Renesas RA Family

Virtual EEPROM Example Using RA4E2

How to Use This Document

This document is intended for audiences with various levels of base knowledge, experience, and confidence. It has been broken up into four main sections:

1. **Introduction** – This is comprised of the introductory pages (before the Table of Contents) and is intended for audiences who need an introduction to Virtual EEPROM (VEEPROM) technology, its intended use, and what is required to work through the example project as written.
2. **Background** – This is comprised of Section 1, which talks extensively about the reasons for use and the functionality of VEEPROM. It is intended for those deciding if VEEPROM is right for their system, or those curious about the technology.
3. **Example** – Section 2 walks the reader through an example use of VEEPROM by first creating a working application based on the standard “Blinky” app that is generated by the FSP code generator. The “Blinky” project is modified to resemble a production application, then VEEPROM technology is implemented to illustratively enhance that application. Section 2 includes both instructions to create a “clean piece of paper” building of that app, and how to add VEEPROM to it.

   Accompanying this application note are two attached project files. One project is the completed application project, for user reference. The other project contains a “base” version of the production application that is functionally complete, awaiting only the VEEPROM components to be added. The readers who would prefer to dive directly into applying VEEPROM to the base app should start with section 2.4.3.
4. **Reference** – For those readers who already have a working app and only want to have the technical information to execute it on their own the Reference Section plus Sections 3-6 address that need.

Introduction [Why Virtual EEPROM?]

Often devices need to hold values that are unique to each manufactured system. The information can range from device ID/serial number or MAC address, which are typically written once and read many times, to calibration data or served bus addresses, which are written rarely and read often. The data must be retained across the power-cycling of the system.

Depending on the frequency of write, size of data, and required retention time, the solution to this design challenge can range from uniquely linked values written into flash at manufacturing, to physical switches/jumpers on the PCB, to I2C/SPI connected storage devices designed for such data storage (some connected devices are so elaborate that they contain SRAM that is battery backed-up, and “shadow” EEPROM that can be periodically updated to limit the write cycles).

Each of these solutions has an advantage and disadvantage, and a cost associated with the solution. With the Renesas RA family, and its integral Data Flash area, there is a more cost-effective solution for rarely written/often read, non-volatile data requirements.

The RA family’s Data Flash, and the associated FSP have support for a virtual form of EEPROM (VEEPROM – flash-based storage that “looks and acts” like EEPROM). This application note walks the reader through the process of including, configuring, and using the VEEPROM hardware and firmware resources.

Target Device

RA4E2 (to follow this APN exactly)

Note: Because of the way the FSP is designed, the description in this app note is easily translated to other RA devices – in many instances with no changes to the procedure or code. Boards without virtual com port (VCP) support will need serial communication routed elsewhere or eliminated.

Required Resources

To build and run the associated example project, you will need the following:

- RA4E2 (https://www.renesas.com/us/en/products/microcontrollers-microprocessors/ra-cortex-m-mcus/ra4e2-entry-line-100mhz-arm-cortex-m33-general-purpose-microcontroller)
- USB to micro-USB-B cable (computer side depends on your system)
- e² studio IDE, version 2023-4
- RA Family Flexible Software Package (FSP), version 4.4.0
- A terminal emulator program (for example, putty) or the use of the terminal in e² studio (*not* the console) The FSP and e² studio are bundled in a downloadable platform installer available on the Renesas website at: renesas.com/ra/fsp

Prerequisites and Intended Audience

This application note assumes you have some experience with the Renesas e² studio ISDE and RA Family Flexible Software Package (FSP). Before you perform the procedures in this application note, follow the procedure in the *FSP User Manual* to build and run the Blinky project. Doing so enables you to become familiar with e² studio and the FSP and validates that the debug connection to your board functions properly.

This app note has an associated project based on the “Blinky” project but modified from that base project. You can start with the blinky project and build up a project just like it, using the included project as a guide if you get stuck.

The intended audience are users who have determined that virtual EEPROM is a valid solution to their storage needs *(see “Classifications of Data Using Non-Volatile Storage” and “Types of Non-Volatile Storage” below).*

Note: If you are using this application note as a reference for implementing on your own system and you understand the background, you can skim Section 1 and skip Section 2 entirely, as stated above.

Contents

1. Background on Non-Volatile Storage

Non-Volatile storage is data storage that is maintained across power cycles. In embedded designs, it is typically used for identification, calibration or configuration data. But non-volatile storage is not limited to that. In desktop computers, for example, anything stored on a drive is non-volatile.

1.1 Why Do We Need Non-Volatile Storage?

As previously stated, non-volatile storage is used to hold data that must be maintained across power-cycles. Three very common use cases are to store:

1. IDENTIFICATION DATA: Data used to indicate model number, serial number manufacturing info, MAC address, so forth.
2. CONFIGURATION DATA: Data used to control the current or available modes, functions, and so forth. (For example, what features/options are paid for, what language, or left/right handedness, or function button configurations/macros). Last calibration date would also fit into this category.
3. CALIBRATION DATA: Data that is used to calibrate, for example, an analog input or output of the device, or a system offset of a sensor in use.

1.2 Types of Data Using Non-Volatile Storage

There are three main types of writing data stored in non-volatile memory. Data can be:

- Written once
- Written infrequently
- Written frequently

1.3 Types of Non-Volatile Storage

Memory that is considered non-volatile comes in four different types:

- Flash with a low number of write-cycles (usually on the order of 10k write-cycles) such as code flash.
- Flash with a high number of write cycles (usually on the order of 100k write-cycles) such as Data Flash.
- EEPROM (usually on the order of 1M write cycles)
- Battery-backed-up SRAM (infinite number of write cycles until the battery runs out)

1.4 Mapping Data Classifications to Types of Non-Volatile Storage

If you look at the above two sections, not all storage types are appropriate for all data types. For example, data that must be changed and written often should not be stored in memory with a low number of write cycles. This is a case where VEEPROM is not an appropriate solution.

> Strategy: if the data changes during execution, but only the last values must be recalled across power cycles, detecting power loss, and having enough bulk storage to keep the system “alive” (really only the MCU) long enough to save the last data, might allow a VEEPROM design to work. The designer must determine the relative cost of bulk power storage and an external storage device, and the risk of power loss before the write is completed.

For this application note, we will be working with data that will be written once, and data that will be written infrequently. This type of data is perfectly suited for storage in VEEPROM.

1.5 How Virtual EEPROM (VEEPROM) Works

VEEPROM works by managing write cycles into Data Flash. It presents itself to user code as EEPROM – as if it were an EEPROM device connected via I2C. Behind the scenes, it manages the movement of data and re-instantiating data when an erase of a section of memory needs to be performed. This is all transparent to the user code and is managed by code in the FSP.

1.6 Hardware Considerations: Size and Wear-Leveling

Since each block in the data flash has limited erase/write cycles, it is important to equally distribute the erase and write cycles within the used data flash space in a process called “wear-leveling.” Since the data to be stored is typically smaller than the memory size, the memory is broken down into several evenly sized sections called “segments”. Flash memory cannot be erased by the byte – it is erased by the block. The last entry is tracked so the latest data is always retrieved.

