Renesas Synergy™ Platform

Getting Started with the Thermostat Application for S7G2 and S5D9

Introduction

This application note describes a simulated Thermostat control application. The Thermostat application is geared towards providing users with a quick out-of-box experience that demonstrates how complex multi-threaded applications with a touch screen graphical Human Machine Interface (HMI) can be developed using the Renesas Synergy™ Software Package (SSP).

The Thermostat application runs on several different development boards including the DK-S7G2, PK-S5D9, SK-S7G2 and PE-HMI boards. These boards differ in screen resolution and have slightly different feature sets. As an example, the more full-featured boards such as the DK-S7G2 and PE-HMI boards provide backlight control of the LCD, while the lower cost PK-S5D9 and SK-S7G2 boards do not include this ability. This application note covers the DK-S7G2 while noting changes required to run the Thermostat application on other boards.

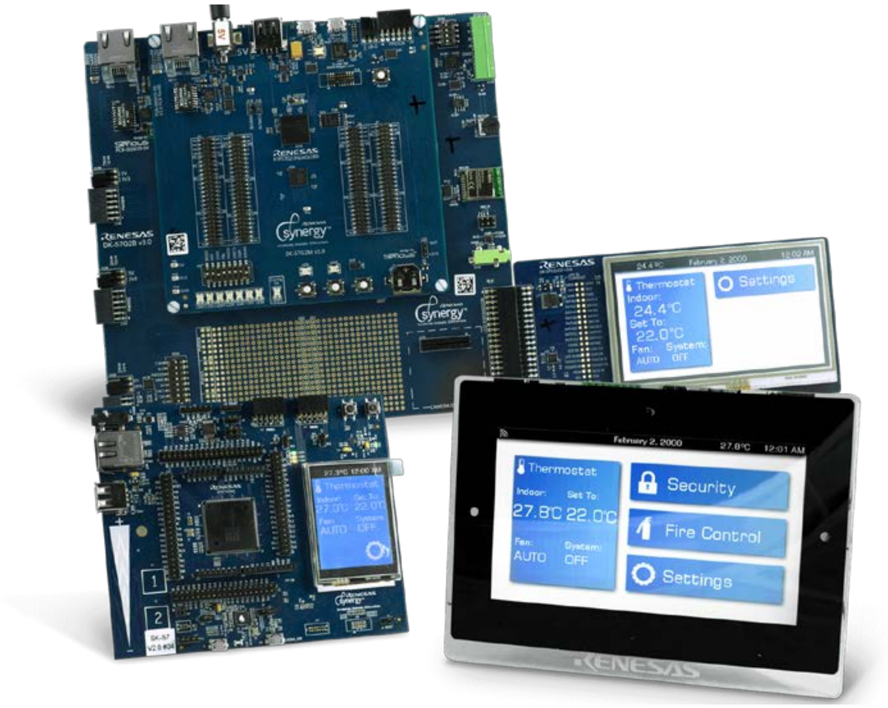


Figure 1. Thermostat Application on Several Development Boards

The Thermostat application was developed using the Synergy Software Package (SSP), version 1.5.0. The SSP is a unified robust Application Framework that includes driver level support for peripherals in the Synergy Arm® Cortex®-M4/M0+ Cores along with ThreadX®, Real Time Operating System (RTOS) from Express Logic, Inc. In addition to ThreadX, full stack support is available through the X-Ware suite of stacks such as NetX™, USBX™, GUIX™, and FileX™ from Express Logic. This powerful suite of tools provides a comprehensive integrated framework for rapid development of complex embedded applications.

This application note assumes that you are familiar with the concepts associated with writing multi-threaded applications under an RTOS such as ThreadX. While specific knowledge of ThreadX makes understanding of the code even easier, you should be able to understand the information provided in this application note if you have any previous experience with RTOS principles such as threads, message queues, semaphores, and mutexes.

The Thermostat application was developed using the Renesas Synergy™ e² studio Integrated Solution Development Environment (ISDE). This eclipse-based ISDE can be freely downloaded from Renesas. While e² studio supports the use of multiple tool chains, this application note was built using the GCC compile tools that come free with the e² studio ISDE environment.

Building applications using the Renesas Synergy™ Platform is considerably faster than developing similar applications in other environments, yet there is a learning curve to understand the required steps to construct complex multi-threaded HMI applications quickly. This application note guides you through all the necessary steps including the following:

- Board setup
- Loading and running the project
- Application overview
- Detailed explanation of the graphical screen uses
- GUIX Studio project integration
- SSP Application Framework configuration
- Application design highlights
- Inter-thread communication using the Synergy messaging framework
- Reading the internal temperature sensor of the MCU with the ADC unit
- Using the General-Purpose Timer (GPT) to drive a PWM backlight control signal
- Using the Audio Framework

Prerequisites

It is assumed that you have some experience with the Renesas Synergy e² studio ISDE and SSP. For example, before you perform the procedure in this application note, you should follow the procedure in the *Quick Start Guide* of your board to build and run the **Blinky** project. By doing so, you will become familiar with e² studio and the SSP and ensure that the debug connection to your board is functioning properly.

Required Resources

The example application targets Renesas Synergy S7G2 and S5D9 MCU Groups. To build and run the application, you will need:

- A Synergy DK-S7G2 board v3.0/3.1 with the included LCD module (see Figure 2).
- A PC running Microsoft® Windows® 7 with the following Renesas software installed:
  - e² studio ISDE v6.2.1 or later
  - Synergy Software Package (SSP) v 1.5.0 or later
  - IAR Embedded Workbench® for Renesas Synergy™ v8.23.1 or later
  - SSC v6.2.1 or later
  - GUIX Studio™ v5.4.0.0 or later

You can download the required Renesas software from the Renesas Synergy™ Gallery at www.renesas.com/synergy/software.

See the *Renesas Synergy™ Project Import Guide* (r11an0023eu0121-synergy-ssp-import-guide.pdf), included in the package, for instructions on importing the project into e² studio and building/running the project.

Target Devices

- DK-S7G2 v3.0/v3.1/V4.1 Development Kit (Synergy S7G2 MCU Group)
- PE-HMI 1 v2.0 Product Example Kit
- PK-S5D9 v1.0 Promotion Kit (Synergy S5D9 MCU Group)
- SK-S7G2 v2.0 Starter Kit (Synergy S7G2 MCU Group)

Contents

1. Board Setup

The DK-S7G2 kit contains three PCBs. The main board provides interconnects for the MCU board and the LCD screen. The MCU main board and the daughter board have several switches that must be configured prior to running the firmware associated with this application note. In addition to these switches, these boards also contain multiple USB connectors, Ethernet connectors, and PMOD connectors. This application note does not use Ethernet and uses only one of the micro USB connectors to access the SEGGER J-Link® programming interface.

Connect the supplied USB cable between J17 of the DK-S7G2 and the PC on which you have loaded the Synergy e[2] studio software. Connector J17 is the upper right USB port as you look at the board from the orientation shown in Figure 2. For proper operation of the application, use the following table to ensure that the three DIP switches (S5, S101, S102) are set correctly.

Table 1. Configuration of S5, S101, and S102 Switches

S5	Switch 1 and 6 on, all others off
S101	All switches off
S102	All switches off

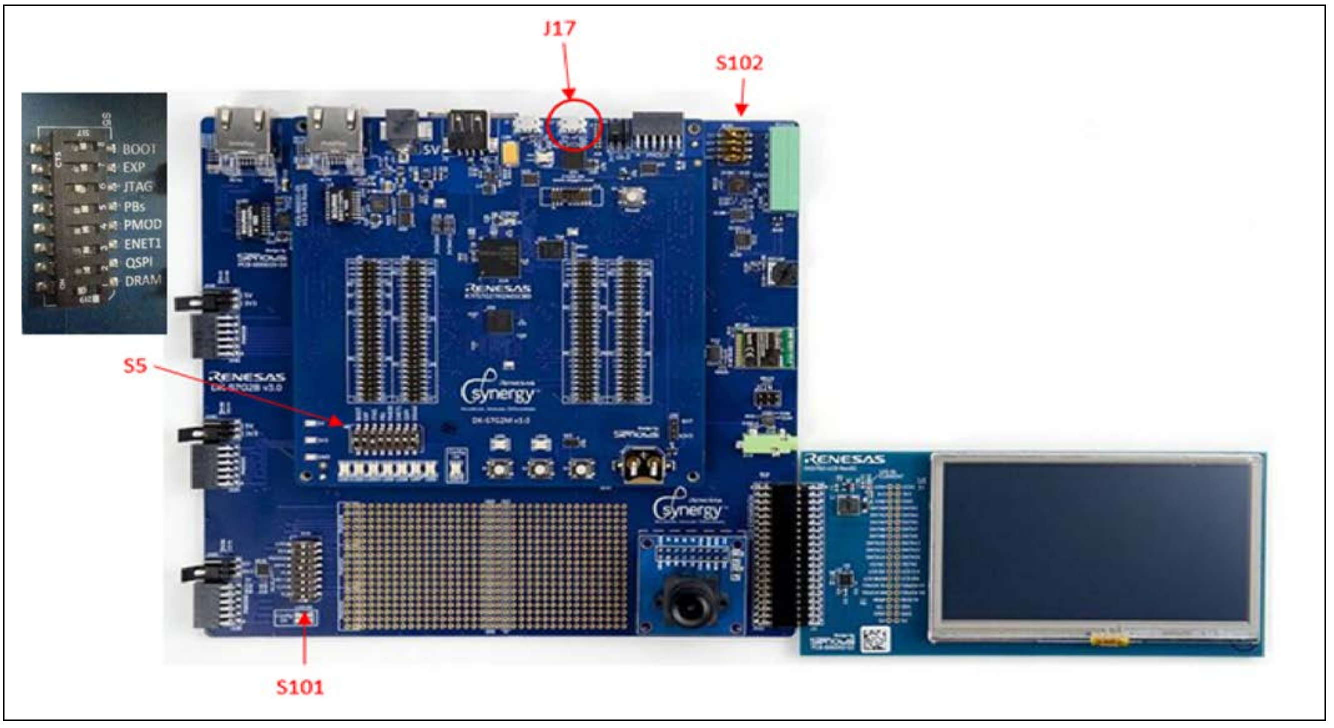


Figure 2. Items to Configure on the DK-S7G2 Board

2. Application Overview

One of the key objectives of the Thermostat application is to demonstrate how to build applications using complex HMI screens with GUIX and GUIX Studio. The Thermostat application is one of the most complex HMI designs among the Synergy application notes. The following list highlights all the key features of the Thermostat application:

- Complex HMI design using GUIX Studio
- Multi-threaded applications using the ThreadX RTOS
  - Queue and Mutex Thread objects used
- Extensive use of Synergy Messaging Framework for inter-thread communication
- GLCD configuration for various screen types and sizes
  - Frame buffer run from internal and external memory
  - External memory interface used
  - Serial Peripheral Interface (SPI) initialization of ILI9341 Graphics Controller (SK-S7G2 board)
- Touch Panel, I²C touch controller driver ft5x06
  - External IRQ mapping required
- Use of the Real Time Clock (RTC) driver for date/time
- PWM control of LCD backlight
- Audio Playback Framework used for LCD screen touch feedback
- ADC system used to read on-board temperature sensor.

In any software design, there are many ways to solve the same problem. The solution given in this application note is one approach.

2.1 S7G2 and S5D9 Synergy MCU Group Peripherals Used by the Thermostat Application

At the time of the writing of this application note, all the development boards that run the Thermostat application use the S7G2 and S5D9 Synergy MCU Groups. The MCU is built around an Arm® Cortex®-M4 core. Developing complex embedded applications is usually a multi-step process.

1. The first step usually involves gathering the application requirements and performing a high-level system design that maps the requirements onto the set of hardware components that are necessary, including the target MCU that is to be used in the design, the tool chains required to build and debug the applications.
2. The next step is to determine which on-board peripherals of the target MCU are to be used. In this step, it is often necessary to spend a considerable amount of time to understand the register map of the on-board peripherals and to write lower level driver code required to expose the peripheral to the upper level application code. However, most of this work has already been done in the SSP Application Framework to considerably streamline the application development.
3. In addition to the on-board peripherals of the target MCU, the design often includes external hardware that specify how it can be controlled. As an example, the DK-S7G2 and PE-HMI boards have LCD screens that may be controlled directly by the on-board Graphics LCD Controller (GLCD) of the S7G2, S5D9 Synergy MCU Groups. The PK-S5D9 and SK-S7G2 boards use a smaller, lower cost LCD over a serial interface that requires some initialization, before it can be controlled by the GLCD of the S7G2 MCU Group.
4. The last step details how an application should be structured on top of the selected hardware to satisfy the initial requirements.

To follow the specified process, the Thermostat application requirements were first mapped to the on-board peripherals of the S7G2 and S5D9 Synergy MCU Groups. The following illustration shows all the internal hardware peripherals utilized by the Thermostat application. This application note describes how each of these peripherals is configured using the SSP Application Framework, and considered for each peripheral when the application was being developed.

Figure 3. S7G2 Synergy MCU Group Peripherals used in the Thermostat Application

2.2 HMI

The most daunting task in many HMI applications is possibly the GUI itself. In applications requiring a graphical HMI, it is best practice to separate the business logic from the presentation. This simply means that the GUI does not make decisions about what to display but rather how to display it. The GUI relies on external logic for what to display and when to display it.

After you have gathered all the requirements, performed a top-level design, and identified the hardware necessary to implement the design, it is useful to design a GUI.

The SSP natively supports the use of GUIX from Express Logic. You may choose to use GUIX primitive calls directly from Express Logic, or you may choose to use GUIX Studio to design the screens. GUIX Studio is a standalone tool that provides a point and click environment for generating all the screens necessary for your embedded application. Once designed, the studio outputs `.c` and `.h` files that you can integrate into your application. All the application screens in the Thermostat application were built using GUIX Studio.

2.3 Thermostat Screens

Screen designs are normally tailored to the size of the screen they are displayed on. This often requires multiple graphical designs when porting an application to different boards with different sized LCD screens. There are two approaches to this problem. The first approach is to build separate static display designs for each screen resolution. GUIX Studio allows you to do this very quickly and this is the approach used for this application note. The second approach is to build the screens dynamically and to size the windows and widgets at run time depending on the screen resolution available. GUIX has a rich API that allows this type of dynamic screen generation, but building screens dynamically using this interface is beyond the scope of this application note.

The Thermostat application has two different designs, one for large screens like the 4.3-inch screen found on the DK-S7G2 board or the 7-inch screen found on the PE-HMI board, and one for smaller screens like the one found on the SK-S7G2 board.

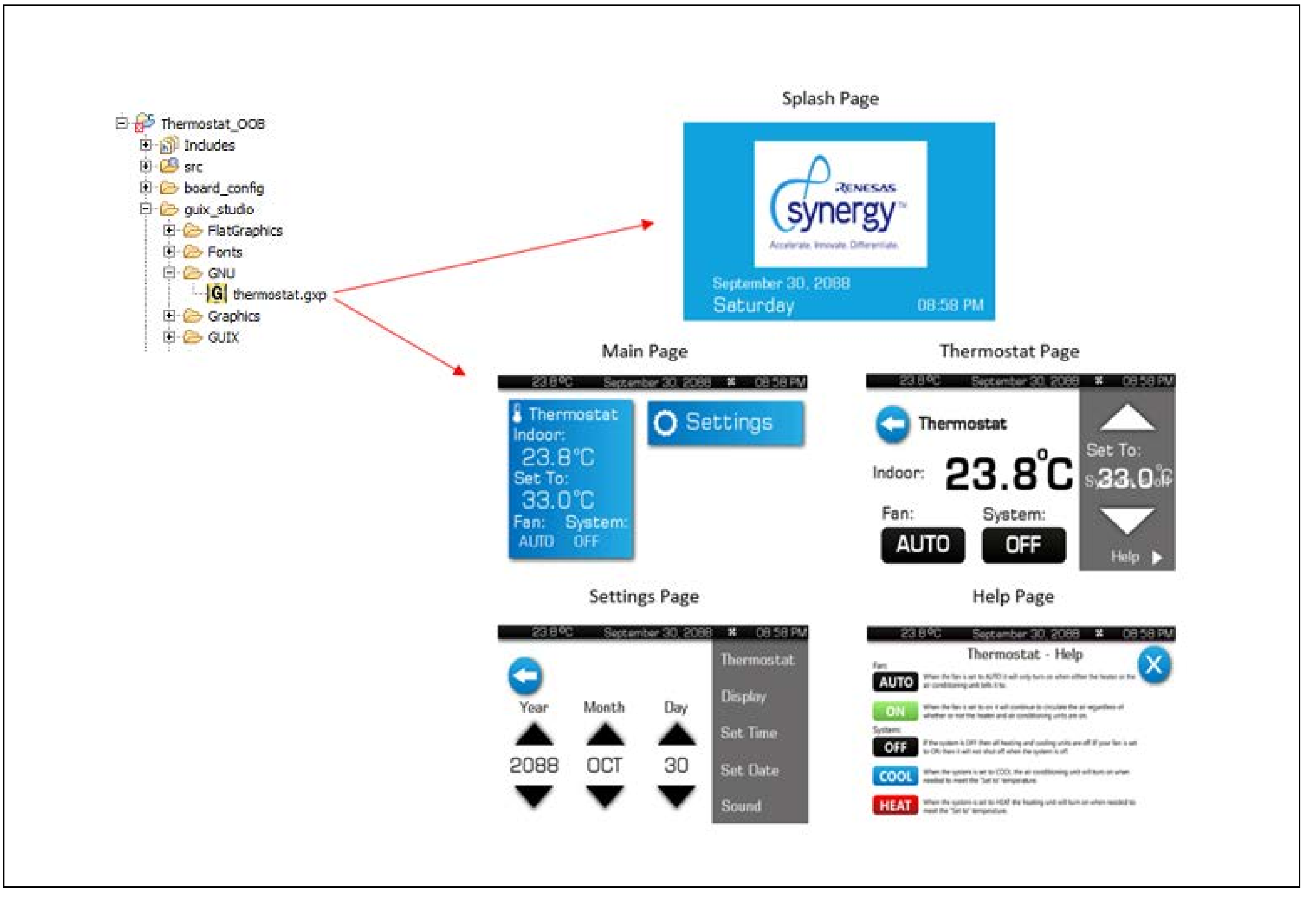


Figure 4. Screen Snapshots for PE-HMI, and DK-S7G2 Boards

2.3.1 Large Screen Design

For larger screens such as the DK-S7G2 or the PE-HMI development boards, the Thermostat application contains five main screens as shown in Figure 4, that are named as follows:

Splash Page	The initial screen that is displayed when the application is first run.
Main Page	The main page displayed after the user touches the splash screen.
Settings Page	Accessed from the main page, this screen allows user to change various system settings.
Thermostat Page	Screen to adjust thermostat settings such as set point, heat/cool mode, and fan state (auto/on).
Help Page	Screen that highlights what each thermostat control button does.

While the same five screens apply to both the DK-S7G2 and the PE-HMI boards, two separate graphical designs exist because it was necessary to scale down the design for the smaller screen of the DK-S7G2 board. A GUIX Studio project includes various resource files such as fonts and images, but as is the case with many IDEs, the project definition itself is actually maintained in a single xml file with a .gxp extension. A separate .gxp file exists for all three board designs.

2.3.2 Small Screen Design

Figure 5 shows a small screen design used for the SK-S7G2 board. The key change in this design is the separation of the various settings screens from the Settings menu and the elimination of the Help Page.

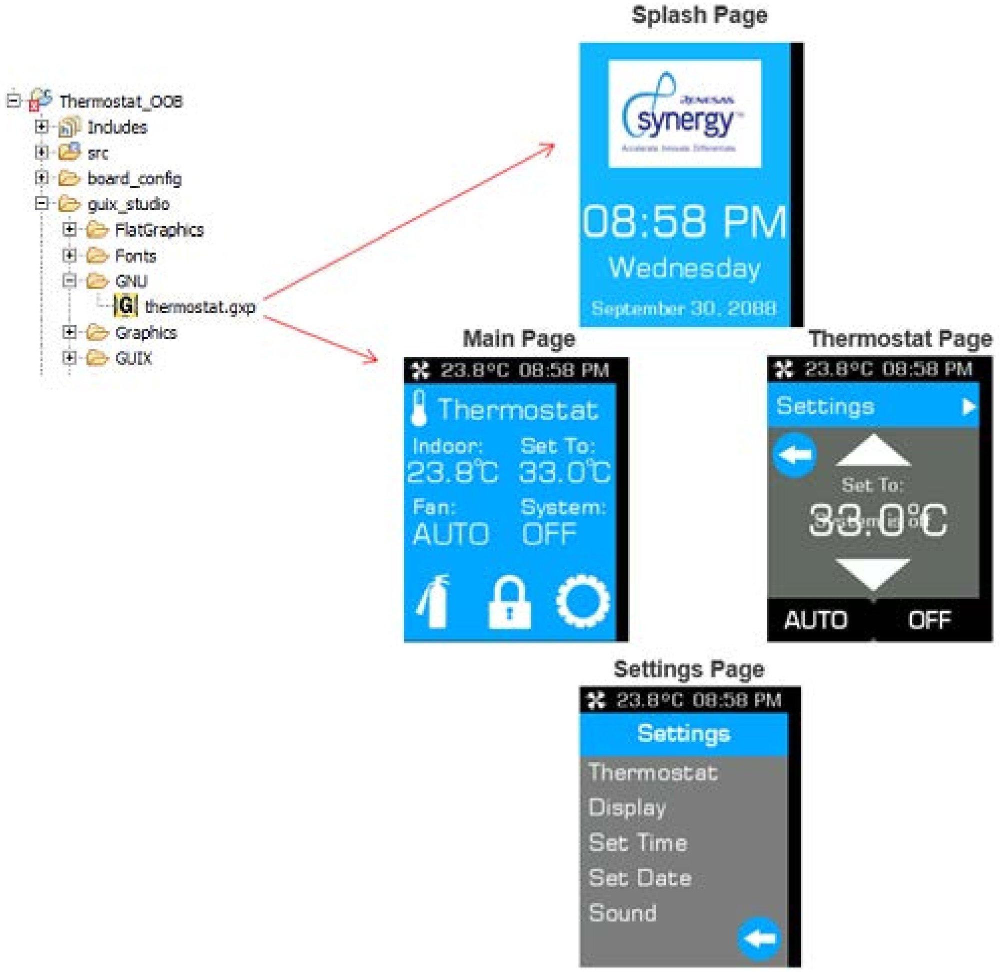


Figure 5. Screen Snapshots for the SK-S7G2 Board

In addition to these main pages, additional settings screens exist for changing the units from Fahrenheit to Celsius, adjusting the screen brightness, setting the time and date, and adjusting the sound volume. You can easily see how these screens look by running the application or examining the thermostat.gxp file using GUIX Studio.

3. GUIX Studio Overview

This section provides an overview of how GUIX screens are designed and integrated into an SSP application using GUIX Studio. It is not meant as a replacement for the GUIX or GUIX Studio documentation. When designing graphical interfaces for the Renesas Synergy Platform, you are encouraged to read the full documentation for GUIX and the GUIX Studio while learning the associated screen handling code in the Thermostat application.

GUIX Studio presents a graphical, point-and-click environment that allows you to quickly create all the required screens for your embedded application. You can specify the screen resolution, color depth, and various other parameters such that what you see in GUIX Studio, running on your desktop PC, is what you get on your embedded screens.

GUIX comes standard with a few fonts and basic graphics for things like button controls. During your screen creation phase, you may provide GUIX with your own external images and font files to make your displays as fancy as needed. GUIX Studio also provides the use of multi-language displays using string tables.

Figure 6 is a screen shot of the Thermostat page designed in GUIX Studio.

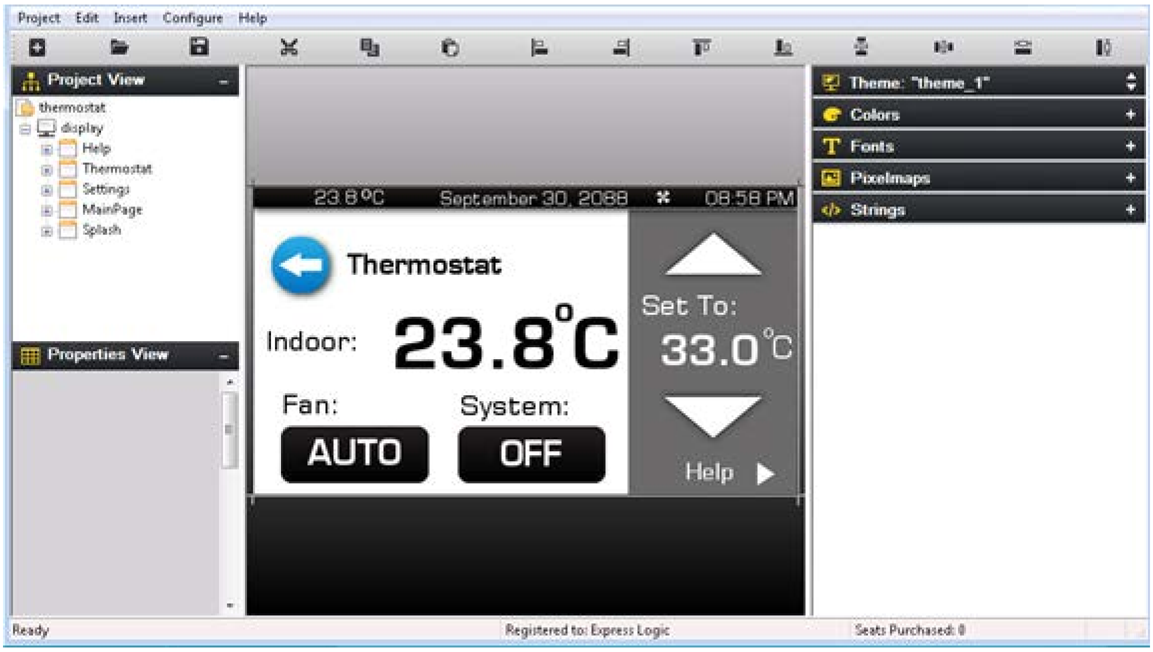


Figure 6. Thermostat Page Design using GUIX Studio

The organization of the GUIX Studio IDE is straightforward. The center screen, known as the **Target View**, contains the screen being designed. The upper left shows the **Project View**. This pane shows the widgets contained in your project. The order you add items to the project determines the order they are drawn in the final screens, and therefore some planning is necessary. As is the case with most graphical design environments, screens are laid out in a hierarchy where the main window is normally the parent window, and all graphical objects contained in the window are children of that parent. The **Properties View** (lower left) displays properties associated with a selected object. You may select objects from the **Project** or **Target View**.

The far right-side of the GUIX Studio screen contains drop-downs for all the various resources such as colors, fonts, images (pixel maps) and strings used to create the screens. GUIX supports multi-language designs using string tables.

The key to making any graphical design interactive is to associate events such as screen touches, with the event handling code that implements the appropriate functionality. As you design your screens, you can associate callback functions with your widgets. These callback functions provide the hooks necessary in your application to respond to GUI events.

GUIX Studio provides both Draw and Event callbacks. Event functions allow you to respond to typical events such as touch events. Draw functions allow you to add customized drawing. The Thermostat application only defines Event function callbacks and only on the top-level windows. The callback function names are entered into the Event function field of the **Properties View** as shown in Figure 7.

