Renesas RA Family

Getting Started with the Graphics Application

Introduction

This application note describes the development of a graphics application that targets the RA8D2 MCU and runs on MIPI graphics expansion board with the RA8D2 evaluation kit (EK). It provides a comprehensive overview of key concepts and implementation techniques for building high-performance Human Machine Interface (HMI) applications using the RA8D2 MCU. The application note covers a variety of graphics-related topics, including:

- Interfacing with peripherals using the MIPI options on the RA8D2.
- Designing a GUI for the EK-RA8D2 MIPI 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, dual-core and FreeRTOS.
- Communicating with the EK-RA8D2 MIPI LCD's capacitive touch panel using custom-written drivers.
- Evaluation of design tradeoffs for graphics applications in the context of the RA8D2 dual core 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 through a single-use case. The application project is a dual-core, multi-threaded graphic application designed in portrait-orientation that runs on the EK-RA8D2 and interfaces as a MIPI DSI-2 Host to the external MIPI LCD panel.

The application's Graphical User Interface (GUI) was designed using the SEGGER AppWizard tool (hereinafter referred to as AppWizard), and it is integrated into an e² studio project for the EK-RA8D2, using the RA Flexible Software Package (FSP) that natively supports SEGGER emWin (hereinafter referred to as emWin), dual-core processing and FreeRTOS, enabling a scalable and efficient foundation for advanced HMI application development.

Target Device

EK-RA8D2

Required Resources

Development tools and software

- e² studio v2025-10
  e² studio | Renesas
- Renesas Flexible Software Package (FSP) v6.2.0
  RA Flexible Software Package (FSP) | Renesas
- AppWizard V1.56_6.48
  SEGGER emWin GUI Library for Renesas RA Products | Renesas
  Note: The version emWin in FSP must match the emWin version in the SEGGER AppWizard. For example, the emWin version 6.44.2 in FSP is equivalent to 6.44b in the AppWizard V1.52_6.44b, and so on.

Hardware

- Renesas EK-RA8D2 kit (RA8D2 MCU Device Group): www.renesas.com/ek-ra8d2
- MIPI Graphics Expansion Board Version 1: MIPI Graphics Expansion Board 2 v1

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 RA8D2 MCU Device Group.

Contents

1. Overview 5
1.1 RA8D2 Device Overview 5
1.1.1 Key Graphics Features of the RA8D2 MCU 5
1.1.2 RA8D2 Evaluation Kit 6
1.1.3 EK-RA8D2 MIPI Graphics Expansion Board 7
1.2 System Overview 8
1.2.1 CPU0 Task Allocation and Responsibilities 8
1.2.2 CPU1 Task Allocation and Responsibilities 8
1.2.3 Graphics Application Layout 9

2. GUI Development with AppWizard 9
2.1 AppWizard and emWin Capabilities 9
2.2 Creating a New AppWizard Project 10
2.2.1 Including AppWizard Project Files in e^2 studio Project 11
2.2.2 AppWizard/emWin Initialization 14
2.3 Custom Designs in AppWizard 14
2.4 Setup AppWizard Interactions 15
2.4.1 AppWizard Defined Interactions 15
2.4.2 User-Defined Slot Code 16
2.4.3 Responding to AppWizard Variables 17
2.5 Add emWin Widget to AppWizard Project 17

3. Thermostats Graphics Application 18
3.1 Source Code Layout 18
3.2 System Design and Operation Flow 21
3.3 Dual-Core Architecture and Component Identification. 22
3.4 Module, Pin and Clock Configuration 24
3.4.1 IPC & FSP Solution Clock Configuration. 25
3.4.2 CPU0 Module Configuration. 26
3.4.3 CPU1 Module Configuration. 31
3.5 Configuring the MIPI Graphic Expansion Board 31
3.6 GT911 Touch Drivers 32
3.7 Placing Graphic Resources in External Flash Memory 34
3.8 System Performance Enhancement 37
3.8.1 Utilizing Cortex®-M85 Core Data Cache 37
3.8.2 Utilizing Helium on Cortex®-M85 for JPEG Decode 37
3.8.3 Leveraging 32-bit SDRAM Bus for High-Speed Framebuffer Access 37

4. Running Thermostats Application 39
4.1 Hardware Setup 40
4.1.1 Attach the MIPI LCD to the MCU 40

4.1.2 EK-RA8D2 Configuration Switch (SW4) Settings 40
4.2 Importing and Building the Project 40
4.3 Downloading and Executing on the EK-RA8D2 Kit 41

5. Graphics Implementation Considerations and Trade-offs 41
5.1 MIPI DSI vs Parallel RGB 42
5.1.1 Data Rate and Bandwidth: 43
5.1.2 Cable Complexity and Length: 43
5.1.3 Power Consumption: 43
5.1.4 System Integration: 43
5.1.5 Cost: 43
5.2 Graphics Configuration Tradeoffs 43
5.2.1 Display Resolution 43
5.2.2 Color Format 44
5.2.3 Framerate 45
5.2.4 Bus Width 45
5.2.5 Internal SRAM 46

6. Introducing QE for Display Application Development 46
6.1 Installation and Uninstallation 47
6.1.1 Install from the "Renesas Software Installer" menu of e^2 studio 47
6.1.2 Install using QE (zip file) downloaded from the Renesas website 47
6.1.3 Uninstalling QE Product 47
6.2 Development Step with RA device. 48

7. References 55

8. Website and Support 56

Revision History 57

1. Overview

One of the key goals of the provided graphics application is to demonstrate how to build applications that require complex HMI screens using AppWizard and emWin library. The following list summarizes its main features.

- Complex GUI design created with AppWizard.
- External Octal OSPI Flash for image storage.
- Multi-threaded applications framework based on FreeRTOS.
- Dual-core architecture for efficient workload distribution.
- Display using graphics LCD controller and MIPI DSI interface.
- I²C-based GT911 touch controller driver.

There can be many ways to achieve the target design, and the approach described in this application note is one possible solution.

1.1 RA8D2 Device Overview

The RA8D2 device group is designed for high-performance HMI and Vision AI applications, combining advanced processing power with robust security features such as cryptographic Security IP, immutable storage for the first-stage bootloader (FSBL), a secure boot, and tamper protection for a truly secure IoT device.

1.1.1 Key Graphics Features of the RA8D2 MCU

The RA8D2 MCUs integrate the high-performance Arm® Cortex®-M85 core running up to 1 GHz and Arm® Cortex®-M33 core running up to 250 MHz, and 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, 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 fetching, processing, and format conversion, and it can interface to camera sensors up to 5M pixels. The on-chip 2MB 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.

RA8D2 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.

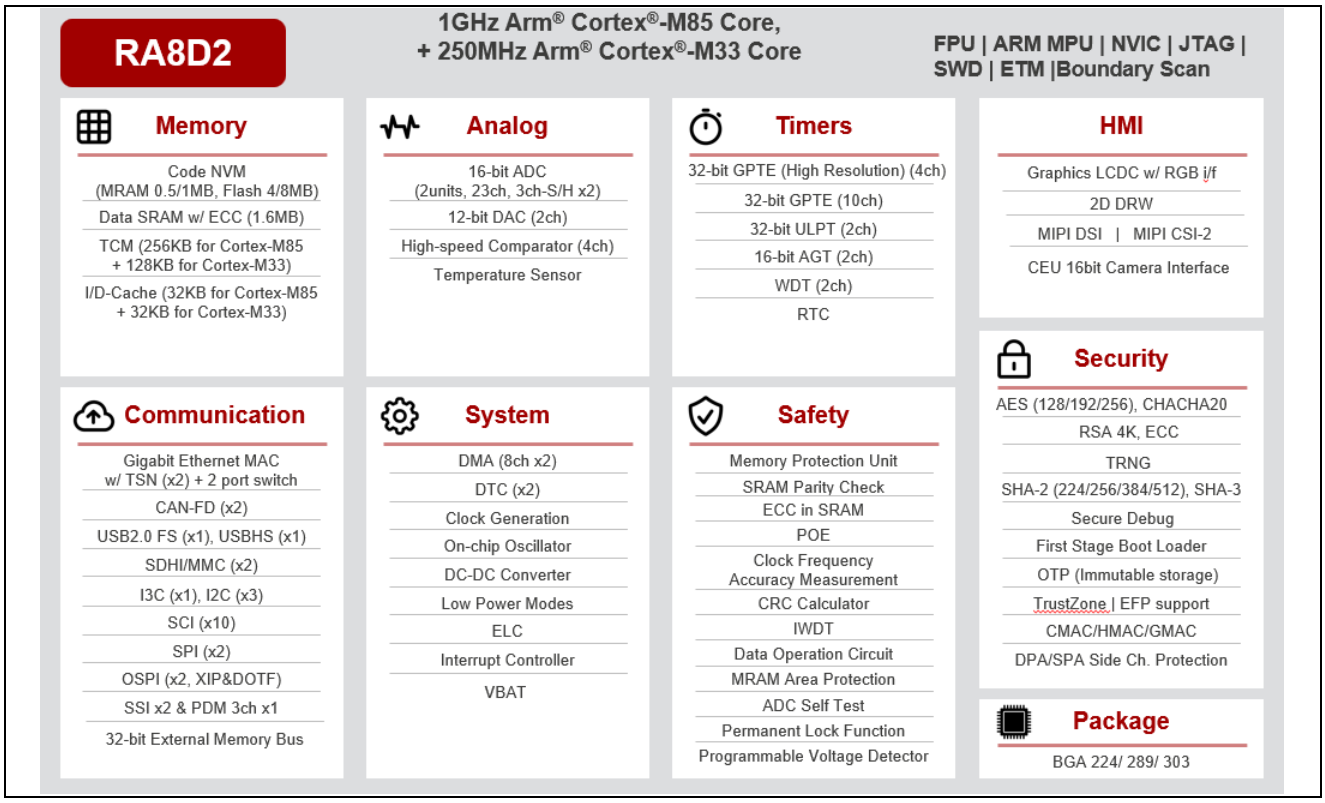


Figure 1. Block Diagram of Key Features in RA8D2 MCU

1.1.2 RA8D2 Evaluation Kit

The EK-RA8D2, an evaluation kit for the RA8D2 MCU Group, enables users to evaluate the features of the RA8D2 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 their required interconnections: the EK-RA8D2 board featuring the RA8D2 MCU with an on-chip Graphics LCD Controller (GLCDC), the Parallel Graphics Expansion Board featuring a 1024×600 TFT LCD with a capacitive touch panel overlay, and the Camera Expansion Board (OV5640) featuring a 5-megapixel CMOS image sensor.

By default, the kit is supplied with the Parallel Graphics Expansion Board using the RGB interface. However, the EK-RA8D2 board also supports MIPI-DSI display connectivity through the MIPI graphics expansion port (J32) as seen in Figure 2, enabling the use of a MIPI Graphic Expansion Board that can be purchased from the Renesas website.

In this application, the development example demonstrates how to build a graphics application using the MIPI LCD screen, showcasing the configuration and performance of the MIPI-DSI interface on the EK-RA8D2 platform.

The EK-RA8D2 board comes pre-programmed with a Quick Start example project. Please refer to the EK-RA8D2 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-RA8D2, check out the EK-RA8D2 Example Projects Bundle document, which can also be found on the Renesas website.

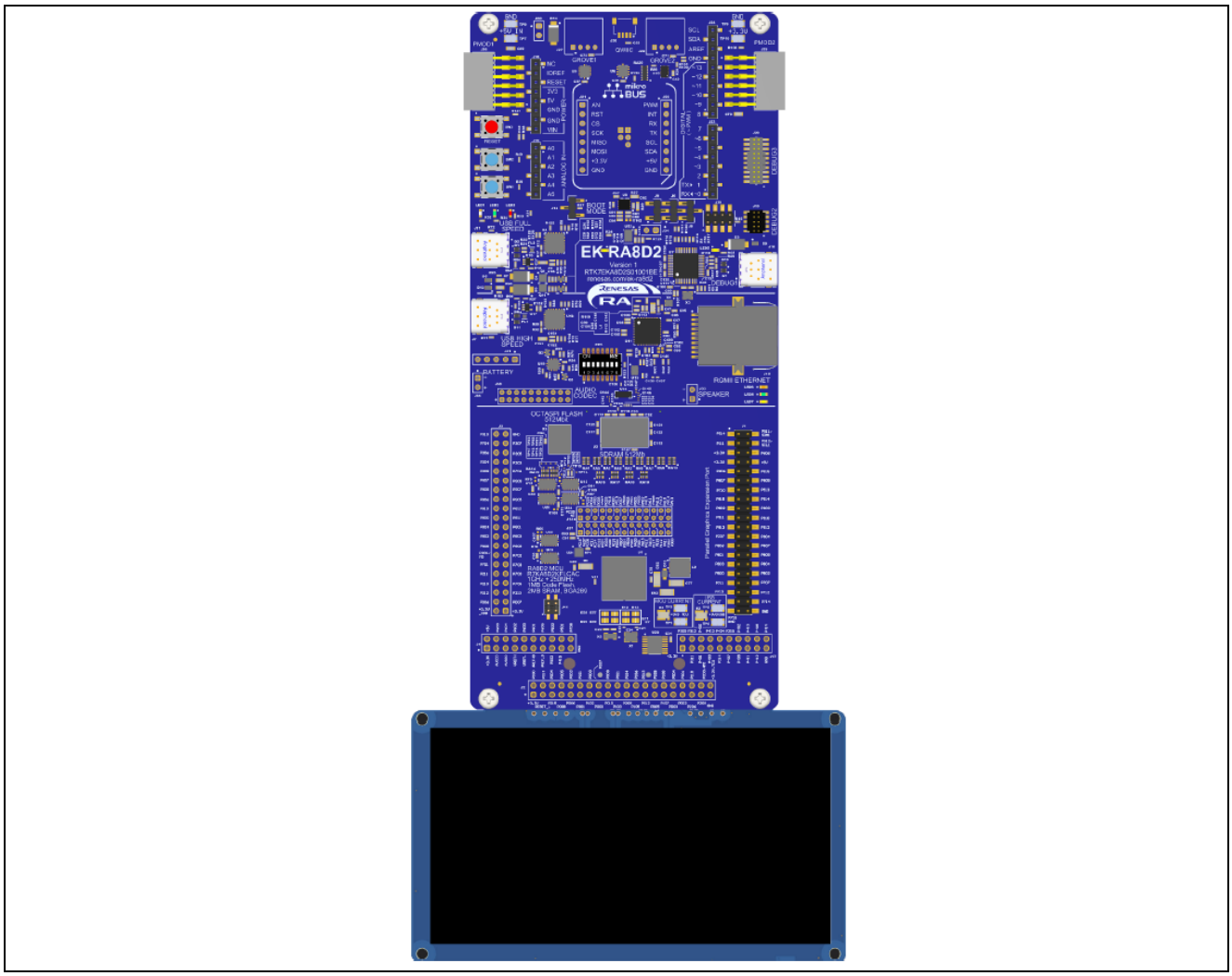


Figure 2. RA8D2 Evaluation Kit with MIPI Graphic Extension Board Version 1

1.1.3 EK-RA8D2 MIPI Graphics Expansion Board

The MIPI Graphics Expansion Board is originally supplied with the EK-RA8D1 kit, it is fully compatible with the EK-RA8D2 and is used in this application for MIPI-DSI display evaluation. The TFT LCD on the expansion board provides the following key features:

- Display type: TFT LCD with capacitive touch panel (CTP) overlay and backlight control
- Diagonal size: 4.5 inch
- Dimensions: 120 mm (width) x 90 mm (length)
- 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 MIPI graphics expansion board is portrait orientation, this application note will only cover designing and drawing a portrait orientation application.

1.2 System Overview

With dual-core architecture, graphic applications can take advantage of parallel processing where one core handles intensive rendering tasks while the other manages system control or user interaction. This division of workload not only improves overall performance but also enhances responsiveness and ensures a smoother user experience.

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.

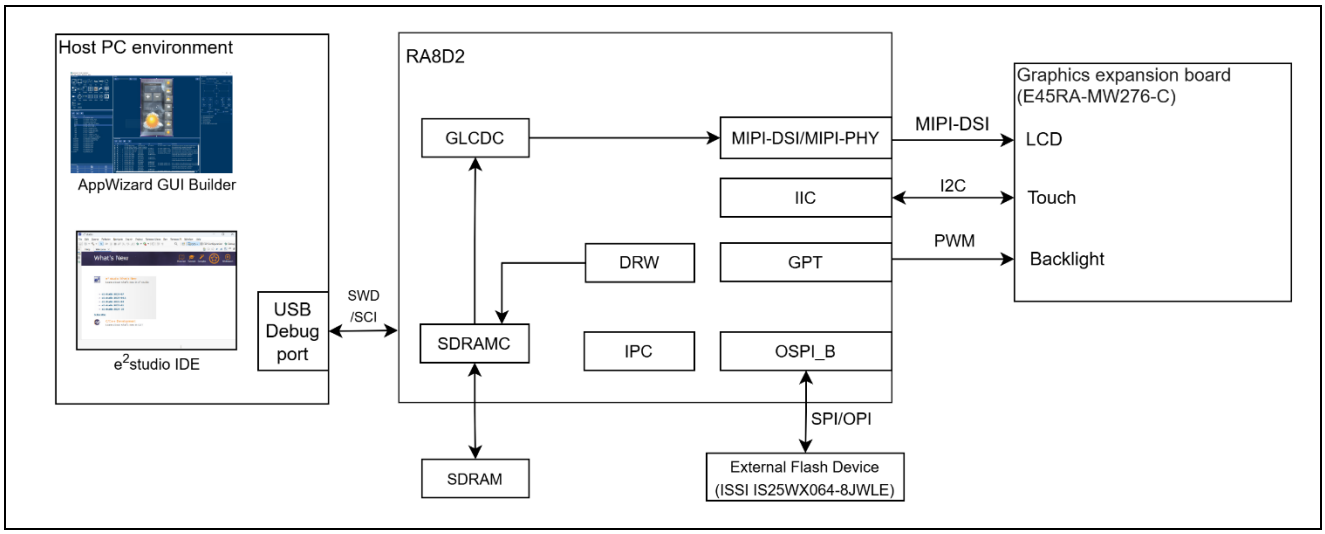


Figure 3. Block Diagram of Dual-Core Graphic Application

1.2.1 CPU0 Task Allocation and Responsibilities

In this architecture, CPU0 serves as the Display Master, taking full responsibility for graphical output and display control to deliver optimal visual performance with low-latency frame rendering. By isolating these compute-intensive tasks, the system ensures a smooth and stable User Interface (UI) experience.

CPU0 executes the main graphics render loop of the application, which integrates both AppWizard auto-generated sources and custom rendering logic. It manages all draw calls, texture and image resources, and buffer swapping to sustain the target frame rate. In addition, CPU0 manages graphics memory usage, including framebuffer allocation, display synchronization, and access to external Flash image resources.

As part of display bring-up, CPU0 performs MIPI LCD initialization, including execution of the MIPI command table to configure the panel. It also manages backlight control directly, enabling synchronized brightness adjustments with ongoing display updates. This localized handling minimizes Inter-Processor Communication (IPC) overhead and provides a seamless, responsive user experience.

1.2.2 CPU1 Task Allocation and Responsibilities

The CPU1 core is designated as the Peripheral Master, exclusively handling all system I/O, core control logic, and driver-level processing. Its primary role is to offload time-critical interrupt processing and non-graphical management tasks from CPU0, thereby ensuring optimal responsiveness and application stability.

CPU1 is responsible for acquiring and processing raw data and system events. After performing the necessary computations or data interpretation, it transmits the resulting high-level events to CPU0 through the Inter-Processor Communication (IPC) mechanism for corresponding updates and actions within the graphical user interface (GUI).

1.2.3 Graphics Application Layout

The graphics application consists of two graphical panels: a **Weather Panel** and a **Logging Panel**. In this application, we build separate static display designs for these two panels. The screen resolution on the EK-RA8D2 kit is 480 x 854 pixels.

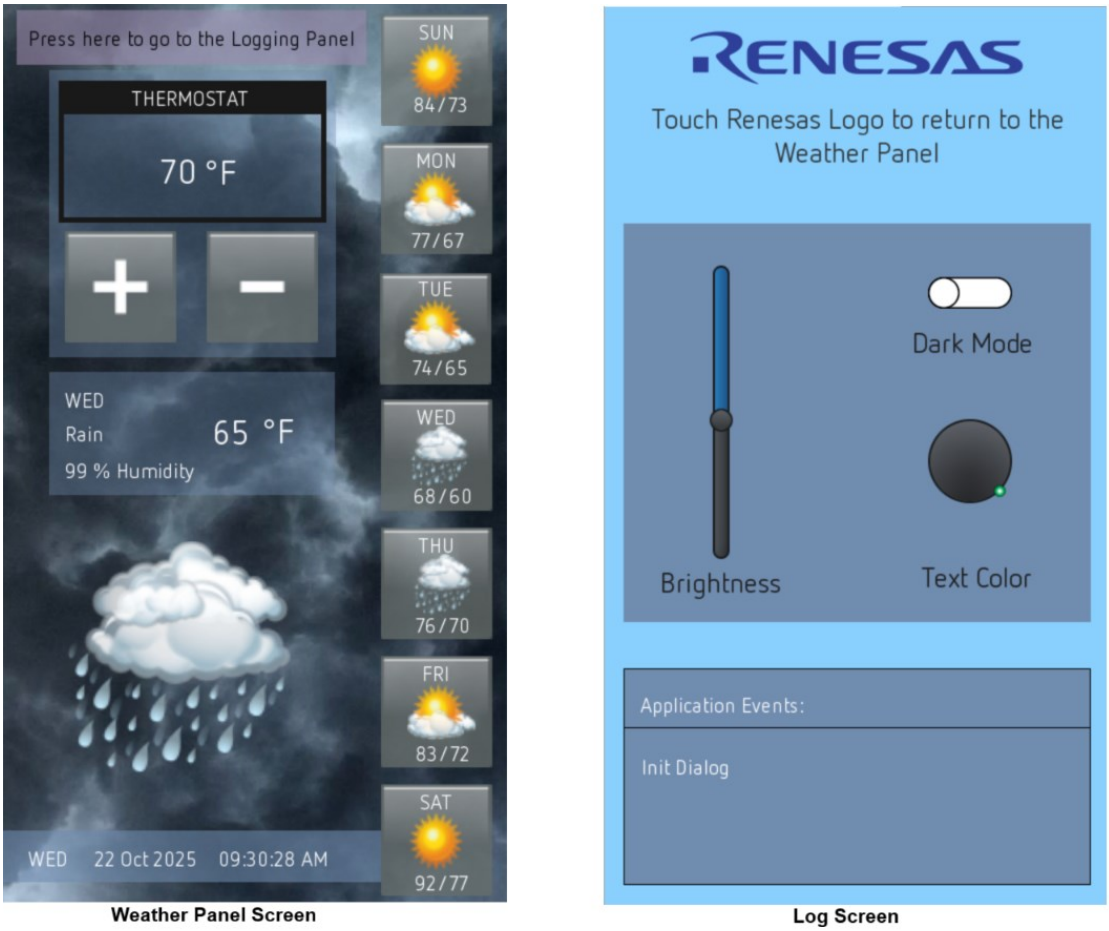


Figure 4. Screenshot of the GUI being designed in the AppWizard

Weather Panel is the first screen that appears on the LCD when booting up. It displays the thermostat information and allows the user to select a day or adjust the temperature.

Logging Panel shows events that occurred in the Weather Panel, adjusts LCD backlight, text color or background color of the Logging Panel.

2. GUI Development with AppWizard

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 RA8D2 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.

2.1 AppWizard and emWin Capabilities

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.

The AppWizard comes with a standard set of fonts and basic interface graphics, including images, text, buttons, rotaries, sliders, and more. During your screen creation phase, you can customize the display by importing image and font files into AppWizard. For optimal performance, images should be prepared in the same color format as the framebuffer. This approach eliminates runtime format conversion and ensures efficient rendering. After designing the GUI application, the AppWizard desktop simulator allows you to preview the precise look and feel of the application. This cross-platform simulation ensures a high degree of fidelity, meaning what you see running in the tool on your PC is what you get displayed on the final embedded screen.

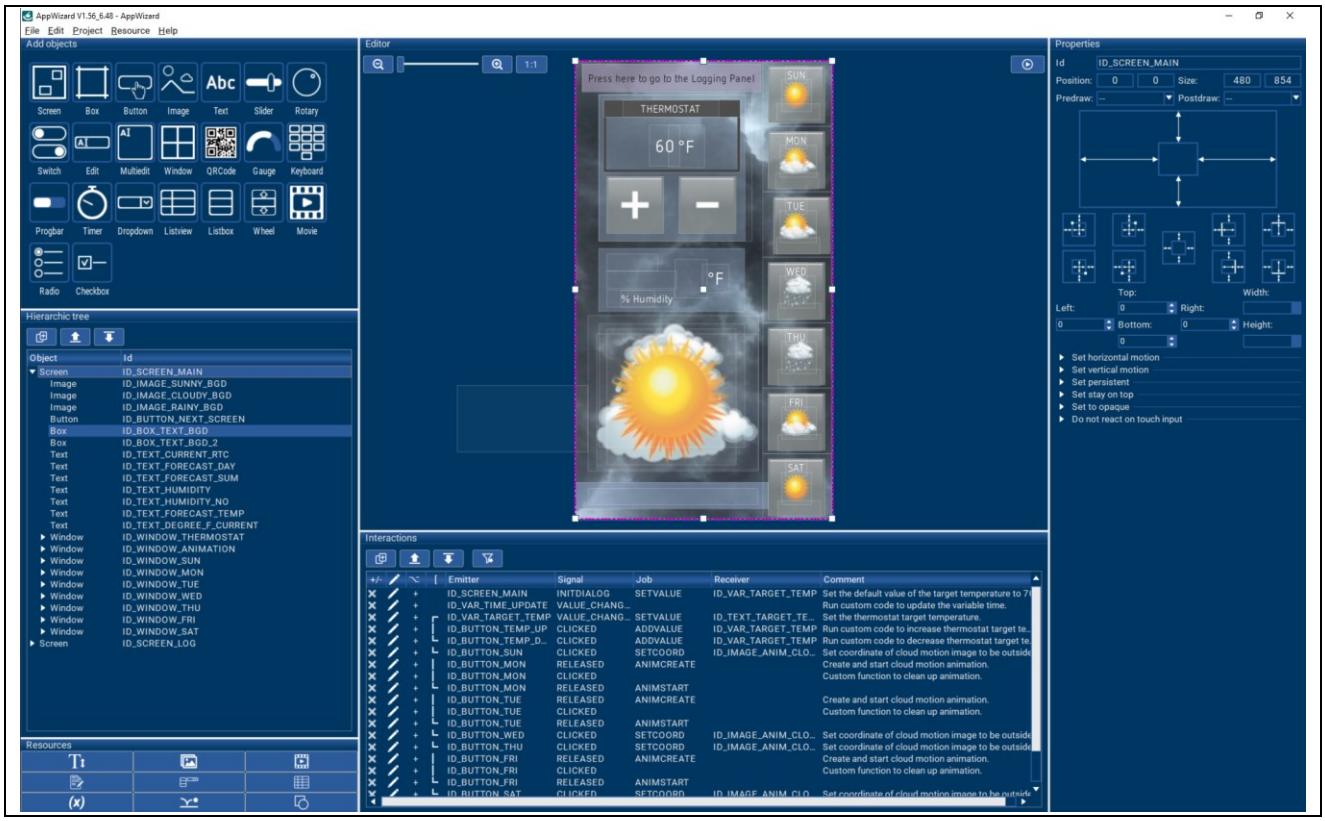


Figure 5. Screenshot of GUI being designed in AppWizard

2.2 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 shown in Figure 6. You can always access this information to view and edit the properties by using the **Project > Edit Options** menu option. During project creation in AppWizard, the display **color format** can be defined. When adding a new image, the default setting for the bitmap format is **Auto**, meaning AppWizard will automatically select the most suitable bitmap format based on the project's color format. However, if the target hardware requires a specific bitmap format, the user can override the default and manually select the desired format.

