

Renesas RA Family

Implementing Production Programming Tools for RA Cortex-M85 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. On RA8 MCUs, boot mode can be entered through a serial port, USB virtual COM port or SWD/JTAG interface. The ability to connect to boot mode via SWD/JTAG enhances convenience, whilst serial/USB connections retain compatibility with existing production programming tools. In addition, to enter boot mode using the serial port or USB virtual COM port, it is necessary to simply reset the MCU while keeping the MD pin on the device pulled low. When the MCU is in boot mode, user code in flash/MRAM will not be active. An MCU in boot mode enumerates as a COM port when accessed 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 RA8 MCU Family devices based on Cortex-M85 and Cortex-M33 that implement Device Lifecycle Management (DLM) capabilities.

The full specification of the boot mode interface for these RA Family MCUs is detailed in Renesas Boot Firmware application note, e.g. R01AN7140 (RA8M1), R01AN7823 (RA8P1), 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. It includes demonstration code in Python on RA8M1 and RA8P1 to illustrate how boot mode access can be accomplished. The process for other RA8 MCUs is comparable to these two devices, users can follow the guidelines and examples outlined in the application note.

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:

- RA8M1 Group
- RA8D1 Group
- RA8T1 Group
- RA8E1 Group
- RA8E2 Group
- RA8P1 Group
- RA8T2 Group
- RA8D2 Group
- RA8M2 Group

Required Resources

Development tools and software

- Python v3.12 or later (<https://www.python.org/downloads/>)
- pySerial v3.5 (<https://pyserial.readthedocs.io/en/latest/pyserial.html#installation>)
- PyCryptodome ([Installation — PyCryptodome 3.4.6 documentation](#))
- Renesas Flash Programmer v3.19.00 or later (<https://www.renesas.com/rfp>)

Hardware

- EK-RA8M1, Evaluation Kit for RA8M1 MCU Group (<http://www.renesas.com/ra/ek-ra8m1>)
- EK-RA8P1, Evaluation Kit for RA8P1 MCU Group (<http://www.renesas.com/ra/ek-ra8p1>)
 - For demonstration purposes, this application note makes use of the RA8M1/P1 MCU and the EK-RA8M1/P1 evaluation board. However, the available functionality will be the same on the other supported MCU groups except when specifically noted.
- Workstation running Windows® 10/11
 - 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/type-C 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 “Renesas Boot Firmware for (the MCU that is under consideration) Group”. E.g. Renesas Boot Firmware for RA8M1 MCU Group ([R01AN7140](#)).
- The MCU User’s Manual: Hardware (for the MCU that is under consideration).
- Application note “Device Lifecycle Management for RA8 MCUs” ([R01AN0785](#)).

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

Contents

1. Production Programming Concepts	5
1.1 Background	5
1.2 Typical Production Programming Flow.....	5
1.3 Flash Programming	6
1.4 Device Lifecycle Management	6
1.5 Secure / Non-secure /Non-secure Callable Regions	9
1.5.1 SAU Registers and Non-TrustZone-using Software	10
1.6 Production Programming Advanced Features	10
1.6.1 First Stage Bootloader (FSBL) Support	11
1.6.2 Secure Factory Programming (SFP) Support	11
1.7 Key considerations for migrating Production Programming from CM33 to CM85	11
2. MCU Hardware Setup for Boot Mode Use	11
2.1 Boot Mode Communication Interfaces Overview	11
2.2 Power	11
2.3 Clock.....	12
2.4 MCU System Mode Control Signals.....	12
2.5 Using the 2-wire Serial Communication	13
2.6 Using the Universal Serial Bus (USB) Communication	13
2.7 Using Serial Wire Debug Interface (SWD)	13
3. Connecting to Boot Mode	14
3.1 Boot mode operational phases.....	14
3.2 Initialization Phase.....	16
3.2.1 Serial Settings	17
3.2.2 USB Settings	17
3.2.3 JTAG/SWD Settings	17
3.3 Communication Setting Phase	17
3.3.1 USB/UART Communication	17
3.3.2 JTAG/SWD Communication	19
4. Boot Mode Commands	19
4.1 Command Acceptable Phase	19
4.1.1 Command Packet Format	19
4.1.2 Data Packet.....	20
4.1.3 Summary of Boot Mode Commands	20
4.1.4 Boot Mode Firmware Operation	21
5. Typical Use Cases of Boot Mode Commands.....	22
5.1 Overview of Use cases.....	22
5.2 Signature Request Command	22

5.3	Inquiry Command	24
5.4	Initialize the MCU	25
5.4.1	Initialize Command	25
5.4.2	Check Whether Initialize Command is Disabled	26
5.4.3	Disable Initialize Command	28
5.5	TrustZone Boundary Region Setup	30
5.5.1	Operational Flow	30
5.5.2	Acquire the Boundary Information from an Application	30
5.5.3	TrustZone Boundary Request Command	31
5.5.4	TrustZone Boundary Setting Command	33
5.6	DLM Authentication Key Handling	35
5.6.1	Inject DLM Authentication Keys	35
5.6.2	Verify DLM Authentication Keys	39
5.7	DLM State, Protection Level and Authentication Level Handling	41
5.7.1	DLM State Request	41
5.7.2	DLM State Transition	42
5.7.3	Protection Level Request	44
5.7.4	Protection Level Transition	46
5.7.5	Authentication Level Request	48
5.7.6	Authentication Level Transition	49
6.	Running the Python Example Code	54
6.1	Set up the Python Environment	54
6.2	Setting Up the Hardware	54
6.3	Running the First Demo Code	58
6.4	Running the Second Demonstration Code	62
6.4.1	Establishing the Connection (USB)	64
6.4.2	Checking Product Type Name, Current DLM State, Protection Level and Authentication Level	65
6.4.3	Configuring TrustZone Partition Boundaries	65
6.4.4	Injecting DLM AL Keys	66
6.4.5	Configuring Final Protection Level State	67
6.5	Testing Authenticated DLM Transitions	67
6.6	Disabling the Initialize Command	68
7.	References	68
8.	Website and Support	69
	Revision History	70

1. Production Programming Concepts

This section introduces some of the concepts behind the operations required to perform production programming of RA8 MCU Family devices based on Cortex-M85 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.

For RA8 MCU family devices based on the Cortex-M85 that have DLM capabilities, the SWD interface can be used to access the MCU boot mode. Accessing boot mode is necessary to carry out various device configuration operations. In addition to SWD, boot mode can also be accessed via SCI/UART or USB. This application note will demonstrate how to invoke and communicate with boot mode over both SCI/UART and USB. The command sequence can be adapted for production programming tools that use the SWD interface.

SCI/USB 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, Cortex-M4 and Cortex-M33 CPUs 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).

The example in this application demonstrates SCI/UART and USB for establishing communication, but to minimize integration effort and maximize consistency across the ecosystem, we recommend that using SWD/JTAG bootmode communication.

1.2 Typical Production Programming Flow

RA8 MCU Family devices based on Cortex-M85 with DLM capabilities are delivered from the factory in the Chip Manufacturing (CM) or Original Equipment Manufacturer (OEM) 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 or SWD. 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 OEM. This step is explained in more detail later in this document.
3. If the MCU is not in the CM or OEM 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 LCK_BOOT or RMA states. At the end of the “Initialize” command, the MCU is changed to the OEM state and Protection Level 2 (PL2) with the Code Flash, Data Flash (or MRAM on RA8P1) 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.
 - 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 are demonstrated in the example code:
 - a) Configure TrustZone partition boundaries.
 - b) Inject DLM/AL Keys.
 - c) Change to DLM/PL states.

1.3 Flash Programming

Generally, RA8 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

With the RA8P1, the types of memory are:

- Code MRAM memory
- Extra MRAM 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 while the MCU is in normal operating mode.

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

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 6.0.0 at:

- https://github.com/renesas/fsp/releases/download/v6.0.0/MDK_Device_Packs_v6.0.0.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 RA8 family MCUs based on Arm Cortex-M85 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, USB or SWD/JTAG 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 a MCU into at least OEM (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 RA8 Family MCU Group Device Lifecycle States

Lifecycle	Definition	Protection Level	Boot mode access
CM	"Chip Manufacturing" (Some MCUs may be delivered in this state)	PL2	Available. Cannot access code/data flash area.
OEM	"Original Equipment Manufacturer" The device is owned by the customer.	PL2 or PL1 or PL0	Available.
LCK_BOOT	"LoCKed BOOT interface" The debug interface and the serial programming interface and permanently disabled.	PL0	Not available.
RMA_REQ	"Return Material Authorization REQuest" Request for RMA. The customer must send the device to Renesas in this state.	PL0	Available. Cannot access code/data flash area.
RMA_ACK	"Return Material Authorization ACKnowledge" Failure analysis in Renesas.	PL2	Available. Cannot access code/data flash area.
RMA_RET	"Return Material Authorization ACKnowledge" The device is back to the customer. The device does not boot.	PL0	Not available.

The three Protection levels are:

- PL2: The debugger connection is allowed, with no restriction on access to memories and peripherals.
- PL1: The debugger connection is allowed, with access to only non-secure memory regions and peripherals.
- PL0: The debugger connection is not allowed.

Figure 1, Figure 2 and Figure 3 describe the possible transitions between DLM, PL and AL states. Production programming tools need to be able to inject the keys required to allow authenticated DLM state changes.

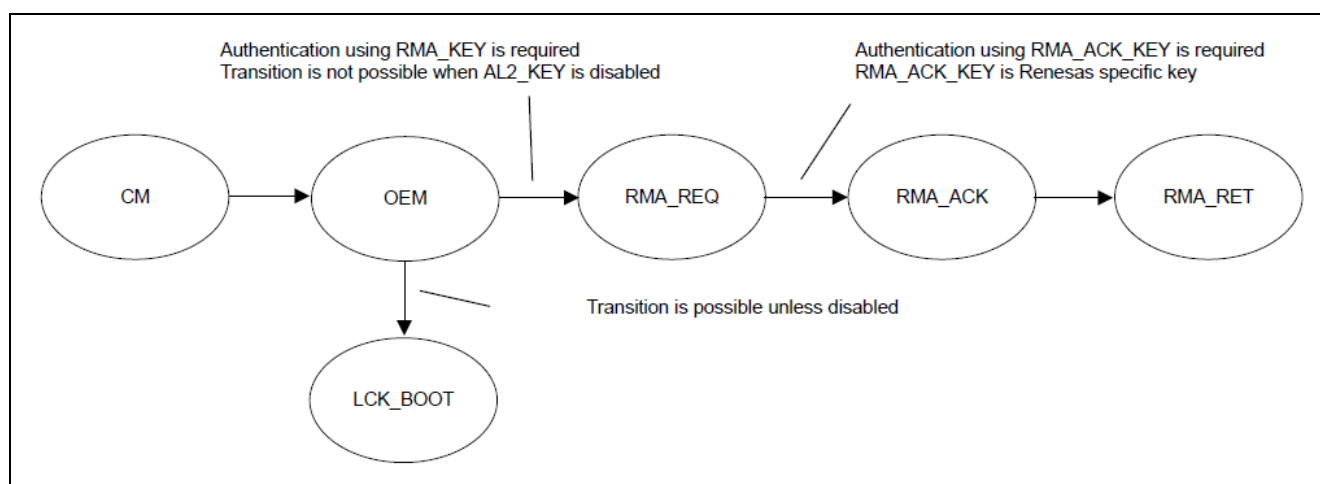


Figure 1. Device Lifecycle Management on RA8M1

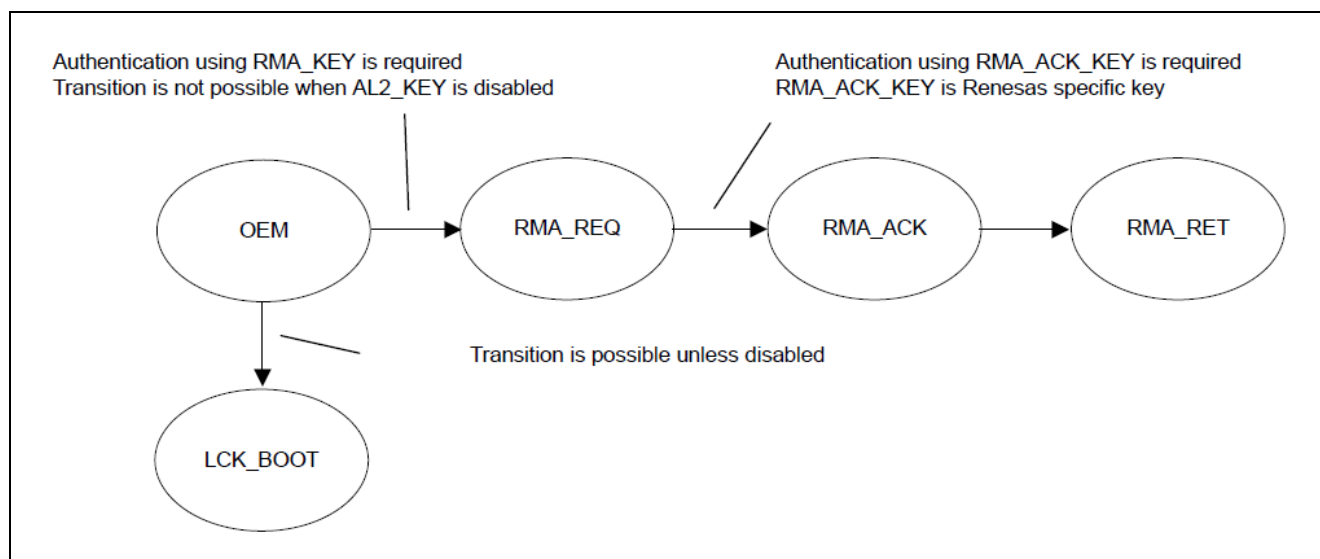


Figure 2. Device Lifecycle Management on RA8P1

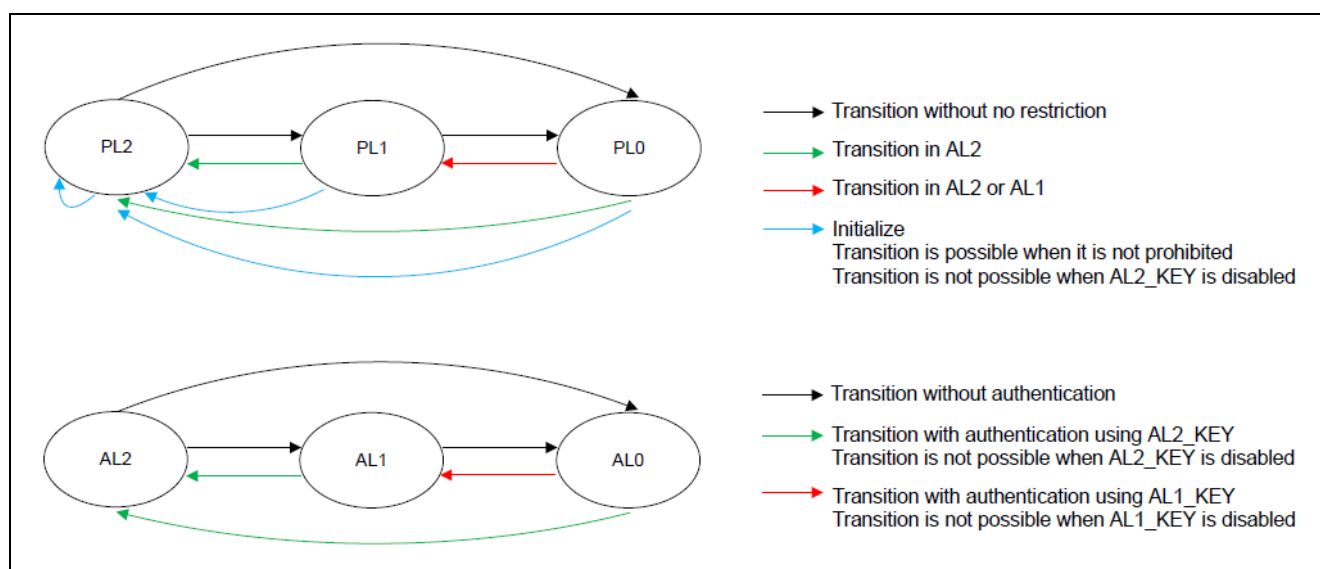


Figure 3. PL and AL states and transitions on RA8 MCU Groups

For production programming, the tool must move a device from CM to OEM for MCUs that are delivered from the factory **if the device is delivered in CM state**. The tool may alternatively need to transit the DLM state back to OEM 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 LCK_BOOT or moving to RMA_REQ state in case of customers need Renesas support. Boot mode also provides a command to disable the “Initialize” command, preventing future erasing of flash/MRAM and resetting of DLM state, Protection Level and Authentication Level.

Authenticated transitions are possible using AL 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:

- **AL2_KEY**
 - The Authentication Level 2 Key can be injected when the MCU is in the OEM state. It can be used when the MCU is in the OEM_PL1 or OEM_PL2 state to regress back to the OEM_PL2 state without erasing flash memory. This key is used in green arrow of Figure 3.
- **AL1_KEY**
 - The Authentication Level 1 Key can be injected when the MCU is in the OEM state. It can be used when the MCU is in the OEM_PL0 state to regress back to the OEM_PL1 state without erasing flash memory. This key is used in red arrow of Figure 3.

