Renesas Synergy™ Platform

Messaging Framework Module Guide

Introduction

This module guide will enable you to effectively use a module in your own design. Upon completion of this guide, you will be able to add this module to your own design, configure it correctly for the target application and write code, using the included application project code as a reference and efficient starting point. References to more detailed API descriptions and suggestions of other application projects that illustrate more advanced uses of the module are available on the Renesas Synergy Knowledge Base (as described in the References section at the end of this document) and should be valuable resources for creating more complex designs.

The Messaging Framework module is implemented on sf_message. It includes a lightweight and event-driven framework API for passing messages between threads. The Messaging Framework module allows applications to communicate messages between two or more threads. The framework uses the ThreadX® message-queue primitive for message passing and provides more benefits than the ThreadX RTOS message-queue services alone. The Messaging Framework API is purely a software API and does not access any hardware peripherals. The Messaging Framework callback allows an event-producer thread and a message-subscriber thread to handshake after the message passing is done.

You can use the messaging tab to either create your own custom event classes, events, and subscribers for the Messaging Framework module, or to customize preconfigured events such as the touch event used by the Touch Panel Framework module.

Contents

1. Messaging Framework Module Features

The Messaging Framework module supports the following functions:

- **Inter-Thread Communication:** Allows application threads which control disparate devices or manage subsystems to communicate with each other.
- **Publishing/Subscribe scheme:** Based on the loosely-coupled messaging paradigm, this scheme allows multiple threads to listen to an event class. The message producer thread does not need to know who is subscribing to a message for the event class. Subscribers do not need to know who produces the message.
- **Message management:** Supports buffer control blocks to manage each message, including flags to control the buffer and a callback function pointer for handshaking.
- **Message buffering:** Manages buffer allocation and release for messaging. An application use the allocated buffer to write a message and discard the message should it is no longer be needed.
- **Synchronous communication:** Supports asynchronous messaging by using the ThreadX message-queue, and also provides an option to create a handshake between a message producer and a subscriber thread. The handshake is implemented by invoking a user-callback function of the producer thread from a subscriber thread.
- **Message formatting:** To provide a predefined common message header and also provides some typical payload structure templates as examples.
- **Message Priority:** Sends a high-priority message so a subscriber thread can retrieve the message prior to other messages which are located in the message queue.

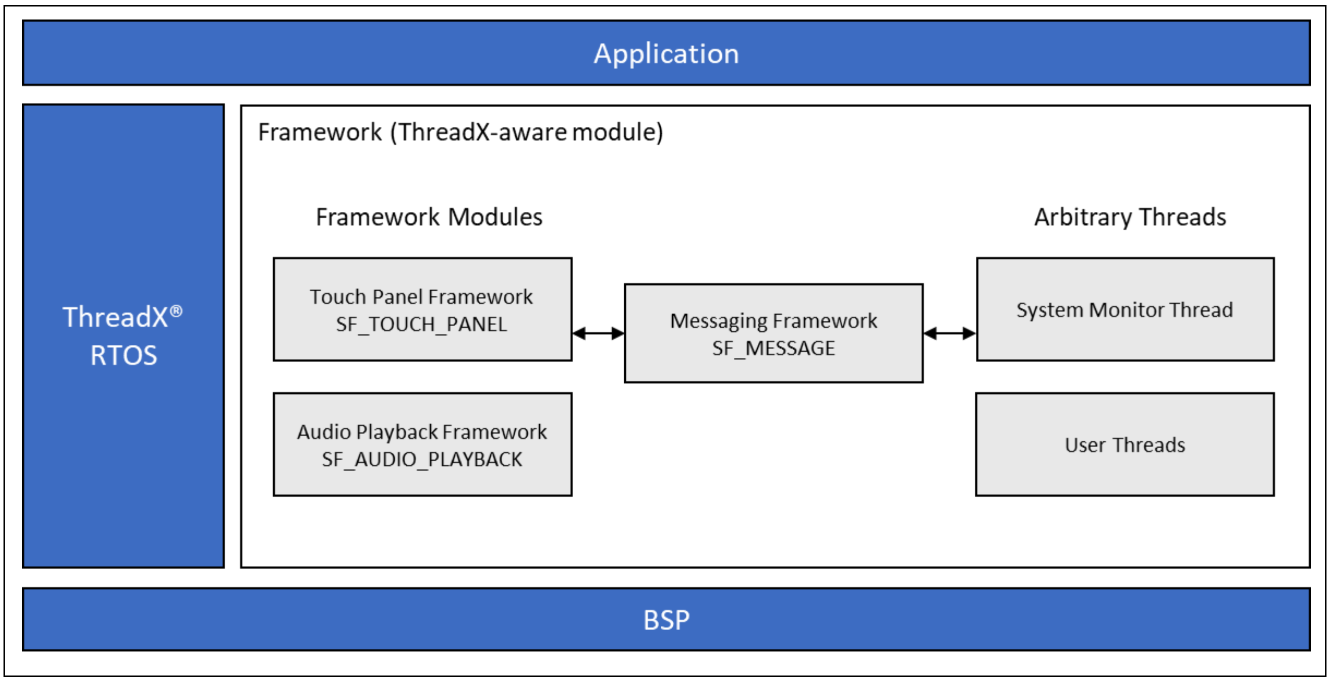


Figure 1. Messaging Framework Module Organization and Use Example

2. Messaging Framework Module APIs Overview

The Messaging Framework module defines APIs for opening and closing the framework, acquiring and releasing buffers, and posting messages to subscribers. A complete list of the available APIs, an example API call, and a short description of each can be found in the following table. A table of status return values follows.

Table 1. Messaging Framework Module API Summary

Function Name	Example API Call and Description
.open	`g_sf_message.p_api->open (g_sf_message.p_ctrl, g_sf_message.p_cfg);` Initialize message framework. Initiate the messaging framework control block, configure the work memory corresponding to the configuration parameters.
.close	`g_sf_message.p_api->close (g_sf_message.p_ctrl);` Finalize message framework.
.bufferAcquire	`g_sf_message.p_api->bufferAcquire ( g_sf_message.p_ctrl, &p_buffer, &acquire_cfg, wait_option);` Acquire buffer for message passing from the block.
.bufferRelease	`g_sf_message.p_api->bufferRelease ( g_sf_message.p_ctrl, &p_buffer, option);` Release buffer obtained from .bufferAcquire API.
.post	`g_sf_message.p_api->post (g_sf_message.p_ctrl, (sf_message_header_t *) p_payload, &post_cfg, &err_post, wait_option);` Post message to the subscribers.
.pend	`g_sf_message.p_api->pend (g_sf_message.p_ctrl, &my_queue, &p_buffer, wait_option);` Pend message.
.versionGet	`g_sf_message.p_api->versionGet (&version);` Retrieve the API version with the version pointer.

Note: For details on operation and definitions for the function data structures, typedefs, defines, API data, API structures, and function variables, see the *SSP User's Manual* API References for the associated module.

Table 2. Status Return Values

Name	Description
SSP_SUCCESS	API call successful.
SSP_ERR_ASSERTION	Required pointer is NULL.
SSP_ERR_BUFFER_RELEASED	The buffer is released.
SSP_ERR_ILLEGAL_SUBSCRIBER_LISTS	Message subscriber lists is illegal.
SSP_ERR_IN_USE	The messaging framework is in use.
SSP_ERR_INTERNAL	OS service call fails.
SSP_ERR_INVALID_MSG_BUFFER_SIZE	Message buffer size is invalid.
SSP_ERR_INVALID_WORKBUFFER_SIZE	Invalid work buffer size.
SSP_ERR_MESSAGE_QUEUE_EMPTY	Queue is empty. (Timeout occurs before receiving a message if timeout option is specified.)
SSP_ERR_MESSAGE_QUEUE_FULL	Queue is full. (Timeout occurs before sending a message if timeout option is specified.)
SSP_ERR_NO_MORE_BUFFER	No more buffer found in the memory block pool.
SSP_ERR_NO_SUBSCRIBER_FOUND	No subscriber found.
SSP_ERR_NOT_OPEN	Message framework module has yet to be opened.
SSP_ERR_TIMEOUT	OS service call returns timeout.
SSP_ERR_TOO_MANY_BUFFERS	Too many message buffers.

Note: Lower-level drivers may return common error codes. Refer to the *SSP User's Manual* API References for the associated module for a definition of all relevant status return values.

3. Messaging Framework Module Operational Overview

The following figure shows the overview of the messaging data flow between a message producer thread and subscriber thread(s) in the system making use of the Messaging Framework module.

