Renesas RA Family

Implementing Production Programming Tools for RA Cortex-M33 with Device Lifecycle Management

Introduction

Renesas RA Family MCUs implement boot mode, which provides access to built-in firmware that allows the system configuration to be interrogated and updated. Boot mode is entered when the MCU is reset with the MD pin on the device pulled low. When the MCU is in boot mode, user code in flash will not be active. An MCU in boot mode enumerates as a COM port through either a serial port or a USB virtual COM port. Tools running on an external system, such as a Windows PC, can then communicate with the MCU over this interface.

During software development or small prototype production runs, standard Renesas tools, such as the Renesas Flash Programmer (RFP), may be used with boot mode. In such cases, the system developer may not need to be aware of the full details of boot mode and how it works.

However, for companies who provide production programming tools—or users who plan to create their own tools for production purposes—such tools may well be required to communicate with boot mode, particularly for the RA4 and RA6 MCU Family devices based on Cortex-M33 and that implement Device Lifecycle Management (DLM) capabilities.

The full specification of the boot mode interface for these RA Family MCUs is detailed in Renesas application note R01AN5562, which is available for download from the Renesas website. This application note expands on the boot mode interface specification to provide more practical examples of how to interface with boot mode, from both the hardware and software perspectives. Demonstration code, written in Python, is provided to illustrate how boot mode access can be accomplished.

Note: We do not guarantee any operations not described in this document.

Supported MCU Groups

At the time of the release, the supported MCU groups are:

- RA4M2 Group
- RA4M3 Group
- RA6M4 Group
- RA6M5 Group
- RA4E1 Group
- RA6E1 Group
- RA6T2 Group

Required Resources

Development tools and software

- Python v3.10 or later (https://www.python.org/downloads/)
- pySerial v3.5 (https://pyserial.readthedocs.io/en/latest/pyserial.html#installation)
- Renesas Flash Programmer v3.11.01 or later (https://www.renesas.com/rfp)
Hardware

- EK-RA6M4, Evaluation Kit for RA6M4 MCU Group (http://www.renesas.com/ra/ek-ra6m4)
  - For demonstration purposes, this application note makes use of the RA6M4 MCU and the EK-RA6M4 evaluation board. However, the available functionality will be the same on the other supported MCU groups except when specifically noted.
- Workstation running Windows® 10
  - Demonstration code should also work on other platforms that support Python and pySerial, but this has not been tested.
- One USB device cable (type-A male to micro-B male) or
- One USB to TTL Serial 3.3-V UART Converter with four pieces of male to female jumper wire.

Prerequisites and Intended Audience

The intended audience is engineers creating production programming tools to use with Renesas RA Family MCUs. Before using this application note and associated demonstration code, users should acquire the following documentation for reference:

- Application note “Standard Boot Firmware for the RA family MCUs Based on Arm® Cortex®-M33” (R01AN5562)
- The MCU User’s Manual: Hardware (for the MCU that is under consideration)
- Application note “Device Lifecycle Management Key Injection for the RA Family MCUs” (R11AN0469)

These documents are available on the Renesas website and are referenced in this application project.

Contents

1. Production Programming Concepts ................................................................. 4
   1.1 Background ................................................................................................. 4
   1.2 Typical Production Programming Flow ....................................................... 4
   1.3 Flash Programming ..................................................................................... 5
   1.4 Device Lifecycle Management ...................................................................... 5
   1.5 Secure / Non-secure /Non-secure Callable Regions ...................................... 7
     1.5.1 IDAU Registers and Non-TrustZone-using Software .............................. 8
2. MCU Hardware Setup for Boot Mode Use .................................................... 8
   2.1 Boot Mode Communication Interfaces Overview ......................................... 8
   2.2 Power ......................................................................................................... 8
   2.3 Clock .......................................................................................................... 9
   2.4 MCU System Mode Control Signals ............................................................. 9
   2.5 Using the 2-wire Serial Communication ..................................................... 9
   2.6 Using the Universal Serial Bus (USB) Communication .............................. 9
   2.7 Using Serial Wire Debug Interface (SWD) ................................................ 10
3. Connecting to Boot Mode .............................................................................. 11
   3.1 Boot mode operational phases .................................................................... 11
   3.2 Initialization Phase .................................................................................... 12
     3.2.1 Serial Settings ..................................................................................... 13
     3.2.2 USB Settings ...................................................................................... 13
   3.3 Communication Setting Phase ..................................................................... 13
4. Boot Mode Commands ................................................................. 15
  4.1 Command Acceptable Phase .................................................. 15
  4.1.1 Command Packet Format .................................................. 15
  4.1.2 Data Packet ................................................................. 15
  4.1.3 Summary of Boot Mode Commands .................................. 16
  4.1.4 Boot Mode Firmware Operation ....................................... 17

5. Typical Use Cases of Boot Mode Commands .................................. 17
  5.1 Overview of Use cases .......................................................... 18
  5.2 Inquiry Command .............................................................. 18
  5.3 Initialize the MCU .............................................................. 18
    5.3.1 Initialize Command ...................................................... 19
    5.3.2 Check Whether Initialize Command is Disabled .......... 20
    5.3.3 Disable Initialize Command ....................................... 22
  5.4 TrustZone Boundary Region Setup ....................................... 24
    5.4.1 Operational Flow ......................................................... 24
    5.4.2 Acquire the Boundary Information from an Application 24
    5.4.3 TrustZone Boundary Request Command ...................... 25
    5.4.4 TrustZone Boundary Setting Command ....................... 27
  5.5 DLM Key Handling ............................................................ 28
    5.5.1 Inject DLM Keys .......................................................... 29
    5.5.2 Verify DLM Keys ......................................................... 33
  5.6 DLM State Handling ........................................................... 34
    5.6.1 DLM State Request ...................................................... 34
    5.6.2 DLM State Transition .................................................. 36

6. Running the Python Example Code ............................................. 38
  6.1 Set up the Python Environment ............................................ 39
  6.2 Setting Up the Hardware ..................................................... 39
  6.3 Running the First Demo Code ............................................... 41
  6.4 Running the Second Demonstration Code ............................. 45
    6.4.1 Establishing the Connection (USB) .............................. 47
    6.4.2 Checking Current DLM State and Configuring TrustZone Partition Boundaries .... 48
    6.4.3 Injecting DLM Keys ...................................................... 49
    6.4.4 Configuring Final DLM State ....................................... 51
  6.5 Testing Authenticated DLM Transitions ............................... 51
  6.6 Disabling the Initialize Command ....................................... 51

7. References ................................................................................. 53

8. Website and Support .............................................................. 53

Revision History ............................................................................... 54
1. Production Programming Concepts

This section introduces some of the concepts behind the operations required to perform production programming of RA4 and RA6 MCU Family devices based on Cortex-M33 and that implement Device Lifecycle Management (DLM) capabilities.

1.1 Background

With many Arm Cortex-M based MCUs from a variety of silicon manufacturers, it is often possible for most production programming operations—in particular programming of an application image into flash memory—to be carried out over a Serial Wire Debug (SWD) connection to the target MCU, as SWD is also used for debugging purposes during the software development process.

However, with the RA4 and RA6 MCU Family devices based on Cortex-M33 with DLM capabilities, just having access to an SWD connection for production programming is not sufficient. It is also necessary to carry out various device configuration operations that can only be done through the MCU’s boot mode, which cannot be accessed through SWD.

Boot mode is entered when the MCU is reset with the MD pin on the device pulled low. In boot mode, rather than any user code in flash being executed, a terminal-like interface is made available through either a serial port (often referred to in Renesas documentation as SCI/UART) or a USB virtual COM port. Tools running on an external system, such as a Windows PC, can then communicate with the MCU over this interface.

Boot mode is also available on other RA Family MCUs based on Cortex-M23 and Cortex-M4 CPUs, as well as on Cortex-M33 based MCUs that do not implement DLM capabilities. However, on such MCUs, the functionality provided by boot mode is somewhat different and production programming can generally be carried out over SWD without requiring any access to boot mode (although programming can also be done through boot mode).

1.2 Typical Production Programming Flow

RA4 and RA6 MCU Family devices based on Cortex-M33 with DLM capabilities are delivered from the factory in the Chip Manufacturing (CM) state. A typical production programming flow includes the following steps:

1. Establish the necessary hardware connections to enable the use of boot mode.
2. Reset the MCU into boot mode and establish communication from the host to the MCU over either SCI/UART or USB. Then, check the current DLM state. If the MCU is in the CM state, use the DLM state transition command to transition the DLM state from CM to SSD. This step is explained in more detail later in this document.
3. If the MCU is not in the CM state—for example, if this is an evaluation board that has already been used for other purposes—an “Initialize” command may be issued if the device is in the NSECSD or DPL state. At the end of the “Initialize” command, the MCU is changed to the SSD state with the Code Flash, Data Flash, and Flash Option settings erased. This step is explained in more detail later in this document.
4. Reset to normal operation and program the MCU’s flash memory over SWD.
   - This can also be done through boot mode but would generally be much slower, especially if SCI/UART communication is used.
   - This step is not demonstrated in this application note.
5. Reset MCU again into boot mode.
6. Set up the required “security related options” using boot mode operations. Details on how to perform these steps are explained in section 4. Items a), c) and d) are demonstrated in the example code. Demonstration for b) will be added at a later release - although it is very similar to c):
   a) Configure TrustZone partition boundaries.
   b) Inject User Keys. (Note: User Key injection is not available on RA4E1 and RA6E1.)
   c) Inject DLM Keys.
   d) Change to final DLM state.
1.3 Flash Programming

Generally, RA MCUs provide three types of flash memory, with slight differences in the way they are erased and programmed:

- Code Flash memory
- Data Flash memory
- Option Flash memory

Although it is possible to program these flash memory areas through the boot mode interface, in many production programming tools it may be preferable to carry out such programming over an SWD connection.

