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

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

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:

- https://www.keil.com/pack/doc/CMSIS/Pack/html/algorithmFunc.html

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

Lifecycle state	Definition and State Features	Debug Level	Boot mode access
CM	• "Chip Manufacturing" • This is the state when the developer receives the device.	DBG2	• Available • No access to code/data flash
SSD	• "Secure Software Development" • The secure part of the application is being developed.	DBG2	• Available • Can program/erase/read all code/data flash
NSECSD	• "Non-SECure Software Development" • The non-secure part of the application is being developed.	DBG1	• Available • Can program/erase/read non-secure code/data flash
DPL	• "DePLoyed" • The device is in-field.	DBG0	• Available. • No access to code/data flash
LCK_DBG	• "LoCKed DeBuG" • Device is in-field and the debug interface is permanently disabled.	DBG0	• Available • No access to code/data flash
LCK_BOOT	• "LoCKed BOOT interface" • Device is in-field and the debug interface and the boot mode interface are permanently disabled.	DBG0	• Not available

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.

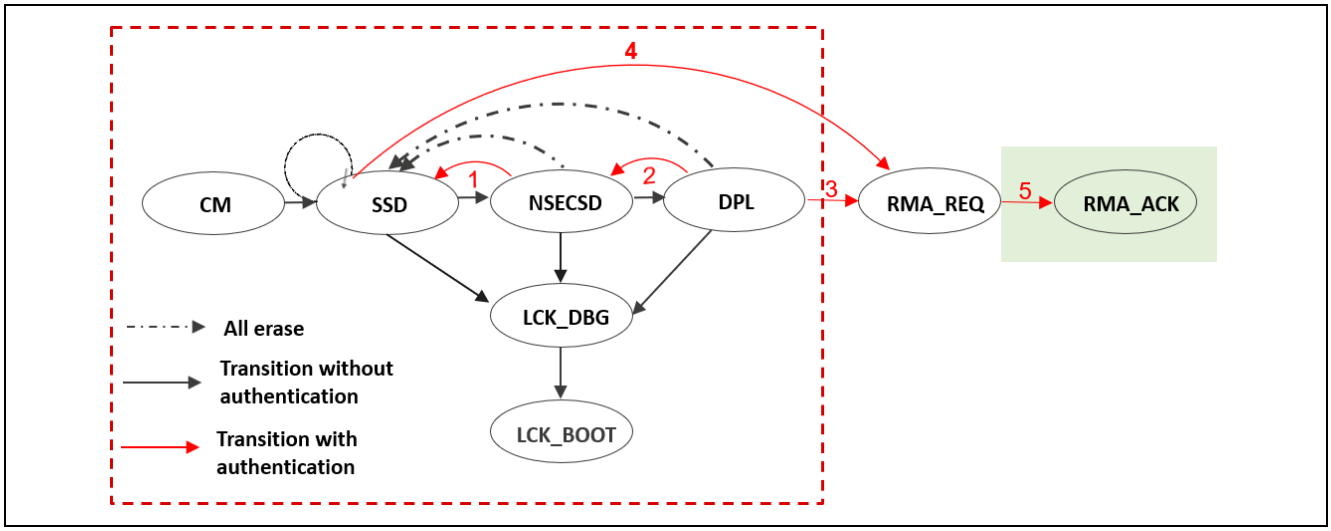


Figure 1. Device Lifecycle Management

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.

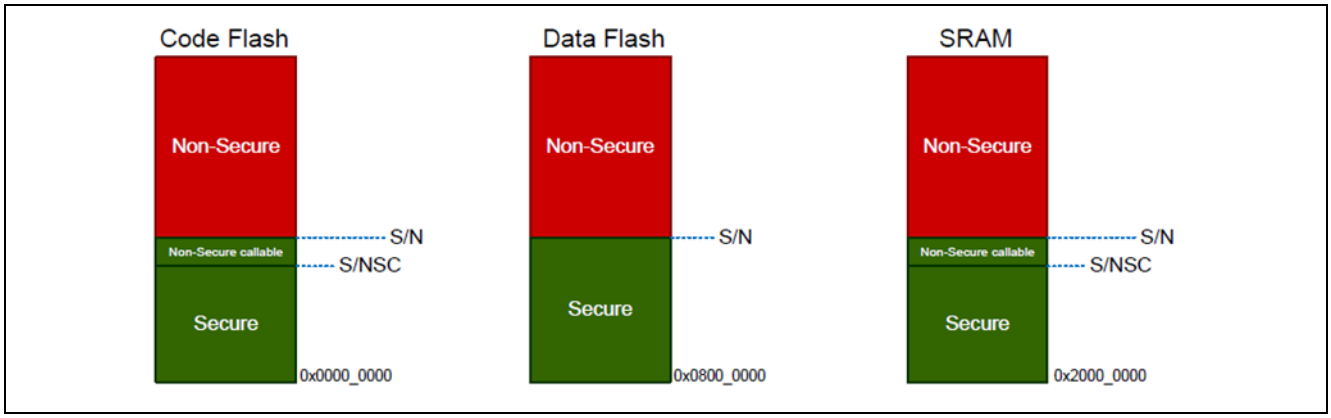


Figure 2. TrustZone Configurations Example using RA6M4

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

Function Name	Signal	IO	Comments
Power supply	VCC	Input	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.
	VCL/VCL0	I/O	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.
	VSS	Input	Ground pin. Connect it to the system power supply (0 V).

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 requirement on the main oscillator frequency. Table 4 shows the requirement for the RA6M4 MCU.

Table 4. Clock Source for Boot Mode Operation

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

Table 5. Clock Signals

Function Name	Signal	IO	Comments
Clock	XTAL	Output	Pins for a crystal resonator. An external clock signal can be input through the EXTAL pin.
	EXTAL	Input	

* 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

Function Name	Signal	IO	Comments
Operating mode control	MD	Input	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.
MCU Reset control	RES	Input	Reset signal input pin. The MCU enters the reset state when the RES signal goes low.

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

Function Name	Signal	IO	Comments
SCI (channel 9)	RXD9	Input	RA4M2/3 RA6M4/5, RA4E1, RA6E1: P110 RA6T2: PA15
	TXD9	Output	RA4M2/3, RA6M4/5, RA4E1, RA6E1: P109 RA6T2: PB03

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

Function Name	Signal	IO	Comments
USB Full Speed	VCC_USB	Input	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.
	VSS_USB	Input	USB Full-speed ground pin. Connect this pin to VSS.
	USB_DP	I/O	D+ pin of the USB on-chip transceiver. Connect this pin to the D+ pin of the USB bus.
	USB_DM	I/O	D- pin of the USB on-chip transceiver. Connect this pin to the D- pin of the USB bus.
	USB_VBUS (P407)	Input	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.
	USB_VBUSEN	Output	VBUS (5V) supply enable signal for external power supply chip.

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.

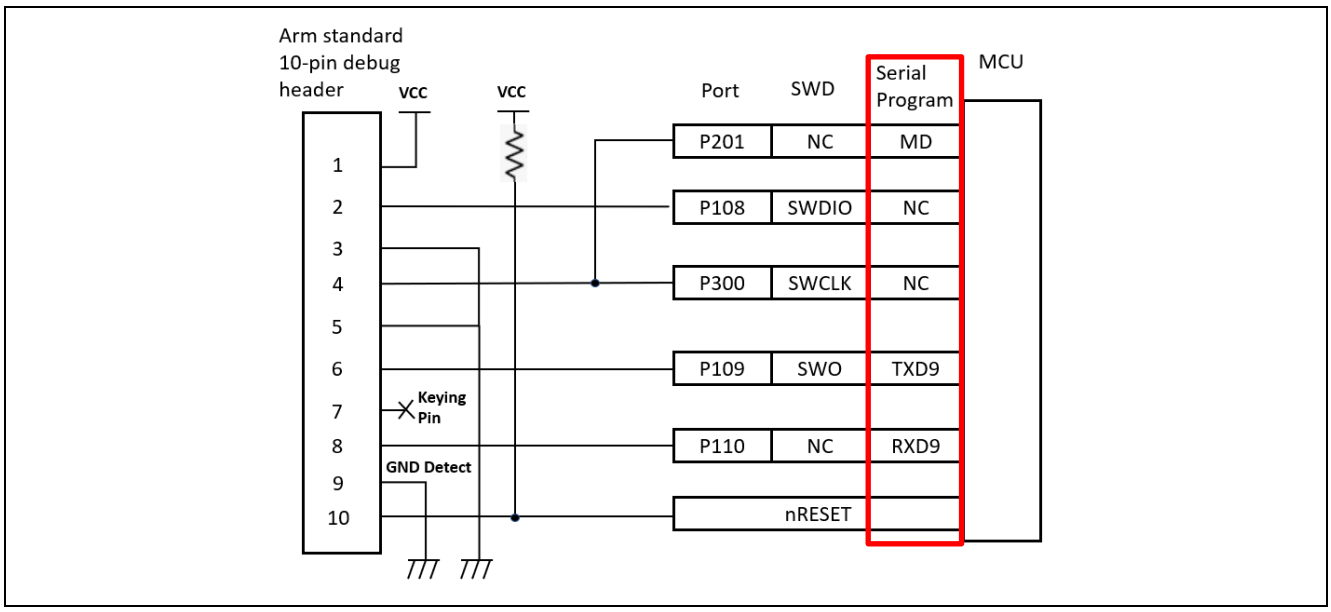


Figure 3. Access the Boot Mode through the Multi-emulator Interface Header

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).

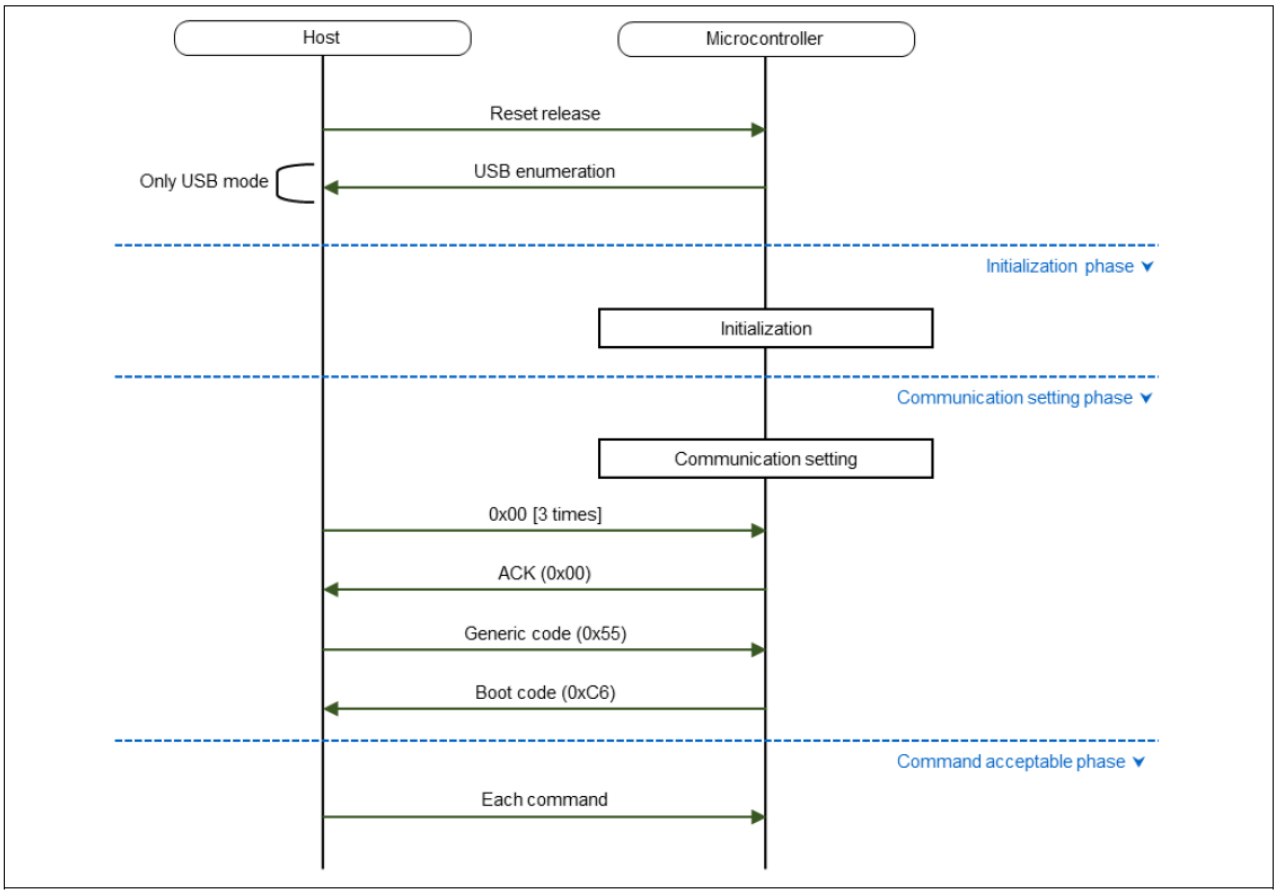
3. Connecting to Boot Mode

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



* 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 4. Boot Mode Operational Phases

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.

