RENESAS

OFFICIAL RENESAS RA FAMILY BEGINNER'S GUIDE

Richard Oed

2023.01

RENESAS RA

Richard Oed

OFFICIAL RENESAS RA FAMILY BEGINNER'S GUIDE

The Renesas RA Family Beginner's Guide is intended to be used as both an introductory guide for engineers taking their first steps with the RA Family, as well as a bookmarked reference book for those already familiar with the RA Family. Please be informed that this version of the book refers to the FSP Version 3.7.0.

As the RA Family of MCUs is being continually developed it's inevitable that by the time we publish this PDF, there will be sections which are no longer up to date. To ensure the book remains useful for the above purposes, there is a supporting online ecosystem which you can find at **www.renesas.com/ra-book**.

CONTENTS

FOREWORD

In our modern world of networked devices, security is an omnipresent issue. Every networked application is a point of attack, a potential vulnerability where potential hackers could gain access to the network and thus access to data worth protecting.

In the rapidly growing area of the IoT, the aspect of embedded security takes on an additional significance, as cost-effective, scalable and energy-efficient security solutions are required here, which are also easy to configure. Therefore, nowadays no manufacturer of embedded applications can avoid taking the topic of security seriously, even if neither available development resources nor the corresponding development time have been budgeted for this.

For this reason, the number of engineers who are forced to deal with the topic of security for the first time under time pressure is constantly growing. The reasons for this are manifold, such as new legislation, competitive pressure, protection of intellectual property, etc. For many of these companies, the first contact with the topic of security can be very challenging, as security is a multifaceted and complex topic that encompasses hardware and software components and must be considered holistically. This means that security cannot be added only at the end of the development process but must be part of the system approach from the very beginning. Often, companies lack the necessary competence, and in addition, a security implementation causes additional costs for their product. In fact, some companies are suffering from this real pressure. In addition, we have carefully evaluated the other challenges our customers may face with the market and technology trends.

Renesas' development departments have made it their business to help our customers. The focus is on developing a cost-effective, scalable and energy-efficient security solution that supports the security requirements of tomorrow's embedded systems.

The outcome of this passionate work to aim for the best for our customers is the Renesas RA Family. The RA Family offers industry leading IoT security with software flexibility based on Arm Cortex-M core with complete ecosystem from Renesas and our partners. This enables our customers to achieve their application goals including security objectives in a much quicker way.

Our passion is the intense attention to detail that simplifies our customers' development work. The 32-bit RA MCU Family is an evidence of such passion.

We hope that this book will help you to enjoy your next project using the RA Family.

Bernd Westhoff
Director – RA MCU Marketing
Renesas Electronics

1 INTRODUCTION TO THE RENESAS RA FAMILY

Introduced in October 2019, the RA Family of microcontrollers (MCUs) extended the 32-bit MCU range of Renesas. It compliments both the existing Renesas Synergy Platform and the RX families. The Renesas Synergy™ Platform, which is also based on Arm® Cortex®-M cores combines MCUs with commercial-grade, warranted software and development tools. The Renesas eXtreme (RX) Family features the proprietary RX core offering an industry-leading 32-bit CoreMark® per MHz performance and a large on-chip Flash memory and SRAM. All of them provide unique differentiation and value to customers. The new RA Family consists of the RA2 Series for low power applications, the RA4 Series for devices needing low power, but high performance and security, the RA6 Series offering advanced performance with connectivity and security, and the RA8 Series offering the highest performance for applications employing human-machine interfaces, connectivity, security, and analog interfaces.

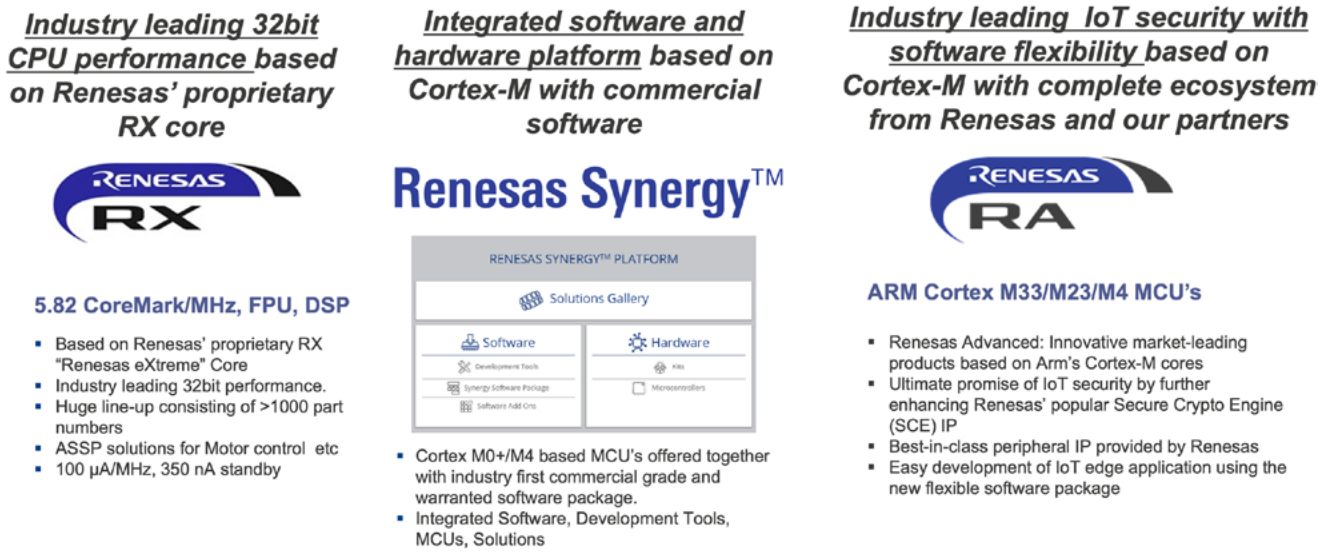


Figure 1-1: The RA Family complements Renesas' offer of 32-bit microcontrollers

The currently released devices of the RA Family incorporate hardware-based security features ranging from AES (Advanced Encryption Standard) acceleration and a True Random Number Generator (TRNG) to fully integrated crypto subsystems isolated within the microcontroller. Renesas' popular Secure Crypto Engine (SCE) was further enhanced for the RA Family. It provides together with the NIST CAPV (Cryptographic Algorithm Validation Program) certification symmetric and asymmetric encryption and decryption, hash functions, and advanced key handling, including key generation and MCU-unique key wrapping. An access management circuit shuts down the crypto engine if the correct protocol is not followed, and dedicated RAM ensures that plaintext keys are never exposed to any CPU or peripheral bus. Multiple groups also incorporate Arm's v8-M TrustZone®.

The whole family is Level 1 PSA Certified® and provides tamper detection and reinforces resistance to side-channel attacks. These features enable developers to improve the security and safety in their high-performing Internet of Things (IoT) endpoint and Edge devices for industrial and building automation, metering, healthcare, and home appliance applications.

Transition within the RA Family is easy, thanks to its feature and pin compatibility, and the commonality of its peripheral IP-(Intellectual Property-)blocks across the different series. All key parts, silicon, tools, and software, have been optimized to work together, creating new blocks where necessary, but also reusing proven IP-blocks, where they are already cutting edge.

For developers this ensures that they can start their hardware and software development without encountering incompatibilities between the different bits and pieces of their workflow.

Engineers are supported by the RA Family's tools ecosystem, which offers Integrated Development Environments (IDEs) like the Eclipse-based e^2 studio, compiler, on-chip debugging, evaluation kits, design files, schematics, PCB (Printed Circuit Board) layouts, and BOMs (Bill of Material). The easy-to-use Flexible Software Package (FSP) developed by Renesas provides an open architecture that allows designers to re-use their legacy code and combines it with ready-to-use software examples from Renesas. Renesas also built a comprehensive RA Partner Ecosystem to deliver additional software and hardware building blocks that will work right out of the box.

All this relieves developers from the burden of taking care about basic tasks and helps them to concentrate on what they really want to do: To create value-added applications.

1.1 Challenges in Todays Embedded System Design

From the Apollo Guidance Computer, developed by the Massachusetts Institute of Technology (M.I.T.) in the early 1960's, to today's high-end automotive applications or fully connected devices used for the Internet of Things (IoT), embedded systems have changed significantly over the last decades. Prior to the turn of the century they employed only a few very basic interfaces such as push-buttons for input or character LEDs or even Nixie tubes for output, and their software consisted of a single function, mostly implemented as a simple loop inside main(), with interrupts to handle a limited amount of tasks. A microcontroller (MCU) with a few MIPS (million instructions per second), a couple of kB of memory and a basic serial communication interface has been sufficient for this kind of application.

Today's embedded systems however are highly interconnected, demanding a wide variety of interfaces like Ethernet, wireless, or graphical displays, all of which need to be configured and handled, and which exchange data and messages not only with each other but also with the outside world to form the complete application. This can require an MCU with clock speeds of 60 MHz or more, several MB of Flash memory and perhaps a couple of tens of thousands kB of on-chip SRAM.

A Real-Time Operating System (RTOS) is not only helpful, but sometimes essential, as different threads need to be prioritized and executed concurrently. Development of such systems is no longer possible in the same way that legacy systems were designed, as increased connectivity, feature-rich human-machine interfaces and security demands make these systems by far less hardware- but much more software-centric.

In addition, development cycles get shorter and requests for new features come in at higher rates. All of this not only places an enormous burden on the engineers who have to tackle new challenges more frequently, but are also a huge investment, not all of which may be visible right from the beginning. This all means that software designers need help to develop their applications, allowing them to handle data and interfaces efficiently without having to write all the code from scratch. They want to concentrate on adding value to the application, not to write low-level drivers or security routines themselves.

Here the Flexible Software Package (FSP) for the RA Family enters the stage. It provides a Board Support Package (BSP) with efficient HAL-drivers and an easy-to-use middleware. Included in the FSP are Amazon's FreeRTOS™ and Microsoft's Azure® RTOS Real-Time Operating Systems, but due to the FSP's CMSIS RTOS compliance, engineers can employ any RTOS of their choice. Re-use of customer legacy code is also possible. With this open software ecosystem, designers can create easily the functions needed for connected IoT endpoint-systems or Edge applications based on Artificial Intelligence.

1.2 The RA Family of Microcontrollers

Initially, the Renesas RA Family of microcontrollers consists of four Series – the already released RA2, RA4, and RA6 Series, and the planned RA8 Series – for use in end-products ranging from small, battery-operated sensor applications, to high-performance, processing-intensive embedded systems. Built-in peripherals for analog, connectivity, human-machine-interfaces, security, motor and inverter control, and more, make this family well suited for the rapidly expanding Internet of Things (IoT) and Edge computing market, but by no means limited to that.

All RA MCUs are based on 32-bit Arm® Cortex®-M cores: The RA2 Series on the M23 core, while the RA4 and RA6 Series devices are based either on the M4F core or on the M33F core with Arm's v8-M TrustZone®. All of them include standard peripherals from Arm like the Nested Vector Interrupt Controller (NVIC), the Arm Memory Protection Unit (MPU), or the Serial Wire Debug (SWD) and Embedded Trace Buffer (ETB)

for easy development. Moreover, Renesas added its own Intellectual Property (IP) blocks where Arm has no solution and where additional performance or features were needed. These extra IP blocks are based on proven technology from Renesas, adapted to the demands of the RA Family.

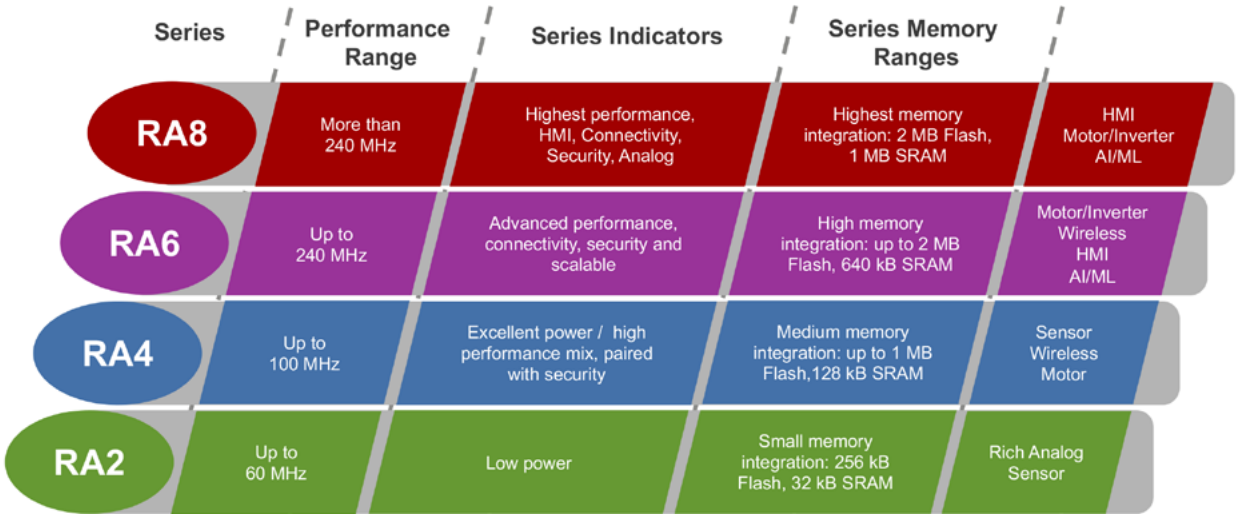


Figure 1-2: The initial series of the RA Family of microcontrollers

The three series of RA microcontrollers released at the time of writing are divided in several groups and exhibit the features below.

- **RA2 Series – Low Power:** Based on the Arm Cortex-M23 core with frequencies of up to 60 MHz, a memory range of up to 256 kB Flash memory and 32 kB SRAM. The supply voltage can range from 1.6 V to 5.5 V. Peripherals include USB Full-Speed, CAN, 24-bit Sigma-Delta analog-to-digital converter (ADC), 16-bit digital-to-analog converter (DAC), capacitive touch sensing, and security and safety.
- **RA4 Series – High Performance & Excellent Power:** Built on top of either the Arm Cortex-M33F with TrustZone or the Arm Cortex-M4F cores with frequencies of up to 100 MHz. Up to 1 MB Flash memory and 128 kB SRAM. Voltage range from 1.6 V to 5.5 V. Peripherals include capacitive touch sensing, segment LCD controller, USB Full-Speed, CAN, security and safety features, as well as data converters and timers. The RA4W1 Group devices come additionally with Bluetooth® low energy (BLE) 5.0.
- **RA6 Series – Advanced Performance:** Based on either the Arm Cortex-M33F with TrustZone or the Arm Cortex-M4F core. Frequencies of up to 240 MHz. Up to 2 MB Flash memory and 640 kB SRAM. Voltage range from 2.7 V to 3.6 V. Peripherals include data converters, timers, an external memory bus, Ethernet, USB Full- and High-Speed, CAN, security and safety features, capacitive touch sensing and a graphic LCD controller for TFT, as well as a 2D-graphics engine. The RA6T1 and RA6T2 Group devices come with enhanced peripherals for motor control, like a high-resolution PWM timer or advanced analog blocks.

Each series will be expanded gradually as new devices are introduced.

All of the MCUs in each series – and to some extend within the whole family – are feature- and mostly pin-compatible. The peripherals on the smaller devices are mainly subsets of the ones found on the larger devices. This allows for scalability and code reuse from one device to another. Similar packages across the different series have pinouts which are almost identical to each other. This way, developers have not to choose the final device at the very beginning because changing to a different one later on is possible. Also, creating PCB-layouts with multiple package footprints inside each other for flexible manufacturing options of the end-product is feasible.

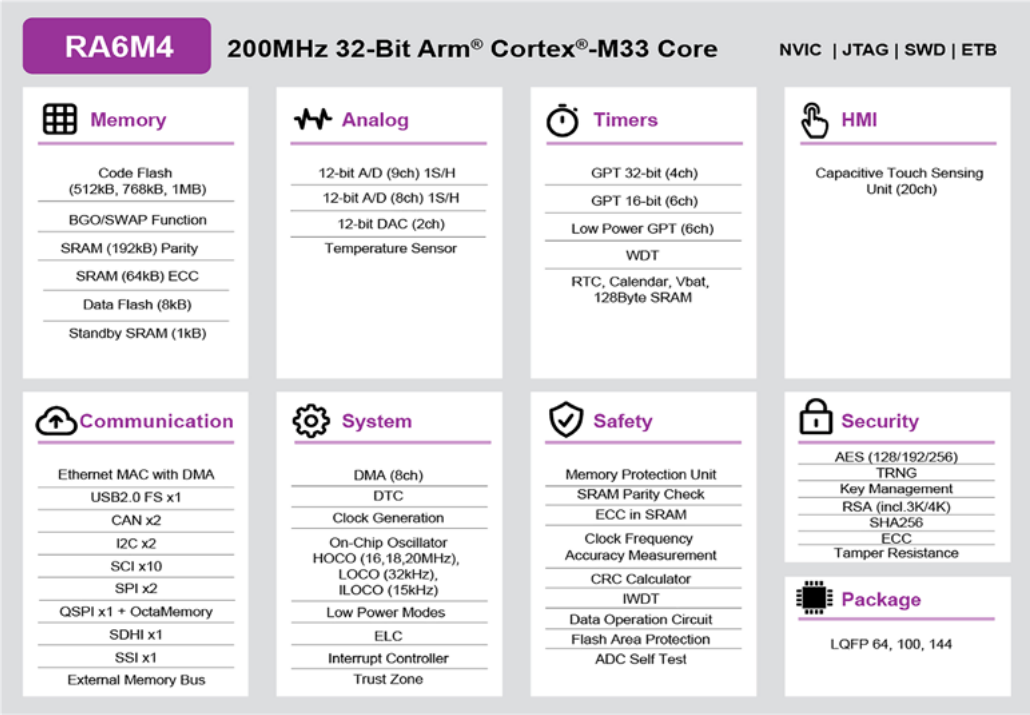


Figure 1-3: Block diagram of the RA6M4 Group of RA-devices

Figure 1-3 shows the key features and peripherals of the RA6M4 Group devices, representing the advanced performance RA6 Series of the RA Family. The majority of the examples and projects throughout this book are based on the Evaluation Kit for this microcontroller.

As it is sometimes, especially for beginners, not easy to decode the different digits and letters inside the part numbers for the RA Family, *Figure 1-4* explains the meaning of the different fields.

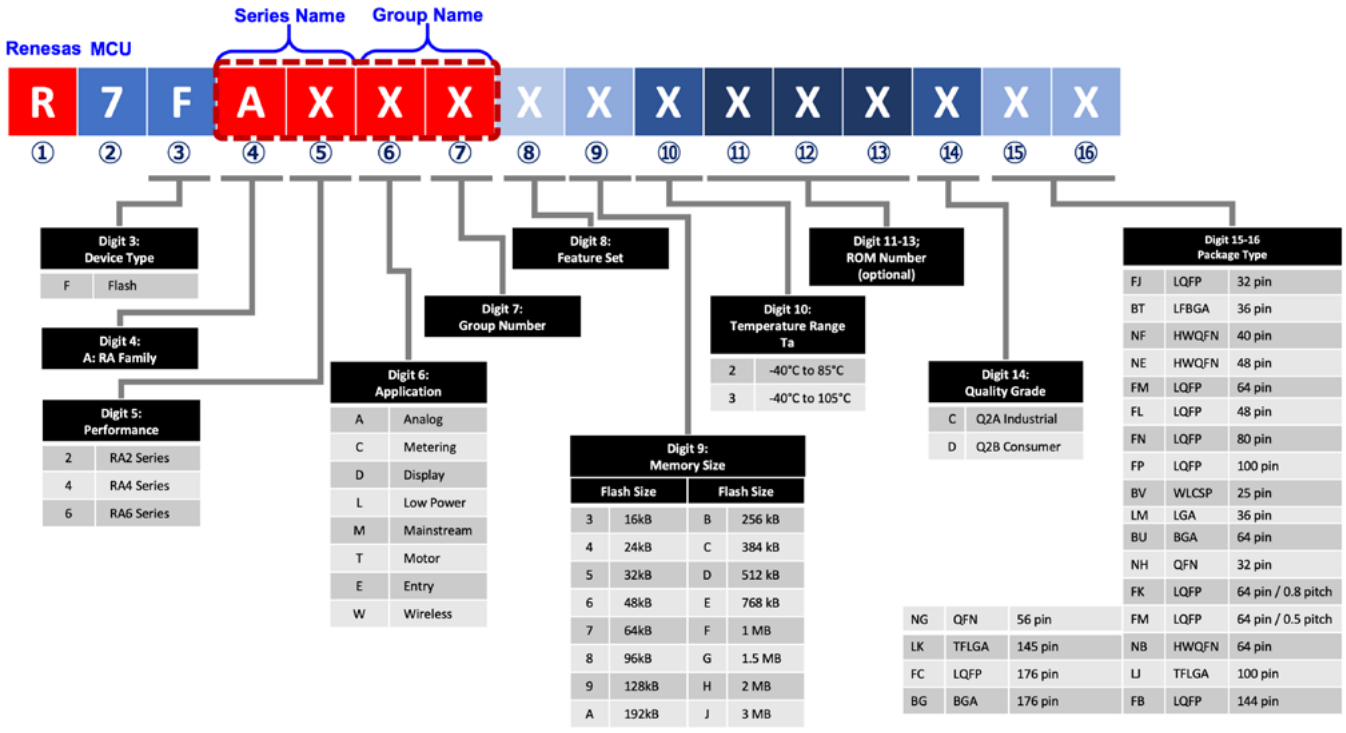


Figure 1-4: Part number decoder for the RA Family

1.3 The Flexible Software Package

The Renesas Flexible Software Package (FSP) for the RA Family of microcontrollers provides a quick and versatile way to create software for secure connected devices for the intelligent Internet of Things and is specifically optimized for the architecture of the RA microcontrollers. It features ready-to-use middleware stacks and protocols, for example for TCP/IP or security, a Board Support Library (BSP) which provides start-up and initialization code for the MCUs and the development kits from Renesas, and Hardware Abstraction Layer (HAL) drivers for all peripherals. These drivers are not only high performant, but also have a very small memory footprint.

All drivers, stacks and middleware functions can be accessed through easy-to-use Application Programming Interfaces (APIs), allowing for effortless interchangeability, and work with Real-Time Operating Systems (RTOS) as well as with bare metal implementations. Also, awareness for Arm's TrustZone is built into all levels of the software and the unified APIs from Arm for security are used. The FSP is Open Source and the full source code is provided but is limited to Renesas hardware.

In addition to the software mentioned above, the FSP also includes FreeRTOS™ from Amazon and Azure™ RTOS from Microsoft as Real-Time Operating Systems which are accessed through Arm's Cortex Microcontroller Software Interface Standard (CMSIS) RTOS interface. This standard interface enables software engineers to use any RTOS of their choice without losing the benefits of the FSP.

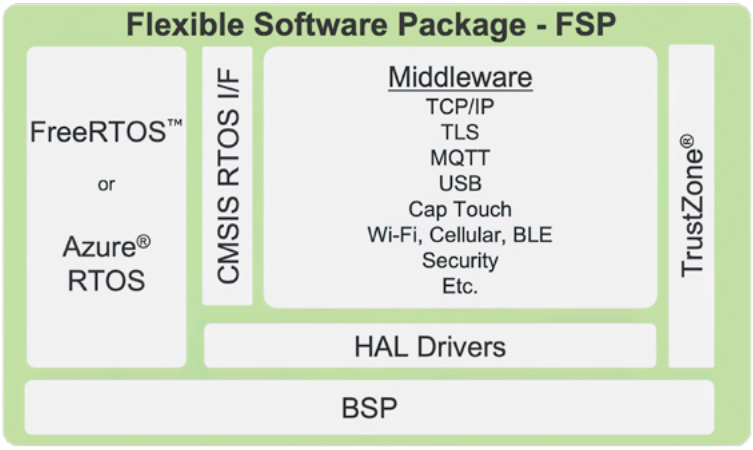


Figure 1-5: The architecture of the Flexible Software Package

To make the use of the FSP as easy and hassle-free as possible, it comes with an intuitive configurator and code generator which not only allows the user to initialize the MCU and its peripherals, but also the chosen RTOS and the middleware modules. Developers are not tied to what the FSP offers: their own application code and middleware modules can be integrated at any time. This is also valid the other way around: The FSP can be used together with the whole Arm software ecosystem.

If you are curious about the details of the FSP please be patient. We will cover them in chapters 2 and 3.

1.4 RA Tools and Kits: A Short Overview

Renesas made a huge effort to create useful software and hardware development tools that can be utilized to explore the technical capabilities of the RA Family and that will take the user beyond the evaluation stage all the way to production.

Beside the Flexible Software Package already explained in chapter 1.3, the development tools available at the time of writing include:

- **e² studio:** The Eclipse-based Integrated Development Environment (IDE) is available for the Microsoft Windows® and Linux operating systems and contains all the tools necessary to create, compile and debug projects for the RA Family. Its configurators allow easy graphical access to tasks like the creation of new projects or to the configuration of different hardware features, like the clock module or the pin functionality. It also enables engineers to pick the various software modules for the middleware, the drivers, the Board Support Package, and the RTOS. And all that with no need for studying the user's guide in great detail.
  These configurators will create all the necessary settings and the initialization code automatically and include an error checking feature to detect problematic setups already at design time. For code generation, e² studio uses the GCC compiler from the GNU Arm® Embedded Toolchain.
- **QE Tools:** QE is an acronym for "Quick and Efficient" and the QE for Capacitive Touch and QE for Bluetooth® low energy 5.0 integrate directly into e² studio. While QE for Capacitive Touch simplifies the initial settings of the touch user interface and the tuning of the sensitivity, QE for BLE 5.0 can be used to test the communication features of the Bluetooth low energy stack.
- **RA Smart Configurator:** The RA Smart Configurator (RASC) is a desktop application providing users with the same functionality the Smart Configurator integrated into e² studio has. With the RASC, developers using third-party toolchains and IDEs get access to the project setup and the graphical configuration of the software system (BSP, HAL drivers, middleware, RTOS), the pin assignments, and the clock setup in the same way as users of e² studio.
- **Evaluation Kits:** Renesas provides individual kits for several products of each RA Series, and at least one kit per superset device in each MCU Group. They integrate a debugger and enable designers to fully evaluate the devices' digital and analog pins, and its connectivity and security features – even before one's own hardware is ready.
  The Evaluation Kits can be expanded further through connectors for boards compatible to some of the most popular ecosystems like Digilent Pmods or Arduino™ compatible boards. As design packages containing BOM, drawings, design- and manufacturing files are provided, the kits can serve as a quick start for building custom hardware, speeding the development of applications even more.
- **Third-Party Tools and the Renesas RA Partner Ecosystem:** A comprehensive partner ecosystem delivers a variety of software and hardware building blocks that will work out-of-the-box with Renesas RA Family MCUs. Developers can choose from an array of solutions, including technologies such as security, safety, and connectivity. Examples of these third-party tools are the Microcontroller Development Kit (MDK) from Keil® and IAR Systems® Embedded Workbench®.

Design support going beyond software and hardware tools is provided as well: On the Renesas webpage of the RA Family (**https://www.renesas.com/ra**), a wide range of white papers, teaching videos and example projects can be found. They help developers and systems designers to come up to speed quickly, and with minimal effort. And, if you want to dig deeper into some topics, the Renesas Academy (**https://academy.renesas.com/?eid=1618**) is a good place to look for training courses.

Sometimes things go wrong or are getting difficult during development, but help is always just a few mouse-clicks away. Good places to start are the RA Forum (**https://www.renesas.com/ra/forum**) or the Knowledge Base (**https://en-support.renesas.com/knowledgeBase**). The Forum and the Knowledge Base can also be accessed right out of e² studio:
Simply go to *Help → RA Helpdesk* or to *Help → RenesasRulz Community Forum*.

Points to take away from this chapter:

- The Renesas RA Family of microcontrollers is delivering the promise of a secure, connected, and intelligent IoT.
- The RA Family is supported by a variety of software and hardware tools and by the RA Partner Ecosystem.
- Help is easily available from the Renesas website.

2 THE FLEXIBLE SOFTWARE PACKAGE (FSP)

What you will learn in this chapter:

- What the different components of the Flexible Software Package (FSP) are and how they are layered.
- Details of the different layers and how they work together.
- Specifics of the FreeRTOS™ and Azure® RTOS Real-Time Operating System and the benefits of using an RTOS.

The best microcontroller (MCU) cannot unlock its full potential if either the software running on it is not up to the task, or the software ecosystem is too complicated to be useful to the developer. A well-thought-out concept like the RA Family of MCUs together with its suite of software development tools and the RA Partner Ecosystem helps engineers to create applications which are not only easy to write and simple to maintain, but which also make the best out of the performance and the features of the microcontrollers of this family.

The simplicity developers find inside the RA Family was attained by making all major parts – MCUs, software, tools, kits, and solutions – work perfectly together and Renesas has put a lot of effort into that to reach this ambitious goal.

The Flexible Software Package (FSP) was specifically optimized for the architecture of the RA Family of MCUs, which, in turn, was developed with the software in mind. The primary goal during the development of the FSP was to supply engineers with lightweight and highly efficient functions and drivers to ease implementation of common use cases in embedded systems, like communication and security. They provide an open software ecosystem which also allows the flexible use of legacy code and collaboration with third parties.

The FSP integrates middleware stacks, production ready RTOS-independent Hardware Abstraction Layer (HAL) drivers, and, as the basis for all of that, the Board Support Package (BSP). Packaged with the FSP come the widely used Real-Time Operating Systems FreeRTOS™ from Amazon Web Services® and the Microsoft Azure™ RTOS. This results in a single optimized and easy-to-use high-quality software package for embedded system design, which is scalable and where all the functionality can be accessed through simple and robust API (Application Programming Interface) calls, allowing for effortless interchangeability. The entire FSP is fully supported by Renesas and developers have complete visibility of the FSP's source code, either from within the Integrated Development Environment or by viewing or downloading it from the FSP's GitHub® repository at **https://github.com/renesas/fsp/releases**.

2.1 Introduction to the FSP

The Flexible Software Package (FSP) is a comprehensive piece of software. Its purpose is to provide fast and efficient drivers and stacks with a low memory footprint that cover most scenarios during an embedded systems software development. Included in the FSP are the following parts:

- The **Board Support Package (BSP)**, customized for every hardware kit and microcontroller of the RA Family. It provides the start-up-code for all supported modules and supplies the foundation to allow the FSP modules to work together without issue. Developers using their own hardware can take advantage of the BSP, as it can be tailored for their end products and their own board with the help of the User Pack Creator built into e^2 studio.
- The RTOS-independent **HAL Drivers**, providing efficient drivers with a small memory footprint for all on-chip peripherals and systems services. They eliminate the need for deep study of the documentation of the underlying hardware in the microcontrollers as they abstract the bit-settings and register addresses from you.
- The **Middleware stacks and protocols,** which work with or without an RTOS and use the unified APIs from Arm®. They ease the implementation of connectivity features like WiFi, Bluetooth® low energy, or MQTT connections to cloud services. Other stacks are included as well, for example support for USB transfers, graphics processing, or capacitive touch.
- The **FreeRTOS™ Real-Time Operating System,** providing a multitasking real-time kernel with preemptive scheduling, a flexible allocation of RAM for objects, and different methods for task notifications, queues, semaphores, and buffers. The libraries FreeRTOS+FAT and FreeRTOS+TCP supply additional functionality for applications needing connectivity. Using FreeRTOS is optional; the FSP can also be used with bare metal implementations or with any other RTOS.
- The **Microsoft Azure® RTOS Real-Time Operating System,** comprising components like Azure RTOS ThreadX® and various stack and middleware elements like file systems (FileX®), graphical user interfaces (GUIX™), USB and TCP/IP communication (USBX™ and NetX Duo). The ThreadX RTOS provides a preemptive scheduler, inter-thread communication and synchronization, software timers, and a small memory footprint. Its memory management allows the flexible allocation of RAM and the detection of stack overflows. All these packages integrate into the FSP ecosystem and simplify the connection to the Azure IoT.
- Additional **third-party software solutions** are included in the FSP as well. Examples are the Arm® Cortex® Microcontroller Software Interface Standard (CMSIS) hardware abstraction layer, the Arm Mbed™ Crypto and TLS cryptography libraries, the Arm Littlefs fail-safe filesystem, the emWin embedded graphics library and J-Link® debugger software from Segger, as well as the TES D/AVE 2D graphics rendering library.

One of the goals during the development of the FSP was to create easy-to-use software with uniform and intuitive APIs that are well documented. Engineers have for each module a detailed user documentation at hand, including example code, either at the GitHub repository or through the Smart Manual feature of e^2 studio, which is displaying the information right where it is needed: inside the development environment. As Doxygen is used as the default documentation tool for the FSP, additional details can also be found in Doxygen comments throughout the source code of the different modules.

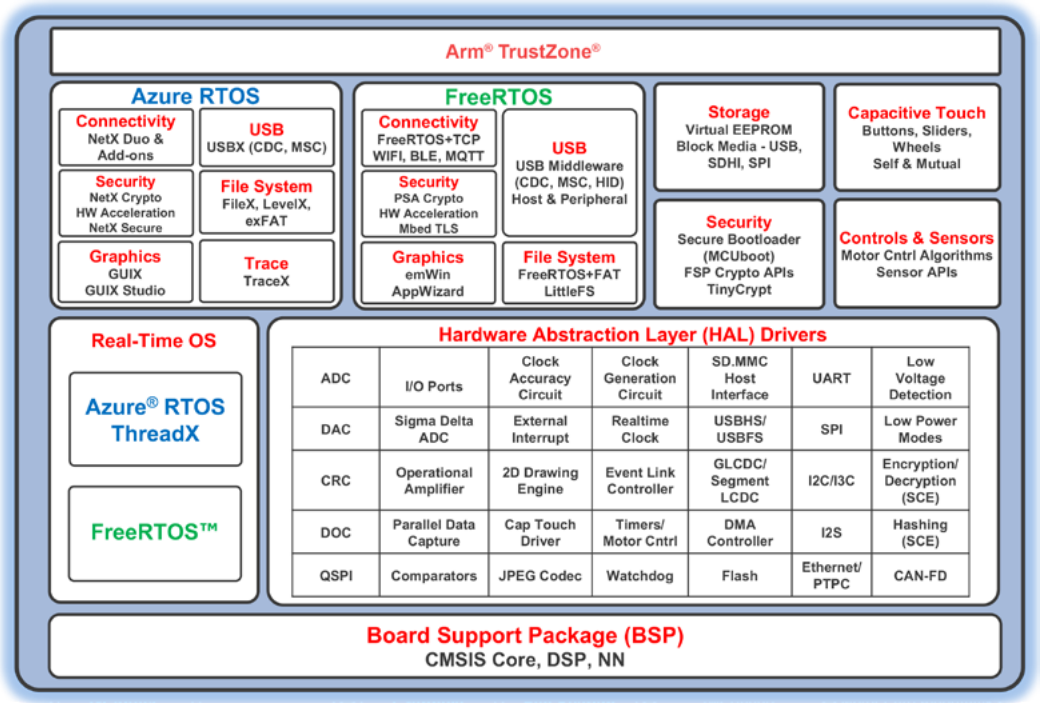


Figure 2-1: The different layers of the Flexible Software Package and their functionality

Build-time configurations for each module are included as well. They can be used to optimize the size of the module for the feature set required by the application. The FSP is highly scalable as its modules are the same for any microcontroller of the RA family, provided the MCU has the peripherals needed by the module to be used. The quality of the FSP is ensured through peer reviews, automated requirements-based testing, and automated static analysis.

Figure 2-1 shows the architecture of the different layers. Each of them can be accessed from the user application through API calls, but it is strongly encouraged that applications call only the API functions of the layer at which they are interfacing. While programmed according to the ISO/IEC 9899:1999 (C99) C programming language standard, all FSP modules allow applications being written in C and C++, in either e^2 studio or one of the supported development environments. Awareness for Arm's TrustZone® is built into all levels of the software to enable designers to create secure programs easily.

Updates and bug-fixes to the FSP will happen regularly, with the scheduled releases following a roadmap and each major or minor release introducing new features. For the long term, Renesas has a great reputation for supporting their products way beyond the point at which other manufacturers would cease maintenance. This means that you are not left alone with software distributed on an "as is" basis.

2.2 Introduction to the Board Support Package (BSP)

The Board Support Package builds the bottom layer of the Flexible Software package (FSP) and it provides the foundation to allow other FSP modules to work together without issues. To achieve this, its main responsibility is to get the MCU from reset to the user application. On its way there, it will set up the clocks, interrupts, stacks, heaps, the stack monitor, and the C runtime environment. It also configures the I/O-pins of the ports and performs any board specific initialization.

Therefore, this package is specific to a certain combination of board and MCU. Both are selected during the setup of a project in e² studio using the project wizard of the IDE (Integrated Development Environment). Every kit provided by Renesas is supported by a BSP. The configurators inside e² studio will extract the necessary files from the FSP and set them up them based on the inputs made in the user interface. The Board Support Package itself is heavily data driven and consists of configuration files, header files and APIs (Application Programming Interfaces).

The core of the BSP is compliant with CMSIS (Arm® Cortex® Microcontroller Software Interface Standard), following the requirements and the naming conventions of that standard. The Board Support Package uses a static macro list of data for each MCU, where the macros themselves are identical for all processors, but they will have different values for the different MCUs.

The BSP provides public functions, available to any project using the package, that allow access to the functionality that is common across the MCUs and boards supported by it. Functions include locking / unlocking the hardware and software, interrupt handling, registering callback functions and clearing flags, software delay and register protection. The names of these routines start with `R_BSP_`, the associated macros with `BSP_`, and the data type definitions with `_bsp` for easy differentiation from other parts of the FSP. The only exception are routines providing functionality described in the CMSIS core. Additionally, the BSP includes some routines for pin-functionality like read or write and for returning information (number and I/O-pins) about the LEDs being available on a board. These functions can be used directly by the application code.

New kits and devices will be added to the BSP once they become available, assuring a solid long-term base for current and new designs. Board level support for custom boards can easily be generated using the User Pack Creator built into e² studio.

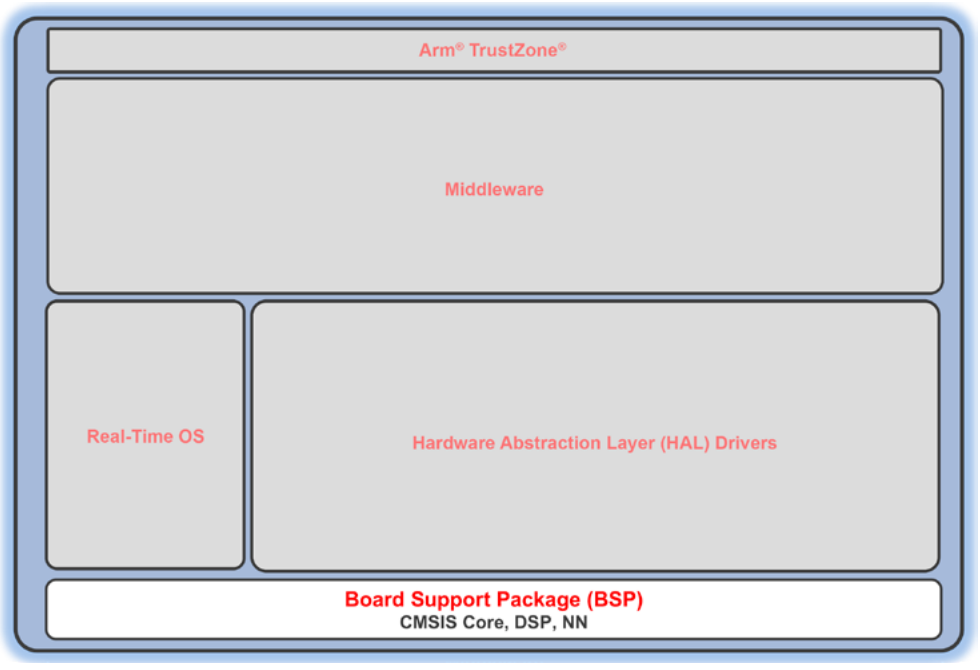


Figure 2-2: The Board Support Package lays the functional foundation of the Flexible Software Package

2.3 Introduction to the HAL Drivers

On top of the Board Support Package (BSP) sits the Hardware Abstraction Layer (HAL), which provides efficient device drivers with a small memory footprint for the peripherals and which aligns with the registers of the MCU to implement easy-to-use interfaces, insulating the programmer from the hardware. It is a collection of modules (see *Figure 2-3*) and each is a driver for a peripheral available in a microcontroller

of the RA Family like the SPI (Serial Peripheral Interface) or the ADC (Analog-to-Digital Converter), and their names begin with `r_` for an easy differentiation from other parts of the Flexible Software Package (FSP). All these modules are inherently RTOS independent.