Note: If programming flash through the boot mode interface, the DLM state of the MCU must first be changed from CM (factory default) to SSD.

For a particular MCU, details such as memory sizes and the mechanisms available to program each area are detailed in the Flash Memory chapter of the corresponding "User's Manual: Hardware".

Example flash source code in Keil MDK flash driver format is available in our Device Family Packs (DFP) on the Renesas RA Flexible Software Package (FSP) GitHub.

At the time of writing, the latest version is available as part of FSP 4.4.0 at:

- [https://github.com/renesas/fsp/releases/download/v4.4.0/MDK_Device_Packs_v4.4.0.zip](https://github.com/renesas/fsp/releases/download/v4.4.0/MDK_Device_Packs_v4.4.0.zip)

Check the FSP GitHub for newer versions.

The Arm/Keil document for the code layout and functions of their flash driver format is available at:


Additional flash programming code is available for reference within the FSP drivers for each MCU group.

1.4 Device Lifecycle Management

Most RA family MCUs based on Arm Cortex-M33 CPUs adopt the concept of a device life cycle and maintain the life cycle state inside the device. The DLM state is used to restrict access to the MCU's internal resources through SWD/JTAG debugger and boot mode interfaces as the device lifecycle states progress. The DLM state is only configurable through boot mode over an SCI/UART or USB connection.

The set of boot mode commands that are possible are controlled by the current lifecycle state. Changing lifecycle state is also only possible using a boot mode command. Note that a production programming tool should always move MCU into at least SSD (not leave it in the CM state).

Table 1 describes the DLM states that may be involved in the production programming stage.
Table 1. TrustZone-Enabled RA Family MCU Group Device Lifecycle States

<table>
<thead>
<tr>
<th>Lifecycle state</th>
<th>Definition and State Features</th>
<th>Debug Level</th>
<th>Boot mode access</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>• “Chip Manufacturing”&lt;br&gt;  This is the state when the developer receives the device.</td>
<td>DBG2</td>
<td>• Available&lt;br&gt;  • No access to code/data flash</td>
</tr>
<tr>
<td>SSD</td>
<td>• “Secure Software Development”&lt;br&gt;  The secure part of the application is being developed.</td>
<td>DBG2</td>
<td>• Available&lt;br&gt;  • Can program/erase/read all code/data flash</td>
</tr>
<tr>
<td>NSECSD</td>
<td>• “Non-SECure Software Development”&lt;br&gt;  The non-secure part of the application is being developed.</td>
<td>DBG1</td>
<td>• Available&lt;br&gt;  • Can program/erase/read non-secure code/data flash</td>
</tr>
<tr>
<td>DPL</td>
<td>• “DePloyed”&lt;br&gt;  The device is in-field.</td>
<td>DBG0</td>
<td>• Available&lt;br&gt;  • No access to code/data flash</td>
</tr>
<tr>
<td>LCK_DBG</td>
<td>• “LoCKed DeBuG”&lt;br&gt;  Device is in-field and the debug interface is permanently disabled.</td>
<td>DBG0</td>
<td>• Available&lt;br&gt;  • No access to code/data flash</td>
</tr>
<tr>
<td>LCK_BOOT</td>
<td>• “LoCKed BOOT interface”&lt;br&gt;  Device is in-field and the boot interface and the boot mode interface are permanently disabled.</td>
<td>DBG0</td>
<td>• Not available</td>
</tr>
</tbody>
</table>

The three debug levels are:
- DBG2: The debugger connection is allowed, with no restriction on access to memories and peripherals.
- DBG1: The debugger connection is allowed, with access to only non-secure memory regions and peripherals.
- DBG0: The debugger connection is not allowed.

Figure 1 describes the possible transitions between DLM states. The states that are related with production programming are included in the red dotted box. The required DLM state may require multiple command invocations to achieve. Authenticated DLM state changes are generally not required for production programming, as they are used during software development or in-field debugging. However, production programming tools need to be able to inject the keys required to allow authenticated DLM state changes.

Figure 1. Device Lifecycle Management

For production programming, the tool must move a device from CM to SSD for MCUs that are delivered from the factory. The tool may alternatively need to transit the DLM state back to SSD using an “Initialize” command for MCUs that have been used in the past. At the end of the sequence, the tool may also need to
support locking down of the device—to prevent user’s proprietary code and data being read back—by moving the DLM state into DPL, LOCK_DBG, or LOCK_BOOT. Boot mode also provides a command to disable the “Initialize” command, preventing future erasing of flash and resetting of DLM state.

Authenticated transitions are possible using DLM keys. These user-defined keys are injected during specific device lifecycle states to allow authenticated regression back to that state.

The primary keys that most applications will use are:

- **SECDBG_KEY**
  - The Secure Debug Key can be injected when the MCU is in the SSD state. It can be used when the MCU is in the NSECSD state to regress back to the SSD state without erasing flash memory. This key is used in transition 1 of Figure 1.

- **NONSECDBG_KEY**
  - The Non-secure Debug Key can be injected when the MCU is in the SSD or NSECSD state. It can be used when the MCU is in the DPL state to regress back to the NSECSD state without erasing flash memory. Authenticated regression from DPL back to SSD must be done in two steps. This key is used in transition 2 of Figure 1.

Note that a DLM key injected during production allows a user to change the DLM state post-production if and only if they have access to the original key.

### 1.5 Secure / Non-secure /Non-secure Callable Regions

Arm TrustZone technology is a core security technology developed by Arm and included as part of the v8-M architecture. It is typically implemented on a wide range of Cortex-M33 based devices, including Renesas’ RA4 and RA6 Family MCUs. The key point about TrustZone technology is that it provides and enforces a partition between trusted and non-trusted portions of the system, which provides the designer of a product with a building block towards producing a more secure MCU application. At a basic level, the way this partitioning is implemented is by use of memory regions, which effectively covers code, data, and peripherals within the overall memory map.

First, we have Secure memory regions. These are the trusted regions covering overall system boot as well as trusted or protected IP such as key storage and data decryption.

Secondly, we have Non-Secure memory regions, which are used for our normal application code and data, which do not require direct access to the trusted data. The important point here is that non-secure operations are only allowed to access Non-Secure regions, thereby preventing unapproved access to trusted information or operations.

Finally, we have Non-Secure Callable regions, which are used to provide a gateway between the secure and non-secure worlds.

RA Cortex-M33 based MCUs with DLM implement a piece of logic called an Implementation Defined Arbitration Unit (IDAU) to partition the flash and RAM into Secure and Non-Secure regions.

![Figure 2. TrustZone Configurations Example using RA6M4](image)

Figure 2. TrustZone Configurations Example using RA6M4
The IDAU registers that control the sizes of these regions are only modifiable through boot mode. Therefore, as part of the production programming sequence, appropriate values for these registers need to be configured by the tools.

When a secure application is built, Renesas tools generate a file (.rpd) that contains details of the required split between Secure and Non-Secure memory. The .rpd file can be used by the production programming tool to configure the appropriate values into the IDAU registers through boot mode.

**IDAU Registers and Non-TrustZone-using Software**

**1.5.1 IDAU Registers and Non-TrustZone-using Software**

Renesas tools also generate the .rpd file for applications that do not use TrustZone technology. In most cases, the IDAU registers could theoretically be left set to the default (maximal) values that will be set by running an Initialize command.

However, in some cases, this is not appropriate. Some applications need some areas of memory set to be Non-Secure. For example, applications that make use of the Ethernet peripheral require some Non-Secure RAM. Configuring the IDAU registers is necessary for such use cases.

In general, to ensure correct application execution, we recommend always setting up the IDAU registers as part of the production programming process.

### 2. MCU Hardware Setup for Boot Mode Use

This section describes the hardware requirements for setting up the production environment, including the power, clock, communication interface connections, and the signals that control the MCU operation mode.

#### 2.1 Boot Mode Communication Interfaces Overview

Boot mode is entered when the MCU is reset with the MD pin on the device pulled low. Boot mode can then be accessed using one of the following communication methods:

- 2-wire serial communication (often referred to in Renesas documentation as SCI/UART)
- Universal Serial Bus (USB) communication (over a virtual COM port)
- Multiplex Serial Wire Debug (SWD) Interface and SCI/UART Interface on the SWD debug header

Communication with boot mode is not carried out directly over SWD. Instead, Renesas has defined a specification for reusing certain pins from an SWD debug header as UART pins. This enables production programming tools to communication over a single physical connector using either SWD (for programming flash) or UART (for communication with boot mode).