For development boards with larger screens, the Thermostat GUIX design has five defined Event functions and are named as follows:

- `help_screen_event`
- `thermostat_screen_event`
- `settings_screen_event`
- `mainpage_event`
- `splashscreen_event`

Development boards with smaller screens such as the SK-S7G2, simply omit the help screen and associated Event function.

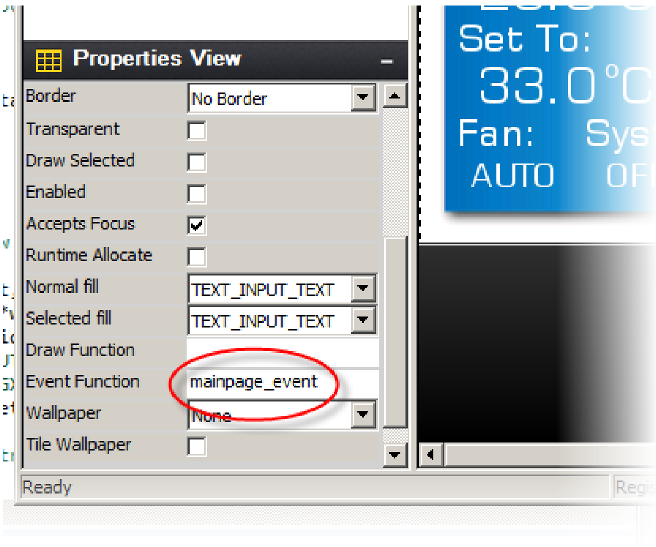


Figure 7. Main Page Event Function Defined

When you are finished with your GUIX Studio design, you can instruct GUIX Studio to generate all the output files by selecting **Configure** -> **Project/Displays** from the top menu ribbon as shown in Figure 8.

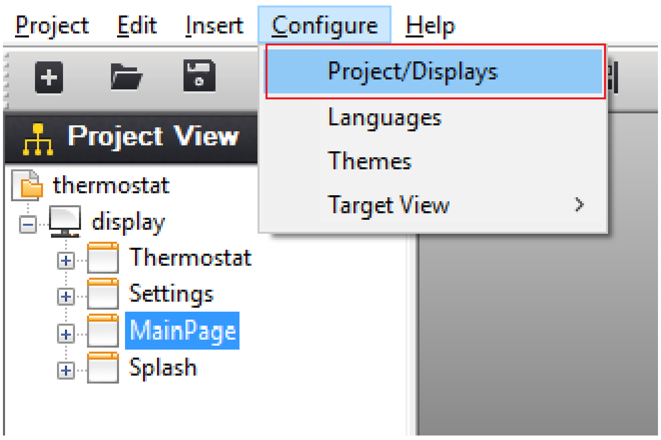


Figure 8. Configure Project/Displays Selection

The Configure Project dialog box is displayed as shown in Figure 9. This dialog box is where you specify project-specific information such as the basic display settings in addition to the path information where GUIX locates the files that result from the build process.

When you build your project, GUIX Studio creates `.c` and `.h` files that contain all the information required to render the screens built with GUIX Studio on the LCD of your embedded Synergy application. The Directories group is where you specify the default output directories for the source and header files. Additionally, you may also specify a directory location where all the resource files such as images are saved.

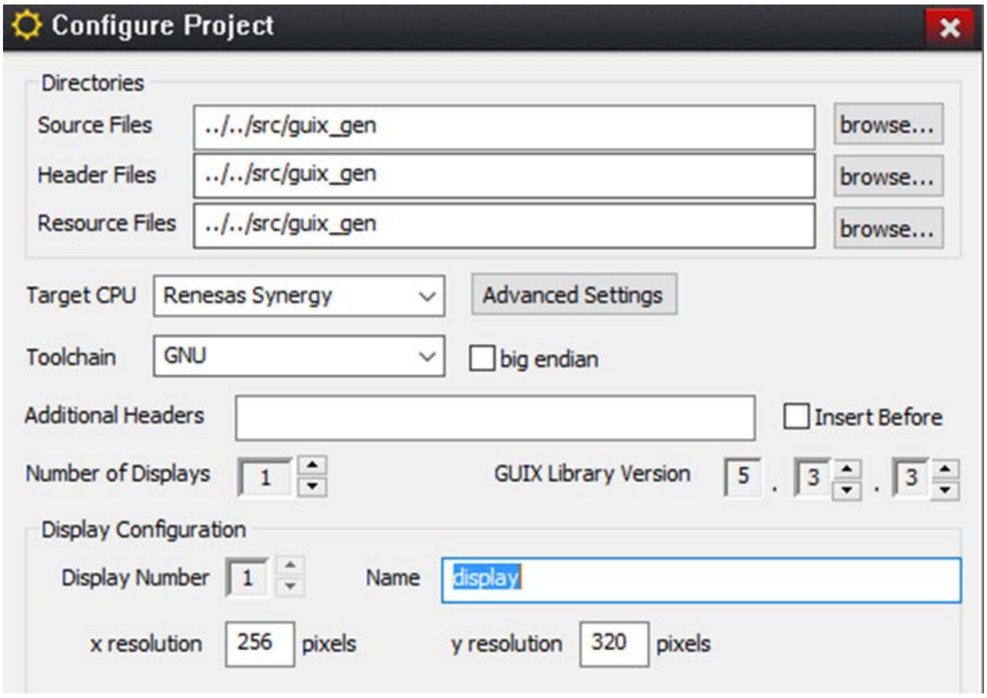


Figure 9. Define Project Paths

It is good practice to save the source, header, and resource files relative to the project location. This makes it easy to move projects from one location to another or from one PC to another. For the Thermostat application, all directories are located in the `src/s7g2_dk` directory.

Initially, the Thermostat application has a `guix_studio` directory containing the original resource files and the `thermostat.gxp` file as shown in Figure 10. If you have GUIX Studio installed, you can click on the `thermostat.gxp` file to launch GUIX Studio.

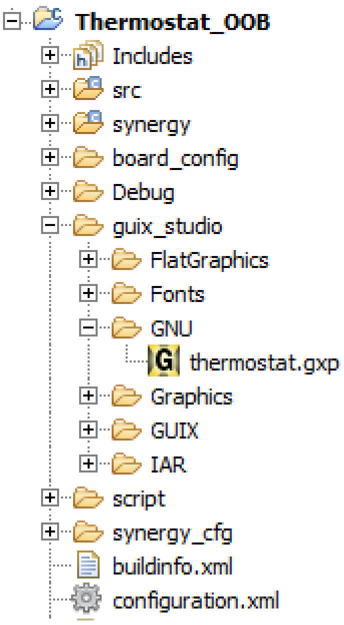


Figure 10. GUIX Resources Folder

As specified earlier, the outputs of the GUIX Studio are `.c` and `.h` source files that must be compiled into your project. The GUIX project for the DK-S7G2 Thermostat application contains five Event Handler functions, one for each top-level screen. GUIX automatically builds function prototypes for these callback functions in the `thermostat_specifications.h` file as shown below:

```
/* Declare event process functions, draw functions, and callback functions     */

UINT help_screen_event(GX_WINDOW *widget, GX_EVENT *event_ptr);
UINT thermostat_screen_event(GX_WINDOW *widget, GX_EVENT *event_ptr);
UINT settings_screen_event(GX_WINDOW *widget, GX_EVENT *event_ptr);
UINT mainpage_event(GX_WINDOW *widget, GX_EVENT *event_ptr);
UINT splashscreen_event(GX_WINDOW *widget, GX_EVENT *event_ptr);
```

In addition, the smaller GUIX Studio designs used for the SK-S7G2 boards do not include a Help page, therefore the `help_screen_event()` handler is not present in these applications.

While GUIX Studio defines function prototypes for event handlers, you can create files that contain the code for each of these handlers. The next section details the source code layout for the Thermostat application.

4. Analyzing the Application

While understanding the HMI is important, there are many other areas that you should understand while developing the Synergy applications. These areas include how the project is physically structured in e[2] studio, how threads and thread resources are added to the project, how threads communicate, how the state machine is designed, and how state data is shared among cooperating threads.

4.1 Source Code Layout

Before diving into the application code, it is best to first understand the overall source code layout of a Synergy project. Synergy applications generally consist of two different types of code, user-generated and auto-generated. The auto-generated code can be further broken down into two sub-categories, code that is auto-generated by the Synergy Framework versus code that is auto-generated by GUIX Studio.

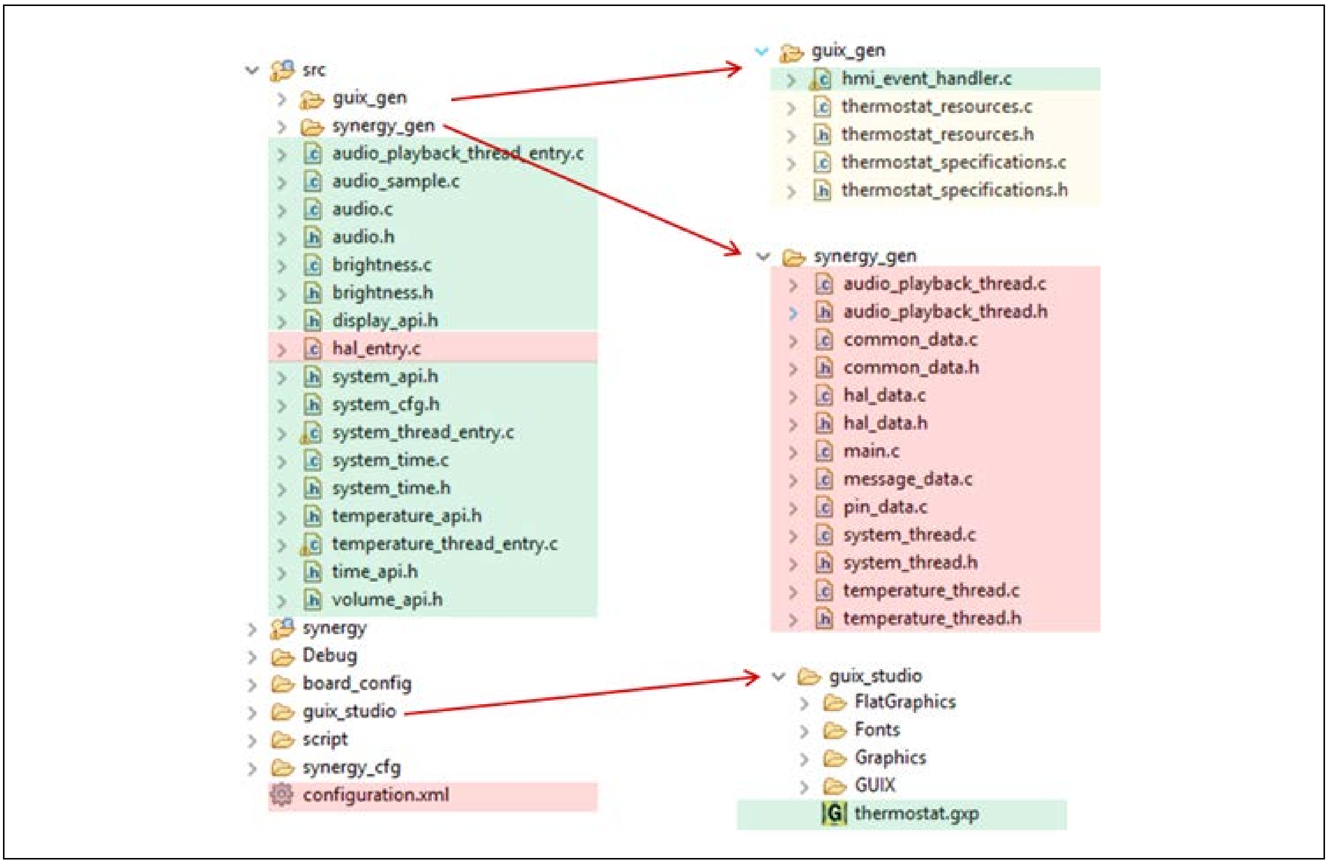


Figure 11. Source Code Layout

Figure 11 shows the source code layout for the DK-S7G2 board. The Application Framework auto-generated code is highlighted in red, GUIX auto-generated code is highlighted in yellow, and user-generated code is highlighted in green.

Note: Most of the user-generated code resides in the `src` directory with the exception of the GUIX Studio project files, `thermostat.gxp`, and `hmi_event_handler.c` which contain the event handlers for the HMI.

4.1.1 State Machine

The cornerstone of most embedded applications is the state machine. This is the part of the code that dictates the overall functionality of the machine. The key feature of the Thermostat application centers around system control based on the indoor temperature versus the set to temperature (set point) programmed into the Thermostat as seen in Figure 12.

In a real thermostat, a separate temperature sensor, perhaps connected to the SPI or I^2C bus of the processor, is most likely used to accurately reflect the indoor temperature. Because most of the currently available Renesas Synergy development boards do not contain a separate external temperature sensor, the decision was made early on to use the on-board temperature sensor of the Synergy MCU to provide a simulated indoor temperature reading.

Note: Some of the development boards have external PMOD connectors. The SK-S7G2 board can also accept an Arduino, which can accommodate an external temperature sensor. If you want to replace the on-board temperature sensor with an external sensor, read an external sensor with either the SPI or I^2C bus.

In separating presentation from business logic, the Thermostat application divides the application into separate parts, the GUI which is responsible for displaying the current state of the machine, and the state machine which maintains the system state and is responsible for determining when changes to that system state are permitted.

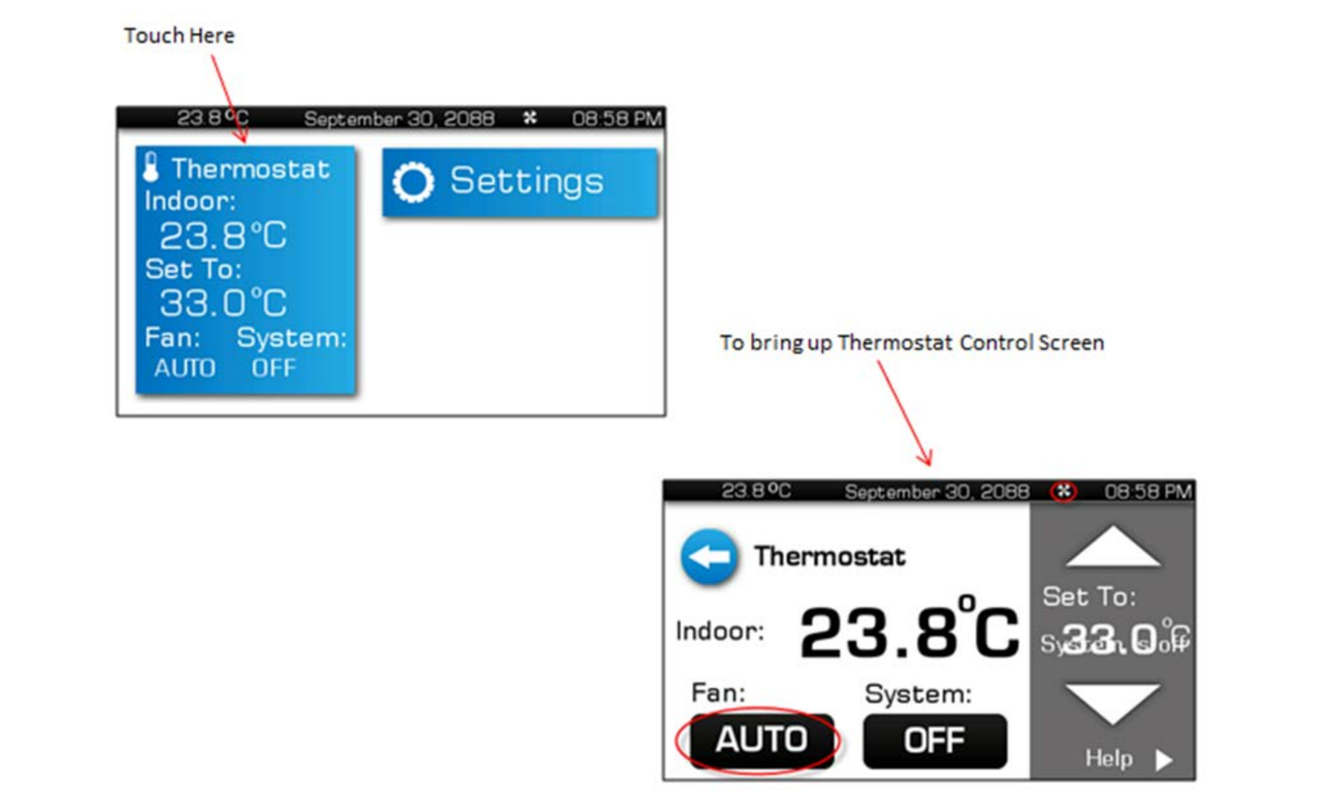


Figure 12. Touch Control for Screen Changes

Touching the Thermostat section of the Main Page display brings you to the Thermostat Control Screen page that should look like the one shown in Figure 12. The controls on this screen are the principle means in which you can change the Thermostat state machine. The flowchart in Figure 13 details the basic operation of the Thermostat system.

When you press the **Auto** button in the lower left-hand corner, the state of the button toggles from auto with a black background to on with a green background. Whenever the fan state is on, the small fan icon in the notification bar appears. When the system is in the auto mode, the fan icon appears and disappears whenever the indoor temperature is above or below the set point temperature depending on the system mode (heat or cool).

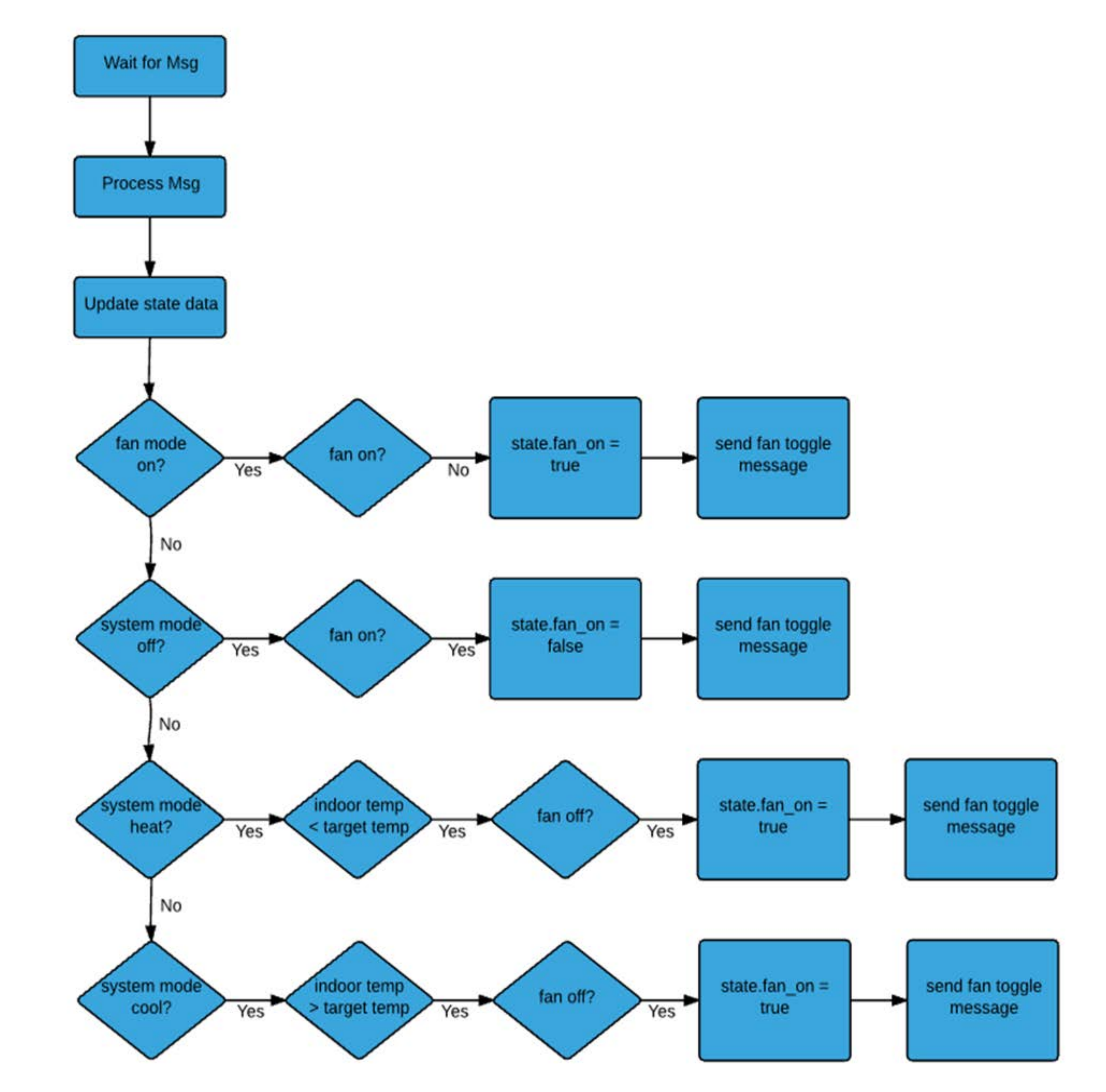


Figure 13. Thermostat Control Flowchart

4.2 Thread Overview

As described in the introduction, the Thermostat application is a multi-threaded application, running in ThreadX. There are two origins of threads in a Synergy application, those created by the programmer, and those created by the SSP Application Framework. While it is obvious what threads you created as a programmer, it is not always obvious what threads are created by the Application Framework. As explained in the *SSP User's Manual*, there are two principle types of modules you add to a Synergy application, the Driver module, and the Framework module. Driver modules are described as RTOS-aware but generally do not use any RTOS objects. Framework modules can use RTOS objects, such as semaphores or mutexes, and can also create their own threads as needed.

The Thermostat application uses two user-created threads, the System thread and the Temperature thread. The two threads communicate through the Synergy messaging framework that is layered on top of the standard ThreadX message queues. The System thread processes touch messages, GUIX events, and maintains the system state data. The Temperature thread periodically reads the on-board temperature sensor of the ARM core and, if it has changed, posts a message to the System thread. The Framework Configuration section that follows provides details on how to add user threads to your application.

Figure 14 shows a high-level design of the threads and messaging running on the Thermostat application. Notice the distinction between user threads and framework threads. From this diagram, in addition to the System thread and Temperature thread, there are also threads associated with GUIX and the touch controller.

Note: Not shown in the diagram is the thread created by the Audio Framework to process the click sound that is played whenever the user touches the screen. Threads created by the SSP Application Framework modules are not always apparent and can generally be disregarded by the application programmer.

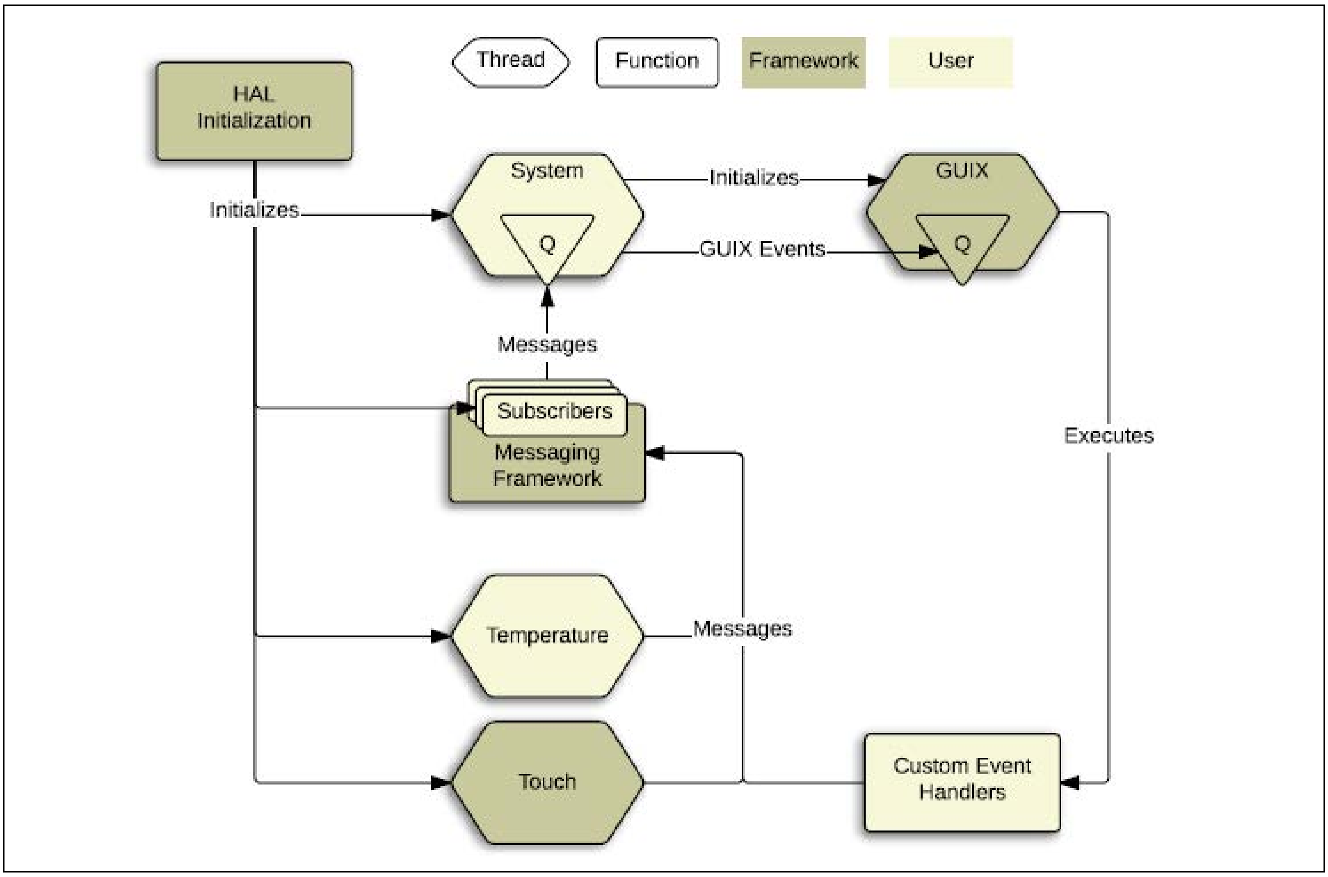


Figure 14. Thermostat threads

In addition to the software component, various hardware components are also accessed through the hardware drivers provided by the SSP Application Framework. These include the clock generation circuit, touch screen controller (I^2C), external interrupts, and the ADC (analog to digital converter) unit of the ARM core.

4.2.1 System Thread

The system thread initializes various services used by the Thermostat application including GUIX, the Audio Playback Framework, and the RTC. On the SK-S7G2 board, the system thread must also initialize the LCD screen to place it into the proper RGB mode so it can be controlled by the GLCD peripheral of the S7G2 Synergy MCU.

Once this initialization is complete, the system thread processes various system inputs including touch messages, GUIX events, RTC interrupt, and temperature update messages from the Temperature thread. If any of these inputs results in a change to the system state, the system thread updates the state data structure, and sends the appropriate update messages to the GUIX thread, resulting in changes to the graphical HMI. The flowchart in the following figure shows the high-level design of the system thread.

