RENESAS
BIG IDEAS FOR EVERY SPACE

BASICS OF THE RENESAS SYNERGY™ PLATFORM

Richard Oed

2020.04

CHAPTER 3

AN INTRODUCTION TO THE APIs OF THE SYNERGY™ SOFTWARE PACKAGE

CONTENTS

3 AN INTRODUCTION TO THE APIs OF THE SYNERGY™ SOFTWARE PACKAGE 03

3.1 API Overview 03

3.2 API Syntax 04

3.3 API Constants, Variables and Other Topics 06

3.4 API Usages 07

Disclaimer 10

3 AN INTRODUCTION TO THE APIs OF THE SYNERGY™ SOFTWARE PACKAGE

What you will learn in this chapter:

- What the different layers of the Synergy™ Software Package are and how they can be accessed.
- Details about the Application Programming Interfaces and how to use them.

Ease of use was in the mind of the designers while working on the Synergy Software Package (SSP). While offering tremendous functionality, the SSP is quite simple to use, as the design of the Application Programming Interface (API) is very straightforward and comprehensive, encapsulating the complexity of the SSP, but still giving the programmer full control of the functionality. 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 detail 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 us have a look at some of the details of the API!

3.1 API Overview

Applications can access all the functions in the SSP through intuitive and comprehensive 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 more often during a design these days.

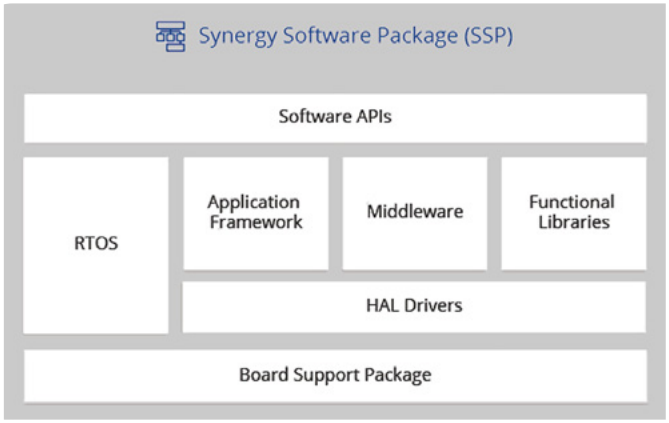

Figure 3-1: The different layers of the Synergy Software Package

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 SSP is used: in the Synergy Configurator the *SPI Driver on r_spi* in the Threads tab needs to be replaced by the *SPI driver on r_sci_spi* driver. As both drivers will use the same instance called *g_spi0*, there is no need to change anything in the application code.

Another possibility is to use the *SPI Framework Device on sf_spi*. In this case, only the *g_spi0 Driver on r_spi* driver inside the framework needs to be replaced by the *g_spi0 SPI driver on r_sci_spi* driver inside the Synergy Configurator. All calls to the frameworks API *g_sf_spi_device0*

stay the same. There is not even a change in the code necessary. Everybody who, like me, already had in the past the task to change peripherals in existing code, will really appreciate this simplicity offered by the Synergy Platform. This means that making design changes after project start does not create big headaches and are perfectly easy to implement and will also reduce the testing time necessary to a bare minimum.

The architecture of the different layers is shown in Picture 3-1: At the bottom is the Synergy MCU, with the Board Support Package (BSP) sitting on top of it, being responsible for getting the MCU from reset to the main application and 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 which makes the software above the HAL more portable across the entire Synergy Platform.

On top of the HAL are the Application Frameworks, the Functional Libraries and the X-Ware™ middleware, as well as the RTOS. Whilst the Application Frameworks implement commonly used system services like communication or audio playback or capacitive touch sensing, the Functional Libraries contain specialized software, for example for digital signal processing or security related functions. At the very top is the user program, making calls to the layers below through the API. Details on the layers and parts of the SSP are available in chapter 2.