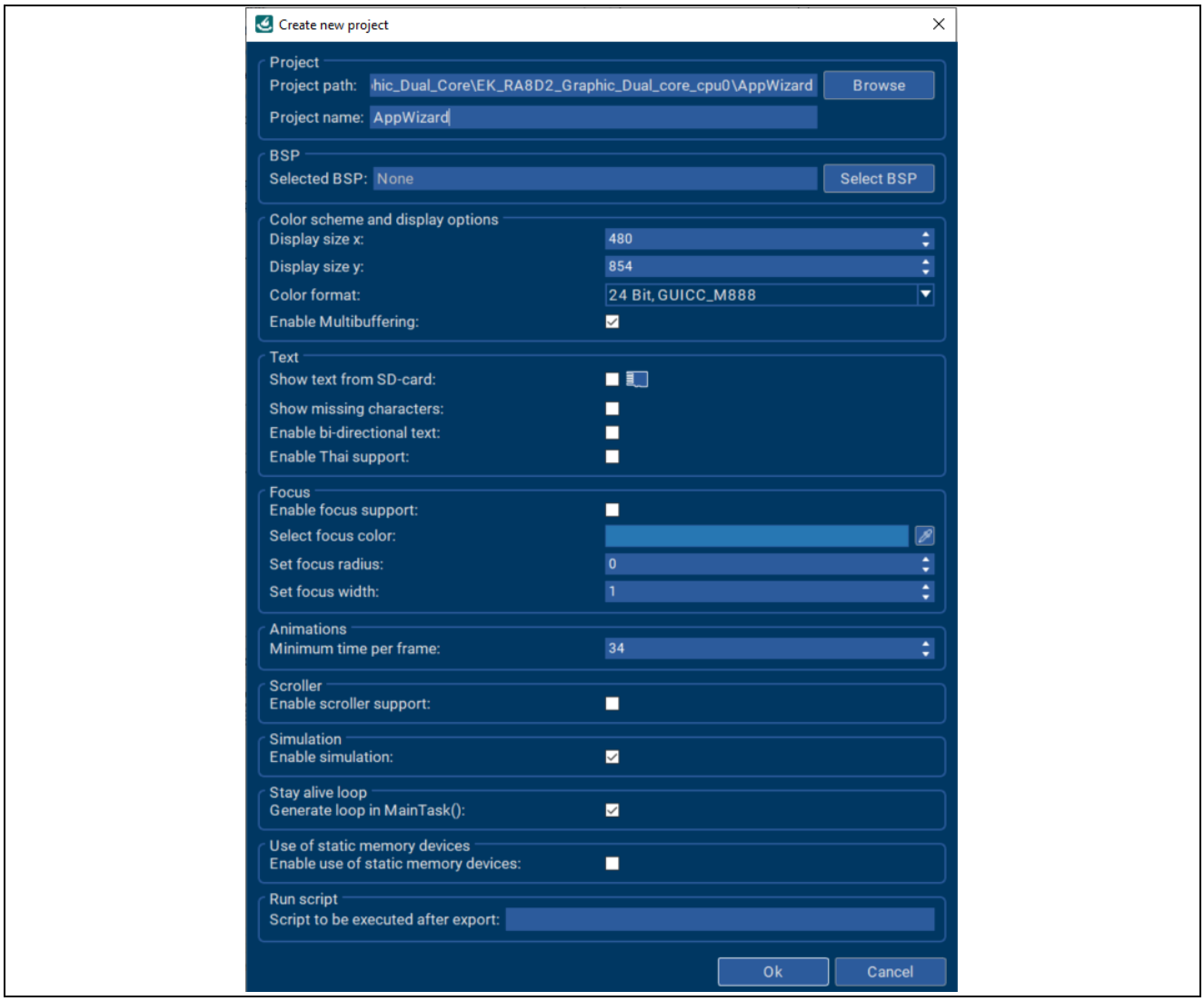


Figure 6. Create a New Project Dialog Box

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.

2.2.1 Including AppWizard Project Files in e² studio Project

It's recommended that you create an e² studio project first and then create an AppWizard folder in the project as an e² studio source folder. The AppWizard folder in the e² 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 e² studio project directory, which was also set as the **Project Path** property in the AppWizard project.

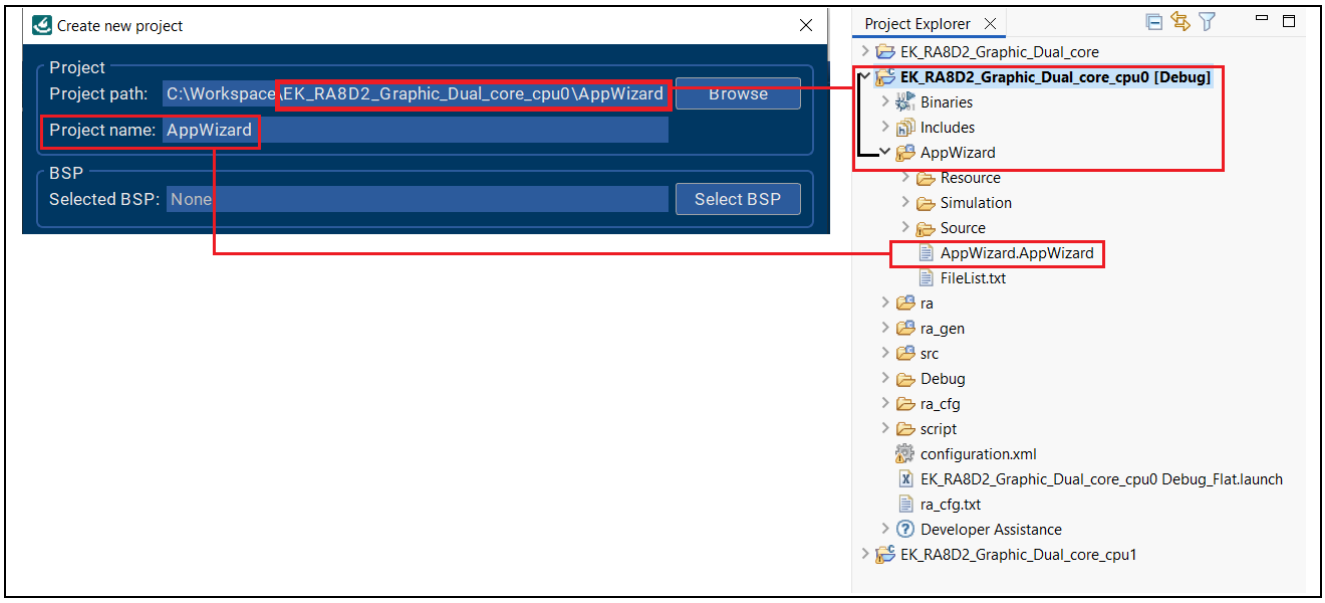


Figure 7. The AppWizard project is contained within the specified project path

To ensure successful compilation and proper access to all required resources, follow these steps to correctly configure the **AppWizard** project in **e² studio**:

1. Set AppWizard as a Source Folder:

- Right-click the “AppWizard” folder in the Project Explorer.
- Select **C/C++ Build**.
- Ensure that **Exclude resource from build** is **not selected**.
- Refer to Figure 8 below for visual confirmation of this setting.

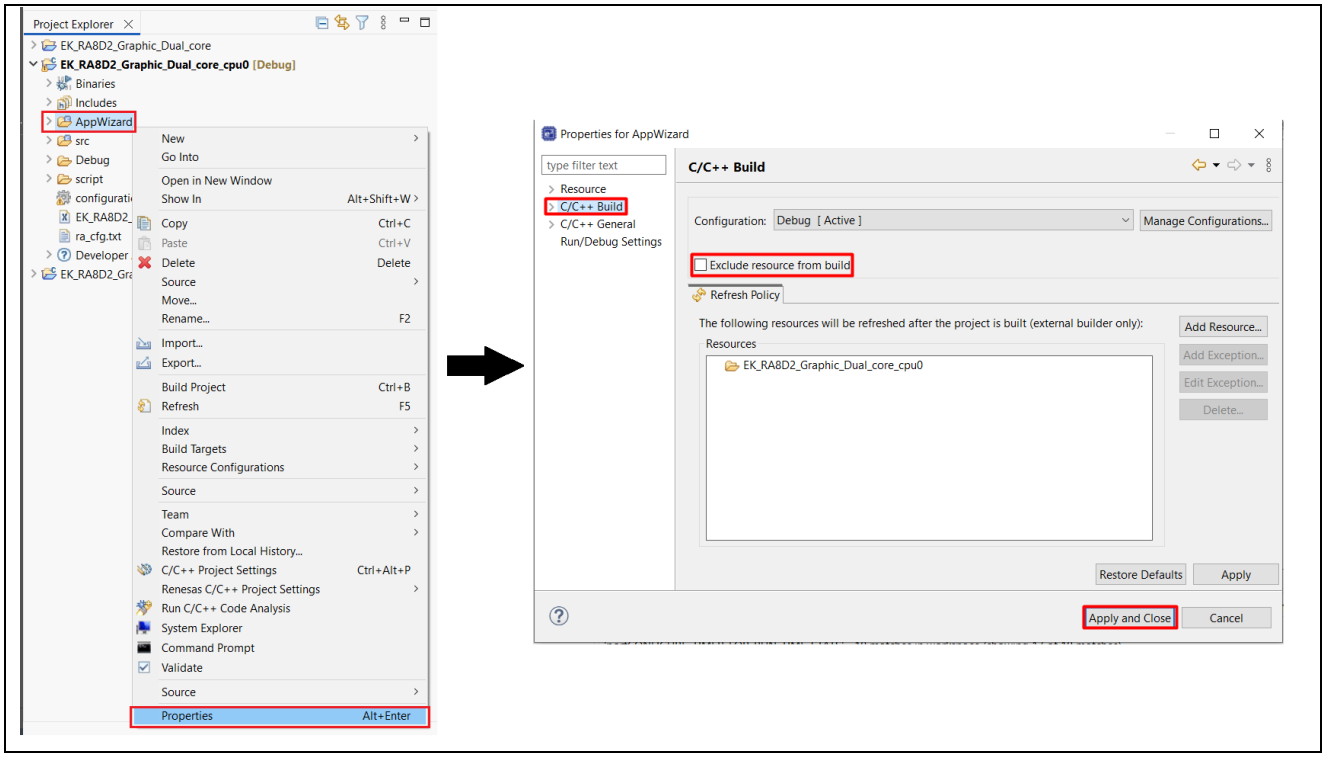


Figure 8. Verify AppWizard as Source Folder

2. Add AppWizard/Source and all subdirectories to the include paths to ensure that all required header files are accessible:

- Right-click your **Graphics development project**. (e.g., EK_RA8D2_Graphic_Dual_core_cpu0)
- Go to **Properties > C/C++ Build > Settings**.
- Under Tool Settings, navigate to: **Compiler > Includes >** Click **Add** as shown in Figure below

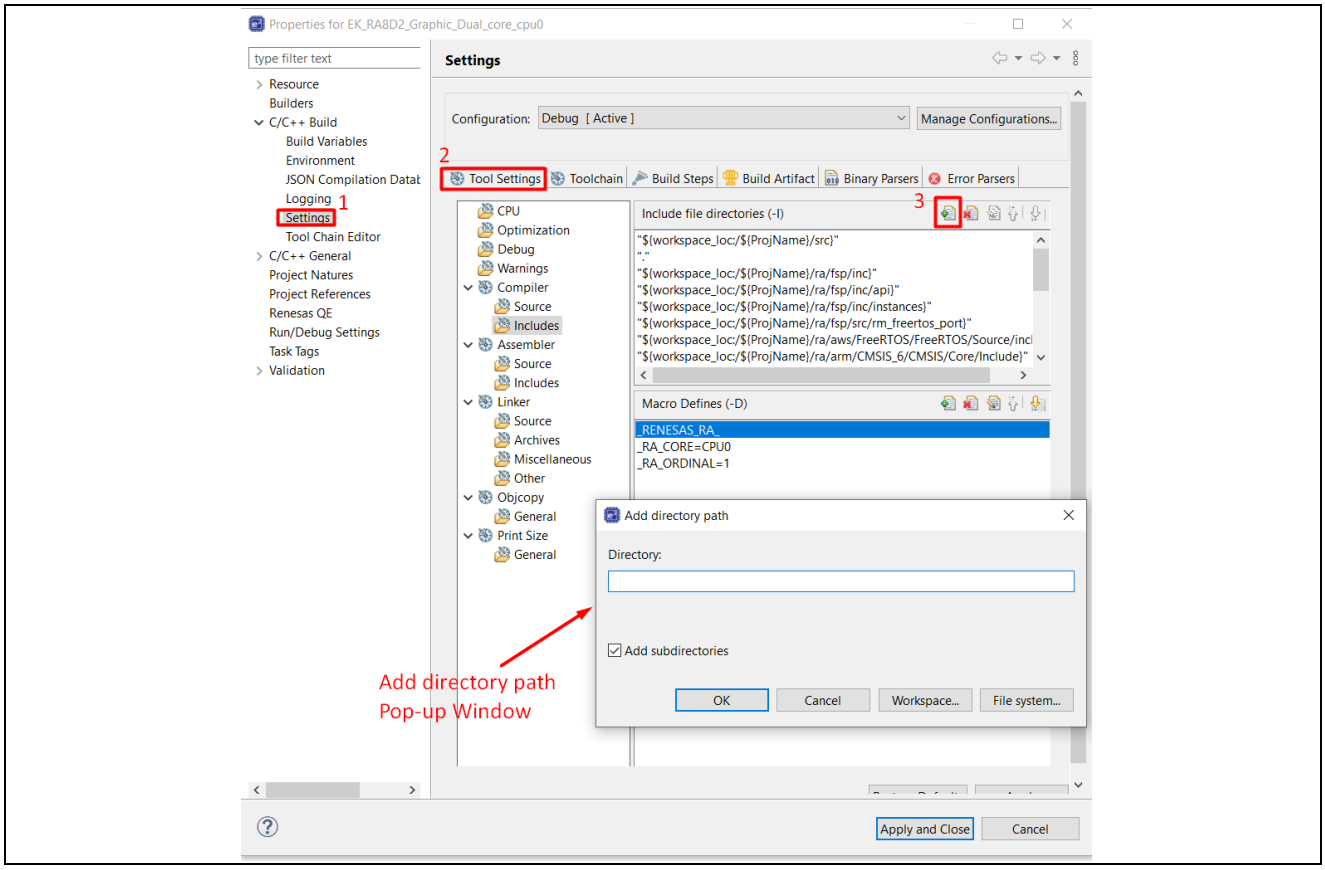


Figure 9. Include Path Configuration in e² studio

- Click **Workspace** and Navigate to the AppWizard/Source folder. Make sure Add subdirectories are checked.
- Click OK, then **Apply and Close** to save the settings.

This ensures that all headers within AppWizard/Source and its subfolders are included during compilation.

3. Exclude the Simulation folder from the build. After including all subdirectories, you should exclude the Simulation folder from the build before building the e² studio project:

- 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**.

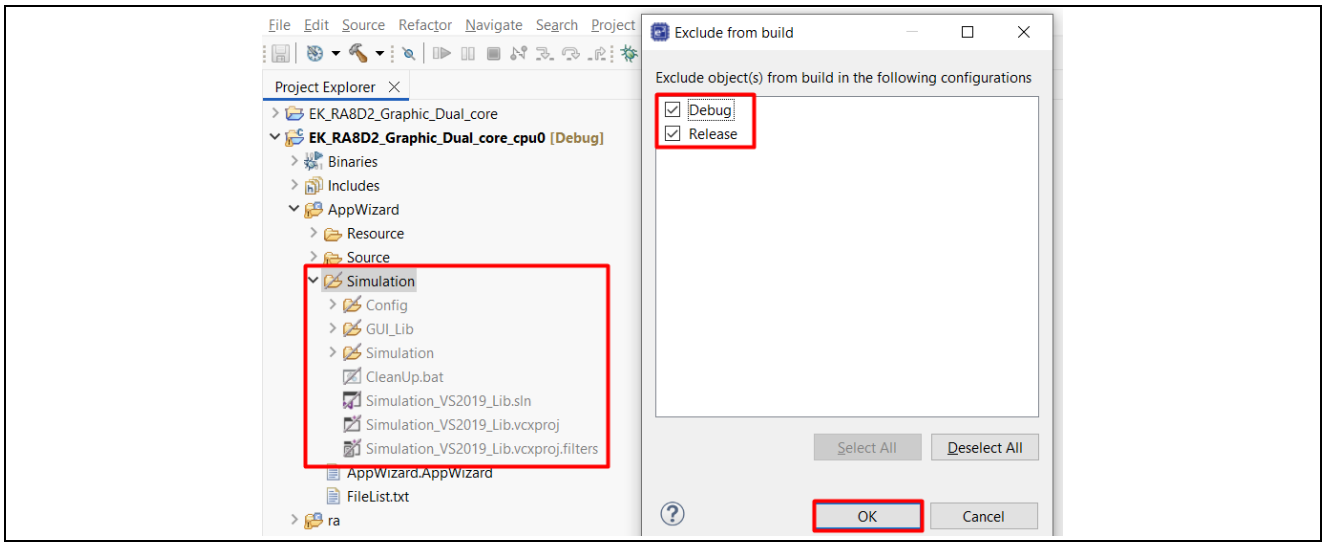
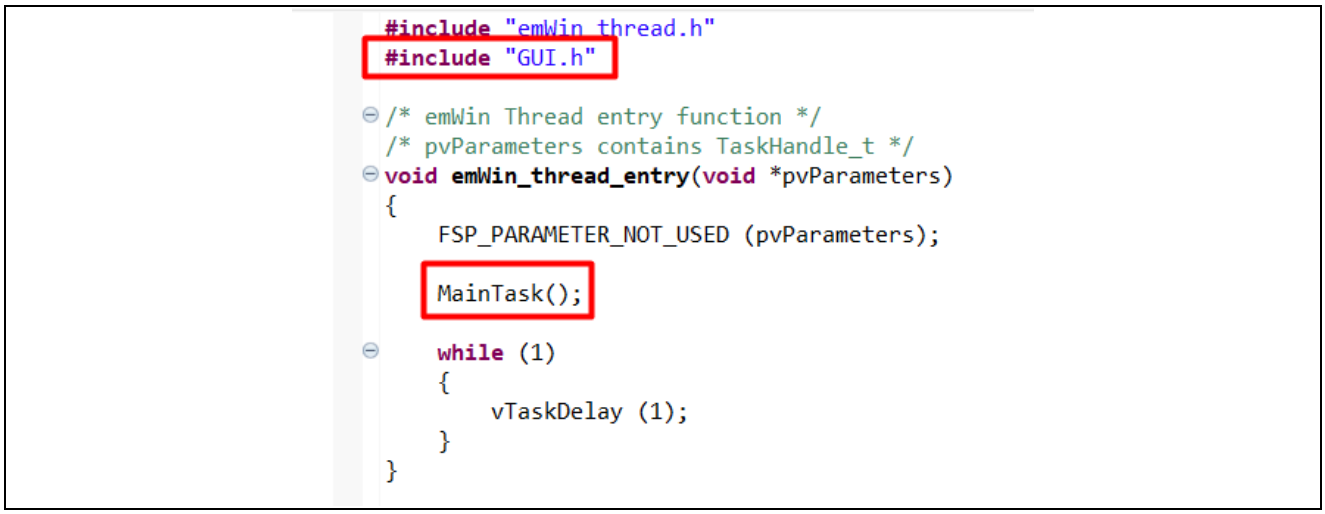


Figure 10. Exclude the Simulation folder and subdirectories for a successful build

2.2.2 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 e^2 studio project as described in section 3.4.2.

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 RA8D2, 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.

2.3 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 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 fonts and 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.

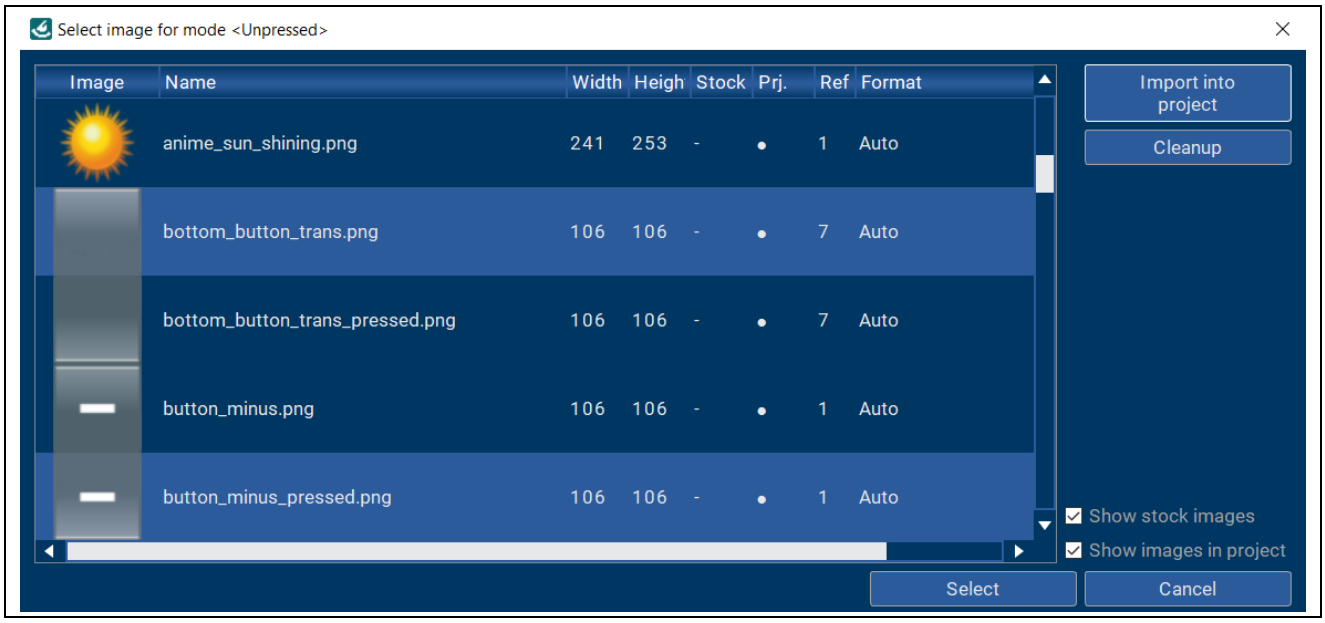


Figure 12. The Select Image Menu supports imported and built-in bitmaps for objects

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 details of all bitmaps in the current AppWizard project. It can also be used to specify the bitmap pixel format for each image. When the format is set to **Auto**, AppWizard automatically selects the most suitable bitmap format based on the project's color format defined for the project.

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.

2.4 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.

2.4.1 AppWizard Defined Interactions

As an example of a predefined interaction, the thermostat application specifies that clicking the **ID_BUTTON_SUN** object moves the **ID_IMAGE_ANIM_CLOUD_MOTION** image outside the visible window. This behavior is configured in AppWizard and does not require additional user code.

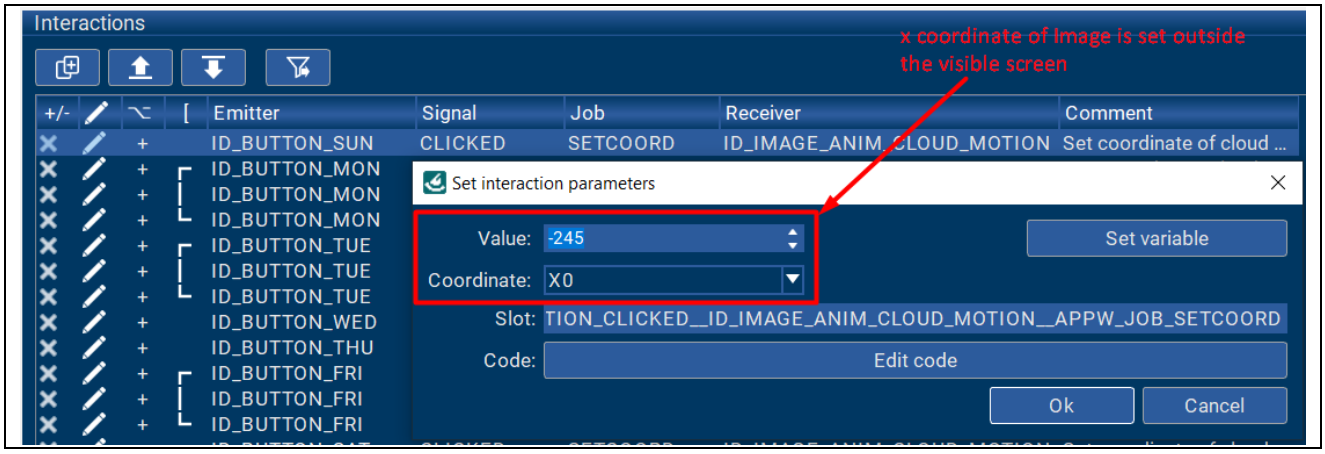


Figure 13. The Example of Setting ID_BUTTON_SUN Interaction

2.4.2 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`.

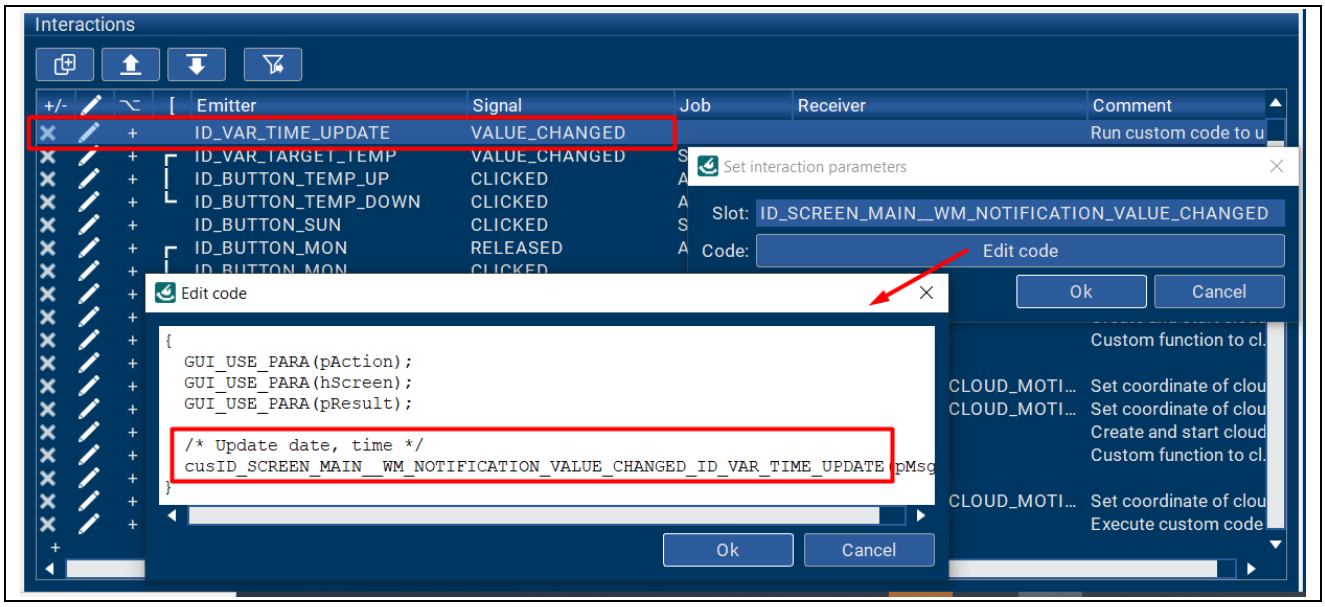


Figure 14. Example of a Custom Slot Routine Triggered on Value Change of ID_VAR_TIME_UPDATE

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.

