RL78 Family

RL78 Dual Image Bootloader Example

Introduction

This example project demonstrates a method to store two user applications in the RL78 and selectively execute each one using the RL78 boot swap feature. The project also uses the Flash Self-programming Library (FSL) to update the applications in flash and the hardware CRC module to verify data integrity.

Note that a bootloader's design is completely custom. Depending upon your project's requirements and limitations, your bootloader will be different. This project shows only one possible example; modifications can and should be made to adapt it to your specific requirements.

Target Device

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

RL78/G14: R5F104ML

Contents

1. Terminology 4

2. Overview 4

2.1 Hardware 4

2.2 Software 5

2.2.1 RL78 Bootloader 5

2.2.2 RL78 Application 5

2.2.2.1 Srec Batch Files 5

2.2.3 PC Program 5

2.3 CRC 5

2.4 Tools 5

2.4.1 e^2 studio 5

2.4.2 CC-RL 6

2.4.3 SREC CAT 6

3. System Architecture and Design 6

3.1 Embedded Memory Allocation 6

3.1.1 Application ID 7

3.1.2 Application Revision 7

3.1.3 ISR Jump Table 7

3.1.4 Application Program 7

3.1.5 Program CRC 7

3.1.6 Monitor Areas 7

3.1.7 Self RAM Area 8

3.2 Boot Swap 8

3.3 Near/Far Memory 8

3.4 Vector Table 8

3.4.1 Vector Table Area 8
3.4.2 Vector Table Contents 8
3.4.3 Interrupt Vectors with Dual Applications 8
3.5 ISR Processing 8
3.5.1 Vector Table Entry 9
3.5.2 Jump Table 9
3.5.3 ISR Handler 9
3.5.4 ISR Processing Example 9
3.5.5 Performance Impact of Jump Table Method 10
3.6 Program Images 10
3.6.1 Application Image 10
3.6.2 Boot Image 10
3.6.3 Combined Image 11
3.7 Embedded Code Flow 11
3.7.1 Downloading the Image 11
3.7.2 System Execution 13

4. Source Code Architecture 14
4.1 RL78 Bootloader 14
4.1.1 Build Configurations 14
4.1.2 Compiler Configuration 14
4.1.2.1 Memory Model 15
4.1.2.2 Locate ROM Data to Near Area 15
4.1.3 Linker Sections 15
4.1.4 Code Generator Output 16
4.1.4.1 Clock Generation Settings 16
4.1.4.2 Relevant Defined Values 16
4.2 RL78 Application 18
4.2.1 Build Configurations 18
4.2.2 Compiler Configuration 18
4.2.2.1 Memory Model 19
4.2.2.2 Locate ROM Data to Far Area 19
4.2.3 Linker Sections 19
4.2.4 Code Generator Output 21
4.2.4.1 Clock Generation Settings 21
4.2.4.2 Serial Settings 21
4.2.5 Flash Libraries 21
4.2.6 Relevant Defined Values 21
4.2.7 Post-Build Processes 23
4.3 PC Application 24
4.3.1 Overview 24

4.3.2 COM Port ... 24
4.3.3 File Reader ... 24
4.4 Srec Batch Files ... 24
4.4.1 Combining Boot and Application Images ... 25
4.4.2 CRC Generation ... 25
4.4.3 Full File Generation ... 25
4.4.4 Download File Generation ... 25

5. Running the Sample Program ... 26
5.1 Building the RL78 Projects ... 26
5.1.1 Importing the Projects ... 26
5.1.2 Selecting the Build Version ... 27
5.1.3 Building the Boot Project ... 27
5.1.4 Building the Application Projects ... 28
5.1.4.1 Initial Image (App 0) ... 28
5.1.4.2 Downloadable Image (App 1) ... 28
5.1.4.3 Downloadable Image (App 0) ... 28
5.2 Setting up the hardware ... 28
5.2.1 Run Mode ... 28
5.2.2 UART Communication ... 28
5.2.3 Power ... 29
5.3 Running the PC Application ... 29
5.3.1 Selecting a COM Port ... 29
5.3.2 Selecting the File to Download ... 29
5.3.3 Download ... 30
5.3.4 Additional Information ... 30
5.3.4.1 Application Region Destination for Downloads ... 30
5.3.4.2 Regression Protection ... 30
5.3.4.3 Additional Interface Controls ... 30

6. Website and Support ... 31

Revision History ... 32

1. Terminology

Table 1. Terms and Definitions

Term	Definition	Hardware/Software
Boot Region A	Denotes the physical memory location from address 0x00000000 to 0x00000FFF.	Hardware Terminology
Boot Region B	Denotes the physical memory location from address 0x00001000 to 0x00001FFF.	
Lower Application Region	Denotes the physical memory location from address 0x00002000 to 0x00040FFF.	
Upper Application Region	Denotes the physical memory location from address 0x00041000 to the end of flash memory.	
Boot program	Refers to a program that resides in one of the two Boot Regions. This program functions as the bootloader, handling the initial program entry and Application verification. “Boot” is also the name of the associated e^2 studio project.	Software Terminology
Application program	Refers to a program that resides in one of the two Application Regions. This program contains the bulk of the user code. “Application” is also the name of the associated e^2 studio project.	
n	Variable used to denote Boot/Application version, either 0 or 1.	
Boot *n*/App *n*	Denotes the Boot and Application image pair *n*.	
Boot 0/App 0	Denotes the Boot-Application pair built for version 0. App 0 goes in the Lower Application Region.	
Boot 1/App 1	Denotes the Boot-Application pair built for version 1. App 1 goes in the Upper Application Region.	
Vector Table Entry	The initial function called when an interrupt occurs. Vector Table Entry can also refer to the function’s 16-bit address location, which is stored in the vector table for a particular interrupt.	

2. Overview

2.1 Hardware

This project takes advantage of the boot swap feature of the RL78 to store two separate user applications in MCU flash and determine which application will execute based on which boot program is at memory address 0x00000000.

This module is designed to run with the RL78/G14 Fast Prototyping Board. The bootloader interfaces with the FSL to program new applications to the device. The techniques used in this module can be adapted to any RL78 device that uses boot swap. Refer to the hardware manual of the specific MCU you are using for any differences.

These applications are downloaded in real time through a UART connection to UART0. The PC system connects to this UART through PMOD1. A USB-to-UART converter is needed for communication. This module was developed using a TTL-232R-3V3 converter. The converter’s TXD line is placed into PMOD1 pin 3, the RXD line is placed into PMOD1 pin 2, and GND to PMOD1 pin 5.

This module also makes use of shorting EJ1. When this jumper is shorted, the on-board debugger is held in reset and the MCU is set to Run mode.

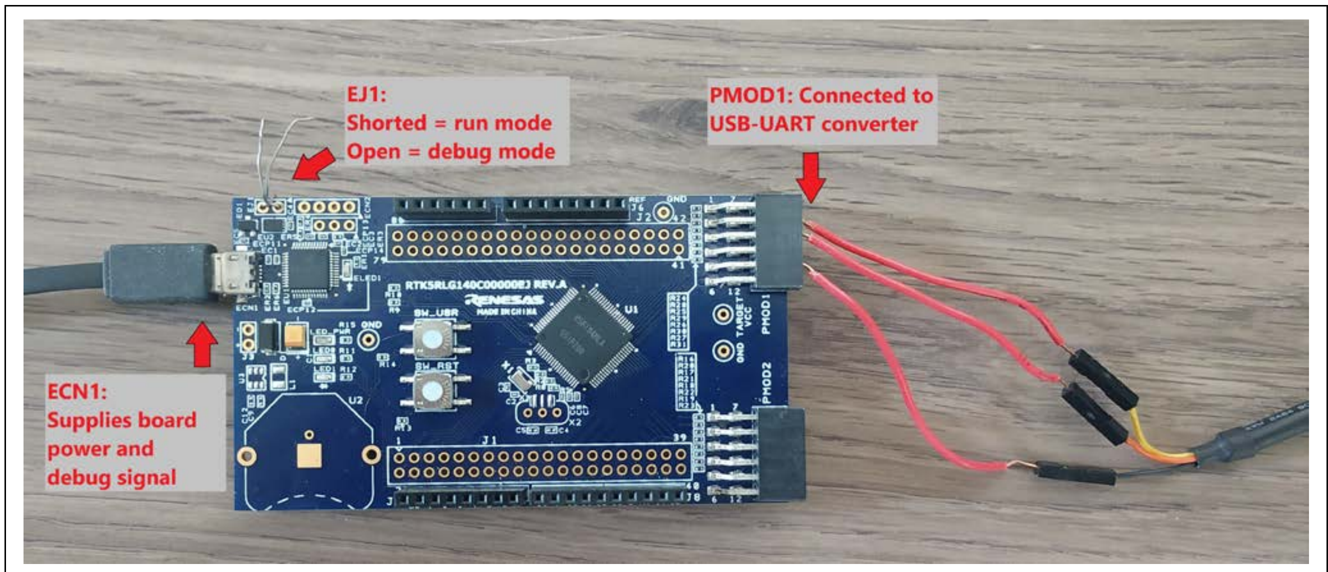

Figure 1. Hardware Setup of RL78/G14 FPB

2.2 Software

This module's software can be viewed as 3 separate components: the RL78 Bootloader, the RL78 Application, and the PC program. The batch files used with the `srec_cat.exe` program are located within the RL78 Application.

2.2.1 RL78 Bootloader

The bootloader is used as the initial program entry. It verifies the integrity of the current Boot/Application image by CRC calculation and, if the CRC is valid, jumps to the current Application start address.

2.2.2 RL78 Application

The Application program is intended to act as a stand-in for a custom application. Its only functions are to turn on an LED, handle UART communication, and flash new application images.

2.2.2.1 Srec Batch Files

The srec files are located inside the RL78 Boot and Application projects, in the `\srec` directory. Each project executes these scripts as part of its post-build process.

The srec batch files are used to generate the files used to program the RL78 device. The CRC for the application is calculated here and placed in the last two bytes of the current Application's memory space. Hex files are generated from the Boot/App pairs and their respective CRCs, and these files are used by the PC program to download a new version of the application to the RL78 device. MOT files are also generated from the Boot/App pairs. The MOT files are used as an initial image to be flashed to the RL78 device with Renesas Flash Programmer.

The batch files are executed with the tool `srec_cat.exe`. This tool can be found online and is not affiliated with Renesas.

2.2.3 PC Program

The PC program is used to parse the files generated from the srec output and download them to the RL78 device.

2.3 CRC

All CRCs used in this application are CRC-CCITT-Kermit functions, with a polynomial of $x^{16} + x^{12} + x^5 + 1$. This is to keep uniform with the RL78's hardware CRC that uses this function.

2.4 Tools

2.4.1 e² studio

This example program uses the e² studio IDE, version 2023-04 for development.

2.4.2 CC-RL

This example program uses the CC-RL v1.12 toolchain/compiler.

2.4.3 SREC CAT

This example program uses the third party `srec_cat.exe`. At the time of writing this document, `srec_cat` and supporting documentation can be found at http://srecord.sourceforge.net/ .

3. System Architecture and Design

3.1 Embedded Memory Allocation

Figure 2 shows the memory segments for this bootloader solution. It is broken down into sections called Boot Region A, Boot Region B, Upper Application Region, and Lower Application Region. This terminology will reference the memory locations themselves.

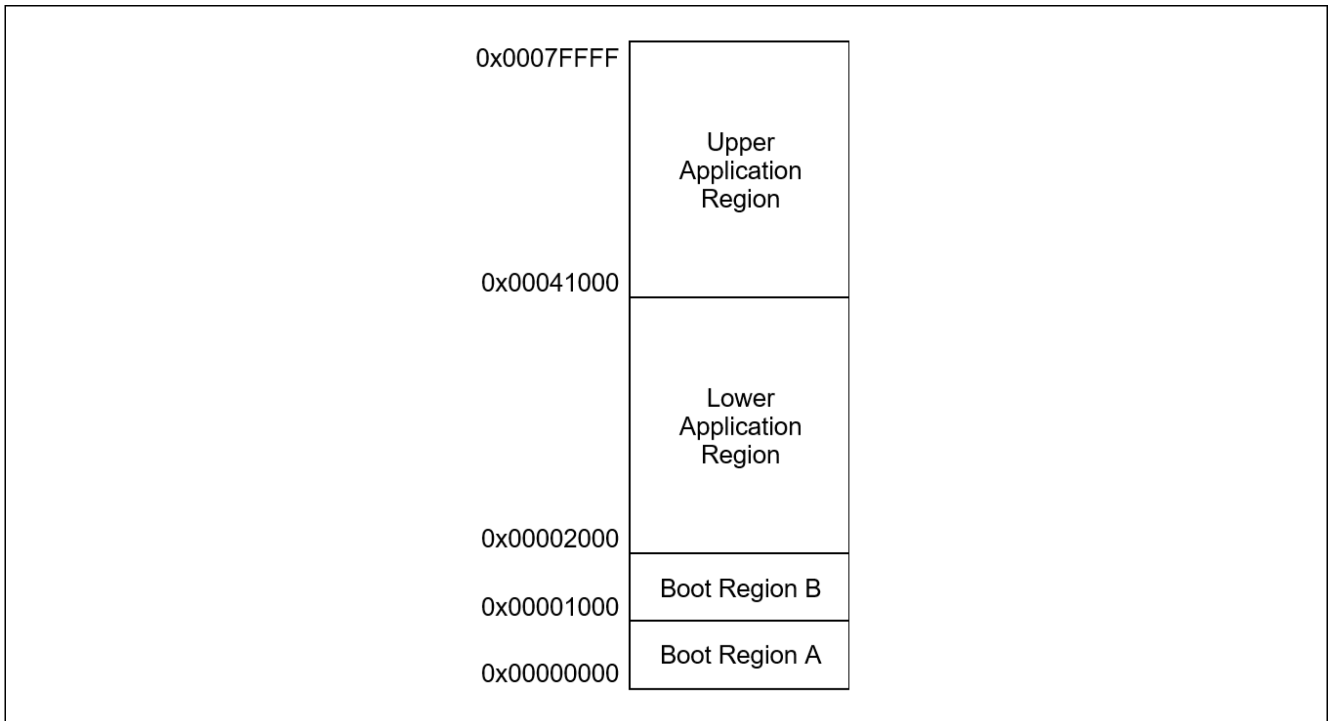

Figure 2. Memory Map of Boot and Application Regions

Boot Region A refers to the first boot block. In the case of the G14, this region exists from memory addresses 0x00000000 to 0x00000FFF.