The hardware requirements of these communication methods are described in the following sections. These sections use the RA6M4 MCU group as an example. For production support, confirm details for the specific MCU being used in the Hardware User's Manual Section “Pin Functions”.

#### 2.2 Power

The production hardware setup needs to provide proper power and ground to the MCU. The example guidelines shown in Table 2 and Table 3 are based on the RA6M4 MCU.

**Table 2. RA6M4 MCU Operating Voltage Range**

| Operating voltage | VCC = 2.7 to 3.6 V |

**Table 3. RA6M4 MCU Power Signals**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Signal</th>
<th>IO</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Power supply</strong></td>
<td>VCC</td>
<td>Input</td>
<td>Power supply pin. Connect VCC to the system power supply. Connect this pin to VSS by a 0.1-μF capacitor. The capacitor should be placed close to the pin.</td>
</tr>
<tr>
<td></td>
<td>VCL/VCL0</td>
<td>I/O</td>
<td>Connect this pin to the VSS pin by the smoothing capacitor used to stabilize the internal power supply. Place the capacitor close to the pin.</td>
</tr>
<tr>
<td></td>
<td>VSS</td>
<td>Input</td>
<td>Ground pin. Connect it to the system power supply (0 V).</td>
</tr>
</tbody>
</table>
2.3 Clock

The clock signal is also mandatory for the MCU and the boot firmware to function. To use the boot mode firmware, there are specific requirements on the main oscillator frequency. Table 4 shows the requirement for the RA6M4 MCU.

Table 4. Clock Source for Boot Mode Operation

<table>
<thead>
<tr>
<th>Clock Source</th>
<th>RA4M2/3, RA6M4/M5, RA4E1, RA6E1:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Main Oscillator Frequency of 8, 10, 16, 20, 24 MHz can be used by the boot mode firmware. Otherwise, HOCO will be used.</td>
</tr>
<tr>
<td></td>
<td>RA6T2:</td>
</tr>
<tr>
<td></td>
<td>HOCO 20 MHz (Does not use Main-OSC).</td>
</tr>
</tbody>
</table>

Table 5. Clock Signals

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Signal</th>
<th>IO</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>XTAL</td>
<td>Output</td>
<td>Pins for a crystal resonator. An external clock signal can be input through the EXTAL pin.</td>
</tr>
<tr>
<td></td>
<td>EXTAL</td>
<td>Input</td>
<td></td>
</tr>
</tbody>
</table>

* When performing USB communication with HOCO, Sub-OSC must be oscillating stably.

2.4 MCU System Mode Control Signals

As mentioned in section 1.1, boot mode is entered when the MCU is reset with the MD pin on the device pulled low. Table 6 describes some more details on these two signals.

Table 6. General Signals for Accessing the Boot Mode

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Signal</th>
<th>IO</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating mode control</td>
<td>MD</td>
<td>Input</td>
<td>Pin for setting the operating mode. The signal level on MD must not be changed during operation mode transition on release from the reset state. For the MCU groups covered in this application note, the MD pin is P201.</td>
</tr>
<tr>
<td>MCU Reset control</td>
<td>RES</td>
<td>Input</td>
<td>Reset signal input pin. The MCU enters the reset state when the RES signal goes low.</td>
</tr>
</tbody>
</table>

2.5 Using the 2-wire Serial Communication

The Serial Communication Interface (SCI) hardware block used for UART communication has several channels. For boot mode use, channel 9 is used to enumerate a COM port. Table 7 provides more details on the UART signals.

Table 7. UART Boot Mode Pins

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Signal</th>
<th>IO</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCI (channel 9)</td>
<td>RXD9</td>
<td>Input</td>
<td>RA4M2/3, RA6M4/5, RA4E1, RA6E1: P110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RA6T2: PA15</td>
</tr>
<tr>
<td></td>
<td>TXD9</td>
<td>Output</td>
<td>RA4M2/3, RA6M4/5, RA4E1, RA6E1: P109</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RA6T2: PB03</td>
</tr>
</tbody>
</table>

2.6 Using the Universal Serial Bus (USB) Communication

USB communication with boot mode can be used by all the supported MCU groups except RA6T2. Table 8 describes the details on the USB signals. The production programming fixture development team can refer to the Renesas evaluation board schematic to provide the signal conditioning circuit for the USB connections.
Table 8. USB Interface and Configurations

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Signal</th>
<th>IO</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB Full Speed</td>
<td>VCC_USB</td>
<td>Input</td>
<td>USB Full-speed power supply pin. Connect this pin to VCC. Connect this pin to VSS_USB through a 0.1 uF capacitor placed close to the VCC_USB pin.</td>
</tr>
<tr>
<td></td>
<td>VSS_USB</td>
<td>Input</td>
<td>USB Full-speed ground pin. Connect this pin to VSS.</td>
</tr>
<tr>
<td></td>
<td>USB_DP</td>
<td>I/O</td>
<td>D+ pin of the USB on-chip transceiver. Connect this pin to the D+ pin of the USB bus.</td>
</tr>
<tr>
<td></td>
<td>USB_DM</td>
<td>I/O</td>
<td>D- pin of the USB on-chip transceiver. Connect this pin to the D- pin of the USB bus.</td>
</tr>
<tr>
<td></td>
<td>USB_VBUS</td>
<td>Input</td>
<td>USB cable connection monitor pin. Connect this pin to VBUS of the USB bus. Designers should scale down the 5V VBUS input to the MCU’s operating VCC voltage range with ESD projection. The VBUS pin status (connected or disconnected) can be detected when the USB module is operating as a function controller.</td>
</tr>
<tr>
<td></td>
<td>USB_VBUSEN</td>
<td>Output</td>
<td>VBUS (5V) supply enable signal for external power supply chip.</td>
</tr>
</tbody>
</table>

2.7 Using Serial Wire Debug Interface (SWD)

For performance reasons, a production programming tool may prefer to program flash memory over an SWD connection, rather than using the boot mode interface. However, a boot mode interface is still required for other operations, such as changing the DLM state.

A dedicated boot mode interface can be implemented by providing a separate serial interface to the target MCU. However, a much cleaner user experience can be achieved if the same header on a board can be used for both SWD and boot mode access. To support this, Renesas has standardized on an extended configuration of the SWD header. This is achieved by reusing pins on the standard debug connector as shown in Figure 3. Refer to the RA6 MCU Family Quick Design Guide (R01AN5775) section “Emulator Support” for more details on the specification of this interface.

![Figure 3. Access the Boot Mode through the Multi-emulator Interface Header](image)

The setup shown in Figure 3 allows the production programming tools to control whether the target MCU will be accessed through serial communications or SWD, based on whether it pulls MD low or not when asserting reset. If boot mode access is in use, pins on the SWD header can be used as SCI/UART RXD/TXD pins by the production programming tool hardware.
This configuration of the SWD header is also commonly used for debugging purposes, where boot mode operations are also required (for example, to inject TrustZone partition boundaries).

3. Connecting to Boot Mode

This section explains the procedure to establish communications with the boot mode.

3.1 Boot mode operational phases

When the MCU is reset with MD pulled low, the MCU enters a sequence of operational phases, as shown in Figure 4:

1. Initialization phase.
2. Communication setting phase.
3. Command acceptable phase

![Figure 4. Boot Mode Operational Phases](image)

*From the Command acceptable phase, the host and microcontroller send data in turn unless otherwise noted. The host executes data transmission after data reception from the microcontroller.*

Figure 5 shows this in more detail, in relation to the MCU DLM state and MD pin status.

The rest of this section examines how production programming tools can make the connection to boot mode, moving the MCU through the Initialization and Communication setting phases, and then entering the Command acceptance phase.
3.2 Initialization Phase

Production programming tools do not need to carry out any actions during the Initialization phase. Once boot mode is entered after release of the reset pin with MD in low state, the boot mode firmware initializes the required hardware modules (including UART or USB) and then transits to the Communication setting phase.

*1: Boot firmware adopts IP(SCI or USBFS) that received 0x00 consecutively three times earlier as the communication method
*2: RA6T2 support SCI only.
3.2.1 Serial Settings
When using serial communication to boot mode, the following default settings are in use:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit rate</td>
<td>9600 bps (minimum, until the baud rate setting command)</td>
</tr>
<tr>
<td>Data length</td>
<td>8 bits (LSB first)</td>
</tr>
<tr>
<td>Parity bit</td>
<td>None</td>
</tr>
<tr>
<td>Stop bit</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

Communication is performed at 9600 bps until the baud rate setting command is invoked (in the Command acceptable phase). After the baud rate setting command has completely successfully, communication is then performed at the desired baud rate. The maximum bit rate that can be communicated with the device is returned by “RMB” of the “signature request” command.