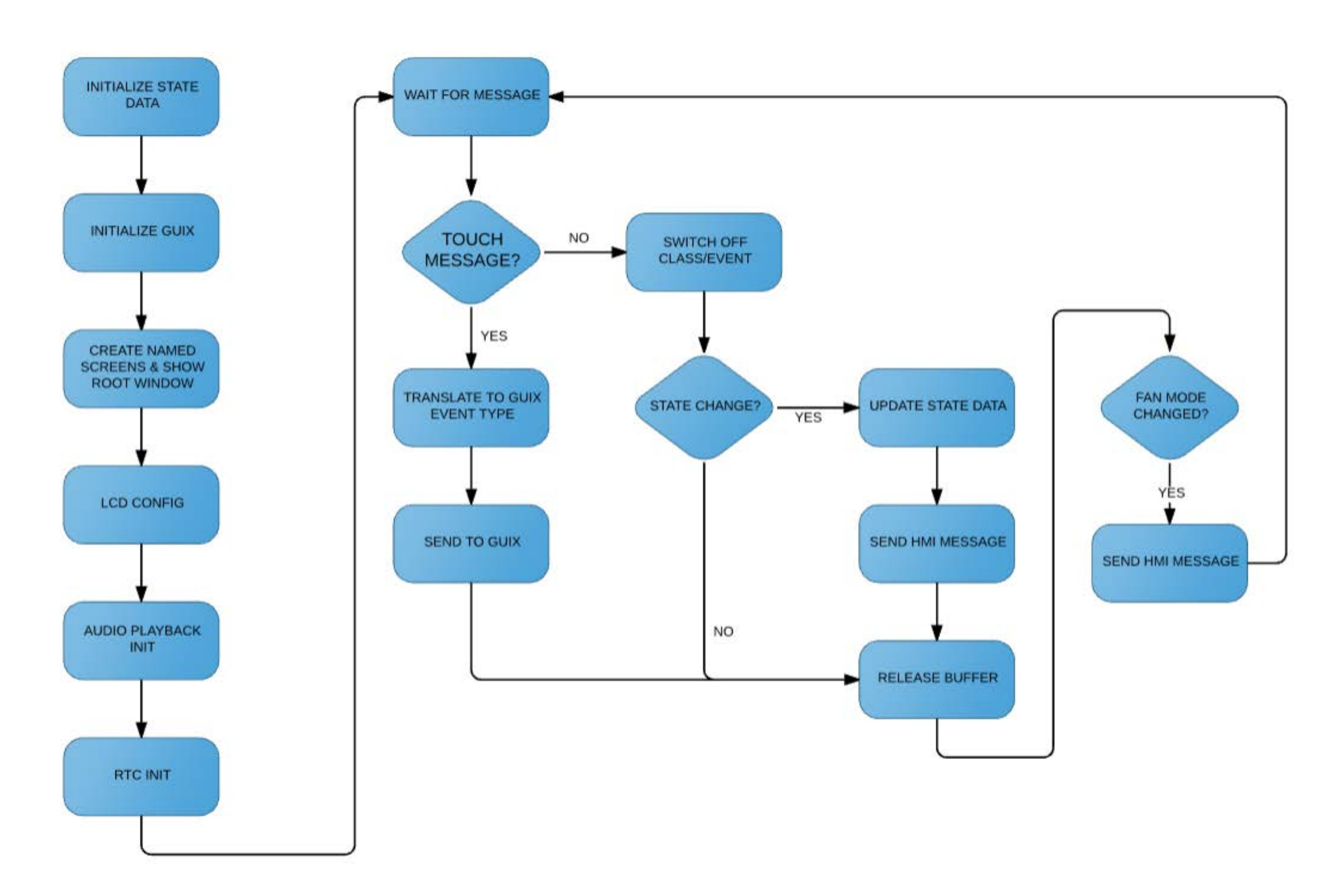


Figure 15. System Thread Flowchart

4.2.2 Temperature Thread

The Temperature thread reads the internal temperature sensor of the S7G2 MCU using the on-board analog to digital converter (ADC). It averages the data, converts the raw ADC value to a temperature reading in Celsius degrees, and sends a message to the system thread anytime the temperature reading changes.

The internal ADC of the S7G2 MCU supports numerous modes including the ability to continuously scan channels and automatically average data read from the ADC before returning the reading. The continuous scan mode is useful for reducing overhead in applications that must read analog channels at a high-speed rate. Because it is generally not necessary to read temperature measurements at a high rate, the single scan mode was selected as the simplest option for reading the internal temperature sensor. The normal thread sleep mechanism of ThreadX is used to determine the rate at which the temperature readings are made. The flowchart in the following figure shows a high-level design of the Temperature thread.

A key issue in multi-threaded applications is thread-safe access to shared data. In the Thermostat application, the system thread owns the state data, however, the GUIX thread needs access to this data when it has to update the screen. It is important to ensure that the data does not change when the GUIX thread accesses this shared memory. To facilitate exclusive access to the state data, a mutex lock is used.

The flowchart in Figure 16 shows how the system thread and the GUIX thread use this mutex to gain exclusive access to shared state data.

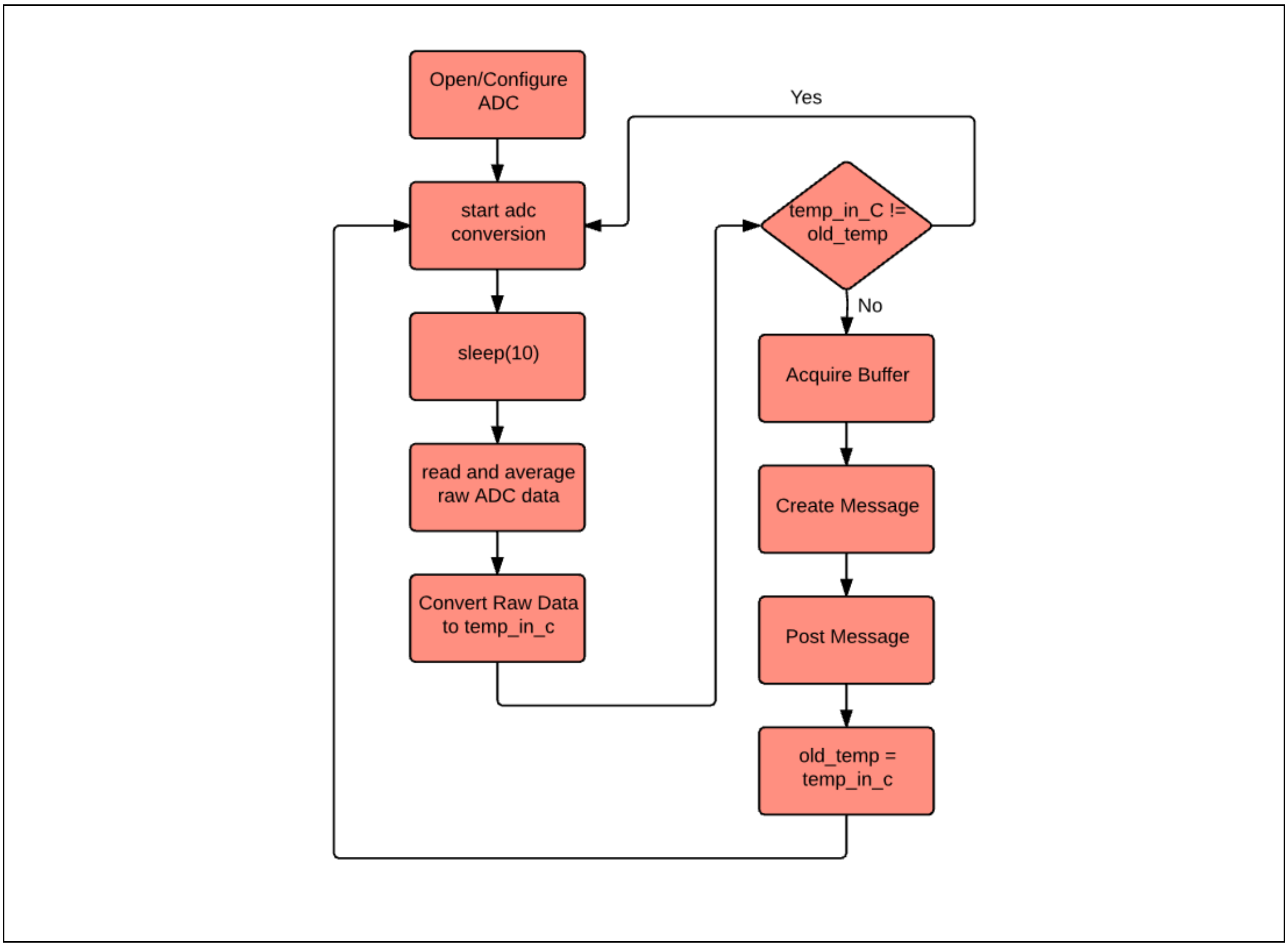


Figure 16. Temperature Thread Flowchart

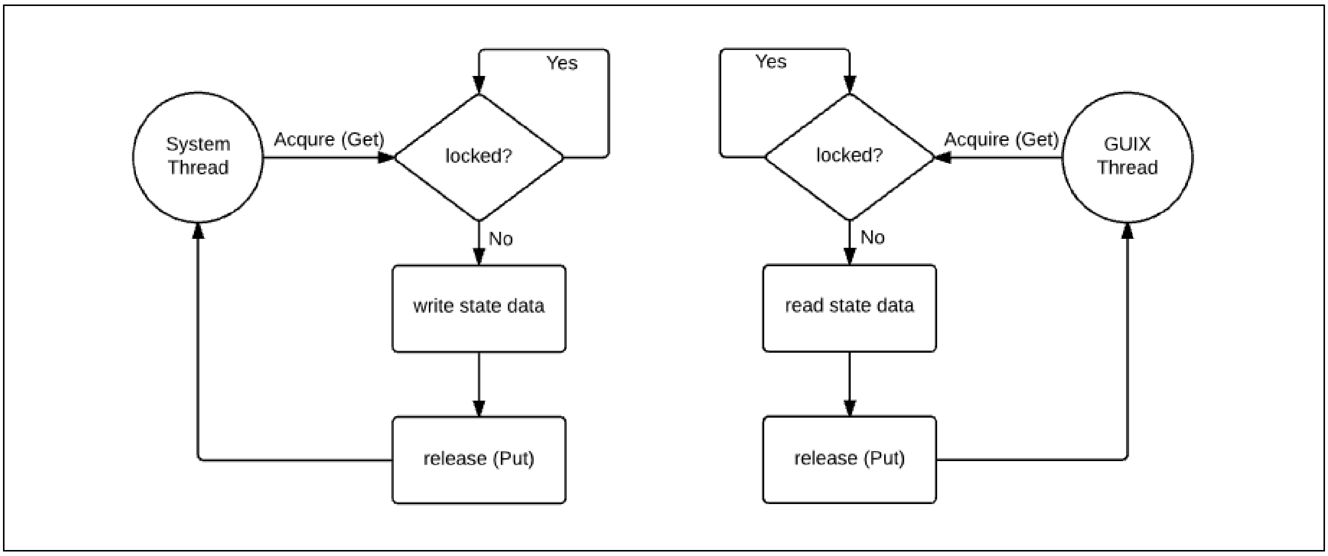


Figure 17. Mutex Lock for System State Data

4.2.3 Thread Layout and the SSP

For those new to the Renesas Synergy Platform, one of the most difficult aspects of learning how to develop complex applications is to learn the various modules defined in the SSP Application Framework, how to add them to your application, and more specifically how to layer these modules on top of each other to form the SSP stacks.

As described in Chapter 2 of the *SSP User's Manual*, Modules are the core building blocks of the SSP. Modules provide functionality upwards and may require downward functionality. The SSP comes with two predefined layers, the Driver layer and the Framework layer. The principle difference between the two is that the Driver layer Modules are peripheral drivers that are RTOS-aware but do not use any RTOS objects or make any RTOS API call. This means the Driver layer Modules can be used in applications with or without an RTOS. Framework layer Modules however, can use RTOS objects such as semaphores, make RTOS API calls, or even create threads as necessary.

Note: Understanding SSP naming conventions early on can help you understand Synergy applications. Driver layer module names always start with an `r_` prefix while framework modules always start with a `sf_` prefix.

The simplest SSP application consists of one module with user application on top.

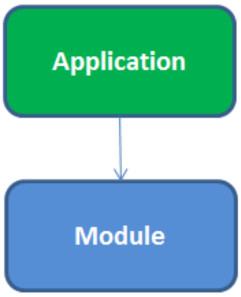


Figure 18. Simple SSP Application

The Temperature thread layout best represents this simple concept and in this case, the Temperature thread requires only the `r_adc` module. Figure 19 shows two ways to represent this. The representation on the right side of the diagram shows the Temperature thread sitting on top of the ADC module. The representation on the right side of the diagram is how you might typically see this concept represented in this document. This can be read as the Temperature thread requiring `g_adc` on `r_adc`.

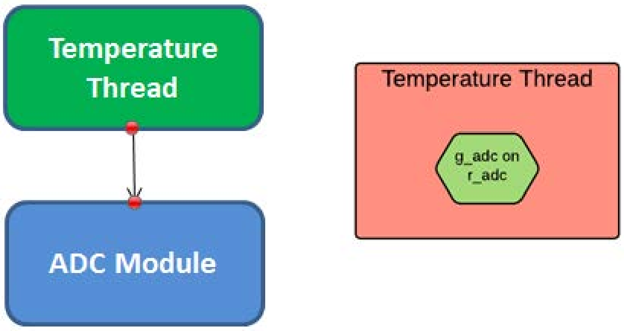


Figure 19. Different Ways to Represent Thread Structure

In SSP, all driver instances have names such as `g_adc` attached to the instance of the driver, in this case the `r_adc` driver. The first thing to recognize is that the `r_` prefix indicates `r_adc` is a driver-level module. The Framework Configuration section shows how to assign these names to a specific instance of a driver.

As shown in Figure 20, the complexity depends on the modules required to accomplish the objectives of the application. The system thread relies on numerous modules, some of which are layered on top of each other to form the SSP stacks. Framework modules are represented with a dark blue color, Driver modules are represented with a light green color, and thread objects such as queues and mutexes are represented in purple.

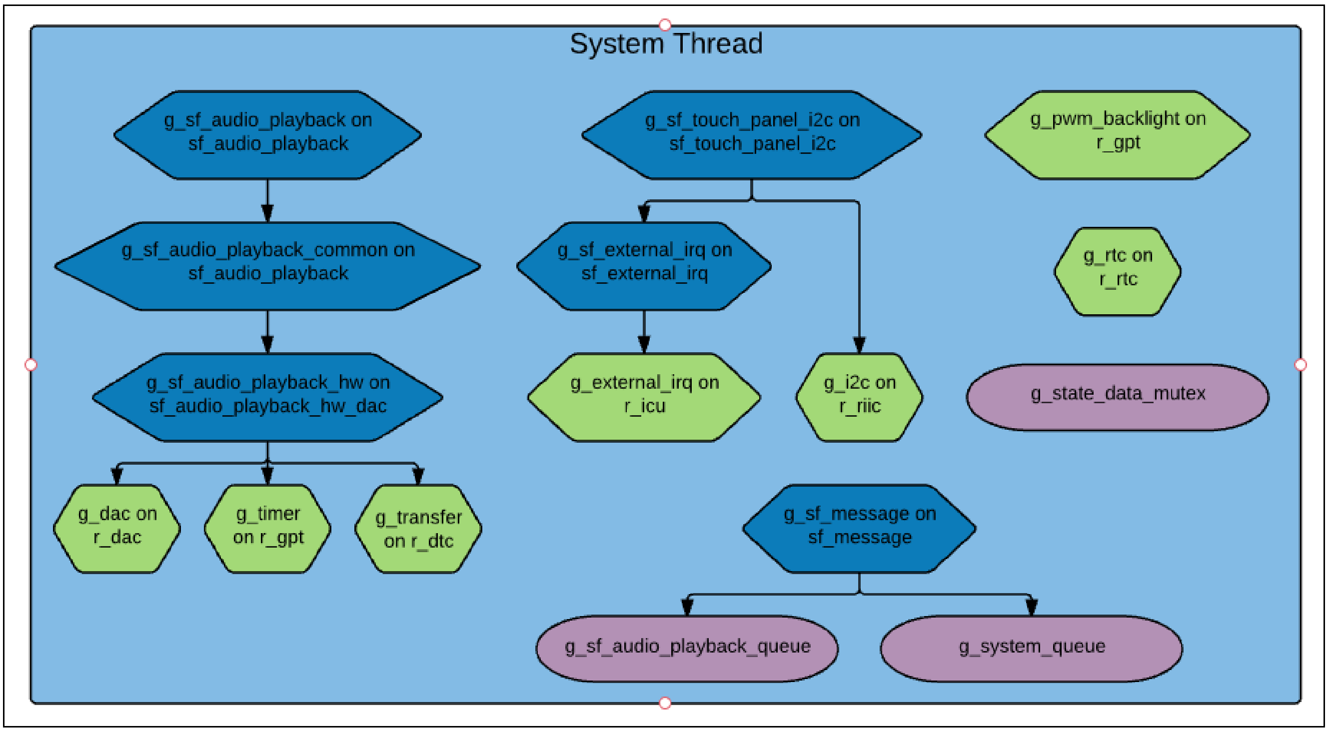


Figure 20. System Thread Module Diagram

The touch controller on most of the development boards generates an IRQ when a touch occurs. The coordinates of the touch are then communicated over the I^{2}C bus. In the center of the diagram, the system thread uses the `sf_touch_panel_i2c` module. For interrupt processing, this module requires the `sf_external_irq` module that in turn requires the `r_icu` module. For I^{2}C communication, the `sf_touch_panel_i2c` module requires the `r_iic` module.

The following figure shows a scenario that highlights some of the nuances in understanding your application architecture with the SSP. It also shows a small shortcoming of the system thread in Figure 20. Even though Figure 21 shows a GUIX thread, you never actually created a GUIX thread in your application. The reason is the `sf_el_gx` module automatically creates this thread when you added the module to your application. The main reason for the GUIX thread box is to have a place holder for the modules you add to your application.

Even though the `sf_audio_playback` and `sf_touch_panel_i2c` modules are added under the system thread, each of these modules creates a separate thread from which they operate. Illustrating these threads would unnecessarily complicate the diagram, yet omitting them does not show a complete picture of what occurs.

The GUIX thread utilizes several modules including the `r_glcd` driver module. The S7G2 MCU includes a Graphics LCD (GLCD) controller that is controlled by the `r_glcd` driver module. This is also perhaps one of the more complex modules to understand. This module allows you to define many properties including the screen resolution, where the frame buffer resides for example, the internal and external memory, and the assignment of video synchronization signals. It is recommended that you have a complete understanding of the `r_glcd` driver module if you want to design embedded systems with graphical displays.

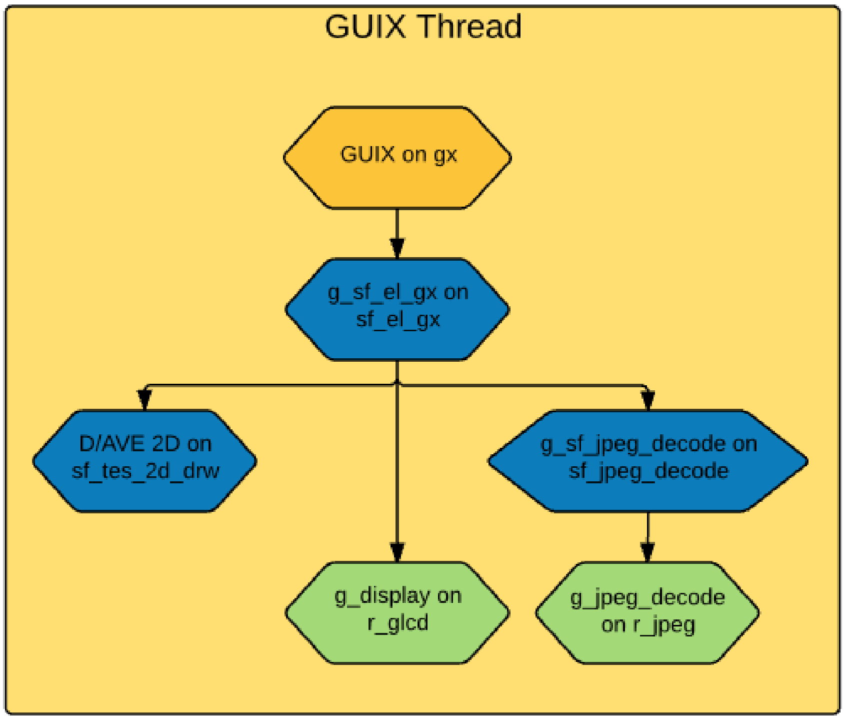


Figure 21. The GUIX Thread Modules

5. Framework Configuration

One of the first things you must do when writing a Synergy application is to configure the framework. To properly configure the framework, you must have detailed knowledge of both the software design that you are implementing along with the specific hardware it is running on. For the hardware, this includes the types of peripherals to be used on the hardware, the pins they are mapped to, and whether they are internal or external to the processor. From the software perspective, you must decide how many threads to be used, what threads need access to what hardware components, and what additional software objects such as semaphores and queues, each thread requires. With this information, you can successfully configure the framework for your specific application needs.

In the Thermostat application, the framework configuration is stored in a file named `configuration.xml` in the `synergy_cfg` folder. Double-click on this file to bring up the Synergy Configuration window for the project. This window is displayed as one of the tabs in e^2 studio. It may take a few seconds for e^2 studio to process the `.xml` file.

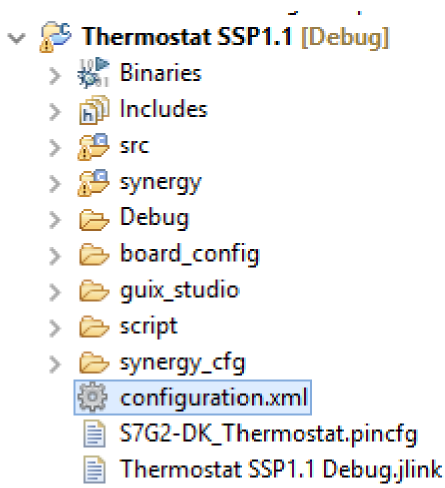


Figure 22. Configuration.xml file Storing the Framework Configuration

When building a project from scratch, this configuration tab is where you perform the initial configuration of the SSP Application Framework. In the following figure, selecting the **Summary** tab at the bottom of the Synergy Configuration **DKS7_Thermostat** pane generates a Summary screen highlighting the items you might configure along with a scrolling window that lists all the software components currently selected for this project. The remaining tabs allow you to tailor the framework to your specific application needs.

This application note only highlights a few details of the application framework configuration as it pertains to the Thermostat application. For more information, refer to the appropriate Synergy Application Framework documentation and application notes.

After you have properly configured the project, click **Generate Project Content** (the green arrow button above the summary screen) to build the auto-generated files required to implement the components you defined.

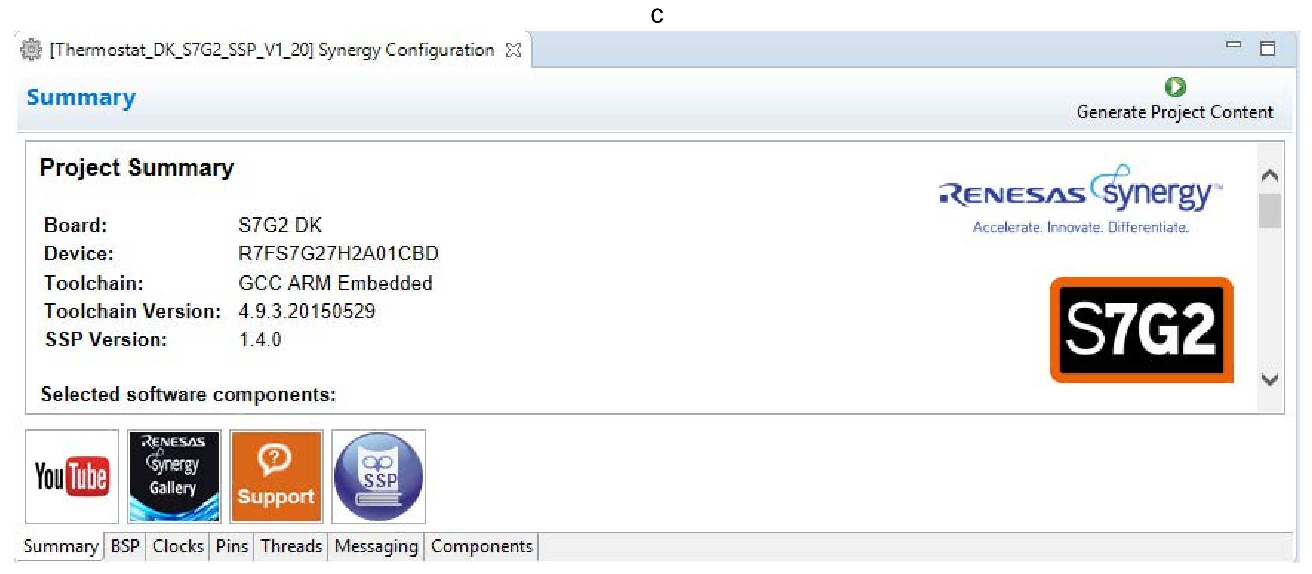


Figure 23. Synergy Configuration Project Summary Screen

5.1 Components Tab

Even though the Components tab is the last tab shown, it is one of the first items you should configure. Selecting components first makes them available in subsequent operations such as the mapping of hardware resource to specific threads in the **Threads** tab. One of the advantages of the Synergy Framework is that it only compiles in the components you choose, therefore reducing the size of your overall application. As shown in Figure 24, the components tab is broken down into seven categories.

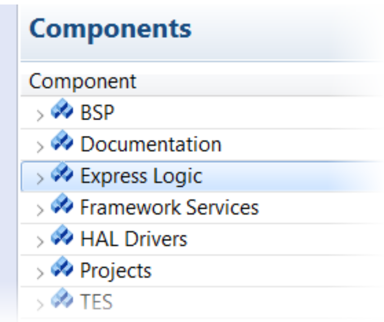


Figure 24. Components Tab

You can expand any of the categories by clicking on the arrow to the left of the category name. The following table highlights the selections used for the Thermostat application. One of the nice features of the Components tab is that it gives you a description of the component and also shows dependencies for each component. As an example, notice that `sf_message`, the Messaging Framework component, requires ThreadX. This dependency listing helps eliminate compile time errors that might result from failing to choose the proper dependent components when making your component selections.