Note that an AL key injected during production allows a user to change the DLM/PL 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-M85 based devices, including Renesas' RA8 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-M85 based MCUs with DLM implement a combination of hardware and software defined attribution units is used to establish secure and non-secure memory regions. The Implementation Defined Attribution Unit (IDAU), a fixed hardware component, and its companion Master Security Attribution Unit (MSAU) for other bus masters, work in parallel with the software configurable Security Attribution Unit (SAU). This parallel query system ensures the stricter security label is applied, guaranteeing that secure memory remains protected even if one of the attribution units is misconfigured.

For more details, refer to The MCU User's Manual: Hardware (for the MCU that is under consideration), section Arm Security Features; or refer to Security Design using Arm TrustZone - Cortex M85 ([R11AN0897](#)).

Address	Address map	Security Attribution partition	Region number
0xFFFF_FFFF	Vendor SYS	Exempt	0
0xE000_0000	PPB		
0xC000_0000	External Device	NS	6
0xA000_0000			
0x8000_0000	External RAM		
0x6000_0000			
	Peripheral	NS	5
0x4000_0000		S	
	SRAM	NS	4
0x2000_0000			NSC
	Code	NS	2
0x0000_0000			NSC

Figure 4. TrustZone Configurations Example using RA8M1

The IDAU provides a fixed, hardware-defined security map for code, SRAM, peripheral regions, and MSAU extended that map for other bus masters, which cannot be changed at runtime. Additional TrustZone boundaries as Secure, NSC, Non-Secure regions will be programmed into the SAU 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/Non-Secure memory. The .rpd file can be used by the production programming tool to configure the appropriate values into the security attributes register through boot mode.

1.5.1 SAU 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 SAU 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. Configuring the SAU regions is necessary for such use cases.

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

1.6 Production Programming Advanced Features

The Renesas RA8 MCU also provides security foundation to authenticate and protect the firmware during production programming. For details on the commands to achieve these functionalities, please refer to the boot firmware application note of the MCU. To review an example usage of these features in an application, please refer to the application projects “Application Design using RA8 First Stage Bootloader ([R11AN0774](#)) and RA8 Secure Factory Programming ([R01AN7516](#))”.

1.6.1 First Stage Bootloader (FSBL) Support

The FSBL, which is masked in ROM and can execute after reset to verify the OEM firmware programmed in the on-chip flash. Due to its immutable nature, the FSBL provides a silicon-based Root of Trust (RoT).

To use the FSBL, the OEM firmware must be verified during production programming by the MCU boot firmware using ECDSA. Production programming software must follow the proscribed order of operations (as per the Boot Firmware documentation for the MCU) for injecting keys, programming the image and authenticating the image using the code certificate, optionally a key certificate and OEM root public key.

1.6.2 Secure Factory Programming (SFP) Support

SFP supports image programming in ciphertext format in a non-secure environment. To use the SFP, the production programming software must follow the proscribed order of operations (as per the Boot Firmware documentation for the MCU) for injecting keys, programming the encrypted image and setting up the FSBL (optional).

When using SFP, the encrypted data write boot mode command is used to program the application. The encrypted data write command also changes the device to the specified state, PL0 or LCK_BOOT when saving the data. Thus, a separate DLM or PL state setting command is not required in this case to lock down the device after programming completes.

1.7 Key Considerations for Migrating Production Programming from CM33 to CM85

For users who are already familiar with production programming on CM33, please note that when transitioning to CM85, it is important to understand the key differences between CM85 and CM33, and update the Production Programming software accordingly:

- Add support for the new Device Lifecycle states not currently available on CM33, such as the OEM state, which introduces Protection Level and Authentication Level configurations.
- Support functionality to inject the OEM root key and certificate for FSBL usage.
- Support for injecting Secure Factory Programming (SFP) files using the encrypted data write command. Note that this feature is also available on certain CM33 MCUs, e.g. RA4L1 and RA4C1.

For some commands, the types of keys eligible for injection have been changed. Refer to the Boot Firmware application note for details.

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 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)
- Serial Wire Debug (SWD) Interface
- Multiplex SWD Interface and SCI/UART Interface on the SWD debug header

Communication with boot mode can be carried directly over SWD. In addition, Renesas has defined a specification for reusing certain pins from an SWD debug header as UART pins. This enables production programming tools to communicate over a single physical connector using either SWD (for communication with boot mode and 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 RA8M1 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 RA8M1 MCU.

Table 2. RA8M1 MCU Operating Voltage Range

Operating voltage	VCC = 1.68 to 3.6 V When using USB communication: VCC = 3.0 to 3.6V
--------------------------	--

Table 3. RA8M1 MCU Power Signals

Function Name	Signal	IO	Comments
Power supply	VCC, VCC2	Input	Power supply pin. Connect it 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.
	VCC_DCDC	Input	Switching regulator power supply pin.
	VLO	I/O	Switching regulator pin.
	VCL	Input	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.
	VBATT	Input	Battery Backup power pin.
	VSS, VSS_DCDC	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 requirements on the main oscillator frequency. Table 4 shows the requirement for the RA8 MCU.

Table 4. Clock Source for Boot Mode Operation

Clock Source	RA8M1/D1/T1/E1/E2, RA8P1: Main Oscillator Frequency of 8, 10, 12, 15, 16, 20, 24, 32, 48 MHz can be used by boot mode firmware. Otherwise, HOCO will be used.
---------------------	---

Table 5. Clock Signals

Function Name	Signal	IO	Comments
Clock	XTAL	Output	Pins for a crystal resonator. Input an external clock signal 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, SCI/USB 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	RA8M1/D1/T1/E1/E2, RA8P1: P208
	TXD9	Output	RA8M1/D1/T1/E1/E2, RA8P1: P209

2.6 Using the Universal Serial Bus (USB) Communication

USB communication with boot mode can be used by all the supported MCU groups. 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. For RA8 MCUs, using the debug connector can conveniently support boot mode access as well as flash programming support over the SWD interface.

In addition, the SCI/UART boot access pins can be shared on the debug connector, allowing both the SCI/UART and SWD interfaces to access boot mode through the debug connector. 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 5. Refer to the RA8 MCU Family Quick Design Guide ([R01AN7087](#)) section "Emulator Support" for more details on the specification of this interface.

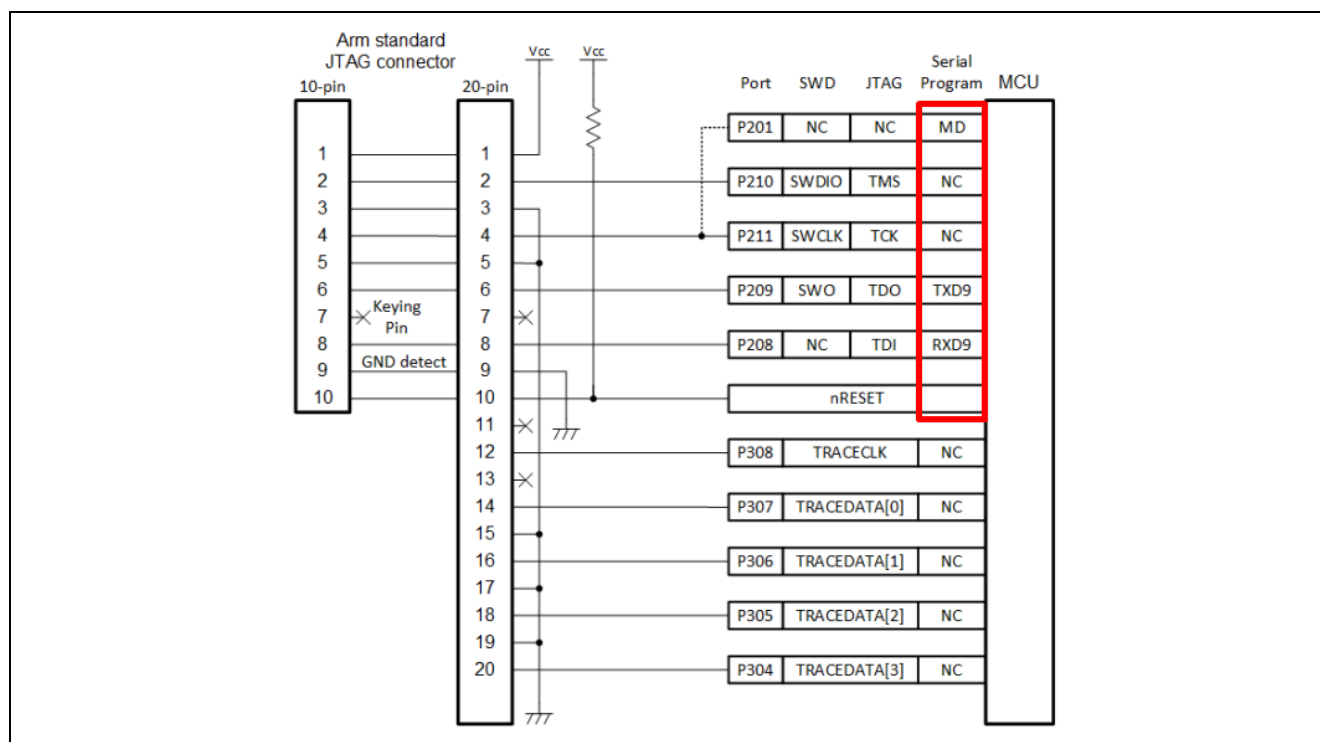


Figure 5. Access the Boot Mode through the Multi-emulator Interface Header

The setup shown in Figure 5 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 for boot mode access. If boot mode access is not 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 6:

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

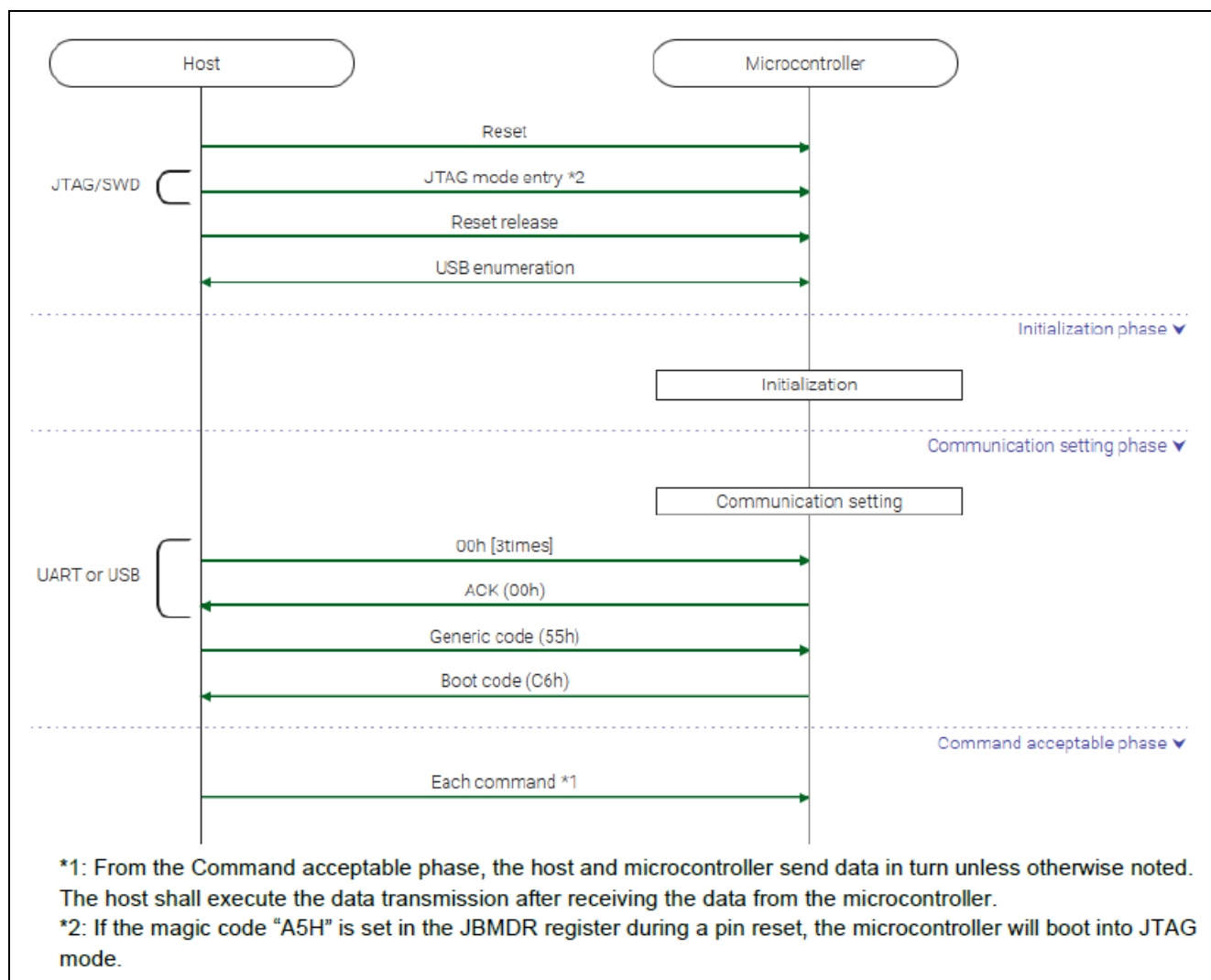


Figure 6. Boot Mode Operational Phases

Figure 7 shows this in more detail, in relation to the MCU DLM state.

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.

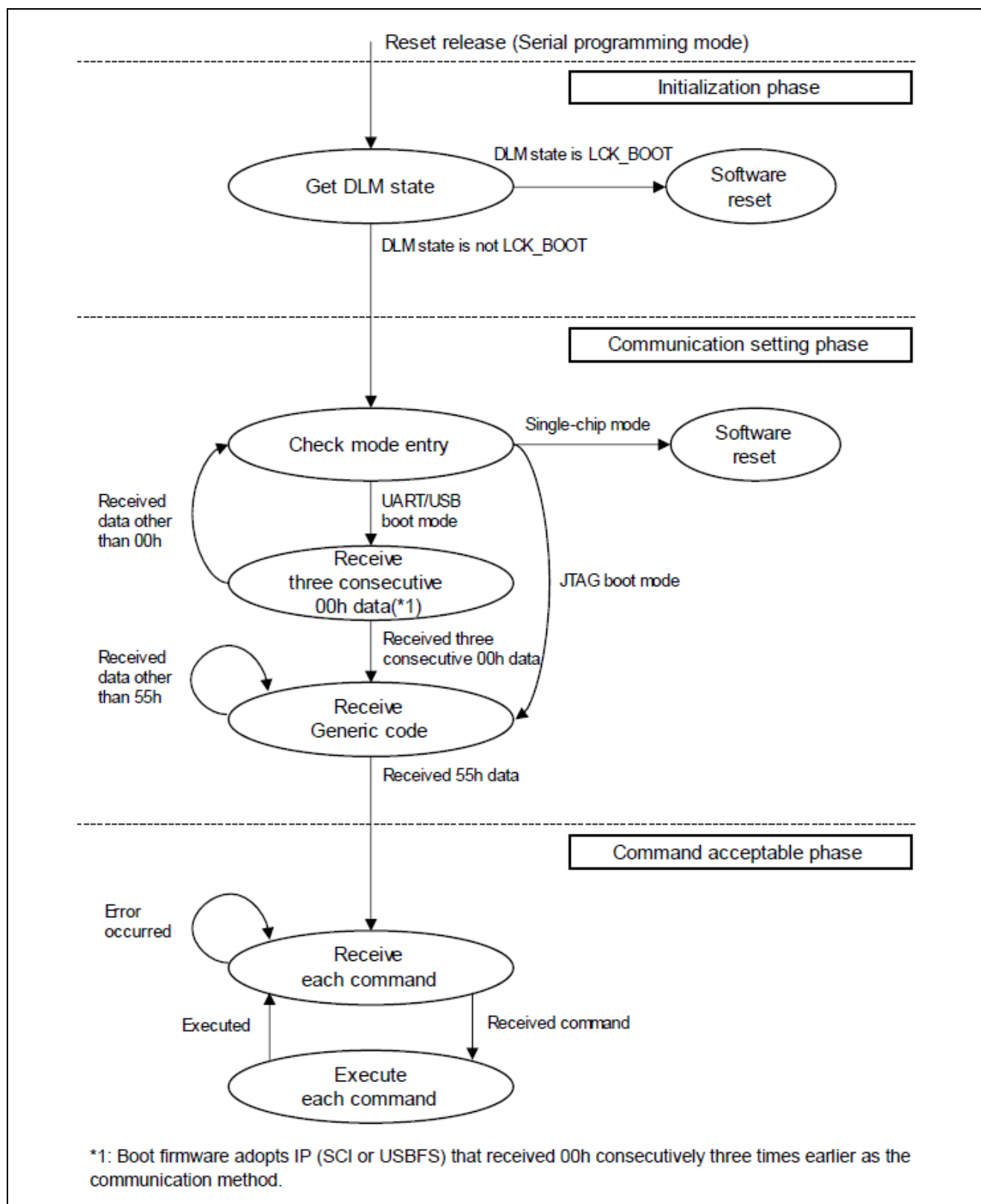


Figure 7. Command Execution State Transition Diagram

3.2 Initialization Phase

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

3.2.1 Serial Settings

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

Bit rate	9600 bps (minimum, until the baud rate setting command) 6Mbps (maximum)
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 communication with the MCU is interrupted during an active boot mode session - for example, due to cable disconnection, host-side timeout, or power loss - the MCU may enter an undefined state. Reset the MCU and reenter the boot mode may be required under certain conditions. The production programming tools should implement timeout handling and retry mechanisms to recover from communication failures whenever possible.
- Communication with the boot firmware through UART is demonstrated in this application note. However, the “Baud rate setting” command is not demonstrated. Refer to Boot Firmware application note, e.g. [R01AN7140](#) for more details of commands. For production use cases requiring custom baud rates, refer to that document for implementation guidance.