Figure 15. Custom routines allow personalized responses to interactions

2.4.3 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.

```
 * @brief     IPC channel 0 callback function.
void g_ipc0_callback(ipc_callback_args_t *p_args)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    switch(p_args->event)
    {
        case IPC_EVENT_IRQ2:
            xSemaphoreGiveFromISR(g_touch_reset_semaphore, &xHigherPriorityTaskWoken);
            portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
            break;
        case IPC_EVENT_IRQ3:
            /* Trigger GUI update date & time*/
            APPW_SetVarData(ID_VAR_TIME_UPDATE, 1);
            break;
        case IPC_EVENT_IRQ4:
        case IPC_EVENT_IRQ5:
        case IPC_EVENT_IRQ6:
        case IPC_EVENT_IRQ7:
            break;
        default:
            break;
    }
}
```

Figure 16. Example of Time Update Request from CPU1 Triggering the Graphic Thread

When ID_VAR_TIME_UPDATE changes, it triggers the execution of the custom user code defined in Figure 15. This interaction was configured during the GUI creation phase, as shown in Figure 14.

2.5 Add emWin Widget to AppWizard Project

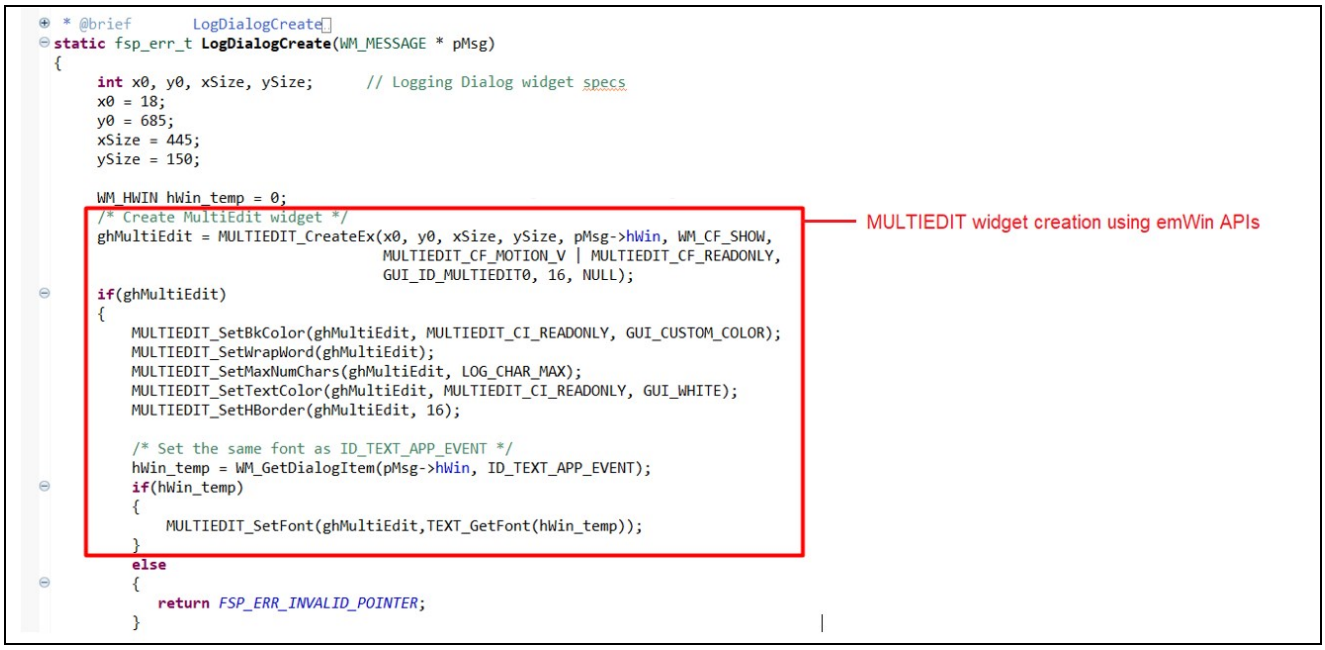
The objects that AppWizard supports are similar to the widgets provided by emWin. A widget is a window with object-type properties and the emWin library contains API functions for the creation, configuration, communication, and more, of widgets. The emWin library also supports the creation and management 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 17 and 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.

```
 * @brief       Custom code for cbID_SCREEN_LOG in ID_SCREEN_LOG_Slots.c
void cuscbID_SCREEN_LOG(WM_MESSAGE * pMsg) {
    switch(pMsg->MsgId)
    {
        case WM_INIT_DIALOG:
          /* Create MultiEdit widget as Logging dialog */
          if(LogDialogCreate(pMsg))
          {
            /* Handle error */
            APP_ERR_TRAP(FSP_ERR_INVALID_POINTER);
          }
          else
          {
```

Figure 17. Example of Adding a MULTIEDIT Widget during the Log Screen Creation Routine




```
 * @brief       LogDialogCreate
static fsp_err_t LogDialogCreate(WM_MESSAGE * pMsg)
{
    int x0, y0, xSize, ySize;      // Logging Dialog widget specs
    x0 = 18;
    y0 = 685;
    xSize = 445;
    ySize = 150;

    WM_HWIN hWin_temp = 0;
    /* 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);

        /* Set the same font as ID_TEXT_APP_EVENT */
        hWin_temp = WM_GetDialogItem(pMsg->hWin, ID_TEXT_APP_EVENT);
        if(hWin_temp)
        {
            MULTIEDIT_SetFont(ghMultiEdit,TEXT_GetFont(hWin_temp));
        }
        else
        {
           return FSP_ERR_INVALID_POINTER;
        }
```

Figure 18. Example of Creating a MULTIEDIT Widget with emWin APIs

3. Thermostats Graphics Application

The accompanying project is a dual-core FreeRTOS application built within the FSP Solution project. In this setup, a portrait-oriented graphical thermostat application runs on CPU0, while CPU1 handles touch input and other peripheral control tasks. This task partitioning demonstrates how graphics can be isolated to CPU0 and supporting functions to CPU1, thereby optimizing overall application performance. The project's purpose is multifaceted and demonstrates how to do the following through a single use case:

- Design a multi-screen GUI in AppWizard, targeting the MIPI Graphic Extension Board.
- Integrate the AppWizard GUI into the e^2 studio project.
- 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-RA8D2'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-RA8D2.
- Set up and store large resources to external flash devices.
- Communicate with the I2C capacitive touch overlay on the LCD using the GT911 drivers.
- Task partitioning and inter-core communication between CPU0 and CPU1.

This section explains the thermostat application's code structure, high-level design, thread functions, and important module configurations. Furthermore, this section explains how to initialize and store image resources in the external flash, as well as the procedures for operating the MIPI graphics expansion display.

For details on how to partition tasks and the methods for exchanging data in dual-core systems, refer to application note "R01AN7881EU Developing with RA8 Dual Core MCU".

Use the instructions in Section 4 to import, build, and run the graphics thermostat application project on your own hardware.

3.1 Source Code Layout

The thermostat application is implemented on an FSP solution project configured with a dual-core FreeRTOS environment.

This architecture enforces clear task partitioning**:** CPU0 is dedicated to graphical user interface (GUI) processing, while CPU1 is assigned to peripheral management tasks. On CPU0, the application integrates

both custom code and auto-generated sources, including the code generated by AppWizard. In contrast, CPU1 executes only peripheral APIs and does not rely on AppWizard auto-generation.

For guidance on creating an FSP solution project with dual-core FreeRTOS, please refer to the reference application R01AN7982 Multicore Setup and Running Hello World on Dual-Core.

The structure of a dual-core project created with an FSP solution project is depicted in Figure 19.

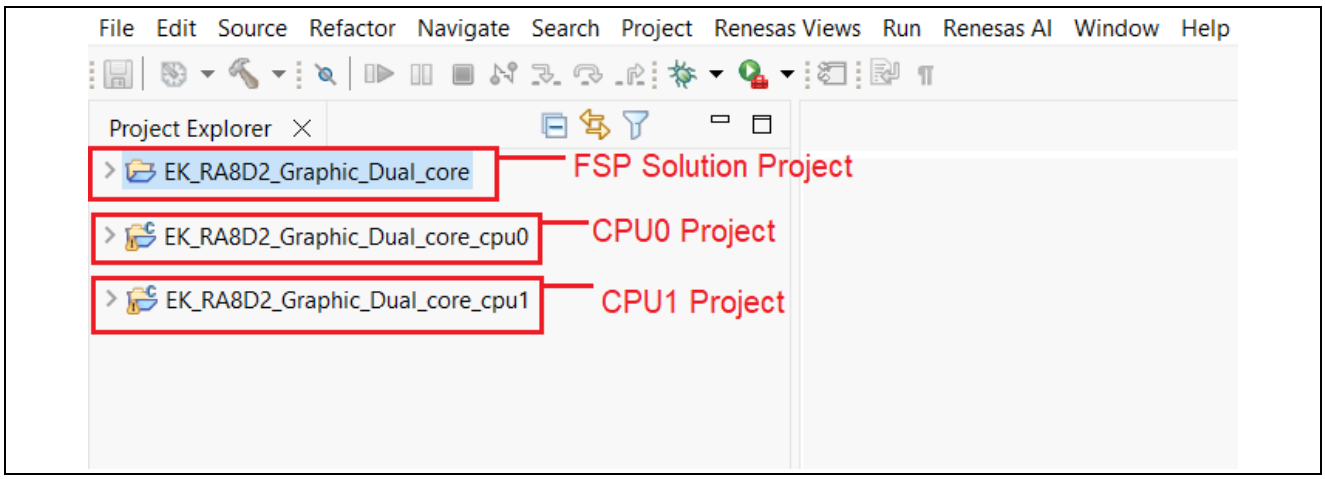


Figure 19. Organization of a Dual-Core FSP Solution Project

Figure 20 shows the source code layout for the thermostat application in the CPU0 project.

The “ra_gen” and “ra” directories include the auto-generated FSP module APIs, middleware, and public emWin library files, while the “src” directory contains the user-defined application thread code. 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 “AppWizard/Source/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.

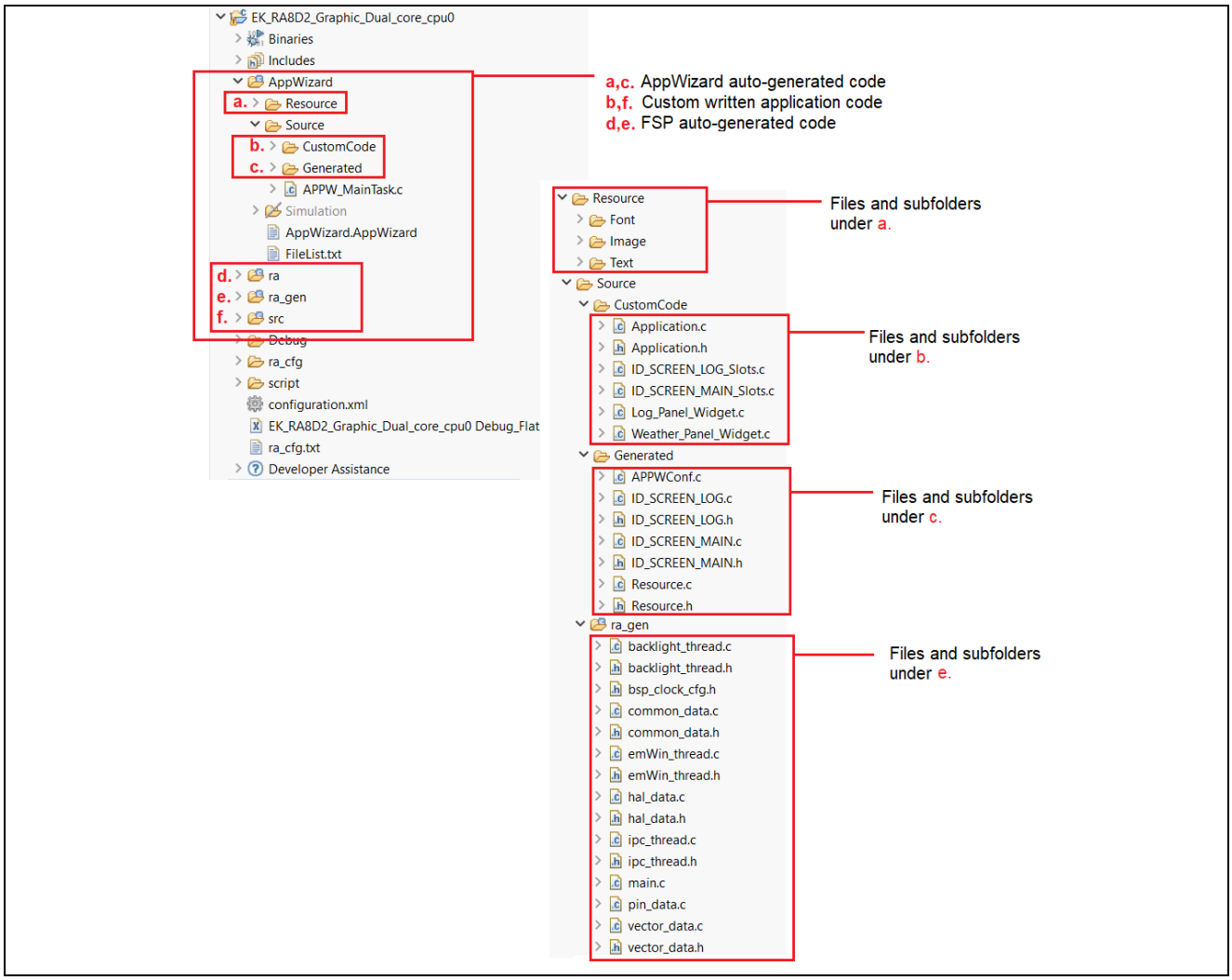


Figure 20. Overview of source code organization in the CPU0 project

The CPU1 project focuses exclusively on peripheral management. Its contents are limited to the peripheral control thread code and the GT911 driver. Consequently, it does not contain any auto-generated sources or user-defined code from the AppWizard project. Figure 21 illustrates the source code layout of the CPU1 project.

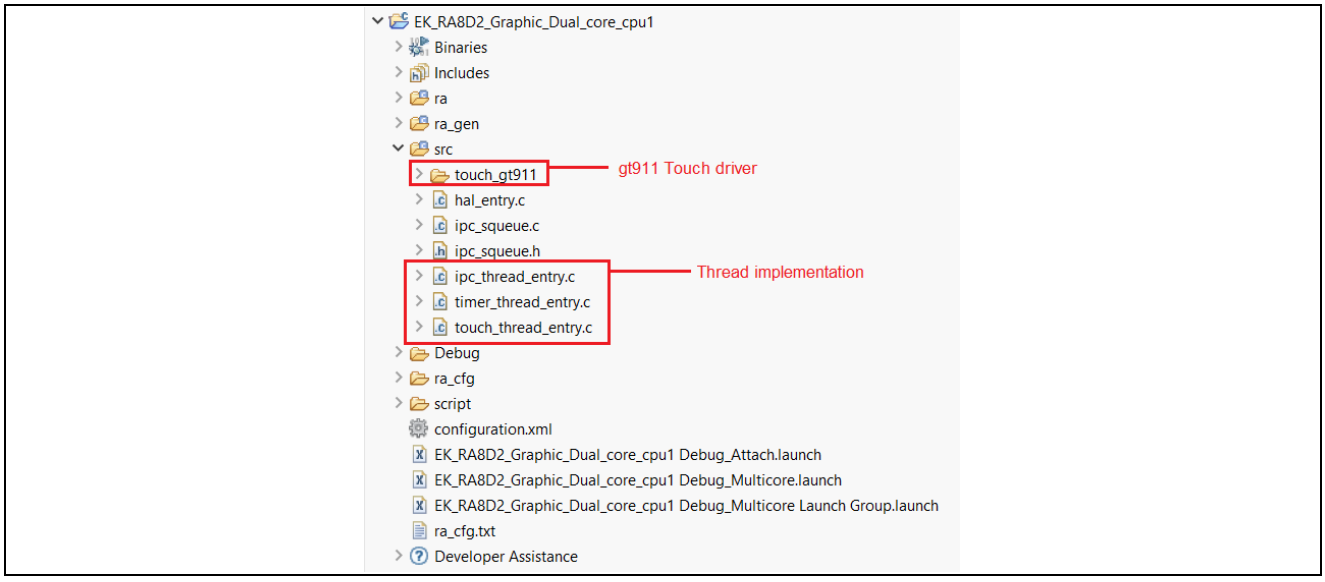


Figure 21. Overview of source code organization in the CPU1 project.

3.2 System Design and Operation Flow

As illustrated in Figure 4, the graphics application consists of two panels: the Weather Panel and the Logging Panel. These panels interact with the graphics framework through touch events and variable updates, which are received and processed by CPU1. CPU1 communicates with the FSP and HAL drivers to send and receive touch sensing data, as well as RTC date and time information, via IPC to CPU0.

The graphics framework comprises the SEGGER AppWizard framework, the emWin library, and the emWin RA port, and it interfaces with CPU0, which manages hardware resources through HAL drivers such as GLCDC, D/AVE 2D, MIPI-DSI / MIPI-PHY, and the backlight (controlled by GPT in the CPU0 project). Figure 22 illustrates the system block diagram of the thermostat application, while Figure 23 presents its operation flow.

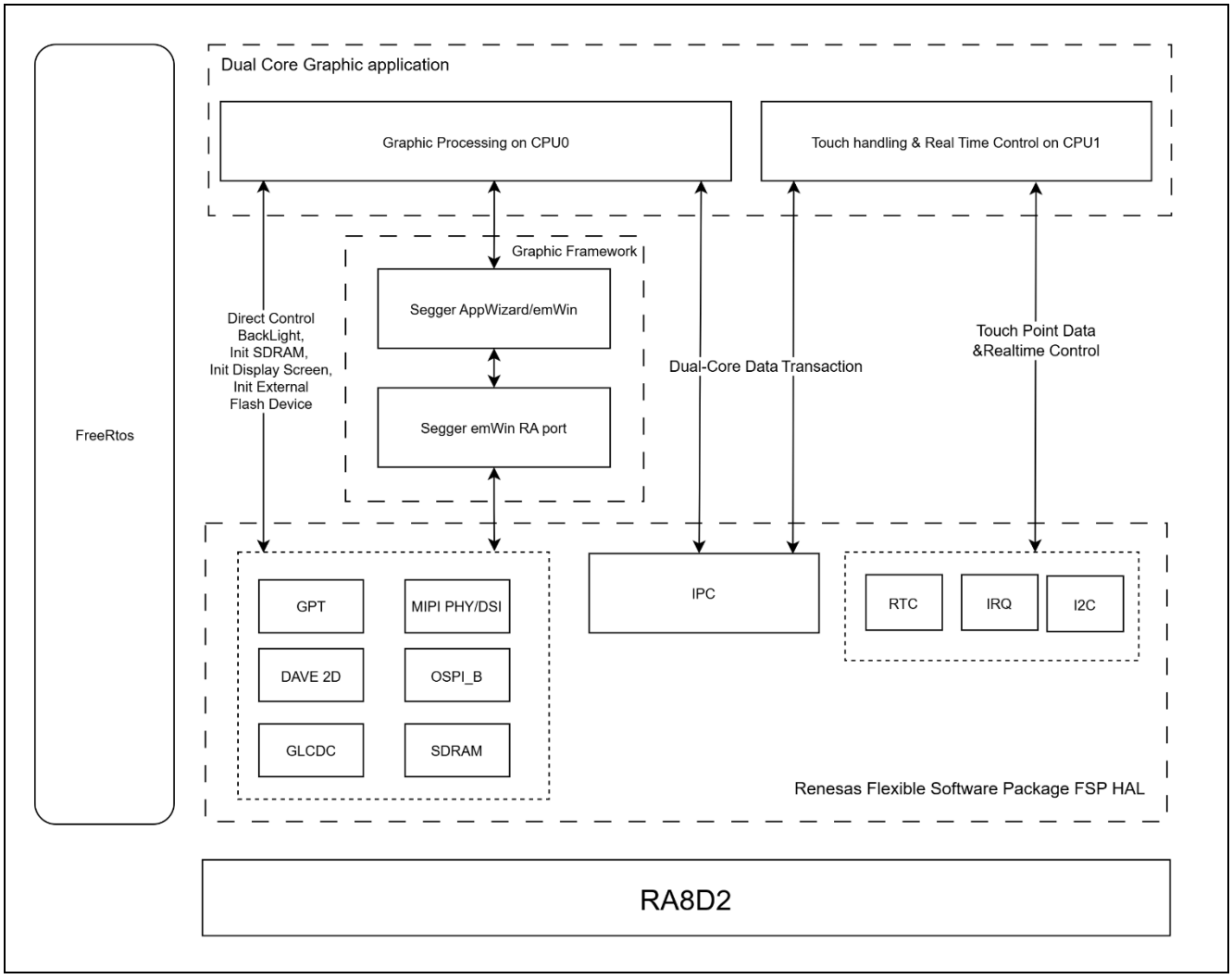


Figure 22. Dual Core Thermostats Application Block Diagram.

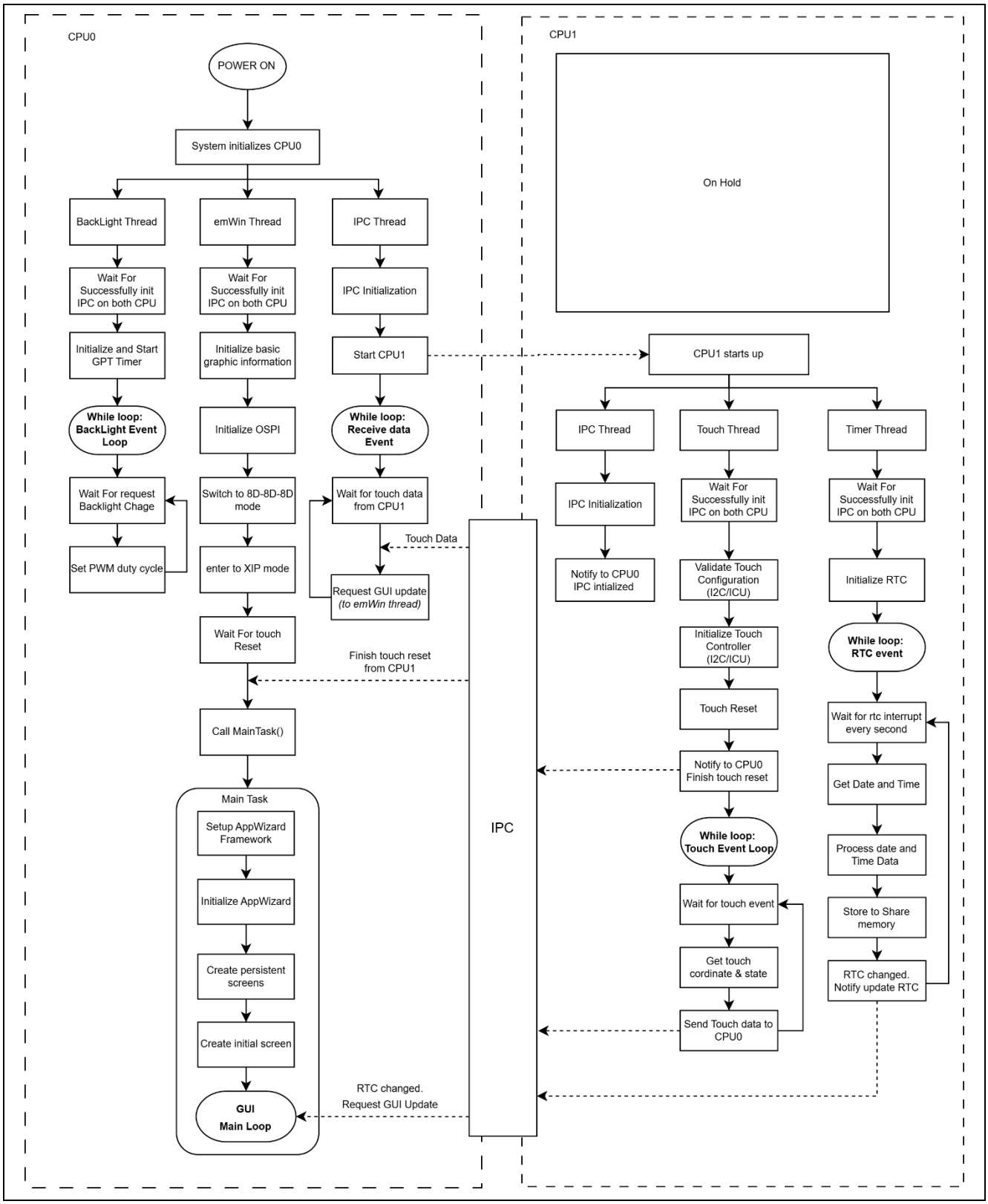


Figure 23. Dual-Core Thermostat Application Operation Flow.

3.3 Dual-Core Architecture and Component Identification.

The RA8 dual core series employs a dual-core architecture based on an Asymmetric Multiprocessing (AMP) model. This architecture is supported in the Flexible Software Package (FSP) through a dedicated dual-core FSP Solution Project.

In this model, one core functions as the primary processor and is always the first to boot following a system reset. By default, CPU0 is designated as the primary CPU. Once initialized, CPU0 can activate CPU1 by invoking the Secondary Core Start API from the user application.

A dual-core application can therefore be developed as two independent FSP projects, one for each core. Component selection and configuration for each CPU are performed individually through the Stacks tab in e² studio.

For system resources shared between CPU0 and CPU1, as well as for inter-core communication, developers must follow specific design principles to avoid bus contention and race conditions. Details of these guidelines are provided in application note “R01AN7881 Developing with RA8 Dual Core MCU”.

Determining the required components for each CPU is a critical step once task allocation between the cores has been defined. The following two tables list the selected components for CPU0 and CPU1 in this thermostat application.

Table 1. Components of the Graphics Application in CPU0

Category	Component	Version	Description
BSP	ra8d2_ek	6.2.0	RA8D2-EK Board Support Package Files
CMSIS	CoreM	6.1.0+fsp.6.2.0	Arm CMSIS Version 6 - Core (M)
Common	fsp_common	6.2.0	Board Support Package Common Files
GUI	emWin	6.48+fsp6.2.0	SEGGER emWin Library
HAL Drivers	r_drw	6.2.0	TES D/AVE 2D Port
	r_glcdc	6.2.0	Graphics LCD Controller
	r_ioport	6.2.0	I/O Port
	r_dmac	6.2.0	Direct Memory Access Controller
	r_gpt	6.2.0	General PWM Timer
	r_ipc	6.2.0	IPC
	r_mipi_dsi	6.2.0	MIPI DSI Host
	r_mipi_phy	6.2.0	MIPI PHY Host
	r_ospi_b	6.2.0	Octa Serial Peripheral Interface Flash
Heaps	heap_4	11.1.0+fsp.6.2.0	FreeRTOS - Memory Management – Heap 4
Middleware	rm_emwin_port	6.2.0	SEGGER emWin RA Port
	rm_freertos_port	6.2.0	FreeRTOS Port
RTOS	FreeRTOS	11.1.0+fsp.6.2.0	FreeRTOS
TES	dave2d	3.8.0+fsp.6.2.0	TES DAVE 2D Drawing Engine

Table 2. Components of the Graphics Application in CPU1

Category	Component	Version	Description
BSP	ra8d2_ek	6.2.0	RA8D2-EK Board Support Package Files
CMSIS	CoreM	6.1.0+fsp.6.2.0	Arm CMSIS Version 6 - Core (M)
Common	fsp_common	6.2.0	Board Support Package Common Files
	bsp_ipc	6.2.0	BSP IPC Semaphore
HAL Drivers	r_dtc	6.2.0	Data Transfer Controller
	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_rtc	6.2.0	Real-Time Clock
	r_ipc	6.2.0	IPC
Heaps	heap_4	11.1.0+fsp.6.2.0	FreeRTOS - Memory Management – Heap 4
Middleware	rm_emwin_port	6.2.0	SEGGER emWin RA Port
	rm_freertos_port	6.2.0	FreeRTOS Port
RTOS	FreeRTOS	11.1.0+fsp.6.2.0	FreeRTOS

After the required components are added in the Stacks tab, they can be reviewed collectively in the Components tab, as illustrated in Figure 24.

