm that P2 is now the highest priority in the CPB:

```
root@linux:~# ./rsu_client --list 1
NAME: P2
OFFSET: 0x0000000002000000
SIZE: 0x01000000
PRIORITY: 1
Operation completed
```

6. Power cycle the board, boot Linux and query the RSU log:

```
root@linux:~# ./rsu_client --log
VERSION: 0x0DCF0202
STATE: 0xF004D003
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000002000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

The current image is P1, and the P2 shows as failed. The top 4 bits of the version field are set to 0x0 which means the currently used DCMF index is zero. The next 12 bits of the version field are set as 0xDC F which means the error was caused by the decision firmware, because it was not able to load the image. The top 16 bits of the state field are set to 0xF004, which indicates an internal error. For more information, refer to [RSU Status and Error Codes on page 114](#).

7. Clear the error status and display the log again to confirm it was cleared:

```
root@linux:~# ./rsu_client --clear-error-status
Operation completed
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```



### Related Information

Using the RSU Client to Perform Basic Operations on page 81

## 7.5.5. Additional Flash Corruption Detection and Recovery

This section presents examples of detecting and recovering corrupted decision firmware, decision firmware data, configuration pointer blocks and sub-partition tables with the aid of the RSU client.

### 7.5.5.1. Corrupted Decision Firmware

This example uses the RSU client to demonstrate detecting that some decision firmware copies are corrupted, and recovering them by running a decision firmware update image. The task uses U-Boot to corrupt flash, as it is the only component which has direct access to decision firmware.

**Note:** The commands listed in this example assume the initial flash image (JIC) was written to flash, with no other changes.

**Note:** The `rsu_display_dcmf_version` command is called automatically by current U-Boot before booting Linux. This is required to have the status available in Linux.

1. Power cycle the board, boot up to Linux.
2. Query RSU status:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

The top four bits of the `state` field are `0x0` which means the currently used decision firmware index is zero. There are no errors.

3. Display decision firmware status, it shows no corruptions:

```
root@linux:~# ./rsu_client --display-dcmf-status
DCMF0: OK
DCMF1: OK
DCMF2: OK
DCMF3: OK
Operation completed
```

4. Power cycle board, stop to U-Boot, corrupt decision firmware copies 0 and 2:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total 256 MiB
SOCFPGA # sf erase 0 0x1000
SF: 4096 bytes @ 0x0 Erased: OK
SOCFPGA # sf erase 0x80000 0x1000
SF: 4096 bytes @ 0x80000 Erased: OK
```

5. Power cycle the board, boot to Linux, query the RSU status and decision firmware status:

```
root@linux:~# ./rsu_client --log
VERSION: 0x10000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
root@linux:~# ./rsu_client --display-dcmf-status
DCMF0: Corrupted
DCMF1: OK
DCMF2: Corrupted
DCMF3: OK
Operation completed
```

The currently used copy of the decision firmware is 1, as indicated by top four bits of the version field. There are no errors reported by firmware. The decision firmware copies 0 and 2 are detected as corrupted.

6. Erase an unused slot, add the decision firmware update image to the slot, verify it was written fine, and confirm it is now the highest priority running slot:

```
root@linux:~# ./rsu_client --erase 2
Operation completed
root@linux:~# ./rsu_client --add-factory-update
decision_firmware_update.rpd --slot 2
Operation completed
root@linux:~# ./rsu_client --verify decision_firmware_update.rpd --slot 2
Operation completed
root@linux:~# ./rsu_client --list 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: 1
Operation completed
```

7. Pass control to the decision firmware update image:

```
root@linux:~# ./rsu_client --request 2
Operation completed
root@linux:~# reboot
```

8. Linux shuts down, then the decision firmware update image writes new decision firmware copies and new decision firmware data to flash, remove itself from CPB, then pass control to the highest priority image. Let it boot to Linux.

9. In Linux, query the RSU status and display the DCMF status:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
root@linux:~# ./rsu_client --display-dcmf-status
DCMF0: OK
DCMF1: OK
DCMF2: OK
DCMF3: OK
Operation completed
```

All decision firmware copies are reported as fine, and copy 0 is the one currently used, as expected.

### 7.5.5.2. Corrupted Decision Firmware Data

This example uses the RSU client to demonstrate detecting that the decision firmware data is corrupted, and recovering it by running a decision firmware update image.

**Note:** The commands listed in this example assume the initial flash image (JIC) was written to flash, with no other changes.

1. Power cycle the board, boot to Linux, and query RSU status:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

There are no errors.

2. Reboot power cycle the board, stop at U-Boot prompt, and corrupt decision firmware data:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x100000 0x1000
SF: 4096 bytes @ 0x100000 Erased: OK
```

3. Power cycle the board, boot to Linux and query the status:

```
root@linux:~# ./rsu_client --log
VERSION: 0x0DCF0202
STATE: 0xF004D00F
CURRENT IMAGE: 0x0000000000110000
FAIL IMAGE: 0x0000000000100000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

The State contains the special error code 0xF004D00F indicating that the decision firmware data was corrupted. The current image is listed as the factory image. The error source is listed as 0xDCF, meaning the error was reported by the decision firmware. The Last Fail Image is set to 0x00100000, which is a special value indicating an error was reported by the decision firmware.

4. Erase an unused slot, add the decision firmware update image to the slot, verify it was written fine, and confirm it is now the highest priority running slot:

```
root@linux:~# ./rsu_client --erase 2
Operation completed
root@linux:~# ./rsu_client --add-factory-update
decision_firmware_update.rpd --slot 2
Operation completed
root@linux:~# ./rsu_client --verify decision_firmware_update.rpd --slot 2
Operation completed
root@linux:~# ./rsu_client --list 2
NAME: p3
```

```

OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: 1
Operation completed

```

5. Pass control to the decision firmware update image:

```

root@linux:~# ./rsu_client --request 2
Operation completed
root@linux:~# reboot

```

6. Linux shuts down, then the decision firmware update image writes new decision firmware copies and new decision firmware data to flash, remove itself from CPB, then pass control to the highest priority image.

7. In Linux, query the RSU status:

```

root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed

```

The highest priority image is loaded, and there are no errors. The decision firmware data was restored.

### 7.5.5.3. Corrupted Configuration Pointer Block

This section uses the RSU client to demonstrate how configuration pointer block corruptions can be detected and recovered. It also uses U-Boot to corrupt the CPB, as it's more conveniently done there.

**Note:** The commands listed in this example assume the initial flash image (JIC) was written to flash, with no other changes.

1. Power cycle the board, boot to Linux, and query RSU status:

```

root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed

```

There are no errors.

2. Reboot or power cycle the board, stop at U-Boot prompt, and corrupt CPB0 by erasing the corresponding flash area:

```

SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x0920000 0x1000
SF: 4096 bytes @ 0x920000 Erased: OK

```

3. Power cycle the board. U-Boot runs, and before starting Linux, it executes a few RSU commands, which causes `rsu_init` to be called, and that restores CPB0 from the CPB1 copy:

```
FW detects corrupted CPB0 but CPB1 is fine
Restoring CPB0
```

4. Once Linux has booted, query the RSU status:

```
root@linux:~# ./rsu_client --log
librsu: load_cpb(): FW detects corrupted CPB0, fine CPB1
[LOW]
librsu: load_cpb(): warning: Restoring CPB0 [LOW]
VERSION: 0x0DCF0202
STATE: 0xF004D010
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x000000000001000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

The State field has the special error code 0xF004D010 which indicates that CPB0 was corrupted. The Last Fail Image has the special value 0x00100000 which is used in this case. The Version field indicates that the error was reported by the decision firmware (0xDCF). Although the U-Boot already recovered CPB0 from CPB1, the firmware still reports CPB0 as corrupted, and LibRSU recovers it again.

5. Clear the error status, so that the firmware stops reporting CPB0 corrupted, and query RSU status again to verify there are no errors:

```
root@linux:~# ./rsu_client --clear-error-status
librsu: load_cpb(): FW detects corrupted CPB0, fine CPB1
[LOW]
librsu: load_cpb(): warning: Restoring CPB0 [LOW]
Operation completed
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

6. Power cycle the board, boot to Linux and query RSU status. Verify there are no errors:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

7. Save the CPB contents to a file, to be used later for recovery. Also call `sync` command to make sure the file is saved to storage:

```
root@linux:~# ./rsu_client --save-cpb cpb-backup.bin
Operation completed
root@linux:~# sync
```

8. Reboot or power cycle, stop in U-Boot and corrupt both CPBs by erasing the flash at their location:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x0920000 0x1000
SF: 4096 bytes @ 0x920000 Erased: OK
SOCFPGA # sf erase 0x0928000 0x1000
SF: 4096 bytes @ 0x928000 Erased: OK
```

9. Power cycle the board, boot to Linux and query RSU status:

```
root@linux:~# ./rsu_client --log
librsu: load_cpb(): FW detects both CPBs corrupted
[LOW]
    VERSION: 0x0DCF0202
    STATE: 0xF004D011
CURRENT IMAGE: 0x0000000000110000
  FAIL IMAGE: 0x0000000000110000
    ERROR LOC: 0x00000000
  ERROR DETAILS: 0x00000000
  RETRY COUNTER: 0x00000000
Operation completed
```

The Current Image is reported as being the factory image, as expected. The State field has the special error code 0xF004D011 which indicates that both CPBs were corrupted. The Last Fail Image has the special value 0x00100000 which is used in this case. The Version field indicates that the error was reported by the decision firmware (0xDCf). The LibRSU reported that both CPBs are corrupted.

10. Try to run a command which requires a valid CPB - verify it is rejected:

```
root@linux:~# ./rsu_client --list 1
librsu: load_cpb(): FW detects both CPBs corrupted
[LOW]
librsu: rsu_cpb_corrupted_info(): corrupted CPB -- [LOW]
librsu: rsu_cpb_corrupted_info(): run rsu_client create-empty-cpb or [LOW]
librsu: rsu_cpb_corrupted_info(): rsu_client restore_cpb <file_name> first
[LOW]
ERROR: Failed to get slot attributes
```

11. Clear errors, so that the CPBs are not reported as corrupted by the firmware anymore:

```
root@linux:~# ./rsu_client --clear-error-status
librsu: load_cpb(): FW detects both CPBs corrupted
[LOW]
Operation completed
root@linux:~# ./rsu_client --log
librsu: load_cpb(): Bad CPB1 is bad [MED]
librsu: load_cpb(): Bad CPB0 is bad [MED]
librsu: load_cpb(): error: found both corrupted CPBs [LOW]
    VERSION: 0x00000202
    STATE: 0x00000000
CURRENT IMAGE: 0x0000000000110000
  FAIL IMAGE: 0x0000000000000000
    ERROR LOC: 0x00000000
  ERROR DETAILS: 0x00000000
  RETRY COUNTER: 0x00000000
Operation completed
```

12. Restore the saved CPB from the backup file that we created:

```
root@linux:~# ./rsu_client --restore-cpb cpb-backup.bin
librsu: load_cpb(): Bad CPB1 is bad [MED]
librsu: load_cpb(): Bad CPB0 is bad [MED]
librsu: load_cpb(): error: found both corrupted CPBs [LOW]
Operation completed
```

13. Query again RSU status, there should be no errors signalled now:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000000110000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

14. Try to run a command which requires a valid CPB - verify it is not rejected:

```
root@linux:~# ./rsu_client --list 1
NAME: P2
OFFSET: 0x0000000002000000
SIZE: 0x01000000
PRIORITY: [disabled]
Operation completed
```

#### 7.5.5.4. Corrupted Sub-Partition Table

This example uses the RSU client to demonstrate how sub-partition table corruptions can be detected and recovered. It also uses U-Boot to corrupt the SPT, as it's more conveniently done there.

**Note:** The commands listed in this example assume the initial flash image (JIC) was written to flash, with no other changes.

1. Power cycle the board, boot to Linux, and query RSU status:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

There are no errors.

2. Reboot or power cycle the board, stop at U-Boot prompt, and corrupt SPT0 by erasing the corresponding flash area:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x0910000 0x1000
SF: 4096 bytes @ 0x910000 Erased: OK
```

3. Power cycle the board. U-Boot runs, and before booting Linux it runs a few RSU commands. Running the first one causes `rsu_init` to be called, which detects corrupted SPT0 and recovers it from SPT1:

```
Bad SPT0 magic number 0xFFFFFFFF
Restoring SPT0
```

4. boot to Linux and query RSU status:

```
root@linux:~# ./rsu_client --log
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
Operation completed
```

There are no errors

5. Save the currently used SPT to a file for backup purposes. Also call `sync` command to make sure the file is committed to flash:

```
root@linux:~# ./rsu_client --save-spt spt-backup.bin
Operation completed
root@linux:~# sync
```

6. Reboot or power cycle, stop to U-Boot and corrupt both SPTs by erasing the flash at their locations:

```
SOCFPGA # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA # sf erase 0x910000 0x1000
SF: 4096 bytes @ 0x910000 Erased: OK
SOCFPGA # sf erase 0x918000 0x1000
SF: 4096 bytes @ 0x918000 Erased: OK
```

7. Power cycle the board, boot to Linux, and query RSU status:

```
root@linux:~# ./rsu_client --log
librsu: load_spt(): Bad SPT1 magic number 0xFFFFFFFF [MED]
librsu: load_spt(): Bad SPT0 magic number 0xFFFFFFFF [MED]
librsu: load_spt(): error: No valid SPT0 or SPT1 found [LOW]
VERSION: 0x00000202
STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
FAIL IMAGE: 0x0000000000000000
ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
RETRY COUNTER: 0x00000000
```

The decision firmware loads the highest priority image, and it does not look at the SPTs. The LibRSU detects that both SPTs are corrupted.

8. Try to run an RSU client command which requires a valid SPT - it fails:

```
root@linux:~# ./rsu_client --count
librsu: load_spt(): Bad SPT1 magic number 0xFFFFFFFF [MED]
librsu: load_spt(): Bad SPT0 magic number 0xFFFFFFFF [MED]
librsu: load_spt(): error: No valid SPT0 or SPT1 found [LOW]
librsu: rsu_spt_corrupted_info(): corrupted SPT -- [LOW]
librsu: rsu_spt_corrupted_info(): run rsu_client restore-spt <file_name>
first
[LOW]
ERROR: Failed to get number of slots
```



9. Restore the SPT from the backup copy that we have created:

```
root@linux:~# ./rsu_client --restore-spt spt-backup.bin
librsu: load_spt(): Bad SPT1 magic number 0xFFFFFFFF [MED]
librsu: load_spt(): Bad SPT0 magic number 0xFFFFFFFF [MED]
librsu: load_spt(): error: No valid SPT0 or SPT1 found [LOW]
Operation completed
```

10. Try again running RSU client commands which require a valid SPT - the commands work fine:

```
root@linux:~# ./rsu_client --count
number of slots is 3
Operation completed
```

## 8. Version Compatibility Considerations

This section lists the component versions that are presented in this document. It also includes considerations about compatibility between different component versions.