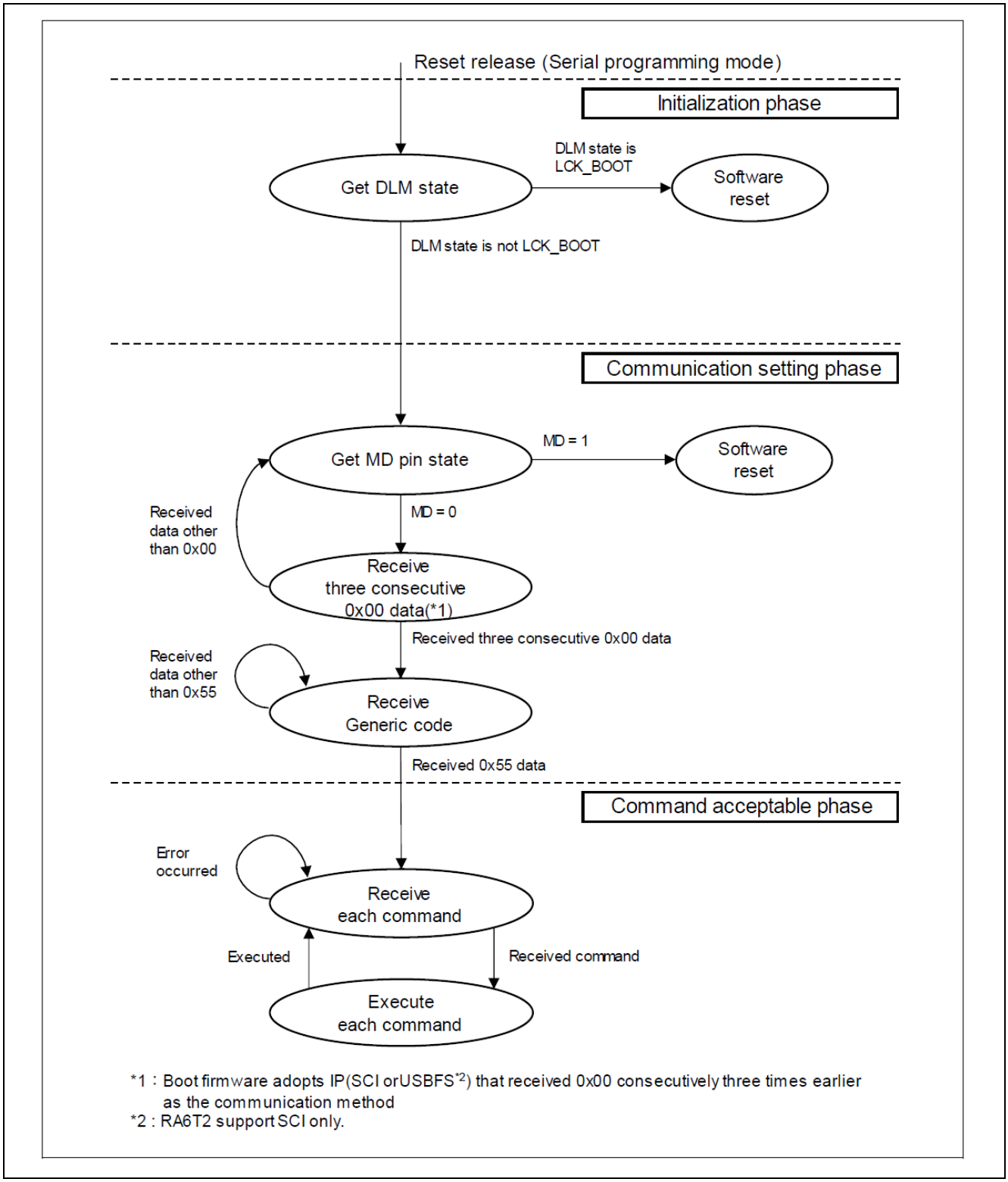


Figure 5. Command Execution State Transition Diagram

3.2 Initialization Phase

Production programming tools do not need to carry out any actions during the Initilization 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.

3.2.1 Serial Settings

When using serial communication to boot mode, the following default settings are in use:

Bit rate	9600 bps (minimum, until the baud rate setting command)
Data length	8 bits (LSB first)
Parity bit	None
Stop bit	1 bit

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:

Transfer rate	12 Mbps (USB 2.0 Full Speed)
Device class	Communication Device Class (CDC) SubClass: Abstract Control Mode (ACM) Protocol: Common AT commands
Vender ID	0x045B
Product ID	0x0261
Transfer mode	Control (in/out) Bulk (in, out) Interrupt (in)

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.

```
# ================================
# This routine demonstrates the communication setting phase handling
def communication_setting():
    loopcount = 20

    print ('Sending three 0x00 to target to start Communication Setting Phase')
    while loopcount != 0:
        # Try this a few times
        ser.write(b'\x00')
        # This sleeping time can change based on what source clock is used for
        # the MCU. Reference the Secttion "Communication setting phase" in
        # R01AN5562 for more details".
        time.sleep(SHORT_DELAY)
        ser.write(b'\x00')
        time.sleep(SHORT_DELAY)
        ser.write(b'\x00')
        time.sleep(SHORT_DELAY)
        h = ser.read()
        if h == b'\x00':
            print ("Success: ACK received")
            break
        loopcount -= 1
        time.sleep(SHORT_DELAY)
    return loopcount
```

Figure 6. Communication Setting – Making the initial connection

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.

```
        print ('Sending GENERIC code to target : 0x55 ')
        ser.write(b'\x55')

        print ('Checking for the Boot code sent back from target')
        time.sleep(SHORT_DELAY)
        h = ser.read()
        print ("Received :" + h.hex())


        match h:
            case b'\xc6':
                print ("CM33 boot code received")
                bootcode = 0xc6
            case b'\xc3':
                print ("CM4/CM23 boot code received")
                print ("They are not supported by this demonstration")
                terminate_execution()
            case _:
                print ("*** ERROR : Unknown code received, closing down")
                terminate_execution()
```

Figure 7. Completing the connection - retrieving the boot code

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

Symbol	Size	Value	Description
SOH	1 byte	0x01	Start of command packet.
LNH	1 byte	-	Packet length (length of 'CMD + command information') [High].
LNL	1 byte	-	Packet length (length of 'CMD + command information') [Low].
CMD	1 byte	-	Command code, as described in section 4.1.3.
Command information	0 to 255 bytes	-	Command information. For example: • For write command: Start/End address. • For erase command: Start/End address. • For DLM state transit command: Source/Destination DLM state code.
SUM	1 byte	-	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

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

Symbol	Size	Value	Description
SOD	1 byte	0x81	Start of data packet.
LNH	1 byte	-	Packet length (length of 'RES + Data') [High] (*1).
LNL	1 byte	-	Packet length (length of 'RES + Data') [Low] (*1).
RES	1 byte	-	Refer to R01AN5562 section "RES: Response code" for all the supported response codes.
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.

Table 11. Boot Command Code Summary

Command Value	Device		Name	Comment
	RA4M2/M3, RA6M4/M5, RA6T2	RA4E1, RA6E1		
0x3B	○	○	Area information request command	Not demonstrated
0x30	○	○	Authentication command	Not demonstrated
0x34	○	○	Baud rate setting command	Not demonstrated
0x4F	○	○	Boundary request command	Demonstrated (see section 5.4.3)
0x4E	○	○	Boundary setting command	Demonstrated (see section 5.4.4)
0x18	○	○	CRC command	Cyclic Redundancy Check of target area
0x2C	○	○	DLM state request command	Demonstrated (see section 5.6.1)
0x71	○	○	DLM state transit command	Demonstrated (see section 5.6.2)
0x12	○	○	Erase command	Not demonstrated
0x50	○	○	Initialize command	Demonstrated (see section 5.3.1)
0x00	○	○	Inquiry command	Demonstrated (see section 5.2)
0x28	○	○	Key setting command	Not demonstrated
0x29	○	○	Key verify command	Not demonstrated
0x2A	○		User key setting command	Not demonstrated
0x2B	○		User key verify command	Not demonstrated
0x52	○	○	Parameter request command	Demonstrated (see section 5.3.2)
0x51	○	○	Parameter setting command	Demonstrated (see section 5.3.3)
0x15	○	○	Read command	Not demonstrated
0x3A	○	○	Signature request command	Not demonstrated
0x13	○	○	Write command	Not demonstrated

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 12. Boot Mode Command Use Cases

Section in this application note	Corresponding section in R01AN5562
Boot Mode Inquiry Command	Inquiry command
Initialize MCU Command	Initialize command
Disable Initialize Command	Parameter setting command
Check Whether Initialize Command is Disabled	Parameter request command
Inject DLM keys	Key setting command
Verify DLM keys	Key verifying command
DLM State Request	DLM state request command
DLM State Transition	DLM state transition command
TrustZone Boundary Setting Command	Boundary setting command
TrustZone Boundary Request Command	Boundary request command

5.2 Inquiry Command

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

6.14.2.1 Command packet

SOH	(1 byte)	0x01
LNH	(10xbyte)	0x00
LNL	(1 byte)	0x01
CMD	(1 byte)	0x00 (Inquiry command)
SUM	(1 byte)	0xFF
ETX	(1 byte)	0x03

Figure 8. Inquiry Command Packet

```
# ==================================
# 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'\x01'
    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

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).

6.8.2.1 Command packet

SOH	(1 byte)	0x01	
LNH	(1 byte)	0x00	
LNL	(1 byte)	0x03	
CMD	(1 byte)	0x50 (Initialize command)	
SDLM	(1 byte)	Source DLM state code	0x02 : SSD 0x03 : NSECSD 0x04 : DPL
DDLM	(1 byte)	Destination DLM state code	0x02 : SSD
SUM	(1 byte)	Sum data	
ETX	(1 byte)	0x03	

6.8.2.2 Data Packet [status OK]

SOD	(1 byte)	0x81
LNH	(1 byte)	0x00
LNL	(1 byte)	0x0A
RES	(1 byte)	0x50 (OK)
STS	(1 byte)	0x00 (OK)
FST	(4 bytes)	0xFFFFFFFF (unused code)
ADR	(4 bytes)	0xFFFFFFFF (unused code)
SUM	(1 byte)	0xAE
ETX	(1 byte)	0x03

Figure 10. Initialize Command Packets