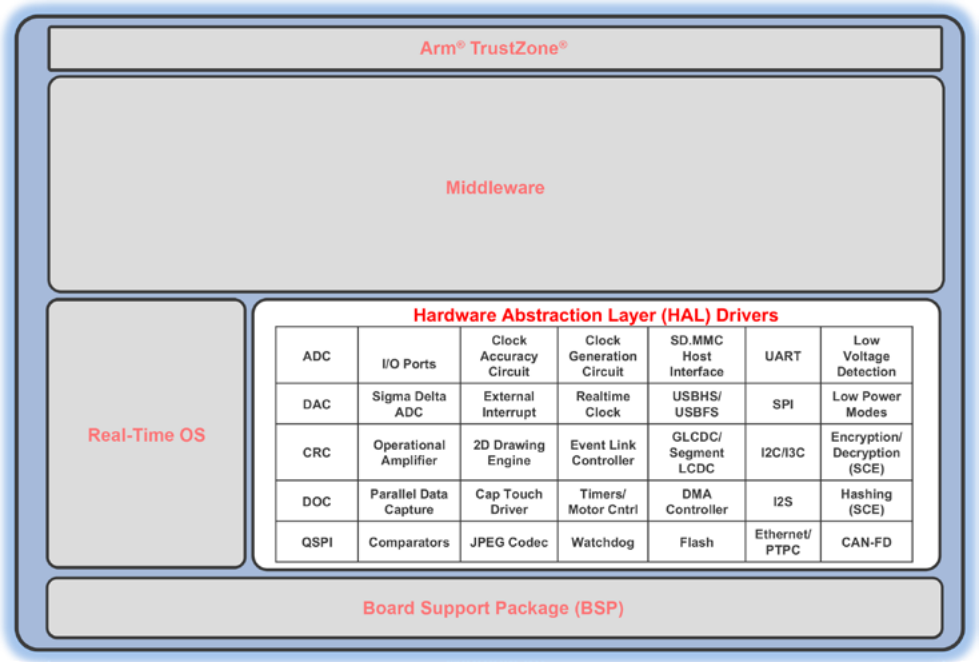
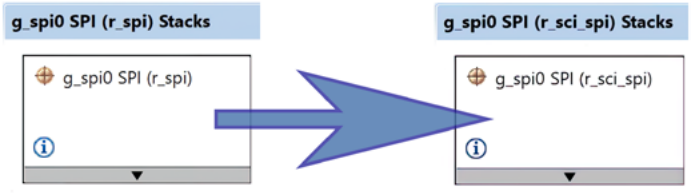


Figure 2-3: The HAL-layer provides drivers for the on-chip peripherals

The interface to abstract the hardware is consistent throughout all modules of the HAL and can be extended. Some of the peripherals support multiple interfaces, while some interfaces are supported by multiple peripherals. The advantage of that is the flexibility gained, as requirements can be modified at a higher level. If, for example, the code was originally written for the dedicated hardware SPI peripheral, but later on the SPI-functionality of the SCI (Serial Communication Interface) needs to be used, it is sufficient to change the configuration information. The application code itself remains unchanged (see *Figure 2-4*). While the API functionality can be accessed directly through the HAL interfaces, most of the functions can also be accessed through the different middleware stacks and protocols available in the FSP, which actually is the preferred method.




```
void hal_entry(void)
{
    fsp_err_t err = FSP_SUCCESS;
    err = R_SPI_Open(g_spi0.p_ctrl,g_spi0.p_cfg)
```

Figure 2-4: Even if the underlying peripheral is changed, in this example from the SPI peripheral to the SCI port, the application code stays the same

The production readiness of the HAL drivers is assured through unit and system tests executed using industry standard tools for static and dynamic analysis. All drivers are non-blocking by default and return an execution status. Also, they do not allocate any memory themselves; the working memory is passed to the drivers by the calling routine.

2.4 Introduction to the Middleware

On top of the HAL layer and just below the user application sits the middleware layer, providing stacks and protocols to the application. Its intention is to ease the use of complex peripherals like Ethernet and USB or the different security features of the RA Family of microcontrollers (MCUs). The modules of the middleware layer provide abstractions of various system-level and technology-specific services, enabling a rich functionality with simple APIs. While they integrate with the FreeRTOS™ to manage resource conflicts and synchronization between multiple user threads, they can also be used on bare-metal implementations or with any other Real-Time Operating System.

The middleware modules of the Flexible Software Package (FSP) provide effortless connectivity options like the scalable and thread-safe TCP/IP stack from FreeRTOS+TCP or easy-to-use APIs to access the Ethernet peripheral on the silicon of the MCUs. Other modules support WiFi and socket implementations, including device configuration. Also, Bluetooth® low energy (BLE) 5.0 functionality is available, enabling users to control the BLE radio peripheral on selected microcontrollers like the RA4W1 Group devices.

USB communication is supported as well in host and device mode with several classes. All of the mentioned middleware modules can be set up easily through the different configurators or the QE Tools.

With FreeRTOS+FAT and Azure RTOS FileX, DOS-compatible file system are available which enable file-based access and storage and which are thread-aware and scalable. They allow implementations using FAT12, FAT16, or FAT32 file systems. Arm's LittleFS, a little fail-safe filesystem developed especially for microcontrollers and embedded systems is supported as well.

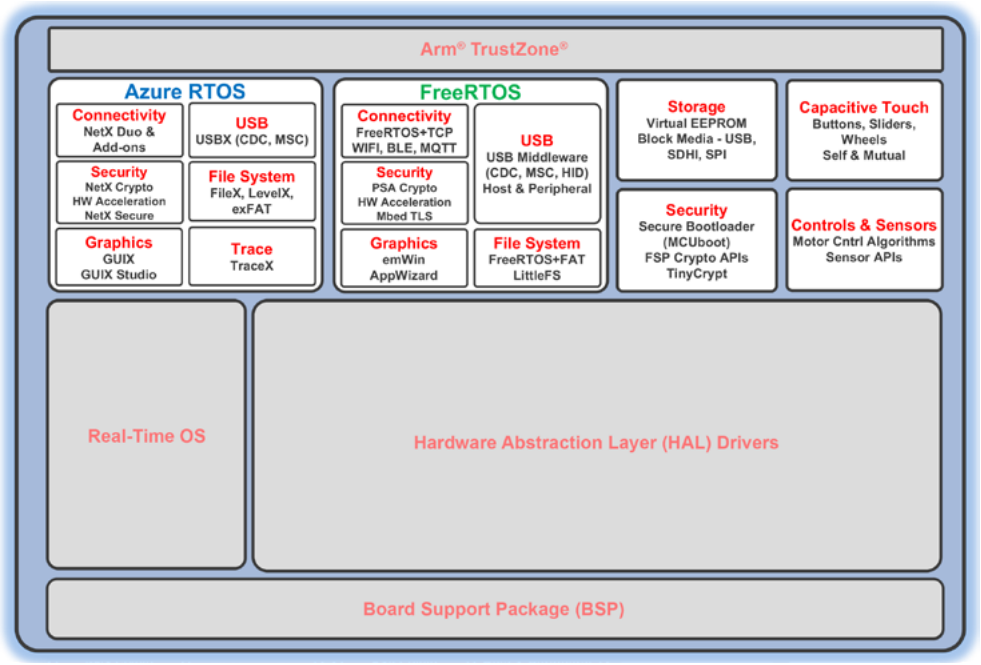


Figure 2-5: The different middleware modules of the FSP provide easy connectivity and other features

Other modules help software engineers to create applications with end-to-end cloud connectivity. These modules include support to directly connect a user's software with Amazon Web Services®, Microsoft Azure® and Google Cloud Platform™ Service. Communication is provided through the Ethernet or WiFi interfaces and supports the MQTT protocol. The security of the connection is assured by the Transport Layer Security (TLS) protocol of Arm's Mbed™ Crypto interface.

The cryptographic APIs of the FSP are based on Arm's Mbed Crypto and they support hardware and software cryptography. AES 128 and AES 256, SHA-256 and SHA-224 calculations, and RSA 2048 are taken care of, as is Elliptic-Curve Cryptography (ECC) and True Random Number Generation (TRNG). A hardware acceleration for the Mbed Crypto implementation of the Arm PSA Crypto API is available as well, as is secure debugging.

For applications using capacitive touch sensing, drivers and middleware based on the Renesas Capacitive Touch Sensing Unit (CTSU) technology are at one's fingertips. They include APIs to simplify the integration of widgets like buttons and sliders, which are easily created through the QE for Capacitive Touch tool.

For some MCUs, for example the RA6M3 Group devices, middleware to support graphics is ready to use: the FSP comes with the Segger emWin graphics libraries for high-quality GUIs. They implement functions for image processing, 2D-acceleration, window manager, event processing, rotation/scaling, alpha/anti-aliasing, and much more.

Developers using the middleware stacks and protocols benefit from a higher level of abstraction, reuse of code on different processors and therefore potentially products, more consistent code, and are offloaded from the burden of having to reinvent commonly used tasks across applications. At the same time, the other parts of the Flexible Software Package reduce the need to directly interface with the hardware available in the microcontroller even further.

2.5 Introduction to the Real-Time Operating Systems

The Flexible Software Package (FSP) comes not just with one, but with two quite popular Real-Time Operating Systems (RTOS): the FreeRTOS™ from Amazon Web Services® and the Microsoft Azure® RTOS ThreadX®. Both provide not only the services software engineers expect from operating systems, such as real-time threads, preemptive schedulers, memory managers, or inter-thread communication via queues, semaphores, and buffers. They also provide additional middleware libraries. These include file systems, communication stacks like USB and TCP/IP, and much more. This way, they actively support engineers while developing deeply embedded applications that are either standalone or connected to various cloud services.

But why use an RTOS at all? And what are the specific features of the respective operating systems? The following paragraphs will give the answers to these questions.

2.5.1 Why Using an RTOS?

Not so long ago, the life of a software-developer for an embedded system was comparably straightforward. Most systems consisted of a single background task that was running in an endless loop inside the main() function. It executed tasks like reading inputs from I/O-pins (for example the state of a push-button), and then performed certain calculations such as multiplying a measured voltage and a current to obtain a power consumption, and finally updated discrete outputs, and so on.

The few peripherals connected to the microcontroller issued an interrupt to notify the CPU that maybe a new conversion result from an analog-to-digital converter is available and should be read. Or that the RS-232 port finished transmitting the last word and is waiting for new data. These asynchronous events have been traditionally handled inside the interrupt service routines. If no interrupt was pending, the software was idling in the background loop, standing by for interrupts to occur.

With embedded systems growing more complex, having a higher connectivity to the outside world, or more user interfaces to serve, and more tasks to execute, the bare metal setup mentioned above is difficult to maintain, and an operating system is

becoming not only something preferable, but a necessity. Why? The response time of the background loop is hard to predict and definitely not deterministic in a bare metal implementation, as it is affected by the decision tree inside the code and would even change once a modification to the software is made. Another disadvantage was that all tasks (or "threads") in the background-loop had the same priority, as the code was executed serially, which means that the reading of the push-button and the power-calculation in the example above would always have been executed one after each other, even if the button was pressed multiple times. In other words, some of these events could have been missed, something which is a no-go in embedded computing. Today's systems need to be more responsive and more predictable; they need to be real-time.

One possibility would be to write the required functionality yourself. But designing and implementing your own scheduler or intertask communication is complex and error-prone. Moreover, such software is hard to port to a new processor and even harder to maintain. So, a Real-Time Operating System makes the life of a programmer easier, especially if the RTOS is included in the software starting from day 1.

Having seen the clear need for an operating system, the next question is: why not use an off-the-shelf OS like Windows® or Linux? They are part of our everyday life and seem to do their job quite well. There are a couple of reasons for not using them. First of all, they offer by far too many features, most of them not necessary in an embedded system, and they are not configurable enough. Secondly, they need more resources and memory space than what is available in a resource-restricted embedded application. And last but not least, their timing uncertainty is still too large to satisfy the demands of real-time systems.

All of this suggests that there are special requirements for an OS running on our embedded systems. The first and most important one is the predictability of the timing behavior. An RTOS needs to be deterministic, which means that the upper limits of the blocking times, like the time during which interrupts are disabled, need to be known and available to the developer.

The second requirement is that the RTOS has to manage the timing and scheduling. It has to be aware of the timing constraints and deadlines and it must provide a time service with a high resolution. And finally, the operating system should be fast, small and configurable.

Another major function made available by an RTOS is thread management, permitting the execution of quasi-parallel tasks on an MCU using threads (a thread can be understood as a lightweight process) by taking care of the state of the threads, the queuing of the processes, and allowing for preemptive threads and interrupt handling. Other services provided to the application are scheduling, thread synchronization, interthread communication, resource sharing, and a real-time clock as time reference.

Many software designers balk at the idea of using an RTOS, because they fear its complexity and learning curve. But a real-time operating system offers plenty of benefits like the increased responsiveness to real-time events, the prioritization and easy insertion of threads, the reduced development time once the first steps are mastered, and the services added to the application.
So, the advantages outweigh the initial difficulties by far.

As with all good things in life, there are of course some disadvantages in using an RTOS: It needs additional RAM and Flash memory, and each thread requires its own stack, increasing the memory need even further. Cost might be a factor as well, but this is not really an argument with FreeRTOS™, as this operating system comes for free.

Having seen the benefits of using an RTOS, it is now time to look into the various features of FreeRTOS and Azure RTOS.

2.5.2 The Main Features of FreeRTOS™

FreeRTOS™ was developed specifically for microcontrollers and small microprocessors in embedded systems where memory resources are scarce, and a proven robustness is necessary. Nowadays it is the de-facto standard for embedded operating systems. It provides a multitasking scheduler, several options for memory allocation of objects, and methods for task notifications, queues, semaphores, and different buffers.

The processing overhead of FreeRTOS is very small, as is the memory footprint. Typically, an FreeRTOS kernel binary image needs between 6 and 12 kB of Flash memory plus a couple of hundred bytes of RAM for the kernel itself, leaving the resources of the microcontroller mostly to the application. This operating system was designed to be simple: its kernel consists of only three common source files and one file being specific to the microcontroller in use.

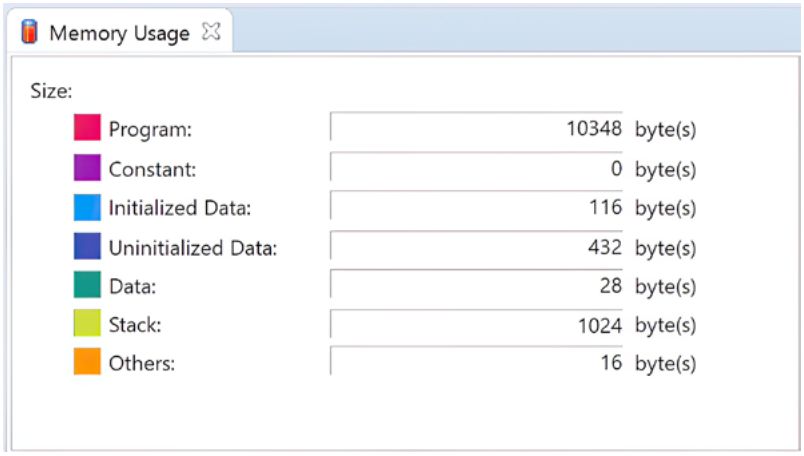


Figure 2-6: FreeRTOS has a very small memory footprint. In this example, an empty FreeRTOS FSP project with no threads uses just a bit over 10 KB Flash memory, and this includes the initialization of the MCU through the BSP

This feature-rich and fully maintained Real-Time Operating System was developed together with world leading MCU manufacturers over a period of more than 15 years and is now owned by Amazon Web Services®. It is distributed freely as source code under the MIT open-source license and a version adapted to the RA Family is integrated into the Flexible Software Package (FSP). The operating system is very strictly quality managed, and the core source files conform to the MISRA® coding standard guidelines.

FreeRTOS allows for a preemptive or a cooperative operation, supports multiple threads, queues, mutexes, semaphores, and software timers, and has a very flexible task priority management. The task notification uses a lightweight mechanism and is fast and versatile. Provisions for trace recording and for gathering statistics of a thread at run-time are available.

Different resources of the RTOS – like threads, mutexes or the memory management – are configurable through graphical user interfaces, either from inside e^2 studio or by using the Smart Configurator available for other Integrated Development Environments (IDEs), making an implementation of the RTOS-functionality quite easy.

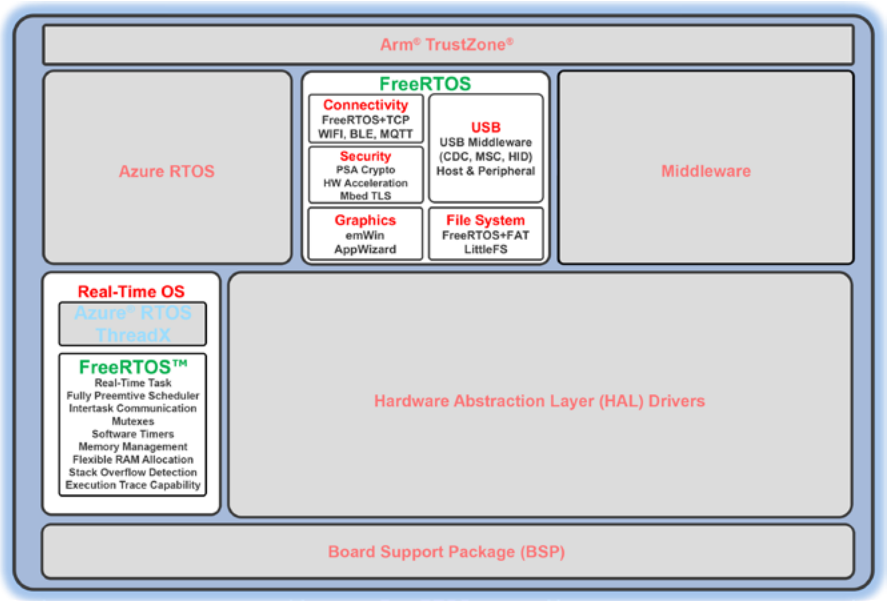


Figure 2-7: FreeRTOS™ provides a multitasking scheduler, options for memory allocation of objects and methods for task notifications, queues, semaphores, and buffers

The real-time scheduling algorithm of FreeRTOS always ensures that the thread with the highest priority queued as ready-to-run will be executed. If more than one thread of the same priority is in ready state, there are two options: If time slicing is disabled in the configuration file of the RTOS, which is the default for the FSP, the scheduler will not switch between the threads once an RTOS tick interrupt has occurred. If time slicing is enabled, the scheduler will switch between the threads of the same priority on every tick interrupt. Independent of the setting for time slicing, round-robin is used to ensure that all threads will enter the running state at some point.

To prioritize threads, there are up to 32 priority levels available to the software-engineer if the so-called architecture optimized method of prioritization is enabled, and an indefinite number if this method is disabled and the generic method is used. As each priority level consumes RAM inside the kernel and has a negative impact on the execution time, the number of levels should be kept as low as possible. Also, it is very rare that an application requires more than 10 to 15 different priorities, as threads sharing the same level will be scheduled using round-robin, assuring that each thread will get executed.

Kernel objects in FreeRTOS, like threads, queues, or semaphores, can be created completely statically at compile-time or dynamically at run-time. Both versions have their benefits and their disadvantages. It is the decision of the developer: a decision he has to do based on the requirements of his application. Dynamic memory object allocation is simpler and can potentially reduce the maximum RAM usage of the application, as the memory can be re-used once an object is deleted. Dynamic allocation is automatic, so the writer does not need to take care of this task.

Static allocation of memory provides the developer with more control, as he can place objects at memory locations of his desire. Also, he can determine the amount of RAM required by the application already at link-time. The decision between static and dynamic allocation is not final: if necessary, both methods can be used inside the same program, allowing for maximal flexibility and an optimal management of the scare resource “memory”.

Other resources managed by the RTOS are the software timers. These application timers are available in two operation modes: one-shot and auto-reload. A one-shot timer will call a user-function only once after the timer expires, while a periodic timer enters the user-function repeatedly at a fixed interval. The implementation of the software timers in FreeRTOS is very efficient, as a timer

does not eat up any processing time unless it has expired, and it does not add any overhead to the tick interrupt. Additionally, the implementation does neither walk any linked-list structures while the interrupts are disabled and does not enter any timer callback function from inside an interrupt service routine (ISR).

Software timers need to be created before use and they execute in the context of the timer service task that is created automatically once the scheduler is started. The creation of the timers can be done either through an API call, or statically at compile-time. The period of timers is specified in ticks and a timer cannot have a finer resolution than one kernel tick period.

Synchronization of threads and communication between threads, or between interrupts and threads, is another big topic in embedded systems. FreeRTOS provides several mechanisms for that and makes this job very easy and straightforward.

The primary form of interthread communication are queues. They are thread-safe, first in, first out (FIFO) buffers and new data is added to the end of the queue by copying the message. This means that the message itself is copied in full and not only a reference.

Semaphores and mutual exclusions (mutex) help preventing priority inversion and serve synchronization purposes. For simple events like unblocking another thread or updating a notification value, there is a lightweight and fast alternative to binary or counting semaphores and sometimes to queues: direct-to-task notifications. They allow threads to interact with other threads or with ISRs without the need for other objects, saving memory and time.

.

The last synchronization features to be mentioned here are message queues and stream buffers. With stream buffers, developers can pass a stream of bytes between threads or from an ISR to a thread. This byte-stream can have any length and any number of bytes can be written and read at one time. On the other hand, message buffers make it possible to pass variable-length discrete messages. The length does not matter, but a, for example, 32-byte message written to the message buffer by the sender needs to be read as 32-byte message by the receiver. They cannot be read as individual bytes.

And finally, FreeRTOS also includes two other very important features: a built-in trace functionality to view the sequence of the different events and the possibility to detect stack related issues like overflow. Both are really helpful when troubleshooting an application!

2.5.3 The Main Features of Microsoft Azure® RTOS ThreadX®

ThreadX® is part of the Microsoft Azure® RTOS. This Real-Time Operating System (RTOS), originally developed by Express Logic, targets high-end and graphic rich applications, as well as embedded systems with limited memory and special requirements in terms of determinism. It features a small Flash-memory, a small RAM requirement and a short context switching time.

As a multi-threading RTOS, ThreadX uses multiple advanced scheduling algorithms, provides real-time event trace, message passing and interrupt management, as well as many other services, and is fully deterministic. It has a proven reliability record with over 10 billion deployments in the consumer, medical electronics, and industrial automation markets, and meets numerous important safety and quality standards.

Other advanced features offered are, for example, its picokernel™ architecture, Preemption-Threshold™ scheduling, Event-Chaining™ and a rich set of system services. It also integrates with the other components of the Microsoft Azure RTOS, like FileX®, GUIX™,

TraceX®, NetX Duo™, LevelX™, or USBX™. It has a pretty small RAM-memory footprint and its designers optimized it for very fast execution and low overhead, leaving the resources of the microcontroller mostly to the application.

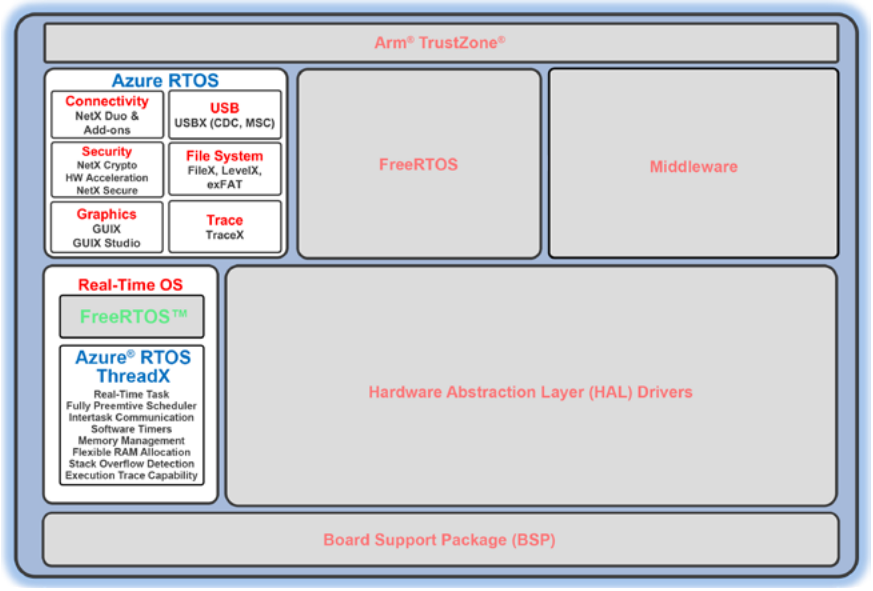


Figure 2-8: ThreadX® includes a preemption-threshold multitasking scheduler, event-chaining and communication, memory management, and other system services

The enhanced real-time scheduling algorithms and efficient multitasking routines provide round-robin scheduling and time slicing, as well as preemptive and Preemption-Threshold™ scheduling. To prioritize threads, there are up to 1024 priority levels available to the software-engineer, with the default number being 32.

Resources in an embedded system are mainly limited by execution time and available memory. ThreadX provides several powerful options to manage them. Memory allocation needs to be fast and deterministic, so ThreadX provides the ability to create and manage any number of pools of fixed-sized memory blocks. As the pools are of a fixed size, memory fragmentation is not a problem, but the size of a memory block must be the same or larger than the largest memory request from the application or the allocation will fail. But making it large enough for that scenario might waste memory if requests come in with different block sizes.
A workaround for that is the creation of several memory block pools that contain different sized memory blocks. The fast allocation and de-allocation of these blocks outweigh this disadvantage, as these allocations are done at the head of the available list. This provides the fastest possible linked list processing and might keep the actual memory block in cache.

ThreadX allows not only block pools with fixed block sizes but also the creation of multiple byte pools, which behave in a similar way to a standard C heap. Memory is only allocated with the desired size in bytes, based in a first-fit manner. The disadvantage of that is that, like heaps in C, byte pools get easily fragmented, creating a somewhat non-deterministic behavior.

Other resources managed by the RTOS are the application timers, allowing an unlimited number of software timers to be created. These application timers are available in three operation modes: one-shot, periodic, and relative. A one-shot timer will call a user-function only once after the timer expires, while a periodic timer calls a user-function repeatedly after a fixed interval. The relative timer is a single continuously incrementing 32-bit tick counter. All the timer expirations are specified in terms of ticks. For example, 1 tick equates to 10 ms, but this is of course configurable.

Synchronization of threads and communication between tasks is another big topic in embedded systems. ThreadX provides several mechanisms for that and makes this task effortless. Semaphores and mutual exclusions (mutex) help to prevent priority inversion and

allow an unlimited number of objects to be created. Sophisticated callback functions and event-chaining are available as well, as is an optional priority inheritance. In addition, ThreadX can suspend in either FIFO or priority order, avoiding the problem of threads starving for processing time.

Event flags are another thread synchronization feature. An event flag group consists of 32-bits, where each bit represents a different logical event, and threads can wait on a subset of these bits. Event flags also support event-chaining. As with semaphores, an unlimited number of objects can be created and information on the run-time performance of the flags is available.

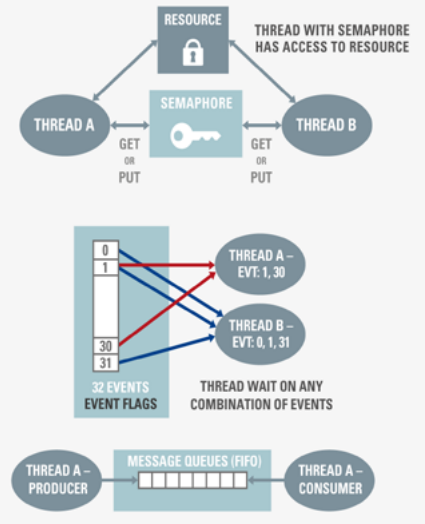


Figure 2-9: ThreadX® synchronization and communication features

The last synchronization feature to be mentioned here is message queues. Using the producer-consumer model for inter-task communication, ThreadX supports messages sizes from 1 to 16 four-bytes words, with mailboxes being a special case of a message queue with the size 1. Application threads can register a callback function for further notifications. As with the event flags, information about the run-time performance is available.

For debugging purposes, an essential feature is implemented: A built-in event trace capability supported by TraceX®, which allows to view the sequence of the different events. TraceX is a standalone Windows® application but programmers can launch it from inside e² studio.

And finally, ThreadX complies with all "required" and "mandatory" rules of Misra-C:2004 and Misra-C2012, is EAL4+ Common Criteria security certified, and is pre-certified to the following standards.

- IEC 61508 SIL 4
- IEC 62304 Class C
- UL/IEC 60730-1 H
- CSA E60730-1 H
- ISO 26262 ASIL D
- UL/IEC 60335-1 R
- UL 1998
- EN 50128 SW-SIL 4

Points to take away from this chapter:

- The Flexible Software Package (FSP) is built in layers and all functionality can be accessed through simple and standardized API-calls.
- The middleware provides rich connectivity and security, as well as graphics and capacitive touch functionality without the need to write code from scratch.
- FreeRTOS™ and ThreadX® are powerful and easy-to-use operating systems with small memory footprints, leaving most of the microcontroller's resources to the application.

3 AN INTRODUCTION TO THE APIs OF THE FLEXIBLE SOFTWARE PACKAGE

What you will learn in this chapter:

- The syntax, style and naming conventions of the FSP code.
- Details about the Application Programming Interfaces and how to use them.

Ease of use was in the mind of Renesas' designers when they developed the Flexible Software Package (FSP) for the RA Family of microcontrollers. While offering tremendous functionality, the FSP is quite simple to use, as the architecture of the Application Programming Interface (API) is very straightforward and comprehensive, encapsulating the complexity of the FSP, but still giving the programmer full control of the different features. Even programming a complex task like a USB-transfer is reduced to a couple of lines of code and can be achieved without having to read tomes of manuals or to study each and every particularity of the register set of a given microcontroller peripheral. This is a huge relief for developers, as they can concentrate on the feature set of their application instead of writing low-level code, which does not add any value to the design, but which takes a good amount of time to write and test. Let's have a look at some of the details of the API.

3.1 API Overview

Applications can access all the functions of the Flexible Software Package through intuitive, simple, and unified API calls, no matter in which of the layers the required functionality resides. This makes it very easy and straightforward to write code which is easy to understand, simple to maintain and painless to port, for example, if a different peripheral of the microcontroller has to be used for a certain task, something which is happening quite often during a design these days.

Staying with this example, changing from the on-chip SPI peripheral to the SPI functionality of the on-chip SCI port means that there is just one simple change to make if the FSP is used: in the FSP Configurator the *SPI (r_spi)* driver in the Stacks tab needs to be replaced by the *SPI (r_sci_spi)* driver. As both drivers use the same instance called *g_spi0*, there is no nothing to change in the application code.

The architecture of the different layers of the FSP is shown in *Figure 3-1*: At the bottom is the RA MCU, with the Board Support Package (BSP) sitting on top of it. This package is responsible for getting the MCU from reset to the main application and for providing other services to the software above. The next layer is the Hardware Abstraction Layer (HAL), which insulates the developer from having to deal directly with the register set of the microcontroller, and makes the software on top of the HAL more portable across the entire RA Family.

Above the HAL is the Middleware Stacks and Protocols layer and one of the Real-Time Operating Systems. The Middleware layer provides connectivity options, graphics processing functionality, and security features to the user application. Finally, at the very top resides the user program, making calls to the layers below through unified API interfaces. Details on the layers and the individual parts of the FSP are available in chapter 2.

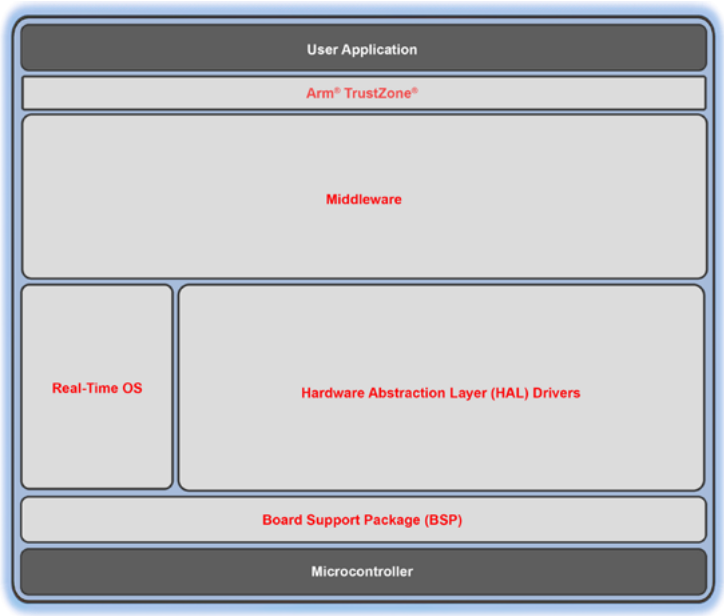


Figure 3-1: The different layers of the Flexible Software Package

Each of the different layers is accessed either through API calls directly, or in a stacked manner. An application needing a USB transfer, for example, would make a call to the USB HCDC driver, which makes use of the basic USB driver, which itself uses two instances of the DMAC (Direct Memory Access Controller) module: one for transmission, one for reception (see *Figure 3-2*).

Of course, an end application could also call the HAL drivers and the Board Support Package directly, but using the Middleware Stacks and Protocols layer is much easier, as no detailed knowledge of the underlying parts is necessary in this case.

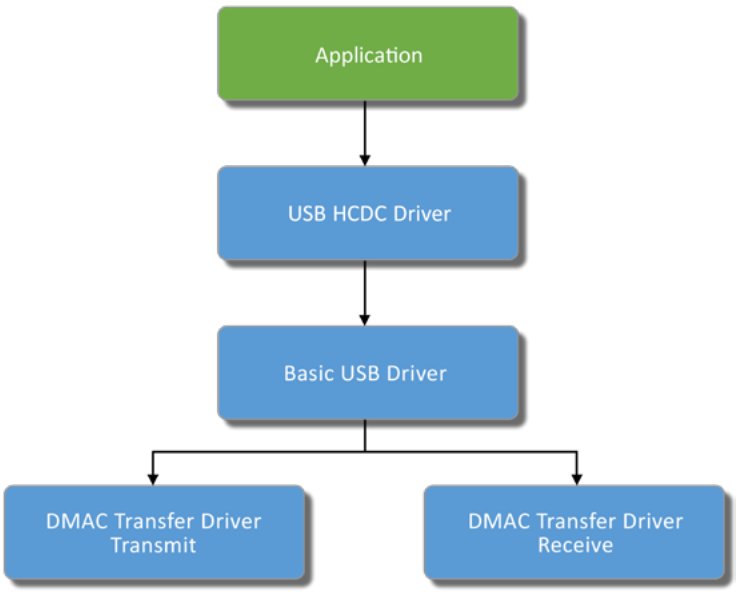


Figure 3-2: Modules can be stacked for ease of use

3.2 API Syntax

Before diving more deeply into the details of the API, we should have a short look at the naming conventions of the different APIs and files. Once understood, not only programming an application gets easier, but also understanding either the code from another programmer or one's own code after several months. The latter is an effort which we shouldn't underestimate, but always do. With all that, the clear structure provided by the Flexible Software Package (FSP) will help a lot.

In general, internal functions follow the "NounNounVerb" naming convention, e.g. communicationAbort(). Any data is returned in the output argument of the function and the first parameter is always the pointer to its control structure. As this structure holds the memory of the module instance, the user application is flexible to decide where to place it. *R_BSP_GET*

Following is a list of commonly used prefixes in the FSP:

R_BSP_xxx: The prefix for a common BSP function, e.g. `R_BSP_VersionGet()`.

BSP_xxx: The prefix for BSP macros, e.g. `BSP_IO_LEVEL_LOW`.

FSP_xxx: The prefix for common FSP defines, mostly error codes,
e.g. `FSP_ERR_INVALID_ARGUMENT` and version information, e.g. `FSP_VERSION_BUILD`.

g_<interface>_on_<instance>: The name of the constant global structure of an instance that ties the interface API functions to the functions provided by the module. One example would be `g_spi_on_spi` for a HAL layer function.

r_<interface>_api.h: The name of an interface module header file, e.g. `r_spi_api.h`.

R_<MODULE>_<Function>: The name of an FSP driver API, e.g. `R_SPI_WriteRead`.

RM_<MODULE>_<Function>: The name of a middleware function, e.g `RM_BLE_ABS_Open()`.

The operating systems also follow naming conventions. In FreeRTOS™, file scope static functions, for example, have the prefix *prv*. API functions adhere to the scheme *<return type><Filename><Operation>* where the return type follows the convention for variables, or in case a function is void, it is prefixed with the letter "v". The *File* field contains the name of the file in which the API function is defined, and the Operation field names the operation being executed in UpperCamelCase syntax. Examples are `vTaskStartScheduler(void)` which is a void function that is defined in *tasks.c* and starts the RTOS scheduler, or `uxQueueMessagesWaiting()`, a function which returns data of the non-standard type `unsigned BaseType_t`, is defined in the file *queue.c*, and returns the number of messages waiting in a queue.

Similar is valid for the macros: They are prefixed with the file in which they are defined in lowercase, followed by the name in all uppercase letters. Here an example would be the macro `taskWAITING_NOTIFICATION`, which is defined in the file *tasks.h*.

More details on the naming conventions of FreeRTOS can be found in the FreeRTOS Coding Standard, Testing and Style Guide on the Internet (**https://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html**).

The naming conventions of the Microsoft Azure RTOS are similar. They are built in the following way: *<ID>_<noun>_<verb>*. ID is the module, noun is the object in question (timer, semaphore, etc.) and the verb is the action to be performed (create, close, receive, ...). For example `tx_queue_create()`, which creates a message queue in ThreadX®.

Below is a summary of several of the IDs for the various Azure® RTOS modules:

fx: FileX® related functions, e.g. `fx_directory_create()`.

gx: GUIX™ related functions, e.g. `gx_display_create()`.

lx: LevelX™ related functions, e.g. `lx_nand_flash_open()`.

nxd: NetX DuoTM related functions, e.g. `nxd_ipv6_enable()`.

tx: ThreadX® related functions, e.g. `tx_thread_create()`.

ux: USBXTM related functions, e.g. `ux_device_stack_initialize()`.

Besides understanding the formal syntax of the API, it is also very helpful to agree on a couple of definitions. All too often, every one of us has a slightly different comprehension of some terms, causing confusion. So, here is the list of terms used throughout the book.

- **Modules:** Modules can be peripheral drivers, pure software, or anything in between, and are the building blocks of the FSP. Modules are normally independent units, but they may depend on other modules. Applications can be built by combining multiple modules to provide the user with the functionality they need.
- **Module Instance:** Single and independent instantiation (configuration) of a module. For example, a USB port might need to transfer data to and from the port, using two instances of the *r_dmac* module.
- **Interfaces:** An interface contains API definitions that can be shared by modules with similar features. They are the way modules provide common features. Through them, modules that adhere to the same interface can be used interchangeably. Think of an interface as a contract between two modules, where the modules agree to work together using the information agreed upon in the contract. Interfaces are definitions only and do not add to code size.
- **Instances:** While interfaces dictate the features provided, instances actually implement them. Each instance is tied to a specific interface, using the enumerations, data structures and API prototypes from the interface. This allows an application to swap out instances when needed.
- **Drivers:** A driver is a specific kind of module that directly modifies registers on the RA Family MCUs.
- **Stacks:** The FSP architecture is designed in a way that modules can work together to form a stack. A stack consists of a top-level module and all its dependencies.
- **Application:** Code that is owned and maintained by the user, even if example code provided by Renesas is used.
- **Callback Functions:** They are called when an event occurs, for example, when the USB has received some data. They are part of the application, and, if used for interrupts, should be kept as short as possible, as they will run inside the interrupt service routine, preventing other interrupts from being executed.

3.3 API Constants, Variables, and Other Topics

As mentioned, modules are the building blocks of the Flexible Software Package. They can perform different tasks, but all modules share the basic concept of providing functionality upwards and requiring functionality from below, as shown in *Figure 3-3*. So, the simplest possible application using the FSP consists of one module with the user code on top.

Modules can be layered on top of one another, building an FSP stack as shown in *Figure 3-2*. The stacking process is performed by matching what one module provides with what another module requires. The Block Media implementation, for example, requires an SD/MMC driver, which in turn needs a data transfer interface, which can be provided by the Data Transfer Controller (DTC) driver module. Its code is not included in the Block Media module by intent, allowing the (underlying) DTC module to be reused by other modules as well, for example by an UART.

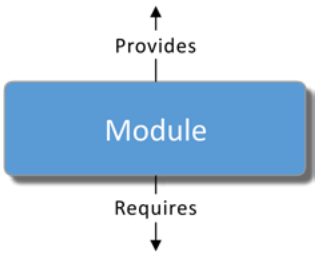


Figure 3-3: Modules are built in a way that they provide functionalityto the caller and require functionality from the level below

The ability to stack modules offers a great benefit, as it ensures the flexibility needed by the application. If they were directly dependent on others, portability across different user designs would be difficult. The FSP architecture, as it was designed, provides the possibility of swapping modules in and out (e.g. exchange the UART driver against the SPI driver) through the use of the FSP interfaces, as modules adhering to the same interface can be used interchangeably. Remember the contract mentioned above? All the modules agree to work together using the information that was agreed upon in the contract.