- If the communication cable is disconnected during communication, subsequent operations are not guaranteed.
- Communication with the boot firmware through UART is demonstrated in this application note. However, the “Signature Request” and “Baud rate setting” commands are not demonstrated. Refer to application note R01AN5562 for more details of these commands.

3.2.2 USB Settings
When using USB communication to boot mode, the following settings are in use:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer rate</td>
<td>12 Mbps (USB 2.0 Full Speed)</td>
</tr>
<tr>
<td>Device class</td>
<td>Communication Device Class (CDC)</td>
</tr>
<tr>
<td></td>
<td>SubClass: Abstract Control Mode (ACM)</td>
</tr>
<tr>
<td></td>
<td>Protocol: Common AT commands</td>
</tr>
<tr>
<td>Vender ID</td>
<td>0x045B</td>
</tr>
<tr>
<td>Product ID</td>
<td>0x0261</td>
</tr>
<tr>
<td>Transfer mode</td>
<td>Control (in/out)</td>
</tr>
<tr>
<td></td>
<td>Bulk (in, out)</td>
</tr>
<tr>
<td></td>
<td>Interrupt (in)</td>
</tr>
</tbody>
</table>

3.3 Communication Setting Phase
Once the system enters the Communication setting phase, boot mode polls the selected communication interface looking for a sequence of 3 consecutive 0x00 characters being transmitted by the production programming tools. When 3 consecutive 0x00 characters are received, boot mode sends an ‘ACK’ (another 0x00) back to the production programming tools to indicate that communication is being established.

Figure 6 shows an example function, communication_setting(), that could be used within the production programming tool to implement the Communication setting phase sequence. In this example, the code will attempt to start communication with boot mode 20 times.

The production programming tools should hold MD low throughout this process.
If no ACK is received, it can be because a previous boot mode connection is still active. This can be verified using a boot mode Inquiry command. This is explained in section 5.2.

Once the production programming tools have received the ACK code, they should then transmit the “generic code” (0x55) to which boot mode will reply with the “boot code”, as shown in Figure 7. For the MCUs described in this application note, the boot code is 0xC6.

In a real production programming tool, the boot code alone is not sufficient to completely identify the MCU type being communicated with and the capabilities available through its boot mode. The “Signature”
command should be used to obtain additional details and determine such information. For example, some RA Family MCUs based on Cortex-M33 do not implement Device Lifecycle Management, so related boot mode commands are not available on those devices.

4. Boot Mode Commands
This section describes how production programming tools can interact with boot mode, after they have retrieved the boot code and the MCU boot mode has entered the Command acceptable phase.

4.1 Command Acceptable Phase
Once in Command Acceptable Phase, the MCU’s boot mode expects to receive command packets from the production programming tools, telling it which boot mode operation is to be carried out. Boot mode responds back to the production programming tools using data packets. Some commands also require data packets to be sent from the production programming tools back to boot mode providing additional information for use in the operation.

Sequence diagrams showing the transmission of packets for each command can be found in application note R01AN5562.

4.1.1 Command Packet Format
The production programming tools send information for the required operation to the MCU’s boot mode in the form of a command packet in format shown in Table 9.

Table 9. Command Packet

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH</td>
<td>1 byte</td>
<td>0x01</td>
<td>Start of command packet.</td>
</tr>
<tr>
<td>LNH</td>
<td>1 byte</td>
<td>-</td>
<td>Packet length (length of ‘CMD + command information’) [High].</td>
</tr>
<tr>
<td>LNL</td>
<td>1 byte</td>
<td>-</td>
<td>Packet length (length of ‘CMD + command information’) [Low].</td>
</tr>
<tr>
<td>CMD</td>
<td>1 byte</td>
<td>-</td>
<td>Command code, as described in section 4.1.3.</td>
</tr>
<tr>
<td>Command</td>
<td>0 to 255 bytes</td>
<td>-</td>
<td>Command information. For example:</td>
</tr>
<tr>
<td>information</td>
<td></td>
<td></td>
<td>• For write command: Start/End address.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• For erase command: Start/End address.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• For DLM state transit command: Source/Destination DLM state code.</td>
</tr>
<tr>
<td>SUM</td>
<td>1 byte</td>
<td>-</td>
<td>Sum data of ‘LNH + LNL + CMD + Command information’ (expressed as two’s complement). For example: LNH + LNL + CMD + Command information (1) + Command information (2) + ... + Command information(n) + SUM = 0x00</td>
</tr>
</tbody>
</table>

4.1.2 Data Packet
The production programming tools and the boot mode firmware send additional data to each other in the format shown in Table 10.
Table 10. Data Packet

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOD</td>
<td>1 byte</td>
<td>0x81</td>
<td>Start of data packet.</td>
</tr>
<tr>
<td>LNH</td>
<td>1 byte</td>
<td>-</td>
<td>Packet length (length of ‘RES + Data’) [High] (*1).</td>
</tr>
<tr>
<td>LNL</td>
<td>1 byte</td>
<td>-</td>
<td>Packet length (length of ‘RES + Data’) [Low] (*1).</td>
</tr>
<tr>
<td>RES</td>
<td>1 byte</td>
<td>-</td>
<td>Refer to R01AN5562 section “RES: Response code” for all the supported response codes.</td>
</tr>
</tbody>
</table>
| Data   | 1 to 1024 bytes (*2) | - | Transmit data:  
  - For write data transmission: Write data.  
  - For status transmission: Status code (STS) (*3), Flash status (FST) (*5), and Failure address (ADR) (*6).  
  - For DLM state request: DLM state code (DLM) (*4). |
| SUM    | 1 byte     | -     | Sum data of ‘LNH + LNL + RES + Data’ (expressed as two's complement).  
  For example: LNH + LNL + RES + Data(1) + Data(2) + ... + Data(n) + SUM = 0x00 |
| ETX    | 1 byte     | 0x03  | End of packet                                                               |

*1: If the host sends a packet whose length is 0 byte or over 1025 bytes, the microcontroller returns a packet with indefinite RES value.  
*2: If the host sends data that exceeds 1030 bytes, subsequent operations are not guaranteed.  
*3: Refer to R01AN5562 section “STS: Status code” for all the supported status code.  
*4: Refer to R01AN5562 section “DLM: Device Lifecycle Management state code” for all the supported DLM state code.  
*5: Refer to R01AN5562 section “FST: Flash status” for all the supported flash status code.  
*6: When a flash access error occurs, boot firmware returns the value of the start address of the flash sequencer command. When a flash access error does not occur, boot firmware returns 0xFFFFFFFF.

4.1.3 Summary of Boot Mode Commands

Table 11 is the summary of all the boot mode commands available on the MCUs described in this application note. For the commands that are not demonstrated in this application project, refer to application note R01AN5562 to understand the command format details, response, and error handling.
4.1.4 Boot Mode Firmware Operation

When the boot mode firmware receives a command packet, it performs packet analysis:

- The boot mode firmware recognizes the start of the command packet by receiving SOH. If the boot mode firmware receives something other than SOH, it waits until SOH is received.
- If ETX is not added to the received command packet, the boot mode firmware sends a ‘Packet error’.
- If the SUM of the received command packet is different from the sum value, the boot mode firmware sends a ‘Checksum error’.
- If the received command packets of LNH and LNL are different from the values specified in the packet format, the boot mode firmware sends a ‘Packet error’.
- If the CMD command in the received command packet is an undefined code, the boot mode firmware sends an ‘Unsupported command error’.
- If the received command packets of LNH and LNL are different from the values specified in each command, the boot mode firmware sends a ‘Packet error’.
- When an error described above occurs, the boot mode firmware does not process and returns to the command waiting state.

When the packet analysis has successfully completed, the boot mode firmware executes command processing. Refer to the explanation of each command for specific details.

When a command completes successfully, the boot mode firmware stays in the ‘Command acceptable phase’.

5. Typical Use Cases of Boot Mode Commands

This section describes several typical use cases of the boot mode commands. The command format and example code are provided.
5.1 Overview of Use cases

For more details on the use cases described in this section, refer to the application note R01AN5562 section Command List. Table 12 details the specific subsections from R01AN5562 that match the following use cases from this application note.

<table>
<thead>
<tr>
<th>Table 12. Boot Mode Command Use Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section in this application note</td>
</tr>
<tr>
<td>Boot Mode Inquiry Command</td>
</tr>
<tr>
<td>Initialize MCU Command</td>
</tr>
<tr>
<td>Disable Initialize Command</td>
</tr>
<tr>
<td>Check Whether Initialize Command is Disabled</td>
</tr>
<tr>
<td>Inject DLM keys</td>
</tr>
<tr>
<td>Verify DLM keys</td>
</tr>
<tr>
<td>DLM State Request</td>
</tr>
<tr>
<td>DLM State Transition</td>
</tr>
<tr>
<td>TrustZone Boundary Setting Command</td>
</tr>
<tr>
<td>TrustZone Boundary Request Command</td>
</tr>
</tbody>
</table>

5.2 Inquiry Command

The Inquiry command checks whether a previous boot mode connection is still alive.