### 8.1. Component Versions

This document applies to and was tested with the Intel Quartus Prime Pro Edition software version 21.2 and the HPS software component versions shown in the table below:

Repository	Branch	Commit ID
<a href="https://github.com/altera-opensource/u-boot-socfpga">https://github.com/altera-opensource/u-boot-socfpga</a>	origin/socfpga_v2021.01	e59d8e9eaa14d2164fe4dac4ab0fa7f5dca3a0fe
<a href="https://github.com/altera-opensource/arm-trusted-firmware">https://github.com/altera-opensource/arm-trusted-firmware</a>	origin/socfpga_v2.4.1	9250b2f0a84d01cc33f2e23f7b69a697928e46b0
<a href="https://github.com/altera-opensource/linux-socfpga">https://github.com/altera-opensource/linux-socfpga</a>	origin/socfpga-5.4.114-lts	0b009da1a50f1521a7454c94873dad51c6cc74d1
<a href="https://github.com/altera-opensource/intel-rsu">https://github.com/altera-opensource/intel-rsu</a>	origin/master	e90cd1e36b3f89df27641f3f932afd56013406d8

**Note:** Intel policy specifies that only the current and immediately previous U-Boot and Linux branches are kept on GitHub. As a new branch is added, the oldest branch is removed. You must keep a local copy of the sources used to build your binaries in case you need to reproduce the build or make changes in the future.

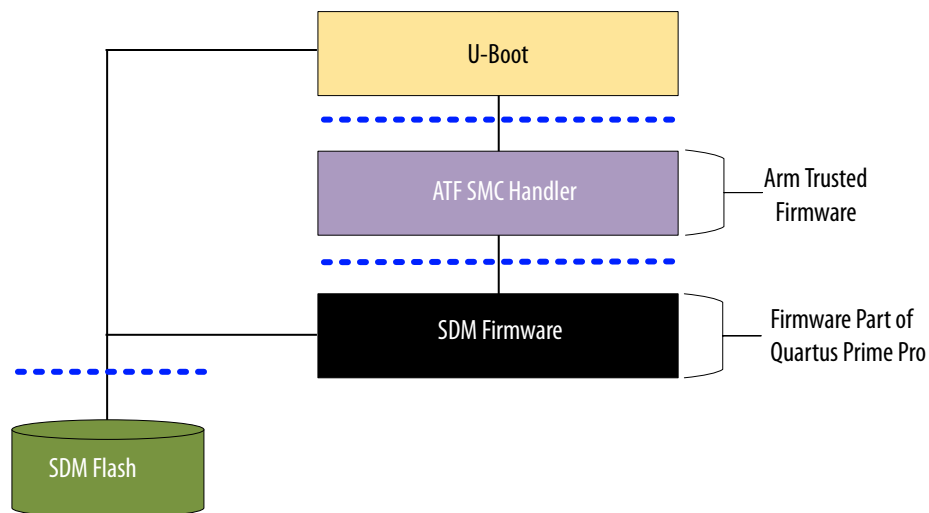
#### Related Information

[Version Compatibility Considerations](#) on page 98

### 8.2. Component Interfaces

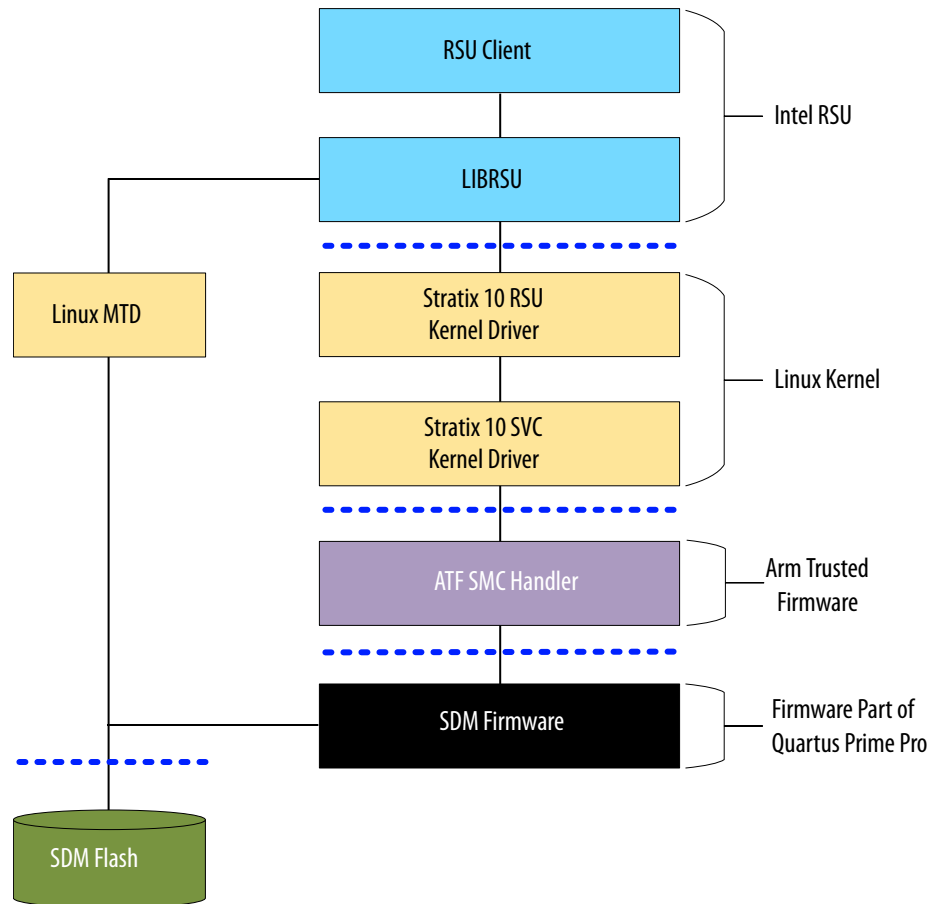
The SDM provides commands which are accessed by ATF (Arm Trusted Firmware). ATF exports the functionality through an SMC (Secure Monitor Call) handler for the rest of HPS software to use.

You can access the RSU functionality through U-Boot. The SMC handler is used by U-Boot to implement the RSU functionality.

**Figure 9. U-Boot RSU Components and Interfaces**

You can also access the RSU services through LIBRSU. In this case the SMC handler is used by the Linux kernel drivers which then expose the functionality to LIBRSU which then is used by Linux applications in general, and RSU Client application in particular.

Figure 10. Linux RSU Components and Interfaces



The horizontal dotted lines in the above diagrams show interfaces between components. The SDM firmware, U-Boot and LIBRSU access the SDM Flash, so an interface is shown there too.

**Important:** Newer versions of the Intel Quartus Prime software typically include new or updated SDM features implemented in firmware. When generating your configuration bitstream, Intel recommends using the latest version of the Intel Quartus Prime Pro Edition software which includes the latest firmware. You do not need to recompile your .sof file to use the firmware from a newer version of the Intel Quartus Prime Pro Edition software. You can simply regenerate your configuration bitstream with the new version of the Programming File Generator.

### 8.3. Component Version Compatibility

The component versions used for creating this document, and listed in [Component Versions](#) on page 98 are tested to be compatible.

**Important:** In order to get the functionality described in this document, you must use the component versions that are presented in [Component Versions](#) on page 98. Using different versions of components than the ones described in this document may result in some functionality not being available or not behaving consistently. It is your responsibility to validate that the combination of components you plan to use meets your requirements.

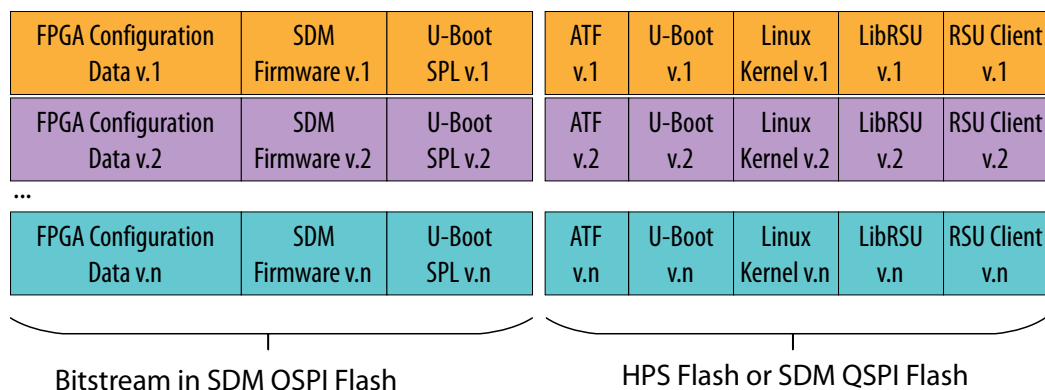
Starting with U-Boot branch `socfpga_v2020.04`, each branch of U-Boot (and corresponding ATF) is tested not only with the current Intel Quartus Prime Pro Edition software version, but also with the two previous Intel Quartus Prime Pro Edition software versions. Refer to the U-Boot file `doc/README.socfpga` for the lists of Intel Quartus Prime Pro Edition software versions that were tested with that U-Boot version.

## 8.4. Using Multiple Intel Quartus Prime Software Versions for Bitstreams

In order to get consistent functionality, you must use component versions which are tested to work together, as described in Component Version Compatibility.

If different bitstreams (application or factory) are generated with different Intel Quartus Prime software versions, this creates the need to use different HPS software component versions (such as U-Boot, ATF, Linux kernel and LIBRSU) for each bitstream as shown in the figure below:

**Figure 11. RSU System Using a Complete set of HPS Software for each Bitstream**



It is recommended to design your system with fully separated software stacks for each bitstream as shown above. This allows you to use any Intel Quartus Prime software version for each bitstream. It also allows you the most flexibility when choosing HPS software components.

## 8.5. Updating U-Boot to Support Different SSBL per Bitstream

By default U-Boot does not support loading a separate SSBL for each bitstream. Instead, the SSBL location is fixed as an address when loading SSBL from QSPI, and as a file name when loading SSBL from a FAT partition.

This section gives guidance on how you may update U-Boot to support having a different SSBL for each bitstream.

### 8.5.1. Using Multiple SSBLs with SD/MMC

This section presents the recommended approach to support multiple SSBLs when they are stored in the HPS SD/MMC.

No changes required in the Programming File Generator, at initial image creation time.

Changes required in FSBL:

- Query SDM and read flash to determine all the partition information, and the currently running bitstream location in flash.
- Look up the currently running bitstream location in the list of the SPT partitions to determine the partition containing the currently running bitstream.
- Instead of using a hardcoded file name for the SSBL, use a name derived from the name of the partition containing the currently running bitstream.

If the SSBL is configured with read-only environment, then no SSBL code changes are needed. If the SSBL is configured with a modifiable environment, then the following changes are recommended:

- Make sure there is enough space between the MBR and the first SD card partition to store the environment for all the bitstreams.
- Change the environment location from the hardcoded value to an address which is different for each bitstream.

The recommended application image update procedure also changes:

1. Use LIBRSU or U-Boot to erase the application image partition. This also disables it, removing it from the CPB.
2. Replace the corresponding U-Boot image file on the FAT partition with the new version, using a filename derived from the partition name.
3. In the case of a modifiable environment, erase the sector(s) associated with the application image partition.
4. Use LIBRSU or U-Boot to write the new application image. This also enables it, putting it as the highest priority in the CPB.

### 8.5.2. Using Multiple SSBLs with QSPI

This section presents the recommended approach to support multiple SSBLs when they are stored in the SDM QSPI flash.

Changes required in the Programming File Generator, at initial image creation time:

- Use the default naming scheme for the bitstream partitions:
  - FACTORY\_IMAGE for factory image
  - P1, P2, and so on for application images
- Make copies of the U-Boot image files (u-boot-socfpga/u-boot-dtb.img) to have the .bin extension, as that is what the Programming File Generator requires for binary files. For example name them u-boot-socfpga/u-boot-dtb.img.bin

**Attention:** Intel Quartus Prime Programming File Generator versions prior to 21.2 have an issue which causes block partitions (specified by entering the start and end address of the partition) containing raw data files (.hex, .bin, .puf or .wkey) to be resized to match the size of the raw data file. When using these versions, pad the raw file with 0xFFs to match the size of the partition.

**Attention:** Intel Quartus Prime Programming File Generator version 21.2 has an issue which causes block partitions (specified by entering the start and end address of the partition) containing raw data files (.hex, .bin, .puf or .wkey) to have incorrect content. Intel Quartus Prime Pro Edition software version 21.2 Patch 0.14 fixes this issue.

- In the Input Files tab, click the **Add Raw** button, then select the .bin file filter at the bottom and browse for the renamed U-Boot image file. Once added, click the file to select it, then click Properties on the right. Make sure to change the "Bit-swap" option from "off" to "on". This is telling the PFG that it is a regular binary file.
- Create new partitions to contain SSBLs, one for each bitstream:
  - SSBL partitions are large enough for the U-Boot image (1 MB suffices for most applications)
  - Name the partition with a name that is derived from the bitstream partition name, and is less than 15 characters long (limit for SPT partition name excluding null terminator). For example "FACTORY\_IM.SSBL", "P1.SSBL", etc.
  - For the initially loaded SSBL partitions, select the corresponding U-Boot binary image files as Input file so that they are loaded with the SSBLs.
- Generate the initial image.

Changes required in FSBL:

- Query SDM and read flash to determine all the partition information, and the currently running bitstream location in flash.
- Look up the currently running bitstream location in the list of the SPT partitions to determine the partition containing the currently running bitstream.
- Add the ".SSBL" to the name of the currently running partition and find the SSBL partition using that name. Treat "FACTORY\_IMAGE" differently as adding ".SSBL" to it can make it longer than the maximum allowable of 15 characters.
- Instead of loading the SSBL for a hardcoded address, load it from the partition found at the previous step.

If the SSBL is configured with read-only environment, then no changes are needed. If the SSBL is configured with a modifiable environment, then the following changes are recommended:

- Make the SSBL partitions larger than the maximum anticipated U-Boot image, to accommodate the environment. 1 MB can still suffice, as typical U-Boot image is smaller, and typical environment size is 4 KB.
- Change the hardcoded value for the SSBL environment address to query SDM for partitioning information and currently running image to determine the name of the current SSBL partition, and use its top portion as environment.

The recommended application image update procedure also changes:

- Use LIBRSU to erase the application image partition. This also disables it, removing it from the CPB.
- Use MTD to erase SSBL partition instead of LIBRSU because it does not support erasing raw partitions yet, just bitstreams.
- Use MTD to write the new contents of the SSBL partition instead of LIBRSU because it does not support writing raw partitions, just bitstreams.
- Use LIBRSU to write the new application image. This also enables it, putting it as highest priority in the CPB.

### 8.5.3. U-Boot Source Code Details

This section presents some details about the U-Boot source code, to aid in the implementation of support for multiple SSBLs.

The U-Boot code which queries SDM for the SPT partitioning information and currently running image and displays them when the `run list U-Boot` command is executed is located in the file `arch/arm/mach-socfpga/rsu_sl0.c`. This code can be used as a starting point for implementing the FSBL changes to allow it to load a different SSBL for each bitstream.