Boot Region B refers to the second boot block. This segment exists from addresses 0x00001000 to 0x00001FFF.

Upper/Lower Application Region refers to the two memory areas reserved for the Application programs. The combined memory region of both extends from 0x00002000 to 0x0007FFFF. Each Application uses half of this available memory, so the Lower Application Region extends from 0x00002000 to 0x00040FFF, and the Upper Application Region extends from 0x00041000 to 0x0007FFFF.

App 0 and App 1 refer to the Application programs that will reside in the Upper and Lower Application Regions. App 0 will always reside in the Lower Application Region while App 1 will always be in the Upper Application Region.

Figure 3 shows a more detailed memory map, breaking down the Upper/Lower Application regions into their individual parts.

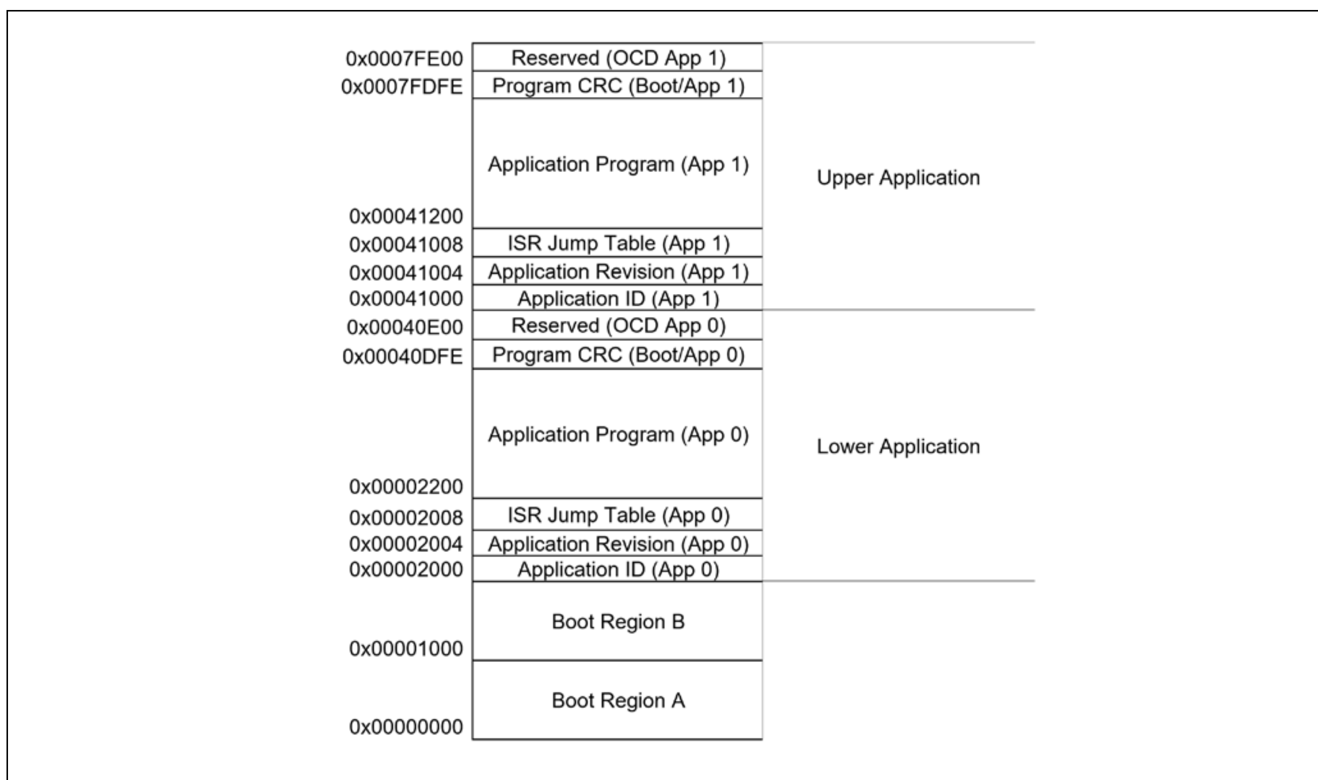

Figure 3. Detailed Memory Map of Boot/App Images

3.1.1 Application ID

Each Application program is compiled with an Application ID field to identify whether it belongs in either the Upper or Lower Application region. The Application ID is an ASCII-coded, 4-byte field located at offset 0x00000000 in the Application region. The Application ID is “APP0” if the Application belongs in the Lower Application Region, and is “APP1” if the Application belongs in the Upper Application Region.

3.1.2 Application Revision

Each Application program contains a 4-byte revision field to identify the firmware build revision. This field is located at address offset 0x00000004 from the start of the Application Region.

3.1.3 ISR Jump Table

The ISR jump table is an array of 32-bit function pointers to the ISR handlers, and it is part of the Application program. This table is necessary due to the ISR vector table being limited to 16-bit addresses. It allows for ISR handlers to be placed in the App 1 region of flash, beyond the 16-bit address range. For details, see section 3.5.2. The jump table is located at address offset 0x00000008 from the start of the Application region.

3.1.4 Application Program

The Application area contains the user code for that Application program, either App 0 or App 1. This area includes the ISR handlers that are pointed to by the ISR jump table. The Application program is located at address offset 0x00000200 from the start of the Application region.

3.1.5 Program CRC

The Program CRC section contains the CRC of the combined Boot/App image. It uses the 16-bit CRC-CCITT with polynomial representation 0x8408. The CRC calculation is performed as if the Boot/App programs are one contiguous block of data. The CRC is located at address offset 0x0003EDFE from the start of the Application region.

3.1.6 Monitor Areas

This example project uses the On-Chip Debug (OCD) option. This is not necessary at all; however, it is left in to showcase an example on how to work with the OCD used. In general, when building the application, the OCD will be set, then subsequently removed once the application is deployed.

The monitor area will be the last 512 or 256 bytes of each application image. This prevents you from using these areas for program space. Any attempt to overlap code with this section will cause the linker to

generate an error. For more information on these sections, refer the hardware manual chapter on the On-Chip Debug Function.

Since the example project uses the OCD setting, even though memory extends to 0x0007FFFF, its program image can only go to address 0x0007FDFF. In this document, the terms "End of memory" and "512k" refer to address 0x0007FDFF as the OCD occupies the final 0x200 bytes of memory.

Also note that, when an image is downloaded, the monitor area is not included in the download. This is because once the image is ready to download, debugging is complete, and the monitor area is no longer necessary.

3.1.7 Self RAM Area

Because this example uses the FSL, there is a dedicated section of RAM known as the Self RAM area that the FSL will use. The project configuration reserves this area specifically for the FSL, and no other data is allowed to use it. The Self RAM area for this MCU (R5F104ML) is the first 1 KB of RAM, from address 0x000F3F00 to 0x000F42FF.

3.2 Boot Swap

An important thing to keep in mind is that, with the RL78's boot swap ability, Boot Region A and Boot Region B are interchangeable. When the boot swap bit is set, these two segments will physically swap memory locations. For example: if the boot swap bit is set, and you attempt to access the memory at address 0x00000080, you will read what you originally programmed to address 0x00001080.

Because of the boot swap feature, Boot 0 or Boot 1 can reside in either Boot Region A or Boot Region B based on the status of the boot swap bit. It is this swap action that allows for the two applications to be "selectable" by pairing Boot 0 with App 0 and Boot 1 with App 1.

3.3 Near/Far Memory

The RL78 can use 16-bit addressing for functions and data. This uses what is known as "near" memory. However, as previously mentioned, 16-bit addressing is not sufficient for the upper Application region, so it is necessary to use the 20-bit "far" memory addressing.

The CC-RL compiler options determine the default near/far placement of functions and variables in the program. In some cases, the type qualifiers `__near` and `__far` are used in declarations to override the default placement and explicitly state the allocation area. See sections 4.1.2 and 4.2.2 for information on what compiler options are used for this example.

3.4 Vector Table

3.4.1 Vector Table Area

The RL78's vector table is fixed in memory at address 0x00000000 and takes up the first 128 bytes of memory. If boot swap is active, then this table will be at address 0x00001000; however, the MCU will only use the vector table contents at address 0x00000000 at MCU reset. Boot 0 contains the vector table for App 0, and Boot 1 contains the vector table for App 1.

3.4.2 Vector Table Contents

The vector table contains the addresses for program start and interrupt processing. Each entry is 16 bits in length, meaning that the possible jump addresses range from 0x00000000 to 0x0000FFFF.

3.4.3 Interrupt Vectors with Dual Applications

Because the vector table addresses are limited to 16 bits, and App 1 lies beyond this addressable memory space, special care must be taken to ensure proper interrupt processing. See section 3.5 for details.

3.5 ISR Processing

The primary ISR handlers reside in application space, and the RL78's vector table is in the boot area. However, because the vector table only allows 16-bit addresses, and since Application 1 lies beyond the 16-bit address range, a jump table is necessary.

Both Boot/App image pairs maintain their own ISR vector tables, jump tables, and ISR handler addresses for the area of flash in which they will be placed.

3.5.1 Vector Table Entry

When an interrupt is triggered, the function address in the vector table (referred to as the Vector Table Entry) is called. This function resides in the program area of the Boot region, and its purpose is to get the appropriate function pointer from the ISR jump table and call that function.

3.5.2 Jump Table

The jump table resides at the beginning of each application and is an array of function pointers to the ISR handlers. When an interrupt occurs, the initial Vector Table Entry function is called, which immediately calls the handler function at the designated jump table address.

Each jump table entry is kept at a fixed address in memory. Since the jump table address is static, and each entry is 4 bytes in length, it is possible for the Boot program to know the location of the appropriate jump table entry. These addresses are defined in the source code.

With this technique, the linker can place the ISR handler function anywhere it determines, and the jump table will always use the correct pointer to the handler.

3.5.3 ISR Handler

An ISR handler is a function located in the Application program that performs the actions required when a particular interrupt occurs. Its address is placed on the ISR jump table at build time.

3.5.4 ISR Processing Example

The following is an example of how an ISR is processed from beginning to end. Figure 4 and Figure 5 show addressing and handling of a theoretical interrupt for App 0 and App 1. The process is the same for both images; the only differences are the memory addresses used, specifically those of the ISR jump table and ISR handler function. The App 0 program memory starts at address 0x00002000 and the App 1 program memory starts at address 0x00041000.