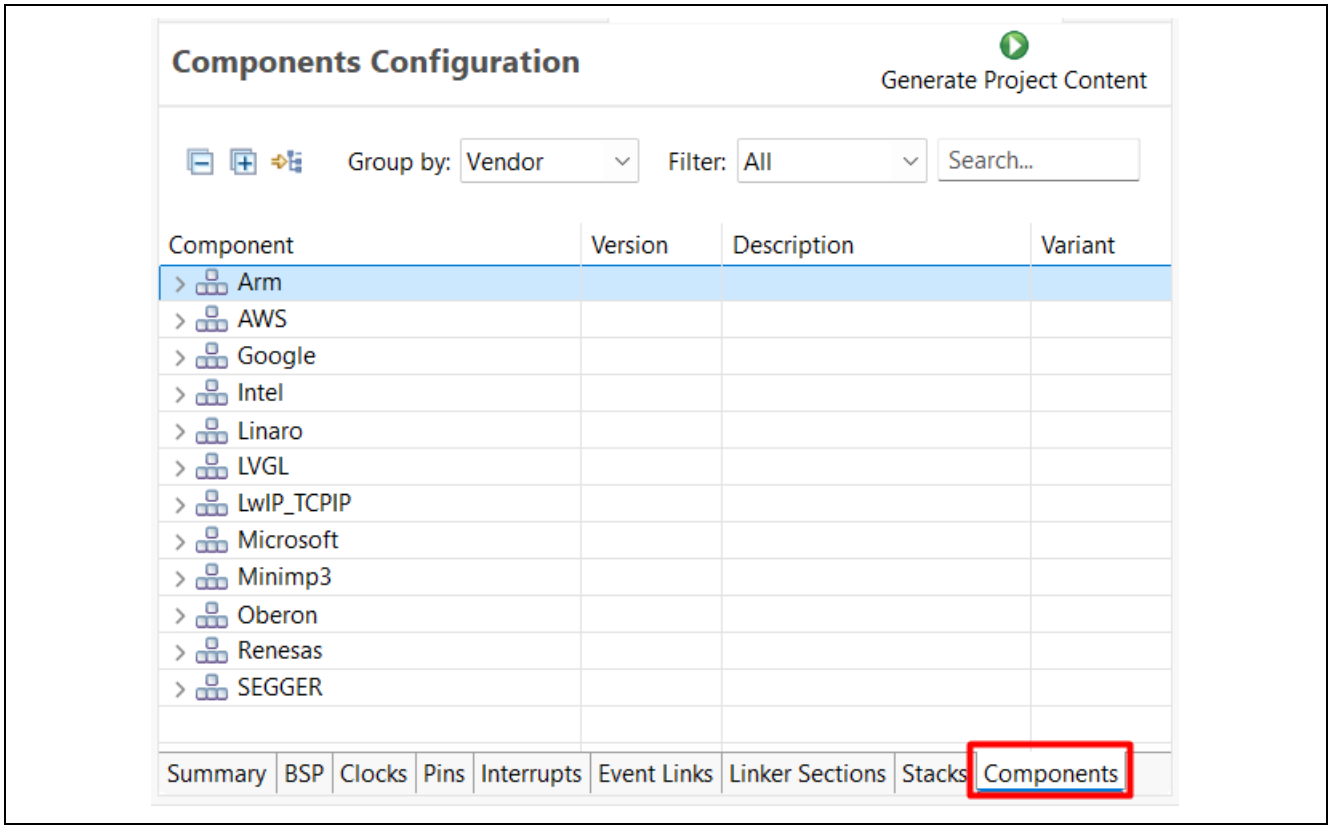


Figure 24. Components Tab Categories

3.4 Module, Pin and Clock Configuration

This section provides details of the configuration required for each component that was identified in the previous section. For every selected component, the corresponding peripheral modules, pin assignments, and clock settings are defined in the FSP configuration. These settings ensure that the hardware resources are properly initialized and available for use by the application software.

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.

The remainder of this section describes the key configuration settings for each module on both cores. These settings ensure cohesive operation of the graphics framework and establish the communication mechanism between the two CPUs.

3.4.1 IPC & FSP Solution Clock Configuration.

In a complex dual-core application, efficient inter-core communication is essential to avoid race conditions and bus contention. In this dual-core thermostat application, inter-core communication is implemented using two mechanisms with the help of the IPC framework and peripheral:

- Message FIFO, provided by the FSP for asynchronous message passing between cores.
- Shared memory, used to exchange data between CPU0 and CPU1.

Allocation of memory regions for shared access between the two cores is supported through the FSP Solution project, which allows defining shared memory areas accessible by both projects. The FSP Solution also provides reference clock settings that can be reused by individual CPU projects. Clock configuration for the entire dual-core thermostat application can be found in the FSP Solution Configuration.

Inter Processor Communication

In the thermostat application, Inter-Processor Communication (IPC) is employed to exchange real-time data and to issue periodic update requests to the graphics screen at one-second intervals. The IPC framework also makes use of interrupts to notify CPU0 whenever a touch event has been processed on CPU1.

- Touch Data Transfer: Touch coordinates are transferred from CPU1 to CPU0 using the IPC message FIFO mechanism.
- RTC Data Exchange: Real-Time Clock (RTC) data is exchanged between the two CPUs via shared memory.

The shared memory is configured within the FSP Solution project under the memory configuration settings. In the thermostat application, this memory region serves as a dedicated area for real-time data exchange between the two cores. The layout and allocation of the shared memory are illustrated in Figure 25.

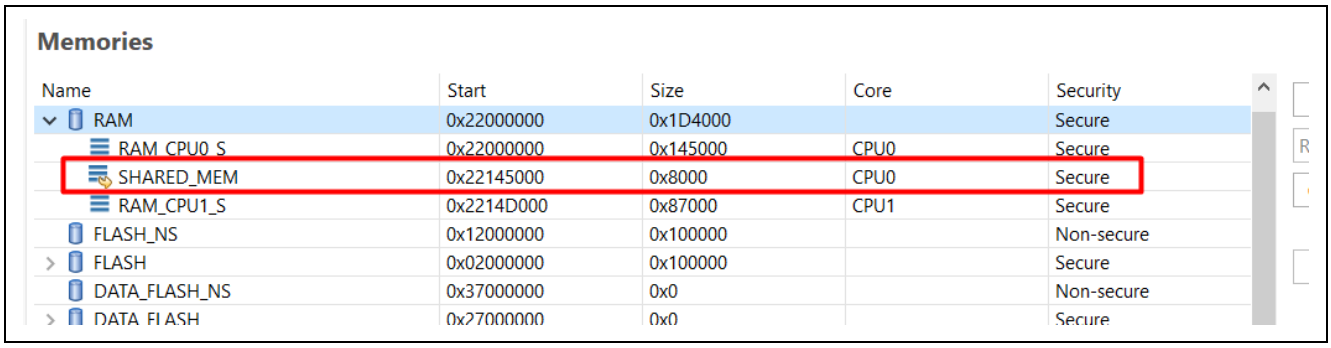


Figure 25. Example of Share Memory Partition in Application

FSP Solution Clock Configuration

The thermostat application is built on an FSP solution project, which supports clock configuration in FSP Solution project that can be inherited by other projects.

The clock configuration for the entire application is carried out by opening the **solution.xml** file in the FSP Solution Project and selecting the **Clocks** tab, as illustrated in Figure 26 .

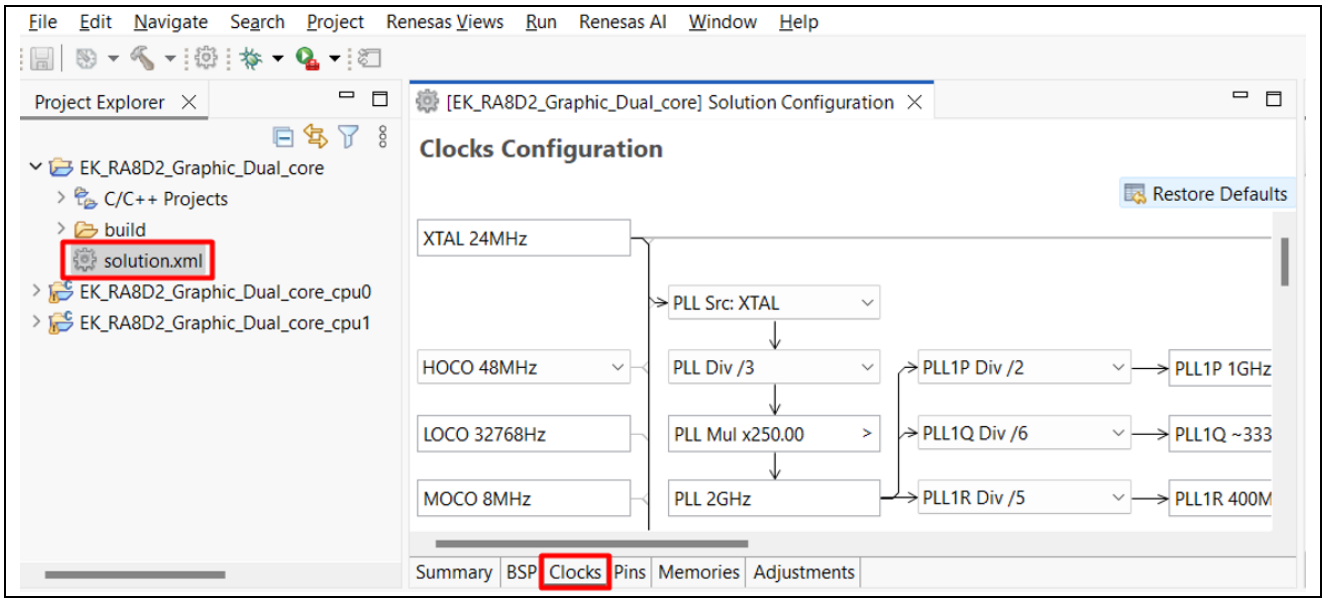


Figure 26. Example of the Clocks configuration tab in the Thermostat application

3.4.2 CPU0 Module Configuration.

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 the FreeRTOS "Heap_4" allocator, which coalesces adjacent free blocks. The heap region itself is placed at an absolute address.

SEGGER emWin RA Port (rm_emwin_port)

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 RA8D2, as well as with FreeRTOS. SEGGER emWin is commercial software. Renesas provides it to its customers free of charge, in binary library form only. Please contact SEGGER for the source code of the emWin library if it needs to be purchased.

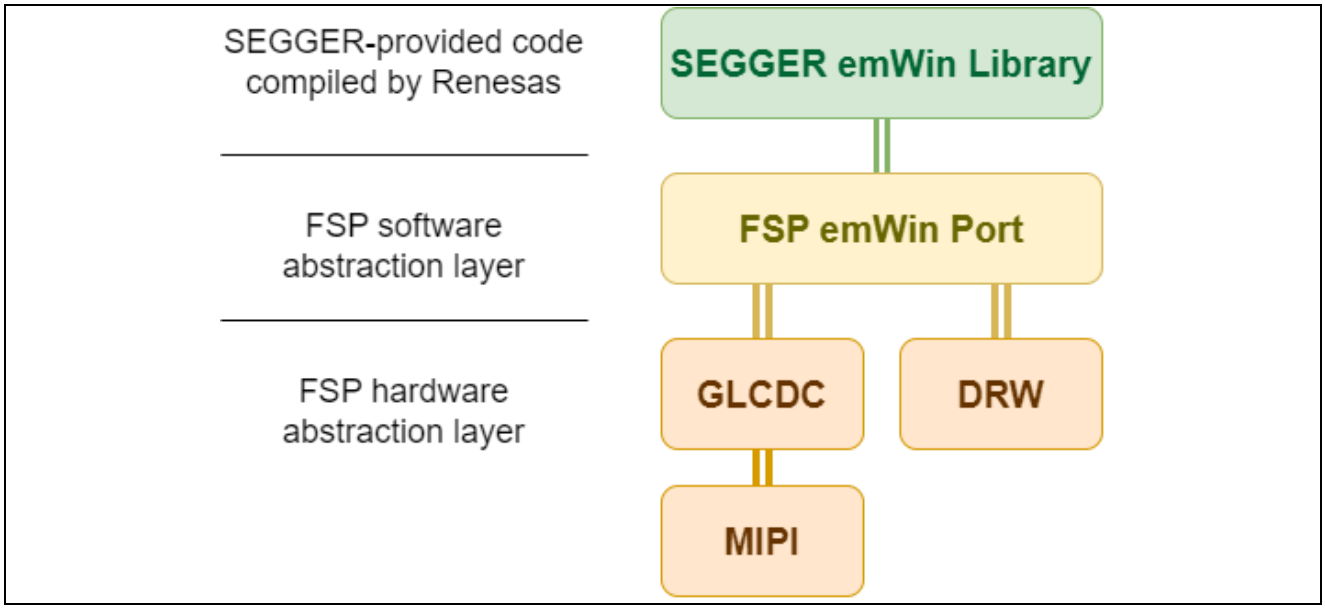


Figure 27. SEGGER emWin FSP port layers block diagram

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 3. 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.	0x20000	Provide sufficient RAM for the JPEG decoding and more.
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 requires 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 RGB888. 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 RA8D2 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 in FSP solution project to enable the correct clock for your application. In themostats application using the PLL1R as the LCDCLK source, resulting in a 200MHz LCDCLK.

Figure 28. PLL1R is the clock source for the LCDCLK in the Thermostat Application

MIPI-DSI/MIPI-PHY (r_mipi_phy and r_mipi_dsi)

When configuring the MIPI-PHY and MIPI-DSI interfaces, attention should be given to several critical parameters that determine both display synchronization and physical layer stability:

- **Clocking** – ensures proper synchronization between the display controller, the DSI transmitter, and the panel interface.
  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

  For the thermostat application, the D-PHY PLL is configured at 1 GHz under the **r_mipi_phy > DSI PLL Frequency** property. This corresponds to a line rate of 500 Mbps.

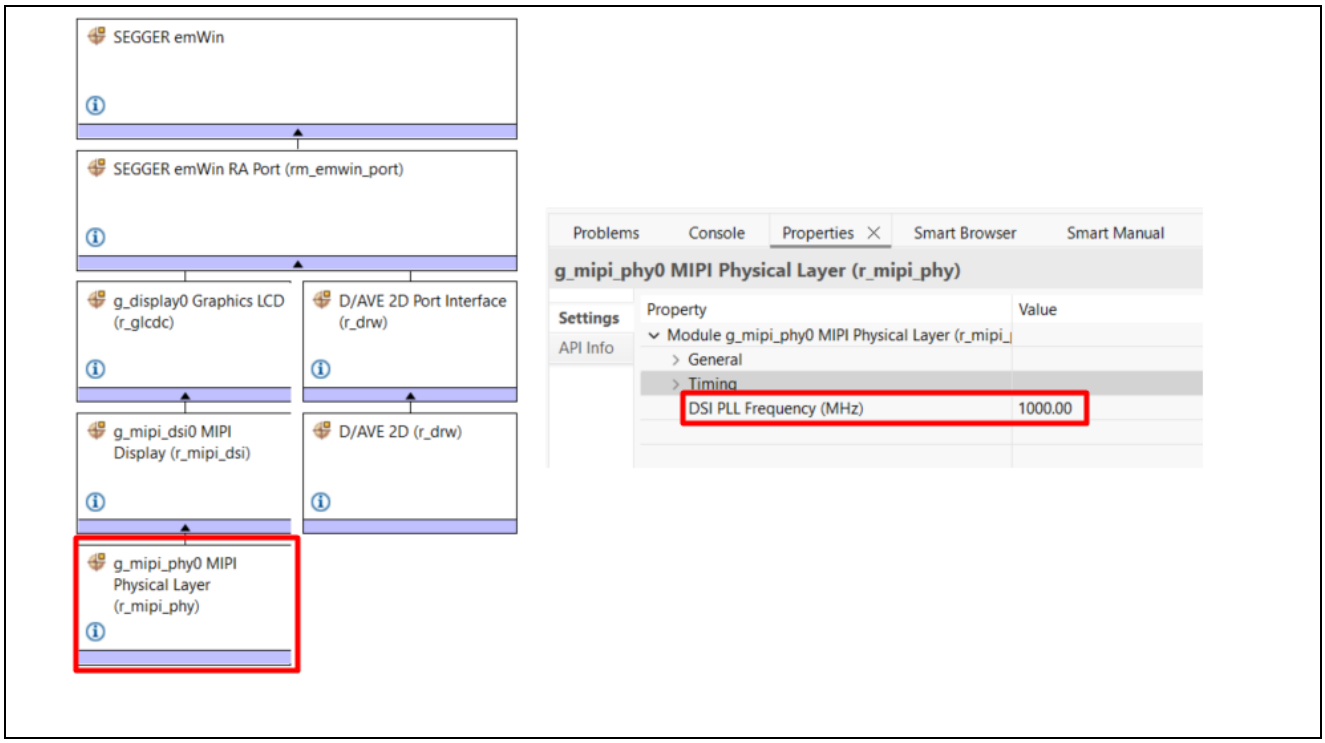


Figure 29. Example DSI PLL Frequency Clock Setting in Application.

- **PHY timing parameters:** Include low-power and high-speed transition timings such as THS-PREPARE, THS-ZERO, THS-TRAIL, TCLK-PREPARE, and TCLK-ZERO, which are critical for maintaining signal integrity and proper lane switching. FSP automatically computes the default timing values for MIPI D-PHY lanes.
- **Video mode selection** – specifies how pixel data is transmitted (Burst or Non-Burst mode).
  In the thermostat application, Burst Mode is used, which has the following characteristics:
  Sync pulses are disabled, meaning that horizontal and vertical sync events (HSE and VSE) are not transmitted.
  The bandwidths are calculated as follows:
  `GLCDC Video Clock Bandwidth (bps) = (Panel Clock MHz) * (Bits per pixel)`
  `MIPI PHY PLL Bandwidth (bps) = (MIPI Phy PLL Clock MHz / 2) * (Number of MIPI data lanes) * 8 - (Configuration Dependent Transmission Overhead).`
  The GLCDC video clock bandwidth is less than the MIPI PHY PLL bandwidth.
- **Lane configuration**: Determines the number of data lanes and their mapping to the DSI interface signals.

In the thermostat application, all these parameters are preconfigured and optimized by the FSP, ensuring proper synchronization and reliable operation of the MIPI display interface.

Octal Serial Peripheral (r_ospi_b)

In the thermostat application, the OSPI controller provides access to external flash for storing image resources used by the GUI. On the EK-RA8D2, the external OSPI flash device is interfaced with the OSPI_B controller using Chip Select 1 (CS1). The corresponding memory-mapped address range for CS1 is shown in **Figure 30**, which allows the application to directly read data from external flash as if it were internal memory.

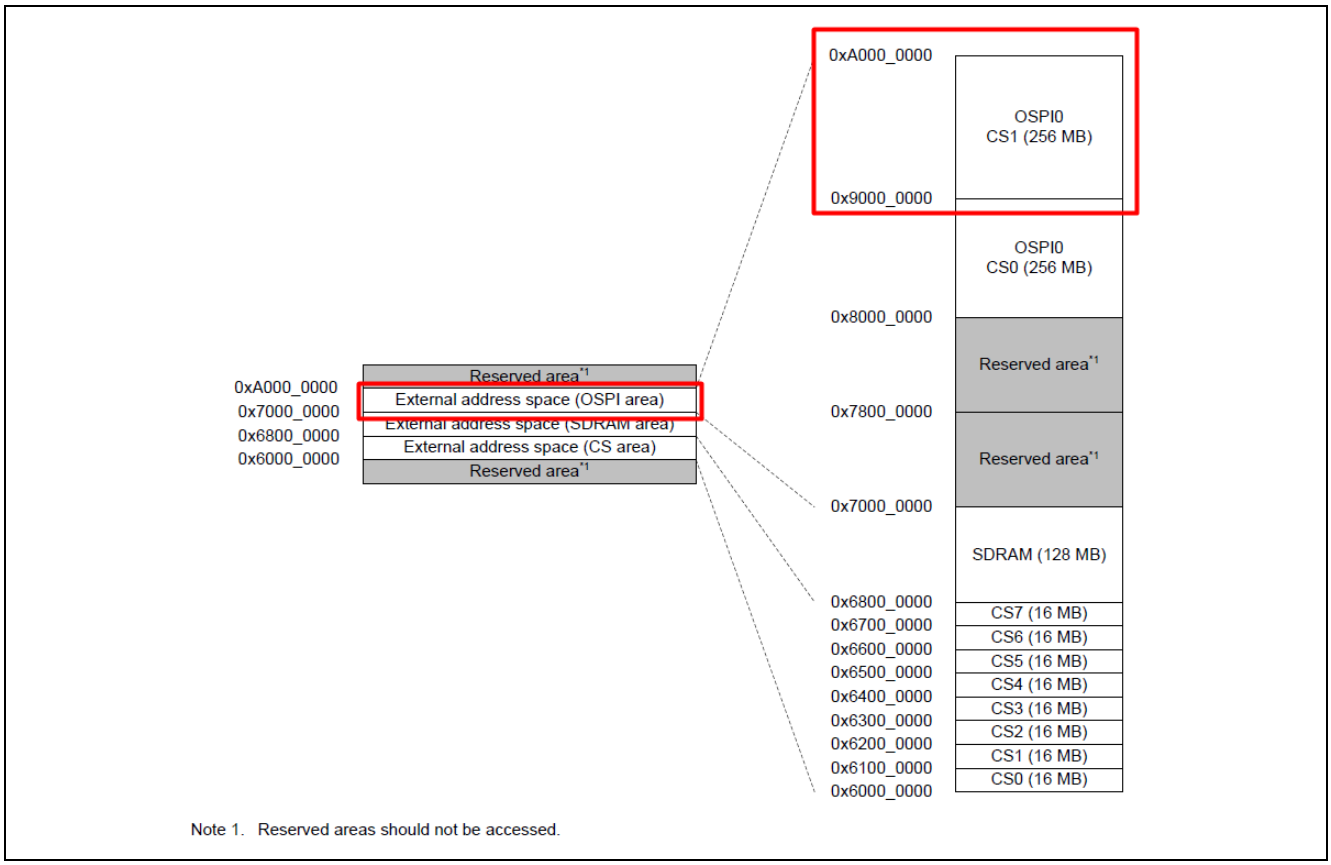


Figure 30. Detailed address map for CS area, SDRAM area, and OSPI area for SiP Product

Figure 31 below shows the OSPI General setting from r_ospi_b properties module in the thermostat application.

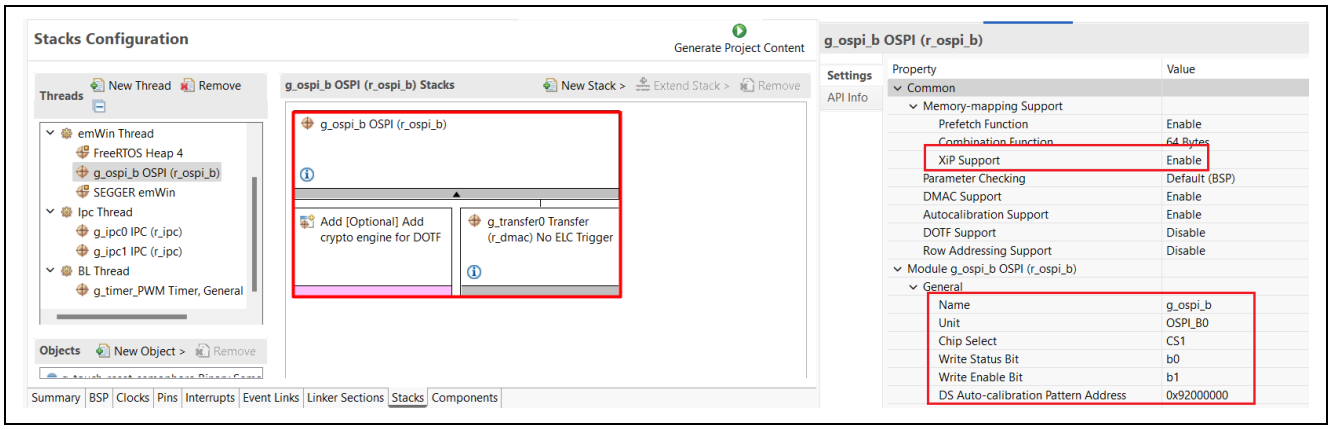


Figure 31. Example of General Setting from OSPI driver configuration in Thermostats Application.

When working with the OSPI controller, all command sequences, timings, and configuration parameters must strictly follow the external memory manufacturer's specifications to ensure reliable operation and data integrity. All command set was configured through Stack Configuration in r_ospi_b **Properties** shown in Figure 33. The pin mapping of the external flash device must also be configured to map to OSPI0 on the RA8D2. Figure 32 illustrates the pin configuration for OSPI in the thermostat application.