The file name for the SSBL binary to be loaded from SD card is defined in `include/configs/socfpga_soc64_common.h`:

```
#define CONFIG_SYS_MMCSL_FS_BOOT_PARTITION    1
#ifdef CONFIG_SPL_LOAD_FIT
#define CONFIG_SPL_FS_LOAD_PAYLOAD_NAME      "u-boot.itb"
#else
#define CONFIG_SPL_FS_LOAD_PAYLOAD_NAME      "u-boot.img"
#endif
```

The MMC device number where the U-Boot environment is stored is also defined in `include/configs/socfpga_soc64_common.h`

```
#define CONFIG_SYS_MMC_ENV_DEV                0    /* device 0 */
```

Location and size of U-Boot environment when it is stored in SD/MMC is defined in `configs/socfpga_agilex_atf_defconfig`:

```
CONFIG_ENV_SIZE=0x1000
CONFIG_ENV_OFFSET=0x200
```

Location of SSBL in QSPI flash is defined in `configs/socfpga_agilex_qspi_atf_defconfig`:

```
CONFIG_SYS_SPI_U_BOOT_OFFS=0x02000000
```



Location and size of U-Boot environment in QSPI flash is defined in `configs/socfpga_agilex_qspi_atf_defconfig`:

```
CONFIG_ENV_SIZE=0x1000  
CONFIG_ENV_SECT_SIZE=0x10000  
CONFIG_ENV_OFFSET=0x020C0000
```

## 9. Using RSU with HPS First

In the HPS First use case, the initial bitstream does not configure the FPGA fabric, and only the HPS FSBL is loaded and executed. Then at a later time the HPS configures the FPGA fabric – for example from U-Boot or Linux.

The potential advantages of using HPS First are:

- HPS can be booted faster.
- The bitstreams are much smaller, requiring smaller SDM QSPI size.
- The FPGA fabric configuration can reside on larger HPS flash or even be accessed remotely over the network.

The RSU fully supports both FPGA first, and HPS first use cases.

The following changes are required to the example presented in the *Remote System Update Example* section to use HPS first instead of FPGA first.

### Related Information

[Remote System Update Example](#) on page 49

### 9.1. Update Hardware Designs to use HPS First

When creating the hardware projects, enable HPS first for each project, either from the Intel Quartus Prime GUI, or as shown in bold below:

```
cd $TOP_FOLDER
# compile hardware designs: 0-factory, 1,2-applications, 3-factory update
rm -rf hw && mkdir hw && cd hw
wget https://github.com/altera-opensource/ghrd-socfpga/archive/\
ACDS-21.1pro-20.1std.zip
unzip ACDS-21.1pro-20.1std.zip
mv ghrd-socfpga-ACDS-21.1pro-20.1std/agilex_soc_devkit_ghrd .
rm -rf ghrd-socfpga-ACDS-21.1pro-20.1std ACDS-21.1pro-20.1std.zip
for version in {0..3}
do
rm -rf ghrd.$version
cp -r agilex_soc_devkit_ghrd ghrd.$version
cd ghrd.$version
make clean
make scrub_clean
rm -rf *.qpf *.qsf *.txt *.bin *.qsys ip/qsys_top/ ip/subsys_jtg_mst/ ip\
/subsys_periph/
sed -i 's/BOOTS_FIRST .*= .*/BOOTS_FIRST := hps/g' Makefile
sed -i 's/ENABLE_WATCHDOG_RST .*= .*/ENABLE_WATCHDOG_RST := 1/g' Makefile
sed -i 's/WATCHDOG_RST_ACTION .*= .*/WATCHDOG_RST_ACTION := remote_update/g'
Makefile
sed -i 's/0xACD5CAFE/0xABAB000'$version'/g' create_ghrd_qsys.tcl
export IP_ROOTDIR=~/.intelFPGA_pro/21.2/ip
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
make generate_from_tcl
echo "set_global_assignment -name RSU_MAX_RETRY_COUNT 3" \
>> ghrd_agfb014r24a3e3vr0.qsf
```

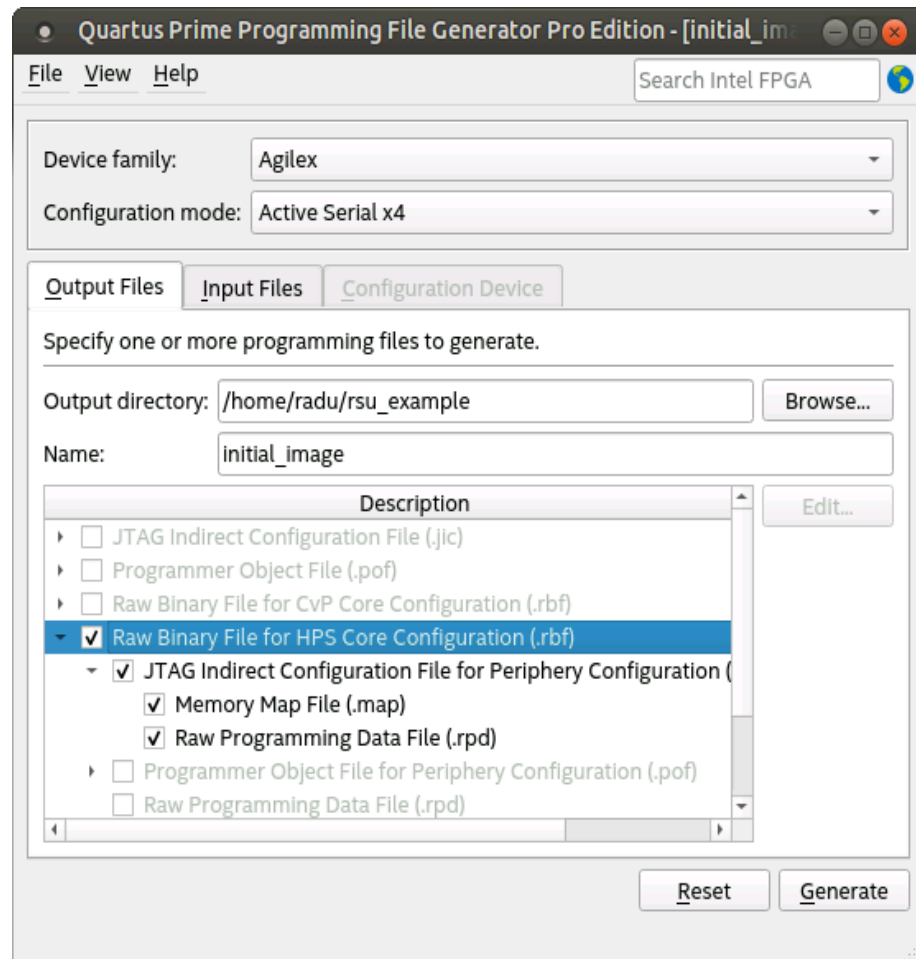
```
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \  
make sof  
cd ..  
done  
rm -rf agilex_soc_devkit_ghrd  
cd ..
```

## 9.2. Creating Initial Flash Image for HPS First

The following changes are required:

- Instead of **JTAG Indirect Configuration File (.jic)**, select the **Raw Binary File for HPS Core Configuration (.rbf)** then check the **JTAG Indirect Configuration File for Periphery Configuration (.jic)**, **Memory Map File (.map)**, and the **Raw Programming Data File (.rpd)** options.

Figure 12. Creating JIC File for HPS First



- You can make the factory image and application image partitions smaller, because they do not store the FPGA fabric configuration data.

After clicking the **Generate** button the following additional files are generated in the \$TOP\_FOLDER folder:

- `initial_image_FACTORY_IMAGE.core.rbf` - containing the FPGA fabric configuration data for the factory image.
- `initial_image_P1.core.rbf` - containing FPGA fabric configuration data for the application image P1.

### 9.3. Creating Application and Update Images for HPS First

The application image is created with the same command as for FPGA first, with an additional parameter specified, as shown below, in bold:

```
cd $TOP_FOLDER
mkdir -p images
rm -rf images/application2.rpd
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
quartus_pfg -c hw/ghrd.2/output_files/ghrd_agfb014r24a3e3vr0.sof \
images/application2.rpd \
-o hps_path=u-boot-socfpga/spl/u-boot-spl-dtb.hex \
-o mode=ASX4 -o start_address=0x000000 -o bitswap=ON \
-o hps=1
```

The following files are created:

- `images/application2.hps.rpd` - Application image
- `images/application2.core.rbf` - Corresponding fabric configuration file

The factory update image is created with the same command as for FPGA first, with the same additional parameter specified, as shown below, in bold:

```
cd $TOP_FOLDER
mkdir -p images
rm -f images/factory_update.rpd
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
quartus_pfg -c hw/ghrd.3/output_files/ghrd_agfb014r24a3e3vr0.sof \
images/factory_update.rpd \
-o hps_path=u-boot-socfpga/spl/u-boot-spl-dtb.hex \
-o mode=ASX4 -o start_address=0x000000 -o bitswap=ON \
-o rsu_upgrade=ON \
-o hps=1
```

The following files are created:

- `images/factory_update.hps.rpd` - Factory update image
- `images/factory_update.core.rbf` - Corresponding fabric configuration file

The decision firmware update image is created with the same command as for FPGA first, with the same additional parameter specified, as shown below, in bold:

```
cd $TOP_FOLDER
mkdir -p images
rm -f images/decision_firmware_update.rpd
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
quartus_pfg -c hw/ghrd.3/output_files/ghrd_agfb014r24a3e3vr0.sof \
images/decision_firmware_update.rpd \
-o hps_path=u-boot-socfpga/spl/u-boot-spl-dtb.hex \
-o mode=ASX4 -o start_address=0x000000 -o bitswap=ON \
-o rsu_upgrade=ON \
-o firmware_only=1 \
-o hps=1
```

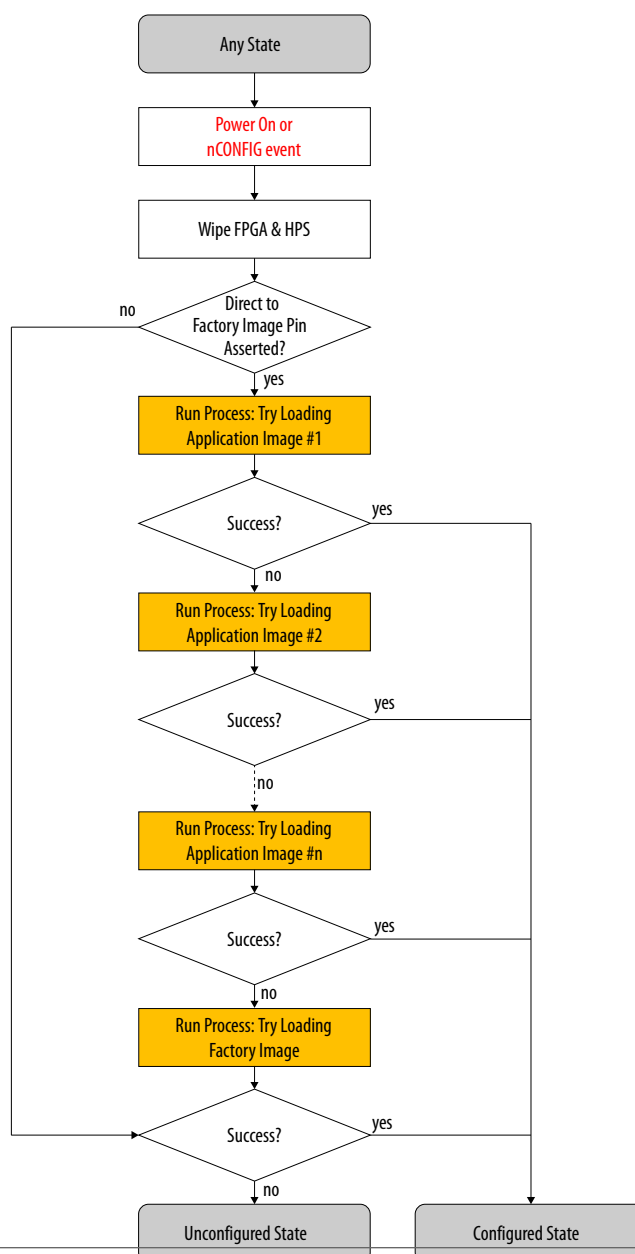
The following files are created:

- `images/decision_firmware_update.hps.rpd` - Decision firmware update image
- `images/decision_firmware_update.core.rbf` - Not used

## A. Configuration Flow Diagrams

### A.1. Load Image on Power Up or nConfig Flow

Figure 13. Load Image on Power Up or nConfig Flow



Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

\*Other names and brands may be claimed as the property of others.