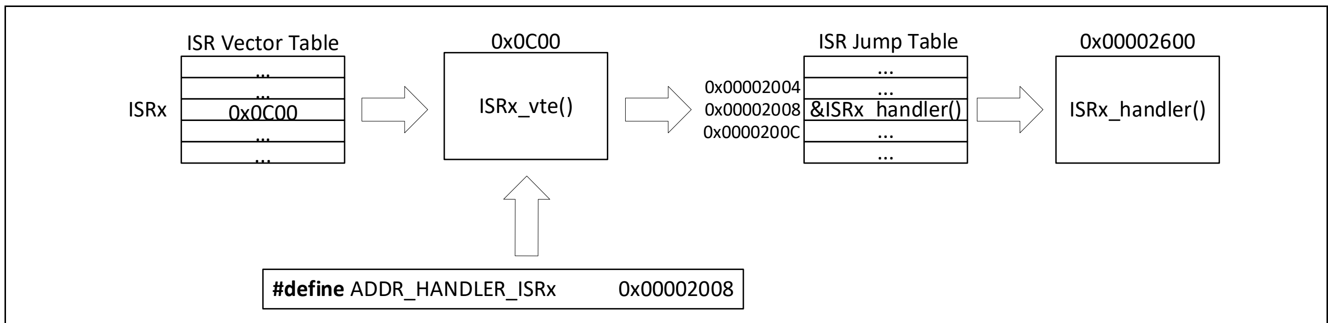

Figure 4. Example ISR Process Flow for App 0

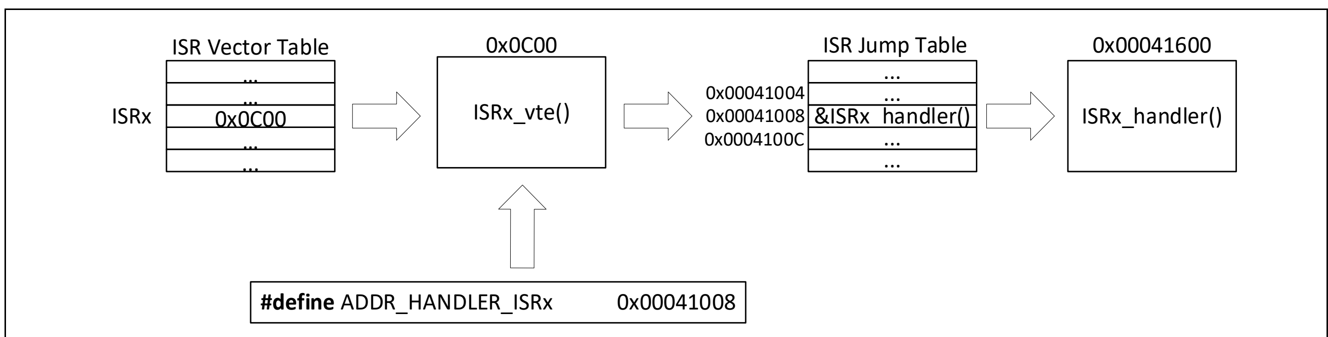

Figure 5. Example ISR Process Flow for App 1

This example uses a theoretical ISR known as ISRx to demonstrate the process flow. The address values used are for example only and will not necessarily be the values used by the program, though the memory range restrictions for each Application still apply.

Referring to Figure 4, the ISR process is as follows:

- The linker places the ISRx Vector Table Entry function at address 0x00000C00 in the Boot region.
- The linker places the ISRx handler function at address 0x00002600 in the Application region.
- ISR jump table is placed at 0x00002004 in the Application region.
- The address of ISRx_handler() is located at index ‘1’ of the jump table, so it will be at address 0x00002004 + (1 x 4), which is 0x00002008. Even though the actual address of ISRx_handler() may be changed by the linker at build time, the jump table entry will always point to the correct address.

Note: The examples in Figure 4 and Figure 5 both use the lower section of boot program space. This is because the active Boot program will always start at address 0x00000000 regardless of the location of the current Application program.

- When ISRx is triggered, its Vector Table Entry (ISRx_vte()) is called.
- ISRx_vte() gets the location in the jump table where the address of function ISRx_handler() is stored, creates a function pointer from that address, and calls the function.
- ISRx_handler() performs the necessary interrupt handling.

3.5.5 Performance Impact of Jump Table Method

Because the ISR Jump Table method introduces an extra interrupt processing step, there is a small impact on MCU performance. For each interrupt instance, the additional function call will require a stack entry as well as some extra CPU cycles to process the added instructions. While the performance impact on a single interrupt event is minimal, it is important to evaluate the extra delay if used in applications that are more time-sensitive.

3.6 Program Images

This section describes each of the images created in the example project and their contents.

3.6.1 Application Image

Figure 6 shows a physical representation of the memory layout of the Application project output. Depending on the build configuration used, the output image will use either the Upper or Lower Application region. Notice that all other memory segments are empty.

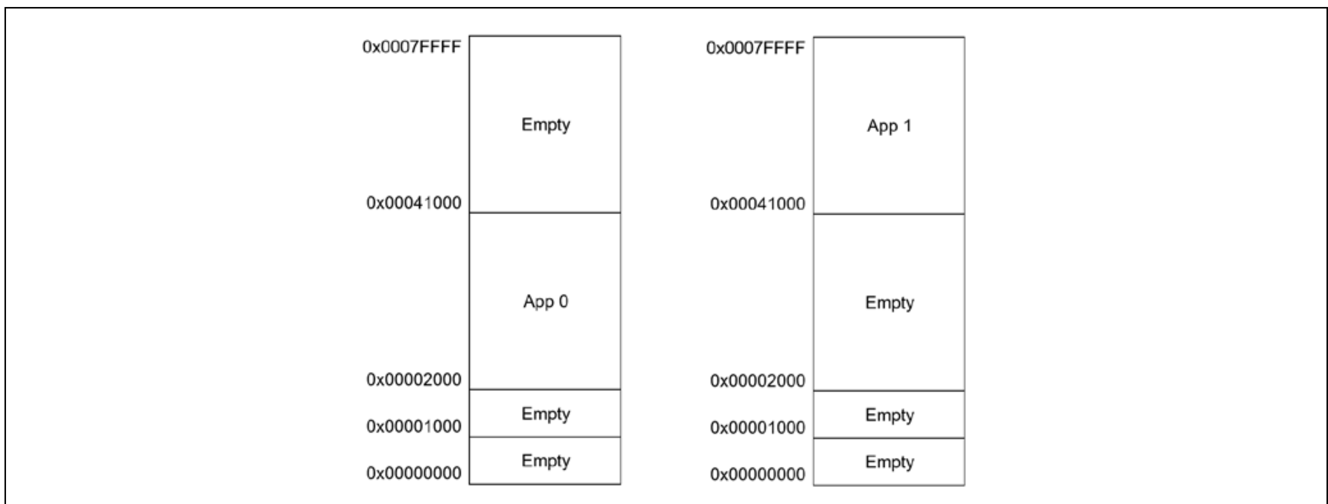

Figure 6. Memory Placement of Application *n* Images

The generated Application image will be combined with the appropriate Boot image in post-build steps.

3.6.2 Boot Image

Figure 7 shows a physical representation of the memory layout of the Boot project output. Both Boot images are built and linked to starting address 0x00000000 by default. However, either Boot image can be stored at address 0x00001000 until needed.

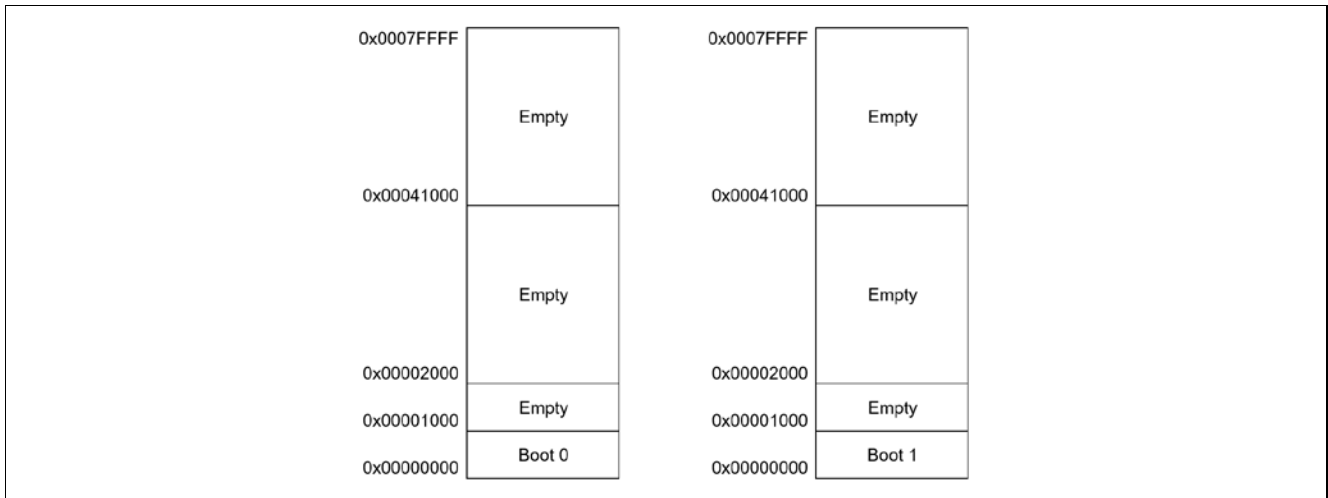

Figure 7. Memory Placement of Boot *n* Images

3.6.3 Combined Image

Figure 8 shows both the Application *n* and Boot *n* programs combined into one image. This figure shows how the image generated during post-build will look for each build configuration.

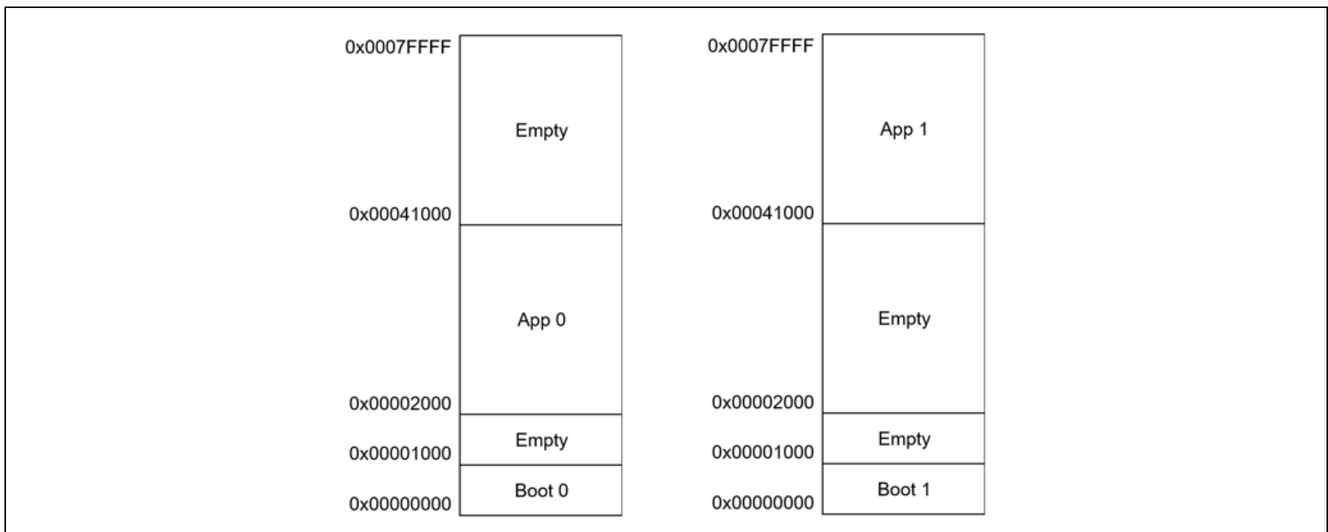

Figure 8. Memory Map of Each Boot *n*/App *n* Image

3.7 Embedded Code Flow

3.7.1 Downloading the Image

The download works on a Command-ACK structure. The PC program sends a command and the RL78 will completes a task according to the received command. When the task is completed, an ACK or NAK is sent to the server, indicating that the MCU is ready to receive another command.

Each command and response is compiled into a packet. Command packets consist of 69 bytes. Figure 9 shows the structure of the command packets; Table 2 shows all the possible types of command packets that can be sent.

The packet CRC is used as a check for packet integrity. Both the packet counter and the data are CRC'd and that CRC is appended to the end of the message. To verify the CRC, the example project runs the same CRC algorithm (in this case the general hardware CRC) on the entire packet, including the appended CRC, and verifies that the CRC operation resolves to 0.

Command Packet (69 bytes)		
Packet Counter	Data	Packet CRC
Bytes 0 - 1	Bytes 2 - 66	Bytes 67 - 68

Figure 9. Command Packet Structure

Table 2. UART Command Codes

Command	Command ID	Description
Data	0x01	This packet contains the next 64 bytes ready to be written to the MCU as well as a counter for the packets. When a data packet is received, the RL78 takes the data and programs it into the next available 64 bytes in flash.
Version	0x02	This packet requests the version number of the application from the RL78 device.
Reset	0x03	This packet instructs the RL78 to reset without initiating a boot swap.
Transfer Complete	0x04	This packet informs the RL78 that the last packet has been transmitted and the download has finished.
Swap and Reset	0x05	This packet instructs the RL78 to reset after initiating a boot swap.
Transfer Start	0x06	This packet signals to the RL78 that a firmware update attempt is starting. The RL78 will erase Boot Region B and the alternate Application to prepare for the new image.
Get Boot Swap State	0x07	This packet requests the RL78 to send the current state of the boot swap bit.
Get Active App	0x08	This packet requests the RL78 to send the App ID of the main Application.

When the RL78 receives a message from the PC program, the RL78 sends a response. These responses follow every message from the PC application. Figure 10 shows the structure of the response packets; Table 3 shows all the possible types of response packets that can be sent.