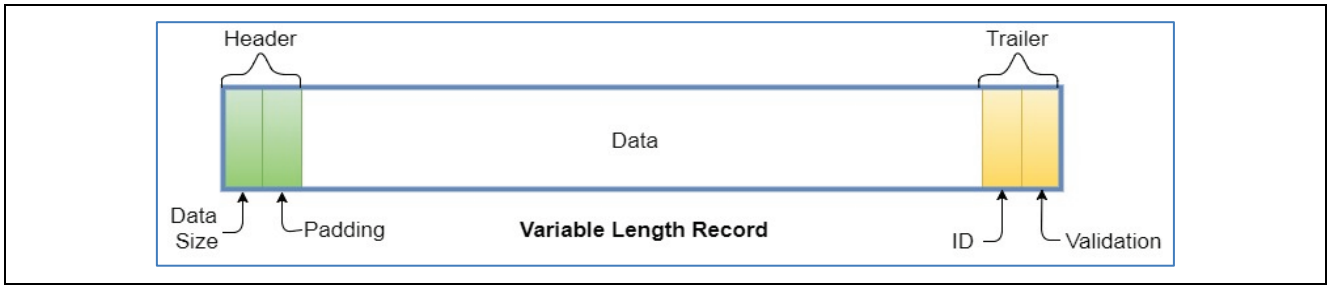
When a segment is “full” the latest data is copied into a new segment, and the old segment is erased to become usable again.

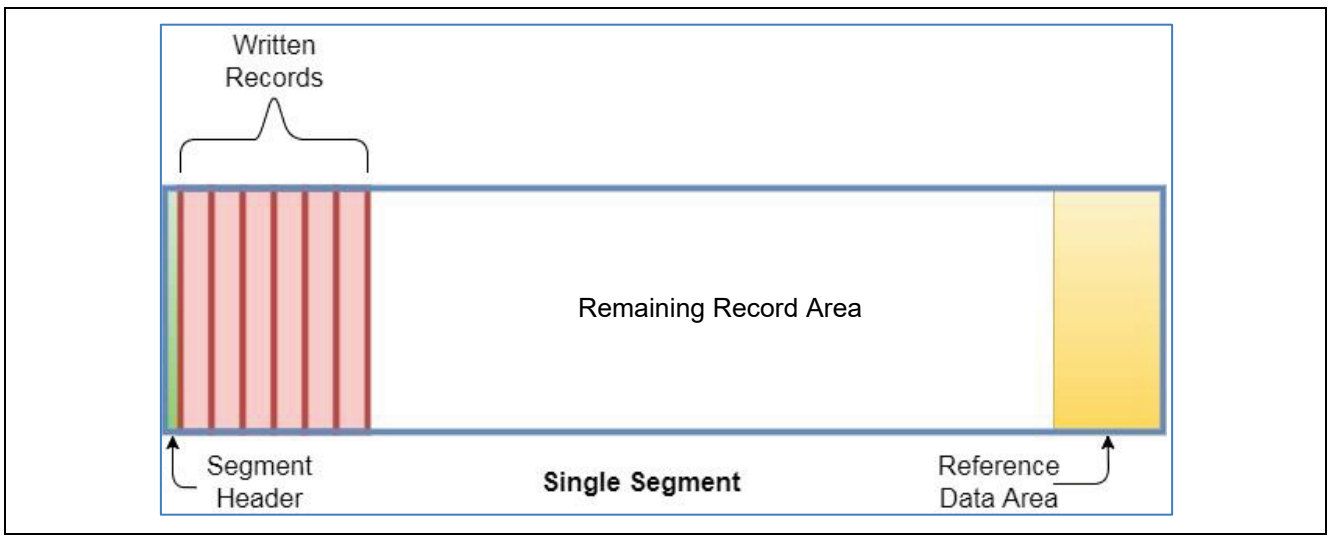
1.7 How the FSP Handles Your Data

FSP support is designed to open/close connections to memory, manage finding the last entry of a stored value, manage memory segments, and write new values to Data Flash. No data moves until explicitly done so under application control. We will give an overview of the API capabilities in the next subsection. The FSP allows for both fixed-length and variable length data. We will see how this is done in the example project and in Section 4 API Details.

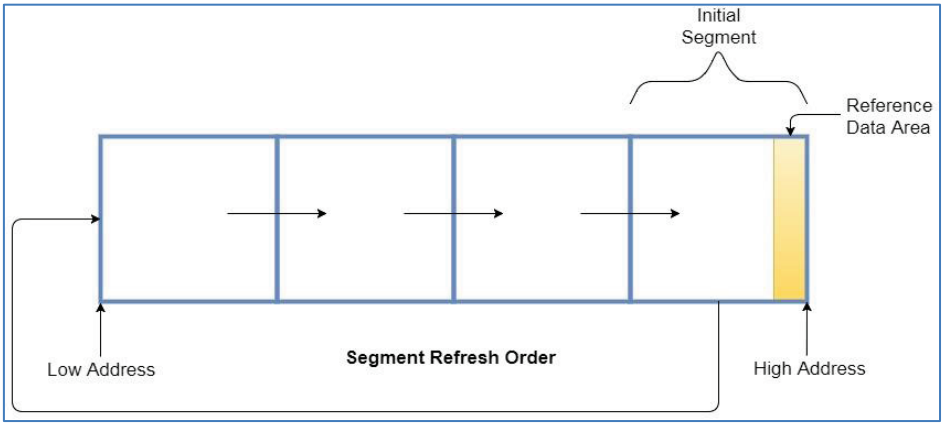
1.8 The Mechanics of the FSP Code for Managing Data Flash as VEEPROM

In section 4 we will look at each API individually, but the overall algorithm bears explaining at this point. The virtual EEPROM FSP stack manages the Data Flash by separating it into Segments. Each segment contains a segment header, x number of records, and (if used) optional reference data (typically used at manufacturing for device information: model, serial number, date of manufacture, MAC address, and so forth.).





Unlike real EEPROM, each memory location can only be written once, and blocks of memory must be erased as a block. To facilitate the storage of new data, and "wear-leveling" the memory usage, the new data record is written at the end of the current data. When a segment is full, the last good data is written into the next segment in a "round-robin" style, using all the allocated memory. Old segments are erased as a block to be used again.



The FSP APIs include read and write functions for both the data records and the reference data. These functions align pointers properly for read and write operations, which must be initiated explicitly from the user code (this means that setting a read location finds the last record but does not retrieve it.)

The functionality of the write functions detects a power failure during a write which leaves a corrupt piece of data. The FSP functions only mark the data as valid after the write is complete. So, any power loss (or interruption) will leave the last correctly written data as the last valid data to use.

Note the difference between this approach and if, for example, the designer chose to just write directly to data flash. In that scenario, let's say there are five pieces of data to be saved. They are written to a block. Now one or more pieces of data need to be changed. Since the data flash needs to be erased in blocks, there are two choices: (1) copy the data to SRAM, erase the block, update the data and re-write it, or (2) write the new data to a second block and then erase the first. This, on the surface, sounds like a reasonable strategy – until a power failure occurs in the midst of the process. Then, in (1) the data is simply lost, and in (2) there are now two versions of the data (power loss before the erase of the old block). In addition, we find that the same two locations are used over and over again in ping-pong style, and it is clear that those data flash locations will wear out while the others are never used. The VEEPROM library manages all of this for you by storing the data in records – each with an ID so the latest version is always identifiable. Since the records are written sequentially in data flash in a round-robin fashion, you get automatic wear-leveling.

Now, we will show an example using the APIs in the example project for better understanding.

2. Example Project

We will use a modified version of the standard "Blinky" project for this application note. This will better approximate and illustrate real-world use of the VEEPROM FSP functionality. The Blinky project will be modified in three ways.

First, the blinking code will be altered so the on and off timing is adjustable. Second, a serial communications module is added in to make it possible for the connected PC to control the Blinky application. Lastly, the standard project blinks all three LEDs (RGB). We will modify the project so it only blinks R&G "out of the box" and adding B is an option that can be enabled after manufacturing.

To help follow along and check your work, it is a good idea at this point to add the included example project to your workspace. Then you can compare your work with the working code. To do this, unzip the project into the workspace folder. Then, from e^2 studio, simply select "**File > Import…**" and then expand "**General**" and choose "**Existing Projects Into Workspace**" and follow the prompts.

The code we will add to your Blinky is already included in module folders inside the project folder, all grouped under one "**Modules**" folder. You can copy the Modules folder and put it into your Blinky project.