Each of the different layers are accessed through API calls directly, or in a stacked manner. An audio application, for example, can call the Audio Playback Framework of the Application Frameworks, which makes use of the IIC (Inter-Integrated Circuit) driver and two Data Transfer Controller (DTC) drivers to provide read / write access to an external audio converter connected to the microcontroller over the IIC interface, and a General PWM Timer for the time base. Of course, an end application could also access the HAL drivers and the BSP directly, but using the Application Frameworks is much, much easier as no detailed knowledge of the underlying parts is necessary in this case.

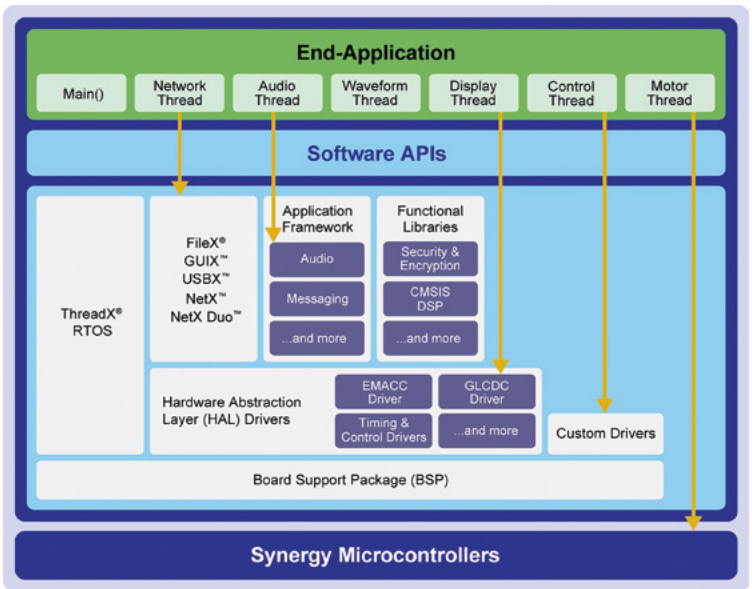

Figure 3-2: User programs can have direct access to the different layers of the SSP, but using the Application Frameworks or the Functional Libraries is much easier

3.2 API Syntax

Before diving more deeply into the details of the API, we should have a short look on the naming conventions of the different APIs and files. Once comprehended, not only programming an application becomes an easy task, but also understanding either code from another programmer or one's own code after several months, an effort which shouldn't be underestimated (something we always do) and where the clear structure provided by the SSP will help a lot.

- **R_BSP_xxx:** The prefix for a common BSP function, e.g. `R_BSP_VersionGet()`.
- **BSP_xxx:** The prefix for BSP defines, e.g. `BSP_CODE_VERSION_MAJOR`.
- **SSP_xxx:** The prefix for common SSP defines, mostly error codes, e.g. `SSP_SUCCESS`.
- **g_<interface>_on_<instance>:** The name of the constant global structure of an instance that implements the API of the interface. E.g. `g_spi_on_rspi` for an HAL Layer function or `g_sf_spi_on_sf_spi` for an Application Frameworks function.
- **r_<interface>_api.h:** The name of an interface module header file, e.g. `r_spi_api.h`.
- **R_<INTERFACE>_<Function>:** The name of an HAL Layer API, e.g. `R_RSPI_Open()`.
- **sf_<framework>_<function>:** An Application Frameworks level module.
- **SF_<FRAMEWORK>_xxx:** An Applications Frameworks Layer API, e.g. `SF_SPI_Open()` or `SF_AUDIO_PLAYBACK_Open()`.

The naming conventions of the X-Ware™ middleware 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®.

Here is a summary of several of the IDs for the X-Ware™ middleware and the ThreadX® real-time operating system:

- **tx:** ThreadX® related functions, e.g. `tx_thread_create()`.
- **nx:** NetX™ related functions, e.g. `nx_packet_copy()`.
- **nxd:** NetX Duo™ related functions, e.g. `nxd_ipv6_enable()`.
- **gx:** GUIX™ related functions, e.g. `gx_display_create()`.
- **fx:** FileX® related functions, e.g. `fx_directory_create()`.
- **ux:** USBX™ 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 SSP. 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 features they need.
- **Module Instance:** Single and independent configuration of a Module.
- **Interfaces:** Interfaces 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.
- **SSP 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 in the Synergy MCUs.