```
# ==================================
# The "Initialize command" clears User area, Data area, Config area,
# Boundary setting, and Key index (Wrapped key). In addition,
# the DLM state transitions to SSD from the NSECSD or DPL.

def command_initialize (current_DLM_state):

    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x03'
    #0x50: Initialie command
    CMD=b'\x50'
    # Source DLM state code
    SDLM=current_DLM_state
    # Destination DLM state code
    DDLM=b'\x02'
    SUM=calc_sum(LNH + LNL + CMD + SDLM + DDLM)
    EXT=b'\x03'
    command = SOH + LNH + LNL + CMD + SDLM + DDLM + SUM + EXT

    print ("Sending MCU Initialize command:")
    print_bytes_hex (command)

    ser.write(command)
    time.sleep(INITIALIZE_DELAY)

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

    if RES != 0x50:
        print('Initialize -FAIL')
    else:
        print ('Initialize - SUCCESS')

    print ("\n==========================\n")
    print ("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
    print ("Warning : After MCU Initialize is run, an MCU reset is")
    print ("          required before further operations. ")
    print ("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
```

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

SOH	(1 byte)	0x01	
LNH	(1 byte)	0x00	
LNL	(1 byte)	0x02	
CMD	(1 byte)	0x52 (Parameter request command)	
PMID	(1 byte)	Parameter ID	0x01 : Enable / Disable of the initialization
SUM	(1 byte)	Sum data	
ETX	(1 byte)	0x03	

6.13.2.2 Data packet [Parameter data]

SOD	(1 byte)	0x81	
LNH	(1 byte)	0x00	
LNL	(1 byte)	N + 1	
RES	(1 byte)	0x52 (OK)	
PRMT	(N byte)	Parameter data	[PMID = 0x01] 0x00 : Initialization is disabled 0x07 : Initialization is enabled
SUM	(1 byte)	Sum data	
ETX	(1 byte)	0x03	

N = 1 to 16

Figure 12. Command Packet: Check whether Initialize Command is Disabled

```
# ==================================
# Reference section "Parameter request command"
# for the definition of the parameters"
def command_check_whether_initialize_is_disabled ():

    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x02'
    #0x51: Parameter request command
    CMD=b'\x52'
    # Parameter ID: enable/disable of the Initialization
    PMID=b'\x01'
    SUM=calc_sum(LNH + LNL + CMD + PMID)
    EXT=b'\x03'
    command = SOH + LNH + LNL + CMD + PMID + SUM + EXT

    print ("Sending MCU check whether Initialize command is
    disabled command:")
    print_bytes_hex (command)

    ser.write(command)
    time.sleep(LONG_DELAY)

    return_packet = receive_data_packet()
    #RES is the fourth byte
    RES = return_packet[3]
    if (RES == 0x52):
        #whether Initialize command is disabled is indicated by the
        fifth byte
        PRMT = return_packet[4]
        if PRMT == 0x00:
            print('Initialization is disabled')
        elif PRMT == 0x07:
            print ('Initialization is enabled')
    else:
        print ('Check Whether Initialize Command is disabled or not
        command failed')

    return PRMT
```

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

SOH	(1 byte)	0x01	
LNH	(1 byte)	0x00	
LNL	(1 byte)	N + 2	
CMD	(1 byte)	0x51 (Parameter setting command)	
PMID	(1 byte)	Parameter ID	0x01 : Setting of disable initialization
PRMT	(N byte)	Parameter data	[PMID = 0x01] 0x00 : Disable initialization
SUM	(1 byte)	Sum data	
ETX	(1 byte)	0x03	

N = 1 to 16

6.12.2.2 Data packet [status OK]

SOD	(1 byte)	0x81
LNH	(1 byte)	0x00
LNL	(1 byte)	0x0A
RES	(1 byte)	0x51 (OK)
STS	(1 byte)	0x00 (OK)
FST	(4 bytes)	0xFFFFFFFF (unused code)
ADR	(4 bytes)	0xFFFFFFFF (unused code)
SUM	(1 byte)	Sum data
ETX	(1 byte)	0x03

Figure 14. Disable the 'Initialize Command' Command Packet

```
# ==================================
# The "Initialize command" can be disabled using the
# "Parameter setting command".
#  !!!!! WARNING !!!!!
#  This step is non-reversable, so invoke with caution!
def command_disable_initialize ():

    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x03'
    #0x51: Parameter setting command
    CMD=b'\x51'
    # Parameter ID
    PMID=b'\x01'
    # Parameter Data 0x00 is for disable Initialze
    PRMT=b'\x00'
    SUM=calc_sum(LNH + LNL + CMD + PMID + PRMT)
    EXT=b'\x03'
    command = SOH + LNH + LNL + CMD + PMID + PRMT + 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('\nDisable Initialize command - FAIL')
    else:
        print ('Disable Initialize command - SUCCESS')
```

Figure 15. Example Code: Disable the ‘Initialize Command’

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 e^2 studio, or a RASC-generated EWARM / MDK project. Table 13 shows how to find the .rpd file based on the IDE.

Table 13. The . rpd File Location Based on IDE

IDE	Location of the .rpd file
e^2 studio	Secure project root folder: <secure_project_name>\Debug\<secure_project_name>. rpd
EWARM	Secure project root folder: <secure_project_name>\Debug\exe\<secure_project_name>.rpd
MDK	Secure project root folder: <secure_project_name>\Objects\<secure_project_name>.rpd

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

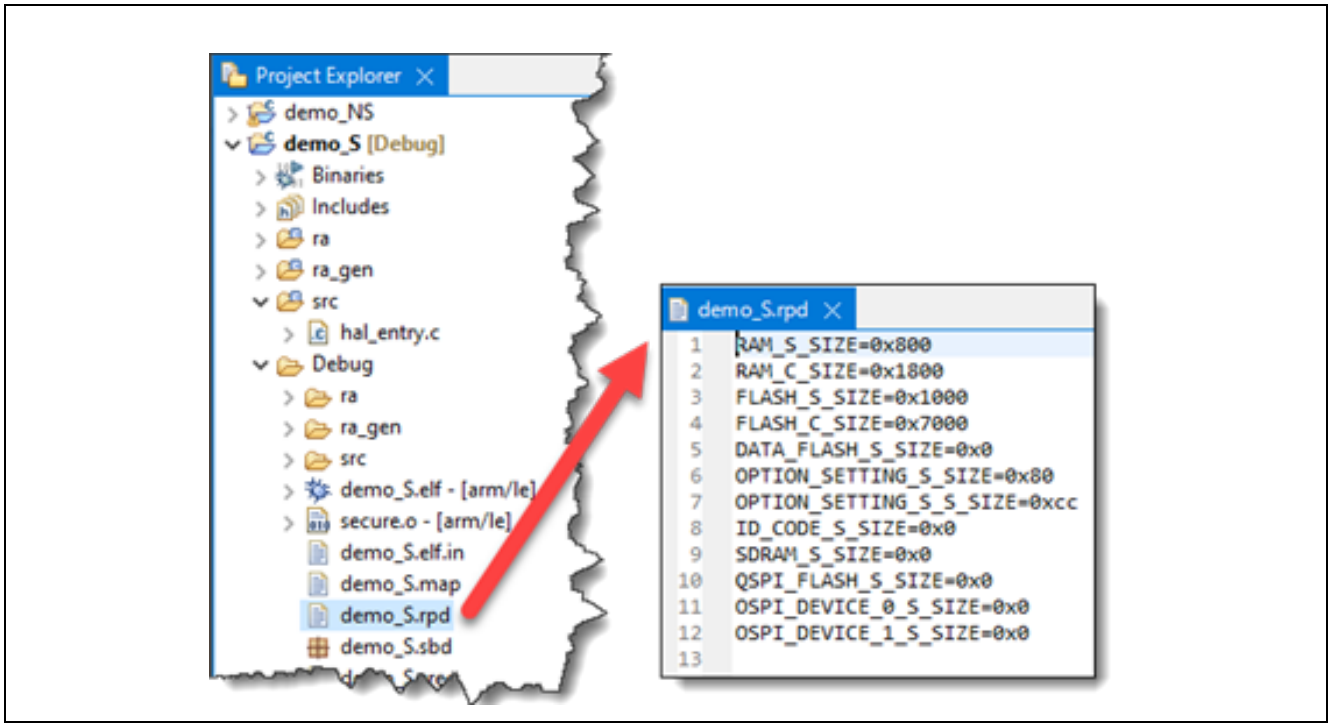


Figure 16. Obtain the IDAU Region Size using EWARM

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 needs 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	(1 byte)	0x01
LNH	(1 byte)	0x00
LNL	(1 byte)	0x01
CMD	(1 byte)	0x4F (Boundary request command)
SUM	(1 byte)	0xB0
ETX	(1 byte)	0x03

6.11.2.2 Data packet [Boundary setting data]

SOD	(1 byte)	0x81	
LNH	(1 byte)	0x00	
LNL	(1 byte)	0x0B	
RES	(1 byte)	0x4F (OK)	
CFS1	(2 bytes)	Size of code flash secure region (without NSC) [KB]	(ex.) 0x0100 -> 0x01, 0x00 (256 KB)
CFS2	(2 bytes)	Size of code flash secure region [KB]	(ex.) 0x0100 -> 0x01, 0x00 (256 KB)
DFS1	(2 bytes)	Size of data flash secure region [KB]	(ex.) 0x0004 -> 0x00, 0x04 (4 KB)
SRS1	(2 bytes)	Size of SRAM secure region (without NSC) [KB]	(ex.) 0x0040 -> 0x00, 0x40 (64 KB)
SRS2	(2 bytes)	Size of SRAM secure region [KB]	(ex.) 0x0040 -> 0x00, 0x40 (64 KB)
SUM	(1 byte)	Sum data	
ETX	(1 byte)	0x03	

NSC: Non-secure callable regions

Figure 17. Command Packet for Reading IDAU Region Setup

```
# ==================================
# 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'
    EXT=b'\x03'
    command = SOH + LNH + LNL + CMD + SUM + EXT

    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 ("-----------------------------\n")
        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.