On the MCUs of the RA Family, there is occasionally an overlap of features between different peripherals. For example, IIC communications can be achieved using the IIC peripheral or the SCI peripheral in its IIC mode, with both peripherals providing a different set of features. The interfaces are built in a way that they provide the common features most users would expect, omitting some of the more advanced features, which however, in most cases, are still available through interface extensions.

At least three data structures are included in each FSP interface: First, a module dependent control structure named `<interface>_ctrl_t`, which is used as a unique identifier for using the module. Second, a configuration structure named `<interface>_cfg_t*` as input to the module for initial configuration during the `open()` call, and, third, an instance structure, consisting of three pointers: a pointer to the control structure, a pointer to the configuration structure and a pointer to the API structure. The name of this latter structure is standardized as `g_<interface>_on_<instance>`, e.g. `g_spi_on_spi` and the structure itself is available to be used through an `extern` declaration in the instance's header file, e.g. *r_spi.h*. While these structures could have been combined in the case of a simple driver, creating them this way expands the functionality of the interfaces.

All interfaces include an API structure containing function pointers for all the supported functions. For example, the structure `dac_api_t` for the digital-to-analog converter (DAC) contains pointers to functions such as *open*, *close*, *write*, *start*, *stop* and *versionGet*. The API structure is what allows modules to be swapped in and out for other modules that are instances of the same interface.

Modules alert the user application once an event has occurred through callback functions. An example of such an event could be a byte being received over a UART channel. Callbacks are also required to allow the user application to react to interrupts. They need to be as short as possible, as they are called from inside the interrupt service routine. Otherwise, they might have a negative impact on the real-time performance of the system.

Whilst interfaces dictate the features that are provided, the instances actually implement them. Each instance is tied to a specific interface and uses the enumerations, data structures, and API prototypes from the interface. This allows for applications that use an interface to swap out the instance when needed, saving a lot of time once changes to the code or the used peripherals are needed. On the RA Family MCUs, some peripherals, for example the IIC, will have a one-to-one mapping (it only maps to the IIC interface), while others, like the SCI, will have a one-to-many mapping (it implements three interfaces: IIC, UART, SPI). Refer to *Figure 3-4* for a graphical representation of this mapping.

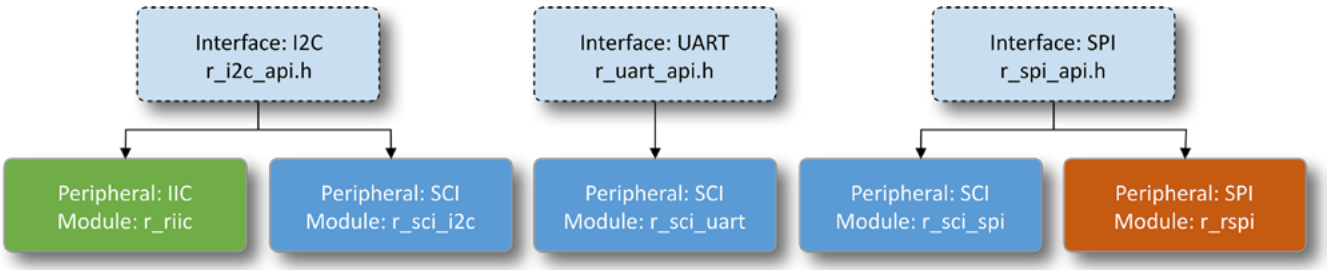


Figure 3-4: On the microcontrollers of the RA Family some peripherals will have a one-to-one mapping between the interface and instance, while others will have a one-to-may mapping

3.4 API Usages

After all the theory, it is now time to finally look at how easy it is to work with the Flexible Software Package (FSP). For this, we will use a simple SPI as example and explain how to use the APIs and where to find information about the different items. If you want to follow this on your PC, you can come back to this section after chapter 5.

The first step in using an FSP module is always to select the right interface for the functionality that is required. In our case, we want to communicate over the SPI interface block of the Serial Communications Interface (SCI), which is available on all series of the RA Family of MCUs. So, inside e^2 studio we choose the *r_sci_spi* driver for the SPI in the FSP Configurator. This is done by first highlighting the *HAL/Common* thread on the left-hand side of the Stacks tab of the configurator and then by selecting *New Stack → Connectivity → SPI (r_sci_spi)* on the right-hand side under *HAL/Common Stacks*. Once added, additional required entries will be added automatically down to a level where the user needs to make a choice or decision on the functionality. In the case of the SPI, an instance of each the *g_transfer0 Transfe*r and the *g_transfer1 Transfer* drivers of the *r_dtc* (data transfer controller) module will be added. The first one handles the data transfer to the SI port and is connected by default to the TXI interrupt of SCI0, while the latter one is associated with the RXI interrupt and retrieves the data from the serial port.

Nothing else needs to be done. In the case of our example above, the receive, transmit, and error interrupts are automatically enabled by the FSP Configurator. If wanted, the module instance, which is called `g_spi0` by default (the number depends on the order in which drivers of the same type are added), can be given a username if desired (e.g. `my_spi`). This adds another layer of abstraction, which, if the name is properly chosen by the software designer, can make the code more portable and better maintainable, not to speak of being more readable as well.

The *Properties* view of the *g_spi0 SPI (r_sci_spi)* module would be the place to configure the parameters, like bit rate, bit order, etc. of the SPI port. With all the configuration done already in the graphical user interface, there is no need to initialize the port with the correct values manually. This is all done by the FSP Configurator for you. Renesas made a huge effort to create useful software and hardware development tools that can be utilized to explore the technical capabilities of the RA Family and that will take the user beyond the evaluation stage all the way to production.

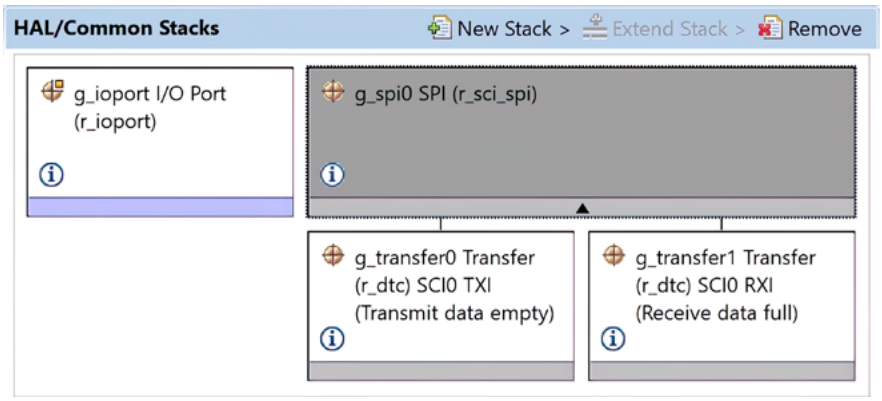


Figure 3-5: The SPI stack in e² studio

Right now, there is no need to care about the other item in the *HAL/Common Stacks* window: this is the basic driver needed by the Board Support Package for the I/O-ports. It is added automatically by the configurator and initialized at startup.

All that is left is to check if the I/O-pins for the SCI-SPI have been properly configured on the *Pins* tab of the FSP Configurator to actually transmit and receive data through this peripheral. Expanding the Peripherals list on the left side of the view will show the interfaces available. Further expanding the *Connectivity:SCI* tree allows the selection of the SCI0 entry and the *Pin Configuration* for this port will display. Selecting *Simple SPI* under Operation Mode enables the *TXD_MOSI*, *RXD_MISO*, *SCK*, and *CTS_RTS_SS* drop-down list box, where the pins used can be matched to the actual hardware.

Saving the changes *using <ctrl>+<s>* and clicking on the *Generate Project Content* icon will create the necessary source and header files and will populate the control and configuration structures, named `p_ctrl` and `p_cfg`, being of the type `spi_ctrl_t*` and `spi_cfg_t*` respectively, and places them in the application-specific header files to be included in the user code, from where they can be accessed using the pointers provided by the *g_spi0* API interface.

With the setup completed and FSP files created, interaction with the interfaces gets easy. In case of the SPI example, the SPI peripheral would be opened and configured by calling the following function:

```
err = g_spi0.p_api->open(g_spi0.p_ctrl, g_spi0.p_cfg);
```

with `g_spi0` being the name given the instance when configuring it in the Properties view and `p_api, p_ctrl` and `p_cfg` being the pointers to the structures generated by the configurator. Similarly, writing to the SPI would require the function call below:

```
err = g_spi0.p_api->write(g_spi0.p_ctrl, tx_buffer, length, bit_width);
```

Here, `tx_buffer` is the pointer to a user-defined array holding the data to be sent over the SPI, `length` the number of units to be written as a 32-bit unsigned integer, and `bit_width`, which has the type `spi_bit_width_t` and holds the size of the units. The exact syntax can be extracted from the interface reference chapter in the RA Flexible Software Package Documentation or by using the smart documentation feature of e² studio.

Instead of using the interface functions as in the example above, the direct implementation functions `R_<MODULE>_<Function>` can also be used. In that case, the code for opening the SPI peripheral and configuring it would look like that:

```
err = R_SPI_OPEN(&g_spi_ctrl, &g_spi0_cfg);
```

with `g_spi0_ctrl` and `g_spi0_cfg` being the data structures created by the FSP configurator. Analogously, a write to the SPI port would require this function call:

```
err = R_SPI_WRITE(&g_spi0_ctrl, tx_buffer, length, bit_width);
```

with `tx_buffer` being again the pointer to the array holding the data to be written over the port, `length` being the number of units and `bit_width` holding the size of the units. Instead of `bit_width`, the macro `SPI_BIT_WIDTH_8_BITS` provided by the FSP could also be used.

These two short examples are an excellent demonstration of the capabilities of the Flexible Software Package. Only two lines of code need to be written in each one of the cases to configure the port and to send an array of data through it. Everything else has already been created and performed by the configurator and the drivers of the FSP. No need to read through the manual of the microcontroller and learn about the different registers involved, and I believe that this saves a lot of time during the development of an application.

Points to take away from this chapter:

- The FSP follows a clear naming convention, making coding simple.
- FSP Modules can easily be stacked to provide a rich functionality to the application.
- The syntax of the API is straightforward and intuitive.
- Instead of using the interface functions, the implementation functions could be used as well.

4 GETTING THE TOOLCHAIN UP AND RUNNING

What you will learn in this chapter:

- Where to get and how to install the toolchain for the RA Family.
- What the necessary steps are to start e² studio for the first time and to create a new project.

In this chapter we will download and install the two tools required for software projects for the Renesas RA Family of microcontrollers: the e² studio Integrated Development Environment (IDE) with the GCC Arm® Embedded toolchain, and the Flexible Software Package (FSP).

The following paragraphs outline all the steps necessary for that. You will find that the installation of the tools is free of any hassle as the installer takes care of all things needed, once you made your selections on what and where to install. Finally, after the installation is complete, we will perform a short sanity check to ensure that everything is working as expected.

4.1 Downloading and Installing e² studio and the FSP

e² studio contains all the tools necessary to create, compile and debug projects for the microcontrollers of the RA Family. It is based on the popular Eclipse™ IDE, but several solution-oriented components and plug-ins have been added by Renesas, making it even more powerful. This is especially true for the configurators, which provide a simple way to generate new projects and which allow an easy graphical access to the different hardware and software features like the pin configuration or the addition of software stacks without the need for deep study of the User's Manual. These configurators will create all the settings necessary and the initialization code automatically, and they include an error-checking element to detect problematic combinations already at design time, saving tons of hours otherwise spent for writing and/or debugging code not adding value to the application itself. At least, I would have been more than happy, if such tools had been available when I first started writing software for embedded systems.

The next paragraphs will walk you through the different phases of downloading and installing e² studio and the FSP on your Windows® workstation. They are intended for users new to the FSP/e² studio toolchain. If you have already installed the current releases of e² studio and the GCC Arm® Embedded tools, you can shorten the process by just downloading the FSP packs separately. They are accessible under the Assets section on the FSP's GitHub page (**https://github.com/renesas/fsp/releases**). Two versions are available: a zip-file named *FSP_Packs_<version>.zip* and an installer called *FSP_Packs_<version>.exe*.

If you plan to use the IAR Embedded Workbench® for Arm or the Keil® MDK Microcontroller Development Kit, you also can take advantage of the configurators for the FSP: the RA Smart Configurator (RASC) integrates with these toolchains and provides the same functionality. It is available in the *Assets* section on GitHub as well.

4.1.1 Download of the Installer

The first step is to locate the latest release of the *FSP with e² studio Installer* in the Assets section on the FSP page on GitHub® (**https://github.com/renesas/fsp/releases**). There you can find the executable of the installer named *setup_fsp_v<fsp_version>_e2s_v<e2 studio version>.exe*, e.g. *setup_fsp_v3_7_0_e2s_v2022-04.exe*. Download this file to your hard drive. While the download is in progress, and it is a fairly large file with a size of more than 1.3 GB, take some time to review the other items on the page. From there, you can also load the latest documentation in HTML-format for the FSP as zip-file or read the release notes in the *README.md* file.

4.1.2 Installing the Toolchain

Once the download of the *FSP with e² studio Installer* is complete, locate the file on your hard drive and double-click on it to start the setup process. The extraction of the installation files will take some time. The first screen appearing queries if you want to install the development tools for all users or just the current user. Select the installation type which suites you best. Next, a Windows® User Account Control dialog box shows, asking "Do you want to allow this app to make changes to your device?" You can safely answer *Yes* to this question.

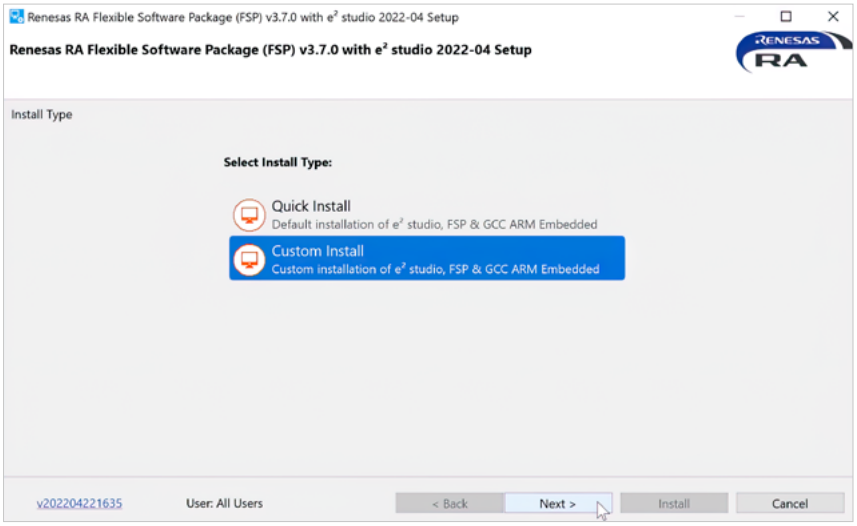


Figure 4-1: The custom installation is the way to go, as it allows to install additional tools and features

On the next screen the installer will ask you for the install type (see *Figure 4-1*). There are two options: Quick Install, which simply installs e² studio, the Flexible Software Package and the GCC Arm® Embedded code generation tools, as well as the User's Manual with their default settings and runs with minimal user interaction. Or Custom Install, which allows not only to install the IDE and the Flexible Software Package, but also optional components like debug support for FreeRTOS™, the integration of the GIT version control system, or the QE Tools for capacitive touch and Bluetooth® low energy. The quick installation is the way to go, if you want to get going as fast as possible. But, if you want to install any additional features or if you are just curious what else is available, the custom installation is the right choice. No matter how you decide now, things can be changed any time later by re-running the installer.

For the purpose of this walk-through, select *Custom Install* and click on *Next*. The installation routine will now check that the install directory is ready and that all components required are available. Here you can also change the folder where the software will be installed. The default path is *C:\Renesas\RA\ e2studio_v<e2studio version>_fsp_v<fsp_version>*, e.g. *C:\Renesas\RA\e2studio_v2022-04_fsp_v3.7.0*. If necessary, change it to a path of your choice. Then dismiss the *Results* screen by clicking on *Next*.

Once the *Extra Features* dialog shows, browse through the list displayed and select the features you want to add to the installation. Make your choices and click on *Next* to move on.

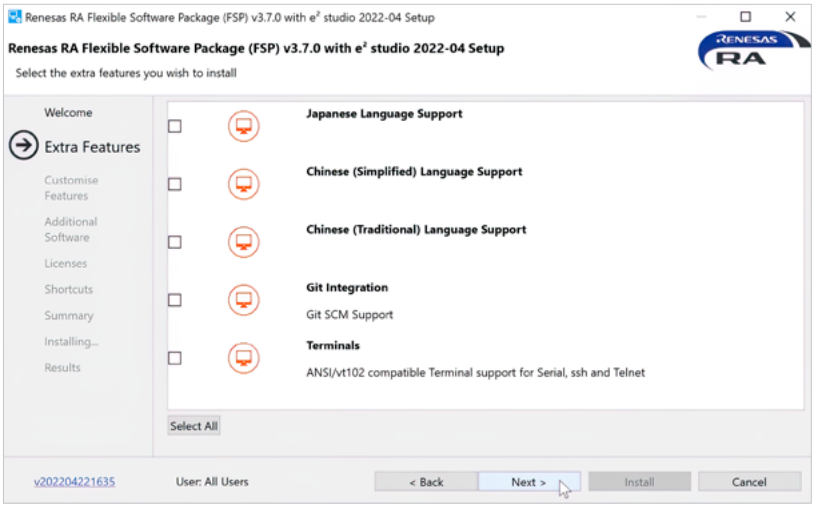


Figure 4-2: This screen allows to install additional features

Another screen will show, named *Customize Features* (see *Figure 4-3*). Here you can add additional components to your installation, like Git integration for Eclipse or Terminal support for serial, ssh, and Telnet. Again, make your choice and click on *Next*.

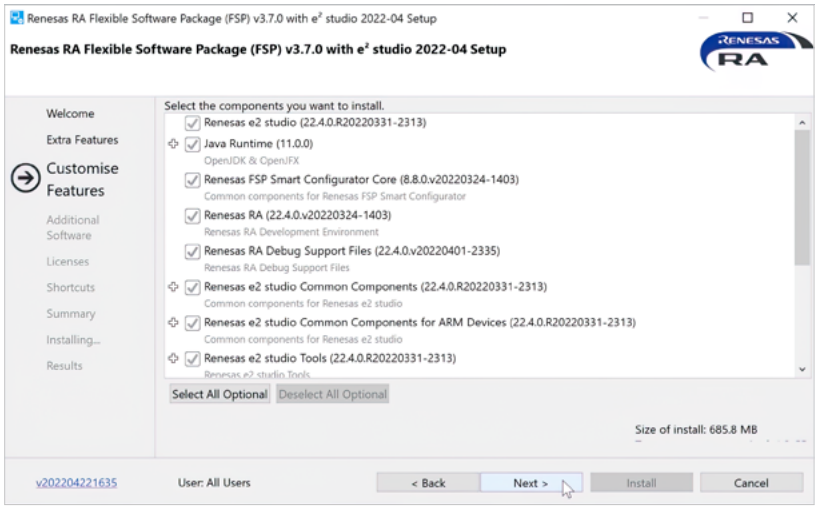


Figure 4-3: This window allows to install further components

The following screen, named *Additional Software*, presents you two tabs (see *Figure 4-4*): one called *Renesas RA* and one called *Renesas QE*. On the *Renesas RA* tab, make sure that *GNU ARM Embedded* is selected under *Toolchains*, as this one is required to compile and link your code. On the second tab, choose the QE Tools of your choice for installation. Refer to Chapter 1, if you need more information about these. As soon as you are ready, click on *Next*.

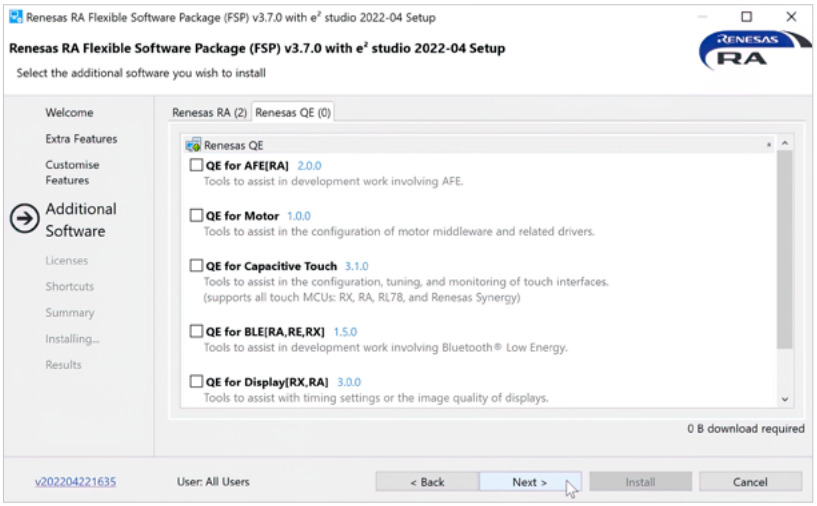


Figure 4-4: Select the QE Tools of your choice for installation

The window showing now will present you the various software agreements for the components to be installed. Read each one carefully and then check the box besides "I accept the terms of the Software Agreements" and click again on *Next*. On the *Shortcuts* screen appearing, either accept the proposed Start menu group, change it to something being more meaningful to you, or clear the check box. After moving on to the next window by clicking on *Next* the installer will present you a summary of the software it is going to install. Browse through it and start the installation by clicking on *Install*.

During the installation process, a Windows security dialog will show, asking if you would "like to install this device software" (see Figure *4-5*). Check the box besides *Always trust software from "Renesas Electronics Corporation"* and click on *Install*.

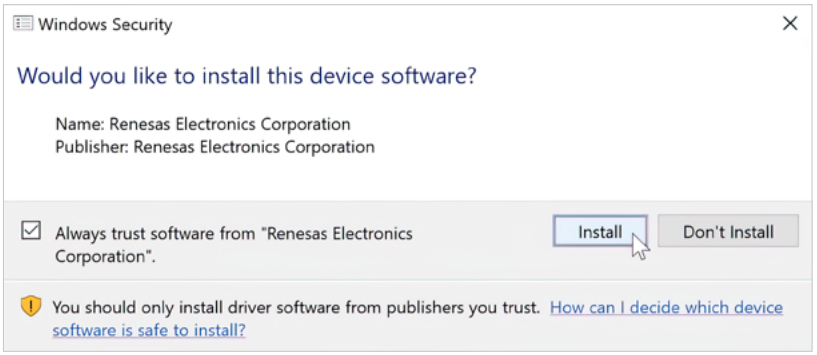


Figure 4-5: Allow the installation of the device software. It is needed for debugging.

Once the installation is complete, the Results window will show. Clear all check boxes and have a look at the links providing more information on the FSP and the GCC Arm Embedded, as well as a link to the FSP User's Manual. Finally, click on *OK* to close the installer. This concludes the installation and you are now ready for a first test of the development environment.

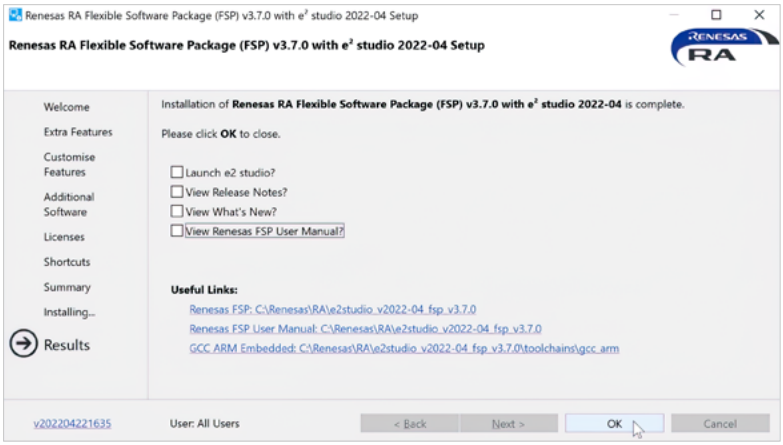


Figure 4-6: Once the installation is complete, you will see this window. Clear all check boxes and click on OK to close the installer.

4.2 Starting for the First Time

Open e² studio from the Start menu of your Windows® workstation. You will find it in the Renesas RA <version number> menu group, or, if you had changed the shortcut during the installation, in the group you entered there. If you can't find it at all in the Start menu, have a look in the installation directory, which is located at C:\Renesas\RA\e2studio_v<e2studio version number>_fsp_v<fsp version number>\eclipse.

e² studio will ask you for a folder for the workspace. You can either accept the default or choose your own. Click on *Launch*. As this is the first time you start the Integrated Development Environment (IDE), it will extract and refresh additional support files, so the start might take a little longer. A screen will show, asking you for your "My Renesas" login credentials. Provide them and decide if you want to consent to record and send usage data or not. If you change your mind later on, you can reverse your decision in the e² studio preferences. This screen lets you also sign up for "My Renesas" if you haven't done so yet. Click on *Login* and provide the answers for the password recovery setup. Click on *OK* and dismiss the *Welcome* screen by clicking on the *Hide* icon at the upper right-hand corner of the screen.

Next step is to create a first simple project to ensure that everything is working. For that, go to *File* → *New* → Renesas *C/C++ Project* → *Renesas RA* on the menu bar, which will bring up a new window called *Templates for Renesas RA Project*. Select *Renesas RA C/C++ Project* (see *Figure 4-7*). Click on *Next*. This will start the Project Configurator. Enter a name for your project, for example "*MyRaProject*" and click on *Next.*

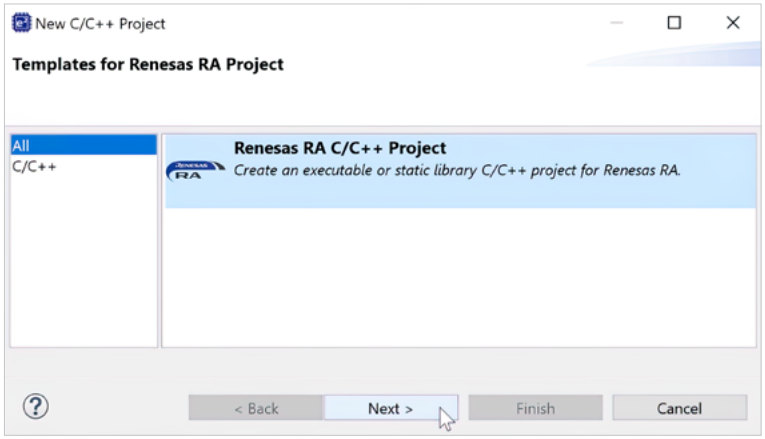


Figure 4-7: Select the Renesas RA C/C++ Project entry to call the Project Configurator for the RA Family

This will get you to the *Device and Tools Selection* screen. First, choose a board. As we just want to test the development environment, select the *EK-RA6M4* and set the corresponding device to *R7FA6M4AF3CFB*, if not already listed. Have a look at the toolchain: it should read *GNU Arm® Embedded*. Click on *Next*.

If you have installed the GNU tools but they are not shown here, you need to integrate them manually. For this, go to the Help menu of e² studio and select the entry *Add Renesas Toolchains*. Please note that the Scan option only works for toolchains having been installed at their default location. If you haven't installed the GNU tools at their default location, either enter the path to your custom folder in the entry field or browse to the location using the *Add* command and the file browser.

The screen showing now lets you choose between non-TrustZone® and secure and non-secure TrustZone projects. Keep the *Flat (Non-TrustZone) Project* selected (see *Figure 4-8*) and click on *Next*. The *Build Artifact and RTOS Selection* window appears. Leave the settings as they are, i.e. *Executable selected* under *Build Artifact Selection* and *No RTOS* under *RTOS Selection*. Don't worry about the RTOS selection: FreeRTOS™ and Azure® RTOS ThreadX® can be activated later on in the RA Configurator, if needed.

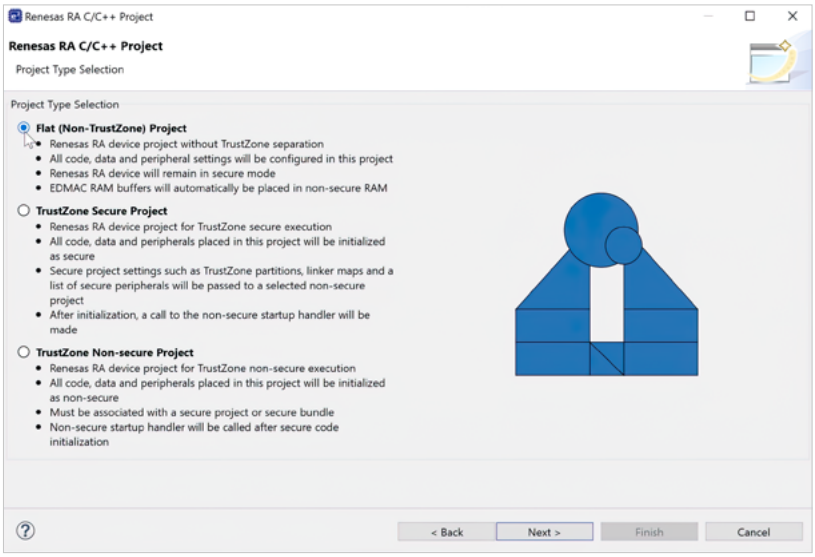


Figure 4-8: Right now, just leave the already selected Flat Project activated

Move on to the next screen named *Project Template Selection* by clicking on *Next*. For our test, it is sufficient to use an example package, so select *Bare Metal – Blinky*. Click on *Finish*. Now the configurator has enough information to create a project template which includes the correct device and board dependent Board Support Library and the example project.

The configurator then extracts all necessary files and will ask you if you want to open the FSP Configuration perspective. Answer *Open Perspective* and the perspective will show. In the configurator, you could configure the FSP and most of the on-chip hardware of the processor. For the template we used for this example, everything has already been set, so click on the small green arrow saying *Generate Project Content* at the upper right-hand corner of the window and the configurator will create all necessary configuration files and add them to the project.

Finally, click on the small "Hammer" symbol near the left-hand corner of the toolbar on top of the IDE and the project will compile and link. Once it has finished, it should show a status of *0 errors, 0 warnings* after completion. Now everything is set up and working and you are ready to start your own project.

And, if something went unexpectedly wrong during the installation, you know help is always available, even right out of e² studio: Go to *Help → Renesas Help → Renesas Helpdesk* and you will be directed to the *Design & Support* page on Renesas' homepage. Another possibility is to click on the Support icon on the Summary tab of the FSP Configuration perspective. This will open the *Knowledge Base* page inside your web browser.

4.3 Keeping your Installation Up to Date

Having installed the tools, you might want to keep them in good shape, which means you want to keep them up to date. Or you are so excited about all the possibilities the Eclipse™ based development environment offers, that you want to install additional software. Keeping the installation current is achieved by loading the latest documentation and installations from the GitHub® repository under **https://github.com/renesas/fsp/releases**. Using the update feature from inside e² studio (*Help → Check for updates*) is not recommended for that purpose, as the updates there may not add anything beneficial for the RA Family of microcontrollers and the search takes quite a while.

If you want to install additional software either run the installer again or go to *Help → Install New Software* inside e² studio. On the following screen, choose the -- *All Available Sites* – entry from the dropdown list under *Work with* at the top of the window to search all sites and you a list of software available for download will be presented. Yet another possibility is to go to the RA Family page on Renesas' website (**https://www.renesas.com/ra**) where you can find additional software and solutions from the Renesas RA Partner Ecosystem.

Now that we know that the installation is working, the next steps will be to check out a development kit and to write your very own code.

Points to take away from this chapter:

- GitHub® is the place to download the latest revision of the toolchain.
- Installation of the toolchain is straightforward if the FSP with e² studio installer is used.
- Configurators will guide you through the process of creating projects and configurations.

5 WORKING WITH e² studio

What you will learn in this chapter:

- The difference between Perspectives, Views, and Editors.
- What the different Configurators are for and how to use them.
- How to import and export projects.

Now that we know the details of the Renesas Flexible Software Package (FSP) and how to use its Application Programming Interfaces, it is time to look into the development environment for the RA Family of microcontrollers, e² studio.

Developed and maintained by Renesas, e² studio is based on Eclipse™, a popular and widespread open-source Integrated Development Environment (IDE) for different programming languages and target platforms. Eclipse can be easily customized and extended, and is therefore the IDE of choice for thousands of developers worldwide and a de facto standard.

e² studio leverages all the benefits of Eclipse and includes additional views and configurator perspectives to support all the features of the RA Family. It contains every tool necessary to create, compile and debug projects of any size and complexity, and guides the developer through the three phases of a software design: preparation, build and debug. And, it will get updated regularly to use the latest Eclipse SDK and CDT tools.

The following parts of this chapter will talk about the details of the Eclipse workbench and how to use the different elements of it. While we will cover some ground, not everything will be explained in here. More information can be found in the Workbench User Guide by going to *Help → Help Contents* inside e² studio.

5.1 Short Introduction to the Philosophy of Eclipse

The main window of e² studio, called the workbench, is created of a few basic user interface elements, like perspectives, views, editors, menu- and toolbars.

Once e² studio is started, it opens the perspective(s) last used. A perspective is a set of views, editors and toolbars, together with their arrangement inside the workbench. If you re-arrange windows, toolbars or views, these changes will be saved in the current perspective and are available once you open it again the next time.

In e² studio, there are several pre-defined perspectives available, and you can have multiple of them visible simultaneously – like the C/C++ perspective and the Resource perspective. Changing from one perspective to another can be done by clicking on the squared icon besides the perspective list at the very right side of the toolbar or by selecting *Window → Perspective → Open Perspective → Other …* from the main menu bar. Both ways bring up a pop-up window itemizing all the available perspectives. You can close a perspective using the same menu entry or by right-clicking on the perspective in the perspective list. If you right-click on the name of an open perspective, you can also detach it and move it around freely inside the workbench.

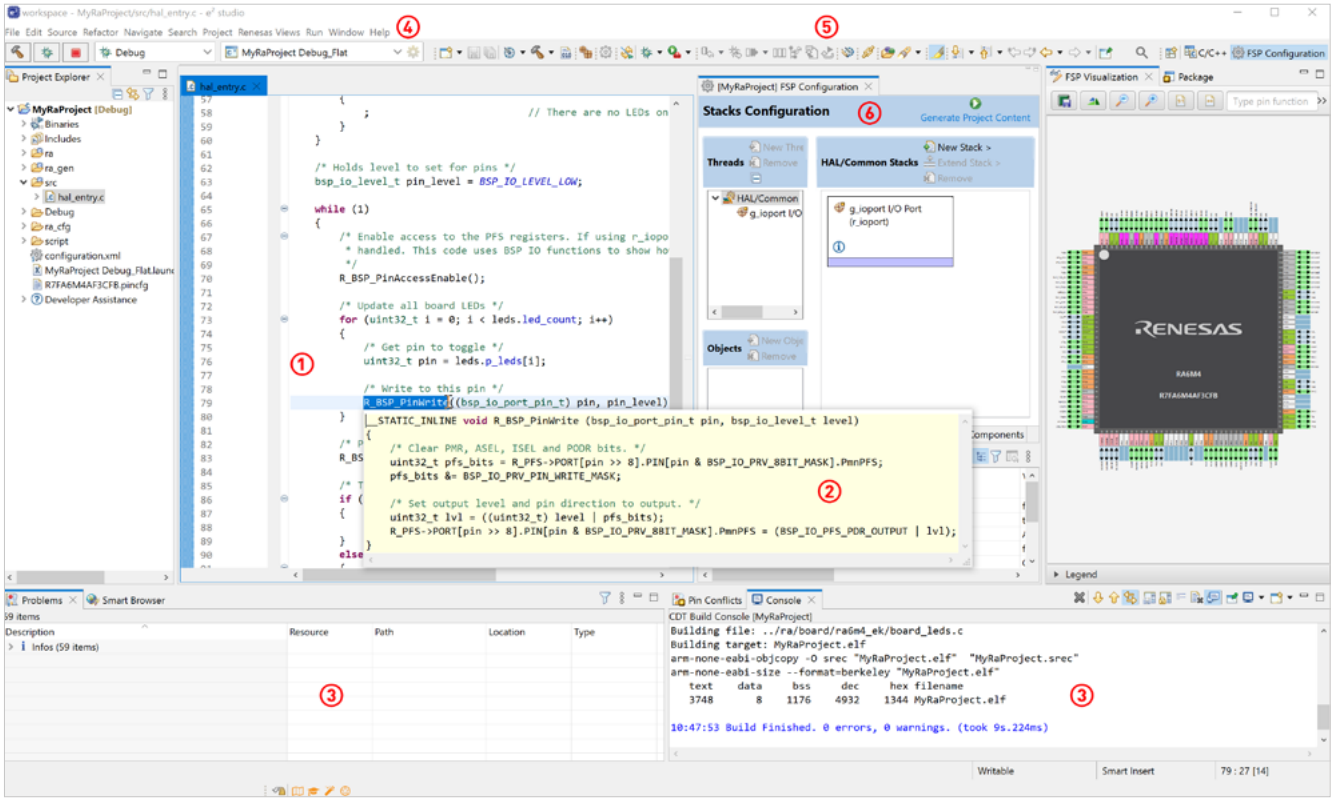


Figure 5-1: The workbench of e² studio consists of editors (1), Smart Manuals (2), views (3), menu bar (4), toolbars (5) and perspectives (6)

5.1.1 Perspectives in e² studio

- **C/C++:** This is the standard perspective of e² studio and is used for developing programs and editing source code. By default, it opens the editor in the middle, the Project Explorer view to the left, where you can manage your projects, the Outline view, showing all variables and definitions of the source file active in the editor, and at the bottom a tabbed notebook with a couple of stacked views like Problems (e.g. any syntax- or compilation errors), Tasks (lists Doxygen tasks) or Properties.
- **Debug:** This perspective is used for running programs and to diagnose and debug problems that occur at runtime. e² studio opens the following views by default: The Debug view (yes, the Debug perspective includes a Debug view), with the Debug Tool Bar, editor window(s), the Outline view (as with the C/C++ perspective) to the right and some stacked views for analysis and visualization at the bottom and a tabbed notebook on the top for watching variables, breakpoints, registers, and others.
- **Resource:** By default, this perspective includes most of the views from the C/C++ perspective, besides that the window at the bottom is showing only the Tasks view. It provides general resource viewing and navigation.
- **FSP Configuration:** This is one of the perspectives, where you will spend a lot of time in at the beginning. The largest part of the workbench is occupied by the FSP (Flexible Software Package) Configuration perspective (a perspective can include other perspectives as well), with several tabbed views for the miscellaneous FSP Configurators, like the Clock Configurator or the Pin Configurator. It also includes the Properties view, where the settings for the different peripherals, threads and drivers can be made, and the Package view, where you see a visual outline of the package chosen for the microcontroller showing the current pin assignments together with a status showing whether a configuration problem has been detected or not. If needed, the FSP Configurator can be accessed from the C/C++ perspective and the Debug perspective as well.
- **Team Synchronization:** In this perspective, you can merge changes you had made with those of other members of your team.
- **Tracing:** This perspective opens the Statistics and Histogram views, as well as the Debug Call Flow view.

If you make changes to your perspective and you do not like them, you can always reset it back to the default by right-clicking on the perspective's name in the main tool bar and selecting *Reset*. This also helps to clean up the mess you created during an intensive coding or debugging session, with lots of views and editors open and undocked! No screen is big enough to keep them nicely arranged, so once finished, you might end up with a very cluttered workbench.

5.1.2 Views

A view is a window where you can examine something, like a set of registers, properties or a list of files. A view can also have alternative representations, like a real-time chart of running threads, or gauges and controllers to be connected to variables. Views can have their own menu- or toolbar. Actions triggered by a view's menu- or toolbar will only affect items within that view, but not in other views, even if they reside in the same perspective.

Multiple views can be stacked together in the same window, which is then called tabbed notebook. You can also move views from one notebook to another or undock them totally. It's one of the nice features of e² studio that you can arrange everything to match your workflow and that the last layout of your perspective will automatically be saved for you.

Besides the numerous views e² studio offers already, Renesas created several additional ones like the Memory Usage view, the Fault Status view, or the RTOS Resources view, making the Integrated Development Environment even more versatile. All of them can be found in the main menu bar under *Renesas Views*.

5.1.3 Editors

This is the view (an editor is considered a special type of view) every software developer will spend most of her or his time with. Besides the usual and expected features like code completion or keyword highlighting, the editor built into e² studio includes a special feature, called Smart Manual. It eliminates the need to study thousands of pages of documentation, as you get context-aware help on the FSP and on the microcontroller itself directly inside the Editor view.

