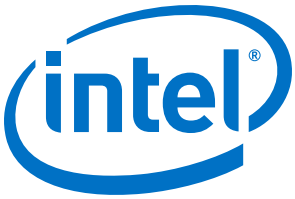


Intel® Stratix® 10 SoC Remote System Update (RSU) User Guide

Updated for Intel® Quartus® Prime Design Suite: **18.1**

UG-20197 | 2019.02.27
Latest document on the web: PDF | HTML

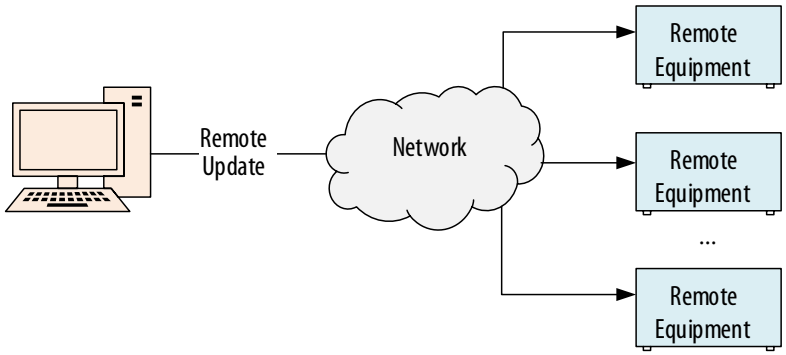
Contents

1. Overview

Remote System Update (RSU) allows you to reliably update the QSPI configuration bitstream of an Intel® Stratix® 10 SoC device with extremely low risk of corrupting the bitstream storage and rendering the system non-functional.

If the configuration bitstream in QSPI flash is corrupted, the only method to recover the device would be to connect to it 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



The RSU solution offered for Intel Stratix 10 SoC focuses on updating the configuration bitstream once a new version is available on the target equipment. It is your responsibility to deploy the image over network to the target remote equipment.

This document details the Intel Stratix 10 HPS remote system update solution and provides an update example using the *Intel Stratix 10 SX SoC Development Kit*.

All Intel Stratix 10 devices also support a RSU procedure which is driven from the FPGA fabric. For information about the FPGA-fabric driven RSU flow, refer to the *Intel Stratix 10 Configuration User Guide*.

Related Information

- Intel Stratix 10 Configuration User Guide
- Intel Stratix 10 SX SoC Development Kit User Guide

1.1. Features

The remote system update solution:

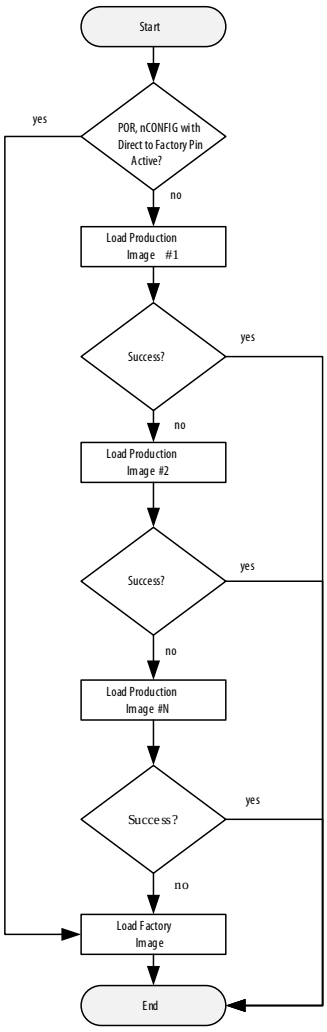
- Provides support for creating the initial flash image for a system to support RSU.
- Allows a number of production images to be tried in a specific order until one of them is successful. Success is defined as SDM being able to configure FPGA (or just the HPS EMIF I/O in case of HPS first), load FSBL and start it on HPS. Optionally an HPS watchdog timeout can be treated as a RSU failure too.

  Note: The number of production images can be greater than 100, but in reality it is limited by the flash size. The number can easily reach up to several dozens with large flash devices and images using HPS first, which are very small. A maximum of three production images can be specified at image creation time, but more can be added later.
- Loads a factory image if no production image is available, or all production images failed.
- Provides you the option to select a pin and define its polarity so that when it is asserted during a configuration caused by a POR or nCONFIG event, it forces the SDM to load the factory image instead of the highest priority production image.
- Provides you with the ability to add and remove production images.
- Provides you with the ability to change the order in which production images are loaded.
- Provides you with the ability to load a specific image from flash. The image is a production or factory image.
- Provides you with information on which image is currently running, and what errors were encountered by RSU.
- Provides you with an API to notify SDM of the state of the HPS software as a numeric value. The state is reported back after the next image is successfully loaded in case the previous image failed due to an HPS watchdog timeout.

The factory and production images are also called bitstreams and they typically contain the FPGA fabric configuration, the SDM firmware and the HPS FSBL (First Stage Bootloader). For HPS first mode, the FPGA fabric configuration is omitted.

The following figure presents the image selection flow that occurs when the device with RSU enabled is configured as a result of a power-up or nCONFIG event.

Figure 2. RSU Image Selection Flow



The HPS watchdog timeouts can be optionally configured to be treated as configuration failures. When that happens, the algorithm above behaves as if the image failed to configure, and it moves to the next image. If the factory image fails to configure, then there is nothing else to try and both the FPGA and HPS are wiped and remain unconfigured.

For more information, refer to "Remote System Update Watchdog" and "Enabling HPS Watchdog to Trigger RSU".

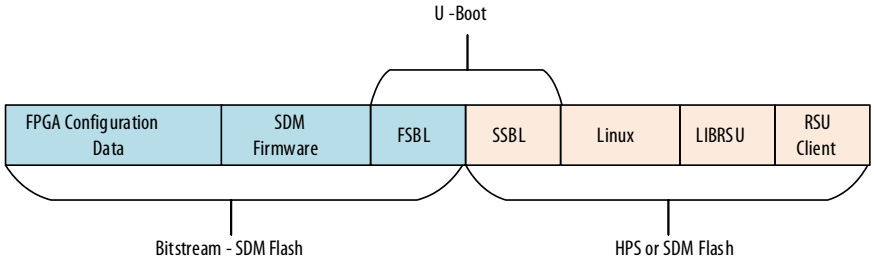
Related Information

- Intel Stratix 10 Configuration User Guide
- Remote System Update Watchdog on page 12
- Enabling HPS Watchdog to Trigger RSU on page 22

1.2. System Components

The following figure presents the typical components of an Intel Stratix 10 SoC based system using RSU.

Figure 3. 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 only the HPS EMIF I/O configuration data.
SDM Firmware	Firmware for the Secure Device Manager: Implements the RSU flow. Provides APIs for querying RSU status, initiating an RSU reload, and querying flash partitioning. These services are not directly accessible to you. Instead both U-Boot and Linux* offer ways of indirectly accessing these services.
FSBL	HPS First Stage Bootloader Initializes hardware and loads SSBL
SSBL	HPS Second Stage Bootloader Loads and boots the OS Uses SDM firmware APIs to query RSU status, initiate RSU reload, and query flash partitioning from U-Boot command line. Provides resident SMC(1) (Secure Monitor Call) handler to allow Linux to use the SDM services.
Linux Drivers	Intel RSU driver uses the APIs provided by U-Boot SMC and makes services available to applications running on Linux.
LIBRSU	Linux user space library providing an API for managing RSU.
RSU Client	Linux sample application which uses LIBRSU for managing RSU.

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

The RSU solution provided by Intel 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 53

1.3. Glossary

Acronym	Description
API	Application Programming Interface
CMF	Control Monitor Firmware
FPGA	Field Programmable Gate Array
FSBL	First Stage Bootloader
HPS	Hard Processor System
`*.jic`	JTAG Indirect Configuration File
JTAG	Joint Test Action Group
LIBRSU	Remote System Update Linux Library
MTD	Memory Technology Device
POR	Power On Reset
QSPI	Quad Serial Peripheral Interface Flash
`*.rpd`	Raw Programming File
RSU	Remote System Update
SDM	Secure Device Manager
SMC	Secure Monitor Call
SOF	SRAM Object File
SoC	System on Chip
SPT	Sub Partition Table
SSBL	Second Stage Bootloader

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 production images. A tool called "Programming File Generator" uses this input to create an image file which is used to set the initial contents of the flash. You must program the flash with this image file to prepare the device for remote system update.

The production and factory images must be suitable for initial configuration. Initial configuration assumes that no other images are pre-configured on the device. Both FPGA first and HPS first image types are supported.

2.2. Production Image Boot

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

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

When attempting to load a production image the SDM traverses the CMF 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 CMF pointer continues to increment to the next image until it reaches a successful image or all images loads fail. 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 production images have all been corrupted (or replaced with broken production images) to obtain a new production image and program that image into QSPI.

Note: Intel does not provide a software solution for obtaining the new product image and programming the image into QSPI, only the capability to load the factory image, when all production images fail to load.

The SDM loads the factory image in two 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 production images fail.

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

2.4. Modifying the List of Production Images

The SDM uses the CMF pointer block to determine the order in which the production images need to be tried.

The CMF pointer block management relies on the following QSPI memory characteristics:

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

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

- All 1’s – means the entry is unused, and a pointer could be written to it. This is the state after a QSPI erase operation on the CMF pointer block.
- All 0’s – means the entry was previously used, but then was cancelled, for example due to erasing a production image.
- A combination of 1's and 0's – a valid pointer to a production image.

Initially when a CMF pointer block is erased, all entries are marked as unused. Adding a production image to the list consists in finding the first unused location and writing the production image address to it. Removing a production image from the list consists in finding it in the CMF pointer block list and overwriting it as all 0s.

After a while the CMF pointer block may become filled with cancelled entries, with no room to add new production image pointers. In such a case, LIBRSU erases the CMF pointer block, copies over all previously valid entries, and adds the new image. This procedure is called CMF pointer block compression. CMF pointer block compression does not occur often because the CMF pointer block has approximately 500 available entries.

There are two CMF pointer blocks: a primary (CPB0) and a backup (CPB1). Two pointers enable the list of production images to be protected when a power failure happens just after erasing a CMF pointer block. For more information, refer to the "Pointer Block Layout on page 19" section.

The only RSU component which updates the list of production images is LIBRSU. The rest of the components only read the CMF pointer blocks.

When the client writes the production image to flash it must ensure that the pointers within the main image pointer of its first signature block are updated to point to the correct locations in flash. For more information, refer to the "Production Image Layout on page 20" section. This functionality is already implemented in LIBRSU APIs which write images to flash.

2.5. Querying Remote System Update Status

The SDM firmware offers a command for querying which image is currently running and the failing reason for the primary image. This functionality is available in the U-Boot command line and also in LIBRSU.

2.6. Loading a Specific Image

The SDM firmware provides a command that loads a specific image from flash. The image can be the factory image or one of the production images. You have access to this functionality from the U-Boot command line and also from LIBRSU.

When this command is issued, the SDM immediately resets and wipes both the FPGA and HPS, then it proceeds to load the specified image. When the equivalent API is called from LIBRSU, the command is not immediately sent to SDM as this would cause Linux to crash. Instead, the U-Boot SMC handler makes a note to call the command on the next cold reset request from the HPS. This way the SDM command only gets issued when the Linux "reboot" command is ran, which causes the kernel to shut down cleanly and the HPS cold reset request to be issued.

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

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

For more information, refer to the "RSU Status and Error Codes" section.

Related Information

RSU Status and Error Codes on page 62

2.7. Protected Access to Flash

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

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

Another potential measure to protect the flash may be to use QSPI vendor specific commands to mark the Decision CMF and factory image areas as read-only. However, that may result in a more complex procedure for updating the Decision CMF and Factory Image areas.

2.8. Remote System Update Watchdog

The HPS watchdogs can be used to trigger a reset in case they are 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 FSBL from the current image again.
- Trigger a warm reset – the SDM causes the HPS to re-start the HPS FSBL, without reloading it.
- Trigger a cold reset and a remote system update – the SDM considers the last loaded production image a failure, and loads the next one in the CMF pointer block list, or 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.

Currently, the FSBL automatically enables the watchdog as one of the first things it does, and it services the watchdog periodically. Then the SSBL disables the watchdog as one of the very first things it does.

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

Note: The current image is not removed from the CPB when a watchdog timeout occurs. That is, 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 on page 21

2.9. RSU Notify

The SDM offers a command called RSU Notify which is used by FSBL and SSBL to let SDM know the current HPS software state as a 16-bit numerical value.

In case of an RSU failure triggered by an HPS watchdog timeout, the next FPGA production (or factory) image is loaded. When HPS queries the RSU state, it sees that the top 16 bits read as `0xf006`. The bottom 16 bits of the RSU state contain the RSU Notify value reported to the SDM.

The RSU Notify is called with a value of "1" from FSBL just before SSBL is entered, and with a value of "2" from SSBL just before the control is passed to the operating system. The possible notify values reported in case of watchdog timeout are presented in the following table:

Value	Description
0	FSBL was either not run or did not reach the point of launching SSBL.
1	FSBL was run, but SSBL was either not run or did not reach the point of launching the operating system.
2	Both FSBL and SSBL were ran and the operating system launch was attempted.

Note: Support added in a future release enabling the RSU Notify functionality available from the U-Boot command line, Linux driver, LIBRSU, and RSU client.

2.10. Updating the Decision CMF And Factory Image

Updating of the Decision CMF and factory image is not supported in Intel Quartus Prime Pro Edition version 18.1.

In order to make your system ready to support updating the Decision CMF and factory images you need to have an available area in flash of the size of the maximum factory image you anticipate to use plus 512KB.

You can temporarily use a production image slot for the update procedure, as they are typically much larger than factory images. However, that means one less production image slot is available during the Decision CMF and factory image update procedure.

