Renesas RA Family

Develop Graphics Applications Using the MIPI DSI Interface on the RA8D1

Introduction

This application note describes the development of a graphics application that targets the RA8D1 MCU and runs on the external MIPI display included with the RA8D1 evaluation kit (EK). The application note covers a variety of graphics-related topics, including:

- Interfacing with peripherals using the MIPI options on the RA8D1.
- Designing a GUI for the EK-RA8D1 LCD panel using the SEGGER AppWizard design tool.
- Integrating the AppWizard project with an e² studio project using the RA Flexible Software Package (FSP), emWin, and FreeRTOS.
- Communicating with the EK-RA8D1 LCD's capacitive touch panel using custom-written drivers.
- Consider design tradeoffs for graphics applications in the context of the RA8D1 MCU architecture to determine the best-case design for your system needs.

The discussion in the application note is supported by an accompanying application project that demonstrates the above topics in a single-use case. The application project is a multi-threaded, portrait-orientation graphics application that runs on the EK-RA8D1 and interfaces as a MIPI DSI-2 Host to the external LCD panel. The application's Graphical User Interface (GUI) was designed using the SEGGER AppWizard tool, and it is integrated into an e² studio project for the EK-RA8D1, using the RA Flexible Software Package (FSP) that natively supports emWin and FreeRTOS.

Target Device

EK-RA8D1

Required Resources

To build and run the Thermostat graphics application accompanying this application note, you will need the following resources:

Development tools and software:

- e² studio IDE, version 2025-10
- RA Family Flexible Software package (FSP) version 6.2.0
- AppWizard version V1.56_6.48

Note: The FSP emWin version must match the Segger AppWizard/emWin version. The FSP emWin version is denoted differently than Segger: e.g., FSP emWin v6.44.2 correlates to Segger's emWin v6.44b and Segger's AppWizard v1.52_6.44b, and so on.

The FSP and e² studio are bundled in a downloadable platform installer available on Renesas' website at: www.renesas.com/ra/fsp

The AppWizard GUI Design Tool can be downloaded from SEGGER's website:
https://www.segger.com/products/user-interface/emwin/tools/appwizard/

Or: SEGGER emWin GUI Library for Renesas RA Products | Renesas

Hardware

- Renesas EK-RA8D1 kit (RA8D1 MCU Device Group): www.renesas.com/ek-ra8d1

Prerequisites and Intended Audience

This application note assumes you have some experience with the Renesas e² studio IDE and RA Family Flexible Software Package (FSP). Before you perform the procedures in this application note, follow the procedure in the FSP User Manual to build and run the Blinky project. Doing so enables you to become familiar with e² studio and the FSP and validates that the debug connection to your board functions properly. Additionally, this application note assumes that you have a theoretical background in graphics applications. The intended audience is users who want to develop graphics applications on the RA8D1 MCU Device Group.

Contents

1. The RA8D1 Device Group 4
1.1 Key Graphics Features of the RA8D1 4
1.2 RA8D1 Evaluation Kit 4
1.2.1 EK-RA8D1 MIPI Graphics Expansion Board 5

2. MIPI DSI Options on the RA8D1 6
2.2 MIPI Design Guidelines 8
2.2.1 Clock Configuration 9
2.2.2 Pin Configuration 9
2.2.3 MIPI Operating Modes 9
2.3 Operating MIPI in e^2 studio 10
2.3.1 Adding MIPI modules to a project 10
2.3.2 Handling MIPI in software 11

3. AppWizard GUI Design Tool 12
3.1 AppWizard and emWin Capabilities 12
3.2 Creating a New AppWizard Project 12
3.2.1 Including AppWizard Project Files in e^2 studio Project 13
3.2.2 AppWizard/emWin Initialization 15
3.3 Custom Designs in AppWizard 16
3.4 Setup AppWizard Interactions 16
3.4.1 AppWizard Defined Interactions 16
3.4.2 User-Defined Slot Code 17
3.4.3 Responding to AppWizard Variables 18
3.5 Add emWin Widget to AppWizard Project 19

4. The Thermostat Application 20
4.1 Source Code Layout 20
4.2 Application Block Diagram 21
4.3 Threads 22
4.3.1 emWin Thread 23
4.3.2 Touch Thread 24
4.3.3 Timer Thread 25
4.4 Configuring the EK-RA8D1’s MIPI LCD 25
4.5 GT911 Touch Drivers 26
4.6 FSP Configuration 28
4.6.1 Components: 29
4.6.2 Thread Objects 30
4.6.3 Module, Pin and Clock Configuration 30

5. Running the Thermostat Application 33
5.1 Hardware Setup 34

5.2 Import, Build, and Run 35

6. Graphics Tradeoffs on the RA8D1 36
6.1 MIPI DSI vs Parallel RGB 36
6.2 Graphics Configuration Tradeoffs 38
6.2.1 Display Resolution 38
6.2.2 Color Format 38
6.2.3 Framerate 40
6.2.4 Bus Width 40
6.2.5 Internal SRAM 41
6.3 RA8D1 Memory Options 41
6.3.1 EK-RA8D1 Memory Devices 42
6.4 Thermostat Application Best Case Design 43

7. References 46

Revision History 48

1. The RA8D1 Device Group

The MCUs in the RA8D1 device group are available in 176- and 224-pin packages. Secure Element-like functionality is built in with advanced cryptographic Security IP, immutable storage for the first-stage bootloader (FSBL), a secure boot, and tamper protection for a truly secure IoT device.

1.1 Key Graphics Features of the RA8D1

The RA8D1 MCUs integrate the high-performance CM85 core with large memory, and they hold a rich peripheral set including a high-resolution TFT-LCD controller with parallel RGB and MIPI-DSI interfaces, 2D drawing engine, 16-bit camera interface, and multiple external memory interfaces like SDRAM and OSPI, which are optimized to address the needs of diverse graphics and Vision AI applications.

The high-resolution graphics LCD controller (GLCDC) can support displays up to 1280x800 WXGA and supports both parallel and MIPI-DSI interfaces. The 2D drawing engine (DRW) offloads the graphics rendering from the CPU, and it can support graphics primitives like lines and polygons as well as functions like alpha blending, rotation/scaling, and color conversions. The 16-bit camera interface (CEU) has support for image data fetch, processing, and format conversion, and it can interface to camera sensors up to 5M pixels. The on-chip 1MB of SRAM can fit single or dual frame buffers for resolutions of 800x480 WVGA or smaller. The external SDRAM supports framebuffers for higher resolutions that cannot fit into SRAM. The xSPI-compliant OSPI interface has XIP and DOTF support for the secure storage of graphics assets.

RA8D1 MCUs are fully supported by the Flexible Software Package (FSP) to provide easy-to-use, scalable, high-quality software for embedded system design. Along with the FSP, there is a comprehensive set of hardware and software tools to assist with embedded application development.

SEGGER's AppWizard GUI design tools assist in creating highly efficient and high-quality graphical user interfaces targeting any RA MCU. The FSP contains the corresponding middleware, the emWin graphics library, facilitating the smooth development of embedded graphics applications. The RA emWin Port middleware layer is regularly tested with each FSP release, ensuring that it fully supports the operation of the emWin library on RA devices.

1.2 RA8D1 Evaluation Kit

The EK-RA8D1, an evaluation kit for the RA8D1 MCU Group, enables users to evaluate the features of the RA8D1 MCU Group and develop embedded systems applications using Renesas' Flexible Software Package (FSP) and e^2 studio IDE.

The kit consists of three boards (and required connections): the EK-RA8D1 board featuring the RA8D1 MCU with on-chip graphics LCD controller, a MIPI graphics expansion board featuring a 4.3-inch TFT color portrait LCD with a capacitive touch panel overlay, and a camera expansion board featuring a 3M pixels CMOS image sensor.

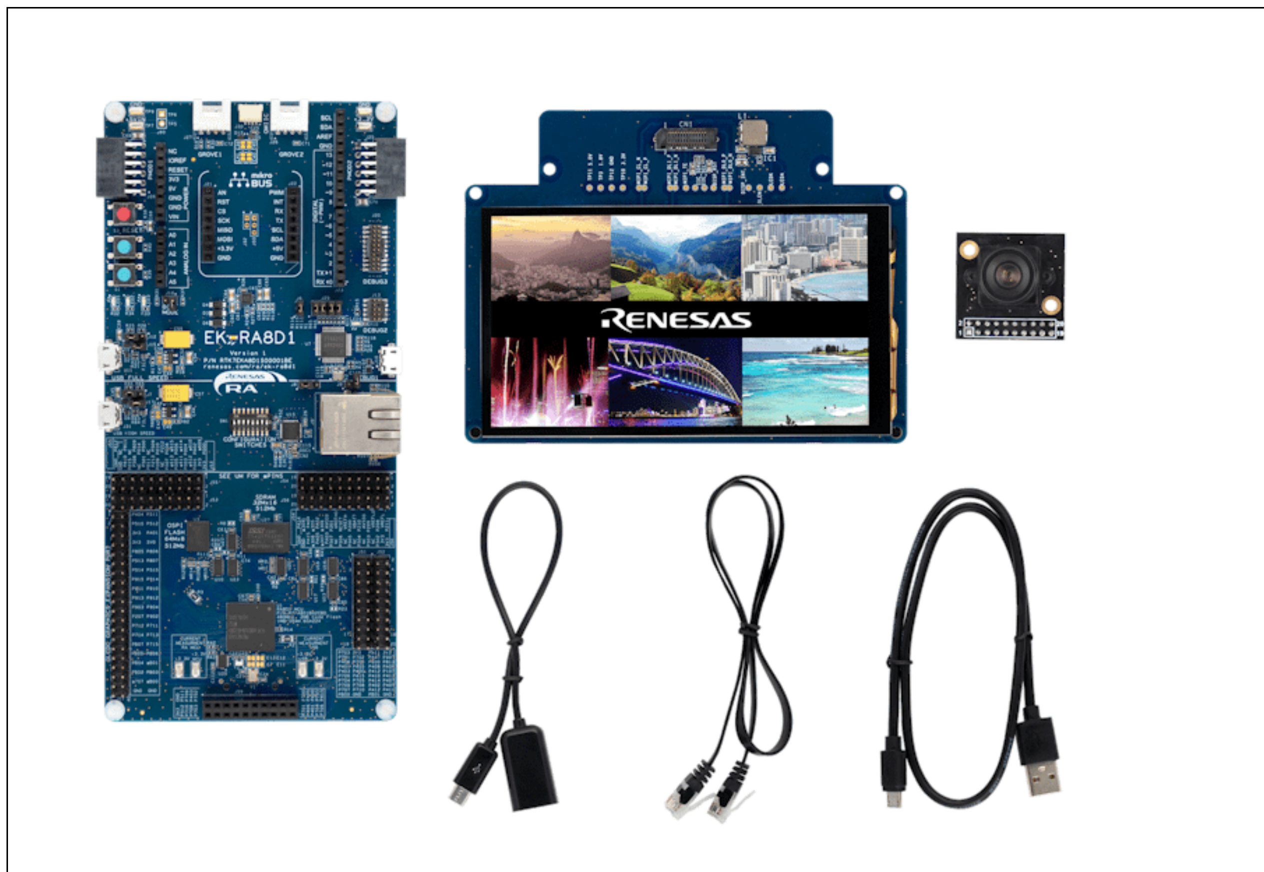

Figure 1. Contents of the RA8D1 evaluation kit

The EK-RA8D1 board comes pre-programmed with a Quick Start example project. Please refer to the EK-RA8D1 Quick Start Guide for instructions on importing, modifying, and building the Quick Start example project.

For more examples demonstrating the operation of the modules on the EK-RA8D1, check out the EK-RA8D1 Example Projects Bundle document, which can also be found on the Renesas website.

1.2.1 EK-RA8D1 MIPI Graphics Expansion Board

Some of the key features of the TFT LCD used on the MIPI Graphics Expansion Board are as follows:

- Display type: TFT LCD with capacitive touch panel (CTP) overlay and backlight control
- Diagonal size: 4.5 inch
- Dimensions: 61.54x110.09 mm
- Resolution: 480x854 pixels
- Touch mode: up to 5 points
- LCD panel controller IC: ILI9806E
- CTP controller IC: GT911

The part number of the TFT LCD used on the MIPI Graphics Expansion Board is E45RA-MW276-C. For more details and to view the LCD's specification sheet, please visit focusLCDs.com.

The LCD and the CTP on the MIPI graphics expansion board are separate and use different controller ICs. The LCD panel uses the ILI9806E controller, and the CTP uses the Goodix GT911 controller. Please refer to the respective data sheets /specification sheets in the reference section at the end of the application note to understand the operation of each controller.

Since the EK-RA8D1's MIPI graphics expansion board is portrait orientation, this application note will only cover designing and drawing a portrait orientation application.

2. MIPI DSI Options on the RA8D1

The graphics subsystem of the RA8D1 features the following hardware modules:

- Graphics LCD Controller (GLCDC): 8/16/18/24-bit RGB LCD interface supported.
- 2D Drawing Engine (DRW): Raster and vector graphics supported.
- MIPI D-PHY controller (MIPI PHY): Used to control MIPI DSI.
- MIPI Display Serial Interface (MIPI DSI): Display Serial Interface 2 supported.

The high-resolution graphics LCD controller (GLCDC) can support displays up to 1280x800 WXGA and has both parallel RGB and MIPI-DSI interfaces, providing the flexibility to drive both widely used RGB external displays and newer high-quality MIPI external displays.

The block diagram below illustrates the connections between the modules in the graphics subsystem of the RA8D1. Note how the LCD_DATA framebuffer data output from the GLCDC routes to parallel RGB output pins on the MCU and to the input of the MIPI DSI module of the MIPI subsystem, where the signal is prepared for output via the D-PHY and connections to the display.

Another important feature in the block diagram is that the same bus interfaces with all the graphics modules: MIPI PHY, DRW, MIPI DSI, and GLCDC. Keep this in mind when evaluating bandwidth and bottlenecks of your own graphics application.

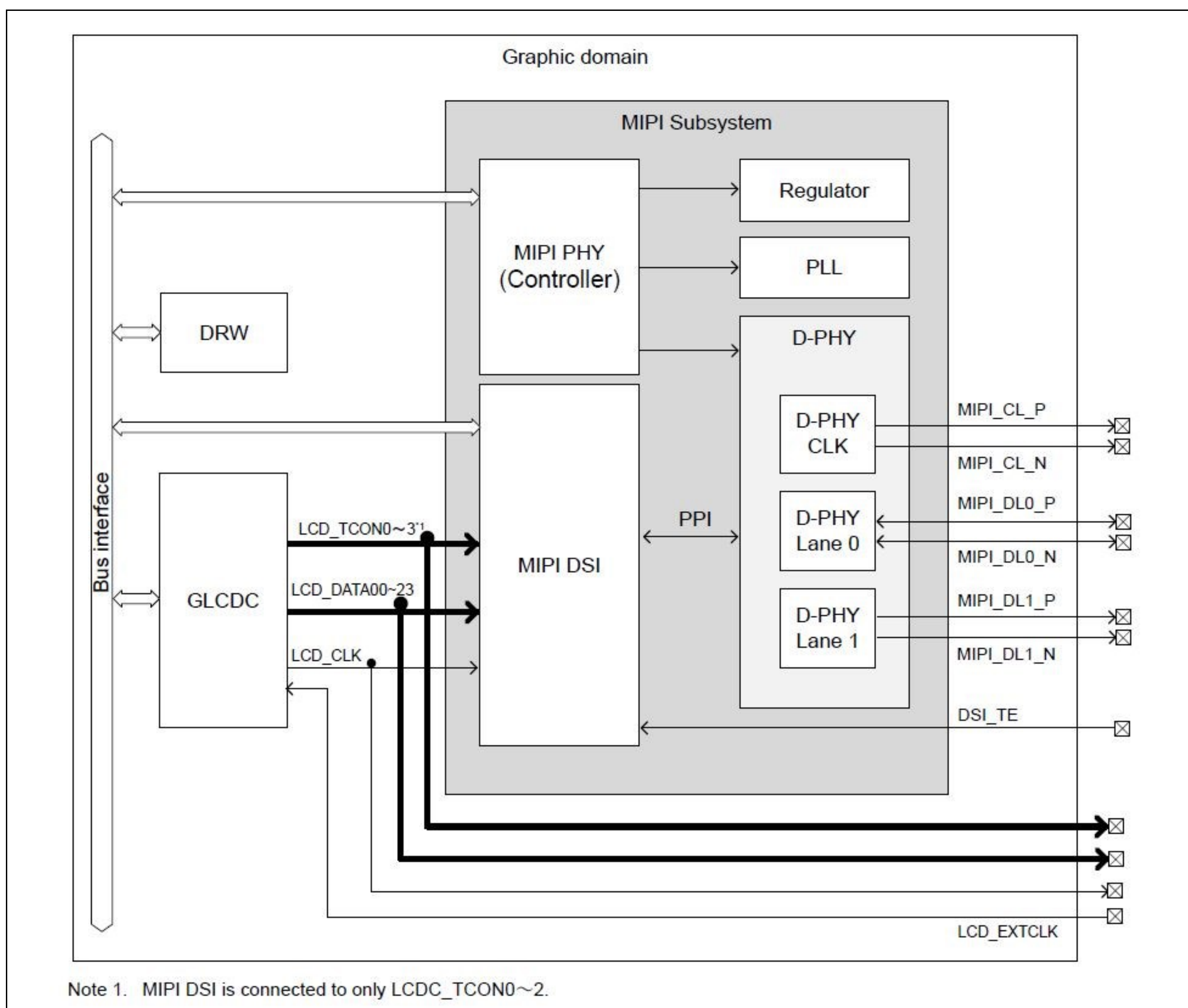

Figure 2. Block diagram of the RA8D1 graphics subsystem

The MIPI subsystem IO pins from the block diagram are explained in the table below:

Table 1. MIPI I/O Pin Functions

Pin Name	I/O	Function
MIPI_CL_P	Output	DSI Clock Lane positive pin
MIPI_CL_N	Output	DSI Clock Lane negative pin
MIPI_DL0_P	I/O	DSI Data Lane 0 positive pin
MIPI_DL0_N	I/O	DSI Data Lane 0 negative pin
MIPI_DL1_P	Output	DSI Data Lane 1 positive pin
MIPI_DL1_N	Output	DSI Data Lane 1 negative pin
DSI_TE	Input	DSI Tearing Effect pin
AVCC_MIPI	Power	D-PHY Analog Power*
VCC18_MIPI	Power	D-PHY I/O Power*
VSS_MIPI	Power	D-PHY Ground

*Connect these pins to VSS_MIPI by a 0.1µF capacitor. The capacitor should be placed close to the pin.

(1) MIPI PHY

The MIPI PHY on the RA8D1 MCU is a MIPI D-PHY module that follows the MIPI Alliance Specification Version 2.1 for D-PHY Specification. The following table summarizes the specifications:

Table 2. MIPI PHY Specifications

Parameter		Specifications
D-PHY	Number of lanes	Up to 2 Lanes
	Maximum rate	720 Mbps / Lane
MIPI PLL circuit	Input clock source	MOSC
	Input pulse frequency division ratio	Selectable from 1/2/3/4
	Input clock frequency	8 MHz to 48 MHz
	Input clock frequency (After DPHYPLFCR.IDIV[1:0] bits	5 MHz to 24 MHz *1
	Frequency multiplication ratio	Selectable from 20 to 180 (after the decimal point : 0/0.33/0.50/0.66)
	VCO frequency	160 MHz to 1440 MHz
	PLL Output clock	160 MHz to 1440 MHz

Note:
Note 1. . DPHYPLFCR.IDIV[1:0] must be set so that the frequency of the clock divided by DPHYPLFCR.IDIV[1:0] is within the range that is between 8 to 24 MHz. However, under the condition of 160 MHz ≤ PLL output clock (fDPHYPLL) < 320 MHz, the allowable frequency range of the clock divided by IDIV[1:0] is extended to 5 to 24 MHz.

In order to use the MIPI DSI function, the D-PHY must first be activated. The MIPI DSI D-PHY configuration controls timing aspects of Low Power (LP) and High Speed (HS) communication. Configure these values to match the specifications listed in the display datasheet. Recommended timing tables based on PCLKA settings can be found in section 57.2.13 of the RA8D1 Group Hardware User's Manual.

(2) MIPI DSI

The MIPI DSI on the RA8D1 MCU is a MIPI DSI-2 Host module, and it supports the MIPI Alliance Specification for Display Serial Interface 2 (DSI-2) Specification. The following table summarizes the available configuration parameters and their specifications for MIPI DSI on the RA8D1:

Table 3. MIPI DSI Specifications

Parameter		Specifications
Compliant specification		• MIPI Alliance Specification for Display Serial Interface 2 Version 1.1 (with Errata 01) • MIPI Alliance Specification for D-PHY Version 2.1 (with Errata 01) (80 Mbps to 720 Mbps/Lane and up to 2 Lanes).
Video Mode Operation	Available input video format from GLCDC	• Parallel RGB888 (24 bits), little endian • Parallel RGB666 (18 bits), little endian • Parallel RGB565 (16 bits), little endian
	Available output format	• RGB (16 bits, 18 bits, 24 bits)
	Available video mode packet sequence	• Non-Burst Mode with Sync Pulse • Non-Burst Mode with Sync Event • Burst Mode
	Others	• Selectable Blanking Packet or LP-11 during each of blanking interval of HSA, HBP, and HFP
Command Mode Operation*1	Sequence Operation Channel-0	LP only packet generation and LP packet reception from descriptor list
	Sequence Operation Channel-1	HS or LP packet generation and LP packet reception from descriptor list
DSI Link support functions		• 1 and 2 Lane configurations • Unidirectional High-Speed mode transfer (HS-TX) • Bidirectional LP mode transfer/receipt (LP-TX / LP-RX) (Only Lane 0) • ECC/Checksum generation for Tx packet • ECC/Checksum verification and ECC error correction for Rx packet • Ultra-Low-Power mode (ULPS) • Automated power change to LP mode and return to HS mode • Automated clock stop and resume (non-continuous clock mode) • Assignment for Virtual Channel in video mode • Assignment for individual Virtual Channel for each packet in Command mode • Detection for PHY contention error and timeout error • Generation of scrambled packets • Input of TE signal
Module-stop function		Module-stop state can be set to reduce power consumption.
TrustZone Filter		Security and Privilege attribution can be set.

