Renesas Synergy™ Platform

Software Development Best Practices Guide for SSP Users

R11AN0235EU0100
Rev.1.00
Oct 18, 2017

Introduction

The Renesas Synergy™ Platform can dramatically decrease time-to-market, ownership costs, and barriers to entry for engineers developing embedded systems. To maximize the benefit from the Renesas Synergy Platform, there are general guidelines that you should follow. This Application Note will show examples in learning best practices with developing your own projects using the Synergy Software Platform (SSP).

Target Device

Renesas Synergy™ Microcontrollers

Recommended Reading

- *SSP User's Manual*, Introduction section
- SSP Datasheet v1.2.0 or later

Note: If you are not familiar with the above documents, you should review them before continuing.

Purpose

This document provides guidelines for developing embedded software using the Renesas Synergy Platform. The purpose of this document is to assist developers to quickly get- up-to-speed with known best practices and tips that will decrease the learning curve.

Intended Audience

The intended audience are users that understand the Synergy Platform's fundamentals, are interested in learning best practices for using the platform, and in developing their own projects using SSP.

Contents

1. Software Architecture Overview

The overall software architecture is a tiered architecture designed to improve reusability, and portability, and to maximize performance. The architecture's foundation is the board support package (BSP). The next tier up, which is dependent upon the BSP, are the drivers. These drivers are comprised of modules for the various peripherals on the microcontroller, such as SPI, I^{2}C, and timers. The application framework tier sits above the drivers and is comprised of modules that provide higher-level services and ThreadX® RTOS integration. Your application code sits at the top and can access any of these tiers, or bypass them, depending on the application needs.

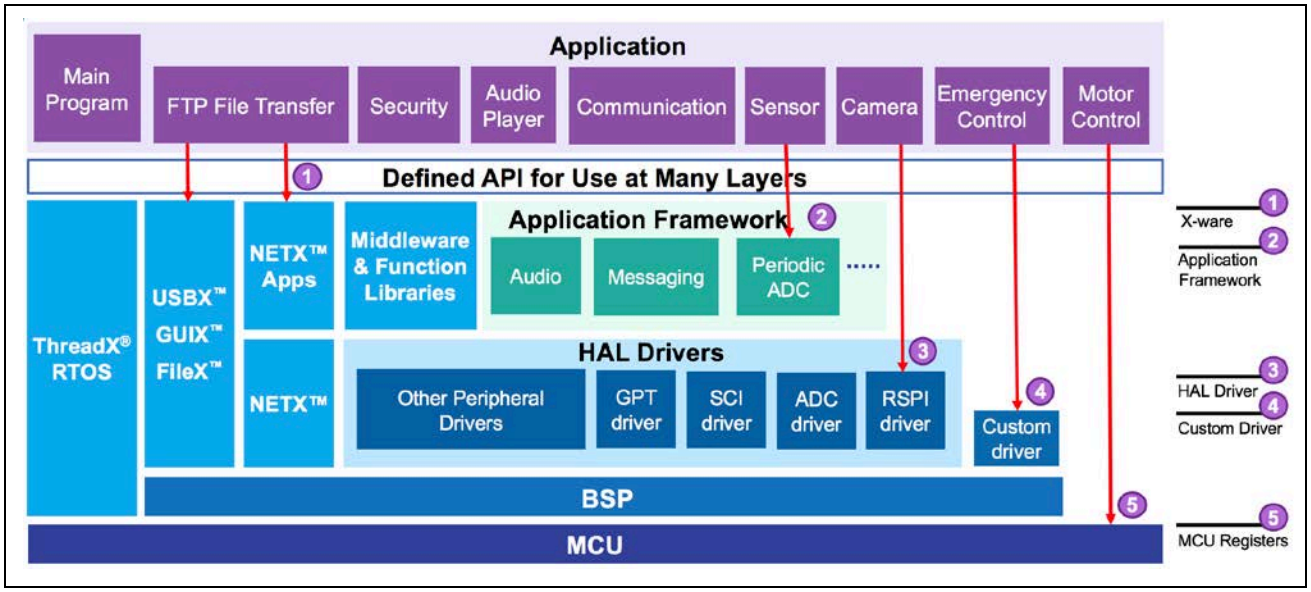

Figure 1 Software Architecture Overview

2. SSP Architecture Overview

Working with the SSP requires a certain understanding of how different terminology is used within the framework. The terms used, in many cases, parallel those used in computer science, but for clarity, the definitions have been listed in the following table.

Table 1 SSP Term Definitions

Term	Description
Module	Modules can be peripheral drivers, application software, or anything in between. Each module consists of a folder with source code, documentation, and everything you need to use the code effectively. Modules are independent units, but they may depend on other modules. Example SSP modules are the UART Driver (UART Interface), Audio Playback Framework, or the Messaging Framework (Messaging Framework Interface). Applications can be built by combining multiple modules to provide developers with the needed features and functionality.
BSP	The SSP Board Support Package (BSP) provides startup code for the MCU plus basic services. Example BSP services include startup code, initial clock setting, functions to globally enable and disable interrupts and to set the global interrupt level.
Callback Function	A callback function is a standard C function that is called when an event occurs. For example, the bus error interrupt handler is implemented in `r_bsp`. You will likely want to know when a bus error occurs. To alert the application, a callback function can be supplied to `r_bsp`. When a bus error occurs, `r_bsp` jumps to the provided callback function, and you can handle the error. Interrupt callback functions should be kept short and handled carefully because they are invoked from within an interrupt service routine. Callback functions should adhere to best practices for interrupts.
Framework	Framework is a collection of drivers, stacks, and modules that work together to solve a common design problem. The framework often abstracts the lower level details, allowing you to provide configuration information to the framework for its specific behavior.

Term	Description
Component	Components are program modules designed to interoperate with each other at runtime. Components and objects are often used synonymously.
Interface	Interfaces can be considered a contract between two modules. The modules agree to work together using the information that was agreed upon in the contract. An interface occurs between two or more separate modules, components, or even software tiers.
Instance	Instance is a single copy of a running module, driver, or stack. There are circumstances such as when multiple channels exist for a peripheral where there may be multiple instances for the driver loaded to handle each hardware channel.
Application	Application is a high-level code that is owned and maintained by a developer for a commercial product. Application code may be based on sample application code provided by Renesas, but is the responsibility of the product company to develop and maintain.
Driver	Driver is a module that directly modifies registers on the microcontroller hardware.
Stacks	The SSP architecture is designed such that modules work together to form a stack. Starting with the uppermost module and going to the bottom most dependency forms a specific stack.
Layer / Level	A layer contains one or more modules that are dependent on the requirements for modules existing in the layer above and even below the modules. Layers contain a consistent interface that allows the layers to be used interchangeably. The interfaces between layers are designed so that the software performance is maximized.

2.1 SSP modules

Modules are the core building blocks of SSP. Modules can do many different things, but all modules share the basic concept of providing functionality upwards and requiring functionality from below. A graphical module representation can be seen in the figure below.

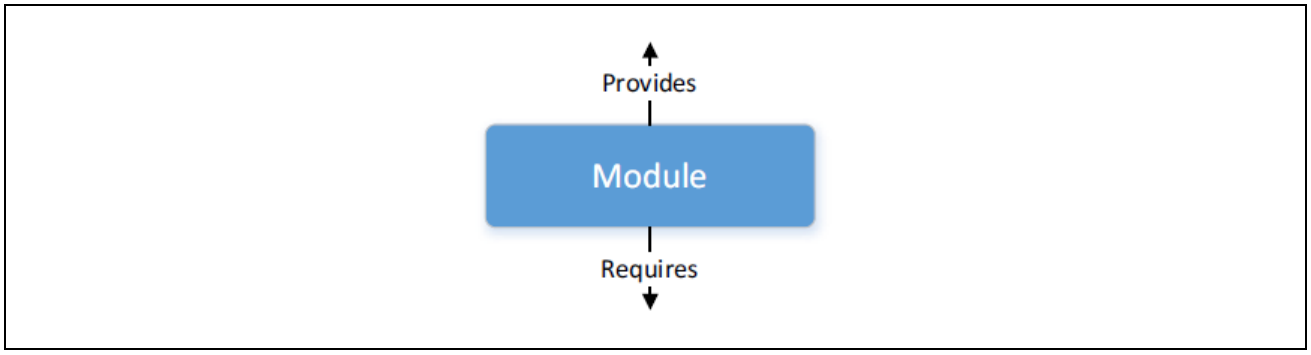

Figure 2 Module overview

The functionality provided by a module is not limited although there are usually points where separation makes sense to provide the module with high cohesion. If too much functionality is provided, then module reuse can become difficult. If not enough functionality is provided, then unnecessary complexity and overhead may be added to make the modules work as expected. The simplest SSP application contains one module with the user application on top as shown in the figure below..

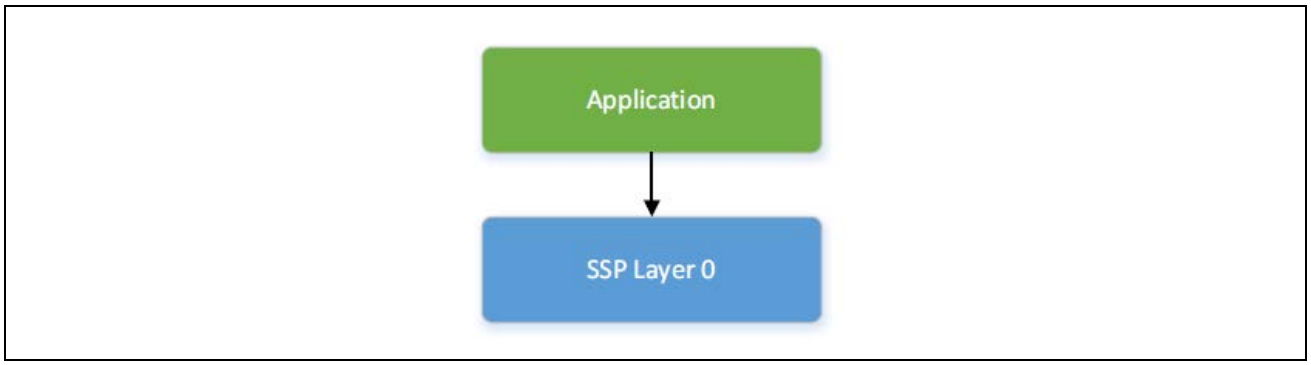

Figure 3 Simplest application with a single module

2.2 SSP stacks

When modules are layered atop one another, an SSP stack is formed. The stacking process is performed by matching what one module provides with what another module requires. For example, the Audio Playback Framework module requires a Transfer interface, which can be fulfilled by the Data Transfer Controller (DTC) Driver module. Instead of including the DTC code in the Audio Playback module, we split these into two modules. This allows for reuse of the underlying modules, that has many benefits. An example can be seen in the figure below.

