RX600 & RX200 Series

The Flash Loader Project

R01AN0287EU0300
Rev.3.00
February 22, 2013

Introduction

The Flash Loader project is a flexible system that's goal is to help users add the ability to upgrade their firmware in-the-field. The Flash Loader project gives users a framework to build off of so that they can customize the project to meet their own specific needs. Each piece of the project is documented in this application note to help the user see how and what needs to be changed for their own implementation.

Target Device

The following is a list of devices that are currently supported by this project:

- **RX610 Group**
- **RX621, RX62N, RX62T Groups**
- **RX630, RX631, RX63N Groups**
- **RX210 Groups**

Related Documents

- The Flash Loader Project – SD Card Implementation (R01AN1535EU0100)
- Simple Flash API for RX (R01AN0544EU0240)

Contents

1. Overview

The Flash Loader project is meant to help users implement in-the-field upgradability in their designs. Other projects have performed aspects of this project but the goal was to cover all required aspects. An example is the Flash-Over-CAN project. This works great for customers using CAN but what if you wanted to use Ethernet or USB? The Flash Loader project was designed to be modular so that users can modify it to meet their needs. Some of the requirements when designing the project were:

- Flexible system capable of adapting to customer's needs
  - Hardware agnostic
- Error checking built into project
  - Not reliant on external protocols
- Packets used for communications to enable transfer retries
- Should not interfere with user's application
- All source code is available for MCU and Windows applications

These requirements became a subset of the features offered and helped shape the Flash Loader project.

1.1 Background Information

To help understand the rest of the Flash Loader project some background information is presented in this section.

1.1.1 Terms Used

This section will define some terms that will be used throughout the rest of this document.

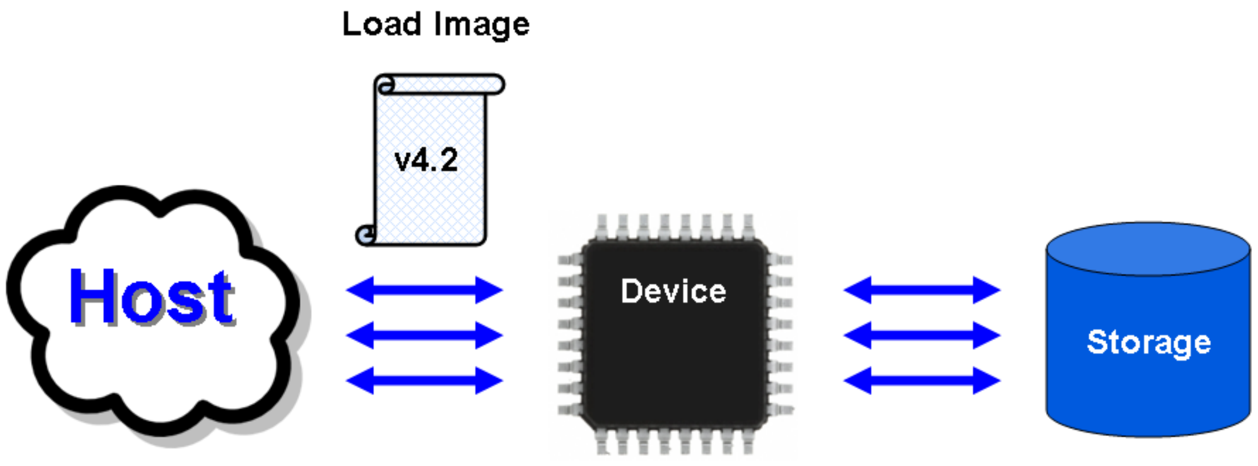


Figure 1-1 : Pieces of the Flash Loader Project

Device: This is the MCU that the user is implementing the Flash Loader project on. It is the MCU that receives the firmware update and uses it at some point.

Host: The Host is the computer, MCU, etc… that communicates with the Device. It sends the Device new firmware images, asks for information about what it is currently running, and has the ability to erase parts of the Device's Flash Loader Storage area.

Load Image: Once the user has a new firmware image they want to transfer to the Device they convert it into a Load Image. The Load Image contains the data the Device needs to update its firmware and check for communications errors.

Storage: The Flash Loader Storage is memory that is set aside to be used by the Flash Loader project. It holds Load Images that the Device receives from the Host.

1.1.2 Load Image Information

When sending a new firmware update to the Device the data is called a Load Image. A Load Image is the user's firmware formatted to work with the Flash Loader project. Load Images are generated from the S-Record (*.mot) file that is generated when you compile and link your project. There are several reasons that we convert the S-Record file instead of just using it straight from the linker. The main reasons are:

1. **File Size:** S-Record files are stored as ASCII text which makes them human readable but large in size.
2. **Line Size:** S-Record files have a very limited line size (257 bytes max per line)
3. **Block ID:** In order to make retries as easy as possible Load Image packets have a unique identifier so the Device can easily figure out where to restart the transfer.
4. **Error Checking:** A checksum is used with an S-Record file while we wanted to use a CRC value.

1.1.3 Load Image Format

Every Load Image is made up of one header which describes the file as a whole and then some number of Data Blocks.

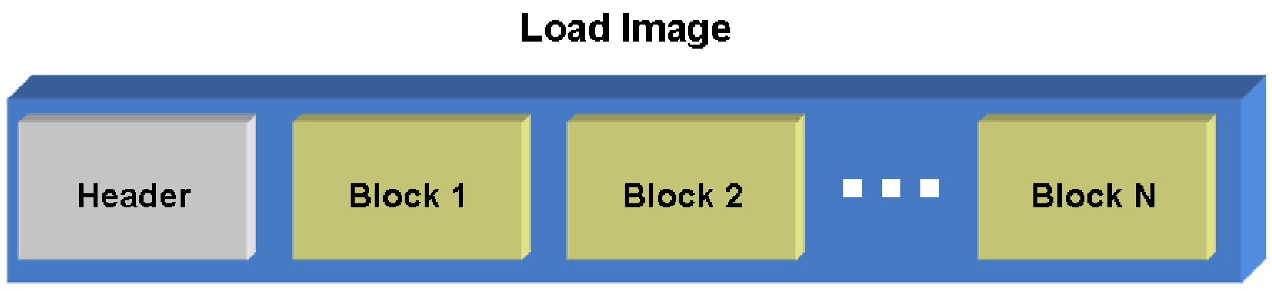


Figure 1-2 : Inside a Load Image

In the Flash Loader project the file header is called a Load File Header. The format for this header can be seen in Appendix A. Each Data Block, which can be thought of as a packet for transmission, has its own header called a Data Block Header. The format for this header can be seen in Appendix B.

2. API Information

This Middleware API follows the Renesas API naming standards.

2.1 Hardware Requirements

This middleware requires your MCU support the following features:

- Timer able to generate timer tick
- Ability to rewrite memory area where application code is stored

2.2 Hardware Resource Requirements

This section details the hardware peripherals that this middleware requires. Unless explicitly stated, these resources must be reserved for the middleware and the user cannot use them.

Note that the resources shown below are only for the default Flash Loader implementation that comes packaged with this application note. The Flash Loader project is meant to be modified to meet the needs of the user's application. This means that these resources will most likely differ for your own design.

2.2.1 CRC

CRCs are used for catching communication errors.

2.2.2 Flash Control Unit (FCU)

The FCU takes care of programming and erasing internal memory. This middleware uses the FCU and therefore should not be used by the middleware user.

2.2.3 RSPI

One RSPI channel is used for communicating with a SPI flash.

2.2.4 SCI

One SCI channel is used for communicating with a Flash Loader Host.

2.2.5 WDT

The watchdog timer is used to reset the MCU when a new firmware image has been downloaded.

2.2.6 CMT (compare match timer)

This timer is used to generate a timer tick that drives the state machine. The user is responsible for calling the state machine so they can use any timer they wish. The CMT is listed here because it is used in the demo and in the bootloader.

2.3 Software Requirements

This middleware depends on the following packages.

2.3.1 r_crc_rx

This package is used for generating CRC codes.

2.3.2 r_flash_api_rx

This package is used for rewriting the MCU's internal ROM and data flash.

2.3.3 r_glyph

This package is used for controlling the LCD on RDK boards (i.e. RDKRX62N, RDKRX63N).

2.3.4 r_rspi_rx

This package is used for communicating with an external SPI flash.

2.3.5 r_sci_async_1ch_rx

This package is used for communicating with a Host over an asynchronous serial channel.

2.3.6 r_spi_flash

This package is used for interfacing to a SPI flash. While the *r_rspi_rx* package implements the low-level drivers to communicate with the SPI flash, this package is used to send commands and control the SPI flash.

2.3.7 r_cmt_rx

This package is used to control a CMT channel which generates a timer tick that drives the state machine.

2.3.8 r_delay

This package is used to implement delays. This is currently only used by the r_glyph module.

2.4 Supported Toolchains

This middleware is tested and working with the following toolchains:

- Renesas RX Toolchain v1.02.01

2.5 Header Files

All API calls are accessed by including a single file: *r_flash_loader_rx_if.h*. This header file is supplied with this middleware's project code.

2.6 Integer Types

This project uses ANSI C99 "Exact width integer types" in order to make the code clearer and more portable. These types are defined in *stdint.h*.

2.7 Configuration Overview

The core Flash Loader code is configured through the *r_flash_loader_rx_config.h* header file. The configuration options available in this header file are shown in the table below.

Configuration Options in *r_flash_loader_rx_config.h*	
FL_CFG_DATA_BLOCK_MAX_BYTES	Maximum block data size supported. sizeof(fl_block_header_t) is added to this #define when declaring the receive buffer. This is done because the r_fl_mot_converter.py program accepts a parameter to set the data block size. That parameter does not take into account the block header size so neither does this one.
FL_CFG_TIMEOUT_ENABLE	Whether to use a timeout or not. A timeout occurs when FL_CFG_TIMEOUT_TICKS go by without receiving an expected response from the host. If a reply is not expected then a timeout will not occur. Using a timeout helps the state machine not get stuck when the host goes down during communications. • '0' means do not use a timeout. • '1' means do use a timeout.
FL_CFG_TIMEOUT_TICKS	Number of ticks of the state machine before a timeout occurs. The time for each tick will depend on the frequency of the timer that is used to call the state machine. For example, if the state machine is called at 50Hz then the time for each tick is 20ms.
FL_CFG_MEM_NUM_LOAD_IMAGES	Number of load image slots available. The larger this number, the more firmware images that can be stored at once. Making this number larger also potentially means that each firmware image will have less room available. The reason for this is that firmware images are stored at reserved static locations.
FL_CFG_MEM_BASE_ADDR	Starting address of where Flash Loader load images are stored. The address for each load image will be based on this address. For example, if FL_CFG_MEM_MAX_LI_SIZE_BYTES is 0x10000 then the addresses would be the following if FL_CFG_MEM_NUM_LOAD_IMAGES was set to 4: • Address of Load Image 0: 0x00000000 • Address of Load Image 1: 0x00010000 • Address of Load Image 2: 0x00020000 • Address of Load Image 3: 0x00030000
FL_CFG_MEM_MAX_LI_SIZE_BYTES	Maximum supported load image size. If a host sends a request to download a new image that is larger than this, the MCU will deny the request.

Table 2-1 : Module configuration options

2.8 Adding Middleware to Your Project

This section details how to add the Flash Loader code to your own project. The Flash Loader project is made up of two projects: a bootloader and a user application.

2.8.1 Flash Loader Bootloader

1. Copy the ‘r_flash_loader_rx’ directory (packaged with this application note) to your project directory.
2. Add the following source files to your project.
    a. src\r_fl_bootloader.c
    b. src\r_fl_downloader.c
    c. src\r_fl_store_manager.c
    d. src\r_fl_utilities.c
3. Add the source file from the ‘communications’ directory that corresponds to your project’s method of communication between the Host and Device.
4. Add the source file from the ‘memory’ directory that corresponds to your project’s method of communication between the Device and Storage.
5. Add an include path to the 'r_flash_loader' directory.
6. Add an include path to the 'r_flash_loader\src' directory.
7. Copy r_flash_loader_config_reference.h from 'ref' directory to your desired location and rename to r_flash_loader_config.h.
8. Configure middleware through r_flash_loader_config.h.
9. If you are placing the bootloader in the User Boot area then make sure to:
    a. Configure your linker to place the code in the correct area.
    b. Configure your BSP to choose User Boot Mode. This is done by configuring *r_bsp_config.h* if you are using the r_bsp package.
    c. Read Section 5.4.10 for information about using the Flash API from the User Boot Area.

2.8.2 Flash Loader User Application

1. Copy the ‘r_flash_loader_rx’ directory (packaged with this application note) to your project directory.
2. Add the following source files to your project.
    a. src\r_fl_app_header.c
    b. src\r_fl_downloader.c
    c. src\r_fl_store_manager.c
    d. src\r_fl_utilities.c
3. Add the source file from the ‘communications’ directory that corresponds to your project’s method of communication between the Host and Device.
4. Add the source file from the ‘memory’ directory that corresponds to your project’s method of communication between the Device and Storage.
5. Add an include path to the 'r_flash_loader' directory.
6. Add an include path to the 'r_flash_loader\src' directory.
7. Copy r_flash_loader_config_reference.h from 'ref' directory to your desired location and rename to r_flash_loader_config.h.
8. Configure middleware through r_flash_loader_config.h.
9. Add a #include for r_flash_loader_rx_if.h to files that need to use this package.

3. Flash Loader on the Device

The Flash Loader project on the Device can be split into two separate distinct parts: the Flash Loader Downloader, and the Flash Loader Bootloader.

3.1 Flash Loader Downloader

The Flash Loader Downloader is the part of the Flash Loader that users add to their own projects. It implements the communications protocols, error checking, and the state machine that handles Flash Loader operations. The Flash Loader Downloader consists of six source files which will be covered in detail.

3.1.1 r_fl_app_header.c

Information about the Load Image you are currently working on is stored in this file. There is a structure named *g_fl_cur_app_header* that holds this information. The members of this structure correspond to the entries in a Load File Header (Appendix A). This structure is the only source of communications between the Load Image currently running on the Device and the Flash Loader Bootloader. The Bootloader can read this structure and find out information about the Load Image used for error checking and for checking against other Load Images.

3.1.2 r_fl_comm_*type*.c

The code associated with enabling communications between the Host and the Device is stored in this file. The *type* portion of the name indicates what communications medium is being used. The default Flash Loader implementation that comes packaged with this application note uses asynchronous serial for communications therefore the name of the file is *r_fl_comm_uart.c*. The functions in this file are:

1. **fl_com_init**: Initializes everything needed to start communications with the host.
2. **fl_com_receive**: Sets up the communications medium for receiving next data.
3. **fl_com_bytes_received**: Reports how many bytes have been received since fl_com_receive was last called.
4. **fl_com_transmit**: Used to transmit data to the host.
5. **fl_com_send_status:** Returns whether the last transmission is complete or not.

3.1.3 r_fl_downloader.c

This file contains the Flash Loader state machine and 2 public API functions: R_FL_DownloaderInit() and R_FL_StateMachine(). The state machine handles the communication protocols and commands from the host. There are configuration options that can be controlled in the associated header file *r_fl_downloader.h*. The functions in this file are:

1. **R_FL_DownloaderInit**: Initializes the state machine and calls functions to initialize communications and storage.
2. **fl_receive_reset**: Resets reception after a timeout has occurred.
3. **R_FL_StateMachine**: The Flash Loader state machine. Controls what operations happen when commands are received from the host. This is where the communications protocols are implemented.

The Flash Loader state machine has the option to use timeouts. By default, timeouts are enabled. Timeouts allow the Flash Loader state machine to reset when the state machine is stuck in a certain state because the Host for some reason could not continue communications. Controlling whether timeouts are enabled and changing the timeout period are discussed in Section 2.7.

3.1.4 r_fl_store_manager.c

The file *r_fl_store_manager.c* contains functions that implement a simple file system on the Flash Loader Storage. *r_fl_store_manager.c* relies on low-level functions in *r_fl_memory_*type*.c* for access to the Flash Loader Storage. This file also has functions for handling retries after a communications error has occurred.

One of the goals of the Flash Loader project was that the code should not interfere with the user's application code. In order to do this some Flash Loader Storage operations were split up. This means that for an erase there could be a function to start the erase, a function to check the erase, and a function to finish the erase. This method was chosen to keep the simplicity of the state machine while not getting in the way of other user code. Because there may be several functions to complete one task, not all of the functions in *r_fl_store_manager.c* are listed below.

1. **fl_is_store_busy :** Returns whether the Flash Loader Storage is currently busy with another operation.
2. **fl_start_erase_load_block :** Starts the erase of a Load Image in Flash Loader Storage.

3. **fl_store_retry_init :** Starts the process of retrying a failed Load Image transfer.
4. **fl_store_block_init :** Starts the process of storing a Load Image into Flash Loader Storage.
5. **fl_get_load_image_headers :** Returns a list of Load Images available in Flash Loader Storage.
6. **fl_verify_load_image :** Runs tests and checks to see if a Load Image in Flash Loader Storage is valid.
7. **fl_get_latest_image :** Returns which Load Image in Flash Loader Storage was the most recent to be downloaded.
8. **fl_find_matching_image :** Checks Load Images in Flash Loader storage to see if there is a match against the information sent in. This is used to make sure a user does not download the same Load Image twice, and for checking to see if a retry is needed.

3.1.5 r_fl_memory_*type*.c

The file *r_fl_memory_*type*.c* contains functions that perform low-level memory operations on the memory used for holding Flash Loader data. While *r_fl_store_manager.c* implements the file system, this file takes care or reading, writing, and erasing actual bytes on the Flash Loader Storage device. The functions contained in this file are:

1. **fl_mem_init**: Initializes resources needed for talking to Flash Loader Storage.
2. **fl_mem_read**: Reads data from Flash Loader Storage.
3. **fl_mem_write**: Writes data to Flash Loader Storage.
4. **fl_mem_erase**: Erases data in Flash Loader Storage.
5. **fl_mem_get_busy**: Returns whether Flash Loader Storage is still busy processing previous operation.

3.1.6 r_fl_utilities.c

The file *r_fl_utilities.c* contains utility functions that may be used in multiple source files and don't necessarily belong in any of the other Flash Loader source files. Users can modify this file if needed and can add their own functions. By default, the functions that come in *r_fl_utilities.c* are:

1. **fl_check_application:** Runs a check on MCU flash (CRC-16 by default) to validate the current running image. If the image in MCU flash is deemed invalid then the Flash Loader Bootloader will wait for a new image.
2. **fl_reset:** Performs an internal reset on the MCU. This is used when the user's application is running and a new image is received. By default the Flash Loader project will reset immediately using the Watchdog Timer (WDT) once the image has been successfully downloaded.
3. **fl_signal:** This is used to signal to the outside world that the MCU does not have a valid image to run and is waiting in the Flash Loader Bootloader for a new image. An example would be to flash an LED, put a message on a LCD, or send out a status command.
4. **fl_check_bootloader_bypass:** This function is used to determine if a user is requesting that the normal Flash Loader Bootloader checking be skipped. If it is skipped then the Flash Loader Bootloader will not check Flash Loader Storage for a new image and will not check the internal Device memory for a valid image. Instead it will go straight into waiting for a new Load Image. This is a helpful bypass for when a bad image has been programmed into the Device. For instance, if the user has a bug in their code that prevents the Flash Loader state machine from running with their application then the Flash Loader Bootloader will still let it run if the image is error free in terms of a valid byte-for-byte image. This bypass effectively allows you to wipe out the bad Load Image without having to erase the Device's internal memory manually. An example of how to implement a bypass would be to have the user hold several buttons at once when resetting the Device.
5. **R_FL_GetVersion:** Returns the current version of the Flash Loader code.

3.2 Flash Loader Bootloader

The Flash Loader Bootloader is the part of the project that uses the Load Images downloaded from the Host. Once the Device has successfully downloaded a new Load Image it will reset when it is ready to update. After the reset the Flash Loader Bootloader is run before any user application code. The Flash Loader Bootloader will look for a new Load Image. If a new Load Image is found the bootloader will validate the image and then program it into the Device's internal memory. If a new Load Image is not found, or if the image is found to be invalid, then the bootloader will look for an image already programmed into the Device's internal memory. If a Load Image Header is found then the application will be validated in the Device's internal memory. If the image is valid the bootloader will jump to the application. If the image is not found to be valid then the code will signal that a valid image has not been found, start the Flash Loader state machine, and will wait for a new Load Image download request. To see a diagram of the execution flow of the Flash Loader Bootloader refer to Appendix C.