3.2.2 USB Settings

When using USB communication to access the 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 (Renesas)
Product ID	0x0261
Transfer mode	Control (in/out) Bulk (in, out) Interrupt (in)

3.2.3 JTAG/SWD Settings

When using JTAG/SWD communication to access the boot mode, the following settings are in use:

Transfer rate	25MHz (maximum)
Data length	32 bits
Magic code	A5h

3.3 Communication Setting Phase

There are 3 methods to communicate with the device in boot mode:

- USB and UART: These two interfaces are demonstrated in this application project.
- JTAG/SWD: This method is not demonstrated in the current application.

More information and signal timing visualization are available in “Communication Methods” and “General Procedure” section in Boot Firmware application note.

3.3.1 USB/UART Communication

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 8 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 Section "Communication Setting Phase" in Renesas Boot Firmware app note.
        # Ex: R01AN7140(RA8M1) for 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 8. 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 5.3.

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 9. 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("CM85/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 9. 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.

3.3.2 JTAG/SWD Communication

This is enabled by setting a magic code in the JBMDR register during terminal reset and the MD pin level is high. When JTAG/SWD communication is selected, boot firmware waits for the generic code without waiting for 0x00 characters. Refer to the Boot Firmware application note for detailed procedures to enter and exit Boot Firmware mode; production programming software should adhere to these steps to establish communication.

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 Boot Firmware application note, e.g. R01AN7140, R01AN7290.

4.1.1 Command Packet Format

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	01h	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. Examples: - For Write command: Start/End address. - For Baudrate setting command: UART baudrate.
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 = 00h.
ETX	1 byte	03h	End of packet.

Note: If the host sends data that exceeds 261 bytes, subsequent operations are not guaranteed.

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	81h	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 Boot Firmware application note - section "RES: Response code" for all the supported response codes.
Data	(*3)	-	Transmit data. Examples: - For Write data transmission: Write data. - For Status transmission: Status code (STS), Status details (ST2) and Failure address (ADR).
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 = 00h.
ETX	1 byte	03h	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 an indefinite RES value.

*2: If the host sends data that exceeds 1030 bytes, subsequent operations are not guaranteed.

*3: The size is 1-1024 bytes. As an exception, the maximum is 1040 bytes only for the Encrypted data write command.

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 R01AN7140 to understand the command format details, response, and error handling.

Table 11. Boot Command Code Summary

Value	Device		Name	Comment
	RA8M1, RA8D1, RA8T1, RA8E1, RA8E2	RA8P1		
71h	○	○	DLM state transit command	Demonstrated
2Ch	○	○	DLM state request command	Demonstrated
72h	○	○	Protection level transit command	Demonstrated
73h	○	○	Protection level request command	Demonstrated
75h	○	○	Authentication level request command	Demonstrated
30h	○	○	Authentication command	Demonstrated
28h	○	○	Key setting command	Demonstrated
2Ah	○	○	User key setting command	Not demonstrated
29h	○	○	Key verify command	Demonstrated
2Bh	○	○	User key verify command	Not demonstrated
50h	○	○	Initialize command	Demonstrated
4Eh	○	○	Boundary setting command	Demonstrated
4Fh	○	○	Boundary request command	Demonstrated
51h	○	○	Parameter setting command	Not demonstrated
52h	○	○	Parameter request command	Not demonstrated
4Ah	○		Lock bit setting command	Not demonstrated
4Bh	○		Lock bit request command	Not demonstrated
4Ch	○	○	ARC configuration setting command	Not demonstrated
4Dh	○	○	ARC configuration request command	Not demonstrated
00h	○	○	Inquiry command	Demonstrated
3Ah	○	○	Signature request command	Demonstrated
3Bh	○	○	Area information request command	Not demonstrated
34h	○	○	Baudrate setting command	Not demonstrated
12h	○	○	Erase command	Not demonstrated
13h	○	○	Write command	Not demonstrated
15h	○	○	Read command	Not demonstrated
18h	○	○	CRC command	Not demonstrated
2Eh	○	○	OEM root public key setting command	Not demonstrated
2Fh		○	OEM root public key verify command	Not demonstrated
26h	○	○	Code certificate update command	Not demonstrated
27h	○	○	Code certificate check command	Not demonstrated
38h		○	SiP Flash various setting command	Not demonstrated
39h		○	SiP Flash various request command	Not demonstrated
36h	○	○	External flash memory setting command	Not demonstrated
1Ah	○	○	Encrypted data 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 section *Command List* in Boot Firmware application note. Table 12 details the specific subsections from Boot Firmware 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 Boot Firmware AN
Device Signature Command	Signature Request Command
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 AL keys	Key setting command
Verify AL 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 Signature Request Command

This command sends the information of the device signature to the host.

6.22.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	01h
CMD	(1 byte)	3Ah (Signature request command)
SUM	(1 byte)	C5h
ETX	(1 byte)	03h

6.22.2.2 Data Packet [Signature]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	2Ah
RES	(1 byte)	3Ah (OK)
RMB	(4 bytes)	Recommended maximum UART baudrate of the device [bps]. *Order of sending: High -> ... -> Low For example: 6 Mbps (6000000bps) -> 00h, 5Bh, 8Dh, 80h
NOA	(1 byte)	Number of accessible areas For example, if the device has 4 areas -> 04h
TYP	(1 byte)	Type code (features and functions of the device): 03h: RA8M1 MCU Group and others
BFV	(3 byte)	Boot firmware version Order of sending: Major version -> minor version -> build For example, v2.4.1.6 -> 02h, 04h, 10h
DID	(16 bytes)	Device ID 16-byte ID code (unique ID) for identifying the particular MCU
PTN	(16 bytes)	Product type name. Character strings (20h for the space) Order of sending example: R7FA6M3AH ->52h, 37h, 46h, 41h, 36h, 4dh, 33h, 41h, 48h, 20h, 20h, 20h, 20h, 20h, 20h
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

Figure 10. Signature Request Command Packets

```
# =====
# This routine demonstrates the "Signature Request Command".
def command_signature_request():
    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x01'
    #0x3A: Signature request command
    CMD=b'\x3A'
    SUM=b'\xC5'
    EXT=b'\x03'
    command = SOH + LNH + LNL + CMD + SUM + EXT

    print("Sending Signature Request 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 != 0x3A:
        print("Signature Request - FAIL")
    else:
        print("Signature Request - SUCCESS")
        # The TYP byte is the tenth index
        TYP = return_packet[9]
        # The PTN byte starts from the thirtieth index
        PTN = return_packet[29:44]
        product_type_name = PTN.rstrip(b'\x20').decode('ascii')
        print("Product Type Name: " + product_type_name)

    return TYP
```

Figure 11. Signature Request Command Example Code

5.3 Inquiry Command

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

6.21.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	01h
CMD	(1 byte)	00h (Inquiry command)
SUM	(1 byte)	FFh
ETX	(1 byte)	03h

Figure 12. 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'
    #0x00: Inquiry command
    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 13. Inquiry Command Example Code

5.4 Initialize the MCU

The commands introduced in this section are used to ensure that the MCU is in the OEM DLM state and Protection Level is PL2 state to ready for other operations, such as flash programming.

5.4.1 Initialize Command

The Initialize command can be executed in the OEM state with PL2, PL1, PL0 of the Protection Level state. It clears the User area, Data area, Config area, EEP config area, Boundary setting, and Key index (Wrapped keys). In addition, the PL state transitions to PL2 from PL1 and PL0. 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 5.7.1).

6.12.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	03h
CMD	(1 byte)	50h (Initialize command)
SDLM	(1 byte)	Source DLM state code: • 04h: OEM
DDLM	(1 byte)	Destination DLM state code: • 04h: OEM
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

6.12.2.2 Data Packet [Status OK]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	0Ah
RES	(1 byte)	50h (OK)
STS	(1 byte)	00h (OK)
ST2	(4 bytes)	FFFFFFFFh (unused code)
ADR	(4 bytes)	FFFFFFFFh (unused code)
SUM	(1 byte)	A Eh
ETX	(1 byte)	03h

Figure 14. Initialize Command Packets

```
# =====
# The "Initialize Command" clears User area, Data area, Config area,
# EEP config area, Boundary setting, and Key index (Wrapped key).
# In addition, the PL state transitions to PL2.
def command_initialize(current_DLM_state):
    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x03'
    #0x50: Initialize command
    CMD=b'\x50'
    # Source DLM state code
    SDLM=current_DLM_state
    # Destination DLM state code
    DDLM=b'\x04'
    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!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
    print("Warning : After MCU Initialize is run, an MCU reset is")
    print("          required before further boot mode operations.")
    print("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
```

Figure 15. Initialize Command Example Code

Note: In the example code, INITIALIZE_DELAY is set 5 seconds. The command response time specified in the “AC Characteristics” section in Boot Firmware documentation. Users can safely adjust this value to compatible with the specific hardware set up used, as long as the hardware still responds correctly.

5.4.2 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.

6.16.2.1 Command Packet

SOH	(1 byte)	01h															
LNH	(1 byte)	00h															
LNL	(1 byte)	02h															
CMD	(1 byte)	52h (Parameter request command)															
PMID	(1 byte)	Parameter ID Specifiable parameter: <table border="1"> <thead> <tr> <th>PMID</th><th>Parameter description</th><th>Specifiable after Encrypted data write command</th></tr> </thead> <tbody> <tr> <td>01h</td><td>Disable initialization</td><td>Specifiable</td></tr> <tr> <td>02h</td><td>Disable LCK_BOOT</td><td>Specifiable</td></tr> <tr> <td>03h</td><td>Disable AL2_key</td><td>Specifiable</td></tr> <tr> <td>04h</td><td>Disable AL1_key</td><td>Non-specifiable</td></tr> </tbody> </table>	PMID	Parameter description	Specifiable after Encrypted data write command	01h	Disable initialization	Specifiable	02h	Disable LCK_BOOT	Specifiable	03h	Disable AL2_key	Specifiable	04h	Disable AL1_key	Non-specifiable
PMID	Parameter description	Specifiable after Encrypted data write command															
01h	Disable initialization	Specifiable															
02h	Disable LCK_BOOT	Specifiable															
03h	Disable AL2_key	Specifiable															
04h	Disable AL1_key	Non-specifiable															
SUM	(1 byte)	Sum data															
ETX	(1 byte)	03h															

6.16.2.2 Data Packet [Parameter Data]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	02h
RES	(1 byte)	52h (OK)
PRMT	(1 byte)	Parameter data: • [PMID=01h] — 00h: Initialization is disabled. — 07h: Initialization is enabled. • [PMID=02h] — 00h: Transition to LCK_BOOT is disabled. — 07h: Transition to LCK_BOOT is enabled. • [PMID=03h] — 00h: Authentication using AL2_KEY is disabled (*1). — 07h: Authentication using AL2_KEY is enabled. • [PMID=04h] — 00h: Authentication using AL1_KEY is disabled. — 07h: Authentication using AL1_KEY is enabled.
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

*1: When disabled, initialization and transition to RMA_REQ are also impossible.

Figure 16. 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'
    #0x52: 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")
            print("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
            print("Warning : If the Initialization is disabled and the DLM state")
            print("          is not OEM, then the bootmode_demonstration_code_RA8.py")
            print("          can not run successfully as the TrustZone boundary")
            print("          can only be set up in OEM state.")
            print("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!")
        elif PRMT == 0x07:
            print("Initialization is enabled")
        else:
            print("Check whether Initialize Command is disabled or not command failed")

    return PRMT
```

Figure 17. Example Code: Check whether Initialize Command is Disabled

5.4.3 Disable Initialize Command

As part of the final production programming process, the Initialize command can be disabled if required.

NOTE: This step is **non-reversible, so exercise with caution!**

This command is included in the `bootmode_demonstration_code_RA8.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.

6.15.2.1 Command Packet

SOH	(1 byte)	01h																				
LNH	(1 byte)	00h																				
LNL	(1 byte)	03h																				
CMD	(1 byte)	51h (Parameter setting command)																				
PMID	(1 byte)	Parameter ID Specifiable parameter: <table><tr><th>PMID</th><th>Parameter description</th><th>Specifiable at</th><th>Specifiable after Encrypted data write command</th></tr><tr><td>01h</td><td>Disable initialization</td><td>AL2/AL1/AL0</td><td>Specifiable</td></tr><tr><td>02h</td><td>Disable LCK_BOOT</td><td>AL2/AL1</td><td>Specifiable</td></tr><tr><td>03h</td><td>Disable AL2_key</td><td>AL2</td><td>Specifiable</td></tr><tr><td>04h</td><td>Disable AL1_key</td><td>AL2/AL1</td><td>Non-specifiable</td></tr></table>	PMID	Parameter description	Specifiable at	Specifiable after Encrypted data write command	01h	Disable initialization	AL2/AL1/AL0	Specifiable	02h	Disable LCK_BOOT	AL2/AL1	Specifiable	03h	Disable AL2_key	AL2	Specifiable	04h	Disable AL1_key	AL2/AL1	Non-specifiable
PMID	Parameter description	Specifiable at	Specifiable after Encrypted data write command																			
01h	Disable initialization	AL2/AL1/AL0	Specifiable																			
02h	Disable LCK_BOOT	AL2/AL1	Specifiable																			
03h	Disable AL2_key	AL2	Specifiable																			
04h	Disable AL1_key	AL2/AL1	Non-specifiable																			
PRMT	(1 byte)	Parameter data: • [PMID=01h] — 00h: Disable initialization • [PMID=02h] — 00h: Disable transition to LCK_BOOT • [PMID=03h] — 00h: Disable of authentication using AL2_KEY (*1) • [PMID=04h] — 00h: Disable of authentication using AL1_KEY																				
SUM	(1 byte)	Sum data																				
ETX	(1 byte)	03h																				

*1: When disabled, initialization and transition to RMA_REQ are also impossible.

6.15.2.2 Data Packet [Status OK]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	0Ah
RES	(1 byte)	51h (OK)
STS	(1 byte)	00h (OK)
ST2	(4 bytes)	FFFFFFFFh (unused code)
ADR	(4 bytes)	FFFFFFFFh (unused code)
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

Figure 18. Disable the 'Initialize Command' Command Packet

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

5.5 TrustZone Boundary Region Setup

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

5.5.1 Operational Flow

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

1. Acquire the TrustZone partition boundary information from the application.
2. Check DLM state and AL as necessary. The TrustZone partition boundaries can only be set up in OEM with PL2 state.
3. Set up boundaries.
4. Verify the boundaries set up properly.

5.5.2 Acquire the Boundary Information from an Application

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

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 20 shows the contents of the rpd file.

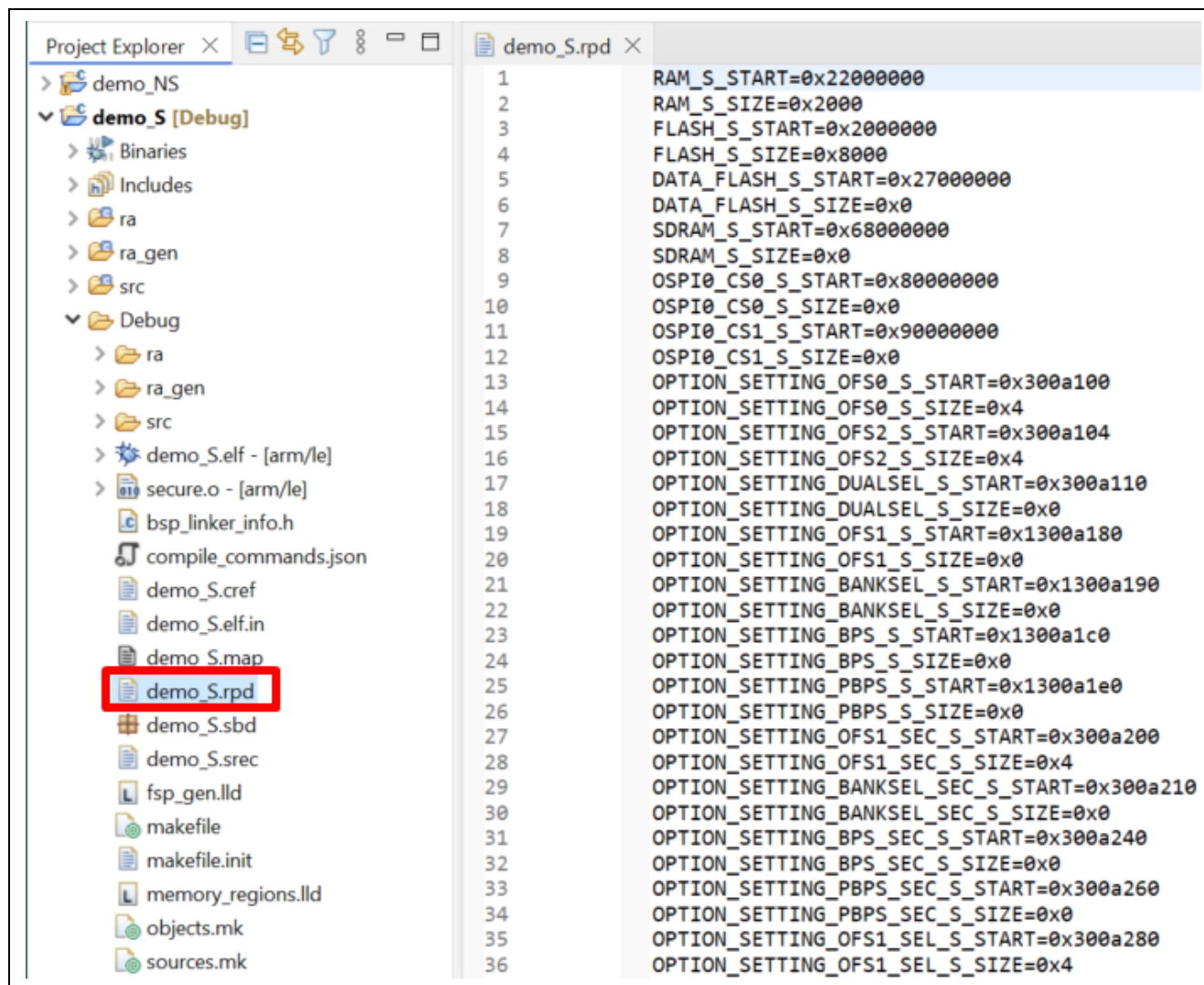


Figure 20. Obtain the IDAU Region Size using EWARM

The FLASH_S_SIZE is the size of the Secure Code Flash region. The production programming tools need to convert this value to KB (kilobytes) and then assign this value to the CFS as shown in Figure 23.

The DATA_FLASH_S_SIZE 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 DFS as shown in Figure 23.

Production programming tools can ignore the other fields.

5.5.3 TrustZone Boundary Request Command

Reading the configured security attribute regions setup can be achieved using the command in Figure 21. The example code to perform this function is shown in Figure 22.

6.14.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	01h
CMD	(1 byte)	4Fh (Boundary request command)
SUM	(1 byte)	B0h
ETX	(1 byte)	03h

6.14.2.2 Data packet [Boundary Setting Data]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	0Bh
RES	(1 byte)	4Fh (OK)
RSV	(2 bytes)	0000h (unused code)
CFS	(2 bytes)	Size of Code Flash Secure region [KB] For example: 0100h -> 01h, 00h (256 KB)
DFS	(2 bytes)	Size of Data Flash Secure region [KB] For example: 0004h -> 00h, 04h (4 KB)
RSV	(2 bytes)	0000h (unused code)
RSV	(2 bytes)	0000h (unused code)
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

Figure 21. Command Packet for Reading Security Region Setup