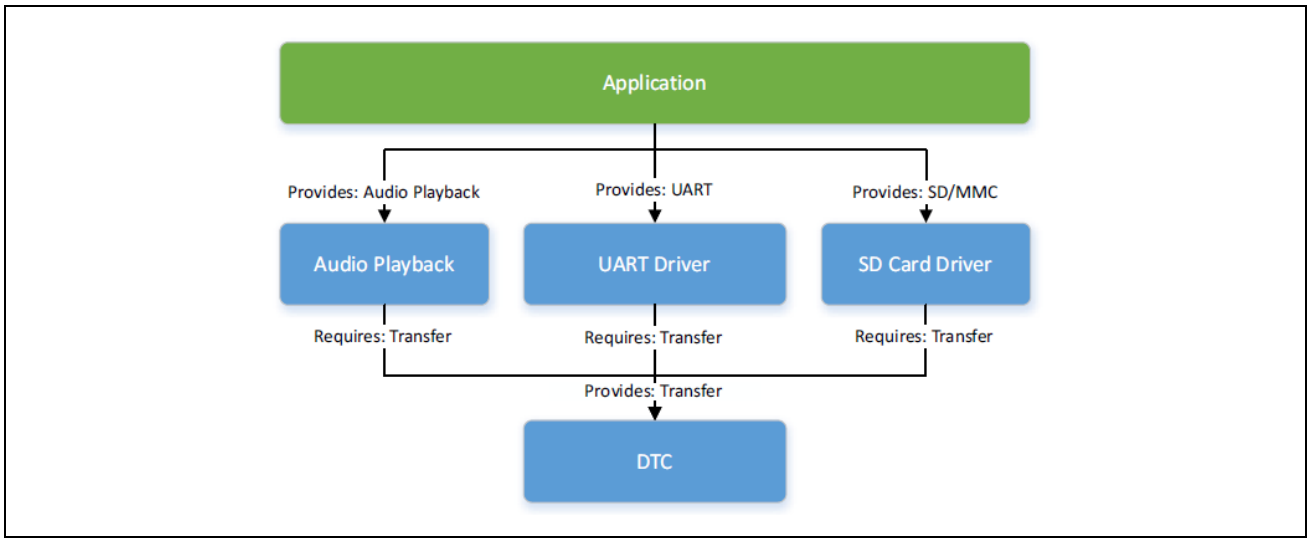

Figure 4 Example SSP stack

By continuing to add layers to the stack using SSP modules, developers can interface with the Synergy MCU Group at a high abstraction level.

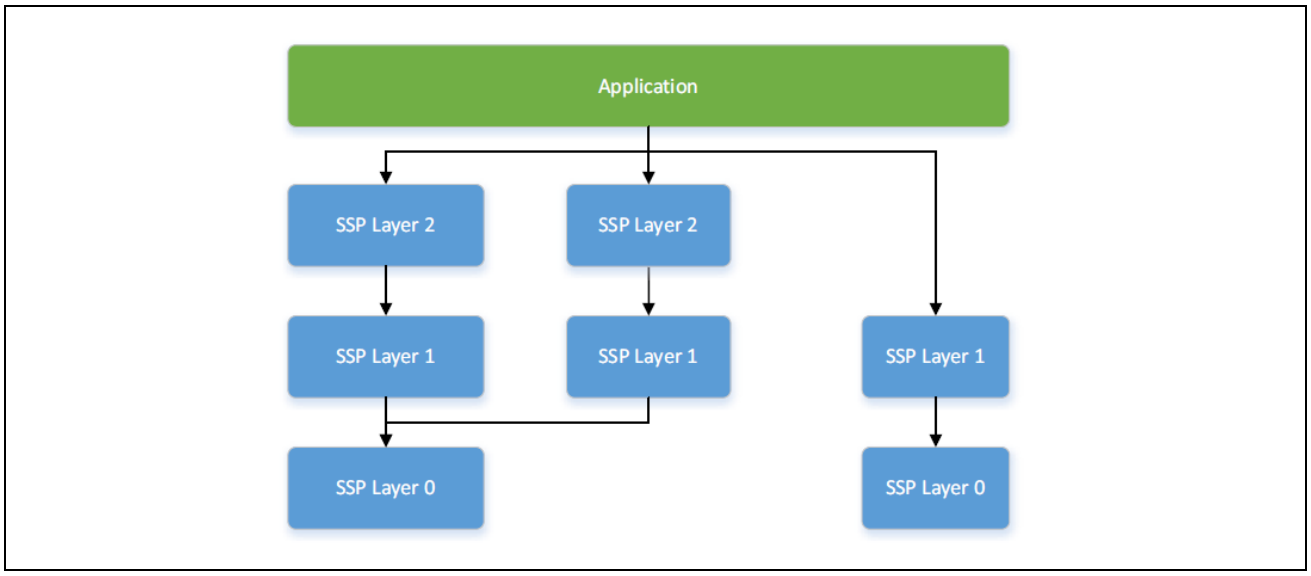

Figure 5 General SSP stack

The ability to stack modules has a great benefit because it ensures that the architecture is flexible. If modules are directly dependent upon other modules, then issues arise when application features must work across different user designs. To ensure that modules are reusable, the modules must be capable of being swapped out for other modules that provide the same features. The SSP architecture provides this flexibility to swap modules in and out through SSP interfaces.

2.3 SSP interfaces

At the architecture level, interfaces are the way that modules provide common features. This commonality allows modules that adhere to the same interface to be used interchangeably. Interfaces are a contract between two modules. The modules agree to work together using the information that was agreed upon.

Interfaces aim to provide support for the common features that most users would expect. This means that some advanced peripheral features might not be available in the interface. In most cases, these features are still available through interface extensions. In design, interfaces are implemented as header files. All interface header filenames end with `_api.h`.

The SSP interface contains five primary parts:

- Enumerations
- Data structures
- Callback functions
- API structure
- Version information

See the *SSP User's Manual* for a full discussion concerning these five primary components. Best practices for using and creating SSP interfaces include:

- Using enumerations to remove uncertainty and to provide known, available values, for parameters.
- Use a configuration structure to set the desired parameters and pass the configuration structure into the open API call.
- Avoid using `malloc()` and `free()`
- Use configuration structures to initialize peripherals and frameworks.
- Callback functions are used to allow modules to asynchronously alert an application when an event occurs.
- Callback functions can be called either from within an interrupt service routine or sometimes in a non-interrupt context such as through error handlers. You should adhere to following interrupt best practices such as:
  - Keep the callback as short as possible
  - Make the callback execute as fast as possible
  - Minimize function calls
  - Minimize writing large amounts of data to and from the stack
  - Do the minimum that must be done and alert a task to continue lower priority processing

- Understand the worst-case execution time
- Declare shared variables as volatile
- Avoid calling functions that are not interrupt safe such as `tx_mutex` or other RTOS calls that are not designed to be executed within an ISR.

2.3.1 SSP interface API structure

All interfaces include an API structure, which contain function pointers for all the supported interface functions. An example structure for the Digital to Analog Converter (DAC), with the comments removed, is shown in the figure below. The API structure is what allows for modules to easily be swapped in and out for other modules that are instances of the same interface. For example, by using a common interface like that shown in the figure below, you can use the same interface for DAC units that are internal to the MCU and external to the MCU. Only the configuration structure determines which is used. The figure below shows how the common interface can be used to access both internal and external DACs.

```
typedef struct st_dac_api
{
  ssp_err_t (*open)(dac_ctrl_t *p_ctrl, dac_cfg_t const *const p_cfg);
  ssp_err_t (*close)(dac_ctrl_t *p_ctrl);
  ssp_err_t (*write)(dac_ctrl_t *p_ctrl, dac_size_t *p_value);
  ssp_err_t (*start)(dac_ctrl_t *p_ctrl);
  ssp_err_t (*stop)(dac_ctrl_t *p_ctrl);
  ssp_err_t (*versionGet)(ssp_version_t *p_version);
} dac_api_t;
```

Figure 6 DAC API structure

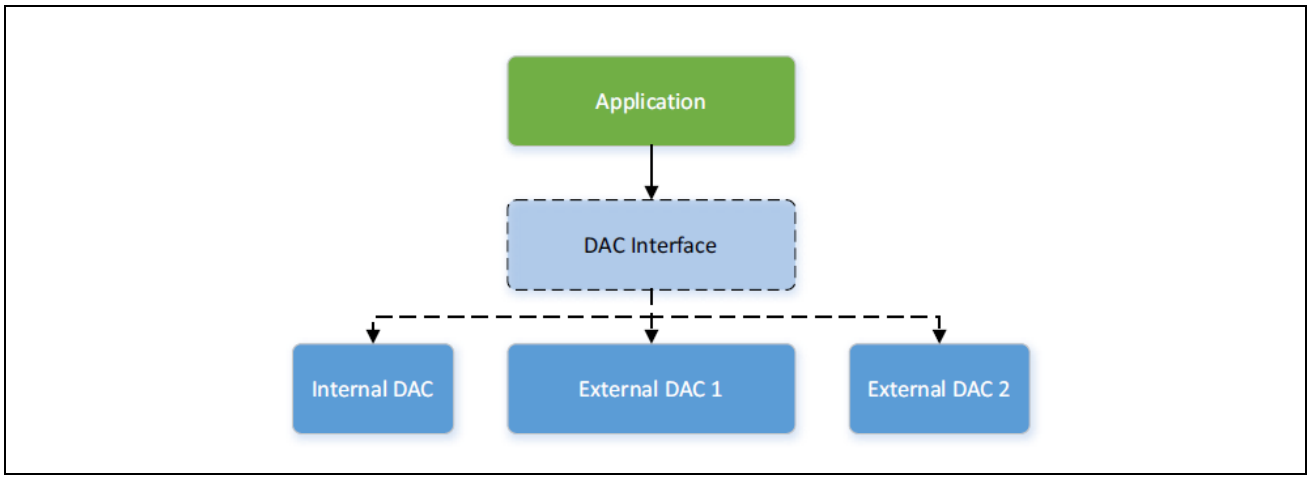

Figure 7 DAC interface diagram

Functions inside of the API structures follow common names. Most modules will have a pair of `open()` and `close()` functions. The `open()` function must be called before any of the other functions. The only exception is the `versionGet()` function that is not dependent upon any provided information you provide. There may also be modules that do not include an `open()` API such as the FMI module. It is always a good idea to refer to the module API and documentation to determine what calls are supported.