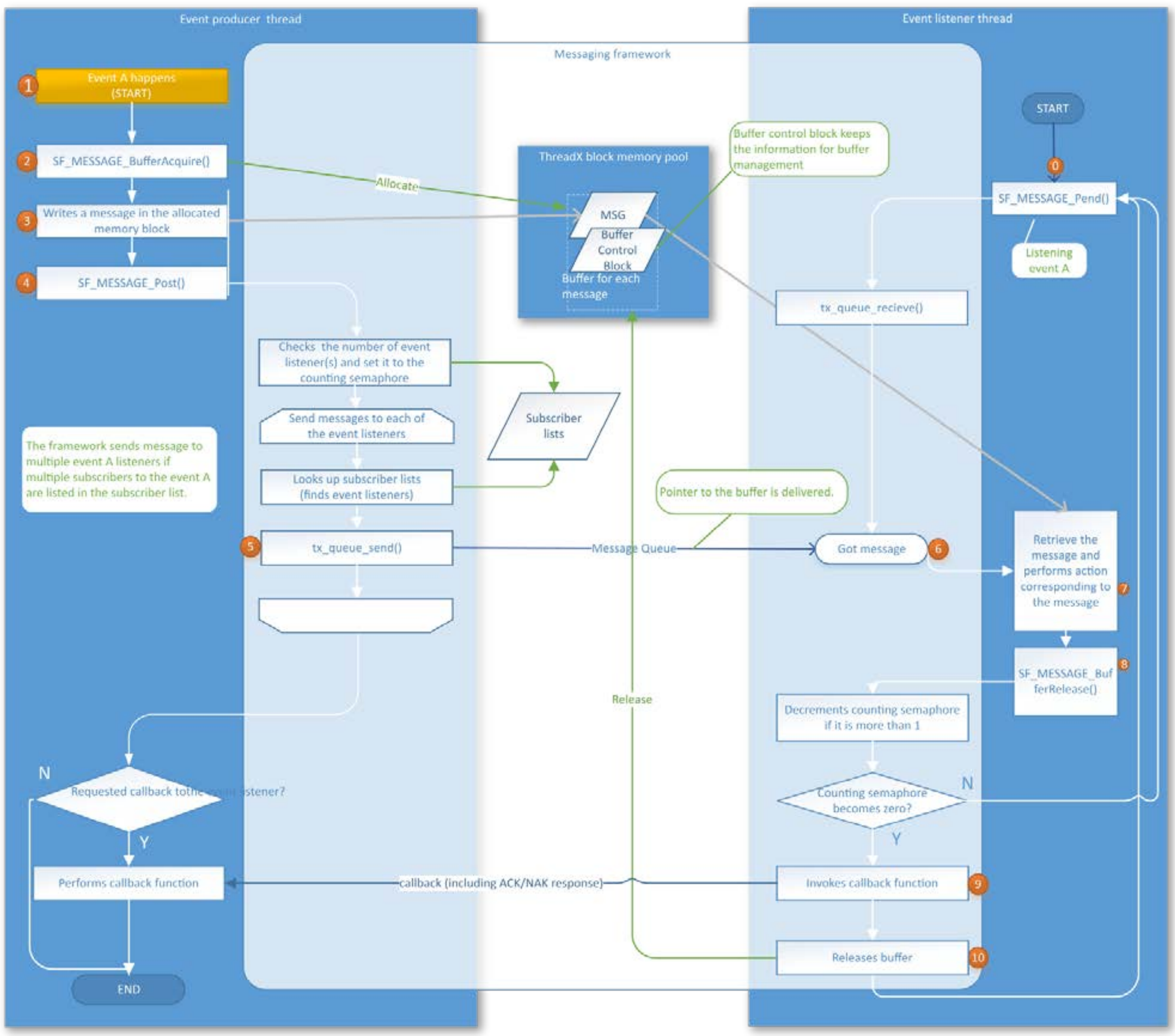

Event producer thread
Messaging framework
Event listener thread
1 Event A happens (START)
2 SF_MESSAGE_BufferAcquire()
3 Writes a message in the allocated memory block
4 SF_MESSAGE_Post()
Allocate
ThreadX block memory pool
MSG
Buffer Control Block
Buffer for each message
Buffer control block keeps the information for buffer management
START
0
SF_MESSAGE_Pend()
Listening event A
tx_queue_recieve()
Checks the number of event listener(s) and set it to the counting semaphore
Send messages to each of the event listeners
Subscriber lists
Looks up subscriber lists (finds event listeners)
The framework sends message to multiple event A listeners if multiple subscribers to the event A are listed in the subscriber list.
Pointer to the buffer is delivered.
5 tx_queue_send()
Message Queue
Got message 6
Retrieve the message and performs action corresponding to the message 7
SF_MESSAGE_BufferRelease() 8
Release
Decrements counting semaphore if it is more than 1
N
Requested callback to the event listener?
Y
Counting semaphore becomes zero?
N
Y
Performs callback function
callback (including ACK/NAK response)
Invokes callback function 9
Releases buffer 10
END

Figure 2. Messaging Framework - Data Flow

The following is a description for each stage of the message passing procedure:

Note: A thread in the system has been called using the `open` API and message subscriber threads have called the `pend` API to pend on a message for the event class.

1. An event (Event A) happens on a message producer thread.
2. A message producer thread calls `bufferAcquire` to acquire a buffer from the ThreadX memory pool managed by the Messaging Framework module. `bufferAcquire` returns the address of the allocated buffer.
3. A message producer writes the message to the allocated buffer.
4. A message producer calls `post` to post the message.
5. The Messaging Framework module looks up the event subscriber list and sends a message to the message queue of the message subscriber threads using the ThreadX message-queue primitive. The framework just sends the pointer to the buffer but does not send the entire message, thereby performing lightweight message passing.

6. The message reaches the message queue of the message subscriber threads and the message subscriber threads return from `pend`. The API function returns the buffer address where the message is stored to message subscriber threads.
7. The message subscriber threads receive the message and perform an action corresponding to the event.
8. The message subscriber threads call `bufferRelease` to try to release the allocated buffer for the message. If the message subscriber thread is not the last one subscribing to the message, the framework does not release the buffer as the message has to be kept in the buffer until all subscribers have received the message.
9. The Messaging Framework module invokes a user-callback function which is specified by an event producer thread if the message subscriber thread is the last one in the message subscriber group.
10. The Messaging Framework module releases the buffer if the message subscriber thread is the last one in the message subscriber group. (There is an option 'SF_MESSAGE_ACQUIRE_OPTION_KEEP' to not release the buffer.)

3.1 Messaging Framework Module Message Producer and Subscribers

The Message Framework module is an inter-thread messaging system based on the publish/subscribe model. A message is posted with an event class code by an event producer thread. The message subscriber threads which subscribe to the event class can check for pending messages. Subscribers are registered in the subscriber list, which is referred to by the framework. The subscriber list allows the framework to deliver a message to multiple subscribers.

Every thread which joins the Messaging Framework module system network can send a message, and all threads in the network can listen to the message.

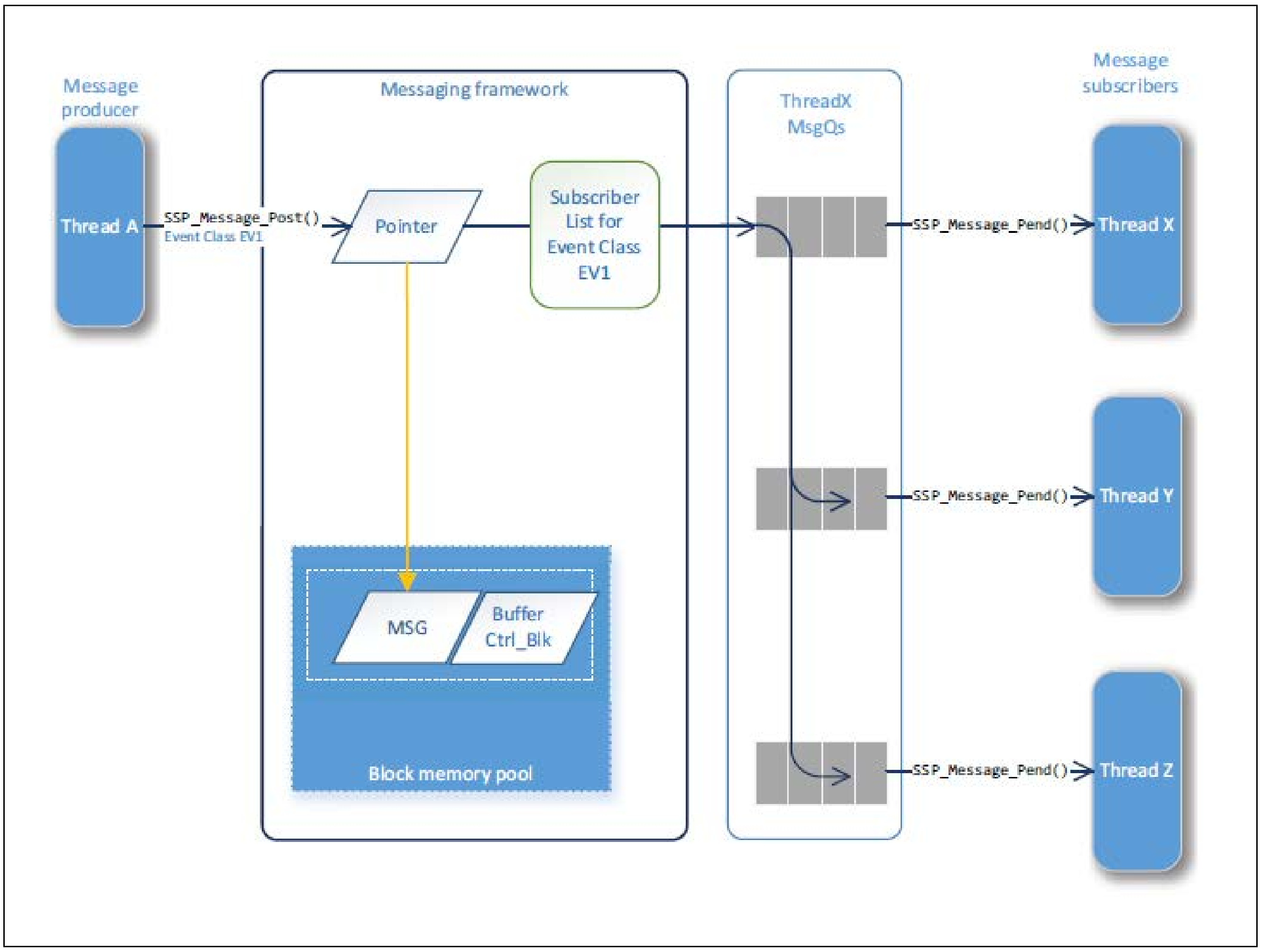


Figure 3. Messaging Framework — Subscribing

3.2 Messaging Framework Module Events, Subscribers, and Messages

3.2.1 Messaging Framework Module Event Class Code

The event class code is the most important definition for the Messaging module. The Messaging module uses the event class code as the mechanism to connect a message producer with subscribers. The event class code is the class definition of the events which occur in the application. The classification of the event class relies on the user definition, but it is intended to be the group name of the particular events which can occur in a subsystem. For example, you can use the following event classes:

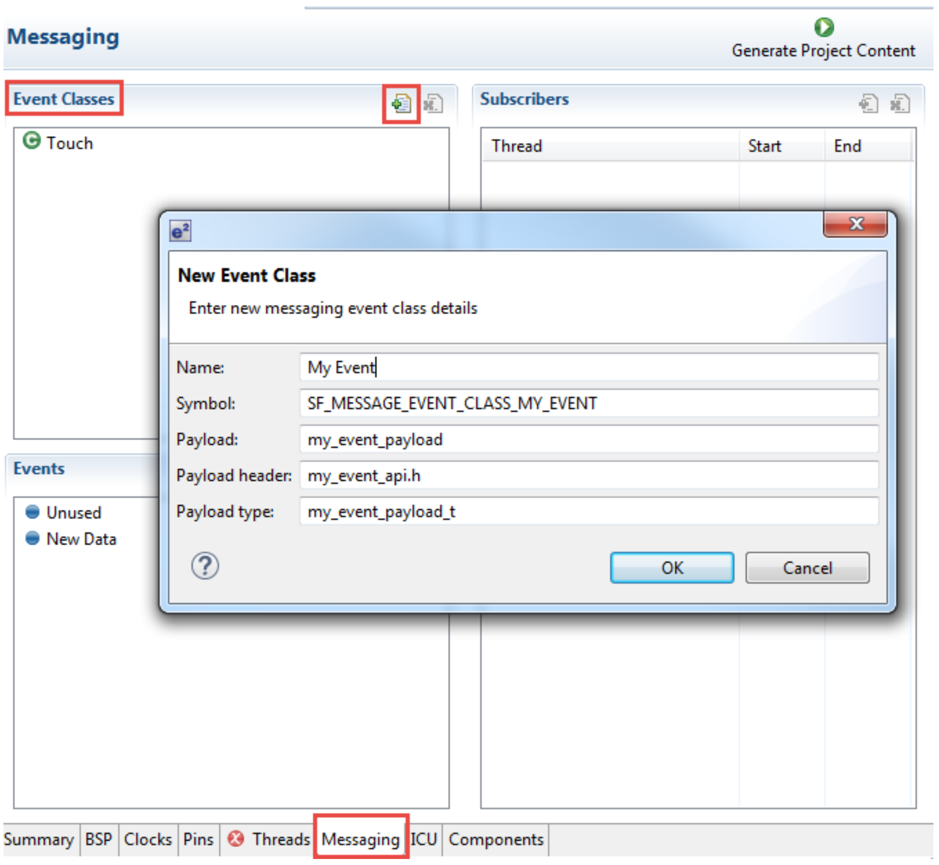
- The 'touch' event class which is part of the touch subsystems. The Touch Event Class is automatically loaded into the event classes' window. This window is available on the Messaging tab the Project Configurator when you add the Touch Panel Framework to your Synergy project.
- The 'time' event class is part of the subsystem that manages time-related applications.

The event class code is defined in the `sf_message_event_class_t` enumeration and has a prefix `SF_MESSAGE_EVENT_CLASS_XXX`. Since the definition of the event class code is different for each system, the framework does not provide a concrete event class code but instead provides a set of event class codes as examples. (See the Configuring the Messaging Framework Module section.) The maximum number for the event class is 255.

An application can use the event class code as follows:

- The message producer thread sets the event class code to the `event_b.class_code` bit fields in the `sf_message_header_t` type common message header before posting the message.
- The message subscriber thread branches to the event processing corresponding to the event class code which is set to the message header after receiving the message.
- The subscribers for the event class code must be grouped and registered in the subscriber list so that the Message Framework can deliver the message to the subscribers.

The following screen capture illustrates how you can configure an event class using the ISDE:



Name: My Event
Symbol: SF_MESSAGE_EVENT_CLASS_MY_EVENT
Payload: my_event_payload
Payload header: my_event_api.h
Payload type: my_event_payload_t

Figure 4. Messaging — ISDE Event Class Configuration

3.3 Messaging Framework Module Event Class Instance Number

The event class instance number is used when an application needs to have different event class instances. For example, the audio streaming event class can have instance *N* which represents the streaming channel N. Message subscribers can receive a message only if the event class instance number in the message common header matches to the number it owns.

In other words, messages for which the event class instance number is out of range for the subscriber are filtered out and not delivered to the subscriber even though it is the event class subscriber. The maximum for the event class instance number is 255.

Note: The Touch Panel Framework typically only has one instance, and therefore the event class instance number is 0 with the start and end values of the subscriber list set to 0.

An application can use the event class instance number as follows:

- The message producer thread sets the event class instance number to the `event_b.class_instance` bit fields in the `sf_message_header_t` type common message header before posting the message.
- Each subscriber instance in the Subscriber List has to specify the range of the event class instance numbers to receive the message (`sf_message_subscriber_t::instance_range.start` and `sf_message_subscriber_t::instance_range.end`).
- If there is no need for multiple instances for an event class, just set `sf_message_header_t::event_b.class_instance`, `sf_message_subscriber_t::instance_range.start`, and `sf_message_subscriber_t::instance_range.end` to zero in the subscriber instance for the subscriber list.

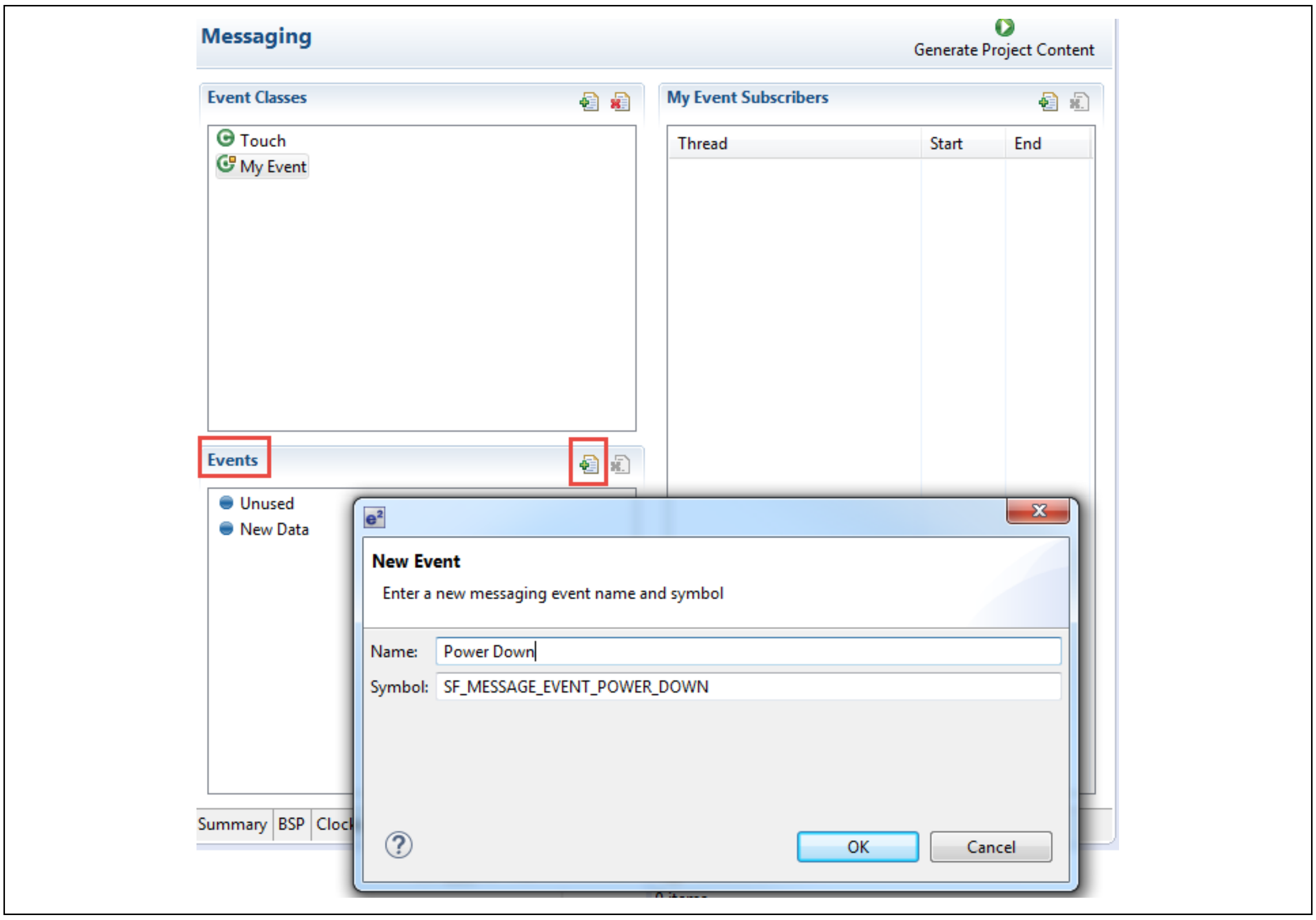
3.4 Messaging Framework Module Event Code

The event code includes the details of the event definition. For instance, the event codes for the audio playback event class are “playback start” and “playback stop.” Another example is ‘set’ or ‘get’ for the ‘time’ Event Class. The event code is enumerated in the `sf_message_event_t` and has a prefix `SF_MESSAGE_EVENT_XXX`. The definition of the event code relies on the user code as well as the event class code. The framework provides some code as examples. See Configuring the event class code and event code for configuring the event code. The maximum for the event class instance number is 65535.

The Touch Panel Framework uses new data as the only event code.

An application can use the event code as follows:

- The message producer thread sets the event code to the `event_b.code` bit fields in the `sf_message_header_t` type common message header before posting the message.
- The message subscriber thread performs an action corresponding to the event code which is set to the message header after receiving the message.