Hovering with the mouse over any keyword, function, variable, or macro within e² studio will bring up a window displaying relevant information. For variables, the declaration will be presented, for structures and enumerations all the members, and for functions, the description, prototype, and parameter details. If a variable is associated with a register of the microcontroller, specific information about the bit definitions will be shown as well. The Smart Manual even pulls in applicable application notes and media-rich instructional material if available and displays them in the Smart Browser view at the bottom of the C/C++ perspective.

This feature is quite cool and saves a lot of time, as you will not have to switch back and forth between the FSP and RA microcontroller manuals and e² studio. You get the relevant information where you need it most: right at the point of your mouse-cursor.

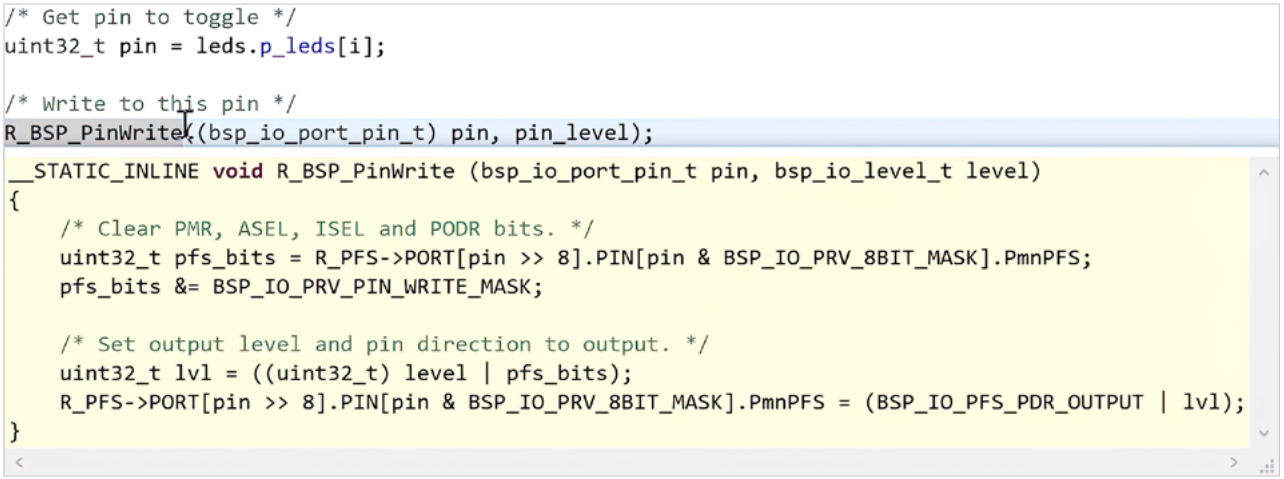


Figure 5-2: The Smart Manual feature of e² studio will display the information right there where it is needed

Code completion is another one of those features of the IDE saving a lot of time during code development. Pressing *<ctrl>-<space>* after a variable will bring up a window with the available options, like the members of an API-structure. Clicking on one of the members will insert them into your code. Again, no need to dig into the manual, just point and click!

5.2 The Configurators: A Short Introduction

The configurators inside e² studio graphically guide the software developer through specific device options of the RA Family microcontrollers (MCUs) and advises on their use. Examples of that are pin choices or warnings of conflicts. The configurators also generate start-up code or place FSP software components within the project.

5.2.1 Project Configurator

There are several configurators available to help the designer and the first one most users will experience is the Project Configurator, which leads them through the process of creating a new project from scratch or from a template provided by the configurator. It allows to choose most settings for the project, like the toolchain to be used, which device and board to use, or if an example project should be produced.

At the end of this process, the project will be automatically generated and all necessary files added to it in the workbench of e² studio. There is no need to create anything manually; no looking up of make-file parameters and settings, no research which header files to include, and how to advise the compiler to use a specific series and device out of the large RA Family of microcontrollers. All is done for you once you click on *Finish* on the last configurator screen.

With code generation complete, the Project Configurator will switch to the FSP Configuration perspective, where the settings for the different FSP components can be made.

5.2.2 FSP Configurator

Once the FSP Configurator shows in its perspective, it displays a read-only summary of the project, stating the selected board and device, and the toolchain used. It also lists the various software components in the project together with their version numbers. At the bottom of the page, shortcuts provide quick access to the Renesas FSP playlist on YouTube™ containing several short videos about the RA Family, to the Renesas FSP Support page on Renesas.com, and to the User's Manual of the Flexible Software Package (see *Figure 5-3*).

The next tab at the bottom of the configurator is the BSP (Board Support Package) tab, allowing to view and edit aspects of the board setup, like device or board selection. In the associated Properties view, additional settings for the BSP can be made, for example the size of the main stack (the stack used outside of a thread context) or for some security features of the MCU.

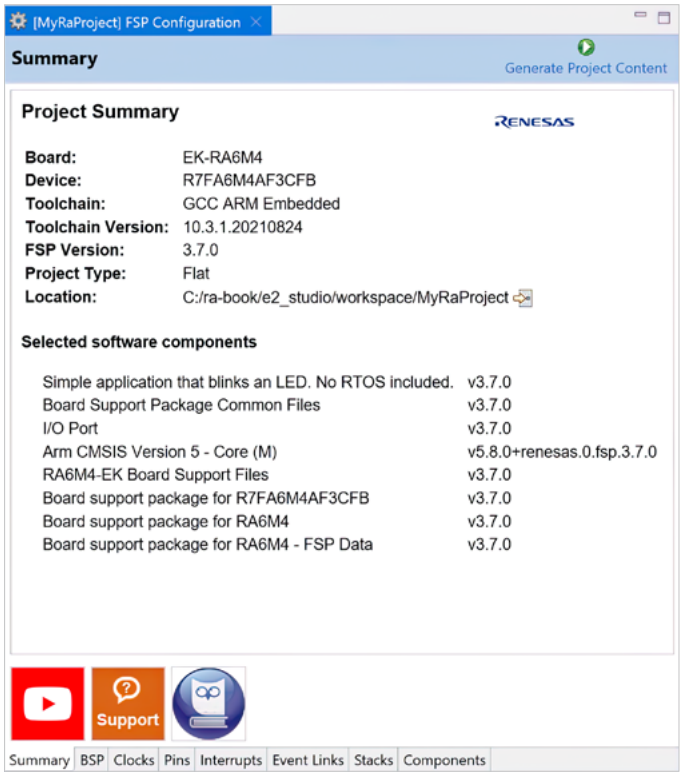


Figure 5-3: Summary tab The of the FSP Configurator

The next tab, named Clocks, is intended for assigning the initial clock configuration. A graphical representation of the on-chip clock system is shown (see *Figure 5-4*), and changes can be made to the clocking tree. Hovering with the mouse over the items will bring up a short description of them. If incompatible settings are made, the respective member will be highlighted in red and an explanation of the problem will be given. Also, the tab itself displays a small exclamation mark, indicating the presence of an issue.

The Pins tab is covering the initial pin setup of the RA MCU. Pins can be listed either based on ports or peripherals. A package view to the right of the configurator shows the package of the device highlighting configured pins and marking errors, if there are conflicts or incomplete settings. These will also be shown in the Problems view, as well as in the Pin Conflicts view.

The tab following, Interrupts, allows to specify how user defined (non-FSP) drivers make use of the microcontrollers Interrupt Controller Unit (ICU) inside an RA project and which interrupt service routine (ISR) is tied to the ICU events (interrupts). It also presents a complete list of all ICU events allocated. This table includes events generated by FSP module instances which have been created in the Stacks view of the configurator as well.

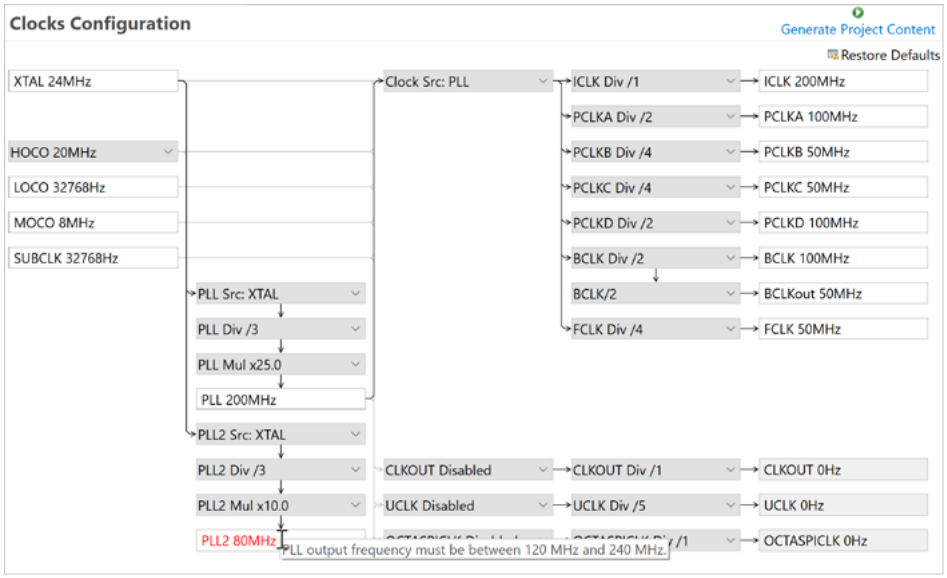


Figure 5-4: The clock view of the FSP-Configurator with an intentionally introduced error

The Event Links tab serves a similar purpose as the Interrupts tab. Here users can specify how their own drivers make use of the Event Link Controller (ELC) within an RA project and can declare that such a driver might produce a set of ELC events or consume a set of ELC events via a set of peripheral functions.

Next is the Stacks tab, which allows to add and configure threads and stacks within an RA project. Different modules and objects can be added to the individual stacks and their properties can be modified in the Properties view. The Stacks view displays the stacks of the different threads, creating a graphical configuration of the modules. New stacks can be added easily, with all modules necessary inserted automatically down to a level, where an intervention of the developer is required. If this is the case, the module will be marked in red and a description of the required settings or problems given, once the mouse is hovering over the module. If resolved, the module turns back to its standard color.

The final tab in the FSP Configuration perspective is named Components, allowing users to display and select different FSP components. It also lists the available RA CMSIS software components. Modifications like adding or removing modules from the current project should preferably be made in the Stacks tab, as the latter will also permit the configuration of them.

Once all settings are made in the FSP Configuration, the related source code can be generated or extracted from the FSP and added to the project. This is done by clicking on the Generate Project Content symbol at the top of the perspective. If you forgot to click on it, don't worry! This will also happen if changes to the configuration or generated files are detected.

The FSP Configurator is a great tool, as it guides you through all the steps needed to prepare a project and to make the initial settings for it. As with the Project Configurator, there is no necessity to peruse thousands of pages of documentation and to study them deeply, as the configurator will provide the necessary information on an abstract level and will assure that all settings are correct, plausible, and consistent. If you remember how much time you spent during your last project not using such configurators on just getting, for example, all the pin-routings and interrupt settings right, and dealing with a lot of interdependent hardware registers, you will surely appreciate all the effort Renesas put into these tools. It is much less trouble this way!

5.3 Importing, Exporting, and Using Projects

From time to time, you will have a need to import or export projects. Maybe you want to use one of the numerous example projects from the Renesas website, or that you need to share your latest development with one of your peers. No matter if you want to import or export, it can be conveniently done from inside e² studio using either the Import Wizard or the FSP Export Wizard.

5.3.1 Importing Projects

There are two ways to call the Import Wizard: you can open it by right-clicking in the Project Explorer view and selecting *Import* from the menu showing, or by going to the menu bar and selecting *File → Import*. In both cases, once the window of the Import Wizard comes up, expand the General entry and either select *Existing Projects into Workspace* or *Rename & Import Existing C/C++ Project into Workspace*. The latter will give you the opportunity to enter a new name for the imported project.

In both cases, you will be presented with the choice of either selecting the source directory, where the project to import resides, or choosing an archive file. If one or more projects are found in a folder, they all are marked for import. Cancel the selection of the projects you do not want to import. Click on *Finish* and the project(s) will be imported into your workspace, or copied there, if you checked the box besides *Copy projects into workspace*. Before you can run the imported project, you will have to recompile it.

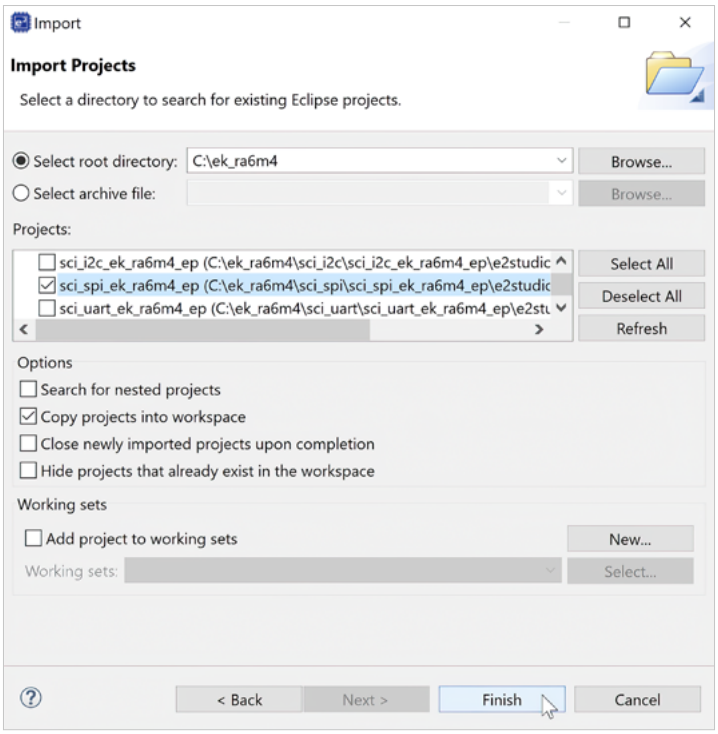


Figure 5-5: The projects found at a given location are listed for import

5.3.2 Exporting Projects

Exporting projects is as easy as importing them. Again, there are two ways to achieve the task. The first one is to go to the menu bar of e² studio and to select *File → Export*. The second one is to right-click on the project you want to export and choose *Export FSP Project*. If you are using the first version, there are two additional steps you have to perform in the export window showing up: The first is to expand the *General* entry and the second to select the *Renesas FSP Archive File* entry from the list appearing. Both versions will bring up the *FSP Export Wizard*.

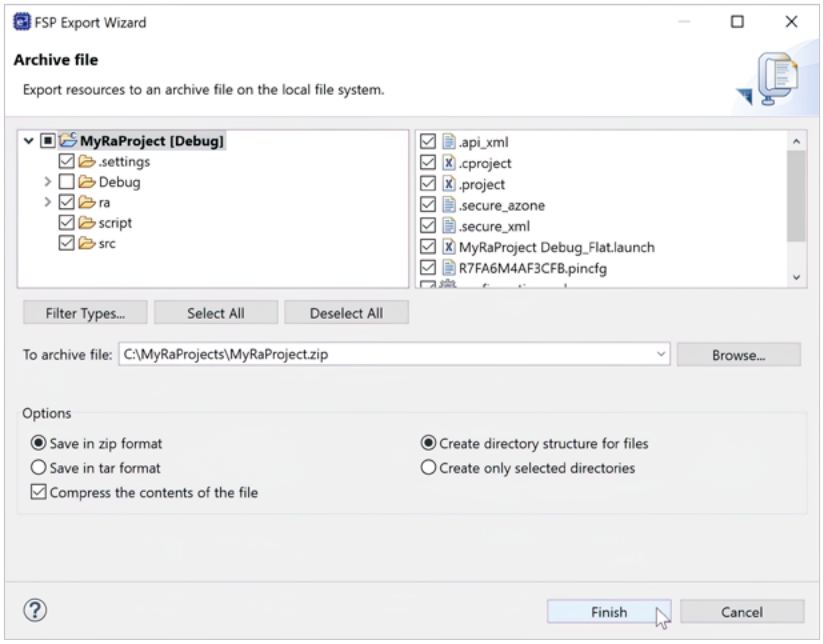


Figure 5-6: The FSP Export Wizard in e² studio

Once the FSP Export Wizard appears, the projects and files to be exported can be selected, along with the compression format and the desired directory structure. Clicking on *Finish* will finalize the export process.

Note that the *Debug* and ra entries are not checked by default. It is not necessary to include them in your export, as the configuration.xml file containing the information is part of the export and the missing content can be recreated by opening the file (which will cause the FSP Configurator to start) and clicking on *Create Project Content* again. This will restore the files omitted during an export. Projects exported this way can easily be shared and imported on other workstations.

Points to take away from this chapter:

- e² studio provides different perspectives and views to group functionality.
- The Smart Manual feature, the Project Configurator and the FSP Configurator speed up the development and reduce errors.
- Projects can easily be imported and exported by using the respective wizard.

6 HARDWARE EVALUATION KITS FOR THE RA FAMILY

What you will learn in this chapter:

- What the different development kits for the RA Family are and their main features.

In every development project, there is a point in time when you will need hardware to run your first tests. And, as all engineers know, this will almost always be the case well before your own board is even close of being ready to use. Or maybe the hardware guys don't want to loan one of their rare prototypes to their beloved software colleagues, who may even destroy the board (not that this happened ever to me!).

Renesas has a solution: It offers various kinds of easy-to-use kits which make starting with the platform effortless and hassle-free for software designers, as they can test their programs right away. Hardware designers will also benefit from the boards, as they can access most of the pins of the microcontroller (MCU) conveniently at the different connectors of the kits. If needed, they can also enrich the functionality of the boards by using popular ecosystem add-ons through standardized interfaces like Digilent Pmod™, SparkFun® Qwiic®, SeeedGrove® system, Arduino™ (Uno R3) connectors, or MikroElektronika™ mikroBUS connectors. These third-party modules can be used individually or simultaneously.

There is one Evaluation Kit (EK) per superset device – the one with the largest memory sizes and the largest choice of on-chip peripherals – in each MCU Group, offering an easy entry to the RA Family of microcontrollers and allowing to switch devices insides the MCU Group later on if smaller memory or less peripherals are needed. All EKs provide a common baseline functionality across the boards and are flexible enough to place specialized features on the PCB. All the kits come with pre-programmed quick-start example projects that you can use to get yourself acquainted with the hardware. Renesas also makes additional sample code available that engineers can utilize as building blocks for more complex custom embedded applications.

All kits are built in a way that they keep the learning curve flat for the user. They provide an inexpensive starting point for evaluation, prototyping, and developing for the RA Family of microcontrollers.

Physically, the boards consist of two different sections: The first section is the System Control and Ecosystem Access Area, where you will find onboard debugging, ecosystem expansion, user I/Os, serial connectivity, power regulation, user I/Os, reset, and boot configuration. This section is standard on all the boards based on this architecture.

The second section is unique to each kit. It comprises the Special Feature Access and Native MCU Pin Access Areas. The Special Feature Access Area includes peripherals which show the unique functionality of each kit such as Ethernet, Octo-SPI, Quad-SPI, and USB High Speed. The MCU Native Pin Access Area incorporates the microcontroller and allows direct access to the MCU terminals through male pin headers and clamping loops for current measurement.

Across all kits, there are some common features:

- Power supply
- Debugging interface
- Serial connectivity
- Microcontroller current measurement test points
- Native access to most of the MCUs digital and analogue pins.
- Wired connectivity
- User LED

Besides the Evaluation Kits, other boards and systems are available as well. These serve the special needs of certain applications like capacitive touch, motor control, or graphics, and contain one or more boards especially tailored to the application space they serve. Common to all kits is their modularity and their expandability, as well as their easy and error-proof configuration.

Each kit has its own website, and their URLs follow the syntax reneasas.com/<kit name>. The page for the EK-RA6M4 we use for our labs could therefore be accessed by typing **renesas.com/ek-ra6m4** into the web browser. There, you will not only find detailed information about each board, but you can also download the User's Manual of the kit, a Quick Start Guide, white papers, example projects, and much more. You will also be able to download a so-called Design Package, which includes schematics, the bill of materials, 3D and mechanical drawings, as well as manufacturing and design files which you could use as a base for your own developments.

And last but not least, the Evaluation Kits comply with many international standards, like the FCC Notice – Part 15 and CE Class A for EMC/EMI, the EU RoHS and China SJ/T 113642014 for waste, recycling and material selection, or the UL 94V-0 for safety, to name just a few.

6.1 The EK-RA6M4 Evaluation Kit

The EK-RA6M4 is an Evaluation Kit for the microcontrollers (MCUs) of the RA6M4 MCU Group. It is built around an R7FA6M4AF3CFB MCU with a 200 MHz Arm® Cortex®-M33 core with TrustZone® in a 144 pin LQFP package. It has 1 MB of on-chip Flash memory for code and 256 kB of internal SRAM. This kit enables users to effortlessly evaluate the features of the RA6M4 MCU group and to develop sophisticated embedded systems applications.

Multiple clock sources provide precision 24.000 MHz and 32,768 Hz reference clocks. Additional low-precision clocks are accessible internal in the RA microcontroller. Current measurement points are placed on the PCB as well to make accurate current consumption measurements possible.

Connectivity to the outside world is available through a USB full speed host and device port, as well as through an Ethernet connector. Access to most of the microcontrollers' features is provided by four 40-pin male headers and through expansion connectors to five of the most popular third-party ecosystems. Easy debugging of software and hardware is possible through the on-board J-Link® debugger. The board is fully supported by the Flexible Software Package (FSP) and the e^2 studio IDE.

Other features of the Evaluation Kit include:

- Memory:
  - On-chip: 1 MB code Flash memory plus 256 kB SRAM
  - Off-chip: 64 MB Octo-SPI Flash memory plus 32 MB Quad-SPI Flash memory
- Wired connectivity:
  - USB (1 x debug, 1 x USB full speed host and device, micro-AB connector)
  - Ethernet with RMII (RJ45 connector)
- Debug modes:
  - Debug on-board (SWD)
  - Debug in (ETM, SWD, JTAG)
  - Debug out (SWD)
- User interaction:
  - 2 x user button
  - Reset button
  - 3 x user LED
  - Power LED
  - Debug LED
- Expansion:
  - Four 40-pin male headers for native pin access
  - 2 x SeedGrove® expansion connectors (I2C / analog)
  - SparkFun® Qwiic® connector
  - 2 x Digilent Pmod™ connectors (SPI and UART)
  - Arduino™ Uno R3 connector
  - MikroElektronika™ microbus connector

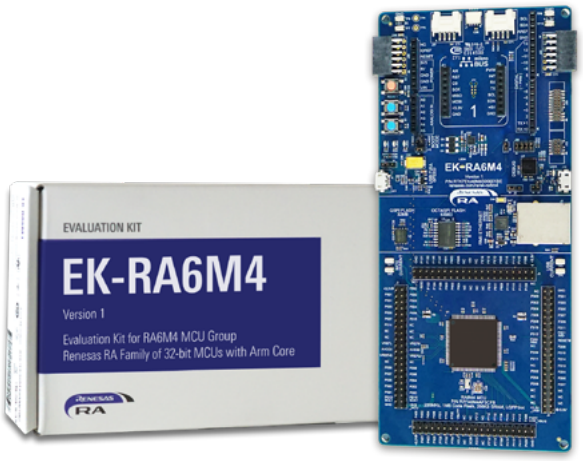


Figure 6-1: The EK-RA6M4 Evaluation Kit

An extensive set of documentation is available for this kit from the Renesas website: a Quick Start Guide walking you through the first start of the kit, a User's Manual, giving you all the technical details of the board and a design files package containing the schematics of the kit, the BOM, Gerber files and much more. As this is the board we will use for the exercises to come, you might want to download the User's Manual and refer to it in case of questions. Accessible from the same web page are several example projects which you might want to peruse once you have completed the labs in this book.

6.2 The EK-RA6M3G Evaluation Kit with Graphics Extension Board

If it comes to powerful human-machine interfaces (HMIs) or colorful displays, the RA6M3 Group inside the RA Family of microcontrollers is a good choice. To assess its on-chip graphic LCD controller for TFT and its 2D-graphics engine, Renesas provides the EK-RA6M3G Graphics Evaluation Kit. It enables engineers to seamlessly evaluate the features of the RA6M3 MCU Group and to develop their own graphics based embedded systems applications using the Flexible Software Package (FSP) and e^2 studio IDE. The kit is built around an R7FA6M3AH3CFC MCU with a 120 MHz Arm® Cortex®-M4 core in a 176 pin LQFP package. It has 2 MB of on-chip Flash memory for code and 640 kB of internal SRAM.

The EK-RA6M3G kit consists of two boards: an EK-RA6M3 board with the MCU and a Graphics Expansion Board featuring a 4.3-inch TFT color LCD with a capacitive touch overlay. To drive the LED backlight on the LCD, the Graphics Expansion Board includes a backlight controller as well.

Connectivity to the outside world is available through full- and high-speed USB ports, and through an Ethernet connector. Access to the features of the microcontroller is provided by four 40-pin male headers and through expansion connectors to four of the most popular third-party ecosystems. Easy debugging of software and hardware is possible through the on-board J-Link® debugger.

Multiple clock sources provide precision 24.000 MHz and 32,768 Hz reference clocks. Additional low-precision clocks are accessible internal in the RA microcontroller. Current measurement points are placed on the PCB as well to make detailed current consumption measurements of the MCU and the USB port possible.

Figure 6-2: The EK-RA6M3G Evaluation Board with graphics expansion board

Other features of the Evaluation Kit include:

- Memory:
  - On-chip: 2 MB code Flash memory plus 640 kB SRAM
  - Off-chip: 32 MB Quad-SPI Flash memory
- Wired connectivity:
  - 3 x USB (1 x debug, 1 x USB full speed, 1 x USB high speed)
  - Ethernet with RMII (RJ45 connector)
- Debug modes:
  - Debug on-board (SWD)
  - Debug in (ETM, SWD, JTAG)
  - Debug out (SWD)
- User interaction:
  - 2 x user button
  - Reset button
  - 3 x user LED
  - Power LED
  - Debug LED
- Expansion:
  - Four 40-pin male headers for native pin access
  - 2 x SeedGrove® expansion connectors (I2C)
  - 2 x Digilent Pmod™ connectors (SPI and UART)
  - Arduino™ Uno R3 connector
  - MikroElektronika™ microbus connector
- Graphics Expansion Board:
  - 4.3-inch TFT color LCD with a resolution of 480 x 272 pixels and a capacitive touch overlay with controller
  - Backlight controller

An extensive set of documentation is available for this kit from the Renesas website: white papers, application notes, a Quick Start Guide walking you through the first start of the kit, a User's Manual, giving you all the technical details of the kit, and a design files package containing the schematics of the boards, the BOMs, Gerber files, and much more. The development of graphical elements is supported by the Segger™ emWin Embedded GUI Library and the accompanying host-based AppWizard for creating ready-to-run applications for the emWin library. Both make the development of powerful HMIs a lot easier.

6.3 The RA6T1 Motor Control Evaluation System

The Motor Control Evaluation System for the RA6T1 Group of the Renesas RA Family of microcontrollers (MCUs) provides a low-voltage permanent magnet synchronous motor (brushless DC motor) solution. It bundles together all the necessary hardware for the evaluation of motor control systems as an easy-to-use kit. It allows an evaluation without difficulties, enabling the highly efficient development of such systems.

This comprehensive solution supports permanent magnet synchronous motors, three-shunt current detection, provides an overcurrent protection, and can communicate with the Renesas Motor Workbench through a USB port. The Motor Workbench allows to auto-tune the parameters of the control system, as well as the real-time monitoring of it. The kit itself consists of an RA6T1 CPU card, a 48 V inverter board, one permanent magnet synchronous motor, and the necessary cabling. By replacing the included CPU card with another CPU card, you can evaluate your motor control system with another MCU.

Figure 6-3: The Motor Control Workbench allows to auto-tune the parameters and to monitor the running system

The kit is built around the 120 MHz R7FA6T1AD3CFP MCU, which comes with an on-chip Flash memory of 512 kB and 64 kB of SRAM. This MCU is based on an Arm® Cortex®-M4 core and features a high-resolution PWM timer, a 12-bit A/D converter, a high-speed analog comparator and various interfaces.

Other features of the Evaluation Kit include:

- RA6T1 CPU Card:
  - R7FA6T1AD3CFP MCU
  - Board-to-board connector
  - USB connector for on-board J-Link™
  - SCI connector for Renesas Motor Workbench communication
  - Connectors for
    - *CAN communication*
    - *SPI communication*
    - *Hall sensor signal input*
    - *Encoder signal input*
    - *10/20 pin through hole for Arm® debugger*
  - Through hole for 2nd inverter
  - 2 x LED for user control
  - Switch for MCU external reset
- Inverter Board:
  - Toggle switch
  - Push switch
  - 3 x user LED
  - LED for power source for inverter control circuit block
  - Board-to-board connector
  - USB connector
  - Motor connector
  - Variable resistor

Figure 6-4: The RA6T1 motor control evaluation system

The kit comes with a pre-programmed software for an easy start. To get the motor turning, only the assembly of the hardware components is necessary. As with the other systems, a complete set of documentation is available from the website of the kit: white papers, application notes, a Quick Start Guide walking you through the first start of the kit, a User's Manual, giving you all the technical details of the boards and a design files package containing the schematics of the boards, the BOMs, Gerber files and much more. Example code can be downloaded as well, giving developers a head-start into their own design.

6.4 The EK-RA6M5 Evaluation Kit

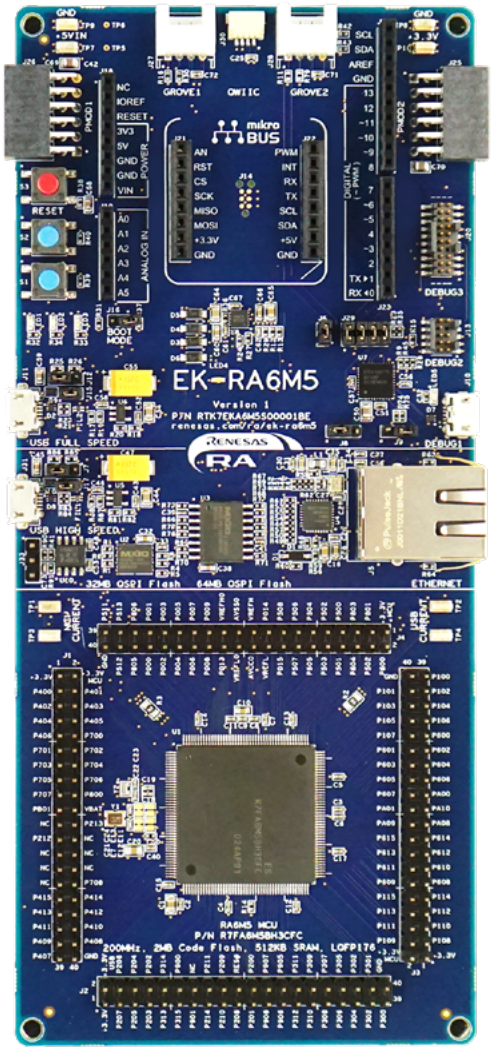


Figure 6-5: The EK-RA6M5 Evaluation Kit

For IoT applications requiring enhanced security, Ethernet or CAN connectivity, and a large, embedded memory, Renesas offers the RA6M5 Group microcontrollers. To support customers in their design, the EK-RA6M5 Evaluation Kit is available. It is built around the R7FA6M5BH3CFC MCU based on a 200 MHz Arm® Cortex®-M33 core in a 176 pin LQFP package. The processor comes with 2 MB of on-chip Flash memory for code and 512 kB of internal SRAM. Power consumption is low, it gets down to 107 µA/MHz running the CoreMark® algorithm from Flash.

Arm's® TrustZone® embedded in the core and a Secure Crypto engine makes the RA6M5 group of MCUs suitable for security applications like fire and burglar detection, or panel control. Other areas benefiting from the security feature and the rich connectivity are electricity meters and automated meter reading devices, as well as industrial applications such as robotics, vending machines, or UPS (Uninterruptible Power Supplies).

The EK-RA6M5 communicates with the outside world in a variety of ways: It has one USB full speed and one high speed host and device port, an Ethernet interface, and a CAN (Controller Area Network) bus transceiver which complies with the CAN FD (Flexible Data Rate) standard. CAN FD extends the original CAN bus standard and allows higher data rates, as well as larger payloads.

Debugging of software and hardware is easy, as the Evaluation Kit comes with an on-board J-Link® debugger. Precise current consumption measurements are possible through measurement points. External 24.000 MHz and 32,768 Hz reference clocks provide a precise time base. Additional low-precision clocks are available internally in the RA6M5 MCU as well.

- Memory:
  - On-chip: 2 MB code Flash memory plus 512 kB SRAM
  - Off-chip: 64 MB Octo-SPI Flash memory plus 32 MB Quad-SPI Flash memory
- Wired connectivity:
  - CAN FD
  - 3 x USB (1 x debug, 1 x USB full speed, 1 x USB high speed)
  - Ethernet (RMII, PHY, RJ45 connector)
- Debug modes:
  - Debug on-board (SWD)
  - Debug in (ETM, SWD, JTAG)
  - Debug out (SWD)
- Expansion:
  - 4 x 40-pin male headers for native pin access
  - 2 x SeedGrove® expansion connectors (I2C)
  - 2 x Digilent Pmod™ connectors (SPI and UART)
  - SparkFun® Qwiic™
  - Arduino™ Uno R3 connector
  - MikroElektronika™ microbus connector
- User interaction:
  - 3 x user LED
  - 2 x user button
  - Power LED
  - Debug LED
  - Reset button

As with the other Evaluation Kits, a User's Manual, application notes and other documentation, and example projects are available. Developers can also download a complete design package containing schematics, design files, the bill of materials, and mechanical drawings from the website of the Evaluation Kit.

Points to take away from this chapter:

- The different Evaluation Kits for the RA Family keep the learning curve flat.
- There is one Evaluation Kit per superset device inside each MCU group available and some kits have special features.

7. STARTING THE RENESAS EK-RA6M4 EVALUATION KIT FOR THE FIRST TIME

What you will learn in this chapter:

- How to connect the Renesas Evaluation Kit to your workstation.
- What a debug configuration is and how to create it.
- How to download and start a program in the integrated development environment.

In this chapter, we will verify that the EK-RA6M4 Evaluation Kit (EK) is working and communicating with your Windows® workstation and the debugger of e² studio. For this, we will use the project we generated in chapter 4.2. If you didn't do this exercise, don't worry, you can download it from the website for this book. But in the latter case, you should have already installed and tested the development toolchain as discussed in chapter 4.

7.1 Connection and Out-Of-The Box Demo

If you didn't unpack the EK from its box, this is now the time for it. Check the contents and verify that everything is there: the EK main board, one USB type A to micro-B cable, one micro USB OTG adapter, and one crossover Ethernet cable. The documentation for the kit, comprising the EK-RA6M4 Quick Start Guide and the User's Manual, can be downloaded from the *Documentation* section of the kit's website at **https://www.renesas.com/ra/ek-RA6M4**. Note that the board contains a SEGGER J-Link® On-Board (OB) debugger, providing full debug and programming capabilities, so there is no need for an emulator when using the EK and the USB cable(s) coming with the kit.

To start, insert the micro-B end of the USB-cable into the USB debug port J10 of the board, which is located at the lower right-hand side of the System Control and Ecosystem Access area, and the other end into a free USB port of your workstation. The white LED4, which forms the hyphen in the lettering "EK-RA6M4", should light up, indicating that the board has power. As soon as the board is up and running, a pre-programmed demo program executes, flashing the blue LED1 in a one-second interval with a luminosity of 10%. The green LED2 will be constantly ON at full intensity and the red LED3 OFF. With the user button S1, you can cycle the intensity of LED1 with each press from 10 percent to 50 percent and to 90 percent and back to 10 percent. With every actuation of push-button S2, you can sequence the blinking frequency of LED1 from 1 Hz to 5 Hz to 10 Hz and back to 1 Hz.

If the orange debug LED (LED5) besides the debug port continues to blink, the Evaluation Kit was not able to detect the J-Link drivers. If this happens, double check that you connected the USB cable to the debug port J10 on the board's right side and not to the USB Full Speed port J11 on the left-hand side. Also verify that the J-Link drivers have been copied to the host workstation during the installation process by opening the *Device Manager* of Windows and expanding the *Universal Serial Bus controllers* tree. The J-Link driver should be listed there (see *Figure 7-1*). If not, re-run the platform installer using the *Custom Install* feature and allow the installation of software from Renesas Electronics Corporation. This should fix the problem.

The board also comes with some other demonstration programs like a web server or a quad-SPI and octo-SPI speed comparison. To run these demos, you will need a terminal program on your workstation to talk to the evaluation kit. Please refer to the EK-RA6M4 Quick Start Guide for setting up the connection and the demos. For just getting going, running these demos is not required.

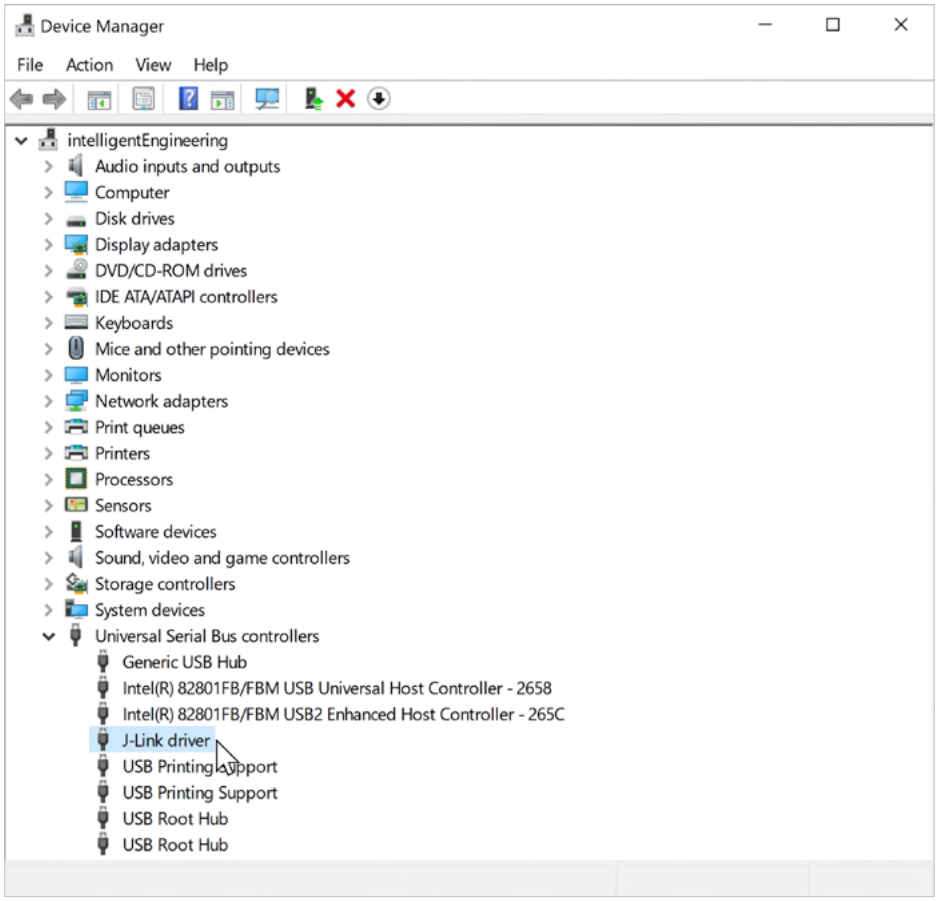


Figure 7-1: If the yellow debug LED does not stop flashing, make sure that the J-Link driver is properly installed

7.2 Downloading and Testing an Example

With the EK-RA6M4 still connected to your Windows® workstation, open e² studio from the Start menu of the operating system. If prompted for the location of the workspace, use the one you entered during the exercise in chapter 4.2 (it should already be listed). If you didn't do the exercise, don't worry, you can download the project from the Website created for this book (**www.renesas.com/ra-book**). Once downloaded, import it into your workspace according to the instructions outlined in chapter 5.3.1. In this case, use a folder of your choice.

Before we can download the program to the kit and run it, you need to create a debug configuration. Click on the small arrow beside the *Debug symbol* and select *Debug Configurations* from the drop-down list box.

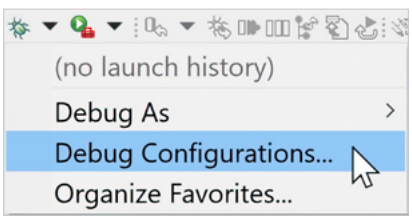


Figure 7-2: To start debugging, select Debug Configurations from the drop-down list box

In the window showing up, highlight *MyRaProject Debug_Flat* under Renesas *GDB Hardware Debugging* in the left-hand tree view. If you used a different name for this project, select the one you used.

Selecting your project will bring up a new screen for the Debug Configuration, showing all the individual options for it (see *Figure 7-3*). No need to change anything in here for our test purpose, just click on *Debug* at the bottom and the debugger will start. Once the *Confirm Perspective Switch* dialog displays, select *YES*.