Response Packet (5 bytes)		
Response type	Data A	Data B
Byte 0	Bytes 1 - 2	Bytes 3 - 4

Figure 10. Response Packet Structure

Table 3. UART Command Response Codes and Data

Response	Response ID	Data A Usage	Data B Usage	Description
Begin Transmission	0x01	Fill - 0xAA	Fill - 0xAA	The device has been set into bootloader mode and is ready for programming.
ACK Packet	0x02	Packet Count	Current running CRC	The packet was flashed successfully.
NAK Packet	0x03	Packet Count	Fill - 0xAA	The packet failed to flash properly.
Send Version	0x04	Version High Bytes	Version Low Bytes	Contains the version of the application currently flashed to the device.
ACK Program	0x05	Packet Count	Current running CRC	The program was flashed correctly and verified.
NAK Program	0x06	Packet Count	Current running CRC	The program failed the CRC check.
NAK CRC	0x07	Packet Count	Fill - 0xAA	The CRC for the last packet sent did not validate.
Boot Swap State	0x08	Boot swap status	Fill - 0xAA	Contains the current state of the boot swap bit.
Active App	0x09	Bytes 0-1 of App ID	Bytes 2-3 of App ID	Contains the App ID field for the current Application.

The example program uses the FSL to program the new application. For more information about the specifics of the FSL, refer to the FSL documentation (https://www.renesas.com/us/en/software-tool/code-flash-libraries-flash-self-programming-libraries).

After the program receives the Transfer Start command, it erases Boot Region B and the alternate Application region to prepare for the new image. Once the erasure is complete, the program notifies the server that it is ready to begin flashing.

As each chunk of data (64 bytes at a time) is received, it is written to the flash area. A counter keeps track of the location of the writes and is incremented with each chunk. The counter begins at address 0x00001000 and increments by 64 until it reaches the end of Boot Region B. Once the counter reaches address 0x00002000, the program sets the counter to the start address of the alternate Application region, either 0x00002000 or 0x000041000. The process continues until the final packet is written.

The total number of packets is determined by the following formula:

$$(Size_{Boot} + Size_{App\ Region} - Size_{OCD}) / Size_{Packet}$$

Where:

$Size_{Boot}$ = 0x1000
$Size_{App\ Region}$ = 0x3F000
$Size_{OCD}$ = 0x200
$Size_{Packet}$ = 64

The calculation for this example results in a total of 4088 packets.

3.7.2 System Execution

The images in Figure 11 show a high-level overview of the code flow and the entire download process. For the purposes of this example, the initial state has both Boot 0/App 0 and Boot 1/App 1 programmed in flash.

Below is a brief description of the steps completed in Figure 11:

- Initial State
  — The part has been flashed with the Boot 0/App 0 and Boot 1/App 1 images as seen in Figure 8.
- Step 1
  — The current Application (App 0) verifies, through the UART, that a new image is ready to download.
  — The current Application erases Boot Region B (which currently contains Boot 1) and the alternate Application area (which contains App 1).
- Step 2
  — The current Application downloads the newest image from the UART and flashes this into Boot Region B and the alternate Application area.
- Step 3
  — The current Application verifies the CRC of the new alternate Application (App 1) before swapping the boot sectors and initiating a reset.
  — The alternate Application that was just updated is now the current Application.

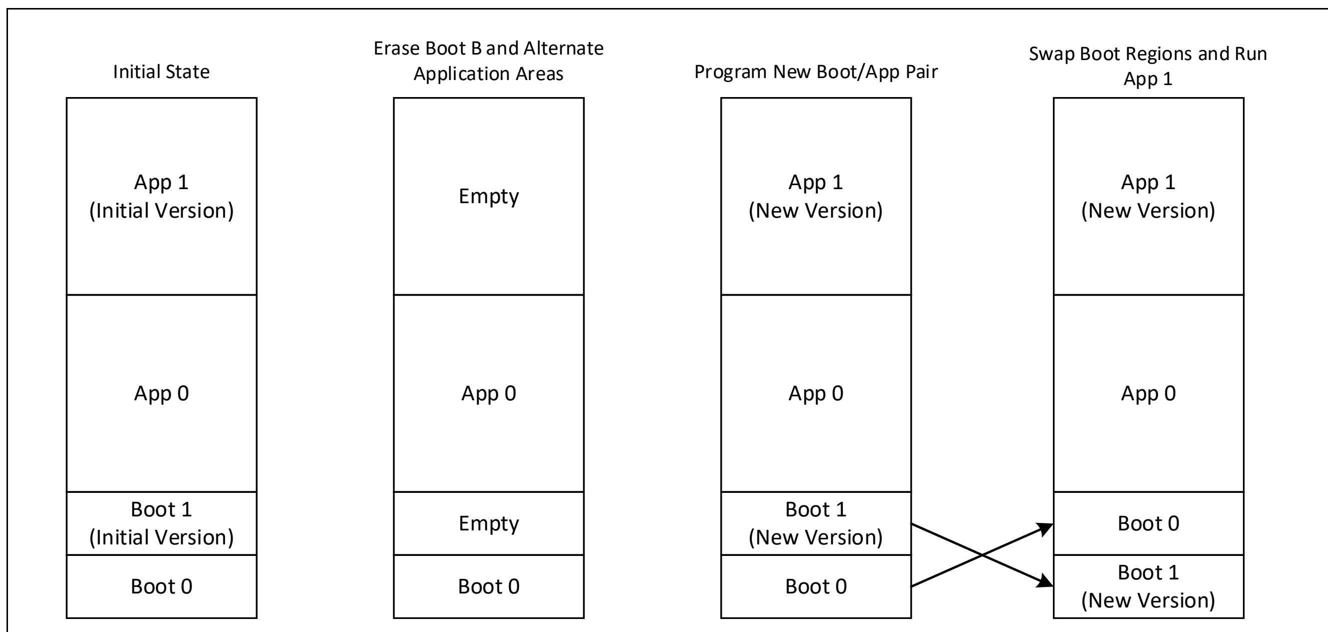

Figure 11. Example Flash Process for Firmware Update

4. Source Code Architecture

This example project is intended as a guide for your own application. It is expected that changes will be made to the example project to accommodate a custom application.

4.1 RL78 Bootloader

The bootloader portion of this example is contained in the e^2 studio project "Boot."

4.1.1 Build Configurations

The Boot project contains two build configurations: one to generate a Boot 0 image, and another to generate a Boot 1 image.

4.1.2 Compiler Configuration

The build configuration for Boot 0 uses the `-D` option to create the macro definition `BUILD_VER_0`. In the source code, there are `#ifdef` statements that are used to define which set of memory addresses will be used in the build. With `BUILD_VER_0` defined, the compiler will use the addresses for the App 0 program.

The Boot build configurations use the near memory area for ROM and the far area for functions by default. To do this, the compiler is configured as shown in Figure 12.

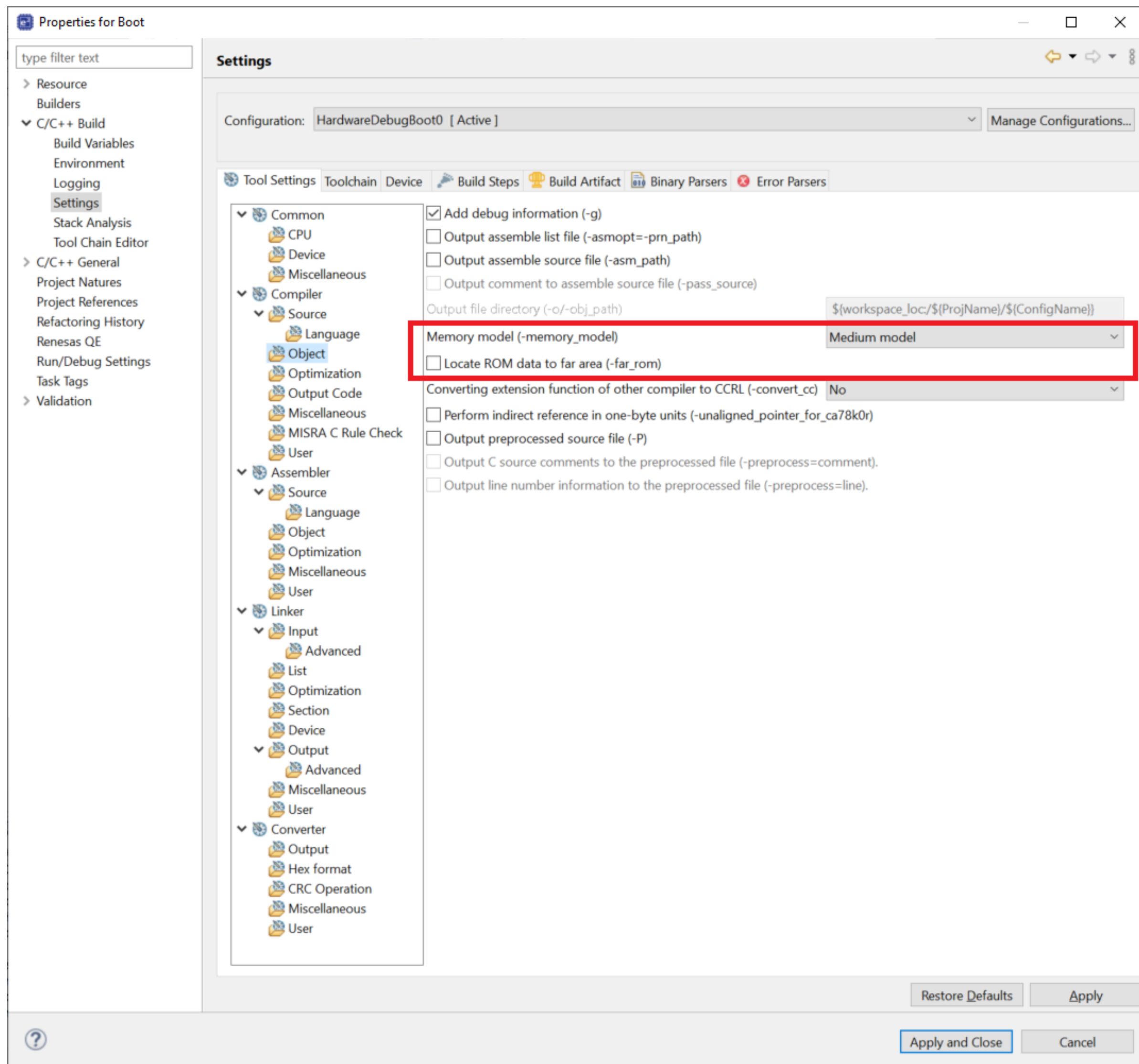

Figure 12. Compiler Options for Boot Memory Configuration

4.1.2.1 Memory Model

The compiler option `-memory_model` is set to "medium" so that functions will use the far attribute by default. The bootloader uses function pointers to memory addresses that can be located outside of the near area, so the compiler needs to use the far attribute for these to avoid compiler warnings.

4.1.2.2 Locate ROM Data to Near Area

The compiler option `-far_rom` is left unused to ensure that any ROM data will use the near attribute by default. This is done to ensure that no data will be placed using 20-bit addresses, which is far beyond our allowed bootloader address space. However, it will still be necessary to verify that no boot code is placed beyond address 0x00001000 since the near memory ranges from 0x00000000 to 0x0000FFFF. See section 3.3 for information on near/far memory.

4.1.3 Linker Sections

Figure 13 shows the section output the bootloader project was built with. Apart from the modifications required from the flash libraries, the bootloader project itself does not require any modifications to the sections. However, there are some important details to observe. It is highly recommended to review the map file to ensure that the code fits to the following criteria:

- **Memory Limitation:** Ensure that all data fits within memory addresses 0x000000D8 to 0x00000FFF. Before 0x000000D8 is reserved for intrinsic values such as the vector table and option bytes. Above this memory range is beyond the boot swap sector.
- **Constant Data:** The `.const` section cannot exist outside of the mirror area. Since the mirrored area of this MCU starts at address 0x00003000, which is outside of the boot area, no data can be placed in the `.const` section. However, the `.constf` section, which uses the far memory attribute, can be used.
- **Flash Sections:** The flash sections must exist in the project. It is unimportant where they are located exactly as long as the location is in the first 0x1000 allocation.
- **RAM Sections:** General RAM sections are allocated starting at address 0x000F4700. This is to avoid allocation in the area reserved for Self RAM and OCD trace.