New Event

Enter a new messaging event name and symbol

Name: Power Down

Symbol: SF_MESSAGE_EVENT_POWER_DOWN

Figure 5. Messaging — ISDE Event Configuration

3.5 Messaging Framework Module Subscriber List

The subscriber list is used for message delivery. The framework looks up the subscriber list.

The framework starts to look up the message queues of each subscriber thread from the subscriber group listed in the head of the pointer array to the `sf_message_subscriber_list_t` instance. The important point of the subscriber list is that it is grouped by event class code (`event_class`).

When the framework looks up the subscriber list in the `post` API function at runtime, it compares the Event Class code in the message header (`sf_message_header_t::event_b.class_code`), which is included in the message payload data, with the one in the subscriber group instance (`event_class`). If there is a match, the framework goes to the next level to get the message queue instance (`sf_message_subscriber_t::queue`) until the iterations reach `number_of_nodes`). If there is no match, the framework looks up the next subscriber group and continues until encountering a NULL in the pointer array to the `sf_message_subscriber_list_t` instance.

In the look-up procedure, the subscriber group listed at the head of the subscriber list gets the highest throughput for messaging, but lower subscriber groups encounter a penalty and get lower throughput for messaging.

The subscriber list is the look-up table for all message subscribers. The subscriber list is configured at compile time. It is statically mapped to the memory and looked up by the framework when the `post` API function is called. The subscriber list allows the framework to determine message queues to deliver a message to. The subscriber list consists of two structures `sf_message_subscriber_list_t` and `sf_message_subscriber_t` as shown in the following figure:

- A queue for a subscriber thread is registered in `sf_message_subscriber_t` instance.
- The instances above for the same event class code are grouped and listed in a pointer array.
- The pointer array for a subscriber group is registered in a `sf_message_subscriber_list_t` instance.
- The subscriber list is the pointer array to the subscriber group structures. Subscribers are grouped by the event class code.
- The pointer array must be terminated by NULL.

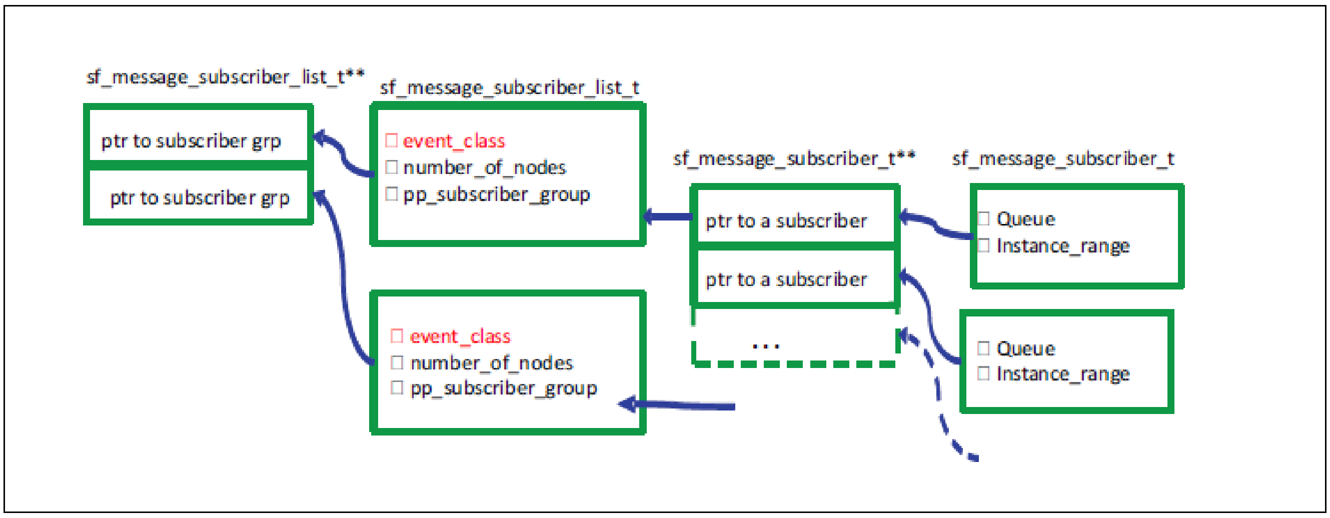
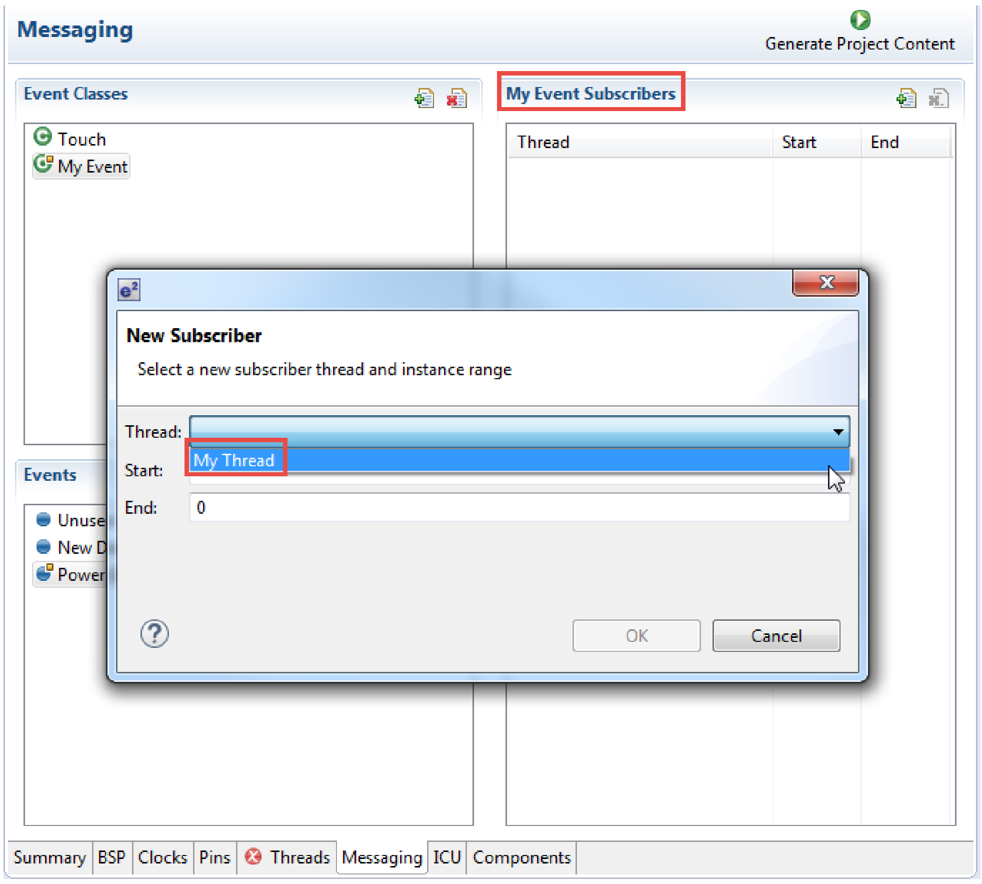


Figure 6. Message Framework — Subscriber List

In the ISDE, you can configure a subscriber grouped by the event class for the thread you named in the Threads tab. In the following example, the thread is named “My Thread” in the Threads tab. The start and end values reflect the event class instance numbers this thread accepts.



New Subscriber

Select a new subscriber thread and instance range

Thread: My Thread

Start:

End: 0

Figure 7. Messaging — ISDE Subscriber Configuration

3.6 Messaging Framework Module Message Payload

The message payload is structured data used by the message producer and the message subscribers to communicate with each other. The message payload contains event class and event code in the common header (`sf_message_header_t` type data, see event class and event code) so that a message producer can post a message to the subscribers to inform the subscribers which event happened.

You must define a system specific message payload structure except for modules for which the SSP provides predefined structures such as the Touch Panel Framework and the audio playback modules. The message payload can contain additional data which is required for the event processing, in addition to the common header.

3.6.1 Messaging Framework Module SSP Predefined Payload

The SSP contains the following predefined message payload structures:

- `sf_touch_panel_payload_t` type for the Touch Panel Framework module
- `sf_audio_playback_data_t` type for the Audio Playback Framework module

These framework modules use the Messaging Framework internally and define suitable message payload structures. An application thread which subscribes to touch event messages from the Touch Panel Framework module can use the predefined message payload by including the header file `sf_touch_panel_api.h`. Likewise, an application thread which posts audio event messages to the Audio Playback Framework module can use `sf_audio_playback_api.h`.

3.6.2 Messaging Framework Module User-Defined Payload

You must define a message payload structure for each event class code; exceptions include the SSP predefined payloads described previously or payloads that require only the common header. To create a new message payload structure, add a common message header (`sf_message_header_t` type structure) at the head in the user-specific message payload structure. The size of the header is 4 bytes.

Note: The payload size must not be greater than the buffer size.

The buffer size limit is critical. Oversized data written beyond the buffer may destroy data in the block memory pool, which is required by ThreadX kernel. Violating the size limit results in a hard fault exception. The buffer size can be configured in `sf_message_ctrl_t::buffer_size`.

3.7 Messaging Framework Module Important Operational Notes and Limitations

3.7.1 Messaging Framework Module Operational Notes

Messaging Framework and OS Message Queue Service

The Messaging Framework module uses the ThreadX primitive-message queue and kernel services and supports some enhancement over the ThreadX RTOS features. For this reason, the Messaging Framework module does not work exactly the same as the ThreadX message-queue service. However, a messaging system with the Messaging Framework module can work simultaneously with the ThreadX message queue services in an application if the two messaging systems are separated.