Other functions that are commonly found are `read()`, `write()`, `get()`, and `set()`. Function names are designed to be a noun followed by a verb. Example names include:

- `read(), write(), writeRead()`
- `statusGet()`
- `calendarAlarmSet(), calendarAlarmGet()`
- `accessWindowsSet(), accessWindowClear()`

3. Guidelines for using Frameworks and HALs

SSP comes with two predefined layers: The Driver layer and the Framework layer. The layers architecture is designed to maximize performance while still maintaining highly portable and reusable software. The layers are easily identifiable because the modules reside in different folders and have different prefixes. Driver layer modules are in the `ssp/src/driver` folder, while Framework layer modules are in the `ssp/src/framework` folder. Modules in the Driver layer start with a `r_ prefix`, while Framework layer modules start with a `sf_ prefix`.

The core difference in the functionality between the layers is that Driver layer modules are restricted to being peripheral drivers that are RTOS aware, but do not use any RTOS objects, or make any RTOS API calls. This means that Driver layer modules can be used in applications with, or without an RTOS. The driver modules provide developers with an abstracted, low-level control for the peripheral. Drivers should be used when:

- A RTOS is not present.
- Access to hardware is required.
  Note: Only write your own drivers if a suitable SSP driver does not exist.
- Performance is a critical requirement.

An example can be seen in the figure below for the external interrupt driver.

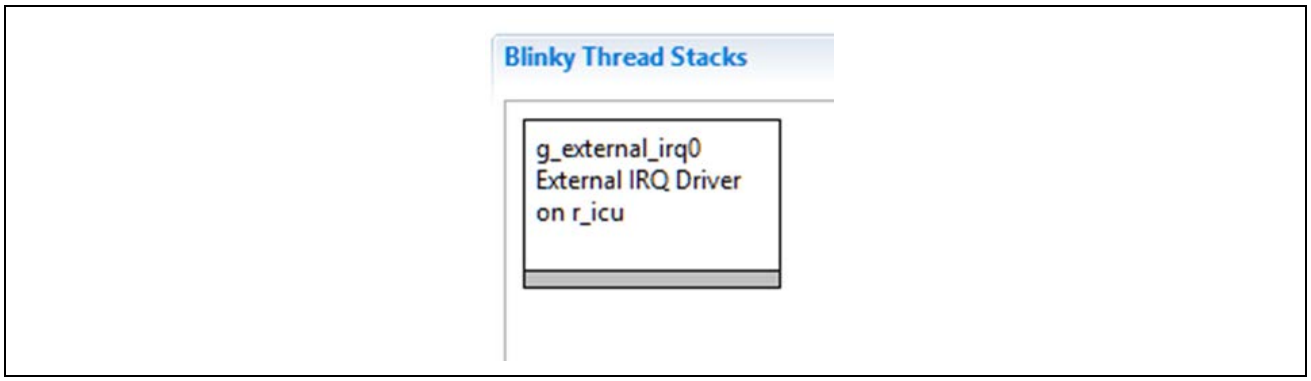

Figure 8 Driver module for external IRQ

Frameworks work at a higher abstraction level and may include driver modules as shown in in the figure below. When a framework uses a driver, the developer is not responsible for creating the driver callback function. Most frameworks will use the callback for their own purposes.

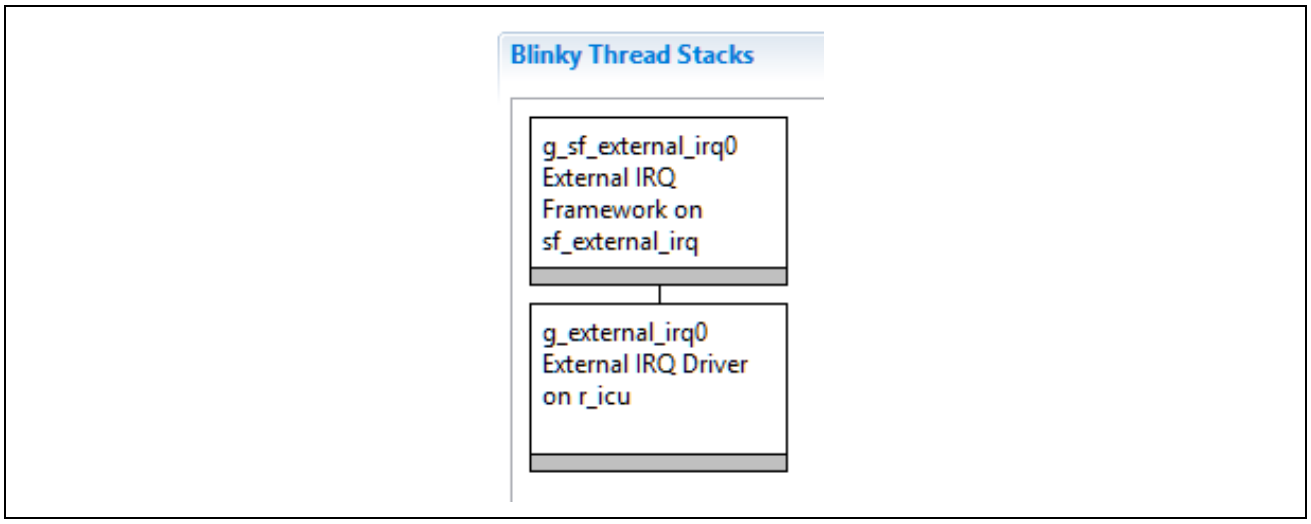

Figure 9 External IRQ framework

Framework layer modules are free to use RTOS objects such as semaphores, mutexes, or event flags. Framework modules may also create their own RTOS objects when needed. Framework layer modules that need to access hardware typically do so through a Driver layer interface. Exceptions can be granted in special cases where multiple peripherals need to be used together in a way that would not be practical through multiple individual interfaces.

3.1 SSP connecting layers

SSP modules are meant to be both reusable and stackable. It is important to remember that modules are not dependent upon other modules, but upon other interfaces. You are then free to fulfill the interface using the instance that best fits the application needs.

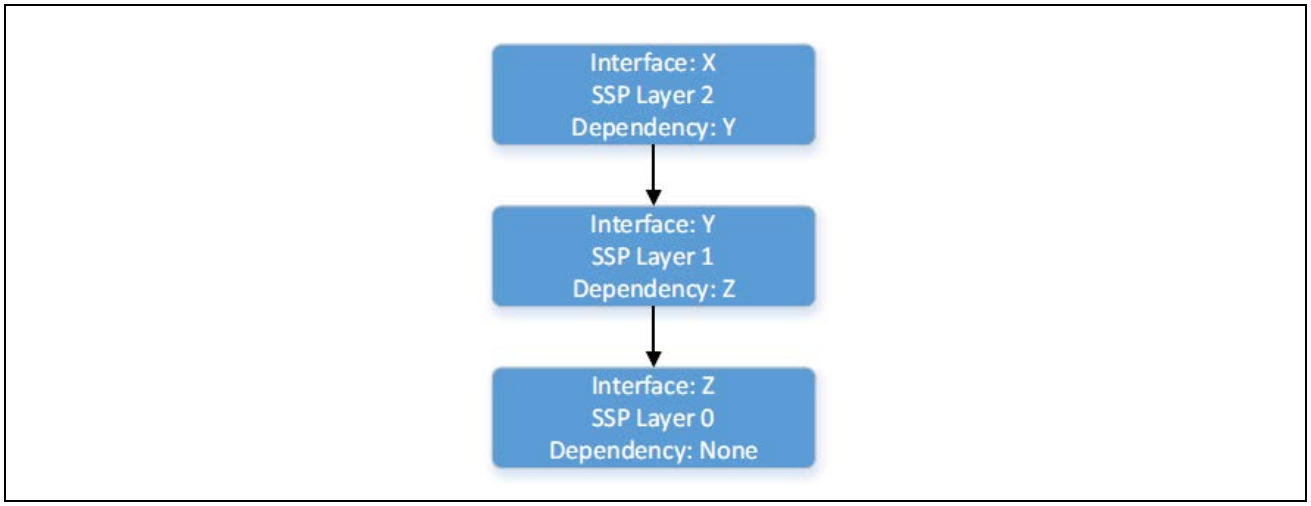

Figure 10 Connecting module layers

In the above figure, Interface Y is a dependency of Interface X, and has its own dependency on Interface Z. Interface X only has a dependency on Interface Y. Interface X has no knowledge of Interface Z. This is a requirement for ensuring that layers can easily be swapped out. This is shown in the figure below.

In this example, the Express Logic, Inc. FileX® file system is used on two storage mediums - SDMMC and SPI Flash. The SPI flash interface takes care of the SPI flash protocol but requires an SPI interface for actual SPI bus communications. The SDMMC interface takes care of the protocol and the bus communications, meaning that it does not have any dependencies.