2.1 Deciding What Data Should be Saved

The easiest way to determine what data to save, how to save it, and in what configuration, is to get your project working with volatile data first. This makes sense because the entire intent of storing data non-volatilely is to be able to use it again in the next power cycle. This implies that the data would be read into live variables and used during execution. Getting your application code to behave exactly as you want it to before you commit to non-volatile storage is a path to the cleanest execution of VEEPROM implementation.

In this case, we would like to adjust the blink rate and enable the blue LED to flash. We will store values of control variables that affect these functions.

2.2 Starting with Blinky

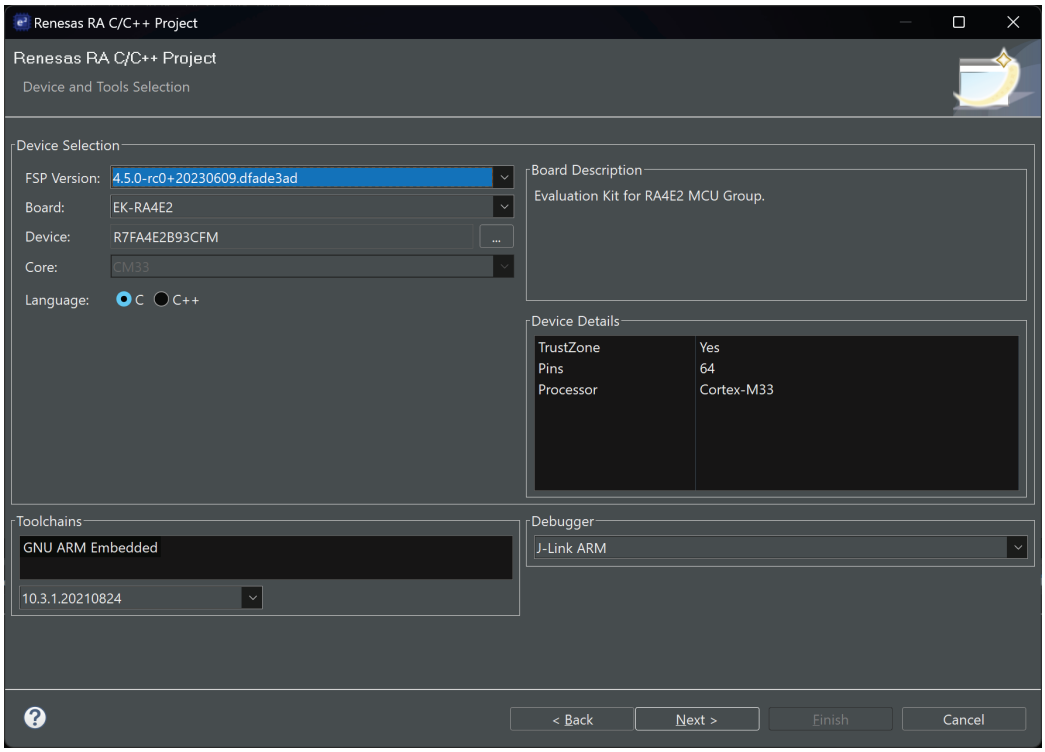
If you have not yet done so, start a new project in e^2 studio and create the “blinky” project:

[Select **File** > **New** > **Renesas C/C++ Project** > **Renesas RA**]

[Select **Renesas RA C/C++ Project**] hit <**Next**>

[Fill in a project name] (Use your own name here, just not the same as the example project) hit <**Next**>:

[Choose the **FSP (4.4.0)** and the **EK-RA4E2** board] hit <**Next**>:



[Select “**Flat**”] hit <**Next**> twice

[Select “**Bare Metal Blinky**”] hit <**Finish**>

2.3 Modifying the Blinky Project with Scaffolding to Add Volatile Data

The standard Blinky project simply creates a while loop with loop delays to blink the LEDs at a set rate. To show how we can use volatile (and then non-volatile) data to affect the blink rate, we will modify the default code to use the SysTick timer to keep time for us. We will do this by incorporating a simple scheduler. To change data "live" as if it were in a manufacturing process or in the field for calibration, we will also add UART support and send our parameters in over a serial connection from a terminal emulator.

2.3.1 Adding the Scheduler

If you have not yet done so, copy the "**Modules**" folder over from the example project into the same place in your Blinky project. All Arm® devices have a SysTick timer – designed for RTOS timing for task switching and other time-related activities. We will take advantage of that here. Also, copy over the "`Project.h`" file in the "src" folder from the example project to the Blinky project.

If you look into the two files in the "Scheduler" module folder ("`Scheduler.c`" and "`Scheduler.h`"), you can see a function called "`SystickHandler()`" which runs every time the SysTick Timer rolls under. There is already a weak reference to this function in the FSP, so we simply need to set the period for the SysTick Timer and it will fire periodically. We will do this in the "`hal_entry()` function located in the "`hal_entry.c`" file. Locate this in project explorer under the "`src`" folder, and open it.

Above the "`while(1)`" in this file, add the following line:

```
SysTick_Config(SystemCoreClock / 1000);
```

This line of code sets the SysTick Timer to 1 mS – a very standard value for SysTick.

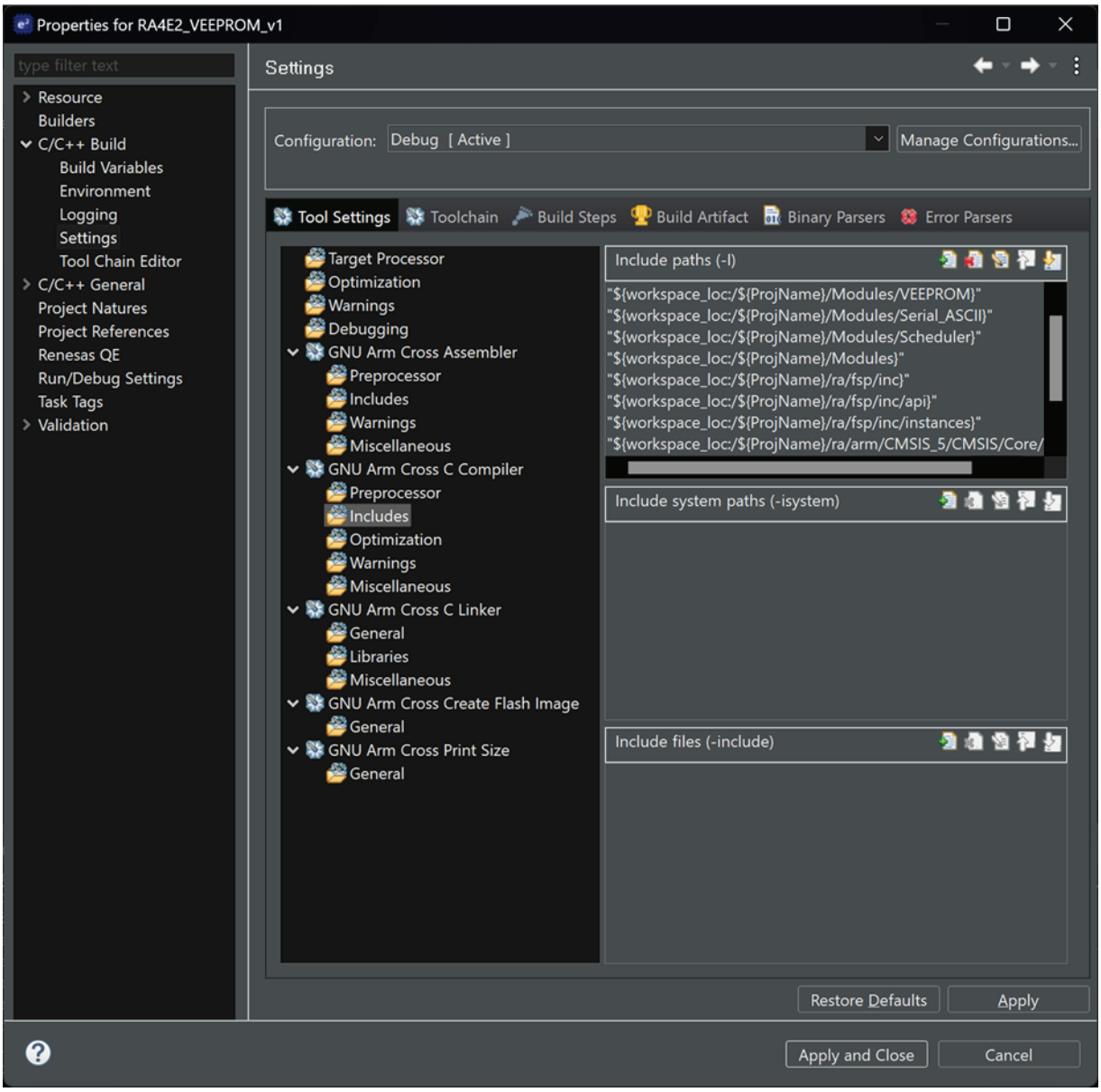
In addition, include the "Scheduler.h" file at the top of the "`hal_entry.c`" file:

```
#include "scheduler.h "
```