```
# =====
# This routine demonstrates "TrustZone Boundary Request Command".
def command_trustzone_boundary_request(sig_type):
    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("Read boundary region - SUCCESS")
    print_boundary_region(return_packet, sig_type)

    return RES
```

Figure 22. Example Code: Request TrustZone Boundary

5.5.4 TrustZone Boundary Setting Command

Figure 23 is the command packet for setting up the TrustZone boundary. The new stored boundary setting becomes effective after resetting the device.

6.13.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	0Bh
CMD	(1 byte)	4Eh (Boundary setting command)
RSV	(2 bytes)	0000h (unused code)
CFS	(2 bytes)	Size of Code Flash Secure region [KB]. For example: 0100h -> 01h, 00h (256 KB) * 32 KB align
DFS	(2 bytes)	Size of Data Flash Secure region [KB]. For example: 0004h -> 00h, 04h (4 KB)
RSV	(2 bytes)	0000h (unused code)
RSV	(2 bytes)	0000h (unused code)
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

* If CFS does not comply with alignment, boot firmware rounds down them.

Figure 23. Command Packet for TrustZone Boundary Setup

Figure 24 shows the example code for setting up the TrustZone boundary cover for both RA8x1 and RA8P1 Groups.

```
# =====
# This routine demonstrates "TrustZone Boundary Setting Command".
def command_setup_trustzone_boundary(sig_type):
    print("Configuring device with :")
    if sig_type == 0x03:
        # RA8x1 MCU Group
        SOH=b'\x01'
        LNH=b'\x00'
        LNL=b'\x0B'
        CMD=b'\x4E'      # Boundary setting command
        RSV_1=b'\x00'    # Unused code
        RSV_2=b'\x00'
        print(" - 512KB of Code Flash Secure region")
        CFS_1=b'\x02'    # 512KB
        CFS_2=b'\x00'
        print(" - 4KB of Data Flash Secure region")
        DFS_1=b'\x00'
        DFS_2=b'\x04'    # 4KB
        RSV_3=b'\x00'    # Unused code
        RSV_4=b'\x00'
        RSV_5=b'\x00'    # Unused code
        RSV_6=b'\x00'
        SUM=calc_sum(LNH + LNL + CMD + RSV_1 + RSV_2 + CFS_1 + CFS_2 + \
                    DFS_1 + DFS_2 + RSV_3 + RSV_4 + RSV_5 + RSV_6)
        EXT=b'\x03'
        command = SOH + LNH + LNL + CMD + RSV_1 + RSV_2 + CFS_1 + CFS_2 + \
                    DFS_1 + DFS_2 + RSV_3 + RSV_4 + RSV_5 + RSV_6 + SUM + EXT
    else:
        # RA8P1 MCU
        SOH=b'\x01'
        LNH=b'\x00'
        LNL=b'\x0B'
        CMD=b'\x4E'      # Boundary setting command
        RSV_1=b'\x00'    # Unused code
        RSV_2=b'\x00'
        print(" - 512KB of Code MRAM Secure region")
        CMS_1=b'\x02'    # 512KB
        CMS_2=b'\x00'
        RSV_3=b'\x00'    # Unused code
        RSV_4=b'\x00'
        RSV_5=b'\x00'    # Unused code
        RSV_6=b'\x00'
        RSV_7=b'\x00'    # Unused code
        RSV_8=b'\x00'
        SUM=calc_sum(LNH + LNL + CMD + RSV_1 + RSV_2 + CMS_1 + CMS_2 + \
                    RSV_3 + RSV_4 + RSV_5 + RSV_6 + RSV_7 + RSV_8)
        EXT=b'\x03'
        command = SOH + LNH + LNL + CMD + RSV_1 + RSV_2 + CMS_1 + CMS_2 + \
                    RSV_3 + RSV_4 + RSV_5 + RSV_6 + RSV_7 + RSV_8 + SUM + EXT

    print("Sending TrustZone boundary setup 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 != 0x4E:
        print("\n*** ERROR : Set up boundary region - FAIL")
    else:
        print("\nSet up boundary region - SUCCESS")

    return RES
```

Figure 24. Example Command Setting up TrustZone Boundary

5.6 DLM Authentication Key Handling

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

Key injection for AL2 and AL1 states is demonstrated in this application note. Injection of the RMA key can follow similar sequence but is not demonstrated in this application note.

Note: The injection of user keys is very similar to AL 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 Authentication keys is described in [R11AN0785](#).

5.6.1 Inject DLM Authentication Keys

Injecting a DLM AL key requires a two-stage sequence, as shown in Figure 25.

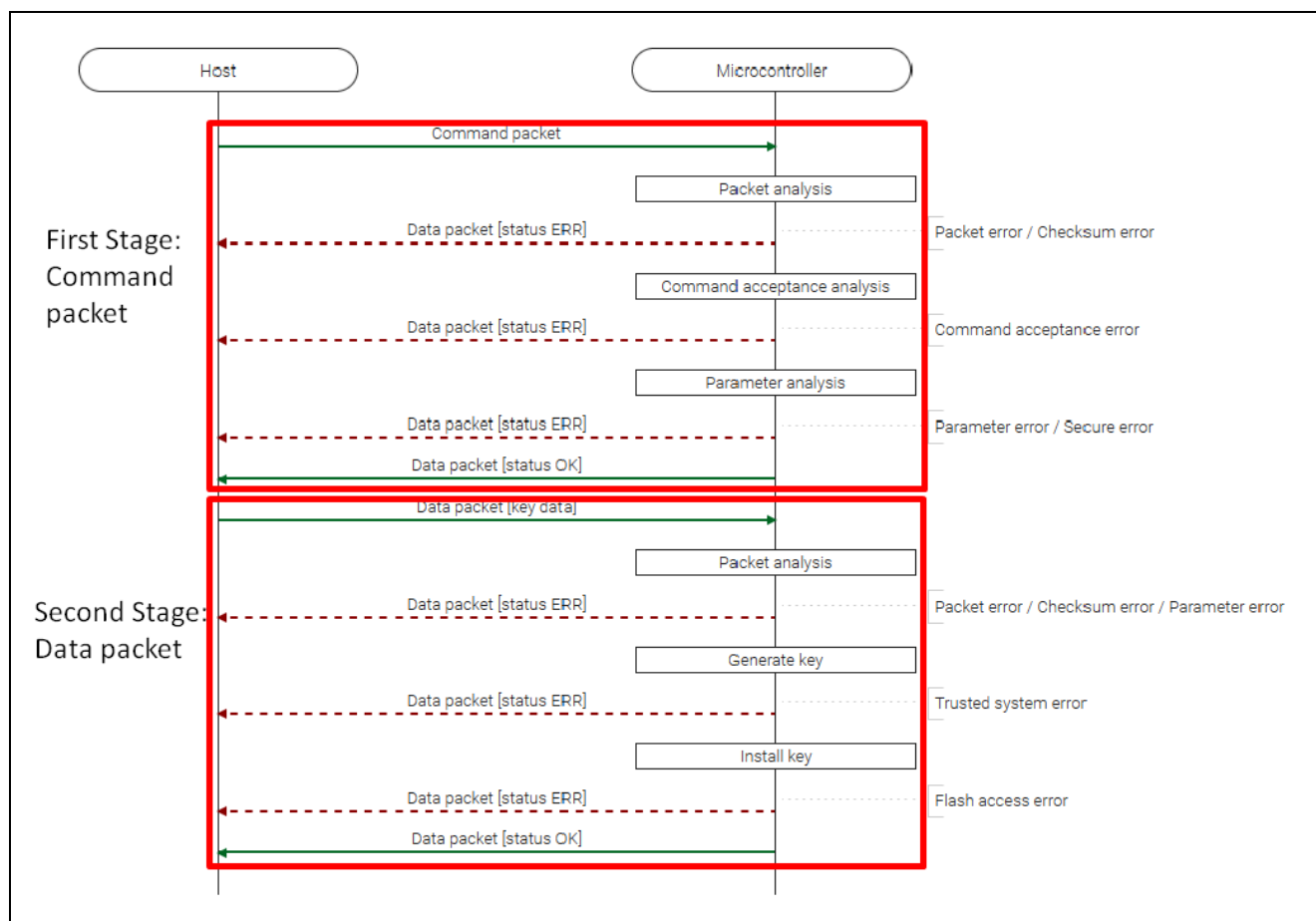


Figure 25. DLM Key Injection Flow

6.8.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	02h
CMD	(1 byte)	28h (Key setting command)
KYTY	(1 byte)	Key type: • 01h: AL2_KEY • 02h: AL1_KEY • 03h: RMA_KEY
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

Figure 26. DLM Key Injection Command Packet

6.8.2.2 Data Packet [Key Data]

SOD	(1 byte)	81h																																																																																																																
LNH	(1 byte)	00h																																																																																																																
LNL	(1 byte)	55h																																																																																																																
RES	(1 byte)	28h (OK)																																																																																																																
SKR	(4 bytes)	Shared key ring number. For example: 01234567h -> 01h, 23h, 45h, 67h																																																																																																																
ESKY	(32 bytes)	Wrapped install key (W-UFPPK). For example: 01234567_89AB ... 2233_44556677h -> 01h, 23h, 45h, ... , 55h, 66h, 77h																																																																																																																
IVEC	(16 bytes)	Initialization Vector. For example: 01234567_89AB ... 2233_44556677h -> 01h, 23h, 45h, ... , 55h, 66h, 77h																																																																																																																
EOKY	(32 bytes)	Install data (Encrypted key MAC). Encrypted key (bytes 0–15) + MAC (bytes 16-31) For example: If install data is as follows, the host should send EOKY in the order shown in the lower table. 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><td>9th</td><td>10th</td><td>11th</td><td>12th</td><td>13th</td><td>14th</td><td>15th</td><td>16th</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><td>17th</td><td>18th</td><td>19th</td><td>20th</td><td>21st</td><td>22nd</td><td>23rd</td><td>24th</td></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>25th</td><td>26th</td><td>27th</td><td>28th</td><td>29th</td><td>30th</td><td>31st</td><td>32nd</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>	Encrypted key								00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	MAC								10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	1st	2nd	3rd	4th	5th	6th	7th	8th	00	01	02	03	04	05	06	07	9th	10th	11th	12th	13th	14th	15th	16th	08	09	0A	0B	0C	0D	0E	0F	17th	18th	19th	20th	21st	22nd	23rd	24th	10	11	12	13	14	15	16	17	25th	26th	27th	28th	29th	30th	31st	32nd	18	19	1A	1B	1C	1D	1E	1F
Encrypted key																																																																																																																		
00	01	02	03	04	05	06	07																																																																																																											
08	09	0A	0B	0C	0D	0E	0F																																																																																																											
MAC																																																																																																																		
10	11	12	13	14	15	16	17																																																																																																											
18	19	1A	1B	1C	1D	1E	1F																																																																																																											
1st	2nd	3rd	4th	5th	6th	7th	8th																																																																																																											
00	01	02	03	04	05	06	07																																																																																																											
9th	10th	11th	12th	13th	14th	15th	16th																																																																																																											
08	09	0A	0B	0C	0D	0E	0F																																																																																																											
17th	18th	19th	20th	21st	22nd	23rd	24th																																																																																																											
10	11	12	13	14	15	16	17																																																																																																											
25th	26th	27th	28th	29th	30th	31st	32nd																																																																																																											
18	19	1A	1B	1C	1D	1E	1F																																																																																																											
SUM	(1 byte)	Sum data																																																																																																																
ETX	(1 byte)	03h																																																																																																																

Figure 27. 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 28, and is further explained in the user manual of Secure Key Management Tool.

Key Data is stored in order and size shown in Figure 28. 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 28. DLM Key Data Structure

The demonstration code for DLM Authentication key injection is included in function `command_inject_dlm_key()`. The main operations carried out by this function are:

- Read the AL2 or AL1 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-uwpk, initialization vector, and the encrypted DLM key to ensure valid content.
- Issue DLM AL Key Injection command packet and verify the response.
- Issue DLM AL Key Injection data packet and verify the response using the decoded key data.

Figure 29 and Figure 30 show the example code.

```
# =====
# This routine demonstrates "DLM Key Injection" command.
# The supported key types are AL1 and AL2 key.
def command_inject_dlm_key(dlm_key_type, sig_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'):
        if sig_type == 0x03:
            # RA8x1 MCU Group
            text_file = open("./dlm_keys/AL2_KEY_RA8M1.rkey", "r")
        else:
            # RA8P1 MCU
            text_file = open("./dlm_keys/AL2_KEY_RA8P1.rkey", "r")
    elif (dlm_key_type == b'\x02'):
        if sig_type == 0x03:
            # RA8x1 MCU Group
            text_file = open("./dlm_keys/AL1_KEY_RA8M1.rkey", "r")
        else:
            # RA8P1 MCU
            text_file = open("./dlm_keys/AL1_KEY_RA8P1.rkey", "r")

    message_bytes, encrypted_key_size = parse_the_dlm_key_field(text_file, dlm_key_type, sig_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("\nSending DLM key injection command packet: ")
        print_bytes_hex(command)
        ser.write(command)
        time.sleep(LONG_DELAY)

        # Acquire the response of the command packet stage
        return_packet = receive_data_packet()
        # The RES byte is the fourth index
        RES = return_packet[3]
```

Figure 29. Example Code: DLM Key Injection – Part 1

```

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")
    # Start the data packet stage
    SOD=b'\x81' # Start of data packet
    LNH=b'\x00'
    LNL=b'\x55' # 85 byte: one byte (RES) + shared key ring number (4 bytes) + w-ufpk (32 bytes)
                #+ initialize vector (16 bytes) + encrypted dlm key and mac (32 bytes)
    RES=b'\x28' # OK byte
    SKR=b'\x00'*4 #Shared key ring number.
    SUM=calc_sum(LNH + LNL + RES + SKR + message_bytes [24:56] + message_bytes [56:72] + \
                message_bytes [72:72+encrypted_key_size])
    ETX = b'\x03'
    command = SOD + LNH + LNL + RES + SKR + message_bytes [24:56] + message_bytes [56:72] + \
                message_bytes [72:72+encrypted_key_size] + SUM + ETX

    print("\nSending DLM key injection data packet: ")
    print_bytes_hex(command)
    ser.write(command)
    time.sleep(LONG_DELAY)

    # Acquire the response of the data packet stage
    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("\nDLM key injection data packet - SUCCESS\n")
            print("Injecting AL2 key is successful")
        elif KYTY == b'\x02':
            print("\nDLM key injection data packet - SUCCESS\n")
            print("Injecting AL1 key is successful")

return RES

```

Figure 30. Example Code: DLM Key Injection – Part 2

5.6.2 Verify DLM Authentication Keys

After injecting the DLM AL keys, the production programming tools should invoke a verify command to confirm correct injection. Figure 31 shows the command packet information for verifying the DLM AL keys.

6.10.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	02h
CMD	(1 byte)	29h (Key verify command)
KYTY	(1 byte)	Key type: • 01h: AL2_KEY • 02h: AL1_KEY • 03h: RMA_KEY
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

Figure 31. DLM Key Verify Command Packet

```
# =====
# This routine demonstrates "DLM Key Verify Command".
# The supported key types are AL1 and AL2 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 AL2 key verify command:")
    elif KYTY == b'\x02':
        print("Sending DLM AL1 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 AL2 key is successful")
        elif KYTY == b'\x02':
            print("Verifying injected AL1 key is successful")

    return RES
```

Figure 32. DLM Authentication Key Verify Command Example Code

5.7 DLM State, Protection Level and Authentication Level Handling

This section explains the DLM state, Protection Level (PL) and Authentication Level (AL) request command and the non-authenticated DLM state, PL and AL transition command.

5.7.1 DLM State Request

The state request command is demonstrated in the included example code. Figure 33 shows the state request command packet format. Figure 34 shows the example code.

6.3.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	01h
CMD	(1 byte)	2Ch (DLM state request command)
SUM	(1 byte)	D3h
ETX	(1 byte)	03h

6.3.2.2 Data Packet [DLM State]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	02h
RES	(1 byte)	2Ch (OK)
DLM	(1 byte)	DLM state code
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

Figure 33. 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(LONG_DELAY)

    # Acquire the response
    return_packet = receive_data_packet()
    # The RES byte is the fourth index
    RES = return_packet[3]
    if RES != 0x2C:
        print("*** ERROR : 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("User must use the DLM state transition command to transition to OEM state")
            case 0x04:
                print("Current DLM state is OEM")
            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 0x09:
                print("Current DLM state is RMA_RET")
            case _:
                print("Unknown DLM state")

    return RES, DLM
```

Figure 34. DLM State Request Command Example Code

5.7.2 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 35. The current DLM state is a required parameter for the DLM state transition command and should be acquired first.