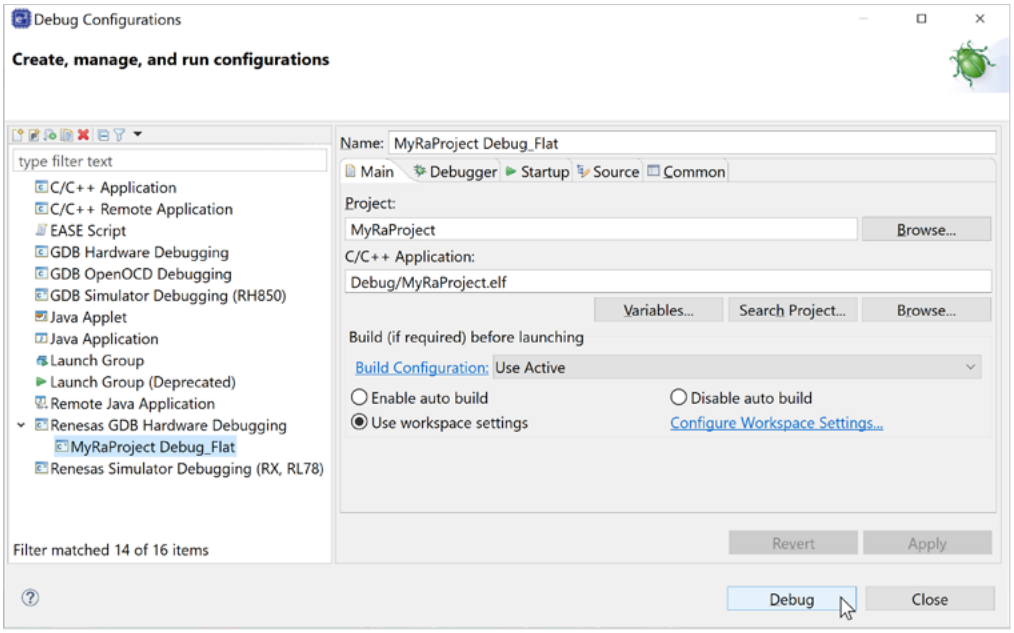


Figure 7-3: Once you selected your project under Renesas GDB Hardware Debugging, there is no need to change anything in the window appearing

There might be a second dialog named *J-Link® Firmware update* showing up, asking to install a new firmware version for the on-board debugger. I highly recommend allowing the update by clicking *Yes*.

Depending on the security settings of your Windows workstation, a dialog window displaying a security alert might appear, telling you that the "Windows Defender Firewall has blocked some features of E2 Server GDB on all public and private networks." To continue, permit the E2 Server GDB to communicate on private networks. For this, select the respective check box and click on *Allow access*.

Once the Debug perspective is open (see *Figure 7-4*), the debugger will set the program counter to the entry point of the program, the reset handler. Click on the *Resume button* and the program will run to the next stop inside the `main()` function, at the line with the call to `hal_entry()`. Click *Resume* again and the program continues to execute, flashing the blue, green, and red LEDs (LED 1 through 3) on the Evaluation Kit simultaneously in a one-second interval.

The final step is to disconnect the debugger from the board by clicking on the *Disconnect button* , stopping the execution of the program.

Now that you are sure that your installation of e² studio works together with your Evaluation Kit, it is time to write your first program for the RA Family of microcontrollers. This will be the topic of the next chapter.

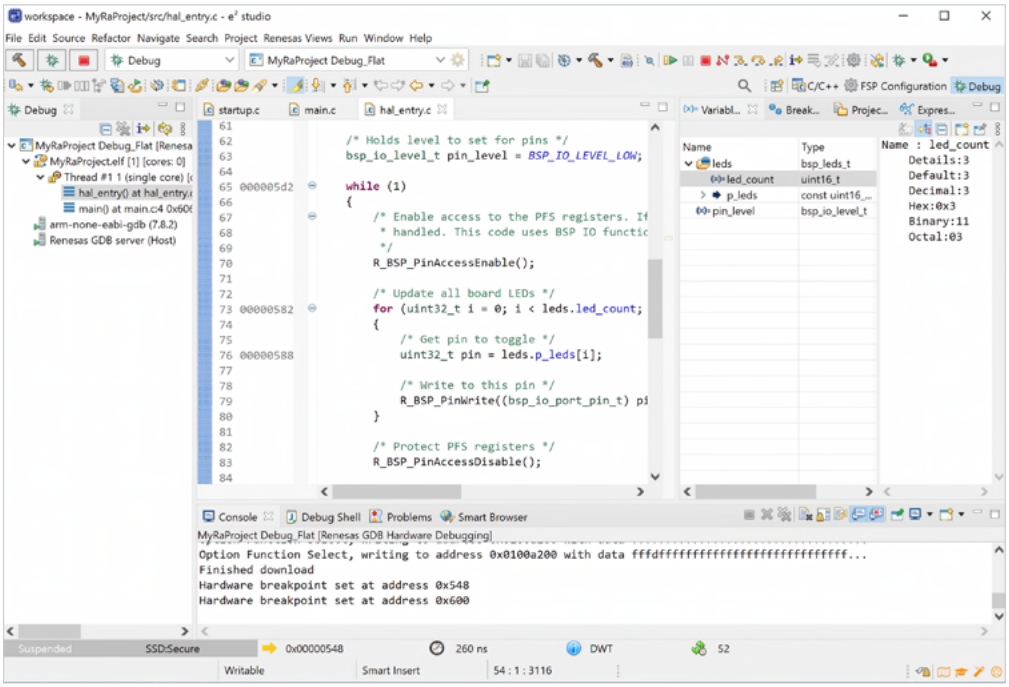


Figure 7-4: The Debug perspective of e² studio

Points to take away from this chapter:

- In order to download and debug a program to any hardware, you need to create a debug configuration first.
- Once loaded, the debugger will set the program counter to the entry point. The next stop will be at main().

8 HELLO WORLD! – HI BLINKY!

What you will learn in this chapter:

- How to create a project for the EK-RA6M4 Evaluation Kit from scratch.
- How to change settings for the Flexible Software Package in the FSP Configurator.
- How to write code to toggle the User LEDs on the EK.
- How to download and test a program.

The very first program most newcomers to a programming language wrote (and still write) is the one which simply puts the string "Hello World" to the standard output device. For me, it was typing "Writeln ('Hello World')" into the editor, as I started with Pascal. Ever since then, I wrote similar lines in several other languages, mostly as a sanity check for the installation of a new development environment.

When I moved on to program embedded systems in the late 1980's, there was no screen where the string could be sent to. So how to instruct the processor to give signs of life? LEDs were barely found in these applications, so toggling one of the very few I/O-pins and observing the waveform with an oscilloscope was the way to go. Over the years, LEDs became a commodity item and we placed plenty of them on our boards, using their blinking as the new "Hello World".

And this is also the objective of this chapter: Toggle LEDs on the Evaluation Kit (EK) for the RA6M4 Group devices, using everything you learned in the chapters before: You will write the code (nearly) from scratch, create a new project using the configurators, employ the APIs of the Flexible Software Package (FSP), and finally download, debug, and run the code. This exercise brings every single thing together.

As a prerequisite, e^2 studio, as well as the FSP should be installed on your Windows® workstation (see chapter 4 for details) and you should have verified that your setup is working as described in chapter 7. And those of you doing the previous hands-on labs: Please bear with the author, as he decided to cover again some of the topics already discussed earlier on for the sake of those of you moving directly from the table of contents to this chapter.

For this exercise, we will once more use the EK-RA6M4, which is extremely well suited for tasks like this, as it allows you to explore the RA6M4 microcontroller and all its peripherals instantly. External hardware can be easily attached, as most of the pins are accessible through either the breakout pin headers in the MCU Native Pin Access area or through the ecosystem connectors in the System Control and Ecosystem Access area of the board. With the RA6M4 Group MCU being the superset device of the RA6 Series of the RA Family MCUs, most features of this Series can be evaluated and the results later on applied to the smaller siblings of the family as well. Figure 8-1 shows the block diagram of the board, highlighting the main components.

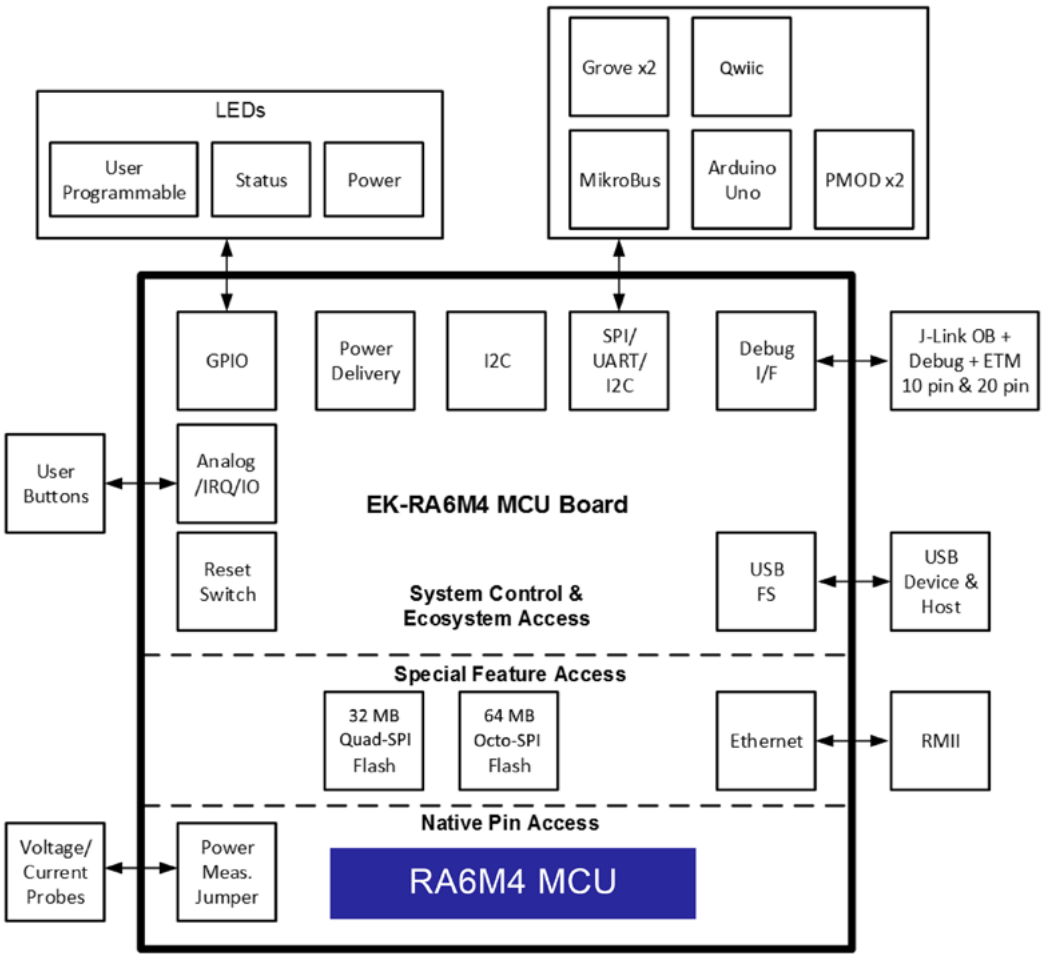


Figure 8-1: Block diagram of the EK-RA6M4 Evaluation Kit

8.1 Creating a Project with the Project Configurator

If not already done, open e² studio from your Windows® workstation Start menu. Once the development environment is up and running, dismiss the *Welcome* screen – if it shows – as it would block other windows from viewing.

As writing a new program for a microcontroller in e² studio always entails the creation of a project, this is the first step you need to take. For this, go either to *File → New → Renesas C/C++ Project → Renesas RA*, or right click in the Project Explorer view and select *New → C/C++ Project*. Both ways will bring up a dialog asking for the template to be used. Select *Renesas RA* on the left sidebar and then *Renesas RA C/C++ Project* from the main window (see *Figure 4-7*). Then click on *Next*.

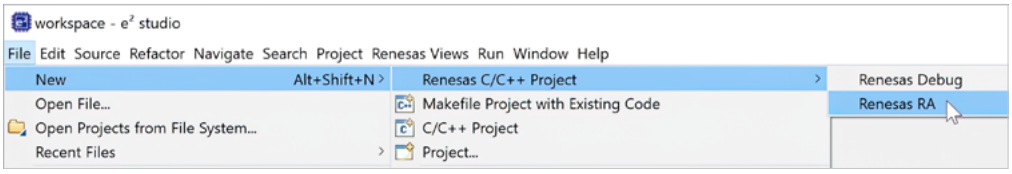


Figure 8-2: First step is to call the Project Configurator

Once the *Project Configurator* shows, give the project a name, for example *MyBlinkyProject* and either accept the default location for the project – which will be your e² studio workspace – or change it to a folder of your preference. Click on *Next* to move on to the *Device* and *Tools Selection* screen.

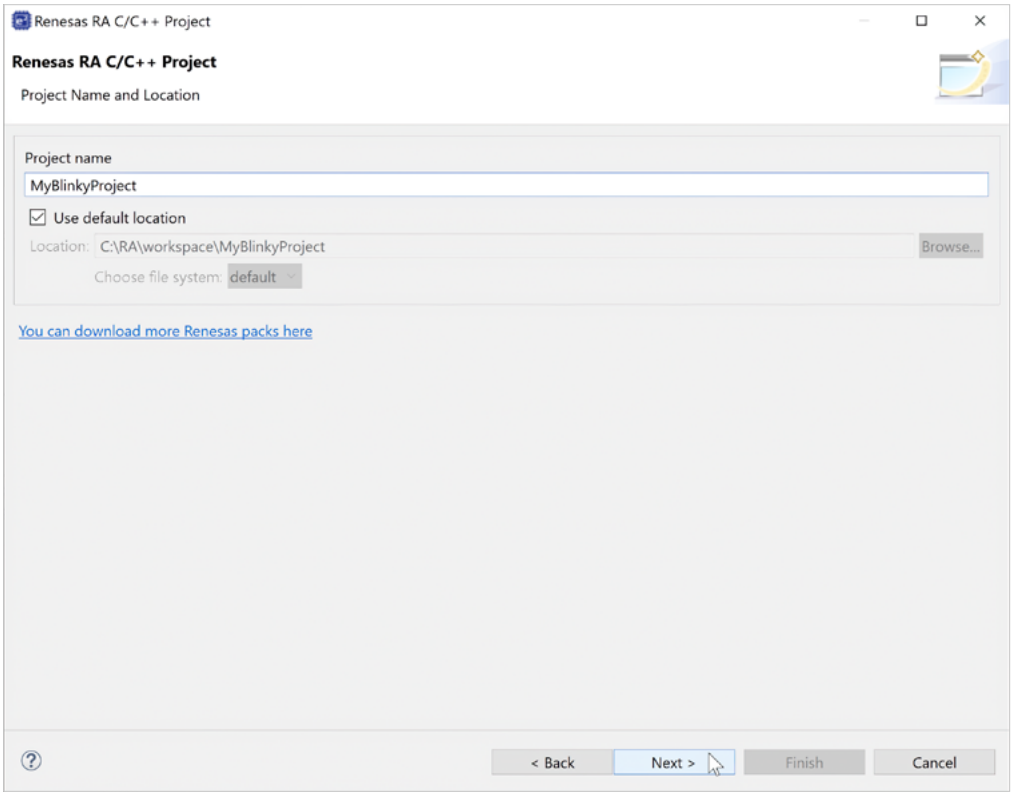


Figure 8-3: The first screen of the project configurator mainly asks for the project's name and location

Under *Device Selection,* look out for the field called *FSP Version*: it should show the same version of the Flexible Software Package you downloaded before. Under *Board*, select *EK-RA6M4* from the drop-down list, as this is the hardware we use for our small "Hello World" program. The list will typically contain the evaluation boards for the RA Family plus a *Custom User Board* entry and is created from the Renesas CMSIS pack files installed for the selected FSP version. Verify that *R7FA6M4AF3CFB* is shown beside Device, it should have been automatically inserted. If not, navigate through the drop-down list until you found it. In the *Toolchains* frame, verify that *GCC ARM® Embedded, 10.3.1.20210824* or later is listed, and that *J-Link® Arm* is selected in the *Debugger* frame. These fields should be pre-populated for you. If not, modify them to match the values given above.

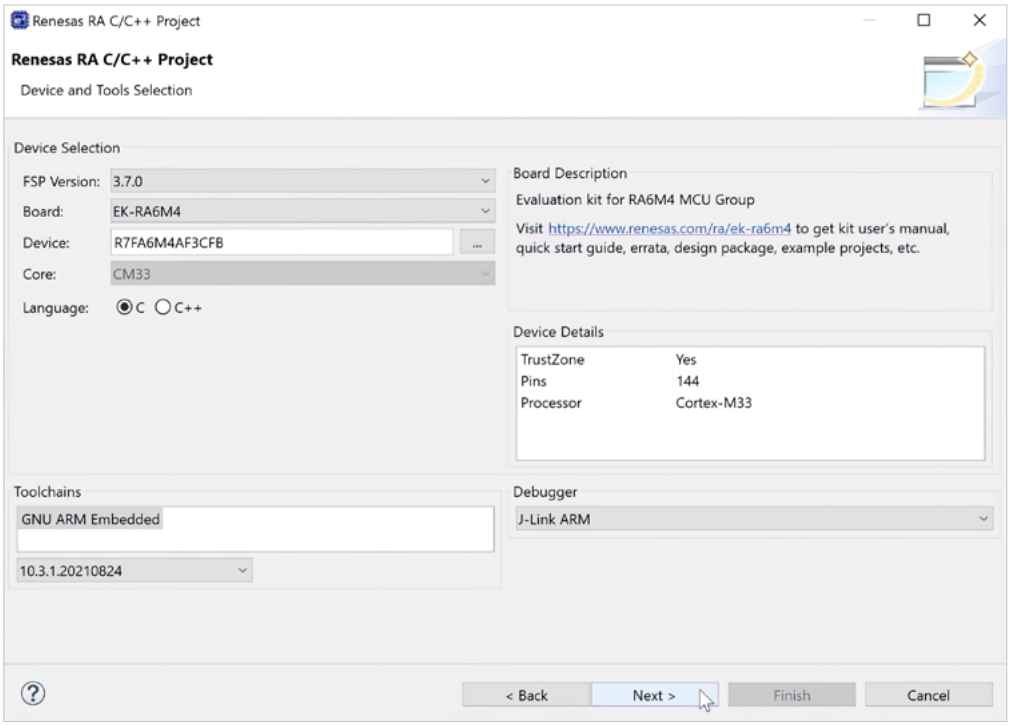


Figure 8-4: This page lets you select the board and the device for your project

If everything looks right to you, click on *Next* to open the *Project Type* Selection screen. Here you can choose whether your project should be a so-called "flat project", which means a project without TrustZone® separation for immediate execution, a secure TrustZone project including secure startup code and other secure code, or a non-secure TrustZone project containing non-secure code for use together with a secure project. For the exercise in this chapter, select *Flat (Non-TrustZone) Project* and click on *Next* to proceed.

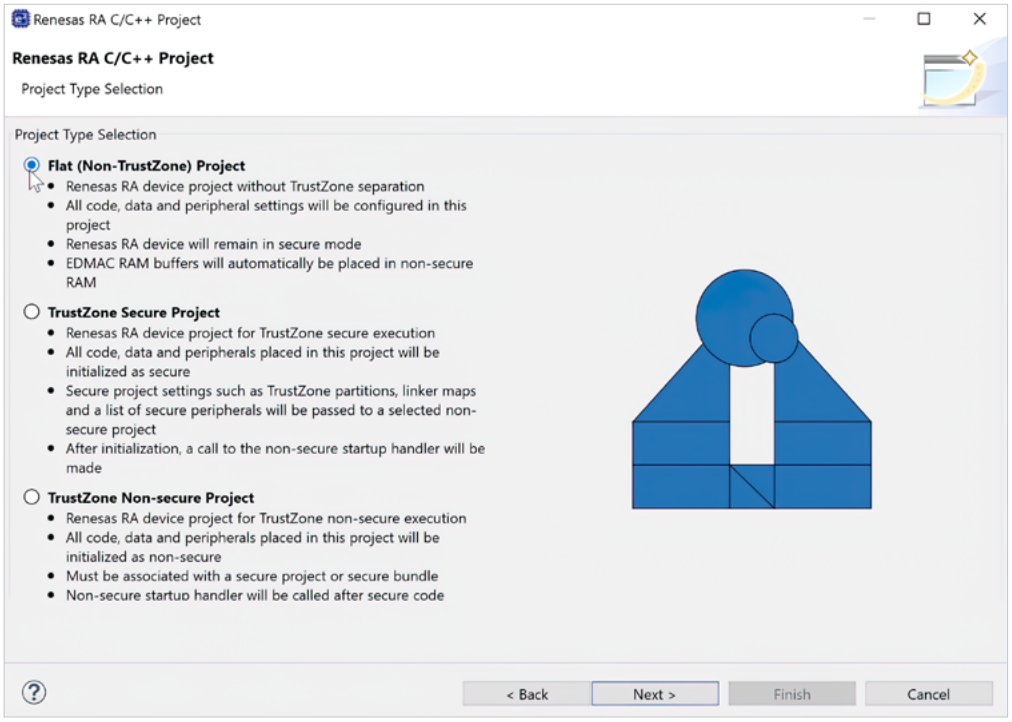


Figure 8-5: The Project Type Selection screen lets you choose between TrustZone and non-TrustZone projects

The next page is the *Build Artifact and RTOS Selection* screen, allowing you to set the type of the build. This screen only shows if a non-TrustZone device or a flat project for a TrustZone device was selected on the previous window. Options available are *Executable*, creating a self-contained ELF (Executable and Linkable Format) executable file, *Static Library* creating an object-code library, and *Executable using an RA Static Library*, which creates an application project configured for use with a static library. At the right-hand side of the page, a drop-down list allows you to choose an optional Real-Time Operating System (RTOS) for your project.

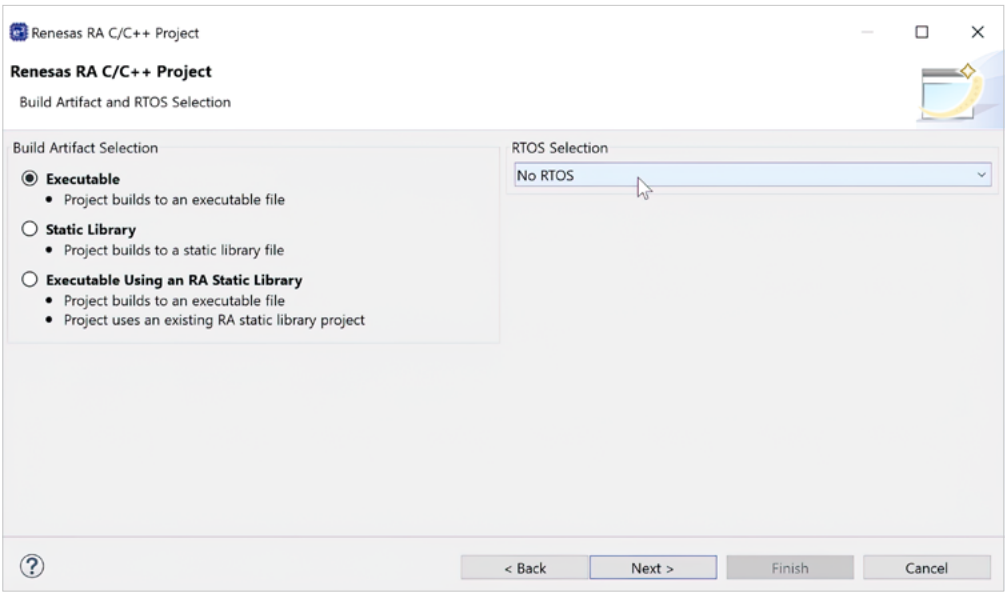


Figure 8-6: We want to build an executable project without RTOS

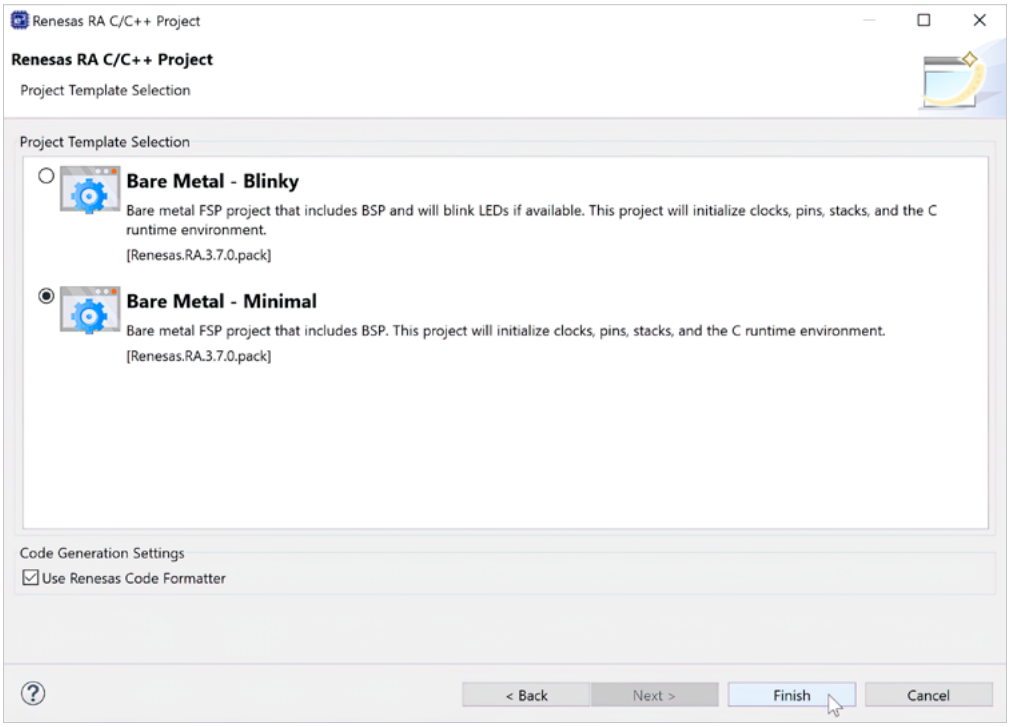


Figure 8-7: This page lets you select the board and the device for your project

For our small hands-on lab, select *Executable* and *No RTOS* before clicking on *Next*.

This will bring up the *Project Template Selection* page where you can pick a template for the initial project content. A project template may include several items; at least it comprises the correct Board Support Package for the selected board / device combination. Some templates even include a complete example project, but the Project Configurator will only present templates matching the selections you made on the previous screens. In our case, select the *Bare Metal – Minimal* entry, which will load the Board Support Package for the Evaluation Kit. Click on *Finish* to finalize the configuration or your project.

The *Project Configurator* will close and will create all the files necessary for the project in a last step. Once this post-processing is complete, a dialog will ask you if you want to open the *FSP Configuration* perspective. Select *Open Perspective*.

8.2 Setting Up the Runtime Environment with the FSP Configurator

Once the FSP Configurator has started, it presents you a read-only summary of the project and a short overview over the software components selected. It also provides convenient shortcuts to the Renesas RA channel on YouTube™, the Renesas Design & Support page on Renesas.com, where you can access the documentation, the Knowledge Base and the Renesas Rulz Forum, and to the FSP User's Manual on your hard drive.

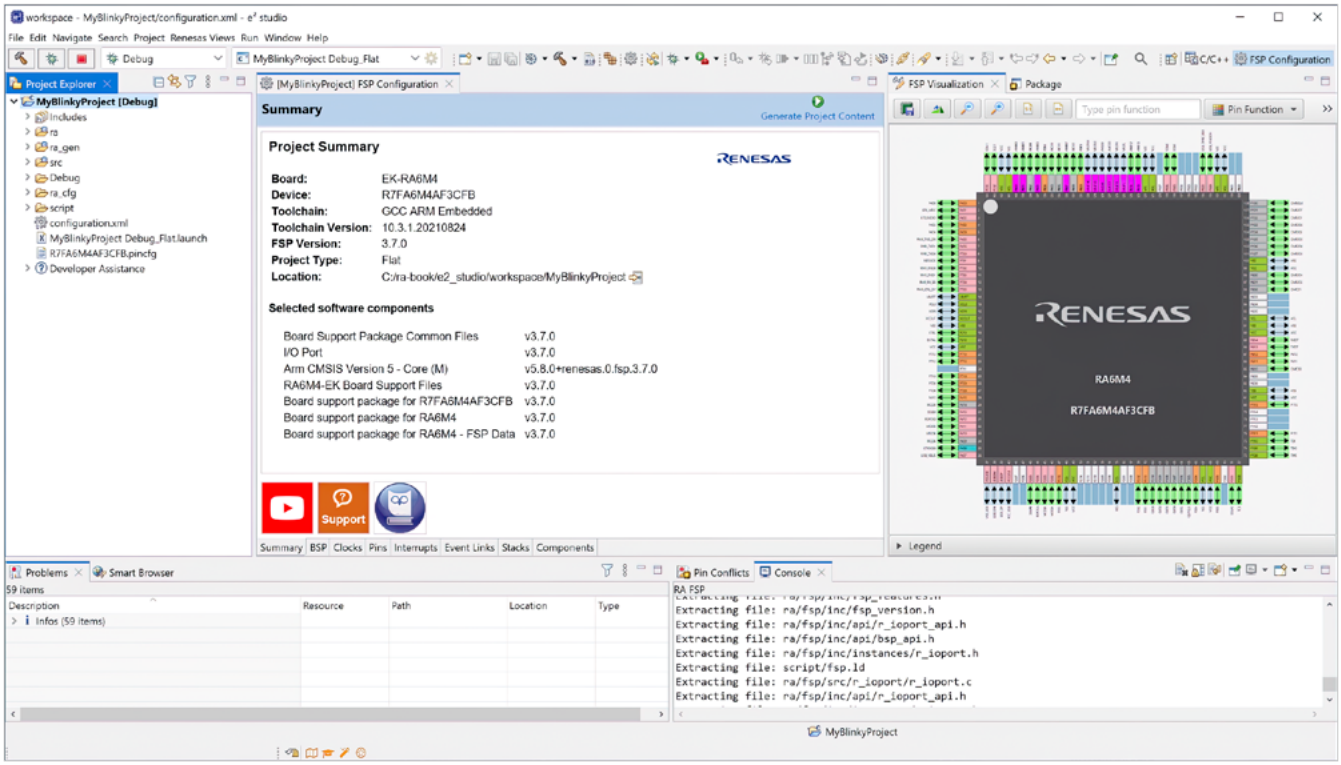


Figure 8-8: The FSP Configuration perspective inside e² studio

The following tab, called *BSP*, allows you to view and edit several aspects of the setup, like the board and device selection. In the properties view for this tab, you can make additional settings for the Board Support Package, for example, the size of the main stack or to some security features of the MCU. In the *Clocks* tab following, you can assign the initial clock configuration for your project. Any potential problem will be highlighted in red, and hovering with the mouse over a highlight will give you an explanation of the conflicting or incomplete setting.

The fourth tab, *Pins*, is covering the pin assignment of your RA MCU. You can list the pins either based on ports or peripherals. A *Package View* to the right of the configurator shows you the package of the device highlighting configured pins and marking errors, if there are incompatible or missing settings. These will also be shown in the *Problems* view, as well as in the *Pin Conflicts* view. This way, possible mistakes are reduced to a minimum.

Next is the *Interrupts* tab. There you can specify how your user-defined (i.e., non-FSP) drivers use the Interrupt Controller Unit (ICU) of the microcontroller and which Interrupt Service Routine (ISR) is tied to the ICE events (interrupts). There, you can also view a complete list of all ICU events allocated, including those generated by the FSP module instances created in the *Stacks* view of the configurator.

The *Event Links* tab serves a similar purpose. Here you can specify how your drivers make use of the Event Link Controller (ELC) within an RA project and you can declare that such a driver might produce a set of ELC events or consume a set of ELC events via a set of peripheral functions.

The page you will spend most of the time on is the *Stacks* page. It allows you to create RTOS threads and kernel objects, as well as FSP software stacks. Different objects and modules can be added, and their properties modified in the *Properties* view. All of them will be inserted automatically until down to a level where a user intervention is required. If this is the case, the module(s) needing attention will be marked in red and a description of the necessary settings or problems given, once the mouse is hovering over the module. If resolved, the module turns back to its standard color. The *Stacks* view itself display the various stacks in a graphical manner, making it easy for you to keep track of the different modules. In the case of our example, only one thread having one module is shown: the HAL/Common thread with the g_ioport I/O Port (r_ioport) driver. It was automatically inserted by the Project Configurator and will allow us to write the program for the blinking LEDs with just a few lines of code.

Components is the name of the final tab displaying the different FSP modules and permitting their selection. It also lists the RA CMSIS software components available. But you should preferably add or remove modules to or from the current project through the *Stacks* page, as you can configure them there as well.

For our project, there is no need to change anything in the FSP Configurator, as all necessary settings have been made for us already by the Project Configurator. As the final step, additional source code based on the current configuration needs to be created. Click on the *Generate Project Content* button at the top right-hand side of the FSP Configurator. This will extract the required files from the FSP, adjust them to the settings made in the configurator, and add them to the project.

8.3 Writing the First Lines of Code

With all the automatically generated files being now in place, it is time to have a look at what was created. The *Project Explorer* at the left-hand side of the IDE lists everything currently included. The *ra_gen* folder holds configuration sets such as channel number, etc. The *src* directory comprises a file named *hal_entry.c*. This is the one you will edit later on. Please note that while there is a file called *main.c* in the *ra_gen* folder, your user code must go to *hal_entry.c*. Otherwise, you will lose your changes, if you make modifications in the FSP Configurator and re-create the project contents, as this file will be overwritten each time you click on *Generate Project Content*.

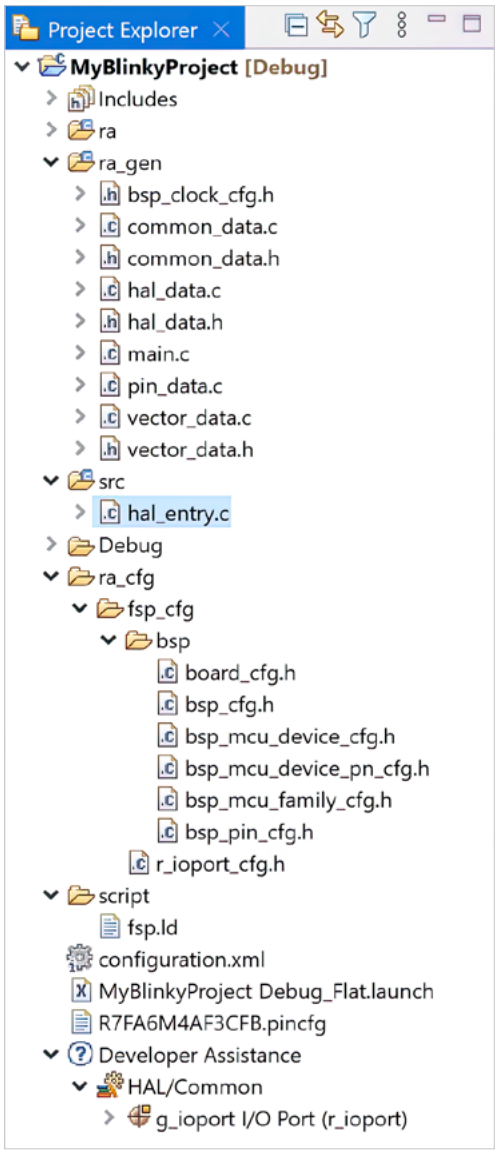


Figure 8-9: The project tree after the FSP Configurator created the files necessary

The project also contains several directories with "ra" or "fsp" in their names, containing source-, include- and configuration files for the FSP. It is a general rule that the contents of these folders (and subfolders) should not be modified. They contain files generated by the configurator(s) and any change made there will be lost the next time the project content is generated or refreshed. The user editable source files are those directly in the root of the *\src* folder or any other folder added by you.

Now it is time for you to write your first real source code for the RA Family of microcontrollers. The plan is to alternate between the green LED2 and the red LED3 on the EK-RA6M4 Evaluation Kit every second, so you will have to add code for turning them on and off and for a delay loop. How to do that?

There are actually two options for that: One is using the API by using the interface functions and one is using the BSP implementation functions. Which one do you think is the better one? You can review chapter 2, if you are unsure about the answer.

Looking at the code in the file *ra_gen\common_data.c*, we find that there is the following definition for the I/O port driver instance `g_ioport`:

```
const ioport_instance_t g_ioport = {.p_api = &g_ioport_on_ioport,
                                    .p_ctrl = &g _ioport_ctrl,
                                    .p_cfg = &g_bsp_pin_cfg, };
```

`g_ioport_on_ioport` is a structure, which declares the possible actions for the port, and which is assigned to the API pointer of the `g_ioport` instance. The contents of the structure can easily be viewed by hovering with the mouse over it, which will reveal that one of its members – `.pinWrite` – is a pointer to the pin write function.

So, turning an LED on, you could write:

```
g_ioport.p_api->pinwrite (&g_ioport_ctrl, pin, BSP_IO_LEVEL_LOW);
```

But this means that you would actually need to know which I/O-ports LED2 and LED3 are connected to and how many LEDs are available for use! For that, we could either read the documentation of the board or scrutinize the schematics to find the correct port. Or you can simply rely on the FSP. Creating a structure of the type `bsp_leds_t` (declared in *board_leds.h*) and assigning the global BSP structure `g_bsp_leds` defined in *board_leds.c* to it will do the trick. Both files reside inside the *ra\boards\ra6m4_ek* folder of your project. So, the following two lines of code are sufficient to get the information about the LEDs on the Evaluation Kit:

```
extern bsp_leds_t g_bsp_leds;
bsp_leds_t Leds = g_bsp_leds;
```

Now you can access all the LEDs on the boards by using the LEDs structure and turn on the green LED2 with the following statement (setting a port to a LOW level, will turn the LED on, setting it to HIGH will turn it off):

```
g_ioport.p_api->pinWrite(&g_ioport_ctrl,
                         Leds.p_leds[BSP_LED_LED2],
                         BSP_IO_LEVEL_LOW);
```

This statement needs to be followed by a second one to turn LED3 off by setting its pin-level to HIGH.
Finally, you need to provide a delay to have the LEDs toggle in a user-friendly way. For that, you again can call a BSP API:

```
R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_SECONDS);
```

The first parameter of the `R_BSP_SoftwareDelay` function is the number of units to delay, while the second one is the base for the units specified, in our case seconds. Other options would be milli- and microseconds.

All what is needed once this is done, is to copy/paste the three lines of code and to reverse the pin-levels of the LEDs in the second set. And finally, as we want to run the program indefinitely, a `while(1)` loop has to be created around the code.

What is left now, is to enter the following lines of code into the *hal_entry.c* file directly after the function signature, replacing the `/* TODO: add your own code here */` line. Please leave the other code inserted by the Project Configurator and the FSP Configurator untouched. It is needed by the microcontroller to operate properly.

```
extern bsp_leds_t g_bsp_leds;
bsp_leds_t Leds = g_bsp_leds;

 while (1)
 {
     g_ioport.p_api->pinWrite(&g_ioport_ctrl,
                              Leds.p_leds[BSP_LED_LED2],
                              BSP_IO_LEVEL_LOW);
     g_ioport.p_api->pinWrite(&g_ioport_ctrl,
                              Leds.p_leds[BSP_LED_LED3],
                              BSP_IO_LEVEL_HIGH);
     R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_SECONDS);

     g_ioport.p_api->pinWrite(&g_ioport_ctrl,
                              Leds.p_leds[BSP_LED_LED2],
                              BSP_IO_LEVEL_HIGH);
     g_ioport.p_api->pinWrite(&g_ioport_ctrl,
                              Leds.p_leds[BSP_LED_LED3],
                              BSP_IO_LEVEL_LOW);
     R_BSP_SoftwareDelay(1, BSP_DELAY_UNITS_SECONDS);
}
```

While writing the code, you can always use the auto-completion feature of e² studio. Just press <ctrl>-<space> and a window displaying possible completions for the structure or function will appear. If you click on an entry, it will be automatically inserted into the code.