Note 1. Ch0 and Ch1 are exclusive.Cannot be used simultaneously.

As supported by the information in the above table, the MIPI DSI interface on the RA8D1 is highly configurable, the communication adheres to the MIPI standards, and access to the peripheral can be secured using Trustzone.

The module's configurability encompasses handling varying bit depths of input from GLCDC and output to the MIPI pins, handling either one or two-lane configs, among other DSI link support functions. The module interfaces using the standardized MIPI Alliance specifications to enable scalable, high-performance, and low-power communications that are used widely in today's industry's displays. A TrustZone filter option with security and privilege attribution options provides an additional layer of security.

2.2 MIPI Design Guidelines

This section serves as a beginning guide to operating MIPI in software with helpful tips, configurations, and restrictions to consider. This is NOT a specification; please refer to the RA8D1 Hardware User's Manual and the MIPI Alliance specifications for DSI and D-PHY concurrently for the complete specifications for operating the MIPI PHY and MIPI-DSI modules.

2.2.1 Clock Configuration

The MIPI DSI D-PHY has a dedicated regulator (D-PHY LDO) and PLL (D-PHY PLL), which are managed by the driver. The D-PHY PLL frequency must be configured between 160 MHz and 1.44 GHz.

The D-PHY High-Speed data transmission rate is determined by the following formula:

Line rate [Mbps] = fDPHYPLL [MHz] / 2

2.2.2 Pin Configuration

Communication to the external display occurs via one or more data lanes and one clock lane. Each of these lanes has dedicated pins. Lane 0 is capable of low-power data transfer and bidirectional communication with a display. Lane 1 is capable of low-power or high-speed data transfer to the external display. Additionally, an optional tearing effect connection (DSI_TE) may be used with this module.

2.2.3 MIPI Operating Modes

The DSI-2 Host supports two basic types of operations: Command Mode and Video Mode.

As defined in the Video Mode section, note that the GLCDC Video Clock Bandwidth must not exceed the data bus bandwidth or MIPI PHY PLL Bandwidth.

2.2.3.1 Command Mode

The command mode of this DSI host is typically used for control communication, such as configuring the settings for the peripheral before sending video signals. A sequence of descriptors is sent to the target peripheral to specify the sending and receiving process in advance. Command mode for the MIPI DSI Host supports both package transmission and reception to and from the peripheral device.

Command mode can also be used to update display data in low-power operations in scenarios where frequent stream updates are not necessary. When using command mode to display pixel information, external displays should have onboard memory for the framebuffer, so the image can be written to and read from memory on the LCD by sending the display-specific commands from the MCU.

Two internal channels, available for all physical lane configurations, may be used for command mode operations. Channel-0 supports only Low-Power (LP) mode (LP-TX, LP-RX), while Channel-1 supports LP mode (LP-TX, LP-RX) and HS mode (HS-TX). While a data lane is in LP operation, command mode may be used for bidirectional communication with a connected display using a pre-defined set of command descriptors.

There are two basic packet formats: short and long. Each format may be transmitted in high-speed or low-power modes. Packets may be followed by a Bus Turn-Around (BTA) request to read information from the display. Once configured and started, video packets are transmitted automatically until video output is stopped.

In addition to the full set of MIPI DSI commands, the application may trigger any of four special commands by setting flags in the message structure. These special commands are Reset Signal, Initial Skew Calibration, Periodic Skew Calibration, and No-Operation.

Note that for peripherals with more than one lane, physical lane 0 is used for all peripheral-to-processor transmissions. Other lanes are unidirectional, from the host processor to the peripheral.

2.2.3.2 Video Mode

Video mode is optimized for streaming large volumes of data, making it suitable for high-resolution and high-refresh-rate displays. In video mode, data is sent as a real-time pixel stream, which the host processor needs to provide and refresh continuously.

In video mode, there are three different packet sequences: Non-Burst Mode with Sync Pulse, Non-Burst Mode with Sync Event, and Burst Mode. A Sync Event specifies the start and end of a sync to represent the active area of the pixel data being sent, similar to the sync in the RGB interface. The sync events are sent in short packets that indicate the location and porch lengths. A Sync Pulse is similar to a Sync Event but requires pre-definition of the sync pulse width.

The timing requirements of the target MIPI peripheral will determine which is the right operating mode. Each operating mode is achieved by configuring the MCU’s MIPI module with specific timing and option settings. For the purposes of this section, use the following definitions:

- GLCDC Video Clock Bandwidth is defined as:

  ```
  GLCDC Video Clock Bandwidth = (Panel Clock MHz) * (Bits per pixel)
  ```

- MIPI PHY PLL Bandwidth is defined as:

```
MIPI PHY PLL Bandwidth = (MIPI Phy PLL Clock MHz / 2) * (Number of MIPI data lanes)
* 8 - (Configuration Dependent Transmission Overhead)
```

(1) Non-Burst Mode with Sync Pulse:

GLCDC Video Clock Bandwidth == MIPI Phy PLL Bandwidth

Sync Pulse is enabled (Horizontal Sync End (HSE) and Vertical Sync End (VSE) signals are transmitted)

(2) Non-Burst Mode with Sync Event:

GLCDC Video Clock Bandwidth == MIPI Phy PLL Bandwidth

Sync Pulse is disabled (HSE and VSE are not transmitted)

(3) Burst Mode:

GLCDC Video Clock Bandwidth < MIPI Phy PLL Bandwidth

Sync Pulse is disabled (HSE and VSE are not transmitted)

3

2.3 Operating MIPI in e² studio

This section will provide an overview of software guidelines for operating the MIPI DSI module on the RA8D1 MCU using the FSP and e² studio configurator.

2.3.1 Adding MIPI modules to a project

In e² studio, the "Stacks" tab of the FSP Configuration view allows you to easily add, remove, and configure your project's threads and modules.

The MIPI DSI Output is an optional sub-module of the graphics LCD controller r_glcdc. The r_glcdc module can be added to a project as a standalone module or as a sub-module of the Segger emWin RA port rm_emwin_port (which will be explained further in Section 3) depending on your project's needs. To add r_glcdc in the "Stacks" tab, use the **New Stack > Graphics** and select either **Graphics LCD (r_glcdc)** or **SEGGER emWin**. Then click the **Add MIPI DSI Output > New > MIPI Display (r_mipi_dsi)**.

The following image illustrates adding the MIPI DSI Output for both the standalone r_glcdc module and the rm_emwin_port module.

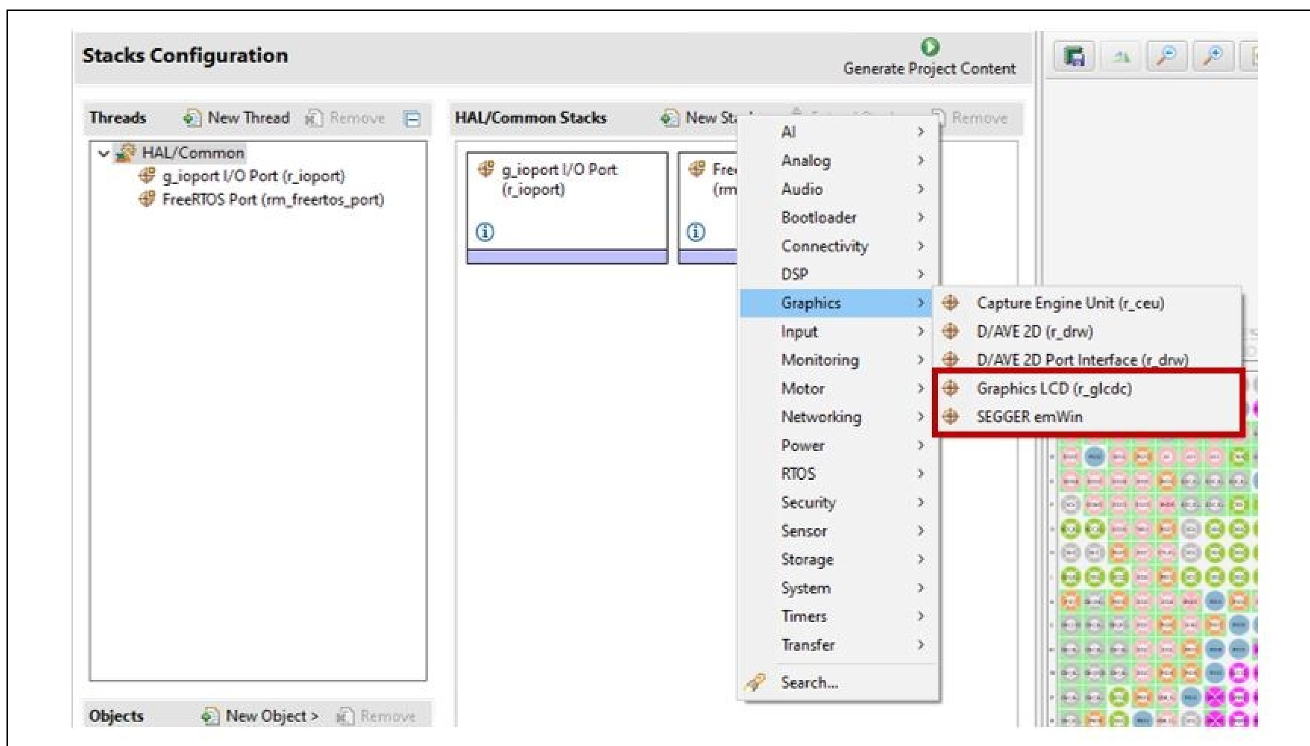

Figure 4. The r_glcdc module must be present to enable MIPI DSI output

Now that both the r_mipi_dsi and the r_mipi_phy modules have been added, you can configure each module's settings by using the "Properties" tab.

Note that the default settings for the MIPI modules are optimized for communication with the LCD display that is included with the RA8D1 evaluation kit. Detailed explanations of the additional MIPI settings are outside the scope of the app note and can be determined by analyzing your graphics system's needs and following the specifications of the RA8D1 Hardware User's Manual.

2.3.2 Handling MIPI in software

The Renesas RA Flexible Software Package (FSP) provides production-ready peripheral drivers to take advantage of the RA FSP ecosystem along with the SEGGER emWin library and FreeRTOS.

The FSP supports pre-configuration and run-time operation of the MIPI DSI and MIPI PHY peripherals under the r_mipi_dsi module with the FSP configurator and R_MIPI_DSI API driver functions, respectively. Refer to the RA Flexible Software Package User's Manual for a complete description of the available API driver functions for r_mipi_dsi.

2.3.2.1 Interrupt handling

When enabled, interrupts will invoke the configured callback function. The callback function is enabled as a default setting for the r_mipi_dsi module in the configuration.xml file in the e² studio project and will be called from the ISR each time any MIPI event occurs.

The FSP provides macro definitions of MIPI PHY and MIPI DSI events, errors, flags, and packet commands in the auto-generated project file `r_mipi_dsi_api.h` when the r_mipi_phy and r_mipi_dsi module stacks are added to an e² studio project. The definitions provided by the FSP facilitate the operation of the modules from software.

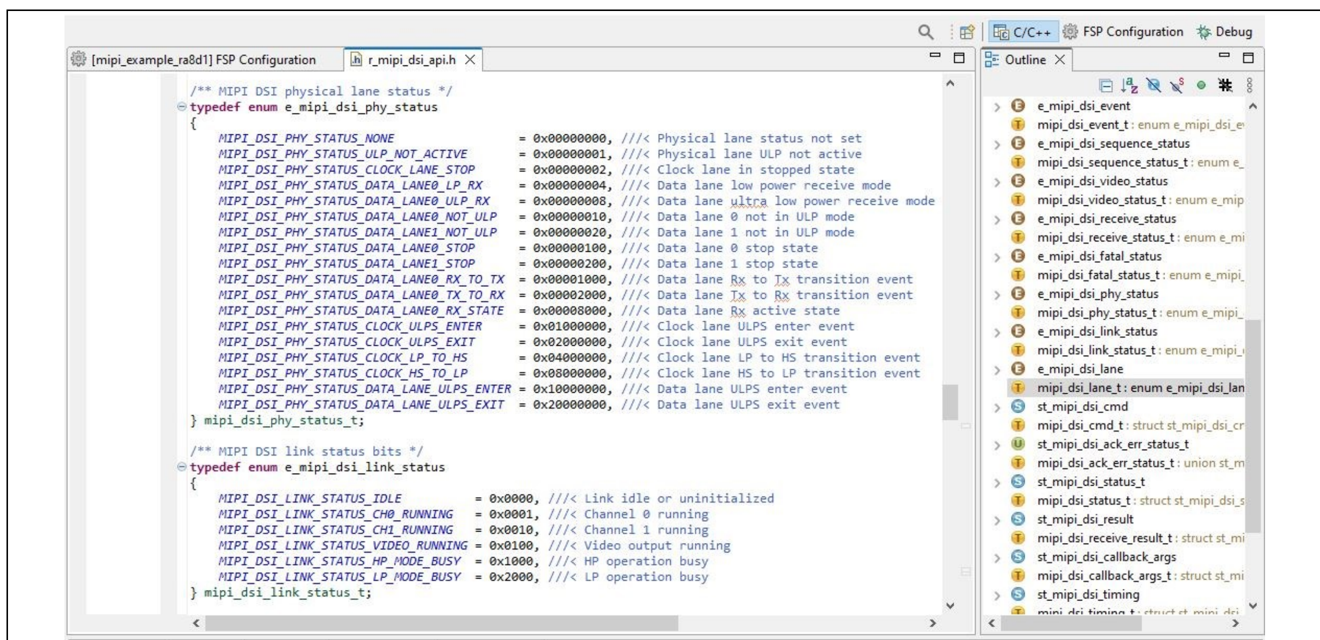

Figure 5. MIPI events. errors, flags and command definitions can be found in r_mipi_dsi_api.h

MIPI DSI on the peripheral will respond to the communication sent by the MCU by sending back data over the MIPI bus, which is captured and stored in the MIPI DSI module on the MCU. This data from the peripheral is passed to the application layer through the mipi_dsi_callback(). It's recommended that the events, errors, and flags in the callback function be serviced to transition the state of the application.

Especially for use with display middleware such as emWin or GUIX, the callback will be invoked with post-open and pre-video-start events. Depending on your target LCD's MIPI controller, it may be necessary to use these interrupt events to perform an action to configure the display. Section 4.4 of this application note gives an example of how to configure the EK-RA8D1's external MIPI graphics LCD display after the post-open event to prepare the display hardware for communication with the EK-RA8D1.

Please refer to Section 58.3.8, Error Handling in the RA8D1 Hardware User's Manual, for complete information about handling MIPI error events.

3. AppWizard GUI Design Tool

This section gives an overview of designing a GUI application with SEGGER's AppWizard software and integrating the graphical application into an FSP project with the emWin library. This section is intended to introduce the design capabilities of the software and highlight the ease of integration with an RA8D1 e^2 studio-embedded graphical application. It is not intended to replace any documentation, so please refer to the AppWizard and emWin User Guide & Reference Manuals.

3.1 AppWizard and emWin Capabilities

SEGGER's AppWizard is a graphical interface software tool for designing and creating emWin embedded graphics applications. The FSP natively supports the use of AppWizard and emWin library, making it quick and easy to get a graphics application up and running on your target RA MCU and external LCD. All the application screens in the thermostat graphics application were built using AppWizard.

You may choose to use emWin primitive calls directly in your e^2 studio project or use AppWizard to facilitate designing your screens. AppWizard is a stand-alone tool that provides a point-and-click environment for generating all the screens necessary for your embedded application. Once designed, the tool outputs "*.c" and "*.h" files to include in your e^2 studio embedded application.

The AppWizard software tool allows for full customization of each screen needed for your GUI application, including the screens' resolutions, color depths, and pixel formats, among various other parameters. AppWizard has a variety of primitives you can add to each application screen, such as boxes, buttons, images, text boxes, sliders, switches, and more.

Primitives come with standard pre-loaded fonts and basic graphics, but you can also provide the AppWizard with your own images and font files to fully customize your display. Adding definitions for AppWizard interactions will allow your application to respond accordingly to specific user inputs, variable changes, and other events. After designing a GUI application, you can preview its functionality in AppWizard, and what you see running on your PC will be displayed on your embedded screens.

Figure 6. Screenshot of the Weather Panel being designed in AppWizard

3.2 Creating a New AppWizard Project

When you open the AppWizard software, you will be prompted to create a new project or to open an existing project. You can always use the File > New Project menu option to create a new AppWizard project.

Upon creating a new project, the **Create New Project** dialog box appears and allows you to specify the project path, project name, the target's display size, and the target's pixel color format. You can always access this information to view and edit the properties by using the **Project > Edit Options** menu option.

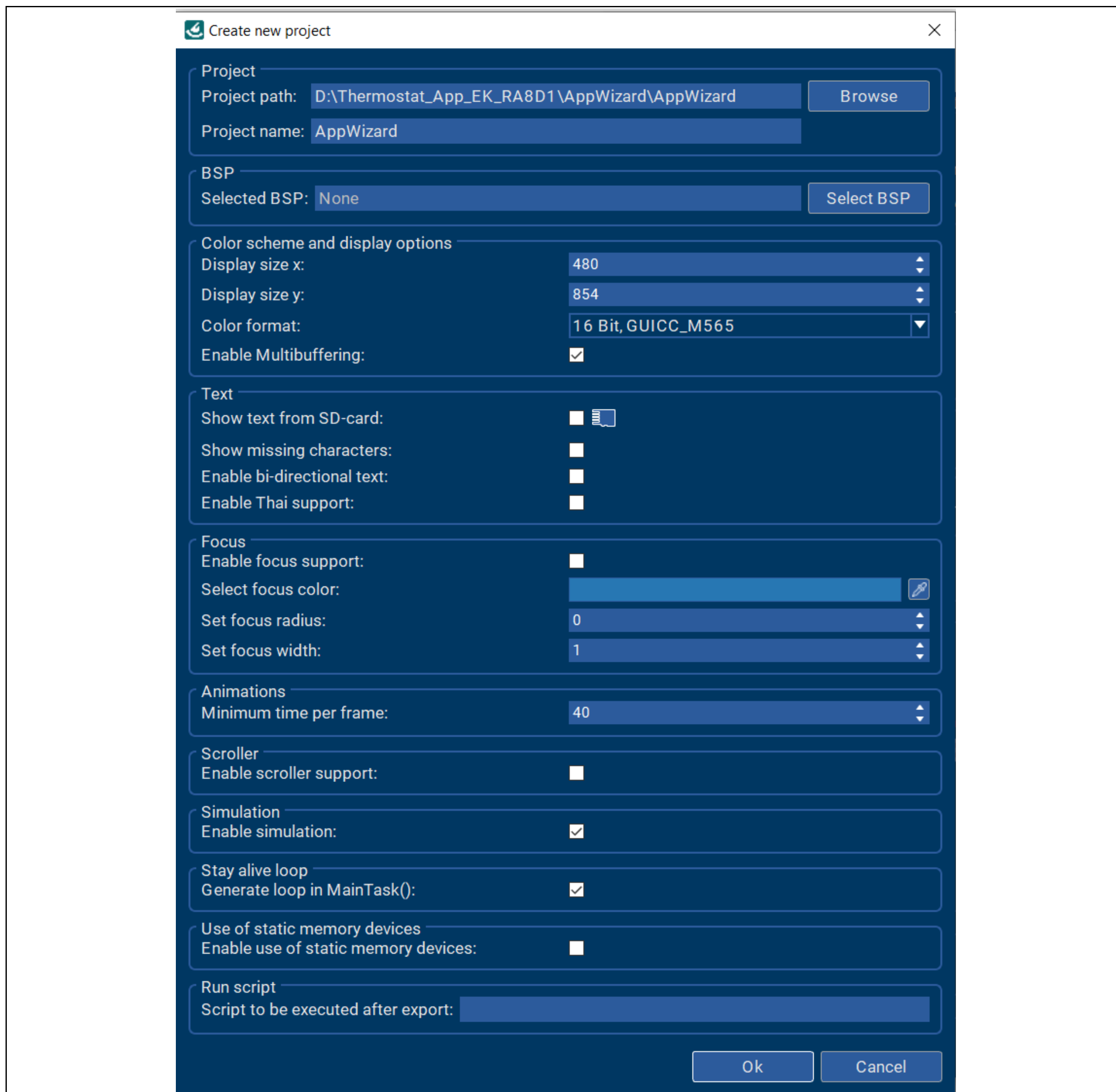

Figure 7. Screenshot of the project properties of the AppWizard thermostat GUI

The **Project Path** is the location where AppWizard will put the .c and .h files that result from the **Export & Save** process. These files contain all the information necessary to render the screens you built in AppWizard onto the LCD in your embedded graphics application. The output files are automatically organized in the project path directory with Resource, Source, and Simulation subdirectories.

3.2.1 Including AppWizard Project Files in e^2 studio Project

It's recommended that you create an e^2 studio project first and then create an AppWizard folder in the project as an e^2 studio source folder. The AppWizard folder in the e^2 studio project workspace will serve as the location of the AppWizard project path for the Resource, Source, and Simulation directory files and make it easy to move the project from one location to another or from one PC to another. In the case of the thermostat application, you can see that all directories are under the "AppWizard" folder in the e^2 studio project directory, which was also set as the **Project Path** property in the AppWizard project.

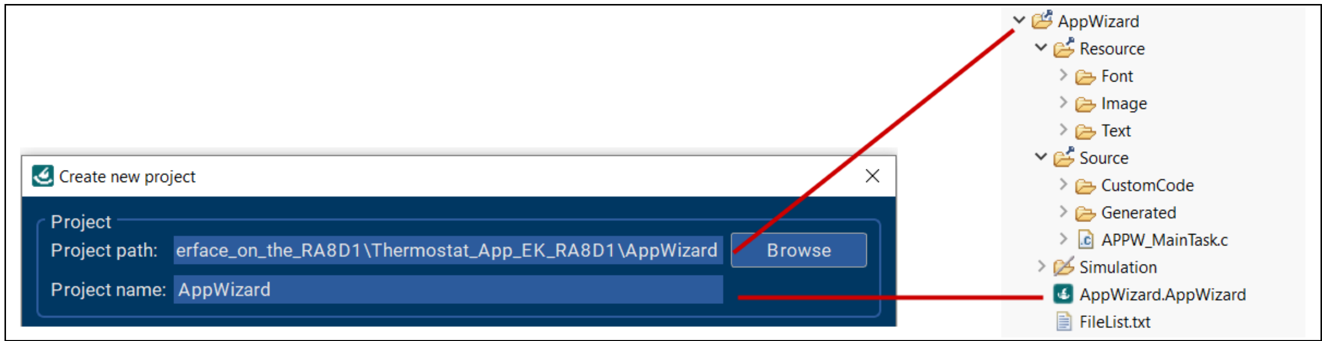

Figure 8. The AppWizard project is contained within the specified project path

As noted, the AppWizard project path folder should be configured in e² studio as a source file folder. Also, in e² studio, you need to go to **Project > Properties > C/C++ Build > Settings > GNU ARM Cross C Compiler > Includes** to add the newly created AppWizard folder and its subfolders to the project include path shown in the figure below.