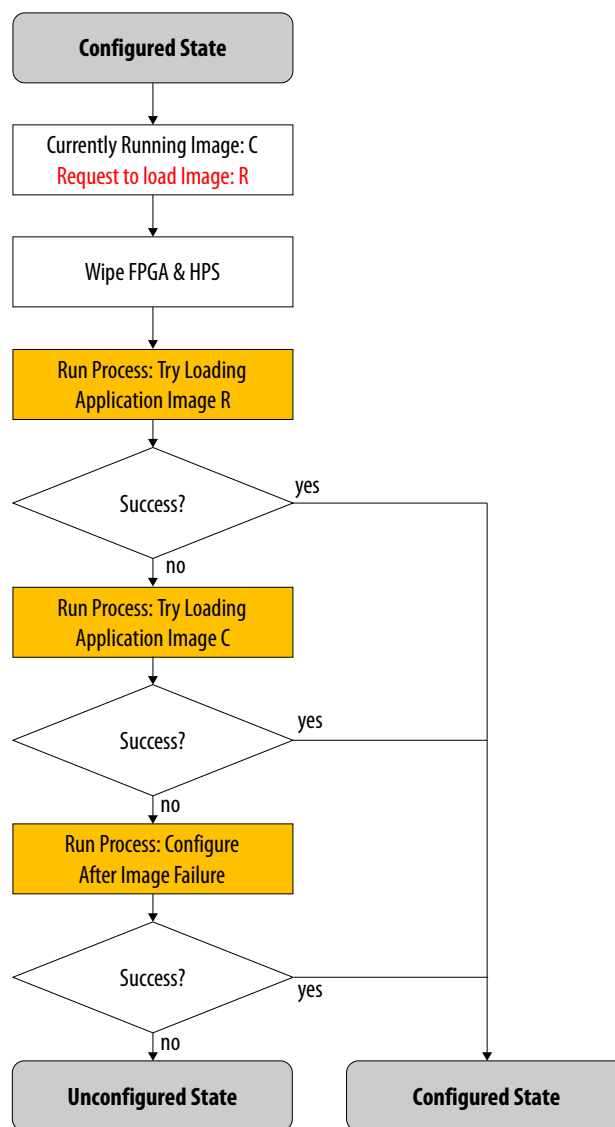
## A.2. Try Loading Image Process

**Figure 14. Try Loading Image Process**



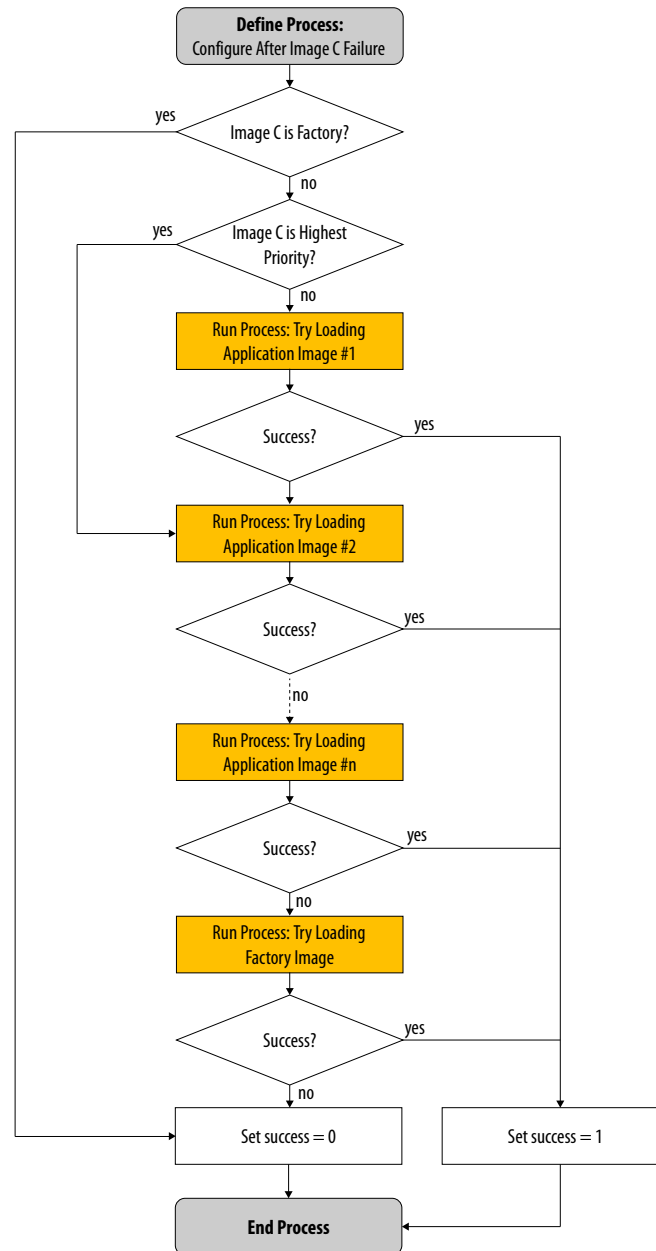
### A.3. Request Specific Image Flow

Figure 15. Request Specific Image Flow



## A.4. Configure after Image Failure Process

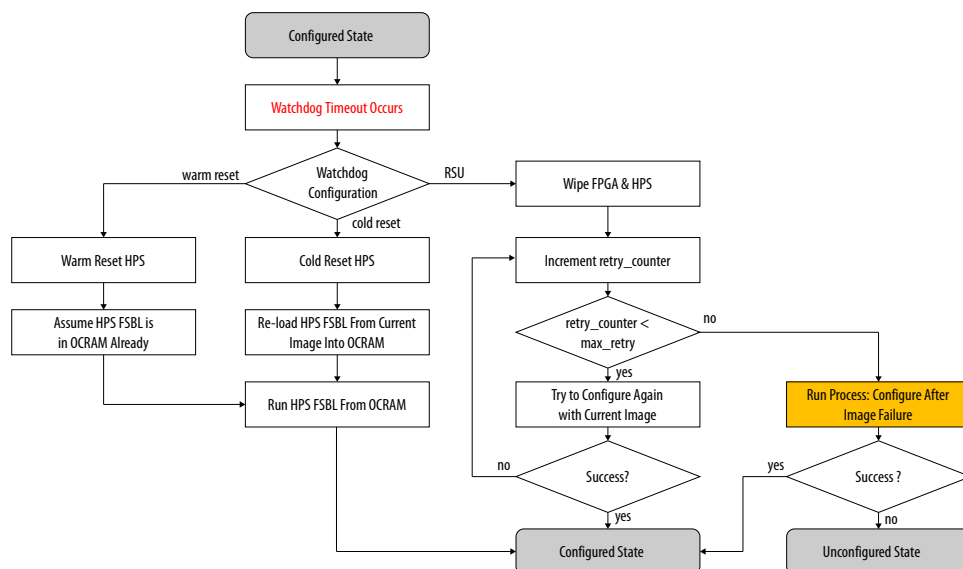
**Figure 16. Configure after Image Failure Process**





## A.5. Watchdog Timeout Flow

Figure 17. Watchdog Timeout Flow



## B. RSU Status and Error Codes

The RSU status can be checked from U-Boot and Linux and contains the following 32-bit fields:

**Table 11. RSU Status Fields**

Field	Description
current_image	Location of currently running image in flash.
failed_image	Address of failed image.
error_details	Opaque error code, with no meaning to users.
error_location	Location of error in the image that failed.
state	State of RSU system.
version	RSU interface version and error source.
retry_counter	Current value of the retry counter.

The failed\_image, error\_details, error\_location, state fields and the error\_source bit field of the version field have a sticky behavior: they are set when an error occurs, then they are not updated on subsequent errors, and they are cleared when one of the following events occur: POR, nCONFIG, a specific image is loaded, or the error status is specifically cleared from either U-Boot or Linux.

The state field has two bit fields:

**Table 12. State fields**

Bit Field	Bits	Description
major_error_code	31:16	Major error code, see below for possible values.
minor_error_code	15:0	Minor error code, opaque value

The following major error codes are defined:

**Table 13. RSU Major Error Codes**

Major Error Code	Description
0xF001	BITSTREAM_ERROR
0xF002	HARDWARE_ACCESS_FAILURE
0xF003	BITSTREAM_CORRUPTION
0xF004	INTERNAL_ERROR
continued...	

Major Error Code	Description
0xF005	DEVICE_ERROR
0xF006	HPS_WATCHDOG_TIMEOUT
0xF007	INTERNAL_UNKNOWN_ERROR

The minor error code is typically an opaque value, with no meaning for you. The only exception is for the case where the major error code is 0xF006 (HPS\_WATCHDOG\_TIMEOUT), in which case the minor error code is the value reported by the HPS to SDM through the RSU Notify command before the watchdog timeout occurred.

Starting with Intel Quartus Prime Pro Edition software version 20.4, the following INTERNAL\_ERROR codes are reported by the decision firmware and have specific meanings, as defined below:

**Table 14. Decision Firmware Error Codes**

Major Error Code	Minor Error Code	Description
0xF004	0xD00F	Decision firmware data was corrupted, factory image was loaded.
0xF004	0xD010	Configuration pointer block 0 was corrupted, configuration pointer block 1 was used instead.
0xF004	0xD011	Both configuration blocks 0 and 1 were corrupted, factory image was loaded.

For a more complete list of possible error codes, refer to [Mailbox Client Intel FPGA IP User Guide, Appendix: CONFIG\\_STATUS and RSU\\_STATUS Error Code Descriptions](#).

The version component has the following bit fields:

**Table 15. Version fields**

Bit Field	Bits	Description
current_dcmf_index	31:28	Index of the decision firmware copy that was used last time. Possible values: 0,1,2,3.
error_source	27:16	Source of the recorded error: <ul style="list-style-type: none"> <li>0x000: for no error</li> <li>0xACF: if the error was produced by an application or factory image firmware</li> <li>0xDCF: if the error was produced by the decision firmware</li> </ul>
acmf_version	15:8	Current image firmware RSU interface version.
dcmf_version	7:0	Decision firmware RSU interface version.

**Note:** Intel Quartus Prime Pro Edition version 20.1 is the first version which support RSU for Intel Agilex, and both acmf\_version and dcmf\_version are set to 0x02 for this release.

## C. U-Boot RSU Reference Information

### C.1. Configuration Parameters

The U-Boot RSU configuration parameters are stored as environment variables. If they are not specified, the default values are used.

Variable	Default Value	Description
<code>rsu_protected_slot</code>	<empty>	Instruct U-Boot to block any attempts to modify the specified slot.
<code>rsu_log_level</code>	7	Instruct U-Boot on how much logging information to be displayed.
<code>rsu_spt_checksum</code>	<empty>	Enable checking and maintaining SPT checksums. Requires initial image to be created using a Quartus Programming File Generator version 20.4 or newer.

#### C.1.1. `rsu_protected_slot`

This option allows protecting a certain slot number, so that U-Boot is not able to modify it. By default, no slot is protected. Only slots between 0 and 31 can be protected by this feature.

#### C.1.2. `rsu_log_level`

Various RSU functions output log messages with the help of the `rsu_log` function. Each message has a log level associated with it, as shown in the table below:

Log Level	Description
0	Emergency messages
1	Alert messages
2	Critical messages
3	Error messages
4	Warning messages
5	Notice messages
6	Info messages
7	Debug messages

The `rsu_log_level` environment variable allows customizing how much logging information is displayed. As long as the message to be displayed has a lower log level than the current `rsu_log_level`, the message is displayed.

If not defined in the environment, the default value for `rsu_log_level` is 7, which means all logging messages except the debug messages are displayed.

To enable all log messages, set the `rsu_log_level` to 8:

```
SOCFPGA # setenv rsu_log_level 8
```

To disable all messages, set the `rsu_log_level` to 0:

```
SOCFPGA # setenv rsu_log_level 0
```

### C.1.3. `rsu_spt_checksum`

This parameter allows the user to enable the checking and maintaining of SPT checksums. The option only has effect if the initial flash image was created with Quartus Programming File Generator version 20.4 or newer.

The default parameter value is empty, which means that checking and maintaining SPT checksums is disabled. Set to "1" to enable the feature, set to "0" to disable it,

## C.2. Error Codes

In case of success, the RSU APIs return the value 0; otherwise, the RSU APIs return the values shown below, as negative values:

```
#define EINTF          1
#define ECFG           2
#define ESLOTNUM       3
#define EFORMAT        4
#define EERASE         5
#define EPROGRAM       6
#define ECMP           7
#define ESIZE          8
#define ENAME          9
#define EFILEIO        10
#define ECALLBACK      11
#define ELOWLEVEL      12
#define EWRPROT        13
#define EARGS          14
#define ECORRUPTED_CPB 15
#define ECORRUPTED_SPT 16
```

These error codes are defined in `arch/arm/mach-socfpga/include/mach/rsu.h`.

### C.3. Using U-Boot RSU Without a Valid SPT or CPB

The U-Boot RSU APIs and commands can also be used when the SPTs or CPBs in flash are corrupted, but with reduced functionality. APIs and commands are provided to restore a saved SPT or CPB, and also to create an empty CPB. After the CPBs and SPTs are repaired using these APIs, the full RSU functionality is available.

If only one SPT is corrupted, the `rsu_init` API restores it from the good copy. If only one CPB is corrupted, the `rsu_init` API restores it from the good copy.

If both SPTs are corrupted, the `rsu_init` is still successful (return code 0) but some of the APIs returns the error code `ECORRUPTED_SPT` when called. Both SPTs being corrupted cause `rsu_init` to not be able to identify the location of the CPBs, and the CPBs is also considered as corrupted.

If both CPBs are corrupted, the `rsu_init` is still successful (return code 0) but some of the APIs returns the error code `ECORRUPTED_CPB` when called.

**Note:** The API `rsu_init` is only called once by U-Boot, the first time an RSU command is called. A reset or power cycle is required to cause `rsu_init` to be called again.

The table below lists which APIs require valid SPT or valid CPB.

**Table 16. APIs which require valid SPT or CPB**

API	Requires Valid SPT	Requires Valid CPB
<code>rsu_init</code>		
<code>rsu_exit</code>		
<code>rsu_slot_count</code>	yes	
<code>rsu_slot_by_name</code>	yes	
<code>rsu_slot_get_info</code>	yes	yes
<code>rsu_slot_size</code>	yes	
<code>rsu_slot_priority</code>	yes	yes
<code>rsu_slot_erase</code>	yes	yes
<code>rsu_slot_program_buf</code>	yes	yes
<code>rsu_slot_program_factory_update_buf</code>	yes	yes
<code>rsu_slot_program_buf_raw</code>	yes	
<code>rsu_slot_program_file_raw</code>	yes	
<code>rsu_slot_verify_buf</code>	yes	yes
<code>rsu_slot_verify_buf_raw</code>	yes	
<code>rsu_slot_enable</code>	yes	yes
<code>rsu_slot_disable</code>	yes	yes
<code>rsu_slot_load</code>	yes	yes
<code>rsu_slot_load_factory</code>	yes	
<code>rsu_slot_rename</code>	yes	
<code>rsu_slot_delete</code>	yes	yes
<code>rsu_slot_create</code>	yes	
<code>rsu_status_log</code>		
<code>rsu_notify</code>		
<code>rsu_clear_error_status</code>		
<code>rsu_reset_retry_counter</code>		
<code>rsu_dcmf_version</code>		
<code>rsu_max_retry</code>		
<code>rsu_dcmf_status</code>		
<code>rsu_create_empty_cpb</code>		
continued...		

API	Requires Valid SPT	Requires Valid CPB
rsu_restore_cpb		
rsu_save_cpb		yes
rsu_restore_spt		
rsu_save_spt	yes	
rsu_running_factory	yes	

## C.4. Macros

The following macros are available to extract the fields from the `version` field of the `rsu_status_info` structure:

```
/* Macros for extracting RSU version fields */
#define RSU_VERSION_CRT_DCMF_IDX(v)    FIELD_GET(RSU_VERSION_CRT_IDX_MASK, (v))
#define RSU_VERSION_ERROR_SOURCE(v)    FIELD_GET(RSU_VERSION_ERR_MASK, (v))
#define RSU_VERSION_ACMF_VERSION(v)    FIELD_GET(RSU_VERSION_ACMF_MASK, (v))
#define RSU_VERSION_DCMF_VERSION(v)    FIELD_GET(RSU_VERSION_DCMF_MASK, (v))
```

The following macros are available to extract the fields from the versions returned by the `rsu_dcmf_version` function:

```
/* Macros for extracting DCMF version fields */
#define DCMF_VERSION_MAJOR(v)          FIELD_GET(DCMF_VERSION_MAJOR_MASK, (v))
#define DCMF_VERSION_MINOR(v)          FIELD_GET(DCMF_VERSION_MINOR_MASK, (v))
#define DCMF_VERSION_UPDATE(v)         FIELD_GET(DCMF_VERSION_UPDATE_MASK, (v))
```

These macros are define in `arch/arm/mach-socfpga/include/mach/rsu.h`.

## C.5. Data Types

### C.5.1. `rsu_slot_info`

This structure contains slot (SPT entry) information.

```
struct rsu_slot_info {
    char name[16];
    u64 offset;
    u32 size;
    int priority;
};
```

### C.5.2. `rsu_status_info`

This structure contains the RSU status information.

```
struct rsu_status_info {
    u64 current_image;
    u64 fail_image;
    u32 state;
    u32 version;
    u32 error_location;
    u32 error_details;
    u32 retry_counter;
};
```

## C.6. Functions

This section presents the RSU functions, as defined in `arch/arm/mach-socfpga/include/mach/rsu.h`.