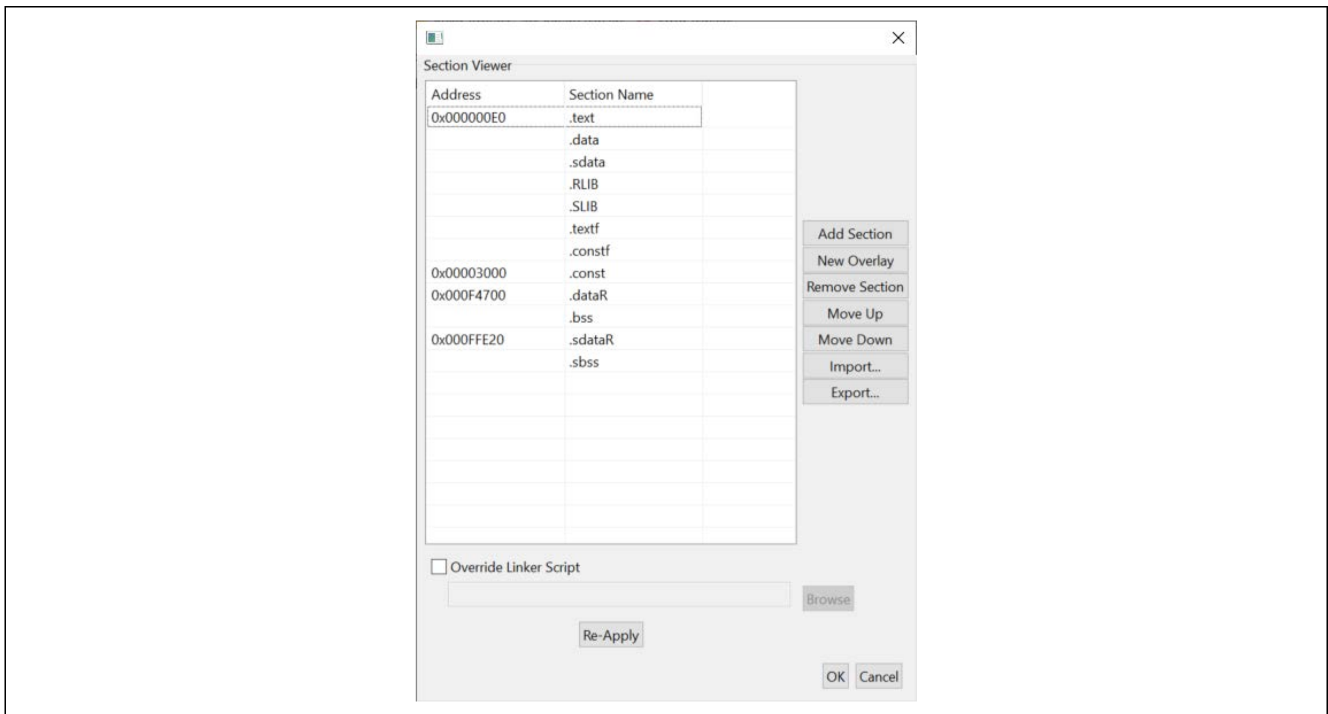

Figure 13. Linker Sections for Boot Project

4.1.4 Code Generator Output

The code generator is used to output most of the driver source code. All code generator source files are in the `/src/CodeGenerator` directory. When implementing the bootloader concepts from this example project into a custom application, it may be necessary to modify the generated code to meet the application's requirements.

4.1.4.1 Clock Generation Settings

The example project uses a HOCO setting of 64 (fHOCO = 64MHz; fIH = 32MHz).

4.1.4.2 Relevant Defined Values

Table 4 describes each of the defines that may need special attention when integrating this example program with a custom application.

Table 4. Bootloader Project Macro Definitions

Header File	Macro Definition	Description	Fixed Value or Build-Dependent
`utility.h`	`UART_PACKET_LENGTH`	Size of packets moving from the PC to the MCU.	Fixed Value
	`UART_RESPONSE_LENGTH`	Size of packets moving from the MCU to the PC.	
	`MAX_PACKETS`	Number of packets in the image.	

Header File	Macro Definition	Description	Fixed Value or Build-Dependent
	CRC_ZERO_SEED	Initial seed for the running CRC of the entire image.	
	CRC_OK	When a CRC verifies against any data segment with that segments CRC appended at the end, the CRC will always evaluate to zero.	
	VERSION_ADDRESS	Physical address where the version information is located.	
	ADDR_BOOTA_START	Starting address of Boot Region A.	
	ADDR_BOOTB_START	Starting address of Boot Region B.	
	ADDR_APPn_FLASH_START	Starting address of Application Region *n.*	
	ADDR_VERSION_APPn	Address where the Application *n* revision is located.	
	ADDR_APPn_ISR_START	Starting address of the App *n* ISR jump table.	
	ADDR_APPn_START	Starting address of the App *n* program.	
	ADDR_APPn_END	End address of the App *n* program; excludes reserved memory for OCD region.	
	ACTIVE_APP_ID	Application ID of the main application.	Build-dependent
	ADDR_MAIN_FLASH_START	Start address of the main Application Region.	
	ADDR_ALT_FLASH_START	Start address of the alternate Application Region.	
	ADDR_MAIN_VERSION	Start address of the main Application revision field.	
	ADDR_ALT_VERSION	Start address of the alternate Application revision field.	
	ADDR_MAIN_APP_START	Start address of the executable portion of the main Application.	
	ADDR_MAIN_APP_END	End address +1 of the executable portion of the main Application (includes CRC).	
	ADDR_ALT_APP_START	Start address of the executable portion of the alternate Application.	
	ADDR_ALT_APP_END	End address +1 of the executable portion of the alternate Application (includes CRC).	
	ADDR_MAIN_ISR	Start address of the main Application ISR jump table.	
	ADDR_ALT_ISR	Start address of the alternate Application ISR jump table.	
	ADDR_ISR_UART0_TX	Address of the jump table entry for the UART0 Tx interrupt.	
	ADDR_ISR_UART0_RX	Address of the jump table entry for the UART0 Rx interrupt.	

Header File	Macro Definition	Description	Fixed Value or Build-Dependent
fsl_interface.h	BLOCK_XXX_YYY	These defines show the start and end blocks of each memory segment. The blocks are the 1KB memory blocks. For example: The define BLOCK_BOOT0_BEGIN gives the first memory block for the Boot 0 memory segment, which is 0. Since the Boot 0 memory segment is 0x1000 bytes long (or 4KB), the last block inside Boot 0 would be the 3rd block or BLOCK_BOOT0_END.	Fixed Value
	BYTES_PER_WRITE	The size of each write to flash memory by the FSL.	
bsp.h	PORT4_OUTPUT	Used for LED initialization settings for the Fast Prototyping Board; a custom board may differ.	
	PM_43_44_MODE_OUTPUT	Used for LED initialization settings for the Fast Prototyping Board; a custom board may differ.	
	LED0_PIN	Used for toggling the pin for LED0.	
	LED1_PIN	Used for toggling the pin for LED1.	

4.2 RL78 Application

4.2.1 Build Configurations

The Boot project contains two build configurations: one to generate the App 0 image, and another to generate the App 1 image.

4.2.2 Compiler Configuration

The build configuration for App 0 uses the -D option to create the macro definition BUILD_VER_0. In the source code, there are #ifdef statements that are used to define which set of memory addresses will be used in the build. With BUILD_VER_0 defined, the compiler will use the addresses for the App 0 program.

The App 1 build configuration needs special attention to ensure that the linker does not place any code or ROM data outside of the Application 1 boundaries. To do this, it is necessary to use the far memory attribute wherever possible. The CC-RL compiler includes options that will automatically use the far memory for the program. See Figure 14 for the relevant compiler options.

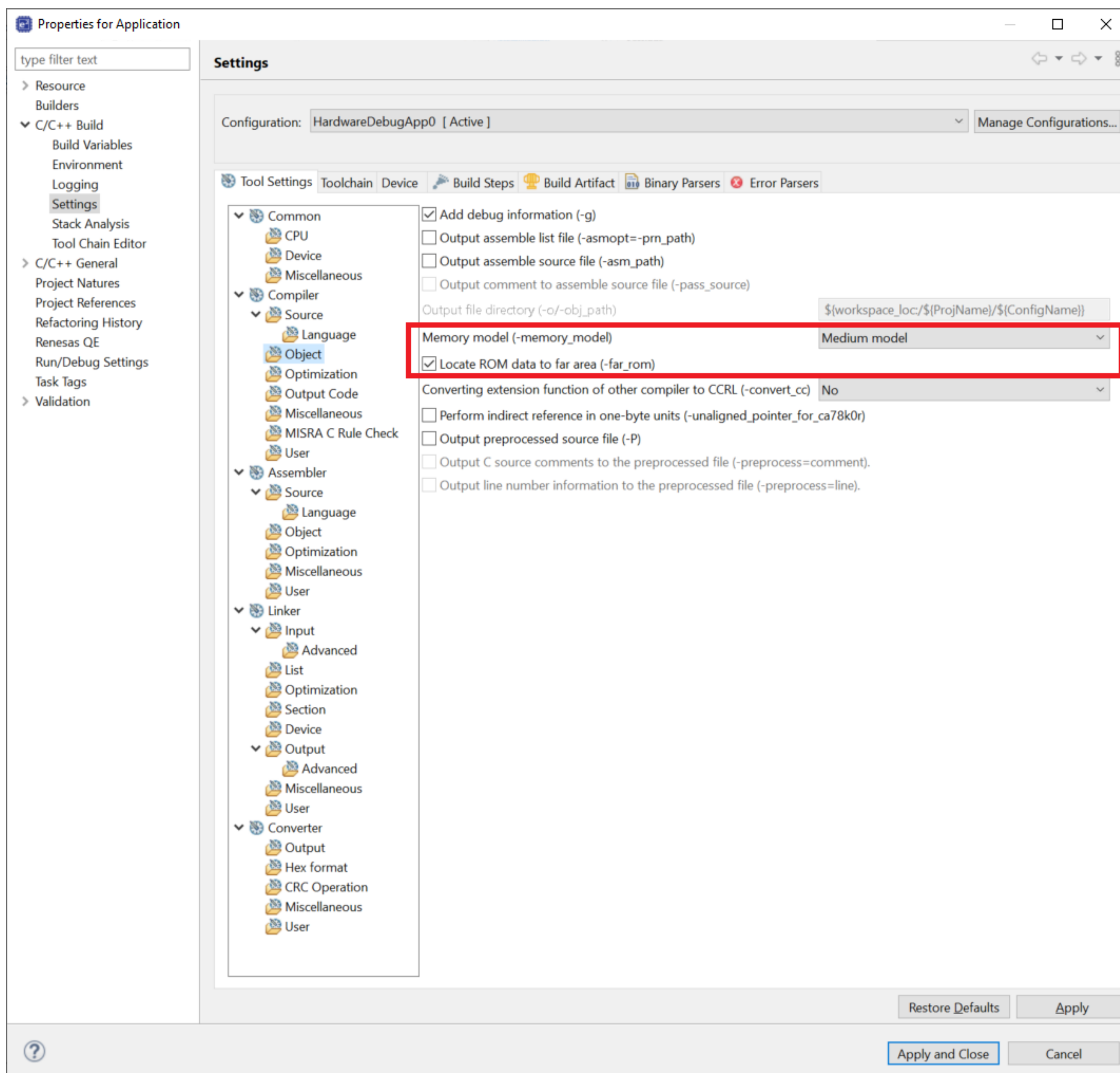

Figure 14. Compiler Options for Application Memory Configuration

4.2.2.1 Memory Model

The compiler option `-memory_model` is set to "medium" so that functions will use the far attribute by default.

4.2.2.2 Locate ROM Data to Far Area

The compiler option `-far_rom` is used to ensure that any ROM data will use the far attribute by default. This is done so that variables do not need the explicit `__far` type qualifier in their declarations. See section 3.3 for information on near/far memory.

4.2.3 Linker Sections

The application project requires a few more changes. Figure 15 shows the section output the Application project was built with. It is highly recommended to review the map file to ensure that the code meets the following criteria:

- **Application ID:** The Application ID is placed in the `.constfAPP_ID_f` section. This is a 4-byte section used to hold a char array identifying the Application image as App 0 or App 1. The array is declared in `main.c`.
- **Application Version:** The version information is held in the `.constfAPP_VERSION_f` section. This is a 4-byte section used to hold the version variable. The version variable is located in the `main.c` file. To make a new version to download to the MCU, ensure that this value has been updated.
- **ISR Jump Table:** The ISR Jump Table is held in the `.constfISR_JUMP_TABLE_f` section. This section will hold the array of function pointers called from the Boot program's ISR vector table entries.
- **Program Memory:** As mentioned previously, it is necessary to ensure that no code exists from 0x00000000 to 0x00001FFF. This blank area will hold the Boot programs. In the Application project, no linker sections are placed before address 0x00002000. For App 1, no linker sections are placed before address 0x00041000.

  All the other code is placed after address 0x00002200. This is accomplished by moving all remaining sections to address 0x00002200 and beyond.

  The `.text` and `.textf` sections will hold the user program code. They can be filled as you see fit, but ensure that they adhere to the memory boundaries for each Application Region and do not extend past address 0x00040DFE for App 0 or 0x0007FDFE for App 1. You can verify this with the map file output in the build directory.