Looking at the code in the systick handler, you can see it is very simple: four counters that each set a flag when they roll under, and then get reset to count down again. These flags are shared with the main loop code, allowing code sections to run on regular intervals.

Even though the new "**Modules**" folder appear in the project explorer pane, the files contained in it are not part of the build process. To add the files to the project, you must add both a search path for `.h` files in the "Includes" section and Source Locations. This is done in the project properties. To do so, do the following:

1. Select the project in the workspace explorer panel and either right-click or choose **Project** from the menu.
2. In either case select "C/C++ Project Settings…"
3. In the dialog box tree on the **Tool Settings** tab, navigate to "GNU ARM Cross C Compiler" and then "Includes".



4. Add the two module folders by clicking on the “+” icon, clicking “Workspace…” and navigating to the folders.
5. Click **Apply and Close**.

2.3.2 Changing the Code to Use the Scheduler

The code to be placed in the main loop is commented out in the “`Scheduler.h`” file. Copy the commented code block and paste it in “`hal_entry.c`” under the “{“ of the “while(1)” loop in hal_entry() and uncomment it. Then comment out the existing Blinky loop code – we will copy and reuse it. (Be sure to leave the closing brace for the `while` loop.)

Copy the following commented out code and place it in the “`if (one_S_Flag) {`“ conditional branch:

```
/* Enable access to the PFS registers. If using r_ioport module then register protection is
automatically
* handled. This code uses BSP IO functions to show how it is used.
*/
R_BSP_PinAccessEnable();

/* Update all board LEDs */
for (uint32_t i = 0; i < leds.led_count; i++)
{
    /* Get pin to toggle */
    uint32_t pin = leds.p_leds[i];
```

```
    /* Write to this pin */
    R_BSP_PinWrite((bsp_io_port_pin_t) pin, pin_level);
}

/* Protect PFS registers */
R_BSP_PinAccessDisable();

/* Toggle level for next write */
if (BSP_IO_LEVEL_LOW == pin_level)
{
    pin_level = BSP_IO_LEVEL_HIGH;
}
else
{
    pin_level = BSP_IO_LEVEL_LOW;
}
```

Hal entry should then look like this:

When you compile and run the current project code, the board should blink at roughly the same rate as before (0.5 Hz – 1 sec on and 1 sec off). Now that we know our scheduler and blinking are working, we can modify the code to let us change the blink rate. We will do this with a minimum of code changes.

First, move the code you put in the 1 sec handler and move it to the 25 ms handler. Wrap the code in two conditional statements: one called `flashEnable` and one called `toggleMe` (the code as it exists writes the same value to the pins over and over because it was meant to only run the loop once for each blink – the 25 ms timer will run many times before the state of the LED changes to count down the on/off timers. While writing the same value out multiple times works, it is better to only write the value out for a change of state – this is the purpose of the `toggleMe` switch):

```
if (flashEnable == true) {

    if (toggleMe == true) {

// the flash code...

    }
}
```

Above the line "`if(toggleMe == true) {`" add the following code (note that it is a modification of some lines copied in the pin setting section):

```
if (BSP_IO_LEVEL_LOW == pin_level) // next state will be low, which means it is currently
high
{
    if (--onTime == 0) {
        onTime = onTimeSetting;
        toggleMe = true;
    }
}
else // next state will be high, which means it is currently low
{
    if (--offTime == 0) {
        offTime = offTimeSetting;
        toggleMe = true;
    }
}

// once the on or off timer is expired, it's time to change the state of the pin
```

This code checks the current state of the LEDs and counts down the appropriate timer for LED on or off, and when the timer expires it enables the pin toggle.

Declare the variables you just added at the top of `hal_entry.c`:

```
uint8_t flashEnable = true;
uint8_t toggleMe = false;
uint16_t onTime = DEFAULT_ON_TIME;
uint16_t offTime = DEFAULT_OFF_TIME;
```

```
uint16_t onTimeSetting = DEFAULT_ON_TIME;
uint16_t offTimeSetting = DEFAULT_OFF_TIME;
```

[The values of the macros used in these and future lines of code are already declared for you in `Project.h`.]

If you run the code now, the LED is still blinking at the familiar 0.5 Hz rate, and 50% duty cycle. Pause and change the value of `onTimeSetting` to 100 and resume execution. Notice that the on time is now more than twice as long as the off time.

We will add additional functionality to show how non-volatile memory can be used to configure a system and enable product features – even in the field. Inside the code that toggles the LED, the code looks like this:

```
/* Update all board LEDs */
for (uint32_t i = 0; i < leds.led_count; i++)
{
    /* Get pin to toggle */
    uint32_t pin = leds.p_leds[i];

    /* Write to this pin */
    R_BSP_PinWrite((bsp_io_port_pin_t) pin, pin_level);
}
```

Modify the code to only flash the blue LED if the “Blue Flashes” product feature is enabled:

```
/* Update all board LEDs */
for (uint32_t i = 0; i < leds.led_count; i++)
{
    /* Get pin to toggle */
    uint32_t pin = leds.p_leds[i];

    if ((enableBlueLED == true) || (i != BLUE_LED_INDEX)) {
        /* Write to this pin */
        R_BSP_PinWrite((bsp_io_port_pin_t) pin, pin_level);
    }
}
```

And add the declaration of the variable at the top of the file with the other variables:

```
uint8_t enableBlueLED = false;
```

Build and run and notice that the Blue LED does not flash. If you stop execution and change the value of `enableBlueLED` to 1 (true), it will flash with the rest of the LEDs.

The last functionality we will add is some information that will get written into the reference area. This will be the serial number in the format of “SNxxxxxx” and will be loaded through the serial port in the manufacturing process. For now, the only thing we need to add is the character array with the other variables:

```
char serialNo[16] = DEFAULT_SER_NO;
```

In the next section, we will modify the `onTimeSetting` and `offTimeSetting` values via the serial port and view the changes happening in real-time. We will also modify the `flashEnable` value to turn flashing on and off (this simulates a feature not stored in VEEPROM) and the `enableBLueLED` value which illustrates a field-upgradable feature that is controlled by a value in VEEPROM. Lastly, we’ll write in some manufacturing information into the `serialNo[]` character array.

2.3.3 Adding Serial Communications Support

[This section assumes the target has VCP support (such as the application note target: EK-RA4E2). If it does not, then the setting here must be modified for your particular hardware.]