API Calls Contexts

- The `open` API can only be called from a thread. It can be called only once per the message framework control block instance. The behavior is undefined if the function is called twice.
- The `close` API can only be called from a thread.
- The `bufferAcquire` API can be called from a thread and an ISR.
- The `bufferRelease` API can only be called from a thread.
- The `post` API can be called from a thread and an ISR.
- The `pend` API can be called from a thread and an ISR.

Estimating the Number of Buffers

The number of buffers available to be allocated in the work memory should be estimated properly when designing the messaging system. The number of buffers is estimated as follows:

$$N \approx \frac{W_m}{M_b + B_{cb} + T_x} = \frac{W_m}{M_b + 12\ bytes + 4\ bytes}$$

where:

N – number of buffers,

W_m – work memory size (in bytes),

M_b – message buffer size (in bytes),

$B_{cb} = 12\ bytes$ – size of buffer control block,

$T_x = 4\ bytes$ – reserved bytes for ThreadX.

The maximum number possible for buffers allocated at the same time equals the total amount of depth of message queues in the system. Ideally, the number of buffers for a robust system should be the sum of the depths of the message queues in the system.

For example, in current project, the work memory size is set to “2048” in g_sfmessage Messaging framework of Threads Configuration which means W_m is equal to “2048”. M_b is “4” bytes. So, for current project, we can estimate

the number of message buffers is $\frac{2048}{4+12+4}$ ≈102.

Message Queue Size and Depth Setting

The Messaging Framework module needs a 4-byte memory block on the message queue as it delivers the pointer to the buffer which contains a message payload. For this reason, the size of the message queue should be fixed to 4 bytes. A size greater than 4 bytes negatively affects the performance, as the extra memory copy resides in the ThreadX message queue service, which is internally invoked by the Messaging Framework API functions.

The depth of the message queue is arbitrary, but it should accommodate the number of queued messages at runtime. As a guideline, estimate the value as follows:

$$D \approx \frac{P_{avg}}{S_{avg}}$$

where:

D – queue depth,

P_{avg} – average message delivery rate from producers,

S_{avg} – average event loop completion time in the subscriber.

For example, the average message delivery rate from producers is 1ms, but the average event loop completion time is 4ms. Then queue depth should be 4.

3.7.2 Messaging Framework Module Limitations

Refer to the latest SSP Release Notes for any additional operational limitations for this module.

4. Including the Messaging Framework Module in an Application

This section describes how to include the Messaging Framework module in an application using the SSP configurator.

Note: It is assumed you are familiar with creating a project, adding threads, adding a stack to a thread, and configuring a block within the stack. If you are unfamiliar with any of these items, refer to the first few chapters of the *SSP User’s Manual* to learn how to manage each of these important steps in creating SSP-based applications.

To add the Messaging Framework to an application, simply add it to a thread using the stacks selection sequence given in the following table. (The default name for the Messaging Framework is `g_sf_message0`. This name can be changed in the associated Properties window.)

Table 3. Messaging Framework Module Selection Sequence

Resource	ISDE Tab	Stacks Selection Sequence
g_sf_message Messaging Framework on sf_message	Threads	New Stack> Framework> Services> Messaging Framework on sf_message

When the Messaging Framework module on `sf_message` is added to the thread stack as shown in the following figure, the configurator automatically adds any needed lower-level modules. Modules with a Blue band are shared or common and need only be added once and can be used by multiple stacks.

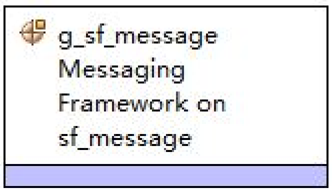


Figure 8. Messaging Framework Module Stack

5. Configuring the Messaging Framework Module

You must configure the Messaging Framework module for the desired operation. The SSP configuration window will automatically identify (by highlighting the block in red) any required configuration selections, such as interrupts or operating modes, which must be configured for lower-level modules for successful operation. Furthermore, only those properties that can be changed without causing conflicts are available for modification. Other properties are 'locked' and not available for changes and are identified with a lock icon for the 'locked' property in the Properties window in the ISDE. This approach simplifies the configuration process and makes it much less error-prone than previous 'manual' approaches to configuration. The available configuration settings and defaults for all the user-accessible properties are given in the properties tab within the SSP configurator and shown in the following tables for easy reference.

One of the properties most often identified as requiring a change is the interrupt priority. This configuration setting is available with the Properties window of the associated module. Simply select the indicated module and then view the Properties window. The interrupt settings are often toward the bottom of the properties list, so scroll down until they become available. Also note that the interrupt priorities listed in the Properties window in the ISDE will include an indication as to the validity of the setting based on the MCU targeted (CM4 or CM0+). This level of detail is not included in the following configuration properties tables but is easily visible with the ISDE when configuring interrupt-priority levels.

Note: You may want to open your ISDE and create the module and explore the property settings in parallel with looking over the following configuration table settings. This helps to orient you and can be a useful 'hands-on' approach to learning the ins and outs of developing with SSP.

Table 4. Configuration Settings for the Messaging Framework Module on sf_message

ISDE Property	Value	Description
Parameter Checking	BSP, Enabled, Disabled Default: BSP	Enables or disables the parameter checking.
Message Queue Depth (Total number of messages to be enqueued in a Message Queue)	16	Specify the size of Thread X Message Queue in bytes for Message Subscribers. This value is applied to all the Message Queues.
Name	g_sf_message	The name of Messaging Framework module control block instance.
Work memory size in bytes	2048	Specify the work memory size in bytes. Choosing a small number results a small number of buffers which can be allocated at the same time (You can confirm the total

ISDE Property	Value	Description
		buffer number on: `sf_message_ctrl_t::number_of_buffers`). If the value is smaller than the peak number of messages to be posted at the same time, the Framework occurs a buffer allocation failure affecting system performance.
Pointer to subscriber list array	p_subscriber_lists	Specify the name of pointer to the Subscriber List array.
name of the block pool internally used in the messaging framework	sf_msg_blk_pool	The name of memory block memory the Framework creates in the control block. This parameter might be useful for debugging purposes, but NULL can be specified for saving memory.
Name of generated initialization function	sf_message_init	Name of generated initialization function
Auto Initialization	Enable, Disable Default: Enable	Auto initialization selection

Note: The example values and defaults are for a project using the Synergy S7G2 MCU Group. Other MCUs may have different default values and available configuration settings.

5.1 Messaging Framework Module Creating a Messaging Queue

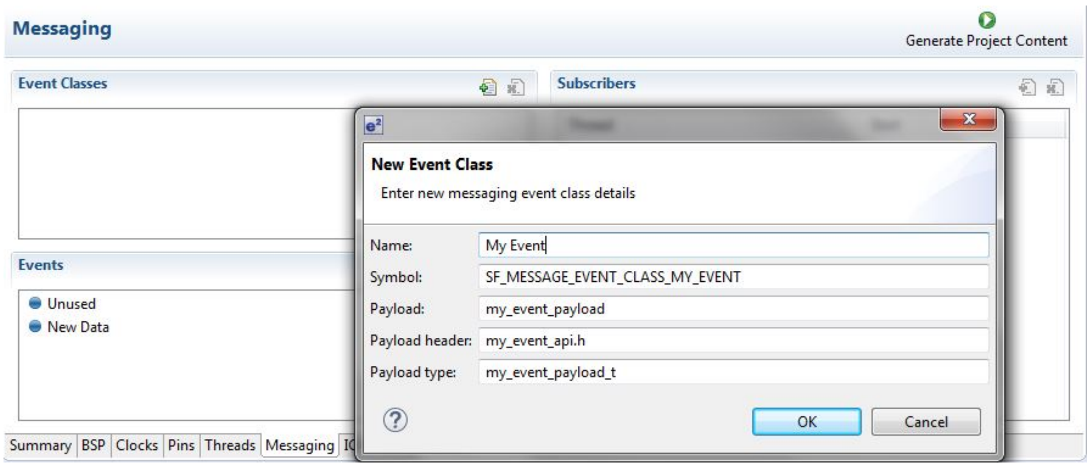
The messaging configurator automatically creates the message queue for the subscribers.

5.2 Messaging Framework Module Configuring an Event Class and Event

To use the Messaging Framework with your own event class, use the Threads tab and the Messaging tab of the project configurator in the ISDE.

In the Threads tab, do the following:

1. Add a new thread in the **Threads** window and give it a unique name.
2. Add the Messaging Framework component in the Thread Stacks panel of the **Threads** window.
3. In the **Messaging** tab (see event class code), do the following:
   A. In the **Event Class** window, add an event class.
   B. Enter the name of the event class for your thread to subscribe to in the **New Event Class** dialog box.