  Because both the `.const` and `.text` sections are limited to the near area, App 1 must exclusively use the `.textf` and `.constf` sections for program memory. See section 4.2.2 for details on how to do this. For the App 1 linker configuration, the `.text` section is placed at address 0x000000D8, and the `.const` section is placed in the mirrored memory region at 0x00003000.
- **Flash Sections:** The flash sections must exist in the project. It is unimportant where exactly as long as the location is not within addresses 0x00000000 and 0x00001FFF.
- **CRC:** The final segment in the Application image is the placeholder for the CRC. This will be generated post-build in the srec batch files for the entirety of the paired Application and Boot images. In the Application code, place the section and an uninitialized data container in the required position. The section `.constfPROGRAM_CRC_f` is used to hold the data.
- **RAM Sections:** General RAM Sections are allocated starting at address 0x000F4700. This is to avoid allocation in the area reserved for Self RAM and OCD trace.

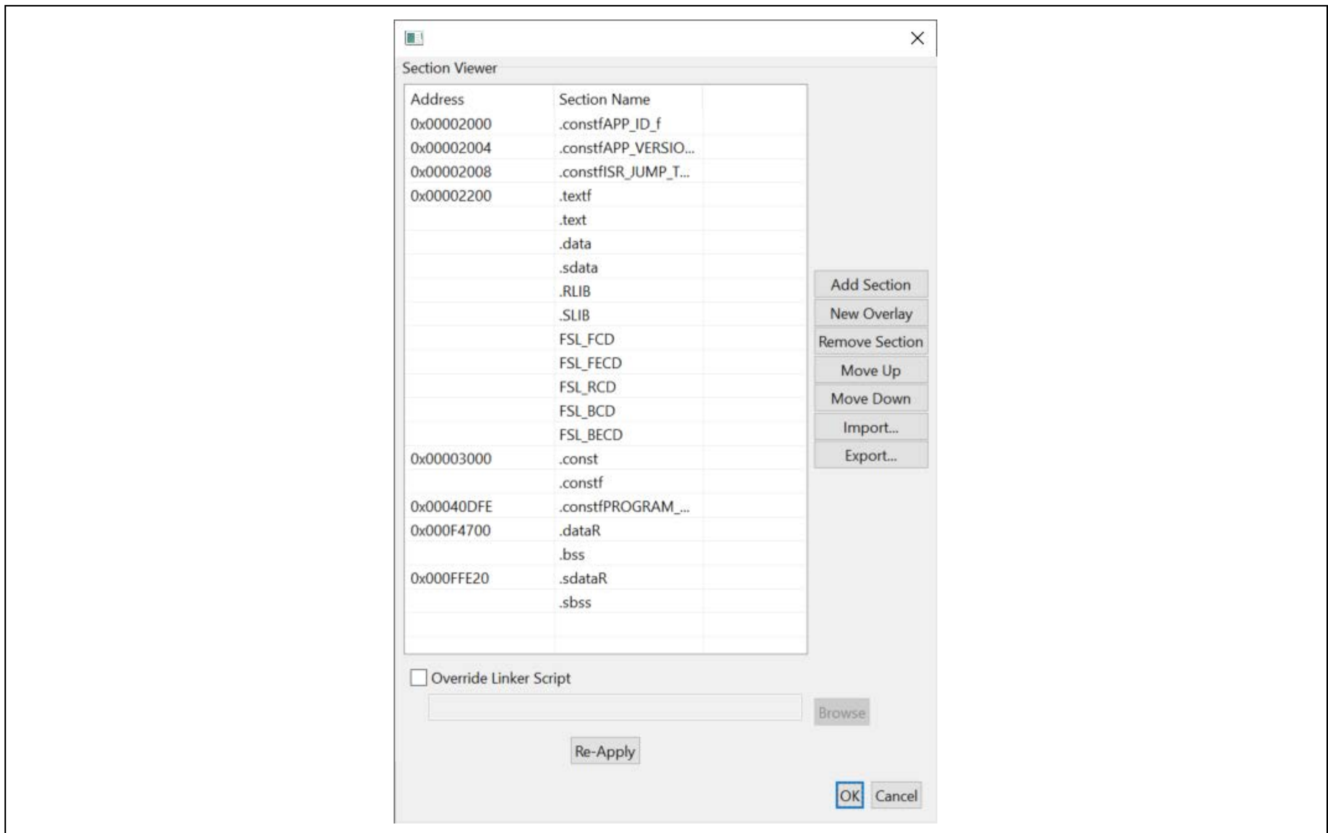

Figure 15. Linker Sections of Application Project

4.2.4 Code Generator Output

The code generator is used to output most of the driver source code. All code generator source files are in the `\src\CodeGenerator` directory. When implementing the bootloader concepts from this example project into a custom application, it may be necessary to modify the generated code to meet the application's requirements.

4.2.4.1 Clock Generation Settings

The example project uses a HOCO setting of 64 (fHOCO = 64MHz; fIH = 32MHz).

4.2.4.2 Serial Settings

The UART is set to a baud of 38,400; this was arbitrarily chosen. This is also the highest speed available in both the RL78 code generator and a standard baud setting in QTs QSerialPort using standard selections.

4.2.5 Flash Libraries

This example project uses the T02 Flash Self-Programming Library. This can be substituted for a different type of the FSL; just verify with the FSL documentation that the type being used can support the functionality required by the application.

4.2.6 Relevant Defined Values

Table 5 describes each of the defines that may need special attention when integrating this example program with a custom application.

Table 5. Application Project Macro Definitions

Header File	Macro Definition	Description	Fixed or Build-Dependent
main.c	USE_LED_0	Determines if LED0 or LED1 will be used when notifying the user that the device is ready to be flashed.	Fixed Value
	version	Stores the current version.	
	program_crc	Stores the CRC value.	
utility.h	UART_PACKET_LENGTH	Size of packets moving from the PC to the MCU.	
	UART_RESPONSE_LENGTH	Size of packets moving from the MCU to the PC.	
	MAX_PACKETS	Number of packets in the image.	
	CRC_ZERO_SEED	Initial seed for the running CRC of the entire image.	
	CRC_OK	When a CRC verifies against any data segment with that segment's CRC appended at the end the CRC will always evaluate to zero.	
	VERSION_ADDRESS	Physical address where the version information is located.	
fsl_interface.h	ADDR_BOOTA_START	Starting address of Boot Region A.	
	ADDR_BOOTB_START	Starting address of Boot Region B.	
utility.h	ADDR_APPn_FLASH_START	Starting address of Application Region *n*.	Build-dependent
	ADDR_VERSION_APPn	Address where the Application *n* revision is located.	
	ADDR_APPn_ISR_START	Starting address of the App *n* ISR jump table.	
	ADDR_APPn_START	Starting address of the App *n* program.	
	ADDR_APPn_END	End address of the App *n* program; excludes reserved memory for OCD region.	
	ACTIVE_APP_ID	Application ID of the main application.	
	ADDR_MAIN_FLASH_START	Start address of the main Application Region.	
	ADDR_ALT_FLASH_START	Start address of the alternate Application Region.	
	ADDR_MAIN_VERSION	Start address of the main Application revision field.	
	ADDR_ALT_VERSION	Start address of the alternate Application revision field.	
	ADDR_MAIN_APP_START	Start address of the executable portion of the main Application.	
	ADDR_MAIN_APP_END	End address +1 of the executable portion of the main Application (includes CRC).	
	ADDR_ALT_APP_START	Start address of the executable portion of the alternate Application.	

Header File	Macro Definition	Description	Fixed or Build-Dependent
	ADDR_ALT_APP_END	End address +1 of the executable portion of the alternate Application (includes CRC).	
	ADDR_MAIN_ISR	Start address of the main Application ISR jump table.	
	ADDR_ALT_ISR	Start address of the alternate Application ISR jump table.	
	ADDR_ISR_UART0_TX	Address of the jump table entry for the UART0 Tx interrupt.	
	ADDR_ISR_UART0_RX	Address of the jump table entry for the UART0 Rx interrupt.	
fsl_interface.h	BLOCK_XXX_YYY	These defines show the start and end blocks of each memory segment. The blocks are the 1KB memory blocks. For example: The define BLOCK_BOOT0_BEGIN gives the first memory block for the Boot 0 memory segment, which is 0. Since the boot 0 memory segment is 0x1000 bytes long (or 4KB) the last block inside boot 0 would be the 3rd block or BLOCK_BOOT0_END.	Fixed Value
	BYTES_PER_WRITE	The size of each write to flash memory by the FSL.	
bsp.h	PORT4_OUTPUT	Used for LED initialization settings for the Fast Prototyping Board; a custom board may differ.	
	PM_43_44_MODE_OUTPUT	Used for LED initialization settings for the Fast Prototyping Board; a custom board may differ.	
	LED0_PIN	Used for Toggling the pin for LED0.	
	LED1_PIN	Used for Toggling the pin for LED1.	

4.2.7 Post-Build Processes

This project uses post-build processes to execute the files for use with `srec_cat.exe`. While the executable can be run from a command prompt, it is possible to wrap the execution into the build process. To do this, you will either need the executable `srec_cat.exe` added to your PC's path or placed directly in the project directory.

The files exist in the `\srec` directory and are executed with post-build steps. To set the post-build steps, go to **Project > Properties > Settings** and select the **Build Steps** tab. You should now see a screen similar to Figure 16. The commands are:

- `srec_cat @..\srec\combine_bootappn.txt`
- `srec_cat @..\srec\crc_genn.txt`
- `srec_cat @..\srec\combine_alln.txt`
- `srec_cat @..\srec\hex_genn.txt`

The '*n*' character can represent either '0' or '1' depending on the selected build. Each command is separated by " & " (including the whitespace). Note that if you are using a Linux system, the ampersand (&) should be replaced with a semicolon (;).

When executing these files, it is recommended to have the Boot application in the workspace as well, if not you will have to modify these files to point to the correct location. For details, see section 4.4.3.

It is also very important that the `crc_gen.txt` file be run before the other two, to ensure that it is placed first. This is because the other files use the `crc.mot` file that it generates.

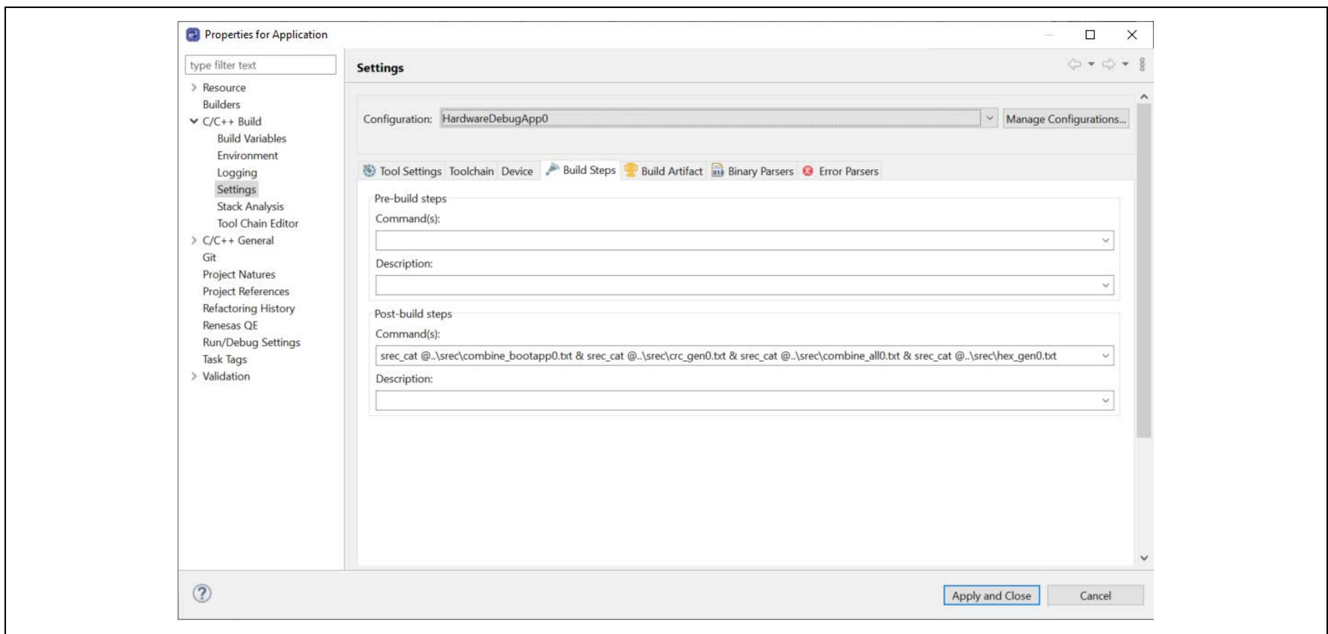

Figure 16. Post-Build Application Project Settings

4.3 PC Application

The PC program is meant as a stand-in for any method of transmitting the data. This example project was built with QT Creator and supplied open-source under the GPL open-source license.

4.3.1 Overview

This application drives the entire download process. Every message sent from the PC will receive a response message. Refer to Table 2 and Table 3 for specifics on what responses are sent. When you click the "Begin Download" button, the process begins.