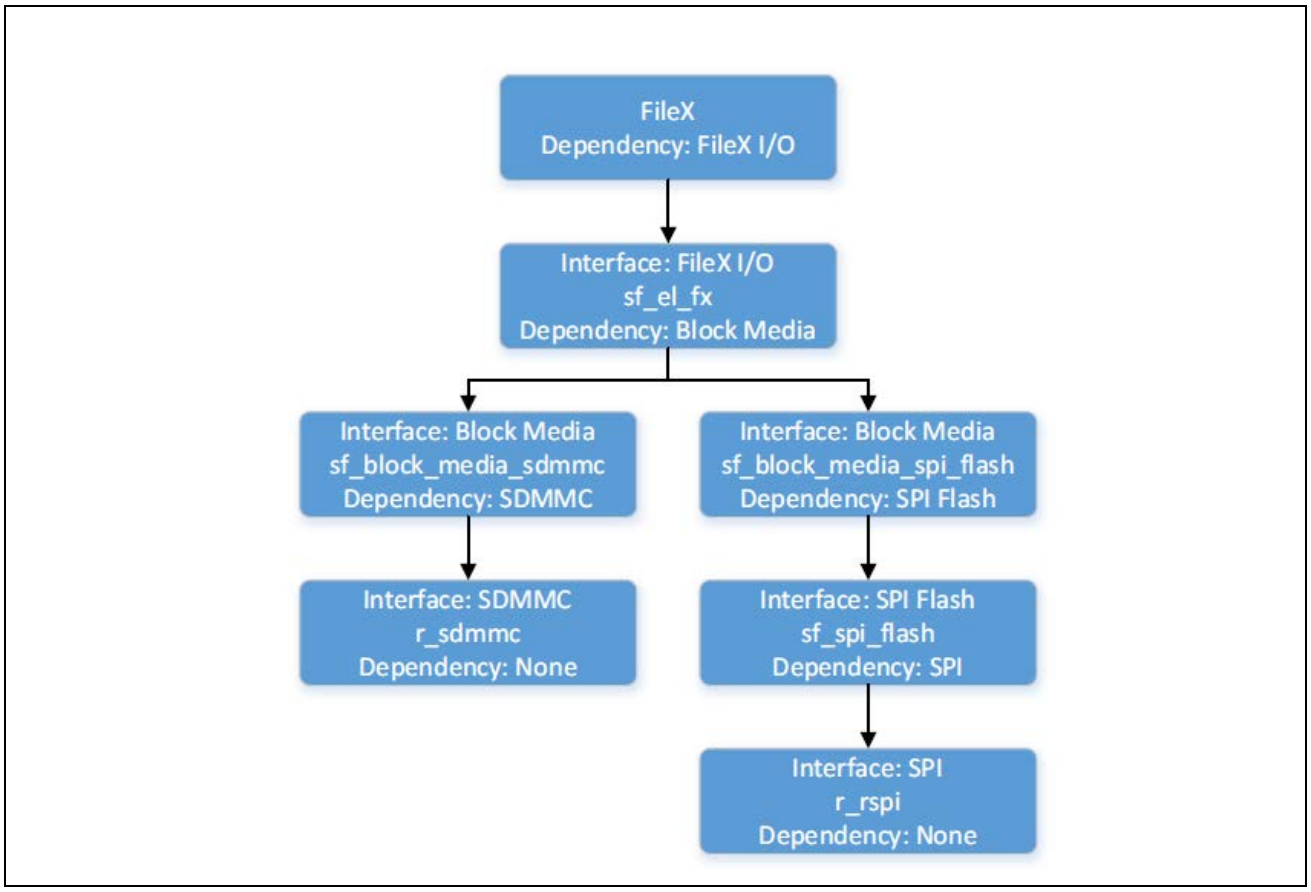

Figure 11 Connecting layers with the FileX interface

3.2 SSP architecture in practice

Each layer in the SSP stack is responsible for calling the API functions of its dependencies. In other words, modules are only responsible for calling the API functions at the layer where they are interfacing. Using the FileX example in the above Figure 11, you are only responsible for calling FileX functions in the application code. Internally, FileX then calls FileX I/O, that in turn calls a Block Media interface module. The Block Media interface can call multiple drivers. At a minimum, an upper layer module calls the `open()` function of the interface it depends upon.

To write an application using a module:

1. Determine the `open()` function to call. Dependencies are based upon interfaces, that means a module must have some way of discerning what instance to call.
2. Determine the configuration parameters. The module also needs to know what configuration information to pass down. In some cases, the module requires certain configuration parameters to be set. If this is the case, the module sets these configuration structure members itself before passing on the rest of the structure. The rest of the configuration structure members must be provided outside of the module.
3. Provide a control structure that is module instance specific and therefore can be allocated by the upper layer module.

Putting this all together, to interact with a module instance, the following pointers are needed:

- A pointer to the instance's API structure
- A pointer to the module instance's configuration structure
- A pointer to the module instance's control structure

Notice that the API structure is the only structure that is instance specific, not module instance specific. This is because the API structure will not vary between multiple uses of the same instance. If SPI is being used on SCI channels 0 and 2, then both module instances use the same API structure, while the configuration and control structures will vary.

To make module instances easier to use, these pieces are encapsulated in instance structures found in each interface. These structures have a standardized name of `<interface>_instance_t`. An example from the WDT interface is shown in the figure below. As can be seen in the watchdog example, each instance contains a pointer to a control, configuration and API structure. These are common to nearly every `<interface>_instance_t.`

```
/* Configuration for Thread Monitor framework */
typedef struct st_sf_thread_monitor_cfg
{
  wdt_instance_t * p_lower_lvl_wdt; /* Pointer to lower level watchdog instance */
  bool profiling_mode_enabled; /* Enables or disables profiling mode */
  UINT priority; /* Priority of thread monitor thread */
  void const * p_extend; /* Instance dependent configuration */
} sf_thread_monitor_cfg_t;
```

Figure 12 Watchdog instance

When using an instance, you should make use of the API interface along with the configuration and control structures. If you add the watchdog driver to your code, with a watchdog instance named `g_wdt0`, the figure below shows how the driver should be initialized using the instance structure. Notice how the watchdog instance is used to setup and use the interface. The open API always requires at least a `p_ctrl` and a `p_cfg` pointer.

```
g_wdt0.p_api->open(g_wdt0.p_ctrl, g_wdt0.p_cfg);
```

Figure 13 Opening the Watchdog instance

Upper layer modules that have a dependency on an interface can then use the instance structure to hold everything needed to interact with an instance of that interface. Continuing with the WDT example in the above figure. Figure 14 below shows the Thread Monitor Framework interface configuration structure. The Thread Monitor interface is dependent upon the WDT interface. The Thread Monitor module has everything it needs to work with the WDT interface in the `p_lower_lvl_wdt` structure member.

```
/* Interface Configuration */
typedef struct st_sf_block_media_cfg
{
  uint32_t block_size; /* Block size in bytes */
  void const * p_extend; /* Instance dependent configuration */
} sf_block_media_cfg_t;
```

Figure 14 Block Media instance

In some cases, module dependencies are not defined in the interface, but instead in the instance. An example is the Block Media interface, that could be implemented on SDMCC, SPI flash, or many other instances. Because of the wide range of implementations, the instance structure for an interface cannot be used directly in the Block Media interface's configuration structure. There are no instance structure pointers provided. The reason for this, as previously mentioned, is that the Block Media interface is too generic to enforce a dependency upon an interface.

When a module is an instance of a generic interface, such as Block Media, and it has dependencies on other modules, the module puts the lower-layer pointers in an extension structure that is referenced through the interface's `p_extend` configuration member. This is required to allow module stacking, while not forcing interfaces to expand and have many optional configuration members as can be seen in the figure below.

```
typedef struct st_block_media_on_sdmmc_cfg
{
  sdmmc_instance_t const * const p_lower_lvl_sdmmc; /* Pointer to SDMMC instance structure */
} sf_block_media_on_sdmmc_cfg_t;
```

Figure 15 Block Media configuration

There are several pointers that you should follow to ensure that they interact with the SSP correctly and efficiently. These include:

- Accessing SSP modules through interfaces, and NOT instances
- Using generated instances (not hand-coded instances)
- Not modifying SSP **owned** resource externally
  - Not manually altering peripheral registers
- Creating the following objects using the Synergy Configurator:
  - Threads, mutexes, semaphores, message queues

Developers may be tempted to try to use the instance rather than the interface. It is easy to tell the difference as shown in the figure below.

```
R_SCI_UartOpen(&uart_ctrl,
               &uart_cfg);
g_uart0.p_api->open(g_uart0.p_ctrl,
                    g_uart0.p_cfg);
```

Figure 16 Instance (wrong) versus interface coding (right)

3.2.1 Using SSP modules

You can follow the standard procedure for using a SSP module in their application. The process that engineers should follow to use a SSP module can be seen in the below figure.

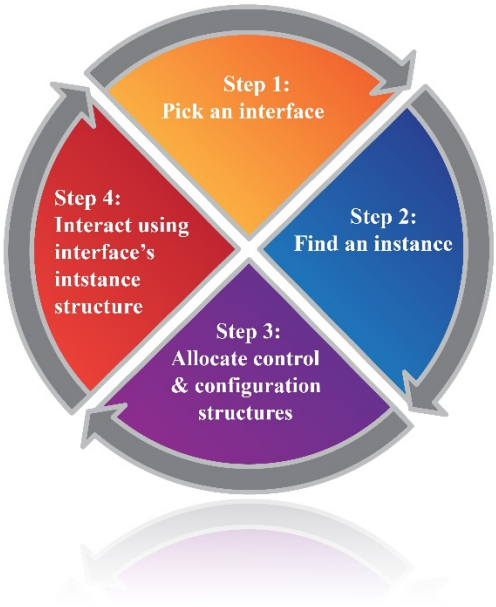

Figure 17 Using a SSP module process

3.2.2 Step 1 - Pick an interface

Start by choosing an interface for the functionality that is required. For example, for UART communications, use the UART interface.

3.2.3 Step 2 - Find a suitable interface instance

After choosing an interface, select a suitable instance. The list of known instances of an interface is listed in the documentation comments for an interface. Include the header file of the selected instance in the source file of the application that uses the instance.

3.2.4 Step 3 - Allocate control and configuration structures

The e^2 studio ISDE provides a graphical user interface for setting the parameters of the interface and instance configuration structures. The ISDE also automatically includes those structures, once they are configured in the GUI, in application-specific header files that you can include into your application code.

The configuration and control structure types follow standard names of `<interface>_ctrl_t` and `<interface>_cfg_t`, respectively. The ISDE allocates storage for both structures in the application specific header files, which the ISDE creates. Use the **ISDE Properties** view to set the values for the members of the configuration structure as needed. Many members will be typed enumerations, in which case, the enumeration can be referenced for available options.

If the interface has a callback function, then you first need to declare and define the function in your source code. The return value is always of type void, and the parameter to the function is a typed structure of name `<interface>_callback_args_t`. Once the function has been defined, assign its name to the `p_callback` member of the configuration structure. If any context information is required in the callback, you can provide a pointer to the `p_context` member. You can assign callback function names through the **ISDE Properties** window for the selected module.

Refer to the instance documentation to see if an interface extension is provided. If so, it will be found in the instance's header file and named `<interface>on<instance>_cfg_t`. It may have several members just like the interface's configuration structure. These members can be set using the ISDE.