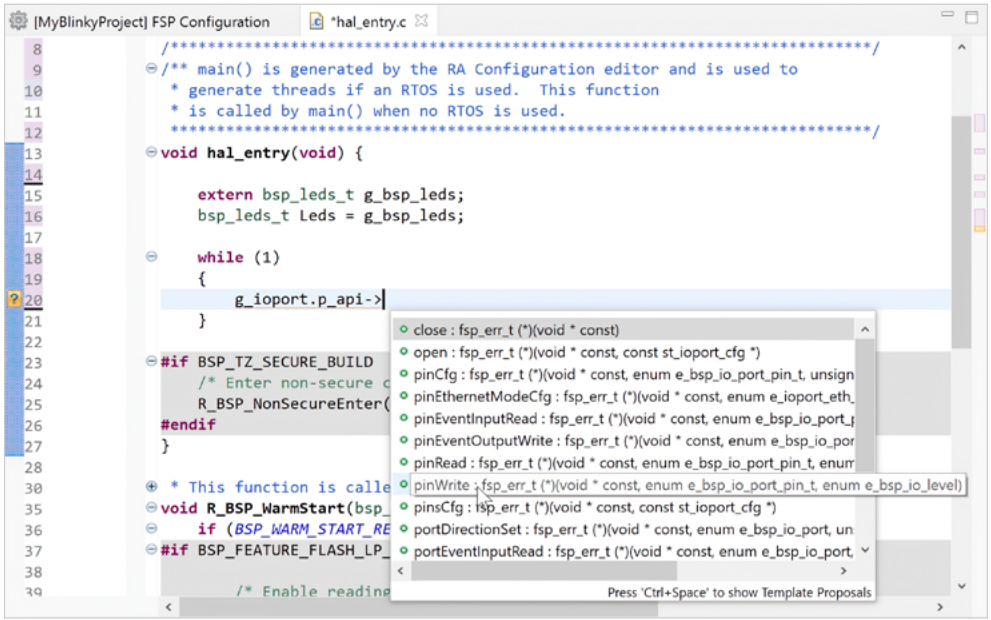


Figure 8-10: Pressing <ctrl>-<space> at a variable or function will activate the code completion feature of e² studio

Another helpful tool while writing your program is the Developer Assistance, which can be accessed from the Project Explorer. After having configured the software stacks for your project with the FSP Configurator, this tool supports you in getting started quickly with your application code. To access the Developer Assistance, first expand your project in the Project Explorer so that the tool is shown. With the tool visible, expand the tree further until you see the stack module and their APIs. Select the API you want to use and drag and drop the call to it into your source file.

It is now your turn: please enter the lines of code above into the *hal_entry.c* file in your project. For that, expand the *src* folder of your project and double-click on the file. This will open it in the editor. If you do not want to type everything yourself, you can also download a complete project from the website of this book (**www.renesas.com/ra-book**).

8.4 Compiling the First Project

When you did all the typing, the program is ready to be built. There are two different configurations for a build: Debug and Release. The Debug configuration will include all information necessary for debugging a program, like variable and function names, and will also turn off certain optimizations of the compiler, for example loop unrolling. This makes debugging easier, but will create larger and slower code. The Release configuration will strip all this information from the output file and turn on full optimization, thereby creating smaller and faster code, but you will no longer, for example, be able to view variables, unless you know their addresses in memory.

For your first test, the  bug configuration, which is also the default, is the way to go. To build your project, click on the *build-button* on the main menu bar and the process will start. If you did everything right, the compilation will finish with 0 errors and 0 warnings. If there are compile-time errors, you need to go back to your code and double-check, if you entered everything correctly. If not, change your code accordingly. To make it easier for you to locate the errors, the feedback of the compiler will be inserted directly into the editor window, where possible.

Once the program build was successful, the output file *MyBlinkyProject.elf* has been created, which needs to be downloaded to the processor before we can run and debug it.

8.5 Downloading and Debugging the First Project

The next step will be to actually run the program on the Evaluation Kit (EK). And this is now the right moment to connect the kit to your Windows® workstation: Insert the micro-B end of the USB-cable delivered with the board into the USB debug port J10 located at the lower right-hand side of the System Control and Ecosystem Access area, and the other end into a free port on your PC. The white LED4 forming the hyphen in the "EK-RA6M4" lettering should light up, indicating that the board is powered. If the kit just came out of the box, the pre-programmed demo will run, signaling that everything is working as expected. The Windows operating system might display a dialogue indicating the installation of the drivers for the J-Link® On-Board Debugger, which should complete automatically. Additionally, a window asking for the update of the J-Link® Debug Probe may appear. It is strongly recommended to allow this update to happen.

If the orange debug LED5 besides the USB port does not stop blinking after a short while, there might be a problem with the J-Link® drivers on your workstation. If this occurs, please consult chapter 7.1 for possible solutions.

DOWNLOADING

To download our program, we will have to create a debug configuration first. Click on the small arrow beside the Debug symbol  and select *Debug Configurations* from the drop-down list box.

In the window showing up, highlight *MyBlinkyProject Debug_Flat* under *Renesas GDB Hardware Debugging*. As all the necessary settings have already been made by the Project Configurator, there is nothing to change in this dialog. Just click on *Debug* in the lower right corner of the window. This will launch the debugger, download the code to the RA6M4 MCU on the EK and will ask you, if you want to switch to the *Debug perspective*. Answer with *Switch*. The *Debug perspective* will open, and the program counter will be set to the entry point of the program, the reset handler. This debug configuration needs to be created only once. The next time you can start the debugger by just clicking the *Debug* symbol  .

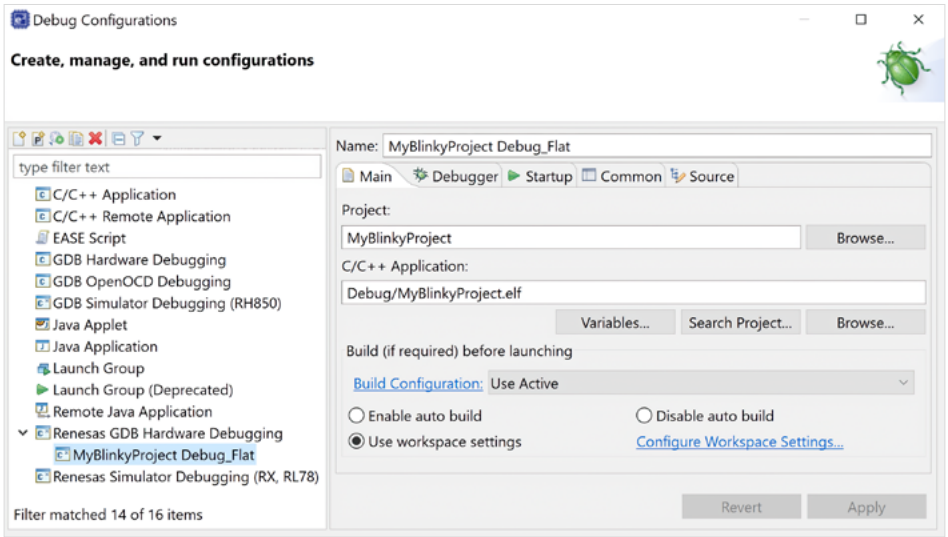


Figure 8-11: Once the MyBlinkyProject Debug_flat is selected, no changes need to be made on the different tabs

RUNNING

Click on the *Resume* button and the next stop will be at `main()`, at the call to `hal_entry()`. Click again on the button and the program continues to execute, toggling between the green and the red LED on the Evaluation Kit in a one-second interval as intended.

WATCHING THE RESULTS

If everything is working as expected, click on the *Suspend* button on the main menu bar. This will stop the execution of the program without terminating it. In the editor view, activate the tab with the file *hal_entry.c*, and right click in one of the lines with a write to the ports; in the menu showing, select *Run to line*. Execution will resume and the program will stop at the line you clicked in. Now have a look at the view with the variables at the right. You will see the `Leds` structure listed. Expand it and browse and analyze the different fields. This view will come handy when debugging a larger project.

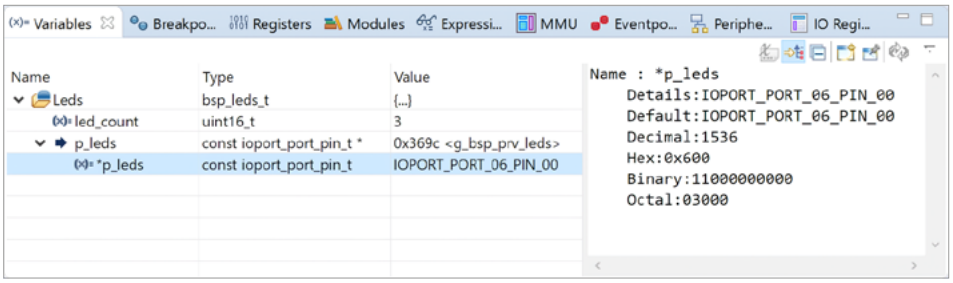


Figure 8-12: Variables and their values can be scrutinized in the Variables view

The final step is to end the debug session by clicking on the *Terminate* button, stopping the execution of the program.

CONGRATULATIONS!

You mastered your first program for the RA Family of microcontrollers!

Points to take away from this chapter:

- The Project Configurator creates all files and settings necessary for a new project.
- The FSP Configurator allows programmers to configure the FSP and the runtime environment easily based on a graphical user interface.
- A Debug Configuration is necessary for debugging a project. It is created automatically and just needs to be activated.
- Only very few lines of code are necessary for the implementation of the desired functionality.

9 INCLUDING A REAL-TIME OPERATING SYSTEM

What you will learn in this chapter:

- What threads, semaphores and queues are, and how to use them.
- How to add threads and semaphores to a program in e² studio.
- How to toggle an LED with a push-button under RTOS control.

The exercise in the previous chapter already made use of a good part of the Flexible Software Package (FSP) for the RA Family of microcontrollers (MCUs) from Renesas. In this chapter, you will create a small application using the FreeRTOS™ real-time operating system utilizing a thread for the LED and a semaphore for synchronization with a push button. And you will experience firsthand that only few steps are actually necessary for this.

We will create the complete project from scratch, so don't worry if you haven't done the previous labs. And, while this chapter uses FreeRTOS, you can also do this exercise with the Azure® RTOS from Microsoft. The steps to perform are similar.

9.1 Threads, Semaphores, and Queues

Before we dive into actually doing this exercise, let's define some of the terms we will use in this and in the next chapter to make sure that we all have a common understanding.

First, we need to define the term "Thread". If you are more accustomed to the expression "Task", just think of a thread being a kind of task. Some even use both phrases interchangeably. When using a real-time operating system (RTOS), the application running on the microcontroller will be broken down into several smaller, semi-independent chunks of code, with each one typically controlling a single aspect of the program; these small pieces are called threads. Multiple threads can exist within one application, but only one can be active at any given time, as the RA Family microcontrollers are single core devices. Each thread has its own stack space, which, if a secure context is needed, can be placed on the secure side of the MCU. Every thread also has an assigned priority in respect to the other threads in the application, and can be in different states like running, ready, blocked, or suspended. In FreeRTOS™, for example, you can query the state of a thread by calling the `eTaskGetState()` API function. Inter-thread signaling, synchronization, or communication is performed by the means of semaphores, queues, mutexes, notifications, direct-to-task notifications, or stream and message buffers. Azure RTOS provides similar functionalities.

A semaphore is a resource of the RTOS, which can be used for signaling events and for thread synchronization (in a producer-consumer fashion). Using a semaphore allows the application to suspend a thread until the event occurs and the semaphore is posted. Without an RTOS, it would be necessary to constantly poll a flag variable or to create code to perform a certain action inside an interrupt service routine (ISR), blocking other interrupts for quite some time. Using semaphores makes it possible to exit ISRs quickly and to defer the operation to the associated thread.

FreeRTOS and Azure RTOS provide counting semaphores and binary semaphores. While binary semaphores can only assume two values, zero and one, and are therefore ideal for implementing synchronization between tasks or between an interrupt and a task, counting semaphores can have a count between zero and a maximum count specified by the user during the creation of the semaphore in the FSP Configurator. The default there is 256, enabling the designer to perform more complex synchronization operations.

Each semaphore has two basic operations associated: `xSemaphoreTake()` (in FreeRTOS) or `tx_semaphore_get()` (in Azure RTOS), will decrease the semaphore by one, and `xSemaphoreGive()` or `tx_semaphore_set()` increase the

semaphore by one. Both FreeRTOS functions come in two flavors: one that can be called from inside an Interrupt Service Routine (`xSemaphoreTakeFromISR()` and `xSemaphoreGiveFromThread()`), and the ones mentioned before that are callable from the normal context of a thread.

The last term we need to talk about – even if we do not use one in this exercise, but in the next chapter's exercise – is the queue. Message queues are the primary method of inter-thread communication and allow to send messages between tasks or between interrupts and tasks. One or more messages can reside inside a message queue. Data, which can also be a pointer to a larger buffer, is copied into the queue, i.e., it does not store a reference, but the message itself. New messages are normally placed at the end of a queue but can also be sent directly to the front. Received messages are removed from the front.

The allowed message size is specified through the FSP Configurator at design time. The default item size there is 4 bytes and the default queue length, which represents the number of items which can be stored in the queue is 20. All items must be of the same size. There is no limit on the number of queues in FreeRTOS; the only limitation is the memory available in the system. Messages are put into the queue by using the `xQueueSend()` function and read from the queue by `xQueueReceive()`. As with semaphores, there are two versions of the functions: one that can be called from the context of a thread and one that can be called from inside an ISR. In Azure RTOS, the two functions are called `tx_queue_send()` and `tx_queue_receive()`respectively.

9.2 Adding a Thread to FreeRTOS using e² studio

The exercise following is again based on the EK-RA6M4 Evaluation Kit. This time, we will use the blue push button S1 at the upper left-hand side of the board to signal an event to the application, which will toggle the green LED2 in response to it. For the implementation, we will use FreeRTOS and the handling of the event will take place inside a thread which will be notified through a semaphore.

As usual, the first step is to create a new project using the Project Configurator, something you already exercised in chapters 4 and 8. To get started, go to *File → New → Renesas C/C++ Project → Renesas RA*. Highlight the *Renesas RA C/C++ Project* entry in the window showing. Click on *Next* and enter a project name on the screen appearing, for example, *MyRtosProject*. Again, click on *Next*. This gets you to the *Device and Tools Selection* window. First, choose a board. Select the *EK-RA6M4* and set the corresponding device to *R7FA6M4AF3CFB*, if not already listed. Have a look at the toolchain: it should read *GCC Arm® Embedded*. Proceed by clicking on *Next*.

The screen showing now lets you choose between non-TrustZone® and secure and non-secure TrustZone projects. Keep the *Flat (Non-TrustZone) Project* selected and click on *Next*. The *Build Artifact and RTOS Selection* window appears. Leave *Executable* selected under *Build Artifact Selection* and chose *FreeRTOS* under *RTOS Selection*. Move on to the next screen named *Project Template Selection* by clicking on *Next*. Here, select *FreeRTOS – Minimal – Static Allocation*.

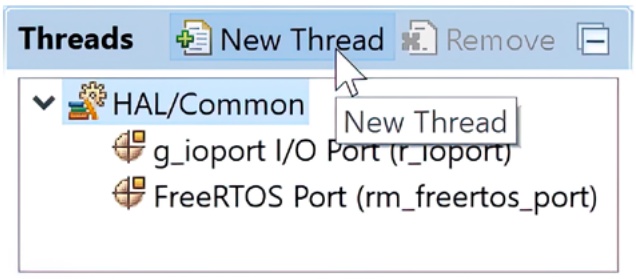


Figure 9-1: After the FSP Configurator shows, only one thread will be shown. Add another one by selecting the "New Thread" button

Finally, click on *Finish* and after the project has been generated by the configurator, e² studio will ask you to switch to the *FSP Configuration perspective*. Once the perspective shows, go directly to the *Stacks* tab. This tab will display two entries for the *HAL/Common* thread in the Threads pane: one containing the drivers for the I/O ports and one for FreeRTOS. Click on the *New Thread* icon at the top of the pane (see *Figure 9-1*) to add a new thread.

Now change the *properties* of the new thread in the Properties view: rename the *Symbol to led_thread* and the *Name* to *LED Thread*. Leave the other properties at their default value. In the *LED Thread Stack* pane, click on the *New Stack* button icon and select *Input → External IRQ Driver (r_icu) as shown in Figure 9-2*.

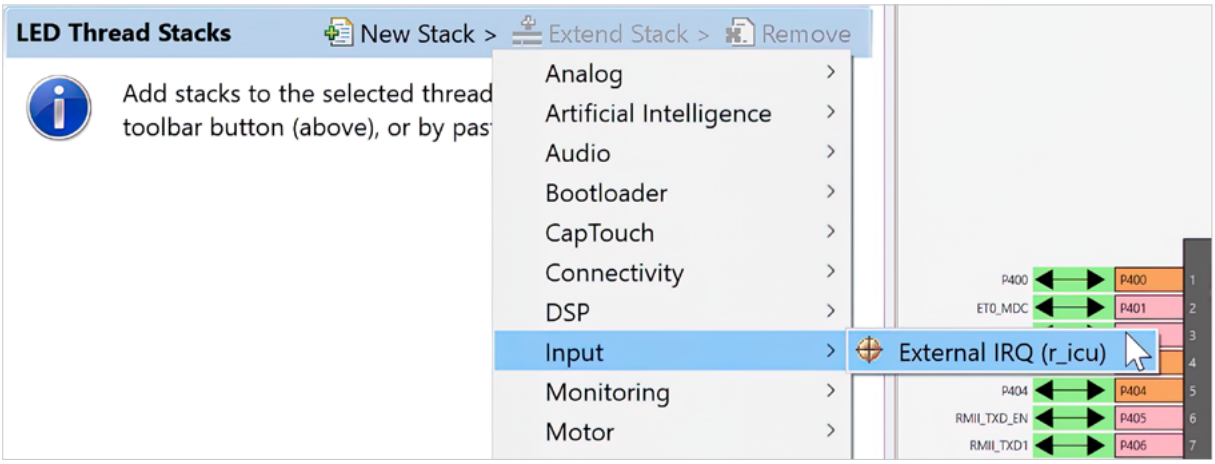


Figure 9-2: Adding a new driver takes only a few mouse clicks

This will add the driver for the external interrupt. Look at the *Properties* for the new driver and make some changes: First, change the *Channel* from *0* to *10*, as the pin where S1 is hooked up to is connected to IRQ10. For the same reason, change the name to *g_external_irq10*, or anything else you like.

The interrupt is assigned priority 12 and it will not be enabled by the FSP during startup. You could choose any other priority as well, but 12 is a good start, as you will rarely run into interrupt priority collisions, even in larger systems. Please note that priority 15 is reserved for the System Tick Timer (systick) and hence should not be used by other interrupts.

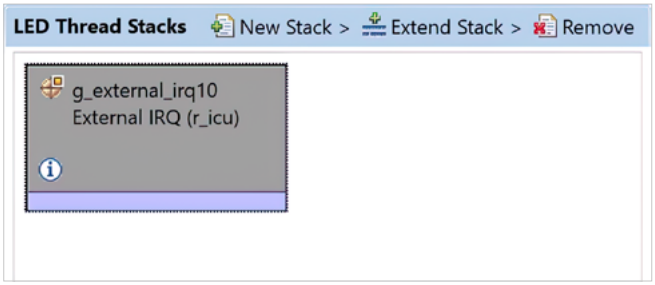


Figure 9-3: The grey color bar of the stack element indicates that this driver is a module instance and that it may be referenced by one other FSP module instance only

Change the *Trigger* from *Rising* to *Falling*, to catch the activation of the button, and the *Digital Filtering* from *Disabled to Enabled*. Keep the setting for the *Digital Filtering Sample Clock* at *PCLK / 64*. This will help to debounce the button. And finally, replace the *NULL* in the *Callback* line with *external_irq10_callback*. This function will be called every time S1 is pressed. We will add the code for the callback function itself later on when creating the application. *Figure 9-4* shows a summary of the necessary settings.

Property	Value
Common	
Parameter Checking	Default (BSP)
Module g_external_irq10 External IRQ (r_icu)	
Name	g_external_irq10
Channel	10
Trigger	Falling
Digital Filtering	Enabled
Digital Filtering Sample Clock (Only valid when Digit	PCLK / 64
Callback	external_irq10_callback
Pin Interrupt Priority	Priority 12
Pins	
IRQ10	P005

Figure 9-4: The properties of the IRQ driver needed for our application

Now, there are only a couple of steps left you need to perform until you can compile and download your program. The next one is to add a semaphore.

For that, click on the *New Object* button in the *LED Thread Objects* pane. If you do not see this pane, but a *HAL/Common Objects* pane, highlight the *LED Thread* in the *Threads* pane and it will become visible. Add a *Binary Semaphore*, which we will need to notify the *LED Thread* once the button is pushed. Change the semaphore's *Symbol* property to *g_s1_semaphore* and leave the *Memory Allocation* at *Static*. Now your *Stacks* tab in the FSP Configurator should look similar to Figure *9-5*.

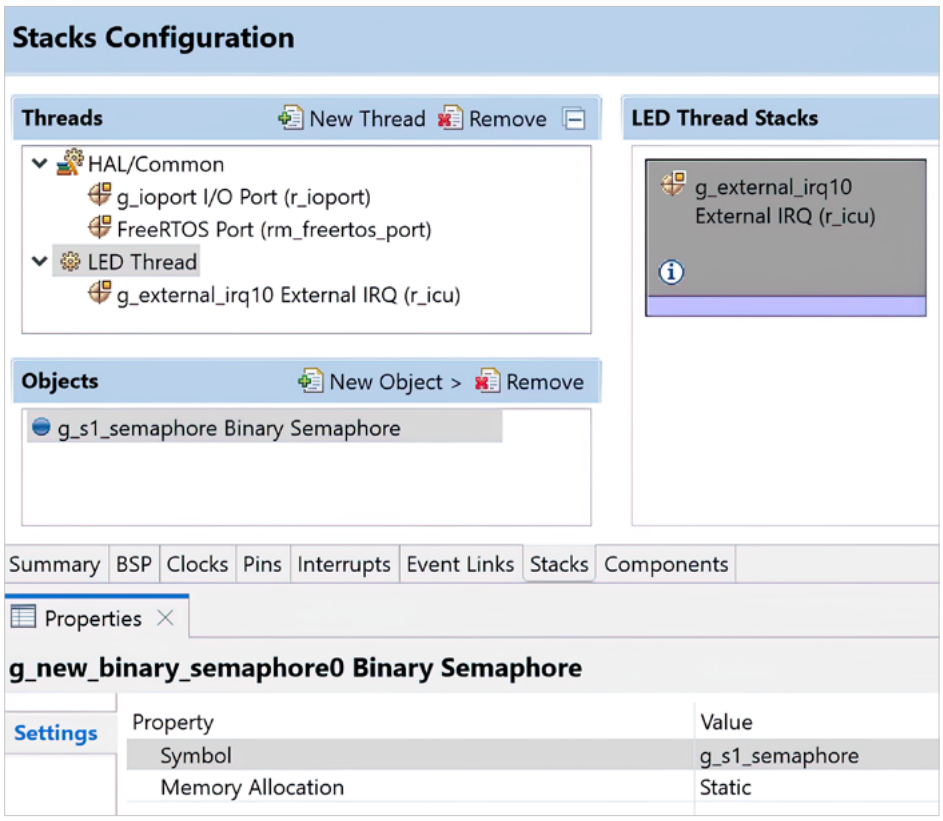


Figure 9-5: This is what the Stacks tab should look like after the LED thread and the semaphore has been added

The final step in the FSP Configurator is to configure the I/O-pin to which S1 is connected to as IRQ10 input. For that, activate the *Pins* tab inside the configurator and expand *Ports* → *P0* and select *P005*. On the RA6M4 Evaluation Kit, this is the port S1 is connected to. In the *Pin Configuration* pane at the right, give it the *Symbolic Name S1* and make sure that the other settings are identical to the one in *Figure 9-6*. Normally, they should have been made for you already by the configurator. Adjust them if not. Note that the package viewer at the right will highlight pin 135 / P005, so you have a graphical reference to the pin's location.

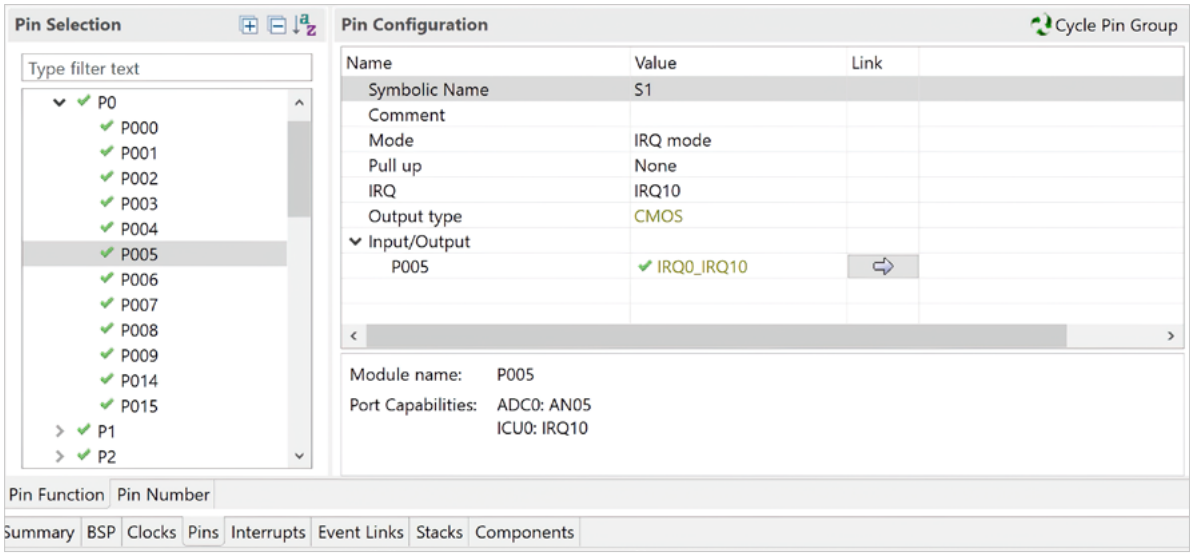


Figure 9-6: Port P005 should already have been configured properly for the IRQ10

With this done, you have finished the settings in the configurator. Save the changes and click on the *Generate Project Content* icon at the top of it to create the necessary files, folders, and settings.

The final task you need to execute is to add the code required for the initialization of the `Leds` structure, write a couple of lines for toggling the LED and to read the semaphore, and to create the callback function, which will set the semaphore. The full code for this can be reviewed at the end of this chapter.

As we are using the *LED Thread* for the handling of the push button and toggling of the LED, we need to add the related code to the *led_thread_entry.c* file this time. Double-click on the file's name in the *Project Explorer* to open it in the editor. If it is not visible, expand the project folder and then the *src* directory. As with the exercise in Chapter 8, add the structure for the LEDs and initialize it. Another variable for the level of the I/O-pin the LED2 is connected to needs to be defined. Name it `led_level`. It needs to be of type `ioport_level_t` and should be initialized to `IOPORT_LEVEL_HIGH` (a "high" level corresponds to "on" on the EK-RA6M4).

The next step will be to open and to enable IRQ10 connected to S1 on the board. For this, use the open and the enable function of the IRQ FSP driver. With that done, initialization is complete.

```
g_external_irq10.p_api->open(g_external_irq10.p_ctrl,
                             g_external_irq10.p_cfg);
g_external_irq10.p_api->enable(g_external_irq10.p_ctrl);
```

Inside the `while (1)` loop, you need to add a couple of statements and to remove the `vTaskDelay(1);` statement. Start with a function call to write the value of the `led_level` to the output register of the I/O-pin for LED2, followed by statements toggling the pin level. There are several ways to do that. Implement your own, review the exercise of Chapter 8, or look up the code at the end of the chapter. Do not forget the Smart Manual feature of e² studio, it helps a lot!

The final statement inside the `while(1)` loop is a call to `xSemaphoreTake()` with the address of the semaphore and the constant `portMAX_DELAY` as parameters. The later one will advise the RTOS to suspend the thread indefinitely until the semaphore is posted from the callback function called by the IRQ 10 interrupt service routine.

And the last thing to do is to add the callback function itself. This function should be as short as possible, as it will be executed in the context of the interrupt service routine. Writing this function is easy: Simply go to *Developer Assistance → LED Thread → g_external_irq10 External IRQ Driver on r_icu* in the Project Explorer and drag and drop the Callback function definition at the end of the list appearing into your source file.

```
void external_irq10_callback(external_irq_callback_args_t *p_args)
```

Inside the callback function, add the following two lines of code:

```
FSP_PARAMETER_NOT_USED(p_args);
xSemaphoreGiveFromISR(g_s1_semaphore, NULL);
```

The macro in the first line will tell the compiler that the callback function does not use the parameter `p_args`, avoiding a warning from the compiler, while the second one sets the semaphore each time button S1 is pressed. Note that the interrupts-save version of the give-function has to be used, as this function call occurs inside the context of the ISR. The second parameter of this call is `*pxHigherPriorityTaskWoken`. If it is possible that a semaphore will have one or more tasks blocked and waiting for the semaphore to become available, and one of these tasks has a higher priority than the one executed when the interrupt has occurred, this parameter will become true after the call to `xSemaphoreGiveFromISR()`. In that case, a context switch should be performed before the interrupt is exited. As in our example no other task depends on this semaphore, we can set this parameter to `NULL`.

Once all the coding is complete, build your project by clicking on the *Build* icon (the "hammer"). If it does not compile with zero errors, go back to your program and fix the problems with the help from the feedback of the compiler being displayed in the *Problems* view.

If the project built successfully, click on the small arrow beside the *Debug* icon, select *Debug Configurations* and expand *Renesas GDB Hardware Debugging*. Select *MyRtosProject Debug_Flat* or the name you gave your project and click on *Debug*. This will start the debugger. If you need more information on that, please review the related section in Chapter 8. Once the debugger is up and running, click on *Resume* twice. Your program is now executing, and each time you press S1 on the EK, the green LED2 should toggle.

One last note: In a real-world application, error checking should be performed to ensure proper operation of the program. For the sake of clarity and brevity, this was omitted in our example.

```
#include "led_thread.h"

 void led_thread_entry(void *pvParameters)
 {
    FSP_PARAMETER_NOT_USED (pvParameters);
    extern bsp_leds_t g_bsp_leds;
    bsp_leds_t Leds = g_bsp_leds;

    uint8_t led_level = BSP_IO_LEVEL_HIGH;

    g_external_irq10.p_api->open(g_external_irq10.p_ctrl,
                                 g_external_irq10.p_cfg);
    g_external_irq10.p_api->enable(g_external_irq10.p_ctrl);

    while (1)
    {
        g_ioport.p_api->pinWrite(&g_ioport_ctrl
                                 Leds.p_leds[BSP_LED_LED2],led_level);
        if (led_level == BSP_IO_LEVEL_HIGH)
        {
            led_level = BSP_IO_LEVEL_LOW;
        }
        else
        {
            led_level = BSP_IO_LEVEL_HIGH;
        }

        xSemaphoreTake(g_s1_semaphore, portMAX_DELAY);
    }
 }

 /* callback function for the S1 push button; sets the semaphore */
 void external_irq10_callback(external_irq_callback_args_t *  p_args)
 {
    FSP_PARAMETER_NOT_USED(p_args);
    xSemaphoreGiveFromISR(g_s1_semaphore, NULL);
 }
```

CONGRATULATIONS!

You successfully finished this exercise.

Points to take away from this chapter:

- Using the functions of the FSP is easy through the use of comprehensive APIs.
- The FSP will take care of most non-user code related things.
- Using FreeRTOS™ is straightforward and adding threads and semaphores is not much of an effort, as the FSP Configurator is intuitive to use.

10 SENDING DATA THROUGH USB USING THE FLEXIBLE SOFTWARE PACKAGE

What you will learn in this chapter:

- How to set up an USB-transfer using the middleware of the Flexible Software Package for the RA Family of microcontrollers.
- How to receive data sent by the MCU on a host workstation.

In this part of the book, we will use the USB middleware of the Flexible Software Package (FSP) for the Renesas RA Family of microcontrollers to send the current state of LED2 as text string through a USB port to your Windows® workstation every time you press user button S1. Contrary to Chapter 9, we will not use a real-time operating system and a semaphore in this lab, but a global variable to indicate that the push switch was activated and the state of the green LED2 has changed.

The update of the LED state (ON or OFF) and the write to the USB port will be done in the callback routine of IRQ10, as well as the update of the global variable holding the information that the button was pressed. The write to the port will trigger a USB transfer, sending the information about the LED to the host. Once returned to the infinite loop inside the `hal_entry()` function, the events of the USB will be handled, and the next update of the LED state prepared by setting the global variable to false and assigning the next string and the next LED level to their respective variables. *Figure 10-1* details the flow of the program and of the callback function for the interrupt.

Most of the setup of the port will be done graphically in the FSP Configurator, so only a few lines of code have to be written by the application programmer. While programming this exercise, you will experience once again the ease of use the FSP provides to users, even when building up complex communication systems like USB.

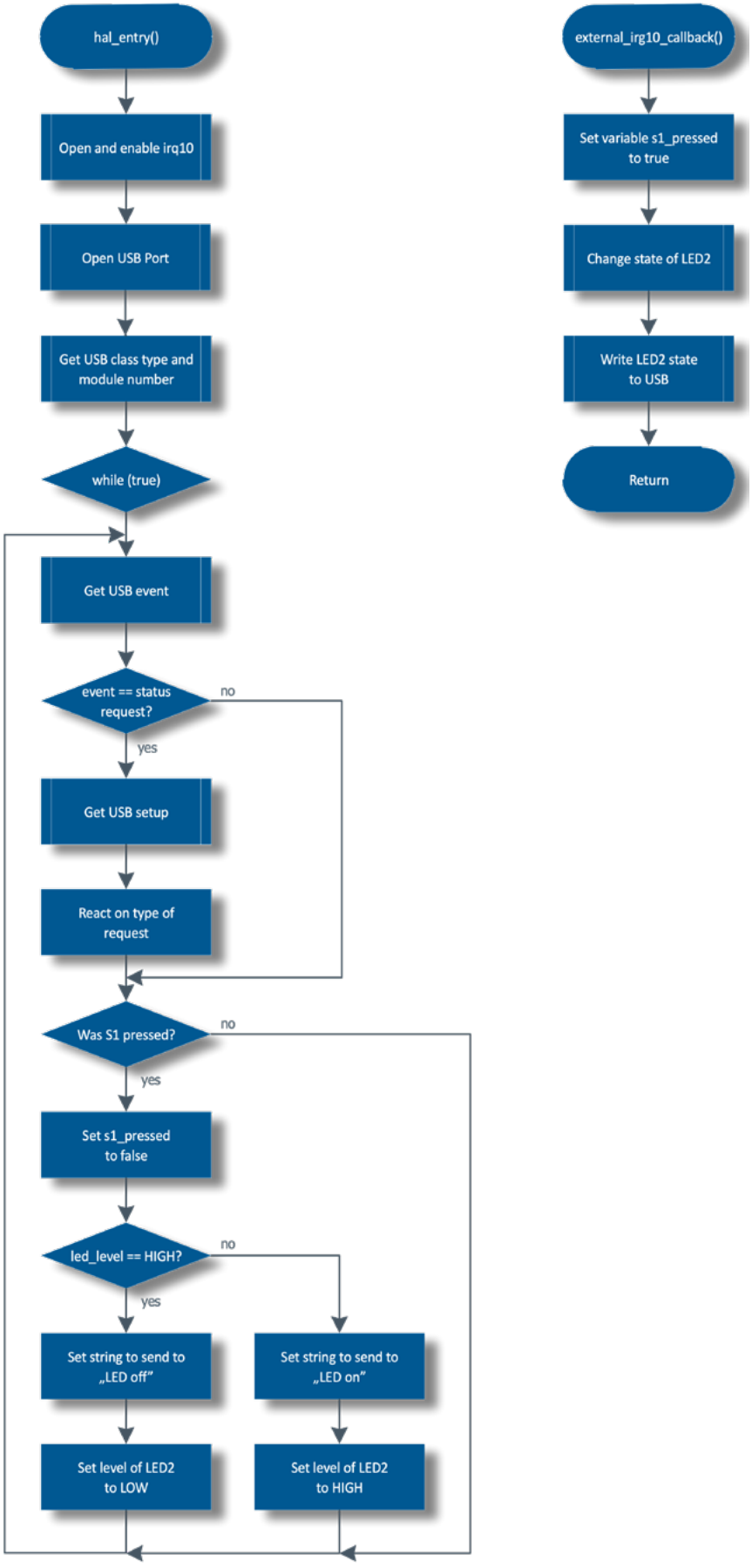


Figure10-1: Flowchart of this chapter's exercise

10.1 Setting Up the USB Port with the FSP Configurator

If you closed e² studio after the last exercise, re-open it and create a new project. As you are an RA expert by now, I will not describe every step in detail, as most of the tasks to be performed have already been described in the previous labs. Name the new project *MyUSBProject*, select the *EK-RA6M4* as board once you are at the *Device and Tools Selection* screen, as we will again use this Evaluation Kit for the lab. On the *Project Type Selection* page, keep the *Flat (Non-TrustZone) Project* enabled and make sure that the *No RTOS* entry is activated under *RTOS Selection*. Finally, select *Bare Metal – Minimal* on the *Project Template Selection* page before clicking on *Finish*.

After the Project Configurator has created the project and the FSP Configurator shows, go directly to the *Stacks* tab. First, we need to add the module for the external interrupt connected to the user push button S1. Click on *New Stack* on the *HAL/Common Stacks* pane and select *Input → External IRQ (r_icu)*.

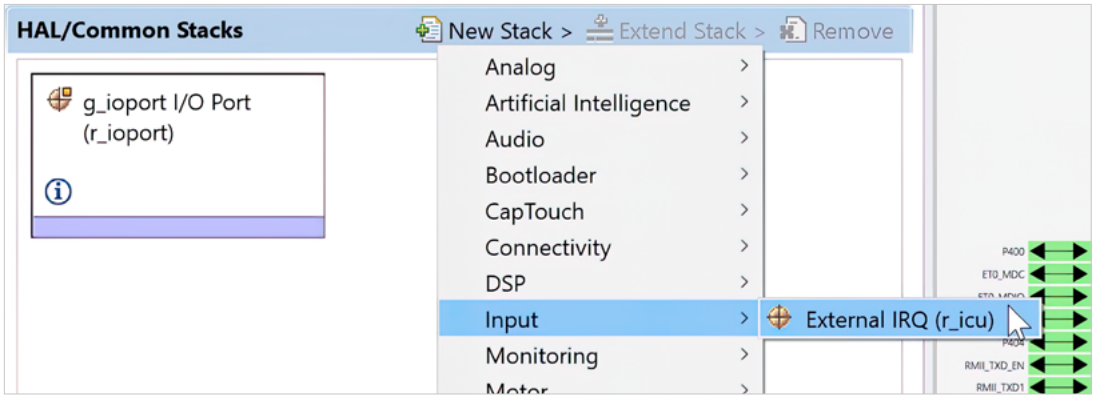


Figure 10-2: First add the driver for the S1 interrupt

In the Properties view, modify the Name of the interrupt to *g_external_irq10* and its *Channel* to *10*, as this is the channel used by the interrupt. Enable the *Digital Filtering* and set the *Trigger* to *Falling*. This helps to debounce the switch. Finally, you need to provide the name of the callback function for this interrupt: name it *external_irq10_callback* and change the *Priority* to *14*, as we want to make sure that the USB interrupts have priority over the push button (see *Figure 10-3*).

Property	Value
▾ Common	
Parameter Checking	Default (BSP)
▾ Module g_external_irq10 External IRQ (r_icu)	
Name	g_external_irq10
Channel	10
Trigger	Falling
Digital Filtering	Enabled
Digital Filtering Sample Clock (Only valid when Digit	PCLK / 64
Callback	external_irq10_callback
Pin Interrupt Priority	Priority 14
▾ Pins	
IRQ10	P005

Figure 10-3: These are the settings necessary for IRQ10

Next, add the middleware for the USB peripheral communications device class (PCDC) to the system: create a new stack and select *Connectivity → USB PCDC (r_usb_pcdc)* as shown in *Figure 10-4*.