Every time a response has been received from the MCU, the PC application knows that any process from the last message has been completed and the MCU is ready for a new message. The logic for this can be followed in the function `command_action()` in the file `mainwindow.cpp`.

4.3.2 COM Port

This application uses the QSerialPort API to communicate with the MCU. The USB-to-UART converter translates the USB communication to UART for the MCU.

4.3.3 File Reader

This application uses the QFile API to read the new MCU application file.

4.4 Srec Batch Files

The srec batch files are used to complete multiple post-build processes. Each one executes automatically, as described in section 4.2.7, when you build the Application project. No extra steps are required to run these tasks.

Ensure that you have built the Boot project as the MOT file from that is also accessed. In this example project, it is assumed that both the Boot and Application projects exist in the same directory. If they do not, it will be necessary to modify the `srec_cat` files to point to the `Boot.mot` file.

Each post-build script is described below.

4.4.1 Combining Boot and Application Images

The script `combine_bootappn.txt` is the first script that should be run. `combine_bootappn.txt` takes the output of the Boot *n* and Application *n* builds and combines them into a single image for CRC calculation, using the following steps:

1. Opens the `Boot.mot` file in the `\Boot\HardwareDebugBootn` directory of the Boot project.
2. Opens the `Application.mot` in the `\HardwareDebugAppn` directory of the Application project.
3. All gaps in the data are filled with 0xFF, and the data are cropped to the appropriate addresses for the Boot and Application regions. In the RL78, unused memory locations are read as 0xFF when read for an internal CRC check.
4. The Application data is offset to be contiguous with the Boot data, and the file `BootApp.mot` is output.

4.4.2 CRC Generation

The script `crc_gen.txt` generates the MOT file that contains the CRC, using the following steps:

1. Opens the file `BootApp.mot` file in the `\HardwareDebugAppn` directory.
2. A CRC is performed on the file data, and the CRC is placed in a new file (`crc.mot`). Note that the CRC matches the internal General CRC of the RL78 and is placed in memory in a byte-reversed order. This is so that when the General CRC is calculated of memory, it will evaluate to 0 once it reaches the last byte of the CRC.
3. Since the file data was offset in the previous script, the generated CRC (placed at the end of the input data) is offset in the reverse direction so that it is placed at its correct memory address.
4. When `srec_cat` execution has completed, the file `crc.mot` should exist in the `\HardwareDebugAppn` folder. Inside it will be the CRC in reverse-byte order at address 0x00040DFE for App 0 or 0x0007FDFE for App 1.

4.4.3 Full File Generation

The `compile_alln.txt` file, when executed with `srec_cat`, generates an image to directly flash to the device:

1. After a build, and the execution of `crc_genn.txt`, this file will access the `Boot.mot` file generated from the Boot project and crop the data to the first 0x1000 bytes.
2. It will also access the `Application.mot` file and fill any unused locations with 0xFF. It will then crop the data down to only the relevant Application region.
3. After it has isolated the application image, and merged it with the bootloader project, it will access the `crc.mot` file generated earlier and place the CRC contained within at the end of the application.
4. When `srec_cat` execution has been completed, the file `BootAppnWithCrc.mot` should exist in the `\HardwareDebugAppn` folder. It should now contain the entire bootloader (0x00000000-0x00000FFF), application (either 0x00002000-0x00040DFD or 0x00041000-0x0007FDFD), and the CRC (either 0x00040DFE-0x00040DFF or 0x0007FDFE-0x0007FDFF).

You should now be able to flash the device using the `BootAppnWithCrc.mot` file. This can be done either with e^2 studio itself or with Renesas Flash Programmer.

4.4.4 Download File Generation

The `hex_genn.txt` file, when executed with `srec_cat`, generates the hex file used to download a new application version to the MCU:

1. After a build, and the execution of `compile_alln.txt`, this file will access the `BootAppnWithCrc.mot`.
2. The file data is output as a hex file named `BootAppnWithCrc.hex` in the `\HardwareDebugAppn` folder. This will be used with the PC application.

The resulting `BootAppnWithCrc.hex` file will be used with the PC application. It is a hex file instead of an S-record/MOT file. This is so it is easier for the PC application to parse and download.

5. Running the Sample Program

Before running the sample program, ensure you have the program `srec_cat.exe` added to your path.

5.1 Building the RL78 Projects

5.1.1 Importing the Projects

Open e^2 studio to a workspace of your choosing and click **File > Import**. Then select **General > Existing Projects into Workspace** as shown in Figure 17.

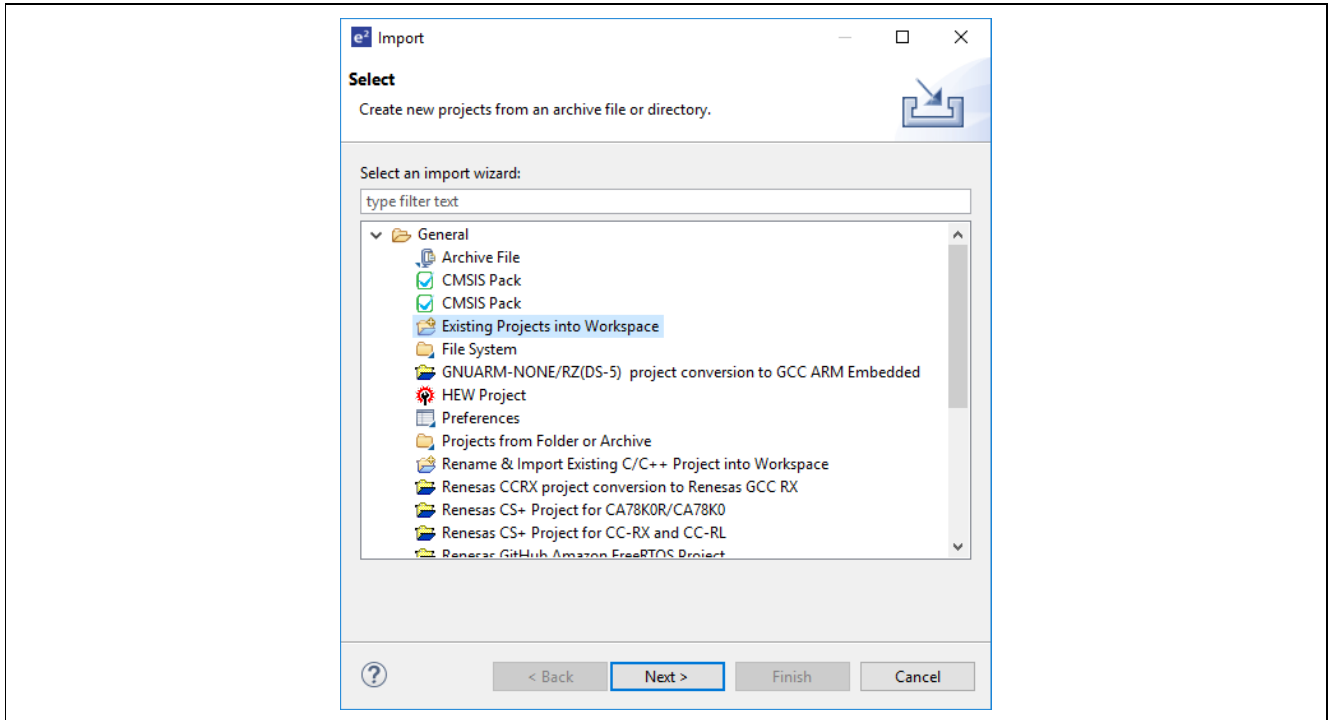

Figure 17. Project Import Screen

Click **Select archive file** and navigate to the RL78 Projects directory in this example project download. Select the `RL78_Dual_Image_Bootloader.zip` file, ensure your screen looks like Figure 18, then click **Finish**. This will import the Boot and Application projects into your workspace.

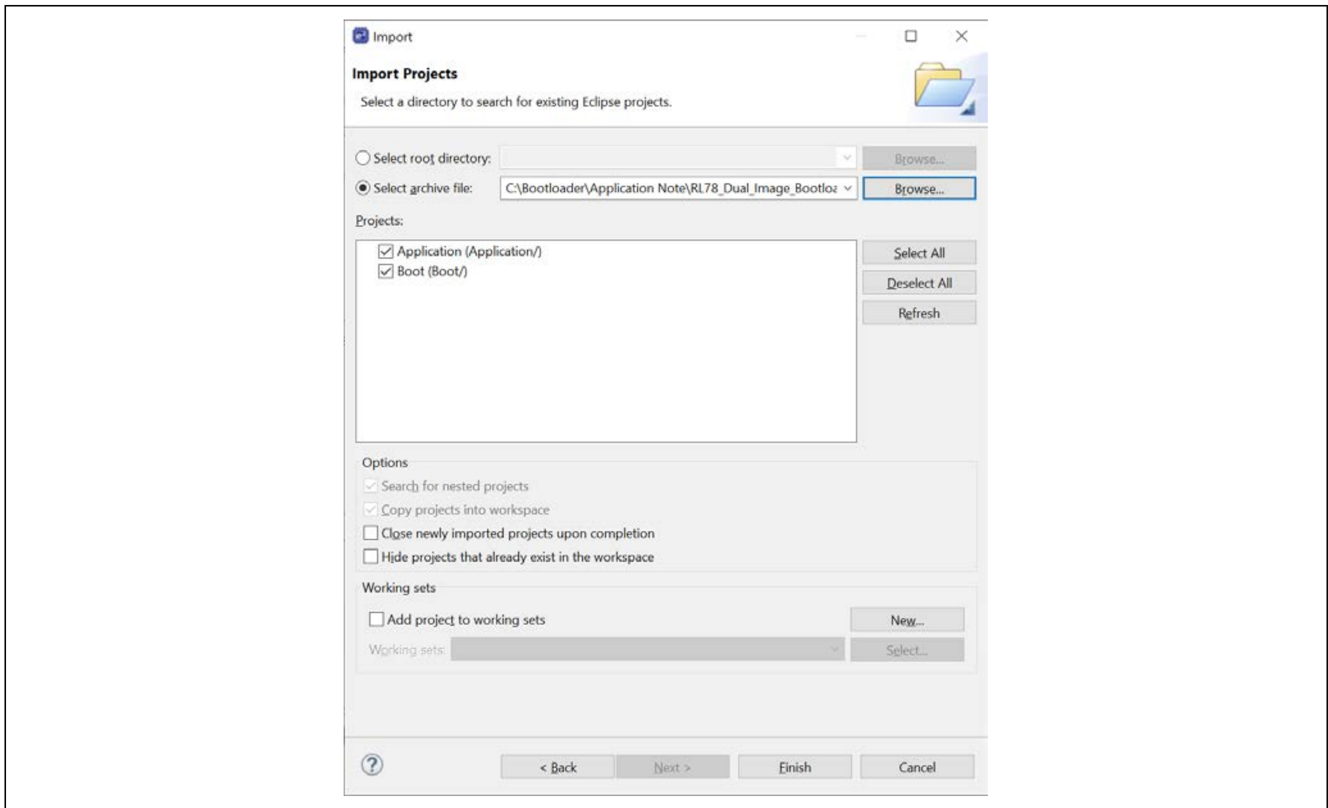

Figure 18. Project Import Details

5.1.2 Selecting the Build Version

Each project has two build versions (0 and 1), so it is necessary to select which version you want to build. To change the current build configuration of the active project, select **Project > Build Configurations > Set Active**, and select the configuration you want to use. For the Boot project, HardwareDebugBoot0 will build Boot 0, while HardwareDebugboot1 will build Boot 1. For the Application project, HardwareDebugApp0 will build App 0, while HardwareDebugApp1 will build App 1.

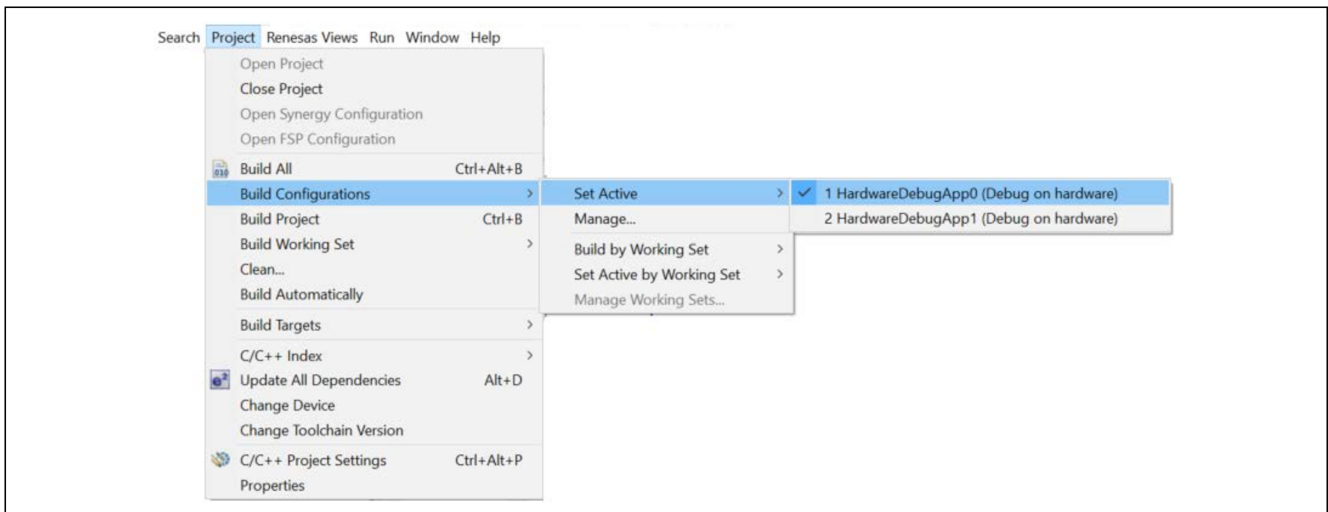

Figure 19. Selecting Build Configuration

5.1.3 Building the Boot Project

Once you have both projects imported, build the Boot project first. This is built first so that the Application project can use its output file to combine the two images into one.