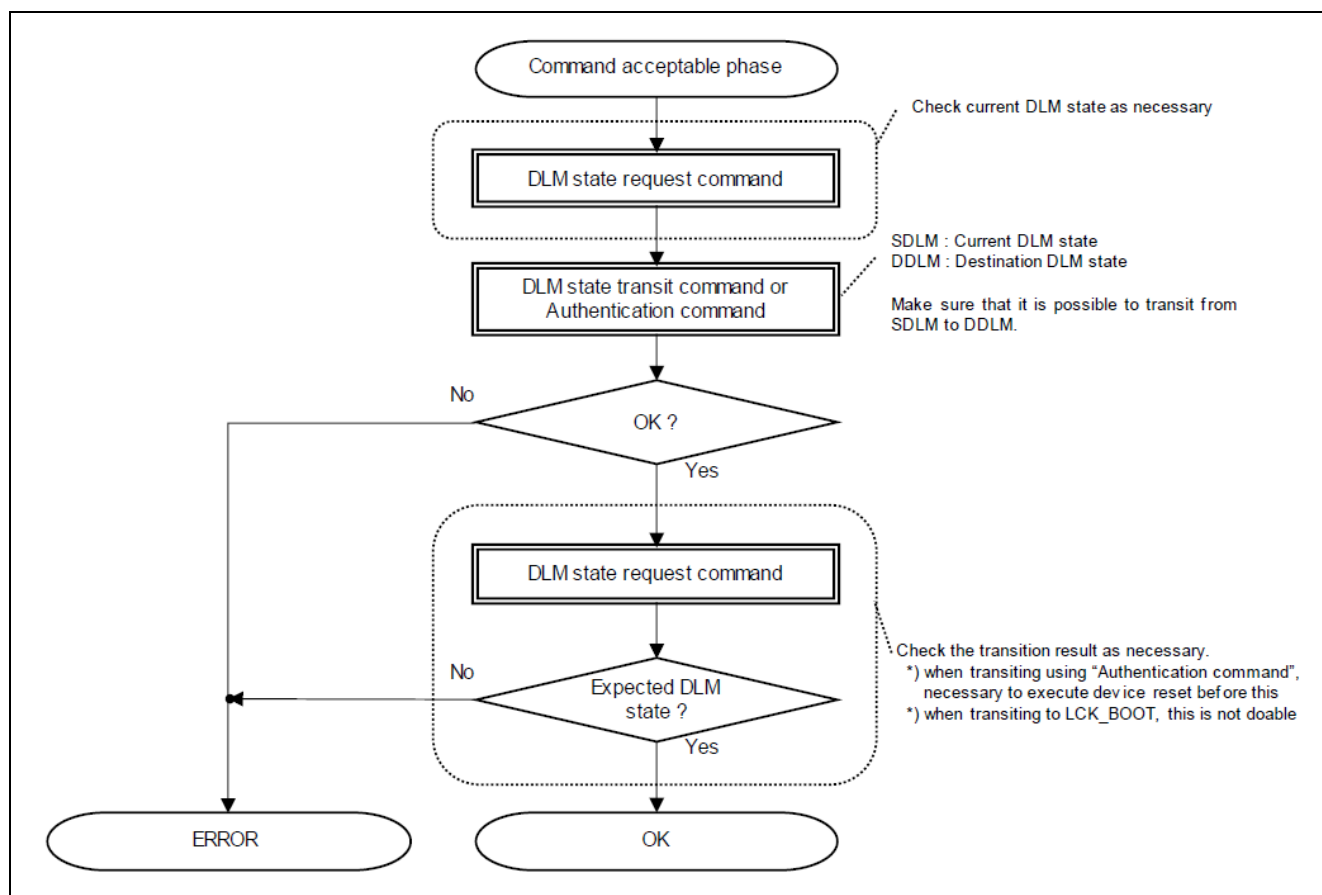


Figure 35. Recommended Flow for Performing DLM State Transition

6.2.1.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	03h
CMD	(1 byte)	71h (DLM state transit command)
SDLM	(1 byte)	Source DLM state code: • 01h: CM • 04h: OEM • 08h: RMA_ACK
DDLM	(1 byte)	Destination DLM state code: • 04h: OEM • 06h: LCK_BOOT • 09h: RMA_RET
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

6.2.1.2 Data Packet [Status OK]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	0Ah
RES	(1 byte)	71h (OK)
STS	(1 byte)	00h (OK)
ST2	(4 bytes)	FFFFFFFFh (unused code)
ADR	(4 bytes)	FFFFFFFFh (unused code)
SUM	(1 byte)	8Dh
ETX	(1 byte)	03h

Figure 36. DLM State Transition Command Packet

```
# =====
# This routine demonstrates the "DLM State Transit 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'\x06' \
            and source_DLM_state != b'\x07' and source_DLM_state != b'\x08' and \
            source_DLM_state != b'\x09':
            SOH=b'\x01'
            LNH=b'\x00'
            LNL=b'\x03'
            #0x71: DLM state transit command
            CMD=b'\x71'
            SDLM=source_DLM_state
            DDLM=target_DLM_state
            SUM=calc_sum(LNH + LNL + CMD + SDLM + DDLM)
            EXT=b'\x03'
            command = SOH + LNH + LNL + CMD + SDLM + DDLM + SUM + EXT

            print("\nSending 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")
            elif DDLM == b'\x04':
                print("DLM state changed to OEM.")
                # the STS is the fifth index
                STS = return_packet[4]
                if STS != 0x00:
                    print("DLM state Transition command - FAIL")

    return RES
```

Figure 37. Example Code for DLM State Transition Command

5.7.3 Protection Level Request

The PL request command is demonstrated in the included example code. Figure 38 shows the PL request command packet format. Figure 39 shows the example code.

6.5.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	01h
CMD	(1 byte)	73h (Protection level request command)
SUM	(1 byte)	8Ch
ETX	(1 byte)	03h

6.5.2.2 Data Packet [Protection Level]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	02h
RES	(1 byte)	73h (OK)
CPL	(1 byte)	Current PL code • 02h: Protection level 2 • 03h: Protection level 1 • 04h: Protection level 0
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

Figure 38. Protection Level Request Command Packet

```
# =====
# This routine demonstrates the "Protection Level Request Command".
# The command execution result and the Protection level are returned.
def command_ProtectionLevelRequest():
    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x01'
    CMD=b'\x73' # Protection level request command code
    SUM=b'\x8C'
    EXT=b'\x03'
    command = SOH + LNH + LNL + CMD + SUM + EXT

    print("Sending Protection Level Request 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 != 0x73:
        print("*** ERROR : Read Protection Level - FAIL")
        CPL = b'\xFF'
    else:
        # The CPL byte is the fifth index
        CPL = return_packet[4]
        match CPL:
            case 0x02:
                print("Current Protection Level (PL) is PL2")
            case 0x03:
                print("Current Protection Level (PL) is PL1")
            case 0x04:
                print("Current Protection Level (PL) is PL0")
            case _:
                print("Unknown Protection Level")

    return RES, CPL
```

Figure 39. Protection Level Request Command Example Code

5.7.4 Protection Level Transition

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

The recommended flow when performing PL transition is described in Figure 40. The current DLM state, PL and AL are necessary for the PL transition command and should be acquired first.

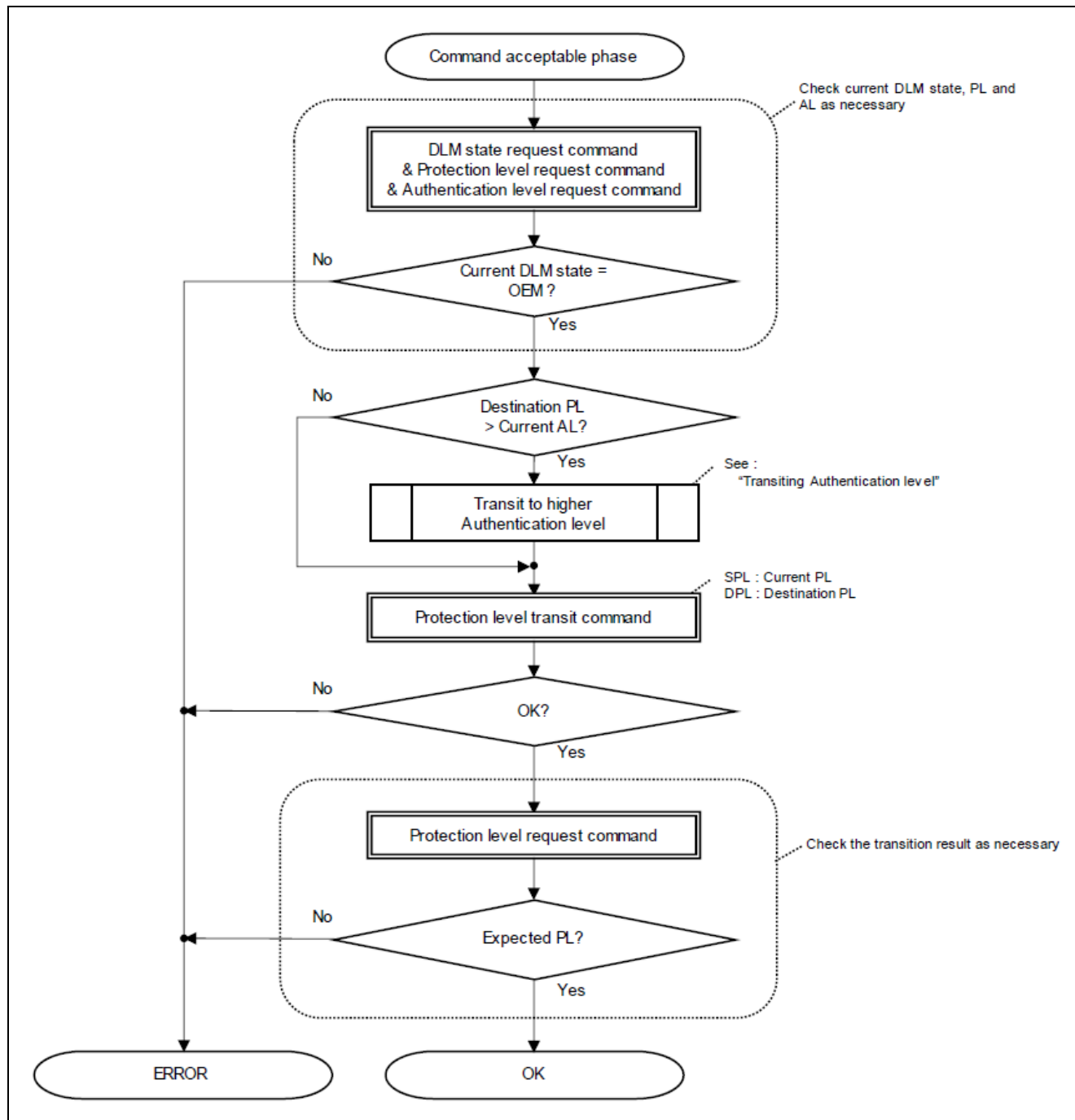


Figure 40. Recommended Flow for Performing Protection Level Transition

6.4.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	03h
CMD	(1 byte)	72h (Protection level transit command)
SPL	(1 byte)	Source PL code: • 02h: Protection level 2 • 03h: Protection level 1 • 04h: Protection level 0
DPL	(1 byte)	Destination PL code: • 02h: Protection level 2 • 03h: Protection level 1 • 04h: Protection level 0
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

6.4.2.2 Data Packet [Status OK]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	0Ah
RES	(1 byte)	72h (OK)
STS	(1 byte)	00h (OK)
ST2	(4 bytes)	FFFFFFFFh (unused code)
ADR	(4 bytes)	FFFFFFFFh (unused code)
SUM	(1 byte)	8Ch
ETX	(1 byte)	03h

Figure 41. Protection Level Transition Command Packet

```
# =====
# This routine demonstrates the "Protection Level Transit Command".
def command_ProtectionLevelTransition(target_PL_state):
    RES, DLM = command_DLMstateRequest()

    if RES != 0x2C:
        print("Read DLM state - FAIL")
        return RES
    elif DLM != 0x04:
        print("DLM state is not OEM, cannot proceed")
        return 0xFF

    RES, CPL = command_ProtectionLevelRequest()

    if RES != 0x73:
        print("Read Protection Level - FAIL")
        return RES

    target_PL_state = int.from_bytes(target_PL_state, "big")
    if target_PL_state < CPL:
        print("Protection Level upgrade -> Checking Authentication Level")
        RES, CAL = command_AuthenticationLevelRequest()
        if RES != 0x75:
            print("Read Authentication Level - FAIL")
            return RES
        elif CAL != 0x02:
            print("Transiting to AL2")
            RES = command_AuthenticationLevelTransition(b'\x02')
            if RES != 0x30:
                print("Authentication Level Transition - FAIL")
                return RES
```

Figure 42. Example Code for Protection Level Transition Command – part 1

```
# Transition to the target Protection Level
SOH=b'\x01'
LNH=b'\x00'
LNL=b'\x03'
#0x72: Protection Level transition command
CMD=b'\x72'
SDLM=CPL.to_bytes(1, "big")
DDLML=target_PL_state.to_bytes(1, "big")
SUM=calc_sum(LNH + LNL + CMD + SDLM + DDLML)
EXT=b'\x03'
command = SOH + LNH + LNL + CMD + SDLM + DDLML + SUM + EXT

print("\nSending Protection Level 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 != 0x72:
    print("Protection Level Transition command - FAIL")
else:
    if DDLML == b'\x02':
        print("Protection Level changed to PL2.")
    elif DDLML == b'\x03':
        print("Protection Level changed to PL1.")
    elif DDLML == b'\x04':
        print("Protection Level changed to PL0.")

return RES
```

Figure 43. Example Code for Protection Level Transition Command – part 2

5.7.5 Authentication Level Request

The AL request command is demonstrated in the included example code. Figure 44 shows the AL request command packet format. Figure 45 shows the example code.

6.6.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	01h
CMD	(1 byte)	75h (Authentication level request command)
SUM	(1 byte)	8Ah
ETX	(1 byte)	03h

6.6.2.2 Data Packet [Authentication Level]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	02h
RES	(1 byte)	75h (OK)
CAL	(1 byte)	Current AL code • 02h: Authentication level 2 • 03h: Authentication level 1 • 04h: Authentication level 0
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

Figure 44. Authentication Level Request Command Packet

```
# =====
# This routine demonstrates the "Authentication Level Request Command".
# The command execution result and the Authentication level are returned.
def command_AuthenticationLevelRequest():
    SOH=b'\x01'
    LNH=b'\x00'
    LNL=b'\x01'
    CMD=b'\x75' # Authentication level request command code
    SUM=b'\x8A'
    EXT=b'\x03'
    command = SOH + LNH + LNL + CMD + SUM + EXT

    print("Sending Authentication Level Request 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 != 0x75:
        print("*** ERROR : Read Authentication Level - FAIL")
        CAL = b'\xFF'
    else:
        # The AL byte is the fifth index
        CAL = return_packet[4]
        match CAL:
            case 0x02:
                print("Current Authentication Level (AL) is AL2")
            case 0x03:
                print("Current Authentication Level (AL) is AL1")
            case 0x04:
                print("Current Authentication Level (AL) is AL0")
            case _:
                print("Unknown Authentication Level")

    return RES, CAL
```

Figure 45. Authentication Request Command Example Code

5.7.6 Authentication Level Transition

This section covers the AL transition. Authenticated transitions are not generally required in production programming tools.