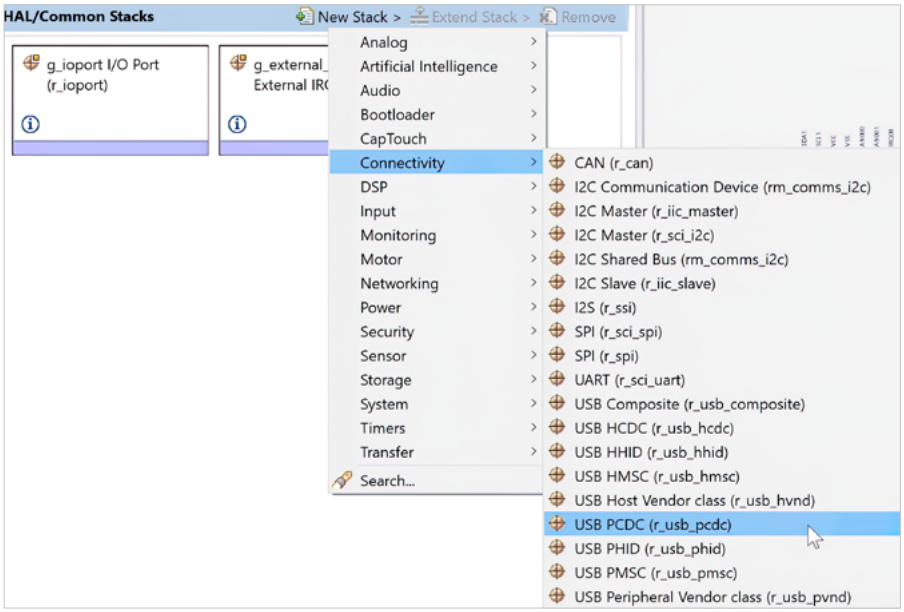


Figure 10-4: The middleware for the USB peripheral communications device class driver needs to be added to the system

This will, in fact, add four modules to the project: the actual PCDC driver for the full speed USB port, which implements the USB PCDC interface for the application level, and the basic USB driver. The stack also shows two modules with a pink color bar. These allow you to add optional drivers for the DMAC (Direct Memory Access Controller) for transmission or reception. As we will send the status messages directly using a USB write API function, there is no need to add them. You might wonder what the meaning of the other color bars of the modules are. It is quite simple: Gray marks module instances which may be referenced by only one other module instance, and blue marks common module instances that can be referenced by multiple other module instances – even across multiple stacks. And the small triangles in the color bars let you expand or collapse the modules tree.

By implementing the USB port as a PCDC device, we can use the USB as a virtual COM port. This simplifies the setup of the receiver on the host side, because after it registers in Windows®, communication with the port is possible using a terminal application. This is how we want to talk with the Evaluation Kit.

With all stacks added, the *HAL/Common Stacks* pane will now look like *Figure 10-5*:

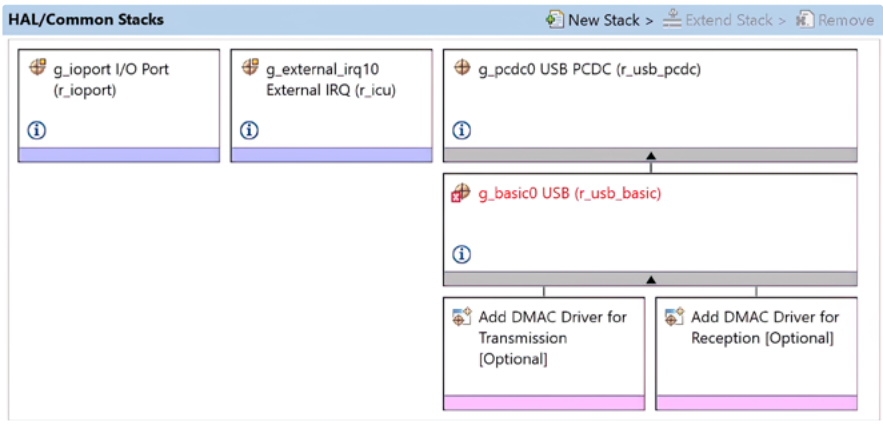


Figure 10-5: The Stacks pane should look like this after adding the USB driver

No changes are needed in the *Properties* of the basic USB driver. Just make sure that the *DMA Support* is set to *Disabled*. Note the name of the *USB Descriptor* in the *g_basic0 USB Driver (r_usb_basic)* section: *g_usb_descriptor*. A structure with this name will be created later on to describe the features of the USB to the system, so you should keep this name in mind. *Figure 10-6* shows the properties of the module.

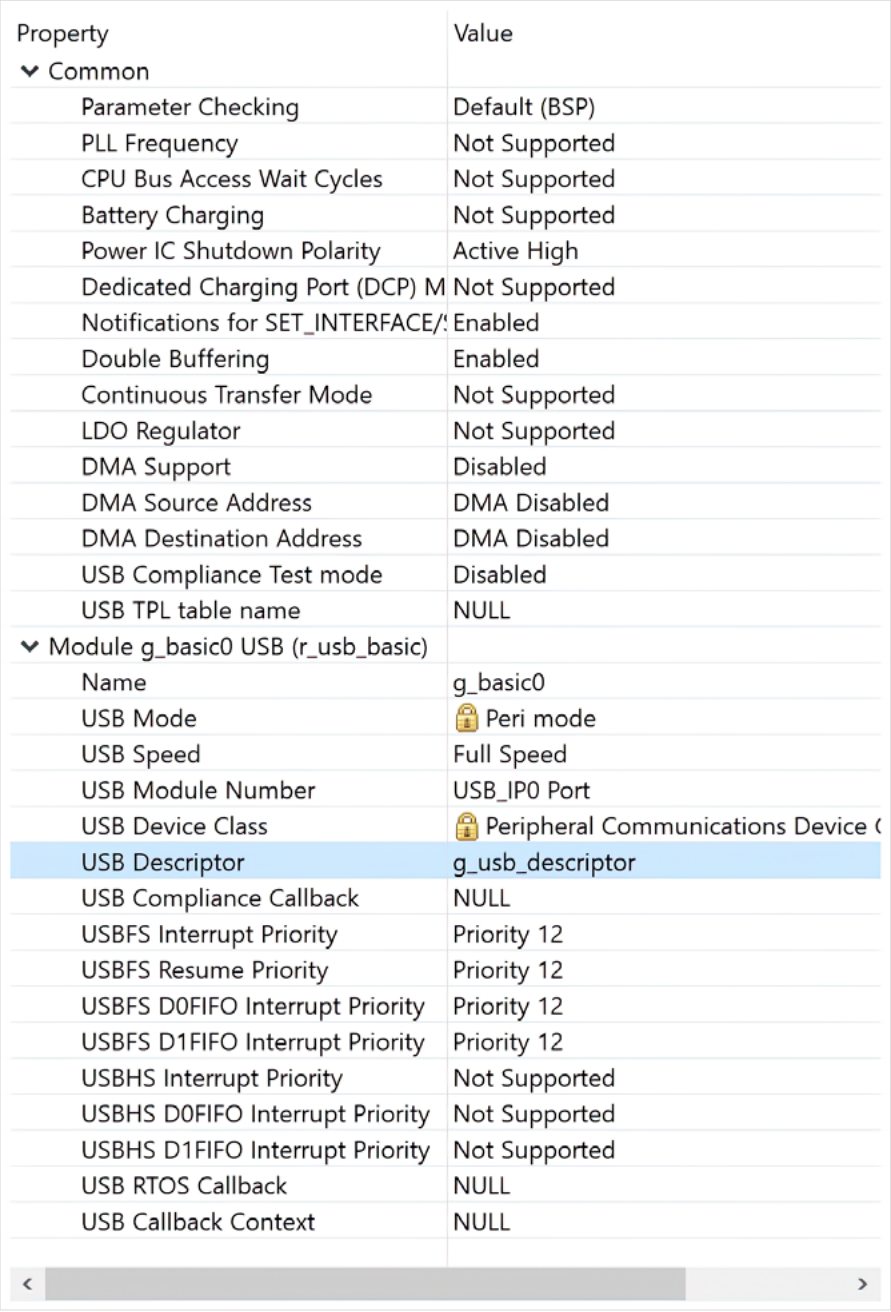


Figure 10-6: The settings for the basic USB driver. Note the name of the USB Descriptor

You will notice that the module *g_basic0 USB (r_usb_basic)* is shown in red. Hovering with the mouse over it reveals the explanation: the USB requires a 48 MHz clock. We can correct this is the *Clocks* tab of the configurator, but before we do this, we will first set the correct operation mode for the USB port.

To do this, switch to the *Pins* tab and expand first the *Peripherals* drop-down list and then the *Connectivity: USB* list in the *Pin Selection* pane. In the *Pin Configuration* pane, modify the *Operation Mode* from *Custom* to *Device*, as this is the mode we want to use. Note that the input/output pin assignments change accordingly.

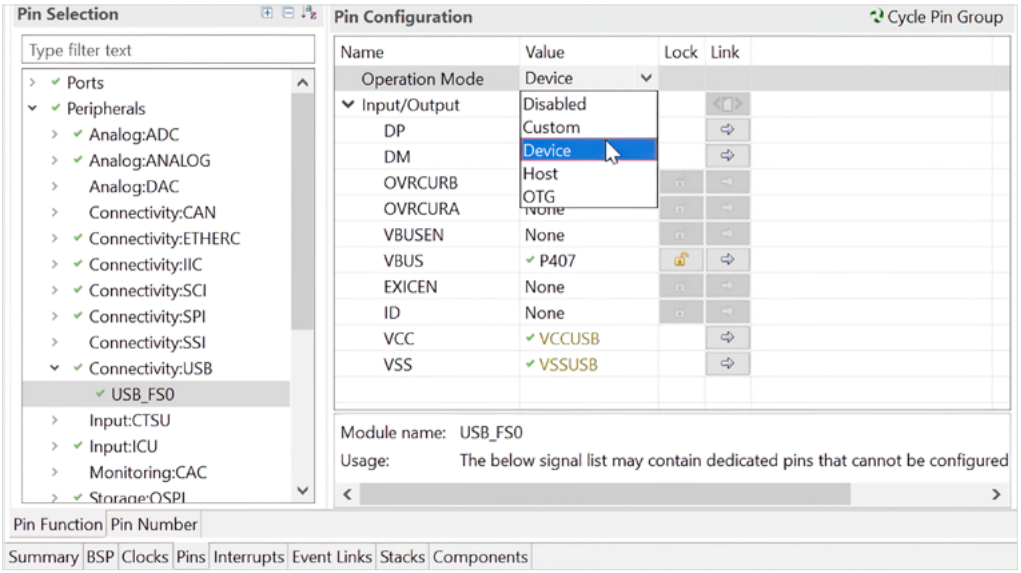


Figure 10-7: Our USB port will use the device mode, so change it accordingly

Now we are almost done with setting up the port. The last step is to enable the *USB clock (UCLK)*, which is the operating clock for the USB full speed (FS) module, and set it to the required frequency of 48 MHz. For this, activate the *Clocks* tab, which allows you to configure the clock generation circuit. First, enable the USB clock by changing the *UCLK* near the bottom of the pane from *Disabled* to *Enabled* with *PLL2* as the source. Next, enable *PLL2*, which is the PLL dedicated especially to the USB module, with the *high-speed on-chip oscillator (HOCO)* as source. You will notice that the *UCLK frequency* is now given as being 40 MHz and is highlighted in red, as the USB requires a frequency of 48 MHz. Change the *multiplier value* for the *PLL2* to *24*, which will result in a PLL2 frequency of 240 MHz. Together with the UCLK divider of 5, the correct UCLK frequency of 48 MHz is now set. Here, a huge advantage of the RA MCU Groups with an Arm® Cortex®-M33 core becomes evident: there is a second PLL available on the microcontroller, which allows to clock the USB with 48 MHz and to run the MCU at maximum speed at the same time.

As a last modification, change the source for the standard PLL from XTAL to HOCO as well. Refer to *Figure 10-8*, if you are unsure which fields to change.

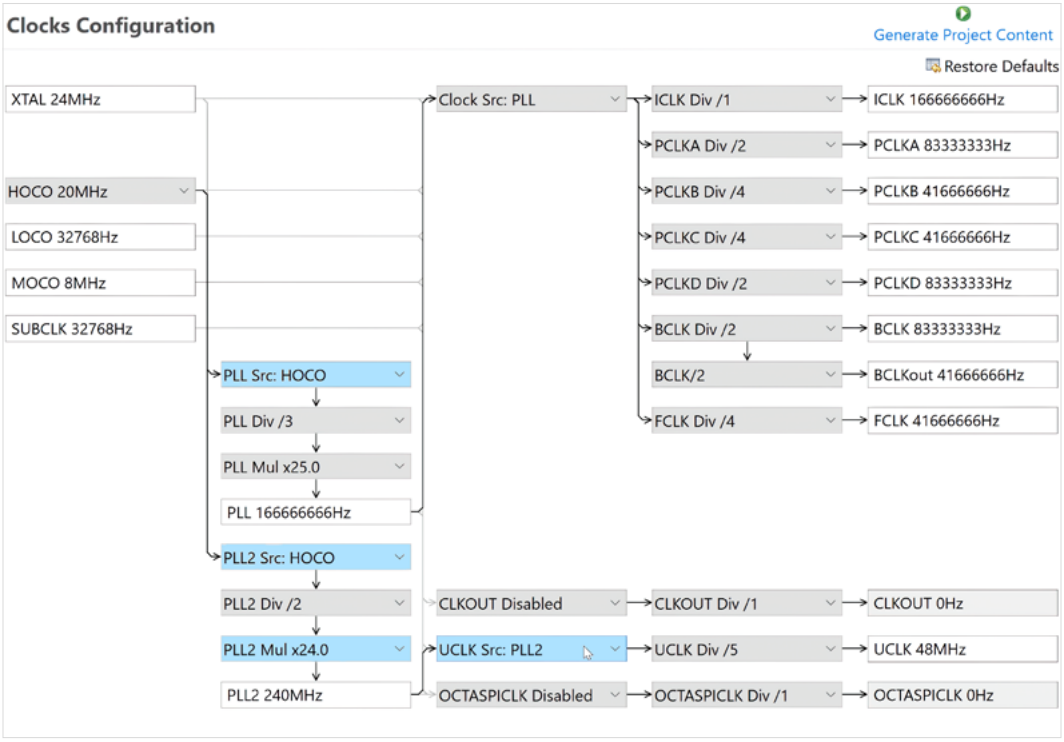


Figure 10-8: The USB FS needs a 48 MHz clock, so the clock generation circuit needs to be changed accordingly. Necessary changes are highlighted

Note that with these changes, the error of the *g_basic0 USB* driver went away, so the Stacks tab should now look like *Figure 10-9*.

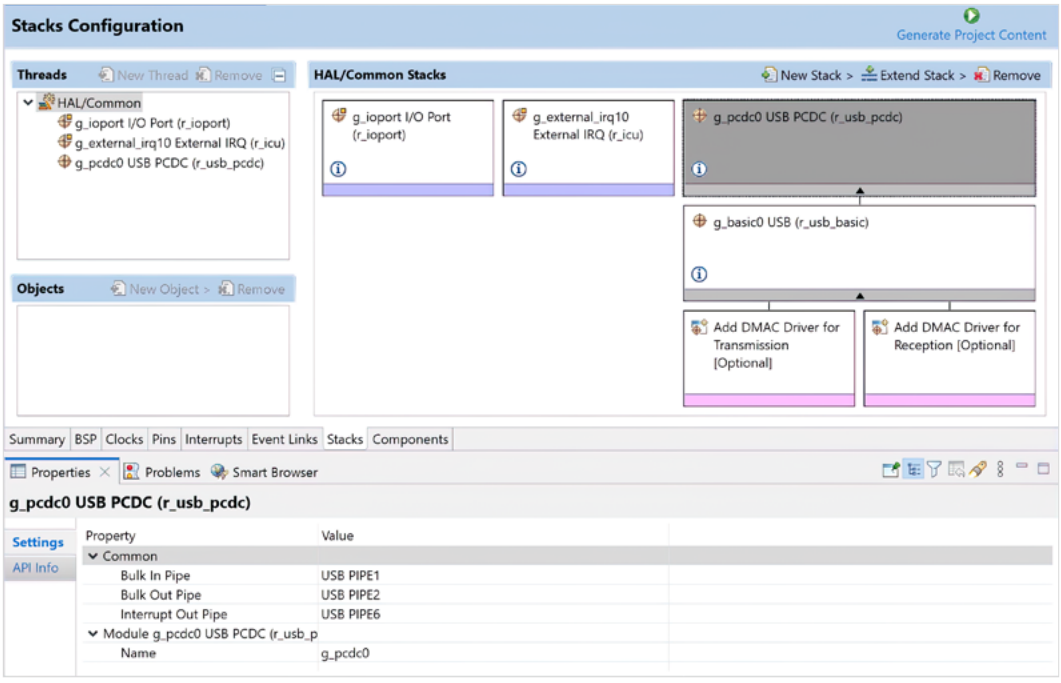


Figure 10-9: With all additions and modifications made, the Stacks tab should look like this

This concludes the settings which had to be made in the FSP Configurator. Save the configuration and click on the *Generate Project Content* button at the upper right-hand corner of the screen to extract the files and to create the required settings. As a last step, switch back to the C/C++ perspective.

10.2 Creating the Code

Now it's time to add the code required for the initialization of the USB port and for performing the write to it. As there is a lot of typing involved with this exercise, you might want to download the solution for this lab from the book's webpage on the Renesas Internet site and just follow along the explanations without doing the actual coding.

If you decide to write everything from scratch, start by opening the file *hal_entry.c* by double-clicking on it in the *Project Explorer*. For our program to work properly, we need a couple of global variables. First, declare an enumeration for the USB driver status right before the head of the *hal_entry()* function. It should be of the type `usb_status_t` and you might name it `usb_event`. Next, add a structure of the type `usb_setup_t` – which is declared in *r_usb_basic_api.h* – and name it `usb_setup`. We will use this variable later on while decoding some of the USB events and it will be initialized inside the USB event loop.

Next, we require a variable to hold the USB module number. Make it of type `uint8_t`, name it `g_usb_module_number` and assign the value "`0x00`" to it. Finally, declare a structure for the USB class type of the type `usb_class_t`, name it `g_usb_class_type` and also assign the value "`0x00`" to it. If you want to have more information on the various types we use, please refer to the Renesas Flexible Software Package (FSP) User's Manual, which can be downloaded from the FSP's GitHub® site.

With these additions done, this part of the code should now look like that:

```
/* Global variables for the USB */
usb_status_t usb_event;
usb_setup_t usb_setup;

uint8_t g_usb_module_number = 0x00;
usb_class_t g_usb_class_type = 0x00;
```

Our own code depends on some static global variables as well. Please add them just right below the ones for the USB:

```
/* Global variables for the program */
static char send_str[12] = { "LED on\n\r" };
static volatile uint8_t s1_pressed = false;
static uint8_t led_level = BSP_IO_LEVEL_HIGH;
```

The array of chars named `send_str` is used to hold the text we want to send over the USB. Initialize it to "`LED on\n\r`", as this variable will be utilized for the first time once we switch LED2 to ON. The next variable, `s1_pressed`, is of type `uint8_t` and needs to be declared as `volatile`, as its value will be changed to `true` inside the callback routine for the user button S1. It is `false` by default and will be set to `true` by the callback routine of the IRQ10 interrupt, signaling that the button was pressed and therefore notifying the main program that the event occurred.

If this variable is not declared as `volatile`, the optimizer of the C-compiler might not re-read its value each time the variable is used and the loop inside `hal_entry()` might therefore not notice a change.

The third variable holds the level of LED2 and should be initialized to `BSP_IO_LEVEL_HIGH` at startup. Its value toggles every time S1 is activated.

With that, we have declared all global variables, and we can continue by writing the code inside the `hal_entry()` function. First of all, we need a static variable which will hold the settings for the virtual UART communication port like the bitrate, the number of stop and data bits, and the parity type. This variable should be of the type `usb_pcdc_linecoding_t` and you might want to name it `g_line_coding`. Replace the "add your own code here" placeholder with the declaration. This variable will later on be initialized inside the USB event handler loop.

```
static usb_pcdc_linecoding_t g_line_coding;
```

Next, write the code to open and to enable the external IRQ10, which is connected to S1 on the Evaluation Kit. As in Chapter 9, use the respective functions of the IRQ FSP driver:

```
g_external_irq10.p_api->open (g_external_irq10.p_ctrl,
                              g_external_irq10.p_cfg);
g_external_irq10.p_api->enable (g_external_irq10.p_ctrl);
```

With the interrupts enabled, the USB needs to be opened and the class type and module number acquired. Use the relevant functions of the *g_basic0 USB Driver on r_usb_basic module* for that and pass the control structures to these functions, as well as the references to the configuration structure (for the `Open()` function) and the respective variables. Remember that both the code completion feature and the *Developer Assistance* of e² studio are there to help you to write these lines.

```
R_USB_Open (&g_basic0_ctrl, &g_basic0_cfg);
R_USB_ClassTypeGet (&g_basic0_ctrl, &g_usb_class_type);
R_USB_ModuleNumberGet (&g_basic0_ctrl, &g_usb_module_number);
```

The initialization of the interrupts and the USB port is now complete. All code coming next should be placed inside a `while (1)` loop, as this part of the program will be executed indefinitely. First, we will write the code for obtaining and handling the USB related events of the port. There are several events associated with the USB driver, but, for the sake of brevity, we will only handle the `USB_STATUS_REQUEST` event. If you are interested in a more complete event handler, have a look into the documentation of the USB Peripheral Communications Device Class (r_usb_pcdc) in the Flexible Software Package (FSP) User's Manual. There you will find a code example for such a handler, as well as a flowchart.

Your first task now is to initialize the `usb_event` variable through a call to the `R_USB_EventGet()` function and then to write the handler, which only should be executed if the `USB_STATUS_REQUEST` event had happened. Inside the if – then – else construct, first get the setup of the USB port and then decide if the line settings were requested. If yes, configure the virtual UART settings by passing the `g_line_coding` variable.

If not, query if the host wants to receive the UART settings. If yes, send them to the host. In the end, if an event was issued we do not handle here, just acknowledge it.

Here is the complete code for our version of the handler:

```
/* Obtain USB related events */
 R_USB_EventGet (&g_basic0_ctrl, &usb_event);

/* USB event received by R_USB_EventGet */
if (usb_event == USB_STATUS_REQUEST)
{
    R_USB_SetupGet (&g_basic0_ctrl, &usb_setup);

    if (USB_PCDC_SET_LINE_CODING == (usb_setup.request_type
                                     & USB_BREQUEST))
    {
        /* Configure virtual UART settings */
        R_USB_PeriControlDataGet (&g_basic0_ctrl,
                          (uint8_t*) &g_line_coding, LINE_CODING_LENGTH);
    }
    else if (USB_PCDC_GET_LINE_CODING == (usb_setup.request_type
                                          & USB_BREQUEST))
    {
        /* Send virtual UART settings back to host */
        R_USB_PeriControlDataSet (&g_basic0_ctrl,
                                  (uint8_t*) &g_line_coding,
                                  LINE_CODING_LENGTH);

    }
    else if (USB_PCDC_SET_CONTROL_LINE_STATE == (usb_setup.request_type
                                                 & USB_BREQUEST))
    {
        /* Acknowledge all other status requests */
        R_USB_PeriControlStatusSet (&g_basic0_ctrl, USB_SETUP_STATUS_ACK);
    }
    else
    {
    }
}
```

You will notice that we use the name LINE_CODING_LENGTH twice inside the handler. As we haven't defined its value yet, please go back to the top of the file and define LINE_CODING_LENGTH with an unsigned value of 0x07.

```
#define LINE_CODING_LENGTH          (0x07U)
```

Back inside the `while (1)` loop, add the code for changing the level of LED2 once S1 has been activated, as indicated by a `true` value of `s1_pressed`. This is similar to what you had written in Chapter 9, but this time, you will also need to copy the string to be sent through the USB into the `send_str` variable and to set the `s1_pressed` variable to false:

```
if (s1_pressed == true)
{
    s1_pressed = false;

    if (led_level == BSP_IO_LEVEL_HIGH)
    {
        strcpy (send_str, "LED off\n\r");
        led_level = BSP_IO_LEVEL_LOW;
    }

else
    {
        strcpy (send_str, "LED on\n\r");
        led_level = BSP_IO_LEVEL_HIGH;
    }
}
```

The last code to add is the one for the callback function for the external IRQ10. Place it just after the closing bracket of the `hal_entry ()` function. You might want to review Chapter 9 for some of the details of the callback. First, you need to import the `g_bsp_leds` structure and initialize our local `Leds` variable with it. Then set `s1_pressed` to true to signal that the event happened, and next write the new value to the pin register. Finally, send the string through the USB port by using the `R_USB_Write()` API of the *r_usb_basic* module.

```
void external_irq10_callback(external_irq_callback_
args_t *p_args)
{
    FSP_PARAMETER_NOT_USED(p_args);
    extern bsp_leds_t g_bsp_leds;
    bsp_leds_t Leds = g_bsp_leds;

    s1_pressed = true;
    g_ioport.p_api->pinWrite (&g_ioport_ctrl,
                              Leds.p_leds[BSP_LED_LED2],
                               led_level);

    R_USB_Write (&g_basic0_ctrl,
                 (uint8_t*) send_str,
                 ((uint32_t) strlen (send_str)),
                 (uint8_t) g_usb_class_type);
}
```

Remember the USB Descriptor named `g_usb_descriptor`? It is now time to take care about that one. The USB needs an exact description of the device, its configuration, and the vendor information. This file is quite complex and with 484 lines of code quite long. An explanation of it is available in the *r_usb_basic* section of the FSP User's Manual, as well as in the Universal Serial Bus Revision 2.0 Specification (**http://www.usb.org/developers/docs/**), which gives detailed instructions on how to construct this descriptor.

But there are two shortcuts for you: One is to download the source files for the exercises of this book from the book's website (insert link here). The other one is to use the template the FSP Configurator placed into the *ra* directory of the project. It is named *r_usb_pcdc_descriptor.c.template* and you can access it by going to the *ra* → *fsp* → *src* → *r_usb_pcdc* folder in the project explorer (see *Figure 10-10*). Copy this file into the *src* folder where *hal_entry.c* resides and rename it to *r_usb_descriptor.c*. Modify the vendor *ID* and the *product ID* to match your own the IDs of your own products. If you don't have them yet, use temporarily the values `0x045BU` and `0x5310U`. This actually concludes the settings to be made and the code to be written.

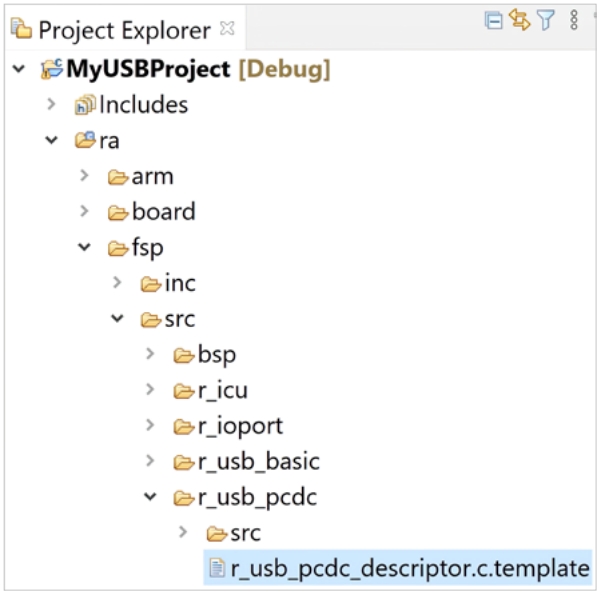


Figure 10-10: The FSP Configurator automatically created a template for the USB descriptor

All what is left, is to build the project. The first time you do that, it will take some time, as the code for all the FSP modules included in the project will need to be compiled as well. As soon as the project has built with zero errors and zero warnings, connect the EK-RA6M4 Evaluation Kit and initiate the debug session. With the *Debug Perspective* open, click on *Resume twice* to start the program. As a quick test, press S1 once to see if LED2 toggles.

10.3 Setting Up a Receiver on the Host Side

With the program running, connect a second USB type A to micro-B cable to the USB port labeled J11 at the lower left-hand side of the *System Control & Ecosystem Access* area of the Evaluation Kit. Insert the other end into your Windows® workstation and wait a couple of seconds until Windows® recognizes the board, enumerates it, and installs the drivers.

Start a terminal emulator program. During the development of this exercise, I used Tera Term, which can be downloaded from **https://ttssh2.osdn.jp/** and I found it quite useful. In Tera Term, you will see the CDC serial port listed. In *Figure 10-11* it is COM3,

but it is likely to be different on other PCs. If you are not sure, use the Device Manager of Windows® to find out the port the board is connected to.

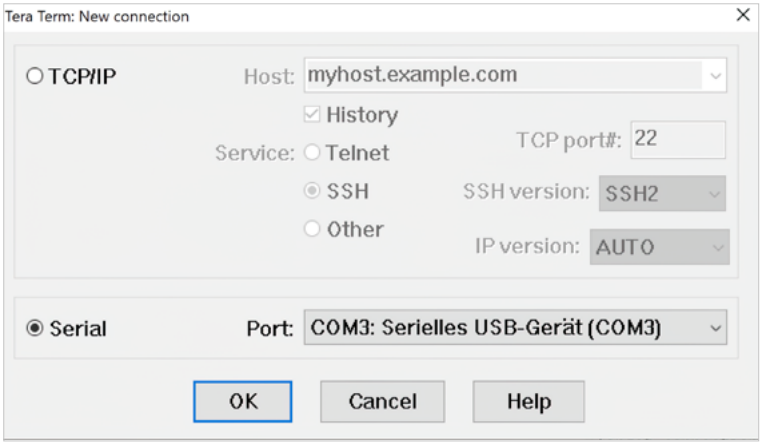


Figure 10-11: If Windows® recognized the board correctly, it will be listed in Tera Term as serial connection

In case the board is not listed at all or the Device Manager indicates an error, there might be a problem with the driver. Please refer to the latest support entry for this topic in the Renesas Knowledge Base to resolve this: **https://en-support.renesas.com/knowledgeBase/18959077**.

With the connection made and Tera Term running, press S1 a couple of times and you should see the green LED2 toggling and the state of it output to the terminal as shown in Figure *10-12*.

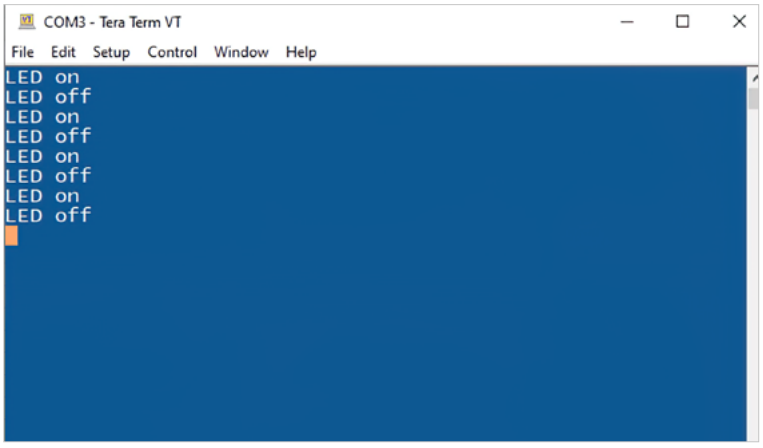


Figure 10-12: With the transfer running, the terminal program will display the state of LED2 each time S1 is pressed

```
#include „hal_data.h“

FSP_CPP_HEADER
void R_BSP_WarmStart(bsp_warm_start_event_t event);
FSP_CPP_FOOTER

#define LINE_CODING_LENGTH          (0x07U)

/* Global variables for the USB*/
usb_status_t usb_event;
usb_setup_t usb_setup;
uint8_t g_usb_module_number = 0x00;
usb_class_t g_usb_class_type = 0x00;

/* Global variables for the program */
static char send_str[12] = { „LED on\n\r“ };
static volatile uint8_t s1_pressed = false;
static uint8_t led_level = BSP_IO_LEVEL_HIGH;

/*******************************************************************************************/
/* main() is generated by the RA Configuration editor and is used to generate threads if  */
/* an RTOS is used.  This function is called by main() when no RTOS is used.              */
/*******************************************************************************************/
void hal_entry(void) {
    static usb_pcdc_linecoding_t g_line_coding;

    /* open and enable irq10 */
    g_external_irq10.p_api->open (g_external_irq10.p_ctrl, g_external_irq10.p_cfg);
    g_external_irq10.p_api->enable (g_external_irq10.p_ctrl);

    /* Open USB instance */
    R_USB_Open (&g_basic0_ctrl, &g_basic0_cfg);

    /* Get USB class type */
    R_USB_ClassTypeGet (&g_basic0_ctrl, &g_usb_class_type);

    /* Get module number */
    R_USB_ModuleNumberGet (&g_basic0_ctrl, &g_usb_module_number);

    while (true)
    {
        /* Obtain USB related events */
        R_USB_EventGet (&g_basic0_ctrl, &usb_event);

        /* USB event received by R_USB_EventGet */
        if (usb_event == USB_STATUS_REQUEST)
        {
            R_USB_SetupGet (&g_basic0_ctrl, &usb_setup);

            if (USB_PCDC_SET_LINE_CODING == (usb_setup.request_type & USB_BREQUEST))
            {
                /* Configure virtual UART settings */
                R_USB_PeriControlDataGet (&g_basic0_ctrl,
                                          (uint8_t*) &g_line_coding, LINE_CODING_LENGTH);
            }
            else if (USB_PCDC_GET_LINE_CODING == (usb_setup.request_type & USB_BREQUEST))
            {
                /* Send virtual UART settings back to host */
```

```
                R_USB_PeriControlDataSet (&g_basic0_ctrl,
                                          (uint8_t*) &g_line_coding, LINE_CODING_LENGTH);

            }
            else if (USB_PCDC_SET_CONTROL_LINE_STATE == (usb_setup.request_type
                                                         & USB_BREQUEST))
            {
                 /* Acknowldedge all other status requests */
                 R_USB_PeriControlStatusSet (&g_basic0_ctrl, USB_SETUP_STATUS_ACK);
            }
            else {
            }
        }

        if (s1_pressed == true)
        {
            s1_pressed = false;

            if (led_level == BSP_IO_LEVEL_HIGH)
            {
                 strcpy (send_str, „LED off\n\r“);
                 led_level = BSP_IO_LEVEL_LOW;
            }

            else
            {
                strcpy (send_str, „LED on\n\r“);
                led_level = BSP_IO_LEVEL_HIGH;
            }
        }
    }

#if BSP_TZ_SECURE_BUILD
    /* Enter non-secure code */
    R_BSP_NonSecureEnter();
#endif
}

/*****************************************************************************************/
/* callback function for the S1 push button; writes the new level to LED2 and sends it */
/* through the USB to the PC                                                             */
/*****************************************************************************************/
void external_irq10_callback(external_irq_callback_args_t *p_args)
{
    /* Not currently using p_args */
    FSP_PARAMETER_NOT_USED(p_args);
    extern bsp_leds_t g_bsp_leds;
    bsp_leds_t Leds = g_bsp_leds;

    s1_pressed = true;
    g_ioport.p_api->pinWrite (&g_ioport_ctrl, Leds.p_leds[BSP_LED_LED2], led_level);

    R_USB_Write (&g_basic0_ctrl,
                 (uint8_t*) send_str,
                 ((uint32_t) strlen (send_str)),
                 (uint8_t) g_usb_class_type);
}
```

CONGRATULATIONS!

You successfully finished this exercise.

Points to take away from this chapter:

- Adding support for a USB port is simplified by the FSP Configurator and the USB middleware.
- To get a USB transfer going, a USB descriptor file is essential.

11. SECURITY AND TrustZone®

What you will learn in this chapter:

- What TrustZone is and how it helps to protect the device.
- Which benefits Renesas' implementation of TrustZone offers.
- How the separation of the secure and non-secure parts of an application works.
- How the device lifecycle management can be implemented.

Security is important! It must be thought of from the beginning, as it cannot be added later on. At least, not without an expensive redesign, which might also involve tearing down the whole application based on it. Think of it as a building foundation: it is not glamorous, and it may take more time to design and build than the creation of the remaining functions together. And when it is done, you won't even notice it. But still, security is essential for all the applications in the connected world.

When do you need security? You will need it, if your product is communicating through wired or wireless interfaces like WiFi, Bluetooth, USB or Ethernet. You will need it, if you store valuable information, like keys, certificates, passwords, personal data, measurement data, or algorithms. You will need it, if your product is upgradable, for example, if firmware updates, feature upgrades or paid services are possible.

That's why security affects so many application spaces. It concerns everything that is "smart", like smart grid, smart cities, smart buildings, smart homes, smart lighting, smart watches, smart appliances, or smart clothes, to mention just a few. It also impacts the Internet of Things or industrial applications, as sensors, robotics, building automation, or farming need to be kept safe as well. And not to forget medical applications, where a life can be quickly endangered, if security is not taken seriously.

The RA Family of microcontrollers from Renesas supports the creation of secure applications through several measures, be it the Secure Crypto Engines on devices using an Arm® Cortex®-M4, -M23, or -M33 core, through secure memory protection units on devices with a Cortex-M4 or -M23 core, or through Renesas' own implementation of Arm's TrustZone® on microcontrollers with a Cortex-M33 core. The latter is the topic of this chapter.

One last thing to mention: the majority of RA Family MCUs are PSA Certified™ Level 1, with the RA6M4 MCU Group being PSA Certified Level 2. This includes PSA Functional API certification of the RA Family's Flexible Software Package with Trusted Firmware M (TF-M). Several RA Family MCU groups have been SESIP1 certified, ensuring fundamental security capabilities regardless of the chosen software platform. In addition, the Secure Crypto Engines hold multiple NIST CAVP certifications, confirming the proper functionality of the various cryptographic functions. This gives users a high level of assurance and confidence for developing their secure product for the connected world.

11.1 What Is TrustZone® and How Does It help?

TrustZone® is a core technology developed by Arm® as part of the Arm v8-M architecture and offers a system-wide approach to security through hardware-enforced isolation. It enables the creation of a protected environment by demanding the separation of software into logical partitions of secure and non-secure MCU regions. It is typically implemented on Arm® Cortex®-M33 core devices and occasionally on Arm Cortex-M23 core devices.

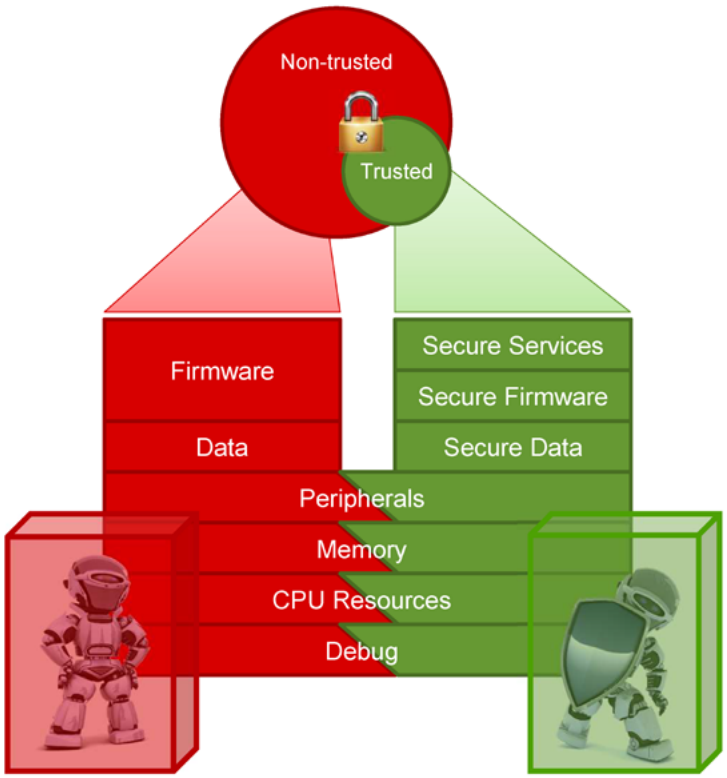


Figure 11-1: TrustZone® allows the split of the microcontroller's resources into non-secure and secure areas

Memory regions are split up into three different zones: a secure area, used for trusted or protected IP like key storage and data decryption, a non-secure area, used for normal applications, and a non-secure callable area acting as a gateway between the two other partitions. The latter one makes it possible that code residing in the non-secure area can call services in the secure world (see *Figure 11-2*). This is done through veneers, allowing the separation of the secure and non-secure zones.

With TrustZone, code from the non-secure world, the Non-Secure Processing Environment (NSPE), cannot access items in the secure world, the Secure Processing Environment (SPE), directly. This means that, for example, keys or certificates in the secure world cannot be changed or used by the non-secure world. Only code residing in the secure world can do that. This enables IP protection and sandboxing and reduces the attack surface of key components. And it simplifies the security assessment of embedded devices. So, if you have intellectual property (IP) in your microcontroller that you want to protect, put it in the secure world!