### C.6.1. `rsu_init`

<b>Prototype</b>	<code>int rsu_init(char *filename);</code>
<b>Description</b>	Load the configuration file and initialize internal data by reading SPT and CPB data from flash. <ul style="list-style-type: none"> <li>• If SPT0 is corrupted, SPT1 is loaded instead. If one SPT is corrupted and one is good, the corrupted SPT is recovered with information from the good SPT.</li> <li>• If CPB0 is corrupted, CPB1 is loaded instead. If one CPB is corrupted and one is good, the corrupted CPB is recovered with information from the good CPB.</li> </ul>
<b>Parameters</b>	filename: NULL for qspi
<b>Return Value</b>	0 on success, or error code

### C.6.2. `rsu_exit`

<b>Prototype</b>	<code>int rsu_exit(void);</code>
<b>Description</b>	Perform cleanup and exit.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.6.3. `rsu_slot_count`

<b>Prototype</b>	<code>int rsu_slot_count(void);</code>
<b>Description</b>	Get the number of slots defined
<b>Parameters</b>	None
<b>Return Value</b>	The number of defined slots

### C.6.4. `rsu_slot_by_name`

<b>Prototype</b>	<code>int rsu_slot_by_name(char *name);</code>
<b>Description</b>	Get slot number based on name
<b>Parameters</b>	name: name of slot
<b>Return Value</b>	0 on success, or error code

### C.6.5. `rsu_slot_get_info`

<b>Prototype</b>	<code>int rsu_slot_get_info(int slot, struct rsu_slot_info *info);</code>
<b>Description</b>	Get the attributes of a slot
<b>Parameters</b>	slot: slot number info: pointer to info structure to be filled in
<b>Return Value</b>	0 on success, or error code



### C.6.6. rsu\_slot\_size

<b>Prototype</b>	<code>int rsu_slot_size(int slot);</code>
<b>Description</b>	Get the size of a slot
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	The size of the slot in bytes, or error code

### C.6.7. rsu\_slot\_priority

<b>Prototype</b>	<code>int rsu_slot_priority(int slot);</code>
<b>Description</b>	Get the load priority of a slot. Priority of zero means the slot has no priority and is disabled. The slot with priority of one has the highest priority.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	The priority of the slot, or error code

### C.6.8. rsu\_slot\_erase

<b>Prototype</b>	<code>int rsu_slot_erase(int slot);</code>
<b>Description</b>	Erase all data in a slot to prepare for programming. Remove the slot if it is in the CPB.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### C.6.9. rsu\_slot\_program\_buf

<b>Prototype</b>	<code>int rsu_slot_program_buf(int slot, void *buf, int size);</code>
<b>Description</b>	Program a slot using FPGA config data from a buffer and then enter the slot into CPB. The slot must be erased first.
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: size of buffer in bytes
<b>Return Value</b>	0 on success, or error code

### C.6.10. rsu\_slot\_program\_factory\_update\_buf

<b>Prototype</b>	<code>int rsu_slot_program_factory_update_buf(int slot, void *buf, int size);</code>
<b>Description</b>	Program a slot using factory update data from a buffer and then enter the slot into CPB. The API also works with decision firmware update images. The slot must be erased first.
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: size of buffer in bytes
<b>Return Value</b>	0 on success, or error code

### C.6.11. rsu\_slot\_program\_buf\_raw

<b>Prototype</b>	<code>int rsu_slot_program_buf_raw(int slot, void *buf, int size);</code>
<b>Description</b>	Program a slot using raw data from a buffer. The slot must be erased first. The slot is not added to the CPB.
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: size of buffer in bytes
<b>Return Value</b>	0 on success, or error code

### C.6.12. rsu\_slot\_verify\_buf

<b>Prototype</b>	<code>int rsu_slot_verify_buf(int slot, void *buf, int size);</code>
<b>Description</b>	Verify FPGA config data in a slot against a buffer
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: size of buffer in bytes
<b>Return Value</b>	0 on success, or error code

### C.6.13. rsu\_slot\_verify\_buf\_raw

<b>Prototype</b>	<code>int rsu_slot_verify_buf_raw(int slot, void *buf, int size);</code>
<b>Description</b>	Verify raw data in a slot against a buffer
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: size of buffer in bytes
<b>Return Value</b>	0 on success, or error code

### C.6.14. rsu\_slot\_enable

<b>Prototype</b>	<code>int rsu_slot_enable(int slot);</code>
<b>Description</b>	Set the selected slot as the highest priority.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### C.6.15. rsu\_slot\_disable

<b>Prototype</b>	<code>int rsu_slot_disable(int slot);</code>
<b>Description</b>	Remove the selected slot from the priority scheme, but don't erase the slot data so that it can be re-enabled.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### C.6.16. rsu\_slot\_load

<b>Prototype</b>	<code>int rsu_slot_load(int slot);</code>
<b>Description</b>	Request the selected slot to be loaded immediately. On success, after a small delay, the system is rebooted.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### C.6.17. rsu\_slot\_load\_factory

<b>Prototype</b>	<code>int rsu_slot_load_factory(void);</code>
<b>Description</b>	Request the factory image to be loaded immediately. On success, after a small delay, the system is rebooted.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.6.18. rsu\_slot\_rename

<b>Prototype</b>	<code>int rsu_slot_rename(int slot, char *name);</code>
<b>Description</b>	Rename the selected slot.
<b>Parameters</b>	slot: slot number name: new name for the slot
<b>Return Value</b>	0 on success, or error code

### C.6.19. rsu\_slot\_delete

<b>Prototype</b>	<code>int rsu_slot_delete(int slot);</code>
<b>Description</b>	Delete the slot from SPT, freeing up space.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### C.6.20. rsu\_slot\_create

<b>Prototype</b>	<code>int rsu_slot_create(char *name, __u64 address, unsigned int size);</code>
<b>Description</b>	Add a new slot to the SPT, using unused space.
<b>Parameters</b>	name: new slot name address: new slot flash start address size: new slot size, in bytes
<b>Return Value</b>	0 on success, or error code

### C.6.21. rsu\_status\_log

<b>Prototype</b>	<code>int rsu_status_log(struct rsu_status_info *info);</code>
<b>Description</b>	Copy firmware status log to info structure.
<b>Parameters</b>	info: pointer to info struct to fill in
<b>Return Value</b>	0 on success, or error code

### C.6.22. rsu\_notify

<b>Prototype</b>	<code>int rsu_notify(int stage);</code>
<b>Description</b>	Report HPS software execution stage as a 16bit number
<b>Parameters</b>	stage: software execution stage
<b>Return Value</b>	0 on success, or error code

### C.6.23. rsu\_clear\_error\_status

<b>Prototype</b>	<code>int rsu_clear_error_status(void);</code>
<b>Description</b>	Clear errors from the current RSU status log
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.6.24. rsu\_reset\_retry\_counter

<b>Prototype</b>	<code>int rsu_reset_retry_counter(void);</code>
<b>Description</b>	Request the retry counter to be set to zero, so that the currently running image may be tried again after the next watchdog timeout.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.6.25. rsu\_dcmf\_version

<b>Prototype</b>	<code>int rsu_dcmf_version(u32 *versions);</code>
<b>Description</b>	Retrieve the version of each of the four decision firmware copies in flash. The versions are also stored to be queried from Linux.
<b>Parameters</b>	versions: pointer to where the four DCMF versions are stored
<b>Return Value</b>	0 on success, or error code

**Note:** Refer to *Decision Firmware Version Information* section for details about the content of the version information.

**Note:** Refer to *Macros* section for macros that can be used to extract the firmware version details: major, minor and update version numbers.

### C.6.26. `rsu_max_retry`

<b>Prototype</b>	<code>int rsu_max_retry(u8 *value);</code>
<b>Description</b>	Retrieve the <code>max_retry</code> parameter from flash. The value is also stored to be queried from Linux.
<b>Parameters</b>	<code>value</code> : pointer to where the <code>max_retry</code> parameter is saved.
<b>Return Value</b>	0 on success, or error code

### C.6.27. `rsu_dcmf_status`

<b>Prototype</b>	<code>int rsu_dcmf_status(int *status);</code>
<b>Description</b>	Determine whether decision firmware copies are corrupted in flash, with the currently used decision firmware being used as reference. The status is an array of four values, one for each decision firmware copy. A value of 0 means the copy is fine, anything else means the copy is corrupted. The status is also stored to be queried from Linux.
<b>Parameters</b>	<code>status</code> : pointer to where the status values are stored
<b>Return Value</b>	0 on success, or error code

### C.6.28. `rsu_create_empty_cpb`

<b>Prototype</b>	<code>int rsu_create_empty_cpb(void);</code>
<b>Description</b>	Create an empty CPB, which includes the CPB header only. All entries are marked as unused.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.6.29. `rsu_restore_cpb`

<b>Prototype</b>	<code>int rsu_restore_cpb(u64 address);</code>
<b>Description</b>	Restore CPB from a bufer in memory. The buffer must contain the 4096 bytes of CPB data, followed by 4 bytes containing the CRC32 checksum of the data.
<b>Parameters</b>	<code>address</code> : address of the buffer from which the CPB is restored
<b>Return Value</b>	0 on success, or error code

### C.6.30. `rsu_save_cpb`

<b>Prototype</b>	<code>int rsu_save_cpb(u64 address);</code>
<b>Description</b>	Save CPB to a memory buffer. A total of 4100 bytes are written: 4096 for the data, plus 4 bytes with a CRC32 checksum of the data.
<b>Parameters</b>	<code>address</code> : address of the buffer where CPB is saved to
<b>Return Value</b>	0 on success, or error code

### C.6.31. rsu\_restore\_spt

<b>Prototype</b>	<code>int rsu_restore_spt(u64 address);</code>
<b>Description</b>	Restore SPT from a bufer in memory. The buffer must contain the 4096 bytes of SPT data, followed by 4 bytes containing the CRC32 checksum of the data.
<b>Parameters</b>	address: address of the buffer from which the SPT is restored
<b>Return Value</b>	0 on success, or error code

### C.6.32. rsu\_save\_spt

<b>Prototype</b>	<code>int rsu_save_spt(u64 address);</code>
<b>Description</b>	Save SPT to an address in memory. A total of 4100 bytes are written: 4096 for the data, plus 4 bytes with a CRC32 checksum of the data.
<b>Parameters</b>	address: address where SPT and CRC32 checksum are saved to
<b>Return Value</b>	0 on success, or error code

### C.6.33. rsu\_running\_factory

<b>Prototype</b>	<code>int rsu_running_factory(int *factory);</code>
<b>Description</b>	Determine if current running image is factory image.
<b>Parameters</b>	factory: value at this address is set to 1 if factory image is currently running, 0 otherwise.
<b>Return Value</b>	0 on success, or error code

## C.7. RSU U-Boot Commands

This section presents the RSU U-Boot commands, as implemented in `arch/arm/mach-socfpga/rsu_s10.c`.  
dtb

### C.7.1. dtb

<b>Command</b>	<code>rsu dtb</code>
<b>Description</b>	Update Linux DTB MTD partition called <code>qspi_boot</code> to start from the SPT0 start address, and adjust its size accordingly. It assumes the <code>qspi_boot</code> partition starts from zero. The <code>rsu dtb</code> operates on the DTB loaded in memory by U-Boot, before passing it to Linux. The sequence to use is: 1. Load DTB. 2. Run <code>rsu dtb</code> command. 3. Boot Linux
<b>Parameters</b>	None <i>Note:</i> If no partition called <code>qspi_boot</code> is found, the command fails.
<b>Return Value</b>	0 on success, or error code

### C.7.2. list

<b>Command</b>	<code>rsu list</code>
<b>Description</b>	List the partitioning information, display the status information and list the contents of the CPB.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.7.3. slot\_by\_name

<b>Command</b>	<code>rsu slot_by_name &lt;name&gt;</code>
<b>Description</b>	Find slot by name and display the slot number.
<b>Parameters</b>	<code>&lt;name&gt;</code> : slot name
<b>Return Value</b>	0 on success, or error code

### C.7.4. slot\_count

<b>Command</b>	<code>rsu slot_count</code>
<b>Description</b>	Display the slot count
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.7.5. slot\_disable

<b>Command</b>	<code>rsu slot_disable &lt;slot&gt;</code>
<b>Description</b>	Remove slot from CPB
<b>Parameters</b>	<code>&lt;slot&gt;</code> : slot number
<b>Return Value</b>	0 on success, or error code

### C.7.6. slot\_enable

<b>Command</b>	<code>rsu slot_enable &lt;slot&gt;</code>
<b>Description</b>	Add slot to CPB and make slot the highest priority.
<b>Parameters</b>	<code>&lt;slot&gt;</code> : slot number
<b>Return Value</b>	0 on success, or error code

### C.7.7. slot\_erase

<b>Command</b>	<code>rsu slot_erase &lt;slot&gt;</code>
<b>Description</b>	Take slot out of CPB then erase slot data.
<b>Parameters</b>	<code>&lt;slot&gt;</code> : slot number
<b>Return Value</b>	0 on success, or error code

### C.7.8. slot\_get\_info

<b>Command</b>	rsu slot_get_info <slot>
<b>Description</b>	Display slot information
<b>Parameters</b>	<slot>: slot number
<b>Return Value</b>	0 on success, or error code

### C.7.9. slot\_load

<b>Command</b>	rsu slot_load <slot>
<b>Description</b>	Load slot immediately
<b>Parameters</b>	<slot>: slot number
<b>Return Value</b>	0 on success, or error code

### C.7.10. slot\_load\_factory

<b>Command</b>	rsu slot_load_factory
<b>Description</b>	Load factory immediately
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.7.11. slot\_priority

<b>Command</b>	rsu slot_priority <slot>
<b>Description</b>	Display slot priority
<b>Parameters</b>	<slot>: slot number
<b>Return Value</b>	0 on success, or error code

### C.7.12. slot\_program\_buf

<b>Command</b>	rsu slot_program_buf <slot> <buffer> <size>
<b>Description</b>	Program buffer into slot, and make it highest priority. The slot must be erased first.
<b>Parameters</b>	<slot>: slot number <buffer>: address of buffer in memory <size>: length of buffer, in bytes
<b>Return Value</b>	0 on success, or error code



### C.7.13. slot\_program\_buf\_raw

<b>Command</b>	<code>rsu slot_program_buf_raw &lt;slot&gt; &lt;buffer&gt; &lt;size&gt;</code>
<b>Description</b>	Program raw buffer into slot. The slot must be erased first. The slot is not added to CPB.
<b>Parameters</b>	<code>&lt;slot&gt;</code> : slot number <code>&lt;buffer&gt;</code> : address of buffer in memory <code>&lt;size&gt;</code> : length of buffer, in bytes
<b>Return Value</b>	0 on success, or error code

### C.7.14. slot\_program\_factory\_update\_buf

<b>Command</b>	<code>rsu slot_program_factory_update_buf &lt;slot&gt; &lt;buffer&gt; &lt;size&gt;</code>
<b>Description</b>	Program factory update buffer into slot, and make it highest priority. The slot must be erased first.
<b>Parameters</b>	<code>&lt;slot&gt;</code> : slot number <code>&lt;buffer&gt;</code> : address of buffer in memory <code>&lt;size&gt;</code> : length of buffer, in bytes
<b>Return Value</b>	0 on success, or error code

### C.7.15. slot\_rename

<b>Command</b>	<code>rsu slot_rename &lt;slot&gt; &lt;name&gt;</code>
<b>Description</b>	Rename slot.
<b>Parameters</b>	<code>&lt;slot&gt;</code> : slot number <code>&lt;name&gt;</code> : new slot name
<b>Return Value</b>	0 on success, or error code

### C.7.16. slot\_delete

<b>Command</b>	<code>rsu slot_delete &lt;slot&gt;</code>
<b>Description</b>	Remove slot from SPT, freeing up flash space.
<b>Parameters</b>	<code>&lt;slot&gt;</code> : slot number
<b>Return Value</b>	0 on success, or error code

### C.7.17. slot\_create

<b>Command</b>	<code>rsu slot_create &lt;slot&gt; &lt;buffer&gt; &lt;size&gt;</code>
<b>Description</b>	Create a slot in the SPT, using unallocated flash space.
<b>Parameters</b>	<code>&lt;name&gt;</code> : slot name <code>&lt;address&gt;</code> : slot start address in flash <code>&lt;size&gt;</code> : slot size, in bytes
<b>Return Value</b>	0 on success, or error code

### C.7.18. slot\_size

<b>Command</b>	rsu slot_size <slot>
<b>Description</b>	Display slot size
<b>Parameters</b>	<slot>: slot number
<b>Return Value</b>	0 on success, or error code

### C.7.19. slot\_verify\_buf

<b>Command</b>	rsu slot_verify_buf <slot> <buffer> <size>
<b>Description</b>	Verify slot contents against buffer.
<b>Parameters</b>	<slot>: slot number <buffer>: address of buffer in memory <size>: length of buffer, in bytes
<b>Return Value</b>	0 on success, or error code

### C.7.20. slot\_verify\_buf\_raw

<b>Command</b>	rsu slot_verify_buf_raw <slot> <buffer> <size>
<b>Description</b>	Verify slot contents against raw buffer.
<b>Parameters</b>	<slot>: slot number <buffer>: address of buffer in memory <size>: length of buffer, in bytes
<b>Return Value</b>	0 on success, or error code

### C.7.21. status\_log

<b>Command</b>	rsu status_log
<b>Description</b>	Display RSU status
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.7.22. update

<b>Command</b>	rsu update <flash_offset>
<b>Description</b>	Initiate firmware to load bitstream as specified by flash_offset
<b>Parameters</b>	<flash_offset>: address of bitstream
<b>Return Value</b>	0 on success, or error code

### C.7.23. notify

<b>Command</b>	rsu notify <value>
<b>Description</b>	Let SDM know the current state of HPS software
<b>Parameters</b>	<value>: state of HPS software as 16-bit number
<b>Return Value</b>	0 on success, or error code

### C.7.24. clear\_error\_status

<b>Command</b>	rsu clear_error_status
<b>Description</b>	Clear the RSU error status
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.7.25. reset\_retry\_counter

<b>Command</b>	rsu reset_retry_counter
<b>Description</b>	Reset the RSU retry counter
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.7.26. display\_dcmf\_version

<b>Command</b>	rsu display_dcmf_version
<b>Description</b>	Display decision firmware versions. The versions are also stored to be queried from Linux.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

*Note:* Refer to *Decision Firmware Version Information* section for details about the content of the version information.