Table 2. Components Selected for the Thermostat Application

Category	Component	Version	Description
BSP	s7g2_dk	1.5.0	Board Support Package for S7G2_DK
Express Logic	gx	1.5.0	Express Logic GUIX: Provides=[GUIX], Requires=[ThreadX]
	tx	1.5.0	Express Logic ThreadX: Provides=[ThreadX]
Framework Services	sf_el_gx	1.5.0	SF_EL_GX GUIX Adaption Framework: Provides=[SSP GUIX Adaptation Framework] , Requires=[ThreadX, GUIX]
	sf_external_irq	1.5.0	Framework External IRQ: Provides=[Framework External IRQ] , Requires=[External IRQ , ThreadX]
	sf_jpeg_decode	1.5.0	Framework JPEG Decode: Provides=[SF JPEG Decode] , Requires=[ThreadX ,JPEG Decode]
	sf_message	1.5.0	Messaging Framework: Provides=[Message] , Requires=[ThreadX]
	sf_tes_2d_drw	1.5.0	TES Dave/2d(DRW) Framework: Provides=[SF_TES_2D_DRW] , Requires=[ThreadX ,TES Dave/2d]
	sf_touch_panel_i2c	1.5.0	Framework Touch Panel using I2C: Provides=[Framework Touch Panel] , Requires=[ThreadX ,Message ,I2C , Framework External IRQ]
HAL Drivers	r_adc	1.5.0	A/D Converter: Provides=[ADC]
	r_cgc	1.5.0	Clock Generation Circuit: Provides=[CGC]
	r_elc	1.5.0	Event Link Controller: Provides=[ELC]
	r_glcd	1.5.0	Graphics LCD: Provides=[Display]
	r_gpt	1.5.0	General Purpose Timer: Provides=[Timer ,GPT]
	r_icu	1.5.0	External IRQ: Provides=[External IRQ]
	r_ioport	1.5.0	I/O Port: Provides=[IO Port]
	r_jpeg_decode	1.5.0	JPEG Decode: Provides=[Key Matrix]
	r_rtc	1.5.0	Real Time Clock: Provides=[RTC]
	r_sci_common	1.5.0	SCI Common: Provides=[SCI]
	r_sci_i2c	1.5.0	SCI I2C: Provides=[I2C Master] , Requires=[SCI Common]
TES	dave2d	1.5.0	TES Dave/2d: Provides=[Dave/2d]

5.2 Threads Tab

The **Threads** tab is where you can add, delete, or review existing application threads. You define a new thread by clicking the **New Thread** button and then entering a unique name for your new thread. After you added a new thread, define the modules for the thread to use in the **System Thread Stacks** pane. You may also add thread objects such as queues, mutexes, semaphores, and event flags using the **System Thread Objects** pane.

As an example, if you click the **Threads** tab and then single click the **System Thread** in the Threads pane, you should see something like the screen shown in the following figure. In this figure, the System Thread has two system thread objects and several system thread stacks. The SSP v1.5.0 thread stacks are conveniently shown in graphical format, this makes it easy to see the relationship that exists between the driver and framework modules.

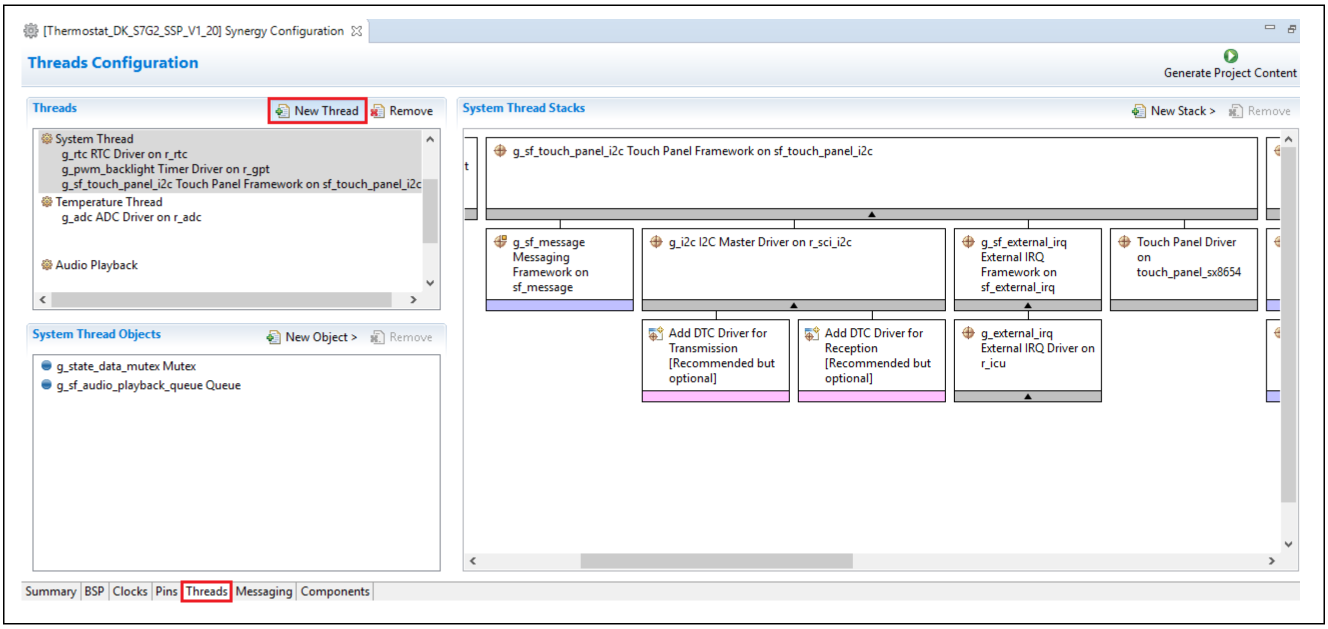


Figure 25. Threads Tab

The interface remains consistent across all three panes. You can add or delete additional threads using **Remove** at the top right-corner of the Threads window. You can also add or delete additional modules to any thread by clicking the same button above the **System Thread Stacks** pane.

If you have chosen the appropriate components before adding modules to your threads, you should not receive any errors. As an example, **Figure 26** shows how to add the ADC driver r_adc to the Temperature thread. In this figure, the driver is added by selecting **Driver** > **Analog** > **ADC driver on r_adc**.

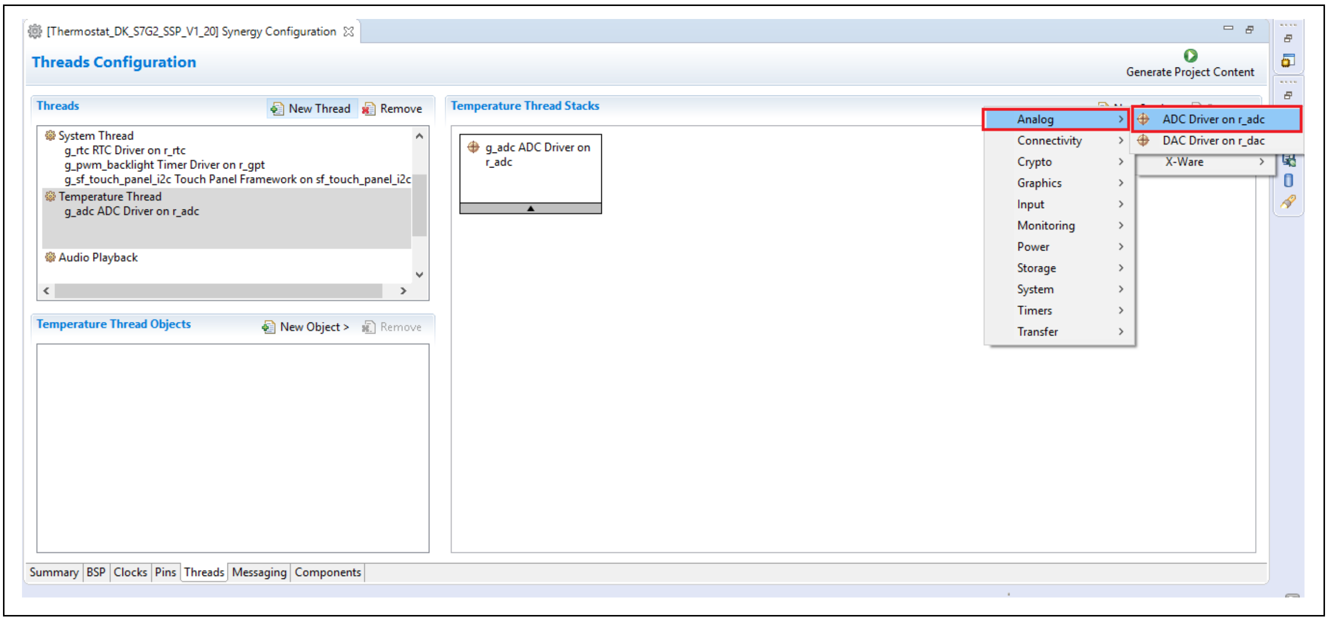


Figure 26. Adding an ADC driver to the Temperature Thread

Note: Drivers that are prefaced with r_ are driver level modules.

If you pick a module but you have not selected the appropriate component first, the Application Framework automatically selects the component for you. If the Application Framework detects errors with the module addition, it prefaces the module with an error. You might examine the errors by hovering over the module name.

5.2.1 Thread Objects

ThreadX supports all typical objects in a RTOS. These include semaphores, mutexes, event flags, and message queues. When you click the **System Thread** in the **Threads** pane, there is one Queue object and is Mutex object allocated for this thread.

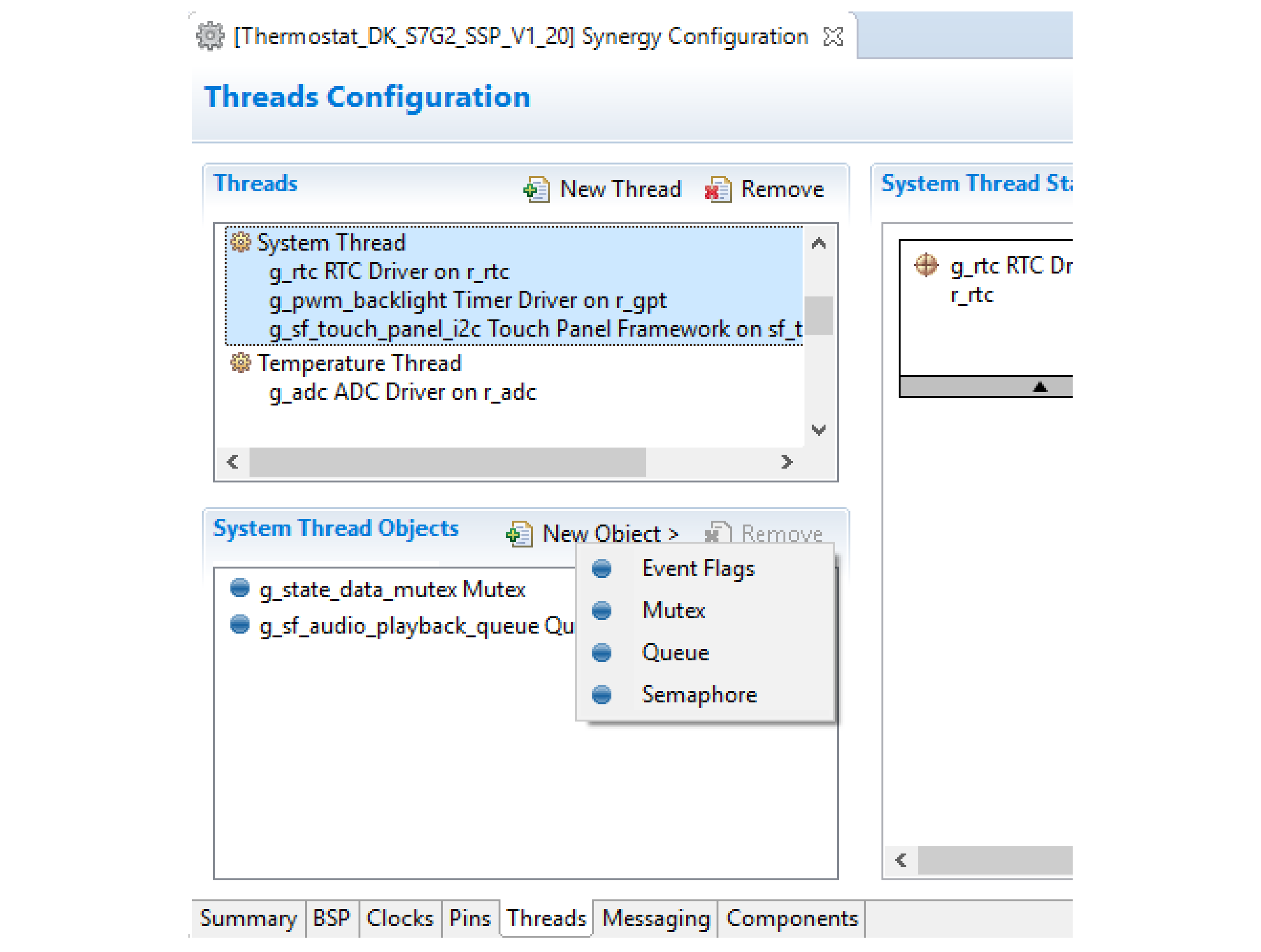


Figure 27. Adding Thread Objects to a Thread

You can allocate additional thread objects in the same way you allocate additional threads. Clicking the **New Object** displays a drop-down list of the standard thread objects supported by ThreadX. Click the object you want to add and the object is added to the System Thread Objects Window.

When adding or reviewing threads, thread modules, or thread objects, in general you want the **Properties** tab enabled so you can examine or change the properties associated with the item. If your **Properties** tab is not showing, you can display it by going to **Window** > **Show View** > **Other**… > **General** and then selecting **Properties**. The following figure shows an example of the Audio Playback Queue (`g_sf_audio_playback_queue`) that has a message size of one word and a queue size of 64 bytes. To change these values, simply update them in the **Properties** view and then click the **Generate Project Content** button to update your project code with the new value.

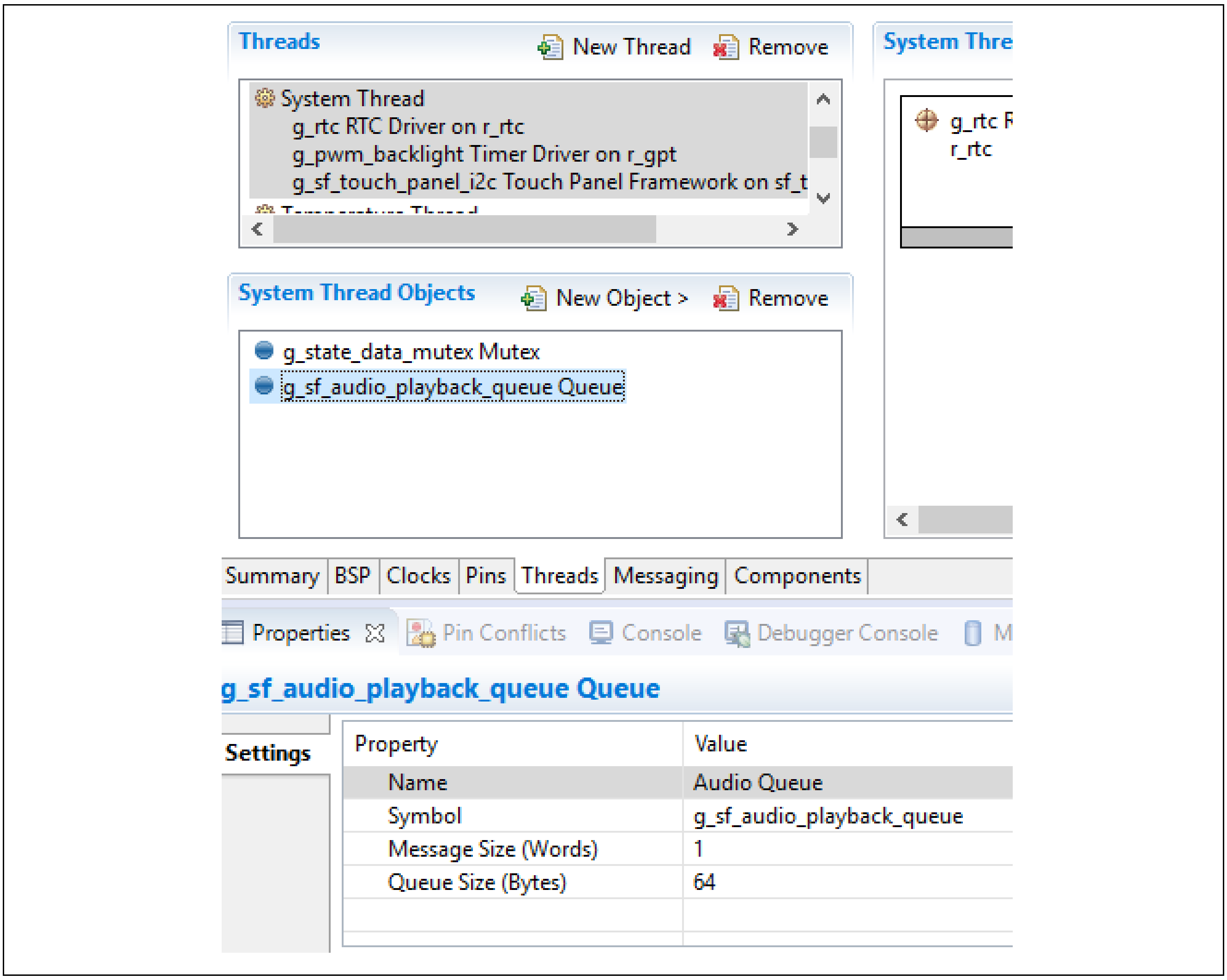


Figure 28. Object Properties Displayed in the Properties Tab

5.3 Module Configuration

Most driver or framework modules added to your project have properties associated with them. The properties are dependent on the drivers you add. The **Properties** tab is used to configure them. This application note does not attempt to cover all the properties configured for every driver. Only a representative few are covered, but the rest are configured in the same way. Since the basic element of the Thermostat application is the HMI, the `r_glcd` driver module is added. This module is used to configure the Graphics LCD (GLCD) peripheral of the Cortex-M4 core. While the properties of each development board might differ slightly, the process of configuring these properties is generally the same on all the development boards.

5.3.1 GLCD Configuration

In Figure 29, selecting the `g_display` display driver on the `g_glcd` module from the System Thread Stacks displays the associated GLCD properties under the **Properties** tab. The first thing to notice is a list of properties with two groupings, ICU and Module. The ICU group is where you configure IRQ priorities.

The Module group is where you configure the GLCD controller itself. These properties can be a bit daunting at first. There are a few broad categories inside the Module grouping:

- **Name** – The name given to this instance of the module `g_display` by default and the name of a user-defined callback function if used. The Thermostat application does not use a callback.
- **Input** – This block of module properties defines the input to the GLCD, most notably the size of the frame buffer, and the source of the dot clock, where the frame buffer is located. This section allows you to

define two graphics screens. The Thermostat application only uses one screen, so the Input-Graphics Screen 1 is set to Used.

- **Output** – This is the area where you define the output properties of the GLCD. This includes properties such as the total Horizontal and Video Cycles, the active video cycles both horizontal/vertical and front/back porch duration.
- **TCON** – Use these lines in conjunction with the **Pins** tab, to map the Horizontal Sync (Hsync), Vertical Sync (Vsync), and Data Enable signals. You can specify the LCD panel clock divisor, which divides the clock input to the GCLD. This divisor ratio currently ranges from 1/1 to 1/32.
- **Color Correction** – This is where you can add various levels of color correction such as brightness, contrast, and gamma to your display. Color correction of LCD screens is outside the scope of this application note but this is the area where you can make this type of adjustment.

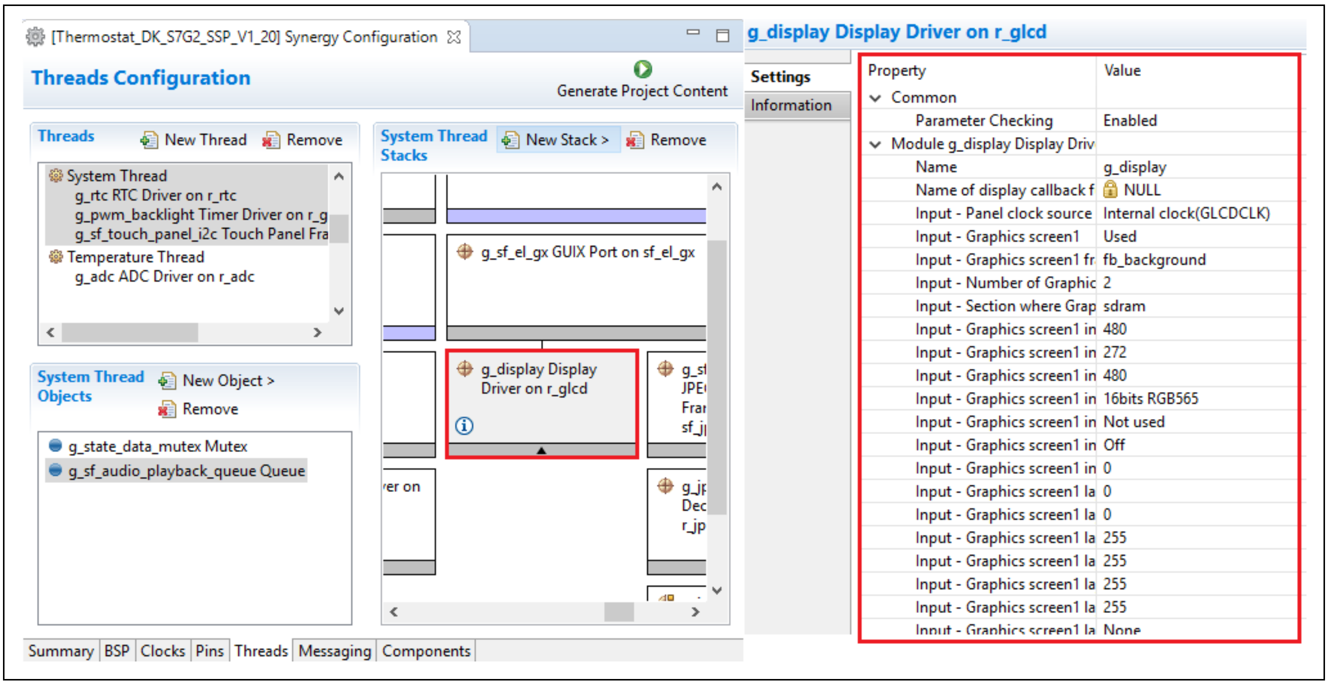


Figure 29. GLCD Properties

5.3.2 TCON Configuration

If you scroll down a little farther in the **Properties** tab, there are four TCON properties. One of these is associated with the panel clock division ratio. This allows additional division of the dot clock, which is driven directly from the PLLOUT branch of the clock tree. The other three TCON properties are associated with the LCD sync signals. These three signals can be confusing to new users. The following figure shows how these signals are mapped to the physical pins they are connected to.

TCON - Hsync pin select	LCD_TCON1
TCON - Vsync pin select	LCD_TCON2
TCON - DataEnable pin select	LCD_TCON0
TCON - Panel clock division ratio	1/32

Figure 30. GCLD TCON properties

To provide some flexibility, the GLCD controller of the S7G2 MCU provides two pin grouping options. Each option uses different pins on the MCU to drive the data lines connected to the LCD display. It is up to the hardware designer to pick the group of pins to be used. Picking one or the other may free up some MCU pins that are required in some other part of the hardware design.

If you look at the schematics for the DK-S7G2 board, you can see all the pins connected to the LCD data lines. You can see the four pins connected to the sync signals which are highlighted in red. The data lines chosen by the hardware designer must match one of the two pin groupings available in the GLCD module. Extra flexibility is provided for the LCD sync signals.

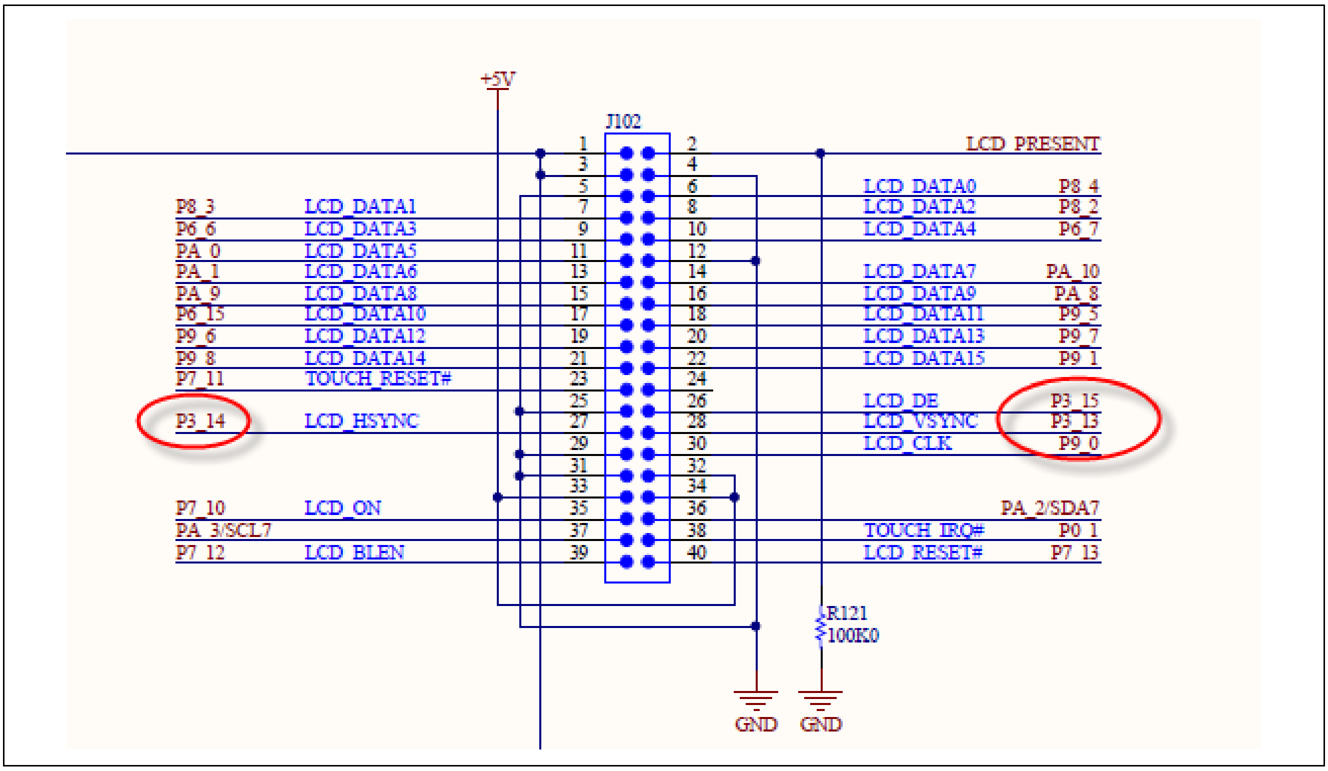


Figure 31. DK_S7G2 LCD Electrical Interface

The easiest way to understand this is to go to the **Pins** tab in the Synergy Configuration window. The selections for **Ports**, **Peripherals**, **Analog Pins**, and **Other Pins** are shown in the following figure. If you expand the **Peripherals** dialog, all the various Arm® core peripherals that can be configured are displayed in this screen.

If you scroll down to the `LCD_GRAPHICS` entry and click the small **+** sign next to it, the `GLCD_Controller_Pin_Option_A` and `GLCD_Controller_Pin_Option_B` options are displayed. There should be a green check mark next to `GLCD_Controller_Pin_Option_B` indicating this is the pin group associated with driving the LCD display.