The recommended flow when performing AL transition is described in Figure 46. The current DLM state and AL are necessary for the AL transition command and should be acquired first.

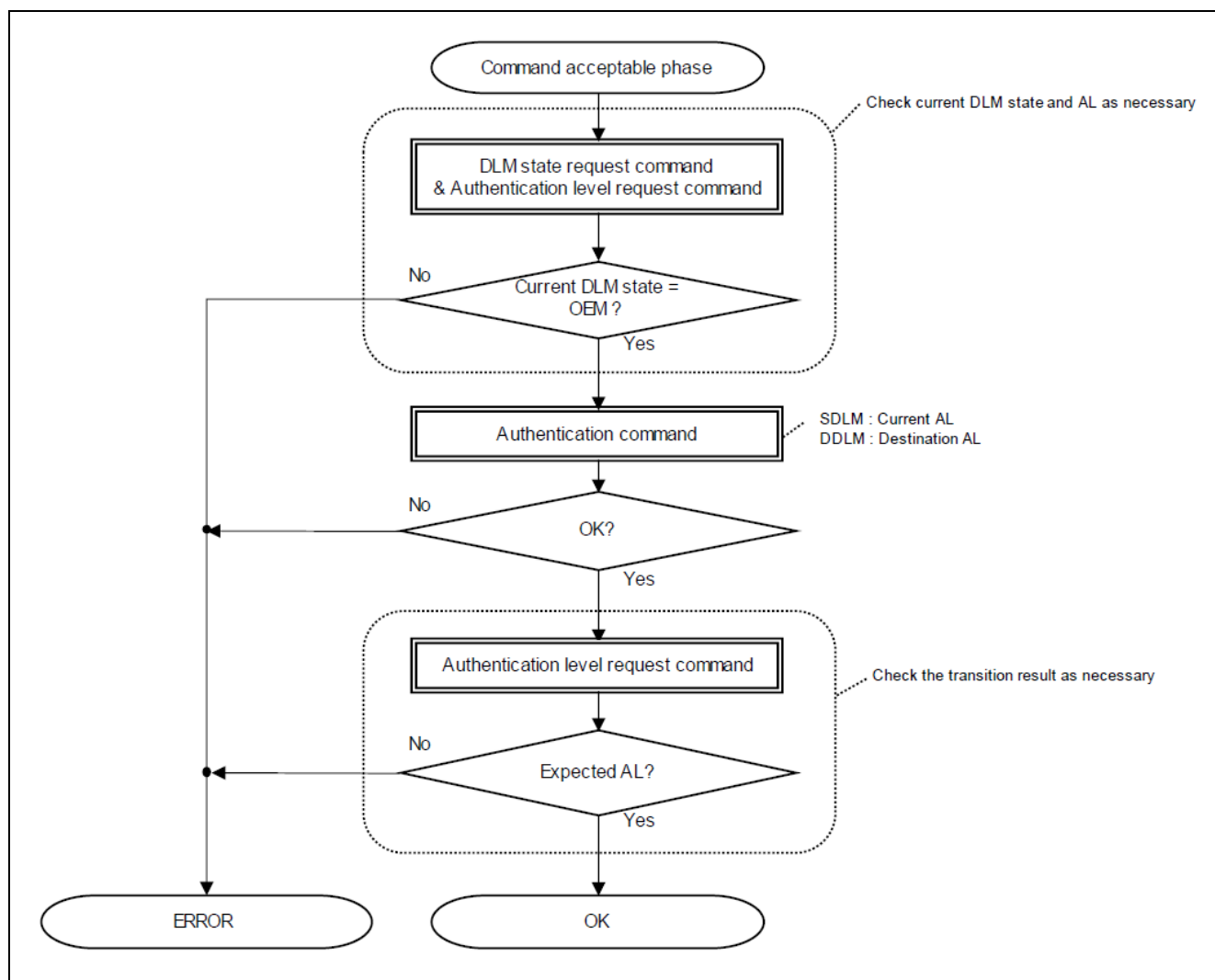


Figure 46. Recommended Flow for Performing Authentication Level Transition

6.7.2.1 Command Packet

SOH	(1 byte)	01h
LNH	(1 byte)	00h
LNL	(1 byte)	04h
CMD	(1 byte)	30h (Authentication command)
SDLM	(1 byte)	Source DLM/AL code. For DLM transitions: • 04h: OEM • 07h: RMA_REQ For AL transitions: • 03h: AL1 • 04h: AL0
DDLML	(1 byte)	Destination DLM/AL code. For DLM transitions: • 07h: RMA_REQ • 08h: RMA_ACK For AL transitions: • 02h: AL2 • 03h: AL1
CHCT	(1 byte)	Authentication type: • 00h: Random number (Can be used in all transit cases.) • 01h: MCU unique ID (Can be used only in transit to RMA_REQ.)
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

6.7.2.2 Data Packet [Challenge Value or Unique ID]

SOD	(1 byte)	81h
LNH	(1 byte)	00h
LNL	(1 byte)	11h
RES	(1 byte)	30h (OK)
CHCD	(16 bytes)	Challenge value or Unique ID For example: 01234567_89AB ... 2233_44556677h -> 01h, 23h, 45h, ... , 55h, 66h, 77h
SUM	(1 byte)	Sum data
ETX	(1 byte)	03h

Figure 47. Authentication Level Transition Command Packet

```
# =====
# This routine demonstrates the "Authentication Level Transit Command".
def command_AuthenticationLevelTransition(target_AL_state):
    RES, DLM = command_DLMstateRequest()

    if RES != 0x2C:
        print("Read DLM state - FAIL")
        return RES
    elif DLM != 0x04:
        print("DLM state is not OEM, cannot proceed")
        return 0xFF

    RES, CAL = command_AuthenticationLevelRequest()

    if RES != 0x75:
        print("Read Authentication Level - FAIL")
    elif CAL != int.from_bytes(target_AL_state, "big"):
        SOH=b'\x01'
        LNH=b'\x00'
        LNL=b'\x04'
        #0x30: Authentication command
        CMD=b'\x30'
        SDLM=CAL.to_bytes(1, "big")
        DDLM=target_AL_state
        CHCT=b'\x00'
        SUM=calc_sum(LNH + LNL + CMD + SDLM + DDLM + CHCT)
        EXT=b'\x03'
        command = SOH + LNH + LNL + CMD + SDLM + DDLM + CHCT + SUM + EXT

    print("\nSending DLM State Transition command:")
    print_bytes_hex(command)
    ser.write(command)
    time.sleep(LONG_DELAY)
```

Figure 48. Example Code for Authentication Level Transition Command – Part 1

```

# Acquire the response
return_packet = receive_data_packet()
# The RES byte is the fourth index
RES = return_packet[3]
if RES != 0x30:
    print("Authentication Transition command - FAIL")
else:
    challenge_data = return_packet[4:20]
    print("Challenge data : ")
    print_bytes_hex(challenge_data)
    # Calculate the response value using AES-128 CMAC
    response_data = calc_response_value(challenge_data, b'\x00'*16)
    print("Response data : ")
    print_bytes_hex(response_data)

# Send Data Packet with Response Value
SOD=b'\x81'
LNH=b'\x00'
LNL=b'\x21'
RES=b'\x30'
MAC=response_data
SUM=calc_sum(LNH + LNL + RES + MAC)
ETX = b'\x03'
command = SOD + LNH + LNL + RES + MAC + SUM + ETX

print("\nSending Authentication Response Value:")
print_bytes_hex(command)
ser.write(command)
time.sleep(AL_DATA_RESPONSE_DELAY)

# Acquire the response
return_packet = receive_data_packet()
# The RES byte is the fourth index
RES = return_packet[3]
# The STS is the fifth index
STS = return_packet[4]
if RES != 0x30 and STS != 0x00:
    print("Authentication Response Value - FAIL")
else:
    if DDLM == b'\x02':
        print("AL state changed to AL2.")
    elif DDLM == b'\x03':
        print("AL state changed to AL1.")

return RES

```

Figure 49. Example Code for Authentication Level Transition Command – Part 2

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>)
- Install PyCryptodome package
 - PyCryptodome ([Installation — PyCryptodome 3.4.6 documentation](#))

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 RA8M1/RA8P1 MCU fitted to an EK-RA8M1/EK-RA8P1 evaluation board.

To cause the RA8M1 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, as shown in Figure 50.

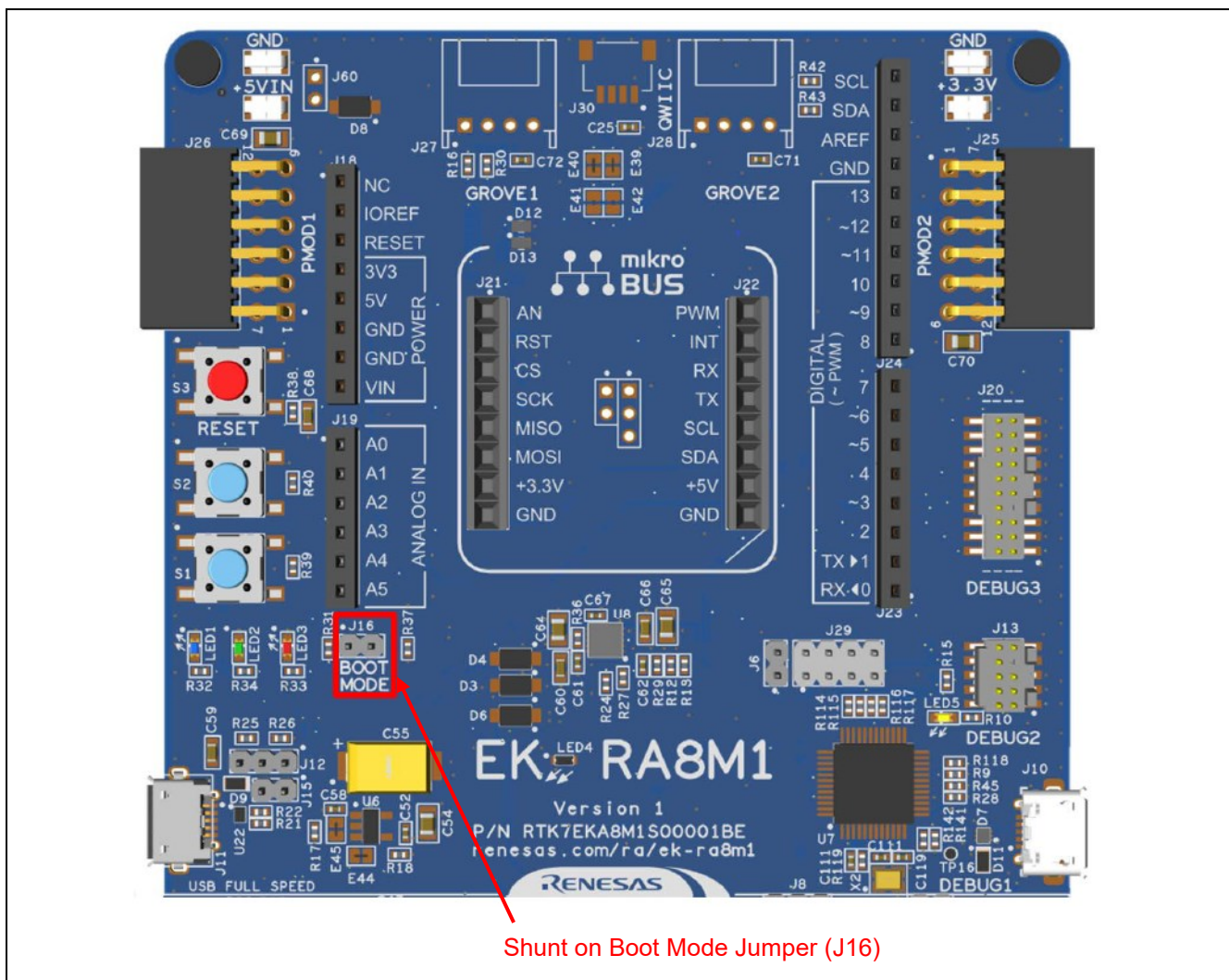


Figure 50. Shunt the MD Pin Jumper on EK-RA8M1

To cause the RA8P1 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 pins 1-2, as shown in Figure 51.

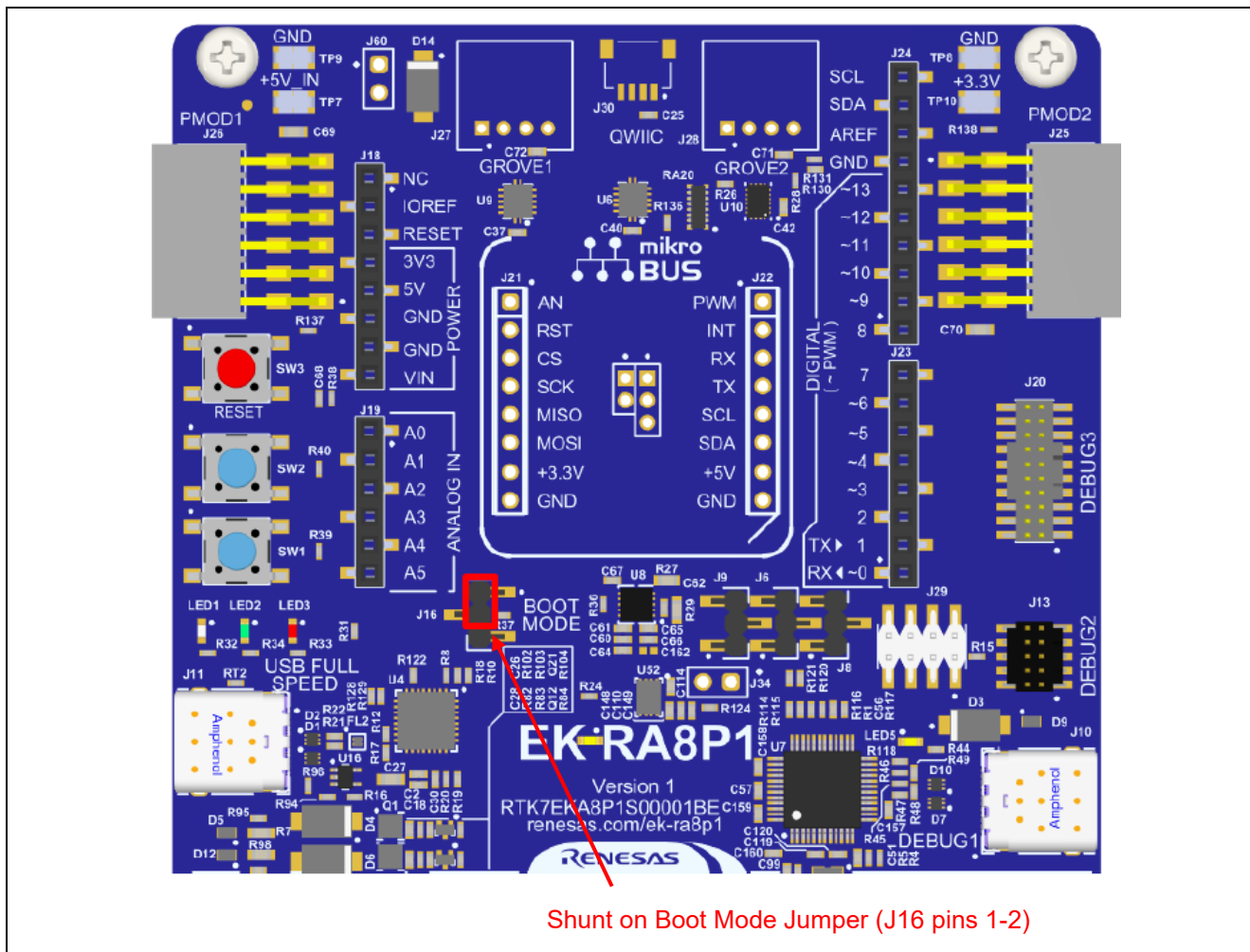


Figure 51. Shunt the MD Pin Jumper on EK-RA8P1

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-RA8M1/EK-RA8P1 to the development PC to provide USB Device connection.
- See Table 8 for more general details on the USB connection.