### C.7.27. display\_dcmf\_status

<b>Command</b>	rsu display_dcmf_status
<b>Description</b>	Display whether decision firmware copies are corrupted in flash, with the currently used decision firmware being used as reference. The status is also stored to be queried from Linux.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.7.28. display\_max\_retry

<b>Command</b>	<code>rsu display_max_retry</code>
<b>Description</b>	Displays the value of the <code>max_retry</code> parameter. The value is also stored to be queried from Linux.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.7.29. restore\_spt

<b>Command</b>	<code>rsu restore_spt &lt;address&gt;</code>
<b>Description</b>	Restore SPT from a bufer in memory. The buffer must contain the 4096 bytes of SPT data, followed by 4 bytes containing the CRC32 checksum of the data.
<b>Parameters</b>	<code>address</code> : address of the buffer from which the SPT is restored
<b>Return Value</b>	0 on success, or error code

### C.7.30. save\_spt

<b>Command</b>	<code>rsu save_spt &lt;address&gt;</code>
<b>Description</b>	Save SPT to an address in memory. A total of 4100 bytes are written: 4096 for the data, plus 4 bytes with a CRC32 checksum of the data. Environment variable <code>\${filesize}</code> is set to "1004" which is 4100 in hex.
<b>Parameters</b>	<code>&lt;address&gt;</code> : address where SPT and CRC32 checksum are saved to
<b>Return Value</b>	0 on success, or error code

### C.7.31. create\_empty\_cpb

<b>Command</b>	<code>rsu create_empty_cpb</code>
<b>Description</b>	Create an empty CPB, which includes the CPB header only. All entries are marked as unused.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### C.7.32. restore\_cpb

<b>Command</b>	<code>rsu restore_cpb &lt;address&gt;</code>
<b>Description</b>	Restore CPB from a bufer in memory. The buffer must contain the 4096 bytes of CPB data, followed by 4 bytes containing the CRC32 checksum of the data.
<b>Parameters</b>	<code>address</code> : address of the buffer from which the CPB is restored
<b>Return Value</b>	0 on success, or error code

### C.7.33. save\_cpb

<b>Command</b>	rsu save_cpb <address>
<b>Description</b>	Save CPB to an address in memory. A total of 4100 bytes are written: 4096 for the data, plus 4 bytes with a CRC32 checksum of the data. Environment variable <code>\${filesize}</code> is set to "1004" which is 4100 in hex.
<b>Parameters</b>	<address>: address where CPB and CRC32 checksum are saved to
<b>Return Value</b>	0 on success, or error code

### C.7.34. check\_running\_factory

<b>Command</b>	rsu check_running_factory
<b>Description</b>	Check and display whether the currently running image is the factory image.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

## D. LIBRSU Reference Information

### D.1. Configuration File

The LIBRSU library relies on the resource file `/etc/librsu.rc` to set the logging level and the MTD QSPI partition to be used for RSU purposes.

**Table 17. LIBRSU Configuration File Elements**

Element	Description
<b>Element:</b> # COMMENT <b>Usage:</b> // COMMENT <b>Options:</b> None	Single line comments <b>Required?:</b> No
<b>Element:</b> root <b>Usage:</b> root {type} {path} <b>Options:</b> type: Storage type = [qspi, datafile]	Specifies the storage containing the RSU data region that LIBRSU manages. The datafile type is provided for testing purposes and treats an ordinary file as the RSU data region. <b>Required?:</b> Yes
<b>Element:</b> rsu-dev <b>Usage:</b> rsu-dev {path} <b>Options:</b> <ul style="list-style-type: none"> <li>path : Path to the RSU entries in Linux sysfs</li> </ul>	Specifies the path for the RSU sysfs entries in Linux. <b>Required?:</b> No. When not specified, it defaults to <code>/sys/devices/platform/stratix10-rsu.0</code> .
<b>Element:</b> log <b>Usage:</b> log {level} [stderr path] <b>Options:</b> <ul style="list-style-type: none"> <li>level : Verbose level = [off, low, medium, high]</li> <li>path : Path to a logfile for debug information</li> <li>stderr (defaults to stderr)</li> </ul>	Instruct LIBRSU to open a logfile and append debug information as commands are performed. Three levels of verbosity are allowed. The log is directed to stderr by default. <b>Required?:</b> No
<b>Element:</b> write-protect <b>Usage:</b> write-protect {slot} <b>Options:</b> slot : Slot number	Instruct LIBRSU to block any attempts to modify the specified slot. The priority of the selected slot might change based on changes to other slots. This option can be used multiple times. <b>Required?:</b> No
<b>Element:</b> rsu-spt-checksum <b>Usage:</b> rsu-spt-checksum {enable} <b>Options:</b> enable : 0-disabled, 1-enabled	Instruct LIBRSU to enable checking and maintaining SPT checksums. Only has effect when the initial flash image was created with Quartus Programming File Generator v20.4 or newer. When not defined, it means the option is disabled. <b>Required?:</b> No

The default values are:

```
# cat /etc/librsu.rc
log med stderr
root qspi /dev/mtd0
rsu-dev /sys/devices/platform/stratix10-rsu.0
```

The ATF SMC handler, Linux SVC driver and Linux RSU driver do not export the SDM API for determining the SPT addresses. Therefore, the MTD QSPI partition to be used by LIBRSU must start at the location of the SPT0, in order for LIBRSU to be able to determine the flash partitioning information. This can either be hardcoded in the device tree, or U-Boot can edit the device tree with the appropriate information before passing it to Linux using the `rsu dtb` command.

## D.2. Error Codes

In case of success, the LIBRSU APIs return the value 0; otherwise, the LIBRSU APIs return the values shown below, as negative values:

```
#define ELIB          1
#define ECFG          2
#define ESLOTNUM      3
#define EFORMAT       4
#define EERASE        5
#define EPROGRAM      6
#define ECOMP         7
#define ESIZE         8
#define ENAME         9
#define EFILEIO       10
#define ECALLBACK     11
#define ELOWLEVEL     12
#define EWRPROT       13
#define EARGS         14
#define ECORRUPTED_CPB 15
#define ECORRUPTED_SPT 16
```

## D.3. Using LIBRSU Without a Valid SPT or CPB

LIBRSU can also be used when the SPTs or CPBs in flash are corrupted, but with reduced functionality. APIs are provided to restore a saved SPT or CPB, and also to create an empty CPB. After the CPBs and SPTs are repaired using these APIs, the full LIBRSU functionality is available.

If only one SPT is corrupted, `librsu_init` API restores it from the good copy. If only one CPB is corrupted, the `librsu_init` API restores it from the good copy.

If both SPTs are corrupted, the `librsu_init` is still successful (return code 0) but some of the APIs return the error code `ECORRUPTED_SPT` when called. Both SPTs being corrupted cause LIBRSU to not be able to identify the location of the CPBs, and the CPBs are also considered as corrupted.

If both CPBs are corrupted, the `librsu_init` is still successful (return code 0) but some of the APIs return the error code `ECORRUPTED_CPB` when called.

The table below lists which APIs require valid SPT or valid CPB.

**Table 18. APIs which require valid SPT or CPB**

API	Requires Valid SPT	Requires Valid CPB
librsu_init		
librsu_exit		
rsu_slot_count	yes	
rsu_slot_by_name	yes	
rsu_slot_get_info	yes	yes
rsu_slot_size	yes	
rsu_slot_priority	yes	yes
rsu_slot_erase	yes	yes
rsu_slot_program_buf	yes	yes
rsu_slot_program_factory_update_buf	yes	yes
rsu_slot_program_file	yes	yes
rsu_slot_program_factory_update_file	yes	yes
rsu_slot_program_buf_raw	yes	
rsu_slot_program_file_raw	yes	
rsu_slot_verify_buf	yes	yes
rsu_slot_verify_file	yes	yes
rsu_slot_verify_buf_raw	yes	
rsu_slot_verify_file_raw	yes	
rsu_slot_copy_to_file	yes	yes
rsu_slot_enable	yes	yes
rsu_slot_disable	yes	yes
rsu_slot_load_after_reboot	yes	yes
rsu_slot_load_factory_after_reboot	yes	
rsu_slot_rename	yes	
rsu_slot_delete	yes	yes
rsu_slot_create	yes	
rsu_status_log		
rsu_status_log		
rsu_notify		
rsu_clear_error_status		
rsu_reset_retry_counter		
rsu_dcmf_version		
continued...		



API	Requires Valid SPT	Requires Valid CPB
rsu_max_retry		
rsu_save_spt	yes	
rsu_restore_spt		
rsu_save_cpb		yes
rsu_create_empty_cpb		
rsu_restore_cpb		
rsu_running_factory	yes	

## D.4. Macros

The following macros are available to extract the fields from the version field of the `rsu_status_info` structure:

```
#define RSU_VERSION_CRT_DCMF_IDX(v) (((v) & 0xF0000000) >> 28)
#define RSU_VERSION_ERROR_SOURCE(v) (((v) & 0x0FFF0000) >> 16)
#define RSU_VERSION_ACMF_VERSION(v) (((v) & 0xFF00) >> 8)
#define RSU_VERSION_DCMF_VERSION(v) ((v) & 0xFF)
```

The following macros are available to extract the fields from the versions returned by the `rsu_dcmf_version` function:

```
#define DCMF_VERSION_MAJOR(v) (((v) & 0xFF000000) >> 24)
#define DCMF_VERSION_MINOR(v) (((v) & 0x00FF0000) >> 16)
#define DCMF_VERSION_UPDATE(v) (((v) & 0x0000FF00) >> 8)
```

The following macros define error codes returned by the decision firmware:

```
#define STATE_DCIO_CORRUPTED 0xF004D00F
#define STATE_CPB0_CORRUPTED 0xF004D010
#define STATE_CPB0_CPB1_CORRUPTED 0xF004D011
```

Refer to [RSU Status and Error Codes](#) on page 114 for more details on these error codes.

## D.5. Data Types

### D.5.1. `rsu_slot_info`

This structure contains slot (SPT entry) information.

```
struct rsu_slot_info {
    char name[16];
    __u64 offset;
    int size;
    int priority;
};
```

## D.5.2. rsu\_status\_info

This structure contains the RSU status information.

```
struct rsu_status_info {
    __u64 version;
    __u64 state;
    __u64 current_image;
    __u64 fail_image;
    __u64 error_location;
    __u64 error_details;
    __u64 retry_counter;
};
```

## D.5.3. rsu\_data\_callback

This is a callback used in a scheme to provide data in blocks as opposed to all at once in a large buffer, which may minimize memory requirements.

```
/*
 * rsu_data_callback - function pointer type for data source callback
 */
typedef int (*rsu_data_callback)(void *buf, int size);
```

## D.6. Functions

### D.6.1. librsu\_init

<b>Prototype</b>	<code>int librsu_init(char *filename);</code>
<b>Description</b>	Load the configuration file and initialize internal data by reading SPT and CPB data from flash. <ul style="list-style-type: none"> <li>• If SPT0 is corrupted, SPT1 is loaded instead. If one SPT is corrupted and one is good, the corrupted SPT is recovered with information from the good SPT.</li> <li>• If CPB0 is corrupted, CPB1 is loaded instead. If one CPB is corrupted and one is good, the corrupted CPB is recovered with information from the good CPB.</li> </ul>
<b>Parameters</b>	filename: configuration file to load. If Null or empty string, the default is /etc/librsu.rc
<b>Return Value</b>	0 on success, or error code

### D.6.2. librsu\_exit

<b>Prototype</b>	<code>void librsu_exit(void);</code>
<b>Description</b>	Cleanup internal data and release librsu.
<b>Parameters</b>	None
<b>Return Value</b>	None

### D.6.3. rsu\_slot\_count

<b>Prototype</b>	<code>int rsu_slot_count(void);</code>
<b>Description</b>	Get the number of slots defined.
<b>Parameters</b>	None
<b>Return Value</b>	The number of defined slots

### D.6.4. rsu\_slot\_by\_name

<b>Prototype</b>	<code>int rsu_slot_by_name(char *name);</code>
<b>Description</b>	Retrieve slot number based on name.
<b>Parameters</b>	name: name of slot
<b>Return Value</b>	Slot number on success, or error code

### D.6.5. rsu\_slot\_get\_info

<b>Prototype</b>	<code>int rsu_slot_get_info(int slot, struct rsu_slot_info *info);</code>
<b>Description</b>	Retrieve the attributes of a slot.
<b>Parameters</b>	slot: slot number info: pointer to info structure to be filled in
<b>Return Value</b>	0 on success, or error code

### D.6.6. rsu\_slot\_size

<b>Prototype</b>	<code>int rsu_slot_size(int slot);</code>
<b>Description</b>	Get the size of a slot.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	The size of the slot in bytes, or error code

### D.6.7. rsu\_slot\_priority

<b>Prototype</b>	<code>int rsu_slot_priority(int slot);</code>
<b>Description</b>	Get the load priority of a slot. Priority of zero means the slot has no priority and is disabled. The slot with priority of one has the highest priority.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	The priority of the slot, or error code

### D.6.8. rsu\_slot\_erase

<b>Prototype</b>	<code>int rsu_slot_erase(int slot);</code>
<b>Description</b>	Erase all data in a slot to prepare for programming. Remove the slot if it is in the CPB.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### D.6.9. rsu\_slot\_program\_buf