Once the bootloader project is built, open the file `\HardwareDebugBootn\Boot.map` and to verify the following items:

- All your code exists between addresses 0x000000D8 and 0x00000FFF.
- There is nothing in the `.const` section.

5.1.4 Building the Application Projects

The next step is to build two separate images of this project: one that uses the App 0 build configuration, and one that uses the App 1 build configuration.

5.1.4.1 Initial Image (App 0)

The first image will be version 0x01010101. The Application `main.c` file, located in the `\src` directory, has the declaration of the variable `g_version`. To edit `g_version`:

1. In the Boot project, select HardwareDebugBoot0 as the current build configuration and execute the build.
2. In the Application project, select HardwareDebugApp0 as the current build configuration.
3. The Application `main.c` file, located in the `\src` directory, set the value of the variable `g_version` to 0x01010101.

Once this is completed, save the changes to the file and build the application. In the `\HardwareDebugApp0` folder, you should see multiple files created. The file `BootApp0WithCrc.mot` will be used to flash the part with the initial image; this image will contain the application that was just built as well as the bootloader project.

Ensure that jumper EJ1 shown in Figure 1 is open, enabling the onboard debugger. Connect the Fast Prototyping Board to your PC with a USB micro cable. Using Renesas Flash Programmer (RFP), download the `BootApp0WithCrc.mot` file generated to the RL78. Once this is completed, disconnect the USB cable.

5.1.4.2 Downloadable Image (App 1)

The second image to create is version 0x01010101 of App 1.

Repeat the steps in the previous section, but set the active build configurations to HardwareDebugBoot1 and HardwareDebugApp1.

Once this is completed, save the changes and build the project. The resulting `BootApp1WithCrc.hex` file in the `\HardwareDebugApp1` directory will be used in section 5.3.

5.1.4.3 Downloadable Image (App 0)

The next image to create is version 0x01020304 of App 0.

Revert the active build configurations to HardwareDebugBoot0 and HardwareDebugApp0 and repeat the build procedure. Set the value of `g_version` to 0x01020304. The file `BootApp0WithCrc.hex` will be used for the download.

All that is left to do now is set up the hardware to download the new application and run the PC program.

5.2 Setting up the hardware

Connect the hardware so that the new image can be downloaded. To prevent shorts, ensure power is disconnected before making any physical changes to the board.

5.2.1 Run Mode

Short the pins of jumper EJ1 to place the device into Run mode. EJ1 is in the corner of the board, next to the USB port. When in Run mode, the device will not interface with an emulator, and normal programming of the device is impossible.

5.2.2 UART Communication

Connect your USB-to-UART converter to the PMOD pins. Ensure the wires are configured as follows:

- The converter’s RXD line is connected to PMOD1 pin 2.
- The converter’s TXD line is connected to PMOD1 pin 3.
- The converter’s Ground line is connected to PMOD1 pin 5.

5.2.3 Power

Once the above hardware is set up, you can re-connect the target device to the micro-USB cable. This is where the RL78 will draw power from and, since EJ1 is shorted, the on-board debug chip is held in reset.

When power is connected, verify that LED1 is not flashing; this means that the device is indeed in Run mode. Also verify that LED0 has turned on; this notifies you that App 0 is running and ready for a download to occur.

At this point, ensure the USB-to-UART converter is connected to a COM port on your PC.

5.3 Running the PC Application

Run the `BootServer.exe` executable included with this example project. It can be found in the PC `\Application\Executable` directory. You should see a screen like in Figure 20.

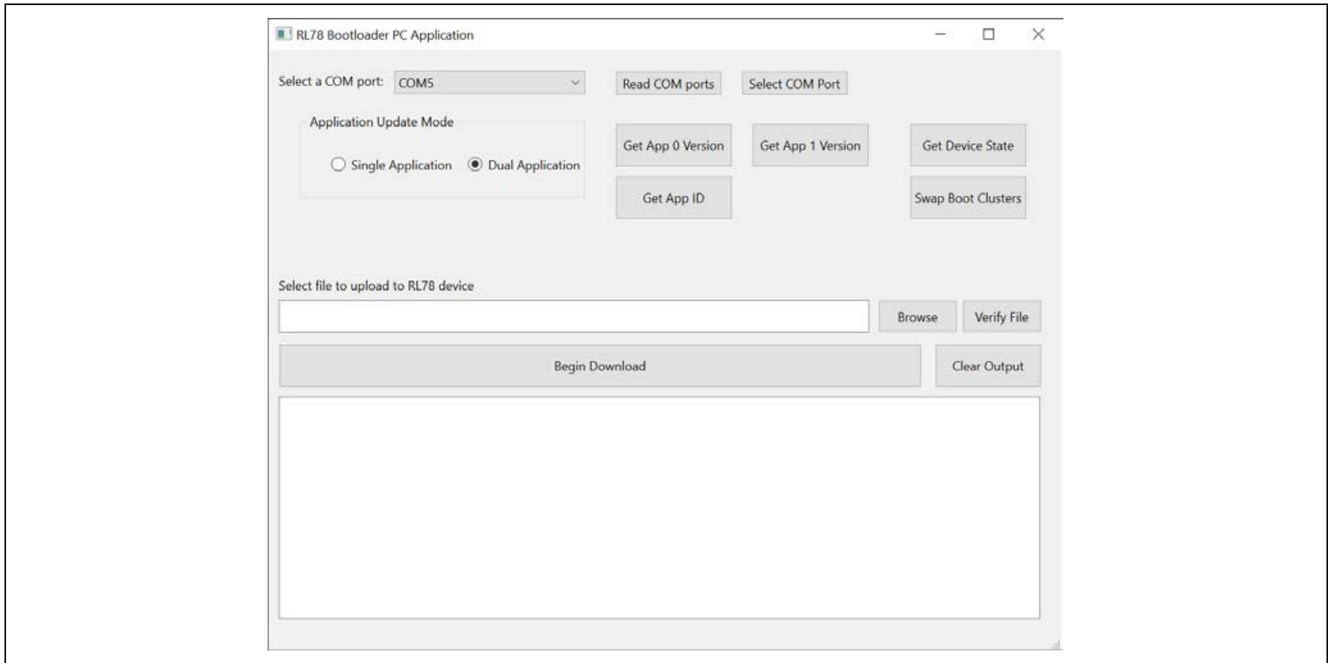

Figure 20. PC Application Interface

5.3.1 Selecting a COM Port

Use the drop-down box to select the COM port connected to your USB-to-UART converter. If you are not sure which one to select, check your PC's device manager.

Once a COM port has been chosen, hit the **Select COM Port** button. Once selected, a message will appear stating "Selected COM port: COMx". If this button is not clicked, communication with the board is impossible.

5.3.2 Selecting the File to Download

Using the **Browse** button, navigate to the `BootApp1WithCrc.hex` file that was created in section 5.1.4.2 (Downloadable Image (App 1)). You can now use the **Verify File** button to verify that the PC application can calculate and match the CRC generated with the srec tools during the build.

Verify that your screen now looks like Figure 21.

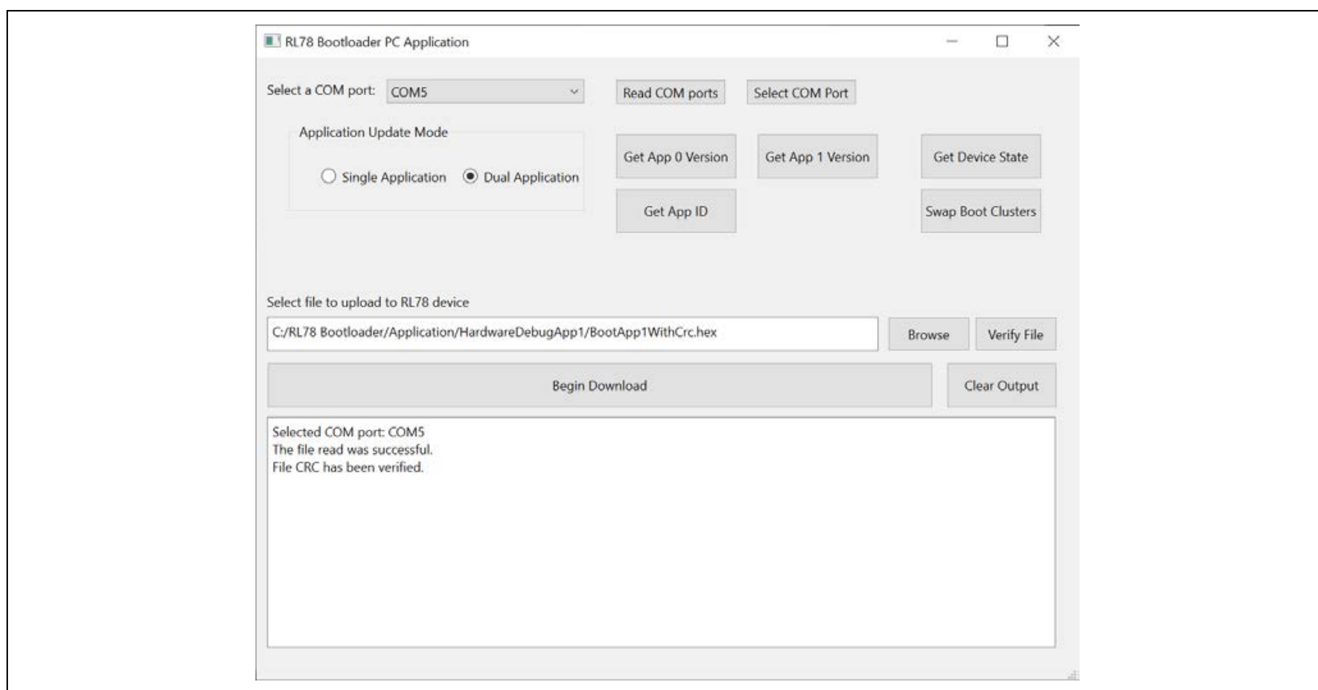

Figure 21. Download File Verified

5.3.3 Download

Now click the **Begin Download** button. This will kickstart the download process and, assuming everything has been set up correctly, the download will begin.

You can view the download progress with the messages output to the message output box, including the packets being transmitted and the ACKs received. The download process may take a few minutes to complete and should end on packet 4087.

When the download is complete, you should see a message "Download Success!". To finally verify that the download worked, the device will automatically reset into the new application (App 1), turning on LED1. If you see LED1, the new image has been downloaded, verified, and is now the active application.

You can repeat the download process with the App 0 hex file generated previously.

5.3.4 Additional Information

5.3.4.1 Application Region Destination for Downloads

When downloading a new image, the PC application will always use the alternate Application area as the destination. This means that if App 0 is currently running, the PC application will expect an App 1 image. Likewise, if App 1 is currently running, the PC application will expect an App 0 image. The PC application checks the App ID field of the current Application and that of the selected file to download. If the file does not have the expected App ID, then the PC application will output an error message.

5.3.4.2 Regression Protection

The download process includes regression protection, so each new Application image will need a higher-valued revision than the image currently in the destination region. An exception is when the revision reads 0xFFFFFFFF, such as when the destination flash is empty.

5.3.4.3 Additional Interface Controls

The PC application interface includes controls to do each of the following operations for the dual-application solution:

- Query the Application ID of the currently running Application.
- Query the Application Revision of App 0.
- Query the Application Revision of App 1.
- Get the current state of the boot swap bit in the MCU.
- Execute a boot swap and reset the MCU.

6. Website and Support

Renesas Electronics Website	www.renesas.com
Inquiries	www.renesas.com/contact

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jul.20.23	—	Initial release

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit 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. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time 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 time 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 time 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 time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance 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 the 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.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is 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. Additionally, 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.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of 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.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit: www.renesas.com/contact/.

© 2023 Renesas Electronics Corporation. All rights reserved.