3.2.1 Using User Boot Mode

The Flash Loader project was designed around the use of a feature in Renesas MCUs referred to as User Boot Mode. Most Renesas MCUs can reset into at least 2 modes. The two most common modes are Boot Mode and Single-Chip Mode. In Boot Mode a factory programmed kernel is run that allows the user to program the MCU's internal memory across a serial interface. No user code can be run from Boot Mode. Single-chip mode is the 'regular' mode where an address is fetched from a location in memory and that is where the MCU will start executing after reset. A bootloader can work in Single-Chip Mode but it makes it more difficult since the user's application needs to be aware of the bootloader. The user will have to do things like change the vector table to make sure that their user application does not interfere with the reset vector the bootloader has already programmed in. Another cause of concern is that bootloaders erase internal memory. If the bootloader is residing in the same memory as the user's application then if the code is not written properly, or if there is an error during erasure/programming, then the bootloader could be compromised.

The Flash Loader project gets around these issues by using User Boot Mode. User Boot Mode is designed to make using custom bootloaders easier. Features that are typical of User Boot Mode are having a separate reset vector (or static reset location) and having a separate memory area. Figure 3-1 graphically shows the User Boot Mode memory area and reset vector. After mentioning the drawbacks of using a bootloader in Single-Chip Mode previously, you can see how User Boot Mode removes these problems. The user's application does have to be cautious of the bootloader because it has its own reset vector that will not interfere. Accidentally erasing or overwriting the bootloader is not a fear either because when executing out of the User Application space ('program ROM' in Figure 3-1) the Device is not allowed to erase or program the User Boot area. The User Boot area can only be programmed or erased when the MCU is put into regular Boot Mode.

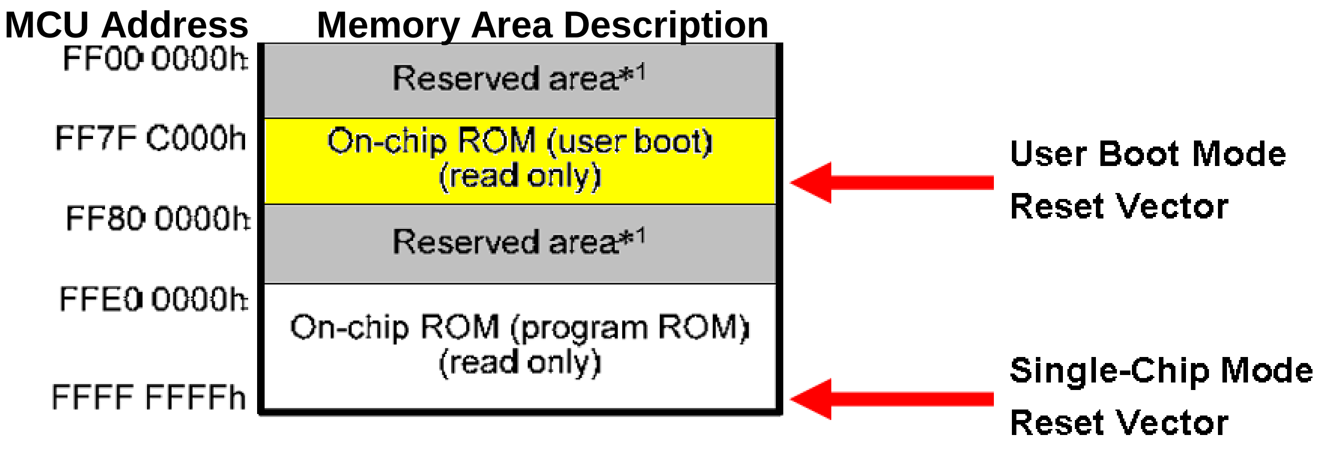

MCU Address
Memory Area Description
FF00 0000h
FF7F C000h
FF80 0000h
FFE0 0000h
FFFF FFFFh
Reserved area*1
On-chip ROM (user boot) (read only)
Reserved area*1
On-chip ROM (program ROM) (read only)
User Boot Mode Reset Vector
Single-Chip Mode Reset Vector

Figure 3-1 : User Boot Mode

If your MCU does not have User Boot Mode this does not mean that you cannot use the Flash Loader project. It does mean that the user will have to make modifications to the default project to handle the issues of keeping the bootloader in user program space (like the issues mentioned previously).

3.2.2 r_fl_bootloader.c

r_fl_bootloader.c contains all of the functions required by the bootloader. These functions are:

- **main –** This is the main function of the bootloader. This is where all of the important bootloader decisions are made. This follows the diagram displayed in Appendix C.
- **fl_write_new_image –** After a new Load Image has been found and verified the bootloader will program it into the Device's internal memory. This function takes in a Load Image and performs the programming.
- **fl_process_write_buffer –** This function writes a Data Block into the Device's internal memory.
- **fl_flush_write_buffer –** Program pages can span blocks. This function needs to be called to flush the buffer in the end.

To be able to erase and write the Device's internal memory the Flash Loader project uses the Renesas supplied Simple Flash API. The Flash API for your device can be found on the Renesas website.

4. Flash Loader on the Host

4.1 Preparing to Use Host Applications

With the default Flash Loader implementation the Host is a PC. The Host applications are written in Python which means it is easy to read, modify, and move to other platforms that support Python. Python 2.7.2 was used for the development of this application note. Any version of Python 2 that is 2.7 or later should work for these applications. Python 2.* can be downloaded from the Python website here:

http://www.python.org/download/

Python 3.* may work with these applications but this has not been tested.

Along with the standard Python libraries, 2 external libraries were also used.

4.1.1 crcmod

The first library is named crcmod and it is used for generating CRC codes for the MOT converter application discussed in Section 4.2. This library can be downloaded from the following link:

http://crcmod.sourceforge.net/

Please read the 'Installation' section on the website for installing this library. At the time this application note was published, the link to the 'Installation' section was the following:

http://crcmod.sourceforge.net/intro.html#installation

4.1.2 pySerial

pySerial is a Python library that is used for accessing the serial port on a PC. It is used by the Host application that transfers a Load Image to the Device. This application is discussed in Section 4.3. This library can be downloaded from the following link:

http://pyserial.sourceforge.net/

Please read the 'Installation' section on the website for installing this library. At the time this application note was published, the link to the 'Installation' section was the following:

http://pyserial.sourceforge.net/pyserial.html#installation

4.2 Making a Load Image

In this section we will talk about the Host application that converts S-Record files into Load Images. The reason for using Load Images and the format used are explained in Section 1.1.2. The S-Record converter that comes standard with the Flash Loader project is named *r_fl_mot_converter.py*. It is a command line application that takes in a number of arguments. Descriptions of the input parameters for the application are shown in Table 4-1. While this application comes standard, it does not have to be used. The protocols used by the Flash Loader project are supplied which means the user could write their own application if they wished. The user would only need to modify this application if they wanted to change something in the protocol/format used for Load Images.

Input Parameter	Description
-i, --input	The path to the input S-Record file
-o, --output [*1]	The name and path desired for the output Load Image. This parameter is optional and if not supplied then the default value will be the original file name with its file extension removed and '.bch' added.
-d, --data_size [*1]	This controls the maximum size of the data field in an individual Data Block. You would want to change this parameter to optimize throughput on your communications medium. For instance, if you are using a wireless connection then you would probably want the block size to be small so that if an error occurs on a block (packet) then you can retry and not lose a lot of previous progress. On the other hand, if you have a wired connection then you might want to use a larger block size to decrease block overhead (header information, CRC, pointer to next block, etc). This value also corresponds to the size of the receive buffer on the Device that will be receiving the file. The Device cannot receive a Load Image if that image's max block size is larger than its internal receive buffer. The size of the Device's internal receive buffer is controlled by the FL_CFG_DATA_BLOCK_MAX_BYTES #define located in *r_flash_loader_rx_config.h*. This parameter is optional and if not supplied then the default value will be 2048.
-f, --fill_space [*1]	This option allows you to merge close blocks of data by putting 0xFF's in between them. The reason you would want to do this would be to save on the overhead of separate blocks when the space in between two blocks is relatively close. For example, if you had two blocks, each 10 bytes wide, that were separated by 2 bytes of empty space then it would be more efficient to have one block and put 2 bytes of 0xFF's in between the data than to have two separate blocks. This parameter allows you to specify the maximum bytes between two blocks that you want to fill. This parameter is optional and if not supplied then the default value will be 64.
-l, --location [*1]	This specifies the address of the structure g_fl_cur_app_header. The MOT File Converter needs to know this so that it can read the ID and version number of the Load Image and fill in the empty fields. You can see which fields are handled by the MOT File Converter by looking in the file *r_fl_app_header.c* and looking for the comment '(Handled by file converter)' above a structure member. This parameter is optional and if not supplied then the default value will be 0xFFFFFE00.
-m, --mask [*1]	This specifies the value the user used for the valid mask in the g_fl_cur_app_header structure. This enables the application to verify that a application header was actually found. This value should match the definition for the FL_LI_VALID_MASK macro. This parameter is optional and if not supplied then the default value will be 0xAA.
--formatting	This is a flag, not a parameter. If you use this option then the script will return information on the layout of the binary file. Headers, blocks, and other details are described.

Table 4-1 : r_fl_mot_converter.py Input Parameters

[1] This parameter is optional

4.3 Transferring a Load Image

The *r_fl_serial_flash_loader.py* application is used by the Host to transmit a Load Image from the Host to the Device. This application is written in Python which has advantages discussed in Section 4.1. By default this application will transmit using asynchronous serial. If the user decides to use another communications medium, then they will either need to modify this file, or write their own. This is encouraged and is the reason the communications protocols are supplied with the Flash Loader project. If the user writes their own application the input parameters will likely change, but the parameters used with the default application are explained below in Table 4-2.

Input Parameter	Description
`-p, --port`	The port number to use for communications (e.g. to use COM4 you would use '-p 3' since COM1 = '-p 0')
`-c, --command` [2]	The command you want to send to the Device. Available commands are: • **'info'** - This is an information request that responds back with what image is currently running on the Device and what Load Images are residing in Flash Loader Storage. • **'load'** - This command requests permission and transmits a Load Image to the Device. The Load Image transferred will be stored in Flash Loader Storage. • **'erase'** - This command will erase Load Images from Flash Loader Storage. This parameter is optional and if not supplied then the 'info' command will be used.
`-f, --file`	If the command specified is a Load command then this parameter is the path to the Load Image you want to transfer to the Device
`-q, --quiet` [2]	If specified some messages will be suppressed.
`-b, --block`	If the command specified is an Erase command then this parameter will specify which Load Block should be erased.

Table 4-2 : *r_fl_serial_flash_loader*.py Input Parameters

[2] This parameter is optional

4.4 Host Applications Example

Below is an example showing how to use the Host applications to make a Load Image and transfer it to the Device. Listed below is the information we have:

- Input File = input.mot
- Desired output file = load_image.bch
- Maximum data block size = 2048 bytes
- Fill space if space between blocks is less than or equal to 50 bytes
- The application header is at flash address 0xFFFFFE00
- COM4 is used on the Host

We'll start by converting the S-Record file into a Load Image. Note that these examples assume that python.exe has been added to the system path. If this has not been done then the user will need to specify the location of python.exe instead of just calling 'python'. An example would be 'C:\Python27\python.exe r_fl_mot_converter.py …'.

```
>>python r_fl_mot_converter.py –i input.mot –o load_image.bch –d 2048 –f 50 –l
  0xFFFFFE00
```

Now that we have a Load Image we need to check to see if the Device has an open spot in Flash Loader Storage. We can get information about the status of Flash Loader Storage by issuing an Information command.

```
>>python r_fl_serial_flash_loader.py –p 3 –c info
```

Let's now assume that the Information command came back and all the spots were full. We can delete the first spot using an erase command.

```
>>python r_fl_serial_flash_loader.py –p 3 –c erase –b 0
```

After the erase has finished we can send a command to download the new Load Image.

```
>>python r_fl_serial_flash_loader.py –p 3 –c load –f load_image.bch
```

5. Implementing Flash Loader with Your Project

This section will cover making the Flash Loader project work with your system. Each sub-section will cover a topic that will need to be addressed when modifying the project to meet your requirements. The example projects that come with this application note and are referenced throughout this section have these properties:

- RX62N/RX63N MCU for Device
    - Has User Boot Mode and User Boot area
    - Has 16-bit CRC Generator
- PC for Host
- Asynchronous Serial for communications between Host and Device
- External SPI Flash Memory for Flash Loader Storage
- 16-bit CRC-CCITT for error checking
- Using Renesas High-Performance Embedded Workshop (HEW) or E2Studio for IDE
- Using Renesas RX Toolchain

5.1 Modifying Flash Loader Device Source Files

In this section we will go through each file and discuss what, if anything, needs to be changed and why you would change it. For a general overview of each file refer to Section 3.1.

5.1.1 r_fl_app_header.c

r_fl_app_header.c contains information about this application image. By default, the only two values that need to be changed in the g_fl_cur_app_header structure are the Image ID (2nd entry) and the Version Number (3rd entry). For more information on each structure entry refer to Appendix A. The user can set these two entries to anything they want. The intent of these two entries was to have the Image ID be static for a particular application with the Version Number updating for each revision. When setting the entries, users should take care to recognize the format of each member. The Image ID and Version Number are 1 byte unsigned entries for example. The fl_image_header_t structure is defined in *r_fl_types.h*.

The other entries in the structure are handled by other applications. More information on these entries is below.

- **Size of Load Image** [*3] – This entry represents the total size of the Load Image in bytes.
- **Max Block Data Size** [*3] – This entry represents the maximum number of bytes that will be used for the data portion of a Data Block.
- **Load Image CRC** [*3] – This entry holds the CRC of the Load Image. The CRC code covers all Data Block headers and data. This CRC code is used for validating Load Images as a whole and is used in the bootloader to validate a Load Image before its data is written into the Device. If the user decides to use another means for error checking, this entry will need to be changed.
- **Raw CRC** – This entry holds the CRC of the actual application image as it sits in the Device's memory. This CRC is used for checking a firmware image that has already been programmed into the Device. This CRC is created using every byte of the Device's user application memory area. This is not handled by any Flash Loader application. In the example Flash Loader project this CRC value is calculated using the RX Toolchain's linker. For more information on how this is done refer to Section 5.4.1**.** If the user does not wish to use this method, or if the toolchain they are using does not support the output of a CRC calculation, then they will need to implement it themselves. A Host or Device-side application could be used to compute this value. For example, if this is done on the Device-side then the same function that is used to check the image CRC could be used to obtain the CRC in the first place.

[3] This entry is handled by *r_fl_mot_converter.py*. If the user does not use *r_fl_mot_converter.py* to make their Load Image then they will need to either hard code this value or implement this functionality in their own application.

- **Address of 1st Data Block** – This entry holds the address of the first data block. Each Data Block header will then have the address of the next block. This is handled by *r_fl_store_manager.c* code when the Load Image is stored in Flash Loader Storage.
- **Successfully Stored** – This value is written by *r_fl_store_manager.c* code after the Load Image has been successfully downloaded and stored into Flash Loader Storage. This entry is used to determine Load Image priority (higher number Load Images are chosen first) and to make sure images were successfully downloaded.

Once the g_fl_cur_app_header structure has been properly filled out then care must also be taken to make sure the structure is put in the proper place in memory. The Flash Loader Bootloader and MOT File Converter both look at a predefined memory address for the g_fl_cur_app_header structure. By default this location is 0xFFFFFE00. If the user changes the location of this structure then they will need to make sure and change the address throughout the Flash Loader project.

5.1.2 r_fl_comm_*type*.c

r_fl_comm_*type.c* contains the Device code for communicating with the Host. The *type* portion of the filename should correspond to what communications medium is being used. The Flash Loader project comes packaged with a UART implementation and the file is named r_fl_comm_uart.c. If the user wishes to use a different communications medium then they will need to create a new r_fl_comm_*type*.c source file and implement the functions prototyped in *r_fl_comm.h*.The basic needs are to be able to transmit a buffer, receive into a buffer, and report back on how many bytes have been received.

5.1.3 r_fl_downloader.c

The state machine for the Flash Loader project is contained in *r_fl_downloader.c*. This file will not be changed by the user unless they want to change the protocols that are used in Flash Loader project. For instance, if the user wanted to add a new command to the communications protocol then they would add that capability in *r_fl_downloader.c*.

5.1.4 r_fl_memory_*type*.c

The low-level communications routines that are used to communicate with the Flash Loader Storage are contained in this source file. The *type* portion of the filename should correspond to what communications medium is being used. The Flash Loader project comes packaged with a SPI flash implementation and the file is named *r_fl_memory_spi_flash.c*. If the user wishes to use a different communications medium then they will need to create a new *r_fl_memory_*type*.c* source file and implement the functions prototyped in *r_fl_memory.h*. The functions in *r_fl_memory_*type*.c* offer the basic functions of reading, writing, and erasing portions of the Flash Loader Storage device.

If the user is going to use a SPI flash for their Flash Loader Storage then they will need to define the specifics for that SPI flash in the g_fl_li_mem_info structure in *r_fl_memory_spi_flash.c*. This structure has the following members:

- **erase_size** – This is the minimum erase size capable of the memory device. This is important when it comes to the ability to retry Load Image transfers. If the erase size is large, this means that more progress might be lost when a retry occurs. An example is shown below in Figure 5-1. In this example each memory is shown with how the data blocks line up in erase sectors. The first memory device with the larger minimum erase size is forced to put multiple blocks inside of each erase sector. The #2 memory device, with the smaller minimum erase size, is able to put each data block in its own erase sector. In the example, Data Block 20 has an error. The #2 memory device in this case will only have to erase sector 20 and will ask the Host to send Data Block 20 again. The device with the larger minimum erase size on the other hand will have to erase sector 1 which means Data Blocks 11-20 will all be deleted. In this case the Device will have to ask the Host to start again with Data Block 11. This is a limitation of using memory devices with large erase sectors.

#1 - Memory Device with Large Minimum Erase Size

Erase Sector 0	Erase Sector 1		Erase Sector 2
Data Block 10	Data Blocks 11-19	Data Block 20 (Bad)	
No Erase	Must Erase	Must Erase	

#2 - Memory Device with Small Minimum Erase Size

Erase Sector 17	Erase Sector 18	Erase Sector 19	Erase Sector 20	Erase Sector 21
Data Block 17	Data Block 18	Data Block 19	Data Block 20 (Bad)	
No Erase	No Erase	No Erase	Must Erase	

Figure 5-1 : Minimum Erase Size

- **max_program_size** – Different types of memory devices vary in how much can be programmed in one operation. This structure member specifies the maximum number of bytes that can be programmed with one program command. Starting with v3.0 of the Flash Loader project, the SPI flash driver code has been updated so that it will perform multiple program commands if necessary. For example, if 128 bytes is sent in to program and the device can only program 64 bytes per program command then 2 program commands will be issued. This means that this member will only dictate when to split up program operations if the user wants to do so. If using a different driver, then this member may be a requirement.

- **addresses[] –** This array contains starting addresses for Load Images. If the user specifies that they want to store 2 Load Images then this array will be 3 entries long. The 1st entry will be the starting address of the 1st Load Image and the 2nd entry will be the starting address for the 2nd Load Image. The 3rd entry is used to store the maximum address for Load Image data and is used for checking purposes.

5.1.5 r_fl_store_manager.c

r_fl_store_manager.c contains functions that implement a simple file system on the Flash Loader Storage device. This file interacts with the Flash Loader Storage device using function calls found in *r_fl_memory_*type*.c*. This separation makes the code in *r_fl_store_manager.c* portable and therefore should not be modified by the user unless they want to change the way the file system works.

5.1.6 r_fl_utilities.c

As mentioned in Section 3.1.6 *r_fl_utilities.c* contains functions that do not necessarily belong in other Flash Loader source files. The user can modify this file to add or remove functions as they see fit.

5.2 Modifying Flash Loader Host Applications

All of the Flash Loader application code that runs on the Host is written in Python. Users can modify this code to meet their needs or they can use their own code. Users should feel free to use their own code as long as the code matches the Flash Loader protocols for communications and the format for Load Images.

5.3 Making a Flash Loader Bootloader Project

5.3.1 Choosing a Communications Medium

Making the Flash Loader Bootloader project is very similar to building your user application. One of the first things the user has to decide is whether or not they are going to include the same communications medium in the bootloader as their user application. For example, if the user application is going to perform Flash Loader communications over USB, will the Flash Loader Bootloader also use USB? The Flash Loader Bootloader does not require that it have any communications at all. If the bootloader does not have its own communication methods then it will require that the Load Images be acquired by some other means (e.g. the user application). This is not the safest option because if an error occurs and no Load Image is present then the bootloader can do nothing. If the regular Flash Loader state machine is also included in the bootloader (as is done in the example project) then there is always a failsafe since the User Boot area can only be erased in Boot Mode.