Figure 9. Adding the AppWizard folder and subdirectories as e² studio project source files

After including all subdirectories, you should exclude the Simulation folder from the build before building the e² studio project. To do this, right-click on the Simulation folder in the Project Explorer. Navigate to Resource **Configurations > Exclude from Build…** and exclude the folder from both Debug and Release. Click **OK**.

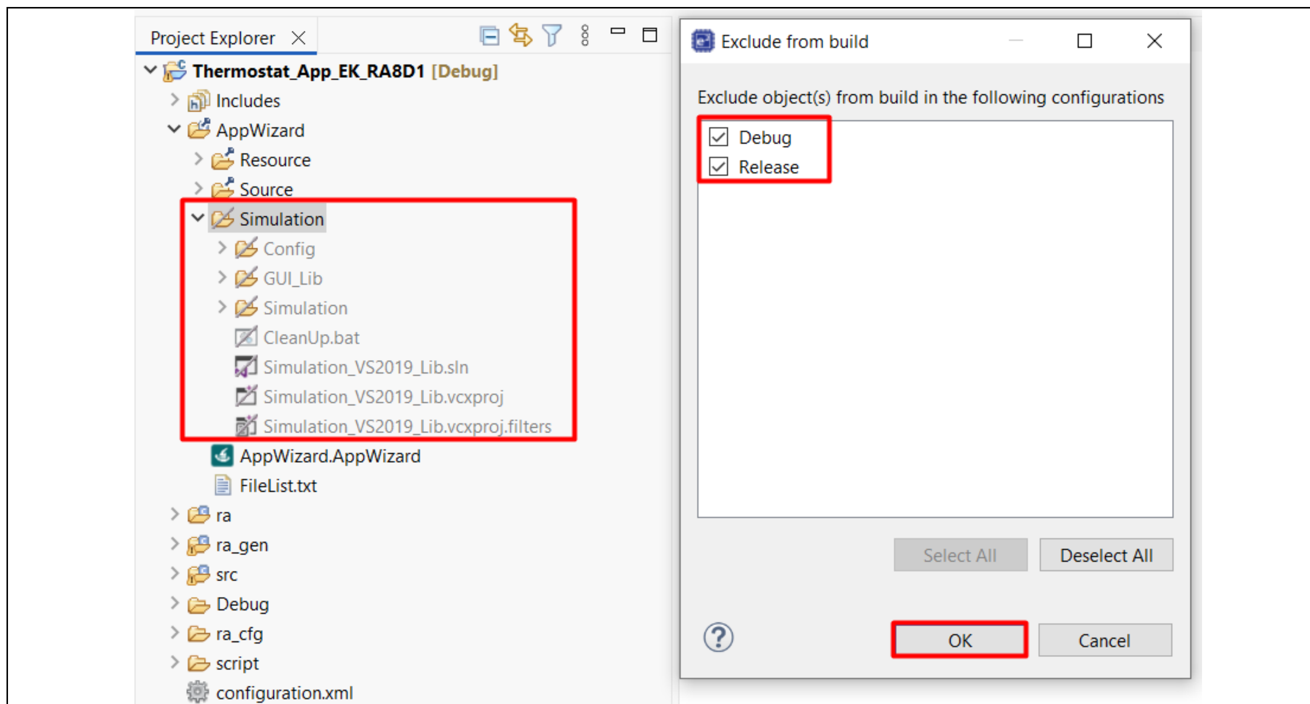

Figure 10. Exclude the Simulation folder and subdirectories for a successful build

3.2.2 AppWizard/emWin Initialization

All of the necessary emWin library and header files for the target board are generated after you finish adding the emWin stack to your e² studio project.

The FSP does not automatically initialize the AppWizard system. To initialize it, simply include `GUI.h` and add the `MainTask()` API call to the appropriate file. The `MainTask()` is a powerful API that for the RA8D1, controls much of the logic for the graphics application, such as the GLCDC start, MIPI start, JPEG decoding and framebuffer rendering.

```
#include "emWin_thread.h"
#include "GUI.h"

/* emWin Thread entry function */
/* pvParameters contains TaskHandle_t */
void emWin_thread_entry(void *pvParameters)
{
    FSP_PARAMETER_NOT_USED (pvParameters);

    MainTask();

    while (1)
    {
        vTaskDelay (1);
    }
}
```

Figure 11. The MainTask() of the Thermostat Application is called from the emWin Thread

In the thermostat application, the `MainTask()` is called in the `emWin_thread_entry()` located in the `emWin_thread_entry.c` file. Visit section 4 for a full explanation of the thermostat project threads and functions.

3.3 Custom Designs in AppWizard

The AppWizard User Guide & Reference Manual covers the primitive objects available to add to your GUI. Primitive objects are straightforward and can operate stand alone, and they can be grouped together in the AppWizard Hierarchic tree to achieve more complex functions.

Every object contains its own set of properties that can be specified, such as the font, bitmap, frame size, etc. AppWizard has some basic fonts and image bitmaps built into the software that you can use out of the box, but you can also import your own bitmaps to create a unique GUI.

For example, a button object has a **Set bitmaps** property where you can set up to three different images for the button's unpressed, pressed, and disabled states, respectively. Clicking on **Set bitmaps > Unpressed** opens the **Select Image** window shown in Figure 12.

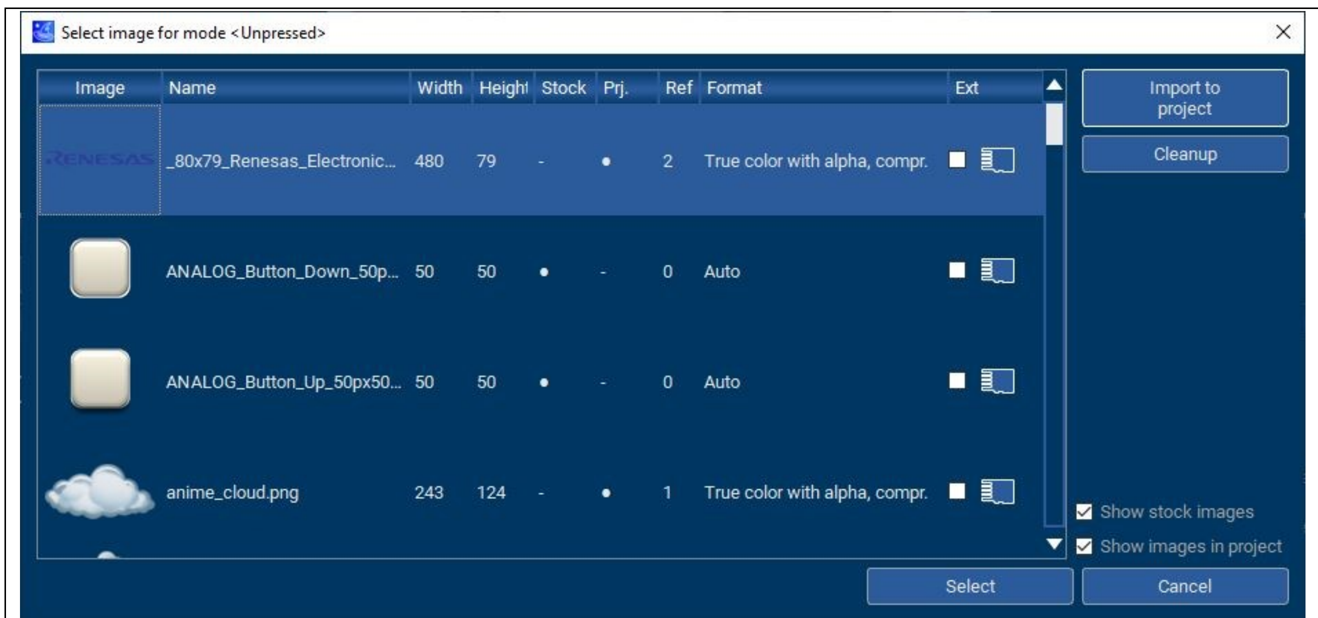

Figure 12. The Select Image Menu supports imported and built-in bitmaps for objects

The built-in AppWizard bitmaps will appear automatically in the list of images in this window, but you can also import your own image using the **Import to Project** button. The image manager window allows you to view the details of all bitmaps in the current AppWizard project, and it can be used to specify each image's bitmap pixel format.

Using a similar procedure, you can import your own font resources to the AppWizard project. All of this and more can be found in the AppWizard User Guide and Reference Manual.

3.4 Setup AppWizard Interactions

The **Interactions** window makes it easy for you to define the application's behavior based on certain actions, events, or user input. These interactions can be done without any extra code, but AppWizard also allows you to add custom slot codes to handle these actions and respond to GUI events. Each defined interaction in the same AppWizard project receives a unique default slot name, which can be changed by the user.

3.4.1 AppWizard Defined Interactions

As an example of interactions without extra code required, the following interactions defined in the thermostat application for the *ID_BUTTON_SUN* object handle the toggling of the Sunday button object's pressed and released images when a user either presses on the Sunday button or releases their touch, respectively:

- When *ID_BUTTON_SUN* is clicked:
  - the *ID_IMAGE_SUN* is hidden.
  - the *ID_IMAGE_SUN_PRESSED* is revealed.
- When *ID_BUTTON_SUN* is released:
  - the *ID_IMAGE_SUN* is revealed.
  - the *ID_IMAGE_SUN_PRESSED* is hidden.

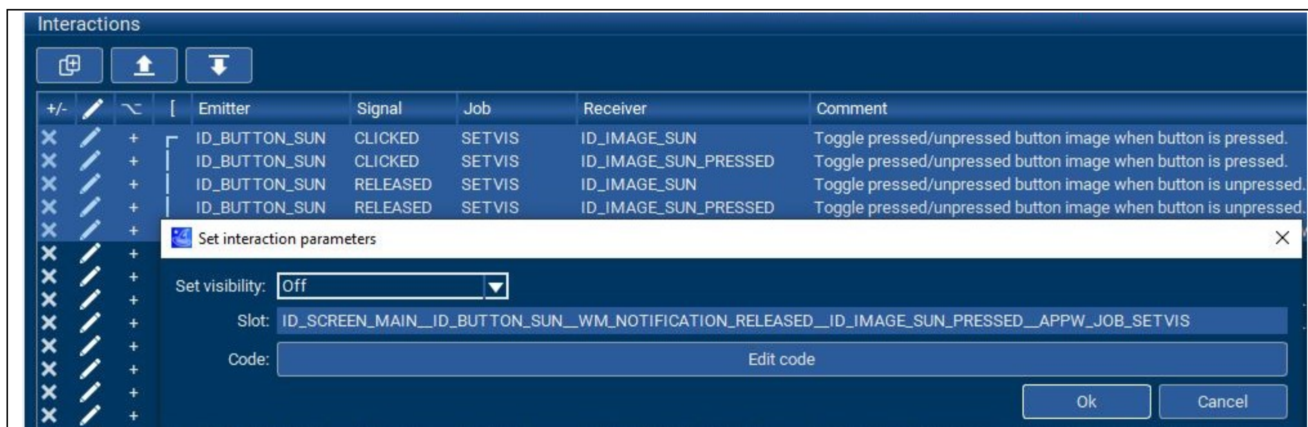

Figure 13. The highlighted interactions toggle the visibility of a button's pressed/unpressed images

3.4.2 User-Defined Slot Code

In some applications, you may need to define an interaction response that is not yet available in AppWizard or respond to an interaction with a more complex routine through a callback function in your code.

For each interaction, AppWizard automatically provides a callback function with the name specified in the **Slot** interaction parameter. AppWizard passes information to the slot routine about the window that caused the event and the actual event that occurred. These events are defined in `WM.h`.

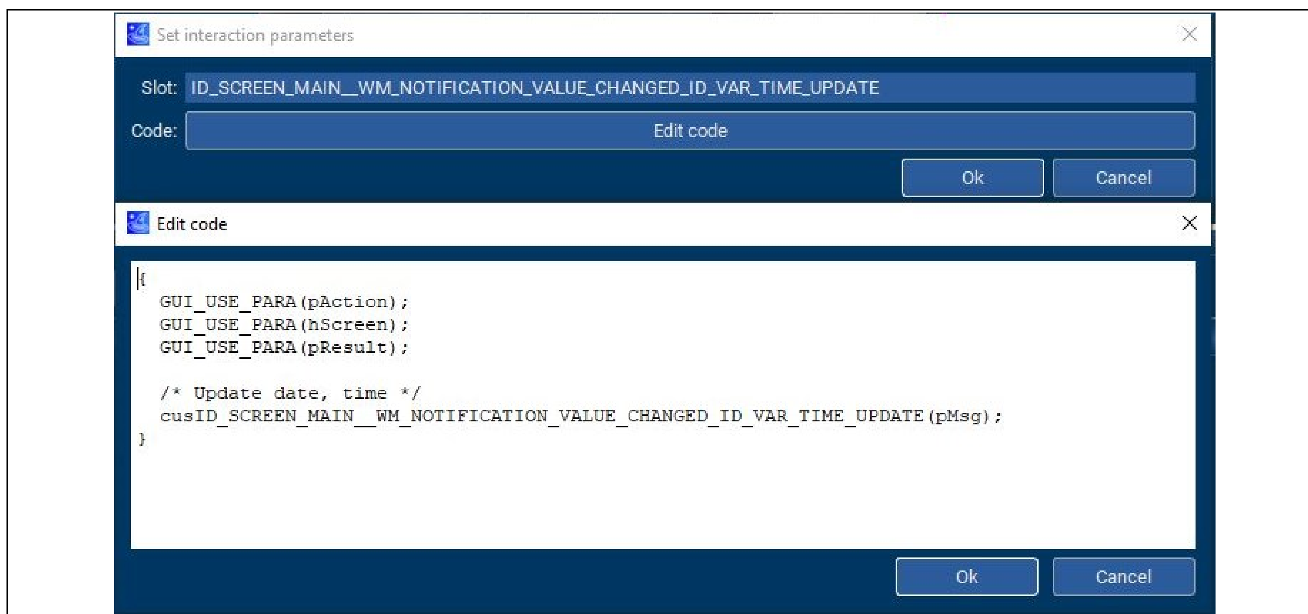

Figure 14. Custom slot routines can be added to interactions under Edit Code

You can view and edit the code of the slot routine using the **Code** parameter, which includes custom function calls for handling various window events. These custom routines will be added in the `<ScreenID>_Slots.c` file located in the `\AppWizard\Source\CustomCode` folder. The `<ScreenID>_Slots.c` file is updated whenever you add and generate new widgets or AppWizard interactions in AppWizard. It is good practice to create your custom code function definitions in a separate file and edit the **Code** interaction parameter to call the right custom function from the slot routine.

```
ID_SCREEN_MAIN_Slots.c

 *      ID_SCREEN_MAIN__WM_NOTIFICATION_VALUE_CHANGED_ID_VAR_TIME_UPDATE
void ID_SCREEN_MAIN__WM_NOTIFICATION_VALUE_CHANGED_ID_VAR_TIME_UPDATE(APPW_ACTION_ITEM * pAction,
                                                                      WM_HWIN hScreen,
                                                                      WM_MESSAGE * pMsg,
                                                                      int * pResult) {
  GUI_USE_PARA(pAction);
  GUI_USE_PARA(hScreen);
  GUI_USE_PARA(pResult);

  /* Update date, time */
  cusID_SCREEN_MAIN__WM_NOTIFICATION_VALUE_CHANGED_ID_VAR_TIME_UPDATE(pMsg);
}

Weather_Panel_Widget.c

 * @brief       Custom code for ID_SCREEN_MAIN__WM_NOTIFICATION_VALUE_CHANGED in ID_SCREEN_MAIN_Slots.c
void cusID_SCREEN_MAIN__WM_NOTIFICATION_VALUE_CHANGED_ID_VAR_TIME_UPDATE(WM_MESSAGE * pMsg) {
  if(WM_NOTIFY_PARENT == pMsg->MsgId)
  {
    /* Update time display */
    AppTimeUpdate();
  }
}
```

Figure 15. Custom routines allow personalized responses to interactions

3.4.3 Responding to AppWizard Variables

AppWizard supports adding variables to a project using the menu option **Resource > Edit Variables**. Variables can be processed by the application through a defined interaction through the GUI or manipulated from outside the GUI through code.

A typical use, as shown in the thermostat application, is to update the variables in a non-GUI thread to trigger data exchange between the AppWizard GUI and non-GUI threads.

```
/* Timer Thread entry function */
/* pvParameters contains TaskHandle_t */
void timer_thread_entry(void * pvParameters)
{
    FSP_PARAMETER_NOT_USED(pvParameters);
    fsp_err_t err = FSP_SUCCESS;

    /* Setup timers */
    err = gpt_timer_PWM_setup();
    APP_ERR_TRAP(err);

    err = rtc_timer_setup();
    APP_ERR_TRAP(err);

    while (1)
    {
        /* Wait for interrupt from RTC timer */
        BaseType_t xResult = xSemaphoreTake(g_timer_semaphore, portMAX_DELAY);
        if(xResult != pdTRUE)
        {
            __asm("BKPT #0\n");
        }
        /* Get date, time */
        R_RTC_CalendarTimeGet(&g_rtc_timer_ctrl, &RtcTimeCurrent);

        /* Trigger GUI update*/
        err = APPW_SetVarData(ID_VAR_TIME_UPDATE, 1);
        APP_ERR_TRAP(err);

        vTaskDelay (1);
    }
}
```

Figure 16. The Timer Thread triggers the ID_VAR_TIME_UPDATE to change the value each second

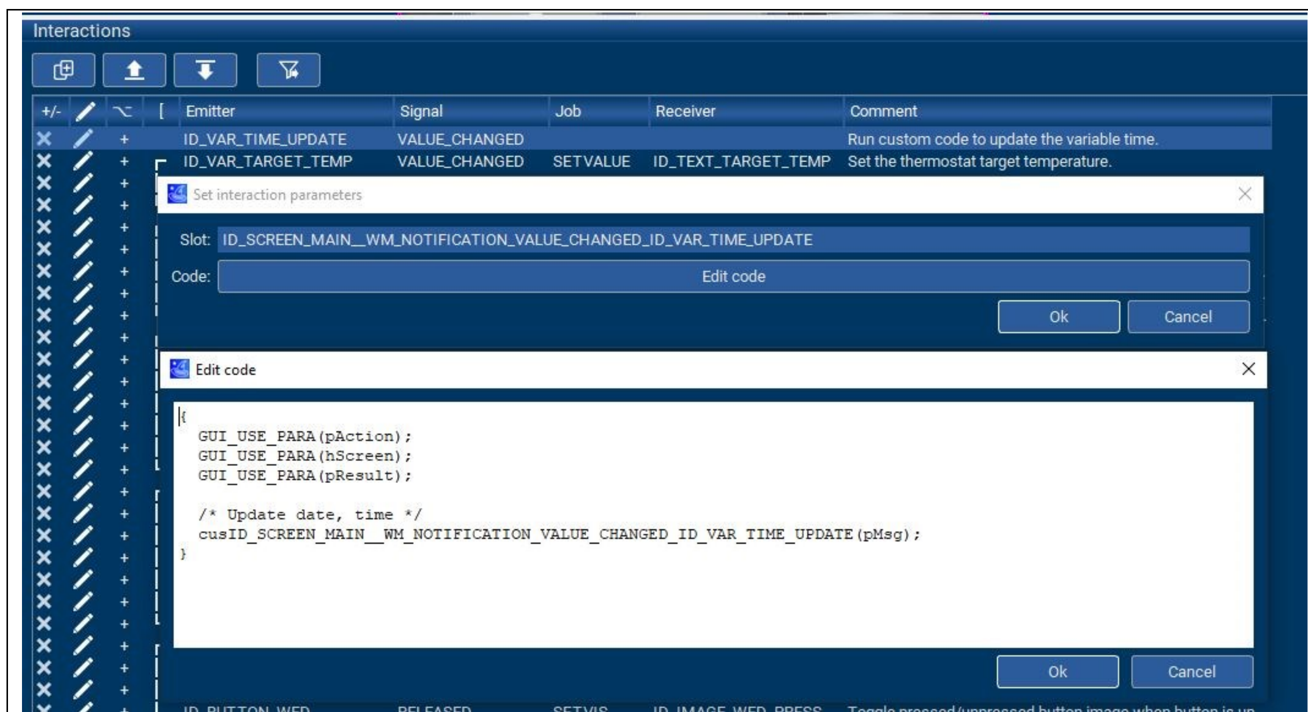

Figure 17. A custom slot routine is called when ID_VAR_TIME_UPDATE changes the value

3.5 Add emWin Widget to AppWizard Project

The objects that AppWizard supports are similar to the widgets provided by emWin. A widget is a window with object-type properties and the emWin library contains API functions for the creation, configuration, communication, and more, of widgets. The emWin library also supports the creation and management of custom widgets.