3.2.5 Step 4 - Interact using interface's instance structure

Once the instance structure has been created, you can interact with the instance as needed. The figure below is code that builds up an instance structure for the UART interface as implemented on SCI.

```
/* Include the header file of the Instance. */
/*This will in turn include the r_uart_api.h Interface */
#include "r_sci_uart.h"

/* Allocate Instance specific control structure. This will be used
in place of uart_ctrl_t. */

sci_uart_instance_ctrl_t my_uart_ctrl;

/* Setup extended UART configuration on SCI. */

uart_on_sci_cfg_t my_uart_extended_cfg =

{

  /* Set extended configuration members... */

};

/* Configure standard UART Interface. */

uart_cfg_t my_uart_cfg =

{

  .data_bits = UART_DATA_BITS_8,

  /* Continue configuring other members... */

  .p_extend = &my_uart_extended_cfg

};

/* Setup instance structure */

uart_instance_t my_uart = {

  .p_ctrl = &my_uart_ctrl,

/* Extended configuration is brought through in p_extend */
.p_cfg = &my_uart_cfg,

.p_api = &g_uart_on_sci *

};
```

Figure 18 UART instance implementation

Now that the instance structure is ready, you can interact with the UART interface. In e^2 studio, the name of the instance structure is the name that you provide when configuring the module instance in the **ISDE Properties** window. Using the instance is shown in the figure below.

```
ssp_err_t err;
/* Initialize UART */
err = my_uart.p_api->open(my_uart.p_ctrl, my_uart.p_cfg);
/* Check return for errors. */
if (SSP_SUCCESS != err)
{
/* Handle error. */
}
/* Use other Interface functions. */
err = my_uart.p_api->write(my_uart.p_ctrl, ...);
err = my_uart.p_api->read(my_uart.p_ctrl, ...);
```

Figure 19 Using the UART instance

3.3 Rules for selecting a HAL or Framework interface

The Renesas Synergy Platform is designed to be flexible so that developers can select to use the high level SSP or directly access low level drivers using the HAL. The question that many developers may ask is **Should I use the SSP Framework or use the HAL?** The answer is not necessarily straightforward.

The general recommendation is that developers should by default use the SSP Framework. The SSP Framework is RTOS aware, which means that framework already knows that it is operating within a multithreaded environment. The SSP Framework also uses RTOS services such as mutexes and semaphores to coordinate access to shared resources to ensure that there are no resource contentions.

There are times when a developer may decide that the SSP Framework should be bypassed and the HAL used instead. The primary reason for doing this is that the component under development has strict timing requirements and the performance and overhead of using the SSP Framework would interfere with the component operation.

3.4 API guidelines

If you are interested in creating their own components you will want to create API's for their own component that are like those used in the SSP. The table below provides a list of recommended API names and their purpose. Specific namespace prefixes (Namespaces) should be added at each layer and for each module. For reference, you can see the existing drivers/modules like the ones you are working on. All APIs must follow the naming convention documented in the Coding Standard. Consistency of APIs must be maintained and common sense must be used. All APIs must be reviewed and approved. If additional review and approval comments are made, those need to be discussed and approved even before being considered for use. Example API's can be found in the table below.

Table 2 Example module APIs

Main Services	Description
`open/close`	Makes module/peripheral available for use by initializing, configuring, allocating resources, locking, and so forth, as needed, and frees up the module/peripheral by resetting, freeing resources, and so forth, as needed.
`read/write`	Reads/writes status, condition, and so forth for a module/peripheral. May be used to read and write to storage devices, files, I2C, SPI, and so forth.
`writeRead`	For writing and then reading in a single transaction as supported by device such as the SPI. Allows different lengths for read and write, but in our case, we will use only one length for write and read (use max and ignore the additional information).
`start/stop`	Starts operation after Opening (for example, start timer) and Stops operation without closing (for example, stop timer)
`enable/disable`	Enable/Disable certain functionality (for example, DMA, Intr).
`set/clear /get`	May be used to set/clear specific state, mode, configuration, and so forth (for example, clear interrupt, set baudrate).
`versionGet`	Provide version of module/peripheral API and code.

Main Services	Description
`reset/blankCheck`	Examples of other APIs that should be used to expose driver functionality that does not fit standard functions above.

4. ThreadX® and Messaging Best Practices

Working in a multithreaded environment is slightly different than working in the standard super loop environment that many embedded software developers are familiar with. When multiple threads are required, using a Real-time Operating System (RTOS) is often the best method to reduce complexity, improve maintainance, and ensure that the system operates reliably. For a developer unfamiliar with design using a RTOS, the book *Real-time Embedded Multithreading using ThreadX* by Edward Lamie is highly recommended.

4.1 General RTOS best practices

Following are a few best practices that you should keep in mind, while designing using an RTOS:

- Avoid dynamic memory allocation. Statically allocate memory. On microcontrollers with small RAM footprints, you may instead use ThreadX memory byte or block pools.
- Create tasks and allocate objects at start-up so that they appear to be statically allocated. On small RAM footprint devices, allocating memory upfront may not be possible. In these cases, you should consider using the ThreadX memory block pool.
- Do not create and destroy objects during application execution as this can lead to heap fragmentation.
- Use memory block pools in preference to memory byte pools. Memory byte pools are prone to fragmentation while memory block pools are not.
- Assign static priority to tasks and do not dynamically change them at run-time. Managing task priorities at run-time can result in confusion, inappropriate settings, and bugs.
- Minimize the number of priorities that are allowed in the system. ThreadX by default allows 0 -31, but can be configured to as many as 0 – 1023 bytes.
- Enable time-slicing when creating multiple tasks at the same priority level. This will provide a round-robin scheduling approach where each task will get the CPU for a time slice.
- The default `TX_MINIMUM_STACK` is probably too small for any given task. A rough rule of thumb is to start with 1024 bytes.
- Perform a worst-case stack analysis to properly size a stack. Alternatively, fill the stack with a known value and monitor how close it comes to overflowing. Use the **RTOS Resources View** in the ISDE (**Renesas Views** > **Partner OS** > **RTOS Resources View**) to get a good indication of thread stack usage.
- Avoid excessive stack usage in tasks. For example, recursive functions and large local structures should be avoided.
- Functions that are called in multiple threads must be re-entrant. To achieve re-entrancy functions must do the following:
  - Store callers' return address on the stack instead of in a register.
  - Cannot use global or static variables that are altered during the function call.
- To optimize performance and minimize code size, the following should be adhered to:
  - Minimize the number of threads in an application. Minimizing the thread number is not always desirable because tasks provide a way to decompose the system, increase reuse and maintenance. In small memory devices though, this may be necessary.
  - Select task priorities carefully so that issues such as priority inversion and thread starvation are prevented.
- Event flags can be used to synchronize tasks and have less overhead than semaphores and mutexes.
- Pass data greater than 16 words through a message queue by using a pointer.

4.1.1 Synchronization and communication recommendations

Using an RTOS with multiple threads will most likely result in those threads needing to synchronize and share resources at some point in time. ThreadX provides multiple methods that are available in most RTOSes to synchronize resources and threads such as mutexes, semaphores, event flags, and message queues. The table below is an excerpt from Real-Time Embedded Multithreading using ThreadX, that shows the recommended resource to use (far left column) with the synchronization need (top row).

Table 3 Recommended ThreadX resources (Source Multithreading using ThreadX)

	Thread Synchronization	Event Notification	Mutual Exclusion	Inter-Thread Communication
Mutex			**Preferred**	
Counting Semaphore	OK – better for one event	Preferred	OK	
Event Flags Group	Preferred	OK		
Message Queue	OK	OK		Preferred

Developers often get confused about the differences between a mutex and a counting semaphore. Once again, the ThreadX book has a great table that shows the differences between mutexes and semaphores in areas such as speed, ownership, and so forth, that is shown in the table below.

Table 4 Comparing Mutex and Counting Semaphore Use and Behavior

	Mutex	Counting Semaphore
Speed	Somewhat slower than a semaphore.	A semaphore is generally faster than a mutex and requires fewer system resources.
Thread Ownership	Only one thread can own a mutex at a given time.	No concept of thread ownership for a semaphore – any thread can decrement a counting semaphore if its current count exceeds zero.
Priority Inheritance	Available only with a mutex.	Feature not available for semaphores.
Mutual Exclusion	Primary purpose of a mutex – a mutex should be used only for mutual exclusion.	Can be accomplished with the use of a binary semaphore, but there may be pitfalls.
Inter-Thread Synchronization	Do not use a mutex for this purpose.	Can be performed with a semaphore but an event flags group should also be considered.
Event Notification	Do not use a mutex for this purpose.	Can be performed with a semaphore.
Thread Suspension	Thread can suspend if another thread already owns the mutex (depends on value of wait option).	Thread can suspend if the value of a counting semaphore is zero (depends on value of wait option).

4.1.2 Avoiding RTOS Issues and Debugging

Using a RTOS does not guarantee that an embedded system will not have any issues. Many issues occur if you're not careful. Following is a list of common RTOS issues and ways that you can avoid.

Table 5 List of common RTOS issues

Issue	Solution
Priority Inversion occurs when a higher priority task is delayed by a lower priority task that has access to a system resource that the higher priority task needs. The priority inversion can become deterministically unbounded especially if there are other tasks in the system that can interfere with the lower priority task from completing.	The solution is to use the **mutex priority inheritance** feature included in ThreadX. When a task is blocked by a mutex with priority inheritance, the lower priority task is promoted to the same task priority as the task waiting for the mutex. Priority inheritance ensures that the task completes its critical section as quickly as possible, freeing the resource and allowing the higher priority task to resume. Alternatively, **preemption hold** can be used and proper **task priority selection** can be used.
Deadlock occurs when two tasks need access to two or more resources to proceed, but each task has only one of the resources. Since neither task can get the needed resource, they are suspended indefinitely.	There are two recommendations for preventing deadlock in a system. First, in any tasks that must acquire more than a single resource, have the tasks acquire those resources in the same order. This prevents two tasks from each getting a resource and then being unable to acquire the second resource. Use timeouts and release the first resource if the second resource could not be acquired.
Thread Starvation occurs when a low priority thread is rarely executed due to higher priority tasks always using the CPU.	Following good design practices and using Rate Monotonic Analysis (**RMA**) can help in minimizing thread starvation. Developers can also set a **preemption threshold** on a task that sets the ceiling on the priority level that can preempt the task. Developers could also use dynamic thread prioritization to occasionally raise the lower thread's priority but this is not a recommended best practice.
Program Counter Corruption commonly occurs when a threads stack overflows.	Fill task stack memory with a known value, and if program counter corruption occurs, review which task overflowed. Increase the stack size, **RTOS Resources** view in the ISDE (**Renesas Views** > **Partner OS** > **RTOS Resources View**) to get a good indication of thread stack usage.

4.1.3 Memory best practices

Real-time operating systems provide many benefits such as decreasing complexity and improving code maintainability, but if you're not careful, they can quickly find themselves out of memory or can run into bottlenecks. Following are a few tips for you on how to minimize memory footprint and improve throughput.