![Figure 8. Inquiry Command Packet](image)

```python
""# This routine demonstrates the "Inquiry command".
""# This command checks if boot firmware is in 'Command acceptable phase' or not.

def command_inquiry():
    SOH = b'\x01'
    LNH = b'\x00'
    LNL = b'\x00'
    CMD = b'\x00'
    SUM = b'\xFF'
    EXT = b'\x03'
    command = SOH + LNH + LNL + CMD + SUM + EXT
    print("Sending Inquiry command:")
    print_bytes_hex(command)
    ser.write(command)
    time.sleep(LONG_DELAY)
```

![Figure 9. Inquiry Command Example Code](image)

5.3 Initialize the MCU

The commands introduced in this section are used to ensure that the MCU is in the SSD DLM state and ready for other operations, such as flash programming.
5.3.1 Initialize Command

The Initialize command can be executed in the SSD, NSECSD, and DPL states. It clears the User area, Data area, Config area, Boundary settings, and Key index (Wrapped keys). In addition, the DLM state transitions to SSD from NSECSD and DPL. Erase processing is performed unaffected by the flash block protection settings (BPS, BPS_SEC). However, if PBPS and PBPS_SEC are set, then the Initialize command cannot be processed. This command is typically not used in the production programming environment unless the MCU has been previously used (for example, an evaluation board being used for testing purposes).

The Initialize command takes the current DLM state as an input parameter. So, prior to executing the Initialize command, the production programming tools need to acquire the current DLM state using the DLM State Request Command (see demonstration in section 5.6.1).

<table>
<thead>
<tr>
<th>Command packet</th>
<th>Source DLM state code</th>
<th>Destination DLM state code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH (1 byte)</td>
<td>0x01</td>
<td></td>
</tr>
<tr>
<td>LNH (1 byte)</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>LNL (1 byte)</td>
<td>0x03</td>
<td></td>
</tr>
<tr>
<td>CMD (1 byte)</td>
<td>0x50 (Initialize command)</td>
<td>0x02 : SSD</td>
</tr>
<tr>
<td>SDLM (1 byte)</td>
<td>Source DLM state code</td>
<td>0x03 : NSECSD</td>
</tr>
<tr>
<td>Source DLM state code</td>
<td>0x04 : DPL</td>
<td></td>
</tr>
<tr>
<td>DDLM (1 byte)</td>
<td>Destination DLM state code</td>
<td>0x02 : SSD</td>
</tr>
<tr>
<td>SUM (1 byte)</td>
<td>Sum data</td>
<td></td>
</tr>
<tr>
<td>ETX (1 byte)</td>
<td>0x03</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Packet [status OK]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH (1 byte)</td>
<td>0x81</td>
</tr>
<tr>
<td>LNH (1 byte)</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL (1 byte)</td>
<td>0x0A</td>
</tr>
<tr>
<td>RES (1 byte)</td>
<td>0x50 (OK)</td>
</tr>
<tr>
<td>STS (1 byte)</td>
<td>0x00 (OK)</td>
</tr>
<tr>
<td>FST (4 bytes)</td>
<td>0xFFFFFFFF (unused code)</td>
</tr>
<tr>
<td>ADR (4 bytes)</td>
<td>0xFFFFFFFF (unused code)</td>
</tr>
<tr>
<td>SUM (1 byte)</td>
<td>0xAE</td>
</tr>
<tr>
<td>ETX (1 byte)</td>
<td>0x03</td>
</tr>
</tbody>
</table>

Figure 10. Initialize Command Packets
Figure 11. Initialize Command Example Code

5.3.2 Check Whether Initialize Command is Disabled

For a non-factory fresh device (for example, a device previously used for development/testing purposes), it is possible that boot mode Initialize command might have been disabled (and other changes made). Checking whether the MCU Initialize command is disabled or not can be achieved by using the Parameter Request command.
### 6.13.2.1 Command packet

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH</td>
<td>1 byte</td>
<td>0x01</td>
</tr>
<tr>
<td>LNH</td>
<td>1 byte</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>1 byte</td>
<td>0x02</td>
</tr>
<tr>
<td>CMD</td>
<td>1 byte</td>
<td>0x52</td>
</tr>
<tr>
<td>PMID</td>
<td>1 byte</td>
<td>Parameter ID</td>
</tr>
<tr>
<td>SUM</td>
<td>1 byte</td>
<td>Sum data</td>
</tr>
<tr>
<td>ETX</td>
<td>1 byte</td>
<td>0x03</td>
</tr>
</tbody>
</table>

### 6.13.2.2 Data packet [Parameter data]

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOD</td>
<td>1 byte</td>
<td>0x81</td>
<td></td>
</tr>
<tr>
<td>LNH</td>
<td>1 byte</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>LNL</td>
<td>1 byte</td>
<td>N + 1</td>
<td></td>
</tr>
<tr>
<td>RES</td>
<td>1 byte</td>
<td>0x52 (OK)</td>
<td></td>
</tr>
<tr>
<td>PRMT</td>
<td>N byte</td>
<td>Parameter data</td>
<td>[PMID = 0x01] 0x00: Initialization is disabled 0x07: Initialization is enabled</td>
</tr>
<tr>
<td>SUM</td>
<td>1 byte</td>
<td>Sum data</td>
<td></td>
</tr>
<tr>
<td>ETX</td>
<td>1 byte</td>
<td>0x03</td>
<td></td>
</tr>
</tbody>
</table>

| N = 1 to 16 |

**Figure 12. Command Packet: Check whether Initialize Command is Disabled**
Figure 13. Example Code: Check whether Initialize Command is Disabled

5.3.3 Disable Initialize Command
As part of the final production programming process, the Initialize command can be disabled if required. This step is non-reversable, so exercise with caution! This command is included in the bootmode_demonstration_code.py code, but the section of the code is not enabled due to the risk of locking up the MCU during the development of tools for production programming. Refer to section 6.6 for details on enabling this command in the demonstration code.
### 6.12.2.1 Command packet

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH</td>
<td>(1 byte)</td>
<td>0x01</td>
</tr>
<tr>
<td>LNH</td>
<td>(1 byte)</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>(1 byte)</td>
<td>N + 2</td>
</tr>
<tr>
<td>CMD</td>
<td>(1 byte) Parameter setting command</td>
<td>0x51</td>
</tr>
</tbody>
</table>
| PMID   | (1 byte) Parameter ID                           | 0x01 : Setting of disable initialization, [PMID = 0x01]
| PRMT   | (N byte) Parameter data                         | 0x00 : Disable initialization, [PMID = 0x01] |
| SUM    | (1 byte) Sum data                               |         |
| ETX    | (1 byte)                                         | 0x03    |

### 6.12.2.2 Data packet [status OK]

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOD</td>
<td>(1 byte)</td>
<td>0x81</td>
</tr>
<tr>
<td>LNH</td>
<td>(1 byte)</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>(1 byte)</td>
<td>0xA</td>
</tr>
<tr>
<td>RES</td>
<td>(1 byte)</td>
<td>0x51 (OK)</td>
</tr>
<tr>
<td>STS</td>
<td>(1 byte)</td>
<td>0x00 (OK)</td>
</tr>
<tr>
<td>FST</td>
<td>(4 bytes)</td>
<td>0xFFFFFFFF (unused code)</td>
</tr>
<tr>
<td>ADR</td>
<td>(4 bytes)</td>
<td>0xFFFFFFFF (unused code)</td>
</tr>
<tr>
<td>SUM</td>
<td>(1 byte) Sum data</td>
<td></td>
</tr>
<tr>
<td>ETX</td>
<td>(1 byte)</td>
<td>0x03</td>
</tr>
</tbody>
</table>

**Figure 14.** Disable the ‘Initialize Command’ Command Packet
5.4 TrustZone Boundary Region Setup

This section explains the operational flow of the TrustZone boundary setup and introduces the command packet and the example code.

5.4.1 Operational Flow

The recommended flow when setting up the TrustZone partition boundary regions is:

1. Acquire the Trust Zone partition boundary information from the application.
2. Check DLM state. The TrustZone partition boundaries can only be set up in SSD DLM state.
3. Set up boundaries.
4. Verify the boundaries are set up properly.

5.4.2 Acquire the Boundary Information from an Application

The IDAU region information is stored in a .rpd file generated as a post-build step in an RA project in e² studio, or a RASC-generated EWARM / MDK project. Table 13 shows how to find the .rpd file based on the IDE.