3.3 API Constants, Variables and Other Topics

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

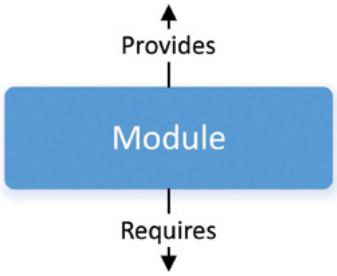

Figure 3-3: Modules are built in a way that they provide functionality to the caller and require functionality from the level below

Modules can be layered on top of one another, building an SSP stack. The stacking process is performed by matching what one module provides with what another module requires. The audio playback framework module, for example, requires a data transfer interface (amongst others), which can be provided by the Data Transfer Controller (DTC) driver module. Its code is not included in the Audio Playback Module by intent, allowing the (underlying) DTC module to be reused by other modules as well, for example by the UART or the SD-card drivers.

The ability to stack modules offers a great benefit, as it ensures the flexibility needed by the application level. If they were directly dependent on others, portability across different user designs would be difficult. The SSP architecture as it was designed provides the flexibility to swap modules in and out (e.g. exchange the UART Driver against the SPI Driver) through the use of the SSP 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 Synergy MCUs, there is occasionally an overlap of features between different peripherals. For example, IIC communications can be achieved through the use of the IIC peripheral or through the use of the SCI peripheral in its Simple 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 SSP interface: a module depending control structure named `<interface>_ctrl_t*`, which is used as unique identifier for using the module, a configuration structure named `*<interface>_cfg_t*` as input to the module for initial configuration during the `open()` call and an instance structure, consisting of a pointer to the control structure, a pointer to the configuration structure and a pointer to the API structure. The name of this 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 instances header file, e.g. `r_spi.h`. While this structure 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 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 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.

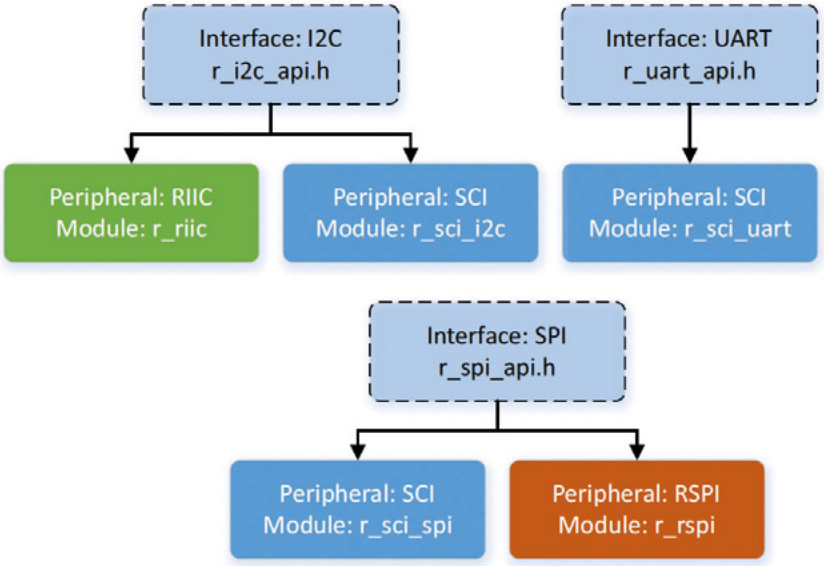

Figure 3-4: On Synergy MCUs some peripherals will have a one-to-one mapping between the interface and instance, while others will have a one-to-many mapping

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 Synergy MCUs, some peripherals, e.g. 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 the mapping.