The main limitation with including the same communications medium in the Flash Loader Bootloader is size. The User Boot area on most devices is usually relatively small compared to the User Application Area which means it might not be possible to use the same communications method. For example, if your USB stack requires 40kB then it will not fit in the 16kB User Boot area which many RX devices have. One option in this case is to use a different communications medium for the bootloader.

5.3.2 Building the Project & Using r_fl_bootloader.c

After choosing the communications medium the user can move on to building their Flash Loader Bootloader project. If the user is including the Flash Loader state machine then they should follow the directions in Section 2.8.1. The bootloader source code in *r_fl_bootloader.c* will not need to be changed unless the user wants to change part of the bootloader protocol. There are other important features that will need to be addressed:

1. **Setting the User Boot area Reset Vector –** As discussed in Section 3.2.1 the User Boot area sometimes has its own reset vector. If this is the case then the user will need to make sure and put the correct value in this address. In the default Flash Loader implementation, this is taken care of in the file *vecttbl.c* which is located in the r_bsp package. The user could do this themselves as well by using code similar to the following:

```
#pragma address UB_ResetVector = 0xFF7FFFFC
void * const UB_ResetVector[] = {
/* User Boot Reset Vector */
  (void *) PowerON_Reset_PC
};
```

   This code takes the address of the function 'PowerON_Reset_PC' and puts it in the User Boot area Reset Vector location. When the Device is reset into User Boot Mode this is the address that will be fetched.

2. **Jumping to the User Application –** After the bootloader has verified the image currently in the Device's internal memory it will jump to the user application. This is done by looking at the reset vector used in Single-Chip Mode. There are two #define's used to add this functionality:

   - **#define MCU_RESET_VECTOR –** This #define gives the memory address for where the reset vector for the user application can be found. This value will need to be changed depending on where the user puts their application's reset vector. This does not have to match the reset vector of the Device but it is convenient to do this so that in the case that the Device is started in Single-Chip Mode it will still have an application to run.

- **#define JUMP_TO_APPLICATION –** This #define makes a function call out of the address defined in MCU_RESET_VECTOR. When the bootloader is ready to jump to the user application it can just call JUMP_TO_APPLICATION(). This #define will only need to be changed if the user's Device does not use 32-bit addressing.

3. **Setting Bootloader Location –** The bootloader should reside in the User Boot area so the user should make sure to configure the linker to place the code at the correct memory location.

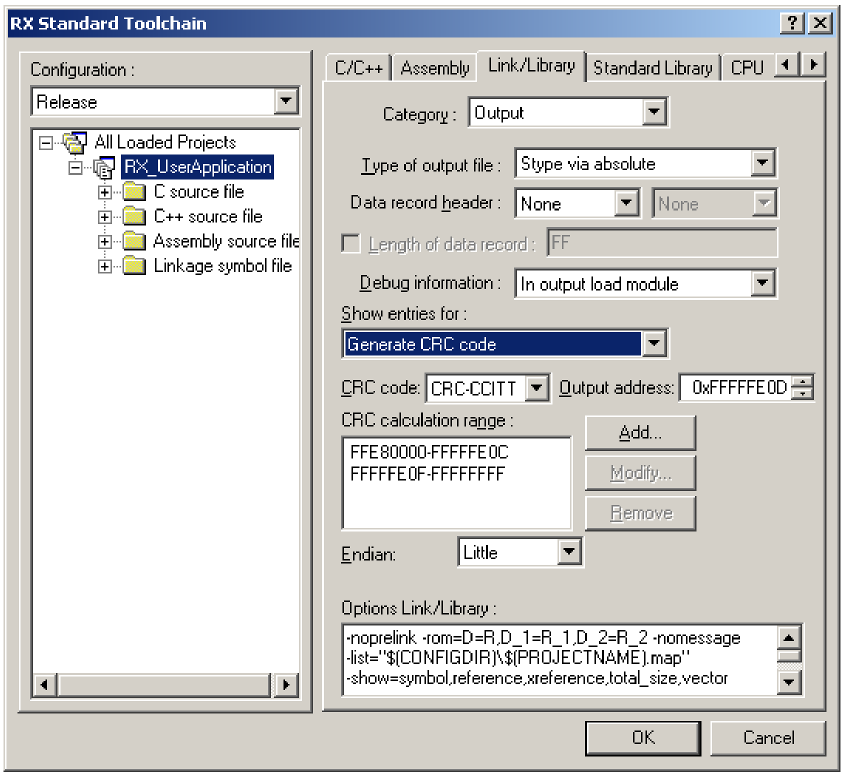
5.4 Usage Notes

5.4.1 Application Code CRC Generation with Renesas RX Linker

The 'Raw CRC' as referenced in Section 5.1.1 is generated by the Renesas RX Linker in the example Flash Loader project that comes with this application note. This, along with the fact that the RX has a hardware 16-bit CRC Generator peripheral, was the reason that 16-bit CRC-CCITT was chosen for error checking. This is also the reason that there are two CRC seeds used by default. In the file *r_fl_utilities.h* there are two CRC seeds: FL_CRC_SEED and RX_LINKER_SEED. The Renesas RX linker uses RX_LINKER_SEED to seed its CRC calculation and also does a bitwise NOT operation at the end. CRC calculations that are done by applications within the Flash Loader project (Host and Device) use FL_CRC_SEED for the seed value and do not perform a bitwise NOT at the end.

To get to the CRC calculation options within HEW do the following:

1. With your HEW project open go to Build >> RX Standard Toolchain'
2. Click on the 'Link/Library' tab.
3. Change 'Category' to 'Output'
4. Change 'Show entries for' to 'Generate CRC Code'



Configuration : Release

Category : Output

Type of output file : Stype via absolute

Data record header : None None

Length of data record : FF

Debug information : In output load module

Show entries for : Generate CRC code

CRC code: CRC-CCITT Output address: 0xFFFFFE0D

CRC calculation range :
FFE80000-FFFFFE0C
FFFFFE0F-FFFFFFFF

Endian: Little

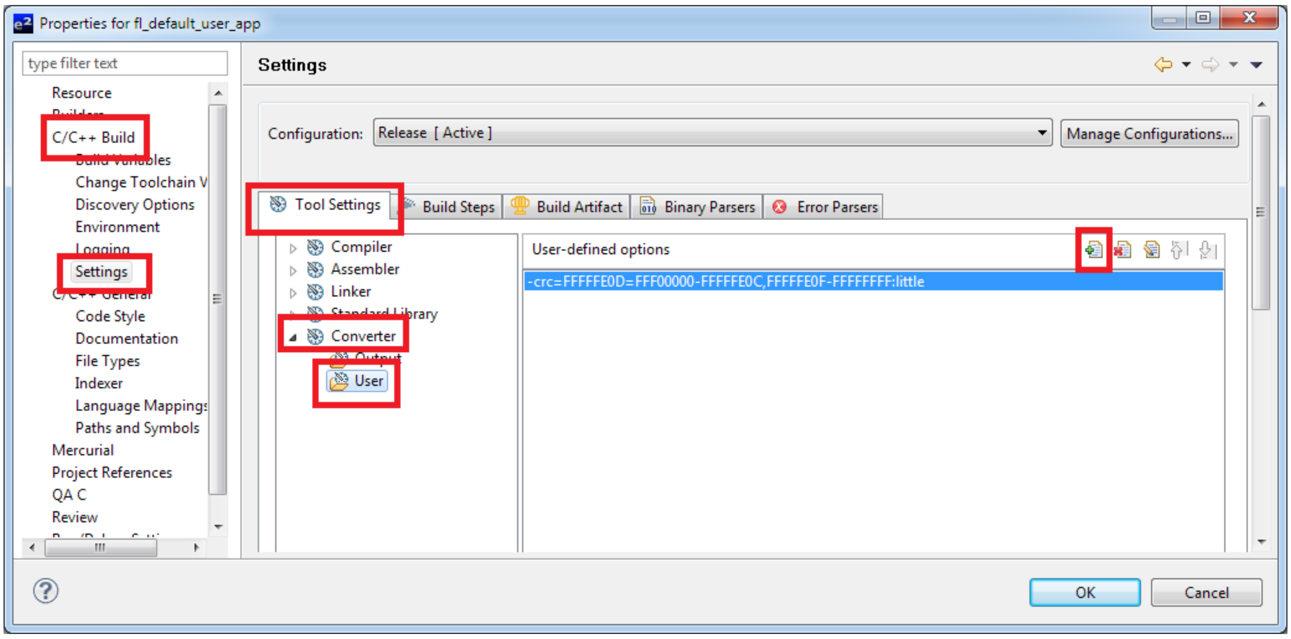
Options Link/Library :
-noprelink -rom=D=R,D_1=R_1,D_2=R_2 -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map"
-show=symbol,reference,xreference,total_size,vector

Figure 5-2 : Example CRC Output By Linker

Now you can choose what kind of CRC calculation you want to use, where to put the result, and what ranges of memory to use for the calculation. Figure 5-2 shows an example setup. Notice that the 'Output address' is not included in the CRC calculation. This is because this address is where the linker will place the result of the CRC calculation when it is done. The value for 'Output address' relates to the location of the 'Raw CRC' value in the g_fl_cur_app_header structure in *r_fl_app_header.c*.

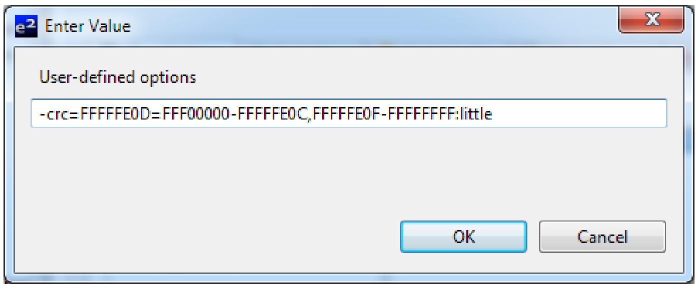
If using E2Studio, the user will need to input the command to generate the CRC manually. Please reference the RX Family C/C++ Compiler Package User's Manual for more information. To setup the CRC calculation options within E2Studio do the following:

1. Right click on your project folder and click Properties.
2. Navigate to C/C++ Build >> Settings >> Tool Settings >> Converter >> User.



```
-crc=FFFFFE0D=FFF00000-FFFFFE0C,FFFFFE0F-FFFFFFFF:little
```

3. Click the Add button to add a new user-defined option.
4. In the window that pops up, add the command-line argument for the RX Toolchain to generate the proper CRC code. Click OK. For this example the following command was used:
   -crc=FFFFFE0D=FFF00000-FFFFFE0C,FFFFFE0F-FFFFFFFF:little
   This specifies to put the result at 0xFFFFFE0D and that the calculated range should be from 0xFFF00000 to 0xFFFFFFFF while skipping the location where the generated CRC will be placed (0xFFFFFE0D-0xFFFFFE0E). The 'little' option specifies to store the CRC little-endian.



```
-crc=FFFFFE0D=FFF00000-FFFFFE0C,FFFFFE0F-FFFFFFFF:little
```

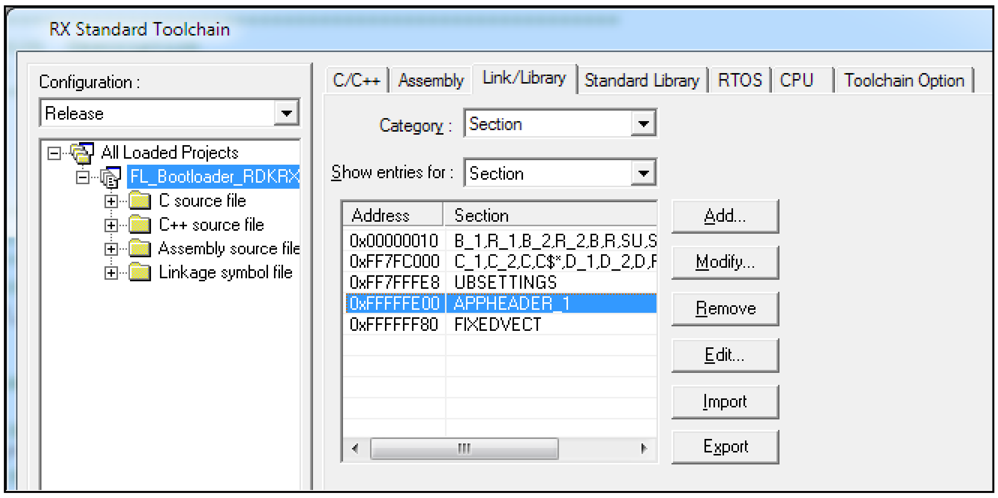
5. Scroll to the bottom of the Properties window and click Apply.

If the toolchain you use for your project does not have the option to output a CRC code then you can still write one yourself. As was discussed above you will still need to make sure to output the CRC result in the correct location in memory and remember to not include the spot in memory where the CRC calculation will be placed.

5.4.2 Setting Address of g_fl_cur_app_header

In *r_fl_app_header.c* the structure g_fl_cur_app_header is placed in a constant section named APPHEADER. This is done using the Renesas RX Toolchain compiler directive '#pragma section C APPHEADER'. The 'C' option specifies that this is constant data. Since a unique section name is used, the user must specify the address of this section in their linker settings. The RX linker allows each section to have 1-byte, 2-byte, and 4-byte aligned subsections where each is differentiated in the linker setup by adding '_1', '_2', or '_4' respectively to the end of the section name. Since the g_fl_cur_app_header structure is specified to be packed (1-byte aligned, no padding bytes) in *r_fl_types.h* the section name that should be added to the linker is APPHEADER_1. The Flash Loader project by default, places the APPHEADER_1 section at address 0xFFFFFE00. The APPHEADER_1 section must be specified at the same address in the linker settings for both the bootloader and user application projects. The section is used by the user application for placing the structure in memory at a specific location. The section is defined in the bootloader application so that the bootloader knows where to look for the image header.

```
#pragma section C APPHEADER
```



Configuration : Release

All Loaded Projects
- FL_Bootloader_RDKRX
  - C source file
  - C++ source file
  - Assembly source file
  - Linkage symbol file

Category : Section

Show entries for : Section

Address	Section
0x00000010	B_1,R_1,B_2,R_2,B,R,SU,S
0xFF7FC000	C_1,C_2,C,C$*,D_1,D_2,D,F
0xFF7FFFE8	UBSETTINGS
0xFFFFFE00	APPHEADER_1
0xFFFFFF80	FIXEDVECT

5.4.3 Programming Bootloader into User Boot area

Renesas' Flash Development Toolkit (FDT) is used for programming the Flash Loader Bootloader into the User Boot area. This is done the same as you would program in any other file except for one difference. After adding the bootloader's MOT File to the FDT project the user should right-click on it and select 'User Boot Flash' as shown in Figure 5-3. This tells FDT that the MOT file contains data that should go in the User Boot area. If this option is not selected then FDT will ignore the User Boot area data contained in the MOT file.

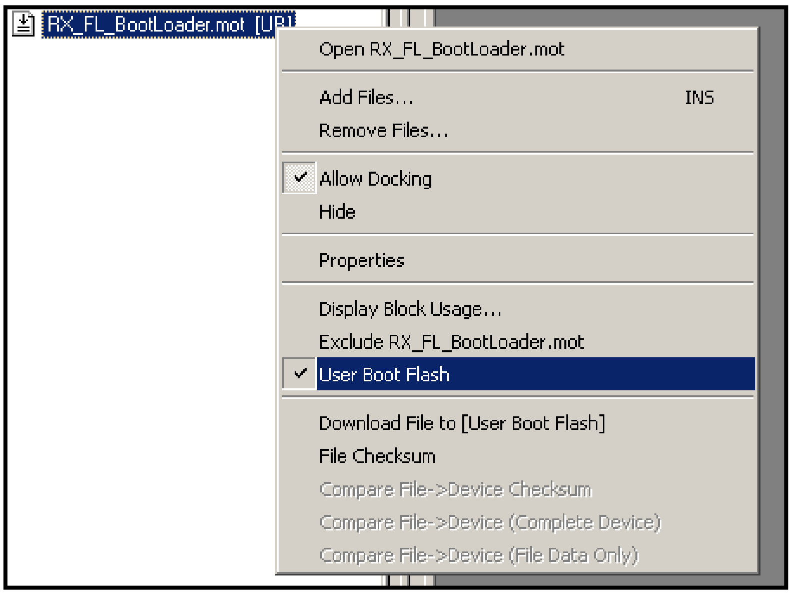

RX_FL_BootLoader.mot [UB]
Open RX_FL_BootLoader.mot
Add Files... INS
Remove Files...
✓ Allow Docking
Hide
Properties
Display Block Usage...
Exclude RX_FL_BootLoader.mot
✓ User Boot Flash
Download File to [User Boot Flash]
File Checksum
Compare File->Device Checksum
Compare File->Device (Complete Device)
Compare File->Device (File Data Only)

Figure 5-3 : Setting up FDT

When programming the User Boot area with FDT you may get a warning saying that the MOT file “exceeds the flash ROM size” of your part. The reason this message is displayed is because the MOT file has data outside of the User Boot area and FDT will not program it in. By default the Flash Loader Bootloader project will define the fixed vector table and will therefore have data outside of the User Boot area. In this case, click ‘Yes’ in the warning dialog. After the download has finished the User Boot area will be programmed but not any other areas of flash. If you wish to program in the other areas, such as the fixed vector table, then uncheck the ‘User Boot Flash’ option that was selected earlier and perform another download. This time the User Boot area will be ignored and the user application flash will be programmed. For more information about using the fixed vector table with the Flash Loader Bootloader please reference Section 5.4.6.

5.4.4 Batch Files Included with Project

There are several batch files that are included with the Flash Loader project. These batch files will most likely not work for your system without some modifications. These batch files are included for convenience. Things that will need to be modified in the files are file locations and port numbers for communications. Once these changes are made the user can add these batch files to their HEW projects and run them by double-clicking them in HEW. This is easier than having to open a command prompt window every time you want to send a Flash Loader command to the Device.

In E2Studio the default action for double-clicking on a batch file is to open it for reading. These batch files can still be used by creating an External Tools configuration. This is accessed in E2Studio by clicking Run >> External Tools >> External Tools Configurations.

5.4.5 Difference between USB Boot Mode and User Boot Mode

USB Boot Mode and User Boot Mode are the same operating mode and both modes use the User Boot area. The name ‘USB Boot Mode’ is used because when a RX with USB support comes from the factory it will have a USB bootloader stored in the User Boot area. In order to use the Flash Loader project the User Boot area will need to be erased which will erase the default USB bootloader. If the user has erased the factory USB bootloader and wishes to get it back, a binary image for their MCU can be downloaded off of the Renesas website.

5.4.6 Bootloader and Fixed Vector Table

The default Flash Loader Bootloader defines the fixed vector table (FVT). Some people see this as a problem because this area will be erased and rewritten when a firmware image is programmed in. The problem is that a good rule of thumb is that you should never erase an active vector. This is true and this was not overlooked in the design of the project. The reason the FVT is defined in the default bootloader project is that the FVT has exception vectors that cannot be turned off. While a bootloader should be well tested and free of exceptions, having a failsafe will appeal to some users. Also, having the FVT defined may be helpful when debugging the bootloader to properly catch any exceptions.

If the user wishes to remove the FVT from their bootloader application then they can do so by deleting the Fixed_Vectors[] array in the file *vecttbl.c*. If this is done then the user must ensure that none of the exceptions in the FVT are triggered. If they are triggered then the MCU will fetch an address from memory that is either erased or has unknown contents.

5.4.7 Getting into User Boot Mode with RX63x and RX200 MCUs

To get into User Boot Mode with RX610 and RX62x MCUs, the user needed to correctly drive the MD0 and MD1 pins. Starting with the RX63x and RX200 MCUs, the user must drive the MD and PC7 pins as well as set the appropriate UB Codes in the Option-Setting Memory registers. These registers are discussed in the hardware manual under the Option-Setting Memory >> UB Codes section. Basically, some constant values have to be put in a flash location to enter User Boot Mode. If the user is using the r_bsp package (the workspace that comes with this application note does) then this is taken care of in the source file *vecttbl.c*.