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

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.

```
# ==================================
# This routine demonstrates "TrustZone boundary setting command".
def command_setup_trustzone_boundary():
    print ("Setting Up the TrustZone boundary:")
    print ("-----------------------------\n")
    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x0B'
    CMD=b'\x4E' #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 = SOH + 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

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.

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.

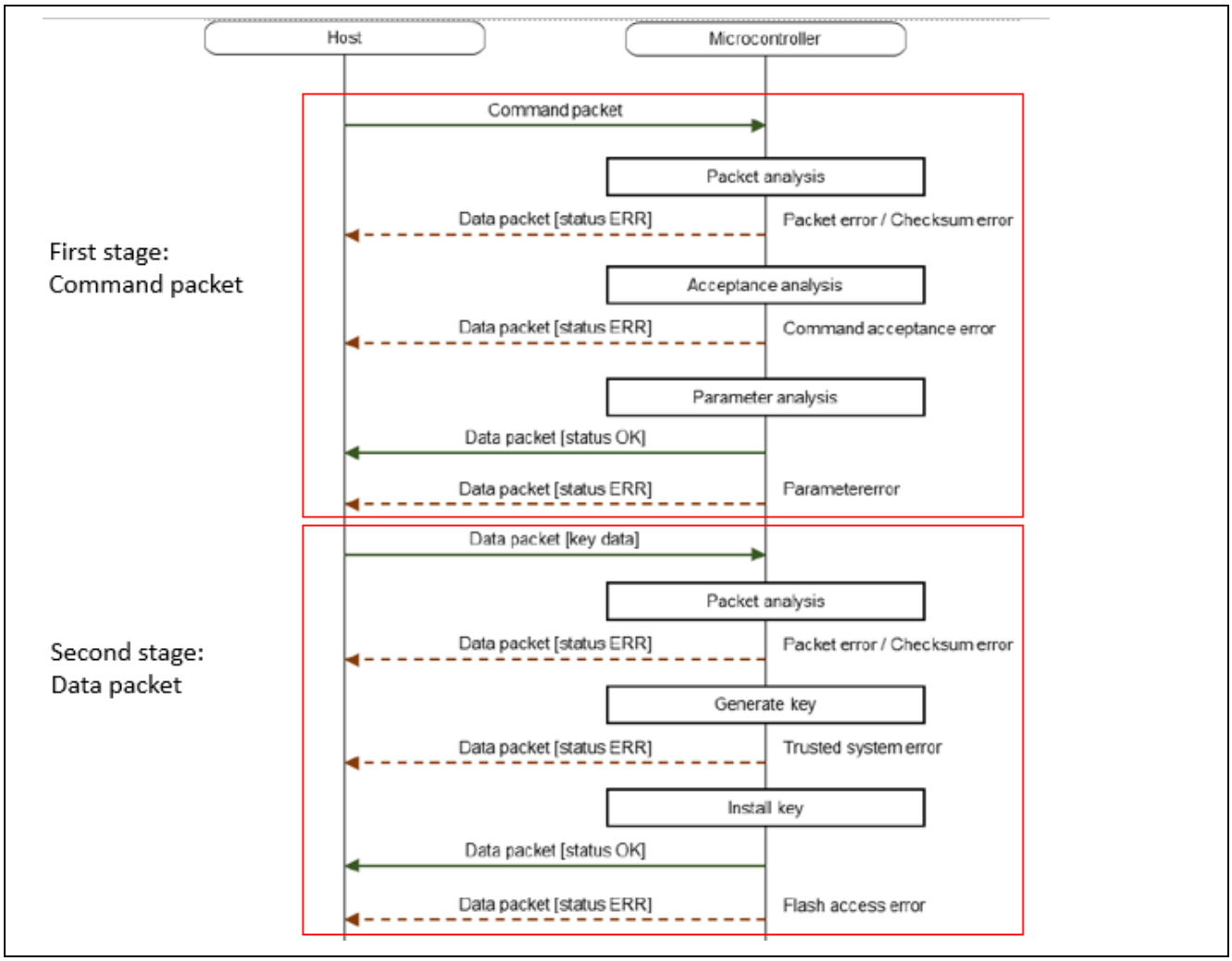


Figure 21. DLM Key Injection Flow

SOH	(1 byte)	0x01	
LNH	(1 byte)	0x00	
LNL	(1 byte)	0x02	
CMD	(1 byte)	0x28 (Key setting command)	
KYTY	(1 byte)	Key type	0x01 : SECDBG_KEY 0x02 : NONSECDBG_KEY 0x03 : RMA_KEY
SUM	(1 byte)	Sum data	
ETX	(1 byte)	0x03	

Figure 22. DLM Key Injection Command Packet

<table>
<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>0x51</td><td></td></tr>
<tr><td>RES</td><td>(1 byte)</td><td>0x28 (OK)</td><td></td></tr>
<tr><td>ESKY</td><td>(32 bytes)</td><td>Session key (W-UFPK)</td><td>(ex.) 0x01234567_89AB ... 2233_44556677
-> 0x01, 0x23, 0x45, ... , 0x55, 0x66, 0x77</td></tr>
<tr><td>IVEC</td><td>(16 bytes)</td><td>Initialization vector</td><td>(ex.) 0x01234567_89AB ... 2233_44556677
-> 0x01, 23, 0x45, ... , 0x55, 0x66, 0x77</td></tr>
<tr><td>EOKY</td><td>(32 bytes)</td><td>Install data (encrypted key | MAC)</td><td>Encrypted key (0-15 bytes) + MAC (16-31 bytes)
(ex.) If install data is as follows, the host must send EOKY in the order shown in the table that follows:

Install data:
<table>
<tr><th colspan="8">Encrypted key</th></tr>
<tr><td>00</td><td>01</td><td>02</td><td>03</td><td>04</td><td>05</td><td>06</td><td>07</td></tr>
<tr><td>08</td><td>09</td><td>0A</td><td>0B</td><td>0C</td><td>0D</td><td>0E</td><td>0F</td></tr>
<tr><th colspan="8">MAC</th></tr>
<tr><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td></tr>
<tr><td>18</td><td>19</td><td>1A</td><td>1B</td><td>1C</td><td>1D</td><td>1E</td><td>1F</td></tr>
</table>
Order of sending EOKY:
<table>
<tr><th>1st</th><th>2nd</th><th>3rd</th><th>4th</th><th>5th</th><th>6th</th><th>7th</th><th>8th</th></tr>
<tr><td>00</td><td>01</td><td>02</td><td>03</td><td>04</td><td>05</td><td>06</td><td>07</td></tr>
<tr><th>9th</th><th>10th</th><th>11th</th><th>12th</th><th>13th</th><th>14th</th><th>15th</th><th>16th</th></tr>
<tr><td>08</td><td>09</td><td>0A</td><td>0B</td><td>0C</td><td>0D</td><td>0E</td><td>0F</td></tr>
<tr><th>17th</th><th>18th</th><th>19th</th><th>20th</th><th>21st</th><th>22nd</th><th>23rd</th><th>24th</th></tr>
<tr><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td></tr>
<tr><th>25th</th><th>26th</th><th>27th</th><th>28th</th><th>29th</th><th>30th</th><th>31st</th><th>32nd</th></tr>
<tr><td>18</td><td>19</td><td>1A</td><td>1B</td><td>1C</td><td>1D</td><td>1E</td><td>1F</td></tr>
</table></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>
</table>

Figure 23. DLM Key Injection Data 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.

Name	Type	Size	Description
Magic code	Char[4]	4 Bytes	"REK1"
Suite Version	Integer	4 Bytes	Data format version. Currently Must be 1.
Reserved	Byte[7]	7 Bytes	Reserved. Must be 0.
Key Type	Byte	1 Bytes	[For DLM key] Must be 0. [For user key] Keytype value constant (Refer to 4.5.2 **keytype** Options)
Encrypted Key Size	Integer	4 Bytes	Size of "Encrypted Key" (= N bytes)
W-UFPK	Byte[36]	36 Bytes	Value of the W-UFPK file sent from the Renesas DLM server The first 4 bytes are Shared Key Number The remaining 32 bytes are the WUFPK value
Initialization Vector	Byte[16]	16 Bytes	Initialization vector value used to Wrap user key.
Encrypted Key	Byte[N]	N Bytes	User key encrypted with UFPK value + MAC value
Data CRC	Byte[4]	4 Bytes	Calculated CRC for all data except this CRC data. Initial Value = 0xFFFFFFFF Magic number = 0x04C11DB7

Figure 24. DLM Key Data Structure

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 SECDBG and NSECDBG key
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 with unknown DLM key type
        return UNKNOWN_KEY
    elif (dlm_key_type == b'\x01'):
        text_file = open("./dlm_keys/SECDBG.rkey", "r")

    elif (dlm_key_type == b'\x02'):
        text_file = open("./dlm_keys/NON-SECDBG.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('\nDLM key injection command packet - SUCCESS')
            print ("\n---------------------------\n")
            # Start the data packet stage
```

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

```
        # Start the data packet stage
        SOD=b'\x81' #start of data packet
        LNH=b'\x00'
        LNL=b'\x51' #81 byte: one byte (RES) + w-ufpk (32byte) + initialize vector (16 byte)
                    #+ encrypted dlm key and mac (32 byte)
        RES=b'\x28' #OK byte
        SUM=calc_sum(LNH+LNL+RES + message_bytes [24:56] + message_bytes [56:72] + \
                    message_bytes [72:72+encrypted_key_size])
        ETX = b'\x03'
        command = SOD + LNH + LNL + RES + message_bytes [24:56]  + message_bytes [56:72] + \
                    message_bytes [72:72+encrypted_key_size] + SUM + ETX
        print ("Sending dlm key injection data packet: ")
        print_bytes_hex (command)
        ser.write(command)
        time.sleep(LONG_DELAY)
        print ("\n----------------------------")
        # Acquire the response of the data packet stage
        print ("\nread the response for the data packet command \n")
        return_packet = receive_data_packet()
        # the RES byte is the fourth index
        RES = return_packet[3]
        if RES != 0x28:
            print('DLM key injection data packet failed')
        else:
            # the STS is the fifth index
            STS = return_packet[4]
            if STS != 0x00:
                print('Injecting DLM key failed')
            elif KYTY == b'\x01':
                print("Injecting SECDBG key is successful")
            elif KYTY == b'\x02':
                print('Injecting NSECDBG key is successful')
return RES
```

Figure 26. Example Code: DLM Key Injection – Part 2

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.

SOH	(1 byte)	0x01	
LNH	(1 byte)	0x00	
LNL	(1 byte)	0x02	
CMD	(1 byte)	0x29 (Key verify command)	
KYTY	(1 byte)	Key type	0x01: SECDBG_KEY 0x02: NONSECDBG_KEY 0x03: RMA_KEY
SUM	(1 byte)	Sum data	
ETX	(1 byte)	0x03	

Figure 27. DLM Key Verify Command Packet

```
# ================================
# This routine demonstrates "DLM Key verification command".
# The supported key types are SECDBG and NSECDBG key
def command_verify_dlm_key(dlm_key_type):
    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x02'
    CMD=b'\x29' #DLM Key verify command
    KYTY = dlm_key_type #DLM Key type

    SUM=calc_sum(LNH+LNL+CMD + KYTY)
    ETX=b'\x03'
    command = SOH + LNH + LNL + CMD + KYTY + SUM + ETX
    if KYTY == b'\x01':
        print ("Sending DLM SECDBG key verify command:")
    elif KYTY == b'\x02':
        print ("Sending DLM NSECDBG key verify command:")
    else :
        print ("Unsupported DLM Key type")
    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 != 0x29:
        print(' DLM key verification - FAIL')
    else:
        # the STS is the fifth index
        STS = return_packet[4]
        if STS != 0x00:
            print('Verifying injected DLM key failed')
        elif KYTY == b'\x01':
            print ("Verifying injected SECDBG key is successful")
        elif KYTY == b'\x02':
            print('Verifying injected NSECDBG key is successful')
    return RES
```

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

SOH	(1 byte)	0x01
LNH	(1 byte)	0x00
LNL	(1 byte)	0x01
CMD	(1 byte)	0x2C (DLM state request command)
SUM	(1 byte)	0xD3
ETX	(1 byte)	0x03