Note that `TCON0` is associated with Port 3 Pin 15 (P315). If this designation is on the schematic, `P3_15` is connected to `LCD_DE` which is the data enable pin for this screen. Referring to Figure 30, `TCON0` is selected to drive the `DataEnable` signal.

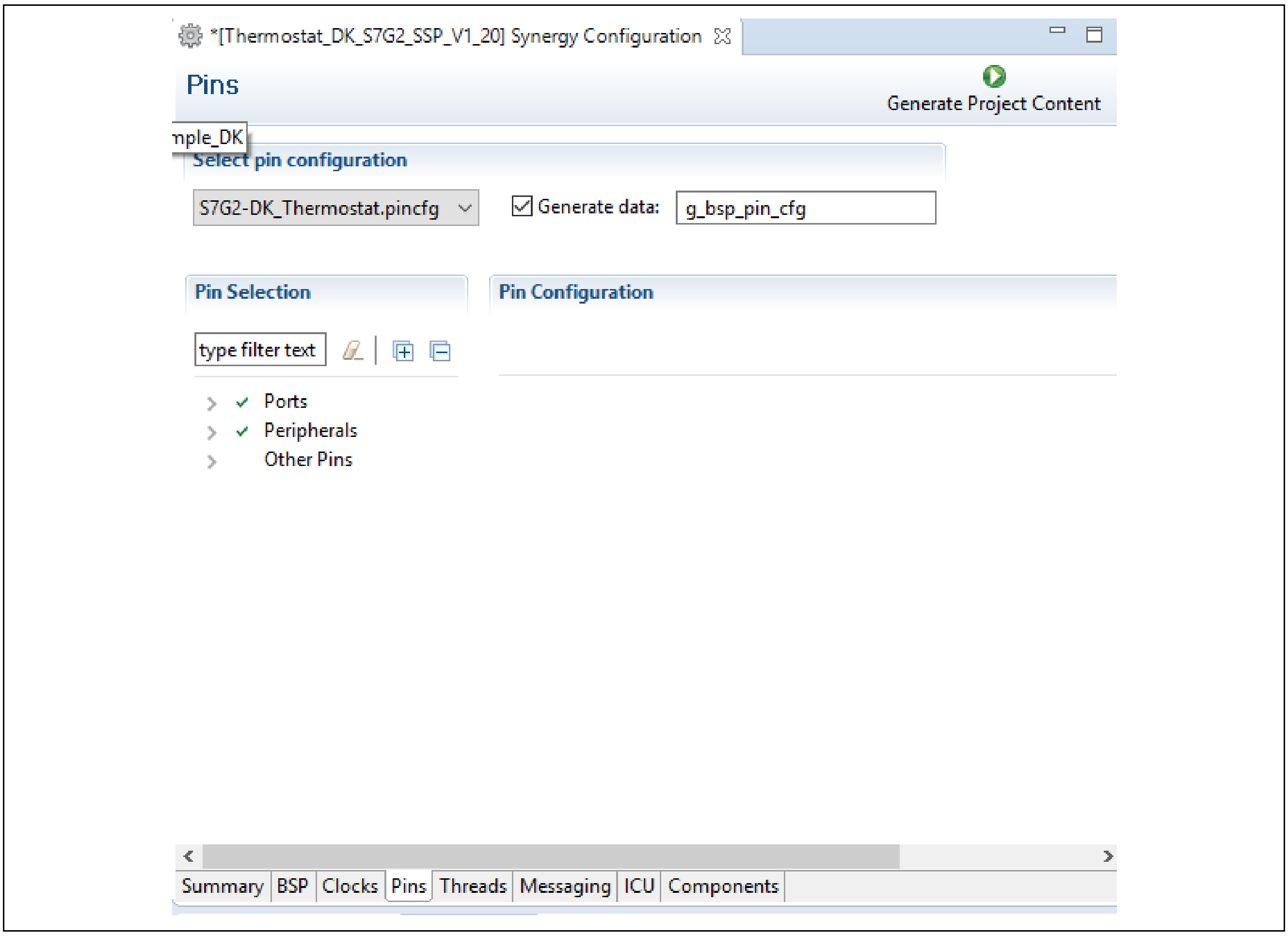


Figure 32. Pins Tab

If you look at all the LCD data lines such as LCD_DATA_DATA00, the pins they are connected to should match the pins they are connected to on the schematic. Clicking on the arrow to the right of the pin takes you directly to the associated **Pin Configuration** dialog, just as if you had selected the Ports Group and specific port and pin.

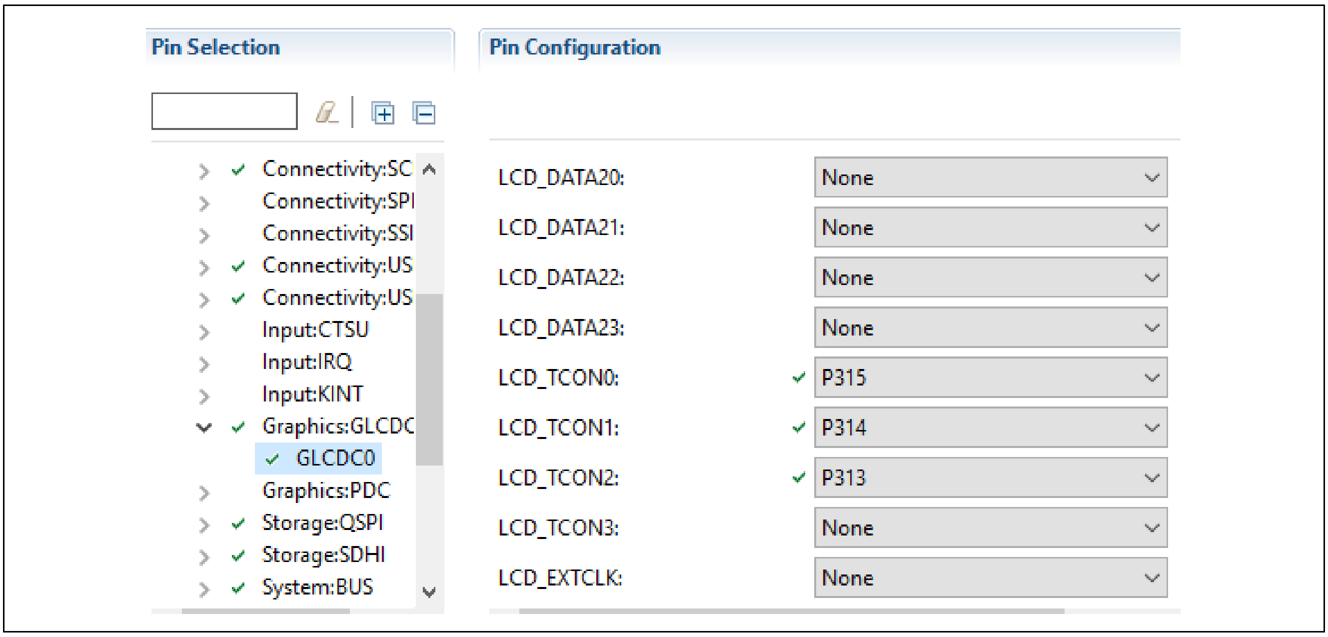


Figure 33. Configuring GLCD Peripheral Pins

For example, clicking on the arrow next to the LCD_TCON0_B pin displays the **Pin Selection** and **Pin Configuration** screens as shown in Figure 34. Notice that the pin is appropriately set to the peripheral mode. As of this writing, the pins default to no Pull Up, Low Drive Capacity, and CMOS output type. Clicking on the arrow button to the right of this screen takes you back to the associated peripheral screen.

Note: At the time of writing this application note, when you select option A or B of the LCD_GRAPHICS peripheral, you must manually enable each pin connected to your display. Using the Arrow button makes it easier to toggle between the **Peripheral** screen and the **Pin Configuration** screen.

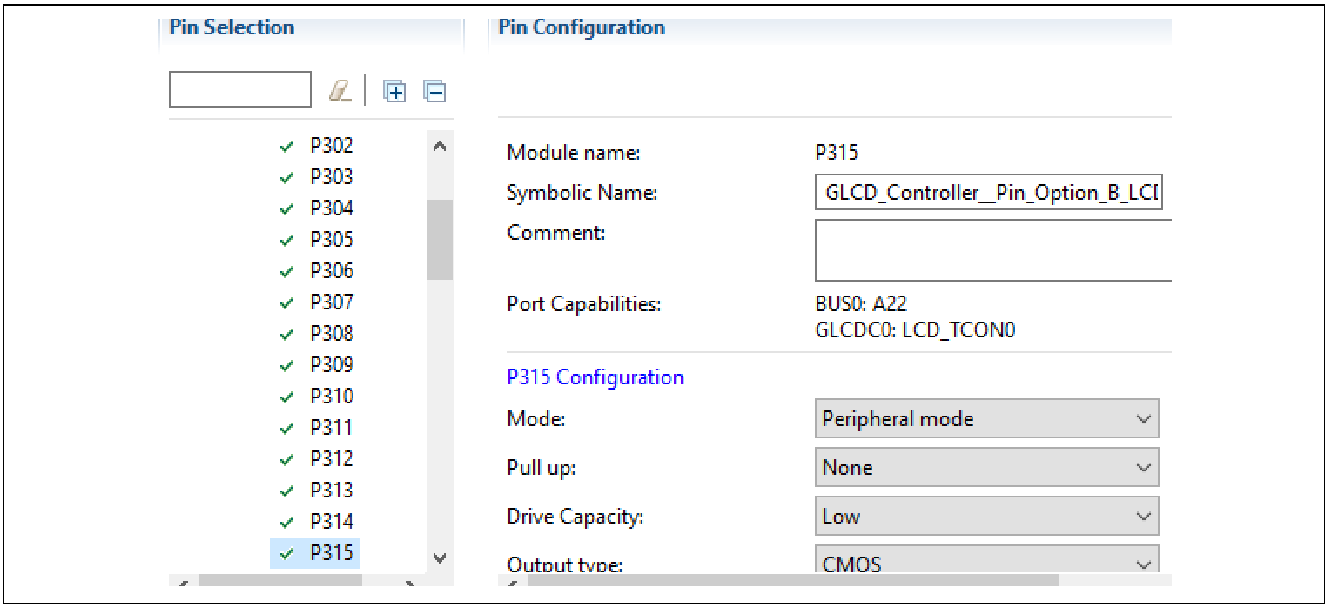


Figure 34. Pin Selection and Configuration Screen

5.3.3 Using External Memory for Frame Buffer

One of the differences between a lower cost development board such as the SK-S7G2 and the more expensive PE-HMI board is the availability of an external memory area for the screen buffer. As the screen size and color depth increases or a more sophisticated display strategy, such as ping pong frame buffering, is used, the available internal memory of the microcontroller may not be sufficient. In this case, an external memory device is usually added to the board.

The S7G2 and S5D9 Synergy MCU Groups support the use of external SDRAM. Figure 35 shows an example of the S7G2 memory map. In this example, the SDRAM address space is associated with address 9000 0000h to 9800 0000h.

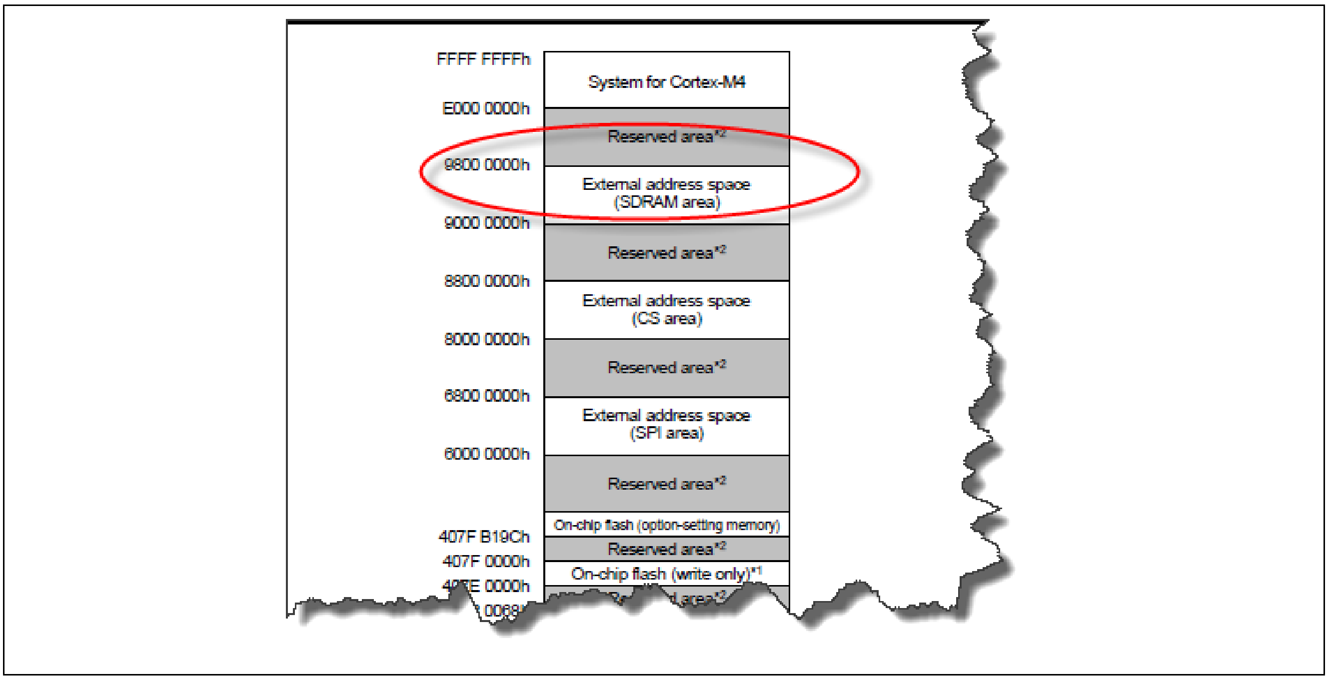


Figure 35. Example S7G2 Memory Map

To use external SDRAM for your frame buffer, you must first locate the following property under the **Properties** tab for the GLCD controller and change it from `bss` to `sdram`. By convention, the `bss` abbreviation instructs the SSP to place the frame buffer in internal memory in a section named `bss`.

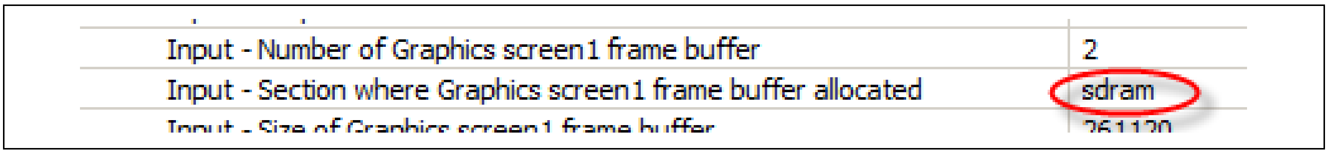


Figure 36. SDRAM Selection in GLCD Properties Tab

You must configure the external memory interface. This is like configuring the GLCD controller. Return to the **Pins** tab of the **Synergy Configuration** window, expand the **Peripherals** selection, then expand the **BUS** selection. Change the **Operation** mode to **Enabled**, then manually enable each line used by your SDRAM by toggling between the **Peripheral** view and the **Pin Configuration** view using the arrow to the right of the pin name.

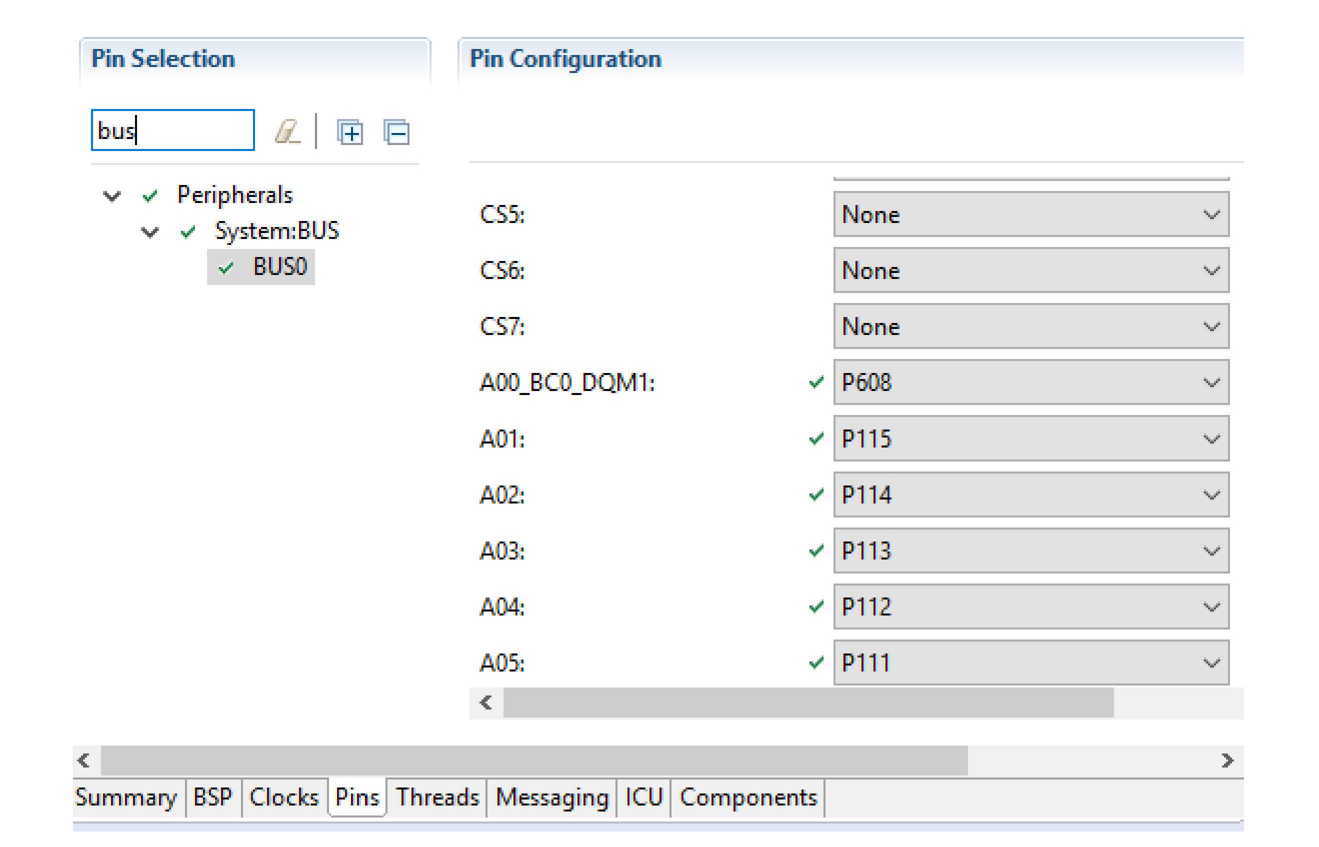


Figure 37. Pin Configuration for the External Memory Interface

After you have correctly configured the external memory interface and changed the GLCD property to point to SDRAM, you may not see any differences on your display. This may be because your current screen resolution fits fine inside the internal memory. When you change to external memory, your GLCD continues to drive the screen as it did before, only now it is pulling the frame buffer from your external SDRAM.

5.3.4 SK-S7G2 LCD

There are several differences in the LCD configuration and control when running the Thermostat application on the SK-S7G2 MCU board. They are:

- Some LCD initializations are required on the SPI interface before running in RGB mode
- No SDRAM to hold GUIX external frame buffers
- No backlight control

This on-board ILI9341 graphics controller of the LCD screen has several operating modes. Placing the controller in the proper mode so it can be controlled by the GLCD controller of the S7G2 MCU, requires some SPI initialization commands to be sent to the ILI9341. This code resides in separate `.c` and `.h` files, which are copied to the project when you run the configuration for the `SK-S7G2.bat` file in the `board_config` directory.

As is the case with most small code changes, the `system_thread_entry.c` file contains the following lines that call the code that initializes the on-board ILI9341 graphics controller if the code executes on the SK-S7G2 Synergy MCU Group development board.

```
#if defined(BSP_BOARD_S7G2_DK)
    InitILI9341V ();
#endif
```

If you were to examine the **Pins** tab after configuring the Thermostat application to run on the SK-S7G2 MCU, you would notice that the external memory controller is not enabled for this board, and there is no PWM signal setup for LCD backlight control.

5.3.5 e[2] studio Tricks

The e[2] studio ISDE has a handy feature you can use to ensure that the images you see on your LCD screen are coming from your external SDRAM. To use this feature, make sure to connect e[2] studio to your board, and run the program in the debugger. Ensure your **Memory** tab is open in the **Console** window, normally located to the bottom of the screen and in **Debug** view. Click the small green plus **(+)** sign to add a memory monitor. The **Monitor Memory** dialog box is displayed as shown in Figure 38. Enter the external address space associated with the SDRAM area in hex format (0x90000000) and press the **OK** button.

A new tab appears in the **Memory** tab that displays the content of the memory area you specified for the memory monitor.

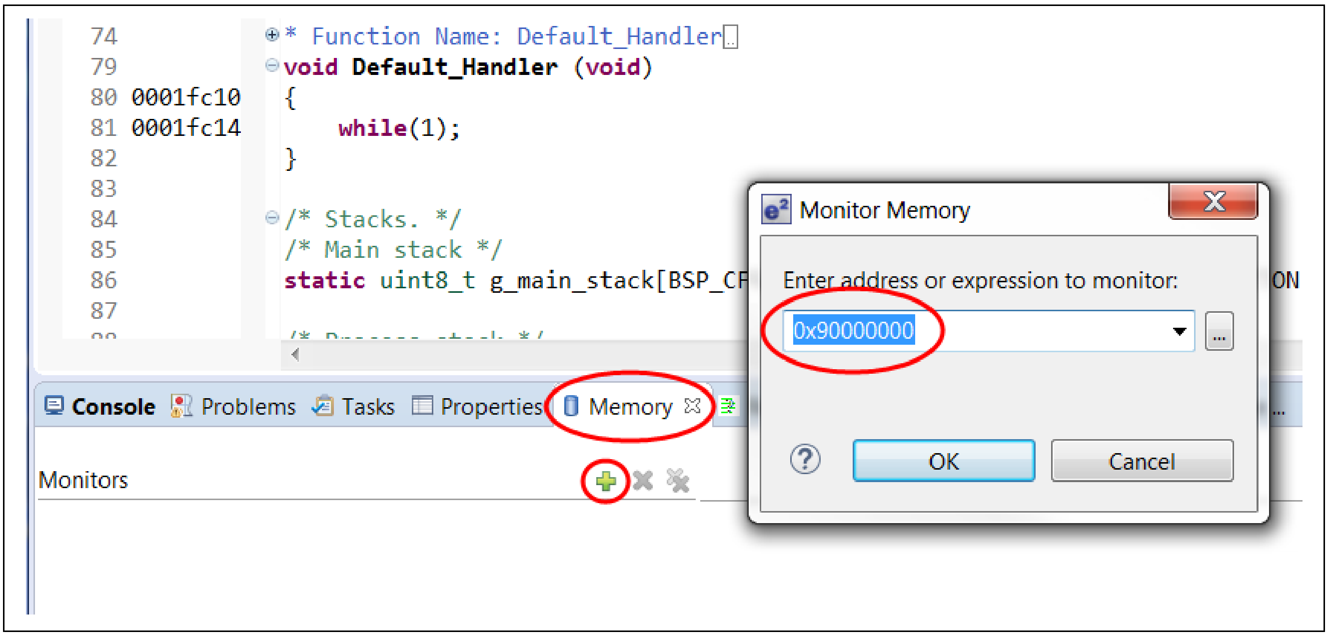


Figure 38. Setting up a Memory Monitor

The content of the SDRAM memory area is displayed in the memory monitor you just created.

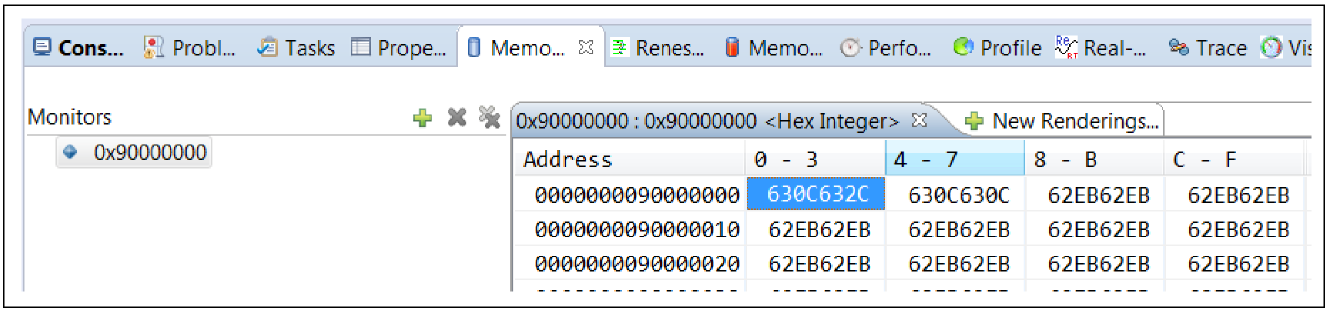


Figure 39. SDRAM Contents

Select the **New Renderings** tab next to the memory monitor you just created, then select **Raw Image** from the list of options and click the **Add Renderings** button.

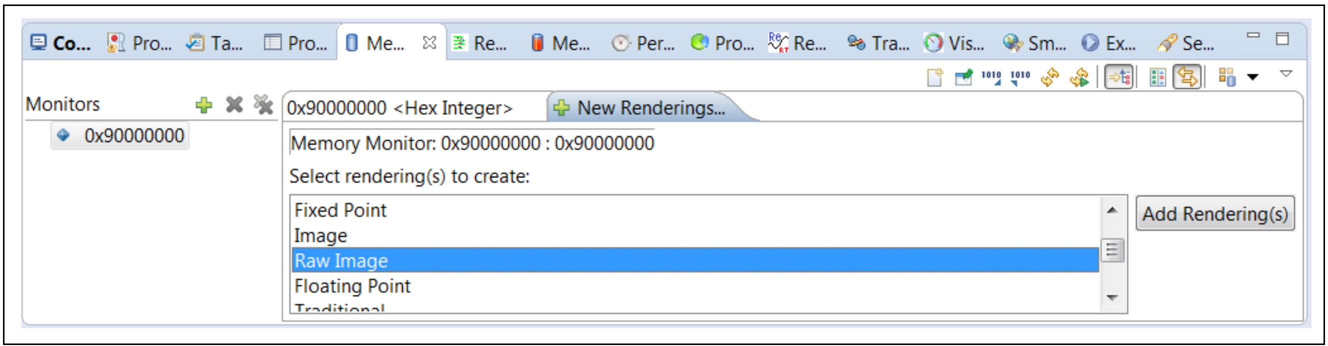


Figure 40. GUIX Rendering Format Selection

When the **Raw Image Format** dialog box displays, enter the screen resolution width and height, along with the encoding, which is 16 bpp (5:6:5) in our case.