5.4.8 Calling Flash Loader State Machine

Flash Loader versions previous to v3.0 had code internal to the Flash Loader project that would setup a timer and call the Flash Loader state machine from an interrupt service routine. Starting with v3.0, calling the state machine is now the responsibility of the user. This was done because many users’ systems will have their own timer ticks and there is no reason for the Flash Loader to ‘waste’ a resource that the user may already have dedicated for another task. The default Flash Loader workspace that comes packaged with this project does include timer code (through r_cmt_rx package) to call the state machine but it is external to the Flash Loader project.

5.4.9 Enabling User Boot Mode in r_bsp Package

The Flash Loader project is built using the Renesas Board Support Package (r_bsp). This package provides all the code needed to bring the MCU out of reset and get it ready to run the user's application. After starting up the MCU the last thing the r_bsp code will do is call main(). The r_bsp package is configured using the file *r_bsp_config.h*. Since the bootloader uses the User Boot Area, which has its own reset vector, the r_bsp package must know to put the reset vector at the correct location. The user configures the r_bsp to do this using the macro BSP_CFG_USER_BOOT_ENABLE in *r_bsp_config.h*. Set this macro to '1' to enable user boot mode. Note that this macro should be set to '1' for the Flash Loader Bootloader and to '0' for the Flash Loader User Application.

5.4.10 Calling Flash API from User Boot Area

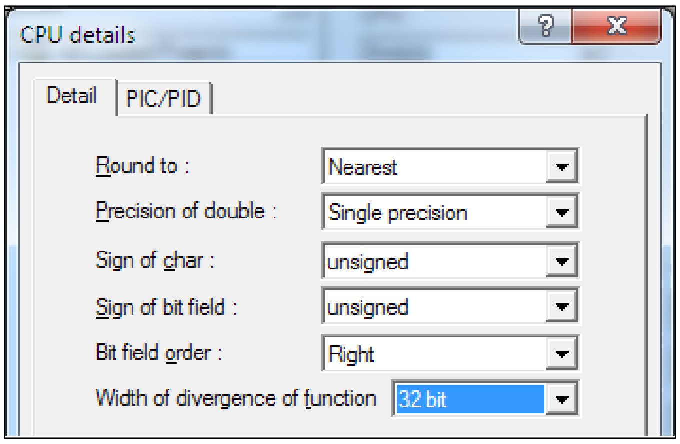
By default, the Renesas RX Toolchain will use 24 bits for the maximum distance that a branch instruction can jump to. The options are 16, 24, or 32 bits. The smaller the chosen value, the smaller the compiled code will be. The reason for this is that if you choose 16 or 24 bits then you are guaranteeing the Renesas RX Toolchain that the destination of all branches will be within the range specified and therefore the compiler does not need to reserve 32 bits for branches. With this guarantee the toolchain can save 1 byte per branch with 24 bit offsets or 2 bytes per branch with 16 bit offsets.

With regular applications, 24 bits will usually be fine. When using User Boot Mode with the Flash API though, 32 bit branches are required. The reason for this is that the Flash API has to put some code in RAM when programming or erasing ROM. Since the end of the User Boot Area is 0xFF800000 and the beginning of RAM is 0x00000000 this means there is a distance of 0x800000. This appears to be within 24 bits, but branches can have positive or negative offsets so the offset is a 2's complement number and therefore the range is half in each direction. This means that calls to Flash API routines in the User Boot Area cannot reach the functions in RAM with 24 bit branches. If the user does not change this setting in the Renesas RX Toolchain then the user will get a 'L2330 (E) Relocation size overflow' error.

Note that this setting only applies to the Flash Loader Bootloader (since it resides in the User Boot Area) and not the Flash Loader User Application.

To set the Renesas RX Toolchain to use 32 bit branches in HEW, follow these steps:

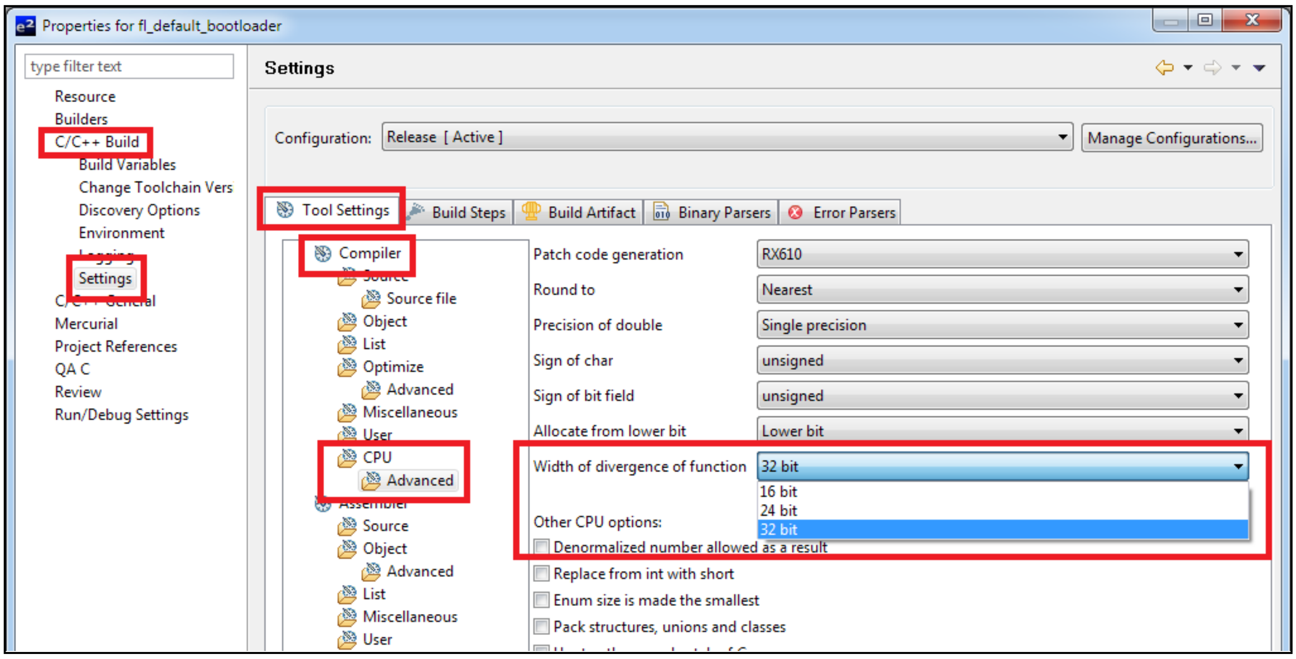
1. Open up your project in HEW.
2. Go to Build >> RX Standard Toolchain.
3. Click the right arrow at the top-right of the 'RX Standard Toolchain' window until you can see the 'CPU' tab.
4. Click the 'CPU' tab.
5. Click the 'Details…' button.
6. In the window that comes up ('CPU details') choose '32 bit' for 'Width of divergence of function'.



Round to : Nearest
Precision of double : Single precision
Sign of char : unsigned
Sign of bit field : unsigned
Bit field order : Right
Width of divergence of function 32 bit

To set the Renesas RX Toolchain to use 32 bit branches in E2Studio, follow these steps:

1. Right click on your project folder and click Properties.
2. Navigate to C/C++ Build >> Settings >> Tool Settings >> CPU >> Advanced.



Properties for fl_default_bootloader

Settings

Configuration: Release [Active]

Patch code generation: RX610
Round to: Nearest
Precision of double: Single precision
Sign of char: unsigned
Sign of bit field: unsigned
Allocate from lower bit: Lower bit
Width of divergence of function: 32 bit

Other CPU options:
- [ ] Denormalized number allowed as a result
- [ ] Replace from int with short
- [ ] Enum size is made the smallest
- [ ] Pack structures, unions and classes

3. Change the dropdown for ‘Width of divergence of function’ to ’32 bit’.
4. Click the Apply button at the bottom of the pane.
5. Click OK to exit the Properties window.

6. Storing Firmware Images in Internal Memory

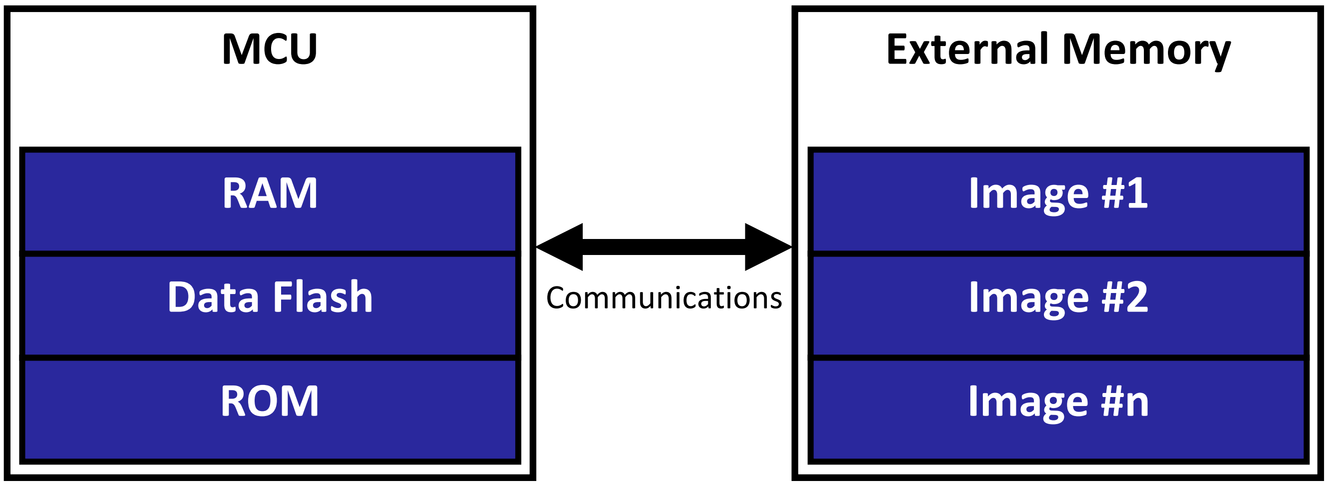
Some users will want to forgo the use of external memory for holding firmware updates and will instead want to store the images internally. While this is possible, there are extra precautions that must be followed that are not present when using external memory. This section will cover the precautions that must be made and why external memory may be the easier choice in your project.

6.1 Example Setups

Throughout this section the following setups will be referenced.

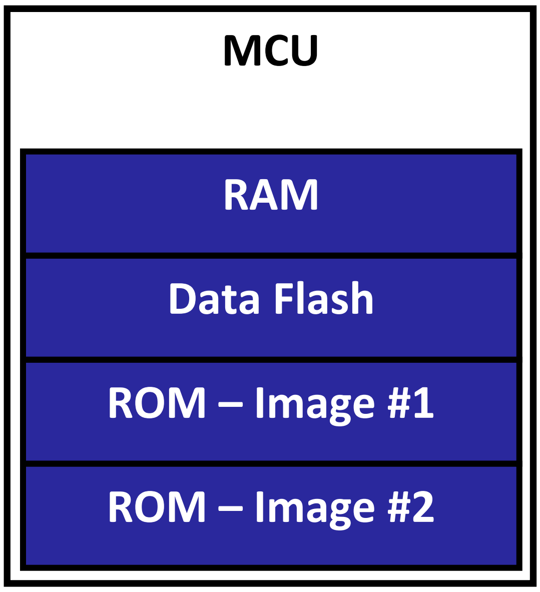
6.1.1 Firmware Images Stored in External Memory

This is how the Flash Loader project is configured 'out of the box'. Firmware images are stored in an external memory area (shown on the right) and programmed into the flash areas on the MCU (shown on left). Only 1 firmware image is stored in the MCU's ROM and Data Flash at one time.



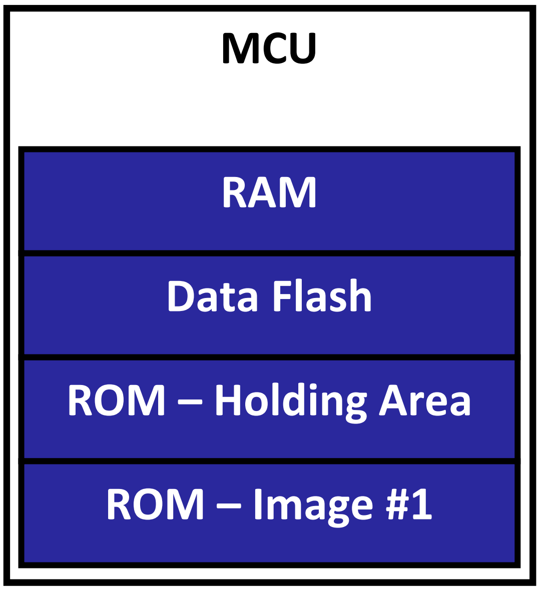
6.1.2 Firmware Images Stored Directly in Internal Memory

Instead of having a separate external memory for firmware images, images are stored in the same ROM area as the user's application. With this setup there is no 'holding area' for the firmware image. A holding area is a memory area where the firmware image is stored, but is not executed from. For example, in Section 6.1.1 the holding area is the external memory. In this example a new firmware image is downloaded in-place. This means that once the image is downloaded, it is ready to be executed. There is no stage where the Flash Loader Bootloader takes the image from a holding area and programs it into the ROM.



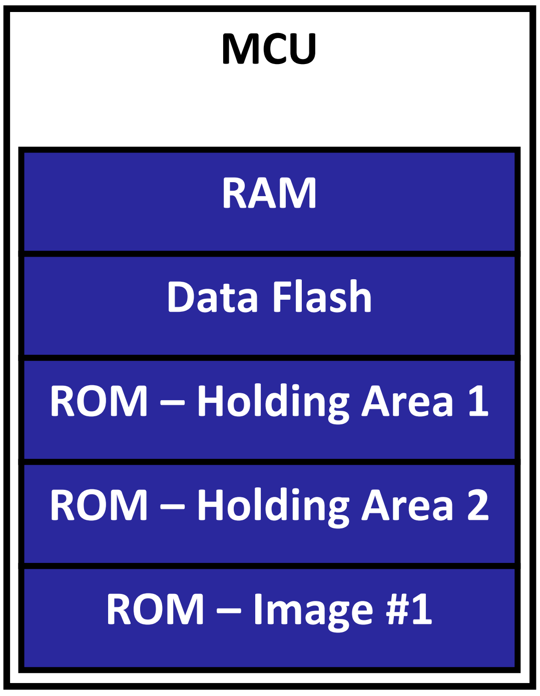
6.1.3 Firmware Images Stored Directly in Internal Memory with Holding Area

This setup is similar to the one described in Section 6.1.2 in that images are stored in MCU ROM except that there is a holding area where images are stored, but not executed from. When a new firmware image is downloaded it will always be stored in the ROM holding area before it is programmed in to the Image #1 space. The differences this leads to in your project are described later in the section.



6.1.4 Firmware Images Stored Directly in Internal Memory with Holding Area

This setup is the same as Section 6.1.3 except that there are multiple holding areas. The differences this leads to in your project are discussed later in this section.



6.2 Cost of Doubling Internal ROM Size

This will have to be investigated on a case-by-case basis but sometimes doubling the size of the internal ROM on a MCU will cost more than buying an external memory like a SPI flash. If board space is an issue, then obviously adding an external memory unit may be impossible.

6.3 Precautions when Storing Images in Internal ROM

The information below is applicable to the setups described in Sections 6.1.2, 6.1.3, and 6.1.4.

One of the features of the Flash Loader project is that it is supposed run alongside the user application while not interfering. This is easy when an external memory is used because the data is sent to the memory and stored. There are no precautions that need to be made as far as accessing the data that has been buffered to be stored. When a receive buffer is full and ready to be stored functions are run from ROM, they access data in the RAM buffer, and write the data to the external memory. This is not as easily done when storing data in the same ROM area as where the program is executing. The reason for this is that when programming the internal ROM on a RX device, any ROM access during the operation will cause a ROM Access Violation error.

To prevent this error from occurring, the user must make sure that their program does not access ROM in any way while the ROM is being programmed. Let's first look at the scenario where the ROM background operation (BGO) feature is not used. This means that when the ROM write routine is started, it does not return until the operation has finished. Some of the ways ROM could be accessed to generate this error are listed below.

- An interrupt occurs and the vector table, which is by default stored in ROM, is accessed.
- An interrupt occurs (vector table is in RAM) and the interrupt is located in ROM.
- An interrupt occurs (vector table is in RAM) and the interrupt service routine accesses constant data in ROM.
- A DMAC/DTC is activated which tries to read or write to ROM.

Let's now look at some extra situations that could occur if the ROM BGO functionality is enabled. Note that these are extra situations, the bullets above also apply.

- After the ROM write has been started the function returns to the calling function which is located in ROM.
- A user function that is located in ROM is accessed while the ROM write operation is still on-going in the background.
- While the ROM write operation is still on-going in the background the user code attempts to read constant data from ROM.

To prevent these errors from occurring the user should take the following precautions:

If ROM BGO is enabled or disabled:

- Copy or move the relocatable vector table to RAM.
- Make sure that any interrupt service routines (ISRs) that need to be run during the flash operation are run from RAM.
- Make sure the addresses for the ISRs in the vector table that might be run during the flash operation point to the location of the ISR in RAM.
- Make sure that any ISRs that run during flash operations do not try to access ROM in any way.
- If a DMAC/DTC is set to access ROM then take the necessary steps to prevent the unit from activating.

If ROM BGO is enabled (the ROM write function will return while operation is still on-going):

- The function that called the ROM write function cannot be located in ROM.
- Any code run before the ROM write operation finishes cannot be located in ROM.
- Any memory accesses made before the ROM write operation finishes cannot be made to ROM.

As stated earlier, one of the features of the Flash Loader project is that firmware image downloads occur in the background. It should be clear that when storing images in internal ROM, this feature begins to fade. While it is easier to disable ROM BGO operations and wait for the operation to finish, this can take a significant amount of time away from the user application. If ROM BGO is used then the user application will not be stalled as long, but extra precautions must be made to make sure ROM is not accessed before the ROM write operation finishes.

6.4 Precautions When Not Using a Holding Area

The information below is applicable to the setup described in Section 6.1.2.

When not using a holding area for a firmware image the user is able to execute the program immediately after it is downloaded. There is no need to move the program to internal ROM because it is already located there. There are issues though that must be addressed with this design.

The main issue that must be addressed is that by default the Renesas RX toolchain outputs position **dependent** code and position **dependent** data. Position dependent code means that branches and jumps can use real addresses inside the MCU. Position dependent data means that data can be referenced from real addresses. This works fine when the program is always programmed into the same location, but this not the case when there is no holding area. If the user has allocated two image sections in the MCU's internal ROM then the image could end up in either section. If the user writes the application for one image location, then it will not work in the other, and vice versa.

There are multiple ways to get around this problem. Two will be discussed in this section.

The first, and most thorough, way to address this problem is to change the RX toolchain to output position **independent** code (PIC) and position **independent** data (PID). As the name suggests this removes the use of real addresses as was done previously. Instead, all function calls and data accesses are done relative to a base register. The downside to this approach is that programs may be less efficient and a 'master' application is required to perform some initialization before the application is run. The master application could reside in the User Boot Area and be part of the bootloader. The generation of PIC and PID was introduced in v1.01 of the Renesas RX Toolchain. Please refer to the RX Family C/C++ Compiler Package V.1.01 User's Manual for more information. The section that should be referenced is Section 8.4 "Usage of PIC/PID Function". Once the user application has been compiled to use PIC and PID and the master application has been written to call the application then the image can be placed anywhere in the internal ROM.

Another approach would be to have rules that are followed on where applications are placed. This is not a thorough fix as the previous example was. If the user forgets one of these rules then the application does have the chance of failing. An example list of rules that could be followed is below.

- There are 2 image 'slots'
- Images that have an odd version number will go in lower addressed slot.
- Images that have an even version number will go in the higher addressed slot.

The downside to this approach it removes the guarantee that users will always have the option of going back to an older firmware image in the event they find something wrong in the current image. As an example, assume the MCU has two images programmed in. The images are version 1 and version 2. Version 2 of the application has a bug that makes the application reset the MCU. The bootloader recognizes that the image has a bug and therefore decides to revert back to version 1 of the application. This can be done according to the rules above as long as the images are odd and even. Now let's assume the current image is version 1, version 2 was never downloaded, and version 3 is requested for download. According to the rules above the version 1 code would have to first be deleted. This means that the bootloader would have to run, delete the current running image, and then download a new image that could have a possible bug. Also, if there was a power down while version 3 was being downloaded (version 1 already deleted) then the MCU would be stuck with only the bootloader.

6.5 Using a Holding Area with Internal ROM

The information below is applicable to the setups described in Sections 6.1.3 and 6.1.4.