New Event Class

Enter new messaging event class details

Name: My Event

Symbol: SF_MESSAGE_EVENT_CLASS_MY_EVENT

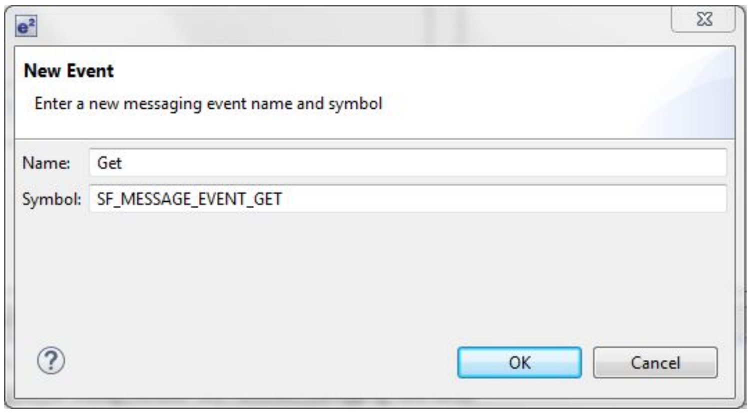
Payload: my_event_payload

Payload header: my_event_api.h

Payload type: my_event_payload_t

Figure 9. Messaging — e^2 studio New Event Class Configuration

4. In the **Events** window, add any events that your application may support (see event code).



Name: Get
Symbol: SF_MESSAGE_EVENT_GET

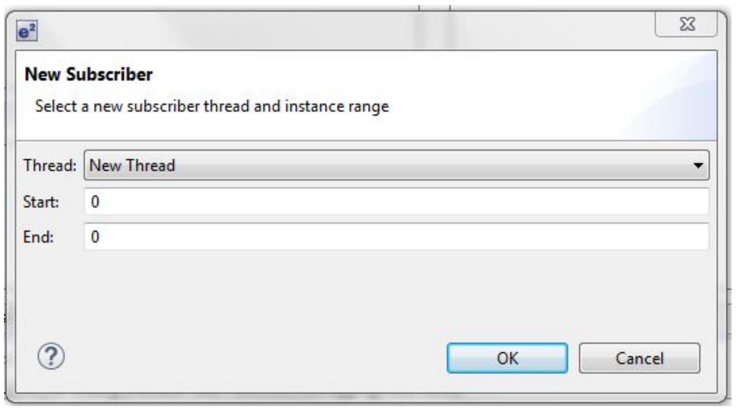
Figure 10. Messaging – e^2 studio New Event Configuration

Your custom event class code and event code are stored in a file named `sf_message_port.h`. The audio playback and the touch event classes are two predefined event classes in the SSP. The Touch Event Class only uses the new data event, `SF_MESSAGE_EVENT_NEW_DATA`.

5.3 Messaging Framework Module Configuring the Subscriber List

In the **Messaging** tab (see also Subscriber List), do the following:

1. Select the event class in the **Event Classes** window and configure a thread for the subscriber list in the event class **Subscribers** window.
2. Select your thread from the drop-down list in the **Threads** dialog box.
3. Next to start, enter the start number of the event class instance(s). If your system does not use multiple event class instances for the event class, or you are not sure what number to specify, just keep the default number (0). Allowed values range from 0 to 255.
4. Next to End, enter the last number of the event class instance(s). If your system does not use multiple event class instances for the event class, or you are not sure what number to specify, just keep the default number (0). Allowed values range from 0 to 255.
5. Click OK. A subscriber for your specified event is added in the subscriber list.
6. Repeat these steps for all event class instances if there are more than one.



Thread: New Thread
Start: 0
End: 0

Figure 11. Messaging — e^2 studio New Subscriber Configuration

5.4 Messaging Framework Module Configuring the Event Class Code and Event Code

5.4.1 Messaging Framework Module Defining the Message Payload

You can define your own message payload structure. Every user-defined message structure must include the `sf_message_header_t` type structure as one of the members, but the other members are entirely user-definable. The Messaging Framework does not care where the message payload structures are defined. You can include the file which defines your own message payload structure in the source file for your message producer and subscriber threads.

5.4.2 Opening the Messaging Module in the Messaging Framework Module

Configure the `sf_message_cfg_t` type configuration parameters to match your system. You can generate code for the configuration structure through the Synergy Configuration tool. Add a Messaging Framework component to the thread stacks in the Threads tab and modify the properties for the Messaging Framework module in the Properties window. When you press the **Generate Project Content** button, the code for the Messaging Framework module is generated on the thread code.

5.4.3 Messaging Framework Module Acquiring a Buffer

Before posting a message, an event producer thread must acquire a buffer for the message from the Messaging Framework module. An event producer thread can acquire the buffer by calling `bufferAcquire`.

When the API function returns SSP_SUCCESS, the buffer with message buffer size in bytes configured on Synergy Configuration tool is allocated in the memory pool managed by the Messaging Framework. The maximum number allowed to be allocated depends on the configuration **Work memory size in bytes** specified on the Synergy Configuration tool. For the estimation of the maximum number, see Estimating the Number of Buffers.

The `bufferAcquire` API has several options to change the message passing behavior:

- **buffer keep:** This option allows the application thread to hold the buffer not to be released by the API function `bufferRelease` if set to true. Typically, the buffer is to be released by `bufferRelease` when the message passing is done; however, in a scenario to have periodical or repeated message passing between threads, this same buffer can be re-used for the messaging without allocating and releasing the buffer each time. Enabling this option reduces the overhead in the buffer allocation/release operation and improve the system throughput.
- **wait_options:** The wait time option is valid if all buffers have been acquired. Any arbitrary thread tick count, TX_WAIT_FOREVER, TX_NO_WAIT can be set for this option. For details, see the `tx_block_allocate()` description for the ThreadX service call in the *ThreadX User Guide*.

5.4.4 Messaging Framework Module Releasing a Buffer

After message subscriber threads receive a message posted by an event producer, the message subscriber threads must release the buffer to the framework. Buffer releasing is performed by calling `bufferRelease`. Since the API function can be called multiple times if there are multiple event subscribers in the system, the actual buffer release is performed only by the last message subscriber thread in the event subscribers. For instance, if there were three subscribers in the subscriber group for the event class, the first and second threads which call `bufferRelease` do not release the buffer, only the third thread releases the buffer. Note that if the buffer keep option is specified by `bufferAcquire`, the buffer is never being released except when option `SF_MESSAGE_RELEASE_OPTION_FORCED_RELEASE` is passed to the API function argument option. (Also see Messaging Framework callbacks for `bufferRelease` API function use.)

The API is also used for invoking a user-callback function to create a handshake between an event producer thread and a message subscriber thread.

Posting a message

1. After getting a buffer by `bufferAcquire`, an event producer can write the message payload data to the buffer location.
2. ATTENTION: Writing data to the buffer is the user's responsibility and writing more data than the buffer size causes a fatal error in the Messaging Framework module.
3. Write an event class code to the `sf_message_header_t::event_b.class_code` in the payload structure.
4. Write an event code to the `sf_message_header_t::event_b.code` in the payload structure. It is not mandatory to specify this but necessary in most cases.
5. Write an event class instance number to `sf_message_header_t::event_b.class_instance`. Specify a number from 0 to 255 if your system has multiple instances for an event class. Specify 0 if your system simply uses single event class instance.
6. Post the message by the `post` API. Note that the pointer to the buffer must be cast to the `sf_message_header_t` * type when given to the API. The message will be delivered to the message

subscribers which are registered in the message subscriber list. The `post` API has several options to change the message passing behavior.

- **Message priority:** Message can take one of two message priority levels, `SF_MESSAGE_PRIORITY_NORMAL` or `SF_MESSAGE_PRIORITY_HIGH`. When `SF_MESSAGE_PRIORITY_HIGH` is specified, the message is queued at the front of the message queue of the message subscriber. This is typically used for the emergency message to make the message subscribers handle the event prior to the events which might have been queued in the message queue.
- **User-callback function:** This function is registered in the buffer control block of the Messaging Framework module. The callback function is invoked by `bufferRelease`. This function can be used for handshaking between an event producer thread and a message subscriber thread.
- **Wait_options:** The wait time option is valid if a message queue of the message subscriber thread is full. Any arbitrary ThreadX tick count, `TX_WAIT_FOREVER` and `TX_NO_WAIT` can be set to this option. For details, see the description of `tx_queue_send()` ThreadX service call in the ThreadX User Guide.

Checking for a Pending Message

1. After the Messaging Framework module is opened, the message subscriber threads can wait for a message by calling `pend`. In general use, the second API argument specifies the pointer to a message queue, which you configured for the message subscriber thread in the Thread Subscribers pane in the Messaging tab, but you can specify the other message queues instead if required.
2. When a message is delivered from an event producer, the thread returns from `pend`.
3. The API returns the pointer to the buffer which contains the message to the thread through the third argument of the API.
4. The message subscriber casts the pointer above with a pointer type for the user custom message payload structure and does the event processing corresponding to the Event Class code sf_message_header_t::event_b.class_code, Event code `sf_message_header_t::event_b.code` and the user defined arbitrary data in the message.

Note that `pend` has the `wait_option` to change the behavior of the API function:

The fourth argument of pend is the wait time option, which is only valid if the message queue of the message subscriber thread is empty. Any arbitrary ThreadX tick count, `TX_WAIT_FOREVER`, and `TX_NO_WAIT` can be set to this option. For details, see `tx_queue_send()` ThreadX service call in the ThreadX User Guide.

5.5 Messaging Framework Module Interrupts

The Messaging Framework module does not use any interrupts.

6. Using the Messaging Framework Module in an Application

The typical steps in using the Messaging Framework module in an application are to first configure all the required settings as follows:

- Create a Message Queue
- Configure the Event Class and Event
- Configure the Subscriber List
- Configure the Event Class Code and Event Code

Once configuration is complete, the module's APIs can be used in the target application as follows:

1. Initialize the Messaging Framework with the `open` API
2. Acquire a buffer with the `bufferAcquire` API
3. Post a message with the `post` API
4. Check for a pending message with the `pend` API
5. Release a buffer using the `bufferRelease` API
6. Close the Messaging Framework with the `close` API

