Renesas RA Family

Getting Started with GUIX Thermostat Application for EK-RA8D2

Introduction

This application, which is a Thermostat application, provides a reference for developing complex dual-core, multi-threaded applications with a touch screen graphical Human Machine Interface (HMI) by using Renesas FSP and Azure RTOS GUIX. It describes steps to create a basic GUIX for FSP, integrates touch driver, handles multiple hardware accesses across cores, manages system updates, and performs event handling.

This application is developed using the Renesas RA Flexible Software Package (FSP), which provides a quick and versatile way to build secure connected Internet of Things (IoT) devices using the Renesas RA family of Arm microcontrollers (MCUs). RA FSP provides production-ready peripheral drivers to take advantage of the RA FSP ecosystem along with Azure RTOS GUIX library and Azure RTOS. In addition, FSP also provides Ethernet, USB, File System, and other middleware stacks. This powerful suite of tools provides a comprehensive, integrated framework for rapid development of complex embedded applications on dual-core platforms.

This application note assumes that you are familiar with the concepts associated with writing multi-threaded applications on a dual-core Real-Time Operating System (RTOS) environment, such as Azure RTOS. This application note makes use of RTOS features such as threads and semaphores and demonstrates how tasks and events can be efficiently distributed across the two cores. Prior experience in using Azure RTOS would be helpful for easy understanding of the provided application project. For more detailed information on Azure RTOS features, refer to the Azure RTOS User Manual.

The Graphics application is developed using the Renesas e2 studio Integrated Solution Development Environment (IDE). e2 studio is integrated with the FSP platform installer, which can be downloaded from the Renesas website. The intuitive configurators and code generators in e2 studio and FSP help application developers create such complex dual-core, multi-threaded graphics applications very quickly.

Required Resources

Development tools and software

- e^2 studio IDE 2025-10.
- Renesas Flexible Software Package (FSP) v6.2.0.
- Azure RTOS GUIX Studio V6.4.0.0.

Hardware

- Renesas EK-RA8D2 kit (RA8D2 MCU Device Group): www.renesas.com/ek-RA8D2

Contents

1. Installing Tools.....4
1.1 Overview.....4
1.2 Procedural Steps.....4

2. Application Overview.....6
2.1 RA8D2 Device Overview.....6
2.1.1 Key Graphics Features of the RA8D2.....6
2.1.2 RA8D2 Evaluation Kit.....7
2.2 Graphics Overview.....7
2.3 Application Architecture.....8

3. GUIX Integration and Display Management.....10
3.1 Overview.....10
3.2 Configuration Steps.....10
3.3 GUIX Studio Project Integration.....12
3.4 Code Highlights.....20
3.5 Enabling SDRAM Support for Framebuffers.....23

4. GUIX Asset Storage in External OSPI Flash.....24
4.1 Overview.....24
4.2 Configuration Steps.....24
4.3 Code Highlights.....28

5. PWM Backlight Control.....28
5.1 Overview.....28
5.2 Configuration Steps.....28
5.3 Code Highlights.....30

6. Dual-Core Touch Handling.....32
6.1 Overview.....32
6.2 CPU0 – GUI Event Handling.....32
6.2.1 Key Configuration.....32
6.2.2 Code Highlights.....34
6.3 CPU1 – Touch Data Acquisition.....36
6.3.1 Key Configuration.....36
6.3.2 Code Highlights.....41

7. Dual-Core RTC/ADC Processing.....42
7.1 Overview.....42
7.2 CPU0 Processing.....42
7.2.1 Key Configuration.....42
7.2.2 Code Highlights.....43

7.3 CPU1 Processing ... 45
7.3.1 Key Configuration ... 45
7.3.2 Code Highlights ... 48

8. Running the Thermostat Application ... 49
8.1 Hardware Setup ... 49
8.2 Import and Building the Project ... 50
8.3 Downloading and Execute to the EK-RA8D2 Kit ... 51

9. Graphics Tradeoffs on the RA8D2 ... 52
9.1 MIPI DSI vs Parallel RGB ... 52
9.1.1 Data Rate and Bandwidth: ... 53
9.1.2 Cable Complexity and Length: ... 54
9.1.3 Power Consumption: ... 54
9.1.4 System Integration: ... 54
9.1.5 Cost: ... 54
9.2 Graphics Configuration Tradeoffs ... 54
9.2.1 Display Resolution ... 54
9.2.2 Color Format ... 54
9.2.3 Framerate ... 55
9.2.4 Bus Width ... 56
9.2.5 Internal SRAM ... 56
9.3 RA8D2 Memory Options ... 57
9.3.1 EK-RA8D2 Memory Devices ... 57
9.4 Thermostat Application Best Case Design ... 58

10. References ... 59

11. Website and Support ... 60

Revision History ... 61

1. Installing Tools

1.1 Overview

In this section you will copy the application note (AN) materials to your PC and install e² studio v2025-10/FSP v6.2.0 and Azure RTOS GUIX Studio v6.4.0.0.

1.2 Procedural Steps

1. If you already have e² studio v2025-10 with FSP v6.2.0, you can skip this step. Otherwise, you can download RA Flexible Software Package (FSP).
2. You can get Azure RTOS GUIX Studio v6.4.0.0 or greater. If it goes well, you will see the window in the next step on the web browser.

Note: It needs Microsoft Store to work on your PC to install Azure RTOS GUIX Studio. If Microsoft Store is not available or disabled, you can alternatively download and install GUIX Studio manually from the official Eclipse ThreadX GitHub repository.

3. Click **Get** to start installing Azure RTOS GUIX Studio.

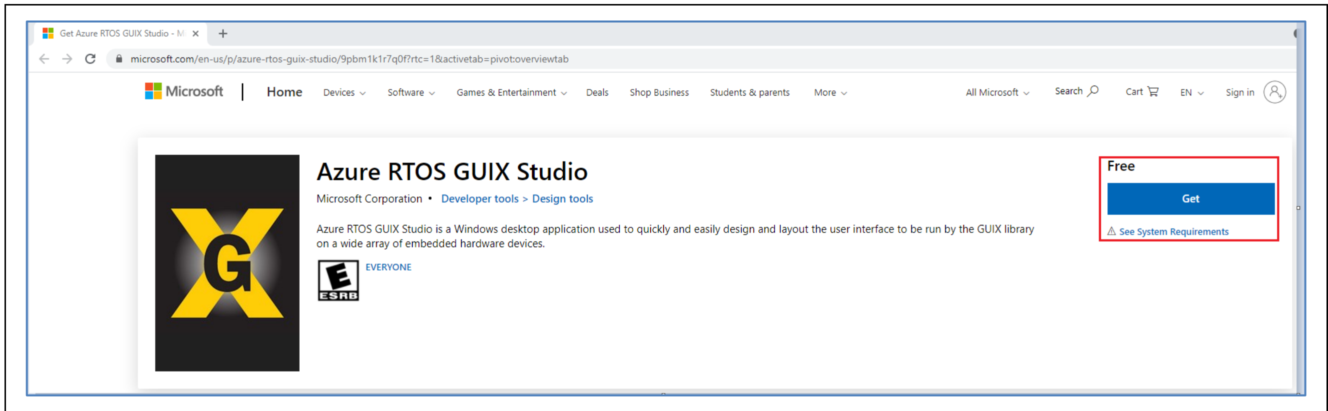


Figure 1. Clicking Get to Start Installing Azure RTOS GUIX

4. Click **Open Microsoft Store** to continue installing Azure RTOS GUIX Studio.

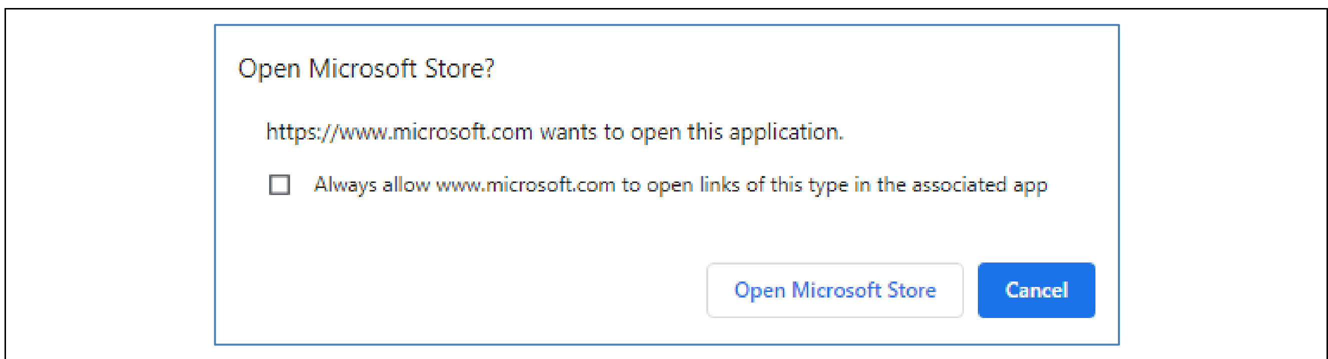


Figure 2. Clicking Open Microsoft Store

5. Click **Install** to continue. A window shows up to ask for a Microsoft account, which is seen in the next step.

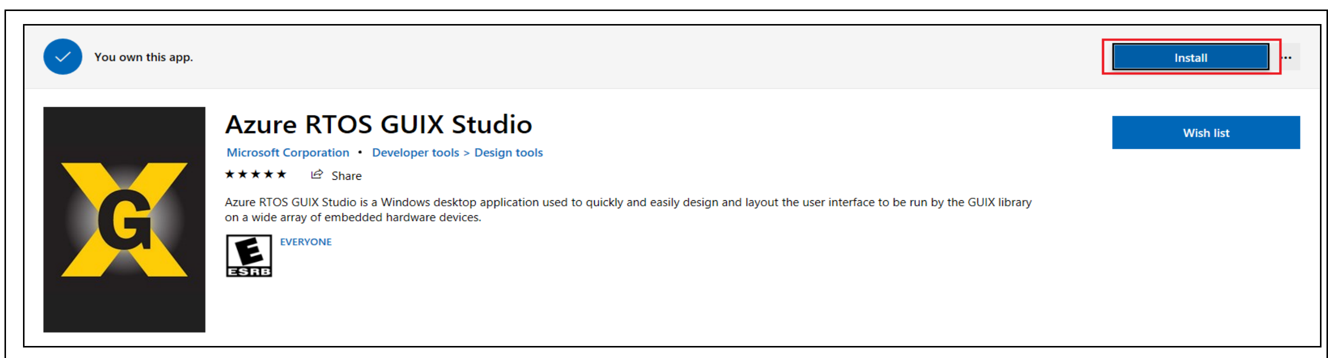


Figure 3. Installing Azure RTOS GUIX

6. Ignore it by clicking "X" on the top-right to close this pop-up window and continue Azure RTOS GUIX Studio installation.

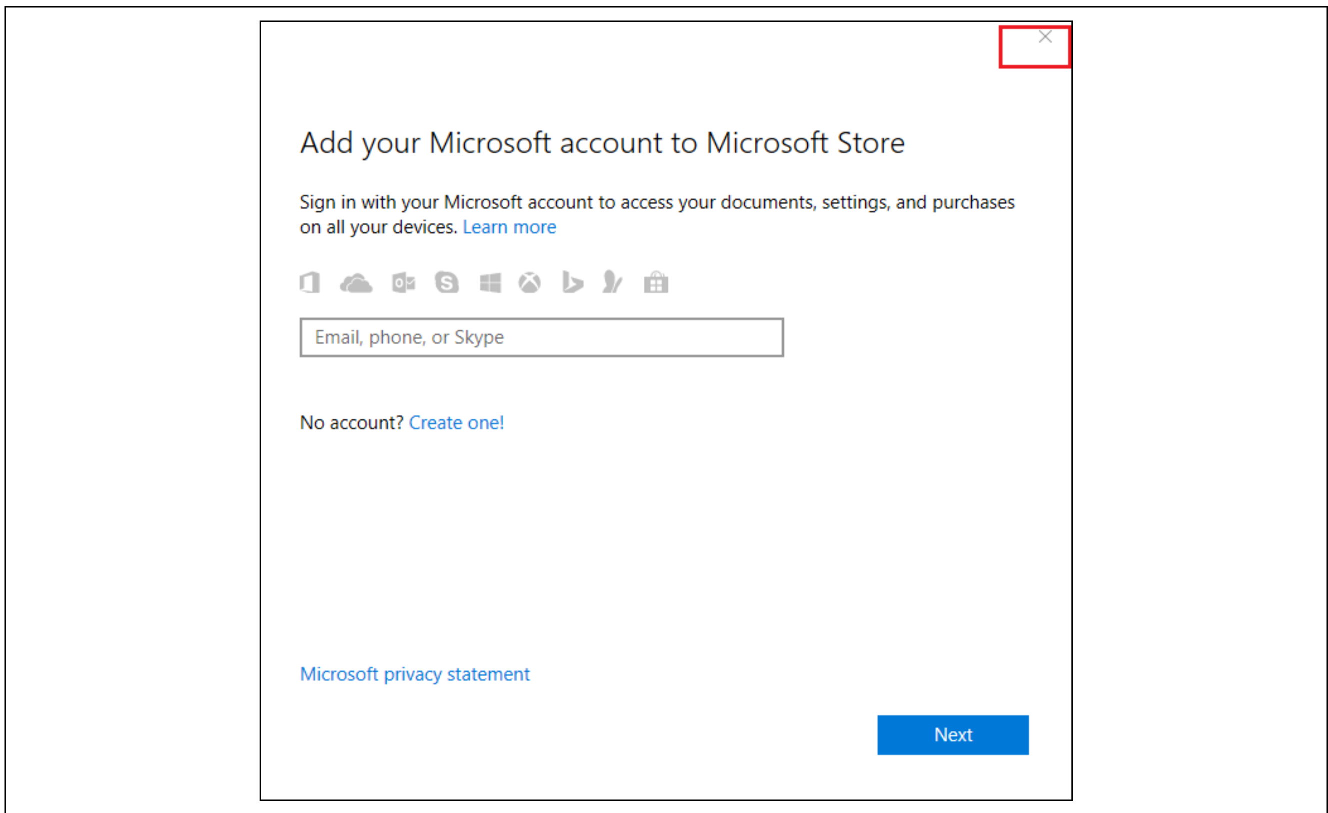


Figure 4. Closing Pop-up Window to Continue Installing Azure RTOS GUIX

7. Downloading and installation of Azure RTOS GUIX Studio starts.

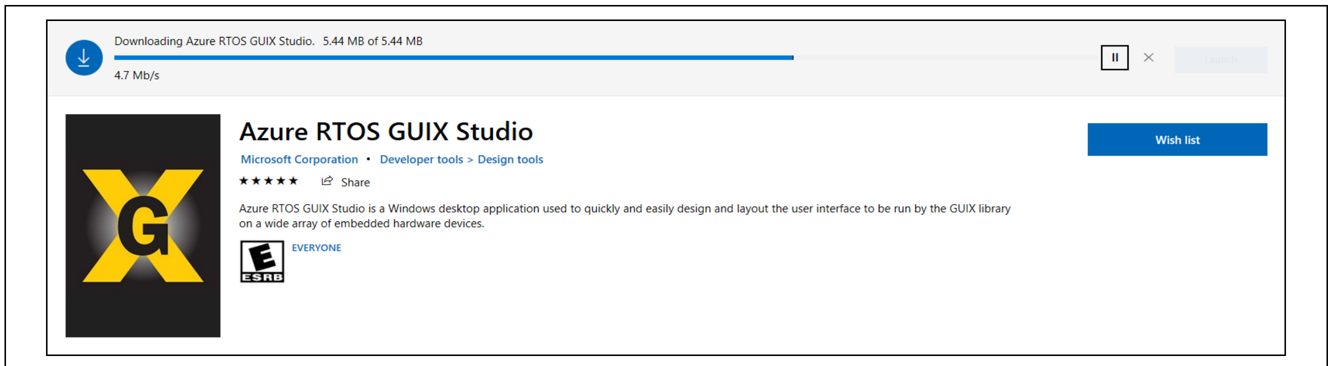


Figure 5. Starting of Downloading and Installation

8. Click **Launch** to launch Azure RTOS GUIX Studio.

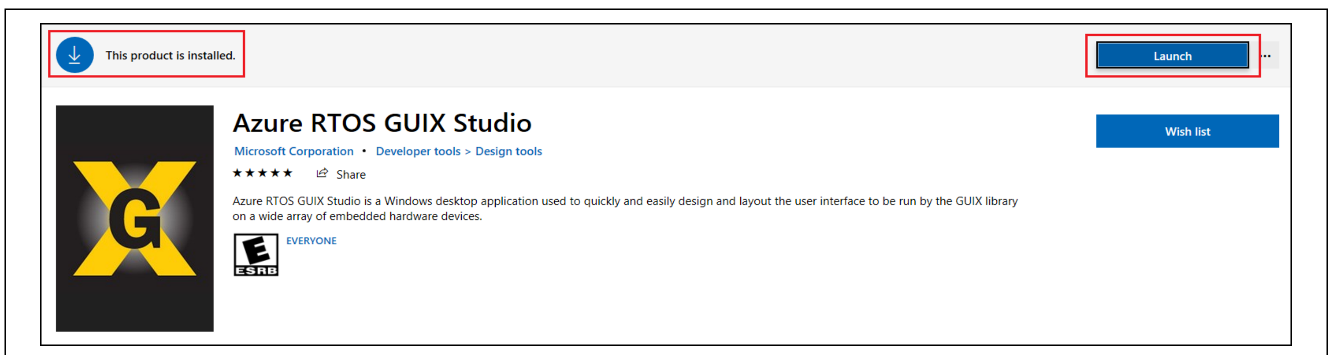


Figure 6. Launching to Start Azure RTOS GUIX Studio

9. **Azure RTOS GUIX Studio** launched.