In some applications, you may need to use an emWin widget that is not yet supported by AppWizard objects or may need to create a custom widget in your code using the API functions. The thermostat application demonstrates creating and using a multiple-line text input or Multiedit widget. The following steps were used to add the widget to the application and highlighted in Figure 18:

- Create an emWin widget by using emWin APIs in the slot routine for the AppWizard screen in the `CustomCode` folder.
- Handle GUI events/message if needed via slot routines in the file `<ScreenID>_Slots.c` located in the `\AppWizard\Source\CustomCode` folder.

```
/* Create MultiEdit widget */
ghMultiEdit = MULTIEDIT_CreateEx(x0, y0, xSize, ySize, pMsg->hWin, WM_CF_SHOW,
                                 MULTIEDIT_CF_MOTION_V | MULTIEDIT_CF_READONLY,
                                 GUI_ID_MULTIEDIT0, 16, NULL);
if(ghMultiEdit)
{
    MULTIEDIT_SetBkColor(ghMultiEdit, MULTIEDIT_CI_READONLY, GUI_CUSTOM_COLOR);
    MULTIEDIT_SetWrapWord(ghMultiEdit);
    MULTIEDIT_SetMaxNumChars(ghMultiEdit, LOG_CHAR_MAX);
    MULTIEDIT_SetTextColor(ghMultiEdit, MULTIEDIT_CI_READONLY, GUI_WHITE);
    MULTIEDIT_SetHBorder(ghMultiEdit, 16);
```

```
Thermostat_App_EK_RA8D1 [Debug] [ra-solutio
  Binaries
  Includes
  AppWizard
    Resource
    Source
      CustomCode
        Application.c
        Application.h
        ID_SCREEN_LOG_Slots.c
        ID_SCREEN_MAIN_Slots.c
```

```
2   /*                          SEGGER Microcontroller GmbH
16
17  #include "Application.h"
18  #include "../Generated/Resource.h"
19  #include "../Generated/ID_SCREEN_LOG.h"
20
21  /*** Begin of user code area ***/
23
26  *       Public code
32  *       cbID_SCREEN_LOG
34  void cbID_SCREEN_LOG(WM_MESSAGE * pMsg) {
35      /* Custom code */
36      cuscbID_SCREEN_LOG(pMsg);
37  }
```

Figure 18. Example of using emWin APIs to create a widget during the screen creation routine

4. The Thermostat Application

The application project accompanying this note is an e² studio project titled Thermostat_App_EK_RA8D1, which is a portrait orientation graphics application of a thermostat. The project's purpose is multifaceted and demonstrates how to do the following with a single use case:

- Design a multi-screen GUI in AppWizard, targeting the EK-RA8D1's external MIPI display.
- Integrate the AppWizard GUI to the e² studio project: add the project folders to the e² studio workspace and communicate between AppWizard and e² studio threads.
- Use emWin widgets and custom slot code to achieve advanced GUI functionality.
- Configure emWin, MIPI, and GLCDC in the FSP to target the external display.
- Initialize the EK-RA8D1's external MIPI display by sending a table of configuration commands after the MIPI post-open event.
- Set up SDRAM to store the framebuffers on the EK-RA8D1.
- Communicate with the I2C capacitive touch overlay on the LCD using the GT911 drivers.

This section explains the thermostat application's code structure, high-level design, thread functions, and important module configurations. It also covers operating the EK-RA8D1's external display, including how to send the MIPI initialization command table to configure the LCD panel and how to use the GT911 drivers to interface with the capacitive touch panel.

Use the instructions in Section 5 to import, build, and run the graphics thermostat application project on your own hardware.

4.1 Source Code Layout

The thermostat application uses both custom code and auto-generated code. The custom code consists of the e² studio threads, gt911 drivers, custom slot routine callback functions, and widget routines for each screen. Both AppWizard and the FSP in e² studio auto-generate code files based on their respective project contents and settings. The thermostat app's auto-generated files include the AppWizard project's exported code, the FSP APIs for each included module, the FSP middleware for the EK-RA8D1 board, and the emWin library.

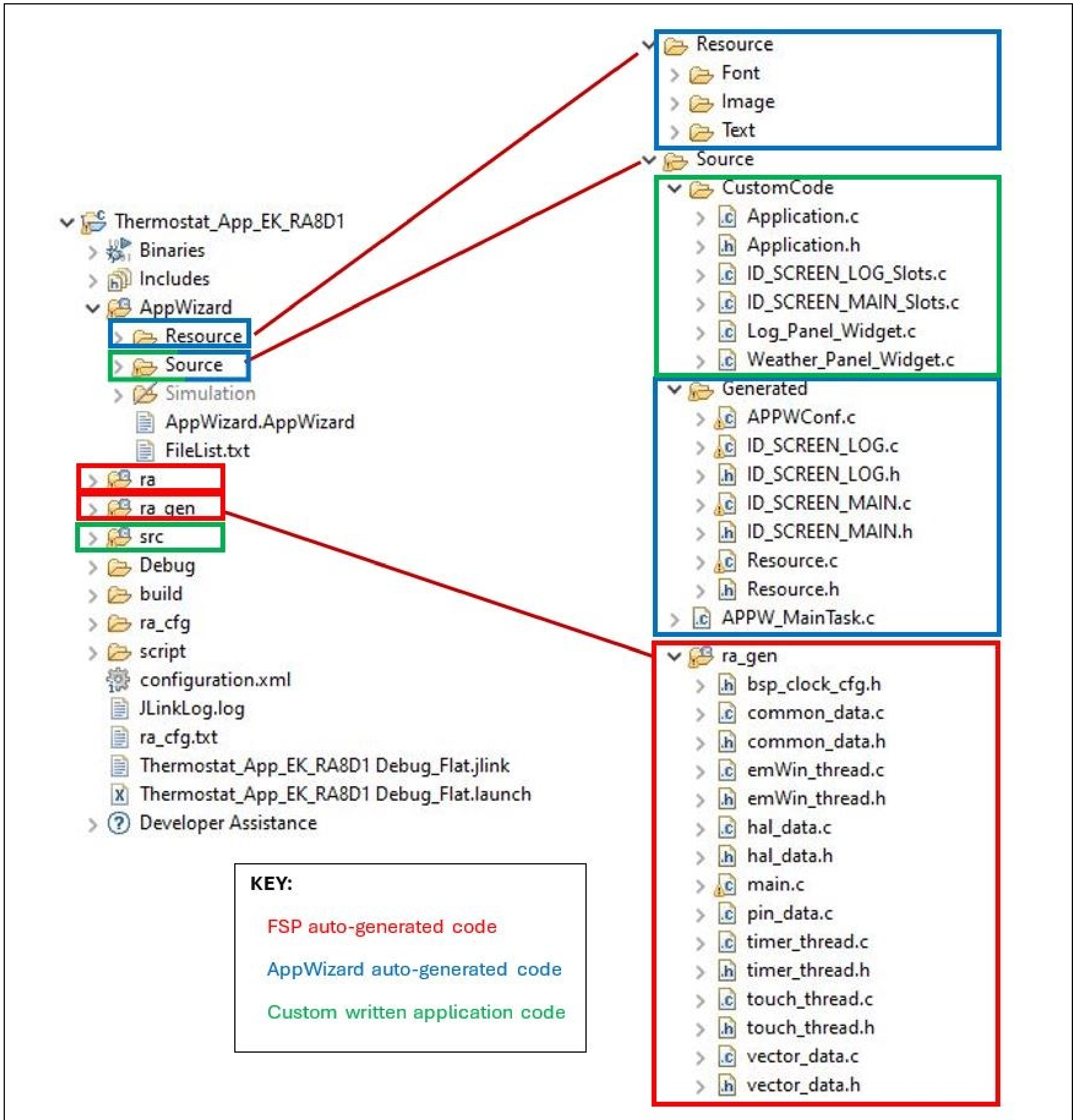

Figure 19. The Thermostat Application's source code layout

The above image shows the source code layout for the thermostat application on the EK-RA8D1 board. The e² studio project folders `ra_gen` and `ra` contain the auto-generated FSP module APIs, middleware, and public emWin library files. The e² studio project folder `src` contains the thread code and gt911 drivers. The AppWizard auto-generated code and support files are in both the Generated and Resource folders, and the custom slot and widget code are in the `CustomCode` folder.

Breaking down the custom code a bit further, the code in the AppWizard folder mainly targets HMI event handling, while the code in the src folder is related to the operation of MCU peripherals and overall application thread logic.

4.2 Application Block Diagram

The thermostat graphics application is a two-screen GUI that consists of a Weather Panel and a Logging Panel screen. The two application screens interface with the AppWizard/emWin graphics framework through interactions like touch events and variable data updates. HMI inputs to the capacitive overlay are processed

by the FSP and HAL I2C drivers to send touch-sensing data to the graphics framework. The FSP and HAL drivers also send GPT PWM duty cycle and RTC date and time data to the graphics framework.

The graphics framework includes the SEGGER AppWizard components, emWin library, and emWin RA port, and it interfaces with several HAL drivers like the GLCDC, DAVE 2D, MIPI DSI, and SDRAM. The following image depicts the application diagram:

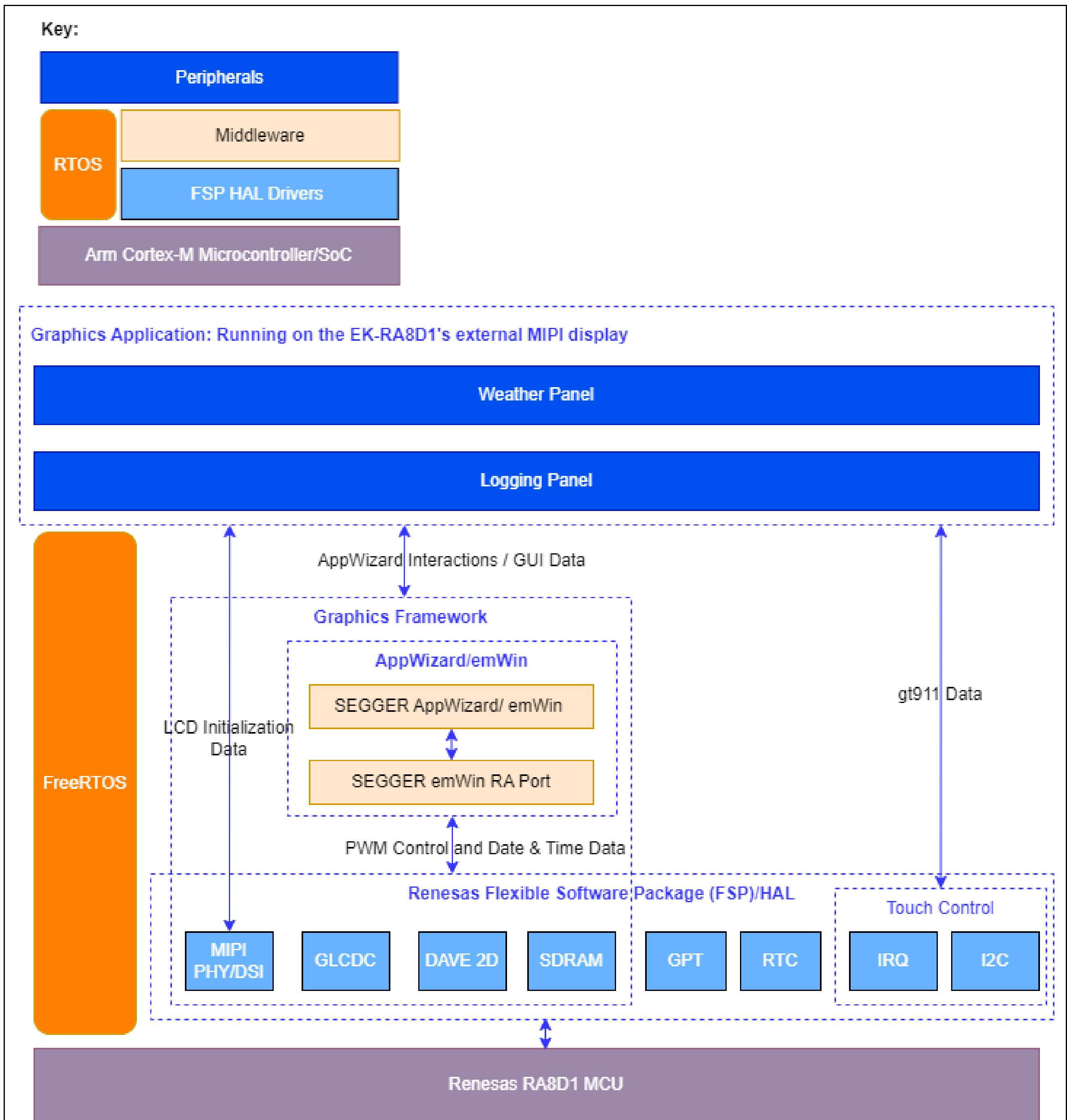

Figure 20. Block diagram of the Thermostat Application on the EK-RA8D1

4.3 Threads

The thermostat application is multi-threaded and runs using the FreeRTOS kernel which is fully supported by the FSP. In any RTOS FSP application there will be threads that are created by the FSP and threads that are created by the embedded software designer.

In the thermostat application, the FSP-created threads are the emWin/AppWizard main task threads, invoked from the emWin thread. The user-created threads include part of the emWin thread and all of the Touch and

Timer threads. Threads communicate through FSP-defined FreeRTOS semaphores, GUI events through AppWizard interactions, and emWin event API calls. The emWin thread processes the data and touch events sent from the Timer and Touch thread.

The following figure shows a high-level diagram of the threads and event flow in the graphics application. Note that the emWin Thread component is colored with a gradient to represent how the thread contains both user-written code and emWin framework code.

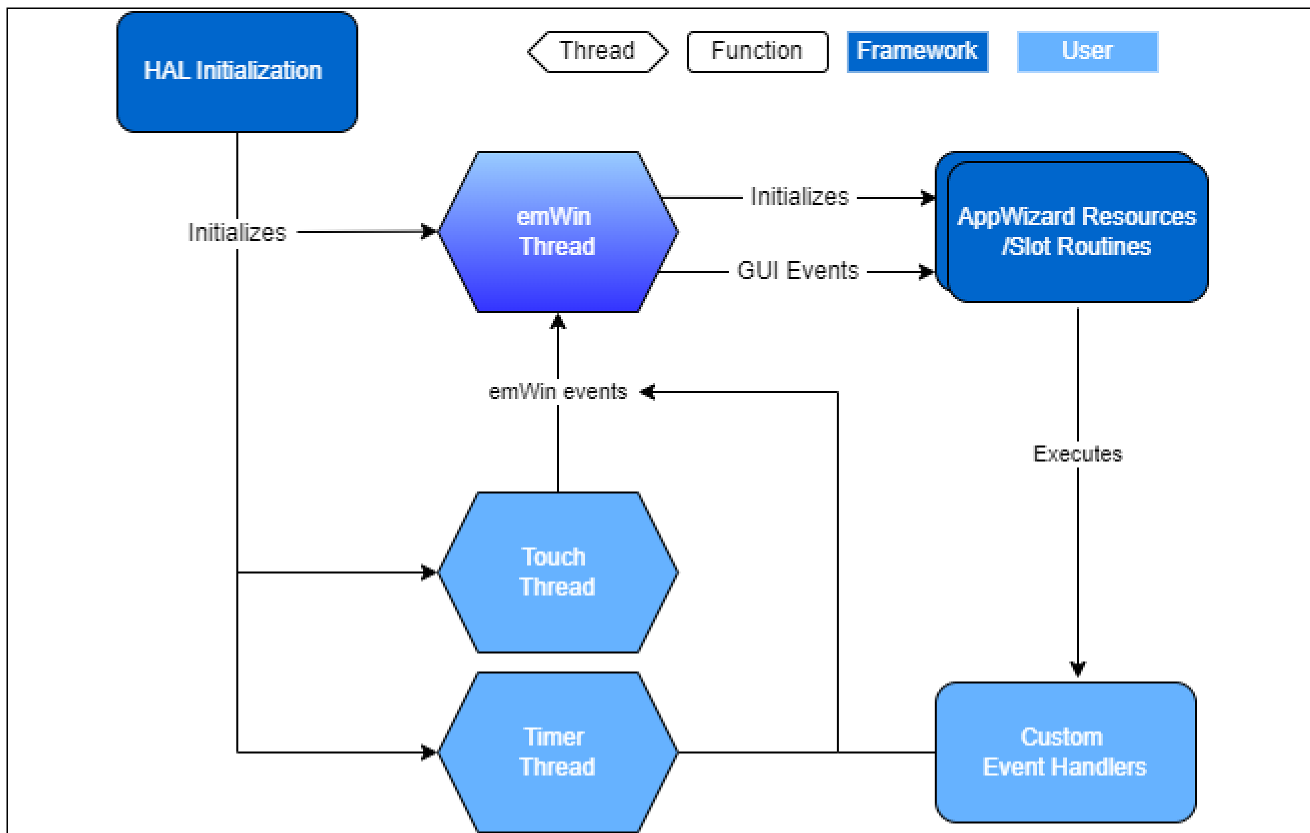

Figure 21. The Thermostat Application's event and thread flow

4.3.1 emWin Thread

The emWin thread initializes the hardware on the MCU and starts the services used by the thermostat application, such as emWin. The thread first initializes the EK-RA8D1' SDRAM by calling the included BSP function `bsp_sdram_init()`. After, it waits for the initialization of the touch device before proceeding. Next, the thread starts the emWin library's MainTask() routine, which is responsible for the bulk of the graphics framework operation, including configuring the GLCDC, DAVE 2D, MIPI PHY, and DSI modules, processing AppWizard touch events and window messages, drawing the next framebuffer, and writing the framebuffer to the external display.

After the MIPI DSI module is first opened, the MIPI DSI Host operates in Command Mode to send the appropriate sequence of descriptors from the MCU to the LCD panel, which configures the external display. See section 4.4, Configuring the EK-RA8D1's MIPI LCD, for specific details on the MIPI initialization sequence operation routine defined in `emwin_thread_entry.c`.

The `MainTask()` routine continues to process touch events, variable changes, and AppWizard window event messages that are sent from the Touch and Timer threads. If any of these result in a change to the system state then emWin evokes the related AppWizard interaction slot routine, resulting in changes to the graphical HMI. The MIPI DSI Host operates in Video Burst Mode to send pixel data to the LCD panel. The flowchart gives a high-level view of the emWin thread.

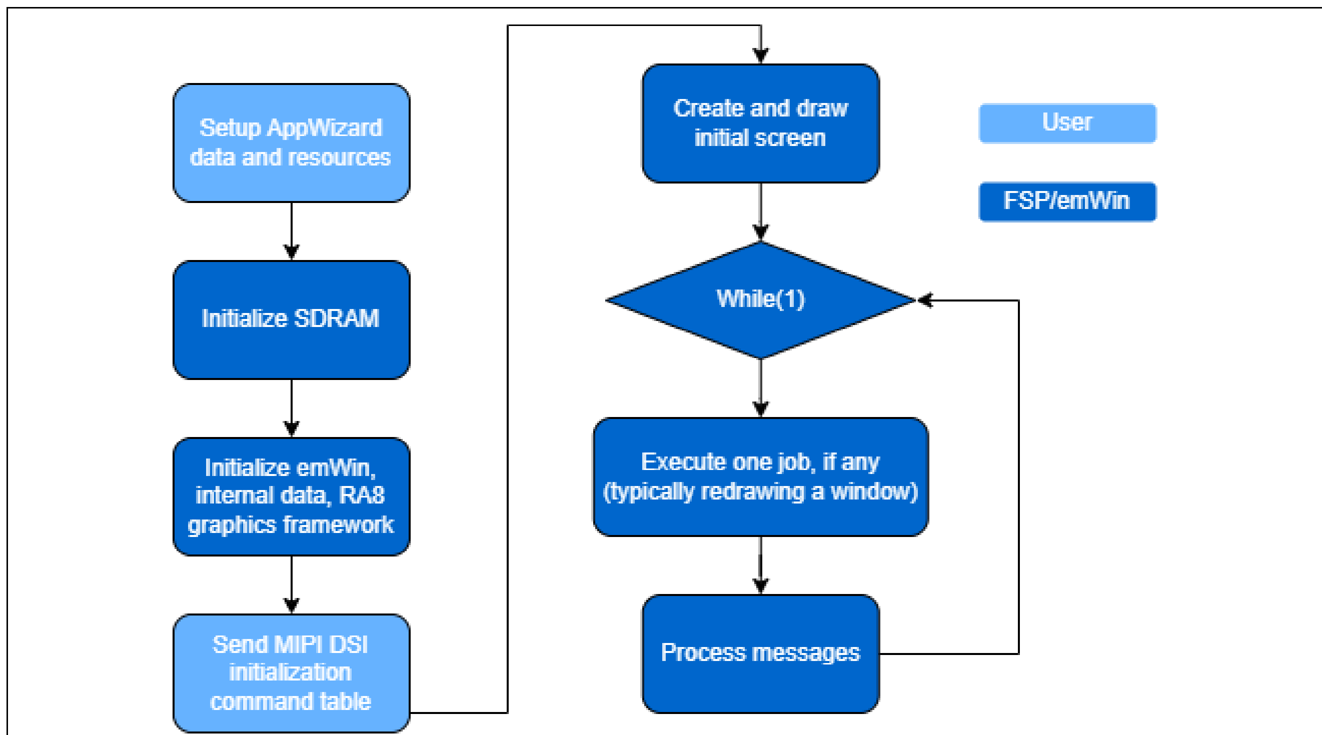

Figure 22. High-level diagram of the emWin Thread

4.3.2 Touch Thread

The Touch thread uses the gt911 drivers, IRQ, and I2C modules to detect and read the touch data input from the capacitive overlay on the display. When changes are detected in the coordinate input from users touching the capacitive touch overlay, like pressing the buttons of the days of the week or toggling a switch, the touch thread sends this coordinate data and presses the state to emWin using the `GUI_PID_StoreState` API.