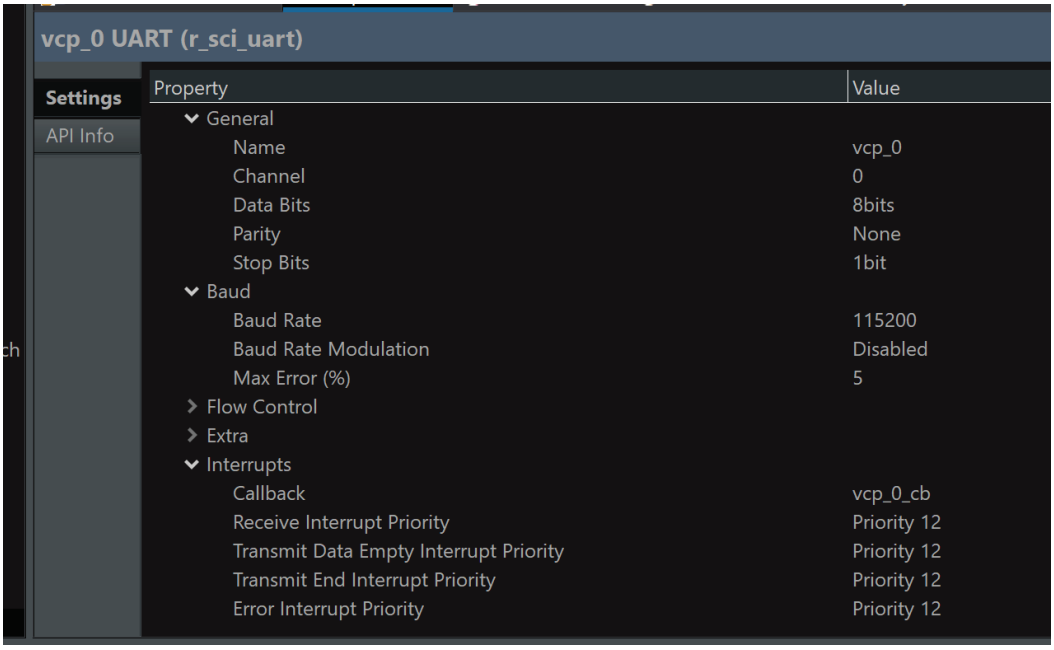
At this point, you should already have the **Modules** folder in your project and references to the folders in it for includes and source files in your project. (See above if you did not do this for the Serial module when doing it for the Scheduler module).

Adding serial communications requires the addition of an FSP stack to manage the UART. Open up the configuration file in the configuration perspective and add the UART (`r_sci_uart`) stack by going to: **New Stack** > **Search**…” and selecting “`r_sci_uart`”.

Set the following properties:

Module > General > Name = vcp_0
Module > General > Channel = 0
Module > Baud > Baud Rate = 115200
Module > Interrupts > Callback = vcp_0_cb
Module > Interrupts > Priority (all of them) = Priority 12

Check:
Pins > RXD0 = P410
Pins > TXD0 = P411



The following commands are implemented in the `serial_user.c` file (more details on the Ref page):

$f1\n | $f0\n
$n<value>\n
$s<value>\n
$b1\n | $b0\n
$i<value>\n
$w\n
$u\n

We will use these commands to test our VEEPROM implementation.

To include the serial support, we simply have to add a header to `hal_entry.c` and add two conditional calls to the main loop.

With the other #include statements at the top of the file, add:

```
#include "serial_user.h"
```

In the main loop of the `hal_entry()` function, enter the following code in the “every time through the loop” section:

```
        if (processPacket == true)
        {
            ProcessPacket ();
        }

        if (!RxBufferEmpty
        {
            ProcessReceiveBuffer ();
        }
```

The first conditional checks to see if a complete packet has been received, and if one has, it processes it. The second conditional checks to see if the receive buffer has anything in it, and if so, processes it one char at a time.

2.4 Modifying the Project to work with Non-Volatile Data

Once we have our application working the way we want it, the next step is to identify and prepare the data for storage. This consists of looking at the data for data types/sizes, frequency of writes, and, determining (1) if VEEPROM is still appropriate for our use and (2) how to configure it. We will start by adding a module called VEEPROM (like the serial and scheduler modules) in the modules folder. This module should contain two files: `veeprom.c` and `veeprom.h`. Add these files to the project as you did for the other two above.

2.4.1 Gathering Up the Data

This first step will determine both the appropriateness of VEEPROM for our application, and the size of the data being stored. In our case, we will not be modifying the blink speed, blink function, or calibration data very often, and the manufacturing information will only be set once. The speed, function, and calibration data will be stored in the “live records” part of the segment, and the manufacturing data in the reference data area.

2.4.2 Adding VEEPROM Support from the FSP

To enable VEEPROM, we must add the appropriate code support from the FSP. This is done in the same way we added the serial port above, by going into the configuration and adding the appropriate stacks and setting the properties.

In the configuration file in the configuration perspective, select: “**New Stack** > **Storage** > **Virtual EEPROM on Flash** (`rm_vee_flash`)” and the stack will be added to the project. For now, we can take the default values for all but two parameters. Change “Reference Data Support” to “Enabled” and “Reference Data Size” to “32.”

2.4.3 Changing the Code to Use VEEPROM

To start, add the following code in the 100 ms time slot in the while loop in `hal_entry()`:

```
if (writeParameters == true)
{
    WriteVEEPROM_Parameters();

}


if (writeReference == true)
{
    WriteVEEPROM_Reference();

}
```

And declare the following variables at the top of the file (with the other declarations):

```
char serialNo[SER_NO_LENGTH] = DEFAULT_SER_NO;
char modelName[MODEL_NAME_LENGTH] = DEFAULT_MODEL_NAME;
uint8_t refData[REF_DATA_LENGTH] = DEFAULT_REF_DATA;

uint8_t writeReference = false;
uint8_t writeParameters = false;
uint8_t callback_called = false;
rm_vee_state_t vee_done_state;
```

And lastly for the `hal_entry.c` file, add the following line at the top of the file with the other include files:

```
#Include”VEEPROM.h”
```

In the `VEEPROM.h` file, add the following code:

```
#ifndef _VEEPROM_H_
#define _VEEPROM_H_


void OpenAndRetrieveVEEPROM Values(void);
void WriteVEEPROM_Parameters(void);
void WriteVEEPROM_Reference(void);
```

```
#endif
```

In the `VEEPROM.c` file, add the following includes and variable declarations:

```
#include "hal_data.h"
#include "veeprom.h"
#include "project.h"

extern uint8_t callback_called;
extern uint16_t onTimeSetting;
extern uint16_t offTimeSetting;
extern uint8_t enableBlueLED;
uint16_t  *p_rec1_in_flash;
uint8_t *p_b_led_feat_in_flash;
uint8_t * p_refData;
uint32_t lengthInBytes;
extern uint8_t callback_called;
extern uint8_t writeReference;
extern uint8_t writeParameters;
extern uint8_t refData[REF_DATA_LENGTH];

uint8_t writeState;
fsp_err_t err;
```

and the following code:

```
void WriteVEEPROM_Parameters(void)
{
    callback_called = false;

    switch (writeState++)
    {
        case 0:
            break;
        case 1:
            break;
        case 2:
            break;
        case 3:
            break;
    }

}


void WriteVEEPROM_Reference(void)
{
    writeReference = false;

}
```

The code added so far creates a structure that is controlled by serial commands. If you are working on a board without serial support, you can still use the infrastructure and pause execution to change the variable values and then continue. Its purpose is to “gate” each record write on a 100 mS boundary giving the VEEPROM system time to create a record or update a segment if necessary. When all writes are done, the switch is turned off and the gating state machine is reset to the beginning. In case 3, you can see some code you might not expect (it is also at the end of the writeReference section below it). It is an inline delay and the stopMe variable with an increment operator. This is code that you would comment out for the final release (or separate with #defines switches, along with the stopMe declaration). It is used to ensure that when you hit a breakpoint, the writes to VEEPROM are completed and when you look at memory you will see your values without a bus conflict (there is technically no conflict – meaning no error is created, you will just read all zeros instead of the values if the bus is in use by the write).