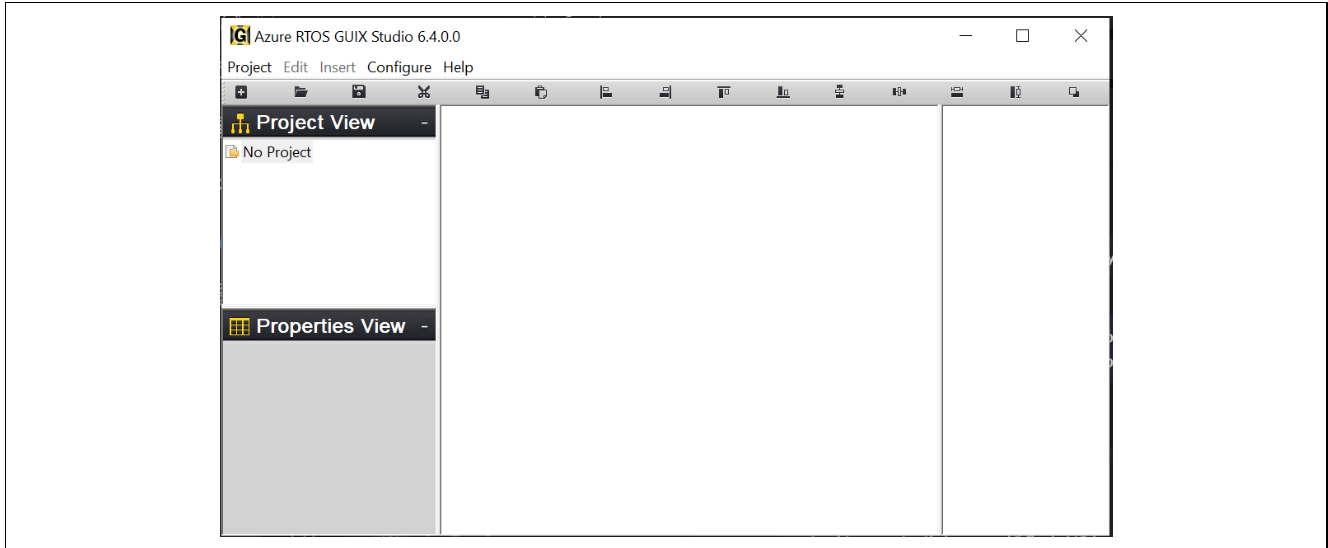


Figure 7. Azure RTOS GUIX Studio Launched

10. Close Azure RTOS GUIX Studio, for now, you will open it again later.

2. Application Overview

2.1 RA8D2 Device Overview

The MCUs in the RA8D2 device group deliver high performance, scalability, and advanced security for modern embedded and IoT applications. Featuring a 1 GHz Arm® Cortex®-M85 and a 250 MHz Cortex®-M33 core, they integrate up to 1 MB MRAM, 2 MB ECC SRAM, and rich peripherals including Ethernet, USB HS, CAN FD, SDHI, I3C, Octal SPI, MIPI DSI/CSI, and a 2D graphics engine. Built-in Security IP with Arm® TrustZone® provides cryptography acceleration, key management, and tamper protection. These devices also maintain a unified peripheral architecture for scalable, platform-based product development.

RA8D2
1GHz Arm® Cortex®-M85 Core,
+ 250MHz Arm® Cortex®-M33 Core
FPU | ARM MPU | NVIC | JTAG | SWD | ETM |Boundary Scan

Memory
Code NVM (MRAM 0.5/1MB, Flash 4/8MB)
Data SRAM w/ ECC (1.6MB)
TCM (256KB for Cortex-M85 + 128KB for Cortex-M33)
I/D-Cache (32KB for Cortex-M85 + 32KB for Cortex-M33)

Analog
16-bit ADC (2units, 23ch, 3ch-S/H x2)
12-bit DAC (2ch)
High-speed Comparator (4ch)
Temperature Sensor

Timers
32-bit GPTE (High Resolution) (4ch)
32-bit GPTE (10ch)
32-bit ULPT (2ch)
16-bit AGT (2ch)
WDT (2ch)
RTC

HMI
Graphics LCDC w/ RGB i/f
2D DRW
MIPI DSI | MIPI CSI-2
CEU 16bit Camera Interface

Communication
Gigabit Ethernet MAC w/ TSN (x2) + 2 port switch
CAN-FD (x2)
USB2.0 FS (x1), USBHS (x1)
SDHI/MMC (x2)
I3C (x1), I2C (x3)
SCI (x10)
SPI (x2)
OSPI (x2, XIP&DOTF)
SSI x2 & PDM 3ch x1
32-bit External Memory Bus

System
DMA (8ch x2)
DTC (x2)
Clock Generation
On-chip Oscillator
DC-DC Converter
Low Power Modes
ELC
Interrupt Controller
VBAT

Safety
Memory Protection Unit
SRAM Parity Check
ECC in SRAM
POE
Clock Frequency Accuracy Measurement
CRC Calculator
IWDT
Data Operation Circuit
MRAM Area Protection
ADC Self Test
Permanent Lock Function
Programmable Voltage Detector

Security
AES (128/192/256), CHACHA20
RSA 4K, ECC
TRNG
SHA-2 (224/256/384/512), SHA-3
Secure Debug
First Stage Boot Loader
OTP (Immutable storage)
TrustZone | EFP support
CMAC/HMAC/GMAC
DPA/SPA Side Ch. Protection

Package
BGA 224/ 289/ 303

Figure 8. Example of Key Features in RA8D2 MCU

2.1.1 Key Graphics Features of the RA8D2

The RA8D2 MCUs integrates a comprehensive graphics subsystem optimized for advanced HMI and Vision AI applications. Key components include a high-resolution TFT-LCD controller (GLCDC) with parallel RGB and

MIPI-DSI interfaces, a 2D drawing engine (DRW), a 16-bit camera interface (CEU), and multiple external memory interfaces for flexible display configurations.

The GLCDC supports display resolutions up to 1280 × 800 (WXGA) and can drive both RGB and MIPI-DSI panels. The 2D drawing engine accelerates rendering of graphics primitives such as lines and polygons, and supports operations like alpha blending, rotation, scaling, and color conversion to reduce CPU workload. The camera interface enables image capture and format conversion for sensors up to 5 megapixels.

On-chip 2 MB SRAM provides frame buffer storage for WVGA (800 × 480) displays, while external SDRAM enables higher-resolution graphics rendering. The xSPI-compliant OSPI interface with Execute-In-Place (XIP) and Decryption-On-The-Fly (DOTF) features ensure fast and secure access to external graphics assets.

Software integration is supported through the Renesas Flexible Software Package (FSP) with Azure RTOS GUIX, providing a robust framework for building high-quality embedded GUIs. GUIX offers advanced widgets, animation, touch input handling, and efficient rendering pipelines. The RA GUIX middleware in FSP ensures optimized performance and seamless operation across RA devices

2.1.2 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 e2 studio IDE.

The kit consists of three boards (and required connections): the EK-RA8D2 board featuring the RA8D2 MCU with on-chip graphics LCD controller, the Parallel Graphics Expansion Board 1 featuring a 1024x600 TFT LCD with a capacitive touch panel overlay, and a camera expansion board OV5640 featuring a 5M pixels CMOS image sensor.

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.

2.2 Graphics Overview

The RA8D2 supports multiple graphics output options, including the parallel RGB interface through the on-chip Graphics LCD Controller (GLCDC) and the high-speed MIPI-DSI interface. Each interface offers distinct advantages in terms of resolution, performance, and system design tradeoffs.

The multifunctional GLCDC supports multiple data formats and panel types, making it well-suited for flexible graphics designs. Key features include:

- GLCDC0 and GLCDC1 bus master function for accessing graphics data
- Superimposition of three planes (single color background plane, graphics 1 plane, and graphics 2 plane)
- Support for various 32- and 16-bit/pixel graphics data and 8-, 4-, and 1-bit LUT data formats
- Digital interface signal output supporting video image sizes up to WXGA (1280 × 800)

For this application, the parallel RGB interface is selected, leveraging the GLCDC to drive a 1024×600 TFT LCD with touch support.

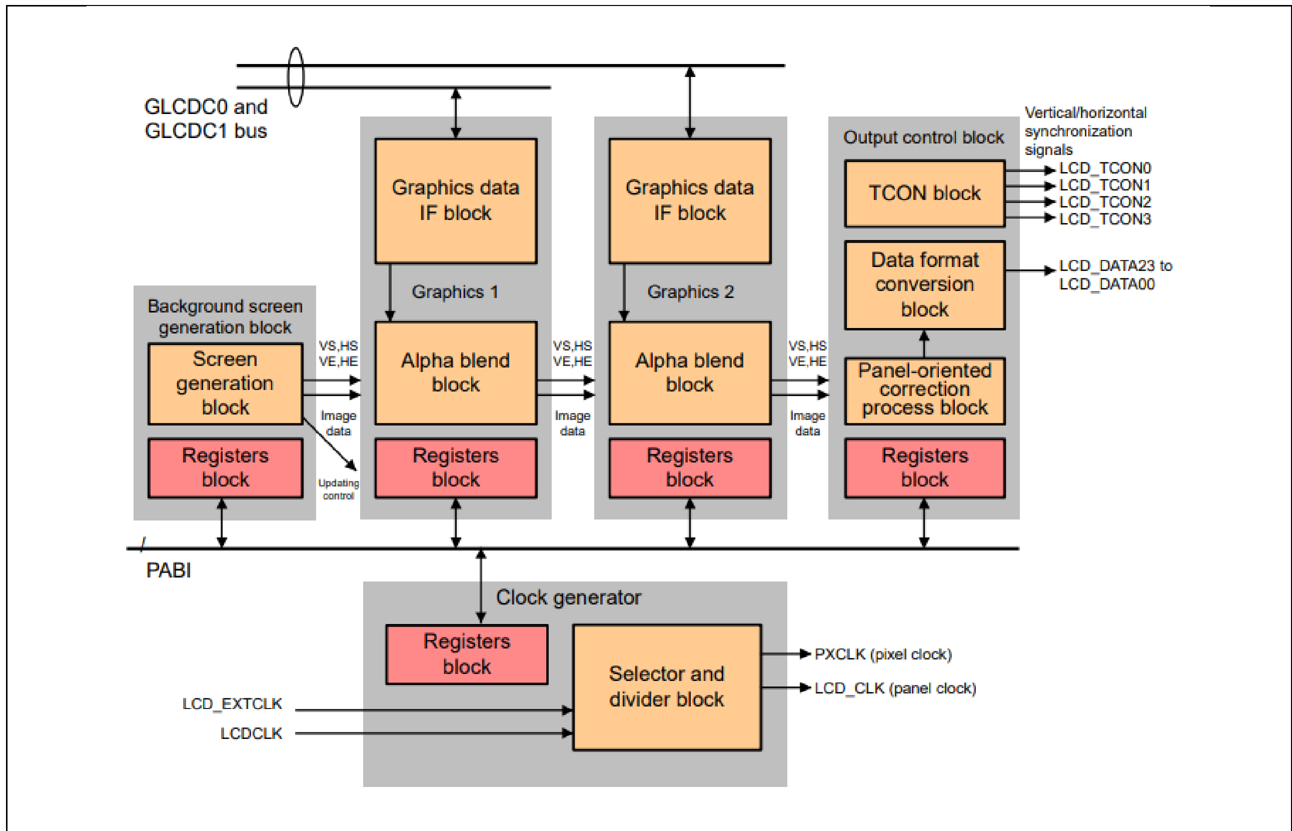


Figure 9. GLCDC block diagram

2.3 Application Architecture

The thermostat application runs on the dual-core RA8D2 platform with Azure RTOS ThreadX and GUIX. CPU0 handles all GUI rendering through the GLCDC over a parallel RGB interface to the TFT panel and also controls display backlight brightness via an FSP GPT (PWM) instance.

Human–machine interaction is processed on CPU1, where touch input from the FT5x06 I²C controller is packaged and sent to CPU0 via inter-core IPC. CPU1 also acquires RTC and temperature (ADC) values, updating shared memory and signaling CPU0 to refresh the display.

The graphics subsystem combines GUIX with DRW (DAVE2D) acceleration, external SDRAM for frame buffering, and OSPI flash for graphics assets, ensuring smooth UI performance while preserving internal memory for real-time tasks.

The following diagram illustrates the high-level data and control flow of the system.

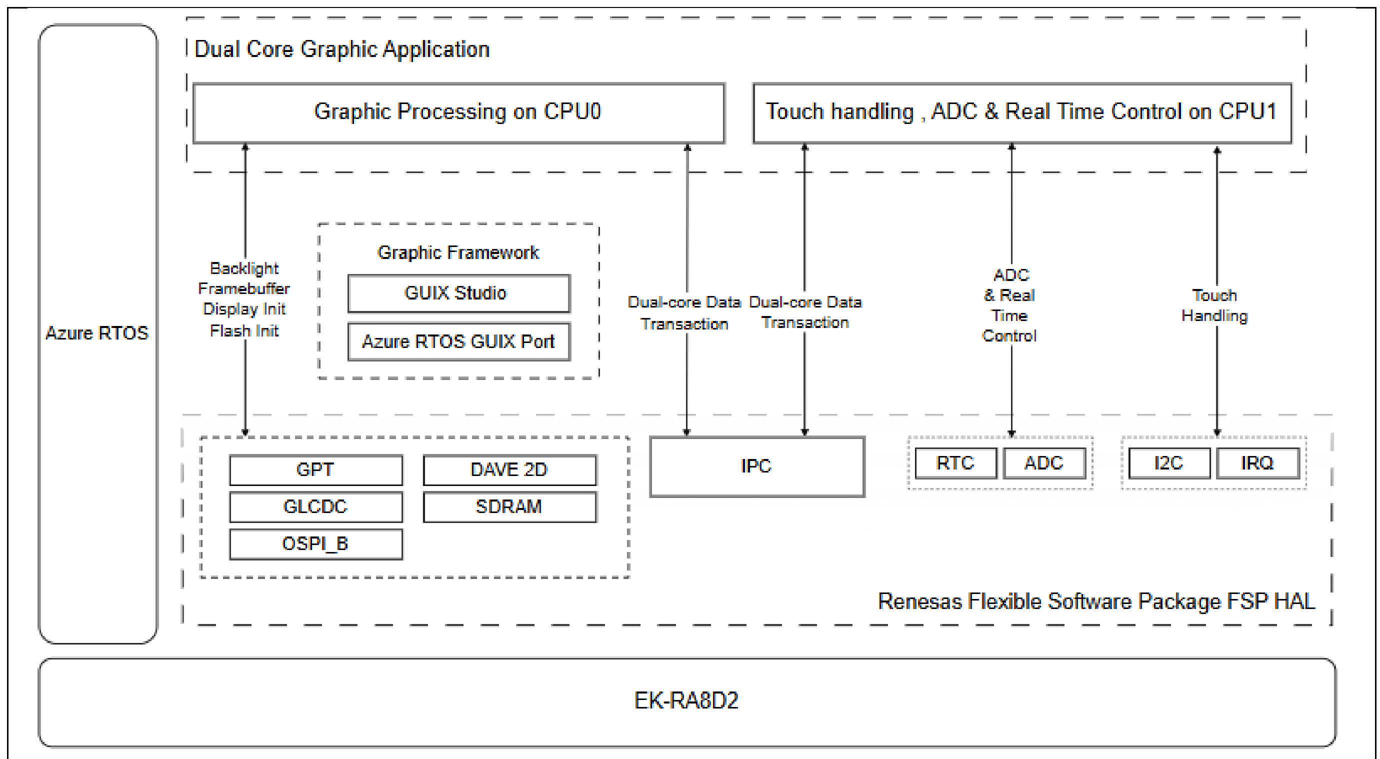


Figure 10. Application Block diagram

In addition to the hardware and communication overview shown above, the application execution flow is illustrated below.