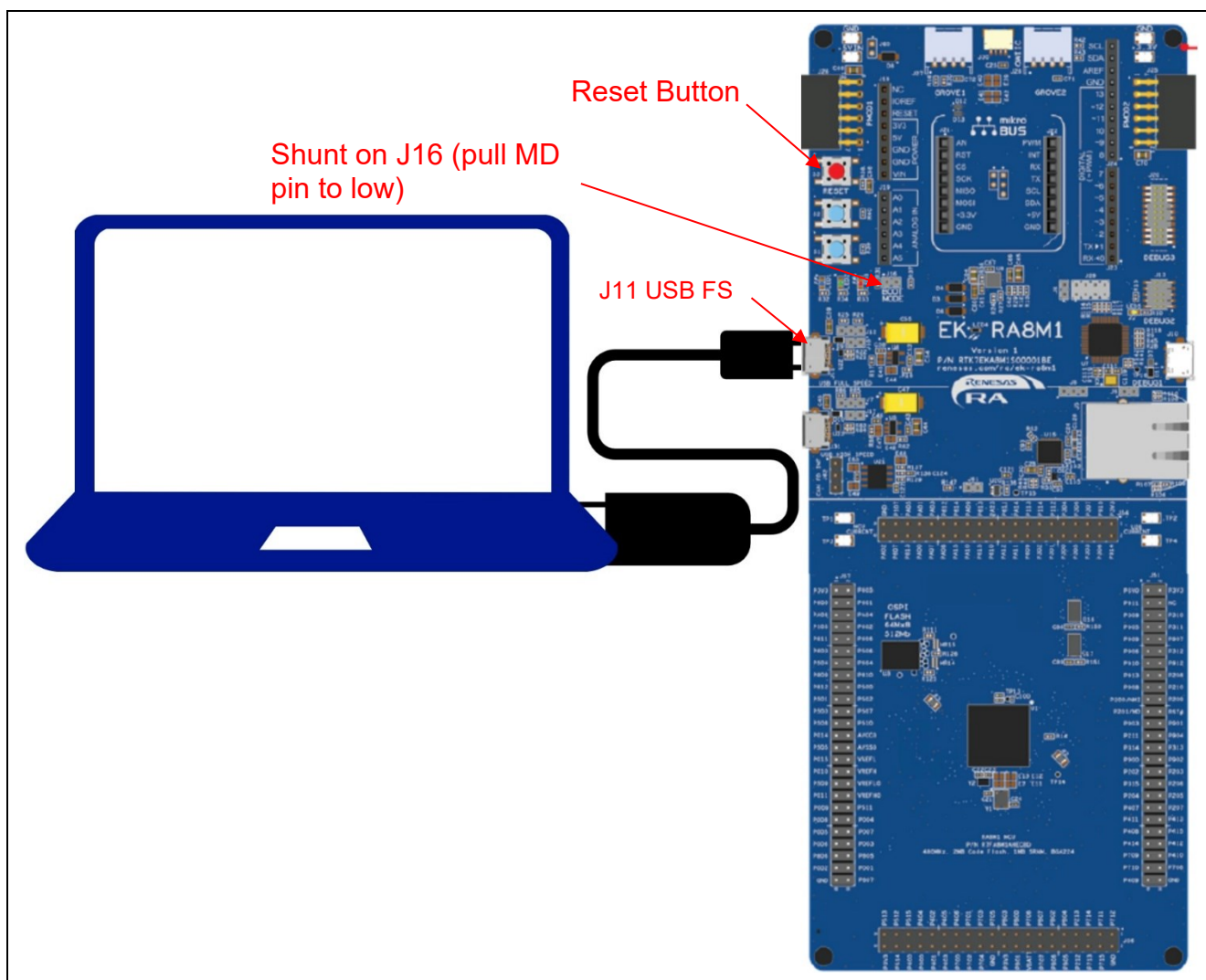


Figure 52. Hardware Setup using USB Full Speed Port

If the serial interface is used:

- Connect the four pins in Table 14 on the UART to USB converter to the EK-RA8M1 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 53, 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-RA8M1	EK-RA8P1
RX	J51: Pin 20 (MCU P209 (TXD9))	J2: Pin 21 (MCU P209 (TXD9))
TX	J51: Pin 16 (MCU P208 (RXD9))	J2: Pin 37 (MCU P208 (RXD9))
+5V (FTDI cable power output voltage. Check the voltage output on the converter used.)	J51: Pin 01 (If +3.3V is provided from the converter, then connect to Pin 02 of J51). If the production programming board has stable power supply, then this pin connection is not needed.	J1: Pin 08 (If +3.3V is provided from the converter, then connect to Pin 01 of J2). If the production programming board has stable power supply, then this pin connection is not needed.
GND	J51: Pin 50 (MCU Ground)	J2: Pin 39 (MCU Ground)

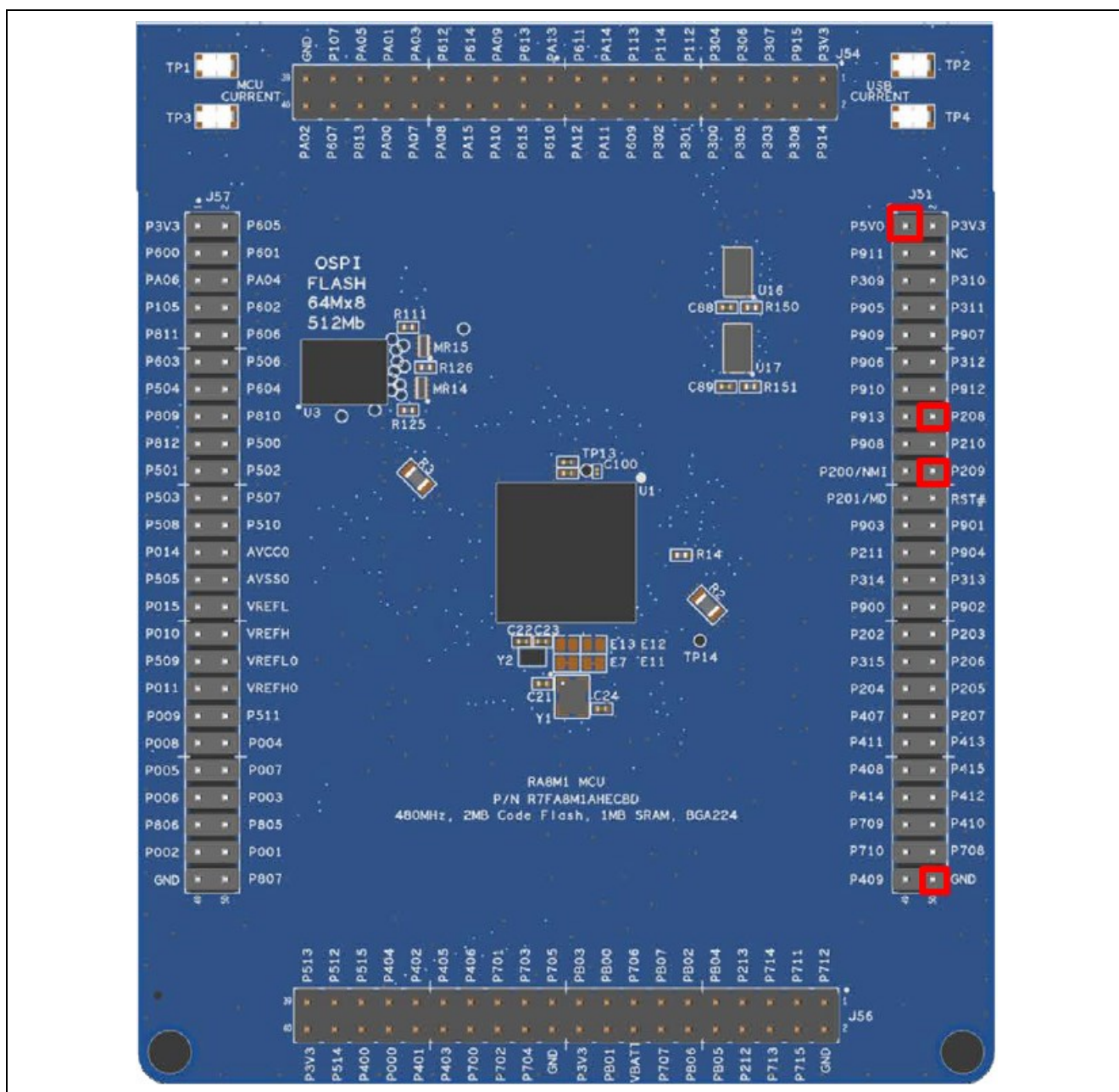


Figure 53. Hardware Setup using UART to USB Converter for EK-RA8M1

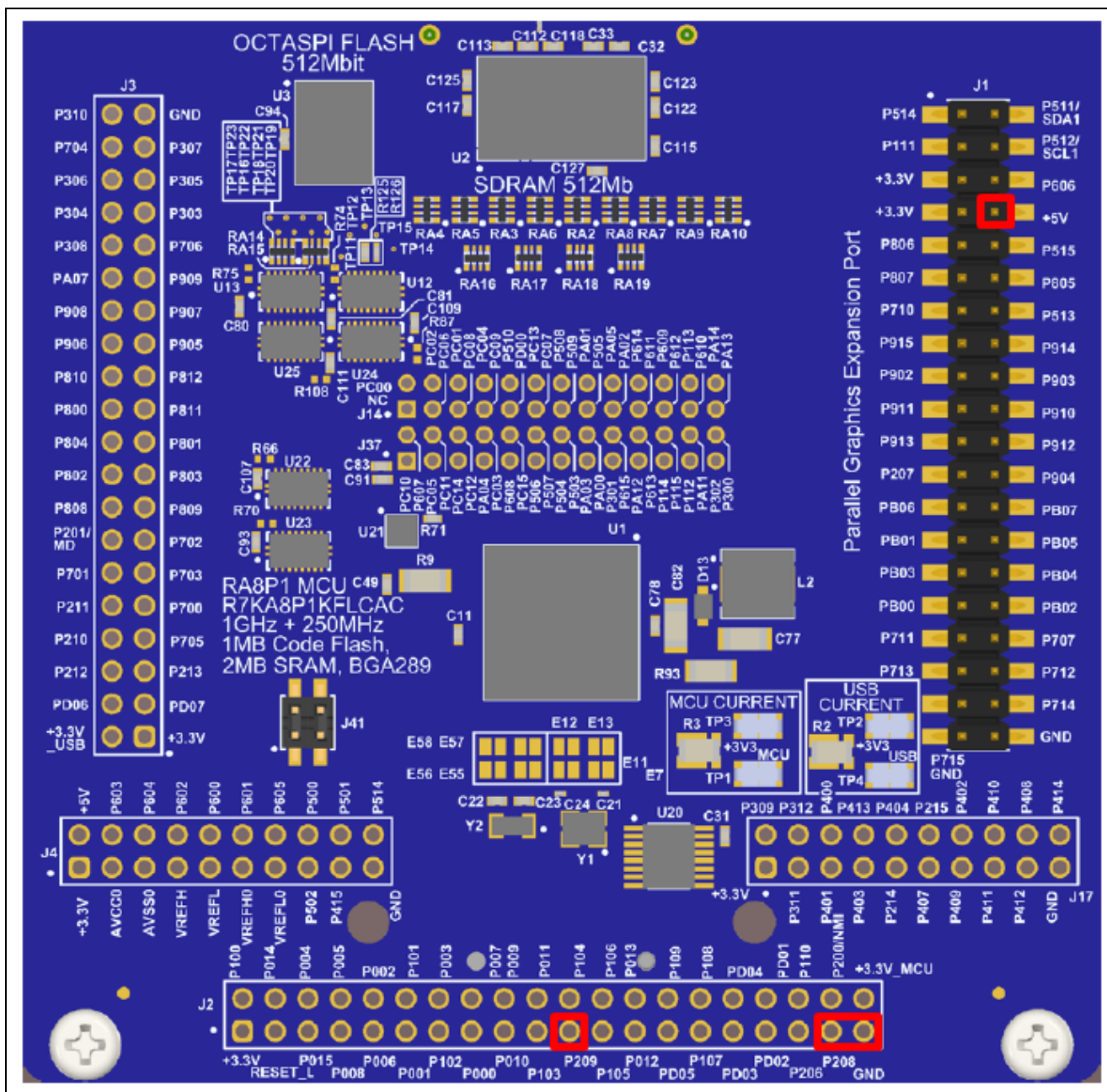


Figure 54. Hardware Setup using UART to USB Converter for EK-RA8P1

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_cm85_dlm.zip` to reveal two Python files and the `\dlm_keys` folder which holds four DLM AL keys: two for RA8M1, two for RA8P1 which can be injected into the MCU.

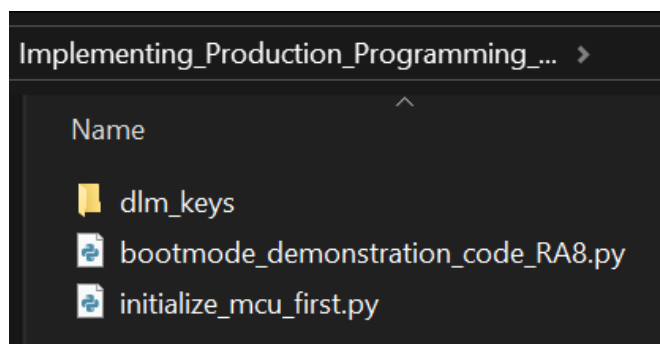


Figure 55. Python Demo Code and Sample DLM Keys

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 56:

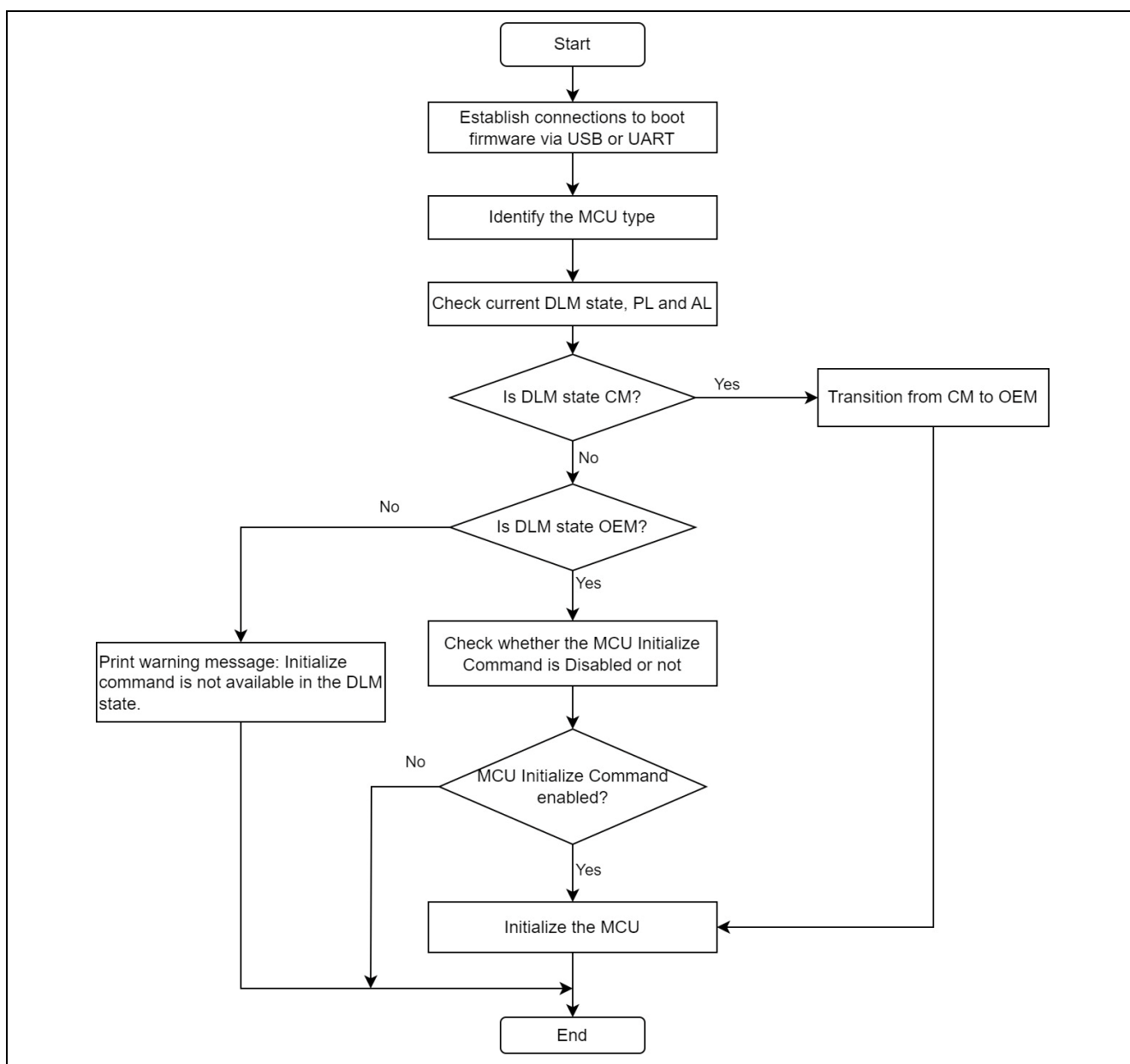


Figure 56. Operational Flow of the First Demo Code

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 57 and Figure 58 shows sample output from running the demonstration with a USB connection to the board.

```
C:\Workspace\02_AP\Production_Programming>python initialize_mcu_first.py

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

COM26 - RA USB Boot(CDC) (COM26)
COM3 - Intel(R) Active Management Technology - SOL (COM3)

Auto-selected COM26 for RA Boot mode USB CDC connection

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

Com port opened : COM26

=====
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
CM85/CM33 boot code received

=====
Requesting Product Type Name
=====

Sending Signature Request command:
b'\x01\x00\x01\x3a\xc5\x03'
Signature Request - SUCCESS
Product Type Name: R7FA8M1AHECBD
```

Figure 57. Demonstration with USB Connection – Part 1

```
=====
Requesting the DLM, PL and AL states
=====

Sending DLM State Request command:
b'\x01\x00\x01\x2c\xd3\x03'
Current DLM state is OEM
Sending Protection Level Request command:
b'\x01\x00\x01\x73\x8c\x03'
Current Protection Level (PL) is PL2
Sending Authentication Level Request command:
b'\x01\x00\x01\x75\x8a\x03'
Current Authentication Level (AL) is AL2

=====
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\x04\x04\xa5\x03'
Initialize - SUCCESS

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

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

Figure 58. Demonstration with USB Connection – Part 2

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 OEM, 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 OEM is a one-way process.

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 58.