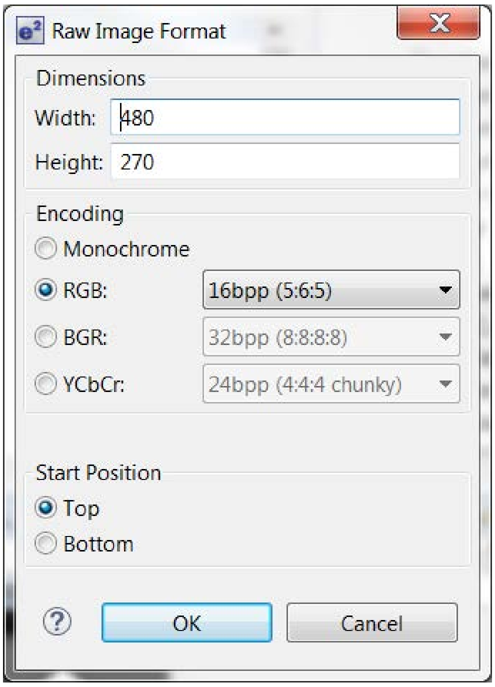


Figure 41. Selecting the Image Format

After pressing **OK**, the memory monitor displays the image based on the parameters you entered. At this point you can even switch back to the **Memory Monitor** tab, modify memory locations, and see the image changes on both the memory monitor and the LCD screen. You can perform these same steps when running out of internal memory but you must first reference the linker map to determine where in the `.bss` section the screen memory was mapped. Open a memory monitor on that location and repeat the specified steps.

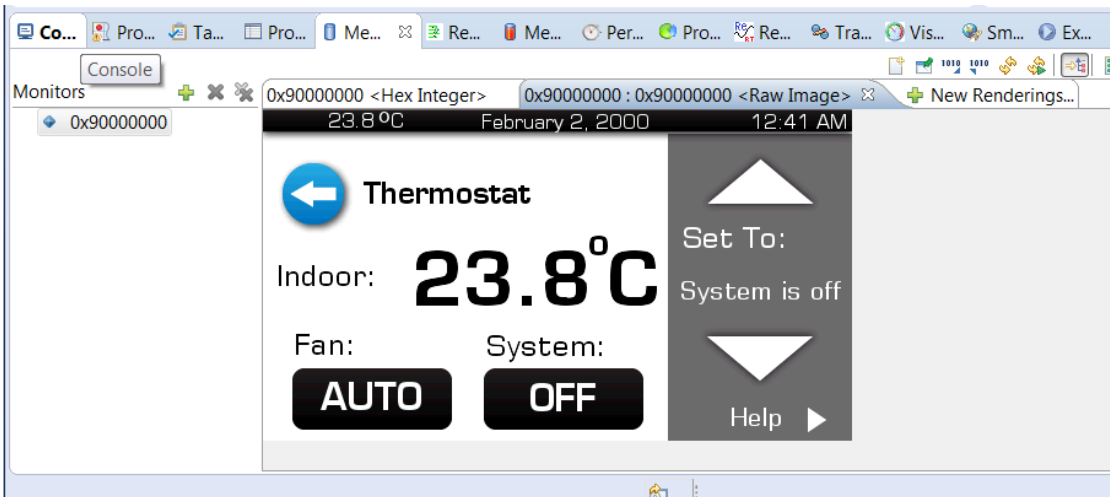


Figure 42. Memory Displayed as an Image

5.4 Messaging Tab

Many modern RTOS environments such as ThreadX, provide a facility for passing messages between threads. The SSP provides a robust messaging framework and the Synergy **Configuration** window provides a point-and-click interface for defining these messages.

The **Messaging** tab of the Synergy Configuration window allows you to define messages (events) that are appropriate to your system. You can double-click on the `configuration.xml` file in the e^2 studio **Project**

explorer to bring up the Synergy Configuration window. Click the **Messaging** tab to display the Messaging window that looks like in the following figure. The **Messaging** pane is divided into three sections:

- Event Classes
- Events
- Touch Subscribers

The **Touch Subscribers** section is prefaced by the currently selected event class.

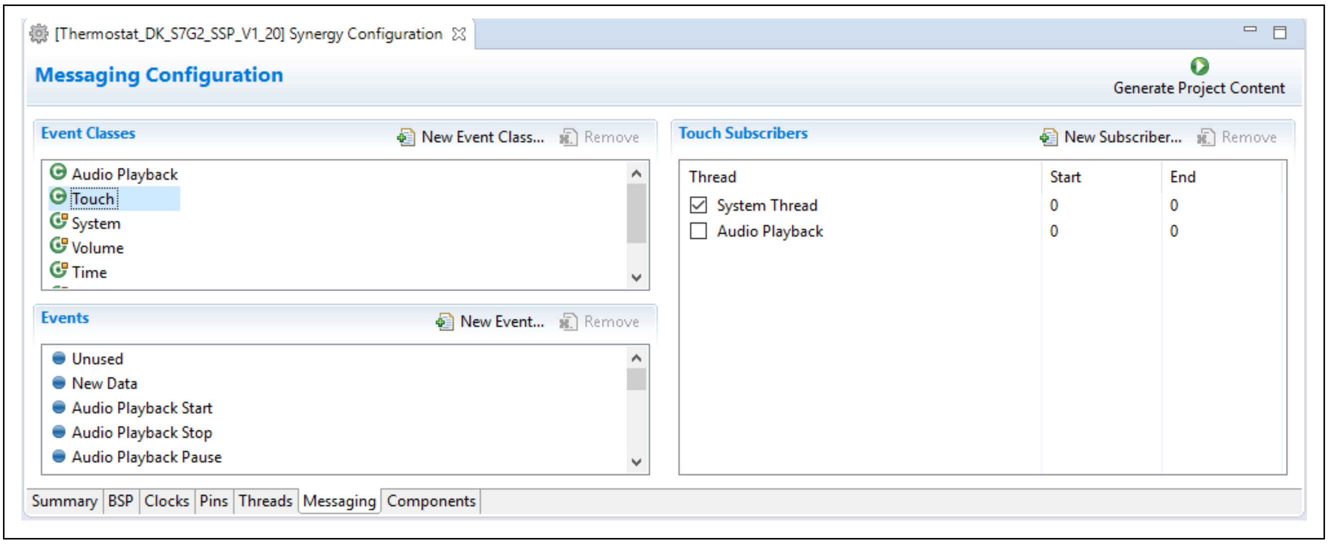


Figure 43. Messaging Tab

5.4.1 Messaging

Messages are used in the system to avoid polling. In event driven systems, respond to asynchronous events in a timely fashion without wasting precious processor cycles to poll for every event that might occur. Messaging systems provide the ability for a thread to suspend execution while waiting for an event. This frees up the processor and simplifies the code. The term message and event are often used synonymously.

How messaging is implemented in the Thermostat application is described later in this document. Detailed description of messaging theory and the SSP messaging framework is outside the scope of this document, however key issues that are often confusing to new users as they implement messaging in their SSP applications are covered.

5.4.2 Event Classes

Event classes logically group specific events together. For example, the **Touch** event class groups all the messages associated with the touch controller of the screen. The **Temperature** class groups all the messages associated with temperature measurements, for example. Event classes are necessary for routing events to subscribers, because threads subscribe to event classes and not to individual events.

In the current SSP, there is no direct linkage between an event class and an event. For example, a **Start** event can be reused across multiple event classes. For instance, an audio recorder and an audio playback subsystem can both require **Start** events so they can share a single start event. For readability, it is better to have separate named events for each event class where the event is preceded by the name of the event class. For example, **Audio Playback Start** is a more descriptive name then the **Start** event.

The SSP has several predefined event classes. The Thermostat application uses two of these, **Touch** and **Audio Playback**. These classes populate in the **Event Classes** pane automatically when you include these modules in the Threads tab. The list of these predefined event classes might grow as the SSP evolves but for the purposes of this application note, we only describe two predefined event classes that the Thermostat application uses.

Besides predefined classes, you can create your own event classes to fit the requirement of your system. As shown in the following figure, the **Thermostat** application has five additional user-defined classes:

- System
- Volume
- Time
- Temperature
- Display

It is easier to understand how to create a new event class by examining the properties of an existing event class.

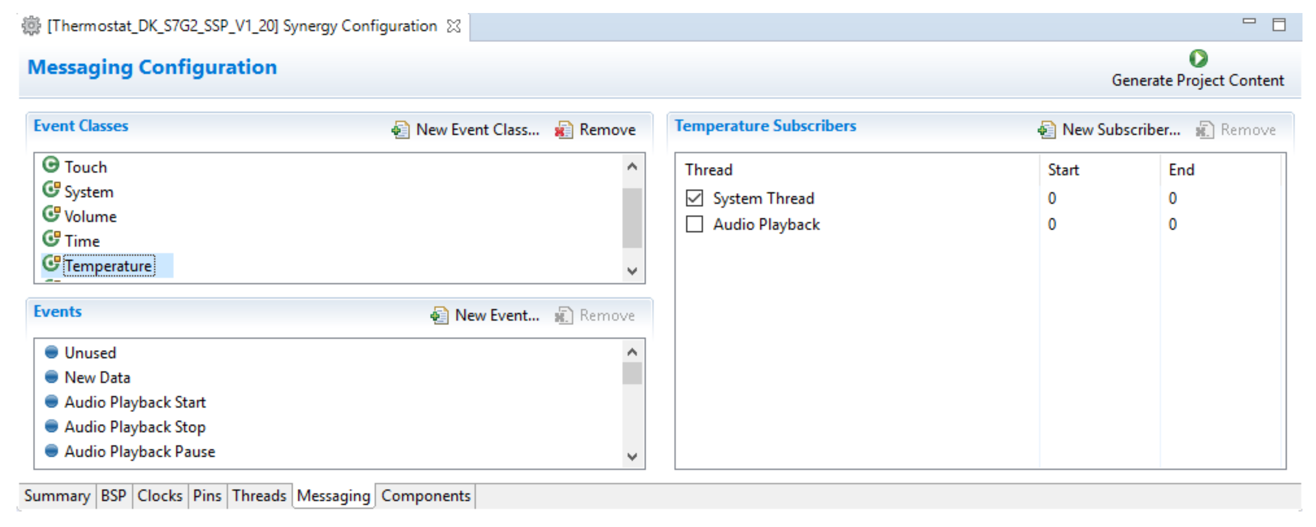


Figure 44. Examining the Temperature Event Class

When you click **Temperature Event Class** and examine the **Properties** tab, there are five properties as shown in the above figure. The five properties are:

- Symbol
- Name
- Payload header file
- Payload
- Payload type

The Synergy message configurator guides you through the process and all you need to do is decide on an appropriate name for your event class.

As shown in the **New Event Dialog** box that displays when you click the **New Event Class** above the **Event Class** window, all five fields are partially filled out for you. As you type the name of your event class, it is appended or prepended according to the Synergy naming conventions.

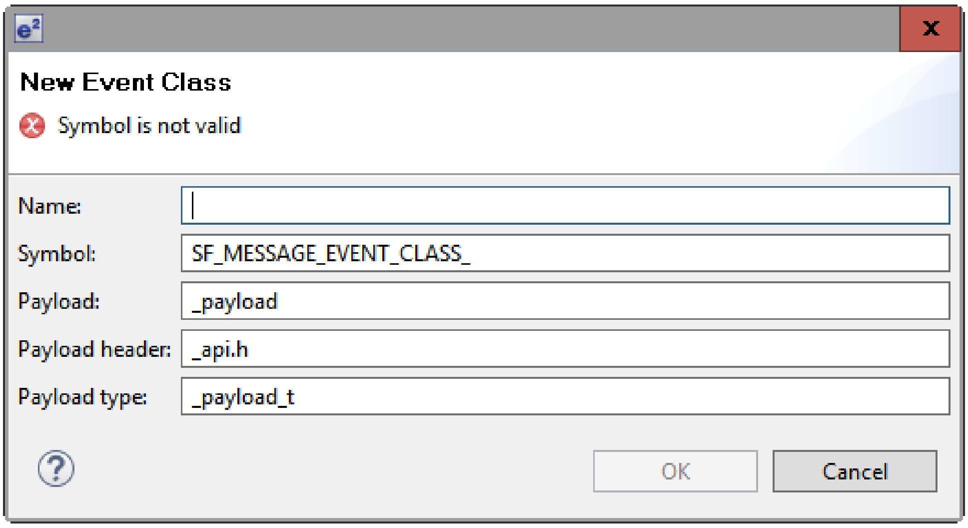


Figure 45. New Event Class Dialog Box Before Entering a Name

As an example, if you have an accelerometer on your hardware and you want to create a new event class for messages related to the accelerometer, you can type **Accelerometer** in the name field and the remaining fields are populated as shown in the following figure. Event class names can be any descriptive string, so consider readability when defining your event classes.

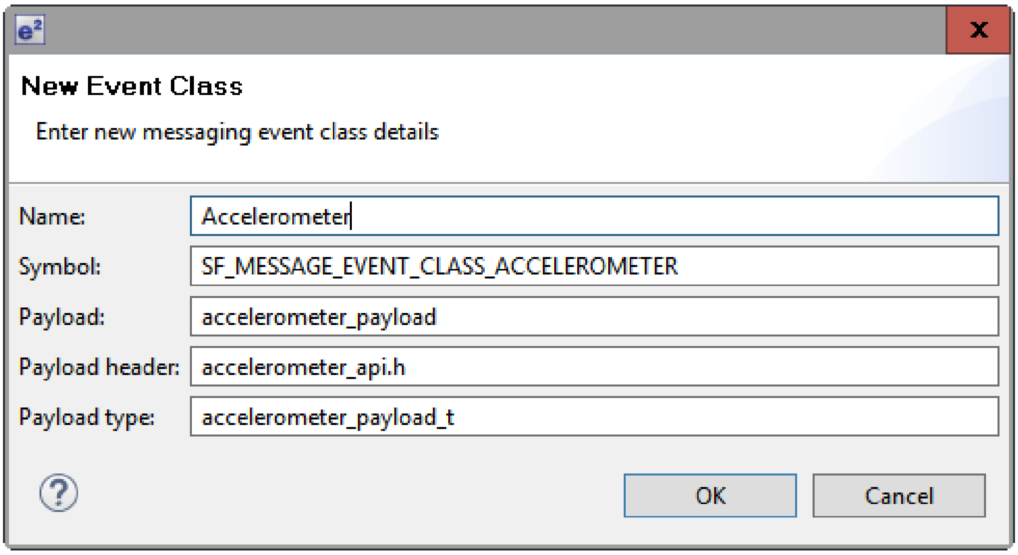


Figure 46. New Event Class Dialog After Entering a Name

You have defined important information that the SSP Framework requires at compile time. This information includes the name of the header file that defines the event class payload along with the **Payload type** name. To complete the creation of your new event class, you must create the header file.

Return to the existing temperature event class that is already defined in the application and examine the key element of the `temperature_api.h` file that is the `temperature_payload_t` structure.

```
#ifndef TEMPERATURE_API_H
#define TEMPERATURE_API_H
typedef struct st_temperature_payload
{
    float temperature;
} temperature_payload_t;
```

```
#endif /* TEMPERATURE_API_H */
```

The structure named `temperature_payload_t`, aligns with the **Payload type** property shown in Figure 46. This structure contains just a single float variable that is used to pass the temperature reading to any event subscribers.

In the background, the SSP builds an enumerated type of event class names in the structure `sf_message_event_class_t` that resides in the auto-generated file `synergy_cfg\ssp_cfg\framework\sf_message_port.h`. Note the comment that corresponds to the **Name** property:

```
typedef enum e_sf_message_event_class
{
  SF_MESSAGE_EVENT_CLASS_TOUCH, /* Touch */
  SF_MESSAGE_EVENT_CLASS_AUDIO, /* Audio Playback */
  SF_MESSAGE_EVENT_CLASS_SYSTEM, /* System */
  SF_MESSAGE_EVENT_CLASS_VOLUME, /* Volume */
  SF_MESSAGE_EVENT_CLASS_TIME, /* Time */
  SF_MESSAGE_EVENT_CLASS_TEMPERATURE, /* Temperature */
  SF_MESSAGE_EVENT_CLASS_DISPLAY, /* Display */
} sf_message_event_class_t;
```

5.4.3 Events

After the **Temperature Event** class is defined, you need to create some events. With the **Properties** tab open, click the **Temperature Increment** event. The **Name** and **Symbol** properties are displayed like the **Name** and **Symbol properties** in the **Event Class**.

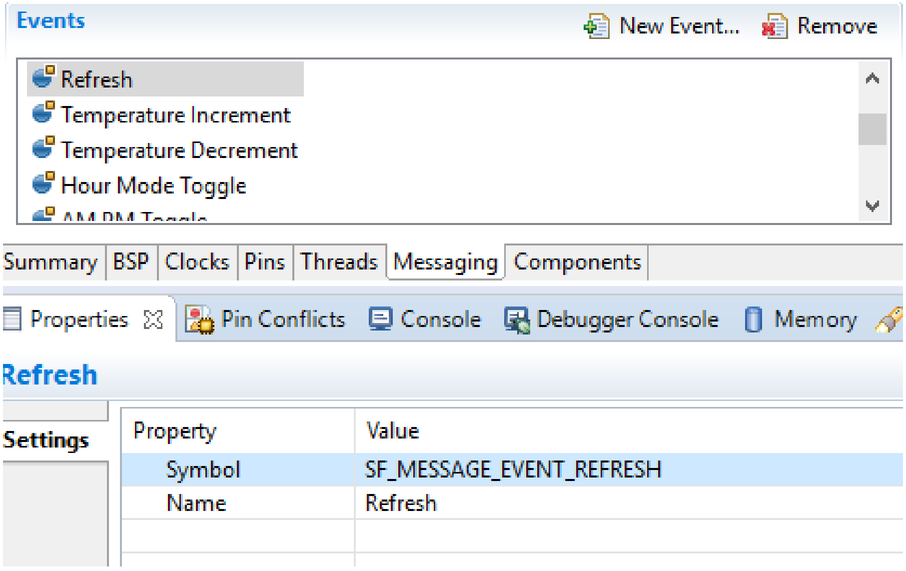


Figure 47. Temperature Increment Event Properties

The **Name** under **Property** contains the name you see in the Synergy Configuration window. The **Symbol** property is the actual #define you use in your code to refer to this event. In a later section, the code used to send and receive these types of events is examined. This only focuses on the location and how to create events.

Creating new events is the same as creating new event classes. When you click the small **New Event** at the top right of the **Events** dialog, the **New Event** dialog appears with the **Symbol** field partially populated.

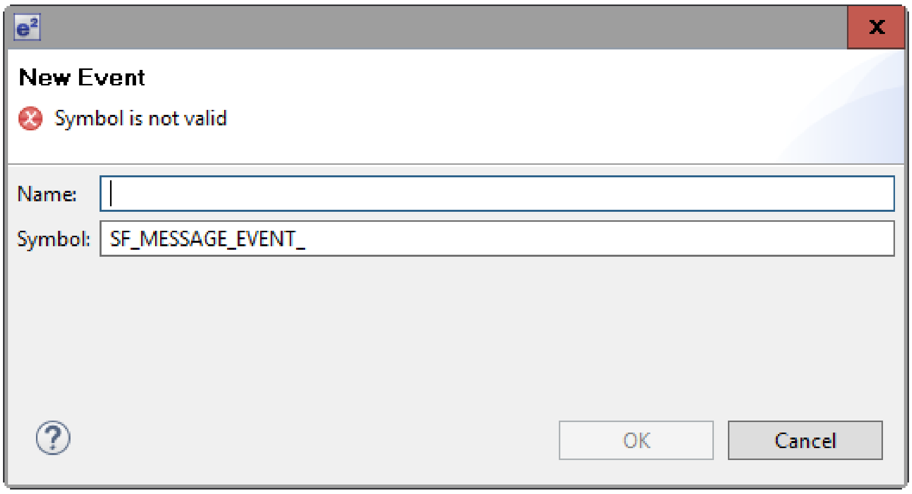


Figure 48. New Event Dialog

As you type the name of your event in the **Name** field, it is appended in upper case to the **Symbol** field. Therefore, if you are designing a system that has an accelerometer, for example, and you want a new data event, you only need to type **Accelerometer New Data** in the **Name** field. The SSP automatically appends this name to the **Symbol**, converting letters to upper case, and replacing spaces with the underscore character.

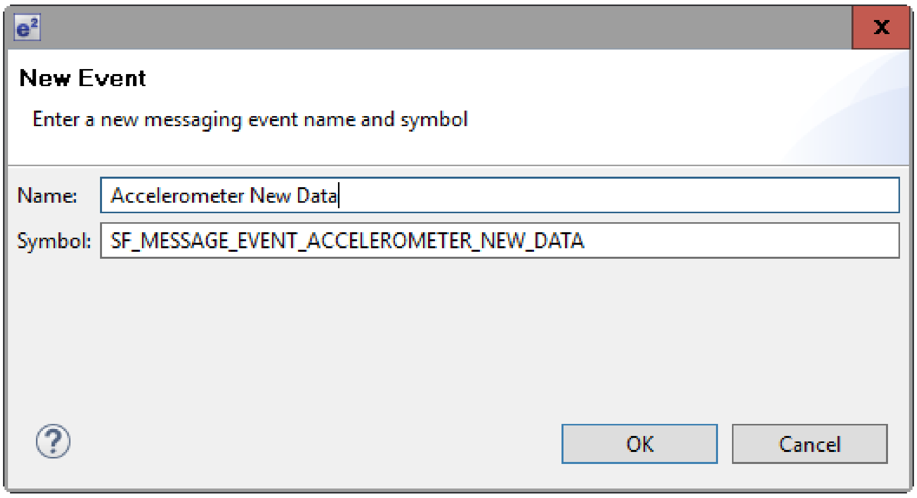


Figure 49. Example for Adding a New Event

While the SSP built an enumerated type containing all the event classes in the auto-generated file `synergy_cfg\ssp_cfg\framework\sf_message_port.h`, it also built an enumerated type `sf_message_event_t` for all the events you define. The comment for each field in the enumerated type aligns with the **Name** field of the **Event**.

Do not edit the `sf_message_port.h` file. It is automatically generated from the data contained in the `configuration.xml` file each time you click the **Generate Project Content** button at the top of the Synergy Configuration window, therefore any changes you make to this file are lost.

To see how it all works in the background, you can open the `configuration.xml` file in a text editor and browse the section labeled <**synergyMessagingConfiguration**>. You can find all the events and event classes you entered using the ISDE.

5.4.4 Event Class Subscribers

ThreadX uses a producer/consumer model for messaging. Producers place messages (events) on a message queue. Subscribers read those messages from the queue and act upon them. The last thing you must do when configuring messages for your system is to define the subscribers. The process is reduced to a simple point-and-click environment in the latest version of the SSP. Clicking on any event class automatically populates the **Subscribers** window to the right of the **Event Classes** window. It also prepends the event class name to the heading of the Subscriber window. As shown in the following figure, clicking on the **Temperature**, **Event Classes** shows the **Temperature Subscribers** what is currently the **System Thread**. **Event Classes** may have more than one subscriber. If you do not see the appropriate thread listed, click the **New Subscriber** to open the **New Subscriber Dialog** box, and use the drop-down arrow on the **Thread** line to select from a list of available threads.

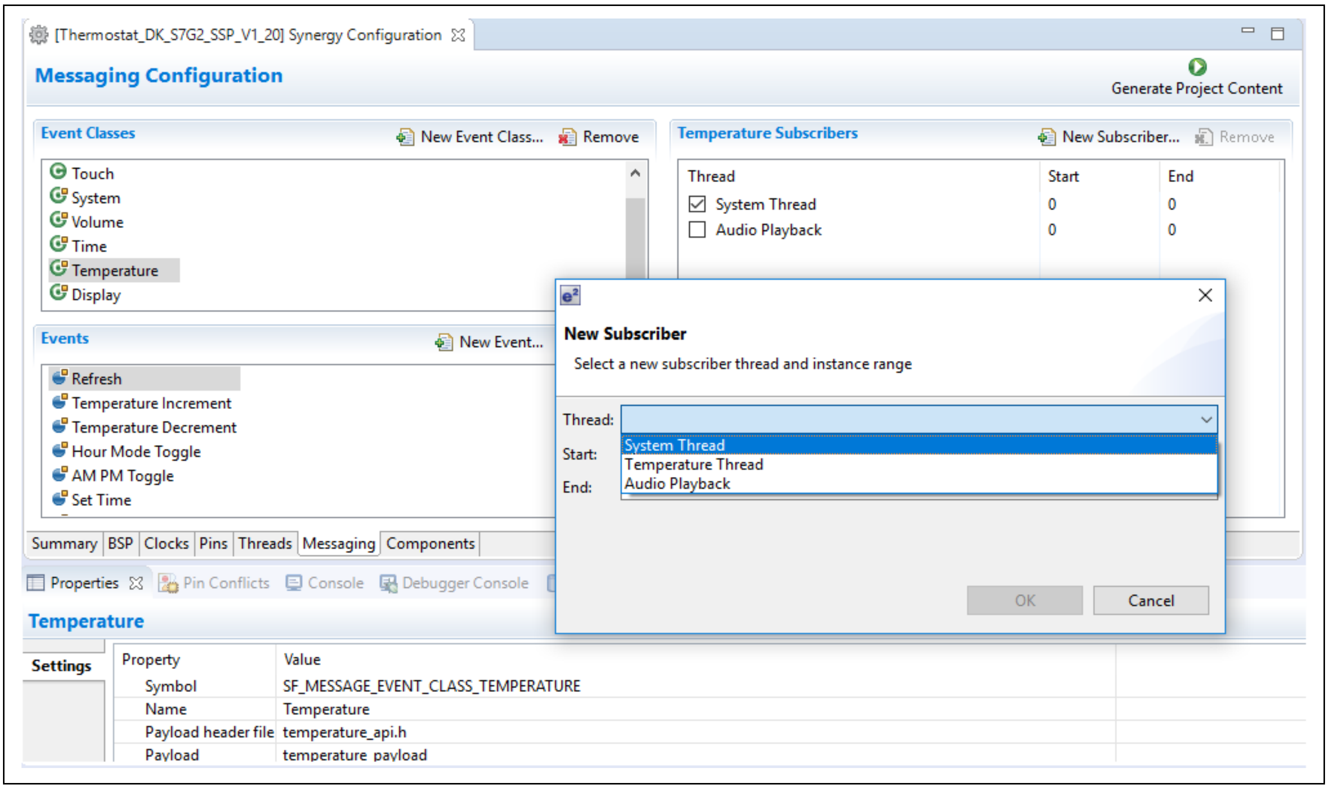


Figure 50. Adding Subscribers to an Event Class

6. Application Source Code Highlights

This section describes some of the more interesting highlights of the Thermostat application. The main purpose of the Thermostat application is to enable you to develop more complex multi-threaded HMI applications using ThreadX and GUIX in the SSP.

The key objective of the SSP is to reduce the complexity of interfacing with a myriad of ARM peripherals and, as quickly as possible, allows you to focus on constructing more complex applications.

Note: The Thermostat application was one of the first SSP applications written using some of the earliest releases of the SSP. Both the application and the SSP have gone through numerous iterations and it is now a stable process. Many of the tools for configuring the SSP framework were not present earlier. While the SSP has considerably improved over the life of the Thermostat application, the SSP is being continually improved for quicker development of complex applications.

6.1 Threads and Main

There are a few subtle differences when using ThreadX in the SSP environment. In a typical ThreadX application, the `main()` function calls `tx_kernal_enter()` that then calls `tx_application_define()`. If you have written ThreadX applications prior to working with the Renesas Synergy Platform, you may be used to creating the main application threads and defining other resources used by the application for example, queues, and semaphores in `tx_application_define()`.