The flowchart shows how CPU0 and CPU1 operate concurrently — CPU0 managing GUI and display updates, and CPU1 handling user input, sensor acquisition, and system logic — while synchronizing through inter-core messaging and shared memory.

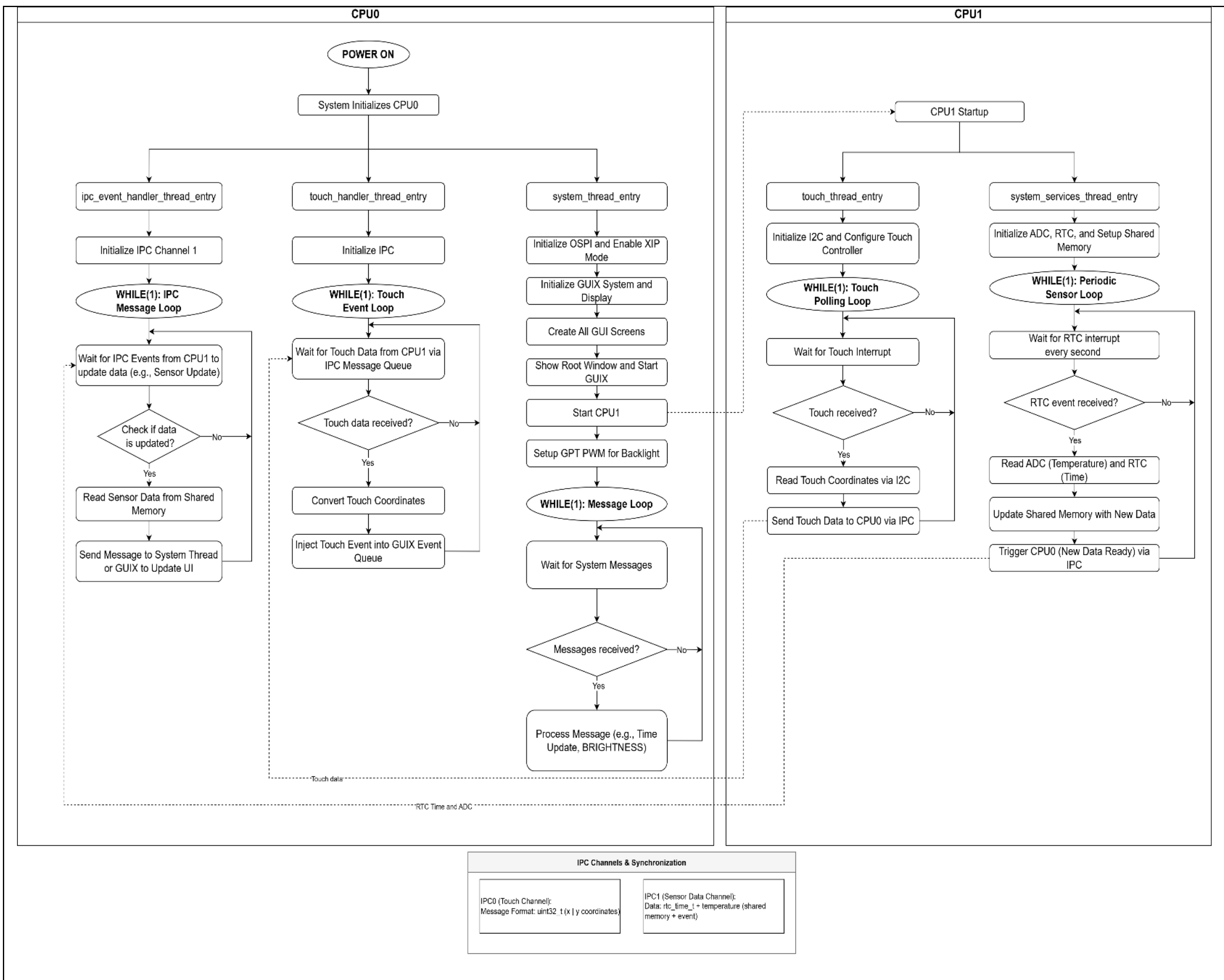


Figure 11. Dual-Core Thermostat Application Operation Flow

3. GUIX Integration and Display Management

3.1 Overview

This section outlines the integration of the Azure RTOS GUIX framework with the RA8D2 graphics subsystem. It introduces the key software and hardware components involved in display management, and provides a summary of how GUIX operates in coordination with the GLCDC, DRW engine, and SDRAM framebuffer.

3.2 Configuration Steps

The GUIX stack provides the interface between Azure RTOS GUIX framework and the RA8D2 display hardware. Configure the rm_guix_port driver to enable GUIX graphics rendering with hardware acceleration support.

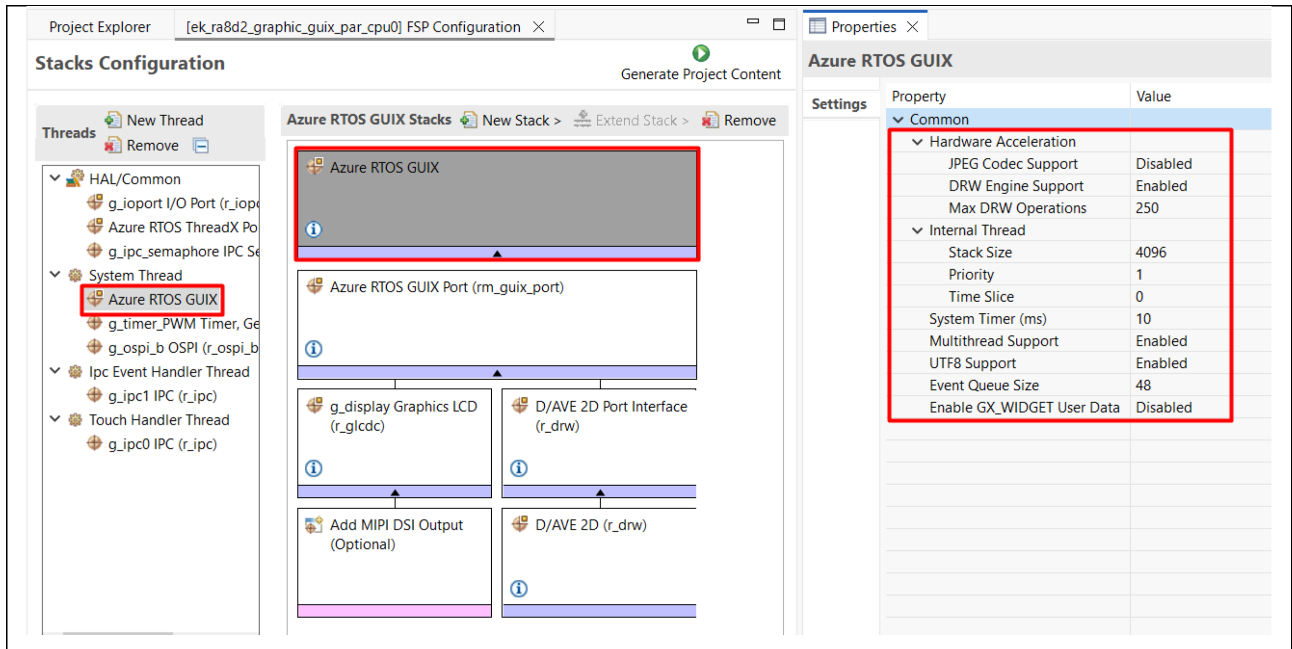


Figure 12. Setting Properties for GUIX Stack

GLCDC configuration requires attention to three key components: Stack configuration, Clock configuration, and Pin configuration.

1. Settings properties for Graphics LCD.

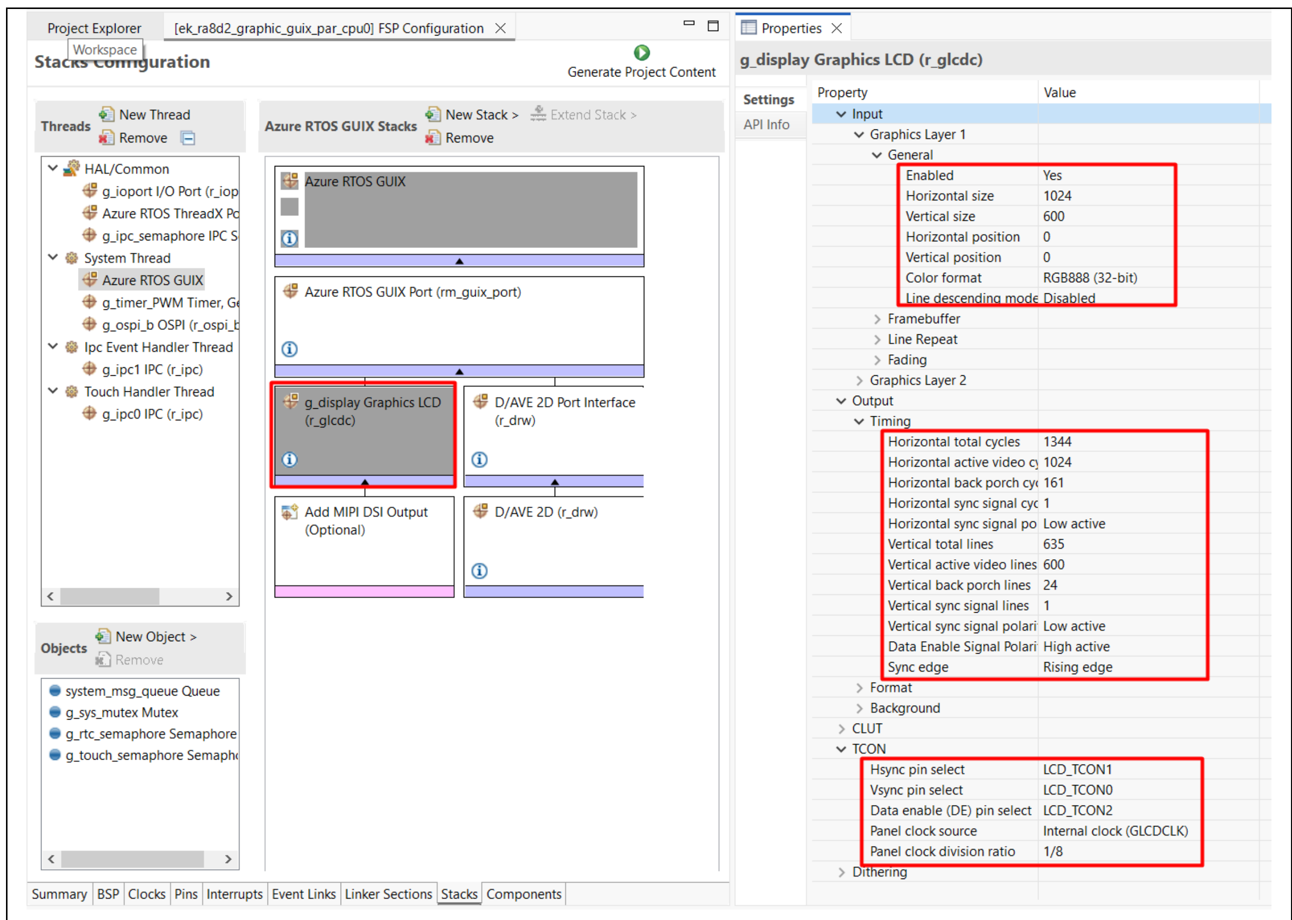


Figure 13. Setting Properties for GLCDC DISPLAY

RENESAS

2. Setting pins configuration for GLCD.

Figure 14. Setting pins configuration for GLCD

3. Setting Clock for GLCD.

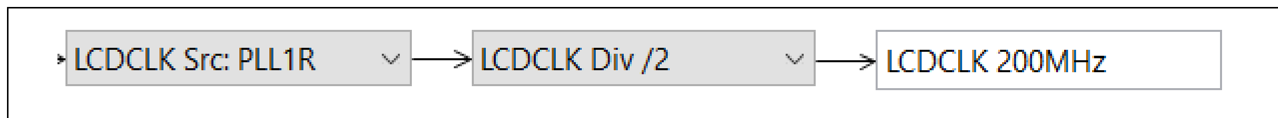


Figure 15. Clock Source Setting

3.3 GUIX Studio Project Integration

Configure GUIX Studio to generate source code and resources for the thermostat application. Follow the steps below to add the GUIX Studio project to the source code.

1. Copy Azure RTOS GUIX Studio project to e2 studio project (Themostat_GUIX_EK_RA8D2) by copying "guix_studio" folder in the application note (AN) folder (FSP_GUIX_Thermostat) and pasting it in the "**ek_ra8d2_graphic_guix_par_cpu0**" project.