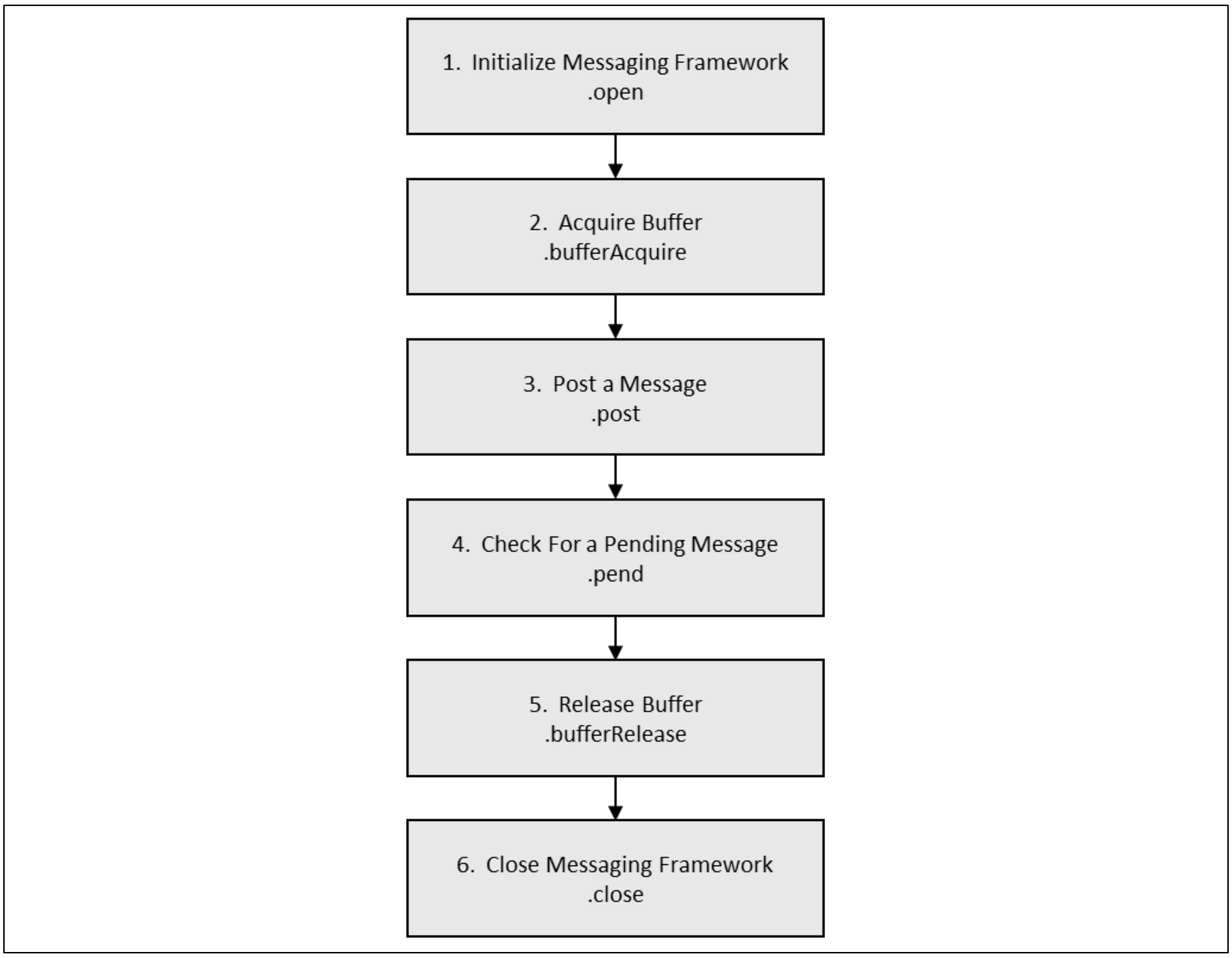


Figure 12. Flow Diagram of a Typical Messaging Framework Module Application

7. The Messaging Framework Module Application Project

The application project associated with this module guide demonstrates the aforementioned steps in a full design. The project can be found using the link provided in the References section at the end of this document. You may want to import and open the application project within ISDE and view the configuration settings for the messaging framework module. You can also read over the code (in `producer_thread_entry.c`, `led1_thread_entry.c`, `led2_thread_entry.c`, and `led3_thread_entry.c`) which illustrates the Messaging Framework APIs in a complete design.

The application project demonstrates the typical use of the Messaging Framework module APIs. The application project producer thread entry initializes the Messaging Framework and periodically selects a random message. The message contains information which determines which LED should be turned on or off. Each LED has its own handler thread which controls the diode; in these threads the message is received and handled. It executes the command and waits for the new one. The producer thread prints sent messages and the handler threads print received messages on the Debug Console using the common semi-hosting function. The following table identifies the target versions for the associated software and hardware used by the application project:

Table 5. Software and Hardware Resources Used by the Application Project

Resource	Revision	Description
e^2 studio	6.2.1	Integrated Solution Development Environment
SSP	1.5.0	Synergy Software Platform
IAR EW for Synergy	8.23.1	IAR Embedded Workbench® for Renesas Synergy™
SSC	1.5.0	Synergy Standalone Configurator
SK-S7G2	v3.0 to v3.3	Starter Kit

A simple flow diagram of the application project is given in the following figure:

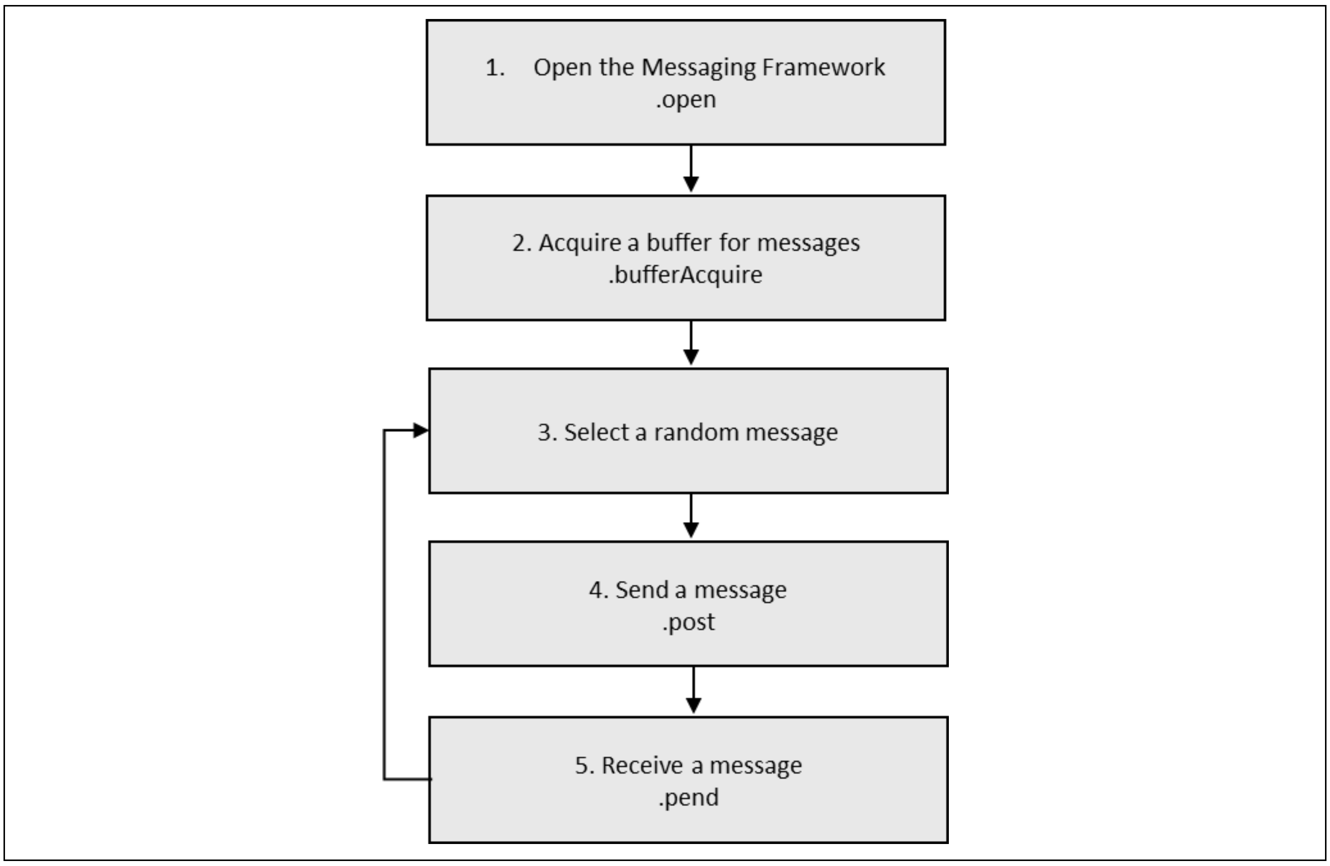

1. Open the Messaging Framework
.open
2. Acquire a buffer for messages
.bufferAcquire
3. Select a random message
4. Send a message
.post
5. Receive a message
.pend

Figure 13. Messaging Framework Module Application Project Flow Diagram

The complete application project can be found using the link provided in the References section at the end of this document. The `producer_thread_entry.c`, `led1_thread_entry.c`, `led2_thread_entry.c`, and `led3_thread_entry.c` files are located in the project once they have been imported into the ISDE. You can open each of these files within the ISDE and follow along with the description provided to help identify key uses of APIs.

The first section of `producer_thread_entry.c` has the header files which reference the Messaging Framework instance structure and a code section which allows semi-hosting to display results using `printf()`. The next section contains macro constants definition for a pseudo-random number generator. The pseudo-random number generator is used to select a message. Afterwards, global variables for the producer thread are defined followed by function prototypes.

The next section has function definitions. The first function toggles the binary state stored in a variable; based on its value, one of two events is returned. The function for selecting a random message follows. There are four event classes to select, each one can be chosen with a 25% probability. If the semi-hosting capability is enabled, a simple function is defined which maps an event code to a string literal. The last function gets a random message and sends it through the Messaging Framework instance. A short information message is displayed using semi-hosting showing which event has been sent.

The last section is the thread-entry function. This function initializes semi-hosting (if necessary) and acquires the message buffer with the `bufferAcquire` API. In the infinite `while` loop, the function for selecting and positing a message is called and the thread falls asleep for several ticks.

The first section of the LED Thread entry file (one of `led1_thread_entry.c`, `led2_thread_entry.c`, or `led3_thread_entry.c`) has header files which reference necessary thread-related variables and structures, including LED Thread configuration structure. Declaration of global variables follows: the first one is used as a pointer to a received message and the second contains thread configuration settings, including index of the controlled LED, LED-specific event class, and event codes for turning the diode on and off. The next section has the thread-entry function. At first, LED configuration is initialized; then in the 'forever' loop, the message is received. If the message is available, then the function for the message processing is called using thread-specific configuration settings. It checks for the type of event (an event class and a code) and executes the appropriate command.