---

```python
# ================
# The "Initialize command" can be disabled using the
# "Parameter setting command".
# !! WARNING !!
# This step is non-reversible, so invoke with caution!

def command_disable_initialize():

    #0x51: Parameter setting command
    CMD=b'\x51'
    # Parameter ID
    PMID=b'\x01'
    # Parameter Data 0x00 is for disable initialize
    FRMT=b'\x00'
    SUM=calc_sum(LNH + LNL + CMD + PMID + FRMT)
    EXT=b'\x03'
    command = SOH + LNH + LNL + CMD + PMID + FRMT + SUM + EXT

    print ("\nSending Disable Initialize command:")
    print_bytes_hex(command)
    print ("\nWarning: After MCU Initialize is disabled, it can never be re-enabled. ")
    ser.write(command)
    time.sleep(LONG_DELAY)

    #acquire the response
    return_packet = receive_data_packet()
    # the RES byte is the fourth index
    RES = return_packet[3]
    if RES != 0x51:
        print ("Disable Initialize command - FAIL")
    else:
        print ("Disable Initialize command - SUCCESS")
```

Figure 15. Example Code: Disable the ‘Initialize Command’
Table 13. The .rpd File Location Based on IDE

<table>
<thead>
<tr>
<th>IDE</th>
<th>Location of the .rpd file</th>
</tr>
</thead>
<tbody>
<tr>
<td>e² studio</td>
<td>Secure project root folder: &lt;secure_project_name&gt;\Debug&lt;secure_project_name&gt;.rpd</td>
</tr>
<tr>
<td>EWARM</td>
<td>Secure project root folder: &lt;secure_project_name&gt;\Debug\exe&lt;secure_project_name&gt;.rpd</td>
</tr>
<tr>
<td>MDK</td>
<td>Secure project root folder: &lt;secure_project_name&gt;\Objects&lt;secure_project_name&gt;.rpd</td>
</tr>
</tbody>
</table>

The format of the .rpd file is identical across the IDEs. Figure 16 shows the contents of the .rpd file.

![Figure 16. Obtain the IDAU Region Size using EWARM](image)

The FLASH_S_SIZE is the size of the Secure Code Flash region without the NSC region. The production programming tools need to convert this value to KB (kilobytes) and then assign this value to the CFS1 as shown in Figure 19.

The FLASH_C_SIZE is the size of the Secure Code Flash region including the NSC region. The production programming tools need to convert the sum of FLASH_S_SIZE and FLASH_C_SIZE to KB and then assign this sum to CFS2 as shown in Figure 19.

The DATA_FLASH_S is the size of the Secure Data Flash region. The production programming tools need to convert this value to KB and then assign this value to DFS1 as shown in Figure 19.

The production programming tools can ignore the other fields.

5.4.3 TrustZone Boundary Request Command

Reading the configured IDAU region setup can be achieved using the command in Figure 17. The example code to perform this function is shown in Figure 18.
### 6.11.2.1 Command packet

- **SOH**: 0x01 (1 byte)
- **LNH**: 0x00 (1 byte)
- **LNL**: 0x01 (1 byte)
- **CMD**: 0x4F (Boundary request command) (1 byte)
- **SUM**: 0x80 (1 byte)
- **ETX**: 0x03 (1 byte)

### 6.11.2.2 Data packet [Boundary setting data]

- **SOD**: 0x81 (1 byte)
- **LNH**: 0x00 (1 byte)
- **LNL**: 0x08 (1 byte)
- **RES**: 0x4F (OK) (1 byte)
- **CFS1**: Size of code flash secure region (without NSC) [KB] (2 bytes)
- **CFS2**: Size of code flash secure region (KB) (2 bytes)
- **DFS1**: Size of data flash secure region [KB] (2 bytes)
- **DFS2**: Size of data flash secure region [KB] (2 bytes)
- **SRS1**: Size of SRAM secure region (without NSC) [KB] (2 bytes)
- **SRS2**: Size of SRAM secure region [KB] (2 bytes)
- **SUM**: Sum data (1 byte)

**NSC**: Non-secure callable regions

---

**Figure 17. Command Packet for Reading IDAU Region Setup**

```python
# This routine demonstrates "TrustZone boundary request command".
def command_trustzone_boundary_request():
    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x01'
    CMD=b'\x4F'  # Boundary request command
    SUM=b'\xB0'
    ETX=b'\x03'
    command = SOH + LNH + LNL + CMD + SUM + ETX

    print("Sending read boundary region command:")
    print_bytes_hex(command)
    ser.write(command)
    time.sleep(LONG_DELAY)
    # acquire the response
    return_packet = receive_data_packet()

    # the RES byte is the fourth index
    RES = return_packet[3]

    if RES != 0x4F:
        print('read boundary region -FAIL')
    else:
        print('read boundary region - SUCCESS')

    print_boundary_region(return_packet)

    return RES
```

**Figure 18. Example Code: Request TrustZone Boundary**
5.4.4 TrustZone Boundary Setting Command

Figure 19 is the command packet for setting up the TrustZone boundary. The new stored boundary setting becomes effective after resetting the device.

<table>
<thead>
<tr>
<th>SOH</th>
<th>1 byte</th>
<th>0x01</th>
</tr>
</thead>
<tbody>
<tr>
<td>LNH</td>
<td>1 byte</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>1 byte</td>
<td>0x0B</td>
</tr>
<tr>
<td>CMD</td>
<td>1 byte</td>
<td>0x4E (Boundary setting command)</td>
</tr>
<tr>
<td>CFS1</td>
<td>2 bytes</td>
<td>Size of code flash secure region (without NSC) [KB] (ex.) 0x0100 -&gt; 0x01, 0x00 (256 KB)</td>
</tr>
<tr>
<td>CFS2</td>
<td>2 bytes</td>
<td>Size of code flash secure region [KB] (ex.) 0x0100 -&gt; 0x01, 0x00 (256 KB) * 32 KB align</td>
</tr>
<tr>
<td>DFS1</td>
<td>2 bytes</td>
<td>Size of Data Flash Secure region [KB] (ex.) 0x0004 -&gt; 0x00, 0x04 (4 KB)</td>
</tr>
<tr>
<td>SRS1</td>
<td>2 bytes</td>
<td>Size of SRAM Secure region (without NSC) [KB] (ex.) 0x0040 -&gt; 0x00, 0x40 (64 KB)</td>
</tr>
<tr>
<td>SRS2</td>
<td>2 bytes</td>
<td>Size of SRAM secure region [KB] (ex.) 0x0040 -&gt; 0x00, 0x40 (64 KB) * 8 KB align</td>
</tr>
<tr>
<td>SUM</td>
<td>1 byte</td>
<td>Sum data</td>
</tr>
<tr>
<td>ETX</td>
<td>1 byte</td>
<td>0x03</td>
</tr>
</tbody>
</table>

NSC: Non-secure callable regions
* If CFS2 or SRS2 does not comply with the alignment, the boot firmware rounds them down.

**Figure 19. Command Packet for TrustZone Boundary Setup**

Figure 20 shows the example code for setting up the TrustZone boundary.
5.5 DLM Key Handling

DLM keys are stored in dedicated, non-user-accessible memory within the MCU, with one slot dedicated to each authenticated DLM transition. Therefore, when injecting the key, it is necessary to specify what target DLM state the key is for, so that the key is placed into the correct slot.

Key injection for DLM SECDBG and NSECDBG states is demonstrated in this application note. Injection of the DLM RMA key can follow similar sequence, but is not demonstrated in this application note.

```python
# This routine demonstrates "TrustZone boundary setting command".
def command_setup_trustzone_boundary():
    print("Setting Up the TrustZone boundary:")
    print("----------------------------------
")
    SCH=b'\x01',
    LNH=b'\x00',
    LNL=b'\x0B',
    CMD=b'\xDE' # Boundary setting command
    print('Set up 8KB of secure code flash region without NSC'
    CFS1_1=b'\x00'
    CFS1_2=b'\x08' #8KB
    print('Set up 32KB of secure code flash region with NSC'
    CFS2_1=b'\x00'
    CFS2_2=b'\x20' #32KB
    print('Set up 4KB of secure data flash region'
    DFS1_1=b'\x00'
    DFS1_2=b'\x04' #4KB
    print('Set up 2KB of secure sram region without NSC'
    SRS1_1=b'\x00'
    SRS1_2=b'\x02' #2KB
    print('Set up 32KB of secure sram region with NSC'
    SRS2_1=b'\x00'
    SRS2_2=b'\x20' #32KB #Size of SRAM Secure region (without NSC) [KB]
    SUM=calc_sum(LNH + LNL + CMD + CFS1_1 + CFS1_2 + CFS2_1 + CFS2_2 + 
                   DFS1_1 + DFS1_2 + SRS1_1 + SRS1_2 + SRS2_1 + SRS2_2) 
    EXT=b'\x03'
    command = SCH + LNH + LNL + CMD + CFS1_1 + CFS1_2 + CFS2_1 + CFS2_2 + 
                   DFS1_1 + DFS1_2 + SRS1_1 + SRS1_2 + SRS2_1 + SRS2_2 + SUM + EXT
    print("----------------------------------\n")
    print("Sending IADU region set up command:")
    print_bytes_hex(command)
    ser.write(command)
    time.sleep(LONG_DELAY)
    print("----------------------------------\n")
    #acquire the response
    return_packet = receive_data_packet()
    # the RES byte is the fourth index
    RES = return_packet[3]

    if RES != 0x4E:
        print('Set up boundary region - FAIL')
    else:
        print('Set up boundary region - SUCCESS')

    return RES
```

Figure 20. Example Command Setting up TrustZone Boundary
Note: The injection of user keys is very similar to DLM keys, except that additional address information is required in the corresponding command, as these keys are stored in user flash.