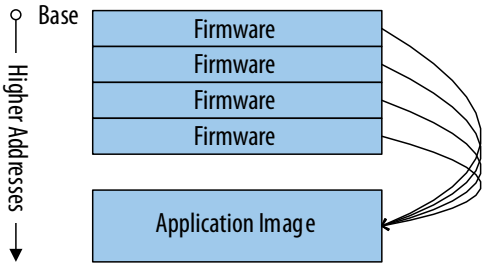
3. QSPI Flash Layout

3.1. High Level Flash Layout

3.1.1. Standard (non-RSU) Flash Layout

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 highest priority application image in flash. Typically the application image is immediately after the four firmware copies, but the Intel Quartus Prime Pro Edition tools do not require this location.

Figure 4. Flash Layout - Non-RSU

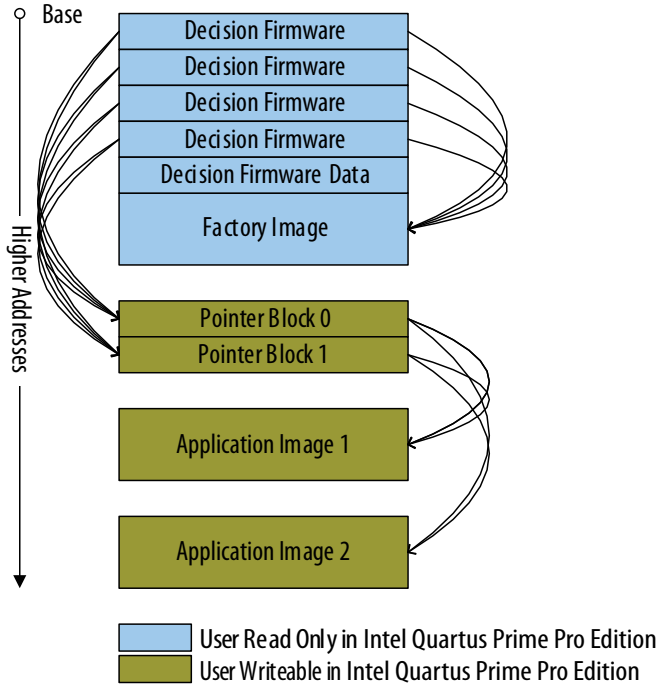


3.1.2. RSU Flash Layout – SDM Perspective

In the RSU case, the CMFs are replaced with Decision CMFs. The Decision CMFs have pointers to the following structures in flash:

- Decision CMF Data
- One Factory Image
- Two CMF Pointer Blocks (CPBs)

Figure 5. RSU Flash Layout - SDM Perspective



The Decision CMF Data is used for storing basic settings, such as the clock and pins to be used for QSPI, and which pin to be used for forcing SDM to load the factory image.

The CMF Pointer Blocks contain a list of production images to be tried until one of them is successful. If none are successful, then the factory image is loaded. The CMF pointer block has a main and a backup copy, to ensure reliability in case an update operation fails at any point.

Both the factory image and the production images start with a CMF. This CMF is loaded by the DCMF as the first step in loading an image, and then the CMF loads the rest of the image. This is an implementation detail and is not shown in the above picture in order to keep is simple. For more information, refer to the "Production Image Layout" section.

Related Information

Production Image Layout on page 20

3.1.3. RSU Flash Layout – Your Perspective

In order to facilitate the QSPI flash management, there is an additional data structure called SPT (Sub-partition Table). This table is not used by SDM, except for reporting to HPS its position in flash.

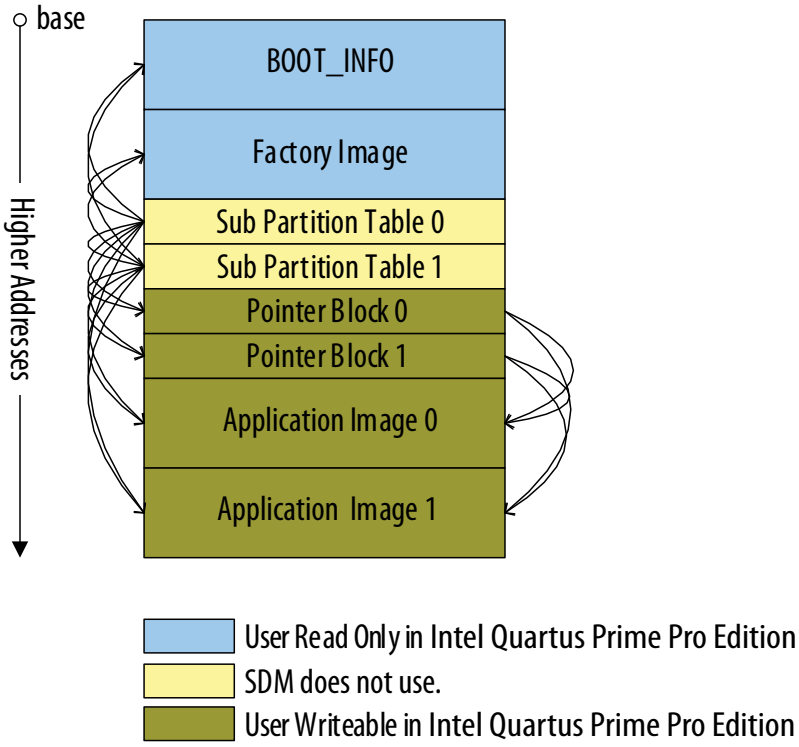
The SPT is created by the Quartus Programming File Generator tool when the initial manufacturing image is created. There are two copies called SPT0 and SPT1, to ensure reliable operation in case of power failure at any point during modifications of the SPT.

The SPT typically contains the following partitions:

Table 2. Typical Sub-Partitions

Sub-partition Name	Contents
BOOT_INFO	Decision CMFs and Decision CMF data
FACTORY_IMAGE	Factory Image
SPT0	Sub-partition Table 0
SPT1	Sub-partition Table 1
CPB0	CMF Pointer Block 0
CPB1	CMF Pointer Block 1
P1 (you enter)	Production Image 1
P2 (you enter)	Production Image 2

Figure 6. RSU Flash Layout - Your Perspective



Summarizing, your view of Flash is different from SDM view in two ways:

- You do not need to know details about where the Decision CMFs and Decision CMF data and factory image are located, since they are read-only in the Intel Quartus Prime Pro Edition version 18.1 release of the tools.
- You have access to the SPTs, which enables you to get access to the data structures required to perform the remote system update process.

3.2. Detailed Quad SPI Flash Layout

3.2.1. RSU Sub-Partitions Layout

The *Flash Sub-Partitions Layout* table shows the layout of RSU flash images.

Table 3. Flash Sub-Partitions Layout

Flash Offset	Size (in bytes)	Contents	Sub-Partition Name
0k	256k	Decision firmware	BOOT_INFO
256k	256k	Decision firmware	
512k	256k	Decision firmware	
768k	256k	Decision firmware	
1M	8k + 24 pad	Decision firmware data	
1M+32k	32k	Reserved for SDM	
1M+64k	varies	Factory image	FACTORY_IMAGE
Next	4k + 28k pad	Sub-partition table (copy 0)	SPT0
Next	4k + 28k pad	Sub-partition table (copy 1)	SPT1
Next	4k + 28k pad	Pointer block (copy 0)	CPB0
Next	4k + 28k 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. When the HPS initiates the RSU, 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 three partitions for application images. There are no limitations on creating empty partitions.

3.2.2. Sub-Partition Table Layout

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

Table 4. Sub-partition Table Layout

Offset	Size (in bytes)	Description
0x000	4	Magic number 0x57713427
0x004	4	Version number (0 for this document)
0x008	4	Number of entries
0x00C	20	Reserved (write as 0)

continued...

Offset	Size (in bytes)	Description
0x020	32	Sub-partition Descriptor 1
0x040	32	Sub-partition Descriptor 2
0xFE0	32	Sub-partition Descriptor 126

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

3.2.3. Pointer Block Layout

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

Table 6. 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	Offset from primary to backup pointer block (sector size)
0x10	4	Offset to image pointers (IPTAB)
0x14	4	Number of image pointer slots (NSLOTS)
0x18	—	Reserved
IPTAB	8	First (lowest priority) image pointer slot
	8	Second (2nd lowest priority) image pointer slot
	8	…
	8	Last (highest priority) image pointer

The pointer block can contain up to 508 application image pointers. The actual number is listed as NSLOTS. A typical 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 remote system updates before the pointer block must be erased. The erase procedure is called *pointer block compression*. This procedure is safe. There are two copies of pointer block. The copies are in different flash erase sectors. While one copy is being updated the other copy is still valid.

3.2.4. Production Image Layout

The production image is composed of a CMF immediately followed by the configuration image. The configuration image is composed of up to four sections, and the CMF contains pointers to those sections. The table below shows the location of the number of sections and the section pointers in a Production Image.

Table 7. Production 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. A couple of options are available to achieve that:

- The Production Image can be created to match the actual location in QSPI. This is not recommended, as different systems may have applied a different set of updates, so they may have different slot(s) available.
- The Production Image can be created as if it is located at address zero, then the pointers can be updated to match the actual location.

LIBRSU supports both options above, by looking at the pointers. If they are all below 1MB, then it updates them to the actual location of the slot to be updated. If not, LIBRSU checks to see whether they are actually pointing to the correct image area. If they do it leaves them untouched, if not it reports an error.

The procedure to update the pointers from a Production Image created for address INITIAL_ADDRESS to a new address called NEW_ADDRESS is as follows:

1. Create Production Image, targeting address INITIAL_ADDRESS location
2. Read the 32-bit value from offset 0xF100 of the Production 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 0xF100 + s * 8
   b. Subtract INITIAL_ADDRESS from section_pointer
   c. Add NEW_ADDRESS to section_pointer
   d. Store back updated section_pointer
4. Recompute the CRC32 for addresses 0x1000 to 0x1FFB then store the new value at offset 0x1FFC.

4. Quartus Tools Support

This section lists the different Quartus tools which are used for RSU scenarios. Refer to each tool documentation for more details.

4.1. Intel Quartus Prime Pro Edition

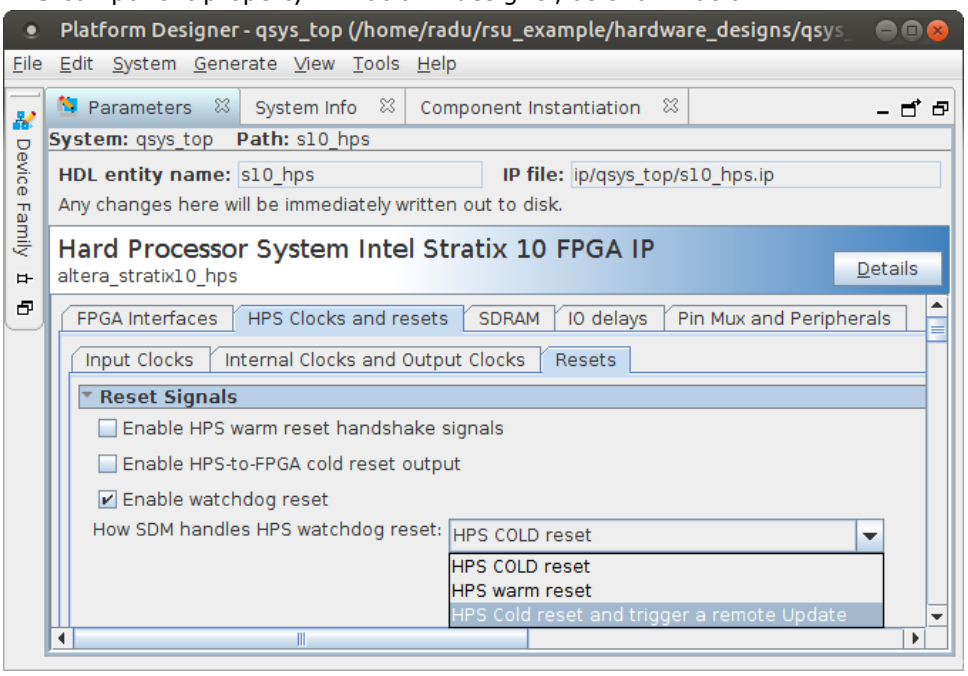
Intel Quartus Prime Pro Edition is used to compile the hardware projects to be used for the Remote System Update.

4.1.1. Selecting Factory Load Pin

The tool offers the option of selecting the pin to be used for forcing the loading of the factory application on a reset. The setting is accessible by going to **Assignments > Device > Device and Pin Options > Configuration > Configuration Pin Options**, then checking the **Direct to Factory Image** checkbox, and selecting the desired pin from the drop box.

4.1.2. Enabling HPS Watchdog to Trigger RSU

The tool also offers the option of selecting what happens when a HPS watchdog is enabled, but not serviced, 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, an HPS cold reset is issued, and SDM considers the current production image as a failure. Then SDM tries to load the next production image in the CMF pointer block list, or the factory image if all production images failed.

If the factory image is currently running, there is no other image to fall back to. In this case both FPGA and HPS are wiped clean by the SDM and remain unconfigured.

4.2. Quartus Programming File Generator

A new Quartus tool called ”Programming File Generator” was introduced in Intel Quartus Prime Pro Edition version 18.0. The tool creates programming files for Intel Stratix 10, including the Intel Stratix 10 SoC RSU initial flash images, and also update production images.

For more information, refer to the "Creating the Initial Flash Image on page 33" section.

For more information about examples on how to use the Programming File Generator tool, refer to "Creating the RSU Update Files on page 40".

For more information about the tool, refer to AN 827: Unified Tool for Generating Programming Files.

4.2.1. Programming File Generator File Types

The most important file types created by the Programming File Generator 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 Quartus Programmer tool. They contain the actual flash data, and also a flash loader, which is a small FPGA design used by the Quartus 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 RSU update file, which is then sent over a network to update the target system.
`.map`	Memory Map File	These files contain details about where the input data was placed in the output file.
`.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 contain configuration images for other types of flash than QSPI (for example parallel NOR flash).

4.2.2. Bitswap Option

The Quartus 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 (both for full flash images and RSU update files).

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

4.3. Quartus Programmer

The Quartus Programmer tool can be used to flash the initial flash image. For an example of how to use the Flash Programmer, refer to "Flashing the Initial Image to QSPI" section.