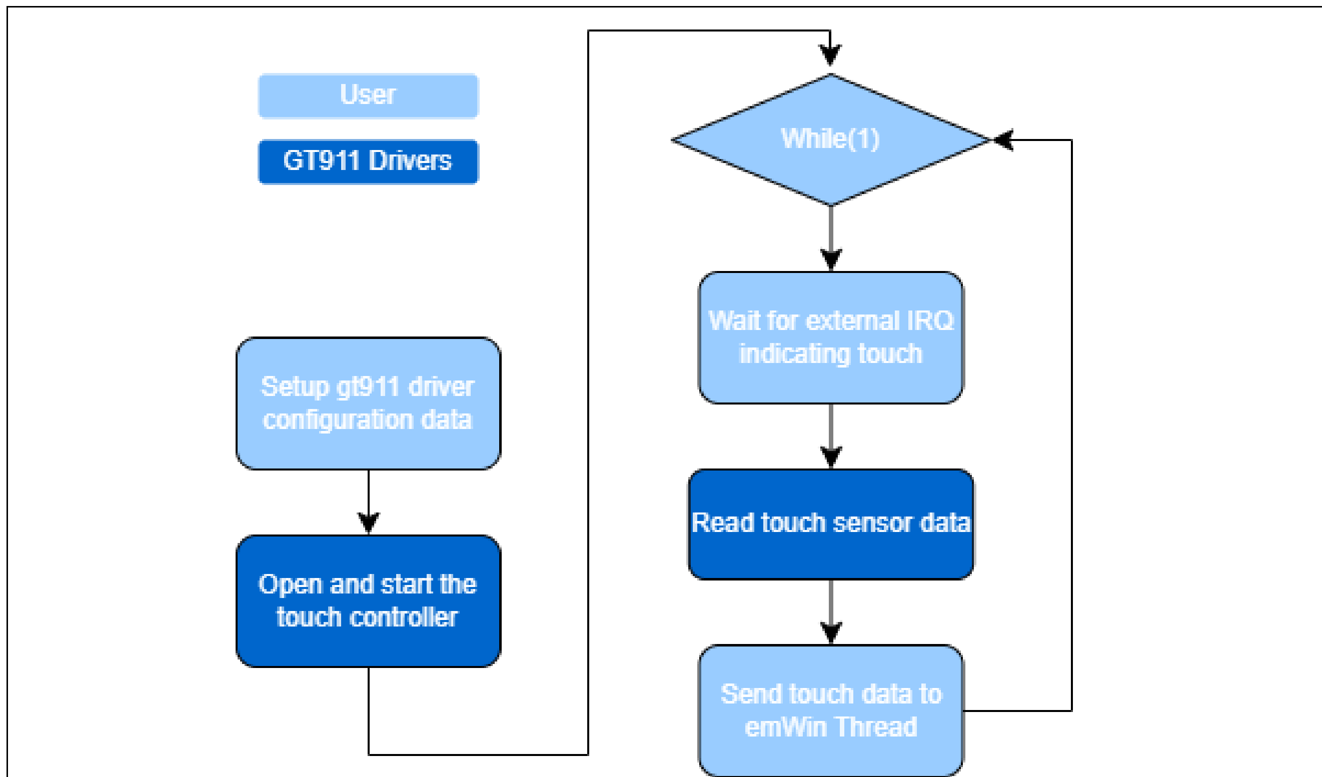

Figure 23. High-level diagram of the Touch Thread

4.3.3 Timer Thread

The timer thread uses the GPT to control the display's backlight, and the RTC keeps track of the date and time. When a user interacts with the Logging Screen and moves the backlight slider, the touch and emWin threads update the duty cycle of the GPT pin connected to the LCD backlight control.

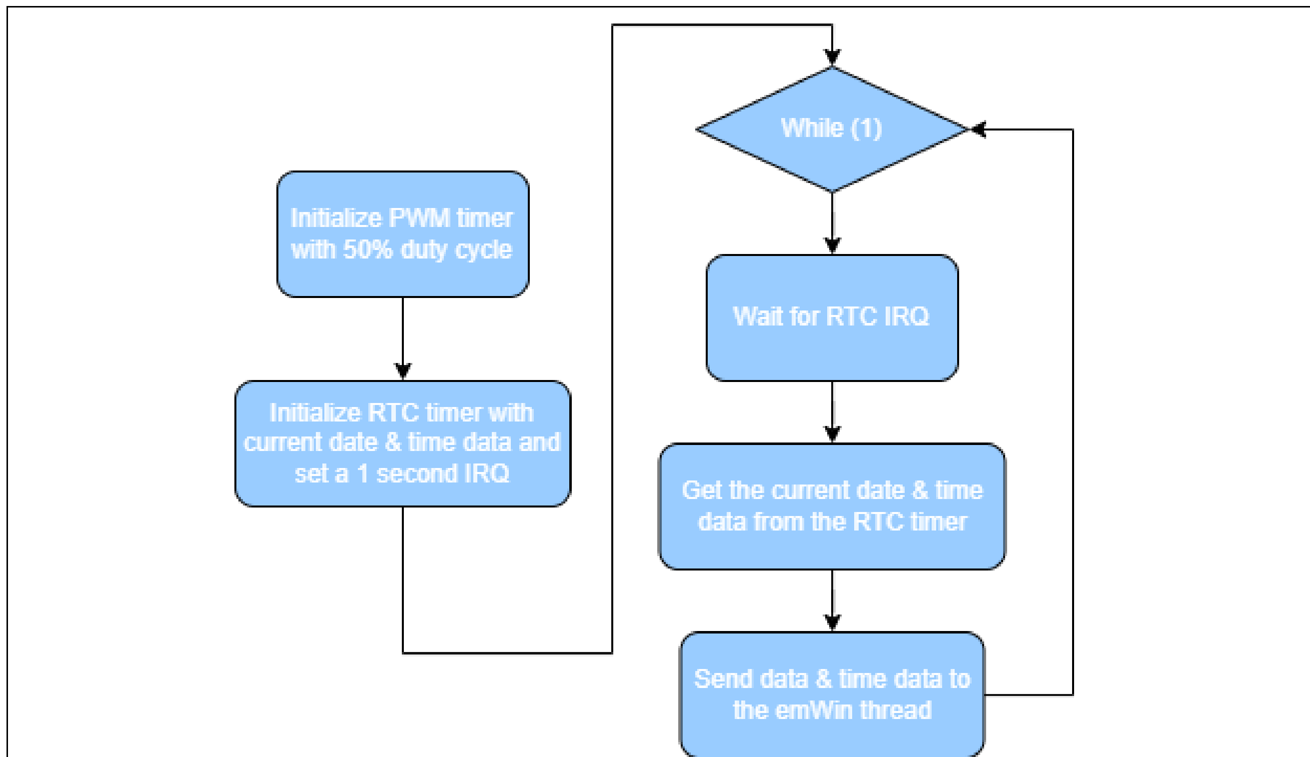

Figure 24. High-level view of the Timer Thread

4.4 Configuring the EK-RA8D1's MIPI LCD

The emWin thread also synchronizes with the custom-written definition of the `mipi_dsi0_callback()` routine. In general, the function filters through the various MIPI DSI or PHY events, flags, and errors and handles the application's response accordingly.

```
void mipi_dsi0_callback(mipi_dsi_callback_args_t *p_args)
{
    switch(p_args->event)
    {
        case MIPI_DSI_EVENT_POST_OPEN:
        {
            /* Initialize sequence for LCD. MIPI operation is in Command Mode */
            mipi_dsi_push_table(g_lcd_init_focuslcd);
        }
        break;
        case MIPI_DSI_EVENT_SEQUENCE_0:
        {
            if(MIPI_DSI_SEQUENCE_STATUS_DESCRIPTORS_FINISHED == p_args->tx_status)
            {
                /* Set MIPI command received semaphore */
                BaseType_t xHigherPriorityTaskWoken = pdFALSE;
                BaseType_t xResult = xSemaphoreGiveFromISR(g_mipi_semaphore, &xHigherPriorityTaskWoken);
                if(xResult != pdTRUE)
                {
                    __asm("BKPT #0\n");
                }

                /* Return to the highest priority available task */
                portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
            }
            break;
        }
        case MIPI_DSI_EVENT_PHY:
        {
            g_phy_status |= p_args->phy_status;
            break;
        }
        default:
        {
            break;
        }
    }

    return;
}
```

Figure 25. The MIPI configuration table is sent after the MIPI DSI module has opened successfully

For the thermostat application, the MIPI DSI post-open event indicates that the EK-RA8D1's external MIPI LCD is ready to be configured. The MIPI DSI module operates in Command Mode to send a series of MIPI descriptors from the MCU to the LCD. The sequence of descriptors or commands will initialize the external display based on the FocusLCD specs and will specify the communication settings to be used when sending pixel data during MIPI video mode. The commands are enumerated in the `src\emWin_thread.c` file as `g_lcd_init_focuslcd[]` and cover settings like bit depth, screen resolution, positive and negative gamma, and more.

It is recommended that you review the `g_lcd_init_focuslcd[]` commands, the ILI9806E data sheet and the corresponding MIPI display specifications of the RA8D1 Hardware User's Manual to understand the mechanisms of the initialization command sequence.

4.5 GT911 Touch Drivers

As mentioned in Section 1, the EK-RA8D1's external MIPI graphics expansion board has a capacitive touch panel (referred to as CTP) overlay, which runs using the GT911 controller IC. Please refer to the GT911 Programming Guide and the GT911 Datasheet from Goodix in the reference section at the end of the application note for more information. This section gives an overview of important GT911 specifications, shows the connection from the MCU to the CTP, and explains how the thermostat application uses the `touch_gt911.c` and `touch_gt911.h` driver files to interface with the touch panel.

The GT911 serves as a slave device in I2C communication to the MCU, sending up to 5 connection points with a maximum transmission rate of 400Kbps. The device has two available 7-bit slave addresses: 0x14/0x5D. For the purposes of this application note, we will only discuss and use the 0x14 address. The IC interfaces with the host via six pins (the parentheses note the equivalent pin name on the RA8D1 MCU): VDD, GND, SCL (IIC_SCL), SDA (IIC_SDA), INT (DISP_INT), and RESET (DISP_RST).

The important signal connections from the CTP to RA8D1 are highlighted in Figure 26. The J58 header refers to the board-to-board connector, which connects the MIPI graphics expansion board to the RA8D1 MCU.

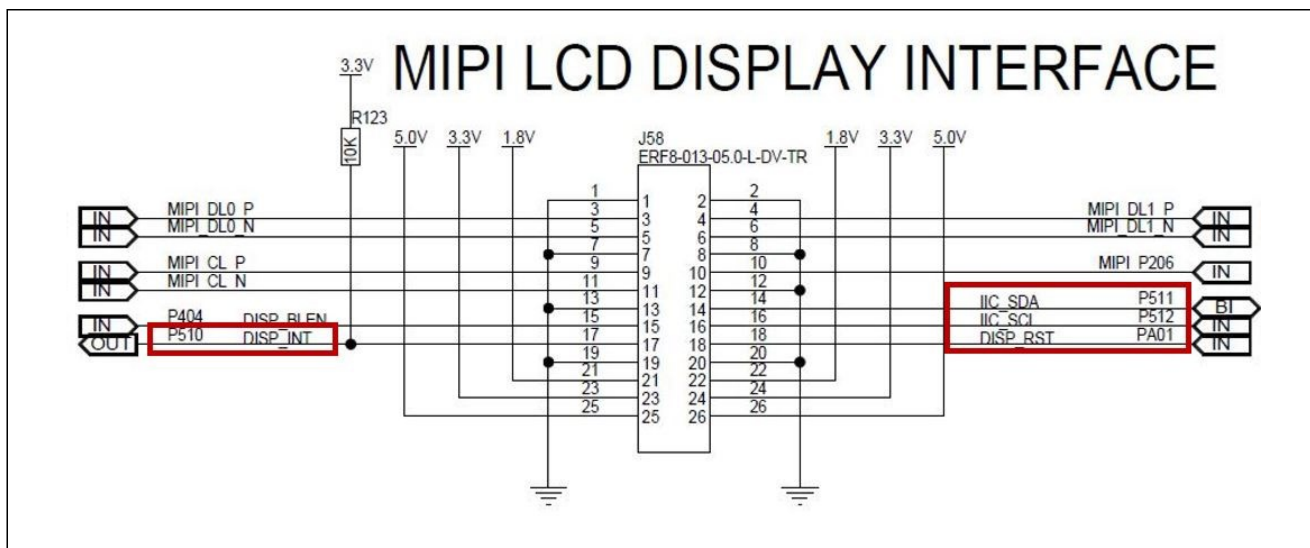

Figure 26. RA8D1 I/O connections to the GT911 IC on the external LCD

The MCU selects the GT911 I2C slave device address after application start-up by controlling and toggling the DISP_RST and DISP_INT timing sequence. The reset logic is provided in the `R_TOUCH_GT911_Reset` driver function defined in `touch_gt911.c`. However, note in the figure above that the DISP_RST display reset signal from the RA8D1 MCU is routed both to the capacitive touch panel and to the MIPI LCD panel. Therefore, it's critical to complete the touch panel setup procedure BEFORE opening and starting the MIPI modules on the RA8D1 so you don't accidentally reset the MIPI after opening. The thermostat application uses the `g_touch_reset_semaphore` to block the emWin thread from opening and starting the MIPI modules until after the touch thread has completed the touch panel startup.

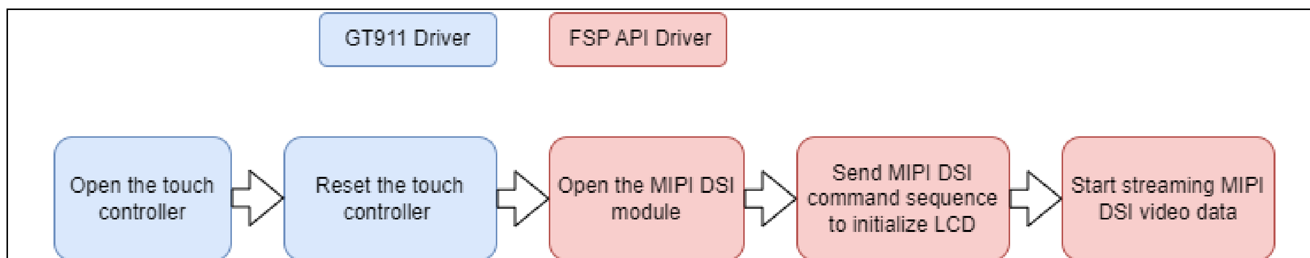

Figure 27. Recommended initialization sequence for the EK-RA8D1 LCD's Touch and MIPI controllers

The following functions are currently provided by `touch_gt911` drivers:

- `R_TOUCH_GT911_Open:` Sets up the signals, like opening the I2C master module on the MCU and specifying the right pins for SDA and SCL.
- `R_TOUCH_GT911_Reset:` Toggles the DISP_INT and DISP_RST pins with the right timing sequence to reset the touch hardware.
- `R_TOUCH_GT911_Close:` Closes the I2C master module on the MCU.
- `R_TOUCH_GT911_StatusGet:` Returns the status of the touch panel.
- `R_TOUCH_GT911_PointsGet:` Returns the number of points in contact with the panel and their coordinates.
- `R_TOUCH_GT911_VersionGet:` Returns the gt911 version.
- `R_TOUCH_GT911_WaitForTouch:` Waits until a gt911 touch event occurs.

The touch thread uses the above functions to properly set up touch panel communication. The following image is a snapshot that encompasses the touch setup procedure in the thermostat application, located in `touch_thread_entry.c`. The key parts include the initialization of gt911 configurations, the sequence of gt911 driver functions used, and unblocking the emWin thread for MIPI setup after the touch reset completes. Note

that during the initialization of the gt911 configurations, the driver configurations are required to have two semaphore objects, one for the irq interrupt and one for the i2c communication.

```
    touch_gt911_ctrl_t gt911_ctrl;
    fsp_err_t err = FSP_SUCCESS;
    uint32_t gt911_err = 0;
    uint16_t gt911_fw_version = 0;

    sync_objects_t gt911_sync =
    {
#if USE_EVENT_FLAGS == 1
     .p_event_flag = &g_event_flags_gt911,f
#elif USE_SEMAPHORES == 1
     .p_semaphore_i2c = g_semaphore_gt911_i2c,
     .p_semaphore_irq = g_semaphore_gt911_irq,
#endif
    };

    /* Initialize touch configuration*/
    touch_gt911_cfg_t gt911_cfg =
    {
     .reset_pin = IOPORT_PORT_10_PIN_01,
     .irq_pin = IOPORT_PORT_05_PIN_10,
     .i2c_sda_pin = IOPORT_PORT_05_PIN_11,
     .i2c_scl_pin = IOPORT_PORT_05_PIN_12,
     .p_i2c_master = &g_i2c_gt911,
     .p_external_irq = &g_int_gt911,
     .p_ioport = &g_ioport,
     .sync = &gt911_sync,
    };

    /* Validate touch configuration */
    err = R_TOUCH_GT911_Validate(&gt911_cfg, &gt911_err);
    APP_ERR_TRAP(err);

    memset(&gt911_ctrl, 0, sizeof(touch_gt911_ctrl_t));

    /* Open the touch controller*/
    err = R_TOUCH_GT911_Open(&gt911_ctrl, &gt911_cfg);
    APP_ERR_TRAP(err);

    /* Reset the touch hardware*/
    err = R_TOUCH_GT911_Reset(&gt911_ctrl);
    APP_ERR_TRAP(err);

    err = R_TOUCH_GT911_VersionGet(&gt911_ctrl, &gt911_fw_version);
    APP_ERR_TRAP(err);

    /* Tell the emWin thread that touch reset has completed */
    xSemaphoreGive(g_touch_reset_semaphore);
```

Figure 28. The GT911 drivers are used in the Touch Thread to setup communication to the capacitive touch panel

4.6 FSP Configuration

When designing any FSP-based application, it is important to have a deep understanding of the software design to be implemented and the hardware it will run on. From the hardware perspective, this means knowing the types of peripherals to be used, the I/O signal connections to the peripherals, any electrical characteristics or restrictions, etc. From the software perspective, this means deciding how many threads to use, which threads need access to which hardware components, and what additional objects, like semaphores, will be needed. After determining the above, it will be easy to use the FSP Configuration View in e^2 studio to set up your application project's contents, like the hardware modules, threads, objects, etc, by editing the `configuration.xml` file.

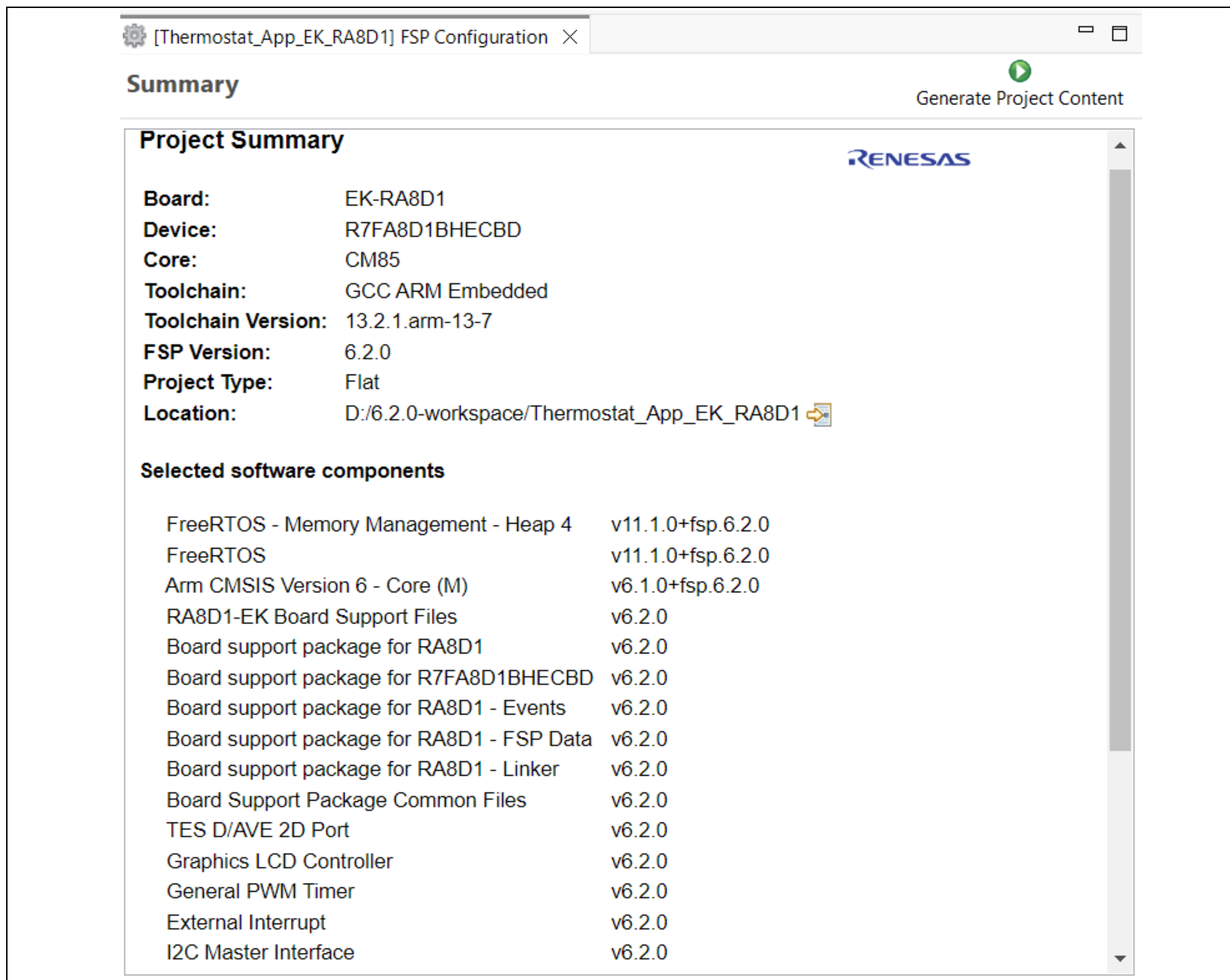

Figure 29. Summary of the Thermostat Application FSP Configuration

4.6.1 Components:

The Components tab enumerates all of the available hardware and software component versions supported by the FSP packs in your e² studio installation. After you add, organize, and configure the hardware and software modules necessary for your application, the Components tab will mark which resources are being used. The FSP will only compile the resources required for the application's components, optimizing the final application size.