In the Synergy framework, `main()` is an auto-generated file which looks similar to the following code. In this case, `tx_application_define()` calls thread entry functions for the threads specified during the SSP framework configuration.

```
void tx_application_define(void* first_unused_memory)
{
    system_thread_create ();
    temperature_thread_create ();

#ifdef TX_USER_ENABLE_TRACE
    TX_USER_ENABLE_TRACE;
#endif

    g_hal_init ();

    tx_application_define_user (first_unused_memory);
}

void main(void)
{
    __disable_irq ();
    tx_kernel_enter ();
}
```

When you create a thread using the **Threads** tab, the application framework creates several files. As an example, when the **System Thread** was added, the application framework created three files as shown in Figure 51:

- `system_thread.h`
- `system_thread.c`
- `system_thread_entry.c`

The first two files are auto-generated and therefore placed in the `synergy_gen` folder. The `system_thread_entry.c` file is the entry point for the system thread and this is where you put your application code. You should not update auto-generated files because they are regenerated every time you build the project or click the **Generate Project Content** button. Auto-generated files always contain some form of **-- do not edit --** message at the top of the file.

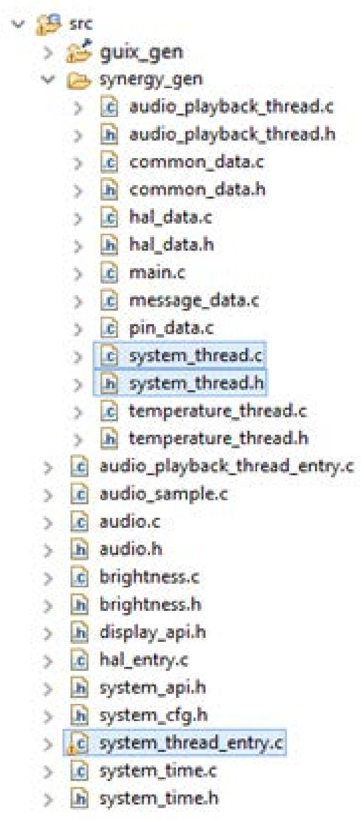


Figure 51. System Thread Files

6.1.1 GUIX Initialization

The GUIX system is not automatically initialized by the application framework. Several calls are required to initialize GUIX and create the initial canvas where the drawing takes place. You can find this initialization code at the top of the `system_thread_entry()` function located in the `system_thread_entry.c` file.

```
/* Initializes GUIX. */
    status = gx_system_initialize();
    if(TX_SUCCESS != status)
    {
        while(1);
    }

    /* Initializes GUIX drivers. */
    err = g_sf_el_gx.p_api->open (g_sf_el_gx.p_ctrl, g_sf_el_gx.p_cfg);
    if(SSP_SUCCESS != err)
    {
        while(1);
    }

    gx_studio_display_configure ( DISPLAY,
                                  g_sf_el_gx.p_api->setup,
                                  LANGUAGE_ENGLISH,
                                  DISPLAY_THEME_1,
                                  &p_root );

    err = g_sf_el_gx.p_api->canvasInit(g_sf_el_gx.p_ctrl, p_root);
    if(SSP_SUCCESS != err)
    {
        while(1);
        }
```

6.1.2 Events and GUIX Message

Touching the screen in the Thermostat application causes GUIX to invoke the specific callback function defined for that screen in GUIX studio. GUIX provides the callback function with specific information about the window that caused the event and the actual event that occurred. There are currently 46 different event types recognized by GUIX. They are defined in the `gx_api.h` file and reproduced in this document for convenience.

```
/* Define the pre-defined Widget event types.  */

#define GX_EVENT_TERMINATE                  1
#define GX_EVENT_REDRAW                     2
#define GX_EVENT_SHOW                       3
#define GX_EVENT_HIDE                       4
#define GX_EVENT_RESIZE                     5
#define GX_EVENT_SLIDE                      6
#define GX_EVENT_FOCUS_GAINED               7
#define GX_EVENT_FOCUS_LOST                 8
#define GX_EVENT_HORIZONTAL_SCROLL          9
#define GX_EVENT_VERTICAL_SCROLL            10
#define GX_EVENT_TIMER                      11
#define GX_EVENT_PEN_DOWN                   12
#define GX_EVENT_PEN_UP                     13
#define GX_EVENT_PEN_DRAG                   14
#define GX_EVENT_KEY_DOWN                   15
#define GX_EVENT_KEY_UP                     16
#define GX_EVENT_CLOSE                      17
#define GX_EVENT_DESTROY                    18
#define GX_EVENT_SLIDER_VALUE               19
#define GX_EVENT_TOGGLE_ON                  20
#define GX_EVENT_TOGGLE_OFF                 21
#define GX_EVENT_RADIO_SELECT               22
#define GX_EVENT_RADIO_DESELECT             23
#define GX_EVENT_CLICKED                    24
#define GX_EVENT_LIST_SELECT                25
#define GX_EVENT_VERTICAL_FLICK             26
#define GX_EVENT_HORIZONTAL_FLICK           28
#define GX_EVENT_MOVE                       29
#define GX_EVENT_PARENT_SIZED               30
#define GX_EVENT_CLOSE_POPUP                31
#define GX_EVENT_ZOOM_IN                    32
#define GX_EVENT_ZOOM_OUT                   33
#define GX_EVENT_LANGUAGE_CHANGE            34
#define GX_EVENT_RESOURCE_CHANGE            35
#define GX_EVENT_ANIMATION_COMPLETE         36
#define GX_EVENT_SPRITE_COMPLETE            37
#define GX_EVENT_TEXT_EDITED                40
#define GX_EVENT_TX_TIMER                   41
#define GX_EVENT_FOCUS_NEXT                 42
#define GX_EVENT_FOCUS_PREVIOUS             43
#define GX_EVENT_FOCUS_GAIN_NOTIFY          44
#define GX_EVENT_SELECT                     45
#define GX_EVENT_DESELECT                   46
```

The Thermostat application uses only a few of these events such as GX_EVENT_CLICKED. GUIX passes these events as a GX_EVENT structure. The first element of the structure is the event type. A GX_EVENT allows data to be sent as part of the message. The final field, gx_event_payload, is a combination of various data types. The Thermostat application uses this payload to send a pointer to the current data structure state.

```
/* Define Event type.  Note: the size of this structure must be less than
or equal to the constant
   GX_EVENT_SIZE defined previously.  */

typedef struct GX_EVENT_STRUCT
{
    ULONG  gx_event_type;                                   /* Global
event type                   */
    ULONG  gx_event_display_handle;
    struct GX_WIDGET_STRUCT *gx_event_target;               /*
receiver of event                      */
    USHORT gx_event_sender;                                 /* ID of
the event sender                 */
    union
    {
        UINT     gx_event_timer_id;
        GX_POINT gx_event_pointdata;
        GX_UBYTE gx_event_uchardata[4];
        USHORT   gx_event_ushortdata[2];
        ULONG    gx_event_ulongdata;
        GX_BYTE  gx_event_chardata[4];
        SHORT    gx_event_shortdata[2];
        INT      gx_event_intdata[2];
        LONG     gx_event_longdata;
    } gx_event_payload;
} GX_EVENT;
```

6.1.3 User defined GUIX messages

When you click on a screen object, it is often you are asking for a change to the system state. As an example, when you toggle the **Fan** mode button on the **Thermostat** screen, you are asking the system to change the state of the fan from Auto to On or visa-versa. Because the **System Thread** is responsible for maintaining and updating the logical state of the machine, the **GUIX Thread** sends a message to the **System Thread** requesting a change to the Fan mode. The following figure illustrates this concept with a simplified message sequence diagram.

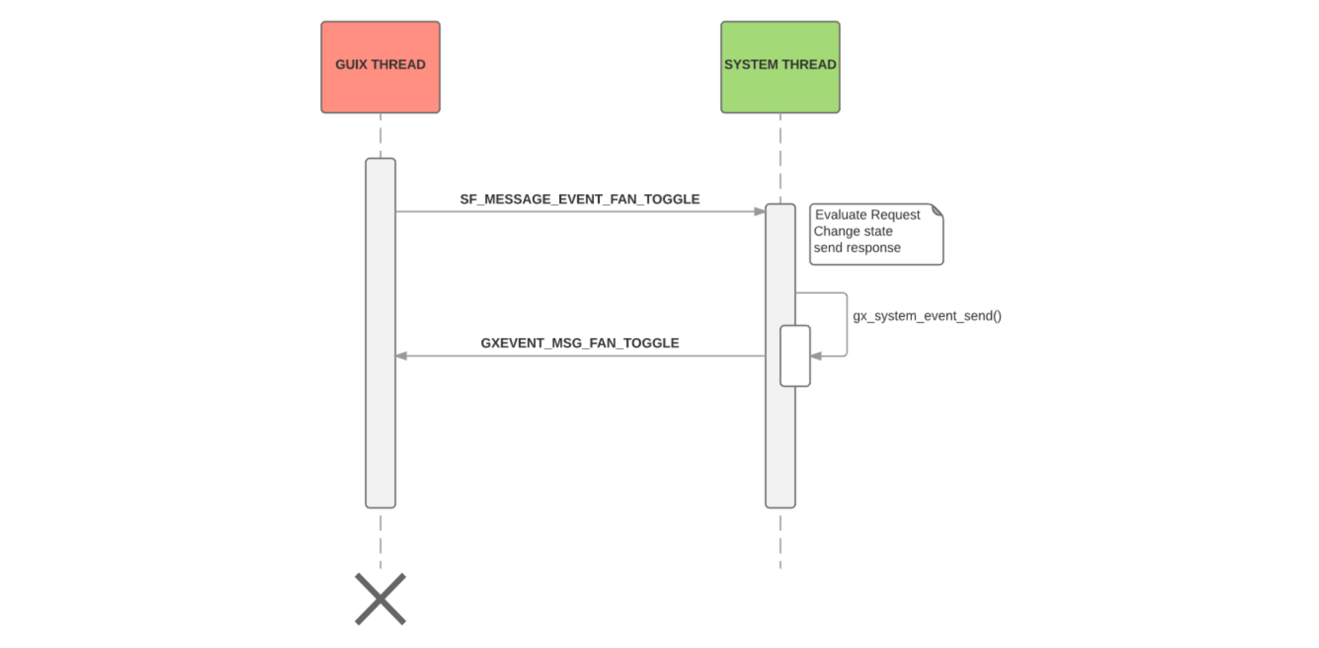


Figure 52. Simplified System Message Sequence Diagram

The **System Thread** evaluates the request, makes any necessary changes to the system state and if appropriate, sends a message back to the **GUIX Thread** to update the screen.

The GUIX system has numerous predefined messages to communicate GUIX events to the application through the provided callback functions. In addition to these predefined events, GUIX also provides a method for other threads to send user-defined messages to GUIX using the `gx_system_even_send()` function.

To separate user-defined events from standard GUIX events, you must pick an event ID numbers larger than the #define value of `GX_FIRST_APP_EVENT` defined in the `gx_api.h` file.

```
/* Define the value of the first application defined event type.  */

#define GX_FIRST_APP_EVENT                          0x40000000
```

GUIX does not attempt to process any message with a `gx_event_type` ID larger than `GX_FIRST_APP_EVENT`; it only passes the event to the event handler of the currently active window through the callback function defined for that screen. This is the method the **System Thread** uses to inform the currently active screen to update to the latest state.

The **System Thread** utilizes `gx_event_payload` element of the `GX_EVENT` structure to pass a pointer to the static data structure containing the system state variables. When the screen event handler receives this message, it uses the state information passed in the message to update the current screen display state.

The following figure shows the list of message requests and responses the Thermostat application uses. Requests are sent from the GUIX event handler functions to the **System Thread** using the SSP messaging framework. You define these messages using the **Messaging** tab in the **Synergy Configuration** window. The system saves them as enumerated types in the file `synergy_cfg/ssp_config/framework/sf_message_port.h`. Responses are sent from the **System Thread** to **GUIX**.

Type	Request	Response
Touch	SF_MESSAGE_EVENT_NEW_DATA	
System	SF_MESSAGE_EVENT_FAN_TOGGLE	GXEVENT_MSG_FAN_TOGGLE
	SF_MESSAGE_EVENT_SYSTEM_MODE_TOGGLE	GXEVENT_MSG_SYSTEM_MODE_TOGGLE
Temperature	SF_MESSAGE_EVENT_TEMPERATURE_INCREMENT	GXEVENT_MSG_TEMPERATURE_INCREMENT
	SF_MESSAGE_EVENT_TEMPERATURE_DECREMENT	GXEVENT_MSG_TEMPERATURE_DECREMENT
	SF_MESSAGE_EVENT_TEMPERATURE_UNIT_C	GXEVENT_MSG_TEMPERATURE_UNIT_C
	SF_MESSAGE_EVENT_TEMPERATURE_UNIT_F	GXEVENT_MSG_TEMPERATURE_UNIT_F
	SF_MESSAGE_EVENT_UPDATE_TEMPERATURE	GXEVENT_MSG_UPDATE_TEMPERATURE
Time	SF_MESSAGE_EVENT_HOUR_MODE_TOGGLE	GXEVENT_MSG_HOUR_MODE_TOGGLE
	SF_MESSAGE_EVENT_AM_PM_TOGGLE	GXEVENT_MSG_AM_PM_TOGGLE
	SF_MESSAGE_EVENT_TIME_UPDATE	GXEVENT_MSG_TIME_UPDATE
	SF_MESSAGE_EVENT_UPDATE_TIME	GXEVENT_MSG_UPDATE_TIME
Date	SF_MESSAGE_EVENT_DATE_UPDATE	GXEVENT_MSG_DATE_UPDATE
	SF_MESSAGE_EVENT_UPDATE_DATE	GXEVENT_MSG_UPDATE_DATE
Brightness	SF_MESSAGE_EVENT_BRIGHTNESS_INCREMENT	GXEVENT_MSG_BRIGHTNESS_INCREMENT
	SF_MESSAGE_EVENT_BRIGHTNESS_DECREMENT	GXEVENT_MSG_BRIGHTNESS_DECREMENT
Volume	SF_MESSAGE_EVENT_VOLUME_INCREMENT	GXEVENT_MSG_VOLUME_INCREMENT
	SF_MESSAGE_EVENT_VOLUME_DECREMENT	GXEVENT_MSG_VOLUME_DECREMENT

Figure 53. System Message/Response Defines

Responses from the **System** thread to the **GUIX** thread are handled differently. GUIX has its own message passing facility. Messages are sent to the GUIX thread using the `gx_system_event_send()` call. The response messages defined in the above figure are not configured in the **Synergy Configuration Messaging Pane**. They must be defined by the application programmer. In the **Thermostat Application**, these messages are defined in the `src/system_api.h` file.

6.2 Handling Screen Events

This section walks you through the code involved in changing the **Fan** mode on the **Thermostat** screen. For this task, you should have the `system_thread_entry.c` and `hmi_event_handler.c` files opened as you follow along with the text.

The following figure contains a detailed message sequence diagram. The logic and data flow are similar for all screen input and update events.

The function prototype for the screen event handler as defined in the `thermostat_specifications.h` file is as follows:

```
UINT thermostat_screen_event(GX_WINDOW * p_window, GX_EVENT *event_ptr)
```

This header file is auto-generated by the application framework but you must write the actual event handler code yourself. If your screens are complex, you may consider separating event handlers for each screen into separate files. In the **Thermostat** application, all the code for all the screen handlers exists in the `hmi_event_handler.c` file.

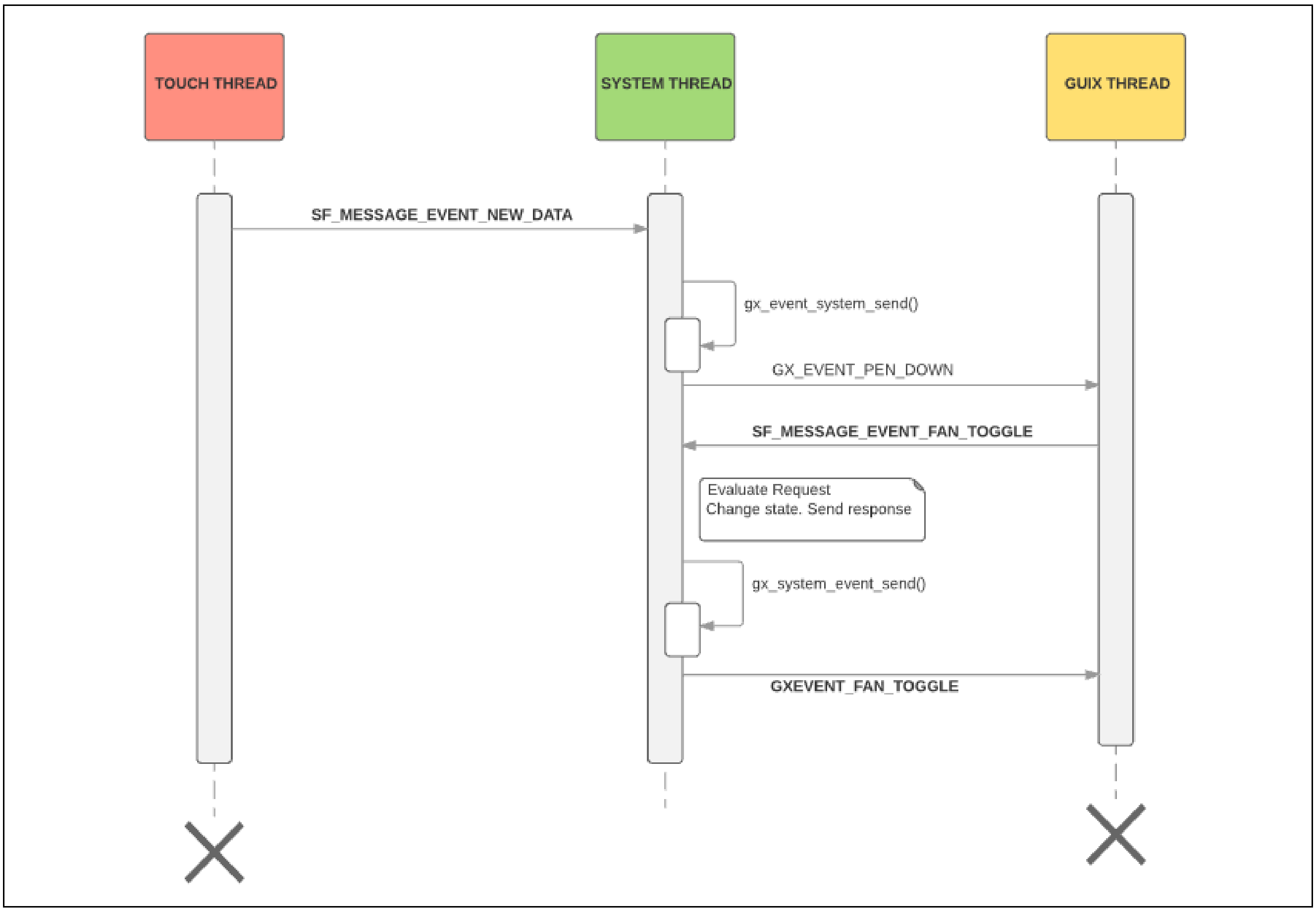


Figure 54. Detailed System Message Diagram

The following section shows a partial extraction of the code for the thermostat screen event handler, focusing on just two cases in the switch statement. Unrelated code has been omitted with the … characters in the code. When you click the **Fan** button in the **Thermostat** screen, GUIX calls this event handler with a pointer to the window that caused the event to occur and a pointer to the GX_EVENT structure which contains the `gx_event_type` field.

```
UINT thermostat_screen_event(GX_WINDOW * p_window, GX_EVENT *event_ptr)
{
    UINT result = gx_window_event_process(p_window, event_ptr);
    GX_WIDGET * p_widget = (GX_WIDGET *) p_window;

    switch (event_ptr->gx_event_type)
    {
        …
        case THERMO_MESSAGE_EVENT_FAN_TOGGLE:
            update_state_data(event_ptr);
            if (SYSTEM_FAN_MODE_AUTO == g_gui_state.fan_mode)
            {
                update_pixelmap_button_id((GX_WIDGET *) p_widget,
ID_FAN_BUTTON,
                                                   GX_PIXELMAP_ID_BLACKBUTTON);
                update_text_id((GX_WIDGET *) p_widget, ID_FAN_MODE_TEXT,
GX_STRING_ID_FAN_MODE_AUTO);
            }
            else
            {
                update_pixelmap_button_id((GX_WIDGET *) p_widget,
ID_FAN_BUTTON,
                                                   GX_PIXELMAP_ID_GREENBUTTON);
```

```
                    update_text_id((GX_WIDGET *) p_widget, ID_FAN_MODE_TEXT,
GX_STRING_ID_FAN_MODE_ON);
                }
                if (g_gui_state.fan_on)
                {
                    show_hide_widget((GX_WIDGET *) p_widget, ID_FAN_ICON, 1);
                }
                else
                {
                    show_hide_widget((GX_WIDGET *) p_widget, ID_FAN_ICON, 0);
                }
                break;
    …
        case GX_SIGNAL(ID_FAN_BUTTON, GX_EVENT_CLICKED):
            /** Create fan toggle message. */
            send_update_request(SF_MESSAGE_EVENT_CLASS_SYSTEM,
SF_MESSAGE_EVENT_FAN_TOGGLE);
            break;
    …
        case GX_SIGNAL(ID_BACK_BUTTON, GX_EVENT_CLICKED):
    …
            /** Returns to main screen. */
            show_window((GX_WINDOW*)&MainPage, (GX_WIDGET*) p_widget,
true);
            break;
    …
    default:
        break;
    }
```

The code switches off `event_ptr->gx_event_type`, the three cases that were selected illustrate most of what you need to follow any GUIX event handling in the Thermostat application. Typically, the screen handler is associated with three different types of events, listed as follows:

1. Events such as the `ID_BACK_BUTTON` return to the previous screen. The event handler processes this type of event immediately. No interaction with the system thread occurs.
2. Events such as the `ID_FAN_BUTTON` that may result in a change to the system state. Because the system thread is responsible for determining the validity of the system state changes, this type of event causes a message to be sent to the **System Thread** requesting a **System State update** using the `send_update_request()` call.
3. User-defined events sent from the **System Thread**, generally to cause an update of display data to occur. These types of events are enumerated to be larger than the `#define` value of `GX_FIRST_APP_EVENT`. This event is the system thread response to an update request instructing the screen to update to the current system state.

Statements that switch off GUIX events typically make use of the GUIX-defined macro `GX_SIGNAL` specified in the `gx_api.h` file.

```
#define GX_SIGNAL(_a, _b)                   (((_a) << 8) | (_b))
```

`GX_EVENT_CLICKED` is one of the standard GUIX events that occurs when you touch a button widget on the screen. The `ID_FAN_BUTTON` define is created by GUIX Studio when you design the screens. From the partial list shown below, there are 71 individually defined elements in the screen design created in GUIX Studio for the Thermostat application.

```
/* Define widget ids
*/

#define ID_HELP_SCREEN 1
#define ID_THERMO_HELP 2
#define ID_HELP_CLOSE_BUTTON 3
#define ID_TIME_TEXT 4
#define ID_DATE_TEXT 5
…
#define ID_FAN_BUTTON 13
#define ID_FAN_MODE_TEXT 14
…
#define ID_MAINPAGE_SCREEN 68
#define ID_THERMO_BUTTON 69
#define ID_SETTINGS_BUTTON 70
#define ID_WEEKDAY_TEXT 71
```

As previously stated, the **Fan** button requires a state change to the **fan** mode. Any screen event that requires a state change calls the following function:

```
send_update_request(SF_MESSAGE_EVENT_CLASS_SYSTEM,SF_MESSAGE_EVENT_REQUEST_FAN_TOGGLE);
```

This function sends an update request message to the **System thread**, as shown in the flow of Figure 54. It is important to remember that the code in `hmi_event_handler.c` is running in the context of the **GUIX thread**. As mentioned earlier, the system thread is responsible for updating the system data with the best practices of separating the business logic from the presentation logic.

6.3 Maintaining and Updating the System State

The **Thermostat screen handler** sends an update request message to the **System thread** to toggle the **fan** mode. The **System thread** is responsible for evaluating the request, updating the state variables, and informing the screen to update if necessary. The state variables for this application are contained in a single structure type `system_state_t` defined in `system_api.h`.

```
typedef struct st_system_state
{
        rtc_time_t          time;
        system_hour_mode_t  hour_mode;
        system_mode_t       mode;
        system_fan_mode_t   fan_mode;
        system_temp_units_t temp_units;
        float               temp_c;       ///< In Celsius
        float               target_temp_c;  ///< In
system_state_t::temp_units
        uint8_t             volume;
        uint8_t             brightness;
        bool                fan_on;
} system_state_t;
```

The structure is allocated and initialized, `g_system_state` at the top of the `system_thread_entry.c` file. You can see all the Thermostat system variables, including the `fan_mode` in the structure definition.

The first thing the system thread does is a pend operation on the system message queue.

```
err = g_sf_message.p_api->pend(g_sf_message.p_ctrl, &g_system_queue,
                    (sf_message_header_t **) &p_message, TX_WAIT_FOREVER);
```

When a message arrives, the system thread executes and switches off the class type, then the message type that are both embedded in the message. In the case of the fan mode, the code toggles the state of the system fan mode variable. It then sets the `system_state_changed` variable to true that causes an update message to be sent to the GUIX thread.

```
switch (p_message->event_b.class)
{
case SF_MESSAGE_EVENT_CLASS_SYSTEM:
                  {
                  switch (p_message->event_b.code)
                      {
              case SF_MESSAGE_EVENT_REQUEST_FAN_TOGGLE:
                  {
                        /** Toggles fan setting. */
                      g_system_state.fan_mode = !g_system_state.fan_mode;

                system_state_changed = true;
                break;
            }
```

Once the message is processed the buffer containing the message is released.

```
      err = g_sf_message.p_api->bufferRelease(g_sf_message.p_ctrl,
(sf_message_header_t *)
                           p_message, SF_MESSAGE_RELEASE_OPTION_NONE);
...
        /** Turn fan on or off based on fan mode, system mode, and target
temperature. */
        if (SYSTEM_FAN_MODE_ON == g_system_state.fan_mode)
        {
            if (!g_system_state.fan_on)
            {
                system_state_changed  = true;
                g_system_state.fan_on = true;
            }
        }
...
        if (system_state_changed)
        {
            /** Create message. */
            gx_message.gx_event.gx_event_type = GX_FIRST_APP_EVENT;
            gx_message.state = g_system_state;
            gx_message.gx_event.gx_event_payload.gx_event_ulongdata = (ULONG)
&gx_message;

            /** Post message. */
            gx_system_event_send(&gx_message.gx_event);
        }
    }
}
```

6.4 LCD Control

The 4.3-inch display has a couple of digital controls that must be driven from the Thermostat application. As is the case with most embedded applications, the first thing you must do is identify the hardware dependencies, and set up the appropriate drivers.