Related Information

Flashing the Initial Image to QSPI on page 44

5. Software Support

This section presents details about the software support for RSU. The information refers to the following versions of software and tools:

- Intel Quartus Prime Pro Edition version 18.1
- U-Boot from github tag: ACDS18.1_REL_S10_GSRD_PR
- Linux from github tag: ACDS18.1_REL_GSRD_PR
- LIBRSU and RSU Client from github tag: ACDS18.1_REL_GSRD_PR

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 the errors that occurred when trying to load an image which failed.
- Get the locations of SPTs.
- Load a specific image.
- Report the state of HPS software.

The SDM commands are not publicly documented. Support is offered for accessing the relevant services from U-Boot command line and LIBRSU.

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

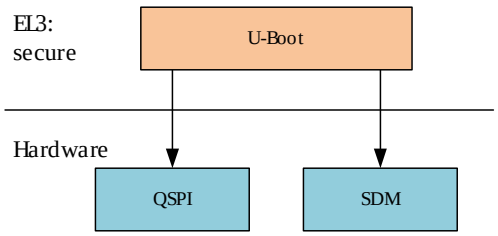
5.2. U-Boot RSU Support

U-Boot provides the following RSU-related features:

- Enables you to access relevant SDM RSU commands from U-Boot command line.
- Implements a resident SMC (secure Monitor Call) handler which is then used by Linux to interact with the relevant SDM services.

U-Boot runs at EL3, the highest execution level, so it is allowed to access the SDM APIs. U-Boot also accesses the QSPI flash to read the contents of the SPTs and CPBs.

Figure 7. U-Boot RSU



U-Boot does not implement RSU commands that would alter the flash, such as updating CPBs or writing a new production image to flash. That functionality is provided by LIBRSU. However, U-Boot does support the generic read, write, and erase flash capabilities – they are used in the example provided in the "Remote System Update Example" section to corrupt production images.

Related Information

Remote System Update Example on page 30

5.2.1. U-Boot RSU Commands

U-Boot offers the `rsu` command, with three options: `list`, `update`, and `dtb`:

```
SOCFPGA_STRATIX10 # rsu
rsu - SoCFPGA Stratix10 SoC Remote System Update

Usage:
rsu list  - List down the available bitstreams in flash
update <flash_offset> - Initiate SDM to load bitstream as specified
              by flash_offset
dtb   - Update Linux DTB qspi-boot parition offset with spt0 value
```

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_STRATIX10 # rsu list
RSU: Remote System Update Status
Current Image  : 0x01000000
Last Fail Image: 0x00000000
State          : 0x00000000
Version        : 0x00000000
Error locaton      : 0x00000000
Error details  : 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
```

The `rsu update` command is used to tell SDM to load a specific image. The following is an example of the `rsu update` command, with a portion of FSBL boot log shown immediately after running the command indicating the reboot.

```
SOCFPGA_STRATIX10 # rsu update 0x03000000
RSU: RSU update to 0x0000000003000000
SOCFPGA_STRATIX10 #
U-Boot SPL 2017.09-00103-g475f14c (October 03 2018 - 21:37:42)
MPU         1000000 kHz
L3 main     400000 kHz
Main VCO    2000000 kHz
…
```

The `rsu dtb` command is used to let U-Boot update the first QSPI partition information in the DTB so that it starts immediately after the BOOT_INFO partition. This way the DCMFs, DCMF data and 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 on page 12" 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

5.2.2. U-Boot SMC Handler

The U-Boot SMC handler runs at EL3, the highest execution level, and allows software running at lower execution levels to access services offered by SDM such as FPGA configuration and SDM APIs.

Getting the location of the SPTs is not offered as a service, since the Linux device tree ensures that the Linux view of the QSPI flash starts from the SPT location.

For the actual interface and implementation, starting with the file: `arch/arm/mach-socfpga/smc_rsu_s10.c`, refer to the "U-Boot Source Code Details" section.

Related Information

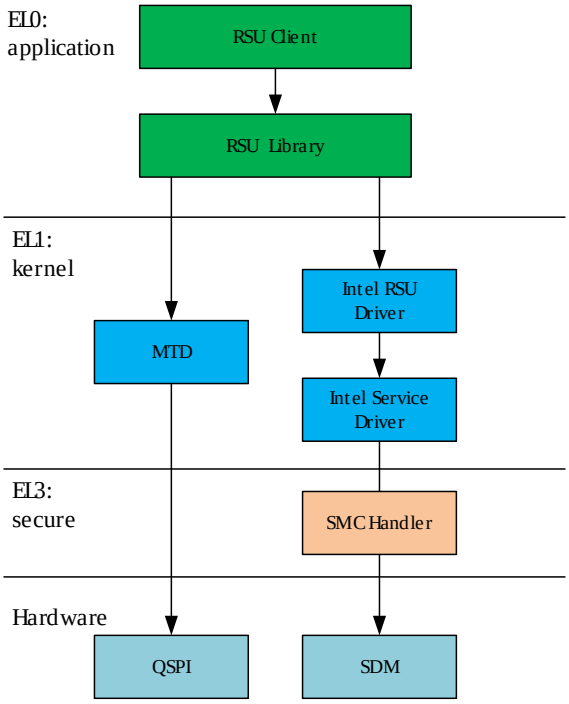
U-Boot Source Code Details on page 58

5.3. Linux RSU Support

The following RSU facilities are offered on Linux:

- Access the APIs provided by U-Boot SMC handler on top of SDM APIs.
- 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 8. Linux RSU Overview



5.3.1. Intel Service Driver

The SMC handler is left resident by U-Boot and is running at EL3, the highest execution level, therefore it can access the SDM services. 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_SERVICE=y`, which means that this driver is part of the kernel. Besides RSU, the Intel Service Driver also provides the FPGA configuration services offered by the U-Boot SMC handler.

The driver does not need to be used directly, so its interface is not documented here. For more information, refer to the source code in the `drivers/misc/intel-service.c` file for details.

5.3.2. Intel RSU Driver

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

The default kernel configuration defines `CONFIG_INTEL_RSU=m` which means it is configured as a loadable module which needs to be loaded with `insmod`.

The Linux device tree must contain the `rsu` node under the `firmware/svc` in order for the driver to be used:

```
firmware {
    svc {
        compatible = "intel,stratix10-svc";
        method = "smc";
        memory-region = <&service_reserved>;
        rsu {
            compatible = "intel,stratix10-rsu";
        };
    };
};
```

The driver offers its services through the following `sysfs` files located in the `/sys/devices/platform/soc:firmware:svc/soc:firmware:svc:rsu/` folder.

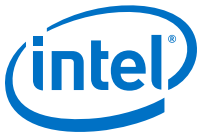
Table 8. SysFS Entries for Intel RSU Driver

File	R/W	Description
`current_image`	RO	Location of currently running image in SDM QSPI flash.
`fail_image`		Location of failed image in SDM QSPI flash.
`error_details`	RO	Opaque error code, with no meaning to users.
`error_location`	RO	Location of error is in the image that failed.
`state`	RO	State of RSU system. For more information, refer to the "RSU Status and Error Codes on page 62" section.
`version`	RO	Currently hard coded to zero. *Note:* May change in the future.
`reboot_image`	WO	Address of image to be loaded on next reboot command

All files except the `reboot_imge` are read-only. The `reboot_image` is write-only.

Example usage commands for the Intel RSU driver are shown below:

```
# cd /sys/devices/platform/soc:firmware:svc/soc:firmware:svc:rsu/
# cat current_image
16777216
# cat error_details
0
# cat error_location
0
# cat state
0
# cat version
0
# printf %i 0x03000000 > reboot_image
```



Related Information

RSU Status and Error Codes on page 62

5.3.3. 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 a production 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 CMF pointer block list.

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

Related Information

LIBRSU Reference Information on page 64

5.3.4. 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 on page 48" section.

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

6. Remote System Update Example

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

- Creating the initial flash image containing the bitstreams for a factory image, one production image, and two empty slots to contain additional production images.
- Creating an SD card with the SSBL, Linux, LIBRSU, RSU client and remote system update images to be deployed.
- Exercising the U-Boot RSU commands.
- Exercising the Intel RSU Linux driver.
- Exercising the Linux RSU example client application.

6.1. Pre-requisites

The following items are required to run the RSU example:

- Host PC running Ubuntu 14.04 LTS (other Linux versions may work too)
- Minimum 48GB of RAM, required for compiling the hardware designs
- SoC FPGA EDS Pro Edition v18.1 – for the hardware projects
- Quartus Prime Pro Edition v18.1 – for compiling the hardware projects, generating the flash images and writing to flash
- Access to Internet – to clone the git trees for U-Boot, Linux, zlib and LIBRSU and to build the Linux rootfs using Yocto.
- Stratix 10 SoC Development kit – for running the example.

6.2. Git Source Versions

The following Git tags were used to test the example presented in this chapter:

Item	Git Tree	Branch	Tag
U-Boot	https://github.com/altera-opensource/u-boot-socfpga	`origin/socfpga_v2017.09`	ACDS18.1_REL_S10_GSRD_PR
Linux Kernel	https://github.com/altera-opensource/linux-socfpga	`origin/socfpga-4.9.78-ltsi`	ACDS18.1_REL_GSRD_PR
Intel RSU	https://github.com/altera-opensource/intel-rsu	`origin/master`	ACDS18.1_REL_GSRD_PR

The tags identify the 18.1 release. The branch names are provided for reference only. For more information, refer to the "Version Compatibility Considerations".

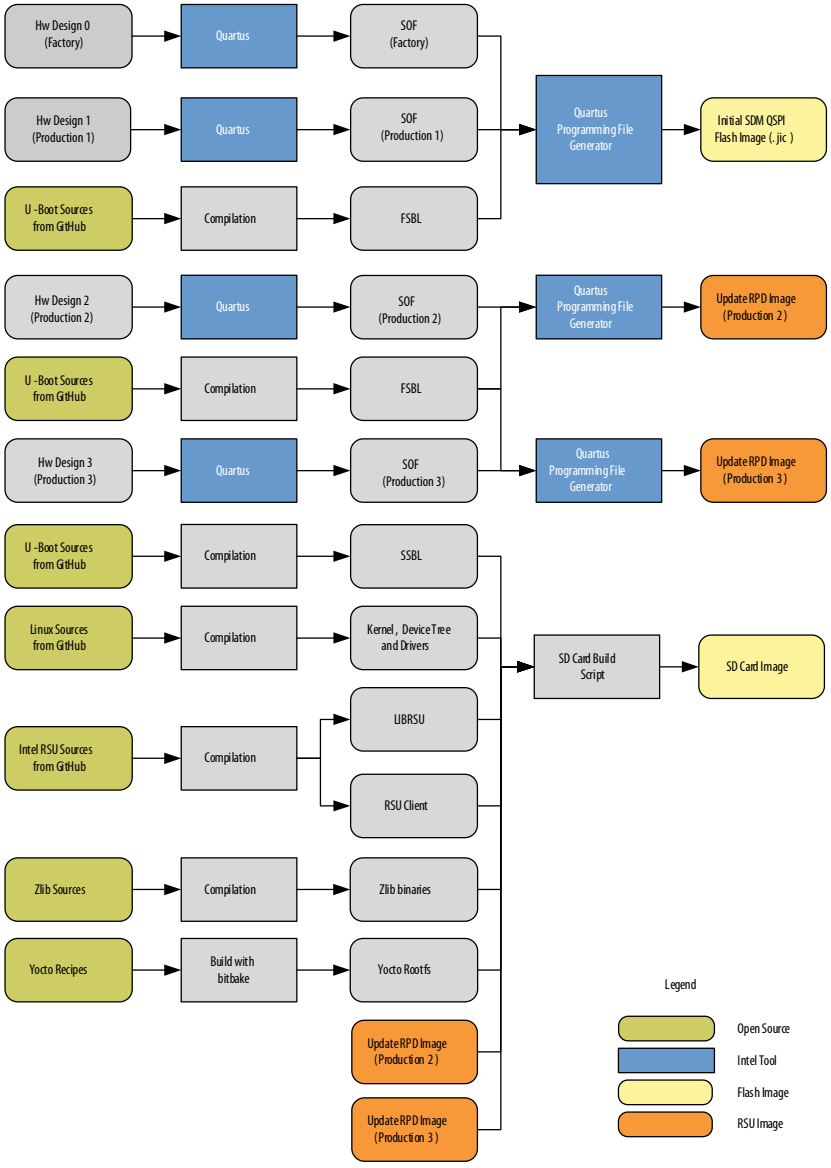
Related Information

Version Compatibility Considerations on page 53

6.3. Building RSU Example Binaries

The diagrams below illustrate the build flow used for this example. In order to simplify the illustration, it is split in three sections: building the initial SDM QSPI flash image, building the update files and building the SD card image. All three sections use U-Boot (either FSBL or SSBL) so U-Boot compilation appears in all three, although the actual compilation only happens once.

Figure 9. RSU Example Build Flow



The end results of the build flow are:

- Initial Flash Image: contains the factory image, a production image and two empty production image partitions aka slots.
- SD Card Image: contains SSBL (U-Boot), Linux device tree, Linux kernel, Linux rootfs with the Intel RSU driver, LIBRSU, RSU Client and the remote system update production images.

6.3.1. Setting up the Environment

All the commands listed in this chapter are run from the context of an Embedded Command Shell, and also use a specific toolchain from Linaro*. Follow the instructions below to set up the required environment:

```
# start an embedded command shell
~/intelFPGA_pro/18.1/embedded/embedded_command_shell.sh
# create the top folder used to store all the example files
sudo rm -rf ~/rsu_example && mkdir ~/rsu_example && cd ~/rsu_example
# retrieve and setup the toolchain
wget https://releases.linaro.org/components/toolchain/binaries/7.2-2017.11/\
aarch64-linux-gnu/gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu.tar.xz
tar xf gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu.tar.xz
export PATH=`pwd`/gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu/bin/:$PATH
# setup the environment variables used to compile Linux and LIBRSU
export ARCH=arm64
export CROSS_COMPILE=aarch64-linux-gnu-
```

6.3.2. Building the Hardware Projects

Create four different hardware projects, based on the GHRD from the Intel SoC FPGA Embedded Development Suite (SoC EDS), with a few changes:

- Target the 1SX280LU2F50E2VGS2 device, which is used by the Intel Stratix 10 Development Kit at this document's creation.
- Use a different ID in the `SystemID` component, to make the binaries for each project slightly different.
- Disable ECC, as there are some issues when using it with 18.1 release.
- Change the behavior of watchdog timeout, to issue a cold reset and trigger an RSU event.

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

```
cd ~/rsu_example
# compile hardware designs: 0-factory, 1..3-production
rm -rf hw && mkdir hw && cd hw
for version in {0..3}
do
mkdir ghrd.$version && cd ghrd.$version
tar xf $SOCEDS_DEST_ROOT/examples/hardware/s10_soc_devkit_ghrd/tgz/*.tar.gz
make clean
make scrub_clean
rm -rf *.qpf *.qsf *.txt *.bin *.qsys ip/qsys_top/ ip/subsys_jtg_mst/ ip\
/subsys_periph/
# use the device on development kit
sed -i 's/QUARTUS_DEVICE := .*/QUARTUS_DEVICE := 1SX280LU2F50E2VGS2\
/g' Makefile
# change the sysid to make projects different
sed -i 's/0xACD5CAFE/0xABAB000'$version'/g' create_ghrd_qsys.tcl
# change the behavior of the watchdog timeput to trigger a cold reset with RSU
sed -i 's/W_RESET_ACTION ./W_RESET_ACTION 2/g' construct_hps.tcl
# disable ECC as there are some issues with it in 18.1
sed -i "s/set HPS_EMIF_ECC_EN ./set HPS_EMIF_ECC_EN 0/g" design_config.tcl
# generate project
```

```
make generate_from_tcl
# build project
make sof
cd ..
done
cd ..
```

After completing the above steps, obtain the following SOF files:

- `~/rsu_example/hw/ghrd.0/output_files/ghrd_1sx280lu2f50e2vgs2.sof`
- `~/rsu_example/hw/ghrd.1/output_files/ghrd_1sx280lu2f50e2vgs2.sof`
- `~/rsu_example/hw/ghrd.2/output_files/ghrd_1sx280lu2f50e2vgs2.sof`
- `~/rsu_example/hw/ghrd.3/output_files/ghrd_1sx280lu2f50e2vgs2.sof`

6.3.3. Building U-Boot

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

```
cd ~/rsu_example
rm -rf u-boot-socfpga
git clone https://github.com/altera-opensource/u-boot-socfpga
cd u-boot-socfpga
git checkout -b test_s10_rsu ACDS18.1_REL_S10_GSRD_PR
make clean && make mrproper
make socfpga_stratix10_defconfig
make -j 24
aarch64-linux-gnu-objcopy -I binary -O ihex --change-addresses \
0xffe00000 spl/u-boot-spl-dtb.bin spl/u-boot-spl.hex
cd ..
```

After completing the above steps, obtain the following files:

- FSBL (U-boot SPL) hex file: `~/rsu_example/u-boot-socfpga/spl/u-boot-spl.hex`
- SSBL (U-Boot) image file: `~/rsu_example/u-boot-socfpga/u-boot-dtb.img`

6.3.4. Creating the Initial Flash Image

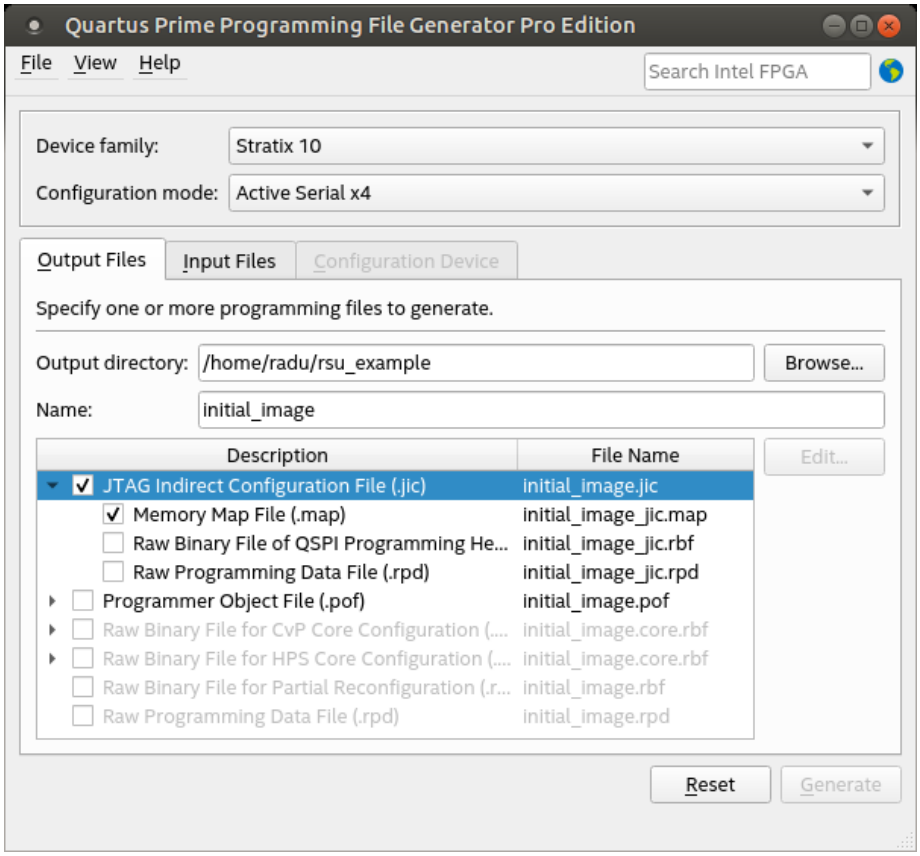
Create an initial flash image containing the factory and production bitstreams and the associated RSU data structures.

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

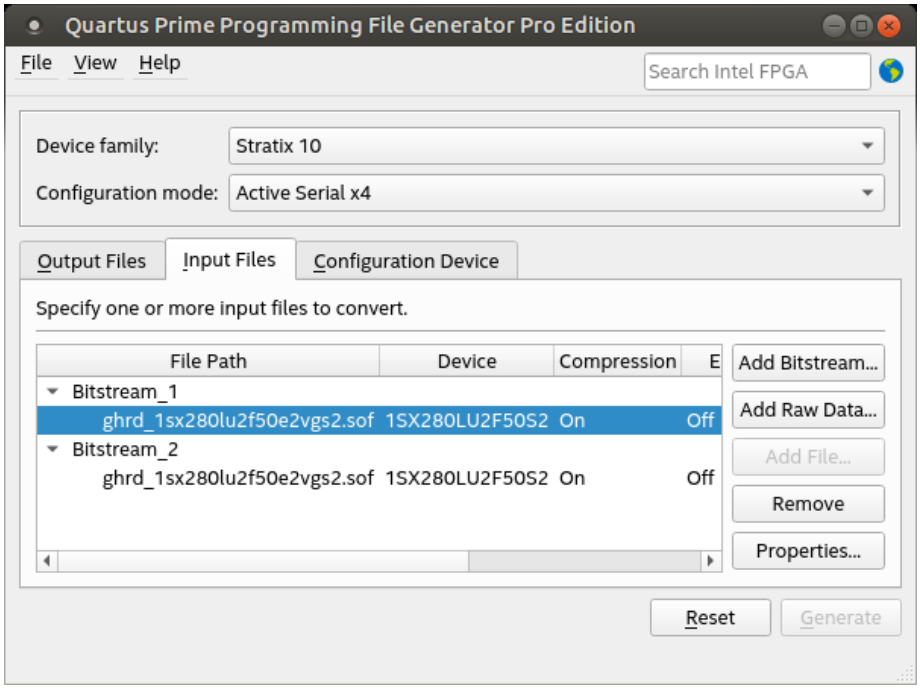
   ```
   cd ~/rsu_example
   qpfgw &
   ```

2. Select the **Device family** as **Intel Stratix 10**, 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 Quartus Programmer tool for writing to the QSPI flash.

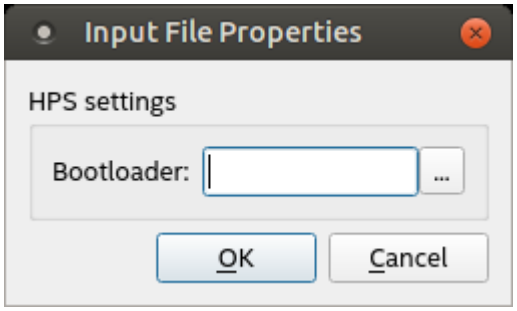
5. Select the optional **Memory Map File (.map)** file so that it is also generated. The .map file contains information about the resulted flash layout. The window looks similar to this:



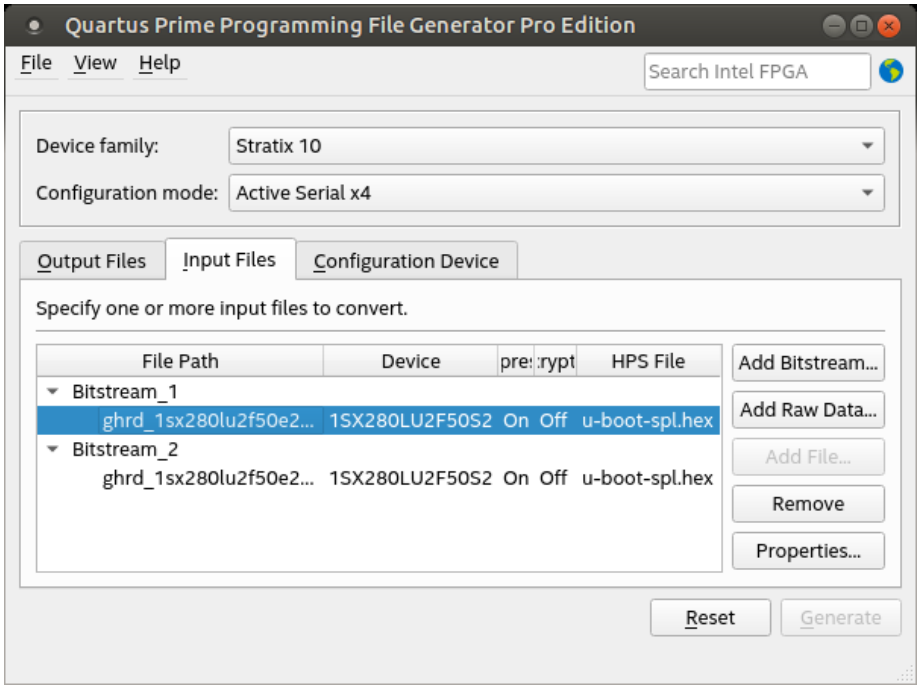
6. Once the output type was selected, click the **Input Files** tab.
7. In the **Input Files** tab click the **Add Bitstream** button, then browse to `~/rsu_example/hw/ghrd.0/output_files`, select the file `ghrd_1sx280lu2f50e2vgs2.sof`, and then click **Open**. This is the factory image. Do the same for the `~/rsu_example/hw/ghrd.1/output_files/ghrd_1sx280lu2f50e2vgs2.sof` image. This is the initial production image. The tab now looks like below:



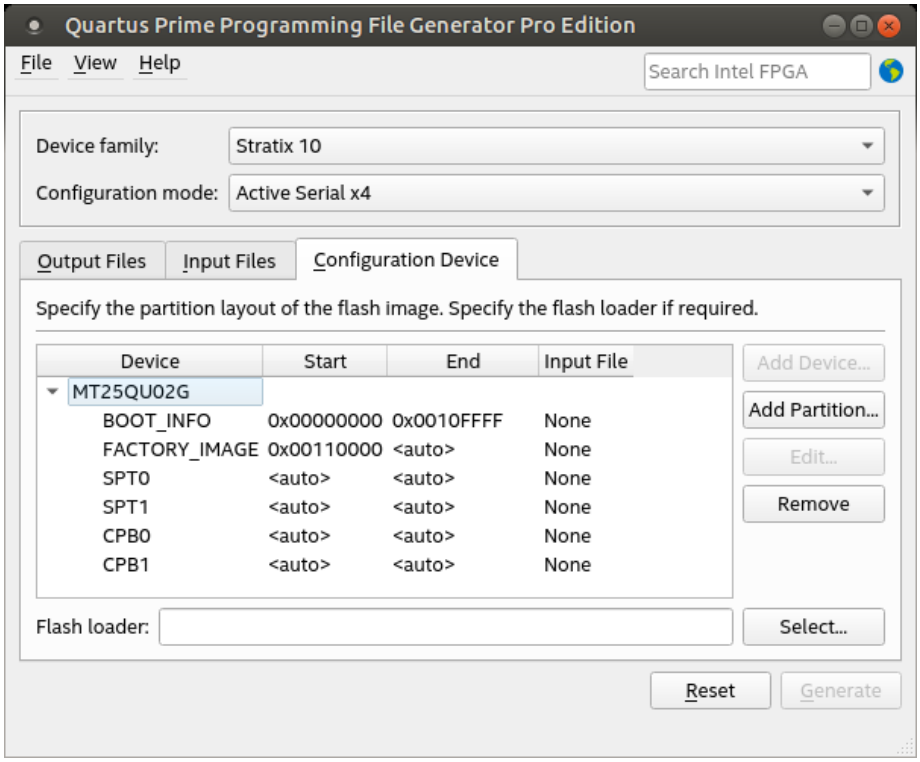
8. click first sof file, then click the **Properties** button on the right side. This opens the window to browse for the First Stage Bootloader



9. click the **... (Browse)** button and select the file `~/rsu_example/u-boot-socfpga/spl/u-boot-spl.hex`, then click OK.
10. click the second .sof file, and add the same First Stage Bootloader file to it. The **Input Files** tab now looks like shown below:

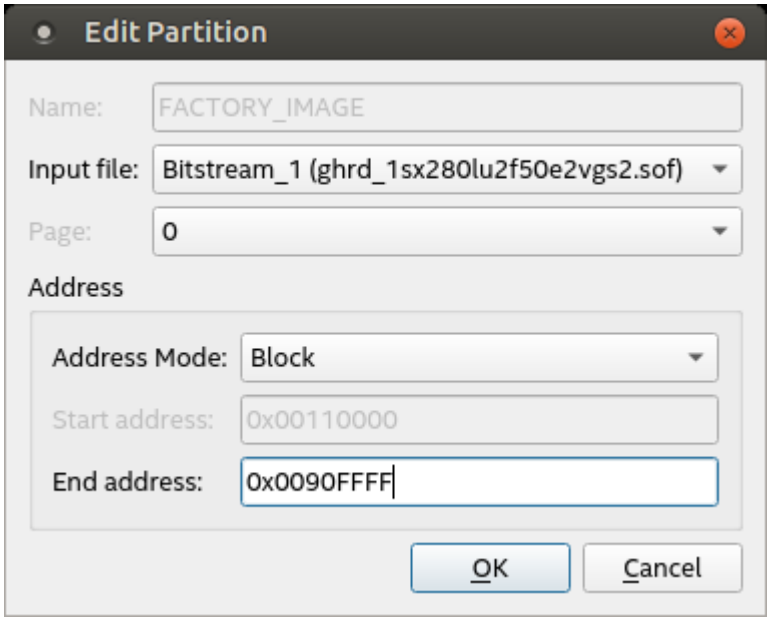


11. Click the **Configuration Device** tab. Note that the tab is only enabled once at least one input file was added in the **Input Files** tab.
12. Because more than one input file was added in the **Input Files** tab, it displays the options for remote system update. Otherwise, it only enables the standard configuration flow.
13. 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, since this example only needs a 512Mb flash device. The Quartus Programmer displays some warnings in case another flash is used, but the example works.

14. Select the **FACTORY_IMAGE** entry, and click the **Edit...** button. The **Edit Partition** window pops up. Select the **Input file** as **Bitstream_1 (ghrd_1sx280lu2f50e2vgs2.sof)**. Change **Address Mode** to **Block** since 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 8MB for the factory image. This end address was calculated by adding 8MB to the end of the **BOOT_INFO** partition. Click **OK**.



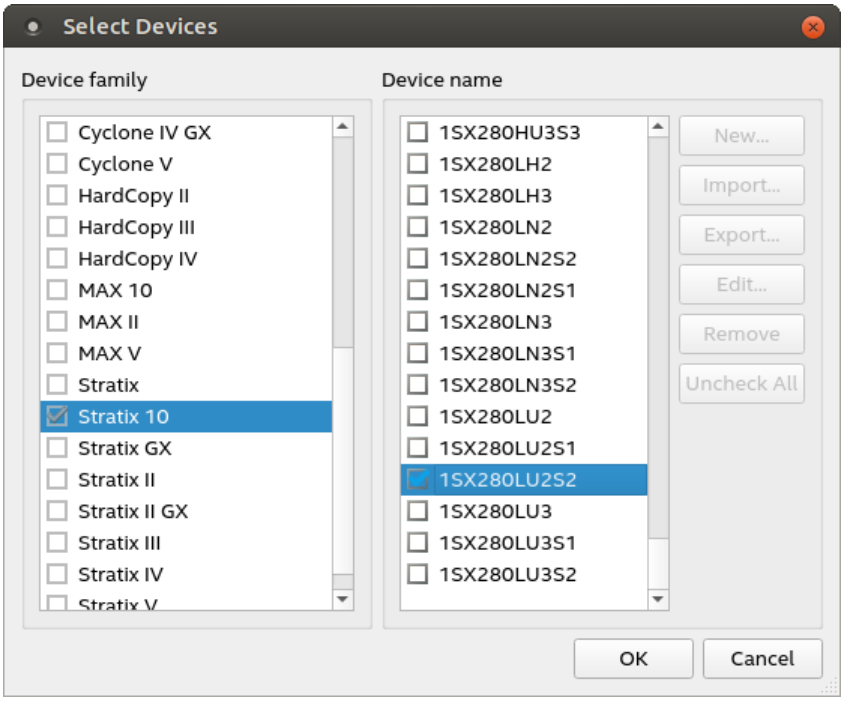
Note: The **Page** property for FACTORY_IMAGE partition must always be set to 0. This means that the FACTORY_IMAGE is tried only after all the production images failed.

15. 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_1sx280lu2f50e2vgs2.sof)**. This becomes the initial production image. Select the **Page** as **1** – this means it has the highest priority of all production images. Select the **Address Mode** as **Block** and allocate 16MB of data by setting **Start Address** = **0x01000000** and **End Address** = **0x01FFFFFF**.

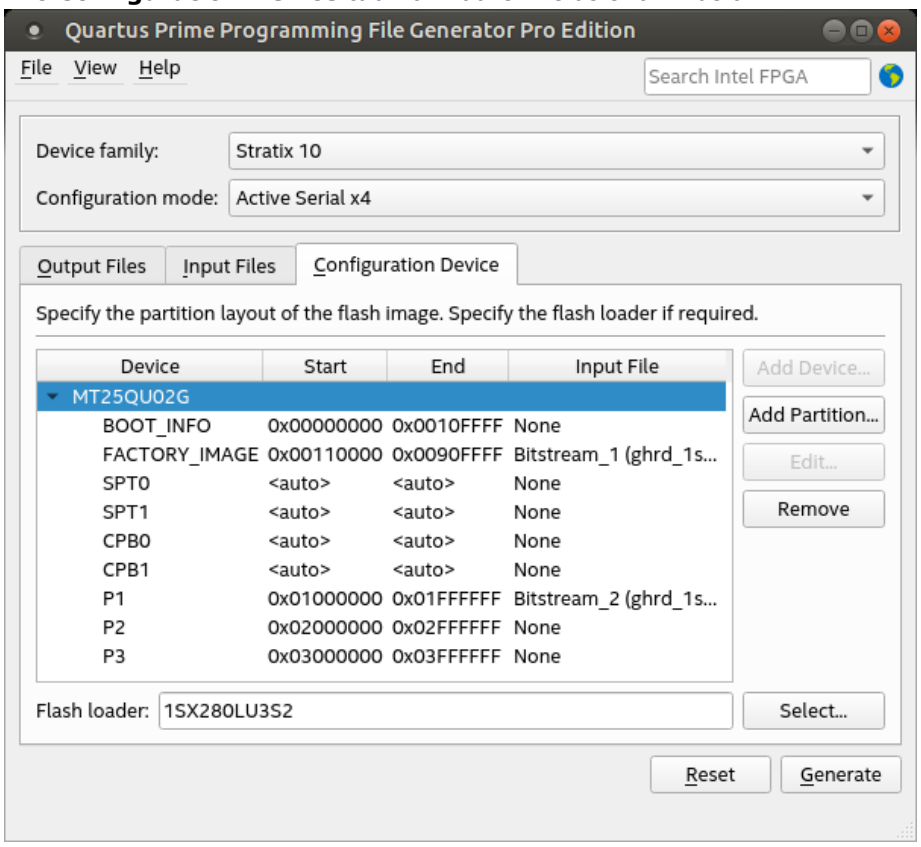
    The Page property is used by the Programming File Generator to create the initial CMF pointer block list. The first image in the list is the one with Page=1, then the one with Page=2, then the one with Page=3. The Programming File Generator issues an error if there are multiple partitions with the same page number, or if there are any "gaps" as in having a Page=1 then a Page=3, without a Page=2.

    Only up to three partitions can contain Production Images at 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 Production Image.

16. 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**.
17. 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**:**


Select Devices
Device family
Cyclone IV GX
Cyclone V
HardCopy II
HardCopy III
HardCopy IV
MAX 10
MAX II
MAX V
Stratix
Stratix 10
Stratix GX
Stratix II
Stratix II GX
Stratix III
Stratix IV
Stratix V
Device name
1SX280HU3S3
1SX280LH2
1SX280LH3
1SX280LN2
1SX280LN2S2
1SX280LN2S1
1SX280LN3
1SX280LN3S1
1SX280LN3S2
1SX280LU2
1SX280LU2S1
1SX280LU2S2
1SX280LU3
1SX280LU3S1
1SX280LU3S2
New...
Import...
Export...
Edit...
Remove
Uncheck All
OK
Cancel

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



18. Click the **Generate** button to generate the initial flash image as `~/rsu_example/initial_image.jic` and the map file as `~/rsu_example/initial_image_jic.map`. A dialog box opens indicating the files were generated successfully.
19. click **File ➤ Save As ..** and save the file as `~/rsu_example/initial_image.pfg`. This file can be useful later, if you wanted to re-generate the initial image by using the command:

```
cd ~/rsu_example
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.

6.3.5. Creating the RSU Update Files

The RSU update files can be created from the Programming File Generator GUI tool, but the easiest way is to create them from command line. The following commands are used to create the files used in this example:

```
cd ~/rsu_example
rm -rf update_files && mkdir update_files
for version in {2..3}
```

```
do
quartus_pfg -c hw/ghrd.$version/output_files/ghrd_1sx280lu2f50e2vgs2.sof \
    update_files/update.$version.rpd \
    -o hps_path=u-boot-socfpga/spl/u-boot-spl.hex \
    -o mode=ASX4 -o start_address=0x00000 -o bitswap=ON
done
```

The following files are created:

- `~/rsu_example/update_files/update.2.rpd`—update bitstream for production image 2
- `~/rsu_example/update_files/update.3.rpd`—update bitstream for production image 3

6.3.6. Building Linux

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

```
cd ~/rsu_example
rm -rf linux-socfpga
git clone https://github.com/altera-opensource/linux-socfpga
cd linux-socfpga
git checkout -b test_s10_rsu ACDS18.1_REL_GSRD_PR
# add the rsu entry to the device tree
sed -i '/memory-region = <&service_reserved>;/a\\t\t\t\trsu\
{\n\t\t\t\tcompatible = "intel,stratix10-rsu";\n\t\t\t\t};\
' arch/arm64/boot/dts/altera/socfpga_stratix10.dtsi
make clean && make mrproper
make s10_devkit_defconfig
make -j 12 Image
make dtbs
make -j 12 modules
make -j 12 modules_install INSTALL_MOD_PATH=modules_install
cd ..
```

The "rsu" entry was added to the device tree by using `sed` to edit the file: `arch/arm64/boot/dts/altera/socfpga_stratix10.dtsi` to contain the text shown in bold below:

```
...
firmware {
    svc {
        compatible = "intel,stratix10-svc";
        method = "smc";
        memory-region = <&service_reserved>;
        rsu {
            compatible = "intel,stratix10-rsu";
        };
    };
};
...
```

After completing the above steps, the following items are available in the folder ~/rsu_example/linux-socfpga:

- `arch/arm64/boot/Image`—kernel image
- `arch/arm64/boot/dts/altera/socfpga_stratix10_socdk.dtb`—kernel device tree
- `modules_install/`—folder containing the loadable modules for the kernel

6.3.7. Building ZLIB

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

```
cd ~/rsu_example
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 ..
```

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

- `~/rsu_example/zlib-1.2.11/zlib.h`—header file, used to compile files using zlib services
- `~/rsu_example/zlib-1.2.11/libz.so*`—shared objects, used to run executables linked against zlib APIs

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 either move to the latest release of all the components, or try using the latest release of Linux, as that should also work.

6.3.8. Building LIBRSU and RSU Client

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

```
cd ~/rsu_example
export ZLIB_PATH=`pwd`/zlib-1.2.11
rm -rf intel-rsu
git clone https://github.com/altera-opensource/intel-rsu
cd intel-rsu
git checkout -b test_s10_rsu ACDS18.1_REL_GSRD_PR
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:

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

6.3.9. 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 Angstrom root file system using Yocto.

1. Various packages may be needed by the build system. On a Ubuntu 14.04 machine the following command was used to install the required packages:

```
sudo apt-get install git chrpath g++ gcc make texinfo
```

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

```
cd ~/rsu_example
sudo rm -rf angstrom
mkdir angstrom && cd angstrom
wget http://commondatastorage.googleapis.com/git-repo-downloads/repo
chmod 777 repo
export PATH=`pwd`:$PATH
repo init -u git://github.com/Angstrom-distribution/angstrom-manifest \
-b angstrom-v2018.06-sumo
repo sync
MACHINE=stratix10 . ./setup-environment
sed -i '/meta-atmel/d' conf/bblayers.conf
sed -i '/meta-optee/d' conf/bblayers.conf
sed -i '/meta-freescale/d' conf/bblayers.conf
sed -i 's/IMAGE_FSTYPES.*/IMAGE_FSTYPES = "tar.gz"/g' conf/local.conf
echo "DISTRO_FEATURES_remove = \" wayland alsa \"" >> conf/local.conf
export KERNEL_PROVIDER=linux-altera-ltsi
export BB_ENV_EXTRAWHITE="$BB_ENV_EXTRAWHITE KERNEL_PROVIDER"
bitbake core-image-minimal-mtdutils
```

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: `~/rsu_example/angstrom/deploy/glibc/images/stratix10/core-image-minimal-mtdutils-stratix10.tar.gz`

For more information about building Linux, including building the Angstrom rootfs using Yocto, refer to the Rocketboards Getting Started web page.

6.3.10. Building the SD Card

In this example the bitstream containing the FPGA image and the FSBL are stored in QSPI flash, while the SSBL and the OS are stored on the HPS SD card. The following commands can be used to create the SD card image used in this example:

```
cd ~/rsu_example
sudo rm -rf sd_card && mkdir sd_card && cd sd_card
# get the sd card script and make it writable
wget https://releases.rocketboards.org/release/2018.10/gsrd/tools/\
make_sdimage.py
chmod +x make_sdimage.py
# copy all the required components to sd_card folder
cp ../u-boot-socfpga/u-boot-dtb.img .
cp ../linux-socfpga/arch/arm64/boot/Image .
cp ../linux-socfpga/arch/arm64/boot/dts/altera/socfpga_stratix10_socdk.dtb .
# create rootfs
mkdir rootfs && cd rootfs
sudo tar xf ../../angstrom/deploy/glibc/images/stratix10/\
core-image-minimal-mtdutils-stratix10.tar.gz
# remove modules compiled as part of Yocto build
sudo rm -rf lib/modules/*
# copy the intel-rsu driver
sudo cp ../../linux-socfpga/modules_install/lib/modules/*/kernel/drivers\
/misc/intel-rsu.ko home/root/
```

```
# copy the remote system update rpd images
for version in {2..3}
do
sudo cp ../../update_files/update.$version.rpd home/root/
done
# copy all the files required by LIBRSU and RSU client
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 ./make_sdimage.py -f \
  -P rootfs/*,num=2,format=ext3,size=300M  \
  -P u-boot-\
dtb.img,Image,socfpga_stratix10_socdk.dtb,num=1,format=vfat,size=100M \
  -s 512M  \
  -n sd_card_image_s10_rsu.img
cd ..
```

This creates the SD card image as:

- `~/rsu_example/sd_card/sd_card_image_s10_rsu.img`

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

- Linux RSU driver
- ZLIB shared objects
- LIBRSU shared objects and resource files
- RSU client application
- Remote system update production images

6.4. Flashing the Initial Image to QSPI

1. Make sure to install the QSPI SDM bootcard on the S10 SoC Development Kit
2. Configure the S10 SoC Development Kit as follows:
   - SW1: 1:OFF, rest:ON
   - SW2: 1:ON 2:ON 3: ON 4: OFF
   - SW3: all OFF
   - SW4: 1:ON 2:OFF 3:OFF 4:ON
3. Run the following command to write the image to SDM QSPI by using the command line version of the Quartus Programmer:

```
cd ~/rsu_example
quartus_pgm  -c 1 -m jtag -o "pvi;initial_image.jic"
```

6.5. Running RSU Example Scenarios

6.5.1. Exercising U-Boot Features

1. Write the SD card image `~/rsu_example/sd_card/sd_card_image_s10_rsu.img` to a microSD card and insert it on the SD card slot on the S10 HPS daughtercard.
2. Configure the Intel Stratix 10 SoC Development Kit to configure FPGA and load FSBL from QSPI:
   - SW1: 1:OFF, rest:ON
   - SW2: 1:ON 2:OFF 3:OFF 4: OFF
   - SW3: all OFF
   - SW4: 1:ON 2:OFF 3:OFF 4:ON
3. Power up the board
4. At the serial console, stop at the U-Boot prompt, by pressing any key during U-Boot countdown.
5. Run the `run list` command to display the RSU partitions, CPBs, the currently running image and the status:

```
SOCFPGA_STRATIX10 # rsu list
          RSU: Remote System Update Status
          Current Image  : 0x01000000
          Last Fail Image: 0x00000000
          State          : 0x00000000
          Version        : 0x00000000
          Error locaton      : 0x00000000
          Error details  : 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
```

The above listing shows that you have one factory image, and three production image slots, with just P1 being used, there are no errors, and the currently loaded image is P1.

6. At the U-Boot console, run the command `rsu update 0x0110000` to tell SDM to load the factory image. The console shows SPL then U-Boot rebooting:

```
RSU: RSU update to 0x0000000000110000
          SOCFPGA_STRATIX10 #
          U-Boot SPL 2017.09-00154-gd7e599a (Sep 25 2018 - 15:19:49)
          MPU         1000000 kHz
          L3 main     400000 kHz
          Main VCO    2000000 kHz
          Per VCO     2000000 kHz
          EOSC1       25000 kHz
          HPS MMC     50000 kHz
          UART        100000 kHz
          DDR: Initializing Hard Memory Controller
          DDR: Calibration success
          ...
          Hit any key to stop autoboot:  0
          SOCFPGA_STRATIX10 #
```

7. Run the command `run list` again to see the new status:

```
SOCFPGA_STRATIX10 # rsu list
          RSU: Remote System Update Status
          Current Image   : 0x00110000
          Last Fail Image : 0x00000000
          State           : 0x00000000
          Version         : 0x00000000
          Error locaton       : 0x00000000
          Error details       : 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
```

The status is basically as before, just that the currently running image is now listed as the factory image.

6.5.2. Exercising the Intel RSU Driver

1. Power cycle the board and let Linux boot.
2. Log in using 'root' as your name and an empty password:

```
.---O---.
|       |                  .-.           o
|   |   |-----.-----.------.| |   .----..-----.-----.
|       |     | __  |  ---'| '--.|  .-'|     |     |
|   |   |  |  |     |---  ||  --'|  |  |  '  | | | |
'---'---'--'--'--.  |-----''----''--'  '-----'-'-'-'
                -'  |
                '---'
The Angstrom Distribution stratix10 ttyS0
Angstrom v2018.06 - Kernel 4.9.78-ltsi-06726-g5704788-dirty

stratix10 login: root
root@stratix10:~#
```

3. Load the intel-rsu kernel driver by running the command:

```
root@stratix10:~# insmod intel-rsu.ko
```

4. Display the currently loaded image and the status by looking at the sysfs files created by the driver:

```
root@stratix10:~# cd /sys/devices/platform/soc:firmware:svc/\
soc:firmware:svc:rsu/
root@stratix10:~# cat current_image | xargs printf 0x%08x\\n
0x01000000
root@stratix10:~# cat error_details | xargs printf 0x%08x\\n
0x00000000
root@stratix10:~# cat error_location | xargs printf 0x%08x\\n
0x00000000
root@stratix10:~# cat state | xargs printf 0x%08x\\n
0x00000000
root@stratix10:~# cat version | xargs printf 0x%08x\\n
0x00000000
root@stratix10:~# cat fail_image | xargs printf 0x%08x\\n
0x00000000
```

5. Initiate reloading the factory image on next reboot:

```
root@stratix10:~# cd /sys/devices/platform/soc:firmware:svc/\
soc:firmware:svc:rsu/
root@stratix10:~# printf %i 0x0110000 > reboot_image
```

6. Initiate a reboot by running the `reboot` command. FSBL, SSBL and Linux reboots.
7. Load again the intel-rsu module and display the status using the files from sysfs:

```
root@stratix10:~# insmod intel-rsu.ko
root@stratix10:~# cd /sys/devices/platform/soc:firmware:svc/\
soc:firmware:svc:rsu/
root@stratix10:~# cat current_image | xargs printf 0x%08x\\n
0x00110000
```

6.5.3. Exercising the RSU Client

1. Power cycle the board and let Linux boot.
2. Log in using 'root' as user name and an empty password
3. Load the intel-rsu kernel driver by running the command:

```
root@stratix10:~# insmod intel-rsu.ko
```

4. Run the rsu_client without parameters, to display its help message:

```
root@stratix10:~# ./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, the default slot is 0 if user doesn't specify
-A|--add-raw file_name [-s|--slot] slot_num  add a new raw image to the
selected slot, the default slot is 0 if user doesn't specify
-v|--verify file_name [-s|--slot] slot_num  verify app image on the
selected slot, the default slot is 0 if user doesn't specify
-V|--verify-raw file_name [-s|--slot] slot_num  verify raw image on the
selected slot, the default slot is 0 if user doesn't specify
-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
-h|--help                           show usage message
```

5. Exercise the rsu client command that displays the current status:

```
root@stratix10:~# ./rsu_client --log
      VERSION: 0x00000000
        STATE: 0x00000000
CURRENT IMAGE: 0x0000000001000000
   FAIL IMAGE: 0x0000000000000000
    ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
Operation completed
```

6. Exercise the rsu client commands that display information about the slots:

```
root@stratix10:~# ./rsu_client --count
number of slots is 3
Operation completed
root@stratix10:~# ./rsu_client --list 0
      NAME: P1
    OFFSET: 0x0000000001000000
      SIZE: 0x01000000
  PRIORITY: 1
Operation completed
root@stratix10:~# ./rsu_client --list 1
      NAME: P2
    OFFSET: 0x0000000002000000
      SIZE: 0x01000000
  PRIORITY: [disabled]
Operation completed
root@stratix10:~# ./rsu_client --list 2
```

```
      NAME: P3
    OFFSET: 0x0000000003000000
      SIZE: 0x01000000
  PRIORITY: [disabled]
Operation completed
```

7. Add the update.2.rpd remote system update image to slot 2 and update.3.rpd to slot 1:

```
root@stratix10:~# ./rsu_client --add update.2.rpd --slot 2
Operation completed
root@stratix10:~# ./rsu_client --add update.3.rpd --slot 1
Operation completed
```

8. List again the slots, it shows the expected priorities, showing the order in which the update images are written, with the most recently written having the highest priority (lowest priority number that is):

```
root@stratix10:~# ./rsu_client --list 0
      NAME: P1
    OFFSET: 0x0000000001000000
      SIZE: 0x01000000
  PRIORITY: 3
Operation completed
root@stratix10:~# ./rsu_client --list 1
      NAME: P2
    OFFSET: 0x0000000002000000
      SIZE: 0x01000000
  PRIORITY: 1
Operation completed
root@stratix10:~# ./rsu_client --list 2
      NAME: P3
    OFFSET: 0x0000000003000000
      SIZE: 0x01000000
  PRIORITY: 2
Operation completed
```

9. Power cycle the board, boot Linux, load the RSU module and display the status – it shows the P2 image running, as expected:

```
root@stratix10:~# insmod intel-rsu.ko
root@stratix10:~# ./rsu_client --log
      VERSION: 0x00000000
        STATE: 0x00000000
CURRENT IMAGE: 0x0000000002000000
   FAIL IMAGE: 0x0000000000000000
    ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
Operation completed
```

10. Instruct the Linux kernel to request slot 2 (P3) from SDM on next reboot command:

```
root@stratix10:~# ./rsu_client --request 2
Operation completed
```

11. Initiate a Linux reboot by running the `reboot` command:

```
root@stratix10:~# reboot
```

12. Log into Linux, load the kernel driver and display the RSU status:

```
root@stratix10:~# insmod intel-rsu.ko
root@stratix10:~# ./rsu_client --log
      VERSION: 0x00000000
        STATE: 0x00000000
CURRENT IMAGE: 0x0000000003000000
```

```
   FAIL IMAGE: 0x0000000000000000
    ERROR LOC: 0x00000000
ERROR DETAILS: 0x00000000
Operation completed
```

The status shows that the P3 image loaded.

6.5.4. Exercising Watchdog Feature

This section demonstrates how a watchdog timeout produces an HPS cold reset followed by an HPS reboot. Since the hardware project is configured to treat the watchdog as a RSU failure, the SDM reports the error, and load the next image in the CMF pointer block list. U-Boot is used since it offers the simplest way to enable the watchdog, by directly writing to its registers.

1. Power cycle the board and stop at U-Boot command prompt.
2. Run `rsu list` to display the current RSU status:

```
SOCFPGA_STRATIX10 # rsu list
RSU: Remote System Update Status
Current Image   : 0x02000000
Last Fail Image : 0x00000000
State           : 0x00000000
Version         : 0x00000000
Error locaton       : 0x00000000
Error details       : 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: 0x0000000002000000 nslot: 2
Priority 2 Offset: 0x0000000003000000 nslot: 1
Priority 3 Offset: 0x0000000001000000 nslot: 0
```

The currently running image is P2, from address 0x02000000.

3. Enable the watchdog in the U-Boot:

```
SOCFPGA_STRATIX10 # mw.l 0xffd00200 1
```

4. After approximately one minute, the watchdog times out and initiates an HPS cold reset. The HPS reboots, and stops again at the U-Boot prompt.
5. Run `run list` to see the current RSU state:

```
SOCFPGA_STRATIX10 # rsu list
RSU: Remote System Update Status
Current Image       : 0x03000000
Last Fail Image : 0x02000000
State           : 0xf0060001
Version         : 0x00000000
Error locaton       : 0x000c9800
Error details       : 0x00000000
...
Priority 1 Offset: 0x0000000002000000 nslot: 2
Priority 2 Offset: 0x0000000003000000 nslot: 1
Priority 3 Offset: 0x0000000001000000 nslot: 0
```

It can be seen that:

- The current image is P3 at 0x03000000
- The last failed image is P2 at 0x02000000
- The state is 0xf0060001
  - Upper 16 bits set to is 0xf006 which means it is a watchdog timeout.
  - Lower 16 bits set to 0x0001 which means the FSBL successfully loaded SSBL and reported this just before passing control to it.

6.5.5. Exercising RSU Fallback

This section demonstrates how the RSU rejects corrupted images and goes to the next ones in the CMF pointer block list. Once all images are exhausted, the RSU falls back to the factory image. U-Boot is used since it offers the quickest way to demonstrate the feature.

1. Power up the board, stop to U-Boot prompt, and check the RSU status – it shows image at 0x02000000 loaded, with no errors:

```
SOCFPGA_STRATIX10 # rsu list
RSU: Remote System Update Status
Current Image       : 0x02000000
Last Fail Image : 0x00000000
State           : 0x00000000
Version         : 0x00000000
Error locaton       : 0x00000000
Error details       : 0x00000000
```

2. Corrupt the current image at 0x02000000 by writing somewhere inside the image:

```
SOCFPGA_STRATIX10 # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA_STRATIX10 # sf write 0x8000 0x02100000 0x100
device 0 offset 0x2100000, size 0x100
SF: 256 bytes @ 0x2100000 Written: OK
```

3. Power cycle the board, and check the RSU status – it shows the image at 0x02000000 had errors, and therefore image at 0x03000000 was loaded:

```
SOCFPGA_STRATIX10 # rsu list
RSU: Remote System Update Status
Current Image       : 0x03000000
Last Fail Image : 0x02000000
State           : 0xf004002d
```

```
Version         : 0x00000000
Error locaton       : 0x00041800
Error details       : 0x00000000
```

4. Corrupt the current image at 0x03000000:

```
SOCFPGA_STRATIX10 # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA_STRATIX10 # sf write 0x8000 0x03000000 0x10
device 0 offset 0x3000000, size 0x10
SF: 16 bytes @ 0x3000000 Written: OK
```

5. Power cycle the board, and check the RSU status - it shows the image at 0x02000000 was corrupted, and that the image at 0x01000000 was loaded instead:

```
SOCFPGA_STRATIX10 # rsu list
RSU: Remote System Update Status
Current Image       : 0x01000000
Last Fail Image : 0x02000000
State           : 0xf004002d
Version         : 0x00000000
Error locaton       : 0x00041800
Error details       : 0x00000000
```

Note: The SDM tried the image at 0x02000000 which failed, then tried the image at 0x03000000 which also failed, then the image at 0x01000000 which succeeded. The SDM reports the first failure, as described in the "RSU Status and Error Codes" section.

6. Corrupt the current image at 0x01000000:

```
SOCFPGA_STRATIX10 # sf probe
SF: Detected mt25qu02g with page size 256 Bytes, erase size 4 KiB, total
256 MiB
SOCFPGA_STRATIX10 # sf write 0x8000 0x01000000 0x100
device 0 offset 0x1000000, size 0x100
SF: 256 bytes @ 0x1000000 Written: OK
```

7. Power cycle the board, and check the RSU status – it shows that the image at 0x02000000 was corrupted, and since there were no more valid images in the CMF pointer block list, the factory image at 0x00110000 was loaded:

```
SOCFPGA_STRATIX10 # rsu list
RSU: Remote System Update Status
Current Image       : 0x00110000
Last Fail Image : 0x02000000
State           : 0xf004002d
Version         : 0x00000000
Error locaton       : 0x00041800
Error details       : 0x00000000
```

Related Information

RSU Status and Error Codes on page 62

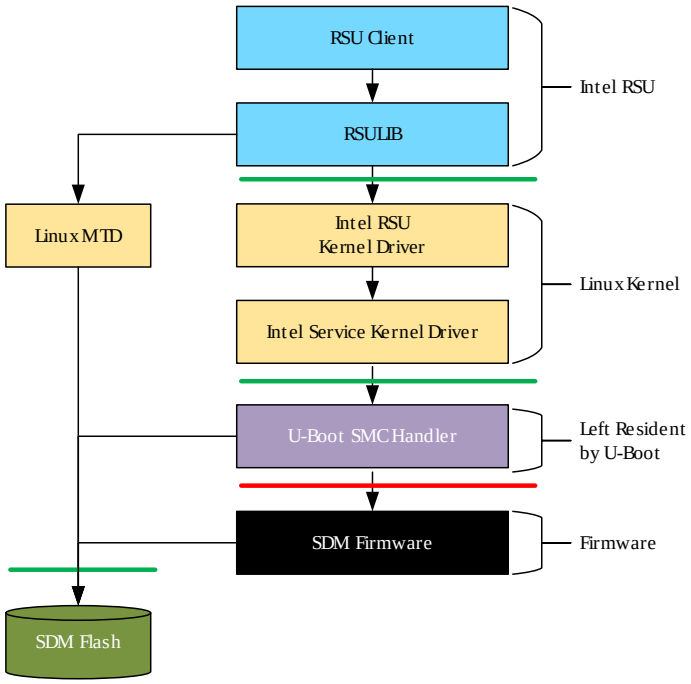
7. Version Compatibility Considerations

7.1. API Version Compatibility

The SDM provides an API which is used by U-Boot. U-Boot exports the services offered by the SDM API to Linux kernel, through the SMC interface. Linux kernel has drivers which then export the services offered through the SMC interface to Linux applications. Each of these interfaces are different, although they offer similar functionality. The following figure identifies these interfaces.

Note: SDM Firmware, U-Boot, and RSULIB directly access the SDM Flash, so an interface is shown there, as well.

Figure 10. API Version Compatibility



The API version compatibility rules are:

- The SDM Firmware/U-Boot interface (shown as a red line above) is allowed to change between versions. You must always use a U-Boot (FSBL and SSBL) version compatible with the SDM Firmware version. The FSBL is not shown in the above figure for brevity. It is used only to load FSBL and it does not remain resident.
- All other interfaces (shown as green lines above) either never change, or always change in a way in which any version of a service client should work with any version of a service provider.

The following table presents the versions for the software running on HPS.

Note: The "Current Branch" column is valid for the 18.1 release. In the future, both U-Boot and Linux may use newer branches.

Table 9. Software Versions

Item	Git Tree	Current Branch	Tag
U-Boot	https://github.com/altera-opensource/u-boot-socfpga	`origin/socfpga_v2017.09`	`ACDS<major>.<minor>_REL_S10_GSRD_PR`
Linux Kernel	https://github.com/altera-opensource/linux-socfpga	`origin/socfpga-4.9.78-ltsi`	`ACDS<major>.<minor>_REL_GSRD_PR`
Intel RSU	https://github.com/altera-opensource/intel-rsu	`origin/master`	`ACDS<major>.<minor>_REL_GSRD_PR`

The SDM firmware is added to the bitstream by the Intel Quartus Prime Programming File Generator, which is part of a Intel Quartus Prime release. A Intel Quartus Prime release is identified by its <major>.<minor> number, such as "18.1" or "19.1". The correct version of U-Boot to use with a certain Intel Quartus Prime Programming File Generator release is given by the tag ACDS`<major>.<minor>`_REL_S10_GSRD_PR tag on GitHub. Both FSBL and SSBL must be built from the same version of U-Boot.

It can be sometimes useful to use the latest on the U-Boot branch instead of the official GSRD tag described above to get access to new functionality and bug fixes which are posted on GitHub between releases. Care must be exercised when doing so, as once a new Intel Quartus Prime version is released, the interface between SDM firmware and U-Boot may change. Intel recommends always using the GSRD official tag, unless explicitly instructed to do otherwise.

Updates to Intel Quartus Prime can sometimes be released which can impact the interface between SDM firmware and U-Boot. In such cases, instructions are provided along with the update to enable the selection of the proper U-Boot version.

Note: Typically the same version of tools is used for both Intel Quartus Prime (which creates the FPGA configuration data as a SOF file) and Intel Quartus Prime Programming File Generator (which creates the bitstreams containing the SDM firmware). You can always generate a bitstream using a newer version of Intel Quartus Prime Programming File Generator with an older version of SOF, but not the other way around. So your SDM firmware can have a newer version than your FPGA configuration data.

7.2. API Version Compatibility Testing

The interface between U-Boot and Linux is designed as version compatible. That is, any version of Linux can work with any version of U-Boot. In order to ensure this compatibility, Intel performs the following tests:

For each U-Boot release (tag ACDS<major.<minor>_REL_S10_GSRD_PR), it tests it with the previous three releases of Linux and Intel RSU (tags ACDS<major>.<minor>_REL_GSRD_PR)

For each Linux and Intel RSU release (tags ACDS<major>.<minor>_REL_GSRD_PR) it tests it with the previous three releases of U-Boot (tag ACDS<major.<minor>_REL_S10_GSRD_PR)

While the above are not an exhaustive set of tests for all possible combinations, they ensure that incompatibilities that can be introduced in the U-Boot/Linux interface or the QSPI flash interface is caught and fixed before each release.

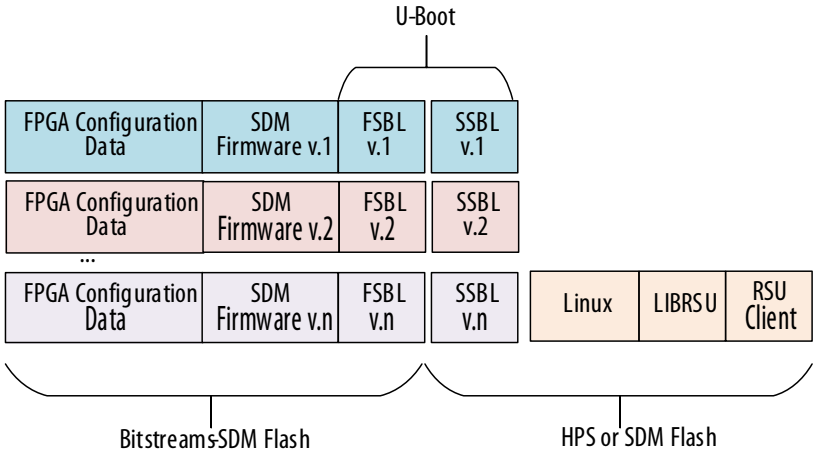
7.3. Using Multiple Intel Quartus Prime Versions for Bitstreams

As mentioned, the version of U-Boot (FSBL & SSBL) needs to be compatible with the version of SDM firmware being used, which is tied to the Intel Quartus Prime version.

If different bitstreams (production or factory) are generated with different versions of Intel Quartus Prime, then each of these bitstreams need to use a compatible version of FSBL. Furthermore each of the FSBLs requires a compatible SSBL. This results in the requirement of having a separate SSBL for each bitstream.

A typical system using multiple versions of Intel Quartus Prime for the bitstreams can look as shown in the following figure:

Figure 11. RSU System Using Multiple Intel Quartus Prime Versions



Some users may even want to duplicate Linux, LIBRSU and RSU Client for each bitstream, but that is not required, as those interfaces are forward and backward compatible. It is only the U-Boot (FSBL and SSBL) which needs to be compatible with the SDM firmware API.

7.4. Updating U-Boot to Support Multiple Quartus Versions

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 multiple Quartus versions by having a different SSBL for each bitstream.

7.4.1. 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 production image update procedure also changes:

1. Use LIBRSU to erase the production image partition. This also disables it, removing it from the CMF pointer block list.
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. Use LIBRSU to write the new production image. This also enables it, putting it in front of the CMF pointer block list.

7.4.2. 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 … etc for production 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`
- In the Input Files tab, click 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 (512KB can suffice 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.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. 512KB can still suffice, as typical U-Boot image is smaller, and typical environment size is 4KB.
- 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 production image update procedure also changes:

- Use LIBRSU to erase the production image partition. This also disables it, removing it from the CMF pointer block list.
- Use MTD to erase SSBL partition. Cannot use LIBRSU since it does not support erasing raw partitions yet, just bitstreams.
- Use MTD to write the new contents of the SSBL partition. Cannot use LIBRSU since it does not support writing raw partitions yet, just bitstreams.
- Use LIBRSU to write the new production image. This also enables it, putting it in front of the CMF pointer block list.

7.4.3. 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_s10.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.

File name for the SSBL binary when it is loaded from SD card is defined in `include/configs/socfpga_stratix10_socdk.h`:

```
#define CONFIG_SYS_MMCSD_FS_BOOT_PARTITION    1
#define CONFIG_SPL_FS_LOAD_PAYLOAD_NAME        "u-boot-dtb.img"
```

Location and size of U-Boot environment when it is stored in SD/MMC is defined in `include/configs/socfpga_stratix10_socdk.h`:

```
#define CONFIG_ENV_SIZE            0x1000
#define CONFIG_SYS_MMC_ENV_DEV        0    /* device 0 */
#define CONFIG_ENV_OFFSET         512    /* just after the MBR */
```

Location of SSBL in QSPI flash is defined in `include/configs/socfpga_stratix10_socdk.h`:

```
#define CONFIG_SYS_SPL_MALLOC_START    (CONFIG_SPL_BSS_START_ADDR \
                    - CONFIG_SYS_SPL_MALLOC_SIZE)
#define CONFIG_SPL_SPI_LOAD
#define CONFIG_SYS_SPI_U_BOOT_OFFS      0x3C00000
```

Location and size of U-Boot environment in QSPI flash is defined in `include/configs/socfpga_stratix10_socdk.h`:

```
#ifdef CONFIG_ENV_IS_IN_SPI_FLASH
#undef CONFIG_ENV_OFFSET
#undef CONFIG_ENV_SIZE
#define CONFIG_ENV_OFFSET         0x710000
#define CONFIG_ENV_SIZE             (4 * 1024)
#define CONFIG_ENV_SECT_SIZE         (4 * 1024)
#endif /* CONFIG_ENV_IS_IN_SPI_FLASH */
```

8. 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.

8.1. Update Hardware Designs to use HPS First

When creating the hardware projects, enable HPS first for each project, either from Quartus GUI, or as shown in bold below:

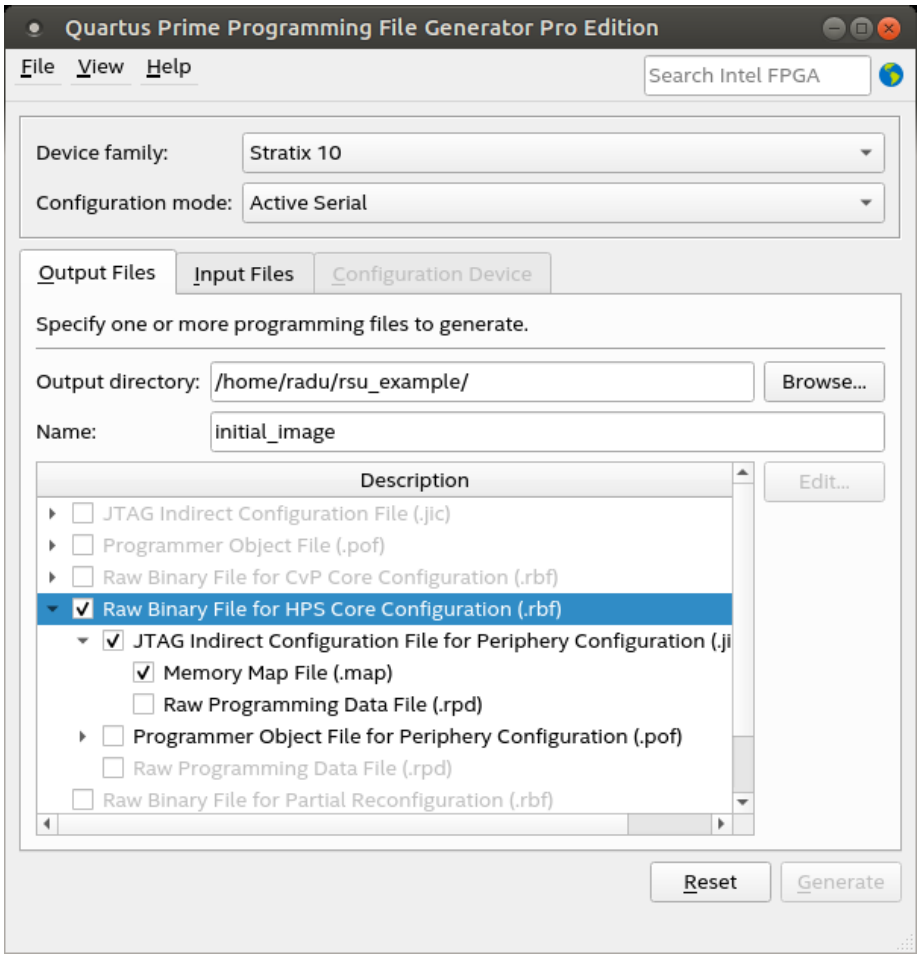
```
cd ~/rsu_example
# compile hardware designs: 0-factory, 1..3-production
rm -rf hw && mkdir hw && cd hw
for version in {0..3}
do
mkdir ghrd.$version && cd ghrd.$version
tar xf $SOCEDS_DEST_ROOT/examples/hardware/s10_soc_devkit_ghrd/tgz/*.tar.gz
make clean
make scrub_clean
rm -rf *.qpf *.qsf *.txt *.bin *.qsys ip/qsys_top/ ip/subsys_jtg_mst/ ip/
subsys_periph/
# use the device on development kit
sed -i 's/QUARTUS_DEVICE := .*/QUARTUS_DEVICE := 1SX280LU2F50E2VGS2/g' Makefile
# enable hps first option
sed -i 's/BOOTS_FIRST := .*/BOOTS_FIRST := hps/g' Makefile
# change the sysid to make projects different
sed -i 's/0xACD5CAFE/0xABAB000'$version'/g' create_ghrd_qsys.tcl
# change the behavior of the watchdog timeput to trigger a cold reset with RSU
sed -i 's/W_RESET_ACTION ./W_RESET_ACTION 2/g' construct_hps.tcl
# disable ECC as there are some issues with it in 18.1
sed -i "s/set HPS_EMIF_ECC_EN ./set HPS_EMIF_ECC_EN 0/g" design_config.tcl
# generate project
make generate_from_tcl
# build project
make sof
cd ..
done
cd ..
```

8.2. 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)** and **Memory Map File (.map)** options.

Figure 12. Creating JIC File for HPS First



- Make the factory image and production image partitions smaller, since they will not store the FPGA fabric configuration data.

After clicking the **Generate** button the following additional files are generated in the `~/rsu_example` 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 production image P1.

8.3. Generating RSU Update Files for HPS First

An additional parameter needs to be added to enable HPS first, as shown below in bold:

```
cd ~/rsu_example
rm -rf update_files && mkdir update_files
for version in {2..3}
do
quartus_pfg -c hw/ghrd.$version/output_files/ghrd_1sx280lu2f50e2vgs2.sof \
    update_files/update.$version.rpd \
    -o hps=1 \
    -o hps_path=u-boot-socfpga/spl/u-boot-spl.hex \
    -o mode=ASX4 -o start_address=0x00000 -o bitswap=ON
done
```

The following files are generated in the folder `~/rsu_example/update_files`:

- `update.2.core.rbf` – FPGA fabric configuration file for production image 2
- `update.2.hps.rpd` – Bitstream configuration file for production image 2
- `update.3.core.rbf` – FPGA fabric configuration file for production image 3
- `update.3.hps.rpd` – Bitstream configuration file for production image 3

A. RSU Status and Error Codes

The status can be checked by one of the following means:

- Running `rsu list` command in U-Boot
- Looking at the files from `/sys/devices/platform/soc:firmware:svc/soc:firmware:svc:rsu/` in Linux
- Running the `rsu client --log` command in Linux

The status is reported by the SDM and contains the following 32-bit fields:

Table 10. RSU Status Fields

Field	Description
`current_image`	Location of currently running image in the SDM QSPI 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`	Currently hardcoded to zero. May change in the future.

The `failed_image`, `error_details`, `error_location`, and `state` are set when an error occurs, then they are not updated on subsequent errors. They are cleared when one of the following events occur: POR, nCONFIG, or RSU_UPDATE command is sent to the SDM.

The "state" field has two parts:

- Upper 16bit: major error code
- Lower 16bit: opaque minor error code.

The following major error codes are defined:

Table 11. 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 users. 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.

B. LIBRSU Reference Information

B.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 12. LIBRSU Configuration File Elements

Element	Description
Element: `# COMMENT` **Usage:** `// COMMENT` **Options:** None	Single line comments **Required?**: No
Element: `root` **Usage:** `root {type} {path}` **Options:** `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. **Required?**: Yes
Element: `log` **Usage:** `log {level} [stderr|path]` **Options:** • `level : Verbose level = [off, low, medium, high]` • `path : Path to a logfile for debug information` • `stderr (defaults to stderr)`	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. **Required?**: No
Element: `write-protect` **Usage:** `write-protect {slot}` **Options:** `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. **Required?**: No

The default values are:

```
# cat /etc/librsu.rc
log med stderr
root qspi /dev/mtd0
```

The U-Boot 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.

B.2. Error Codes

In case of success, the RSULIB APIs return the value 0; otherwise, the RSULIB 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 ECMP          7
#define ESIZE         8
#define ENAME         9
#define EFILEIO      10
#define ECALLBACK    11
#define ELOWLEVEL    12
#define EWRPROT      13
#define EARGS        14
```

B.3. Data Types

B.3.1. `rsu_slot_info`

This structure contains slot (SPT entry) information.

```
struct rsu_slot_info {
    char name[16];
    __u64 offset;
    int size;
    int priority;
};
```

B.3.2. `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;
};
```

B.3.3. `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);
```

B.4. Functions

B.4.1. librsu_init

Prototype	`int librsu_init(char *filename);`
Description	Load the configuration file and initialize internal data by reading SPT and CPB data from flash. • If SPT0 is corrupted, SPT1 is loaded instead. If one SPT is corrupted (no magic number) and one is good, the corrupted SPT is recovered with information from the good SPT. • If CPB0 is corrupted, CPB1 is loaded instead. If one CPB is corrupted (no magic number) and one is good, the corrupted CPB is recovered with information from the good CPB.
Parameters	`filename`: configuration file to load. If Null or empty string, the default is `/etc/librsu.rc`
Return Value	0 on success, or error code

B.4.2. librsu_exit

Prototype	`void librsu_exit(void);`
Description	Cleanup internal data and release librsu.
Parameters	None
Return Value	None

B.4.3. rsu_slot_count

Prototype	`int rsu_slot_count(void);`
Description	Get the number of slots defined.
Parameters	None
Return Value	The number of defined slots

B.4.4. rsu_slot_by_name

Prototype	`int rsu_slot_by_name(char *name);`
Description	Retrieve slot number based on name.
Parameters	`name`: name of slot
Return Value	Slot number on success, or error code

B.4.5. rsu_slot_get_info

Prototype	`int rsu_slot_get_info(int slot, struct rsu_slot_info *info);`
Description	Retrieve the attributes of a slot.
Parameters	`slot`: slot number `info`: pointer to info structure to be filled in
Return Value	0 on success, or error code

B.4.6. rsu_slot_size

Prototype	`int rsu_slot_size(int slot);`
Description	Get the size of a slot.
Parameters	`slot`: slot number
Return Value	The size of the slot in bytes, or error code

B.4.7. rsu_slot_priority

Prototype	`int rsu_slot_priority(int slot);`
Description	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.
Parameters	`slot`: slot number
Return Value	The priority of the slot, or error code

B.4.8. rsu_slot_erase

Prototype	`int rsu_slot_erase(int slot);`
Description	Erase all data in a slot to prepare for programming. Remove the slot if it is in the CMF Pointer Block.
Parameters	`slot`: slot number
Return Value	0 on success, or error code

B.4.9. rsu_slot_program_buf

Prototype	`int rsu_slot_program_buf(int slot, void *buf, int size);`
Description	Program a slot using FPGA config data from a buffer and enter slot into CMF Pointer Block as highest priority.
Parameters	`slot`: slot number `buf`: pointer to data buffer `size`: bytes to read from buffer
Return Value	0 on success, or error code

B.4.10. rsu_slot_program_file

Prototype	`int rsu_slot_program_file(int slot, char *filename);`
Description	Program a slot using FPGA config data from a file and enter slot into CMF Pointer Block as highest priority.
Parameters	`slot`: slot number `filename`: input data file
Return Value	0 on success, or error code

B.4.11. rsu_slot_program_buf_raw

Prototype	`int rsu_slot_program_buf_raw(int slot, void *buf, int size);`
Description	Program a slot using raw data from a buffer. The slot is not entered into the CMF Pointer Block.
Parameters	`slot`: slot number `buf`: pointer to data buffer `size`: bytes to read from buffer
Return Value	0 on success, or error code

B.4.12. rsu_slot_program_file_raw

Prototype	`int rsu_slot_program_file_raw(int slot, char *filename);`
Description	Program a slot using raw data from a file. The slot is not entered into the CMF Pointer Block.
Parameters	`slot`: slot number `filename`: input data file
Return Value	0 on success, or error code

B.4.13. rsu_slot_verify_buf

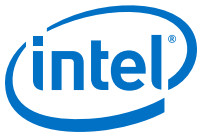
Prototype	`int rsu_slot_verify_buf(int slot, void *buf, int size);`
Description	Verify FPGA config data in a slot against a buffer.
Parameters	`slot`: slot number `buf`: pointer to data buffer `size`: bytes to read from buffer
Return Value	0 on success, or error code

B.4.14. rsu_slot_verify_file

Prototype	`int rsu_slot_verify_file(int slot, char *filename);`
Description	Verify FPGA config data in a slot against a file.
Parameters	`slot`: slot number `filename`: input data file
Return Value	0 on success, or error code

B.4.15. rsu_slot_verify_buf_raw

Prototype	`int rsu_slot_verify_buf_raw(int slot, void *buf, int size);`
Description	Verify raw data in a slot against a buffer.
Parameters	`slot`: slot number `buf`: pointer to data buffer `size`: bytes to read from buffer
Return Value	0 on success, or error code



B.4.16. rsu_slot_verify_file_raw

Prototype	`int rsu_slot_verify_file_raw(int slot, char *filename);`
Description	Verify raw data in a slot against a file.
Parameters	`slot`: slot number `filename`: input data file
Return Value	0 on success, or error code

B.4.17. rsu_slot_program_callback

Prototype	`int rsu_slot_program_callback(int slot, rsu_data_callback callback);`
Description	Program and verify a slot using FPGA config data provided by a callback function. Enter the slot into the CMF Pointer Block as highest priority.
Parameters	`slot`: slot number `callback`: callback function to provide input data
Return Value	0 on success, or error code

B.4.18. rsu_slot_program_callback_raw

Prototype	`int rsu_slot_program_callback_raw(int slot, rsu_data_callback callback);`
Description	Program and verify a slot using raw data provided by a callback function. The slot is not entered into the CMF Pointer Block.
Parameters	`slot`: slot number `callback`: callback function to provide input data
Return Value	0 on success, or error code

B.4.19. rsu_slot_verify_callback

Prototype	`int rsu_slot_verify_callback(int slot, rsu_data_callback callback);`
Description	Verify a slot using FPGA configuration data provided by a callback function.
Parameters	`slot`: slot number `callback`: callback function to provide input data
Return Value	0 on success, or error code

B.4.20. rsu_slot_verify_callback_raw

Prototype	`int rsu_slot_verify_callback_raw(int slot, rsu_data_callback callback);`
Description	Verify a slot using raw data provided by a callback function.
Parameters	`slot`: slot number `callback`: callback function to provide input data
Return Value	0 on success, or error code

B.4.21. rsu_slot_copy_to_file

Prototype	`int rsu_slot_copy_to_file(int slot, char *filename);`
Description	Read the data in a slot and write to a file.
Parameters	`slot`: slot number `filename`: input data file
Return Value	0 on success, or error code

B.4.22. rsu_slot_enable

Prototype	`int rsu_slot_enable(int slot);`
Description	Set the selected slot as the highest priority. This is the first slot attempted after a power-on reset.
Parameters	`slot`: slot number
Return Value	0 on success, or error code

B.4.23. rsu_slot_disable

Prototype	`int rsu_slot_disable(int slot);`
Description	Remove the selected slot from the priority scheme, but do not erase the slot data so that it can be re-enabled.
Parameters	`slot`: slot number
Return Value	0 on success, or error code

B.4.24. rsu_slot_load_after_reboot

Prototype	`int rsu_slot_load_after_reboot(int slot);`
Description	Request that the selected slot be loaded after the next reboot, no matter the priority. A power-on reset ignores this request and uses slot priority to select the first slot.
Parameters	`slot`: slot number
Return Value	0 on success, or error code

B.4.25. rsu_slot_load_factory_after_reboot

Prototype	`int rsu_slot_load_factory_after_reboot(void);`
Description	Request that the factory image be loaded after the next reboot. A power-on reset ignores this request and uses slot priority to select the first slot.
Parameters	None
Return Value	0 on success, or error code

B.4.26. rsu_slot_rename

Prototype	`int rsu_slot_rename(int slot, char *name);`
Description	Rename the selected slot.
Parameters	`slot`: slot number `name`: new name for slot
Return Value	0 on success, or error code

B.4.27. rsu_slot_status_log

Prototype	`int rsu_status_log(struct rsu_status_info *info);`
Description	Copy the SDM status log to info struct.
Parameters	`info`: pointer to info struct to fill in
Return Value	0 on success, or error code

11. Document Revision History for the Intel Stratix 10 SoC Remote System Update User Guide

Table 13. Document Revision History for the Intel Stratix 10 SoC Remote System Update User Guide

Document Version	Changes
2019.02.27	Initial release