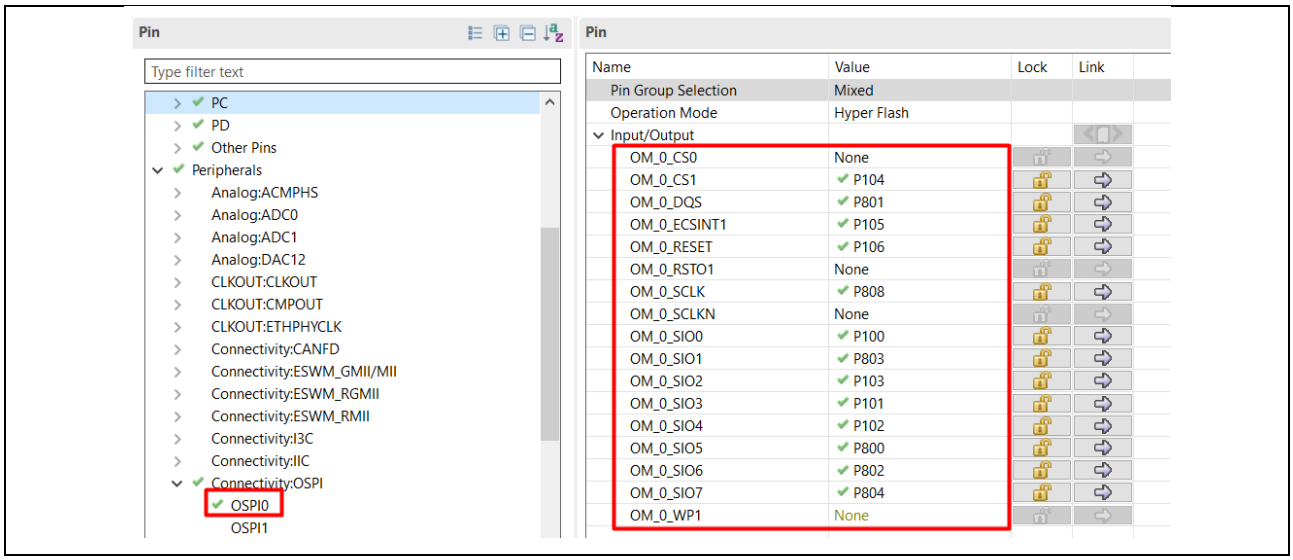


Figure 32. Example of OSPI Pin Configuration in Thermostat Application

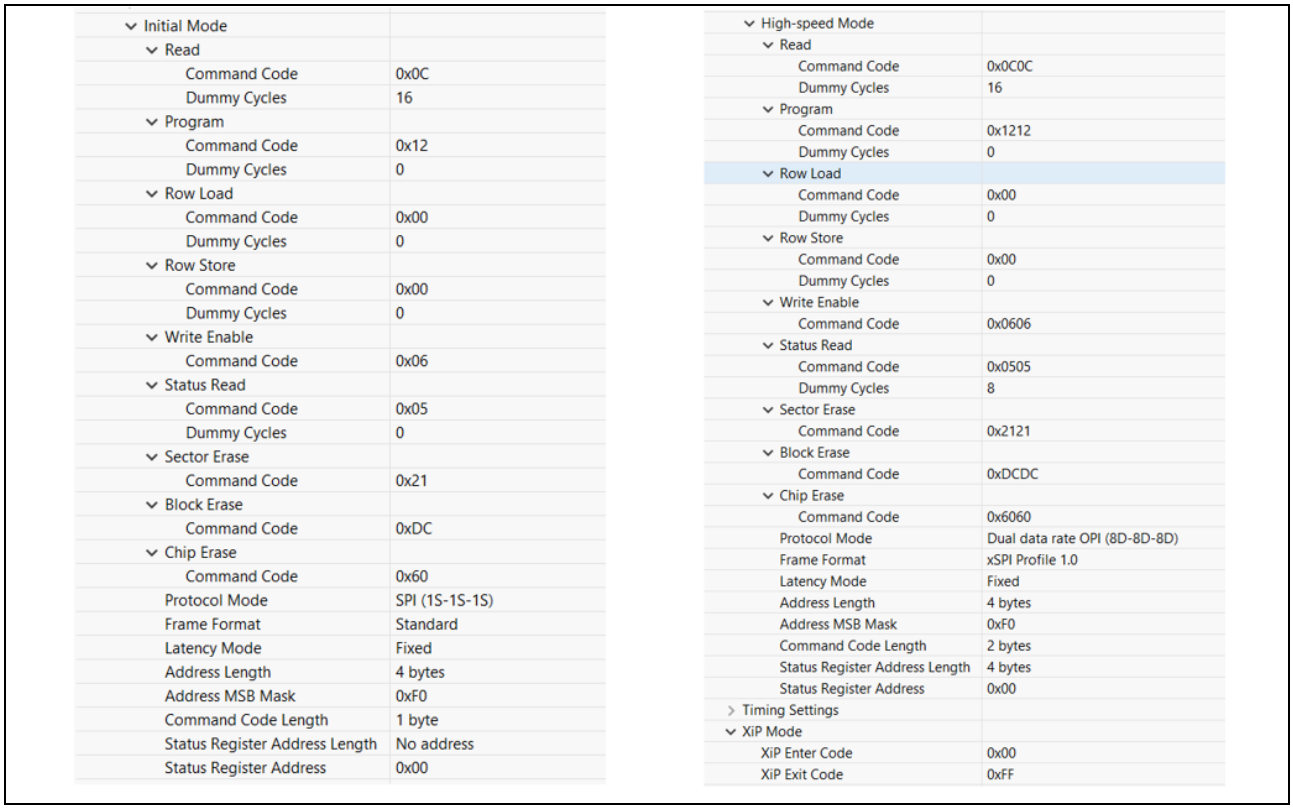


Figure 33. Example of command setting for OSPI

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.

The display's backlight enable signal (DISP_BLEN) is mapped to pin P514 on the RA8D2 MCU. Therefore, the r_gpt module is configured for Channel 13 and GTIOCB, which outputs a PWM signal to P514.

The PWM timer is set to be a sawtooth PWM wave with a 200 Hz 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.

3.4.3 CPU1 Module Configuration.

GT911 External IRQ (r_icu)

The r_icu module is used to detect external interrupt signals coming into the MCU. As shown in Figure 36 in the GT911 Touch Driver section, the GT911 IC uses the DISP_INT pin, connected to P111 on the RA8D2 MCU, to signal when touch events occur. Therefore, in the thermostat application, the external IRQ needs to be set to **Channel > 19** because this channel includes pin P111. Next, click on the **Pins > Pin Function > IRQ** scroll down and choose P111 for IRQ19.

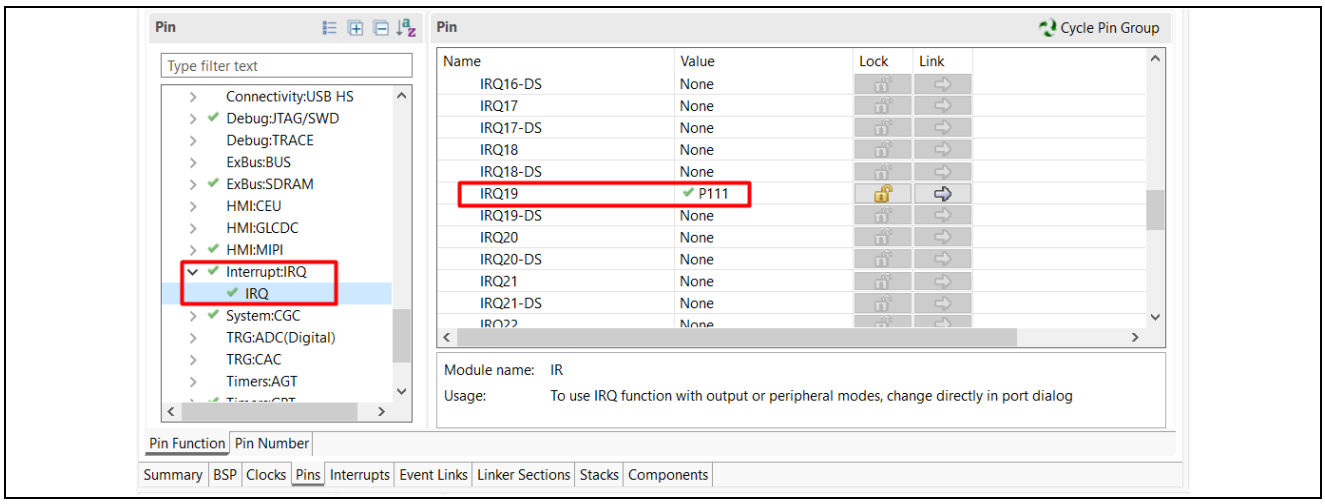


Figure 34. Ensure P111 is selected and enabled for I/O for IRQ19

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 I2C master, communicating with the GT911 slave IC over I2C. Channel 1 of the IIC peripheral is used, and the slave address **0x14** corresponds to the GT911 device.

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`.

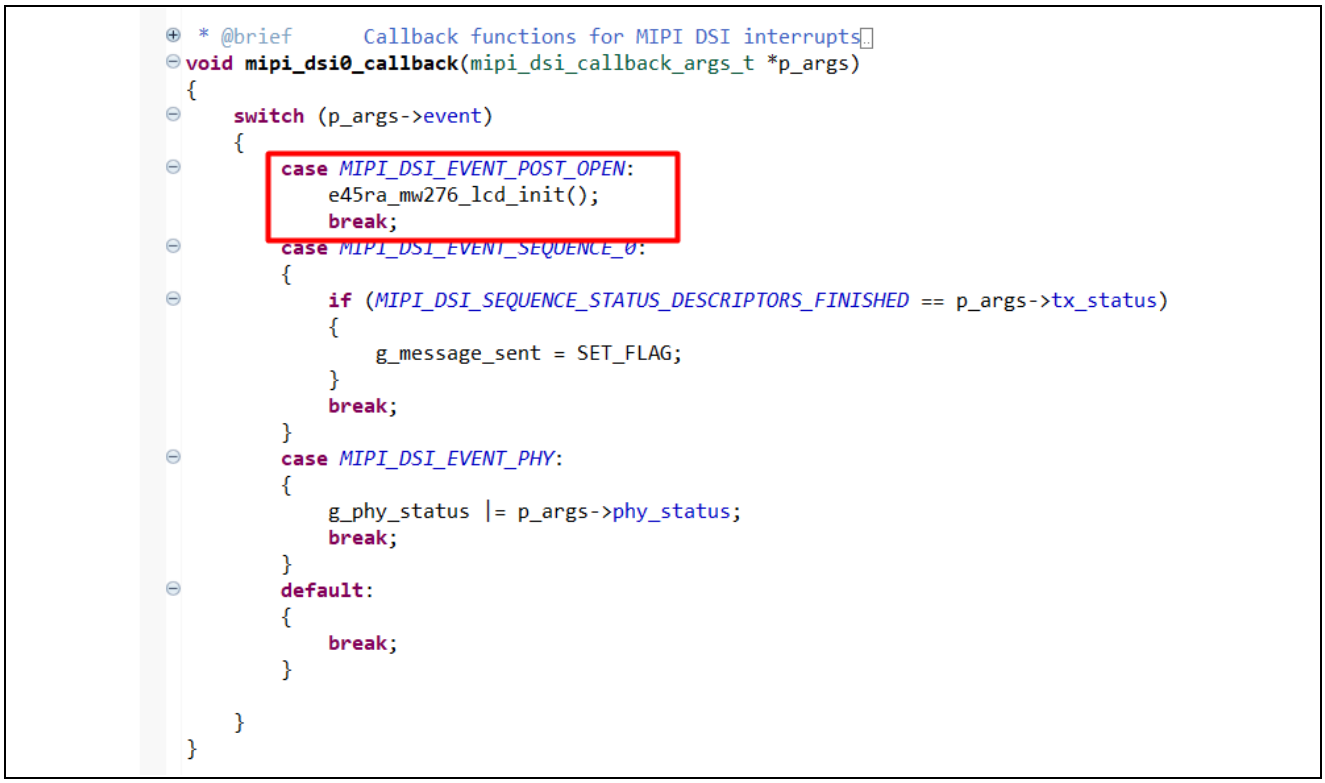
After enabling channel 1, the SCL1 pin should route to **P512,** and the SDA1 pin should route to **P511**. Use the pin configuration tab to set the pins if they don't populate automatically.

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.

3.5 Configuring the MIPI Graphic Expansion Board

The emWin thread on CPU0 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.



```
 * @brief      Callback functions for MIPI DSI interrupts
void mipi_dsi0_callback(mipi_dsi_callback_args_t *p_args)
{
    switch (p_args->event)
    {
        case MIPI_DSI_EVENT_POST_OPEN:
            e45ra_mw276_lcd_init();
            break;
        case MIPI_DSI_EVENT_SEQUENCE_0:
        {
            if (MIPI_DSI_SEQUENCE_STATUS_DESCRIPTORS_FINISHED == p_args->tx_status)
            {
                g_message_sent = SET_FLAG;
            }
            break;
        }
        case MIPI_DSI_EVENT_PHY:
        {
            g_phy_status |= p_args->phy_status;
            break;
        }
        default:
        {
            break;
        }

    }
}
```

Figure 35. 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-RA8D2'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 MIPI 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\mipi_dsi_display_control.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 RA8D2 Hardware User's Manual to understand the mechanisms of the initialization command sequence.

3.6 GT911 Touch Drivers

As mentioned in Section 1, the EK-RA8D2'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 touch 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 (Names in parentheses correspond to RA8D2 pins): VDD, GND, SCL (IIC_SCL), SDA (IIC_SDA), INT (DISP_INT), and RESET (DISP_RST).

The important signal connections from CTP to the RA8D2 are highlighted in Figure 36. The J32 header refers to the board-to-board connector, which connects the MIPI graphics expansion board to the RA8D2 MCU.

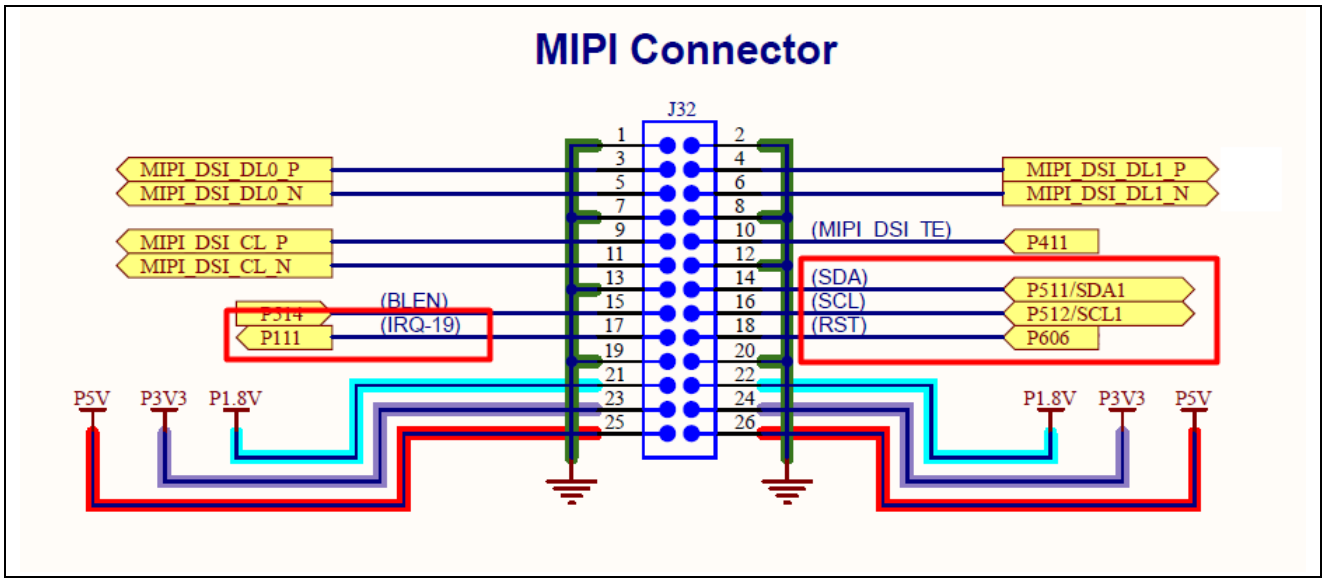


Figure 36. RA8D2 I/O connections to the GT911 IC on the external MIPI LCD

After application start-up, the MCU determines the GT911 I2C slave device address (0x14 or 0x5D) by toggling the DISP_RST and DISP_INT signals in a defined sequence. The reset logic is provided in the `R_TOUCH_GT911_Reset` driver function defined in `touch_gt911.c` from CPU1 project. However, note in the figure above that the DISP_RST display reset signal from the RA8D2 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 RA8D2 to prevent the display's configuration data from being accidentally reset. 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.

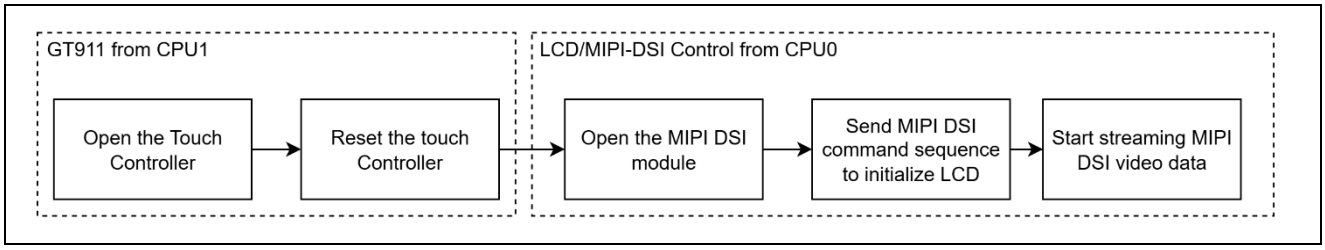


Figure 37. Recommended initialization sequence for the EK-RA8D2 MIPI LCD's Touch and MIPI controllers

The following functions are currently provided by the `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` from CPU1 project. The key parts include the initialization of GT911 configurations,

the sequence of GT911 driver functions used, and unblocking the emWin thread on CPU0 for MIPI setup after the touch reset on CPU1 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.

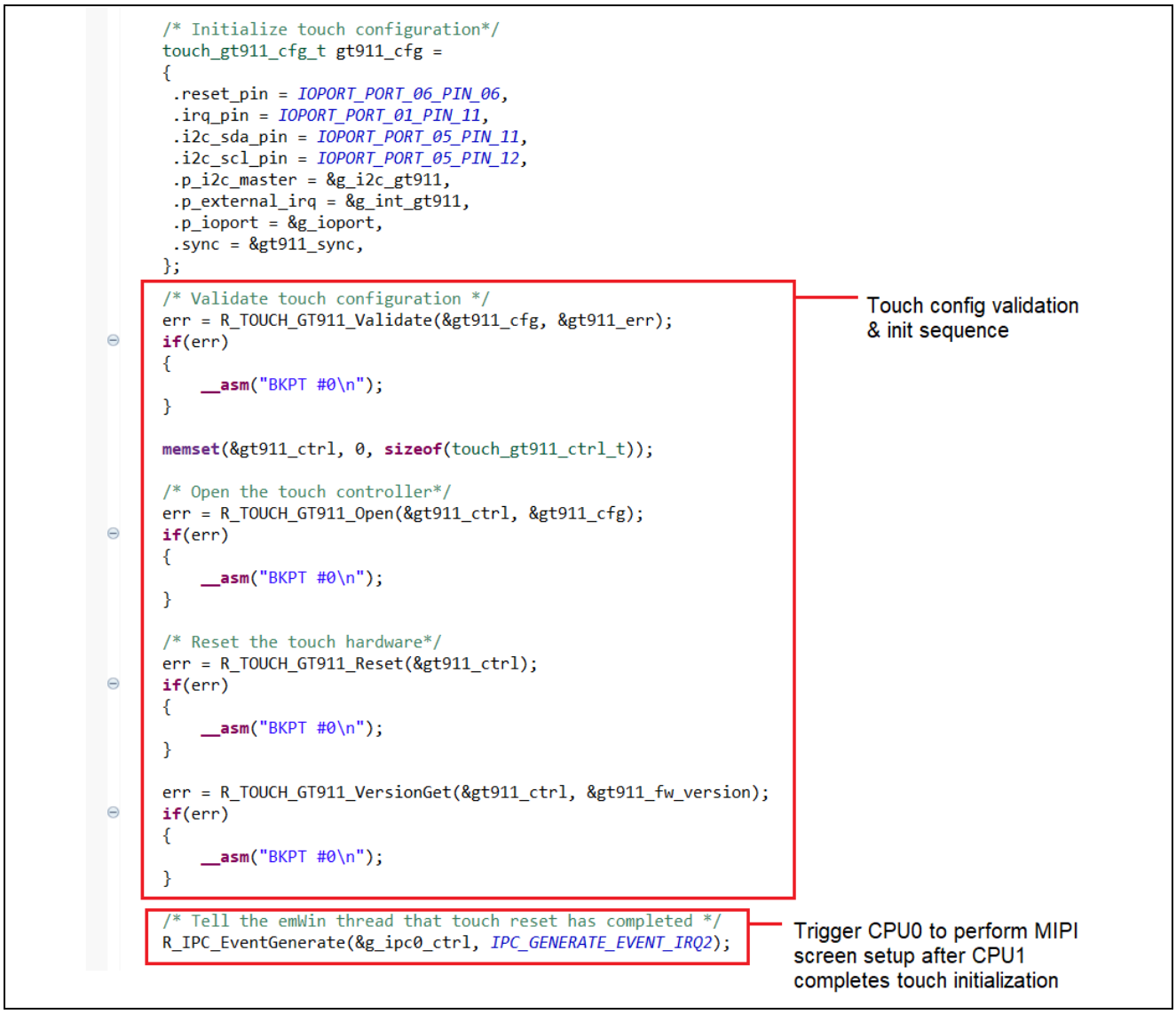


Figure 38. The GT911 drivers are used in the Touch Thread to set up communication to the capacitive touch panel

3.7 Placing Graphic Resources in External Flash Memory

When internal flash memory is insufficient, GUI resources generated by AppWizard can be stored in external OSPI flash to conserve internal memory or expand available storage.

After exporting the AppWizard project, all GUI resources are generated under the AppWizard/Resource/ directory. Image assets are located in the "`AppWizard/Resource/Image/`" folder, where each image is defined as a const C array in a separate *.c file. These arrays are wrapped with a predefined prefix and suffix to ensure consistency and proper linkage.

This application further demonstrates how to store these image assets in external flash memory by using the linker section feature in e² studio. An example configuration for the Graphic application in the CPU0 project is provided below:

1. Double clicks on **configuration.xml** in the application project example. Click **Linker Sections** tab.

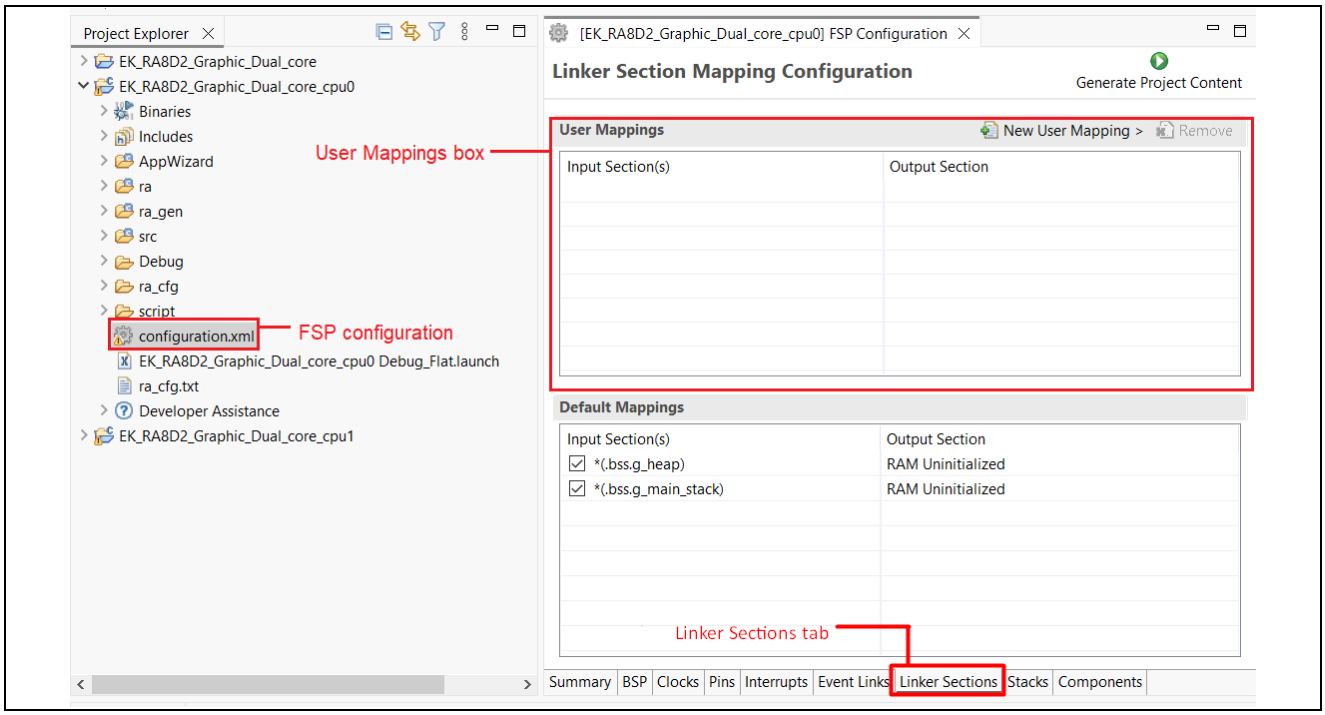


Figure 39. Linker Section Feature in e²studio

2. In **User Mappings** box, Click to **New User Mapping.** Choose **OSPI0_CS1 Constant Data** from **OSPI0_CS1**.

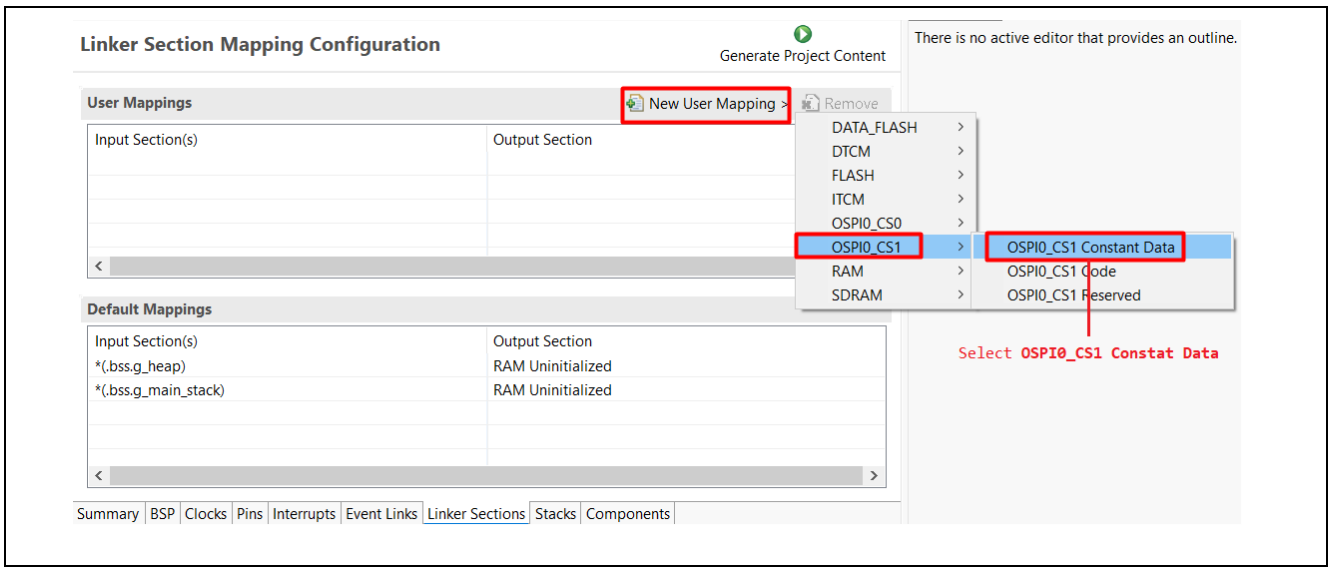


Figure 40. Example of Store Const C Array Buffer Image into OSPI device.

3. Enter the input section name or glob pattern for the new mapping.

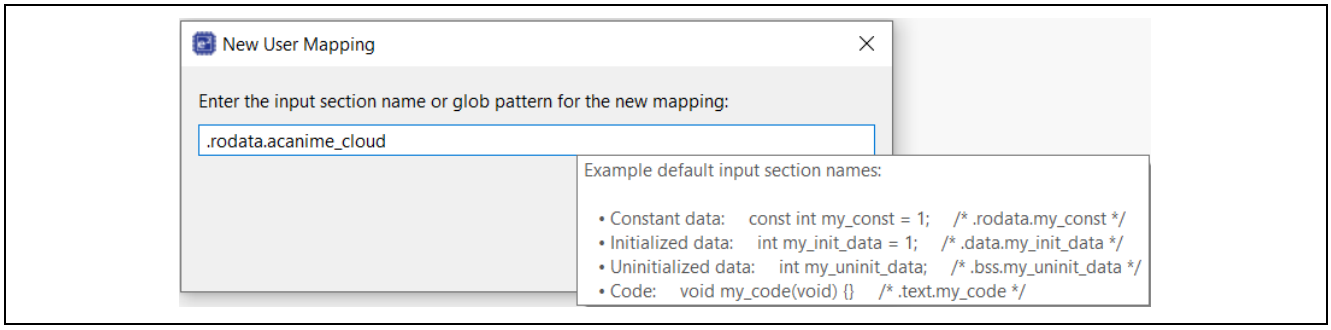


Figure 41. Example of an Input Section Name Used for Mapping Constant Data

Enter the input section name corresponding to type of data or code.

For example, with a constant image buffer “acanime_cloud”, specify the input section as: .rodata.acanime_cloud.

Repeat steps 2 and 3 for each additional image buffer that needs to be placed in external flash memory. As illustrated in Figure 42, all image resources will be placed within the memory space of CPU0 application.