When using a holding area with firmware images that are stored internally the benefit is that the user does not have to use PIC or PID as described in Section 6.4. Since the location of the program is fixed the user can write their application to always reside in that area. The downside to this approach compared to the setup described in Section 6.1.2 is that the user loses the ability to revert back to an earlier version of the code without downloading the previous image again when there is only 1 holding area. The reason for this is that the area that is reserved for a running image in Section 6.1.2 is now reserved for just holding an image.

If the user has a setup like the one shown in Section 6.1.4 that has multiple holding areas then the user would still have the option of reverting to a previous firmware revision. The downside to the approach of having multiple holding areas in internal ROM is that the internal ROM is now divided into 3 or more regions instead of 2. To account for this the user might be required to use a MCU with larger internal ROM or decrease the maximum size allowed for their application.

6.6 Table Comparing Setups

The table below shows the requirements of each setup for comparison purposes. Please refer to the sub-sections of Section 6 for more information on each requirement.

Requirement	**External Memory – Multiple Holding Areas**	**Internal Flash – No Holding Area 2 Images**	**Internal Flash – 1 Holding Area**	**Internal Flash – Multiple Holding Areas**
Able to access ROM while storing firmware image	**True**	**False**	**False**	**False**
No requirement to move data, code, vector table to RAM while storing firmware image	**True**	**False**	**False**	**False**
Must either limit application size or buy MCU with larger ROM area	**True**	**False**	**False**	**False**
No chance of erasing currently running image when erasing firmware download	**True**	**False**	**False**	**False**
Can revert to previous revision without Host intervention	**True**	**True**	**False**	**True**
No requirement to use PIC and PID	**True**	**False**	**True**	**True**
Application is able to use all of MCU's internal ROM	**True**	**False**	**False**	**False**
Requires minimal amount of modification to the 'out-of-box' Flash Loader code	**True**	**False**	**False**	**False**
Requires no extra board space	**False**	**True**	**True**	**True**

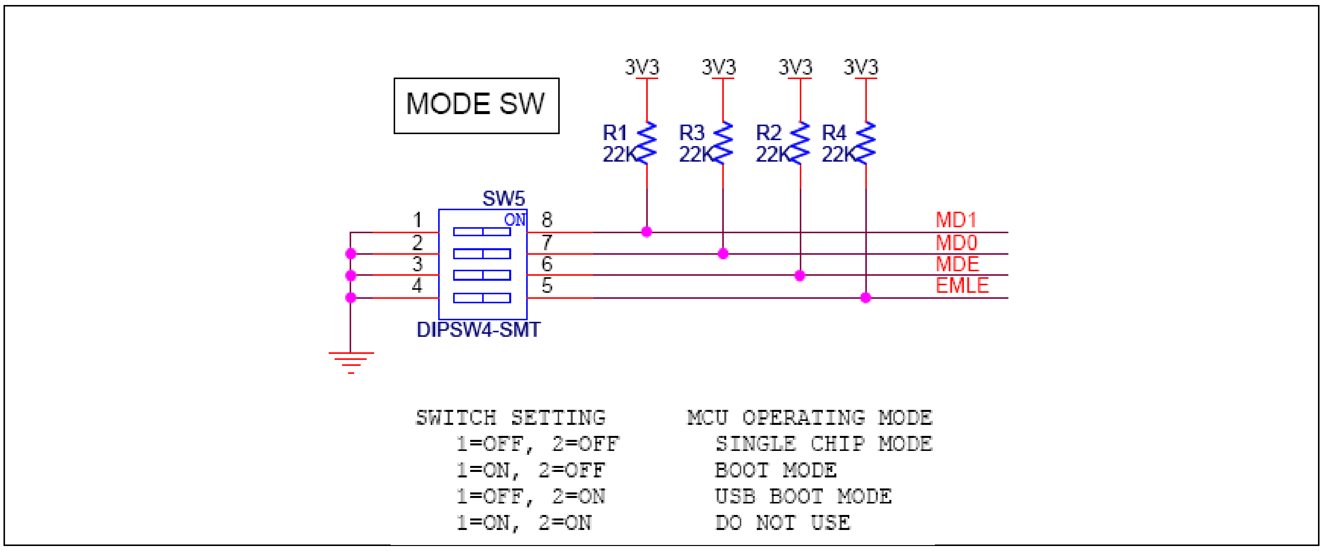
7. Board Notes

This section gives specific information to help with using the Flash Loader project on different boards.

7.1 RDKRX62N Notes – Green Boards (Before Rev 5.0)

7.1.1 Choosing Operating Mode on RX62N RDK

The RDKRX62N allows the user to specify which operating mode the MCU should be in after reset using SW5. Below is a screenshot from the RDK board's schematic that shows how to set the MCU in different operation modes. Note that User Boot Mode is the same as USB Boot Mode. Example usage with the Flash Loader project would be 1=ON, 2=OFF in order to put the MCU in Boot Mode and program the User Boot area and 1=OFF, 2=ON in order to boot from the User Boot area and run the Flash Loader Bootloader.



SWITCH SETTING	MCU OPERATING MODE
1=OFF, 2=OFF	SINGLE CHIP MODE
1=ON, 2=OFF	BOOT MODE
1=OFF, 2=ON	USB BOOT MODE
1=ON, 2=ON	DO NOT USE

Figure 7-1 : Choosing Operating Mode on RDK

7.1.2 Using Serial Boot Mode with the RDKRX62N

In order to program the RX62N using serial boot mode on the RX62N RDK jumpers need to be put in place to connect the RS232 header on the RDK board to SCI channel 1. By default the RDK comes configured with SCI channel 2 connected to the RS232 header. Starting with Revision 3 of the RDK board the user has the option of connecting SCI channel 1 to the RS232 header by shorting JP14 and JP15 on the back side of the RDK board. By default JP14 and JP15 are open. It should be noted that the example Flash Loader project uses SCI channel 2 to communicate with the PC Host and when debugging the E1/E20/JLink use SCI channel 1. This means that you will encounter problems if you leave JP14 and JP15 shorted and try to debug the example Flash Loader project. This is because Flash Loader communications and debugger communications will interfere with each other. To avoid this it is recommended that the user have a way of shorting or opening JP14 and JP15 easily. An example would be to have headers on JP14 and JP15 and use shunts to short or open the connection. When the user wants to program the User Boot area with their bootloader application they would short JP14 and JP15. When the user wants to debug their User Application they would open JP14 and JP15.

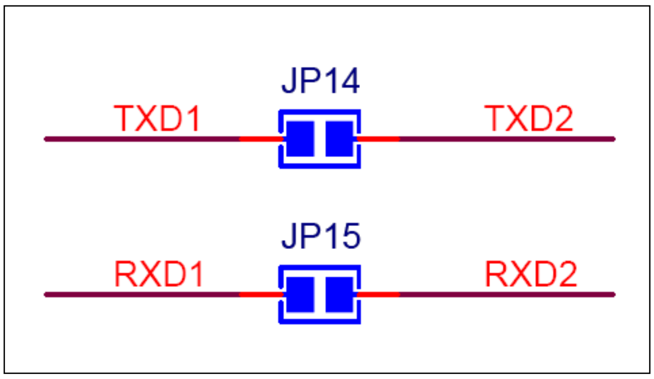
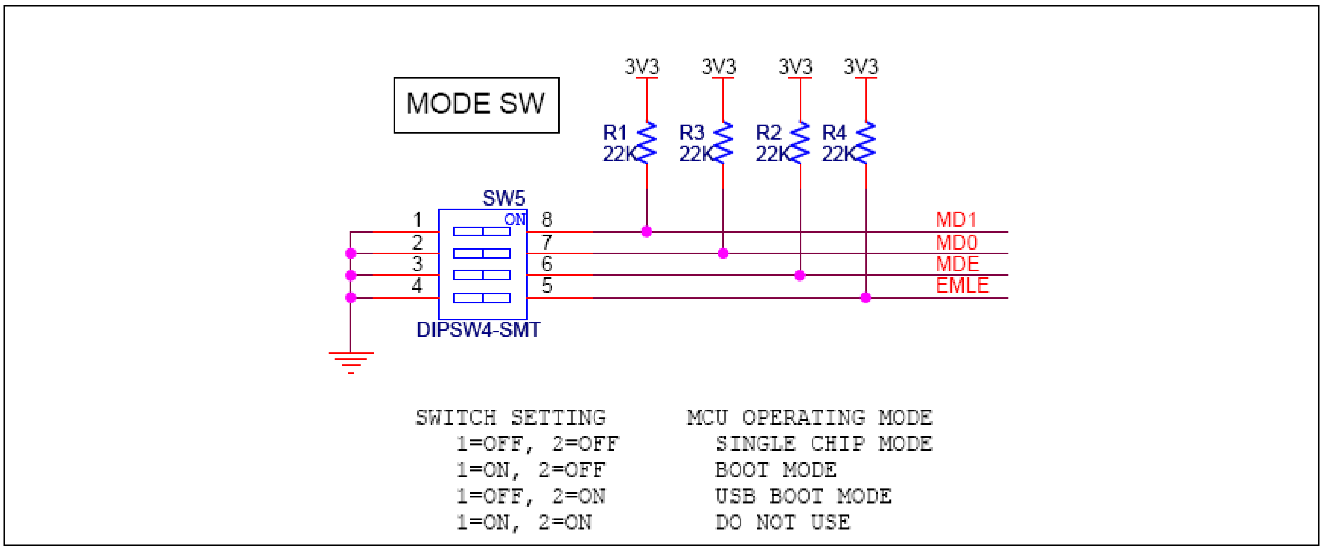


Figure 7-2 : Connecting SCI Channel 1 to RS232 Header on RDK

7.2 RDKRX62N Notes – Blue Boards (Rev 5.0 & Higher)

7.2.1 Choosing Operating Mode on RX62N RDK

The RDKRX62N allows the user to specify which operating mode the MCU should be in after reset using SW5. Below is a screenshot from the RDK board's schematic that shows how to set the MCU in different operation modes. Note that User Boot Mode is the same as USB Boot Mode. Example usage with the Flash Loader project would be 1=ON, 2=OFF in order to put the MCU in Boot Mode and program the User Boot area and 1=OFF, 2=ON in order to boot from the User Boot area and run the Flash Loader Bootloader.

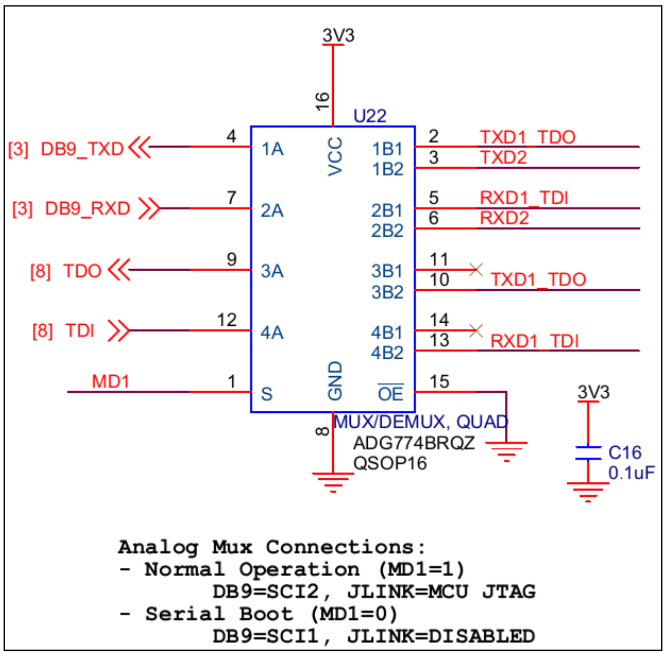


SWITCH SETTING	MCU OPERATING MODE
1=OFF, 2=OFF	SINGLE CHIP MODE
1=ON, 2=OFF	BOOT MODE
1=OFF, 2=ON	USB BOOT MODE
1=ON, 2=ON	DO NOT USE

Figure 7-3 : Choosing Operating Mode on RDK

7.2.2 Using Serial Boot Mode with the RDKRX62N

Starting with Rev 5.0 of the RDKRX62N an analog mux was added to automatically choose the correct SCI channel for the on-board DB9 connector based on the operating mode selection from SW5. When the user chooses Single-Chip Mode, SCI2 will be chosen for the DB9 connector. When the user chooses Boot Mode, SCI1 will be chosen for the DB9 connector enabling serial programming with FDT.



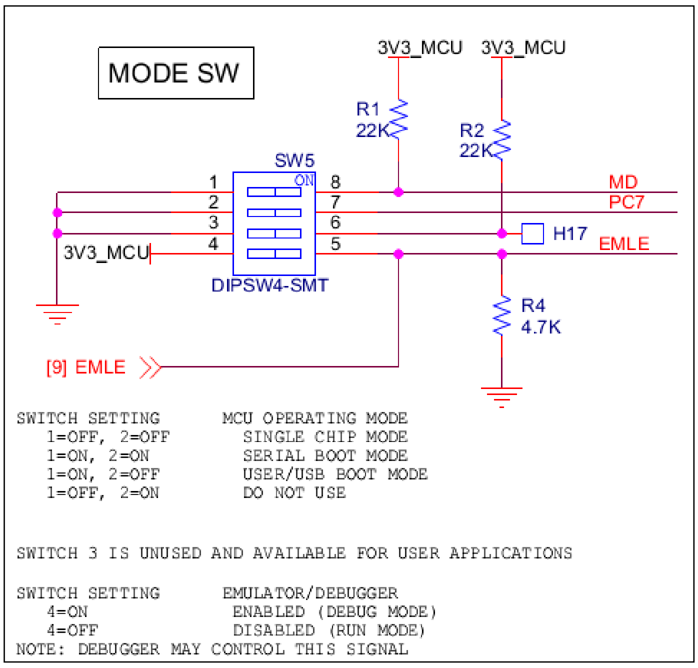
```
Analog Mux Connections:
- Normal Operation (MD1=1)
        DB9=SCI2, JLINK=MCU JTAG
- Serial Boot (MD1=0)
        DB9=SCI1, JLINK=DISABLED
```

Figure 7-4 : Analog mux now chooses correct SCI channel

7.3 RDKRX63N Notes

7.3.1 Choosing Operating Mode on RX63N RDK

The RDKRX63N allows the user to specify which operating mode the MCU should be in after reset using SW5. Below is a screenshot from the RDK board's schematic that shows how to set the MCU in different operation modes. Note that User Boot Mode is the same as USB Boot Mode. Example usage with the Flash Loader project would be 1=ON, 2=ON in order to put the MCU in Boot Mode and program the User Boot area and 1=ON, 2=OFF in order to boot from the User Boot area and run the Flash Loader Bootloader.

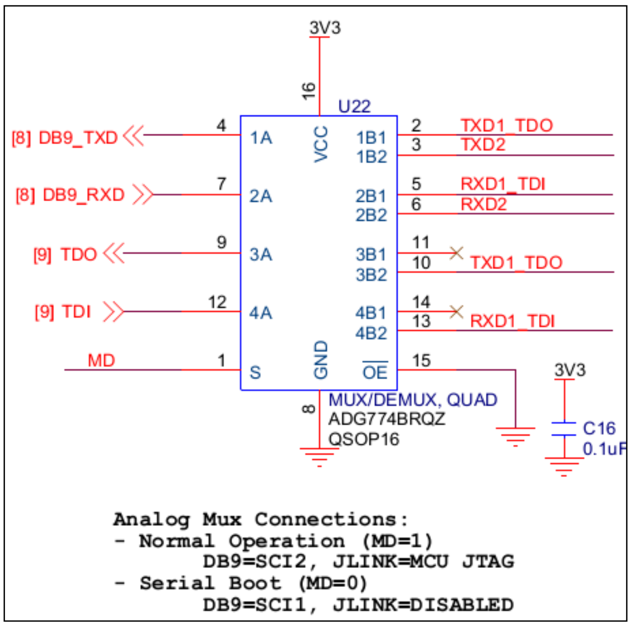


```
SWITCH SETTING      MCU OPERATING MODE
   1=OFF, 2=OFF       SINGLE CHIP MODE
   1=ON,  2=ON        SERIAL BOOT MODE
   1=ON,  2=OFF       USER/USB BOOT MODE
   1=OFF, 2=ON        DO NOT USE

SWITCH 3 IS UNUSED AND AVAILABLE FOR USER APPLICATIONS

SWITCH SETTING      EMULATOR/DEBUGGER
   4=ON              ENABLED (DEBUG MODE)
   4=OFF             DISABLED (RUN MODE)
NOTE: DEBUGGER MAY CONTROL THIS SIGNAL
```

Figure 7-5 : Choosing Operating Mode on RDK

7.3.2 Using Serial Boot Mode with the RDKRX63N

The RDKRX63N has an analog mux that automatically chooses the correct SCI channel for the on-board DB9 connector based on the operating mode selection from SW5. When the user chooses Single-Chip Mode, SCI2 will be chosen for the DB9 connector. When the user chooses Boot Mode, SCI1 will be chosen for the DB9 connector enabling serial programming with FDT.



```
Analog Mux Connections:
- Normal Operation (MD=1)
        DB9=SCI2, JLINK=MCU JTAG
- Serial Boot (MD=0)
        DB9=SCI1, JLINK=DISABLED
```

Figure 7-6 : Analog mux now chooses correct SCI channel

7.3.3 User Boot DB9 Conflict

When putting the RDKRX63N in User Boot Mode using SW5, the analog mux will automatically choose SCI1 for the on-board DB9 connector. This means that User Boot Mode cannot be debugged because SCI1 will not be connected to the JLink debug pins. Since SCI2 is used for Host communications, this also means that the Device cannot communicate with the Host unless the code is changed. To get around this perform the following actions:

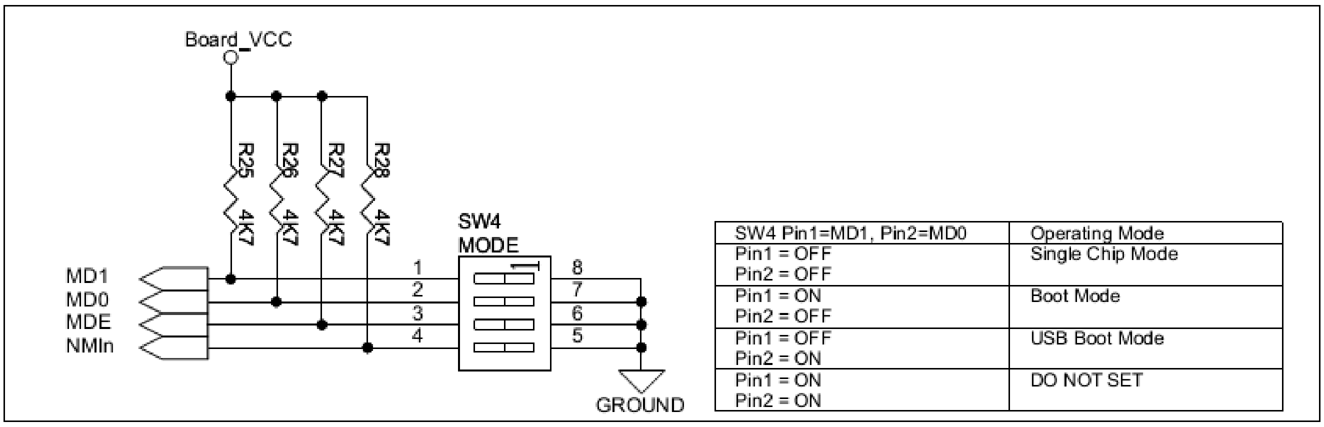
1. Boot the MCU into User Boot Mode following the instructions in Section 7.3.1
2. After the device has reset, change the first switch on SW5 to OFF.

By changing SW5 after the MCU has reset, the MCU will stay in User Boot Mode, but the SCI channels will be connected as they would be in Single-Chip Mode. When doing this the user must remember to change the first switch of SW5 every time they perform a reboot. If this switch is not made every time then the MCU will reboot into Single-Chip Mode instead of User Boot Mode.

7.4 RSK+RX62N Notes

7.4.1 Choosing Operating Mode on RSK+RX62N

The RSK+RX62N allows the user to specify which operating mode the MCU should be in after reset using SW4. Below is a screenshot from the RSK board's schematic that shows how to set the MCU in different operation modes. Note that User Boot Mode is the same as USB Boot Mode. Example usage with the Flash Loader project would be 1=ON, 2=OFF in order to put the MCU in Boot Mode and program the User Boot area and 1=OFF, 2=ON in order to boot from the User Boot area and run the Flash Loader Bootloader.