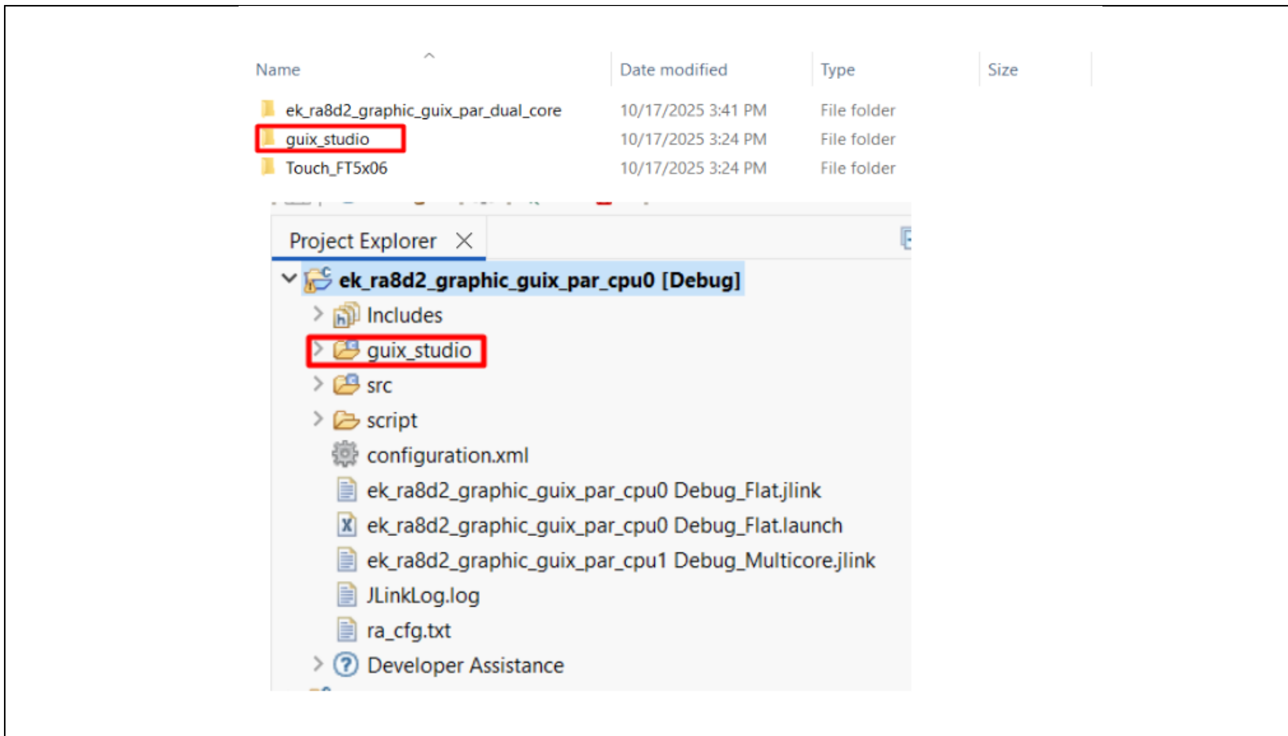
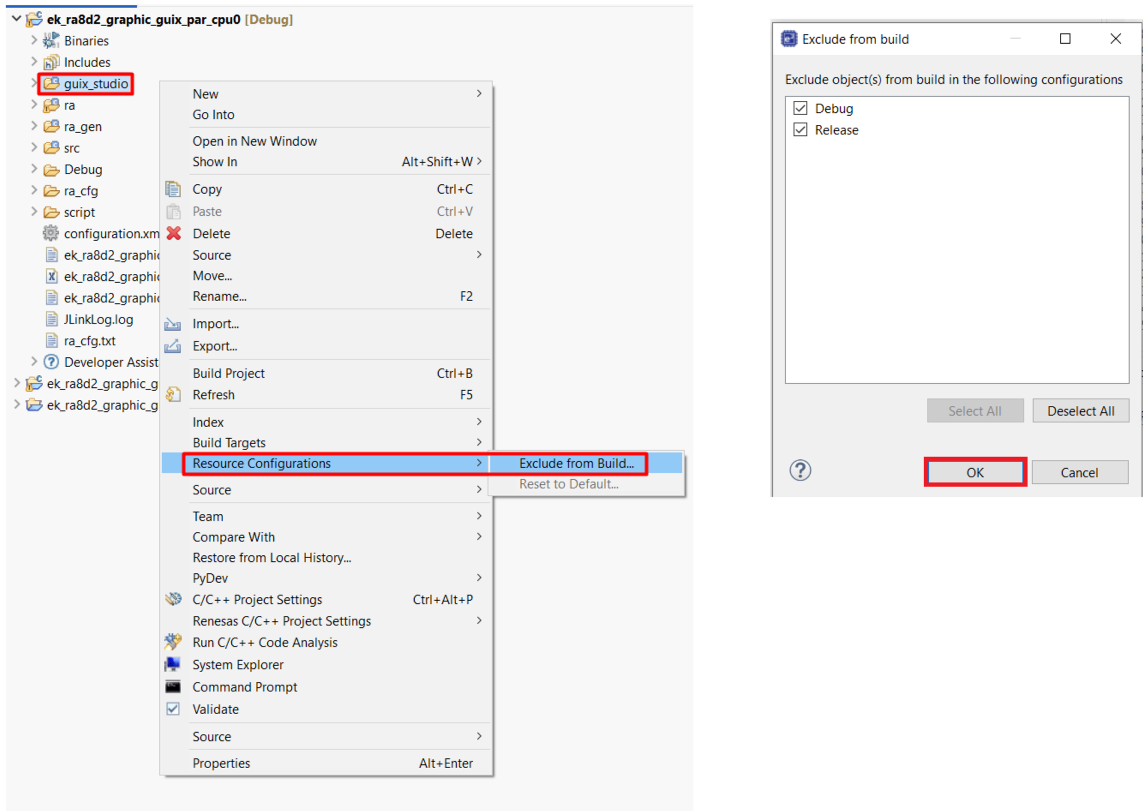


Figure 16. Copying the Azure RTOS GUIX Studio Project to e2 studio

2. GUIX Studio project is now in "**ek_ra8d2_graphic_guix_par_cpu0**" project. In e2 studio, right-click the "guix_studio" folder and exclude it from the build since it contains the Azure GUIX Studio project, which will not be built by FSP.

   The guix_studio folder holds the GUIX thermostat project, the source of the graphics, and the fonts. The graphics and the fonts will be used by the GUIX thermostat project when it is compiled by the GUIX Studio application. The content in this folder will be used in a later step to generate the GUIX .c and .h source files using the GUIX Studio Application. This folder will not be compiled by e2 studio IDE.



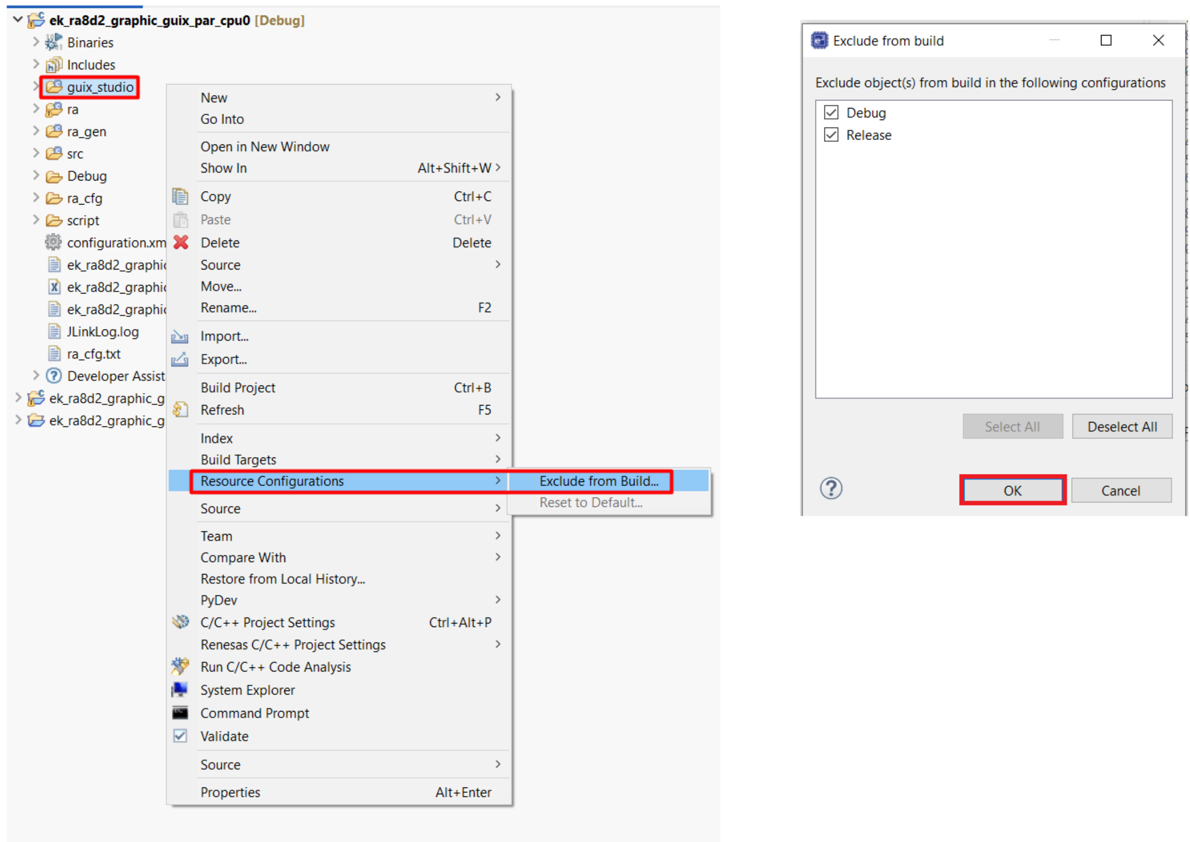


Figure 17. guix_studio folder containing

3. Get to "**ek_ra8d2_graphic_guix_par_cpu0**" project folder by right clicking the e² studio project and select "**System Explorer**" as shown below.

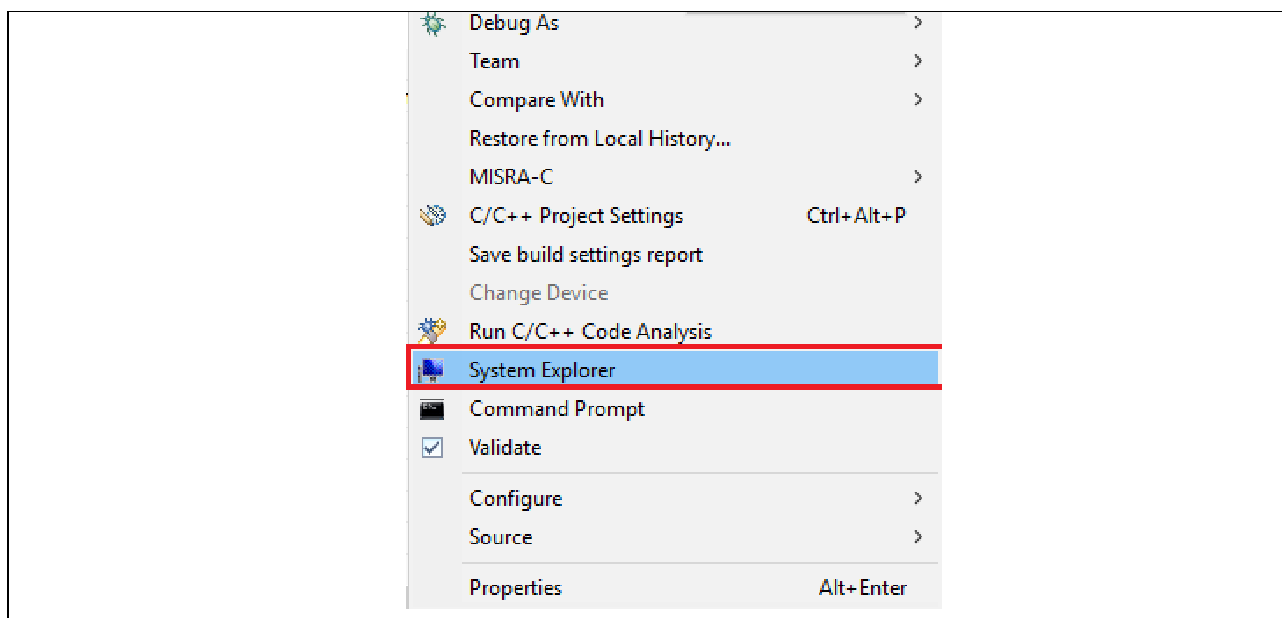


Figure 18. Selecting System Explorer

4. Open thermostat.gpx project file in "**guix_studio > GNU**" sub-folder in your "**ek_ra8d2_graphic_guix_par_core0**" folder. If you have several GUIX Studio versions in your system, make sure you choose the right one, which is v6.4.0.0.

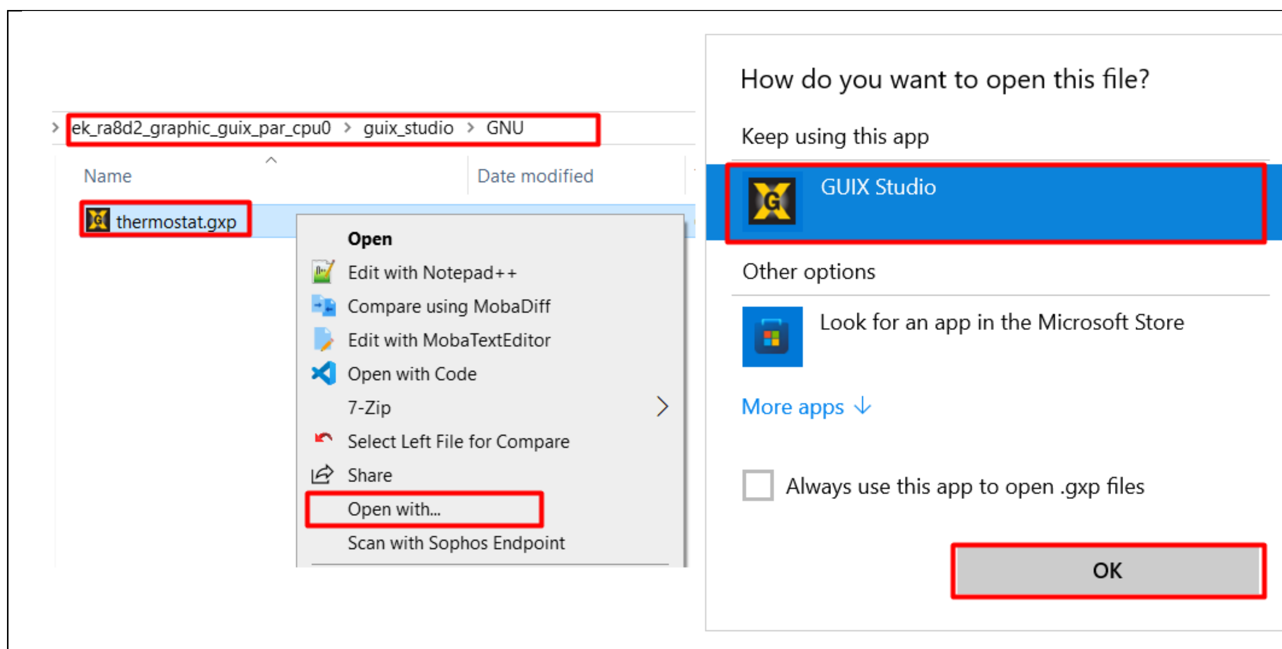


Figure 19. Opening the Project File

5. This GUIX Studio project has a complete design of this Thermostat application. The next several steps describe the process to generate resources, application code and integrate them with an e² studio project.