<b>Prototype</b>	<code>int rsu_slot_program_buf(int slot, void *buf, int size);</code>
<b>Description</b>	Program a slot using FPGA config data from a buffer and enter slot into CPB as highest priority. The slot must be erased first.
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: bytes to read from buffer
<b>Return Value</b>	0 on success, or error code

### D.6.10. rsu\_slot\_program\_factory\_update\_buf

<b>Prototype</b>	<code>int rsu_slot_program_factory_update_buf(int slot, void *buf, int size);</code>
<b>Description</b>	Program a slot using a factory update image data from a buffer and enter slot into CPB as highest priority. The command also works with Decision Firmware update images. The slot must be erased first.
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: bytes to read from buffer
<b>Return Value</b>	0 on success, or error code

### D.6.11. rsu\_slot\_program\_file

<b>Prototype</b>	<code>int rsu_slot_program_file(int slot, char *filename);</code>
<b>Description</b>	Program a slot using FPGA config data from a file and enter slot into CPB as highest priority. The slot must be erased first.
<b>Parameters</b>	slot: slot number filename: input data file
<b>Return Value</b>	0 on success, or error code

### D.6.12. rsu\_slot\_program\_factory\_update\_file

<b>Prototype</b>	<code>int rsu_slot_program_factory_update_file(int slot, char *filename);</code>
<b>Description</b>	Program a slot using a factory update image data from a file and enter slot into CPB as highest priority. The command also works with Decision Firmware update images. The slot must be erased first.
<b>Parameters</b>	slot: slot number filename: input data file
<b>Return Value</b>	0 on success, or error code

### D.6.13. rsu\_slot\_program\_buf\_raw

<b>Prototype</b>	<code>int rsu_slot_program_buf_raw(int slot, void *buf, int size);</code>
<b>Description</b>	Program a slot using raw data from a buffer. The slot is not entered into the CPB. The slot must be erased first.
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: bytes to read from buffer
<b>Return Value</b>	0 on success, or error code

### D.6.14. rsu\_slot\_program\_file\_raw

<b>Prototype</b>	<code>int rsu_slot_program_file_raw(int slot, char *filename);</code>
<b>Description</b>	Program a slot using raw data from a file. The slot is not entered into the CPB. The slot must be erased first.
<b>Parameters</b>	slot: slot number filename: input data file
<b>Return Value</b>	0 on success, or error code

### D.6.15. rsu\_slot\_verify\_buf

<b>Prototype</b>	<code>int rsu_slot_verify_buf(int slot, void *buf, int size);</code>
<b>Description</b>	Verify FPGA config data in a slot against a buffer.
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: bytes to read from buffer
<b>Return Value</b>	0 on success, or error code

### D.6.16. rsu\_slot\_verify\_file

<b>Prototype</b>	<code>int rsu_slot_verify_file(int slot, char *filename);</code>
<b>Description</b>	Verify FPGA config data in a slot against a file.
<b>Parameters</b>	slot: slot number filename: input data file
<b>Return Value</b>	0 on success, or error code

### D.6.17. rsu\_slot\_verify\_buf\_raw

<b>Prototype</b>	<code>int rsu_slot_verify_buf_raw(int slot, void *buf, int size);</code>
<b>Description</b>	Verify raw data in a slot against a buffer.
<b>Parameters</b>	slot: slot number buf: pointer to data buffer size: bytes to read from buffer
<b>Return Value</b>	0 on success, or error code

### D.6.18. rsu\_slot\_verify\_file\_raw

<b>Prototype</b>	<code>int rsu_slot_verify_file_raw(int slot, char *filename);</code>
<b>Description</b>	Verify raw data in a slot against a file.
<b>Parameters</b>	slot: slot number filename: input data file
<b>Return Value</b>	0 on success, or error code

### D.6.19. rsu\_slot\_program\_callback

<b>Prototype</b>	<code>int rsu_slot_program_callback(int slot, rsu_data_callback callback);</code>
<b>Description</b>	Program and verify a slot using FPGA config data provided by a callback function. Enter the slot into the CPB as highest priority.
<b>Parameters</b>	slot: slot number callback: callback function to provide input data
<b>Return Value</b>	0 on success, or error code

### D.6.20. rsu\_slot\_program\_callback\_raw

<b>Prototype</b>	<code>int rsu_slot_program_callback_raw(int slot, rsu_data_callback callback);</code>
<b>Description</b>	Program and verify a slot using raw data provided by a callback function. The slot is not entered into the CPB.
<b>Parameters</b>	slot: slot number callback: callback function to provide input data
<b>Return Value</b>	0 on success, or error code

### D.6.21. rsu\_slot\_verify\_callback

<b>Prototype</b>	<code>int rsu_slot_verify_callback(int slot, rsu_data_callback callback);</code>
<b>Description</b>	Verify a slot using FPGA configuration data provided by a callback function.
<b>Parameters</b>	slot: slot number callback: callback function to provide input data
<b>Return Value</b>	0 on success, or error code

### D.6.22. rsu\_slot\_verify\_callback\_raw

<b>Prototype</b>	<code>int rsu_slot_verify_callback_raw(int slot, rsu_data_callback callback);</code>
<b>Description</b>	Verify a slot using raw data provided by a callback function.
<b>Parameters</b>	slot: slot number callback: callback function to provide input data
<b>Return Value</b>	0 on success, or error code

### D.6.23. `rsu_slot_copy_to_file`

<b>Prototype</b>	<code>int rsu_slot_copy_to_file(int slot, char *filename);</code>
<b>Description</b>	Read the data in a slot and write to a file.
<b>Parameters</b>	slot: slot number filename: input data file
<b>Return Value</b>	0 on success, or error code

### D.6.24. `rsu_slot_enable`

<b>Prototype</b>	<code>int rsu_slot_enable(int slot);</code>
<b>Description</b>	Set the selected slot as the highest priority. This is the first slot attempted after a power-on reset.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### D.6.25. `rsu_slot_disable`

<b>Prototype</b>	<code>int rsu_slot_disable(int slot);</code>
<b>Description</b>	Remove the selected slot from the priority scheme, but do not erase the slot data so that it can be re-enabled.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### D.6.26. `rsu_slot_load_after_reboot`

<b>Prototype</b>	<code>int rsu_slot_load_after_reboot(int slot);</code>
<b>Description</b>	Request the selected slot to be loaded after the next Linux <code>reboot</code> command which would otherwise trigger an HPS cold reset. Unless the Linux <code>reboot</code> command is issued, this function has no effect. If the Linux <code>reboot</code> command is configured to trigger a warm reset by passing the <code>reboot=warm</code> parameter to the kernel, the function also has no effect.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### D.6.27. `rsu_slot_load_factory_after_reboot`

<b>Prototype</b>	<code>int rsu_slot_load_factory_after_reboot(void);</code>
<b>Description</b>	Request the factory image to be loaded after the next Linux <code>reboot</code> command which would otherwise trigger an HPS cold reset. Unless the Linux <code>reboot</code> command is issued, this function has no effect. If the Linux <code>reboot</code> command is configured to trigger a warm reset by passing the <code>reboot=warm</code> parameter to the kernel, the function also has no effect.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.6.28. rsu\_slot\_rename

<b>Prototype</b>	<code>int rsu_slot_rename(int slot, char *name);</code>
<b>Description</b>	Rename the selected slot.
<b>Parameters</b>	slot: slot number name: new name for slot
<b>Return Value</b>	0 on success, or error code

### D.6.29. rsu\_slot\_delete

<b>Prototype</b>	<code>int rsu_slot_delete(int slot);</code>
<b>Description</b>	Delete the slot from SPT, freeing up space.
<b>Parameters</b>	slot: slot number
<b>Return Value</b>	0 on success, or error code

### D.6.30. rsu\_slot\_create

<b>Prototype</b>	<code>int rsu_slot_create(char *name, __u64 address, unsigned int size);</code>
<b>Description</b>	Add a new slot to the SPT, using unused space.
<b>Parameters</b>	name: new slot name address: new slot flash start address size: new slot size, in bytes
<b>Return Value</b>	0 on success, or error code

### D.6.31. rsu\_status\_log

<b>Prototype</b>	<code>int rsu_status_log(struct rsu_status_info *info);</code>
<b>Description</b>	Copy the SDM status log to info struct.
<b>Parameters</b>	info: pointer to info struct to fill in
<b>Return Value</b>	0 on success, or error code

### D.6.32. rsu\_notify

<b>Prototype</b>	<code>int rsu_notify(int value);</code>
<b>Description</b>	Report HPS software execution stage as a 16-bit number.
<b>Parameters</b>	value: HPS software execution stage - only 16-bit lower bits are used
<b>Return Value</b>	0 on success, or error code



### D.6.33. `rsu_clear_error_status`

<b>Prototype</b>	<code>int rsu_clear_error_status(void);</code>
<b>Description</b>	Clear the sticky error fields from the current status log.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.6.34. `rsu_reset_retry_counter`

<b>Prototype</b>	<code>int rsu_reset_retry_counter(void);</code>
<b>Description</b>	Reset the <code>retry</code> counter, so that the currently running image can try again after a watchdog timeout, if the value of <code>max_retry</code> is greater than one.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.6.35. `rsu_dcmf_version`

<b>Prototype</b>	<code>int rsu_dcmf_version(__u32 *versions);</code>
<b>Description</b>	Retrieve the four decision firmware versions. Requires U-Boot API <code>rsu_dcmf_version</code> or U-Boot command <code>rsu_display_dcmf_version</code> to be called before booting Linux, otherwise the version is reported as 0.0.0 The reported versions are the ones valid at the time the U-Boot command was called, and do not reflect any changes that may occur after that.
<b>Parameters</b>	<code>versions</code> : pointer to store the four decision firmware versions.
<b>Return Value</b>	0 on success, or error code

**Note:** Refer to *Decision Firmware Version Information* section for details about the content of the version information.

**Note:** Refer to *Macros* section for macros that can be used to extract the firmware version details: major, minor and update version numbers.

#### Related Information

- [Firmware Version Information](#) on page 30
- [Macros](#) on page 137

### D.6.36. `rsu_max_retry`

<b>Prototype</b>	<code>int rsu_max_retry(__u8 *value);</code>
<b>Description</b>	Retrieve the <code>max_retry</code> parameter from flash. Requires U-Boot API <code>rsu_max_retry</code> or U-Boot command <code>rsu_display_max_retry</code> to be called before booting Linux. Otherwise, the reported value is 0, which is an invalid value.
<b>Parameters</b>	<code>value</code> : pointer to where the <code>max_retry</code> parameter is saved.
<b>Return Value</b>	0 on success, or error code

### D.6.37. rsu\_dcmf\_status

<b>Prototype</b>	<code>int rsu_dcmf_status(int *status);</code>
<b>Description</b>	Determine whether decision firmware copies are corrupted in flash, with the currently used decision firmware being used as reference. The status is an array of four values, one for each decision firmware copy. A value of 0 means the copy is fine, anything else means the copy is corrupted. Requires U-Boot API <code>rsu_dcmf_status</code> or U-Boot command <code>rsu display_dcmf_status</code> to be called first. Otherwise, all decision firmware copies are reported as not corrupted. The reported status is valid at the time the U-Boot command was called, and does not reflect any changes that may have occurred after that.
<b>Parameters</b>	status: pointer to where the status values are stored
<b>Return Value</b>	0 on success, or error code

### D.6.38. rsu\_save\_spt

<b>Prototype</b>	<code>int rsu_save_spt(char *name);</code>
<b>Description</b>	Save the current SPT to a file. The file content is protected with a CRC32 checksum.
<b>Parameters</b>	name: name of the file where SPT is saved
<b>Return Value</b>	0 on success, or error code

### D.6.39. rsu\_restore\_spt

<b>Prototype</b>	<code>int rsu_restore_spt(char *name);</code>
<b>Description</b>	Restore the SPT from a saved file. The file content is protected with a CRC32 checksum.
<b>Parameters</b>	name: name of the file where SPT is restored from.
<b>Return Value</b>	0 on success, or error code

### D.6.40. rsu\_save\_cpb

<b>Prototype</b>	<code>int rsu_save_cpb(char *name);</code>
<b>Description</b>	Save the working CPB to a file. The file content is protected with a CRC32 checksum.
<b>Parameters</b>	name: name of the file where CPB is saved
<b>Return Value</b>	0 on success, or error code

### D.6.41. rsu\_create\_empty\_cpb

<b>Prototype</b>	<code>int rsu_create_empty_cpb(void);</code>
<b>Description</b>	Create an empty CPB, which includes the CPB header only. All entries are marked as unused.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.6.42. rsu\_restore\_cpb

<b>Prototype</b>	<code>int rsu_restore_cpb(char *name);</code>
<b>Description</b>	Restore the CPB from a saved file. The file content is protected with a CRC32 checksum.
<b>Parameters</b>	name: name of the file where CPB is restored from.
<b>Return Value</b>	0 on success, or error code

### D.6.43. rsu\_running\_factory

<b>Prototype</b>	<code>int rsu_running_factory(int *factory);</code>
<b>Description</b>	Determine if current running image is factory image.
<b>Parameters</b>	factory: value at this address is set to 1 if factory image is currently running, 0 otherwise..
<b>Return Value</b>	0 on success, or error code

## D.7. RSU Client Commands

### D.7.1. count

<b>Command</b>	<code>./rsu_client -c --count</code>
<b>Description</b>	Display the number of slots
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.7.2. list

<b>Command</b>	<code>./rsu_client -l --list &lt;slot_num&gt;</code>
<b>Description</b>	List the attribute info from the selected slot
<b>Parameters</b>	<slot_num>: slot number
<b>Return Value</b>	0 on success, or error code

### D.7.3. size

<b>Command</b>	<code>./rsu_client -z --size &lt;slot_num&gt;</code>
<b>Description</b>	Display the slot size in bytes
<b>Parameters</b>	<slot_num>: slot number
<b>Return Value</b>	0 on success, or error code

### D.7.4. priority

<b>Command</b>	<code>./rsu_client -p --priority &lt;slot_num&gt;</code>
<b>Description</b>	Display the priority of the selected slot
<b>Parameters</b>	<slot_num>: slot number
<b>Return Value</b>	0 on success, or error code

### D.7.5. enable

<b>Command</b>	<code>./rsu_client -E --enable &lt;slot_num&gt;</code>
<b>Description</b>	Set the selected slot as the highest priority
<b>Parameters</b>	<slot_num>: slot number
<b>Return Value</b>	0 on success, or error code

### D.7.6. disable

<b>Command</b>	<code>./rsu_client -D --disable &lt;slot_num&gt;</code>
<b>Description</b>	Disable selected slot but to not erase it
<b>Parameters</b>	<slot_num>: slot number
<b>Return Value</b>	0 on success, or error code

### D.7.7. request

<b>Command</b>	<code>./rsu_client -r --request &lt;slot_num&gt;</code>
<b>Description</b>	Request the selected slot to be loaded after the next Linux <code>reboot</code> command which would otherwise trigger an HPS cold reset. Unless the Linux <code>reboot</code> command is issued, this function has no effect. If the Linux <code>reboot</code> command is configured to trigger a warm reset by passing the <code>reboot=warm</code> parameter to the kernel, the function also has no effect.
<b>Parameters</b>	<slot_num>: slot number
<b>Return Value</b>	0 on success, or error code