Keys can be generated using the following systems:

- The “Security Key Management Tool"
- The Renesas Device Lifecycle Management server available from the Renesas website

The procedure for generating the DLM keys is described in R11AN0469.

### 5.5.1 Inject DLM Keys

Injecting a DLM key requires a two-stage sequence, as shown in Figure 21.

![DLM Key Injection Flow](image)

#### Figure 21. DLM Key Injection Flow

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH</td>
<td>(1 byte)</td>
<td>0x01</td>
</tr>
<tr>
<td>LNH</td>
<td>(1 byte)</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>(1 byte)</td>
<td>0x02</td>
</tr>
<tr>
<td>CMD</td>
<td>(1 byte)</td>
<td>0x28</td>
</tr>
<tr>
<td>KYTY</td>
<td>(1 byte) Key type</td>
<td>0x01: SECDBG_KEY, 0x02: NONSECDBG_KEY, 0x03: RMA_KEY</td>
</tr>
<tr>
<td>SUM</td>
<td>(1 byte) Sum data</td>
<td></td>
</tr>
<tr>
<td>ETX</td>
<td>(1 byte)</td>
<td>0x03</td>
</tr>
</tbody>
</table>

![DLM Key Injection Command Packet](image)

#### Figure 22. DLM Key Injection Command Packet
To properly prepare for the DLM key injection, the production programming tools need to understand the .rkey file format. The .rkey file is also base64 encoded, so the production programming tools need to first decode the data prior to accessing the fields. Once the .rkey file is decoded, the .rkey file data fields can then be accessed. The format of the fields is shown in Figure 24, and is further explained in the user manual of Secure Key Management Tool.

Key Data is stored in the order and size shown in Figure 24. The byte order is big-endian.
The demonstration code for DLM key injection is included in function `command_inject_dlm_key()`. The main operations carried out by this function are:

- Read the SECDBG or NSECDBG key file (.rkey) to an array.
- Decode the base64 array so all the data fields can be accessed.
- Parse the .rkey file for the field of magic code, key type, w-ufpk, initialization vector, and the encrypted DLM key to ensure valid content.
- Issue DLM Key Injection command packet and verify the response.
- Issue DLM Key Injection data packet and verify the response using the decoded key data.

Figure 25 and Figure 26 show the example code.
# This routine demonstrates "DLM key injection" command.
# The supported key types are SECDDBG and NSECDDBG key

```python
def command_inject_dlm_key(dlm_key_type):
    if (dlm_key_type != b'\x01' and dlm_key_type != b'\x02'):
        print('This key type is not supported')
        return UNKNOWN_KEY
    elif (dlm_key_type == b'\x01'):
        text_file = open("./dlm_keys/SECDDBG.rkey", "r")
    elif (dlm_key_type == b'\x02'):
        text_file = open("./dlm_keys/NON-SECDDBG.rkey", "r")

    message_bytes, encrypted_key_size = parse_the_dlm_key_field(text_file, dlm_key_type)
    if (message_bytes != 0 and encrypted_key_size != 0):
        # Proceed with the DLM key injection command packet stage
        SOH = b'\x01'  # start of command packet
        LNH = b'\x00'
        LNL = b'\x02'
        CMD = b'\x28'  # DLM Key injection command
        KYTY = dlm_key_type
        SUM = calc_sum(LNH + LNL + CMD + KYTY)
        EXT = b'\x03'
        command = SOH + LNH + LNL + CMD + KYTY + SUM + EXT
        print("n----------------------------------------------n")
        print("Sending dlm key injection command packet: ")
        print_bytes_hex(command)
        ser.write(command)
        time.sleep(LONG_DELAY)

    return_packet = receive_data_packet()
    # the RES byte is the fourth index
    RES = return_packet[3]

    # Acquire the response of the command packet stage
    if RES == 0x28:
        print('DLM key injection command packet - FAIL')
    else:
        # the STS is the fifth index
        STS = return_packet[4]
        if STS == 0x00:
            print('DLM key injection command packet - FAIL')
        # the SUM is the 14th index
        SUM = return_packet[13]
        if SUM == 0xD6:
            print('DLM key injection command packet - SUCCESS')

    # start the data packet stage
```
```

Figure 25. Example Code: DLM Key Injection – Part 1
5.5.2 Verify DLM Keys

After injecting the DLM keys, the production programming tools should invoke a verify command to confirm correct injection. Figure 27 shows the command packet information for verifying the DLM keys.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH</td>
<td>1 byte</td>
<td>0x01</td>
</tr>
<tr>
<td>LNH</td>
<td>1 byte</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>1 byte</td>
<td>0x02</td>
</tr>
<tr>
<td>CMD</td>
<td>1 byte</td>
<td>0x29 (Key verify command)</td>
</tr>
<tr>
<td>KYTY</td>
<td>1 byte</td>
<td>Key type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x01: SECDBG_KEY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x02: NONSECDBG_KEY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x03: RMA_KEY</td>
</tr>
<tr>
<td>SUM</td>
<td>1 byte</td>
<td>Sum data</td>
</tr>
<tr>
<td>ETX</td>
<td>1 byte</td>
<td>0x03</td>
</tr>
</tbody>
</table>
Figure 28. DLM Key Verify Command Example Code

5.6 DLM State Handling
This section explains the DLM state request command and the non-authenticated DLM state transition command.

5.6.1 DLM State Request
The state request command is demonstrated in the included example code. Figure 29 shows the state request command packet format. Figure 30 shows the example code.
### 6.9.2.1 Command packet

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH</td>
<td>Start Of Header</td>
<td>0x01</td>
</tr>
<tr>
<td>LNH</td>
<td>Length Header</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>Length subheader</td>
<td>0x01</td>
</tr>
<tr>
<td>CMD</td>
<td>Command</td>
<td>0x2C</td>
</tr>
<tr>
<td>SUM</td>
<td>Sum</td>
<td>0xD3</td>
</tr>
<tr>
<td>ETX</td>
<td>End Of Transaction</td>
<td>0x03</td>
</tr>
</tbody>
</table>

### 6.9.2.2 Data packet [DLM state]

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOD</td>
<td>Start Of Data</td>
<td>0x81</td>
</tr>
<tr>
<td>LNH</td>
<td>Length Header</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>Length subheader</td>
<td>0x02</td>
</tr>
<tr>
<td>RES</td>
<td>Result</td>
<td>0x2C</td>
</tr>
<tr>
<td>DLM</td>
<td>DLM state code</td>
<td>DLM state code</td>
</tr>
<tr>
<td>SUM</td>
<td>Sum data</td>
<td></td>
</tr>
<tr>
<td>ETX</td>
<td>End Of Transaction</td>
<td>0x03</td>
</tr>
</tbody>
</table>

Figure 29. DLM State Request Command Packet
5.6.2 DLM State Transition

This section covers the non-authenticated DLM state transition. Authenticated transitions are not generally required in production programming tools.

The recommended flow when performing DLM state transition is described in Figure 31. The current DLM state is a required parameter for the DLM state transition command and should be acquired first.
Figure 31. Recommended Flow for Performing DLM State Transition

6.2.2.1 Command packet

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH</td>
<td>SOH byte</td>
<td>0x01</td>
</tr>
<tr>
<td>LNH</td>
<td>LNH byte</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>LNL byte</td>
<td>0x03</td>
</tr>
<tr>
<td>CMD</td>
<td>CMD byte (DLM state transit command)</td>
<td>0x71</td>
</tr>
<tr>
<td>SDLM</td>
<td>Source DLM state code</td>
<td></td>
</tr>
<tr>
<td>DDL</td>
<td>Destination DLM state code</td>
<td></td>
</tr>
<tr>
<td>SUM</td>
<td>Sum data</td>
<td></td>
</tr>
<tr>
<td>ETX</td>
<td>ETX byte</td>
<td>0x03</td>
</tr>
</tbody>
</table>

6.2.2.2 Data packet [status OK]

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOD</td>
<td>SOD byte</td>
<td>0x81</td>
</tr>
<tr>
<td>LNH</td>
<td>LNH byte</td>
<td>0x00</td>
</tr>
<tr>
<td>LNL</td>
<td>LNL byte</td>
<td>0x0A</td>
</tr>
<tr>
<td>RES</td>
<td>RES byte (OK)</td>
<td>0x71 (OK)</td>
</tr>
<tr>
<td>STS</td>
<td>STS byte (OK)</td>
<td>0x00 (OK)</td>
</tr>
<tr>
<td>FST</td>
<td>FST byte (unused code)</td>
<td>0xFFFFFFFFFF</td>
</tr>
<tr>
<td>ADR</td>
<td>ADR byte (unused code)</td>
<td>0xFFFFFFFFFF</td>
</tr>
<tr>
<td>SUM</td>
<td>SUM byte</td>
<td>0x8D</td>
</tr>
<tr>
<td>ETX</td>
<td>ETX byte</td>
<td>0x03</td>
</tr>
</tbody>
</table>

Figure 32. DLM State Transition Command Packet
Figure 33. Example Code for DLM State Transition Command

6. Running the Python Example Code

Many of the typical boot mode commands are implemented in the included example code, which can be used as a starting point for writing a complete production programming tool. The code snippets provided in earlier sections are taken from the examples.
6.1 Set up the Python Environment
To execute the demonstration code supplied along with this application note, it is necessary to install the following software packages first. Follow the links below to acquire and install the software needed:

- Install Python:
  - Python 3.10 or later (https://www.python.org/downloads/)
- Install the pySerial package
  - pySerial 3.5 (https://pyserial.readthedocs.io/en/latest/pyserial.html#installation)

6.2 Setting Up the Hardware
The demonstration code works with all the MCU groups covered in this application project. The example shown here uses an RA6M4 MCU fitted to an EK-RA6M4 evaluation board.

To cause the MCU to enter boot mode on reset, first ensure that a jumper has been placed on the MD ("BOOT MODE") jumper, in this case J16. This is shown in Figure 34, which also highlights the location of the reset button.
Next, decide whether the serial or USB interface will be used for boot mode communication.

If the USB interface is used:

- Using a USB micro to B cable, connect J11 (USB FS) from the EK-RA6M4 to the development PC to provide USB Device connection.
- See Table 8 for more general details on the USB connection.

![Hardware Setup using USB Full Speed Port](image)

**Figure 35. Hardware Setup using USB Full Speed Port**

If the serial interface is used:

- Connect the four pins in Table 14 on the UART to USB converter to the EK-RA6M4 and connect the other end of the converter to the PC’s USB port. Note that there may be variations on the voltage output from the converter cable. For the FTDI cable demonstrated in Figure 36, the voltage supply to the MCU is 5V. Another converter may output 3.3V, so the production programming tool should take this into consideration when setting up the hardware.
- See Table 7 for more details of the serial interface.
Table 14. Connection through the UART Interface

<table>
<thead>
<tr>
<th>UART to USB Converter</th>
<th>EK-RA6M4</th>
</tr>
</thead>
<tbody>
<tr>
<td>RX</td>
<td>J3: Pin 4 (MCU P109 (TXD9))</td>
</tr>
<tr>
<td>TX</td>
<td>J3: Pin 5 (MCU P110 (RXD9))</td>
</tr>
<tr>
<td>+5V (FTDI cable power output voltage. Check the voltage output on the converter used.)</td>
<td>J3: Pin 21 (MCU power) (If +3.3V is provided from the converter, then connect to Pin 1 of J3). If the production programming board has stable power supply, then this pin connection is not needed.</td>
</tr>
<tr>
<td>GND</td>
<td>J3: Pin 39 (MCU Ground)</td>
</tr>
</tbody>
</table>

![Figure 36. Hardware Setup using UART to USB Converter](image)

Once the physical communication mechanism is connected, whether serial or USB, ensure the board is powered up and then press the Reset button to enter boot mode.

6.3 Running the First Demo Code

Unzip `production_programming_demo_cm33_dlm.zip` to reveal two Python files and the `\dlm_keys` folder which holds two DLM keys which can be injected into the MCU.
Figure 37. Python Demo Code and Sample DLM Keys

The first of the demonstration examples, initialize_mcu_first.py, is intended to ensure that the MCU is correctly configured for production programming, running an Initialize command if required.

The full functionality of this code is described in Figure 38:

Figure 38. Operational Flow of the First Demo Code

To execute the example, open a command line prompt and navigate to the folder where the Python example code is stored. Then enter:

```
python initialize_mcu_first.py
```

Figure 39 shows sample output from running the demonstration with a USB connection to the board.
Some Renesas evaluation boards might be distributed in the CM state. In this case, the demonstration code will transition the DLM state from CM to SSD, then the Initialize command will be executed. In this case, there is no need to check whether the Initialize command is disabled or not as the Initialize command cannot be issued in the CM state and transitioning from CM to SSD is a one-way process.
Figure 40. Running the Demo on a MCU in CM State

After the demonstration example finishes running, follow the warning in the output to reset the board before running the second demonstration example.

Note that with the USB connection, the code has automatically identified the RA boot mode USB CDC interface, and automatically connected to it.

With a serial connection, it is necessary to enter the COM port to use manually. This is shown in partial output in Figure 39.
6.4 Running the Second Demonstration Code

The second of the demonstration examples, `bootmode_demonstration_code.py`, is intended to show the main steps likely to be required for a real production programming sequence (except for programming an application image into flash).

The full functionality of this code is described in Figure 42.
To execute the example, ensure that you have reset the board, then enter the following command in the previously opened command prompt:

```
python bootmode_demonstration_code.py
```

Below is sample output from running the demonstration with a USB connection to the board.

Establishing the Connection (USB)
6.4.1 Establishing the Connection (USB)

If a serial connection is used, it will be necessary to enter the COM port manually, as shown in Figure 41.

Figure 43. Establishing the Connection USB
6.4.2 Checking Current DLM State and Configuring TrustZone Partition Boundaries

Figure 44. Checking Current DLM State and Configuring TrustZone Partition Boundaries
6.4.3 Injecting DLM Keys

Figure 45. Injecting DLM Keys
If the Python code is modified to change the value of `DEBUG_OUTPUT_ENABLE` from 0 to 1, then additional details will be displayed as the content of the .rkey file is processed.
6.4.4 Configuring Final DLM State

![Figure 46. Configuring Final DLM State](image)

6.5 Testing Authenticated DLM Transitions

Authenticated DLM transitions are not generally required to be supported in production programming tools, as such transitions are generally only required during product development, or for in-field debug.

This means that the example code simply uses the DLM key verify command to check that keys have been injected correctly.

However, if required, it is possible to test that the injected keys do indeed allow authenticated DLM transitions by referring to the following two sections from Application Note R11AN0469 to perform the authenticated transitions using the Renesas Flash Programmer:

- This step is typically not needed in a production programming environment. Perform DPL to NSECSD transition following section “Authenticated Transition from Deployed State to Non-secure Debug State”. The plaintext raw NSECDBG key value for the example NON-SECDBG.rkey file is “010102030405060708090A0B0C0D0E0F”. This value needs to be used when transitioning from the DPL state to the NSECSD state.
- Perform NSECSD to SSD transition following section “Authenticated Transition from Non-secure Debug State to Secure Debug State”. The plaintext raw SECDBG key value for the example SECDBG.rkey file is “000102030405060708090A0B0C0D0E0F”. This value needs to be used when transitioning from the NSECSD state to the SSD state.

Note: Unlike using the “Initialize” command, the Code Flash, Data Flash, and TrustZone partition boundary settings are preserved in this process.

6.6 Disabling the Initialize Command

The bootmode demonstration code.py example contains a function called command_disable_initialize(), which will cause the ‘Initialize’ boot mode command to be disabled. Executing command_disable_initialize() prevents the DLM state from being reset to the SSD state using the ‘Initialize’ command in the future.
This action may be required during a real production programming run, but not while doing testing. Therefore, the calling of this function is disabled by default. To enable the call, change the value of `INVOKE_DISABLE_INITIALISE_COMMAND` from 0 to 1.

Once changing the value of `INVOKE_DISABLE_INITIALISE_COMMAND` from 0 to 1, as an extra level of protection, it will still be necessary to enter `YES` when prompted in order for the call to `command_disable_initialize()` to be made. When `INVOKE_DISABLE_INITIALISE_COMMAND` is set to 1, the demonstration code will display the following prompt before ending:

![Image](image.png)

**Figure 47. Disable the Initialize Command**

The next time that `initialize_mcu_first.py` is run, the system will report that the Initialize command is disabled:

![Image](image.png)

**Figure 48. Verify that the Initialize Command is disabled**

In this case, the only way to partially recover the board is to use the injected DLM keys to move the DLM state back to SSD as explained in section 6.5.
7. References

- Standard Boot Firmware for the RA Family MCUs Based on Arm Cortex-M33 (R01AN5562)
- Renesas RA Family Device Lifecycle Management Key Injection Application Note (R11AN0469)
- Renesas RA Family RA6M4 User’s Manual: Hardware (R01UH0890)
- Renesas RA Family MCU Security Design with TrustZone – IP Protection (R11AN0467)
- Renesas RA6 Family Quick Design Guide (R01AN5775)
- Renesas RA4 Family Quick Design Guide (R01AN5988)
- Security Key Management Tool User's Manual (R20UT5254)

8. Website and Support

Visit the following URLs to learn about the RA family of microcontrollers, download tools and documentation, and get support.

- EK-RA6M4 Resources: renesas.com/ra/ek-ra6m4
- RA Product Information: renesas.com/ra
- Flexible Software Package (FSP): renesas.com/ra/fsp
- RA Product Support Forum: renesas.com/ra/forum
- Renesas Support: renesas.com/support
## Revision History

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Description</th>
<th>Page</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>May 25, 2023</td>
<td>—</td>
<td>—</td>
<td>First release of this document.</td>
</tr>
</tbody>
</table>
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 VIL (Max.) and VIH (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 VIL (Max.) and VIH (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.
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 the circuits, software, or information.

2. Renesas Electronics hereby expressly disclaims any warranties against 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 and 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 depend 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 systems; 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.