- Perform your design up front so that the appropriate number of threads, mutexes, semaphores, and other objects are used. Unnecessary use of RTOS features can unnecessarily eat into the available system RAM.
- Every thread has a stack. Perform a worst-case stack analysis and appropriately size the stack.
- Minimize the use of RTOS services such as mutexes, semaphores, event flags, and message queues. Once again, all these have a control block associated with them that will use some memory overhead to manage the resource.
- Memory block pools are preferred to byte pools because they are faster and do not have the ability to become fragmented. Memory byte pools can be used if the resource will be allocated once and not released back to the pool. Consider a memory byte pool to be like the heap.
- When adding a component to the SSP stack, at the lowest levels, there will be options to select either `r_dmac` or `r_dtc`. `r_dmac` is the direct memory access controller, which contains registers and a limited number of channels that can be used to move data around the microcontroller without the CPU's intervention. The `r_dmac` is slightly faster than the `r_dtc`. The `r_dtc` data transfer controller stores its information in SRAM. You can

create a nearly unlimited number of `r_dtc` instances, that are only limited by the amount of RAM on the MCU. You need to take care when deciding to use the `dmac` over the `dtc`.

4.1.4 Defining ThreadX resources

In a normal application, you would directly write the code that would create and set the properties for RTOS tasks, semaphores, mutexes, and message queues. Using the SSP should instead use the Synergy Configuration tool (`configuration.xml`). The configurator contains a **Threads** tab for creating tasks with associated SSP stacks and synchronization objects. The messaging tab contains the ability to create message queues and assign them to system tasks. An example can be seen in figure below. You simply click the green addition button, select the object, and then adjust the properties as needed. When the configuration is completed and then generated, the necessary objects are automatically created for developers. Figure 21 shows an example of the code that was generated for a thread named Main Thread, that contained a single semaphore for synchronization. A developer did not have to create this code but instead, added the object and configured it in the Synergy Configurator. Using a configurator in this way drastically simplifies using an RTOS with an embedded system. All RTOS related objects can be found in a single place and be easily adjusted as necessary through-out the development and maintenance cycle.

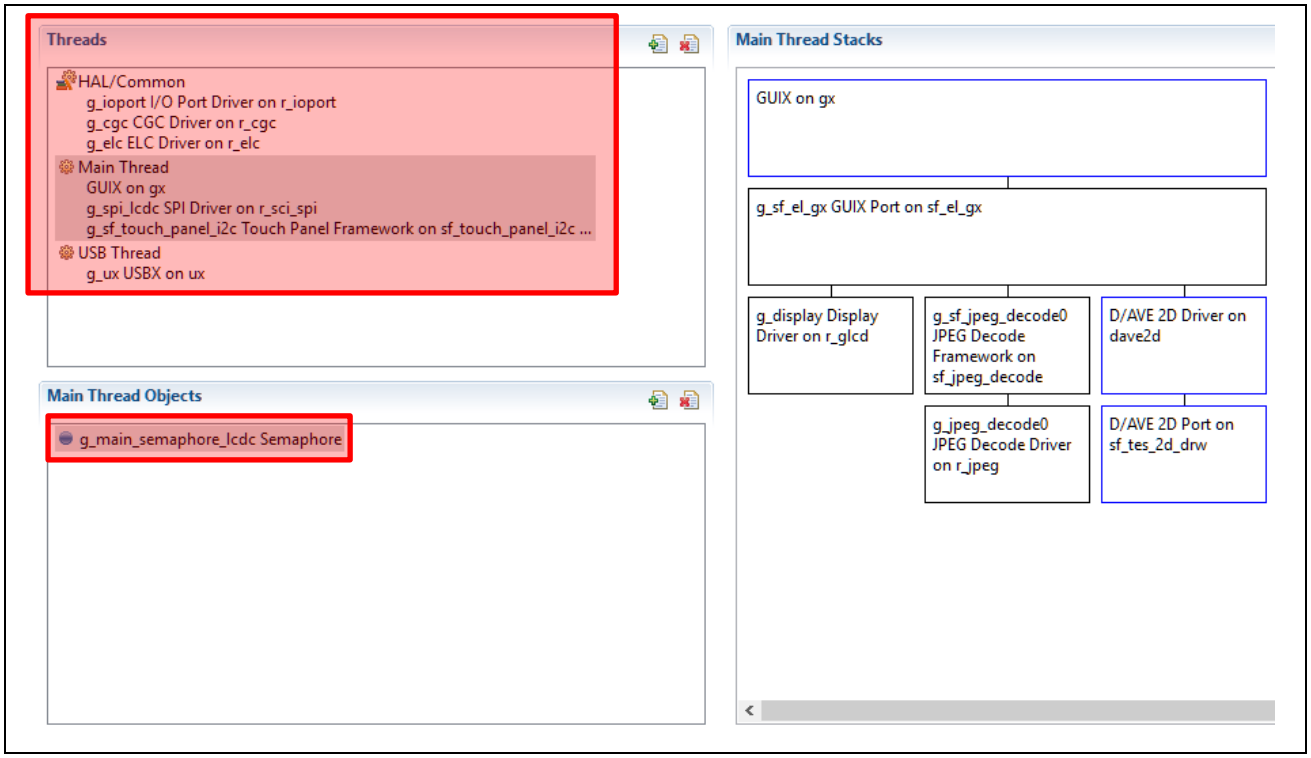

Figure 20 ThreadX object creation on the Threads tab

```
void main_thread_create(void)
{
    /* Initialize each kernel object. */
    tx_semaphore_create (&g_main_semaphore_lcdc, (CHAR *) "Main Semaphore", 0);

    tx_thread_create (&main_thread, (CHAR *) "Main Thread", main_thread_func, (ULONG) NULL, &main_thread_stack, 2048, 6,
                      6, 10, TX_AUTO_START);
}
```

Figure 21 Automatically generated ThreadX objects

5. Using and Creating Synergy Configurators

Creating and using Synergy Configurators provides you with a way to visualize the software that is being developed. It is recommended that you configure and create objects as much as possible through the Synergy Configurator rather than hand code these objects. The configurator provides:

- Ease-of-use
- Configurable properties presented graphically
- Less support issues

6. Generating a Distribution Package

The SSP and add-on software associated with the Synergy Platform are distributed in a pack format. The packs are a convenient way to collect and distribute software. Packs are located within the installation directory located under `\e2_studio\internal\projectgen\arm\Packs`. Custom components and frameworks should be distributed in the same manner. Exporting a custom pack is relatively straight forward since the release of e^2 studio 5.2.1.

To create a pack file, open the project that contains the add-on code. From the **File** menu, select **Export**. The figure belowshows the dialog box that is presented. Under **General**, select the **Renesas Synergy User Pack** option and click **Next**.

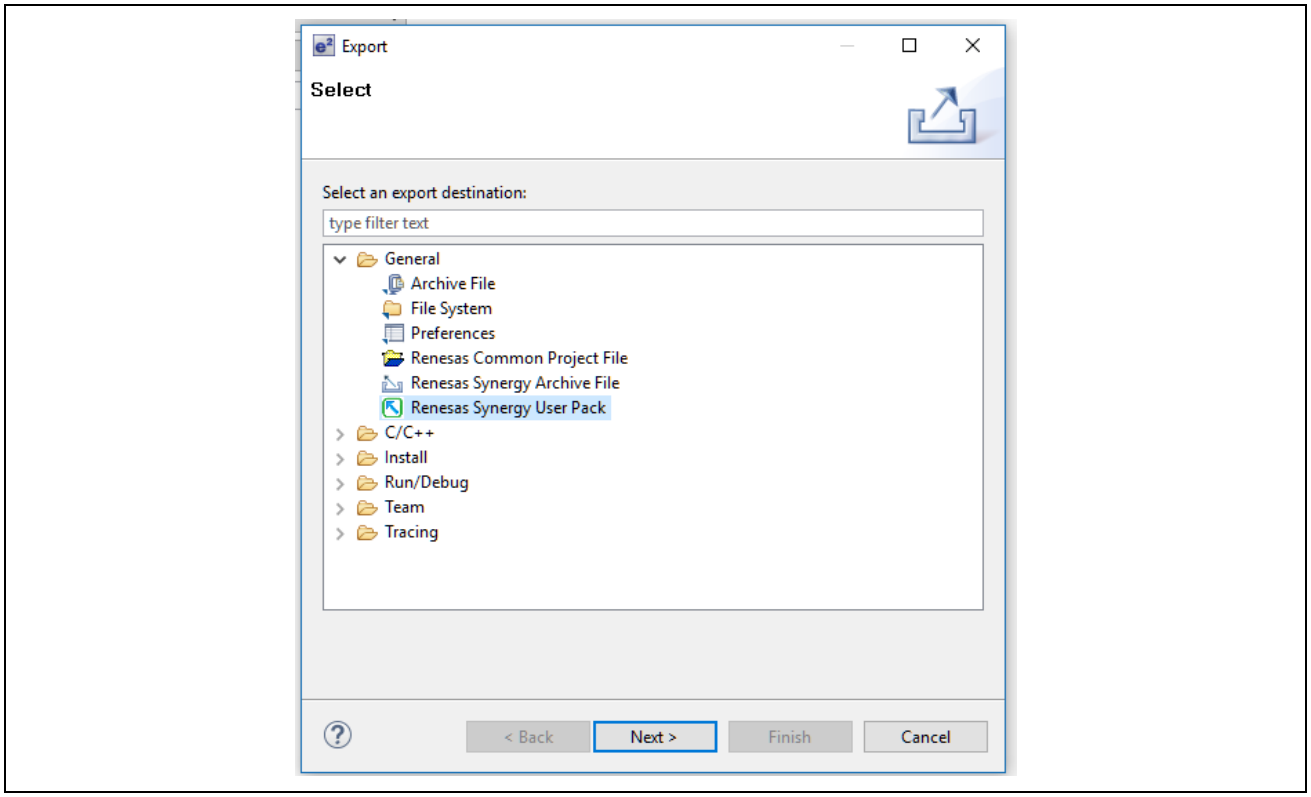

Figure 22 Exporting to a Pack

The next dialog box that is displayed gives you the opportunity to provide not only the pack name but also a variety of other parameters about the pack such as the version, description, and even contact information. At this stage, a developer will want to fill in all the details that are shown in the following figure.