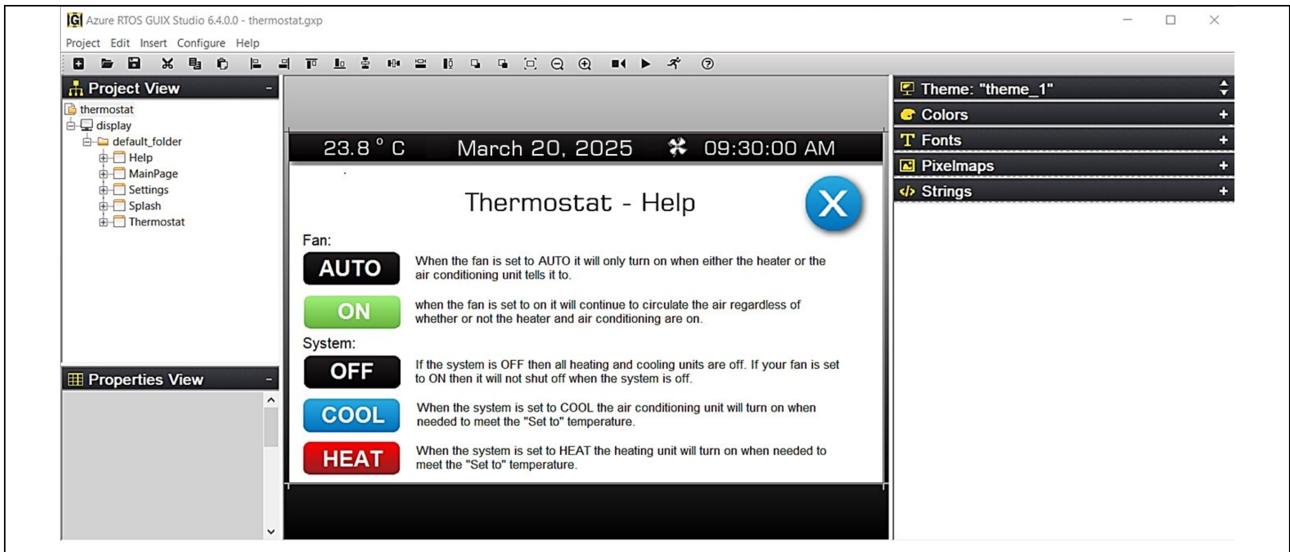


Figure 20. GUIX Studio Thermostat Application View

6. The Azure RTOS GUIX Studio project consists of five screens — Splash, Main Page, Settings, Thermostat, and Help — as shown in the figures below:

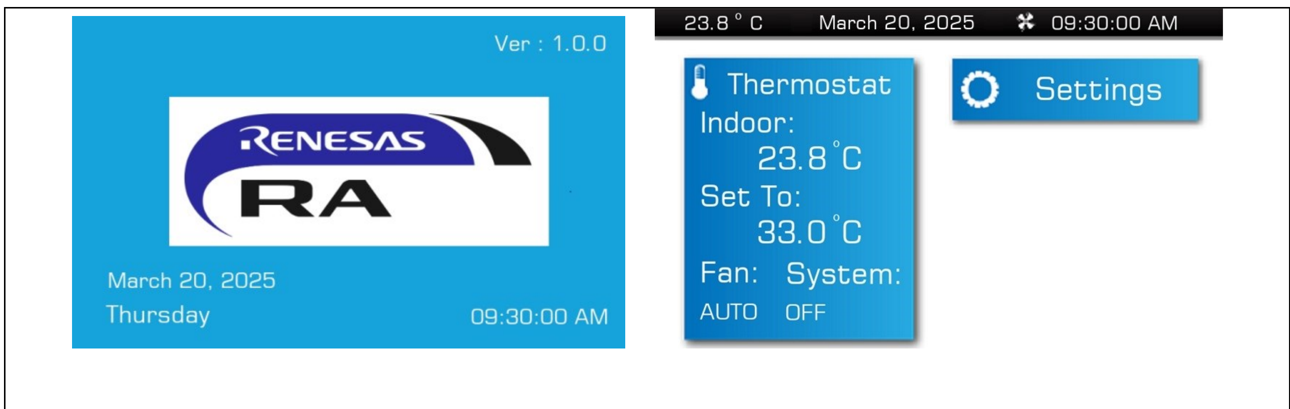


Figure 21. Splash and Main Page Screens

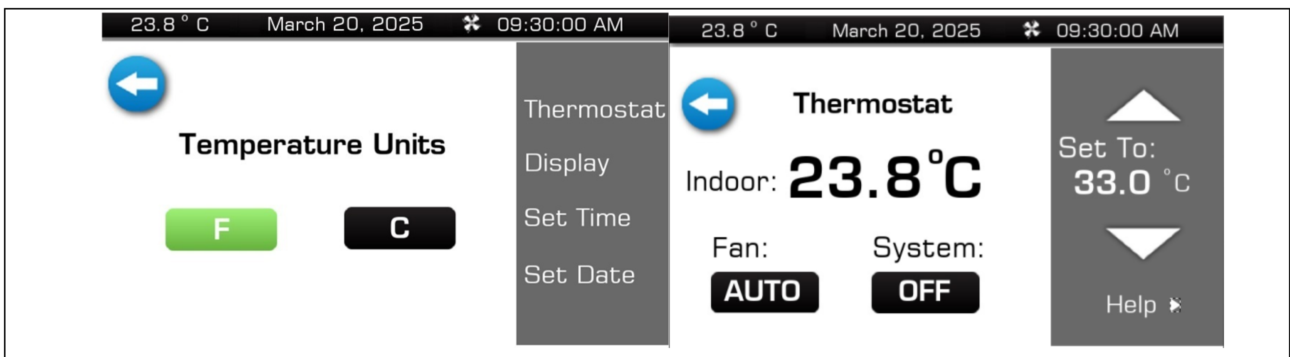


Figure 22. Settings and Thermostat Screens

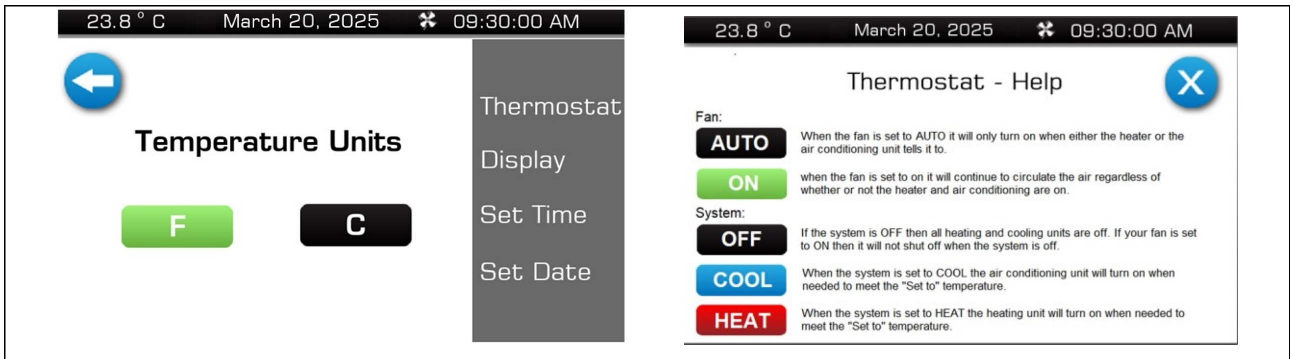


Figure 23. Help Screen

7. Click "**Configure->Project/Display**" and confirm the following settings.

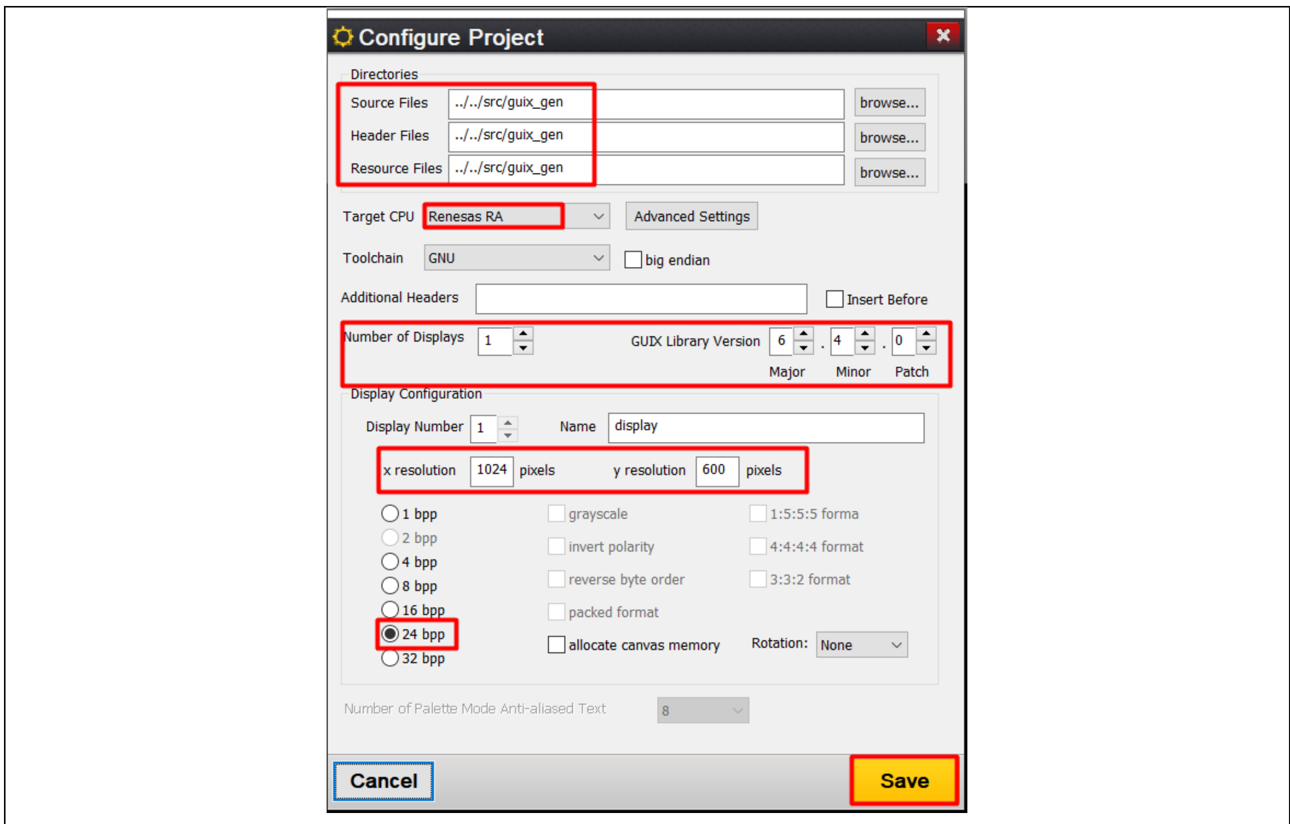


Figure 24. Configure Project Settings

8. Go back e^2 studio project (**ek_ra8d2_graphic_guix_par_cpu0**), right click "**src**", then select "**New->Folder**" and create a folder named "**guix_gen**".

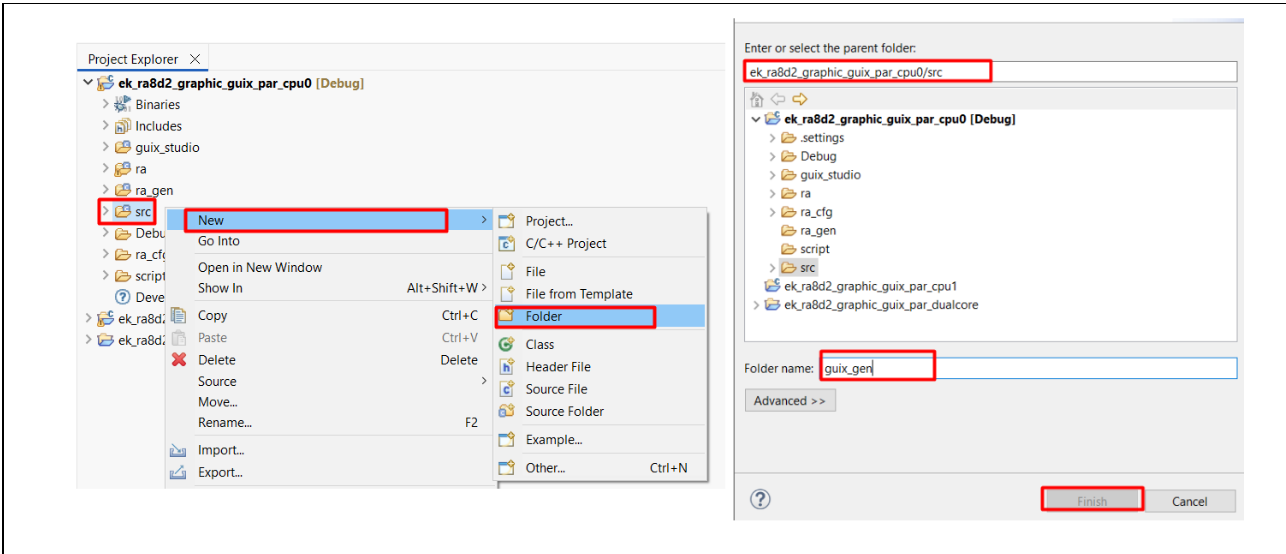


Figure 25. Creating a "guix_gen" in e2 studio Project

9. Confirm "**guix_gen**" is created before moving to next step.

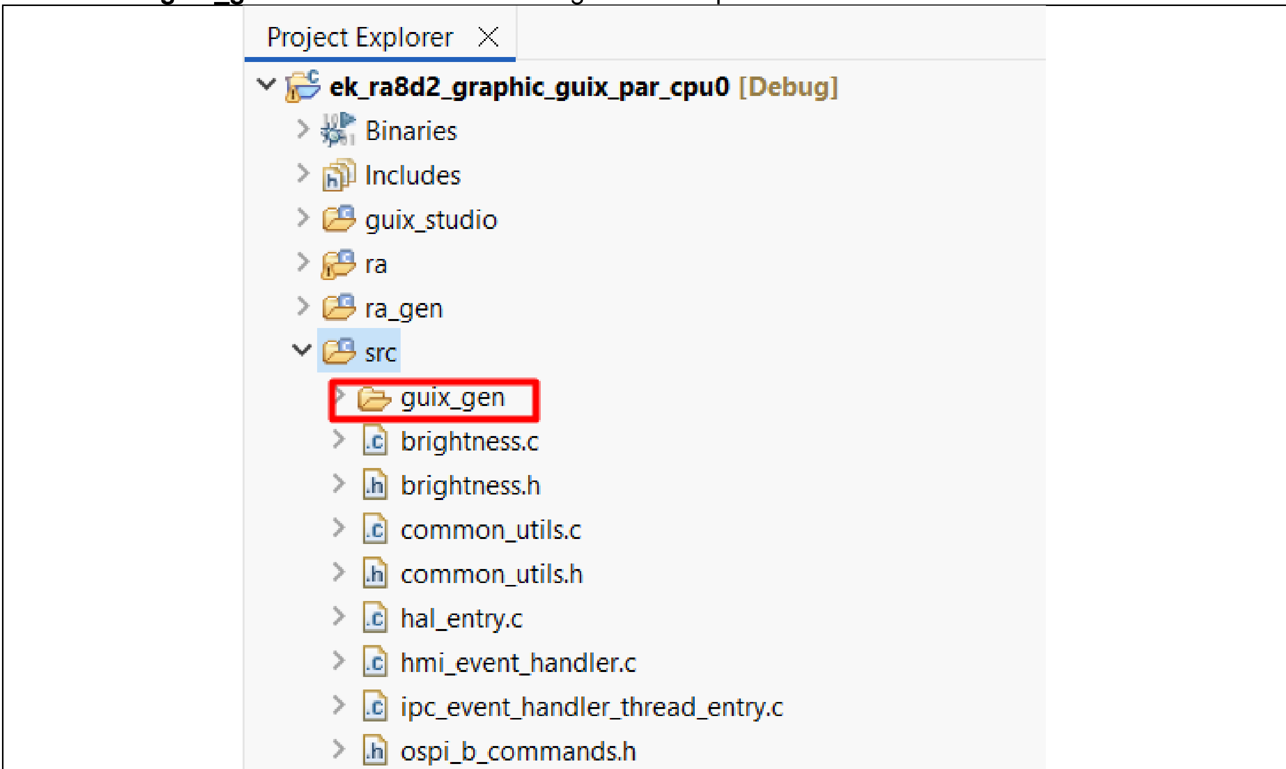


Figure 26. Confirming Creation of "guix_gen"

10. In Azure RTOS GUIX Studio, click **Project->Generate All Output Files** to generate resource files, header files and source files of this GUIX design.