6.9.2.2 Data packet [DLM state]

SOD	(1 byte)	0x81
LNH	(1 byte)	0x00
LNL	(1 byte)	0x02
RES	(1 byte)	0x2C (OK)
DLM	(1 byte)	DLM state code
SUM	(1 byte)	Sum data
ETX	(1 byte)	0x03

Figure 29. DLM State Request Command Packet

```
# ================================
# This routine demonstrates the "DLM state request command".
# The command execution result and the DLM state are returned.
def command_DLMstateRequest():
    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x01'
    CMD=b'\x2c'          # DLM state request command code
    SUM=b'\xd3'
    EXT=b'\x03'
    command = SOH + LNH + LNL + CMD + SUM + EXT

    print ("Sending DLM State Request command:")
    print_bytes_hex (command)

    ser.write(command)
    time.sleep(SHORT_DELAY)
    #acquire the response
    return_packet = receive_data_packet()
    # the RES byte is the fourth index
    RES = return_packet[3]
    if RES != 0x2C:
        print('Read DLM state - FAIL')
        DLM = b'\xFF'
    else:
        # the DLM byte is the fifth index
        DLM = return_packet[4]
        match DLM:
            case 0x01:
                print('Current DLM state is Chip Manufacturing,')
                print('Use the DLM state transition command to transition to SSD state!')
            case 0x02:
                print('Current DLM state is SSD.')
            case 0x03:
                print('Current DLM state is NSECSD')
            case 0x04:
                print('Current DLM state is DPL')
            case 0x05:
                print('Current DLM state is LCK_DBG')
            case 0x06:
                print('Current DLM state is LCK_BOOT')
            case 0x07:
                print('Current DLM state is RMA_REQ')
            case 0x08:
                print('Current DLM state is RMA_ACK')
            case _:
                print('Unknown DLM state')
    return RES, DLM
```

Figure 30. DLM State Request Command Example Code

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.

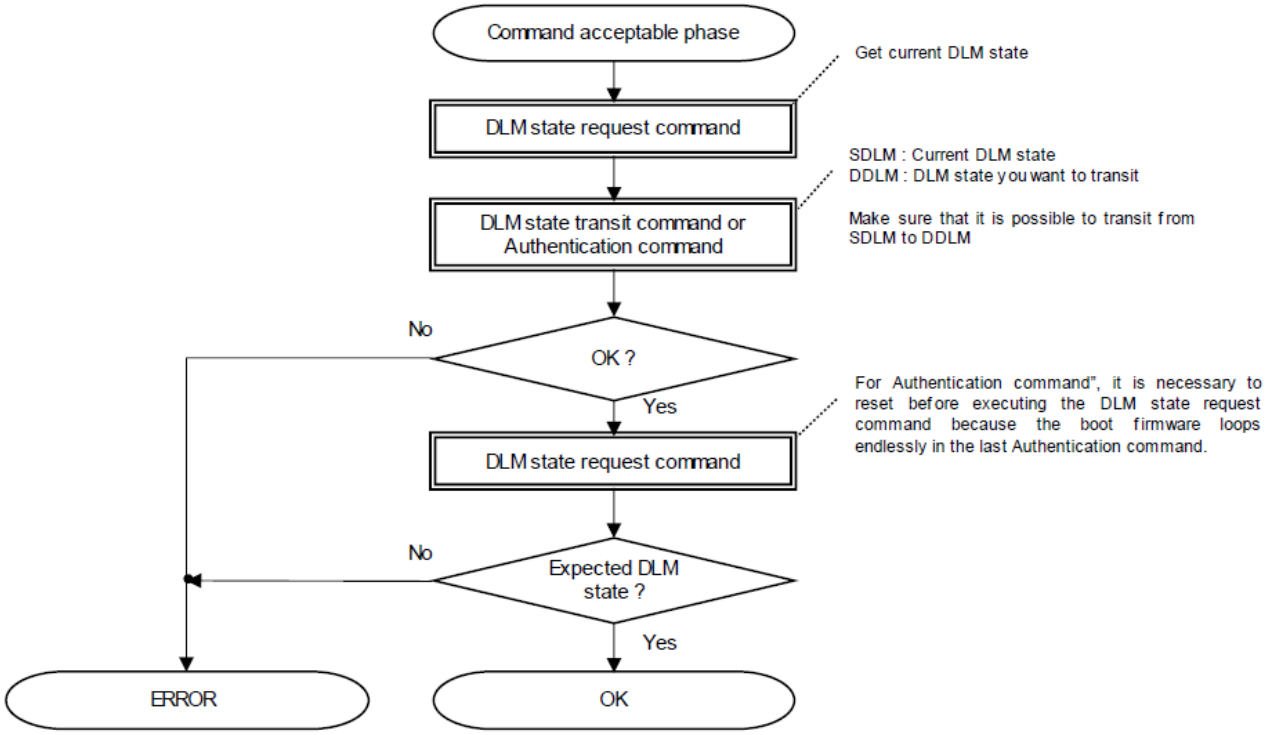


Figure 31. Recommended Flow for Performing DLM State Transition

6.2.2.1 Command packet

SOH	(1 byte)	0x01
LNH	(1 byte)	0x00
LNL	(1 byte)	0x03
CMD	(1 byte)	0x71 (DLM state transit command)
SDLM	(1 byte)	Source DLM state code
DDLM	(1 byte)	Destination DLM state code
SUM	(1 byte)	Sum data
ETX	(1 byte)	0x03

6.2.2.2 Data packet [status OK]

SOD	(1 byte)	0x81
LNH	(1 byte)	0x00
LNL	(1 byte)	0x0A
RES	(1 byte)	0x71 (OK)
STS	(1 byte)	0x00 (OK)
FST	(4 bytes)	0xFFFFFFFF (unused code)
ADR	(4 bytes)	0xFFFFFFFF (unused code)
SUM	(1 byte)	0x8D
ETX	(1 byte)	0x03

Figure 32. DLM State Transition Command Packet

```
# ================================
# This routine demonstrates the "DLM state transition command".
def command_DLMstateTransition(source_DLM_state, target_DLM_state):

    RES, SDLM = command_DLMstateRequest()
    if RES != 0x2C:
        print(' Read DLM state - FAIL')
    elif SDLM != int.from_bytes(source_DLM_state,"big"):
        print(' adjust the source DLM state! ')
    else:
        if target_DLM_state != source_DLM_state and source_DLM_state != b'\x05' \
            and source_DLM_state != b'\x06' and source_DLM_state != b'\x07' \
            and source_DLM_state != b'\x08':
            SOH=b'\x01'
            LNH=b'\x00'
            LNL=b'\x03'
            #0x71: DLM State Transition Command
            CMD=b'\x71'
            #SDLM = source_DLM_state
            DDLM=target_DLM_state
            SUM=calc_sum(LNH + LNL + CMD + source_DLM_state + DDLM)
            EXT=b'\x03'
            command = SOH + LNH + LNL + CMD + source_DLM_state + DDLM + SUM + EXT
            print ("\n--------------------------\n")
            print ("Sending DLM State Transition 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 != 0x71:
                print('DLM Transition command - FAIL')
            else:
                if DDLM == b'\x02':
                    print('DLM state changed to SSD')
                elif DDLM == b'\x03':
                    print('DLM state changed to NSECSD')
                elif DDLM == b'\x04':
                    print('DLM state changed to DPL')
                # the STS is the fifth index
                STS = return_packet[4]
                if STS != 0x00:
                    print('DLM state Transition command - FAIL')
    return RES
```

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.

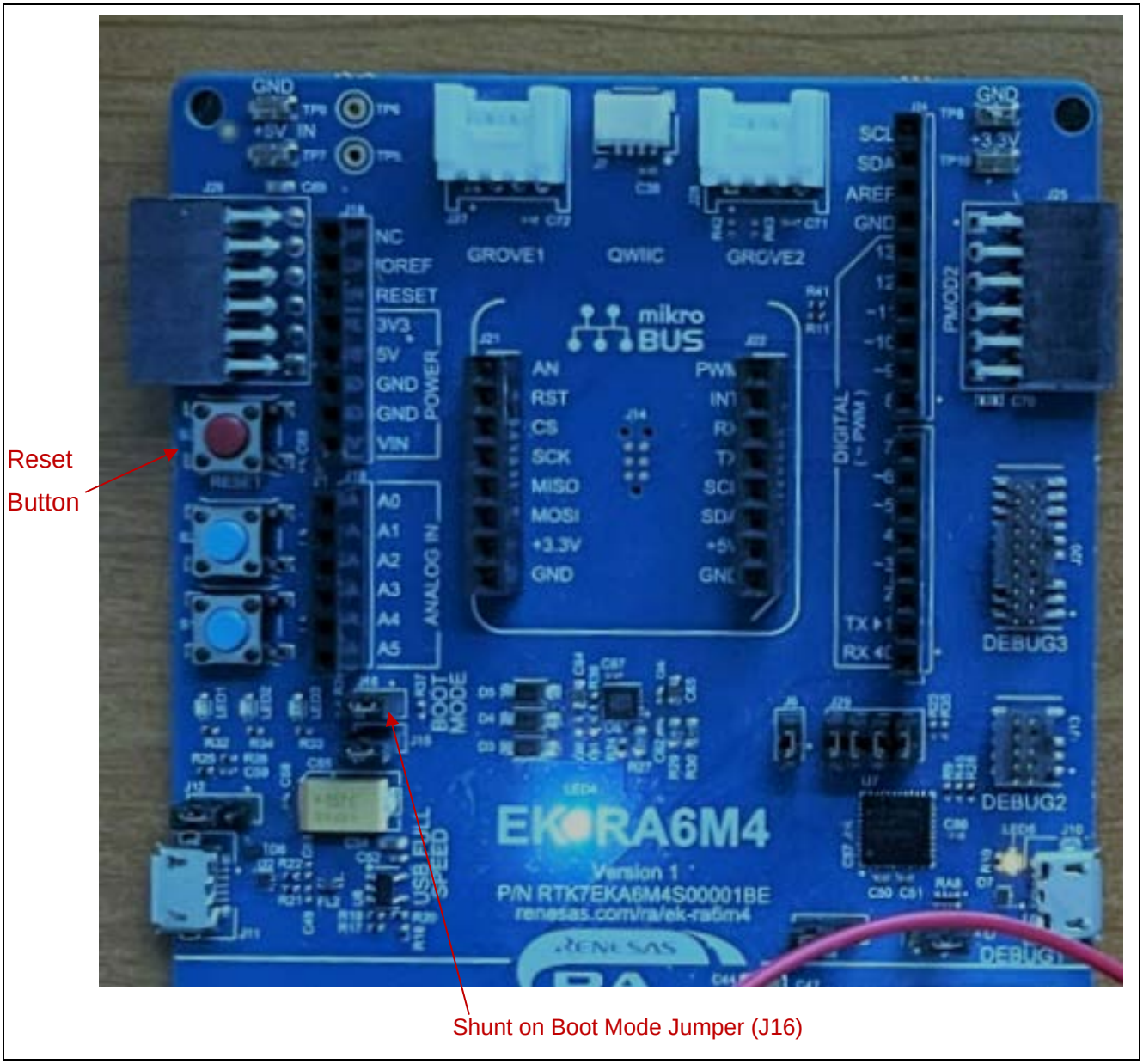


Figure 34. Shunt the MD Pin Jumper

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.