We are almost ready to create the write functionality, but we must first create the ID info. Create (in `project.h`):

```
enum VEPROM_IDS {
    ON_TIME_ID = 1,
    OFF_TIME_ID,
    BLUE_LED_FEATURE_ID
};
```

We can now put in each of the three writes for the parameters into the switch statement you created in the `WriteVEEPROM_Parameters()` function in `VEEPROM.c`. For each case, enter the following line (modified for each of the three parameters):

```
err = RM_VEE_FLASH_RecordWrite(&g_vee0_ctrl, ON_TIME_ID, (uint8_t *)&onTimeSetting,
sizeof(onTimeSetting));
```

The modifications would include the correct ID and the correct variable name (in 2 places) for each of the other two function calls. So, one for OFF_TIME_ID and one for BLUE_LED_FEATURE_ID.

The switch statement should now look like this:

```
    switch (writeState++)
    {
        case 0:
            err = RM_VEE_FLASH_RecordWrite(&g_vee0_ctrl, ON_TIME_ID, (uint8_t
*)&onTimeSetting, sizeof(onTimeSetting));
            break;
        case 1:
            err = RM_VEE_FLASH_RecordWrite(&g_vee0_ctrl, OFF_TIME_ID, (uint8_t
*)&offTimeSetting, sizeof(offTimeSetting));
            break;
        case 2:
            err = RM_VEE_FLASH_RecordWrite(&g_vee0_ctrl, BLUE_LED_FEATURE_ID, (uint8_t
*)&enableBlueLED, sizeof(enableBlueLED));
            break;
        case 3:
            writeParameters = false;
            writeState = 0;
            break;
    }
```

Next we must include the `OPEN` API. This is done in the `OpenAndRetrieveVEEPROM_Values()` function in `VEEPROM.c`:

```
err = RM_VEE_FLASH_Open(&g_vee0_ctrl, &g_vee0_cfg);
```

Note: This just opens the VEEPROM and creates an internal table of pointers. It does not retrieve any data and does not “serve up” the pointers in the table. You must ask for each one of them individually by parameter. (This will be added next.)

This takes care of the writes. Now we must take care of the reads. This is important because typically this is done after the open, but only if the open is successful. That means that we must have default values if the open did not succeed, or if it created a new VEEPROM space (first power-up). We can do it like this (place after the open code you just put in the function):

```
    // Grab the onTime value stored in VEEPROM
    lengthInBytes = 2;
    callback_called = false;
    err = RM_VEE_FLASH_RecordPtrGet(&g_vee0_ctrl, ON_TIME_ID, (uint8_t **)&p_rec1_in_flash,
&lengthInBytes);
    if (err == FSP_SUCCESS)
    {
        onTimeSetting = *p_rec1_in_flash;
    }
    else if (err != FSP_ERR_NOT_FOUND)
    {
        while(1);
    }

    // Grab the offTime value stored in VEEPROM
    lengthInBytes = 2;
    callback_called = false;
    err = RM_VEE_FLASH_RecordPtrGet(&g_vee0_ctrl, OFF_TIME_ID, (uint8_t
**)&p_rec1_in_flash, &lengthInBytes);
    if (err == FSP_SUCCESS)
    {
        offTimeSetting = *p_rec1_in_flash;
    }
```

```
    else if (err != FSP_ERR_NOT_FOUND)
    {
        while(1);
    }

    // Grab the pointer to the Blue LED enabled feature status value stored in VEEPROM
    // ...the code will use this value while it is in flash
    lengthInBytes = 1;
    callback_called = false;
    err = RM_VEE_FLASH_RecordPtrGet(&g_vee0_ctrl, BLUE_LED_FEATURE_ID, (uint8_t
**)&p_b_led_feat_in_flash, &lengthInBytes);
    if (err == FSP_SUCCESS)
    {
        enableBlueLED = *p_b_led_feat_in_flash;
    }
    else if (err != FSP_ERR_NOT_FOUND)
    {
        while(1);
    }

    err = RM_VEE_FLASH_RefDataPtrGet(&g_vee0_ctrl, &p_refData);
    if (err == FSP_SUCCESS)
    {
        strcpy((char *)&refData, (char *)p_refData);

    }
    else if (err != FSP_ERR_NOT_FOUND) {
        while(1);
    }
```

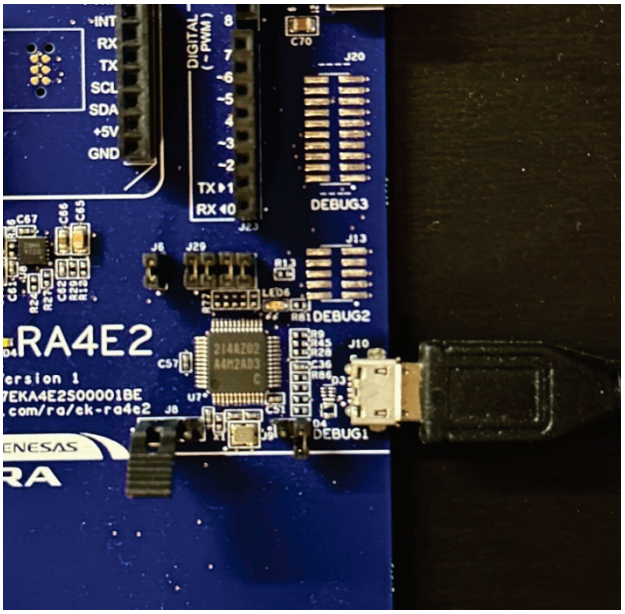
Note: There are various places where execution will be stopped in a while(1); loop if there are serious errors. By default, the first time you run the code it will fail all the read operations (because when you create the VEEPROM environment you have not yet written anything to read), but there is a separate conditional that accepts this and leaves the values at the default values from compile time.

3. Testing Our Code

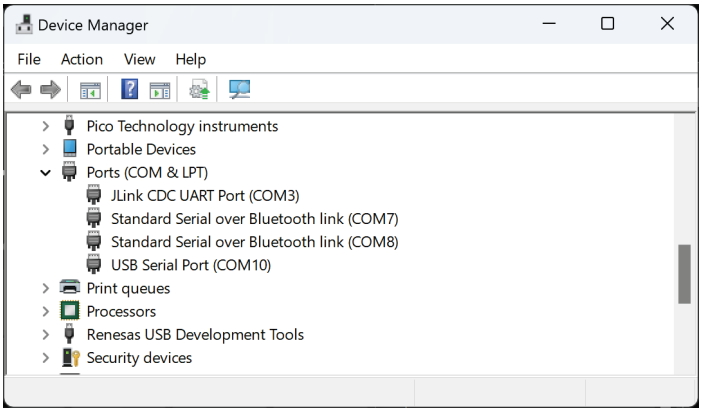
[If you have skipped to this section and you are unfamiliar with the Reness debugging process, please refer to the debugging sections on GitHub: https://renesas.github.io/fsp/_s_t_a_r_t__d_e_v.html#tutorial-your-first-ra-mcu-project-blinky]

3.1 Verifying Your Setup

The connection to the EK-RA4E2 board should look like this:



The other end of the USB cable is connected to your computer. If the connection is successful, you should see a JLink COM port show up in Device Manager:



This application note has been supplied with two projects zip files. One is the base code and one is the final project, which you used for comparison as you built your project on the base Blinky code-generated project. The base project has all of the modification code added, and is ready for the VEEPROM support to be added.