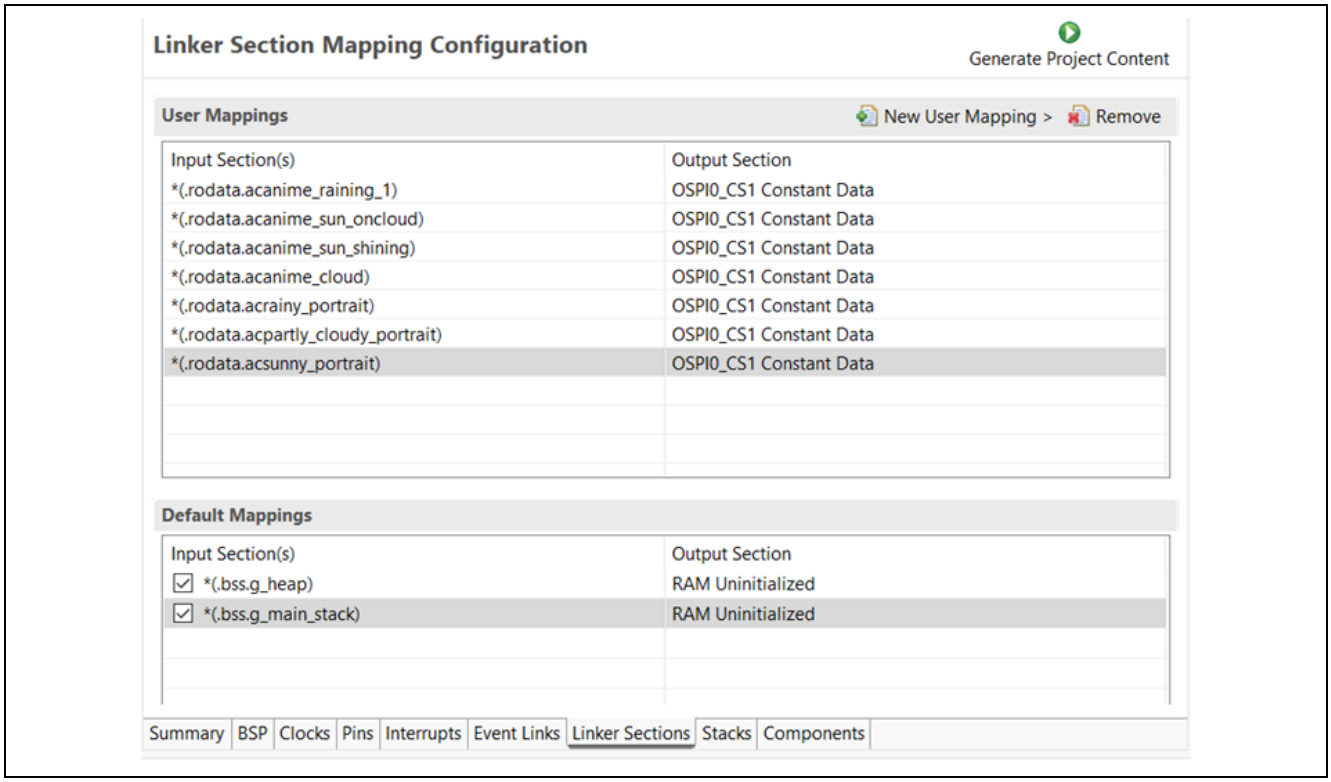


Figure 42. Example of Input Section Name in EK_RA8D2_Graphic_Dual_core_cpu0 Application.

To access GUI resources stored in external OSPI flash, the OSPI_B module must be properly configured and the external OSPI flash device initialized. The OSPI must be initialized early in the emWin thread, as shown in Figure 43 . For detailed OSPI configuration, refer to Section 3.4.

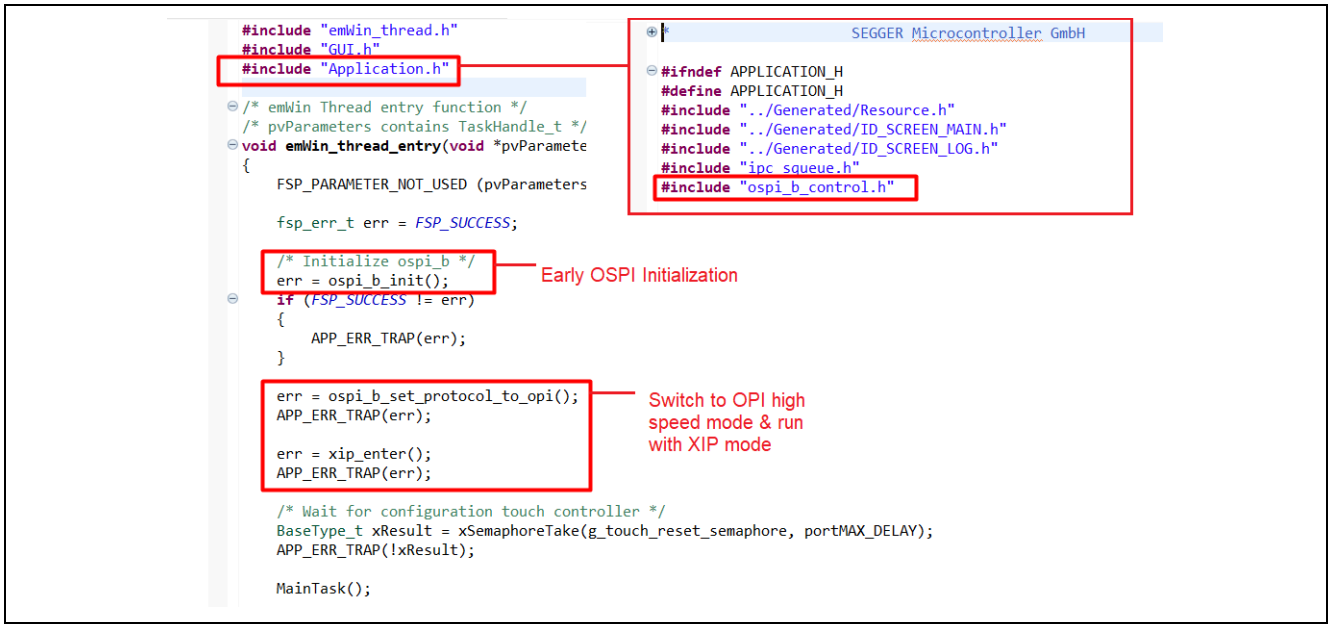


Figure 43. Example of OSPI Initialization in Application

3.8 System Performance Enhancement

In addition to the effective utilization of the dual-core architecture to distribute workloads and improve overall responsiveness, the system performance can be further enhanced by leveraging several key hardware features of the RA8D2. These hardware capabilities are designed to maximize computational throughput, improve memory access efficiency, and enhance real-time responsiveness, particularly in graphics and data-intensive applications.

3.8.1 Utilizing Cortex®-M85 Core Data Cache

In the default configuration for RA8 devices, the FSP always enables the Cortex®-M85 Instruction Cache (I-Cache). The FSP also allows for the optional enabling of the Cortex®-M85 Data Cache (D-Cache) in the BSP configuration settings, and it is disabled by default. To use D-Cache, enable it in the BSP configuration settings.
Double-click on configuration.xml, select the **BSP** tab. **Enable Data cache** in **RA8D2 Family** > **Cache settings.**

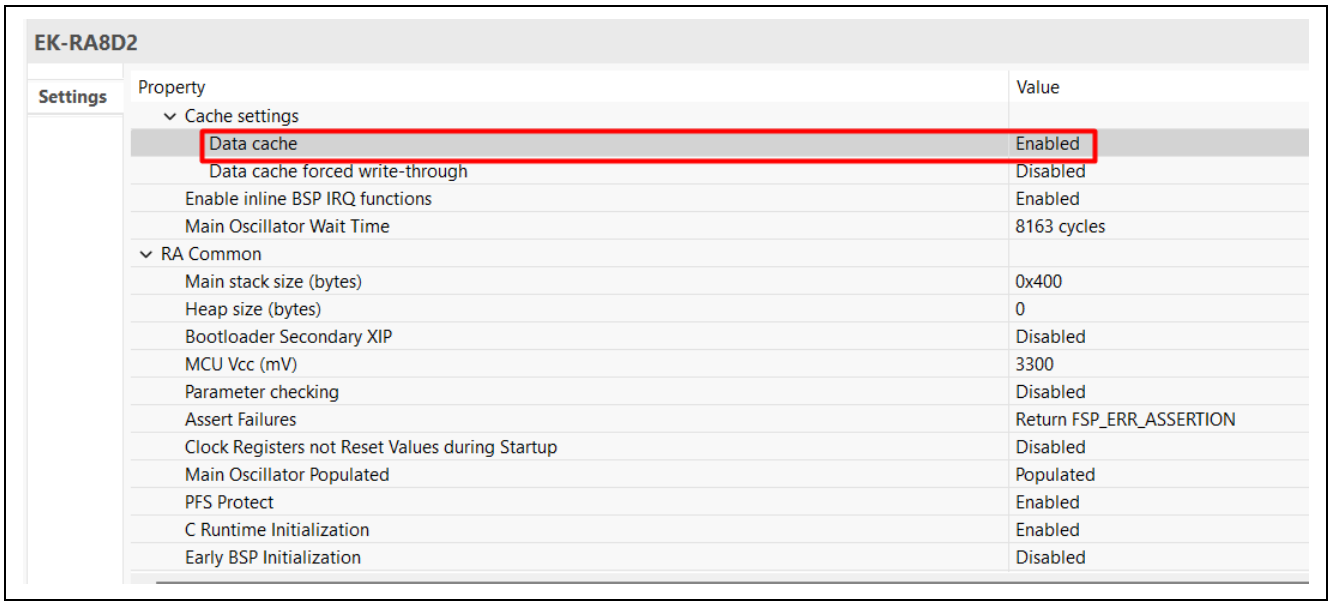


Figure 44. Example of Enable Data Cache in CPU0 project

Cache coherence should be considered when using any type of cache in a system. For further details, refer to the Cortex®-M85 Caches documentation.

3.8.2 Utilizing Helium on Cortex®-M85 for JPEG Decode

The emWin graphics library provides a software JPEG decoder that can take advantage of Arm® Helium (M-Profile Vector Extension, MVE) instructions on Cortex®-M85-based MCUs to significantly accelerate image decoding.

emWin's JPEG decode pipeline can transparently leverage the Arm Helium vector extension when available on the MCU core. The integration does not require application-level changes. The Helium-optimized firmware extensions are automatically used when the project is built with MVE-enabled toolchain flags. For detail of using Helium on Cortex®-M85, refer to the application "R01AN7127 High Performance with RA8 MCU using Arm® Cortex®-M85 core with Helium™"

3.8.3 Leveraging 32-bit SDRAM Bus for High-Speed Framebuffer Access

In modern embedded graphics applications, the speed at which framebuffers are read and written directly impacts display performance, especially for high-resolution, high-refresh-rate LCDs. A 32-bit SDRAM bus provides a wider data path than a 16-bit bus, effectively doubling the memory throughput under similar clock conditions. This allows the MCU to fill framebuffers faster, supporting smoother animations, higher color depth, and multi-layer rendering in GUIs.

Leveraging the full 32-bit bus width ensures that the graphics controller (GLCDC) or emWin framebuffer operations are not bottlenecked by memory access, which is particularly critical when performing partial or full-screen refreshes.

For the thermostat application, the framebuffer is allocated in SDRAM, as illustrated in **Figure 45**, and a 32-bit SDRAM interface is employed to provide high-speed framebuffer access.

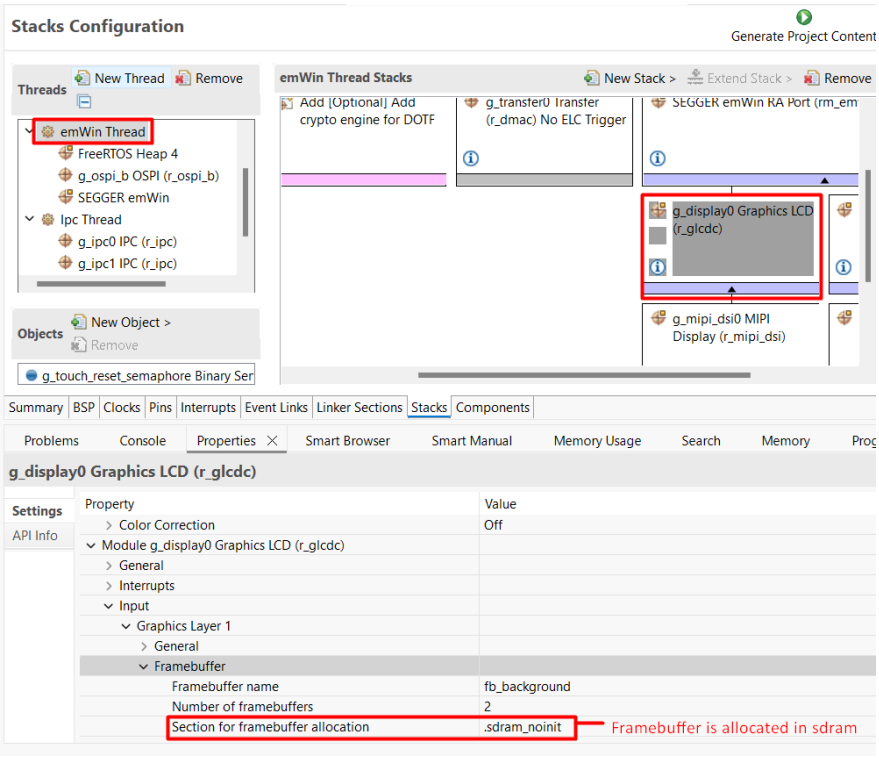


Figure 45. Example of allocating Framebuffer to SDRAM

The 32-bit bus can be enabled simply via the BSP Properties tab in the FSP Configurator, providing full memory throughput for high-speed framebuffer operations.

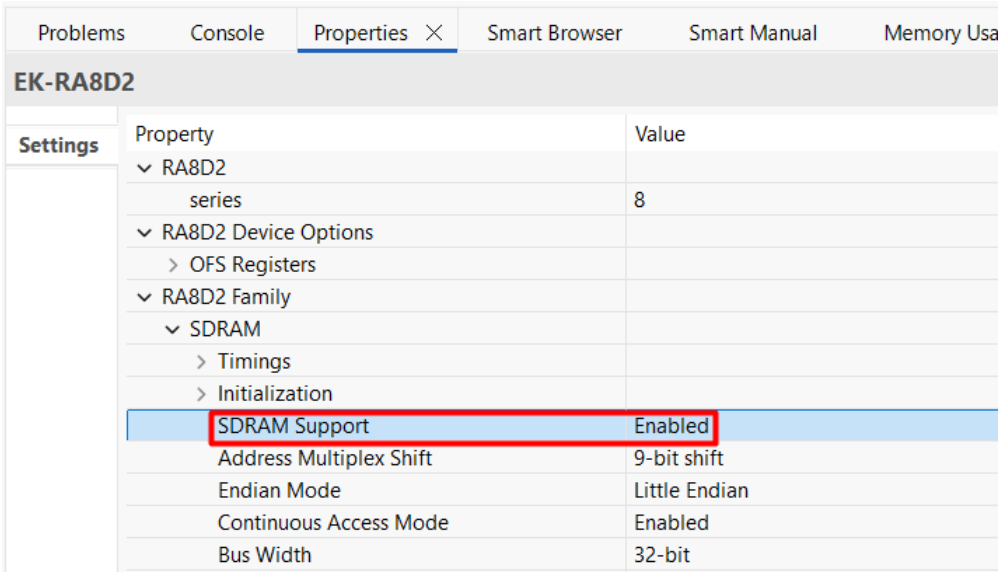


Figure 46. SDRAM Enable in BSP Properties tab

The initialization code for the SDRAM bus is included in the "`bsp_sdram.c`" file, so users do not need to configure the low-level settings manually. The only requirement for the user is to configure the SDRAM pins correctly in the **Pins** tab setup. If a different type of SDRAM is used on a custom board compared to the one on the EK board, parameters such as pin assignments, mode register settings, timing registers, and related configuration values should be set according to the SDRAM datasheet.

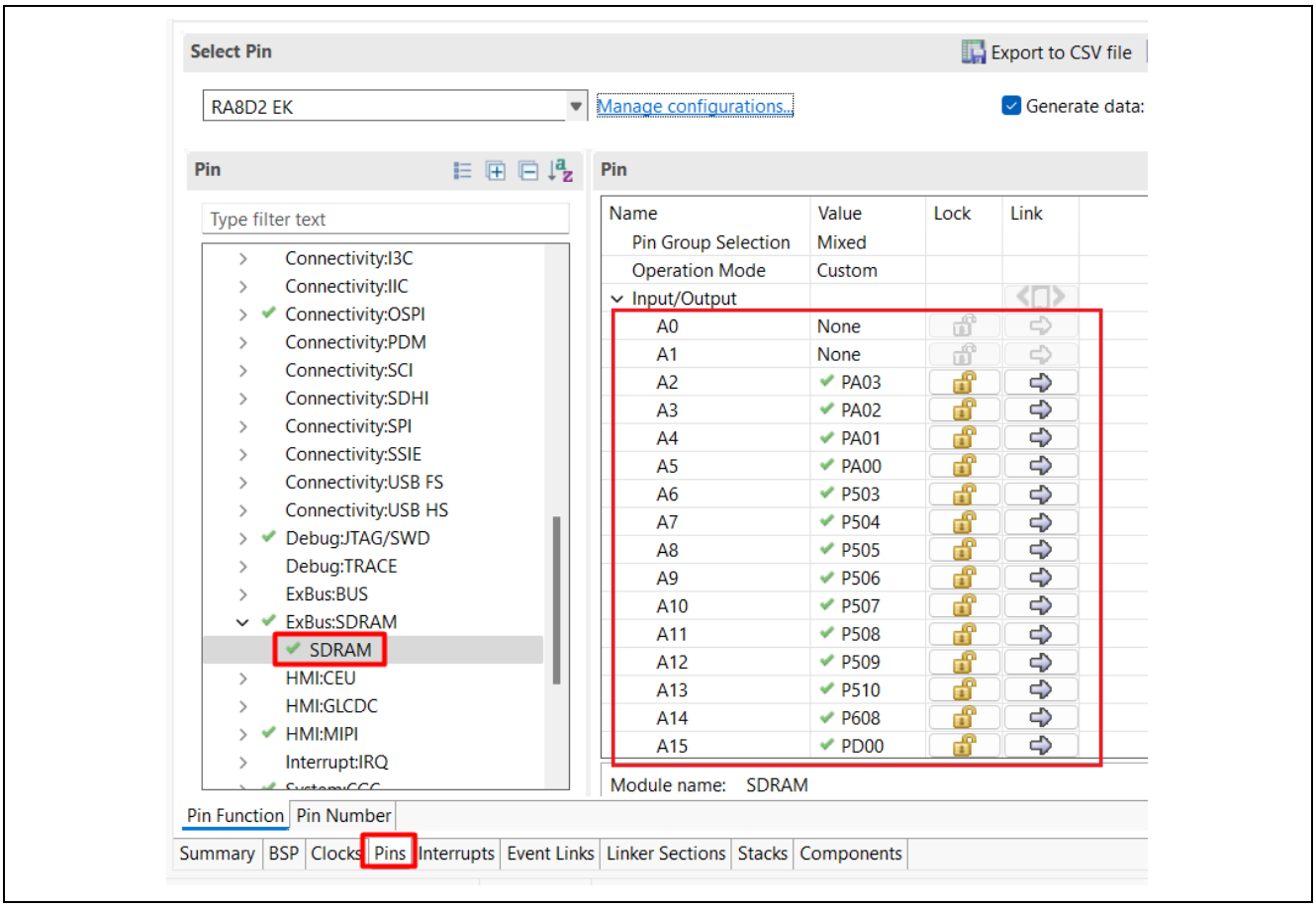


Figure 47. SDRAM Pin Configuration

```
 * This function is called at various points during the startup process.  Th
void R_BSP_WarmStart (bsp_warm_start_event_t event)
{
    if (BSP_WARM_START_RESET == event)
    {
#if BSP_FEATURE_FLASH_LP_VERSION != 0
    }

    if (BSP_WARM_START_POST_C == event)
    {
        /* C runtime environment and system clocks are setup. */

        /* Configure pins. */
        R_IOPORT_Open(&IOPORT_CFG_CTRL, &IOPORT_CFG_NAME);

#if BSP_CFG_SDRAM_ENABLED

        /* Setup SDRAM and initialize it. Must configure pins first. */
        R_BSP_SdramInit(true);
#endif
    }
}
```

Figure 48. Automatically generated SDRAM initialization added to the build.

4. Running Thermostats Application

This section outlines the key features of the graphics application. The goal of the application is to demonstrate how to develop more advanced multi-core and multi-threaded HMI applications using the Flexible Software Package (FSP), AppWizard, and the emWin graphics library.

The key goal of the FSP is to abstract much of the complexity of interfacing with various Renesas peripherals and to quickly get you to the point where you can focus on constructing more complex applications as quickly as possible.

4.1 Hardware Setup

4.1.1 Attach the MIPI LCD to the MCU

Connect CN1 on the included MIPI LCD graphics expansion board to J32 on the EK-RA8D2, located towards the bottom of the kit on the underside. J32 on the EK-RA8D2 should also be labeled “MIPI GRAPHICS EXPANSION BOARD”. If included, use the included screw to secure the display’s connection.

Connect a type C-USB cable to the Debug J10 on the EK-RA8D2 and to the host PC.

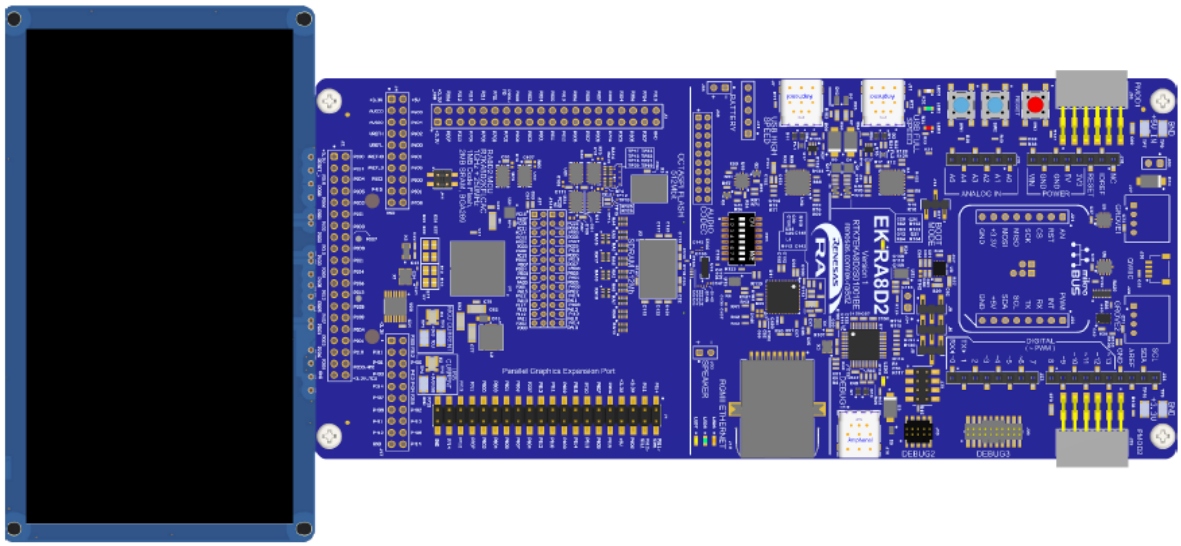


Figure 49. MIPI Graphics Expansion Board 1 connected to the EK-RA8D2

4.1.2 EK-RA8D2 Configuration Switch (SW4) Settings

Ensure that SW4 on the EK-RA8D2 has the settings listed in the table below:

Table 4. Required SW4 configurations for running the Thermostat Application Project

SW4-1 PMOD1	SW4-2 PMOD1	SW4-3 Octo-SPI	SW4-4 Arduino	SW4-5 I2C/I3C	SW4-6 MIPI Display	SW4-7 USBFS Role	SW4-8 USBHS Role
OFF	OFF	**OFF**	OFF	OFF	**ON**	OFF	OFF

4.2 Importing and Building the Project

Complete these steps to run and verify the Thermostat Graphics Application on your own EK-RA8D2:

1. Ensure that the application project folder *ek_ra8d2_graphic_mipi.zip* is downloaded onto your host PC.
2. Follow the connection steps from Section 4.1.
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. **Select archive file** *ek_ra8d2_graphic_mipi.zip*.
8. Make sure the option **Copy projects into workspace** is selected. Click **Finish**.

9. Right-click the solution project **“EK_RA8D2_Graphic_Dual_core”** and select **Build Project**. This process may take some time as it sequentially builds both subprojects: “EK_RA8D2_Graphic_Dual_core_cpu0” and “EK_RA8D2_Graphic_Dual_core_cpu1”.

4.3 Downloading and Executing on the EK-RA8D2 Kit

To connect and run the code, follow these steps:

1. Connect your PC to the DEBUG USB port of the board using a USB cable.
2. Go to **Run > Debug Configurations.**
3. Expand **Launch Group**. Click **EK_RA8D2_Graphic_Dual_core_cpu1 Debug_Multicore Launch Group > Debug**.
4. Click **Switch** to the **Debug perspective** when prompted by the e^2 studio.
5. Click **Resume** > **Resume**.
6. The Weather Panel will show as in Figure 50. Interact with the MIPI LCD to observe the full functionality of the thermostat application.

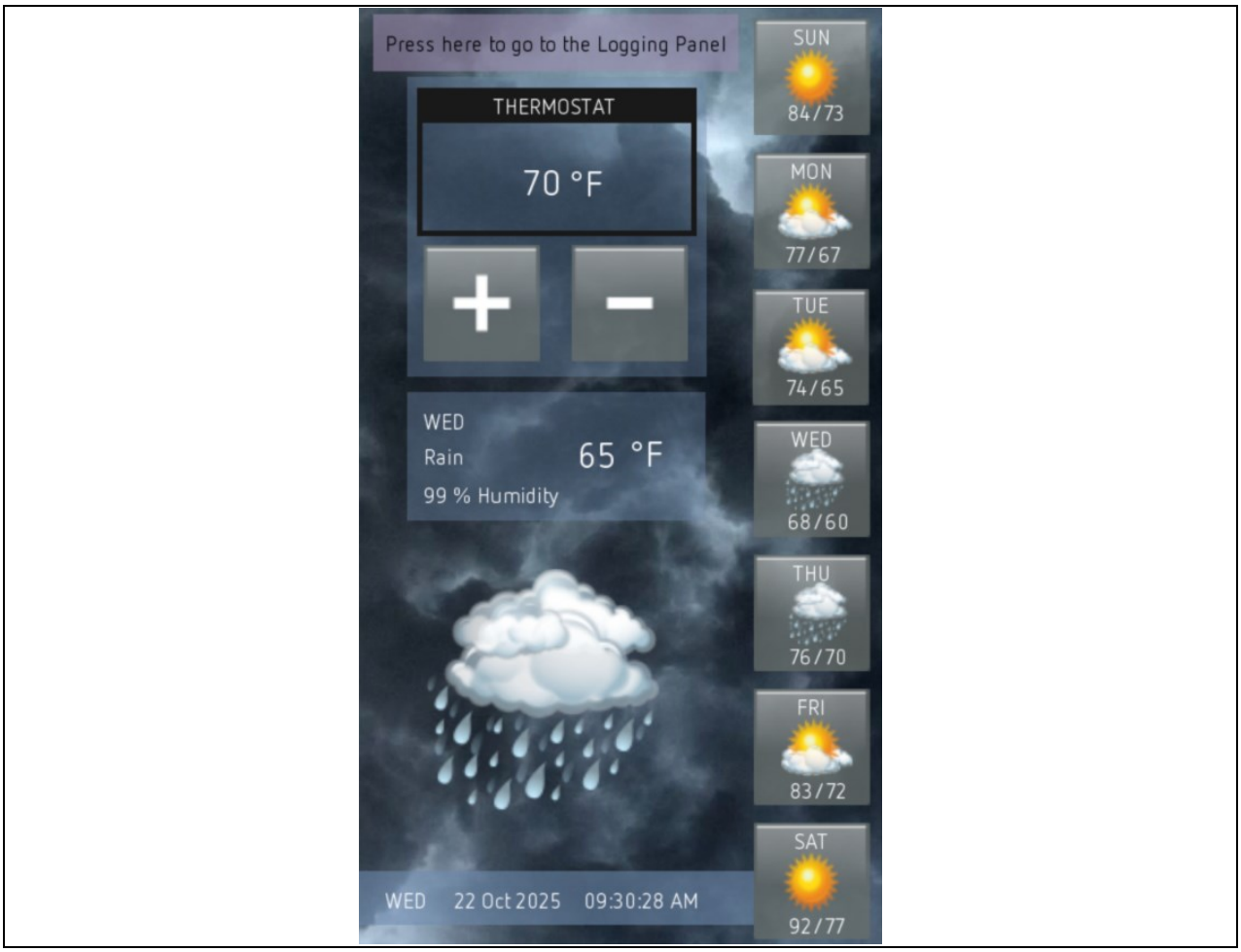


Figure 50. The Weather Panel screen is displayed after startup

5. Graphics Implementation Considerations and Trade-offs

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 RA8D2. Topics cover comparing MIPI DSI vs RGB interfaces, reviewing the memory options on the RA8D2, and understanding tradeoff relationships in the context of the RA8D2. 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 RA8D2.