### D.7.8. request-factory

<b>Command</b>	<code>./rsu_client -R --request-factory</code>
<b>Description</b>	Request the factory image to be loaded after the next Linux <code>reboot</code> command which would otherwise trigger an HPS cold reset. Unless the Linux <code>reboot</code> command is issued, this function has no effect. If the Linux <code>reboot</code> command is configured to trigger a warm reset by passing the <code>reboot=warm</code> parameter to the kernel, the function also has no effect.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.7.9. erase

<b>Command</b>	<code>./rsu_client -e --erase &lt;slot_num&gt;</code>
<b>Description</b>	Erase application image from the selected slot
<b>Parameters</b>	<code>&lt;slot_num&gt;</code> : slot number
<b>Return Value</b>	0 on success, or error code

### D.7.10. add

<b>Command</b>	<code>./rsu_client -a --add &lt;file_name&gt; -s --slot &lt;slot_num&gt;</code>
<b>Description</b>	Add a new application image to the selected slot, and make it the highest priority. The slot must be erased first.
<b>Parameters</b>	<code>&lt;file_name&gt;</code> : file name <code>&lt;slot_num&gt;</code> : slot number
<b>Return Value</b>	0 on success, or error code

### D.7.11. add-factory-update

<b>Command</b>	<code>./rsu_client -u --add-factory-update &lt;file_name&gt; -s --slot &lt;slot_num&gt;</code>
<b>Description</b>	Add a new factory update image to the selected slot, and make it the highest priority. The command also works with Decision Firmware update images. The slot must be erased first.
<b>Parameters</b>	<code>&lt;file_name&gt;</code> : file name <code>&lt;slot_num&gt;</code> : slot number
<b>Return Value</b>	0 on success, or error code

### D.7.12. add-raw

<b>Command</b>	<code>./rsu_client -A --add-raw &lt;file_name&gt; -s --slot &lt;slot_num&gt;</code>
<b>Description</b>	Add a new raw image to the selected slot. The slot must be erased first. The slot is not added to CPB.
<b>Parameters</b>	<code>&lt;file_name&gt;</code> : file name <code>&lt;slot_num&gt;</code> : slot number
<b>Return Value</b>	0 on success, or error code

### D.7.13. verify

<b>Command</b>	<code>./rsu_client -v --verify &lt;file_name&gt; -s --slot &lt;slot_num&gt;</code>
<b>Description</b>	Verify application image on the selected slot
<b>Parameters</b>	<code>&lt;file_name&gt;</code> : file name <code>&lt;slot_num&gt;</code> : slot number
<b>Return Value</b>	0 on success, or error code

## D.7.14. verify-raw

<b>Command</b>	<code>./rsu_client -V --verify-raw &lt;file_name&gt; -s --slot &lt;slot_num&gt;</code>
<b>Description</b>	Verify raw image on the selected slot
<b>Parameters</b>	<file_name>: file name <slot_num>: slot number
<b>Return Value</b>	0 on success, or error code

## D.7.15. copy

<b>Command</b>	<code>./rsu_client -f --copy &lt;file_name&gt; -s --slot &lt;slot_num&gt;</code>
<b>Description</b>	Read the data in a selected slot then write to a file
<b>Parameters</b>	<file_name>: file name <slot_num>: slot number
<b>Return Value</b>	0 on success, or error code

## D.7.16. log

<b>Command</b>	<code>./rsu_client -g --log</code>
<b>Description</b>	Print the status log
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

## D.7.17. notify

<b>Command</b>	<code>./rsu_client -n --notify &lt;value&gt;</code>
<b>Description</b>	Report HPS software state as a 16-bit numerical value
<b>Parameters</b>	<value>: HPS software state as a 16-bit numerical value
<b>Return Value</b>	0 on success, or error code

## D.7.18. clear-error-status

<b>Command</b>	<code>./rsu_client -C --clear-error-status</code>
<b>Description</b>	Clear the sticky errors from RSU status.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.7.19. reset-retry-counter

<b>Command</b>	<code>./rsu_client -Z --reset-retry-counter</code>
<b>Description</b>	Reset the retry counter. This allows the current image to be tried again in case of a failure, if the <code>max_retry</code> parameter was set to a value greater than one.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.7.20. display-dcmf-version

<b>Command</b>	<code>./rsu_client -m --display-dcmf-version</code>
<b>Description</b>	Display the four decision firmware versions. Requires U-Boot API <code>rsu_dcmf_version</code> or U-Boot command <code>rsu display_dcmf_version</code> to be called before booting Linux, otherwise version is reported as 0.0.0 The reported versions are the ones valid at the time the U-Boot command was called, and do not reflect any changes that may occur after that.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

**Note:** Refer to *Decision Firmware Version Information* section for details about the content of the version information.

### D.7.21. display-dcmf-status

<b>Command</b>	<code>./rsu_client -y --display-dcmf-status</code>
<b>Description</b>	Determine whether decision firmware copies are corrupted in flash, with the currently used decision firmware being used as reference. The status is an array of four values, one for each decision firmware copy. A value of 0 means the copy is fine, anything else means the copy is corrupted. Requires U-Boot API <code>rsu_dcmf_status</code> or U-Boot command <code>rsu display_dcmf_status</code> to be called first. Otherwise all decision firmware copies is reported as not corrupted. The reported status is valid at the time the U-Boot command was called, and does not reflect any changes that may have occurred after that.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.7.22. display-max-retry

<b>Command</b>	<code>./rsu_client -x --display-max-retry</code>
<b>Description</b>	Displays the value of the <code>max_retry</code> parameter from flash. Requires U-Boot API <code>rsu_max_retry</code> or U-Boot command <code>rsu display_max_retry</code> to be called before booting Linux. Otherwise, the reported value is 0, which is an invalid value.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.7.23. create-slot

<b>Command</b>	<code>./rsu_client -t --create-slot &lt;slot_name&gt; -S --address &lt;slot_address&gt; -L --length &lt;slot_size&gt;</code>
<b>Description</b>	Create a new slot in the SPT, using unused space.
<b>Parameters</b>	<slot_name>: new slot name <slot_address>: address in flash for new slot <slot_size>: new slot size in bytes
<b>Return Value</b>	0 on success, or error code

### D.7.24. delete-slot

<b>Command</b>	<code>./rsu_client -d --delete-slot &lt;slot_num&gt;</code>
<b>Description</b>	Delete selected slot, freeing up allocated space.
<b>Parameters</b>	<slot_num>: slot number to be deleted
<b>Return Value</b>	0 on success, or error code

### D.7.25. restore-spt

<b>Command</b>	<code>./rsu_client -W --restore-spt &lt;file_name&gt;</code>
<b>Description</b>	Restore the SPT from a saved file. The file content is protected with a CRC32 checksum.
<b>Parameters</b>	<file_name>: name of the file where SPT is restored from.
<b>Return Value</b>	0 on success, or error code

### D.7.26. save-spt

<b>Command</b>	<code>./rsu_client -X --save-spt &lt;file_name&gt;</code>
<b>Description</b>	Save the current SPT to a file. The file content is protected with a CRC32 checksum.
<b>Parameters</b>	<file_name>: name of the file where SPT is saved.
<b>Return Value</b>	0 on success, or error code

### D.7.27. create-empty-cpb

<b>Command</b>	<code>./rsu_client -b --create-empty-cpb</code>
<b>Description</b>	Create an empty CPB, which includes the CPB header only. All entries are marked as unused.
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code



### D.7.28. restore-cpb

<b>Command</b>	<code>./rsu_client -B --restore-cpb &lt;file_name&gt;</code>
<b>Description</b>	Restore the CPB from a saved file. The file content is protected with a CRC32 checksum.
<b>Parameters</b>	<code>&lt;file_name&gt;</code> : name of the file where CPB is restored from.
<b>Return Value</b>	0 on success, or error code

### D.7.29. save-cpb

<b>Command</b>	<code>./rsu_client -P --save-cpb &lt;file_name&gt;</code>
<b>Description</b>	Save the working CPB to a file. The file content is protected with a CRC32 checksum.
<b>Parameters</b>	<code>&lt;file_name&gt;</code> : name of the file where CPB is saved.
<b>Return Value</b>	0 on success, or error code

### D.7.30. check-running-factory

<b>Command</b>	<code>./rsu_client -k --check-running-factory</code>
<b>Description</b>	Check if currently running image is the factory image
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

### D.7.31. help

<b>Command</b>	<code>./rsu_client -h --help</code>
<b>Description</b>	Show usage message
<b>Parameters</b>	None
<b>Return Value</b>	0 on success, or error code

## E. Combined Application Images

---

A combined application image is an application image which is able to update the decision firmware and decision firmware data in flash while it is being loaded. The update only happens if the combined image was created with a newer version of the Intel Quartus Prime software than the decision firmware that is currently used in the system. Intel Quartus Prime software patches have the same version as the Intel Quartus Prime software product they patch, so they are not detected as newer.

*Note:* Combined application image support was added in Intel Quartus Prime software version 20.4, to enable a very specific use case where support for deploying and running a factory update image or a decision firmware update image was not deployed in the system. The combined application image enabled deploying new features in the decision firmware (like the `max_retry` option) by just deploying this special type of application image.

*Important:* You must provide support for updating the factory image, by having the ability to download the factory update image, write it to flash, and pass control to it. If this is done, you do not need to use the combined application image feature.

The combined application image is up less than 512 KB larger than the regular application image created with the same options and using the same SOF file.

The combined application firmware contains the following components: new decision firmware, new decision firmware data, application image section, and specialized firmware which performs the version checking and the conditional update.

The combined application image always checks the decision firmware version at load time, by looking at the decision firmware copy that was last used successfully. In the cases where it does not need to perform the update, the increase in load time is negligible. In case the update needs to be performed, this adds a few seconds to the load time.

The factory update flow is resilient to power loss. If the power is lost during the update, the next time the power is back up, the factory update image resumes the update process from where it stopped.

The combined application image can be written to flash with both the APIs designed to write application images, and with the APIs and commands designed to write factory update images and decision firmware update images to flash. It achieves this by not having internal pointers that need to be relocated, so it can be written to and loaded from any address in flash.

## E.1. Creating the Combined Application Image

The following shows the commands that can be used to create a combined application image for the FPGA first example presented in this document.

```
cd $TOP_FOLDER
mkdir -p images
rm -f images/combined_application.rpd
~/intelFPGA_pro/21.2/nios2eds/nios2_command_shell.sh \
quartus_pfg -c hw/ghrd.3/output_files/ghrd_agfb014r24a3e3vr0.sof \
  images/combined_application.rpd \
  -o app_image=hw/ghrd.2/output_files/ghrd_agfb014r24a3e3vr0.sof \
  -o hps_path=u-boot-socfpga/spl/u-boot-spl-dtb.hex \
  -o app_image_hps_path=u-boot-socfpga/spl/u-boot-spl-dtb.hex \
  -o mode=ASX4 -o start_address=0x00000 -o bitswap=ON \
  -o rsu_upgrade=ON \
  -o firmware_only=1
```

The following decision firmware update image is created: `images/combined_application.rpd`.

### Notes:

- The first SOF file contains the factory image, from which data is taken to fill out the new decision firmware data structure. This includes QSPI clock and pin settings, the value of `max_retry` parameter, and the selected behavior of the HPS watchdog. The actual configuration data from the SOF file is not used.
- The `app_image` parameter contains the SOF that is used for the application image section of the combined image.
- The `hps_path` parameter is unused, and may be removed in the future.
- The `app_image_hps_path` parameter contains the HPS FSBL hex file to be used for the application image section of the combined image.

When using HPS first, the additional parameter "`-o hps=1`" needs to be added, and the following files are created:

- `images/combined_application.hps.rpd` - Combined application image
- `images/combined_application.core.rbf` - Corresponding fabric configuration file

## E.2. Using the Combined Application Image

This section shows an example of using a combined application image, from U-Boot. Similar commands can be used from Linux.

The combined application images are used the exact same way as regular application images, just that they first update the decision firmware and decision firmware data if necessary, before the device is configured with the functionality from the application SOF.

**Note:** The combined application images do not have absolute pointers inside like the regular application images, the factory update images, or the decision firmware update images. Because of this, they can be written to flash with any of the U-Boot and LibRSU APIs that write to slots.

1. Create an initial flash image using the instructions from this document, but use an older version of Quartus, for example 20.2.
2. Build the combined application image as described in the previous section, using Intel Quartus Prime Pro Edition version 21.2. Put the combined application image on the SD card, on the FAT partition.
3. Boot the system created with Quartus 20.2 and query the decision firmware information from U-Boot:

```
SOCFPGA # rsu display_dcmf_version
DCMF0 version = 20.2.0
DCMF1 version = 20.2.0
DCMF2 version = 20.2.0
DCMF3 version = 20.2.0
```

4. Find an unused slot, erase it, write the combined application image to it, and check it is now the highest priority:

```
SOCFPGA # rsu slot_erase 2
Slot 2 erased.
SOCFPGA # fatload mmc 0:1 ${loadaddr} combined_image.rpd
151552 bytes read in 9 ms (16.1 MiB/s)
SOCFPGA # rsu slot_program_buf 2 ${loadaddr} ${filesize}
Slot 2 was programmed with buffer=0x0000000002000000 size=3510272.
SOCFPGA # rsu slot_verify_buf 2 ${loadaddr} ${filesize}
Slot 2 was verified with buffer=0x0000000002000000 size=3510272.
SOCFPGA # rsu slot_get_info 2
NAME: P3
OFFSET: 0x0000000003000000
SIZE: 0x01000000
PRIORITY: 1
```

5. Pass control to the combined application update image:

```
SOCFPGA # rsu slot_load 2
```

6. The combined application image checks the currently used decision firmware copy, it sees that it is older, then it updates the decision firmware and decision firmware data, then it loads the actual application image section. Everything takes a few seconds.
7. Stop at U-Boot prompt and confirm the decision firmware is updated, and the application image is running fine:

```
SOCFPGA # rsu status_log
Current Image : 0x03000000
Last Fail Image : 0x00000000
State : 0x00000000
Version : 0x00000202
Error location : 0x00000000
Error details : 0x00000000
Retry counter : 0x00000000
SOCFPGA # rsu display_dcmf_version
DCMF0 version = 21.2.0
DCMF1 version = 21.2.0
DCMF2 version = 21.2.0
DCMF3 version = 21.2.0
```

8. Power cycle the board, the same combined application image is loaded, as it is the highest priority. But it takes a couple of seconds less, as the decision firmware does not need to be updated.

## 15. Document Revision History for the SoC Remote System Update User Guide

---

Document Version	Changes
2021.09.03	Updated a note in the <i>Component Interfaces</i> section that explains that using a newer firmware is permitted
2021.08.04	<ul style="list-style-type: none"> <li>Added a note about configuring the QSPI erase granularity in order to successfully update: <ul style="list-style-type: none"> <li>SPTs in <i>Sub-Partition Table Layout (S10/FM)</i></li> <li>CPBs in <i>Modifying the List of Application Images and Configuration Pointer Block Layout (S10/FM)</i></li> </ul> </li> <li>Updated the Linux kernel version to 5.4.114-lts.</li> <li>Added example maintenance procedures to the <i>Flash Corruption - Detection and Recovery</i> sections: <ul style="list-style-type: none"> <li><i>Maintenance Procedure</i></li> <li><i>Recover Decision Firmware Procedure</i></li> </ul> </li> <li>Added all the new features and scenarios that were introduced in the last year, to match the Intel Stratix 10 version.</li> </ul>
2020.07.10	Initial release