The following table highlights the selections used in the thermostat application:

Table 4. Components used in the Thermostat Graphics Application

Category	Component	Version	Description
CMSIS5	`CoreM`	6.1.0 + fsp.6.2.0	Arm CMSIS Version 6 – Core (M)
RTOS	`FreeRTOS`	11.1.0 + fsp.6.2.0	FreeRTOS
Heaps	`heap_4`	11.1.0 + fsp.6.2.0	FreeRTOS – Memory Management – Heap 4
BSP	`ra8d1_ek`	6.2.0	RA8D1-EK Board Support Files
Common	`fsp_common`	6.2.0	Board Support Package Common Files
HAL Drivers	`r_drw`	6.2.0	TES D/AVE 2D Port
	`r_glcdc`	6.2.0	Graphics LCD Controller
	`r_gpt`	6.2.0	General PWM Timer
	`r_icu`	6.2.0	External Interrupt
	`r_iic_master`	6.2.0	I2C Master Interface
	`r_ioport`	6.2.0	I/O Port
	`r_mipi_dsi`	6.2.0	MIPI DSI Host
	`r_mipi_phy`	6.2.0	MIPI PHY Host

	r_rtc	6.2.0	Real Time Clock
Middleware	rm_emwin_port	6.2.0	SEGGER emWIn RA Port
	rm_freertos_port	6.2.0	FreeRTOS Port
TES	dave2d	3.8.0 + fsp.6.2.0	TES D/AVE 2D Drawing Engine
GUI	emWin	6.48.0 + fsp.6.2.0	SEGGER emWin Library

4.6.2 Thread Objects

The thermostat application uses FreeRTOS, which supports various objects like mutexes, queues, semaphores, and timers, which can be defined in the **Objects** window of the **Stacks** tab. The thermostat application uses five semaphore objects to communicate between threads and functions:

- **g_semaphore_gt911_irq:** Required when using the `gt911.c` and `gt911.h` drivers. This object is used to block the touch thread until the drivers indicate that the coordinate data is ready to read.
- **g_semaphore_gt911_i2c:** Required when using the `gt911.c` and `gt911.h` drivers. This object is used internally by the drivers for I2C communication timing.
- **g_timer_semaphore:** This object blocks the timer thread from updating the time variable until the RTC callback indicates that a second has passed.
- **g_mipi_semaphore:** This object is used in the emWin thread when pushing the sequence of startup commands to the MIPI display. After each command sent by the `mipi_dsi_push_table()` function, the function blocks until the `mipi_dsi_callback` indicates that the command was sent successfully.
- **g_touch_reset_semaphore:** This object blocks the emWin thread from initializing the MIPI module until the touch thread indicates that the touch panel has been initialized first. Recall that the DISP_RST line, which connects to both the touch panel and the MIPI panel, must be reset to start the touch panel.

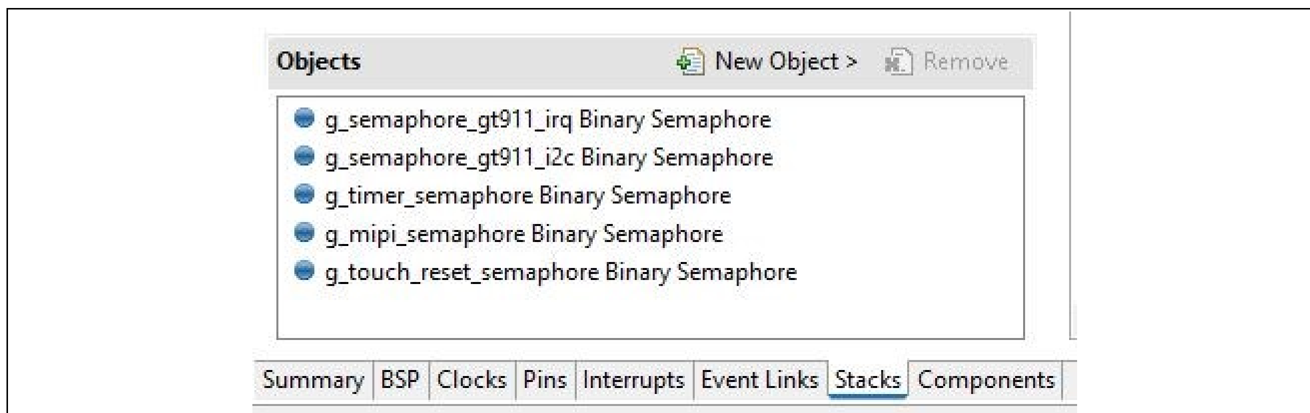

Figure 30. The FreeRTOS semaphore objects used by the thermostat graphics application

4.6.3 Module, Pin and Clock Configuration

The **Stacks** tab is where the thermostat application's threads, modules, and FreeRTOS objects have been added, managed, and configured by the designer for the FSP. Once the proper modules are included in your project, the **Generate Project Content** button will add the corresponding FSP support files for operating the modules defined in the Stacks. As mentioned earlier, the generated FSP files lie within the `ra` and `ra_gen` folders of the e^2 studio project.

In Figure 31 the Threads window shows which hardware peripheral resources have been assigned to each of the thermostat application's threads. The rest of this section will list each module and discuss the important settings that enable the graphics framework to operate cohesively.

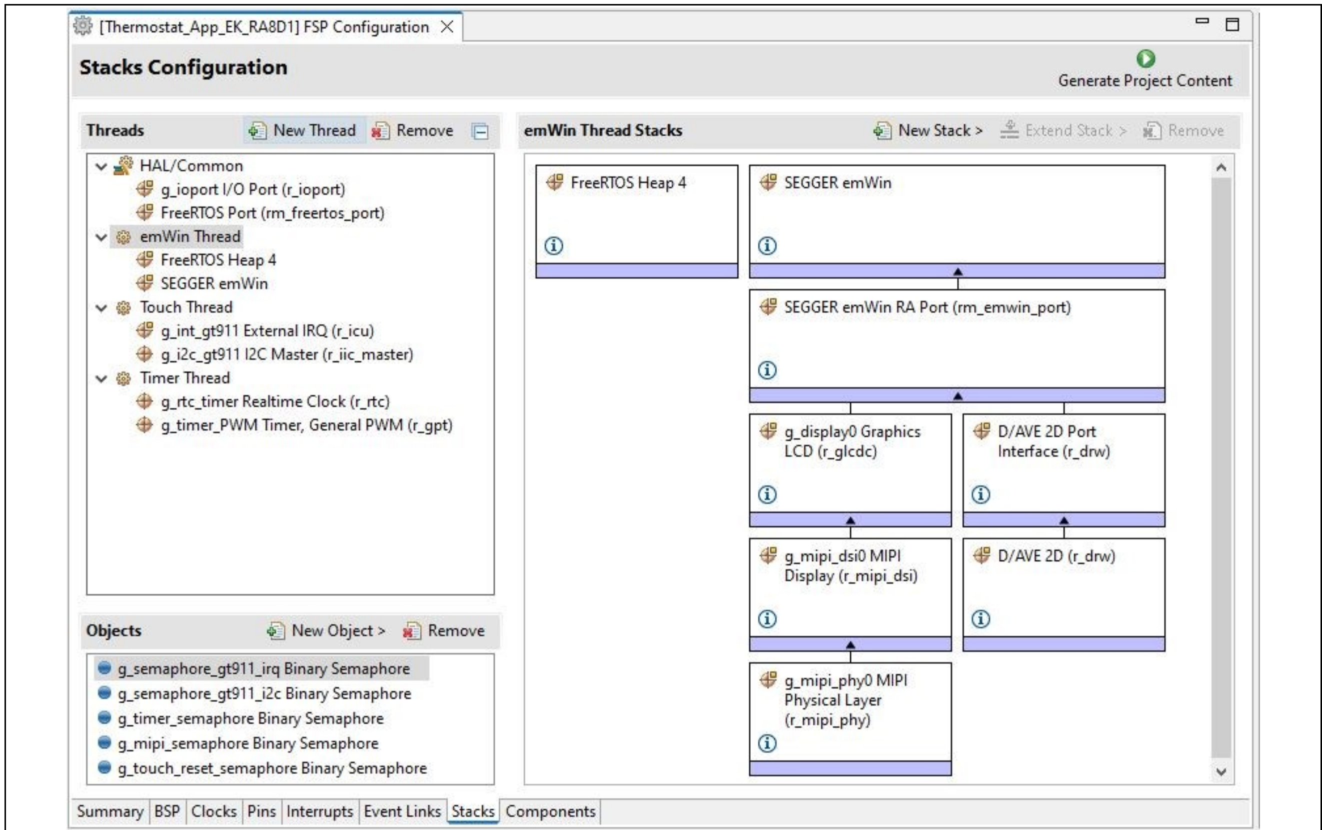

Figure 31. The Stacks tab of the configuration.xml shows threads, stacks, and RTOS objects

FreeRTOS Heap 4

A heap memory allocation is required for FreeRTOS to operate correctly because the kernel requires RAM each time a task, queue, mutex, semaphore, etc, is created. The thermostat application uses “Heap 4”, described by FreeRTOS, to coalesce adjacent free blocks to avoid fragmentation and includes absolute address placements.

SEGGER emWin RA Port (rm_emwin_port)

As depicted in Figure 31 above, the rm_emwin_port is a submodule of the SEGGER emWin graphics framework. It provides the configuration and hardware acceleration support necessary for the use of emWin on RA MCUs, allowing for full integration with the GLCDC, DRW, and MIPI graphics peripherals on the RA8D1, as well as with FreeRTOS. Please contact SEGGER for the source code of the emWin Library.

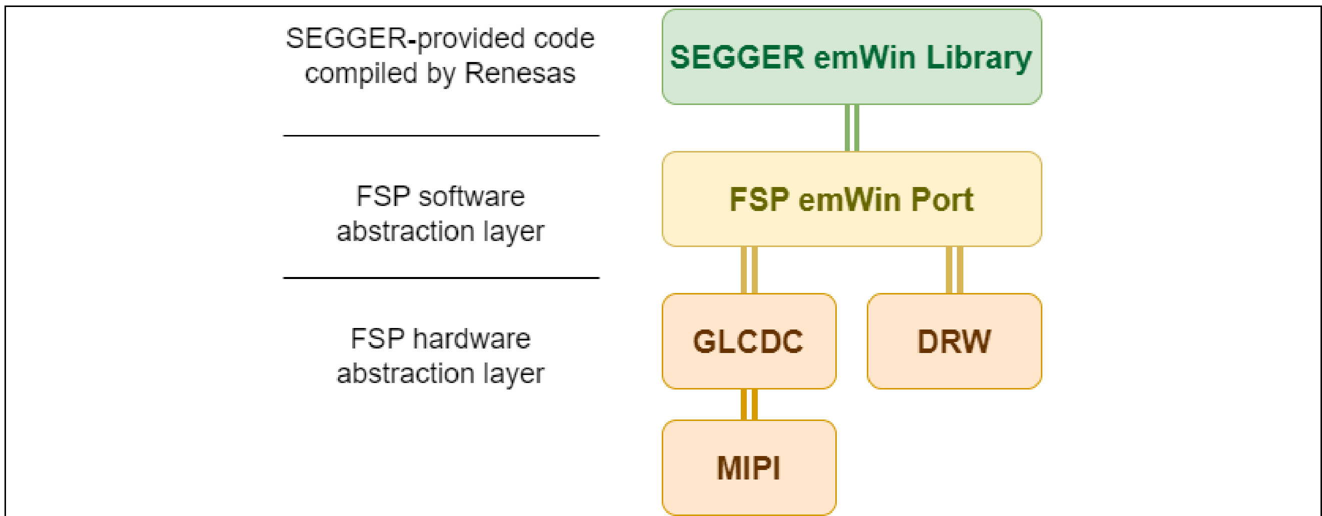

Figure 32. SEGGER emWin FSP port layers block diagram

Please review the table below for an explanation of the FSP properties of the rm_emwin_port in the thermostat application that differs from the default:

Table 5. SEGGER emWin RA Port (rm_emwin_port) configurations for the Thermostat Application

Property	Description	Value Used	Explanation
Memory Allocation > GUI Heap Size	Set the size of the heap to be allocated for use exclusively by emWin.	0x80000	Provide sufficient RAM for the JPEG decoding and more. See section 9.2.5 Memory usage in the emWin User Guide for calculation suggestions.
Memory Allocation > Section for GUI Heap	Specify the section in which to allocate the GUI heap.	.bss	Place the GUI heap in the on-chip SRAM region.
JPEG Decoding > General > Double-Buffer Output	Configure whether JPEG decoding operations use a double-buffer pipeline.	Enabled	This allows the JPEG to be rendered to the display while decoding another in, at the cost of additional RAM usage.
JPEG Decoding > Buffers > Section for Buffers	Specify the section in which to allocate the JPEG buffers.	.bss	Place the JPEG decoding into the on-chip SRAM

Graphics LCD (r_glcdc)

The graphics LCD submodule required a few configuration changes to get operational. The GLCDC input format must match the AppWizard project color format settings, which in the thermostat application are RGB565. This is the default setting, so no adjustment was needed, but pay attention to the color format of your own graphics application and ensure that the right format is being used throughout.

When you add the r_glcdc module to an RA8D1 project for the first time, it will show up with a red warning because the default FSP clock setting provides no input source for the LCD clock (LCDCLK). Navigate to the Clocks tab to enable the correct clock for your application. The thermostat application uses the PLL1P as the LCDCLK source, resulting in a 240MHz LCDCLK.

Figure 33. PLL1P is the clock source for the LCDCLK in the Thermostat Application

MIPI (r_mipi_phy and r_mipi_dsi)

The default settings of the MIPI PHY and MIPI DSI graphics submodules on the RA8D1 are preconfigured for communication with the EK-RA8D1's external MIPI display, and no changes were necessary for MIPI operation in the thermostat application.

GT911 External IRQ (r_icu)

The r_icu module is used to detect external interrupt signals coming into the MCU. Recall from Figure 26 in the GT911 Touch Driver Section the GT911 IC uses the DISP_INT pin, which is connected to P510 on the RA8D1 MCU, to signal when GT911 touch events occur. Therefore, in the thermostat application, the external IRQ needs to be set to **Channel > 3** because this channel includes pin P510. Next, click on **the Pins > IRQ3** arrow to be taken to the pin configuration screen for IRQ3. Ensure that the pin is enabled, and if need be, select **Operation Mode > Custom** and **IO > IRQ3** to set **P510**.

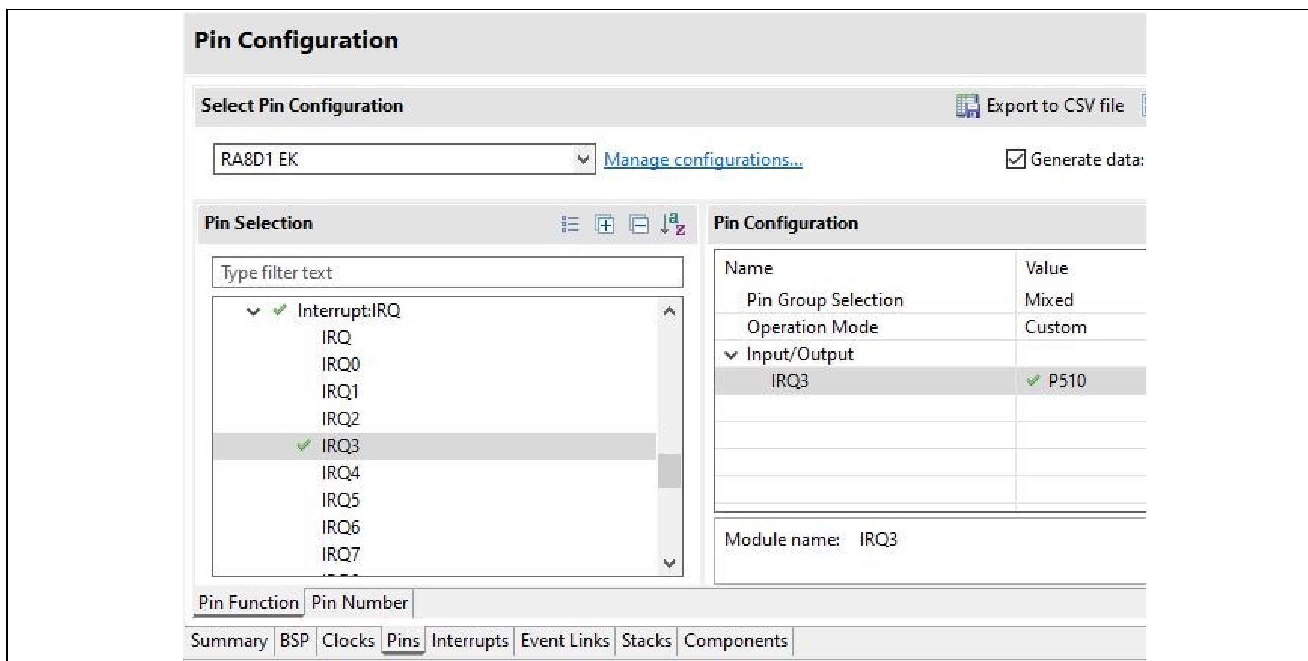

Figure 34. Ensure P510 is selected and enabled for I/O for IRQ3

Additionally, the `gt911.c` driver file includes an interrupt callback routine for handling when the GT911 IC triggers an interrupt to the MCU. When using these driver files, it's necessary to provide the callback routine definition using the same name as the driver, which is `touch_int_callback`.

GT911 I2C Master (r_iic_master)

The r_iic_master module is configured as the master host communicating via I2C to the slave GT911 IC peripheral. Channel 1 sets the right IIC channel for communication from the MCU to the display, and the **0x14** slave address corresponds to the GT911 slave address.

Like with the r_icu, the `gt911.c` file includes an interrupt callback routine for the I2C transaction completion, which provides the callback name `touch_i2c_callback`.

After enabling channel 1, the SCL1 pin should route to **P512,** and the SDA1 pin should route to **P511**. Use the pin configuration tab to set the pins if they don't populate automatically.

Real-time Clock (r_rtc)

The real-time clock is used to update the date and time data in the thermostat application each second. The r_rtc module is set with a 1-second period IRQ rate using the FSP APIs in the timer thread, and from the `timer_rtc_callback` routine, an emWin event will update the time & date variable.

General PWM Timer (r_gpt)

The general PWM timer is used in the thermostat application as the LCD's backlight signal, where the duty cycle directly correlates with the intensity of the display's brightness. On the RA8D1 MCU, the display's backlight enables (DISP_BLEN) maps to the input P404, so the r_gpt was set to Channel 3 and to GTIOB, which sets the PWM timer output to P404 and out to the display.

The PWM timer is set to be a sawtooth PWM wave with a 100-microsecond period. Other wave forms will work, but it must be a PWM wave, and the wave must have a period fast enough for the flicker to be undetectable to the human eye.

5. Running the Thermostat Application

This section explains how to get the Graphics Thermostat Application for the EK-RA8D1 up and running, including steps for setting up the hardware and building and running the project in e^2 studio.

5.1 Hardware Setup

(1) Attach the LCD to the MCU

Connect CN1 on the included LCD graphics expansion board to J58 on the EK-RA8D1, located towards the bottom of the kit on the underside. J58 on the EK-RA8D1 should also be labeled "MIPI GRAPHICS EXPANSION BOARD". If included, use the included screw to secure the display's connection.

Connect a micro-USB cable to the Debug J10 on the EK-RA8D1 and to the host PC.

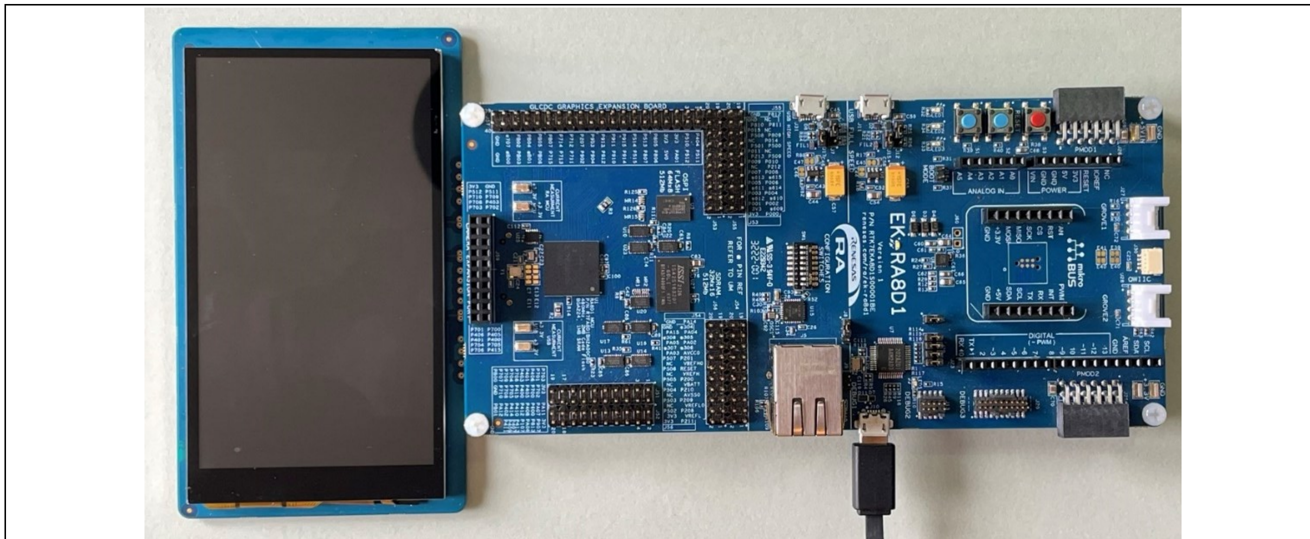

Figure 35. Connect the graphics expansion board to the EK-RA8D1 in portrait orientation

(2) EK-RA8D1 Configuration Switch (SW1) Settings

On the EK-RA8D1, there is a set of configuration switches (labeled SW1) that selects the operational peripheral pins on the board based on circuit groups. Refer to the board schematic for a complete understanding of the peripheral circuit groups.