Note: This description assumes you are familiar with using `printf()` with the Debug Console in the Synergy Software Package. If you are unfamiliar with `printf()`, see the Knowledge Base article, *How do I Use `Printf()` with the Debug Console in the Synergy Software Package*, available in the References section at the end of this document. Alternatively, you can see results via the watch variables in the debug mode.

A few key properties are configured in this application project to support the required operations and the physical properties of the target board and MCU. The properties with the values set for this specific project are listed in the following tables. You can also open the application project and view these settings in the Properties window as a hands-on exercise.

Table 6. Messaging Framework Configuration Settings for the Application Project

ISDE Property	Value Set
Parameter Checking	Default (BSP)
Message Queue Depth (Total number of messages to be `enqueued` in a Message Queue)	16
Name	g_sf_message
Work memory size in bytes	2048
Pointer to subscriber list array	p_subscriber_lists
Name of the block pool internally used in the messaging framework	sf_msg_blk_pool

To configure event classes, events, and subscriber lists, you should open Messaging tab and use the following settings.

Table 7. LED1 Event Class definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_CLASS_LED1
Name	LED1
Payload header file	led1_api.h
Payload	led1_payload
Payload type	led1_payload_t

Table 8. LED2 Event Class definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_CLASS_LED2
Name	LED2
Payload header file	led2_api.h
Payload	led2_payload
Payload type	led2_payload_t

Table 9. LED3 Event Class definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_CLASS_LED3
Name	LED3
Payload header file	led3_api.h
Payload	led3_payload
Payload type	led3_payload_t

Table 10. LED All Event Class definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_CLASS_LED_ALL
Name	LED All
Payload header file	led_all_api.h
Payload	led_all_payload
Payload type	led_all_payload_t

Table 11. LED1_ON Event definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_LED1_ON
Name	LED1_ON

Table 12. LED1_OFF Event definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_LED1_OFF
Name	LED1_OFF

Table 13. LED2_ON Event definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_LED2_ON
Name	LED2_ON

Table 14. LED2_OFF Event definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_LED2_OFF
Name	LED2_OFF

Table 15. LED3_ON Event definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_LED3_ON
Name	LED3_ON

Table 16. LED3_OFF Event definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_LED3_OFF
Name	LED3_OFF

Table 17. LED_ALL_ON Event definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_LED_ALL_ON
Name	LED_ALL_ON

Table 18. LED_ALL_OFF Event definition for the Application Project

ISDE Property	Value Set
Symbol	SF_MESSAGE_EVENT_LED_ALL_OFF
Name	LED_ALL_OFF

Table 19. Event Subscribers configuration for the Application Project

Thread\Event Class	LED1	LED2	LED3	LED All
LED1 Thread	✓			✓
LED2 Thread		✓		✓
LED3 Thread			✓	✓

Note: The application project uses only one instance of each event class, so start and end values for all event classes and threads should be set to 0.

8. Customizing the Messaging Framework Module for a Target Application

Some configuration settings will normally be changed by the developer from those shown in the application project. For example, the user can easily add new event classes or events. The user can also change the subscriber list for each event; this can be done using the Messaging tab in the configurator.

9. Running the Messaging Framework Module Application Project

To run the Messaging Framework module application project and to see it executed on a target kit, you can simply import it into your ISDE, compile, and run debug.

To implement the Messaging Framework application in a new project, follow the steps for defining, configuring, auto-generating files, adding code, compiling, and debugging on the target kit. Following these steps is a hands-on approach that can help make the development process with SSP more practical, while just reading over this guide tends to be more theoretical.

Note: The following steps are described in sufficient detail for someone experienced with the basic flow through the Synergy development process. If these steps are not familiar, refer to the first few chapters of the SSP User's Manual for a description of how to accomplish these steps.

To create and run the Messaging Framework application project, simply follow these steps:

1. Create a new Renesas Synergy project for the SK-S7G2 called Messaging_FW_MG_AP.
2. Select the **Threads** tab.
3. Add a new thread called

Symbol	led1_thread
Name	LED1 Thread

4. Add a new thread called

Symbol	led2_thread
Name	LED2 Thread

5. Add a new thread called

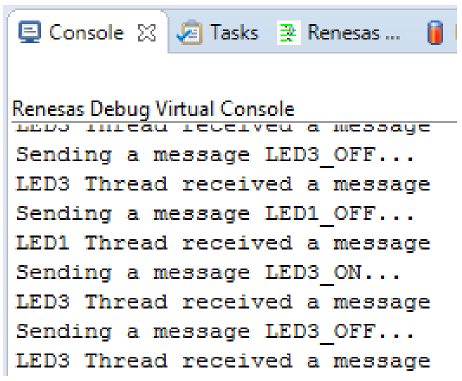
Symbol	led3_thread
Name	LED3 Thread

6. Add a new thread called

Symbol	producer_thread
Name	Producer Thread

7. Add the Messaging Framework to Producer Thread.
8. Configure the Messaging Framework instance.
9. Select the **Messaging** tab.

10. Add a new event class called
    Name LED1
11. Add a new event class called
    Name LED2
12. Add a new event class called
    Name LED3
13. Add a new event class called
    Name LED All
14. Add LED1 Thread to LED1 Subscribers and LED All Subscribers.
15. Add LED2 Thread to LED2 Subscribers and LED All Subscribers.
16. Add LED3 Thread to LED3 Subscribers and LED All Subscribers.
17. Add an event called
    Name LED1_ON
18. Add an event called
    Name LED1_OFF
19. Add an event called
    Name LED2_ON
20. Add an event called
    Name LED2_OFF
21. Add an event called
    Name LED3_ON
22. Add an event called
    Name LED3_OFF
23. Add an event called
    Name LED_ALL_ON
24. Add an event called
    Name LED_ALL_OFF
25. Click on the **Generate Project Content** button.
26. Add the code from supplied project files `led1_api.h`, `led2_api.h`, `led3_api.h`, `led_all_api.h`, `led_api.c`, `led_api.h`, `semihosting_cfg.h`, `led1_thread_entry.c`, `led2_thread_entry.c`, `led3_thread_entry.c`, and `producer_thread_entry.c` or copy over these files.
27. Connect to the host PC via a micro USB cable to J19 on SK-S7G2 Kit.
28. Start to debug the application.
29. The output can be viewed on the debug console (Renesas Debug Virtual Console).



```
LED3 Thread received a message
Sending a message LED3_OFF...
LED3 Thread received a message
Sending a message LED1_OFF...
LED1 Thread received a message
Sending a message LED3_ON...
LED3 Thread received a message
Sending a message LED3_OFF...
LED3 Thread received a message
```

Figure 14. Example Output from Messaging Framework Application Project

10. Messaging Framework Module Conclusion

This module guide has provided all the background information needed to select, add, configure, and use the module in an example project. Many of these steps were time consuming and error-prone activities in previous generations of embedded systems. The Renesas Synergy™ Platform makes these steps much less time consuming and removes the common errors, like conflicting configuration settings or the incorrect selection of lower-level drivers. The use of high-level APIs (as demonstrated in the application project) illustrates additional development time savings by allowing work to begin at a high level and avoiding the time required in older development environments to use or, in some cases, create, lower-level drivers.

11. Messaging Framework Module Next Steps

After you have mastered a simple Messaging Framework project, you may want to review a more complex example. You may see the Audio Playback Framework or the Touch Panel Framework Module Guides, both of which make use of the Messaging Framework.

You may find that simple ThreadX message queue is a better fit for your target application. If you would like to review ThreadX APIs, then the *ThreadX User's Manual* is the right place to start.

The *SSP User's Manual* and *ThreadX® User's Manual* are available as described in the References section at the end of this document.

12. Messaging Framework Module Reference Information

SSP User Manual: Available from the Renesas Synergy Software Package website: https://www.renesas.com/us/en/products/synergy/software/ssp.html

Links to all the most up-to-date sf_message module reference materials and resources are available on the Synergy Knowledge Base: https://en-support.renesas.com/knowledgeBase/16977549.

Website and Support

Visit the following vanity URLs to learn about key elements of the Synergy Platform, download components and related documentation, and get support.

Synergy Software	www.renesas.com/synergy/software
Synergy Software Package	www.renesas.com/synergy/ssp
Software add-ons	www.renesas.com/synergy/addons
Software glossary	www.renesas.com/synergy/softwareglossary
Development tools	www.renesas.com/synergy/tools
Synergy Hardware	www.renesas.com/synergy/hardware
Microcontrollers	www.renesas.com/synergy/mcus
MCU glossary	www.renesas.com/synergy/mcuglossary
Parametric search	www.renesas.com/synergy/parametric
Kits	www.renesas.com/synergy/kits
Synergy Solutions Gallery	www.renesas.com/synergy/solutionsgallery
Partner projects	www.renesas.com/synergy/partnerprojects
Application projects	www.renesas.com/synergy/applicationprojects
Self-service support resources:	
Documentation	www.renesas.com/synergy/docs
Knowledgebase	www.renesas.com/synergy/knowledgebase
Forums	www.renesas.com/synergy/forum
Training	www.renesas.com/synergy/training
Videos	www.renesas.com/synergy/videos
Chat and web ticket	www.renesas.com/synergy/resourcelibrary

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jun.15.17	—	Initial version
1.01	Aug.01.17	—	Update to Hardware and Software Resources Table
1.02	Sep.06.17	—	Added checks for debugging to semi-hosted output Updated this doc with feedback from outside reviewer
1.03	Feb.01.19	—	Updates to api calls and configuration settings throughout

.

Corporate Headquarters

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

Trademarks

Contact information

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