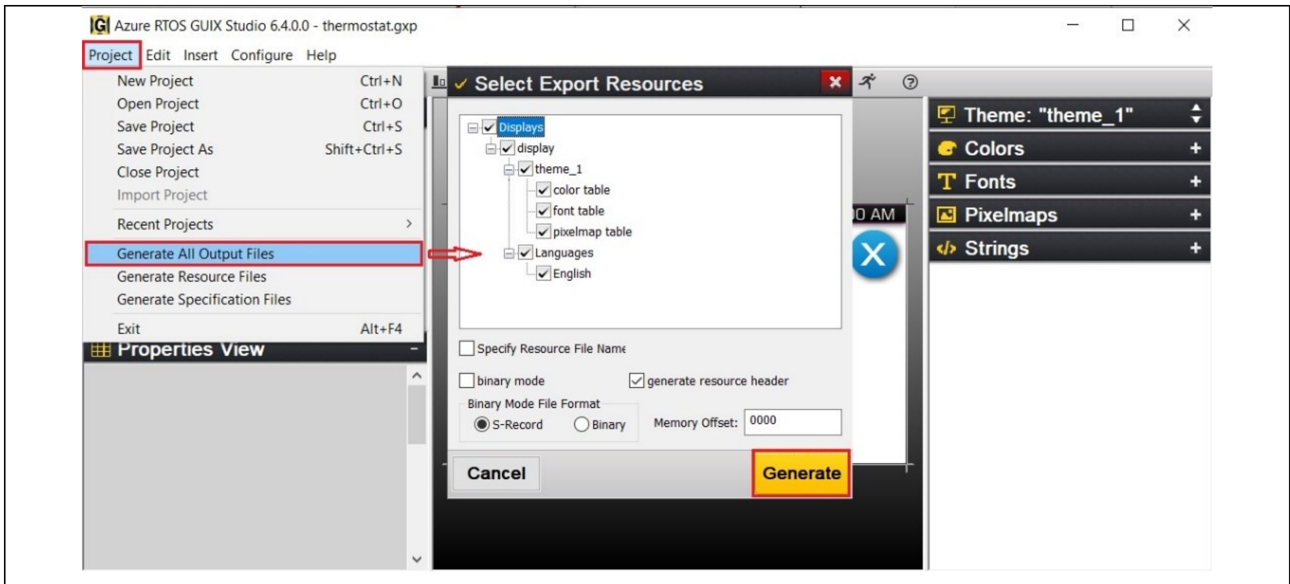


Figure 27. Clicking Generate All Output Files

11. Click **Generate** to generate all output files. If you succeed, you will see the notification below.

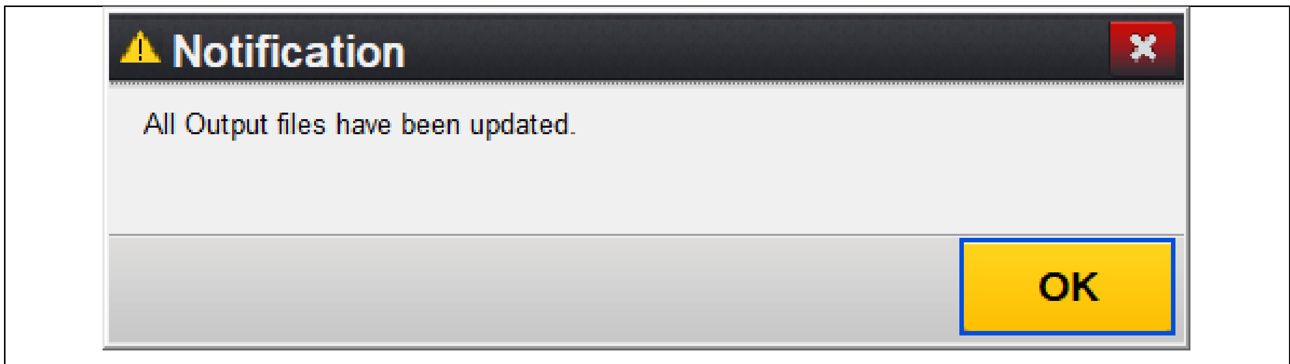


Figure 28. All Output Files Updated Notification

12. All output files are now in “**guix_gen**” folder.

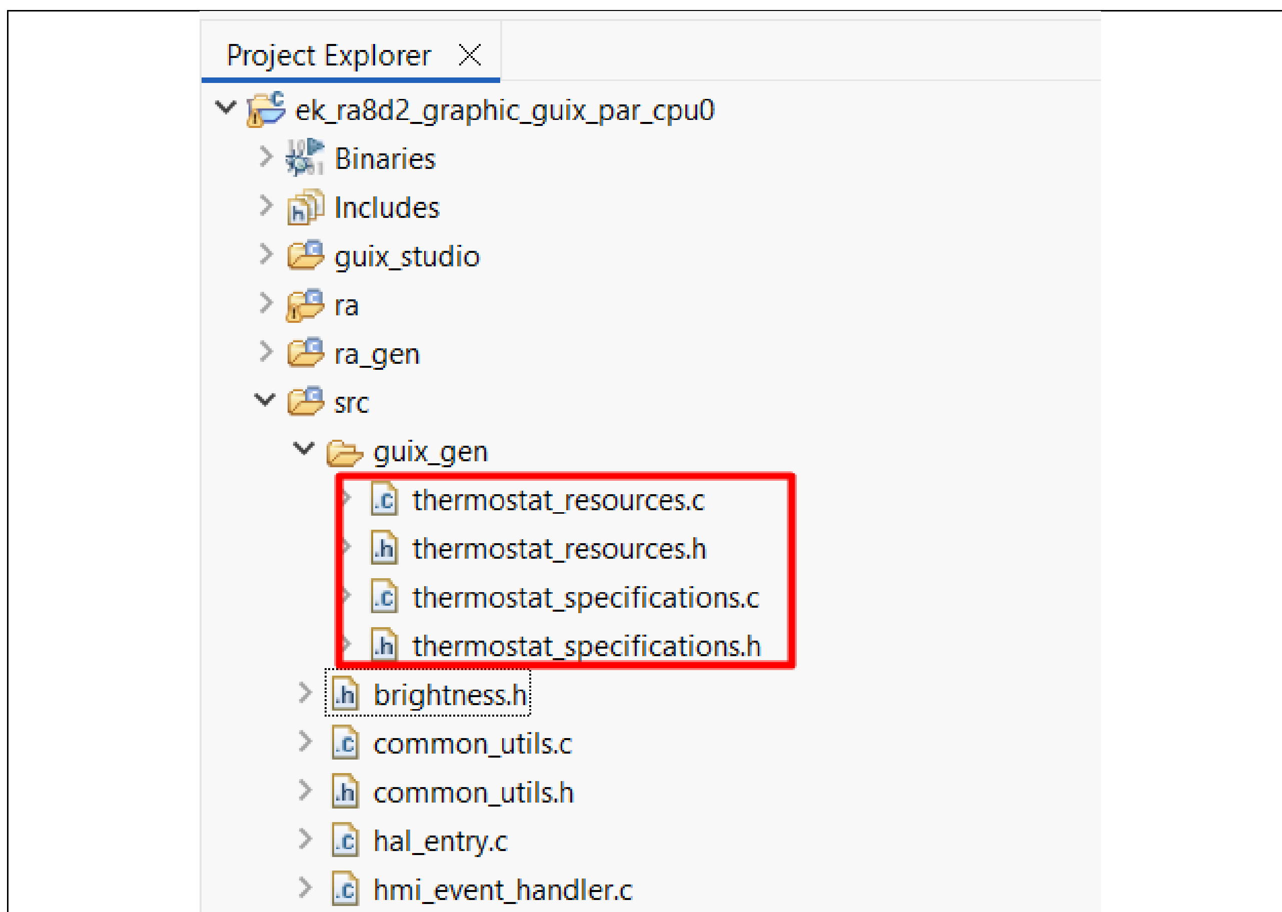


Figure 29. Location of Output Files

3.4 Code Highlights

The graphic layer of the GUIX Thermostat application is implemented using GUIX Studio and Azure RTOS ThreadX on a dual-core RA8D2 system. Most graphic processing is handled in system_thread_entry.c (initialization, display setup, framebuffers) and hmi_event_handler.c (widget updates, event handling).

This chapter highlights the key graphic structures and patterns that drive screen updates and provide smooth, responsive visual behavior.

- GUI Initialization and Display Setup

```
/* Configure main display and create root window */
gx_err = gx_studio_display_configure(DISPLAY,
                                     rm_guix_port_hw_initialize,
                                     LANGUAGE_ENGLISH,
                                     DISPLAY_THEME_1,
                                     &p_root);
if (GX_SUCCESS != gx_err)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}

/* Assign canvas memory buffer for rendering */
gx_err = gx_canvas_memory_define(p_root->gx_window_root_canvas,
                                 rm_guix_port_canvas,
                                 p_root->gx_window_root_canvas->gx_canvas_memory_size);
if (GX_SUCCESS != gx_err)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}
```

Figure 30. GUI Initialization and Display Setup

- Creating and displaying main GUIX widgets

```
/* Create the widget and attached to root window */
gx_err = gx_studio_named_widget_create("Splash", (GX_WIDGET *)p_root, (GX_WIDGET **)&p_splash_screen);
if (GX_SUCCESS != gx_err)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}

gx_err = gx_studio_named_widget_create("Settings", GX_NULL, (GX_WIDGET **)&p_settings_screen);
if (GX_SUCCESS != gx_err)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}

gx_err = gx_studio_named_widget_create("MainPage", GX_NULL, (GX_WIDGET **)&p_mainpage_screen);
if (GX_SUCCESS != gx_err)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}

gx_err = gx_studio_named_widget_create("Thermostat", GX_NULL, (GX_WIDGET **)&p_thermostat_screen);
if (GX_SUCCESS != gx_err)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}

gx_err = gx_studio_named_widget_create("Help", GX_NULL, (GX_WIDGET **)&p_help_screen);
if (GX_SUCCESS != gx_err)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}

/* Shows the root window */
gx_err = gx_widget_show(p_root);
if (GX_SUCCESS != gx_err)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}
```

Figure 31. Creating and displaying main GUIX widgets

Additionally, to gain a deeper understanding of how the application handles events, switches screens, updates text, etc., refer to the functions in the hmi_event_handler.c file. The figures below illustrate examples of the application's screen transitions and event handling.

- Screen switching & event focus

```
 * @brief   Toggles between top level screens defined with .
static void toggle_screen(GX_WINDOW *new_win, GX_WINDOW *old_win)
{
    UINT gx_err = GX_SUCCESS;

    if (!new_win->gx_widget_parent)
    {
        gx_err = gx_widget_attach(p_root, (GX_WIDGET *) new_win);
        if (GX_SUCCESS != gx_err) {
            APP_ERR_TRAP(FSP_ERR_ASSERTION);
        }

    }
    else
    {
        gx_err = gx_widget_show((GX_WIDGET *) new_win);
        if (GX_SUCCESS != gx_err) {
            APP_ERR_TRAP(FSP_ERR_ASSERTION);
        }

        /** User defined events are routed to the widget that has the current
        gx_err = gx_system_focus_claim(new_win);
        if (GX_SUCCESS != gx_err && GX_NO_CHANGE != gx_err) {
            APP_ERR_TRAP(FSP_ERR_ASSERTION);
            }

    }
    gx_err = gx_widget_hide((GX_WIDGET *) old_win);
    if (GX_SUCCESS != gx_err) {
        APP_ERR_TRAP(FSP_ERR_ASSERTION);
        }

}
```

Figure 32. Screen switching & event focus

- Handle touch event when Thermostat button and Settings button are clicked.

```
 * @brief   Handles all events on the main screen.
UINT mainpage_event(GX_WINDOW *widget, GX_EVENT *event_ptr)
{
    UINT gx_err = GX_SUCCESS;
    switch (event_ptr->gx_event_type)
    {
     case GX_SIGNAL(ID_THERMO_BUTTON, GX_EVENT_CLICKED):
         /** Shows the thermostat control screen. */
         toggle_screen (p_thermostat_screen, p_mainpage_screen);
         break;
     case GX_SIGNAL(ID_SETTINGS_BUTTON, GX_EVENT_CLICKED):
         /** Shows the settings screen and saves which screen the user is currently viewing. */
         toggle_screen (p_settings_screen, widget);
         break;
     case GX_EVENT_SHOW:
         /** Update initial text fields according to system settings before the window shows. */
         /** Do default window processing first. */
         gx_err = gx_window_event_process(widget, event_ptr);
         if(GX_SUCCESS != gx_err) {
             while(1);
         }
         break;
     default:
         gx_err = gx_window_event_process(widget, event_ptr);
         if(GX_SUCCESS != gx_err) {
             while(1);
         }
         break;
    }
    return gx_err;
}
```

Figure 33. Thermostat button and Settings button clicked

3.5 Enabling SDRAM Support for Framebuffers

This project uses Azure RTOS GUIX with a framebuffer configuration for a display resolution of 1024 × 600 pixels at 32 bpp, which requires approximately 2.4 MB of memory — exceeding the 1 MB on-chip MRAM (Code Flash) capacity of the MCU. Therefore, the framebuffer cannot be stored in the internal MRAM and must be allocated in external SDRAM (512 Mbit = 64 MB) available on the EK-RA8D2 board, providing sufficient capacity for single or multiple framebuffers as required by the application.

To enable the use of SDRAM for the framebuffer, follow the steps below:

1. Open the project configuration, navigate to the **BSP** tab, and enable and configure SDRAM.

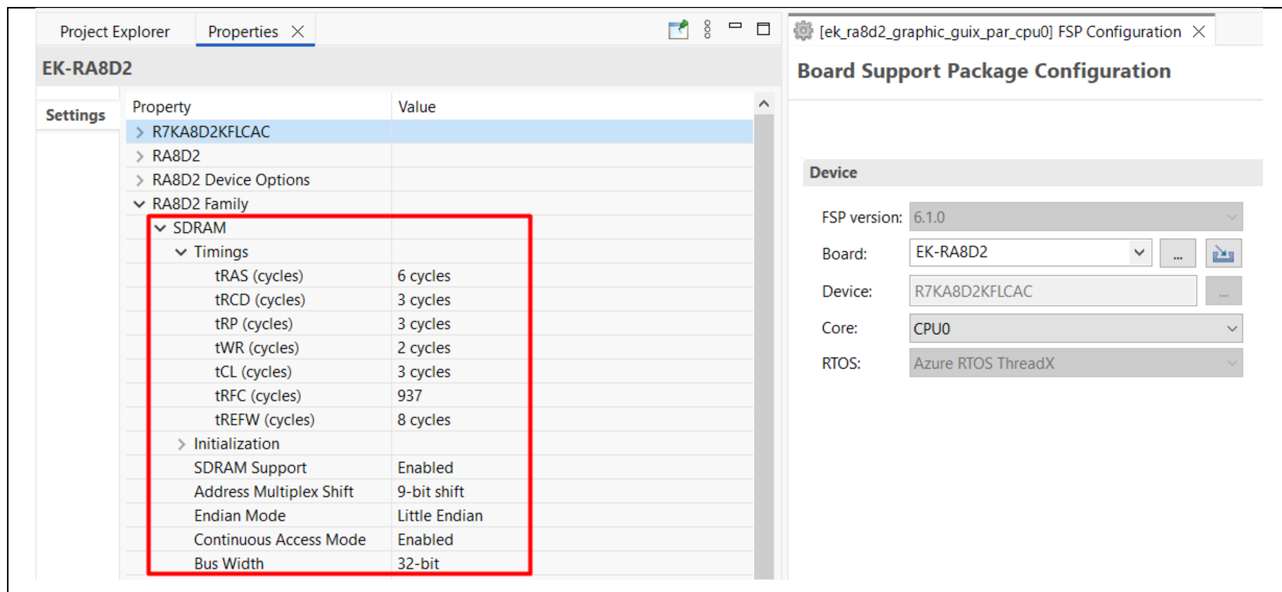


Figure 34. Setting SDRAM Properties

2. In the **Graphics LCD** settings, set the framebuffer section to **.sdram_noinit** to allocate it in external SDRAM.

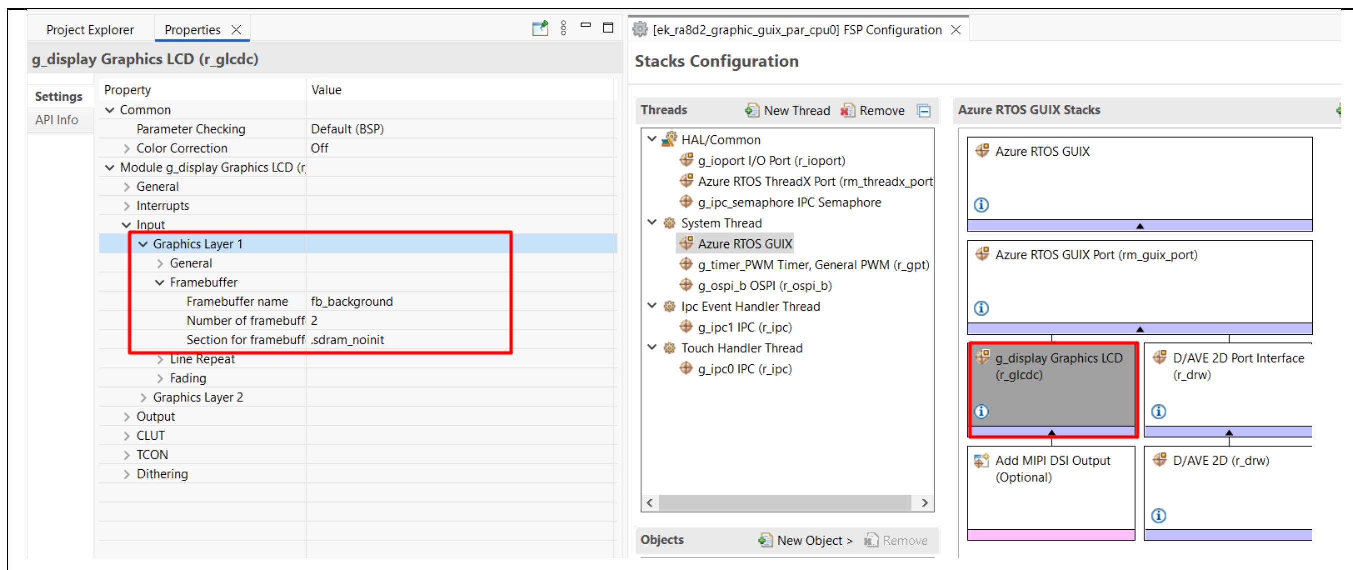


Figure 35. Framebuffer allocation in SDRAM

3. After SDRAM is enabled, it will be initialized in the application code by the function **"R_BSP_SdramInit(true)".**

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

Figure 36. SDRAM Initialization

4. GUIX Asset Storage in External OSPI Flash

4.1 Overview

Due to the 1 MB MRAM limit on EK-RA8D2, GUIX image and font assets cannot be stored internally without causing overflow. These resources are therefore placed in external OSPI flash (CS1), while MRAM is reserved for application and RTOS code. This setup ensures sufficient capacity and reliable non-volatile storage for graphics assets.