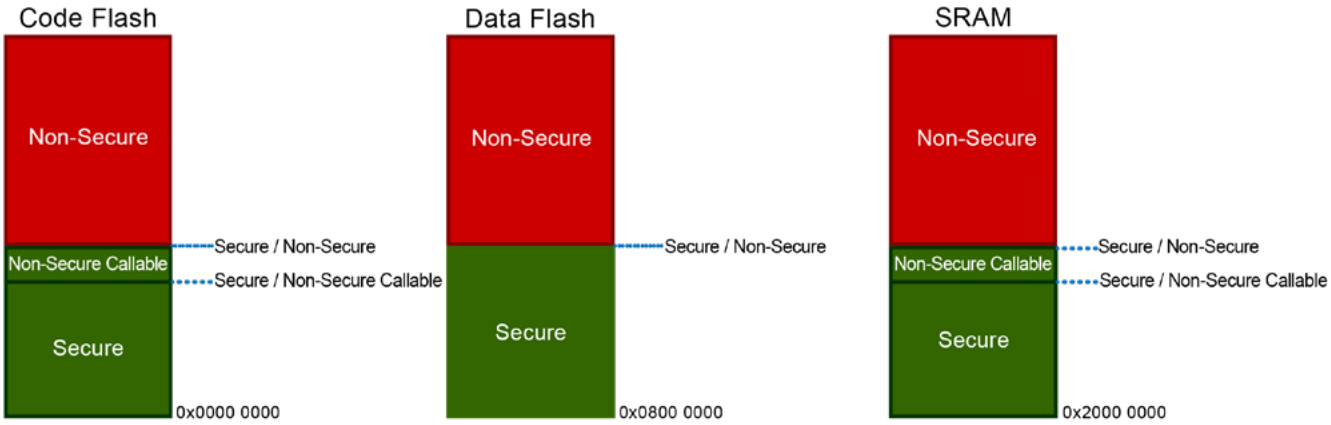


Figure 11-2: TrustZone® allows the segmentation into secure and non-secure worlds

Functions and data in the non-secure world may be accessed from the secure and the non-secure state while routines from the non-secure world can only access services in the secure world through the call of a veneer residing in the non-secure callable world (NSC). This way, a defined entry point into the secure world is provided.

While Arm's original definition of TrustZone applies to Flash memory, RAM, and built-in components like the SysTick timer, or the MPU, and calls for having application code and data separated from the secure items, it does not include protection for a DMA controller or for the dissociation of peripherals.

Renesas extended that concept and applied TrustZone filters to other buses and to pins and peripherals. The bus-filters prevent non-secure code from extracting secure code and data via DMA, DTC, and similar mechanisms. Or in other words: it prevents peripherals to access memory it should not be allowed to access. On the other hand, the TrustZone filters applied to pins and peripherals allow to lock down pins and to make them available only to the secure world rather than allowing them to be accessed from anywhere.

This not only protects the external interfaces, but additionally prevents non-secure code from eavesdropping on pins. For example, a malicious piece of code residing in the non-secure world cannot listen to the status of a pin assigned to the secure world and rebuild the content of, e.g., a transfer through the UART. Non-secure code can also not override outputs.

Renesas also ensured that there is no security gap at startup. Where other manufacturers use a software configured Security Attribution Unit (SAU) for their implementations, which leaves TrustZone unconfigured for a short amount of time after reset, Renesas decided to use an Implementation Defined Attribution Unit (IDAU) for its RA Family that is configured via non-volatile registers. This means that TrustZone is already configured and the device secure when the reset vector is fetched and therefore before any application executes. The security attributions are set in non-volatile memory via the SCI (on-board factory) boot mode and the application cannot modify their contents, only read it.

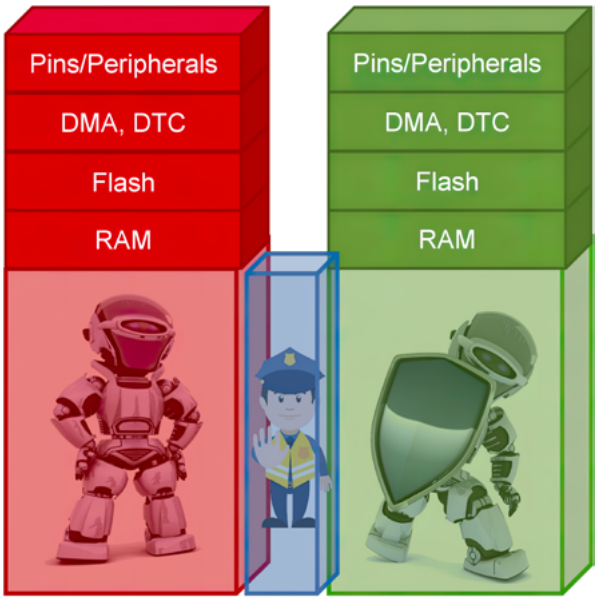


Figure 11-3: Renesas' implementation of TrustZone applies filters not only to memory, but also to other busses, pins, and peripherals

There is one thing to note: Application security is not achieved through the usage of TrustZone alone. Simply using it does not mean the device becomes secure. TrustZone is just another tool in your security toolbox, and you have to use it in the right way. If your secure application has vulnerabilities – like sloppy written code – then TrustZone by itself will not avert attacks and exploits. Your secure code must be proven to be correct and trusted.

But TrustZone helps you to provide and enforce a root of trust (RoT). A RoT is a source that can always be relied on and that is the base on which all secure operations, data, and algorithms depend. It makes not only services like code and data integrity and separation, system verification, confidentiality, or authentication available to the current or other trusted systems or devices, but it additionally incorporates keys, certificates, or passwords. But this also means that if the root of trust gets compromised, it's all over.

TrustZone provides data and firmware protection by allowing to store sensitive data in the secure world. It also permits to pre-load critical software as secure firmware and to configure peripherals to permit access only from the secure world, supplying operation protection.

For more information and a more detailed explanation of TrustZone, please refer to the documentation from Arm (**https://developer.arm.com/ip-products/security-ip/trustzone**), but please note that Arm often talks about Trusted and Non-Trusted. Definitions, which map directly to Secure and Non-Secure.

11.2 Partitioning of the Secure and Non-Secure Worlds

Now that we know that our program needs to be split between the secure and non-secure world, how do we partition our software accordingly? For this, a TrustZone® based systems always consist of two different projects: one secure and one non-secure. Both can utilize SRAM and code and data Flash memory but, only the secure code can access both – secure and non-secure – areas directly.

These projects are set up with the help of the Project Configurator in e² studio. Once you create a new project, you will be prompted to select the type your project should have:

- Flat (Non-TrustZone) Project
- TrustZone Secure Project
- TrustZone Non-secure Project

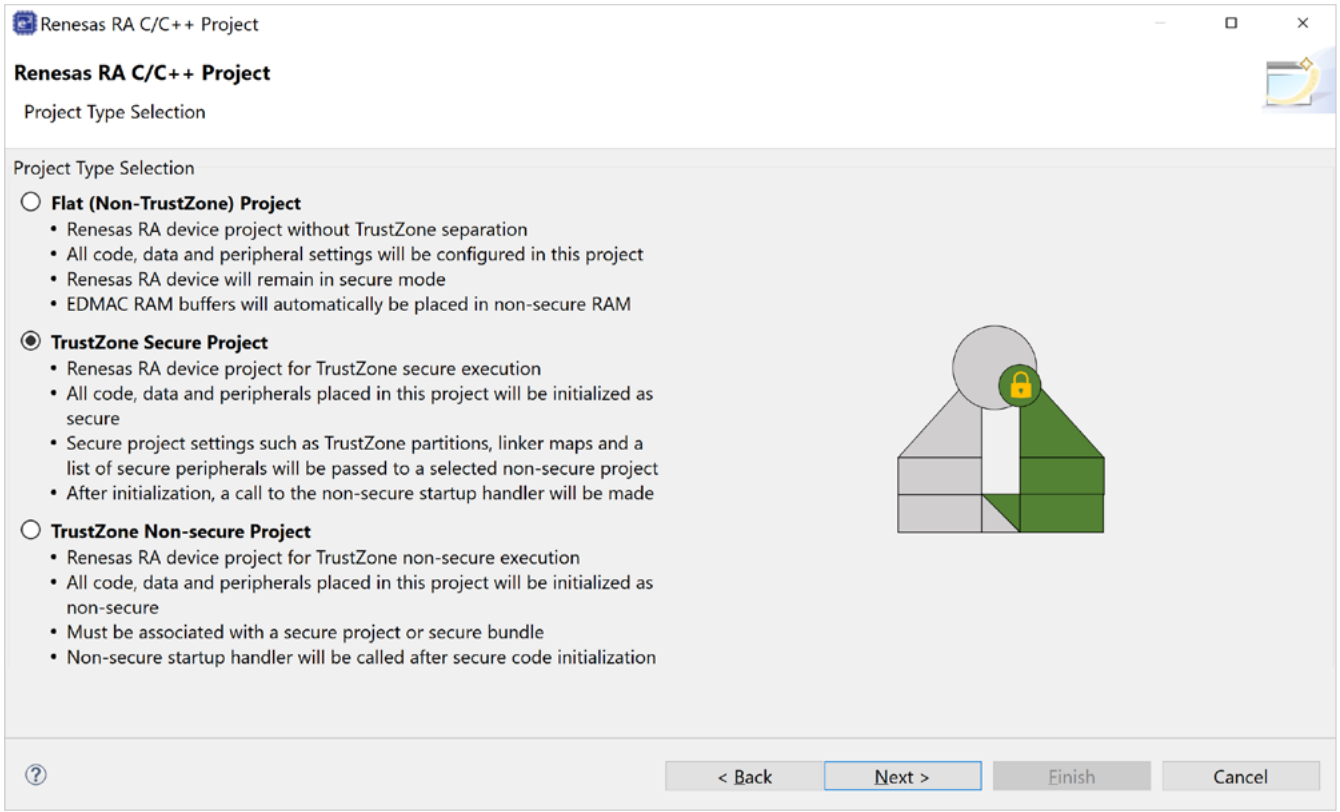


Figure 11-4: The Type Selector page of the Project Configurator lets you select between Flat, Secure, and Non-secure projects

Note that if you select the Flat (non-TrustZone) project, the microcontroller will remain in secure mode after boot. Also, care is needed when setting up a TrustZone project to ensure that the connection between the secure and non-secure partitions are managed correctly. This is done by associating a non-secure project with a secure project or bundle during the creation of the non-secure project in the Project Configurator. e² studio will ask you to assign a secure project to your non-secure project once you selected *TrustZone Non-secure Project* on the *Project Type Selection* screen and clicked on *Next*.

Once you created the secure project, you can make your secure stacks and drivers available to the non-secure world. For this, right click on the topmost module and choose *Non-secure Callable* from the pop-up menu. Note the little arrow at the left after you selected that entry: it indicates that this module is now non-secure callable (see *Figure 11-5*).

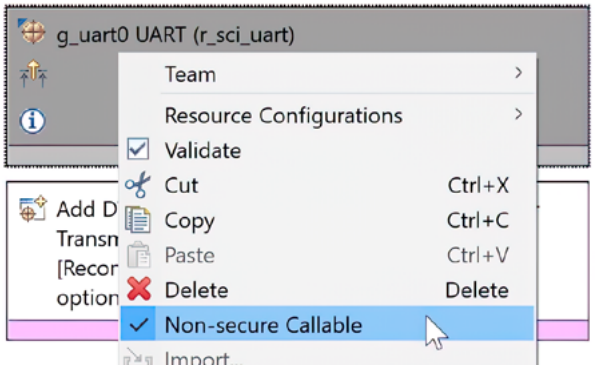


Figure 11-5: The topmost module of a stack in a secure project can be made non-secure callable

And last but not least, you can also partition your memory from inside e² studio: go to *Run → Renesas Debug Tools → Renesas Device Partition Manager* and a utility will run. The Device Partition Manager allows you to perform the Lifecycle State Management during development and additionally lets you setup and query the IDAU regions, and unlock erased Flash memory blocks.

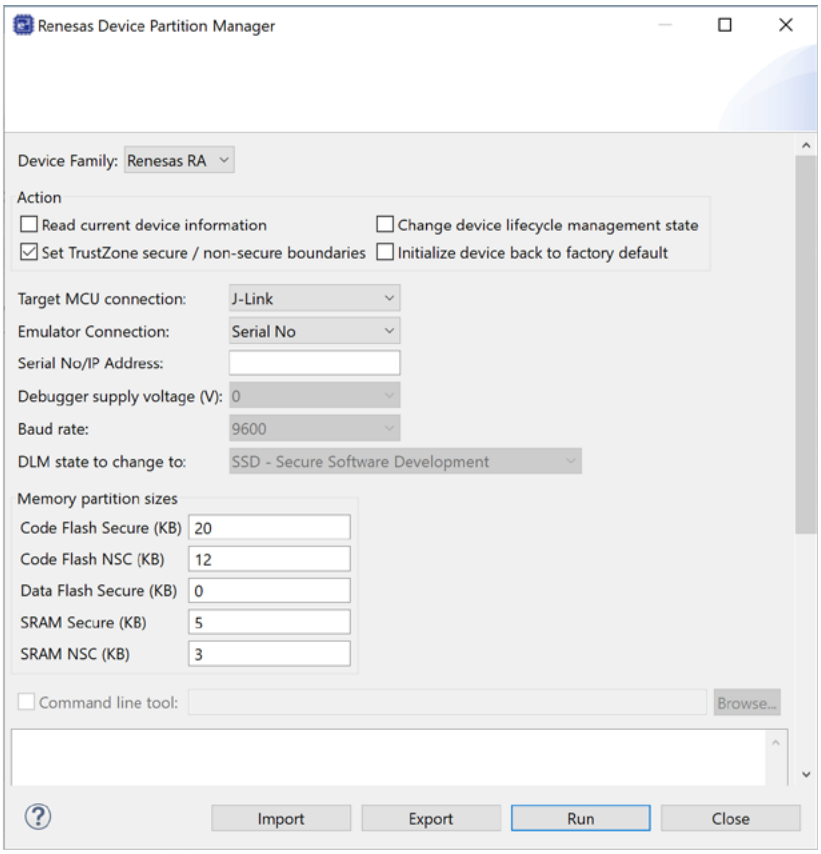


Figure 11-6: The Device Partition Manager lets you define the sizes of the different memory partitions

And, if you want to dive deeper into the tools supporting the configuration of Arm's TrustZone for the RA Family of microcontrollers and the associated workflow: Renesas provides an RA Arm TrustZone Tooling Primer on their internet site (**https://www.renesas.com/eu/en/document/apn/ra-arm-trustzone-tooling-primer**) which will introduce you to the basic concepts of using the tools and setting up a project.

11.2.1 Function Calls Across the Borders

Now, what happens, if for example, a part of an application residing in the non-secure world wants to call the Flash peripheral – which is located in the secure world – to program the non-secure data Flash? For this, a new instruction has been added to the instruction set of the Arm® v8M Cortex®-M33 cores: SG or Secure Gateway. This instruction must be located in the non-secure callable (NSC) region between the secure and the non-secure part of the memory. This ensures that even if the SG opcode is found elsewhere in the secure world, it could not be used as an entry point. After the SG instruction, a call can be made into the code on the secure side (see *Figure 11-7*).

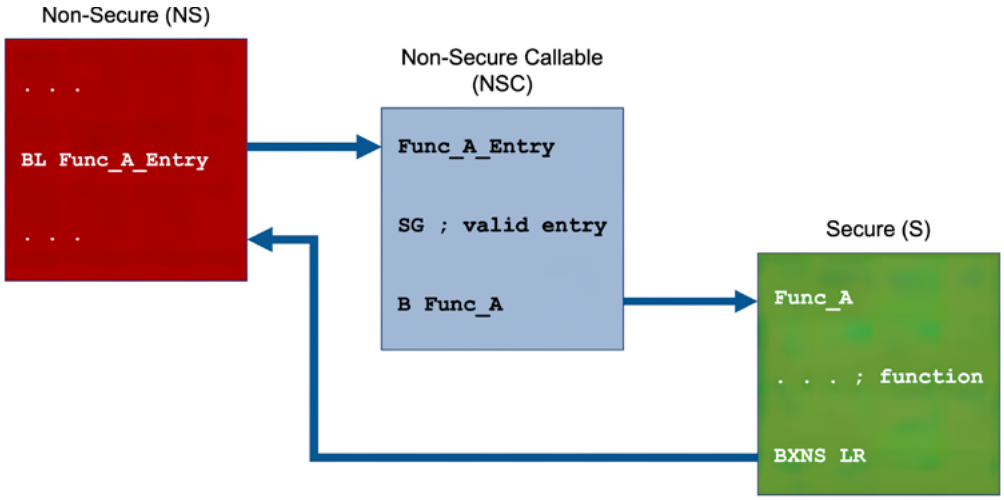


Figure 11-7: Calling secure functions from a non-secure context

The return from the secure side to the non-secure side will be performed through the `BXNS LR` (BXNS = branch with exchange to non-secure state) opcode, which will branch back to the address contained in the Link Register (LR) placed there during the `BL Func_A_Entry` branch. On the return of the function, the return state of the function is stored in the LSB of the return address in the LR. This bit is checked against the return state on arrival at the calling function to prevent the secure API – called from the non-secure code – from returning to a fake return address pointing to a secure address.

If a call from code residing in the non-secure world to code in the secure world would be performed where the first instruction is not the SG opcode in a NSC region, a secure fault would occur on a microcontroller with a CM33 core. This fault will be handled in the secure state.

Calling non-secure code from secure code is feasible as well, but not recommended, as it is a security concern due to the possibility of leaked data. The secure code can transfer some register values to the non-secure world through parameters and the compiler will clear the other secure data from the remaining registers. This mechanism also hides the return address of the secure software, ensuring that the code in the non-secure world does not manipulate the return address (see Figure *11-8*).

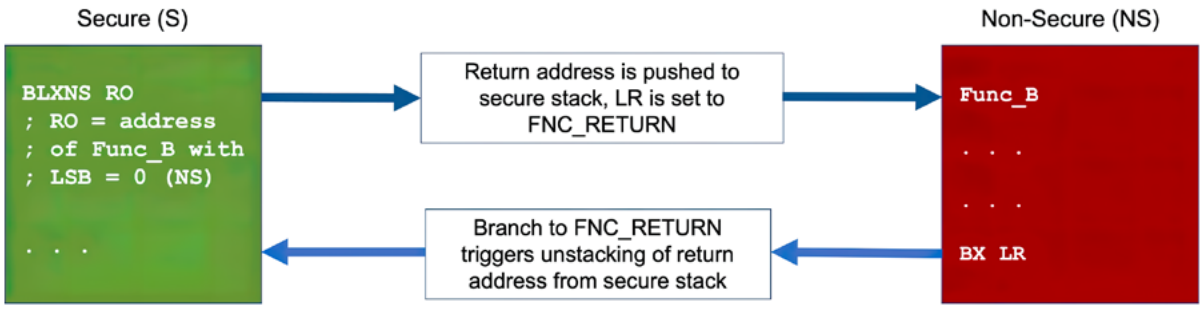


Figure 11-8: Calling non-secure functions from a secure context

A better approach to calling non-secure code from secure code through the BLXNS (branch with link and exchange to non-secure state) instruction initialize the code in the secure world at first startup, and then to pass program control to the non-secure world. Any data transfer from the non-secure world to the secure world thereafter should be managed through FSP callbacks.

11.2.2 Callbacks from Secure Code to Non-Secure Code

How do we handle a scenario, where a peripheral is on the secure side of the world, but the callback function serving its interrupts is located in the non-secure world? Normally, in an FSP callback, the callback structure is allocated on the stack by the ISR and would then be used by the callback function. As the interrupt service routine (ISR) and the callback are residing in different worlds, a security fault would occur, as the callback function would try to access the callback structure in the secure world.

The FSP solves this problem by allocating the callback structure in an area of memory available to both worlds. This is initialized by using the `callbackSet()` API, which is a guard function (see chapter 11.2.3 for a description of the guard functions) allowing access to the secure world. This call would look like that:

```
fsp_err_t (* callbackSet)(uart_ctrl_t * const p_ctrl,
                          void (* p_callback) (uart_callback_args_t *),
                          void const * const p_context,
                          uart_callback_args_t * const p_callback_memory);
```

The callback function pointer and context pointer are already provided in the configuration structure of the module, but they must be created again, as the configuration structure on the secure side is built separately from the structure on the non-secure side. The pointer to the volatile callback memory points to a location where memory for the structure accessible from both worlds can be allocated. This way, the secure fault is eliminated.

11.2.3 Guard Functions

The application programming interfaces (APIs) of the guard functions allow the access to drivers residing in the secure world from a non-secure project. This feature implemented by Renesas is unique and a patent for it is pending. The Flexible Software Package (FSP) will automatically generate guard functions for all the top-of-stack modules and/or driver APIs marked in the FSP configurator as non-secure callable and will add them to the project in the NSC region. Additionally, the FSP will create non-secure module instances for the corresponding NSC instances.

These instances can be used the same way as usual but have their `p_ctrl` and `p_cfg` members set to `FSP_SECURE_ARGUMENT`, which equates `NULL`, and their `p_api` member points to the guard functions instead of the actual member functions. The guard function itself has the `p_ctrl` and `p_cfg` members hard coded into the memory of the secure world. With drivers existing in both secure and non-secure memory where different channels are used on different sides, this eliminates the possibility that by manipulating the `p_ctrl` and/or the `p_cfg` structure a secure channel could be accessed directly from the non-secure code. Guard functions also check any input pointer to ensure that the caller does not overwrite secure memory.

```
const uart_api_t g_uart0_api = {
            .open           = guard_g_uart0_R_SCI_UART_Open,
            .close          = guard_g_uart0_R_SCI_UART_Close,
            .write          = guard_g_uart0_R_SCI_UART_Write,
            .read           = guard_g_uart0_R_SCI_UART_Read,
            .infoGet        = guard_g_uart0_R_SCI_UART_InfoGet,
            .baudSet        = guard_g_uart0_R_SCI_UART_BaudSet,
            .versionGet     = guard_g_uart0_R_SCI_UART_VersionGet,
            .communicationAbort = guard_g_uart0_R_SCI_UART_Abort,
            .callbackSet    = guard_g_uart0_R_SCI_UART_CallbackSet,
};
/* Create non-secure instance that has a non-secure callable API */

const uart_instance_t g_uart0 =
{
.p_ctrl = FSP_SECURE_ARGUMENT, // CTRL is static in guard function
.p_cfg  = FSP_SECURE_ARGUMENT, // CFG is static in guard function
.p_api = &g_uart0_api
};
```

Furthermore, designers can choose to add additional levels of access control or to delete guard functions if they wish to expose only a limited range of APIs to a non-secure programmer. Staying with the example of the SCI, a channel may be opened and configured for the desired baud rate by the programmer of the secure world, but by deleting all but the `g_uart0_write_guard()` API, only the write API is available to the developer of the non-secure application.

11.3 Device Lifecycle Management

The device lifecycle defines the different phases of a device's life and controls the capabilities of the debug interface, the serial programming interface and the Renesas test mode. This way, the writing of the code residing in the secure world and writing of the applications running in the non-secure world can be separated for security reasons and a product can be developed by two independent teams: a team of secure developers creating the Root of Trust (RoT) or an isolated subsystem, and the designers of the non-secure world, which create the application that uses that RoT or the subsystem. This design split is supported by the Flexible Software Package (FSP) and e^2 studio.

Once the code for the secure world is ready, it can either be pre-programmed into the device and the lifecycle set to NSECSD, which locks the secure world, or referenced as bundle by a non-secure project. The application designers would take over from here and write the application in the non-secure world, debug it and flash it into the device. If desired, they could also disable the program and erase functionality of the Flash memory blocks used. As a final step, they would set the lifecycle state to either deployed, debug locked, or boot locked. This way, the entire device is protected, and the programming interface and the device cannot be debugged, read, or programmed. *Figure 11-9* shows the possible states and transitions, while the table in *Figure 11-10* provides an explanation for each lifecycle.

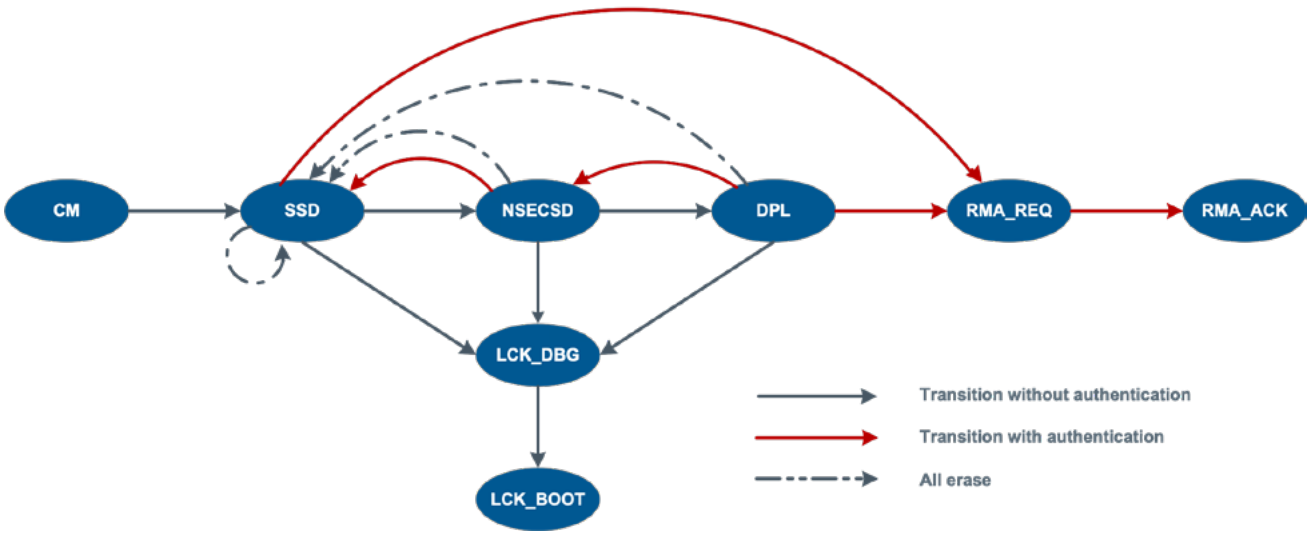


Figure 11-9: The different states of the Device Lifecycle Management

There are three different debug access levels, which change according to the lifecycle state:

- **DBG2:** The debugger connection is allowed, and there are no restrictions to access the memories and peripherals.
- **DBG1:** The debugger connection is allowed, but the access is restricted to non-secure memory regions and peripherals only.
- **DBG0:** No debug connection is allowed at all.

Lifecycle	Definition	Debug Level	Serial Programming	Test Mode
CM	"Chip Manufacturing" The state when the customer received the device.	DBG2	Available, cannot access code/data Flash memory	Not available
SSD	"Secure Software Development" The secure part of the application is being developed.	DBG2	Available can program/erase/read all code/data Flash memory areas	Not available
NSECSD	"Non-Secure Software Development" The non-secure part of the application is being developed.	DBG1	Available can program/erase/read non-secure code/data Flash memory areas	Not available
DPL	"Deployed" The device is in the field.	DBG0	Available cannot access code/data Flash memory area	Not available
LCK_DBG	"Locked Debug" The debug interface is permanently disabled.	DBG0	Available cannot access code/data Flash memory area	Not available
LCK_BOOT	"Locked Boot Interface" The debug interface and the serial programming interface are permanently disabled.	DBG0	Not available	Not available
RMA_REQ	"Return Material Authorization Request" Request for RMA. The customer must send the device to Renesas in this state.	DBG0	Available cannot access code/data Flash memory area	Not available
RMA_ACQ	"Return Material Authorization Acknowledged" Failure analysis at Renesas	DBG2	Available cannot access code/data Flash memory area	Available

Figure 11-10: Description of the different stages of the Device Cycle Management

The transitions from one state to another can be performed using either the Renesas Flash Programmer or the Renesas Device Partition Manager, but the latter one will allow only a limited selection of states. The transitions between the states can be secured by using authentication keys. You can find more information on the different DLM states and the device specific transitions in the User's Manual of the respective microcontroller.

11.4 Use Cases for TrustZone®

Now that we know what TrustZone® is, how it helps to put data and intellectual property (IP) protection in place, what the benefits of its implementation by Renesas are, and how the separation of the secure and the non-secure world looks, it is time to have a glance at some use cases. The following parts of this chapter will cover how TrustZone assists to protect IP, supports the isolation of legally relevant code, and keeps the Root of Trust (RoT) safe.

11.4.1 IP Protection of Pre-Programmed Algorithms

In case an application must have access to a functional algorithm which has to be protected against tampering, the possibility of splitting the development between the implementation of the secure algorithm and the remaining code is a huge benefit. The designers of the algorithm would first create the secure project with a defined application programming interface (API) using the Project and FSP Configurators in e² studio, write the algorithm and debug it with any of the debug interfaces available. Then they would flash it into the microcontroller and protect it by disabling the program or erase functionality on the used blocks of Flash memory, if desired. The secure project will automatically configure TrustZone. Before handing the pre-programmed device over to the application developers, the secure team will set the lifecycle state to non-secure software development (NSECSD) so that the algorithm cannot be read by the debugger or a Flash programmer.

The application writers would then create a non-secure project in e² studio, write their application and debug it employing any of the debug interfaces. Their application can call any of the secure project's exposed APIs without problem. Once they are done, they program the final code into the microcontroller, disable program or erase of the Flash memory blocks used, and set the device lifecycle state to deployed (DPL), debug locked (LCK_DBG), or boot locked (LCK_BOOT). Now the entire unit is protected and ready to be shipped to customers.

The big advantage TrustZone offers here is that TrustZone protects against algorithm misuse, be it inadvertently or maliciously, and that it allows to have the application design split between secure and non-secure parties. But what happens if there is a flaw in the secure algorithm which needs to be corrected? From *Figure 11-9,* you can see that there is a way back to the SSD state, by erasing the protected algorithm if the erase functionality was not disabled by the secure developers. This minimizes scrappage of pre-programmed devices.

11.4.2 Code Separation for Legally Relevant Code in a Smart Meter

The European specifications for smart meters define legally relevant code, which gets certified. This code has to be isolated from the rest of the meter. Currently, most customers employ some kind of physical separation by using two microcontrollers. This is expensive but simplifies certification.

Another possibility would be to do a logical split between the legally relevant code, data and peripherals, and the application code, like the code for display or the DLMS / Cosem (Device Language Message Specification / Companion Specification for Energy Metering), on a single microcontroller using TrustZone. This way, TrustZone would provide a provable isolation and shields against code misuse and corruption on a single device.

11.4.3 Protection of the Root of Trust

As explained in Chapter 11.1, the Root of Trust (RoT) builds the security foundation of the complete product and must therefore be protected. All higher-level security is built on top of the RoT and it provides authenticated firmware updates and secure communication. Furthermore, the RoT enables the recovery from higher-level security failures, but if it is compromised, nothing built on it is secure anymore.

This means that all the factory keys, the device identity, any checksums, verification of Flash images, crypto services, and keys and certificates, as well as sensitive date should be kept inside the secure world. Everything else, like the primary application, user interfaces, the non-secure aspects of interface protocols, services, and others, should be placed in the non-secure world. To keep the surface for attacks to the secure world as small as possible, as much as practicable of the overall application should be kept in the non-secure world.

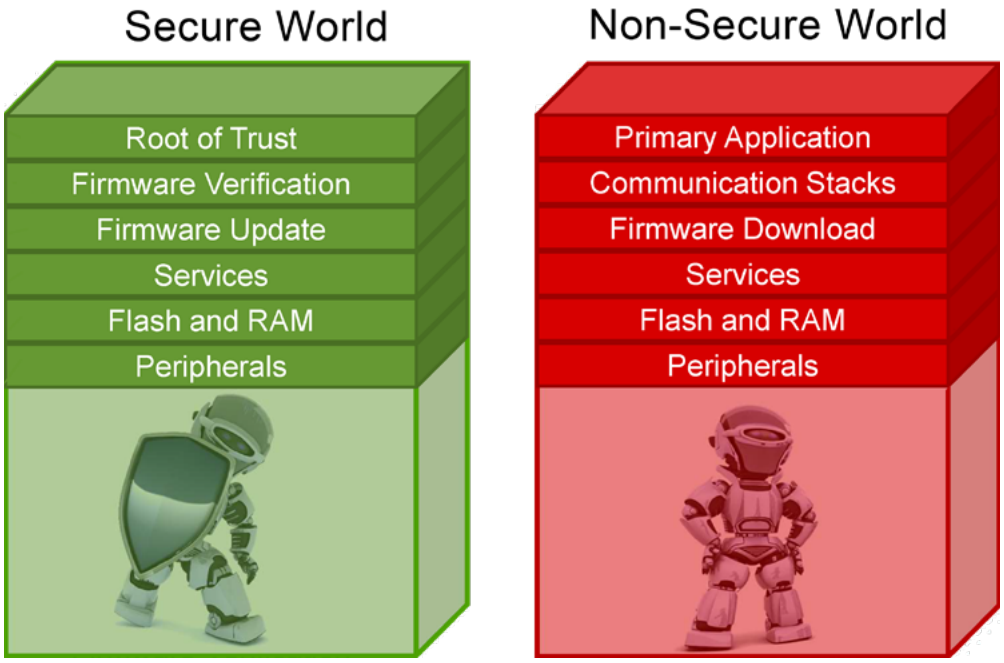


Figure 11-11: What to keep in the secure and non-secure worlds

Points to take away from this chapter:

- TrustZone eases the separation of the secure and the non-secure world.
- Renesas' implementation of TrustZone closes the security gap at startup.
- Two different projects are necessary: one secure and one non-secure.
- The non-secure callable section allows calls into the secure area through guard-functions.
- Lifecycle managements helps to split the development between the different teams.

12 WHERE TO GO FROM HERE

What you will learn in this chapter:

- What the RA Partner Ecosystem is and how it helps to shorten development times.
- Where to get example projects.
- Where to go for additional online-trainings, whitepapers and application notes.

Now that you have learned the basics features of the RA Family of microcontrollers, how to write programs for it, and which support in terms of software and hardware tools is available, you might ask "Is there more"? The answer is yes. This chapter will provide you with links and hints to additional support available either from Renesas or from its partners.

12.1 The Renesas RA Partner Ecosystem

The design complexity of embedded systems has increased exponentially over the last years and with it has the number of design problems and the number of new technologies. Developers of Edge or IoT devices are struggling more than ever to deliver their projects on time, as the level of difficulty goes up, and development times continue to shorten. This raises the need for flexible and ready-to-use solutions and building blocks, be it software or hardware.

Renesas supports this approach with a broad portfolio of third-party applications and building blocks inside the RA Partner Ecosystem with over 50 partners. This ecosystem helps engineers to accelerate their development of embedded systems for the Internet of Things (IoT) by providing plug and play options that work right out-of-the-box. These range from security, over voice user interfaces, graphics, machine learning, to human-machine-interfaces and cloud connectivity, among others. Each new partner building block solution is labeled with an "RA READY" badge, indicating the product is designed to solve real-world customer problems. You can find more information on the website of the Partner Ecosystem: **https://www.renesas.com/ra-partners**.

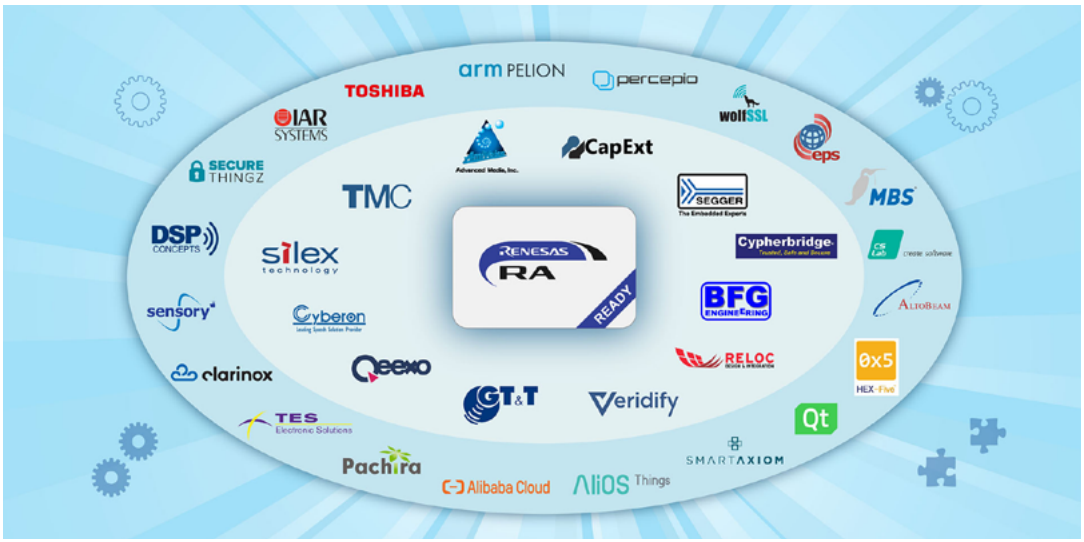


Figure 12-1: The Renesas RA Partner Ecosystem comprises ready-to-use software and hardware components from more than 50 partners

12.2 Example Projects®

To help you to speed up your development and to deliver your project on time, Renesas provides several example projects you can download and use freely. While the User's Manual of the Flexible Software Package features short examples for each API and module, there are more comprehensive example project bundles available as well, ready-to-run on the Evaluation Kit of your choice. For these bundles, a Usage Guide is at hand, which explains how to install the tools and how to import and run the projects.

The bundles include example projects for most of the on-chip peripherals, like the A/D-converter, the Flash peripheral, USB, SPI or UART connectivity, FreeRTOS™, Azure® RTOS, and much more. Each project inside a bundle comes with a ReadMe-file outlining the functionality provided, the hardware requirements, jumper settings, and hardware connections.

Other sample projects are available as well, as for example an IEC 60730/60335 self-test library, motor control software, or sample code for a sensor network solution. You can access the downloads from your Evaluation Kit's homepage or from the homepage of your RA microcontroller.

12.3 Online Trainings, White Papers and Application Notes®

With this book coming slowly to its end, the question "Where can I learn more about the RA Family of microcontrollers?" may surface. If you want to dive further into this topic, there is an abundance of information, trainings and tutorials available for your personal education. The following paragraphs of this chapter will highlight a few of them, but there are others as well and links to most of them can be found on the RA Family's website (**https://www.renesas.com/ra**) under *Support*.

12.3.1 Online Trainings

There are two sites available which provide you with online trainings: the Renesas Academy and the RA channel from Renesas on YouTube™. The Renesas Academy is an online learning platform that brings all learning materials together in one place, providing the convenience of learning what you want, when you want, and at your own pace. The courses available there are not only covering the RA Family of microcontrollers, but also the other product families from Renesas like Synergy, RX, RZ, or analog and power products.

You can go online and quickly find in the course catalogue the trainings you need. Every course is broken down into short modules that you can fit into your schedule. All that is required is a simple one-time registration on the Renesas Academy site (**https://academy.renesas.com**) and you are ready to take whatever training you want. Current courses include topics like introductions to different microcontrollers of the RA Family and trainings for the RA Partner Ecosystem. New and updated courses, content and features are added continuously, so be sure to keep checking back.

There is also an RA family video library available on the Renesas website (https://www.renesas.com/eu/en/products/microcontrollers-microprocessors/ra-cortex-m-mcus/ra-video-library). From there, you can access videos covering various topics. These include, but are not limited to, processor and kits presentations, introductions into the RA Family or the RA ecosystem, and several webinars. Videos on how to use the tools have their own website: The RA Family Software & Tool Course page (https://www.renesas.com/eu/en/software-tool/ra-software-tool-course), which features introductions and "How To" trainings, as well as solution development learning.

Finally, there is the official Renesas RA Platform channel on YouTube (**https://www.youtube.com playlist?list=PLgUXqPkOStPv2KcNs_1Wc5MY31GOPPAd3**). It provides you with tutorials and training videos for the RA Family of microcontrollers. Topics include introductions to the RA microcontrollers, tools and kits, and videos explaining the usage of boards and e^2 studio. Additional content will be added continuously.

12.3.2 White Paper and Application Notes

Renesas provides a huge number of white papers and applications notes covering hardware and software design topics, security, IoT connectivity, motor control, human-machine-interfaces and much more. All of them will help you to solve your design problems without having to re-invent the wheel and will therefore speed up your development. As with the example projects, they can be accessed either from the pages of the microcontroller or from the pages of the Evaluation Kit you use. Also, the homepage of the RA Family is a good starting point.

If you want to browse all the white papers and application notes Renesas offers, you could go the following page and enter the part numbers of interest as a filter: **https://www.renesas.com/eu/en/support/document-search?doc_category_tier_1=&doc_category_tier_2=&doc_category_tier_3=&doc_category_tier_4=&doc_part_numbers=&title=&sort_order=DESC&sort_by=field_document_revision_date#documentation-tools-results**.

Points to take away from this chapter:

- Example projects cover whole applications or certain aspects of an RA microcontroller.
- Online videos provide a fast way to familiarize yourself with tools and devices.
- The RA Partner Ecosystem provides ready-to-use building blocks from third parties.
- White papers and application notes cover a wide range of topics.

ABOUT THE AUTHOR:

Richard Oed has worked as Field Applications and Systems Engineer for more than 23 years in the field of Embedded Processing, providing support and writing software for DSPs, microcontrollers and microprocessors, as well as for data converters. Since autumn 2015 he works as freelance author, writer and journalist. Richard is co-inventor of three patents and is member of the IEEE, the ACM and the VDE. He is also the author of the book „Basics of the Renesas Synergy-Platform", available from Renesas at **renesas.com/synergy-book**.

Before purchasing or using any Renesas Electronics products listed herein, please refer to the latest product manual and/or data sheet in advance.

Renesas Electronics Corp.

www.renesas.com

Document No. R01TU0337ED0200