To verify the operation of the download, execution, and connection with your terminal emulator, use the final project. Connect the application board and set the project to download. The program counter indicator (arrow) should be on `main()`. At this point, you know your debug environment is working.

Set your terminal emulator communication properties to 115200, N, 8, 1 and select the “JLink CDC UART Port”.

Serial commands to the board must be terminated with a ‘\n’ (<LF> - 0x0A) character. This can be done in two ways: if the terminal emulator works in **character mode** (each character is sent as it is typed) and will transmit an **Enter** key as <CR><LF> (usually this is selectable as some combination or subset of <CR> and <LF>) or if the terminal emulator is on **line mode** (a group of typed characters are sent with an **Enter** key or **Send Test** button) it should append a trailer (that includes <LF>) at the end. Note that <CR> is ignored, and only <LF> is processed.

If your terminal is running when the debug starts, you will see a welcome message with the software rev. To verify two-way communications, hit “run” on e^2 studio and the LEDs should be flashing. Type this command into the terminal and send it to the board:

$F0

The flashing should stop. By entering this command and sending it to the board:

$F1

The flashing should start again. At this point you know your debugging connection is working and your serial connection is working. You can find the rest of the commands in the Reference section that follows.

It is time to start testing our code. Select your own project in e^2 studio and download it to the board.

3.2 Testing Code

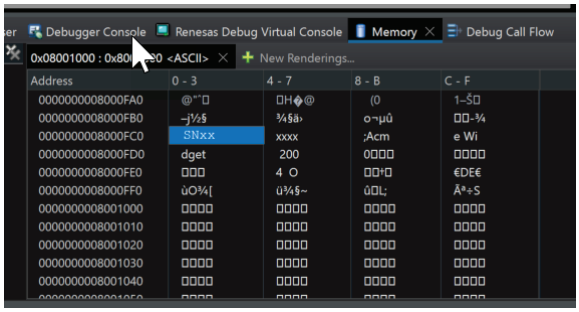
Testing the code is straightforward because of the way we have structured the application. Using the commands ($b, $f, $n, $s) you can modify the behavior of the application flashing. You can then issue the $w and $u to store the values of `onTime/offTimef/blueEnable` and to store the modified model number in VEEPROM (issuing the $b command not only enables/disables the blue LED flashing, but also alters the model number from “Widget 2000” to “Widget 2000B” to indicate the feature has been added).

[You can see the complete list of commands and options in the Reference Sheet: Virtual EEPROM section.]

You can watch each of the values change in the watch window by adding the appropriate watch expressions. (enable real-time updating so you can see them change without stopping execution).

By placing a breakpoint in the VEEPROM callback function, you can be free to examine memory after you write to see the values as stored. Simply go to the **Memory** tab and view from 0x08001000 which is the end

of the last segment in our 4 segments. If you render in ASCII and scroll back slightly, you can see the reference info (which is stored in ASCII) as “SNxxxxxx;Acme Widget 2000” just before that address:



4. API Details

Each of the API calls will be detailed in this section for your reference. To better understand their functionality, we will start with a deeper dive into the functionality and requirements of VEEPROM.

4.1 VEEPROM Functionality

It was mentioned earlier that the VEEPROM functions store the data in records contained within segments, which are in turn stored sequentially in a round-robin fashion. But what does that mean?

First, there are some small requirements to make this all work, and to have efficient code. Each piece of data (not data type, but each variable to be stored) must have its own unique ID. The maximum number of IDs is settable in the stack properties, which defaults to 16 and is a reasonable number for most applications. This is mostly used to construct the lookup or reference table, which stores pointers to the latest version of that data’s value. It is a 0-based indexing system. So, for example, if you are storing limits of `MAX_TEMPERATURE`, `MAX_PRESSURE`, `MAX_FLOW`, they would be assigned index, 0, 1, 2 respectively.

The database is made up of records, and each record contains a header, made up of the data size and a padding, the data, and a trailer which is made up of the ID and a validity flag (set by convention to `0xBEAD`).

New records (updated data) are written sequentially throughout the memory page. When a segment cannot hold additional records, the latest data is copied and written to the “top” of the next segment in rotation, then the old segment is invalidated.

On “powerup” (opening of VEEPROM), this set of records is crawled through until the last entry for each of the IDs is found, and these ultimate addresses are stored in the above-mentioned indexing table for later use.

The stack defaults to 2 segments, and this will work, but the erase times when switching between segments may be too long for the application. The number of segments can be increased, but the memory size *must be* evenly divisible by segments. For example, a memory size of B4k cannot have 3 or 6 segments, but 2, 4, or 8 segments work fine. The smaller the segments, the faster the erase time. Note that segments cannot be so small that the storable data does not fit in them. You typically want the ability to store multiple records of data in a single segment to avoid too many erase cycles. We will use 4 segments.

Each API call returns an error code.

4.2 API Open

```
err = RM_VEE_FLASH_Open(&g_vee0_ctrl, &g_vee0_cfg);
```

This is the start of it all. Typically at power-up, this API is called and depending on the state of data flash (VEEPROM) it will do one of three things:

1. Open the data flash and walk through the records, saving the location of the last entry of each data type in the index table.
2. If VEEPROM has not been initialized, it will do that now.
3. If corruption is found, it will copy the last valid data to a new segment to start over.

In any case, the index table will point to the last valid data (all nulls if it is first initialization).

Error codes of `FSP_ERR_PE_FAILED` (this is a failure to program or erase - likely data flash is write protected) or `FSP_ERR_NOT_INITIALIZED` (likely the segment size is not evenly divisible into the allocated memory).

4.3 API Write Record

```
err = RM_VEE_FLASH_RecordWrite(
          &g_vee0_ctrl, <ID>,
          (uint8_t *)&source,
          sizeof(source_t));
```

where,

source = variable to be written from.

Note: We are using the address of the source, which is a pointer, and we are casting it to a byte pointer (a uint8_t).

source_t = datatype for source *[resolved to a size by sizeof()]*

Note: All writes occur in the background. This means that if you are not careful about what happens right after you trigger a write, you will get “bus errors” (they are not “real errors” but if you try to read you will get all zeros back). If you think your code will modify the variable value being written, or want to examine data flash after the write, insert a wait loop for the write to finish before moving on. (See “2.4.4 Testing Our Code” above.)

4.4 API Get Record Pointer

```
err = RM_VEE_FLASH_RecordPtrGet(&g_vee0_ctrl,
          Record_ID,
          &p_flash_bytes,
          &byte_length);
```

where,

Record_ID = the ID for that data type

destination = address of pointer where you would like the pointer to go (&p_flash_bytes above)

byte_length = pointer where you’d like the number of bytes read to go &p_byte_length above – *(not really needed if you know what you are reading, but a good check).*

Note: This API does not return the data, but a pointer to the data. You can read it whenever you like.

[If the data is re-written, you must call this function again to update your pointer before reading.]

4.5 API Get Reference Data Pointer

```
err = RM_VEE_FLASH_RefDataPtrGet(&g_vee0_ctrl, &p_refdata_bytes);
```

where,

p_refdata_bytes = the pointer to the reference data bytes

Note: There is no size associated with the reference data because the reference data is a fixed number of bytes. As in the parallel record API, this API does not return the data, but a pointer to the data.

4.6 API Write Reference Data

```
err = RM_VEE_FLASH_RefDataWrite(&g_vee0_ctrl, (uint8_t *)&ref_type2);
```

where,

&ref_type2 = pointer to the data to be written (notice it is cast to a byte pointer – all reference data is an array of bytes (typically a string but could be anything).

This API function should be rarely used. The reference data is meant to be written once in manufacturing, and perhaps updated once in the field (but, again, this is rare). There is no limit to the number of times it is written… but… after the second write any additional writes cause a segment refresh (copy/erase) so it is slow. If there is something in this area that is written more frequently, consider making it a normal data type and writing it with the other variables.