SW4 Pin1=MD1, Pin2=MD0	Operating Mode

Pin1 = OFF
Pin2 = OFF
	Single Chip Mode

Pin1 = ON
Pin2 = OFF
	Boot Mode

Pin1 = OFF
Pin2 = ON
	USB Boot Mode

Pin1 = ON
Pin2 = ON
	DO NOT SET

Figure 7-7 : Choosing Operating Mode on RSK+RX62N

7.4.2 Using Serial Boot Mode with the RSK+RX62N

The RSK+RX62N allows the user to choose which SCI channel is connected to the on-board DB9 connector using the J15 and J16 jumpers. A screenshot from the board's schematic is shown below. By default SCI2-A is chosen (pins 2 and 3 are shorted) which is OK during Single-Chip Mode. For Boot Mode SCI1-B will need to be selected so the user must change the jumpers on J15 and J16 to short pins 1 and 2.

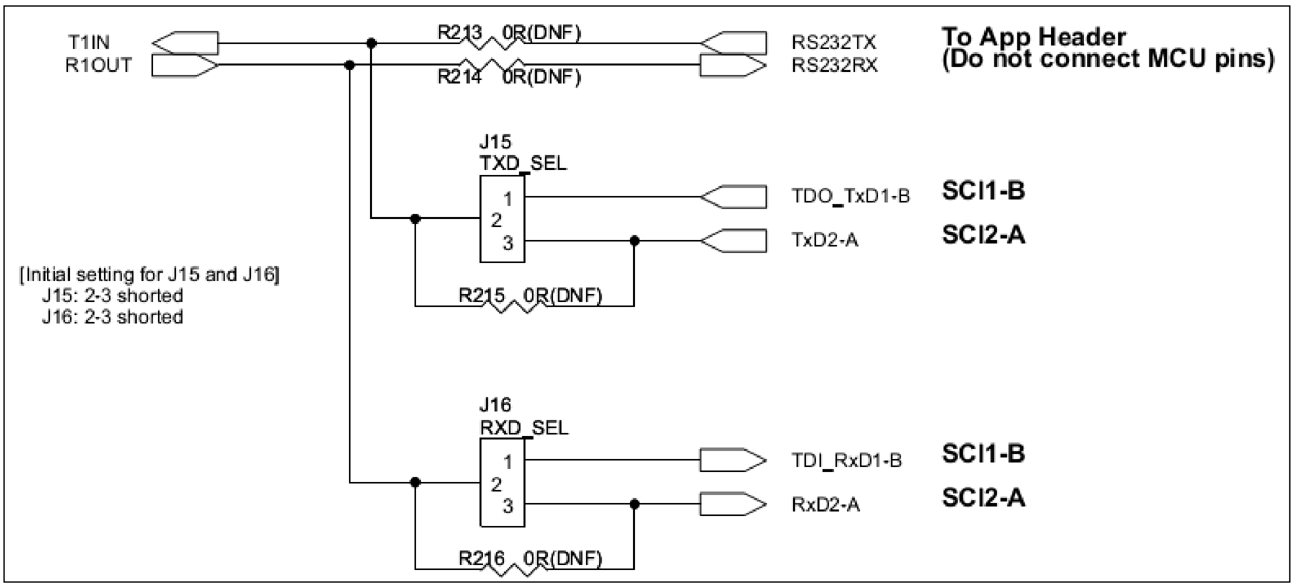
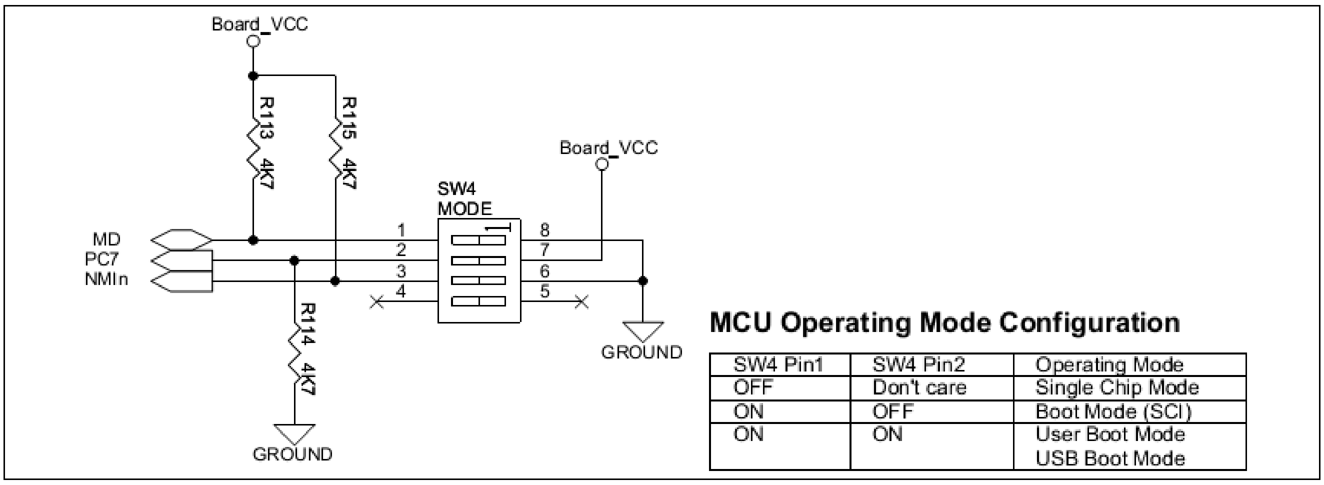


Figure 7-8 : Connecting SCI Channel 1 to RS232 Header on RSK+RX62N

7.5 RSK+RX63N Notes

7.5.1 Choosing Operating Mode on RSK+RX63N

The RSK+RX63N allows the user to specify which operating mode the MCU should be in after reset using SW4. Below is a screenshot from the RSK board's schematic that shows how to set the MCU in different operation modes. Note that User Boot Mode is the same as USB Boot Mode. Example usage with the Flash Loader project would be 1=ON, 2=OFF in order to put the MCU in Boot Mode and program the User Boot area and 1=ON, 2=ON in order to boot from the User Boot area and run the Flash Loader Bootloader. When the user wants to use User Boot mode they should also verify that the J5 header has PC7 chosen (short pins 2-3).



SW4 Pin1	SW4 Pin2	Operating Mode
OFF	Don't care	Single Chip Mode
ON	OFF	Boot Mode (SCI)
ON	ON	User Boot Mode USB Boot Mode

Figure 7-9 : Choosing Operating Mode on RSK+RX63N

7.5.2 Using Serial Boot Mode with the RSK+RX63N

The RSK+RX63N allows the user to choose which SCI channel is connected to the on-board DB9 connector using the J12 and J13 jumpers. A screenshot from the board's schematic is shown below. By default SCI0 is chosen (pins 1 and 2 are shorted) which is OK during Single-Chip Mode. For Boot Mode, SCI1 will need to be selected so the user must change the jumpers on J12 and J13 to short pins 2 and 3.

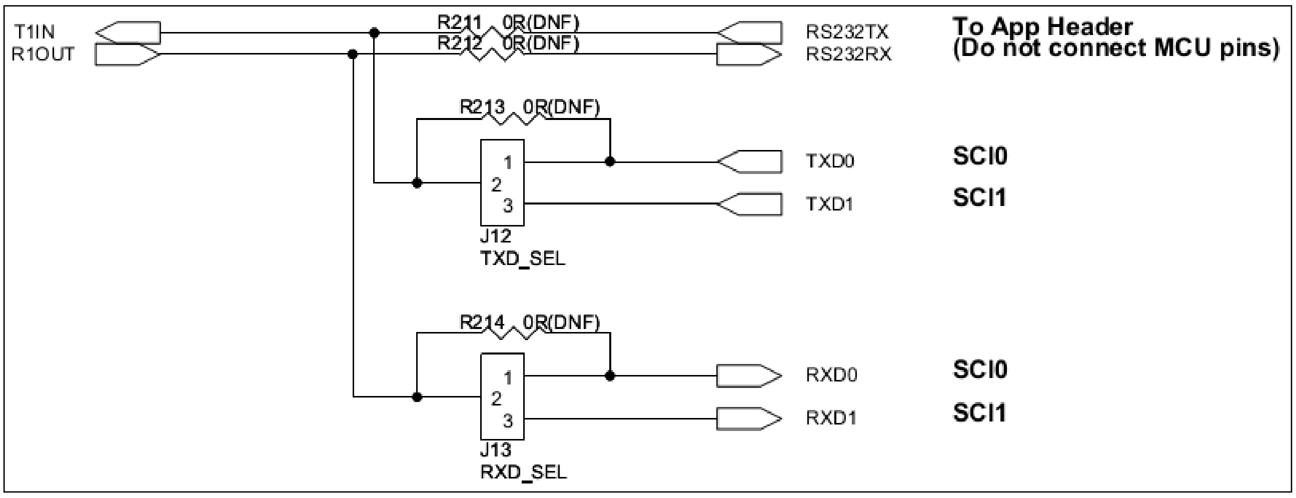


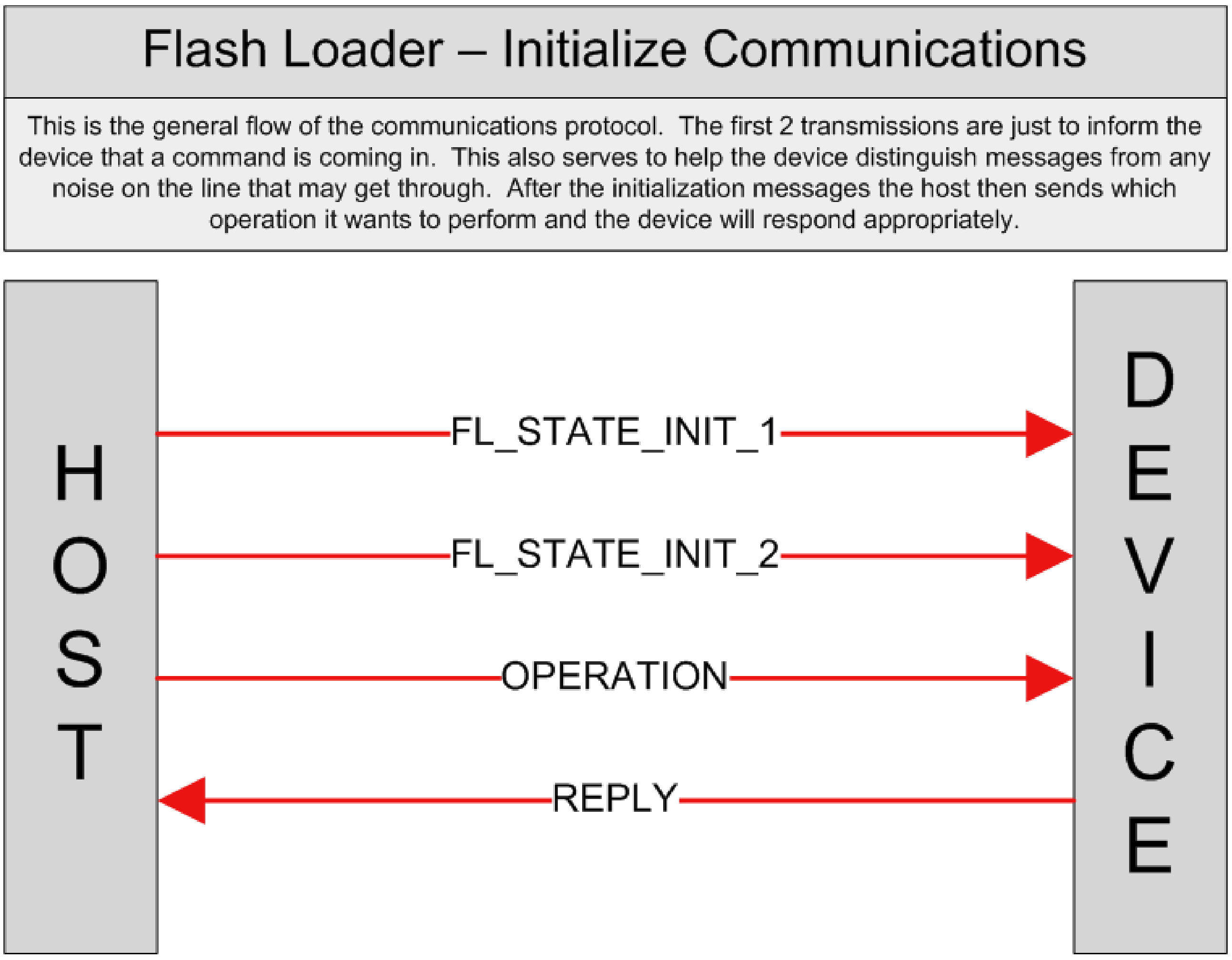
Figure 7-10 : Connecting SCI Channel 1 to RS232 Header on RSK+RX63N

8. Flash Loader Communications Protocol

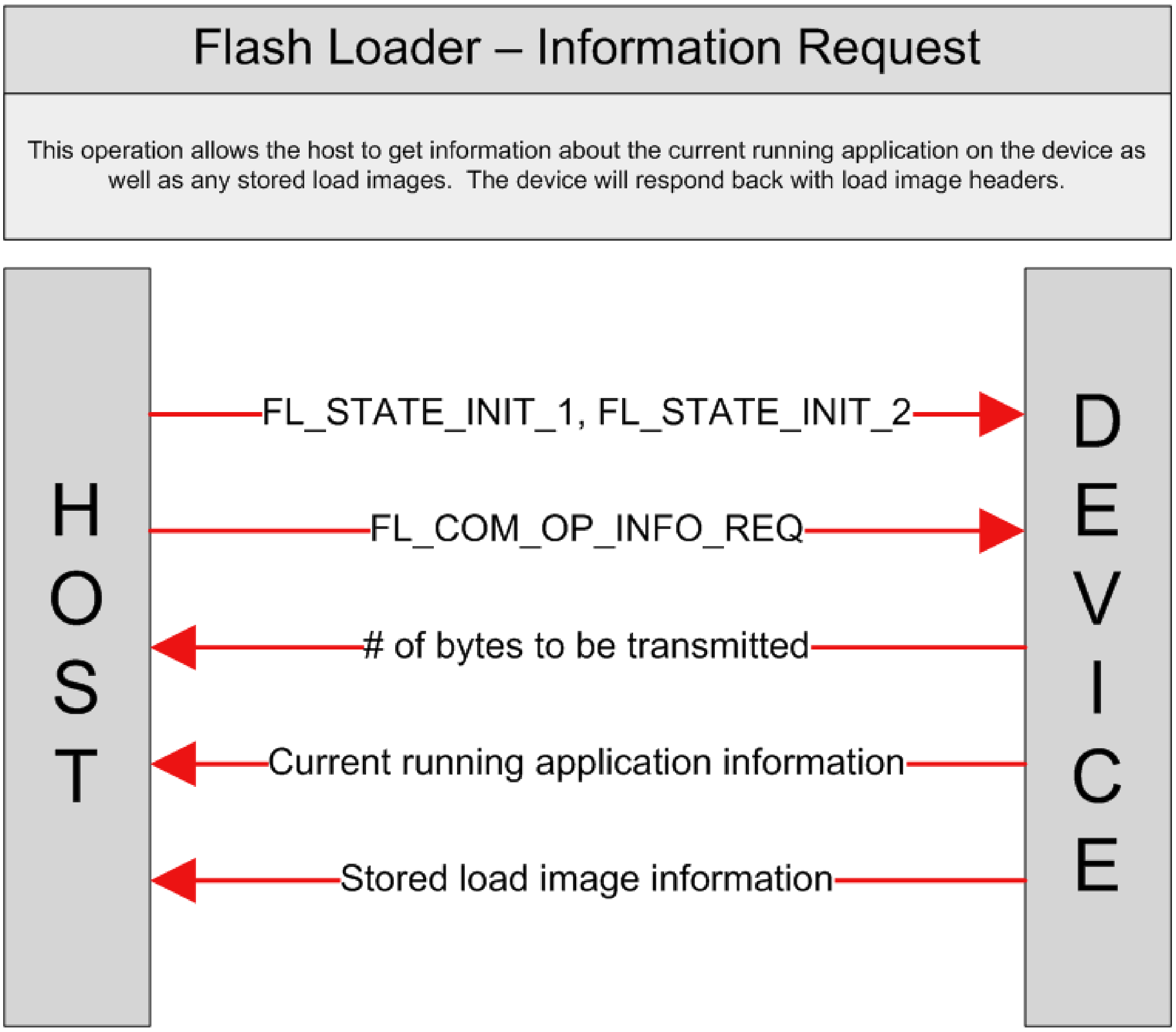
Diagrams showing the communications protocol used with the Flash Loader project are shown in this section. All of the communications protocols are handled by the Host in *r_fl_serial_flash_loader.py* and by the Device in *r_fl_downloader.c*.

8.1 Initializing Communications

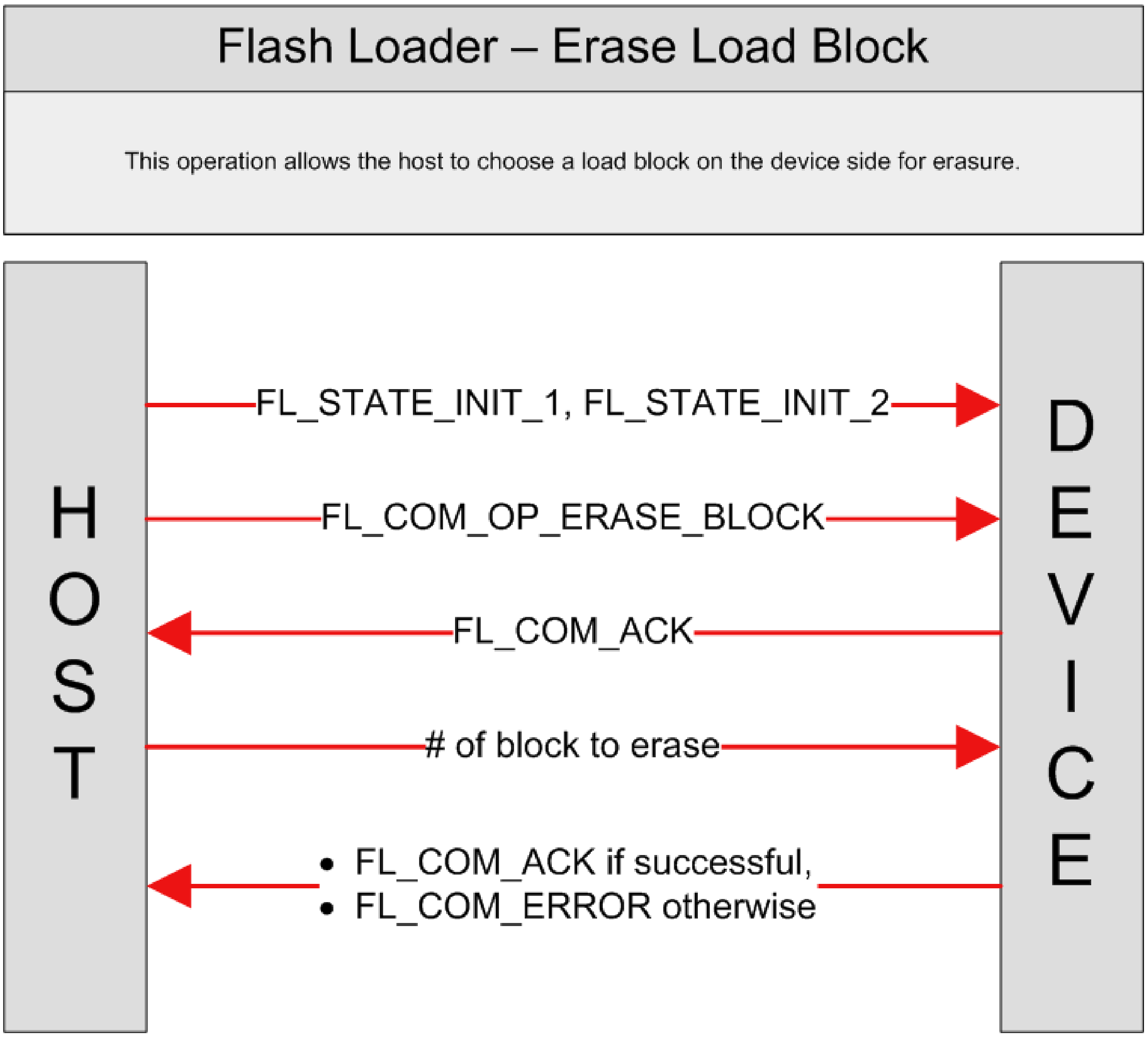
Shown below is not an actual command but how communications start between the Host and the Device.