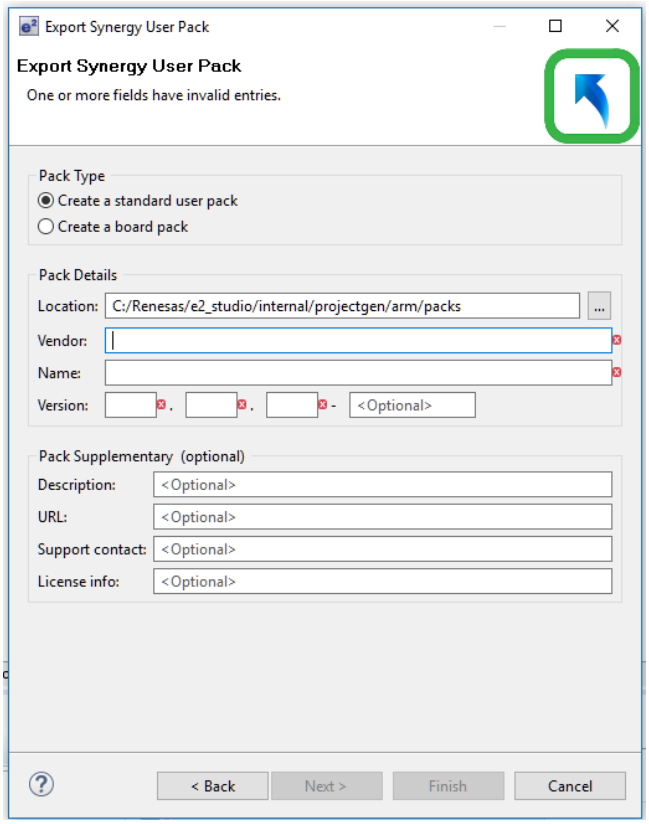

Figure 23 Setting the Pack information

Once those details have been filled in, you can now select the components, threads, and messaging components that will be included in the pack as shown in the figure below. The green plus in the component selection box can be used to filter and review available components for export.

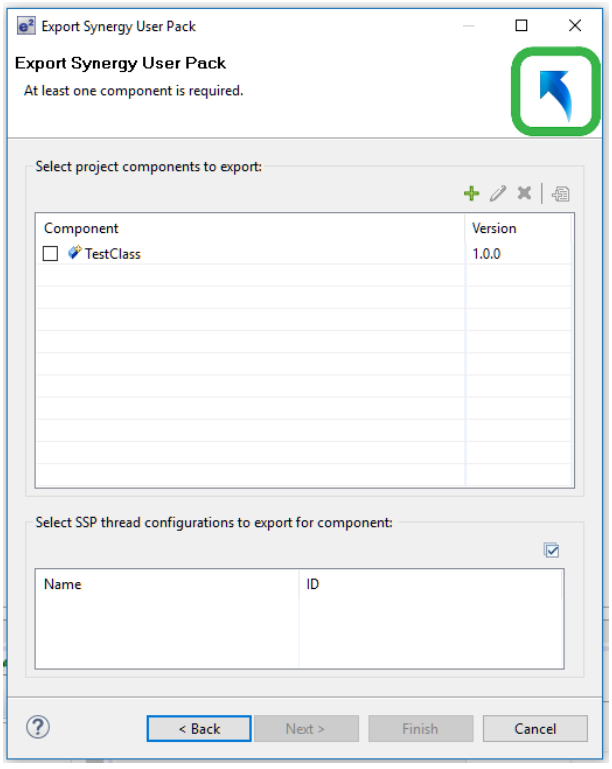

Figure 24 Adding Components to Export

Once the components have been selected, choose **Finish** to generate the pack. Once the pack has been generated, it can be directly distributed to other Synergy Platforms, that you can add to your projects.

7. Optimized Software to Interact with SSP

This section provides you with a checklist that can be used to verify smooth interactions with Renesas Synergy SSP.

7.1 Component organization

You should organize their components and examples so that:

- Demo code is separate from software port and/or component(s)
- The port is in a separate folder
- Code is located under `synergy/<company_name>/`

The preferred organization that you should follow can be seen in the figure below.

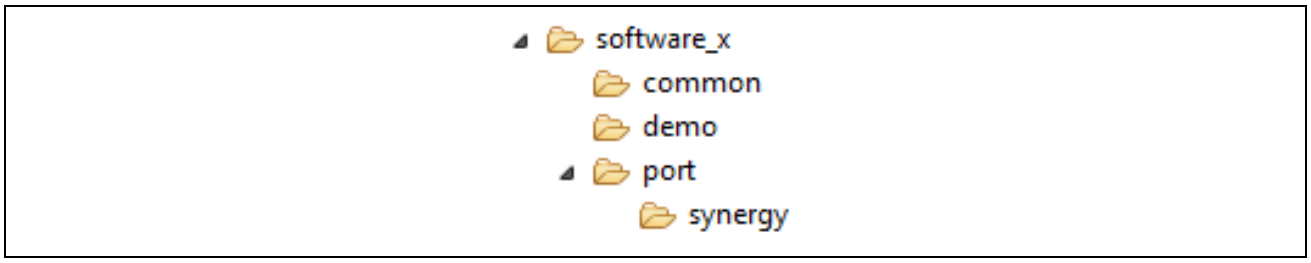

Figure 25 Recommended organization

7.2 Code behavior and implementation

There are many little details that you need to consider when creating their application. First, you will need to make sure that your code works in the SSP / ISDE environment. To work within the Synergy environment, you will need to:

- Create header files with macros, typedefs, and so forth
- Create runtime initialization code that runs in a thread context
- Create data structures with initialized values
- Add source code to a project
- Add include files to a project
- Add libraries to a project and to the linker
- Make sure the module does NOT
  - modify the BSP sequence
  - handle reset vector
  - define its own `main`.
- Use RTOS objects and services correctly including
  - Making functions re-entrant
  - Avoiding `malloc` and `free`
  - Setting appropriate stack sizes

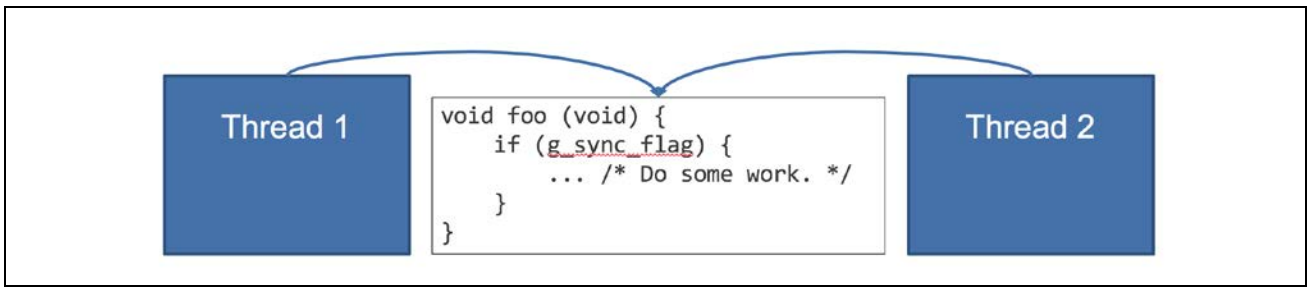

Figure 26 Functions must be reentrant

The code must meet the following conditions:

- Builds under the ISDE of choice, either e^2 studio with the default toolchain (GCC) using the configuration supported by SSP or under IAR with the default toolchain (IAR Embedded Workbench® for Renesas Synergy™) using the configuration supported by SSP.

To maximize portability and compatibility, the code should not reinvent the wheel or override any SSP functionality. You should:

- Use the built-in drivers and frameworks that are included in the SSP
- Use the components and drivers that are currently in place and supported by Renesas. Only customize drivers or lower level components when features are missing from the Renesas supplied drivers.
- Use the Messaging Framework and ThreadX objects to communicate between tasks and components.

Creating threads is an acceptable practice but developers need to be careful about how they use internal threads. When using an internal thread, you will need to:

- Set the stack size intelligently. For example, the need to perform a worst-case stack analysis, run the code under its greatest stress, and use the RTOS viewer capabilities to monitor the minimum, average, and maximum stack usage, and then apply some additional **buffer**s to prevent overflow.
- The documentation needs to make it clear to you that the thread exists and what the default settings are. This could be done using a combination of tables, ThreadX traces, and configurator does screen shots.
- The internal thread priority must be configurable. You need to have the ability to adjust thread priorities based on their RMA analysis for the system.

8. Appendix - The Renesas Coding Standard

There are no requirements that you adopt the Renesas Coding Standard that was used to develop the SSP and associated framework software. Every development team has different guidelines as to what is deemed to be an acceptable subset for C/C++ along with the programming style that is used by developers. What is important is that development teams agree beforehand on what the standard will be and that every developer adheres to that standard. To help teams along, this section provides the reader with a few best practices from the Renesas Coding Standard.

8.1 Coding standard template

The Barr Group **Embedded C Coding Standard** was used as the basis for the original Renesas coding standard (document RWS-1020). More information can be found at http://www.barrgroup.com/Coding-Standard-Muse. The standard provides recommendations for best practices concerning acceptable C constructs to get you started.

8.2 Which C?

Rules

- All programs shall be written to comply with the ISO/IEC 9899:1999 (i.e. C99) C programming language standard.

Guidelines

- The use of proprietary compiler language keyword extensions, #pragmas, and inline assembly should be kept to the minimum necessary to get the job done.

Reasoning

Even **standard C** varies by compiler, but we need as common of a platform as we can find to make possible the rules and enforcement mechanisms that follow. C++ is a different language and the use of C++ and C should not be mixed in the same design.

Exceptions

In the case of necessary (and tested, and documented) use of `#pragma` or `__attribute__` (for example, to place code in a specific linker section), a local exception can be approved. That is when the other rules kick in. In the case of an approved exception, add an explanatory comment, and a comment `/*LDRA_INSPECTED 293 S */`, above the use of the compiler extension like this:

```
/* SSP requires alignment support and there is no better option than
 * to use an attribute in GCC. */
/*LDRA_INSPECTED 293 s */
#define BSP_ALIGN_VARIABLE(x)  __attribute__ ((aligned (x)))
```

8.3 Braces

Rules

- Braces shall always surround the blocks of code (that is, compound statements), following `if`, `else`, `switch`, `while`, `do`, and `for` statements. Single statements and empty statements following these keywords shall also always be surrounded by braces.
- Each left brace (‘{‘} shall appear by itself on the line below the start of the block it opens. The corresponding right brace (‘}‘) shall appear by itself in the same position the appropriate number of lines later in the file.
  - Exception: The right brace (‘}’) may be followed by a type name during a typedef declaration.