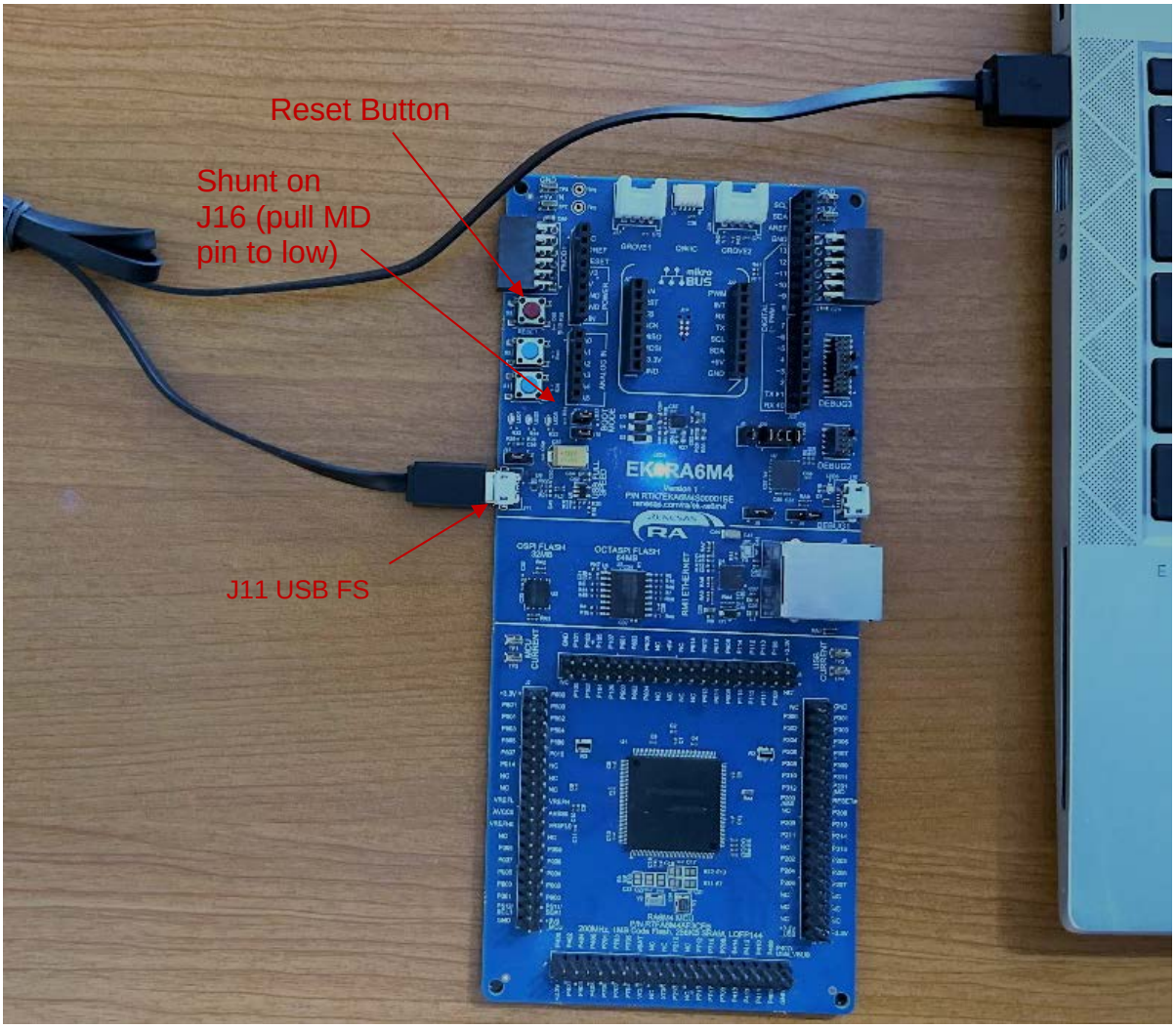


Figure 35. Hardware Setup using USB Full Speed Port

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

UART to USB Converter	EK-RA6M4
RX	J3: Pin 4 (MCU P109 (TXD9))
TX	J3: Pin 5 (MCU P110 (RXD9))
+5V (FTDI cable power output voltage. Check the voltage output on the converter used.)	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.
GND	J3: Pin 39 (MCU Ground)

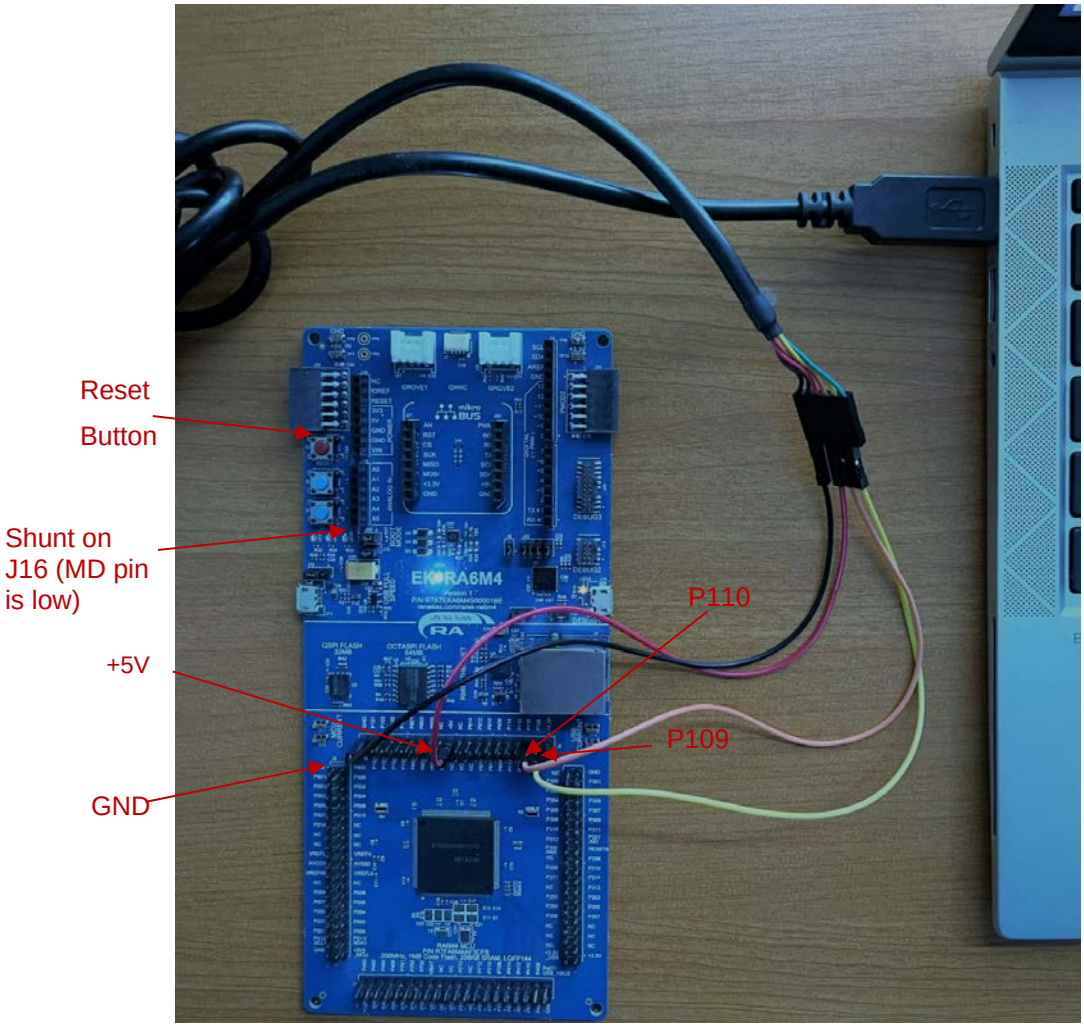


Figure 36. Hardware Setup using UART to USB Converter

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.

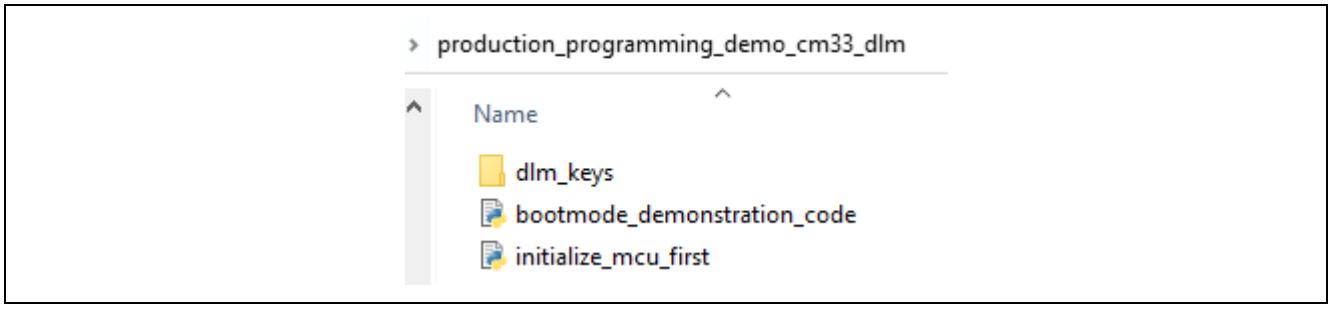


Figure 37. Python Demo Code and Sample DLM Keys

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:

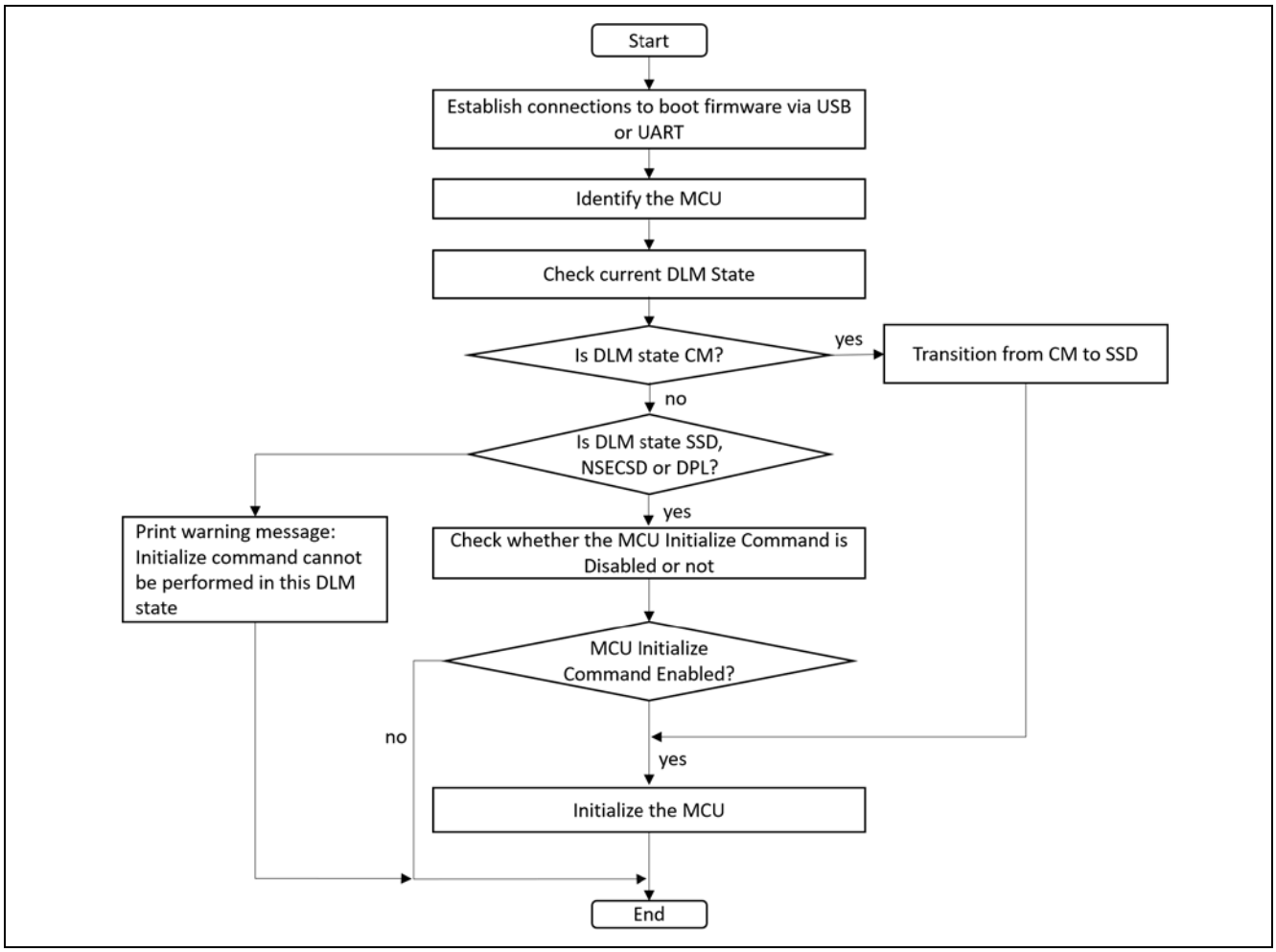


Figure 38. Operational Flow of the First Demo Code

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.

```
C:\RA Production Programming\K2_code>initialize_mcu_first.py

==================
Com port selection
==================

COM13 - RA USB Boot(CDC) (COM13)
COM3 - Intel(R) Active Management Technology - SOL (COM3)

Auto-selected COM13 for RA Boot mode USB CDC connection

===============
Opening com port
===============

Com port opened : COM13

=========================
Connecting to RA Boot Mode
=========================

Sending three 0x00 to target to start Communication Setting Phase
Success: ACK received
Sending GENERIC code to target : 0x55
Checking for the Boot code sent back from target
Received :c6
CM33 boot code received

========================
Requesting the DLM State
========================

Sending DLM State Request command:
b'\x01\x00\x01\x2c\xd3\x03'
Current DLM state is SSD.

=============================================
Checking whether Initialize command is disabled
=============================================

Sending MCU check whether Initialize command is disabled command:
b'\x01\x00\x02\x52\x01\xab\x03'
Initialization is enabled

====================
Initializing the MCU
====================

Sending MCU Initialize command:
b'\x01\x00\x03\x50\x02\x02\xa9\x03'
Initialize - SUCCESS

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Warning : After MCU Initialize is run, an MCU reset is
          required before further boot mode operations.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

=======================
Demonstration Finished.
=======================
```

Figure 39. Demonstration with USB Connection

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.

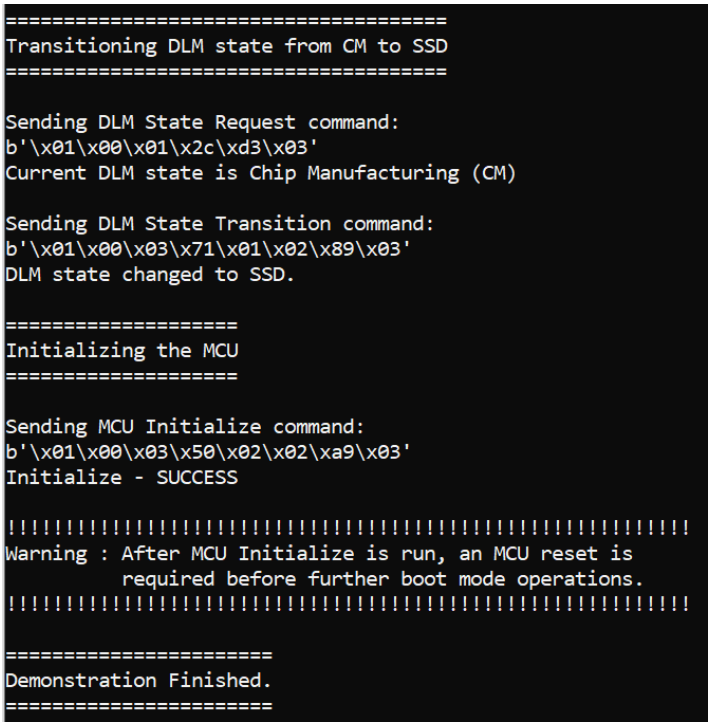


Figure 40. Running the Demo on a MCU in CM State

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.

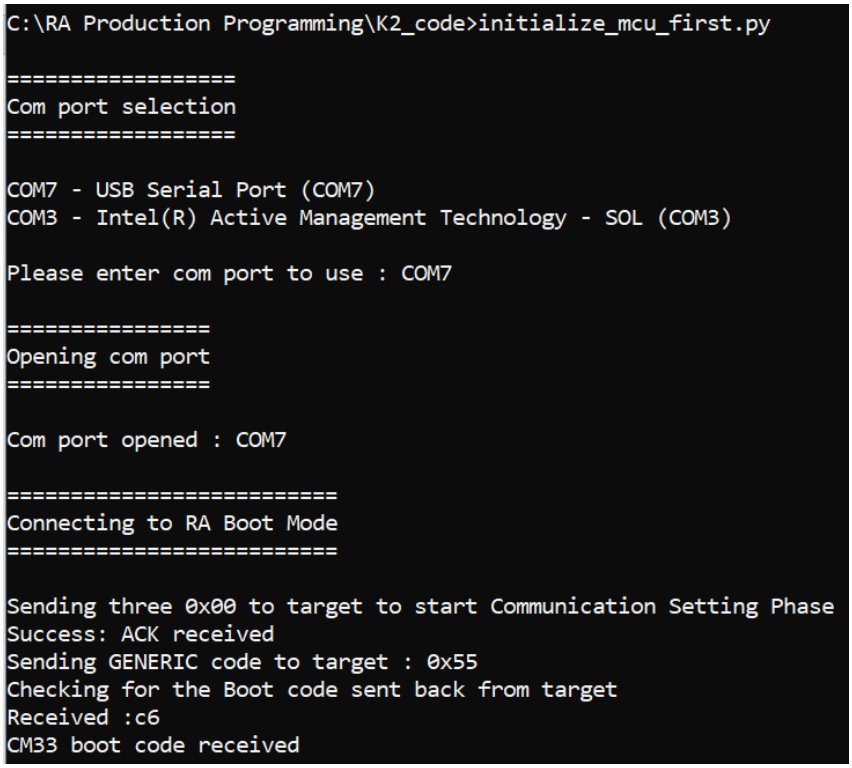


Figure 41. Demonstration with Serial Connection

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.

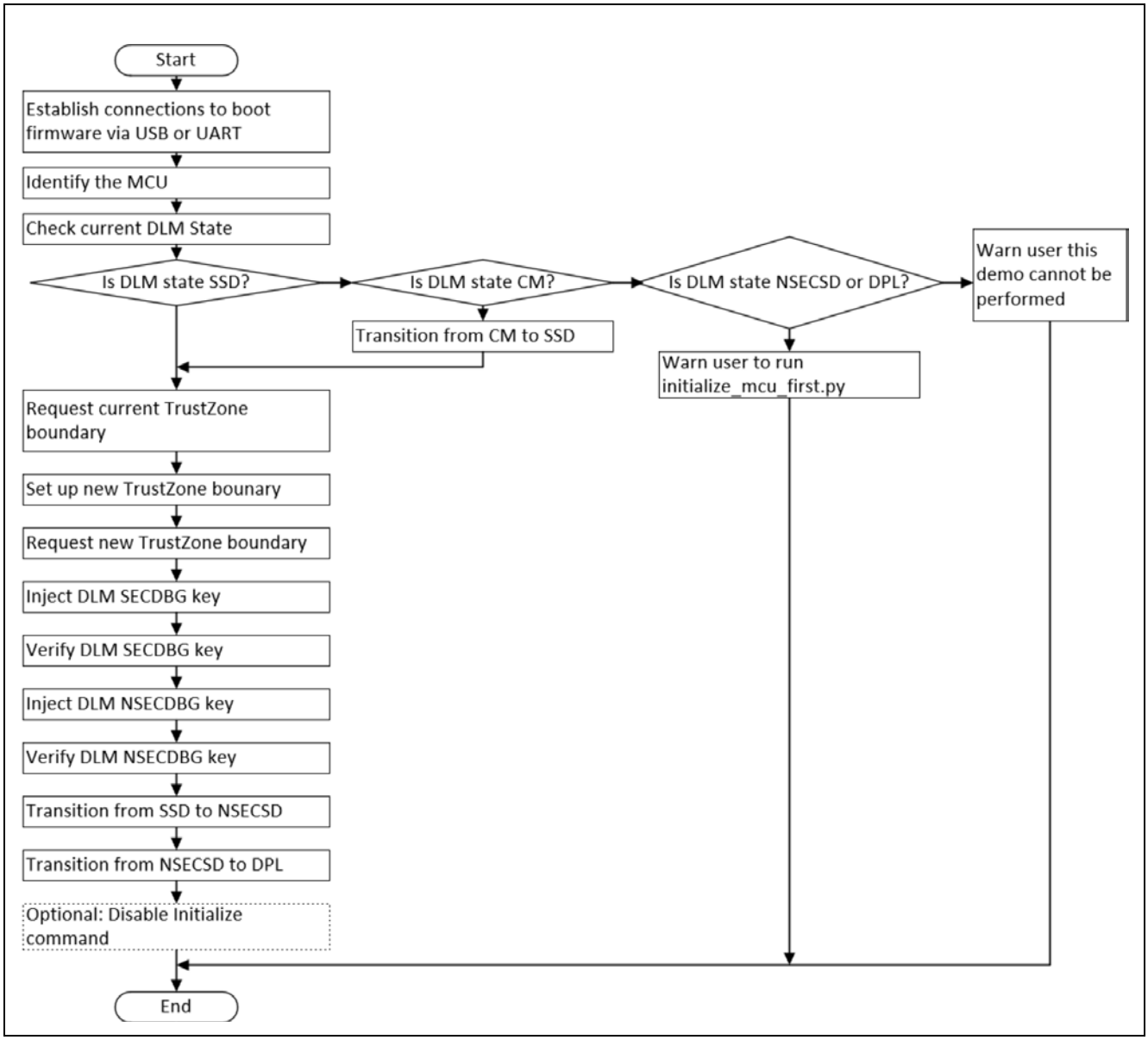


Figure 42. Operational Flow of the Second Demonstration Code

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)