3.4 API Usage

After all the theory, it is now time to finally look at how easy it is to work with the Synergy Software Package. 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 SSP 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 Synergy MCU Family, so inside e² studio we select the *SPI driver on r_sci_spi* in the Synergy Configurator. This is done by first highlighting the *HAL/Common thread* on the left hand side of the Threads page and then by selecting *New Stack → Driver → Connectivity → SPI Driver on 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 of the *g_transfer0* and *g_transfer1 Transfer Drivers on r_dtc Event SCI0 TXI (RXI)* will be added.

Nothing else needs to be done. In our case of the example above, the receive, transmit and error interrupts are automatically enabled by the Synergy 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 user name 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*-window of the *g_spi0 SPI Driver on r_sci_spi* module would be the place to configure the parameters, like bitrate, 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 Synergy Configurator automatically.

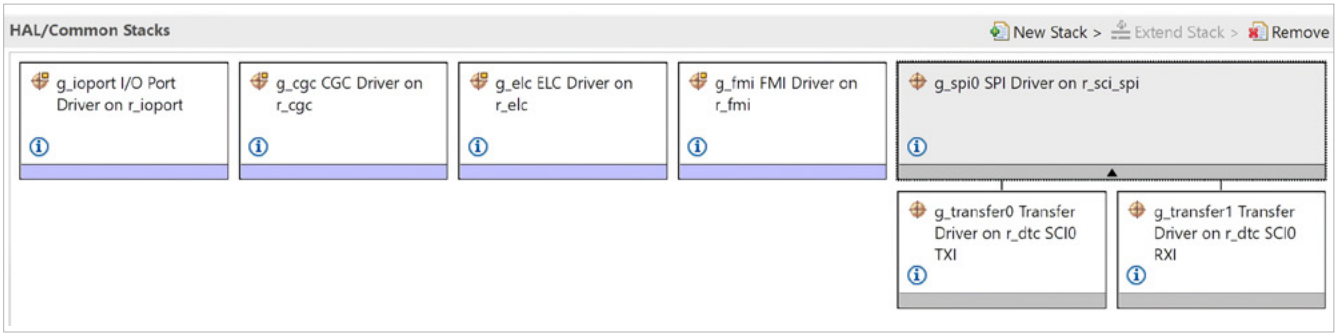

Figure 3-5: The SPI stack displayed in e² studio

Right now, there is no need to care about the other items in the *HAL/Common Stacks* window. These are the basic drivers needed by the Board Support Package for the I/O-ports, the Event Link Controller (ELC), the Clock Generation Controller (CGC) and the Factory MCU Information (FMI). The FMI-driver includes a generic API for reading the records from the Factory MCU Information Flash Table, providing information about the features and peripherals implemented on the microcontroller used. All these drivers are added automatically by the configurator.

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 Synergy Configurator to actually transmit and receive data through this peripheral. Expanding the *Peripherals* list on the left side of the view will show the available interfaces. 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* will configure the pins correctly.

Saving the changes using *<ctrl>-<s>* and clicking on the Generate Project Content icon will create the necessary source files and will populate the control and configuration structures, named `spi_ctrl_t*` and `*spi_cfg_t*` respectively, and place 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* interface.

With the setup completed and SSP-files generated, 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 window 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, p_src, numUnits, bitWidth);
```

with `p_src` being the pointer to a user defined array holding the data to be sent over the SPI, `numUnits` being the number of units to be written as a 32-bit unsigned integer and `bitWidth`, which has the type spi_bit_width and holds the size of the units. The exact syntax can be extracted from the HAL interfaces reference chapter in the SSP User's Guide or by using the smart documentation feature of e² studio. This short example is a very good demonstration of the capabilities of the Synergy Software Package. Only two lines of code have to be written to configure the port and to send an array of data through it. Everything else has already been created and performed by the Configurators and the drivers of the SSP. 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 SSP is built in layers.
- Each layer can be accessed directly through Application Programming Interfaces (API), but users are encouraged to use the APIs of the Application Frameworks and the Functional Libraries instead of calling the lower levels directly.
- Using the Application Frameworks and the Functional Libraries makes the code more portable and maintainable.
- The syntax of the API is straightforward and intuitive.