The two signals in this section are the `LCD_ON` and `LCD_BLEN` with Blanking Enable. The following figure shows an excerpt from the DK-S7G2 v3.0 schematic that shows the J102 connector. This is the connector the LCD screen plugs into. The two signals list the associated MCU pins, P7 10 and P7 12. The `LCD_ON` signal requires a simple Hi/Lo state that turns the LCD ON and OFF. The `LCD_BLEN` signal requires a PWM signal that modulates the display intensity.

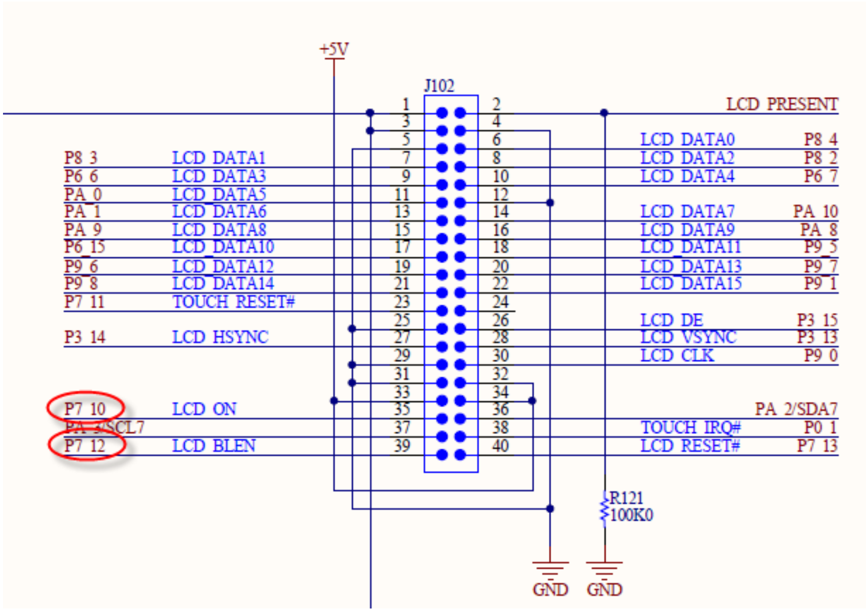


Figure 55. LCD Control Signals

The following figure shows the **Pins** tab for the DK-S7G2 Thermostat application. The first thing you do when configuring a pin is to select the port from the **Pin Selection** dialog box on the left-hand side of the screen, in this case P7 (port 7). The port selection expands to show the pins associated with the I/O port.

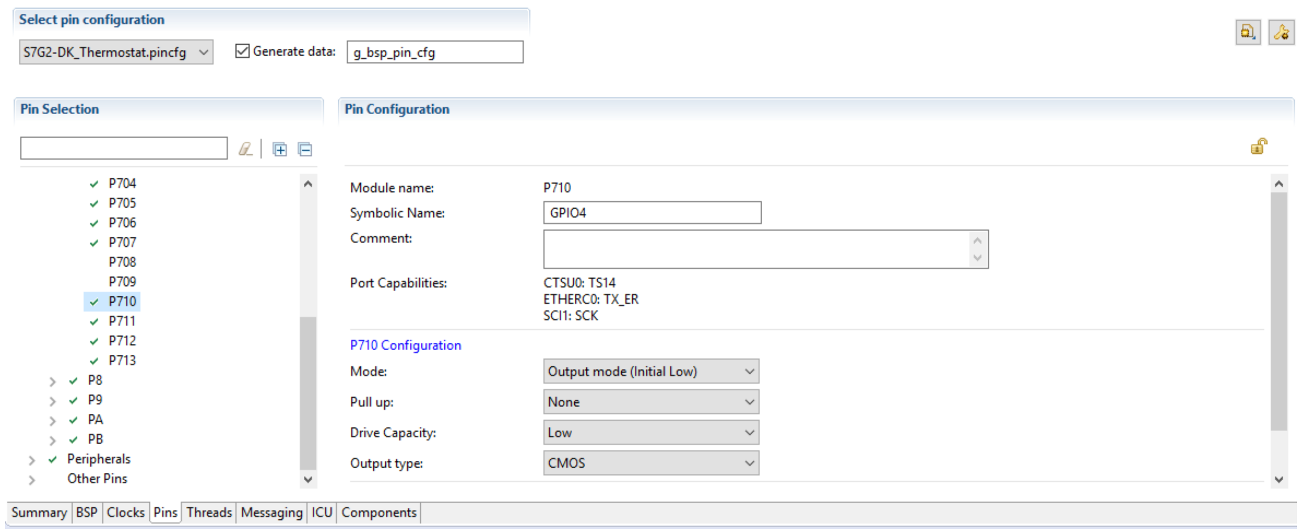


Figure 56. Configuring an Individual Pin

This is the simplest configuration you can have for a GPIO pin; the mode is set to **Output mode**, the input/output of the MCU is set to **GPIO**. Notice that the module name is P710, which is the naming convention you see in the Synergy pin configuration. Declaring an I/O pin in this manner causes the Synergy framework to create an instance of the pin in the `hal_data.c` file that is an auto-generated file.

```
const ioport_instance_t g_ioport =
{ .p_api = &g_ioport_on_ioport, .p_cfg = NULL };
```

This provides driver level access to the pin but you can write code that sets the state of the pin. In this case, all that is required for the Thermostat application is to set the pin high to turn the LCD display on. This is accomplished during the initialization code in the `system_thread_entry.c` file.

```
/* Controls the GPIO pin for LCD ON. */
 err = g_ioport.p_api->pinWrite(LCD_ON_PIN, IOPORT_LEVEL_HIGH);
 if (err)
 {
     while(1);
 }
```

For this demonstration application, the error handling loops with a `while(1)` condition if an error is returned from the `g_ioport.p_api->pinWirte()` call. It causes the system thread to stop responding should an error be returned from the `pinWrite` call.

6.5 Backlight Control

The Thermostat application, running on the DK-S7G2 MCU board, provides an example of how the screen brightness is typically varied on embedded systems. The **Display** option, in the **Settings** screen, brings up the **Brightness Control** screen as shown in the following figure. This screen is used for controlling the screen brightness.

There are two items for discussion in this section. First, there are no callback functions defined for these sub-screens in GUIX Studio. Second, the brightness control relies on a Pulse Width Modulation, (PWM) signal to control the brightness of the screen.

Figure 57. Brightness Control Screen

Referring to the schematic for the DK-S7G2 LCD, you see that a `CAT4139 LED` driver is used to drive the backlight LEDs on the LCD screen. Driving pin 4 of this device with a PWM signal can vary the brightness of the screen. Figure 55 shows this pin connected to Port 7 Pin 12 so configure this pin to have a PWM output. From Figure 3 earlier, this project includes the **General PWM Timer** resource of the S7G2 MCU that required the `r_gpt` driver to be loaded. When you click the **System Thread** tab in the **Synergy Configuration** window, and then click on the `g_pwm_backlight Timer Driver` on `r_gpt` in the **System Threads** module, you can see the following displayed in the **Properties** tab.

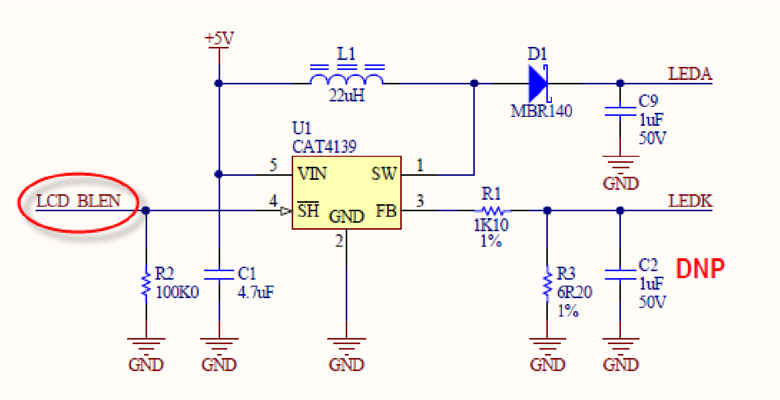


Figure 58. DK-S7G2 Backlight Control Circuit

The GPT driver for this pin is set to the **PWM** mode with the units set to **milliseconds**. The **Duty Cycle Unit** is set to **Unit Percent**. The **Period Value** is set to **10 milliseconds**, and the **Duty Cycle Value** is set to **50**.

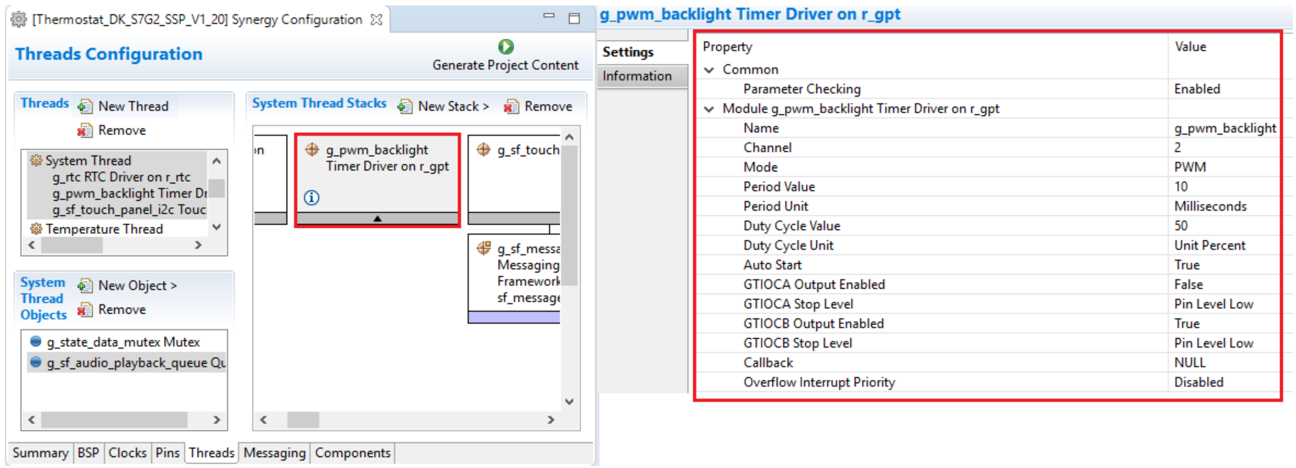


Figure 59. PWM Properties for Backlight Control

Enabling this driver in the application framework causes an instance of the driver to automatically be initialized in `system_thread.c` file:

```
/* Instance structure to use this module. */
const timer_instance_t g_pwm_backlight =
{ .p_ctrl = &g_pwm_backlight_ctrl, .p_cfg = &g_pwm_backlight_cfg, .p_api =
&g_timer_on_gpt };
```

The *Synergy Software Package (SSP) User's Manual* contains a detailed API Reference for the various modules you may install in the Synergy framework. The code in many cases is self-documented and finding the various API calls for a given module can be done quickly in e² studio. For example, the `p_api` element of this array points to the API structure defined in the system `r_gptc` file. Examining this structure reveals the API calls available for the **General Purpose Timer** (GPT) driver. These are the same API calls described in the *SSP User's Manual*.

```
/** GPT Implementation of General Timer Driver  */
const timer_api_t g_timer_on_gpt =
{
    .open            = R_GPT_TimerOpen,
    .stop            = R_GPT_Stop,
    .start           = R_GPT_Start,
    .reset           = R_GPT_Reset,
    .periodSet       = R_GPT_PeriodSet,
    .counterGet      = R_GPT_CounterGet,
    .dutyCycleSet    = R_GPT_DutyCycleSet,
    .infoGet         = R_GPT_InfoGet,
```

```
        .close              = R_GPT_Close,
        .versionGet         = R_GPT_VersionGet
    };
```

As is the case with most driver related calls, the first thing to do is call the driver open call. This is done in the `system_thread_entry.c` file that turns on the LCD:

```
    /* Opens PWM driver and controls the TFT panel back light. */
    err = g_pwm_backlight.p_api->open(g_pwm_backlight.p_ctrl,
    g_pwm_backlight.p_cfg);
    if (err)
    {
        while(1);
    }
```

The GUIX screen interaction with the system thread works like the method described above for the **Fan Mode** selection. Pressing the increment or decrement arrows on the Brightness screen causes GUIX to call the `settings_screen_event()` routine with a `GX_EVENT_CLICKED` message. The handler calls `send_update_request()` to request a system state change. The system thread processes the request and calls either one of the helper functions, `brightness_up()` or `brightness_down()` located in the `brightness.c` file to update the state data appropriately. The system thread then sends the GX_FIRST_APP event message to the GUIX thread causing the screen to be updated.

The following figure shows the PWM signal generated with the brightness set to 95% and 10%. In both cases, a 100 Hz signal is generated, setting the intensity to 10% on the Display screen to yield a signal that is low most of the time, as shown in the following image. This disables the LED driver most of the time when dimming the screen.

Figure 60. PWM Signals for Backlight Control

6.6 Temperature Thread

To simulate a real-world temperature measurement in the Thermostat application, the on-board temperature sensor of the S7G2 Synergy MCU Group is used. The ADC unit of the S7G2 MCU is a complex unit with many operating modes and features. The Synergy framework provides an easy way to access the unit for common use cases.

The use of a separate thread is more than required for this application but it illustrates how you can easily separate measurements from your main thread. In more complex control and measurement applications, this can simplify the application logic required for the task.

6.6.1 Configuring the ADC

You can configure the hardware peripherals of the S7G2 MCU using the Synergy framework by:

- Determining what components are required and adding the appropriate application framework and HAL drivers. You can add them in the **Components** tab of the **Synergy Configuration** window.
- Determining what thread processes the peripheral data and adding the appropriate modules to it in the **Threads** tab.
- Adding any supporting thread objects such as mutex, and semaphore required by the thread handling the data and creating those in the **Thread Objects** dialog.
- Identifying any actual physical connections to the peripherals, if any, and mapping specific MCU pins to the associated peripheral unit in the **Pins** tab.
- Writing the supporting user code required to open the driver and to process the data.

In the Thermostat application, rather than using the internal temperature sensor of the S7G2 MCU, it is most likely that an external temperature sensor is connected. If the temperature sensor provides an analog output that can be connected to one of the various input channels of the ADC, you can use any of the external temperature sensors that provides a digital interface. In that case, you should connect the analog output to the sensor and read data using any of the available communication methods such as SPI, I²C, and CAN. Read the temperature directly from the sensor module as the internal ADC unit is not needed.

For demonstration purposes on the DK-S7G2 board, the internal temperature sensor was selected to allow for a changing temperature display and to provide some levels of detail on how to use the ADC unit.

6.6.2 Accessing the ADC Unit

You can access the ADC unit in the temperature thread by adding the appropriate module to the thread. In this example, add the `g_adc` module. The ADC unit on the S7G2 MCU supports three different modes of operation:

- **Single scan mode** - Convert the analog inputs of selected channels in ascending order of the channel number
- **Continuous scan mode** - Sequentially convert the analog inputs of selected channels continuously in ascending order of the channel number
- **Group scan mode** - Divide the analog inputs of channels into two groups (Group A and Group B) and convert the analog inputs of selected channels for each group in ascending order of the channel number.

In more complex applications, the continuous scan mode or group scan mode might make sense for converting multiple analog channels and having the application framework with data already buffered.

Note: One of the modules you can add to a thread is `SF_ADC_Periodic` which also requires a timer driver (`r_gpt`) and a data transfer mechanism (`r_dtc`).

Because only reading a single analog channel is read, the temperature sensor at a slow rate, one of the most basic ways to access the ADC is used in this application note. In the following figure, the only module added to the temperature thread is the ADC driver module `g_adc ADC Driver` on `r_adc`. As is the case with most drivers in the Synergy Framework, you can configure various ADC driver parameters using the **Properties** tab.

Note: When adding modules to a thread, the ISDE turns the module outline red if it detects conflicts. As an example, if you are to set the mode of the `g_adc` module to continuous scan while defining a callback name, the outline turns red. Hovering over the box produces a warning tip that tells you the callback must be null in continuous mode.

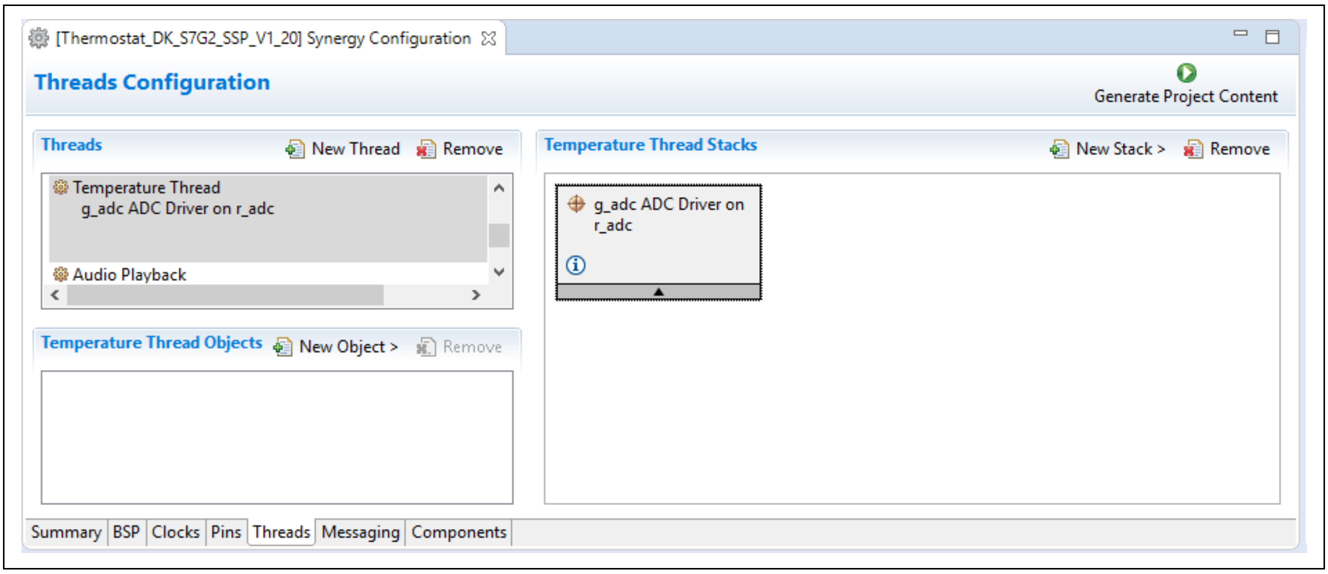


Figure 61. Temperature Thread with Only One Module

Because a large portion of the properties page for the ADC driver module is the numerous channels you can select from, the entire properties page is not reproduced in the example. The key properties to note are shown in the following table.

Table 3. Key ADC Properties to Configure

ICU	ADC1 SCAN END	Priority 3
	ADC1 SCAN END B	Priority 3
Module	Mode	Single Scan
	Channel Scan Mask	Temperature Sensor – Use in Normal/Group A

The `temperature_thread_entry.c` file is the code that runs inside the temperature thread. In summary, the code initializes the `g_adc` driver.

```
void temperature_thread_entry(void)
{

    /** Configure the temperature sensor **/
    ssp_err_t err = g_adc.p_api->open(g_adc.p_ctrl, g_adc.p_cfg);
    err += g_adc.p_api->scanCfg(g_adc.p_ctrl, g_adc.p_channel_cfg);
    err += g_adc.p_api->sampleStateCountSet(g_adc.p_ctrl,
&adc_sample_state);
```

Inside the main loop of the thread, the code starts an A/D scan, sleeps then reads the A/D. So essentially, this code is polling the A/D using the `tx_thread_sleep()` call to determine the polling rate.

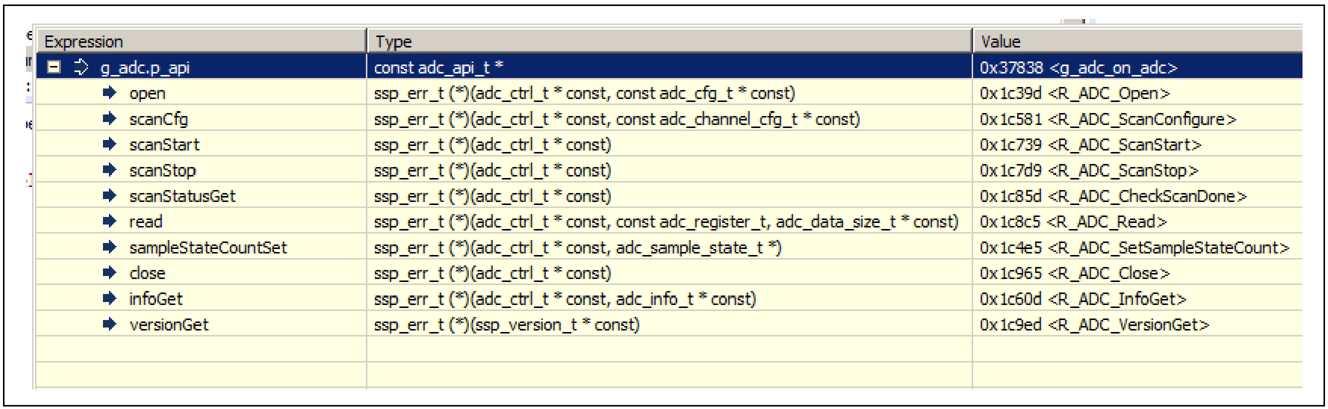
```
    while(1)
    {
        /** Start the ADC conversion*/
        g_adc.p_api->scanStart(g_adc.p_ctrl);

        /** Only update every 100 ms. */
        tx_thread_sleep(10);

        /** Read the raw ADC value. */
        g_adc.p_api->read(g_adc.p_ctrl, ADC_REG_TEMPERATURE,
&adc_data[index]);
```

The remaining code averages the values read from the A/D. If the value changes, the code sends a message to the system thread to update the current temperature.

The e² studio provides quick API information you can reference while working in the code. Hovering over the `p_api` element of the `g_adc` instance produces the following tool tip as shown in the following figure.



Expression	Type	Value
g_adc.p_api	const adc_api_t *	0x37838 <g_adc_on_adc>
open	ssp_err_t (*)(adc_ctrl_t * const, const adc_cfg_t * const)	0x1c39d <R_ADC_Open>
scanCfg	ssp_err_t (*)(adc_ctrl_t * const, const adc_channel_cfg_t * const)	0x1c581 <R_ADC_ScanConfigure>
scanStart	ssp_err_t (*)(adc_ctrl_t * const)	0x1c739 <R_ADC_ScanStart>
scanStop	ssp_err_t (*)(adc_ctrl_t * const)	0x1c7d9 <R_ADC_ScanStop>
scanStatusGet	ssp_err_t (*)(adc_ctrl_t * const)	0x1c85d <R_ADC_CheckScanDone>
read	ssp_err_t (*)(adc_ctrl_t * const, const adc_register_t, adc_data_size_t * const)	0x1c8c5 <R_ADC_Read>
sampleStateCountSet	ssp_err_t (*)(adc_ctrl_t * const, adc_sample_state_t *)	0x1c4e5 <R_ADC_SetSampleStateCount>
close	ssp_err_t (*)(adc_ctrl_t * const)	0x1c965 <R_ADC_Close>
infoGet	ssp_err_t (*)(adc_ctrl_t * const, adc_info_t * const)	0x1c60d <R_ADC_InfoGet>
versionGet	ssp_err_t (*)(ssp_version_t * const)	0x1c9ed <R_ADC_VersionGet>

Figure 62. Useful e² studio Tool Tips Showing API Calls

6.7 Weak Callback Functions and the g_adc

After adding a driver module that can make use of a callback function, as is the case with the `g_adc` module, the application framework defines a callback function using a weak attribute. The weak attribute, associated with GNU and other compilers, is a method that allows you to override a system-defined function. The following is an excerpt from the auto-generated `temperature_thread.c` file defining the weak internal `adc_callback_internal()` function. This call is called by the ADC whenever a scan completes. The default implementation in the application framework has a `/** Do nothing **/` comment.

```
#if ADC_CALLBACK_USED_g_adc
#if defined(__ICCARM__)
#define adc_callback_WEAK_ATTRIBUTE
#pragma weak adc_callback                            =
adc_callback_internal
#elif defined(__GNUC__)
#define adc_callback_WEAK_ATTRIBUTE       __attribute__ ((weak,
alias("adc_callback_internal")))
#endif
void adc_callback(adc_callback_args_t * p_args)
adc_callback_WEAK_ATTRIBUTE;
#endif


#if ADC_CALLBACK_USED_g_adc
/**********************************************************************
******************************************//**
 * @brief      This is a weak example callback function.  It should be
overridden by defining a user callback function
 *             with the prototype below.
 *               - void adc_callback_internal(adc_callback_args_t *
p_args)
 *
 * @param[in]  p_args  Callback arguments used to identify what caused the
callback.

**********************************************************************
********************************************/
void adc_callback_internal(adc_callback_args_t * p_args)
{
```

```
        /** Do nothing. */
    }
```

```
You may override the callback function in your code. For example, you can define
a callback function in the system_thread_entry.c file that reads the temperature
sensor whenever the A/D finishes a scan and places the data inside a variable
called adc_read_data.
```

```
    void adc_callback(adc_callback_args_t * p_args) {
        g_adc.p_api->read(g_adc.p_ctrl, ADC_REG_TEMPERATURE, &adc_read_data);
        return;
    }
```

The main thread code can then pick up this value whenever it wakes up, process it, and call the `scanStart()` API call again before going back to sleep.

7. Importing a Project into e^2 studio

Refer to the *Renesas Synergy™ Project Import Guide* (r11an0023eu0121-synergy-ssp-import-guide.pdf) for instructions on importing the project into e^2 studio and build/run the project.

8. Reloading the Demonstration Program

Refer to the *DK-S7G2 v3.0 Out-of-Box Demonstration Programming Guidelines* (r12an0024eu0129-synergy-dk-s7g2-oob.pdf) for instructions on reloading the Out-of-Box demonstration onto the DK-S7G2 Synergy MCU Group board, included in this kit. Similar guides are available for each of the boards that supports running the Thermostat application.

9. Known Issues

Each `GX_EVENT_CLICKED`, sends `GX_EVENT_KEY_DOWN` event to parent and then parent routes it to selected single line text input widget. With GUIX 5.3.3, if you press a button with auto-repeat, then move your finger outside of the button boundaries while still holding your finger pressed on the screen and then release it, the button will stay locked in the auto-repeat state and generates `GX_EVENT_CLICKED` periodically despite the screen being released. The only way to restore proper behavior is to press and release the button again.

10. Reference Documents

You can download the required Renesas software and documents from the Renesas Synergy™ Gallery at www.renesas.com/synergy/software

- *Synergy Software Package User's Manual (SSP)* (available in **HTML**).
- *ThreadX® User's Manual*: Express Logic Inc.
- *Renesas Synergy™ Project Import Guide* (r11an0023eu0121-synergy-ssp-import-guide.pdf)
- *DK-S7G2 v3.0 Out-of-Box Demonstration Programming Guidelines* (r12an0024eu0129-synergy-dk-s7g2-oob.pdf).

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	Sep.18.17	—	Initial release for v1.3.0
1.01	Nov.17.17	—	Bug fix for splash screen and date configuration, and updated software versions
1.02	Jan.18.18	—	Updated for SSP v1.3.3
1.03	Mar.02.18	—	Updated for SSP v1.4.0
1.04	Jun.18.18	—	Sample codes updated.
1.05	Sep.24.18	—	Updated for SSP v1.5.0
1.06	Mar.15.19	—	Updated for SSP 1.6.0

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/.