4.2 Configuration Steps

To configure GUIX asset storage in external OSPI flash, perform the following steps:

1. Add and configure **OSPI driver** as shown in the figures below.

Figure 37. OSPI driver configuration – General settings

Initial Mode	
Read	
Command Code	0x0C
Dummy Cycles	2
Program	
Command Code	0x12
Dummy Cycles	0
Row Load	
Command Code	0x00
Dummy Cycles	0
Row Store	
Command Code	0x00
Dummy Cycles	0
Write Enable	
Command Code	0x06
Status Read	
Command Code	0x05
Dummy Cycles	0
Sector Erase	
Command Code	0x21
Block Erase	
Command Code	0xDC
Chip Erase	
Command Code	0x60
Protocol Mode	SPI (1S-1S-1S)
Frame Format	Standard
Latency Mode	Fixed
Address Length	4 bytes
Address MSB Mask	0xF0
Command Code Length	1 byte
Status Register	No address
Status Register Address	0x00

High-speed Mode	
Read	
Command Code	0x0C0C
Dummy Cycles	13
Program	
Command Code	0x1212
Dummy Cycles	0
Row Load	
Command Code	0x00
Dummy Cycles	0
Row Store	
Command Code	0x00
Dummy Cycles	0
Write Enable	
Command Code	0x0606
Status Read	
Command Code	0x0505
Dummy Cycles	8
Sector Erase	
Command Code	0x2121
Block Erase	
Command Code	0xDCDC
Chip Erase	
Command Code	0x6060
Protocol Mode	Dual data rate OPI (8D-8D-8D)
Frame Format	xSPI Profile 1.0
Latency Mode	Fixed
Address Length	4 bytes
Address MSB Mask	0xF0
Command Code Length	2 bytes
Status Register	4 bytes
Status Register Address	0x00

Figure 38. OSPI driver configuration – Extended settings

2. Set OSPI pins configuration.

Pin

Type filter text

- PC
- PD
- Other Pins
- Peripherals
  - Analog:ACMPHS
  - Analog:ADC0
  - Analog:ADC1
  - Analog:DAC12
  - CLKOUT:CLKOUT
  - CLKOUT:CMPOUT
  - CLKOUT:ETHPHYCLK
  - Connectivity:CANFD
  - Connectivity:ESWM_GMII/MII
  - Connectivity:ESWM_RGMII
  - Connectivity:ESWM_RMII
  - Connectivity:I3C
  - Connectivity:IIC
  - Connectivity:OSPI
    - OSPI0
    - OSPI1

Pin

Name	Value	Lock	Link
Pin Group Selection	Mixed		
Operation Mode	Hyper Flash		
Input/Output			
OM_0_CS0	None		
OM_0_CS1	P104		
OM_0_DQS	P801		
OM_0_ECSINT1	P105		
OM_0_RESET	P106		
OM_0_RSTO1	None		
OM_0_SCLK	P808		
OM_0_SCLKN	None		
OM_0_SIO0	P100		
OM_0_SIO1	P803		
OM_0_SIO2	P103		
OM_0_SIO3	P101		
OM_0_SIO4	P102		
OM_0_SIO5	P800		
OM_0_SIO6	P802		
OM_0_SIO7	P804		
OM_0_WP1	None		

Figure 39. OSPI pins configuration

3. Placing GUIX Resources in External Flash Memory

You need to add the “User Mappings” in **configuration.xml > Linker Sections** to allocate GUI resource images in external flash. This is because most image resources generated by GUIX are declared as constants (GX_CONST).

Choose **New User Mapping > OSPI_CS1 > OSPI0_CS1 Constant Data** to add New User Mapping to OSPI CS1.

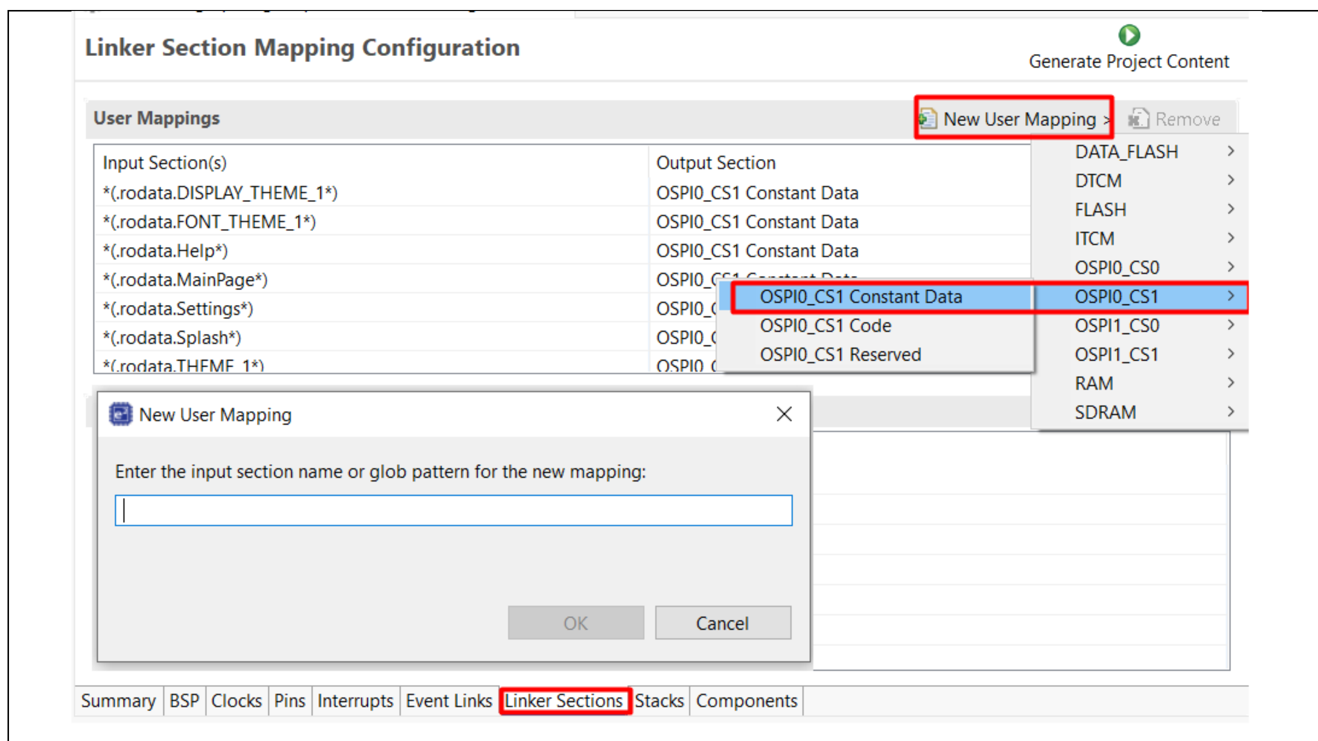


Figure 40. Add New User Mapping to allocate GUI resource to OSPI CS1

In this step, user needs to check the name of the data that needs to be allocated to the OSPI CS1 section. For example, in GUI resource file (src/guix_gen/thermostat_resources.c), there is a const array named **DISPLAY_THEME_1_RADIO_ON_pixelmap_data**. User can allocate this array to OSPI CS1 by entering the name **.rodata.DISPLAY_THEME_1*** into the input box and click **OK,** as shown below:

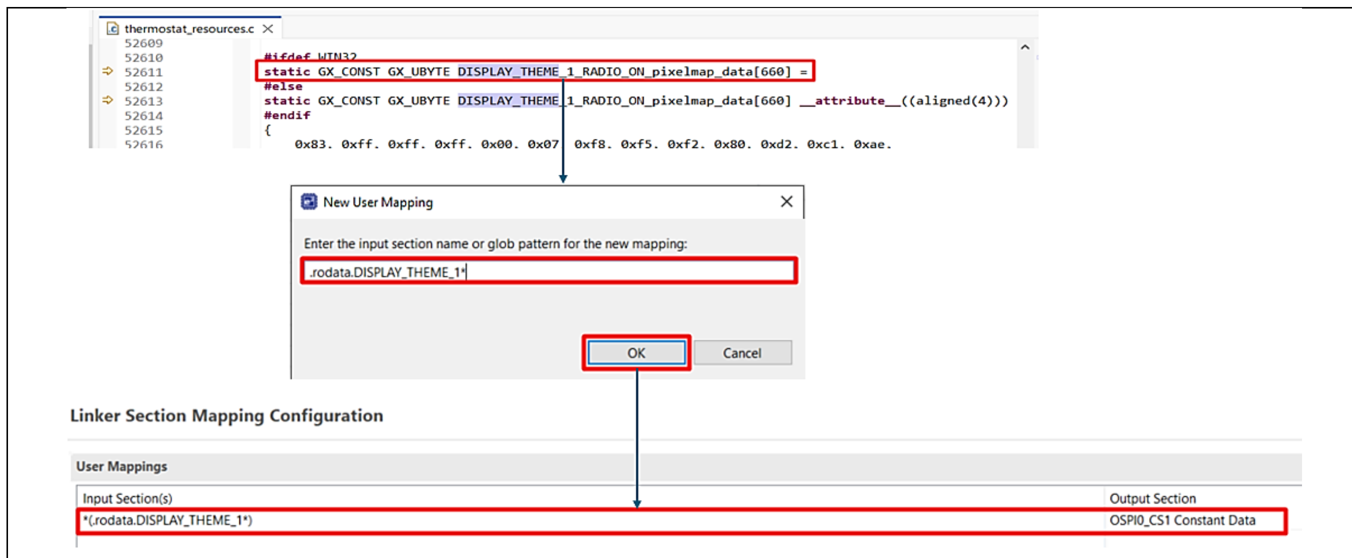


Figure 41. Add User Mapping for GUI resource – rodata

Other arrays in the resource follow a similar pattern. Please continue adding the input sections below in order to allocate the GUI resource to OSPI CS1 and run the application.

Table 1 Input sections

(.rodata.DISPLAY_THEME_1)
(.rodata.FONT_THEME_1)
(.rodata.Help)
(.rodata.MainPage)
(.rodata.Settings)
(.rodata.THEME_1)
(.rodata.Thermostat)
*(.rodata.display_theme_1_color_table)
*(.rodata.thermostat_widget_table)

After adding the input section, the linker section will appear as shown below.

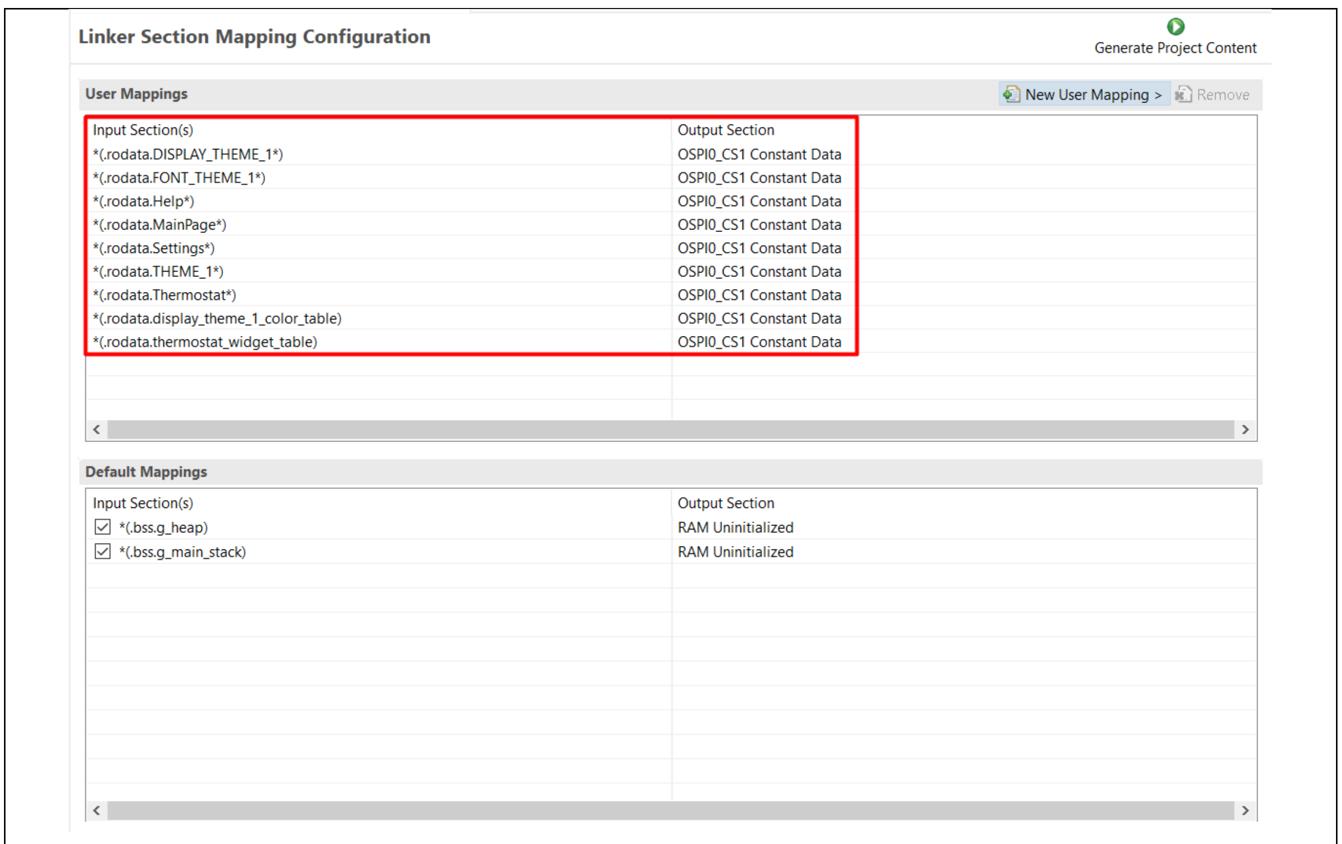


Figure 42. Linker Sections after adding all the input section

Note: The above list is intended only for the generated GUIX source included with this application. Users can check and modify it according to their corresponding source code to ensure that GUI resources are allocated to the OSPI CS1 region.