Flash Loader – Initialize Communications
This is the general flow of the communications protocol. The first 2 transmissions are just to inform the device that a command is coming in. This also serves to help the device distinguish messages from any noise on the line that may get through. After the initialization messages the host then sends which operation it wants to perform and the device will respond appropriately.
HOST
DEVICE
FL_STATE_INIT_1
FL_STATE_INIT_2
OPERATION
REPLY

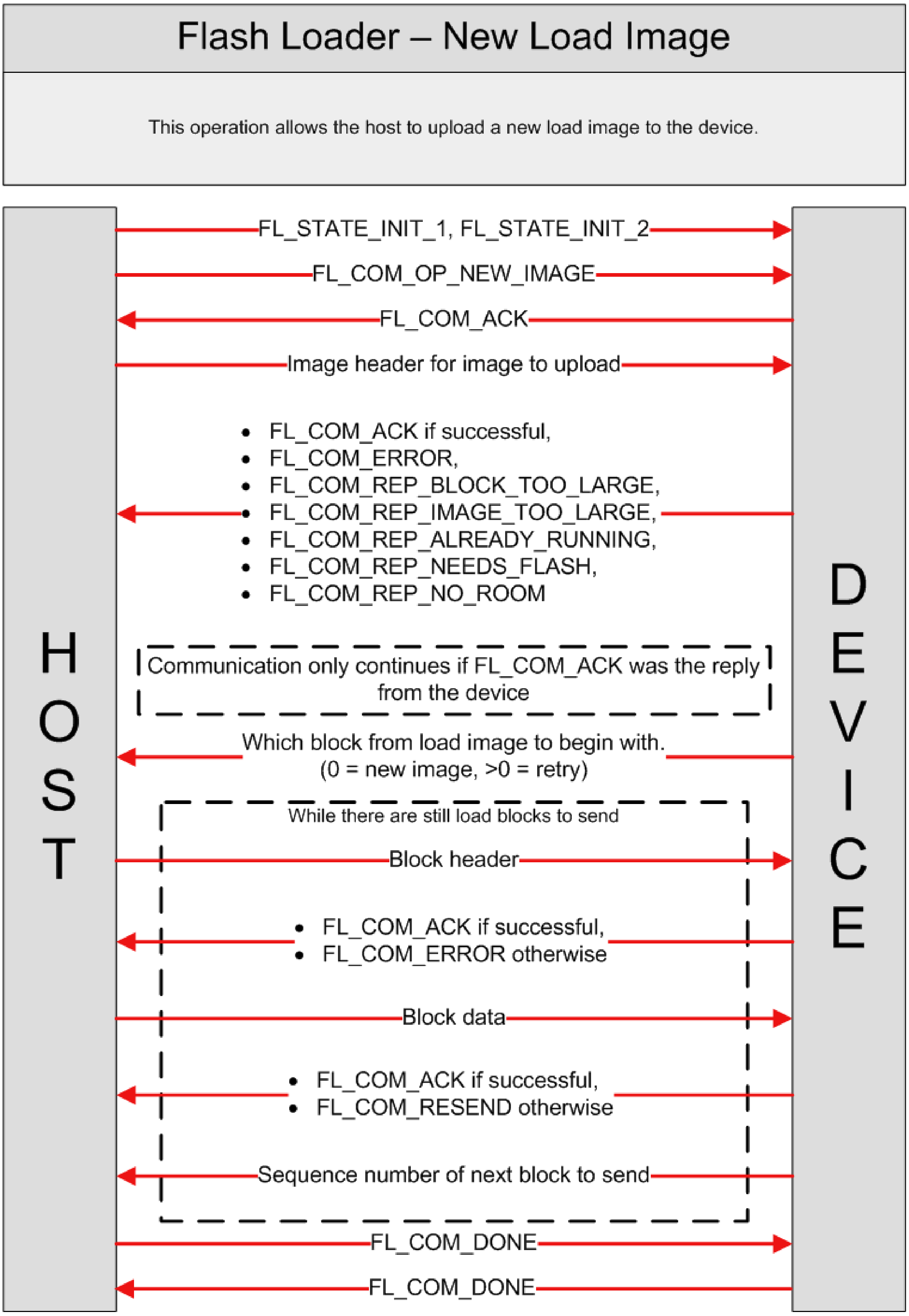
8.2 Information Request Command


Flash Loader – Information Request
This operation allows the host to get information about the current running application on the device as well as any stored load images. The device will respond back with load image headers.
HOST
DEVICE
FL_STATE_INIT_1, FL_STATE_INIT_2
FL_COM_OP_INFO_REQ
# of bytes to be transmitted
Current running application information
Stored load image information

8.3 Erase Command


Flash Loader – Erase Load Block
This operation allows the host to choose a load block on the device side for erasure.
HOST
DEVICE
FL_STATE_INIT_1, FL_STATE_INIT_2
FL_COM_OP_ERASE_BLOCK
FL_COM_ACK
# of block to erase
• FL_COM_ACK if successful,
• FL_COM_ERROR otherwise

8.4 Load Command


Flash Loader – New Load Image
This operation allows the host to upload a new load image to the device.
HOST
DEVICE
FL_STATE_INIT_1, FL_STATE_INIT_2
FL_COM_OP_NEW_IMAGE
FL_COM_ACK
Image header for image to upload
• FL_COM_ACK if successful,
• FL_COM_ERROR,
• FL_COM_REP_BLOCK_TOO_LARGE,
• FL_COM_REP_IMAGE_TOO_LARGE,
• FL_COM_REP_ALREADY_RUNNING,
• FL_COM_REP_NEEDS_FLASH,
• FL_COM_REP_NO_ROOM
Communication only continues if FL_COM_ACK was the reply from the device
Which block from load image to begin with. (0 = new image, >0 = retry)
While there are still load blocks to send
Block header
• FL_COM_ACK if successful,
• FL_COM_ERROR otherwise
Block data
• FL_COM_ACK if successful,
• FL_COM_RESEND otherwise
Sequence number of next block to send
FL_COM_DONE
FL_COM_DONE

9. API Functions

9.1 Summary

The following functions are included in this API:

Function	Description
R_FL_DownloaderInit()	Initializes Flash Loader Downloader
R_FL_StateMachine()	Calls the Flash Loader State Machine
R_FL_GetVersion()	Returns the current version of this API

9.2 R_FL_DownloaderInit

Initializes everything needed to run the Flash Loader Downloader.

Format

```
void R_FL_DownloaderInit(void);
```

Parameters

None

Return Values

None

Properties

Prototyped in file “r_flash_loader_if.h”
Implemented in file “r_fl_downloader.c”

Description

This function initializes everything needed before starting the Flash Loader state machine. Examples include initializing hardware for: Host to Device communications, Device to Storage communications, and CRC calculations.

Reentrant

Yes

Example

```
/* Initialize the Flash Loader code. */
R_FL_DownloaderInit();

/* Now start timer tick that will trigger Flash Loader state machine. */
...
```

9.3 R_FL_StateMachine

Calls the state machine that runs the Flash Loader Downloader.

Format

```
void R_FL_StateMachine(void);
```

Parameters

None

Return Values

None

Properties

Prototyped in file “r_flash_loader_if.h”
Implemented in file “r_fl_downloader.c”

Description

This function implements the Flash Loader state machine. The communications protocol between the Host and Device is implemented inside this function. This function also takes care of storing the downloaded firmware image. This function should be called periodically by the user (i.e. from a timer tick). When called the function will first check the receive buffer to see if any data has been received from the Host. If data has been received then the state machine will process the data accordingly.

Reentrant

No, but the state machine does protect against multiple calls (i.e. only one process is allowed in at any given time)

Example

```
bool g_sm_process;

void main(void)
{
    uint32_t cmt_channel;

    /* Initialize state machine process flag. */
    g_sm_process = false;

    /* Initialize the Flash Loader code. */
    R_FL_DownloaderInit();

    /* Create periodic timer to call Flash Loader state machine. */
    R_CMT_CreatePeriodic(USER_APP_CMT_FREQUENCY,
                         fl_trigger_sm,
                         &cmt_channel);

    while (1)
    {
        /* Call state machine after flag has been set. */
        if (true == g_sm_process)
        {
            /* Trigger state machine. */
            R_FL_StateMachine();

            g_sm_process = false;
        }

        /* Do other work. */
    }
}

/* CONTINUED ON NEXT PAGE */
/* CMT Interrupt Service Routine that will set a flag which will alert the
   main() loop that the Flash Loader state machine should be called. */
static void fl_trigger_sm (void * pdata)
{
    /* Create periodic timer to call Flash Loader state machine. */
    g_sm_process = true;
}
```

Special Notes:
The user gets to control how often the state machine is called. The frequency at which the state machine needs to be called will depend upon the response time requirements of the user. Since the sent data will never overfill the receive buffer, there is no reason to worry about buffer overflow. The faster the state machine is called, the quicker the Device will respond to the host. The downside of this is that as the frequency at which the state machine is called increases, so does the overhead imposed on the entire system by state machine processing. While this overhead may not be large since the state machine will exit if there is nothing new to process, it should not be ignored.

9.4 R_FL_GetVersion

Returns the current version of the module.

Format

```
   uint32_t R_FL_GetVersion(void);
```

Parameters

None.

Return Values

Version of the Flash Loader project.

Properties

Prototyped in file “r_flash_loader_rx_if.h”
Implemented in file “r_fl_utilities.c”

Description

This function will return the version of the currently installed Flash Loader code. The version number is encoded where the top 2 bytes are the major version number and the bottom 2 bytes are the minor version number. For example, Version 4.25 would be returned as 0x00040019.

Reentrant

Yes.

Example

```
uint32_t cur_version;

/* Get version of installed Flash Loader. */
cur_version = R_FL_GetVersion();

/* Check to make sure version is new enough for this application’s use. */
if (MIN_VERSION > cur_version)
{
    /* This Flash Loader version is not new enough and does not have XXX feature
       that is needed by this application. Alert user. */
    ....
}
```

Special Notes:

- This function is specified to be an inline function in *r_fl_utilities.c.*

10. Demo Projects

This application note contains demo projects for both HEW and E2Studio. For HEW the demo is packaged as an entire HEW workspace which contains projects for each Renesas development board. For E2Studio, each Renesas development board has its own zipped project that can be imported into an existing E2Studio workspace. This version of the Flash Loader Project includes projects for the following boards:

- RSKRX62N
- RSKRX63N
- RDKRX62N
- RDKRX63N

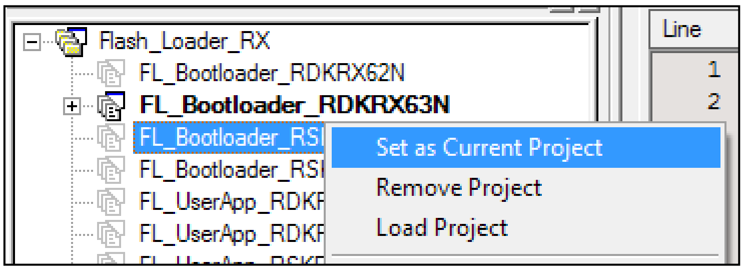
There are two projects per supported board (e.g. 2 for RDKRX62N, 2 for RSKRX63N, etc). These projects are:

- **FL_Bootloader_*board* –** This is the Flash Loader Bootloader project prebuilt for the board referenced by *board*. If the user wants to modify the bootloader then they can edit this project. When the user is ready to program in the bootloader to their MCU then they can get the S-Record file from this project's 'Release' directory.
- **FL_UserApp_*board* –** This is a pre-setup Flash Loader user application for the board referenced by *board*. It does nothing but run the Flash Loader state machine and show a message on the LCD. The purpose of this project is to give the user a shell project that they can use if they wish. Everything is preconfigured for Flash Loader use so the user can also use it as a reference for their own Flash Loader project.

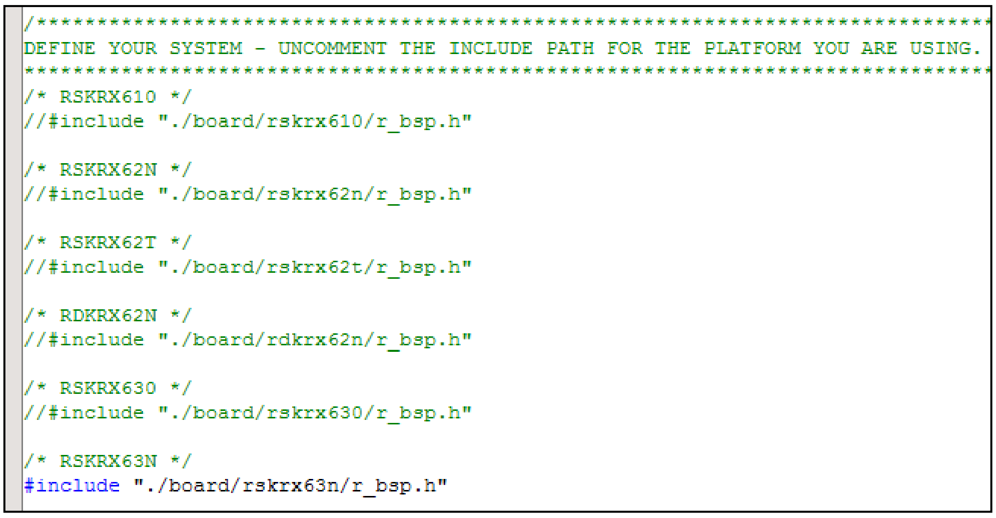
10.1 HEW Workspace

The HEW workspace that comes packaged with this application note has a project for each supported Renesas development board. The only code that changes between these projects is the board support code that is used along with the demo and Flash Loader code. To choose a project follow these steps:

1. Open the HEW workspace
2. Right-click on the project you wish to load in the navigation pane (by default on left) and click 'Set as Current Project'.



3. Flash Loader code uses the r_bsp package for startup code, board support code, and for getting MCU information. The r_bsp package is easily configured through the *platform.h* header file which is located in the r_bsp folder. To configure the r_bsp package, open up *platform.h* and uncomment the #include for the board you are using. For example, to run the demo on a RSK+RX63N board, the user would uncomment the #include for './board/rskrx63n/r_bsp.h' macro and make sure all other board #includes are commented out.



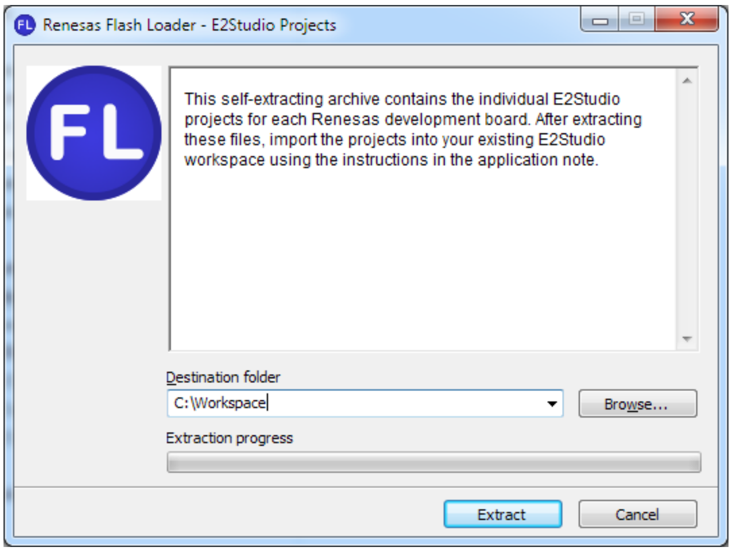
```
/***************************************************************************
DEFINE YOUR SYSTEM - UNCOMMENT THE INCLUDE PATH FOR THE PLATFORM YOU ARE USING.
***************************************************************************
/* RSKRX610 */
//#include "./board/rskrx610/r_bsp.h"

/* RSKRX62N */
//#include "./board/rskrx62n/r_bsp.h"

/* RSKRX62T */
//#include "./board/rskrx62t/r_bsp.h"

/* RDKRX62N */
//#include "./board/rdkrx62n/r_bsp.h"

/* RSKRX630 */
//#include "./board/rskrx630/r_bsp.h"

/* RSKRX63N */
#include "./board/rskrx63n/r_bsp.h"
```

4. You can now build the demos.

10.2 E2Studio Projects

E2Studio handles workspaces differently than HEW and therefore projects must be imported into your existing E2Studio workspace. In order to use the demo for your development board follow these steps:

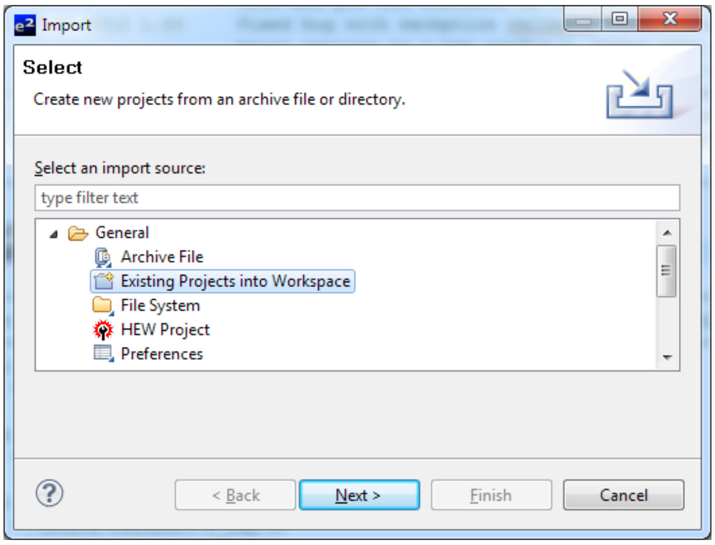
1. The E2Studio projects are distributed as a self-extracting archive with this application note. The first thing that will need to be done is to extract this archive. Double click on the self-extracting archive file (should be *.exe under Workspace\e2studio directory).
2. Choose where to extract the projects and click Extract.



This self-extracting archive contains the individual E2Studio projects for each Renesas development board. After extracting these files, import the projects into your existing E2Studio workspace using the instructions in the application note.

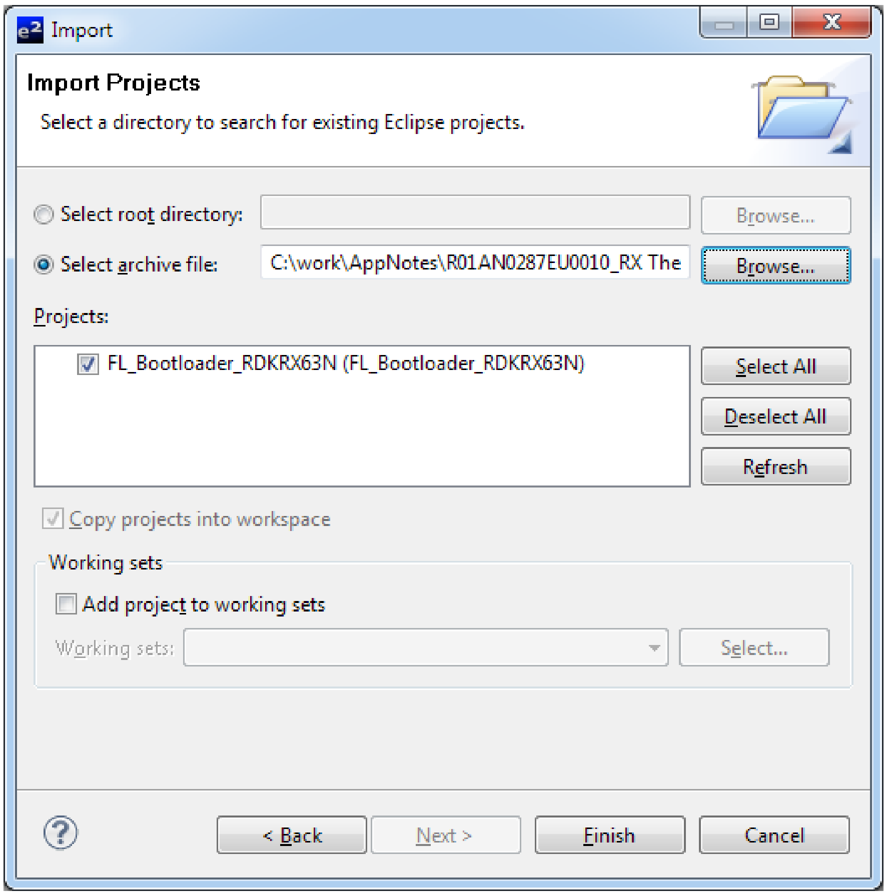
Destination folder: C:\Workspace

3. Open your E2Studio workspace
4. Click File >> Import
5. Choose General >> Existing Projects into Workspace and click Next.