Examples

 Valid	`if (var1 == var2)`
	`{`
	`    ... (empty or may contain lines of code)`
	`}`
	`else if (var1 > var2)`
	`{`
	`    ...`
	`}`
	`else`
	`{`
	`    ...`
	`}`

 Invalid	`if (var1 == var2){` ← **`Braces must be on their own line`**
	`    ...`
	`}`

Reasoning

There is considerable risk associated with the presence of empty statements and single statements that are not surrounded by braces. Code constructs like this are often associated with bugs when nearby code is changed or commented out. This risk is entirely eliminated by the consistent use of braces. The placement of the left brace on the following line allows for easy visual checking for the corresponding right brace.

8.4 Casts

Rules

- Each cast shall feature an associated comment describing how the code ensures proper behavior across the range of possible values on the right side.

Examples

<table>
<tr>
<td> 
Invalid</td>
<td>

```
uint8_t abs (int8_t arg)
{
    return ((arg < 0) ? –arg : arg);
}

uint8_t y;
y = adc_read();
z = abs((int8_t) y);
```

← **‘y’ is unsigned and therefore could have a value such as ‘240’ that would not fit in a signed 8-bit variable. If you know that this is safe for some reason then explain it in a comment before the cast.**
</td>
</tr>
<tr>
<td> 
Valid</td>
<td>

```
/* Function Prototype */
uint8_t R_EXAMPLE_FooBar(uint16_t arg);

uint8_t * ptr;

...

/* Casting the pointer to a uint16_t type is valid because even
though
   the Renesas Compiler uses 4 bytes per pointer, this pointer
will never
   reference memory over 0xFFFF. */
result = R_EXAMPLE_FooBar((uint16_t)ptr);
```

</td>
</tr>
</table>

Reasoning

Casting can be dangerous. In the example above, unsigned 8-bit **y** can take a larger range of positive values than a signed 8-bit value. In that case, the absolute value will be incorrect as well. The above cast was likely used to silence an important warning about possible loss of precision.

Exceptions

None

8.5 Keywords that should be used

Rules

- The `static` keyword shall be used to declare all functions and variables that do not need to be visible outside of the module where they are declared.
- The `const` keyword shall be used whenever appropriate. Examples include:
  - To declare variables that shall not be changed after initialization
  - To define call-by-reference function parameters that shall not be modified (for example, `char * const` param)
  - To define fields in structs and unions that shall not be modified (for example, in a struct overlay for memory-mapped I/O peripheral registers)
  - As a strongly typed alternative to #define for numerical constants.

- The volatile keyword shall be used whenever appropriate. Examples include:
  - To declare a global variable accessible by any interrupt service routine
  - To declare a global variable accessible by two or more tasks
  - To declare a pointer to a memory-mapped I/O peripheral register set (for example, `timer_t volatile * const p_timer`),
  - To declare a pointer to shared memory
  - To declare a delay loop counter

Examples

 Valid	`/* As an alternative to '#define TEST_VALUE (3.14)' */`
	`const float TEST_VALUE = 3.14f;`
	
	`/* If the #define above was used then it is possible that double-precision`
	`math would be used because the compiler would automatically have chosen`
	`the larger data type. */`
	`float my_pieces_of_pi = TEST_VALUE * (float)2.0;`

Reasoning

C's static keyword has several meanings. At the module-level, global variables and functions declared static are protected from external use. Heavy-handed use of static in this way decreases coupling between modules. The `const` and `volatile` keywords are even more important. The upside of using `const` as much as possible is compiler-enforced protection from unintended writes to data that should be read-only. Proper use of `volatile` eliminates a whole class of difficult-to-detect bugs by preventing compiler optimizations that would eliminate requested reads or writes to variables or registers.

Exceptions

None

8.6 Build warnings

Rules

- Projects shall build with no warnings or errors.

Reasoning

Customers expect our code to work right out of the box. If our projects have warnings or errors, then customers lose confidence in our code.

Exceptions

If a project generates a warning, and if the warning is invalid, then a comment should be placed above the code that generates the warning stating why the warning is invalid.

Enforcement

These rules shall be enforced during code reviews.

8.7 Naming conventions – types and variables

Rules

- The names of all new data types, including structures, unions, and enumerations, shall consist only of lowercase characters and internal underscores and end with **_t**.
- All new structures, unions, and enumerations shall be named using a typedef.
- All structure and union member names shall consist of lowercase characters, underscores, and numbers.

Examples

 Valid	`typedef struct`
	`{`
	`  uint16_t count;`
	`  uint16_t max_count;`
	`  uint16_t padding;`
	`} timer_t;`

 Valid	`typedef enum`
	`{`
	`  VEE_STATE_READY,`
	`  VEE_STATE_READING,`
	`  VEE_STATE_WRITING,`
	`  VEE_STATE_ERASING`
	`} vee_states_t;`

Reasoning

Type names and variable names are often appropriately similar. For example, a set of timer control registers in a peripheral can be named **timer**. To distinguish the structure definition that defines the register layout, it is valuable to create a new type with a distinct name, such as **timer_t**. If necessary, this same type could then be used to create a shadow copy of the timer registers, say called **timer_shadow**.

Exceptions

It is not necessary to use typedef with anonymous structures and unions.

8.8 Naming conventions – functions

Rules

- No procedure shall have a name that is a keyword of C, C++, or any other well-known extension of the C programming language, including specifically K&R C and C99. Restricted names include interrupt, inline, class, true, false, public, private, friend, protected, and many others.
- No procedure shall have a name that begins with an underscore.
- Any procedure that configures or uses a specific peripheral shall use the abbreviated name that is used in the MCU's hardware manual.
- No macro name shall contain any lowercase letters.
- Naming API functions shall be done using these steps:
  - Start off with the common prefix of **R** or **SF** which alerts you they are using Renesas software. See SWP Design Spec for details.
  - Next add a functional prefix in uppercase (for example, **LCD_**, **DMAC_**, **FAT_**, **ADC_**).
  - Last, add the function name. API function names shall be named using Pascal case (XxxXxx) notation. This means the beginning letters of each word shall be capitalized followed by lowercase letters.
- Internal function names shall be done entirely in lowercase, with underscores between words.

Examples

 Valid	`/* Public API function name examples */`
	`R_DMAC_Create();`
	
	`R_LCD_Init();`
	
	`R_FAT_SectorSearch();`

 Valid	`/* Internal function name examples */`
	`transmit_bytes();`
	
	`change_channel();`
	
	`calculate_buffer_size();`

 Invalid	`void timer0_init(void)` **←Peripheral is called CMT so name should have been 'CMT0_init'.**
	`{`
	`  /* Configure MCU's compare match timer (CMT) channel 0 */`
	`  ...`
	`}`

Reasoning

Good function names make reviewing and maintaining code easier. The data (variables) in programs are nouns. Functions manipulate data and are thus verbs. The use of prefixes is in keeping with the important goal of encapsulation and helps avoid procedure name overlaps. Using **R_** in front of function names helps customers know exactly what functions are provided by Renesas.

Exceptions

Linker variables and GCC keywords like **__attribute__** may start with underscore.

8.9 Threads

Guidelines

- All functions that represent threads should be given names ending with **_thread**.

Examples

 Valid	`void alarm_thread (void * p_data)`
	`{`
	`  ...`
	`}`

Reasoning

Each thread in a real-time operating system (RTOS) is like a mini-`main()`, typically running forever in an infinite loop. It is valuable to easily identify these important functions during code reviews and debugging sessions.

Exceptions

None

8.10 Scope

Rules

- Variables with file-scope shall use the static keyword to limit their access to the file they are declared in.

Examples

 Invalid	`uint8_t g_count;` ← **Variable should have *static* to limit scope**
	
	`void R_EXAMPLE_Bar (void)`
	`{`
	`  /* Holds result of function. */`
	`  uint8_t result;`
	
	`  result = R_EXAMPLE_Foo(g_count);`
	`}`

 Valid	`static uint8_t count;`
	
	`void R_EXAMPLE_Bar (void)`
	`{`
	`  /* Holds result of function. */`
	`  uint8_t result;`
	
	`  result = R_EXAMPLE_Foo(count);`
	`}`

Reasoning
Sharing global variables between files has the potential to break the separation of different software layers. Using a global variable's name instead of having access functions also creates a dependency between the layers on the variable's name and use, that can make future refactoring difficult.

Exceptions
None

Website and Support

Support: https://synergygallery.renesas.com/support

Technical Contact Details:

- America: https://www.renesas.com/en-us/support/contact.html
- Europe: https://www.renesas.com/en-eu/support/contact.html
- Japan: https://www.renesas.com/ja-jp/support/contact.html

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Oct 18, 2017	-	Initial release

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other disputes involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawing, chart, program, algorithm, application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics products.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.

   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots etc.

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Renesas Electronics products are neither intended nor authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems, surgical implantations etc.), or may cause serious property damages (space and undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. When using the Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat radiation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions or failure or accident arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please ensure to implement safety measures to guard them against the possibility of bodily injury, injury or damage caused by fire, and social damage in the event of failure or malfunction of Renesas Electronics products, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures by your own responsibility as warranty for your products/system. Because the evaluation of microcomputer software alone is very difficult and not practical, please evaluate the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please investigate applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive carefully and sufficiently and use Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall not use Renesas Electronics products or technologies for (1) any purpose relating to the development, design, manufacture, use, stockpiling, etc., of weapons of mass destruction, such as nuclear weapons, chemical weapons, or biological weapons, or missiles (including unmanned aerial vehicles (UAVs)) for delivering such weapons, (2) any purpose relating to the development, design, manufacture, or use of conventional weapons, or (3) any other purpose of disturbing international peace and security, and you shall not sell, export, lease, transfer, or release Renesas Electronics products or technologies to any third party whether directly or indirectly with knowledge or reason to know that the third party or any other party will engage in the activities described above. When exporting, selling, transferring, etc., Renesas Electronics products or technologies, you shall comply with any applicable export control laws and regulations promulgated and administered by the governments of the countries asserting jurisdiction over the parties or transactions.
10. Please acknowledge and agree that you shall bear all the losses and damages which are incurred from the misuse or violation of the terms and conditions described in this document, including this notice, and hold Renesas Electronics harmless, if such misuse or violation results from your resale or making Renesas Electronics products available any third party.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.3.0-1 November 2016)

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HAL II Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 6.0