Ensure that SW1 on the EK-RA8D1 has the settings listed in the table below:

Table 6. Required SW1 configurations for running the Thermostat Application Project

Location	Circuit Group	Setting	Function Restrictions
SW1-1	PMOD 1	OFF	PMOD1 connectivity conflicts with SDRAM
SW1-2	DEBUG TRACE	OFF	TRACE conflicts with ETH-A and SDRAM
SW1-3	CAMERA	OFF	CAMERA conflicts with ETH-B and I3C
SW1-4	ETHERNET A	OFF	ETH-A conflicts with ETH-B, TRACE and SDRAM
SW1-5	ETHERNET B	OFF	ETH-B conflicts with ETH-A, CAMERA, and I3C
SW1-6	GLCD (not MIPI)	OFF	GLCD conflicts with USB-HS
SW1-7	SDRAM	**ON**	SDRAM conflicts with TRACE, ETH-A, and PMOD1
SW1-8	I3C	OFF	I3C conflicts with ETH-B and CAMERA

Once properly set, SW1 on the EK-RA8D1 should look like the following image, with all switches OFF except for SW1-7 for SDRAM:

Figure 36. Enabling SW1-7 for SDRAM on the EK-RA8D1

5.2 Import, Build, and Run

Complete these steps to run and verify the Thermostat Graphics Application on your own EK-RA8D1:

1. Ensure that the application project folder *Thermostat_App_EK_RA8D1* is downloaded onto your host PC.
2. Follow the connection steps from Section 5.1.
3. Open an instance of e^2 studio IDE.
4. In the workspace launcher, either create or browse to the workspace location of your choice and select it.
5. In e^2 studio, navigate to **File** > **Import**.
6. In the Import dialog box select **General** > **Existing Projects into Workspace.**
7. Click **Select root directory** and use the **Browse…** button to point to the location of the *Thermostat_App_EK-RA8D1* folder. If the folder is zipped, click **Select archive file** and use the **Browse…** button to point to the zip folder location.
8. Make sure the option **Copy projects into workspace** is selected. Click **Finish**.
9. Open the *configuration.xml* file in the Project Explorer view and click **Generate Project Content** to create the remaining files required for the project.
10. Build the project. This step may take some time, as there are many files to build.
11. Debug and run the project. It also may take a while to load the project.
12. Interact with the LCD to observe the full thermostat application capabilities.

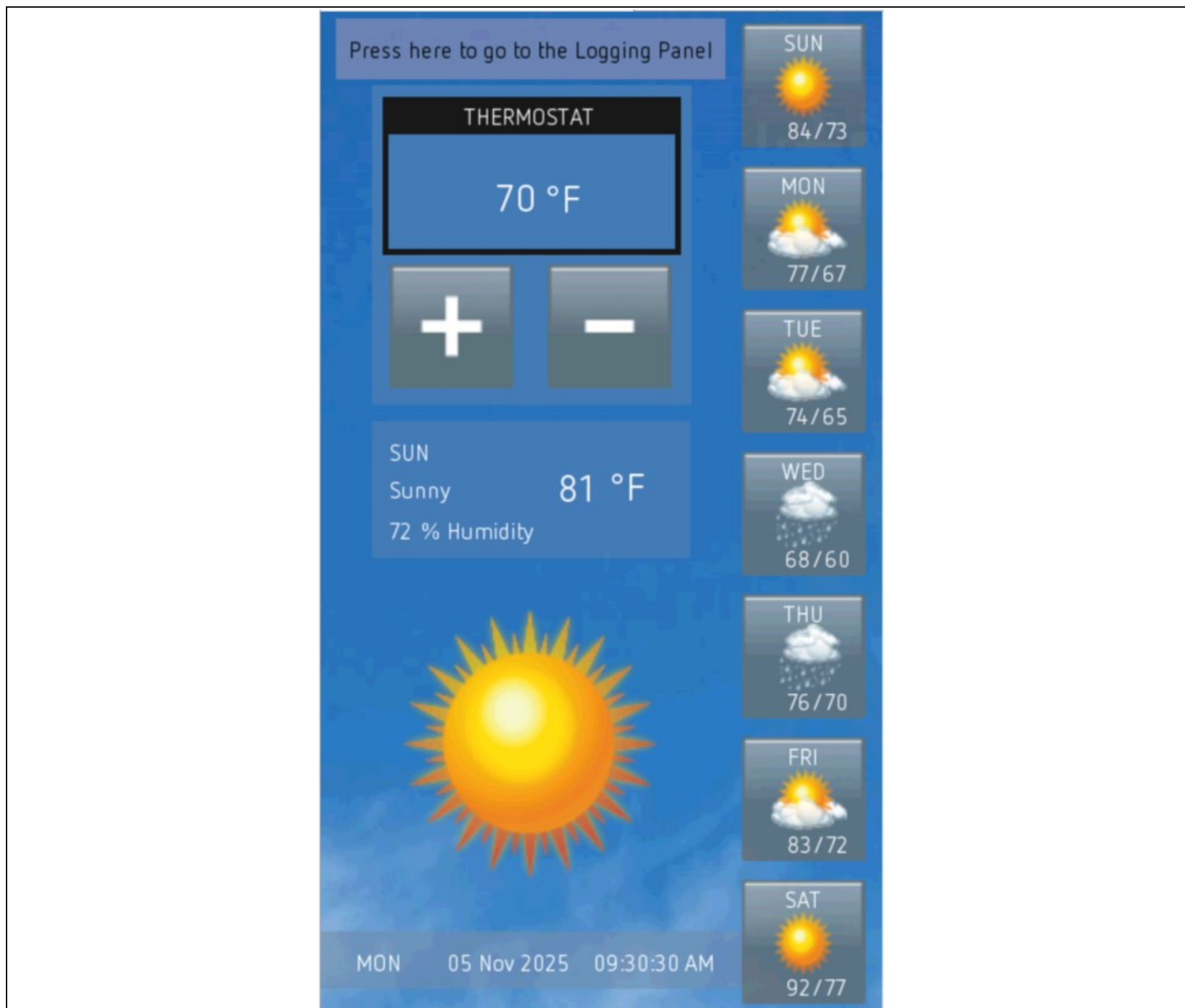

Figure 37. The Weather Panel screen is displayed after startup

6. Graphics Tradeoffs on the RA8D1

In all embedded graphics applications, realizing a best-case design is about finding a balance between various factors like resolution, color depth, framerate, bus width, and memory options to find the optimal performance sweet spot to suit your application needs.

To find the balance between the aforementioned graphic resources, it's integral that the application designer thoroughly understands the topology of the target MCU. They should have a deep knowledge of how the graphics framework functions on a high and low level, what internal and external memory resources are available, and how the bus architecture of the MCU may affect performance. Additionally, they should come prepared with a list of requirements and constraints for their system and relative priorities.

This section will examine various resource tradeoffs based on the hardware available on the RA8D1. Topics cover comparing MIPI DSI vs RGB interfaces, reviewing the memory options on the RA8D1, and understanding tradeoff relationships in the context of the RA8D1. The section concludes with a review of the design considerations when choosing the best-case design for the thermostat application.

This is not meant to serve as a replacement for the deep analysis and decision-making required when creating your own graphics application. It is meant to provide initial pointers before beginning to create a graphics application on the RA8D1.

6.1 MIPI DSI vs Parallel RGB

On RA8D1 MCUs, users have the option to interface via parallel RGB or MIPI DSI to drive external displays. The thermostat application demonstrates how to configure and operate the MIPI PHY and MIPI DSI modules

to send pixel data via MIPI to the external LCD. While parallel RGB communication is supported, the connection details are outside the scope of the application note but can be found in the RA8D1 Hardware User's Manual. This section, instead, will focus on the high-level differences and tradeoffs between the two graphics interfaces on RA8D1.

Take a look at the block diagram of the graphics subsystem on the RA8D1 in Figure 38 below. The graphics LCD controller (GLCDC) outputs the framebuffer pixel data via parallel RGB, denoted as LCD_DATA00-23. The parallel RGB data is routed to output pins on the MCU, and it is also routed as input to the MIPI DSI module. The MIPI subsystem converts the pixel data from parallel RGB to send it out of the MCU following the MIPI specification. It's important to be aware that the GLCDC is a strong candidate for a bottleneck since the graphics subsystem first routes through the GLCDC, whether it is output as MIPI or parallel RGB.

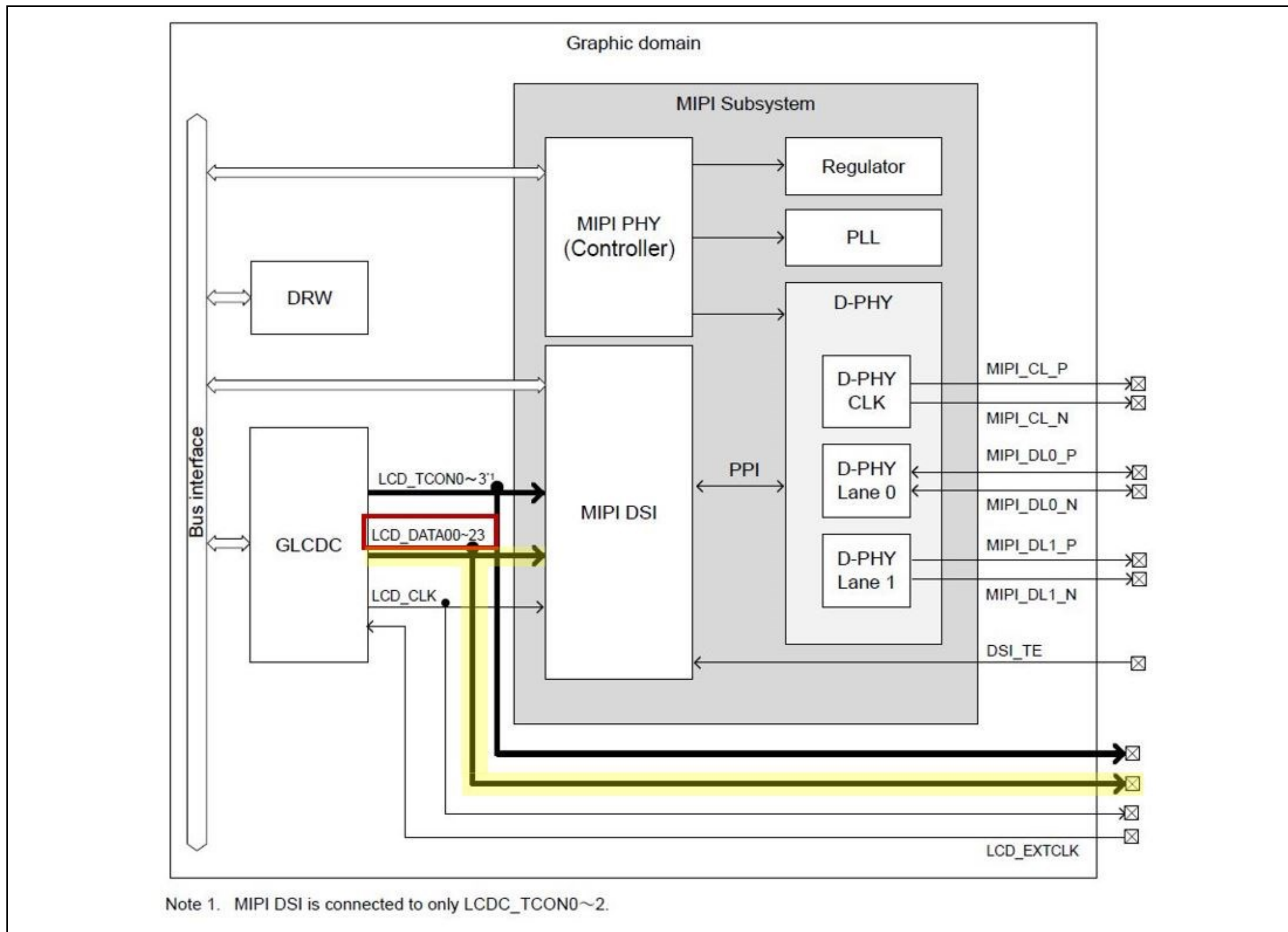

Figure 38. The GLCDC outputs parallel RGB data, routing to MCU output pins and to MIPI DSI input

MIPI DSI offers advantages in terms of higher data rates, simpler cable designs, lower power consumption, and better integration into compact systems. However, MIPI DSI may have a higher implementation cost and may not be suitable for all applications. Also, MIPI DSI requires high clock frequencies (high speed is 720 Mbps/lane), so countermeasures like reducing noise are needed. The parallel RGB interface is simpler and more cost-effective to implement but may be limited in terms of bandwidth, cable length, and power efficiency, especially for high-resolution displays. The choice between the two interfaces depends on the specific requirements and constraints of the display system.

Let's break down some of the technical aspects and compare how MIPI DSI and parallel RGB excel in their respective areas:

(1) Data Rate and Bandwidth:

MIPI DSI typically offers higher data rates compared to parallel RGB interfaces. MIPI DSI can achieve multi-gigabit per second data rates, enabling high-resolution and high-refresh-rate displays.

Parallel RGB interfaces have a limited bandwidth due to the clock rate available for data transmission. This limits the resolution and refresh rate that can be supported, especially for high-resolution displays.

This analysis applies generally, but please note that on the RA8D1 devices, since everything output by RGB or MIPI is first routed through the GLCDC, then the rate of the GLCDC output will be the determining factor for the final data rate. Additionally, on the RA8D1, the maximum achievable throughputs of the MIPI DSI and Parallel RGB interfaces are equivalent to one another.

(2) Cable Complexity and Length:

MIPI DSI uses a serial interface, which means fewer wires are needed for communication between the MCU and the display, resulting in simpler cable designs.

Parallel RGB interfaces require a larger number of wires (one for each color channel plus synchronization signals), which can lead to cable clutter and increased complexity, especially when driving high-resolution displays. Additionally, parallel RGB interfaces are limited in cable length due to signal degradation over longer distances.

While the RGB interface uses more pins and cables, it is the industry standard for graphics, and there will be more displays available that use a parallel RGB interface instead of a MIPI DSI interface.

(3) Power Consumption:

MIPI DSI typically consumes less power than parallel RGB interfaces due to its serial nature and ability to utilize lower-voltage signaling.

Parallel RGB interfaces may consume more power, especially at higher data rates, due to the need to drive multiple data lines simultaneously.

(4) System Integration:

MIPI DSI interfaces are commonly found in mobile devices and other compact systems where space is limited. The compact nature of MIPI DSI allows for easier integration into such systems.

Parallel RGB interfaces are more commonly used in larger display systems such as desktop monitors and TVs, where space constraints are less of an issue.

(5) Cost:

MIPI DSI interfaces may have a higher initial implementation cost due to the need for specialized hardware such as MIPI DSI controllers.

Parallel RGB interfaces are often simpler and more straightforward to implement, which can lead to lower initial costs. However, this might not hold true for high-resolution displays where the complexity of the interface increases.

6.2 Graphics Configuration Tradeoffs

Optimizing the overall configuration of an embedded graphics application involves carefully considering graphic trade-offs to meet the specific requirements and constraints of the target application. This section will generally discuss how resolution, color format, framerate, bus bandwidth, and size of internal SRAM all interact when trying to pick the optimal design. Any restrictions due to the constraints of the RA8D1 will be mentioned for each aspect.

6.2.1 Display Resolution

The resolution of a graphics application is based on the number of pixels on the target display. The developer will need to evaluate the target display and select an MCU with graphics hardware that can support the chosen display’s resolution. On RA8D1, the resolution is constrained by the GLCDC since it drives the pixel data output. The digital interface signal output supports video image sizes up to WXGA, or 1280x800 pixels. Default FSP configurations correspond to the external MIPI display included in the evaluation kit for the RA8D1, which is a portrait screen with a resolution of 480x854 pixels.

Higher resolution screens provide clearer images and can show greater visual details, but they also require more memory bandwidth and processing power, which can impact framerate and may necessitate a wider bus. Lowering the resolution can increase framerate by reducing the number of pixels that need to be processed and rendered per frame. Increasing the resolution increases the amount of data that needs to be transferred between components on the MCU, and processing may exceed the bandwidth capabilities of a narrower bus.

6.2.2 Color Format

The color format specifies the number of bits that represent the red, blue, and green information for each pixel on a display. RA8D1’s GLCDC supports the following pixel formats:

- RGB-888 progressive format (-: 8 bits, R: 8 bits, G: 8 bits, B: 8 bits; 32 bits in total)
- ARGB8888 progressive format (A: 8 bits, R: 8 bits, G: 8 bits, B: 8 bits; 32 bits in total)
- RGB565 progressive format (A: None, R: 5 bits, G: 6 bits, B: 5 bits; 16 bits in total)
- ARGB1555 progressive format (CLUT: 1 bit, R: 5 bits, G: 5 bits, B: 5 bits; 16 bits in total)
- ARGB4444 progressive format (A: 4 bits, R: 4 bits, G: 4 bits, B: 4 bits; 16 bits in total)
- CLUT8 progressive format (CLUT: 8 bits)
- CLUT4 progressive format (CLUT: 4 bits)
- CLUT1 progressive format (CLUT: 1 bit)
- CLUT memory: 512 words × 32 bits per graphics plane (ARGB8888)

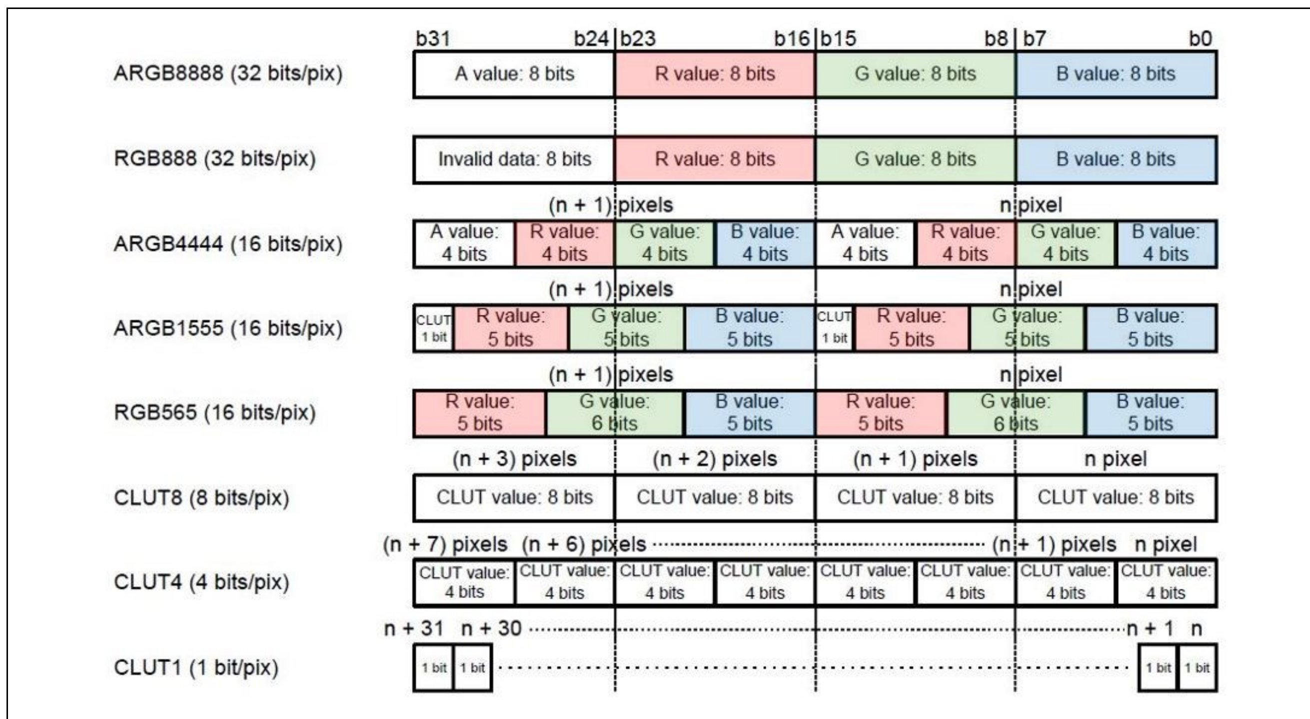

Figure 39. Pixel data formats of the RA8D1 GLCDC

The external MIPI display accepts pixel data input in the 24 bits per pixel (bpp) format RGB888. The default FSP configuration for the output of the GLCDC, therefore, is the 24bit RGB888 format. When 16 bpp or lower are selected for the framebuffer color depth, in the graphics framework, the pixel data is extended to 24bpp data before output from the GLCDC.

The quality of the final image displayed on the LCD depends on the lowest-quality color format used along the entire pixel data path. In addition to the display's color format, it's important to be aware of the color format of the image bitmaps and the format used by each step of the graphics framework to draw the framebuffers.

Higher-end graphics typically use either RGB565 (65k colors) or RGB888 (16.7M colors). Choosing between these formats involves a trade-off between color accuracy and resource consumption. Lower bpp formats, like RGB565, reduce the memory bandwidth and the overall processing time but may result in color banding and reduced image quality, especially in images with gradients.

In terms of memory, higher bpp formats, like RGB888, require more memory to store the pixel data, which can strain the available internal SRAM. In terms of processing time, they may achieve slower framerates due to the increasing amount of data that needs to be processed and transferred for each pixel. It's also critical to analyze if the graphic system's bus bandwidth and transfer rate can support the higher bpp color formats and the desired framerate. Section 6.4 analyzes the choice made for the color format in the context of the Thermostat Project.

6.2.3 Framerate

The framerate is measured in frames per second (FPS) and refers to the number of times the framebuffer is refreshed on the display. Unlike the resolution and color format, the framerate is not a pre-set value restricted by one component on the RA8D1. It depends on the processing capabilities of the graphics framework as a whole, including the software components like emWin. It's commonly stated that human eyes can detect flicks at around 24 fps or lower, so this is a good starting benchmark for the framerate of a graphics application.

Framerate speed is interrelated to all other features mentioned: the resolution, bpp, bus width, and SRAM. Higher framerates provide smoother animations and improve the user experience, especially in interactive applications. However, achieving higher framerates requires a tradeoff with the other aspects. Lowering the resolution can increase framerate by reducing the number of pixels that need to be processed and rendered per frame. Choosing a lower pixel depth can increase framerate by reducing the amount of data that needs to be processed and transferred for each pixel. Higher framerates often require fast access to graphics data, which can benefit from a larger internal SRAM and from a larger bus width but will increase the overall system cost.

Framerates also depend on the overall processing consumption of the CPU. When only LCD output is occurring, then all hardware resources, including the CPU, can be utilized for drawing. However, when several other system operations are utilizing the CPU, then the graphic hardware resources are strained and aspects like bus widths and SRAM usage will start to have a larger effect on the framerate. On the RA8D1, the D/AVE 2D hardware module helps to offload some of the drawing tasks from the CPU and increase overall performance.