5.1 MIPI DSI vs Parallel RGB

On RA8D2 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 RA8D2 Hardware User's Manual. This section, instead, will focus on the high-level differences and tradeoffs between the two graphics interfaces on RA8D2.

Take a look at the block diagram of the graphics subsystem on the RA8D2 in Figure 51 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.

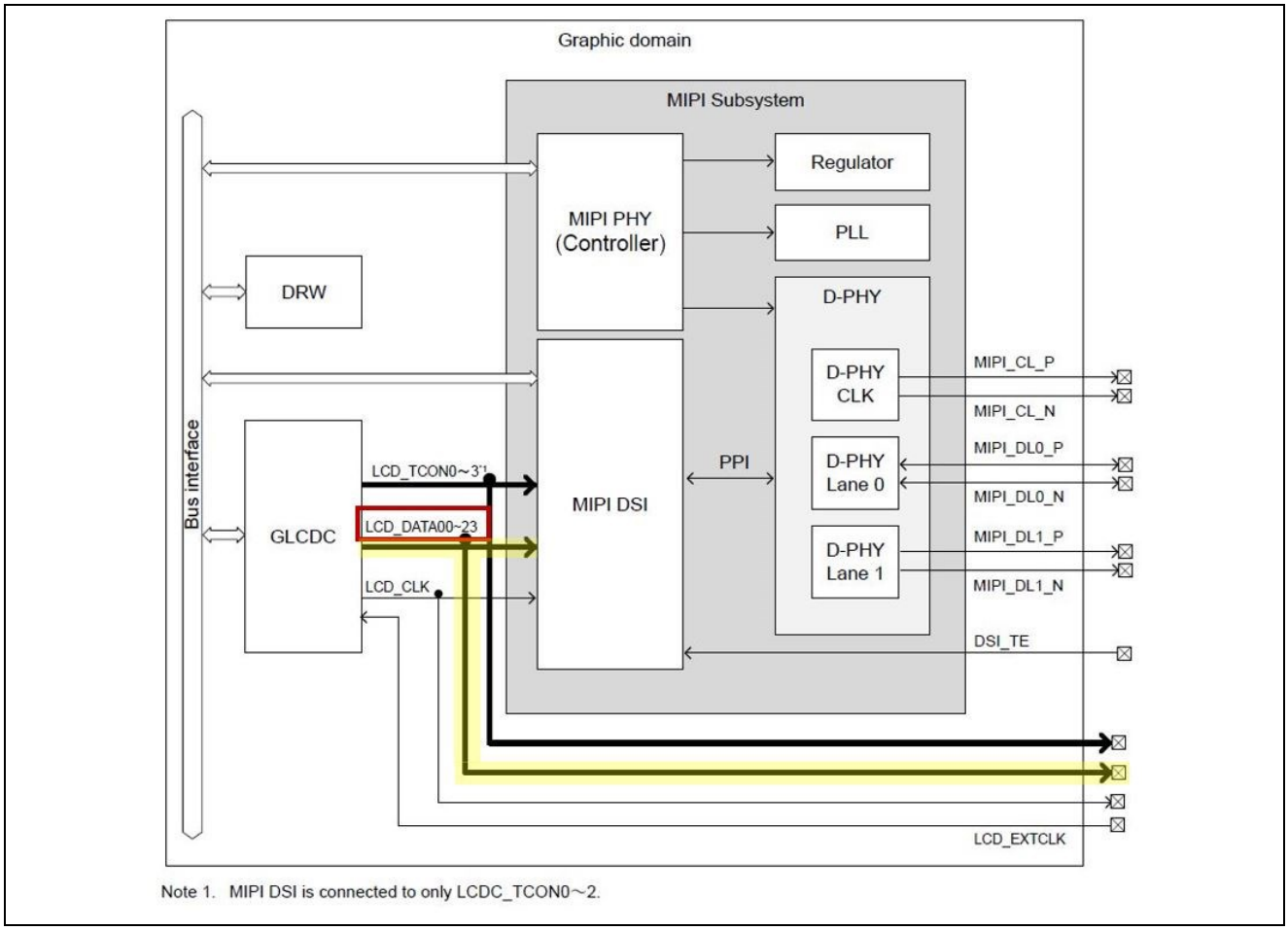


Figure 51. The GLCDC outputs parallel RGB data, routing to MCU output pins and to MIPI DSI input

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:

5.1.1 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 RA8D2 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 RA8D2, the maximum achievable throughputs of the MIPI DSI and Parallel RGB interfaces are equivalent to one another.

5.1.2 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.

5.1.3 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.

5.1.4 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.

5.1.5 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.

5.2 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 RA8D2 will be mentioned for each aspect.

5.2.1 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 the RA8D2, 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 RA8D2, 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.

5.2.2 Color Format

The color format specifies the number of bits that represent the red, blue, and green information for each pixel on a display. RA8D2'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)

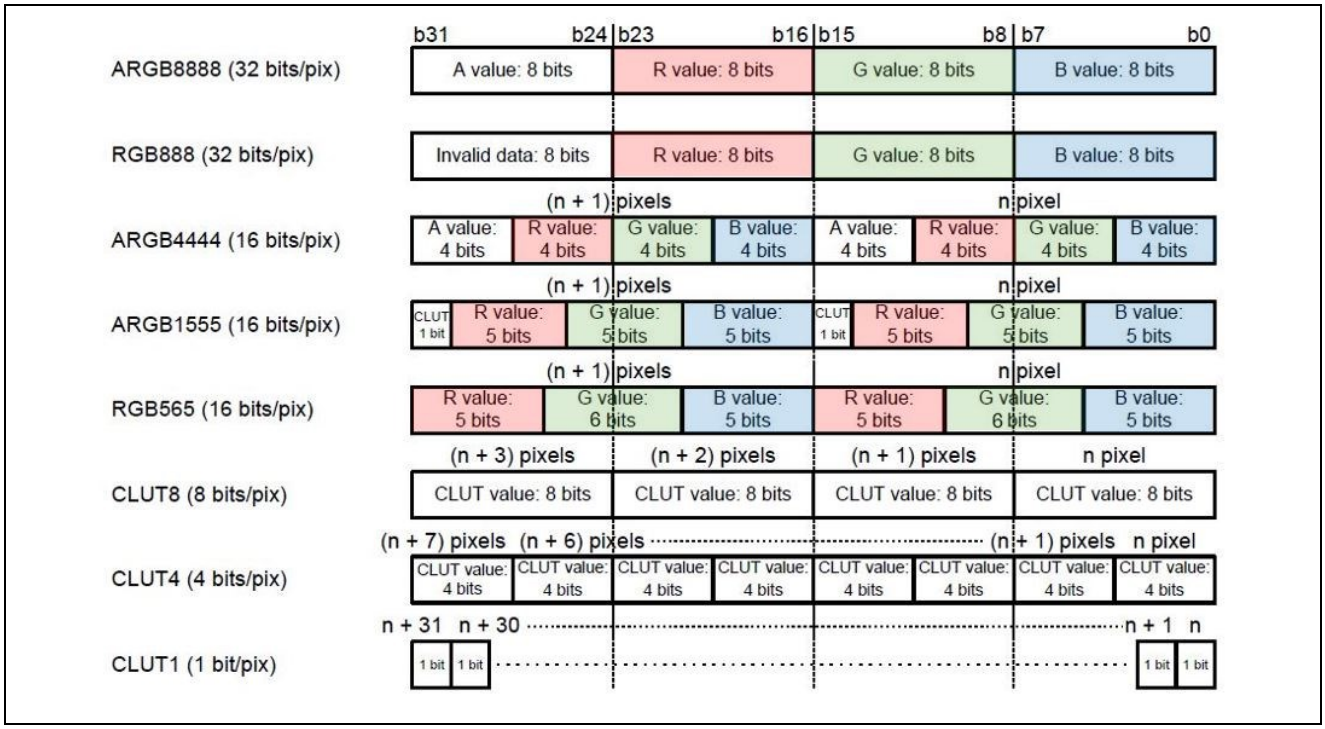


Figure 52. Pixel data formats of the RA8D2 GLCDC

The external MIPI display on the EK-RA8D2 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.

5.2.3 Framerate

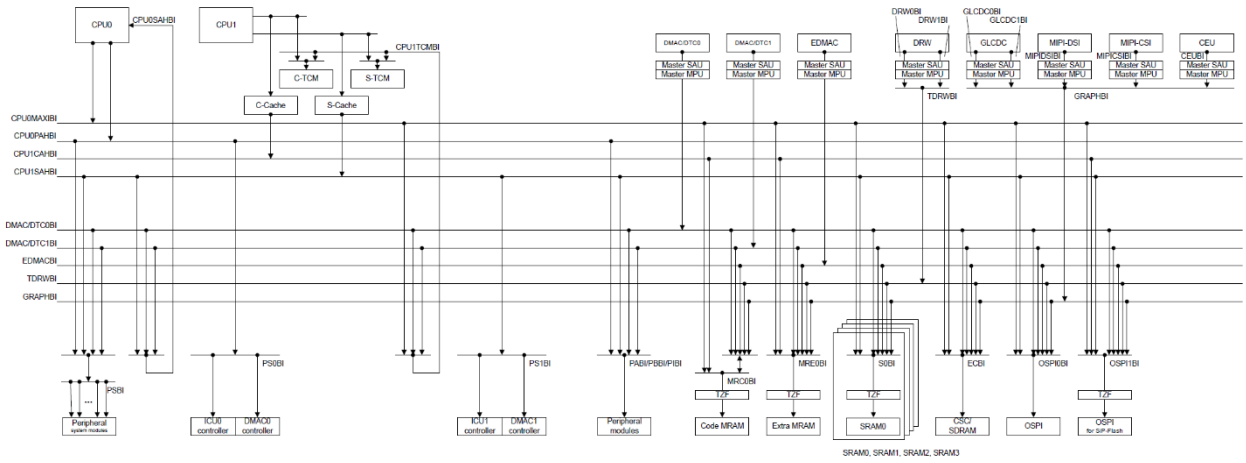
The framerate is measured in frames per second (FPS) and refers to the number of times the framebuffer is refreshed on the display. Unlike the resolution and color format, the framerate is not a pre-set value restricted by one component on the RA8D2. It depends on the processing capabilities of the graphics 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 RA8D2, the D/AVE 2D hardware module helps to offload some of the drawing tasks from the CPU and increase overall performance.

5.2.4 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 buses interfacing between the CPU, GLCDC, DRW, MIPI, and memory components like the SDRAM, SRAM and OSPI 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 the RA8D2:



Note: "TZF" is Slave TrustZone Filter.

Figure 53. Bus map of the RA8D2 indicating commonly used modules for graphics applications

In the RA8D2 MCU group, there are devices with a 16-bit SDRAM bus and devices with a 32-bit SDRAM bus like that of the EK-RA8D2. For more information, visit the Buses section of the RA8D2 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.

5.2.5 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 the RA8D2, the total SRAM is 2MB (256 KB of CM85 TCM RAM, 128 KB CM33 TCM RAM, 1664 KB of user SRAM)

Since there's no hardware JPEG decoder on RA8D2 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. Introducing QE for Display Application Development

QE for Display [RX, RA] is an intuitive plug-in for the **e² studio** development environment, designed to accelerate GUI-based development on **Renesas RA microcontrollers**. With its powerful graphical configuration and visualization capabilities, the tool simplifies display setup, shortens development time, and enhances the overall efficiency of embedded system design involving display devices.

QE for Display [RX, RA] V3.0.0 and later supports RA family. With the FSP Visualization function, the results of timing adjustment and image quality adjustment for LCD panel are directly reflected in FSP. Also, a GUI can be created using SEGGER's emWin software package.

QE for Display [RA] covers everything from the initial adjustment of the display to the creation of designs for screens. It can also be interlinked with various GUI development solutions to provide total support for the development of GUIs within short timeframes.

Note: The FSP visualization function is available by opening the “Stacks” tab in FSP Configuration and selecting the added “r_glcdc”.

6.1 Installation and Uninstallation

6.1.1 Install from the "Renesas Software Installer" menu of e² studio

1. Start e² studio.
2. Select the [Renesas Views] - [Renesas Software Installer] menu of e² studio to open the [Renesas Software Installer] dialog box.
3. Select the [Renesas QE] and click the [Next>] button
4. Select the [QE for Display[RX,RA](v3.0.0)] check box, and click the [Finish] button.
5. Check that [Renesas QE for Display[RX,RA]] is selected in the [Install] dialog box, and click the [Next>] button.
6. Check that [Renesas QE for Display[RX,RA]] is selected as the target of installation, and click the [Next>] button.
7. After confirming the license agreements, if you agree to the license, select the [I accept the terms of the license agreements] radio button, and click the [Finish] button.
8. If the dialog of the trust certificate is displayed, check that certificate, and click the [OK] button to continue installation.
9. When prompted to restart e² studio, restart it.
10. Start this product from the [Renesas Views] - [Renesas QE] menu of e² studio. For details about how to use this product, see the [Help] menu of e2 studio.

6.1.2 Install using QE (zip file) downloaded from the Renesas website

1. Start e2 studio.
2. From the [Help] menu, select [Install New Software...] to open the [Install] dialog box.
3. Click the [Add...] button to open the [Add Repository] dialog box.
4. Click the [Archive] button, select " RenesasQE_Display_RXRA_V370.zip" in the opened dialog box, and click the [Open] button.

   Note: The file "RenesasQE_Display_RXRA_V370.zip" can be obtained by extracting "RenesasQE_Display_V370.zip" downloaded from the Renesas website.
5. Click the [OK] button in the [Add Repository] dialog box.
6. Expand the [Renesas QE] item shown in the [Install] dialog box, select the [Renesas QE for Display[RX,RA]] check box, and then click the [Next>] button.
   * If you check off the [Contact all update sites during install to find required software] checkbox, you can shorten the installation time.
7. Check that [Renesas QE for Display[RX,RA]] is selected as the target of installation, and click the [Next>] button.
8. After confirming the license agreements, if you agree to the license, select the [I accept the terms of the license agreements] radio button, and click the [Finish] button.
9. If the dialog of the trust certificate is displayed, check that certificate, and click the [OK] button to continue installation.
10. When prompted to restart e² studio, restart it.
11. Start this product from the [Renesas Views] - [Renesas QE] menu of e² studio. For details about how to use this product, see the [Help] menu of e² studio.

6.1.3 Uninstalling QE Product

Use the following procedure to uninstall this product.

1. Start e² studio.
2. Select [Help -> About e2 studio] to open the [About e2 studio] dialog box.
3. Click the [Installation Details] button to open the [e2 studio Installation Details] dialog box.
4. Select [Renesas QE for Display[RX,RA]] displayed on the [Installed Software] tabbed page and click the [Uninstall...] button to open the [Uninstall] dialog box.
5. Check the displayed information and click the [Finish] button.
6. When prompted to restart e2 studio, restart it.

6.2 Development Step with RA device.

Follow the procedure below to develop graphics application with QE tool support.

1. Create a new e2 studio project.

Start e² studio and create new project with RA device that supports graphic application.

2. Add the "Display User Interface Application" stack to the e2 studio project.

Open **Stacks** tab in "**configuration.xml**". Add **New Stack** for Graphic QE illustrated in Figure 54.

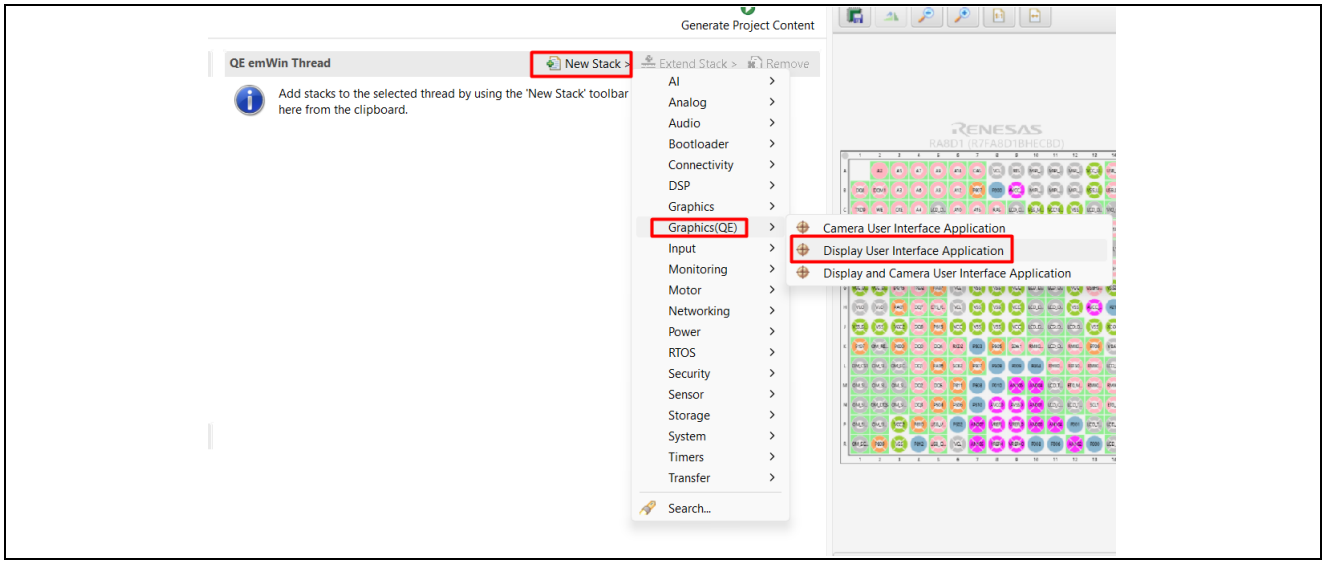


Figure 54. Example of Add Graphic (QE) stacks for Display User Interface Application

3. Open the LCD/Camera Workflow (QE) view.

Click **Renesas Views > Renesas QE > Display/Camera Tuning RX, RA(QE)** to follow configure graphic application with QE tool.

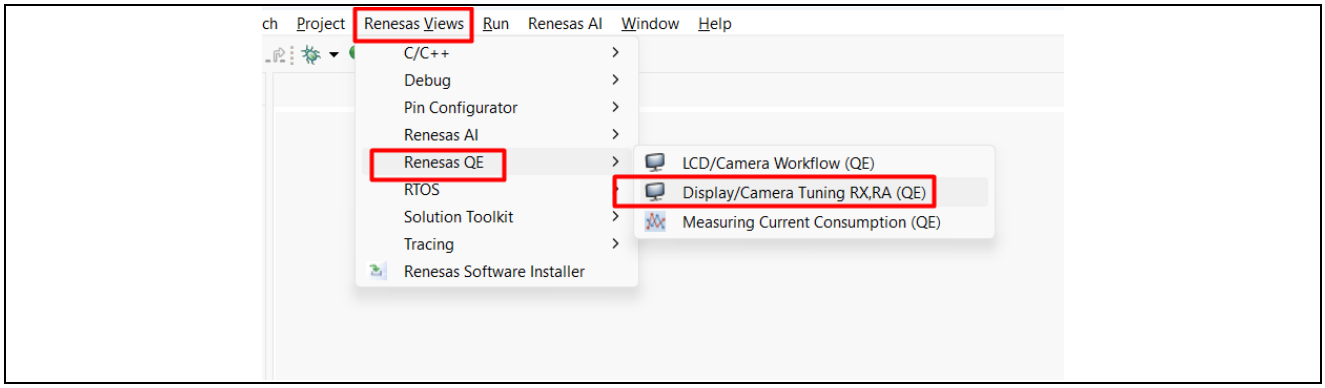


Figure 55. Open Display/Camera Tuning RX, RA (QE)

As shown in the Figure 56, QE for Display [RA] provides a clear workflow interface that visually represents the development process. Each step in the workflow, from Preparation to LCD and Camera adjustment is

intuitively arranged to support efficient and error-free development. Follow step by step in Graphic Workflow to configure whole Display application with QE.

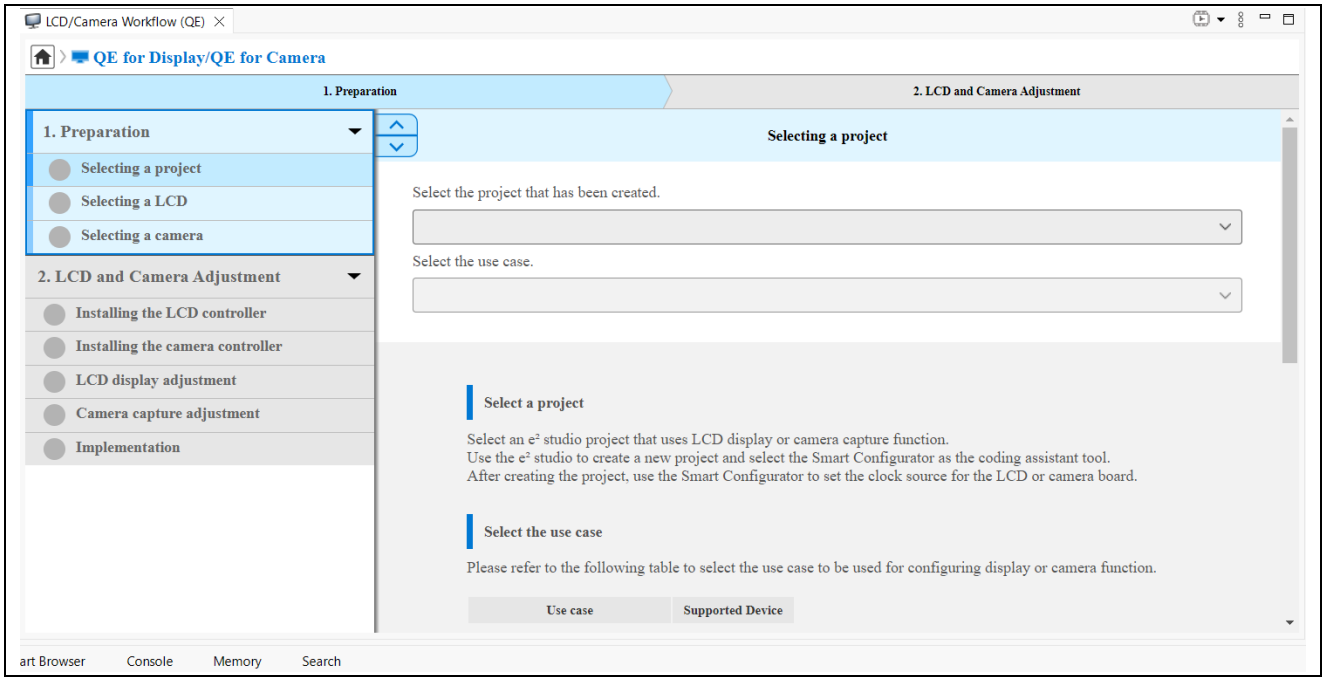


Figure 56. LCD/Camera Workflow (QE) View

4. Select a project.

From **Selecting a Project**, select the project created before with **Display User Interface Application** stack added then select the use case for application.

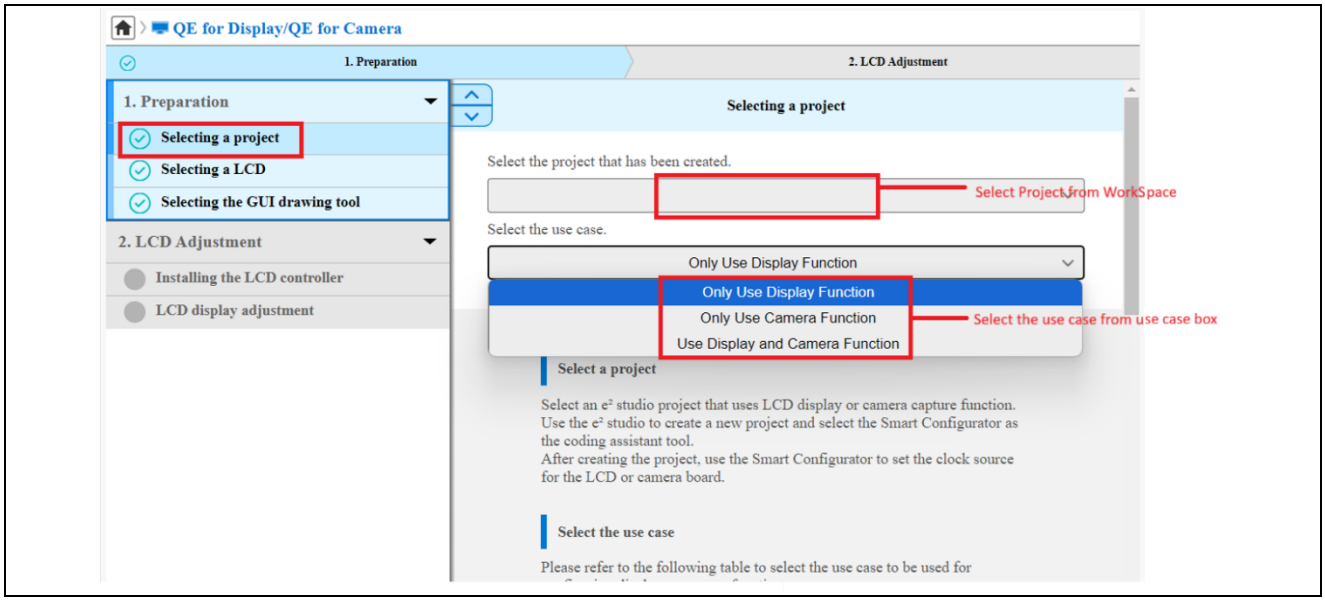


Figure 57. Project and Use Case Selection

5. Select an LCD.

Choose the appropriate LCD panel used in your application. For projects created using a Renesas Evaluation Kit, the LCD settings are preconfigured to match the corresponding RA board. However, when working with a custom board, you can manually add and configure a custom LCD in this section.