```
C:\Workspace\02_AP\Production_Programming>python initialize_mcu_first.py

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

COM4 - Silicon Labs CP210x USB to UART Bridge (COM4)
COM3 - Intel(R) Active Management Technology - SOL (COM3)

Please enter com port to use : COM4

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

Com port opened : COM4

=====
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
CM85/CM33 boot code received
```

Figure 59. Demonstration with Serial Connection

6.4 Running the Second Demonstration Code

The second of the demonstration examples, `bootmode_demonstration_code_RA8.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 60.

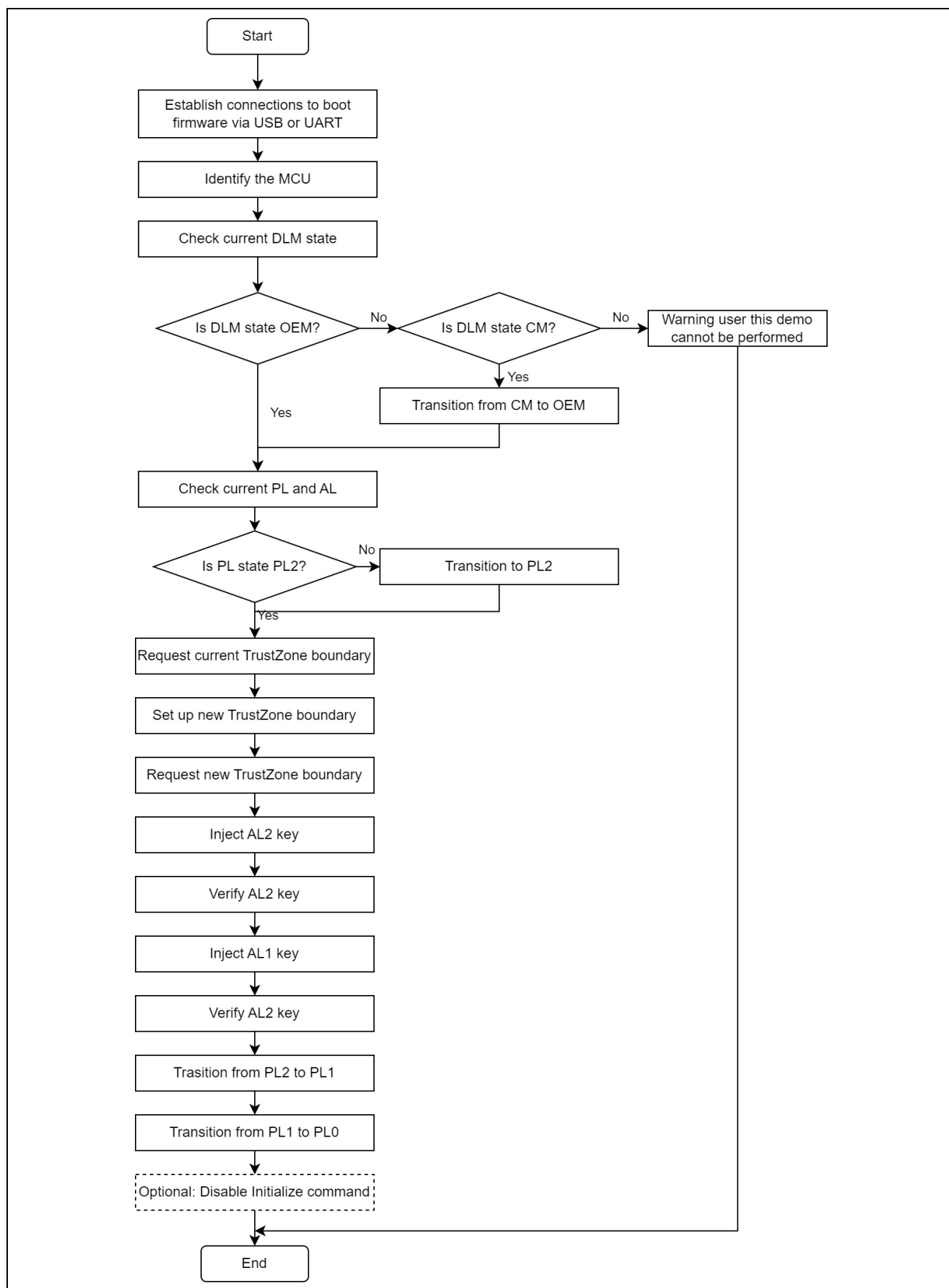


Figure 60. Operational Flow of the Second Demonstration Code

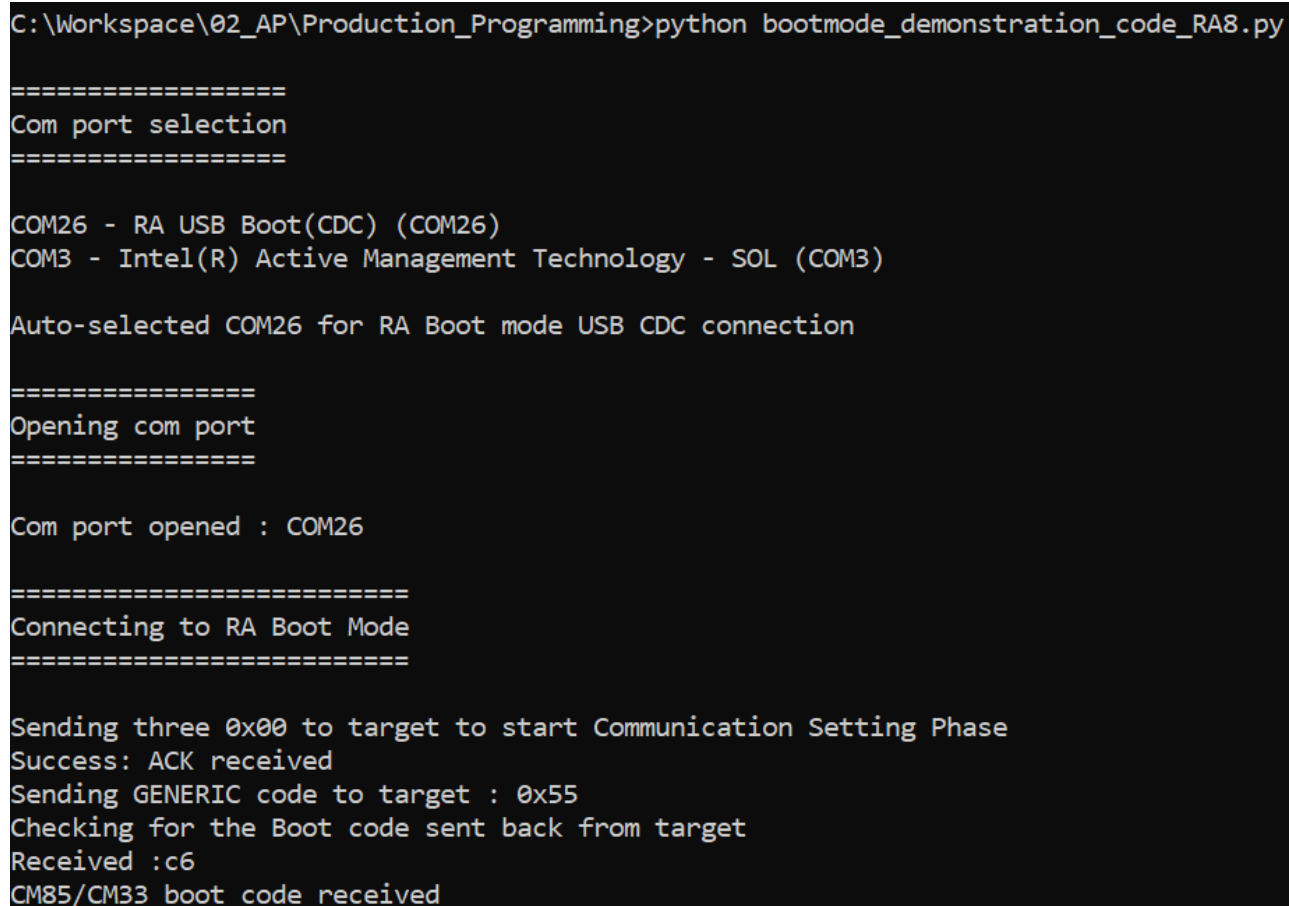
To execute the example, ensure that you have reset the board, then enter the following command in the previously opened command prompt:

```
python bootmode_demonstration_code_RA8.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)



```
C:\Workspace\02_AP\Production_Programming>python bootmode_demonstration_code_RA8.py

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

COM26 - RA USB Boot(CDC) (COM26)
COM3 - Intel(R) Active Management Technology - SOL (COM3)

Auto-selected COM26 for RA Boot mode USB CDC connection

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

Com port opened : COM26

=====
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
CM85/CM33 boot code received
```

Figure 61. Establishing the Connection USB

If a serial connection is used, it will be necessary to enter the COM port manually, as shown in Figure 59.

6.4.2 Checking Product Type Name, Current DLM State, Protection Level and Authentication Level

```
=====
Requesting Product Type Name
=====

Sending Signature Request command:
b'\x01\x00\x01\x3a\xc5\x03'
Signature Request - SUCCESS
Product Type Name: R7FA8M1AHECBD

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

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

=====
Requesting the Protection and Authentication Level
=====

Sending Protection Level Request command:
b'\x01\x00\x01\x73\x8c\x03'
Current Protection Level (PL) is PL2
Sending Authentication Level Request command:
b'\x01\x00\x01\x75\x8a\x03'
Current Authentication Level (AL) is AL2
```

Figure 62. Checking Product Type Name, Current DLM State, PL and AL

6.4.3 Configuring TrustZone Partition Boundaries

```
=====
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 is 16352 KB
- The secure data flash region size is 63 KB

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

Configuring device with :
- 512KB of Code Flash Secure region
- 4KB of Data Flash Secure region
Sending TrustZone boundary setup command:
b'\x01\x00\x0b\x4e\x00\x00\x02\x00\x00\x04\x00\x00\x00\x00\xa1\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 is 512 KB
- The secure data flash region size is 4 KB
```

Figure 63. Configuring TrustZone Partition Boundaries

6.4.4 Injecting DLM AL Keys

```
Injecting DLM - AL2 Key
=====
Start Processing AL2 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\x55\x28\x00\x00\x00\x00\x2a\x84\x34\xca\x97\xd0\x31\x32\x79\x38\x9d\xd8\xf1\x55\x23\xdb\x2a\x84\x34\xca\x97\xd0\x31\x32\x79\x38\x9d\xd8\xf1\x55\x23\xdb\xe2\x03\xad\x93\x9c\xe4\xcc\x00\x56\xfe\x67\x0f\xb5\x9c\x36\x6f\x39\x73\xf9\xab\xf0\x76\xea\x54\xac\x04\x23\x9e\xfe\x5f\x71\xa8\x0c\x5c\x94\x84\x70\x50\x48\xaf\x2e\xdf\xff\xb7\xac\x49\xf7\x56\xbb\x03'

DLM key injection data packet - SUCCESS

Injecting AL2 key is successful

=====
Verifying the AL2 Key
=====

Sending DLM AL2 key verify command:
b'\x01\x00\x02\x29\x01\xd4\x03'
Verifying injected AL2 key is successful

=====
Injecting DLM - AL1 Key
=====

Start Processing AL1 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\x55\x28\x00\x00\x00\x00\x2a\x84\x34\xca\x97\xd0\x31\x32\x79\x38\x9d\xd8\xf1\x55\x23\xdb\x2a\x84\x34\xca\x97\xd0\x31\x32\x79\x38\x9d\xd8\xf1\x55\x23\xdb\x5b\x90\x91\x3b\xcf\x42\xef\x4e\xce\xfc\xb5\x05\x03\x7f\xb0\xa5\xa3\xc5\x9c\x4a\x51\x7a\x0a\x06\xe4\xff\xcc\x35\x34\x27\x70\x80\xb3\x95\xeb\x56\xaf\x32\xea\xc8\xd2\x70\x9a\x8f\x19\x6e\x63\x6f\x9b\x03'

DLM key injection data packet - SUCCESS

Injecting AL1 key is successful

=====
Verifying the AL1 Key
=====

Sending DLM AL1 key verify command:
b'\x01\x00\x02\x29\x02\xd3\x03'
Verifying injected AL1 key is successful
```

Figure 64. Injecting DLM AL 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.

6.4.5 Configuring Final Protection Level State

```
=====
Transitioning Protection Level from PL2 to PL1
=====

Sending DLM State Request command:
b'\x01\x00\x01\x2c\xd3\x03'
Current DLM state is OEM
Sending Protection Level Request command:
b'\x01\x00\x01\x73\x8c\x03'
Current Protection Level (PL) is PL2

Sending Protection Level Transition command:
b'\x01\x00\x03\x72\x02\x03\x86\x03'
Protection Level changed to PL1.

=====
Transitioning Protection Level from PL1 to PL0
=====

Sending DLM State Request command:
b'\x01\x00\x01\x2c\xd3\x03'
Current DLM state is OEM
Sending Protection Level Request command:
b'\x01\x00\x01\x73\x8c\x03'
Current Protection Level (PL) is PL1

Sending Protection Level Transition command:
b'\x01\x00\x03\x72\x03\x04\x84\x03'
Protection Level changed to PL0.

=====
Demonstration finished.
=====
```

Figure 65. Configuring Protection Level 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/PL transitions by referring Authenticated Transition using RFP section from Application Note [R11AN0785](#) to perform the authenticated transitions using the Renesas Flash Programmer:

- This step is typically not needed in a production programming environment. Perform PL0 to PL1 or PL2 transition following case 1 or case 3.
The plaintext raw AL2 key value for the example `AL2.rkey` and `AL2_RA8P1.rkey` files is “000102030405060708090A0B0C0D0E0F”. This value needs to be used when transitioning from the PL0 state to the PL1 or PL2 state.
- Perform PL1 to PL2 transition following case 2.
The plaintext raw AL1 key value for the example `AL1.rkey` and `AL1_RA8P1.rkey` files is “010102030405060708090A0B0C0D0E0F”. This value needs to be used when transitioning from the PL1 state to the PL2 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_RA8.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 OEM 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_INITIALIZE_COMMAND` from 0 to 1.

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

```
=====
Disabling ability to invoke Initialize command
=====

!!! ===== WARNING ===== !!!
!!! Executing 'command_disable_initialize ()' will prevent the !!!
!!! Initialize command from working in the future.             !!!

Enter YES if you are really sure you want to do this: YES
You entered YES - calling function

Sending Disable Initialize command:
b'\x01\x00\x03\x51\x01\x00\xab\x03'

Warning : After MCU Initialize is disabled, it can never be re-enabled.
Disable Initialize command - SUCCESS

=====
Demonstration finished.
=====
```

Figure 66. Disable the Initialize Command

The next time that `initialize_mcu_first.py` is run, the system will report that the Initialize command is disabled. In this case, the only way to partially recover the board is to use the injected AL keys to move the PL state back to PL2 as explained in section 6.5.

7. References

- Renesas RA Family Device Lifecycle Management for RA8 MCUs ([R11AN0785](#))
- Renesas RA Family Renesas Boot Firmware for RA8M1 MCU Group ([R01AN7140](#))
- Renesas RA Family Renesas Boot Firmware for RA8D1 MCU Group ([R01AN7290](#))
- Renesas RA Family Renesas Boot Firmware for RA8T1 MCU Group ([R01AN7291](#))
- Renesas RA Family Renesas Boot Firmware for RA8E1 MCU Group ([R01AN7535](#))
- Renesas RA Family Renesas Boot Firmware for RA8E2 MCU Group ([R01AN7547](#))
- Renesas RA Family Renesas Boot Firmware for RA8P1 MCU Group ([R01AN7823](#))
- Renesas RA Family Renesas Boot Firmware for RA8T2 MCU Group
- Renesas RA Family RA8M1 User's Manual: Hardware ([R01UH0994](#))
- Renesas RA Family RA8D1 User's Manual: Hardware ([R01UH0995](#))
- Renesas RA Family RA8T1 User's Manual: Hardware ([R01UH1016](#))
- Renesas RA Family RA8E1 User's Manual: Hardware ([R01UH1129](#))
- Renesas RA Family RA8E2 User's Manual: Hardware ([R01UH1130](#))
- Renesas RA Family RA8P1 User's Manual: Hardware ([R01UH1064](#))
- Renesas RA Family RA8T2 User's Manual: Hardware ([R01UH1067](#))
- Renesas RA Family RA8 Quick Design Guide ([R01AN7087](#))
- Security Key Management Tool User's Manual ([R20UT5349](#))

8. Website and Support

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

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	Jun.30.25	—	First release of this document.
1.01	Oct.08.25	—	Update minor contents.

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. RENASAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENASAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENASAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION ("Vulnerability Issues"). RENASAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENASAS 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/.