6.2.4 Bus Width

Bus width refers to the number of bits that can be transmitted simultaneously between components in the graphics framework system. The bus width of the busses interfacing between the CPU, GLCDC, DRW, MIPI, and memory components like the SDRAM and SRAM all impact the system performance. Developers should evaluate the bus architecture of the target MCU to understand how the pixel data moves through the MCU and note any bottlenecks. The following image shows the bus map on RA8D1:

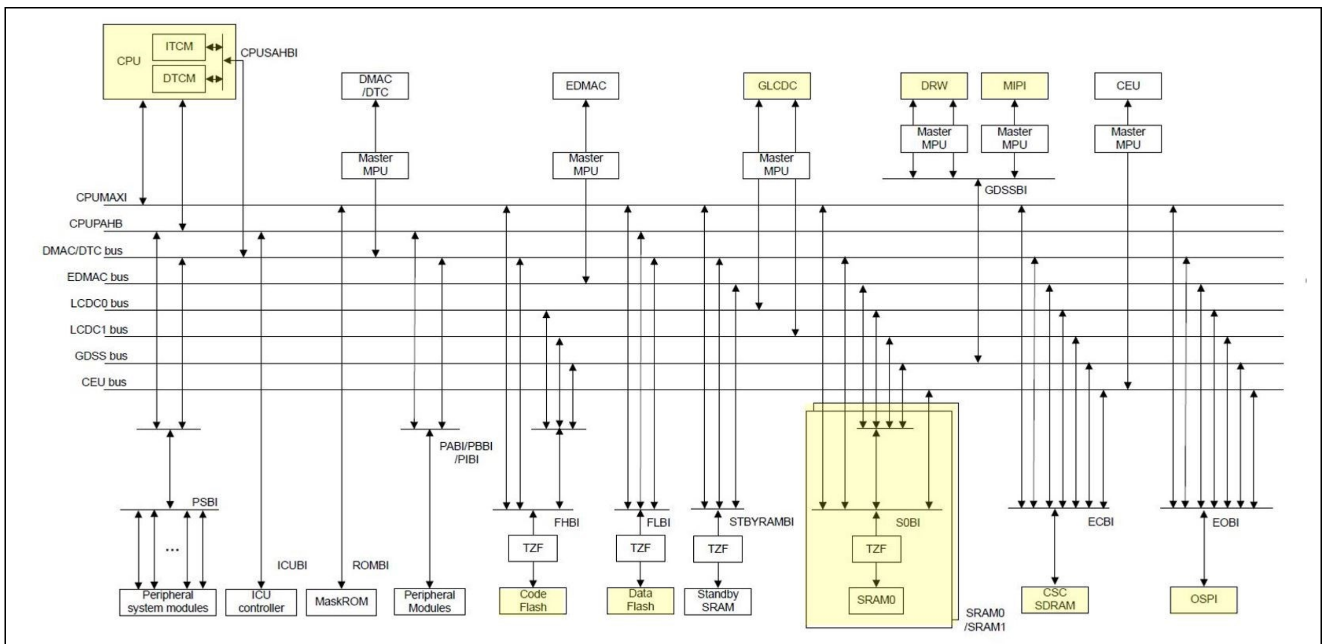

Figure 40. Bus map of the RA8D1 indicating commonly used modules for graphics applications

In the RA8D1 MCU group, there are devices with a 32-bit SDRAM bus and devices with a 16-bit SDRAM bus like that of the EK-RA8D1. For more information, visit the Buses section of the RA8D1 Hardware User's Manual.

A wider bus allows for faster data transfer rates, which can improve overall system performance. However, increasing the bus width may also require more power and board space, which can be limiting factors in embedded systems with size or power constraints. A wider bus can also improve memory bandwidth and, along with increased processing, can allow the system to handle higher resolutions and faster framerates more effectively.

6.2.5 Internal SRAM

The size of internal SRAM (Static Random-Access Memory) directly affects the amount of memory available for storing graphics data and processing intermediate results. On RA8D1, the total SRAM is 1MB, and it is broken into subregions: SRAM with ECC (384kB), SRAM with Parity (512kB), and TCM (128kB) with ECC.

Since there's no hardware JPEG decoder on RA8D1 MCUs, if JPEG images are being used, it's highly recommended that software decoding in the SRAM region is performed at the fastest speeds. In general, the SRAM region is a great candidate for drawing the framebuffer(s) and performing other intensive processes because it reduces the need to access slower external memory during rendering. Note the emWin library supports Helium-accelerated instructions for software decoding of JPEGs.

If you use the SRAM to render the framebuffer(s), then the color format and resolution are directly tied to the required size of the SRAM. As an example, the following shows the calculation comparison for the size of a double framebuffer in SRAM between 32-bit RGB888 and 16-bit RGB565 with a 480x854 pixel resolution:

2 (framebuffers) X 480x854 (pixels) X 32 bpp / 8 = 3.28 MB

2 (framebuffers) X 480x854 (pixels) X 16 bpp / 8 = 1.64 MB

A larger SRAM allows for buffering of graphics data and intermediate results, reducing the need to access slower external memory during rendering. This, in turn, helps to achieve faster framerate performance, but it also adds to the cost and complexity of the system.

6.3 RA8D1 Memory Options

When creating a graphics application on an embedded system, developers need to carefully consider how to manage the system's memory usage. This includes where to store the permanent graphic object data for the code to process and where to store the framebuffer(s) the graphics subsystem draws at runtime. There may be additional memory considerations depending on the target application, like determining where to store software-decoded JPEGs and provisioning memory resources to services like emWin and FreeRTOS. By selecting the appropriate memory hardware and optimizing memory usage, developers can achieve efficient performance and meet the requirements of their applications.

Typically, a developer will know the end application's target resolution and color format and know the graphic content (total image sizes) required to create the end application. Then, the optimization process involves choosing the right MCU and external memory devices to meet system requirements. Other times, the order is reversed, the hardware and its cost will be the design priority, and tradeoffs will need to be made for the other application features.

In either case, once the target MCU has been chosen, it's essential to evaluate the available memory hardware options on the MCU and be aware of the connection options for adding additional external memory devices, if needed.

The RA8D1 supports the following memory options:

- 1MB/2MB Code Flash option
- 12kB Data Flash
- 1MB internal SRAM
- 32kB Instruction and Data Caches
- 512MB external Octo-SPI Flash
- 128MB external SDRAM with 16/32-bit external memory bus interface

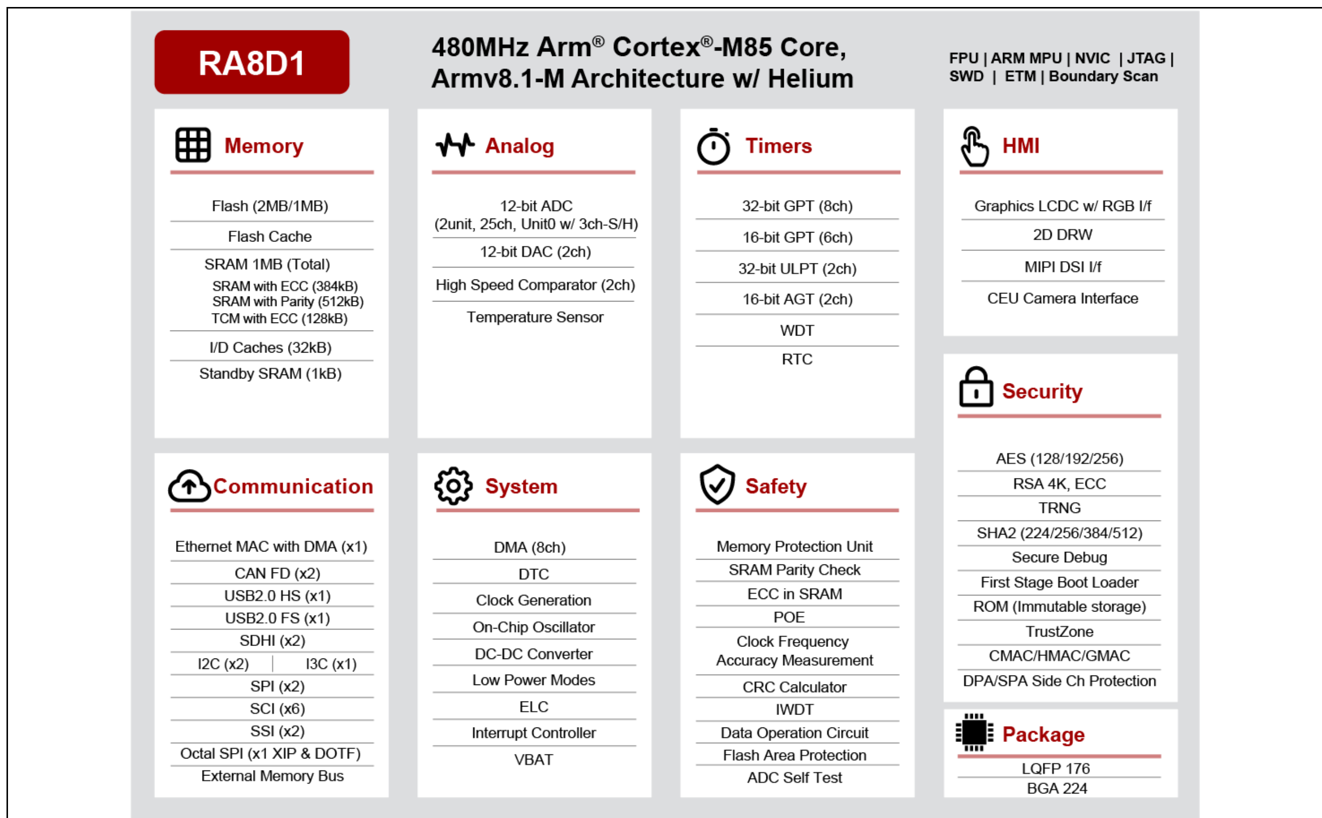

Figure 41. Summary of RA8D1 options

6.3.1 EK-RA8D1 Memory Devices

The EK-RA8D1 has the following memory devices:

- 2MB Code Flash
- 12kB Data Flash
- 1MB internal SRAM
- 32kB Instruction and Data Caches
- 64MB external Octo-SPI Flash
- 64MB external SDRAM with a 16-bit bus interface

As concluded in the previous section, it's impossible to discuss memory considerations without mentioning how memory is affected by other graphic application factors like screen resolution, color format, and framerate.

Sometimes, you may need to decide whether to place the framebuffer in SRAM or SDRAM. The decision depends on the framebuffer size (directly related to the resolution and bpp) and the presence of other resources requiring SRAM memory. If there are time-intensive processes like JPEG decoding, they should be prioritized to happen in the faster SRAM region. Other programs like FreeRTOS and emWin may also need to use SRAM and take up some memory bandwidth.

Higher resolutions and larger color formats increase the footprint of the JPEG framebuffers and the display framebuffers. If JPEG framebuffers and other processes take up all the limited SRAM space, then the display framebuffers may need to be kept on external memory like SDRAM, flash, or in the Octal Serial Peripheral Interface (OSPI) area. Also, a double framebuffer scheme stores twice as much pixel data when compared to the single framebuffer scheme, but the benefit of a double buffer is achieving faster framerates and smoother visual responses.

Note that if you are using SDRAM for framebuffers, there is a notable difference between the 32-bit vs the 16-bit width of the external SDRAM bus. The EK-RA8D1 is limited to the 16-bit SDRAM bus, which simply cannot support the bandwidth requirements for pixel data with the RGB888 data format when the output resolution is 480x854.

When you can, keep the display framebuffers stored in internal SRAM, and use the OSPI area or flash region to store any other graphic objects/bitmaps for the best performance. Adding external memory will increase the system cost, but more memory can support uncompressed graphic formats and more complex graphic applications that have a larger number of image objects because you are no longer limited by memory footprint. If you have the space for them, it’s recommended to use uncompressed image formats over compressed formats, like choosing PNG bitmaps over JPEG bitmaps, because you will achieve faster framerates without needing to decode the bitmaps at runtime.

Enabling the Data Cache assists in the graphics system data processing and rendering to achieve faster framerates. The FSP includes data cache maintenance instructions through the RA emWin port to ensure deterministic behavior and no visual artifacts.

6.4 Thermostat Application Best Case Design

The best-case design for your graphics application depends on the system’s pre-defined constraints and final image requirements. Then, the method is to manage the remaining resource tradeoffs based on priority to achieve the desired final application. This section provides insight into the tradeoff considerations leading to the final thermostat application design.

Before beginning to create the thermostat application’s e^2 studio project, the system’s software components were decided, and the graphic objects making up the thermostat GUI application were defined. It was decided that the project would use AppWizard, emWin, and FreeRTOS components. The image contents of the AppWizard thermostat GUI were also pre-defined. It was known that the application would require at least three full-screen weather background images, various weather graphics, button images, logo graphics, buttons, a slider and rotary, font information, etc.

The final image requirements (resolution, color format, and framerate) of the thermostat application were affected by the target MCU’s memory and bus width and by the LCD’s specifications. Using the external MIPI display with the EK-RA8D1 meant that the resolution was fixed at 480x854 pixels. The GLCDC is specified to support both RGB888 and RGB565 inputs, but both the memory usage requirements and 160-bit external memory bus width shaped the decision to use RGB565 as the color format for the pixelmap input. Note that the GLCDC internally extends the pixelmap color format to RGB888 for processing, drawing the framebuffer, and outputting the final image to the LCD.

One goal was to avoid adding external flash memory to minimize the cost of running a demonstration of the application project. On the EK-RA8D1, the code flash is limited to 2MB. For one full-screen image stored as an uncompressed bitmap, like PNG, using the RGB888 color format, the bitmap requires about 1.64 MB of memory, and the RGB565 requires half that memory space at 0.82 MB. It was only possible to fit all the graphics objects required for the thermostat application inside the internal code flash if they were formatted with RGB565.

The figure below shows the **Memory Usage** view of the thermostat application in e^2 studio, which is a great tool for analyzing the memory usage of the resources allocated in the C project by region. The top right panel **Memory Region Usage** shows that Flash is 89% full, and the bottom panel’s Object tab enumerates the object files in Flash in descending order by size. The green highlights the objects, which are mostly animated images and smaller graphics stored as PNG bitmaps. Because PNGs are uncompressed and require no additional processing to draw, they help achieve the fastest framerates possible during the animated periods of the application. The orange highlights the three full-sized background image objects, which are stored as compressed JPEG bitmaps. Being stored as JPEGs enabled them to fit inside the flash region but at the cost of requiring SRAM to decode the buffers fast enough to achieve acceptable framerates. The blue highlights the font resources allocated by AppWizard. With the JPEG buffers in SRAM, there was not enough space for the double framebuffers, so they were placed in the SDRAM region.

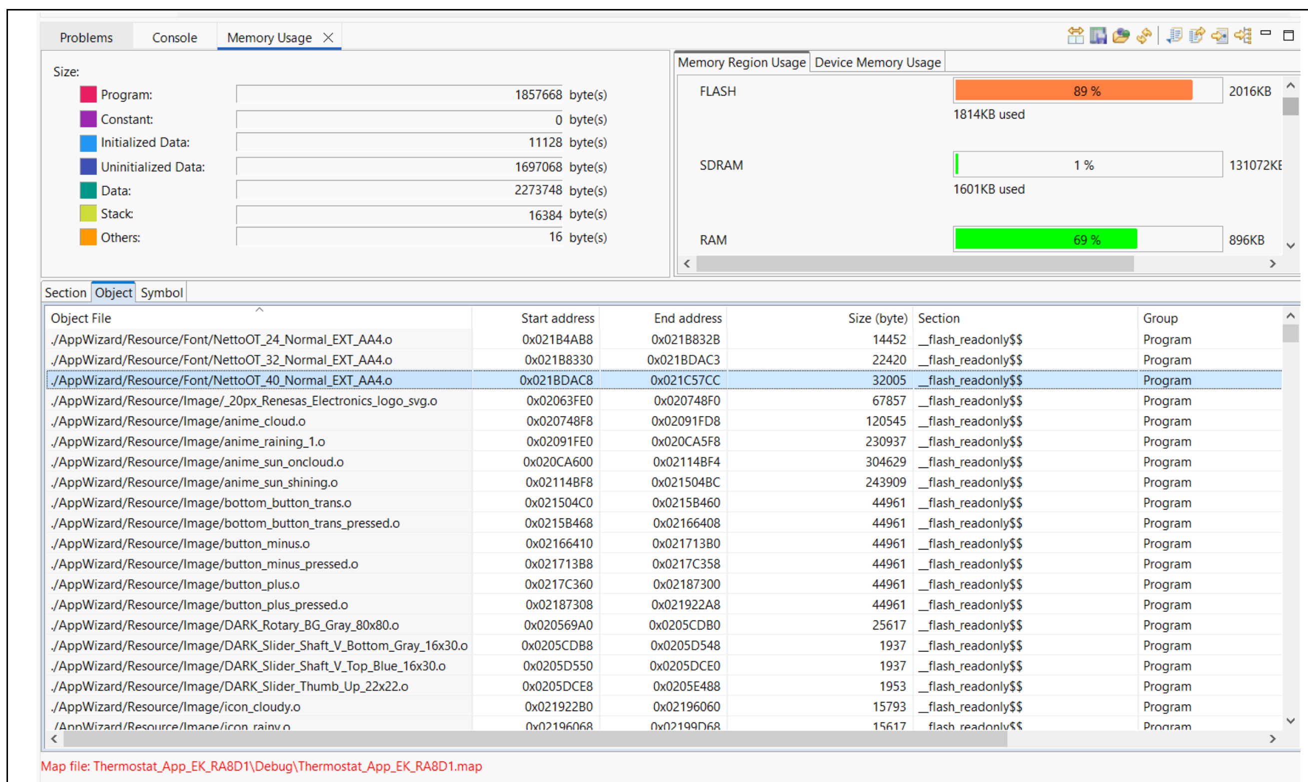

Figure 42. The Memory Usage tab lists memory details about the project's resources

Since there are no gradients or subtle color changes in the images of the thermostat application, there is no discernible difference in image quality between RGB565 and RGB888. However, there is a significant difference in the system resource consumption and achievable framerates. And because the framebuffers are stored in SDRAM, which is limited to a 16-bit bus on the EK-RA8D1, the bus throughput can only handle RGB565 at the highest format to avoid a GLCDC underflow and visual artifacts.

The D-Cache is enabled in the FSP project properties to assist in data processing and rendering and achieve faster frame rates. The FSP includes data cache maintenance instructions through the RA emWin port to ensure deterministic behavior and no visual artifacts. The D-Cache was measured to improve the framerate speeds by a factor of 3 in the thermostat application. The values were measured using the DWT cycle counter to measure the time between framebuffer refreshes in `LCDConf.c`.

```
uint32_t t_samp[1000] = {0};
uint32_t count = 0;

* _SwitchBuffersOnVSYNC
static void _SwitchBuffersOnVSYNC (int32_t Index)
{
    uint32_t enable_mask = 1;

    //Check if the cycle counter is enabled, and if not, enable it
    if(!(DWT->CTRL & enable_mask))
    {
        DWT->CTRL |= enable_mask;
    }
    if(count<1000)
    {
        t_samp[count] = DWT->CYCCNT;
        ++count;
    }
    else count = 0;

    //
    // Swap buffer
    //
    fsp_err_t err;
    do
    {
        err =
            R_GLCDC_BufferChange(_g_display_emwin->p_ctrl,
                                 (uint8_t *) pp_buffer_address[Index],
                                 (display_frame_layer_t) 0);
        if (err)
        {
            vsync_wait();
        }
    } while (FSP_ERR_INVALID_UPDATE_TIMING == err);

    //Set counter to 0
    DWT->CYCCNT = 0;

    GUI_MULTIBUF_ConfirmEx(0, Index);  // Tell emWin that buffer is used

    vsync_wait();
}
```

Figure 43. Example of using the DWT to calculate framerate in the Thermostat Application

Figure 43 shows an example of using the DWT in the `_SwitchBuffersOnVSYNC()` function to determine the time it takes to refresh the framebuffer. The values are stored in the run-time variable `t_samp`, which can be viewed during debugging in the Expressions View.

7. References

Renesas:

- RA8D1 MCU Group Hardware User's Manual: https://www.renesas.com/us/en/document/mah/ra8d1-group-users-manual-hardware
- EK-RA8D1 Quick Start Guide: https://www.renesas.com/us/en/document/qsg/ek-ra8d1-quick-start-guide
- FSP User's Manual (most recent version): https://renesas.github.io/fsp/index.html

EK-RA8D1 LCD:

- MIPI Graphics Expansion Board (E45RA-MW276-C) Datasheet: focusLCDs.com/specs/E45RA-MW276-C
- TFT LCD Controller (ILI9806E) Datasheet: https://focuslcds.com/wpcontent/uploads/Drivers/ILI9806E.pdf
- Capacitive Touch Panel GT911 Controller Datasheet: https://www.crystalfontz.com/controllers/GOODIX/GT911/
- Capacitive Touch Panel GT911 Controller Programming Guide: https://www.crystalfontz.com/controllers/GOODIX/GT911ProgrammingGuide/478/

SEGGER:

- AppWizard User Guide & Reference Manual: https://www.segger.com/downloads/emwin/UM03003_AppWizard.pdf
- emWin User Guide & Reference Manual: https://www.segger.com/downloads/emwin/UM03001

FreeRTOS:

- FreeRTOS Reference Manual v10.0.0: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf

RA8 Related Application Notes:

- EK-RA8D1 Example Project Bundle: https://www.renesas.com/us/en/document/apn/ek-ra8d1-example-project-bundle
- GUIX Hello World for EK-RA8D1 MIPI Display: https://www.renesas.com/us/en/document/apn/guix-hello-world-ek-ra8d1-mipi-lcd-display
- RA8 Quick Design Guide: https://www.renesas.com/us/en/document/apn/ra8-mcu-quick-design-guide
- Getting Started with RA8 Memory Architecture, Configurations and Topologies: https://www.renesas.com/us/en/document/apn/getting-started-ra8-memory-architecture-configurations-and-topologies

Website and Support

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

RA Product Information	www.renesas.com/ra
RA Product Support Forum	www.renesas.com/ra/forum
RA Flexible Software Package	www.renesas.com/FSP
Renesas Support	www.renesas.com/support

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jul.18.24	—	Initial version
1.10	Dec.02.24	—	Minor update for AppWizard v1.52_6.44b + FSP V5.6.0
1.20	Jan.09.26	—	Updated to FSPv6.2.0

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan

www.renesas.com

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/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

© 2025 Renesas Electronics Corporation. All rights reserved.