6. Click 'Select archive file' and click browse.
7. Browse to the directory where you extracted the E2Studio projects and choose the zip file for your development board.

8. Check the box next to the project you wish to import and click Finish. In this screenshot the RDKRX63N project is being imported.



Import Projects

Select a directory to search for existing Eclipse projects.

Select root directory:

[✓ Select archive file:] C:\work\AppNotes\R01AN0287EU0010_RX The

Projects:

☑ FL_Bootloader_RDKRX63N (FL_Bootloader_RDKRX63N)

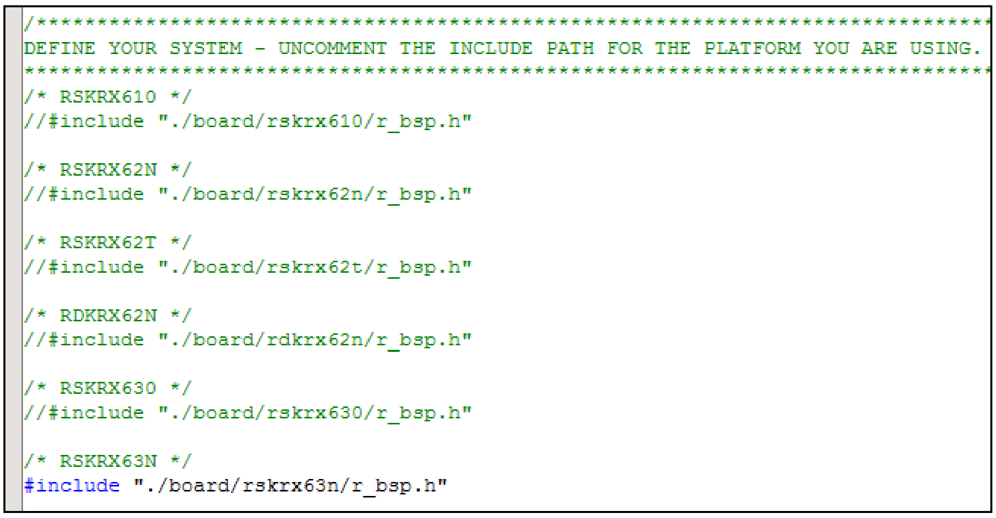
☑ Copy projects into workspace

Working sets

☐ Add project to working sets

Working sets:

5. The VEE API code and demo workspace use the r_bsp package for startup code, board support code, and for getting MCU information. The r_bsp package is easily configured through the *platform.h* header file which is located in the r_bsp folder. To configure the r_bsp package, open up *platform.h* and uncomment the #include for the board you are using. For example, to run the demo on a RSK+RX63N board, the user would uncomment the #include for './board/rskrx63n/r_bsp.h' macro and make sure all other board #includes are commented out.



```
/*****************************************************************************
DEFINE YOUR SYSTEM - UNCOMMENT THE INCLUDE PATH FOR THE PLATFORM YOU ARE USING.
*****************************************************************************
/* RSKRX610 */
//#include "./board/rskrx610/r_bsp.h"

/* RSKRX62N */
//#include "./board/rskrx62n/r_bsp.h"

/* RSKRX62T */
//#include "./board/rskrx62t/r_bsp.h"

/* RDKRX62N */
//#include "./board/rdkrx62n/r_bsp.h"

/* RSKRX630 */
//#include "./board/rskrx630/r_bsp.h"

/* RSKRX63N */
#include "./board/rskrx63n/r_bsp.h"
```

6. You can now build the demo.

10.3 Debugging Projects

10.3.1 E1 or JLink

All UserApp demos can be debugged with no issues. If using an E1/E20 for debugging then the Bootloader demos can be debugged with no issues. If using JLink for debugging then extra steps will need to be followed for debugging. The reason there is a difference is because the E1 controls the mode pins on the RX MCU while the JLink does not. This means that the E1 has the ability to put the MCU into factory boot mode while the JLink does not. The User Boot Area on RX MCUs can only be programmed in factory boot mode. This means that the JLink has no way of programming the Bootloader into the User Boot Area. There are two ways around this:

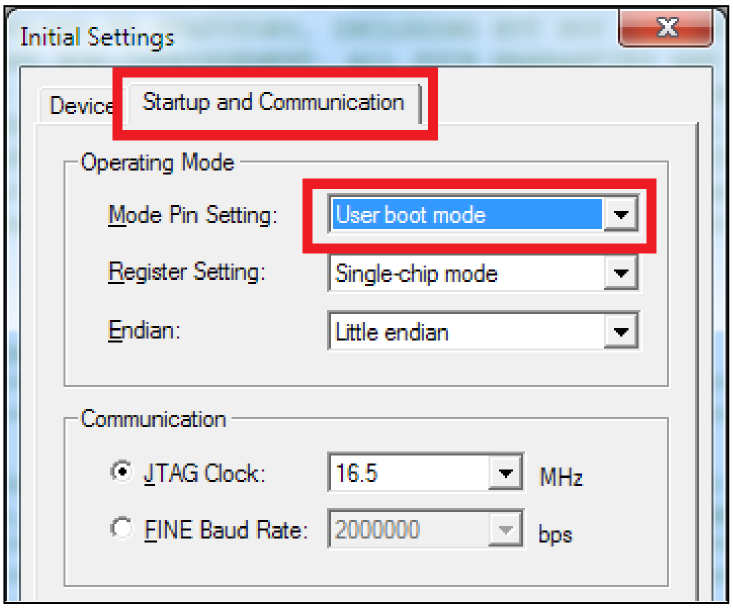
1. Modify the Bootloader project’s linker settings so that the Bootloader is placed in User Application space. By doing this you cannot test the programming of new firmware images (since it would overwrite the currently running program) but you can test out the code and download firmware updates into the Flash Loader Storage area.
2. Program the Bootloader image into the User Boot Area using FDT (Renesas Flash Development Toolkit) and then debug. By doing this the user is ‘fooling’ the JLink into believing that it actually programmed the User Boot Area. The way this works is that the user first builds their project. They then program the image into the User Boot Area using FDT. The user then switches back to their development project and connects as they normally would. When the user chooses to download the code to the MCU the JLink will attempt to program the image but it will not actually do anything. At this point the user can set hardware breakpoints, eventpoints, and debug as usual. Software breakpoints cannot be used because they require that ROM be reprogrammed. Anytime the Bootloader code is modified the user will have to switch over to FDT to program the User Boot Area again before debugging.

10.3.2 Configuring Debug Session for User Boot Mode

No matter what board and debugger are being used the user should always put the MCU into Single-Chip Mode when connecting for debugging. If Single-Chip Mode is not selected then the debugger will not be able to connect to MCU.

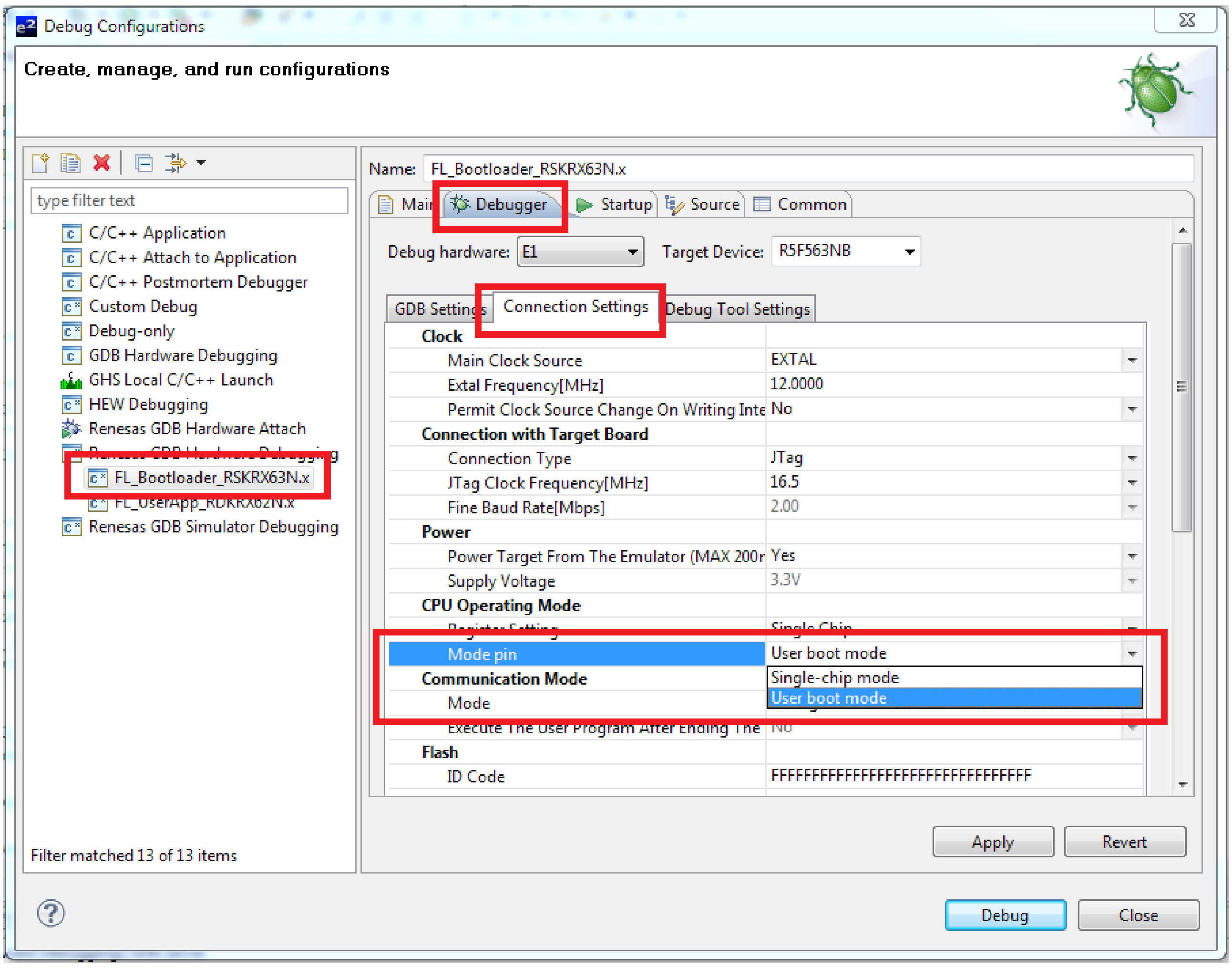
When debugging code in User Boot Mode the user needs to configure the debugger to use User Boot Mode. The main difference with this configuration is that the MCU will fetch the User Boot reset vector instead of the User Application reset vector on reset.

In HEW this is done in the Initial Settings window that pops up when the user wishes to debug. Click the ‘Startup and Communication’ tab and choose ‘User boot mode’ from the ‘Mode Pin Setting’ drop down.



Mode Pin Setting: User boot mode
Register Setting: Single-chip mode
Endian: Little endian
JTAG Clock: 16.5 MHz
FINE Baud Rate: 2000000 bps

In E2Studio this is done through the Debug Configurations window. Select the Debugger >> Connection Settings and then change the ‘Mode pin’ drop down to ‘User boot mode’. Click Apply to ensure the settings are accepted.


Debug Configurations

Create, manage, and run configurations

type filter text

- C/C++ Application
- C/C++ Attach to Application
- C/C++ Postmortem Debugger
- Custom Debug
- Debug-only
- GDB Hardware Debugging
- GHS Local C/C++ Launch
- HEW Debugging
- Renesas GDB Hardware Attach
- Renesas GDB Hardware Debugging
  - FL_Bootloader_RSKRX63N.x
  - FL_UserApp_RDKRX62N.x
- Renesas GDB Simulator Debugging

Filter matched 13 of 13 items

Name: FL_Bootloader_RSKRX63N.x

Main | Debugger | Startup | Source | Common

Debug hardware: E1 Target Device: R5F563NB

GDB Settings | Connection Settings | Debug Tool Settings

Clock	
Main Clock Source	EXTAL
Extal Frequency[MHz]	12.0000
Permit Clock Source Change On Writing Inte	No
Connection with Target Board	
Connection Type	JTag
JTag Clock Frequency[MHz]	16.5
Fine Baud Rate[Mbps]	2.00
Power	
Power Target From The Emulator (MAX 200r	Yes
Supply Voltage	3.3V
CPU Operating Mode	
Register Setting	Single Chip
Mode pin	User boot mode
Communication Mode	
Mode	
Execute The User Program After Ending The	No
Flash	
ID Code	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Appendix A : Load File Header Format

All fields are least-significant-byte (LSB) first.

Load File Header			
Field	**Number of Bytes**	**Name in C Structure**	**Description**
Valid Mask	1	valid_mask	This is a constant value that signifies that this is a Load Image. If this mask is not correct then no other checking will be done on the Load Image.
Image ID	1	image_ID	This is an identifier for the application that is currently running. This helps distinguish between different applications that might run on the Device.
Version Number	1	version_num	This is the version number that applies to this Load Image. This helps distinguish between different versions of the same application.
Size of Load Image	4	load_image_size	This tells how large this Load Image is in bytes.
Max Block Data Size	4	max_block_size	Since packets are used to split up the data, this identifies the largest size a data block can be. If this block size is too large then the Device will not be able to download the Load Image when the Host requests a new transfer.
Load Image CRC	2	image_crc	This is the CRC of the Load Image as it sits in the Flash Loader Storage area. This is used for verifying that a Load Image was successfully downloaded.
Raw CRC	2	raw_crc	This is the CRC of the entire memory area of the Device when this Load Image is programmed into the Device's User Application area.
Address of 1st Data Block	4	start_address	Data blocks are stored in a linked list fashion inside of the Flash Loader Storage area. This address points to the first data block.
Successfully Stored	4	successfully_stored	This entry denotes that this image was successfully downloaded and verified using the Image CRC. It is also a unique identifier that allows for Load Images to be compared against when they were downloaded. An example, the first Load Image downloaded might have a value of 1 for this field while an image downloaded later would have 2.

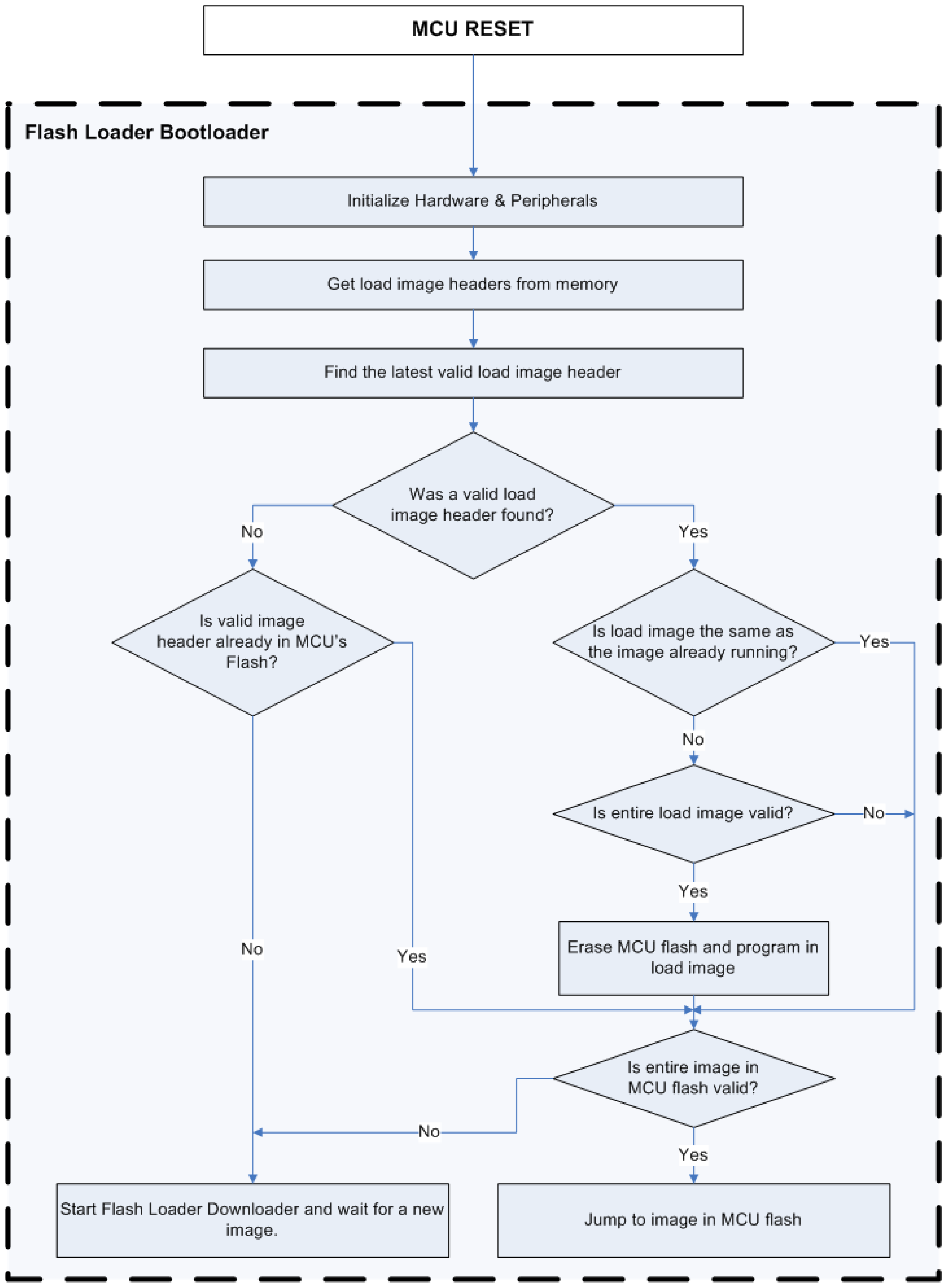
Appendix B : Data Block Header Format

All fields are least-significant-byte (LSB) first.

Data Block Header			
Field	**Number of Bytes**	**Name in C Structure**	**Description**
Valid Mask	1	valid_mask	This is a constant value that signifies that this is a Data Block Header. If this mask is not correct then no other checking will be done on the Data Block.
Sequence ID	2	sequence_ID	A unique identifier for this block. This is what allows retries to be done easily since the Device can send to the Host the last Data Block it received successfully.
Flash Address	4	flash_address	The address in the Device's memory where this data should go.
Size of Data	4	data_size	Number of bytes of data after the block header.
Data CRC	2	data_crc	CRC-16 value of the data
Next Block Address	4	next_block_address	Blocks are connected in a linked list fashion. Each block points to the next block. This is the address of the next block.
Data	data_size	data	This is not part of the Data Block Header but it is part of the Data Block. This is the data that will be programmed into the Device at the address specified by 'flash_address'.

Appendix C : Flash Loader Bootloader Diagram

This flowchart shows the execution flow of the default Flash Loader Bootloader.


MCU RESET
Flash Loader Bootloader
Initialize Hardware & Peripherals
Get load image headers from memory
Find the latest valid load image header
Was a valid load image header found?
No
Yes
Is valid image header already in MCU's Flash?
Is load image the same as the image already running?
Yes
No
Is entire load image valid?
No
Yes
Erase MCU flash and program in load image
No
Yes
Is entire image in MCU flash valid?
No
Yes
Start Flash Loader Downloader and wait for a new image.
Jump to image in MCU flash

Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Sep.28.10	—	First edition issued
2.00	Mar.30.11	—	Updated for use with RX62N RDK
2.10	Jun.21.11	—	Added Section 6 on storing firmware images internally
3.00	Feb.05.13	—	• Moved to FIT template (added Overview, API Information, and API Functions sections). • Updated document to reflect new names based on Coding Standards v4.0 compliance. • This package was modified to be FIT compliant and more modular. This meant many smaller changes were made. • Added 'Board Notes' section which details how to configure supported boards. • Because of FIT compliance this code now works with many RX boards instead of just the RDKRX62N. • Added information about fixed vector table when using bootloader in User Boot Area. • Added more information on using Python and which external packages were used. • Added 'Calling Flash API from User Boot Area' section. • Added references to SD card implementation and the Flash API for RX • Added RX200 to list of available devices. • Added R_FL_GetVersion API information. • Added 'Enabling User Boot Mode in r_bsp Package' subsection. • Added steps for using E2Studio throughout "Implementing Flash Loader with Your Project" section. • Added 'Demo Projects' section.

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accord with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700, Fax: +44-1628-651-804

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-65030, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141