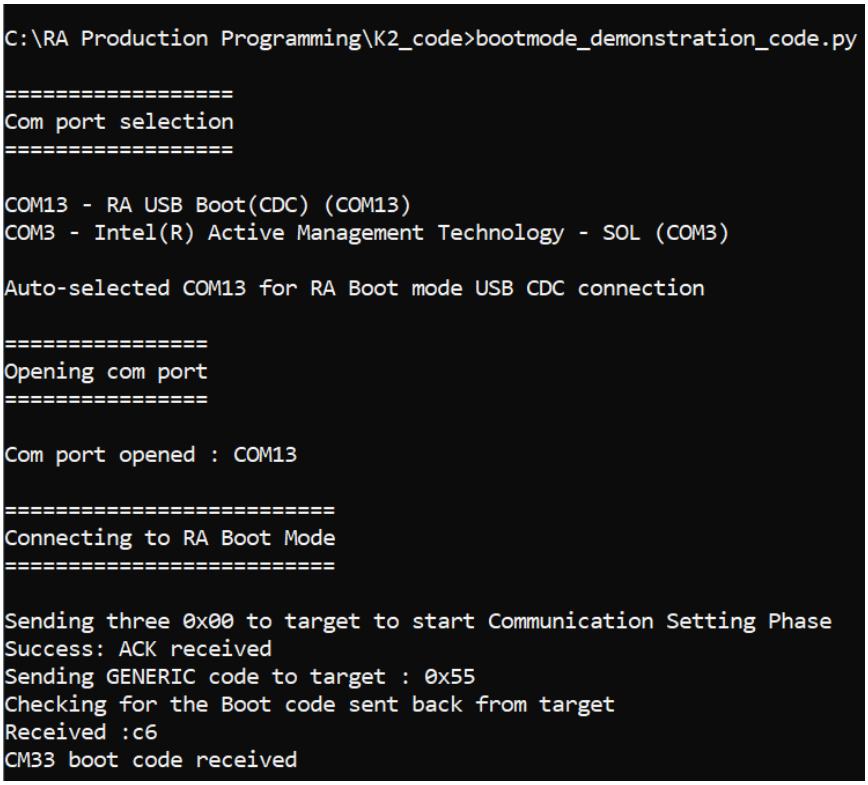


Figure 43. Establishing the Connection USB

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

6.4.2 Checking Current DLM State and Configuring TrustZone Partition Boundaries

```
========================
Requesting the DLM State
========================

Sending DLM State Request command:
b'\x01\x00\x01\x2c\xd3\x03'
Current DLM state is SSD.

================================================
Requesting current TrustZone Boundary information
================================================

Sending read boundary region command:
b'\x01\x00\x01\x4f\xb0\x03'
Read boundary region - SUCCESS
 - The secure code flash region size without NSC is 16383 KB
 - The secure code flash region size with NSC is 16383 KB
 - The secure data flash region size is 63 KB
 - The secure SRAM region size without NSC is 2047 KB
 - The secure SRAM region size with NSC is 2047 KB

================================
Configuring TrustZone Boundaries
================================

Configuring device with :
 - 8kB of secure code flash region without NSC
 - 32kB of secure code flash region with NSC
 - 4kB of secure data flash region
 - 2kB of secure sram region without NSC
 - 32kB of secure sram region with NSC
Sending TrustZone boundary setup command:
b'\x01\x00\x0b\x4e\x00\x08\x00\x20\x00\x04\x00\x02\x00\x20\x59\x03'

Set up boundary region - SUCCESS

================================================
Requesting updated TrustZone Boundary information
================================================

Sending read boundary region command:
b'\x01\x00\x01\x4f\xb0\x03'
Read boundary region - SUCCESS
 - The secure code flash region size without NSC is 8 KB
 - The secure code flash region size with NSC is 32 KB
 - The secure data flash region size is 4 KB
 - The secure SRAM region size without NSC is 2 KB
 - The secure SRAM region size with NSC is 32 KB
```

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

6.4.3 Injecting DLM Keys

```
=========================
Injecting DLM - SECDBG Key
=========================

Start Processing SECDBG rkey file
Base64 Decoding key
Key Type = ' DLM Authentication Key '
Finished : Processing rkey file.

Sending DLM key injection command packet:
b'\x01\x00\x02\x28\x01\xd5\x03'

DLM key injection command packet - SUCCESS

Sending DLM key injection data packet:
b'\x81\x00\x51\x28\x6f\xee\x15\x03\x6a\x3b\x4e\x72\x6f\x0b\x3f\x9e\x1f\x74\xb7\x07\x6f\xee\x15\x03\x6
a\x3b\x4e\x72\x6f\x0b\x3f\x9e\x1f\x74\xb7\x07\xbd\x34\x64\x85\x82\xec\x47\xaf\x25\x2b\x6e\x74\xd3\x89
\x9a\x8f\x09\x39\x55\x7a\xc6\x5c\x07\x81\xbe\xa5\xcc\x22\x75\xb3\xcc\x34\xac\xd2\xc1\x60\x2f\xfd\xe2\
xfb\xaf\x11\x70\x05\xf1\x66\xf5\xc5\x6c\x03'

DLM key injection data packet - SUCCESS

Injecting SECDBG key is successful

========================
Verifying the SECDBG Key
========================

Sending DLM SECDBG key verify command:
b'\x01\x00\x02\x29\x01\xd4\x03'
Verifying injected SECDBG key is successful

==========================
Injecting DLM - NSECDBG Key
==========================

Start Processing NSECDBG rkey file
Base64 Decoding key
Key Type = ' DLM Authentication Key '
Finished : Processing rkey file.

Sending DLM key injection command packet:
b'\x01\x00\x02\x28\x02\xd4\x03'

DLM key injection command packet - SUCCESS

Sending DLM key injection data packet:
b'\x81\x00\x51\x28\x6f\xee\x15\x03\x6a\x3b\x4e\x72\x6f\x0b\x3f\x9e\x1f\x74\xb7\x07\x6f\xee\x15\x03\x6
a\x3b\x4e\x72\x6f\x0b\x3f\x9e\x1f\x74\xb7\x07\xe4\x64\x49\x01\xa6\x48\xb2\x60\xce\x08\x80\x1a\xb8\xb1
\xc4\xe0\xc7\xf9\x9f\x1f\x71\x52\x38\x37\x95\x5e\xc5\xe0\xf3\xbb\x25\x93\xb9\x0e\xff\x22\x0c\xe9\xc9\
xd5\xa9\xc9\x87\xc5\x2a\x4d\xc5\x32\x1f\x03'

DLM key injection data packet - SUCCESS

Injecting NSECDBG key is successful

==========================
Verifying the NSECDBG Key
==========================

Sending DLM NSECDBG key verify command:
b'\x01\x00\x02\x29\x02\xd3\x03'
Verifying injected NSECDBG key is successful
```

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

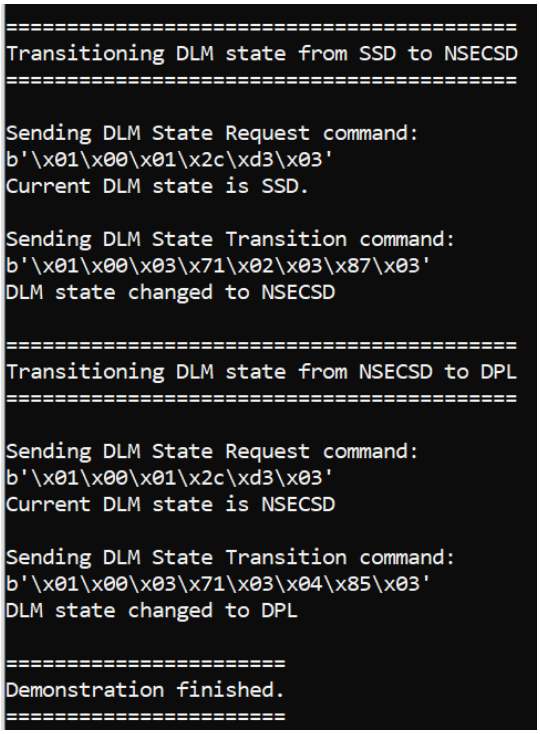


Figure 46. Configuring Final DLM State

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 “0101020304050607080 90A0B0C0D0E0F”. 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 “0001020304050607080 90A0B0C0D0E0F”. 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:

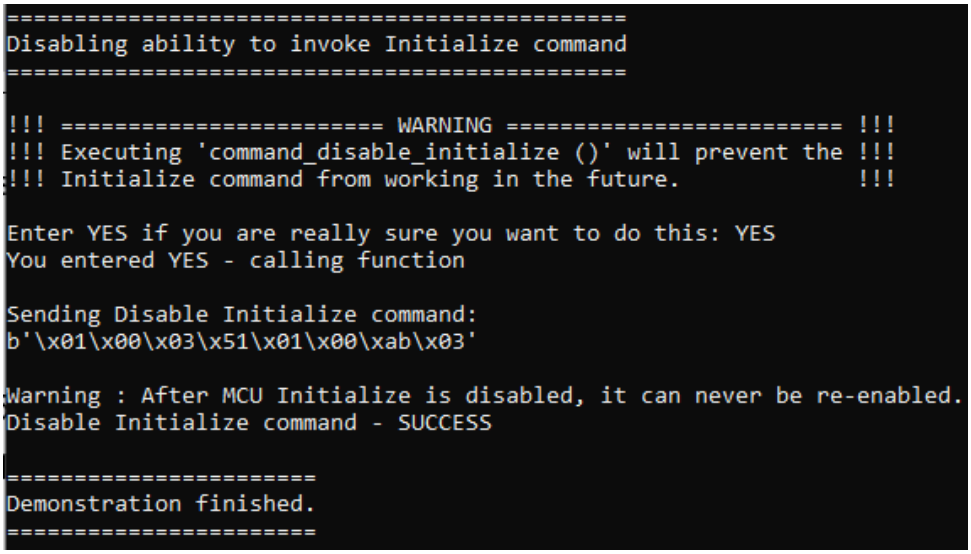


Figure 47. Disable the Initialize Command

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

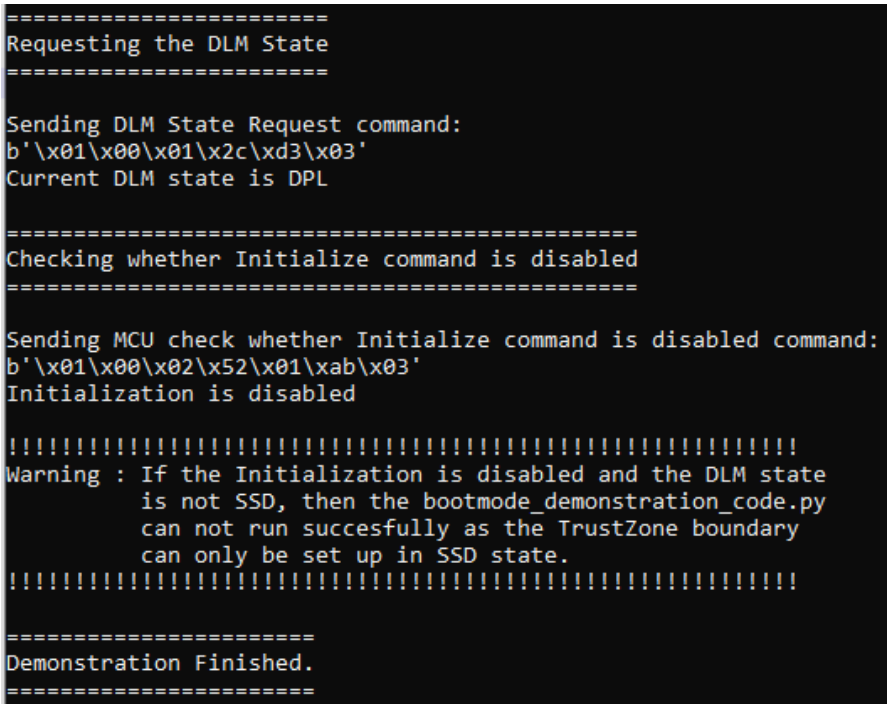


Figure 48. Verify that the Initialize Command is disabled

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

Rev.	Date	Description	
		Page	Summary
1.00	May 25, 2023	—	First release of this document.

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/.