4.7 API Get Status

```
RM_VEE_FLASH_StatusGet()
```

This returns a structure with a lot of information. Please see the FSP reference manual for contents.

4.8 API Close

```
err = RM_VEE_FLASH_Close(&g_vee0_ctrl);
```

This is another API call that is infrequently used. Normally the VEEPROM remains open whenever the code is running. But, there may be times – say, before invoking a bootloader, that you may want to ensure the VEEPROM is closed properly.

5. Modifying Standard Implementation

The standard implementation of VEEPROM assumes all Data Flash will be utilized for VEEPROM storage. But there are times when we might want to use a small amount of the available Data Flash for other reasons. There is a way to modify the parameters of the FSP VEEPROM stack to allow for this configuration.

There are two things to consider when doing this: one is that the total memory allocated must be a multiple of the segment size, or the initialization will fail. The second is that you must be very careful how you access the portion you have allocated for other functions which may attempt to access the data flash area– if the flash bus is busy with Virtual EEPROM, activity, it will fail. And if your other code is accessing it, the VEEPROM code will fail. You can use the above mentioned `RM_VEE_FLASH_StatusGet()` function to see if EE is busy.

6. Equations and Resources for Calculation

During the design process, we will need to ensure that we are selecting the correct parameters for the VEEPROM stack, some of which are calculated. In addition, in many applications, either the read time, write time, or both may be critical to the proper operation of our system. (The initial read is usually done at the start of the application, increasing the time from power-on to functionality. During a write, critically-timed operations may be affected.)

The included spreadsheet can help you with those calculations, but we will derive them here first, so you have full understanding.

To ensure that the correct number and size of segments is selected, the easiest way is to find the start and end addresses of data flash. If you are using the entire data flash area for VEEPROM, then you can find this information in the hardware user's manual. If you are reserving some data flash for other purposes, then you must subtract that from the whole space. Once you have the start and end addresses, you need to select the number of segments. The minimum number of segments is 2, but the suggested number is 4 or 8 (this makes erases faster, but not too frequent).

Divide the memory size by the number of segments. If there is no decimal, you are likely ok, but to ensure that all is correct, round down to the nearest 1 (that is, no decimals) and multiply by the number of segments. This should equal the size of data flash reserved for VEEPROM.

7. Reference Sheet: Virtual EEPROM

APIs

API Open

```
err = RM_VEE_FLASH_Open(&g_vee0_ctrl, &g_vee0_cfg);
```

Typically at power-up, this API is called and depending on the state of data flash (VEEPROM) it will do one of three things:

1. Open the data flash and walk through the records, saving the location of the last entry of each data type in the index table.
2. If VEEPROM has not been initialized, it will do that now.
3. If corruption is found, it will copy the last valid data to a new segment to start over.

In any case, the index table will point to the last valid data (all nulls if it is first initialization).

Error codes of `FSP_ERR_PE_FAILED` (this is a failure to program or erase - likely data flash is write protected) or `FSP_ERR_NOT_INITIALIZED` (likely the segment size is not evenly divisible into the allocated memory).

API Write Record

```
err = RM_VEE_FLASH_RecordWrite(
        &g_vee0_ctrl, <ID>,
        (uint8_t *)&source,
        sizeof(source_t));
```

where,

source = variable to be written from. Note that we are using the address of the source, which is a pointer, and we are casting it to a byte pointer (a uint8_t).

source_t = datatype for source *[resolved to a size by sizeof()]*

Note: All writes occur in the background. This means that if you are not careful about what happens right after you trigger a write, you will get “bus errors” (they are not “real errors” but if you try to read you will get all zeros back). If you think your code will modify the variable value being written, or want to examine data flash after the write, insert a wait loop for the write to finish before moving on. (See section, 2.4.4 Testing Our Code.)

API Get Record Pointer

```
err = RM_VEE_FLASH_RecordPtrGet(&g_vee0_ctrl,
        Record_ID,
        &p_flash_bytes,
        &byte_length);
```

Where,

Record_ID = the ID for that data type

destination = address of pointer where you would like the pointer to go (&p_flash_bytes above)

byte_length = pointer where you’d like the number of bytes read to go &p_byte_length above – *(not really needed if you know what you are reading, but a good check).*

Note: This API does not return the data, but a pointer to the data. You can read it whenever you like.

[If the data is re-written, you must call this function again to update your pointer before reading.]

API Get Reference Data Pointer

```
err = RM_VEE_FLASH_RefDataPtrGet(&g_vee0_ctrl, &p_refdata_bytes);
```

Where,

p_refdata_bytes = the pointer to the reference data bytes

Note: There is no size associated with the reference data because the reference data is a fixed number of bytes. As in the parallel record API, this API does not return the data, but a pointer to the data.

API Write Reference Data

```
err = RM_VEE_FLASH_RefDataWrite(&g_vee0_ctrl, (uint8_t *)&ref_type2);
```

Where,

&ref_type2 = pointer to the data to be written (notice it is cast to a byte pointer – all reference data is an array of bytes (typically a string but could be anything).

This API function should be rarely used. The reference data is meant to be written once in manufacturing, and perhaps updated once in the field (but, again, this is rare). There is no limit to the number of times it is written… but… after the second write any additional writes cause a segment refresh (copy/erase) so it is

slow. If there is something in this area that is written more frequently, consider making it a normal data type and writing it with the other variables.

API Get Status

```
RM_VEE_FLASH_StatusGet()
```

This returns a structure with a lot of information. Please see the FSP reference manual for contents.

API Close

```
err = RM_VEE_FLASH_Close(&g_vee0_ctrl);
```

This is another API call that is infrequently used. Normally, the VEEPROM remains open whenever the code is running. But, there may be times – say, before invoking a bootloader, that you may want to ensure the VEEPROM is closed properly.

Parameter Settings

Number of Segments must be an even division of the allocated data flash area (allocated data flash are must be equal to n * segment size).

Error Codes

FSP_SUCCESS – This means everything executed properly
FSP_ERR_NOT_FOUND – This will occur if you search for a pointer for an ID that has no data stored. You will see this the first time your code runs for every reed, because the first call to *open* created the VEEPROM area, and there is no data stored in it yet.
FSP_ERR_PE_FAILED – This typically means the data flash area is write protected.
FSP_ERR_NOT_INITIALIZED – This error occurs if there is an attempt to access VEEPROM and it is not initialized (no *open* was ever performed).

Serial Commands

Serial command format is: **$<cmd>[<parameter>]\n**
(Commands can be upper or lower case)
(baud rate is 115200)

$f1\n | $f0\n	(Turn flashing on/off - volatile)
$n<value>\n	(Adjust value of on time in 25mS steps – range of <value> is 5-150 – nonvolatile)
$s<value>\n	(Adjust value of off time in 25mS steps – range of <value> is 5-150 – nonvolatile)
$b1\n | $b0\n	(Enable/disable blue LED flashing – nonvolatile)
$i<value>\n	(Change serial number – <value> is a zero-terminated string – nonvolatile)
$w\n	(Write the parameters to VEEPROM)
$u\n	(Write the ref data to VEEPROM)

8. Website and Support

Visit the following URLs to learn about key elements of the RA family, download components and related documentation, and get support:

RA Product Information	renesas.com/ra
RA Product Support Forum	renesas.com/ra/forum
RA Flexible Software Package	renesas.com/FSP
Renesas Support	renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Oct.30.23	—	Initial release

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

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

1. Precaution against Electrostatic Discharge (ESD)

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

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

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

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

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

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

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

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

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

(Rev.5.0-1 October 2020)

Corporate Headquarters

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

Trademarks

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

Contact information

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

© 2023 Renesas Electronics Corporation. All rights reserved.