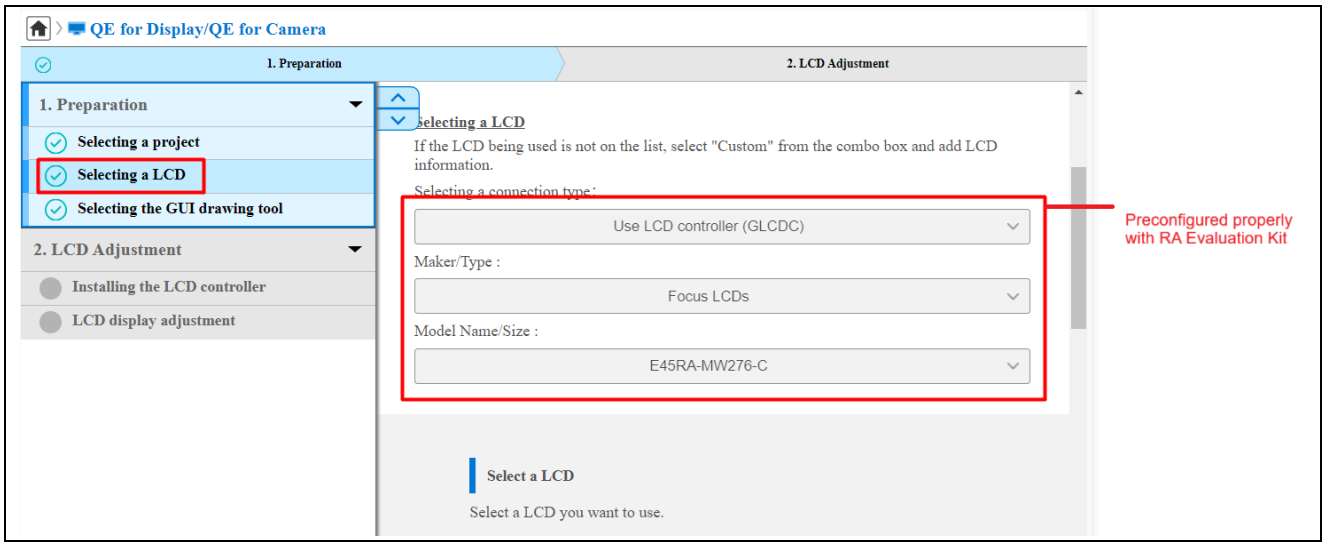


Figure 58. Example of Selecting a LCD

6. Choose the GUI Drawing Tool.

The QE for Display [RA] application currently supports SEGGER's emWin as the embedded graphics builder for GUI design and development.

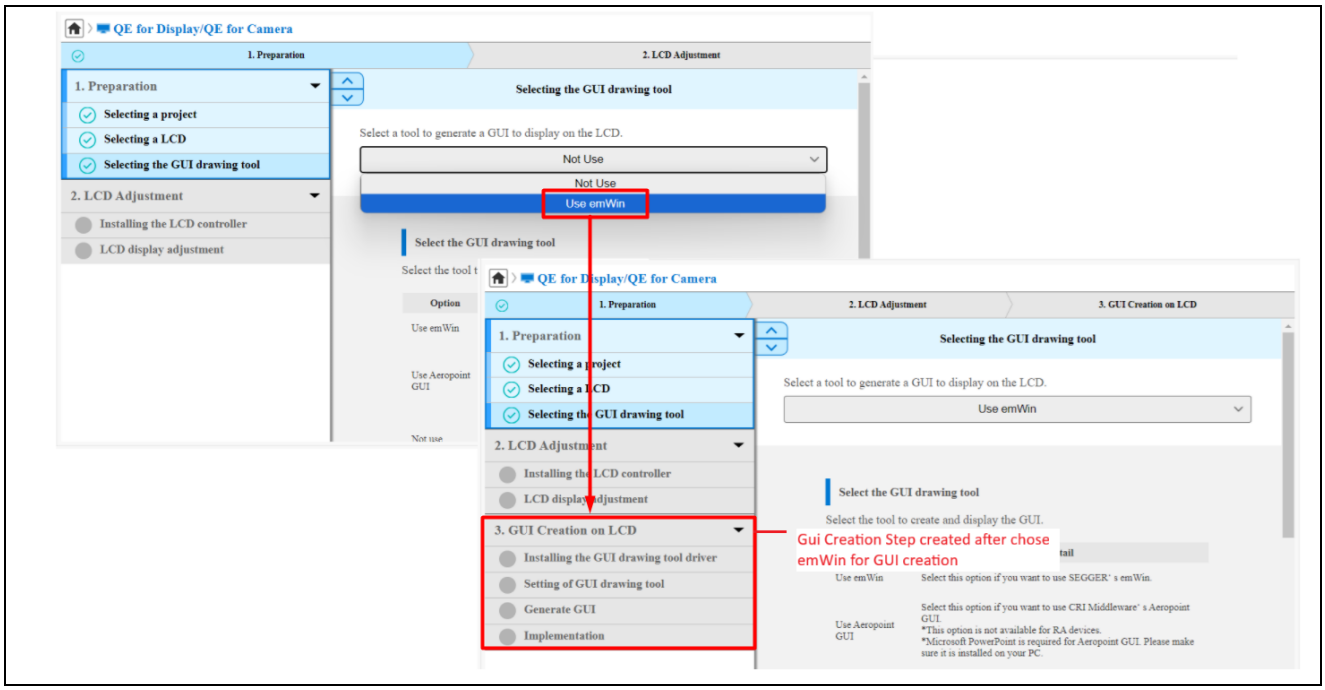


Figure 59. Chose emWin as Embedded Graphics Builder.

7. Add an LCD controller.

Under the Display User Interface Application Stacks section in configuration.xml, add SEGGER emWin as the emWin Driver.

This component automatically includes the r_glcdc LCD controller.

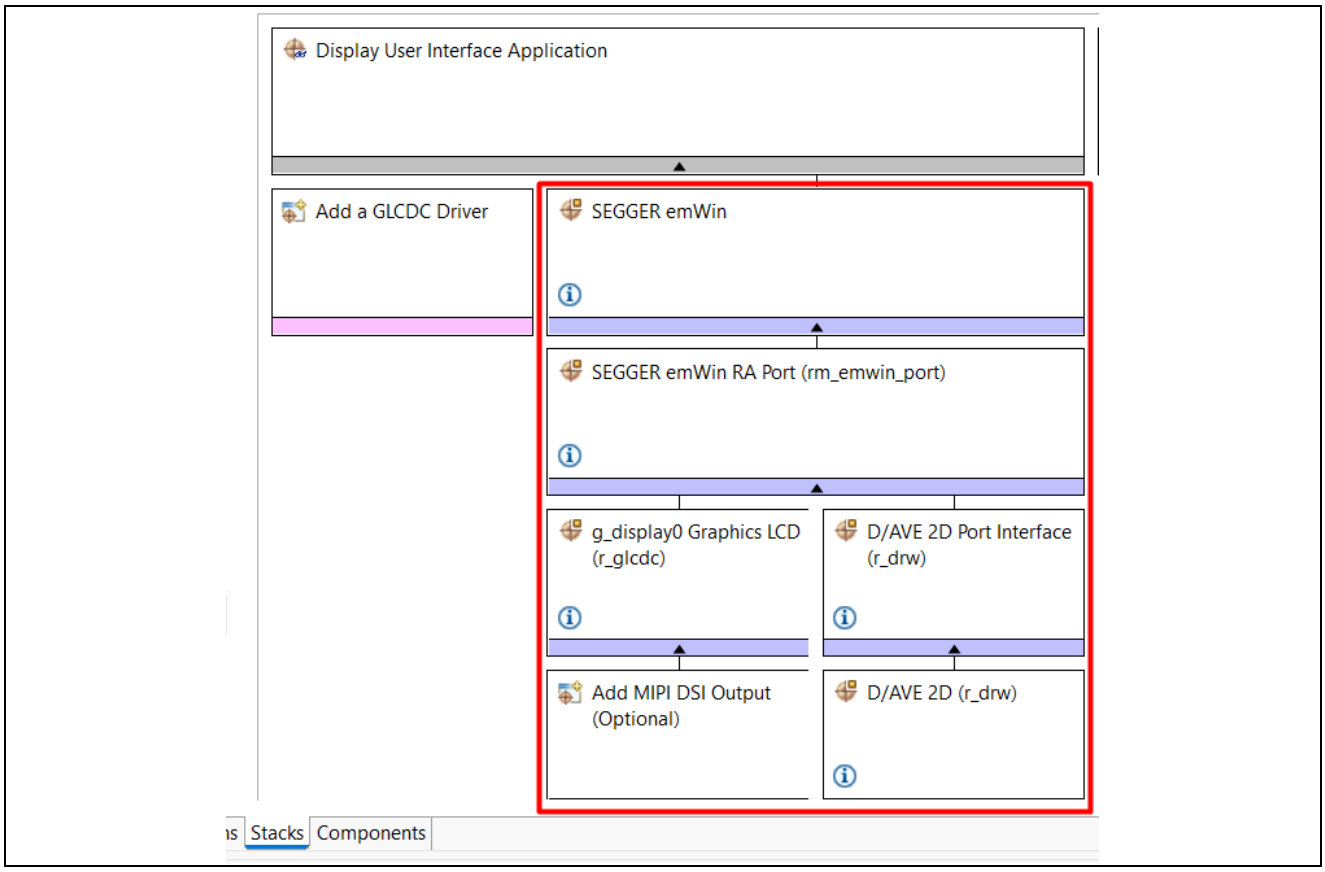


Figure 60. Add r_glcdc LCD controller under SEGGER emWin Driver

Configure the LCD clock and pins for the r_glcdc module and enable the heap for the r_drw module. Then generate and build the project. The figure below shows that the LCD controller has been successfully added.

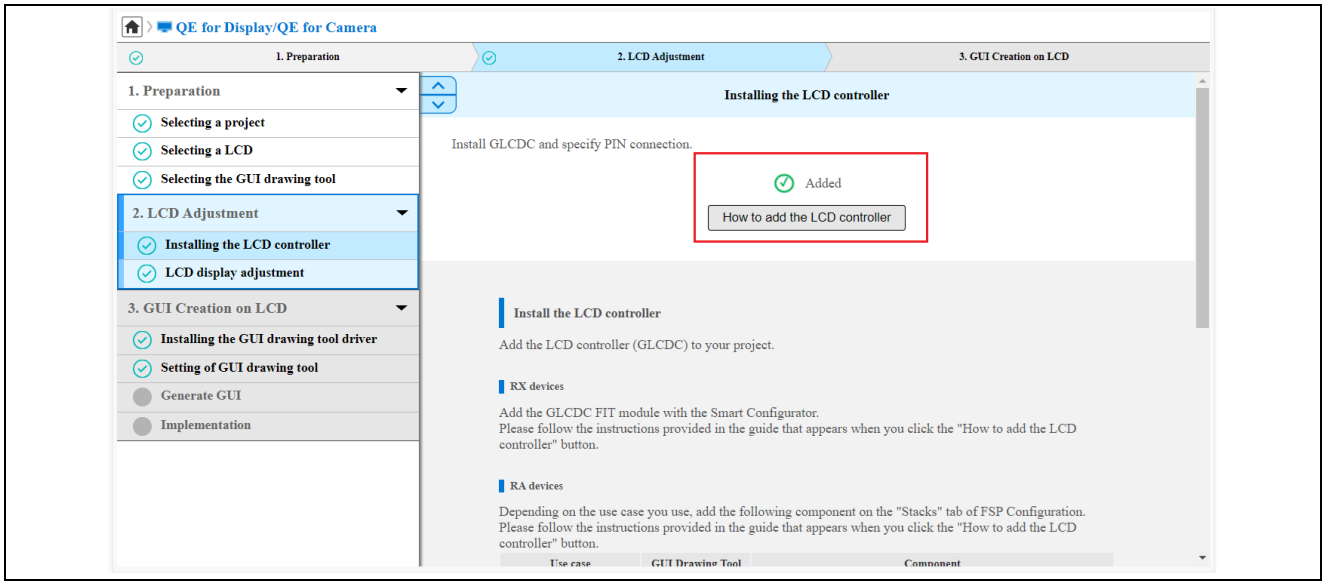


Figure 61. Chose emWin as Embedded Graphics Builder.

8. Adjust the LCD display settings.

Timing adjustment, layer settings and TCON/LCD settings are configured in this tab.

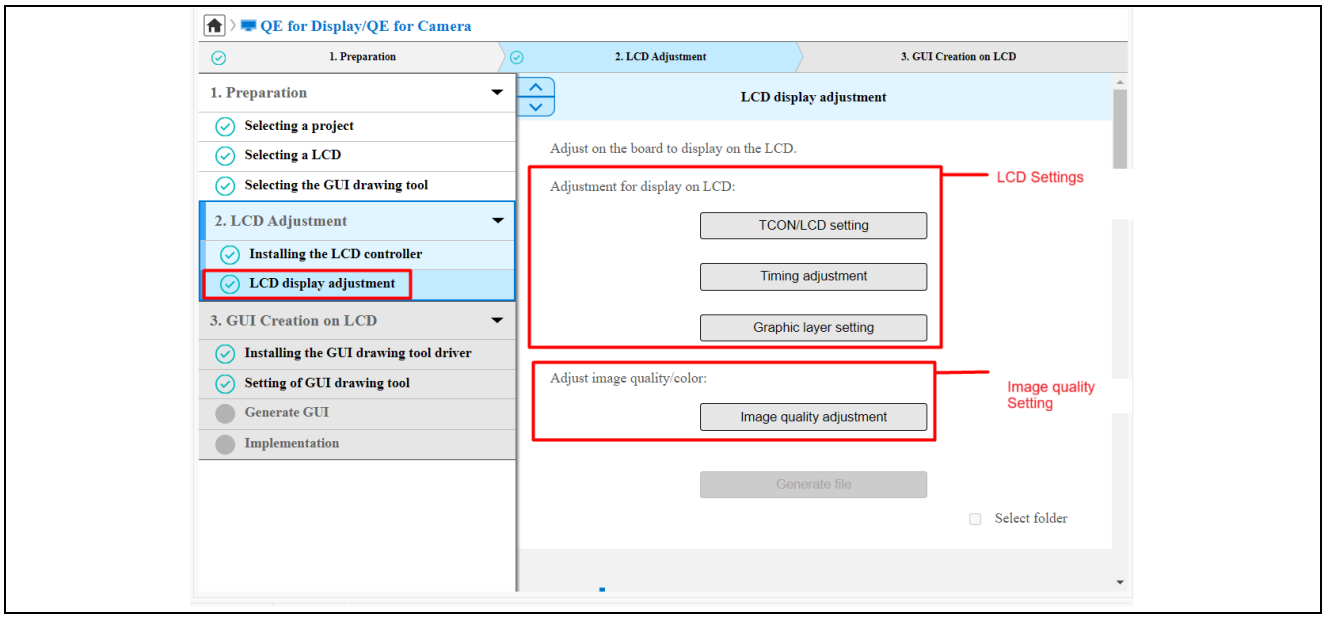


Figure 62. Chose emWin as Embedded Graphics Builder.

Click each setting and navigate to r_glcdc under the SEGGER emWin stack in the **Stacks** tab to view the LCD adjustment view in the **FSP Visualization** tab.

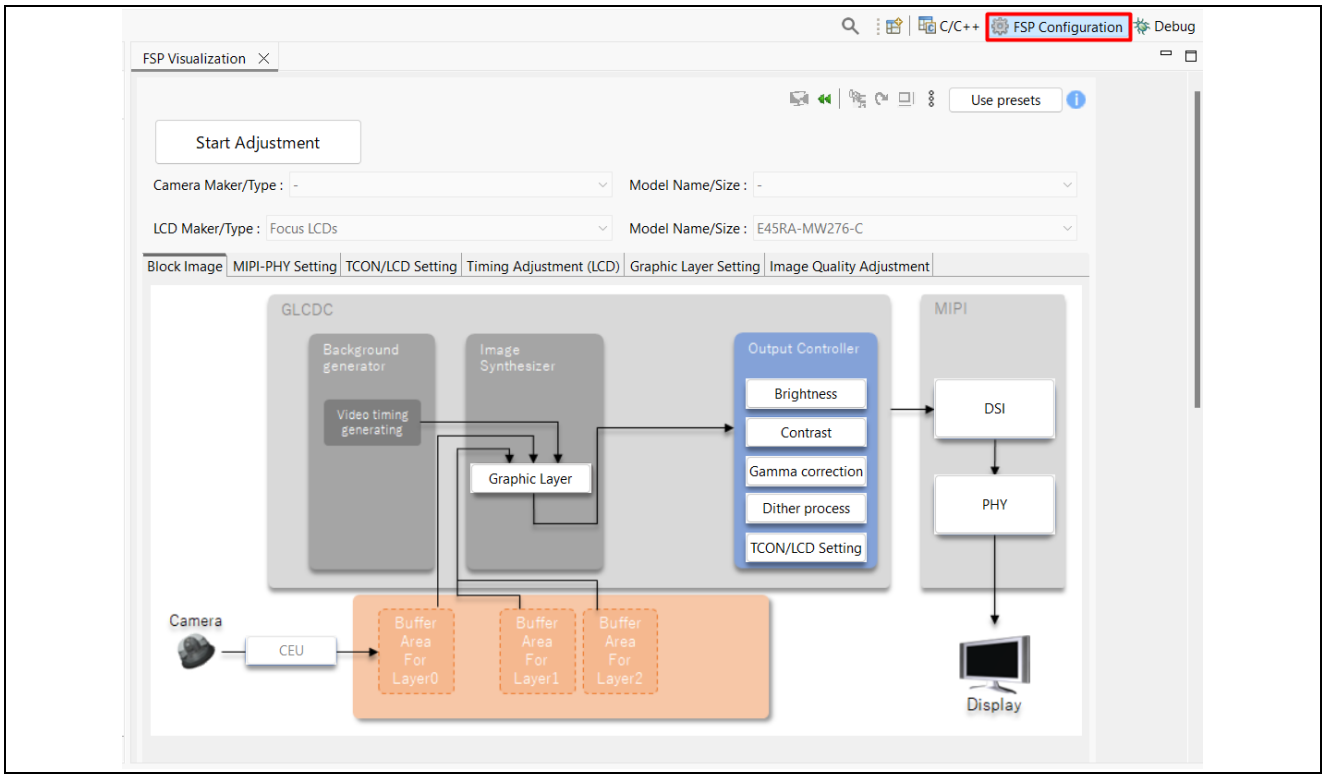


Figure 63. LCD Adjustment view

All configurations related to TCON, Timing, Clock, or Image Quality adjustments made in this adjustment view are applied to the r_glcdc property settings.

One of the advantages of the QE tool is that it allows users to select the panel clock frequency and provides automatic timing calculations to match the desired display refresh rate.

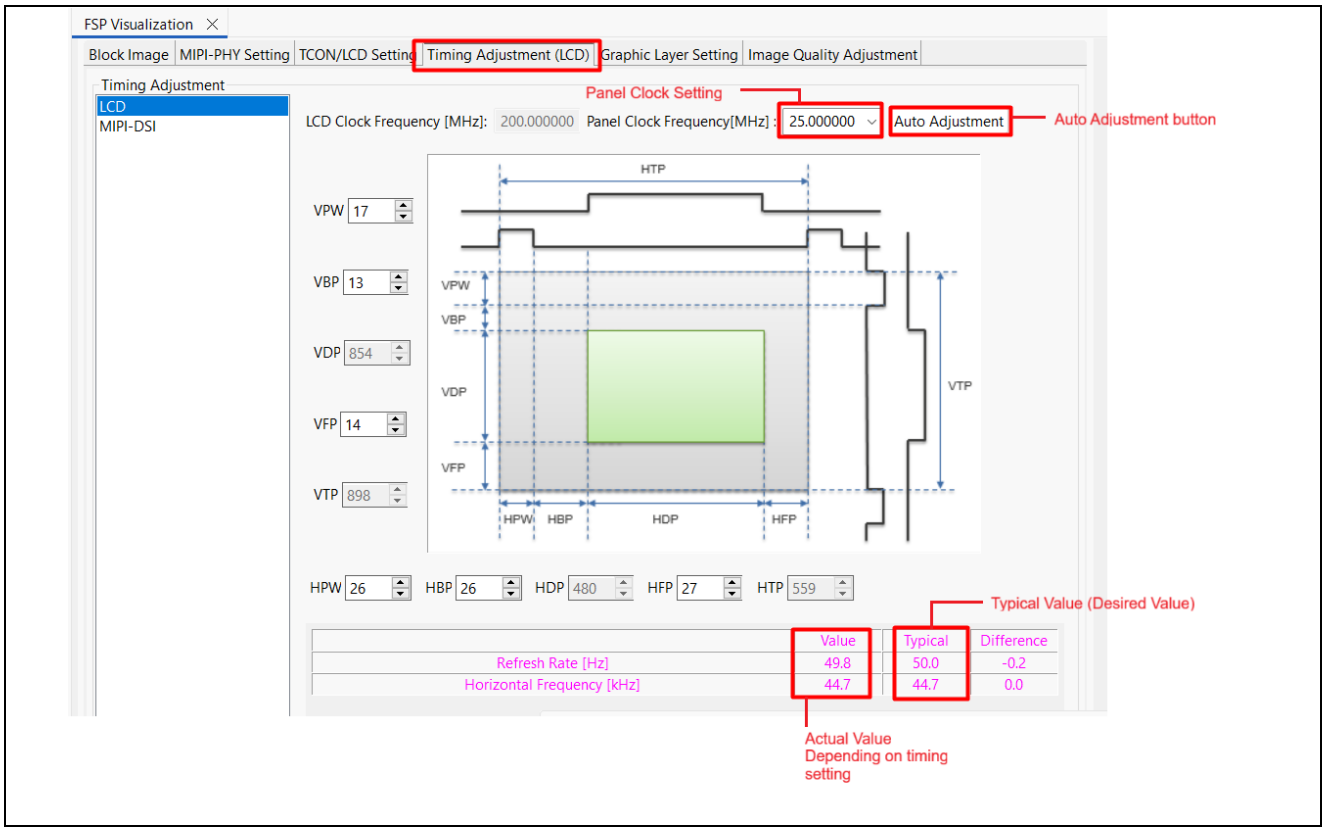


Figure 64. Timing Adjustment Setting

9. Install the GUI drawing tools.

The QE tool also allows users to download the graphic drawing tool directly, or alternatively, specify the installation directory if the tool has already been installed.

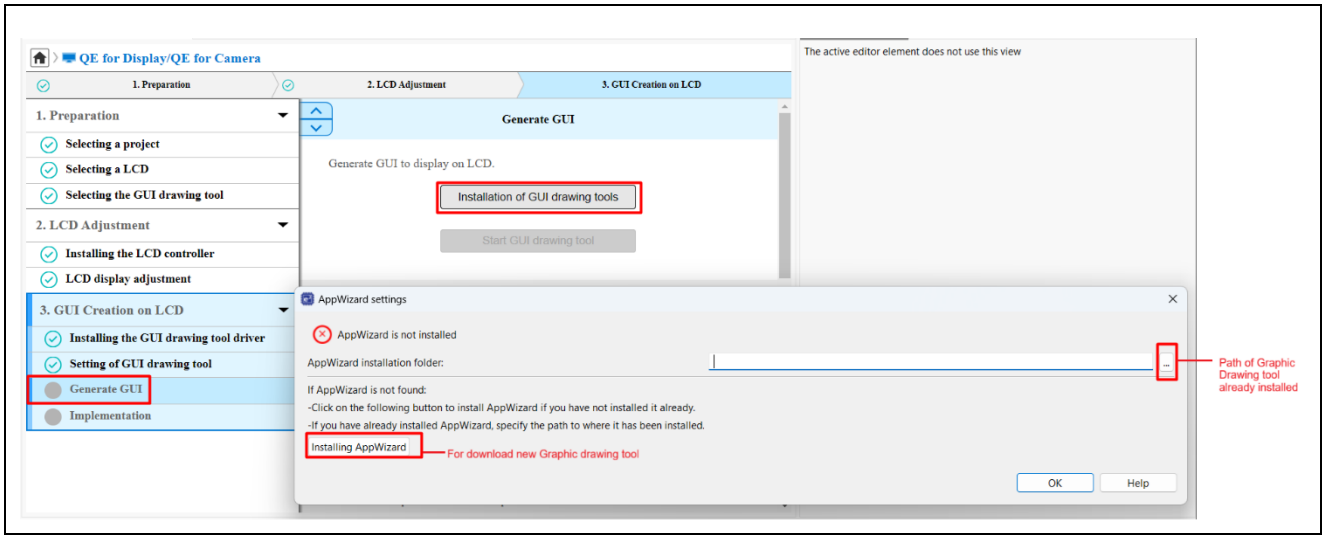


Figure 65. Install Graphic Drawing Tool

10. Launch the drawing tool to develop the GUI.

After successfully downloading Graphic Drawing tool. Open and start building the application GUI.

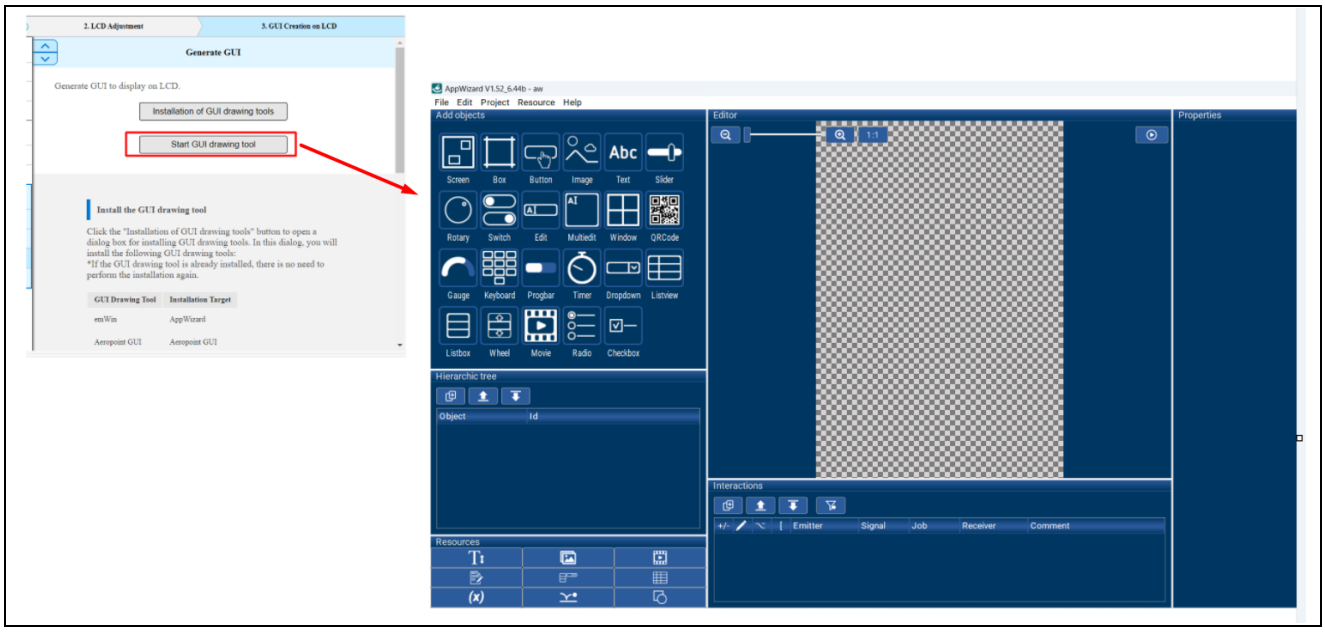


Figure 66. Start Graphic Drawing Tool

After Start the AppWizard Graphic Drawing Tool. The “aw” folder auto generated in application project with all necessary setting for build.

11. Implement the application.

After building the GUI for the graphics application, the sample code provided in the “Implementation” section can be called directly from the application thread, or you can invoke MainTask() directly from the application thread.

7. References

- EK-RA8D2 Quick Start Guide, document No. R20QS0077
- RA8D2 Group User's Manual: Hardware, document No. R01UH1065
- Developing with RA8 Dual Core MCU, document No. R01AN7881
- High Performance with RA8 MCU using Arm® Cortex®- M85 core with Helium™ No. R01AN7127
- MIPI Graphics Expansion Board (E45RA-MW276-C) Datasheet: E45RA-MW276-C - Focus LCDs
- ILI9806E Datasheet: Microsoft Word - ILI9806E-IDT_095.doc
- Capacitive Touch Panel GT911 Controller Datasheet: GOODIX GT911 Datasheet

8. Website and Support

Visit the following 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	Nov.19.25	-	Initial version

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, reWfer 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

Trademarks

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

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/.