4. Enable Data cache to optimize performance.

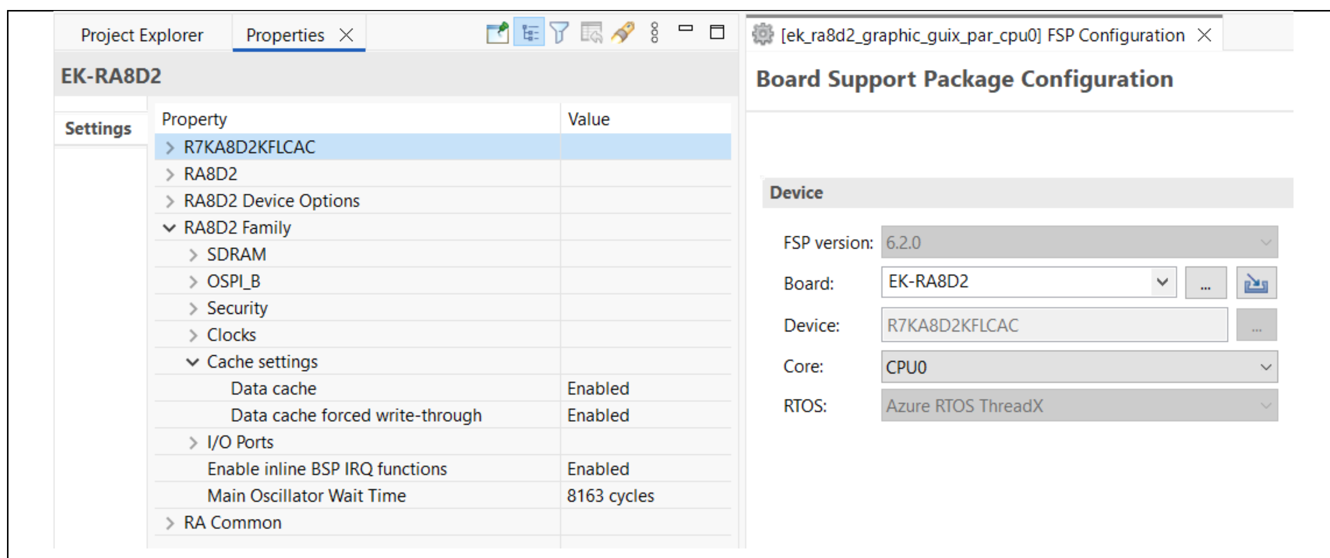


Figure 43. Enable Data cache

4.3 Code Highlights

The initialization flow starts with the OSPI driver, sets the OPI protocol, and enters XIP mode to allow direct access to GUIX assets stored in external flash. The application also initializes OSPI and enables memory mapping in **"system_thread_entry.c".** For details, refer to this file and related OSPI driver sources.

```
/* Initialize OSPI flash driver */
err = ospi_b_init();
if (err != FSP_SUCCESS)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}

/* Set OSPI flash to Octal-SPI mode for high-speed access */
err = ospi_b_set_protocol_to_opi();
if (err != FSP_SUCCESS)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}

/* Enable XIP mode (execute code directly from OSPI flash) */
err = ospi_b_xip_enter();
if (err != FSP_SUCCESS)
{
    APP_ERR_TRAP(FSP_ERR_ASSERTION);
}
```

Figure 44. OSPI initialization

5. PWM Backlight Control

5.1 Overview

This chapter describes how the backlight brightness of the display is controlled using PWM generated by the GPT timer. The PWM duty cycle is adjusted based on user input from the GUI, allowing efficient backlight control.

5.2 Configuration Steps

The backlight enable signal (Blen) is connected to pin P514 on the MCU, as shown in Figure 45. To control this pin, the GPT timer is configured to generate a PWM signal whose duty cycle determines the display brightness.

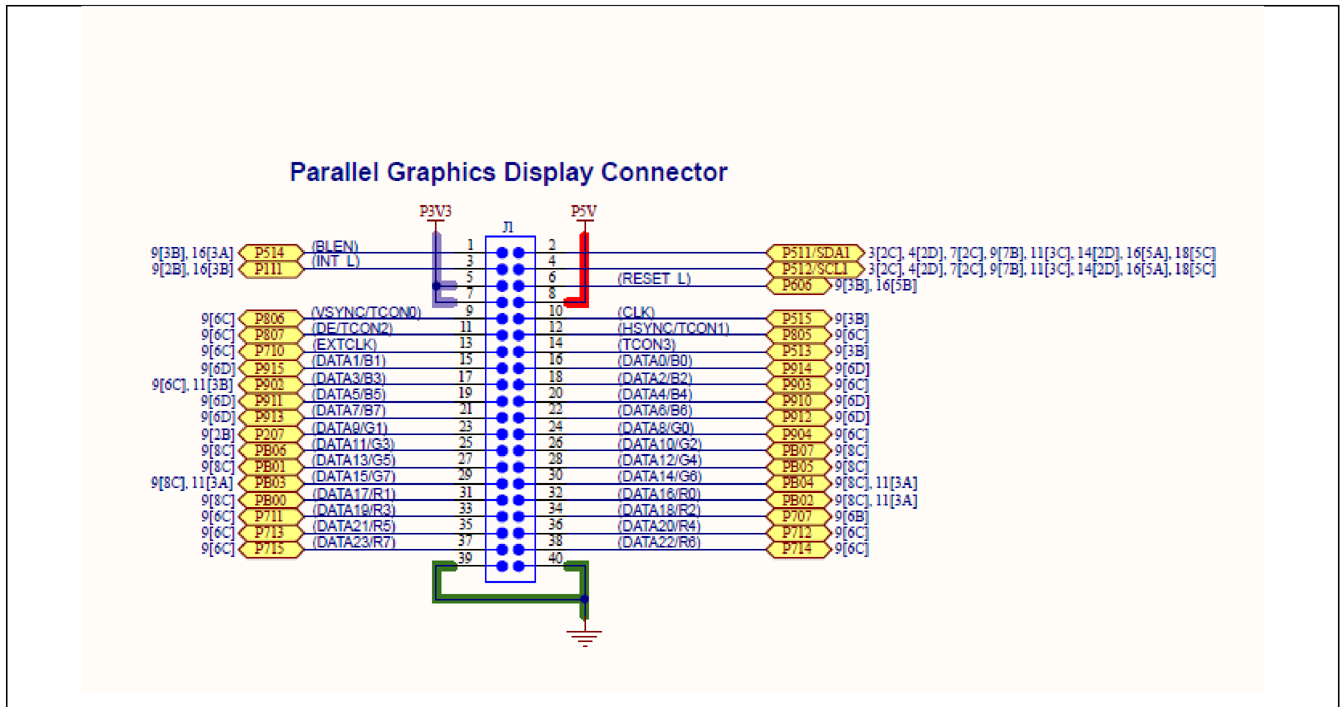


Figure 45. Parallel Graphics Display Connector with Blen connected to P514

The following steps describe how to configure the GPT timer in the FSP tool and assign pin P514 as the PWM output for driving the backlight. Refer to the configuration screenshots for detailed settings.

1. Add and configure **Timer** driver as shown in the figures below.

g_timer_PWM Timer, General PWM (r_gpt)

Settings
API Info

Property	Value
Common	
Parameter Checking	Default (BSP)
Pin Output Support	Enabled
Write Protect Enable	Disabled
Module g_timer_PWM Timer, General PWM (r_gpt)	
General	
Compare Match	
Name	g_timer_PWM
Channel	13
Mode	Saw-wave PWM
Period	200
Period Unit	Hertz
Output	
Custom Waveform	
Duty Cycle Percent (only applicable in PWM mode)	50
GTIOCA Output Enabled	False
GTIOCA Stop Level	Pin Level Low
GTIOCB Output Enabled	True
GTIOCB Stop Level	Pin Level Low
Input	
Pin Polarity	
Interrupts	
Extra Features	
Pins	
GTCPPO13	None
GTIOC13A	None
GTIOC13B	P514

Stacks Configuration

Threads New Thread Remove

- HAL/Common
  - g_ioport I/O Port (r_ioport)
  - Azure RTOS ThreadX Port (rm_threadx_p
  - g_ipc_semaphore1 IPC Semaphore
  - g_ipc_semaphore0 IPC Semaphore
- System Thread
  - Azure RTOS GUIX
  - g_timer_PWM Timer, General PWM (r_g
  - g_ospi_b OSPI (r_ospi_b)
- Ipc Event Handler Thread
  - g_ipc1 IPC (r_ipc)
- Touch Handler Thread
  - g_ipc0 IPC (r_ipc)

Objects New Object > Remove

- system_msg_queue Queue
- g_sys_mutex Mutex
- g_rtc_semaphore Semaphore
- g_touch_semaphore Semaphore

g_timer_PWM Timer, General PWM

g_timer_PWM Timer, General PWM (r_gpt)

Summary | BSP | Clocks | Pins | Interrupts | Event Links | Linker Sections | Stacks | Components

Figure 46. Timer driver configuration

2. Set Timer pin configuration.

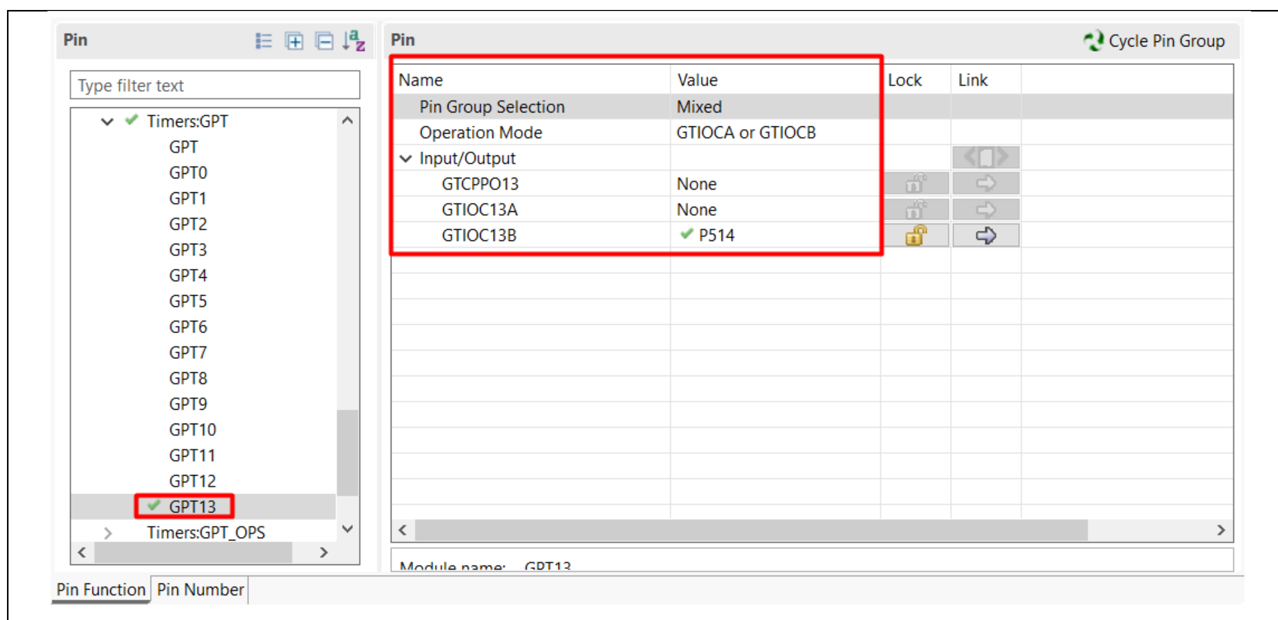


Figure 47. Timer pin configuration

5.3 Code Highlights

The implementation of the backlight control includes the initialization procedure and the brightness adjustment routines. These routines regulate the PWM duty cycle of the GPT timer to increase or decrease the display brightness. The implementation is illustrated in the following code example.

1. The function gpt_timer_PWM_setup is used to initialize the GPT timer for PWM generation and also to configure the duty cycle that determines the LCD backlight brightness, as shown in Figure 48.

```
 * @brief This function is setting up GPT/PWM timer
static fsp_err_t gpt_timer_PWM_setup(void) // @suppress("8.1b, 8.1f Non-API function naming")
{
    fsp_err_t err = FSP_SUCCESS;
    /* Open GPT */
    err = R_GPT_Open(&g_timer_PWM_ctrl, &g_timer_PWM_cfg);
    if(FSP_SUCCESS != err)
    {
        return err;
    }
    /* Enable GPT Timer */
    err = R_GPT_Enable(&g_timer_PWM_ctrl);
    /* Handle error */
    if (FSP_SUCCESS != err)
    {
        return err;
    }
    /* Start GPT timer */
    err = R_GPT_Start(&g_timer_PWM_ctrl);;
    if(FSP_SUCCESS != err)
    {
        return err;
    }

    /* Set brightness (LCD backlight) level: 50 = (45+5) */
    g_gui_state.brightness = 45;
    brightness_up(&g_gui_state.brightness);

    return err;
}
```

Figure 48. Backlight initialization

2. The **brightness_up** and **brightness_down** functions in **brightness.c** are responsible for adjusting the PWM duty cycle to control the LCD backlight brightness, as illustrated below.

```
void brightness_up(uint8_t * p_brightness)
{
    fsp_err_t err                   = FSP_SUCCESS;
    timer_info_t info               = {0};
    uint32_t duty_cycle_count       = 0;
    int8_t brightness               = (int8_t)*p_brightness;

    brightness = (int8_t)(brightness + BRIGHTNESS_INC);
    if ((uint8_t)brightness > BRIGHTNESS_MAX)
    {
        brightness = BRIGHTNESS_MAX;
    } else
    if (brightness < *p_brightness)
    {
        brightness = (100);
    }
    else
    {
    }
    /* Get the current period setting. */
    R_GPT_InfoGet(&g_timer_PWM_ctrl, &info);

    /* Calculate the desired duty cycle based on the current period.
     * Note that if the period could be larger than
     * UINT32_MAX / 100, this calculation could overflow. */
    duty_cycle_count = (uint32_t) ((info.period_counts * brightness)/GPT_PWM_MAX_PERCENT);

    err = R_GPT_DutyCycleSet(&g_timer_PWM_ctrl, duty_cycle_count, GPT_IO_PIN_GTIOCB);

    if (FSP_SUCCESS == err)
    {
        *p_brightness = (uint8_t)brightness;
    }
}
```

Figure 49. Brightness increase function

```
void brightness_down(uint8_t * p_brightness)
{
    fsp_err_t err                   = FSP_SUCCESS;
    timer_info_t info               = {0};
    uint32_t duty_cycle_count       = 0;
    int8_t brightness               = (int8_t)*p_brightness;

    brightness = (int8_t)(brightness - BRIGHTNESS_INC);
    if (brightness < BRIGHTNESS_MIN)
    {
        brightness = BRIGHTNESS_MIN;
    } else
    if (brightness > (int8_t) *p_brightness)
    {
        brightness = (10);
    }
    else
    {
    }

    /* Get the current period setting. */
    R_GPT_InfoGet(&g_timer_PWM_ctrl, &info);

    /* Calculate the desired duty cycle based on the current period.
     * Note that if the period could be larger than
     * UINT32_MAX / 100, this calculation could overflow. */
    duty_cycle_count = (uint32_t) ((info.period_counts * brightness)/GPT_PWM_MAX_PERCENT);
    err = R_GPT_DutyCycleSet(&g_timer_PWM_ctrl, duty_cycle_count, GPT_IO_PIN_GTIOCB);
    if (FSP_SUCCESS == err)
    {
        *p_brightness = (uint8_t)brightness;
    }
}
```

Figure 50. Brightness decrease function

6. Dual-Core Touch Handling

6.1 Overview

In this project, the touch system is handled using dual-core architecture. CPU1 is responsible for receiving touch input and transmitting the coordinates to CPU0, which handles graphics-related tasks such as screen transitions and updating display information. This section focuses on how to configure touch across both cores and highlights key aspects of touch event handling.

6.2 CPU0 – GUI Event Handling

6.2.1 Key Configuration

The following highlights illustrate the key configuration steps for handling touch input on CPU0. These steps provide a reference for setting up graphics event processing based on data received from CPU1.

In this project, IPC0 and g_touch_semaphore will be configured to signal touch_handler_thread that there is touch data sent from CPU1.

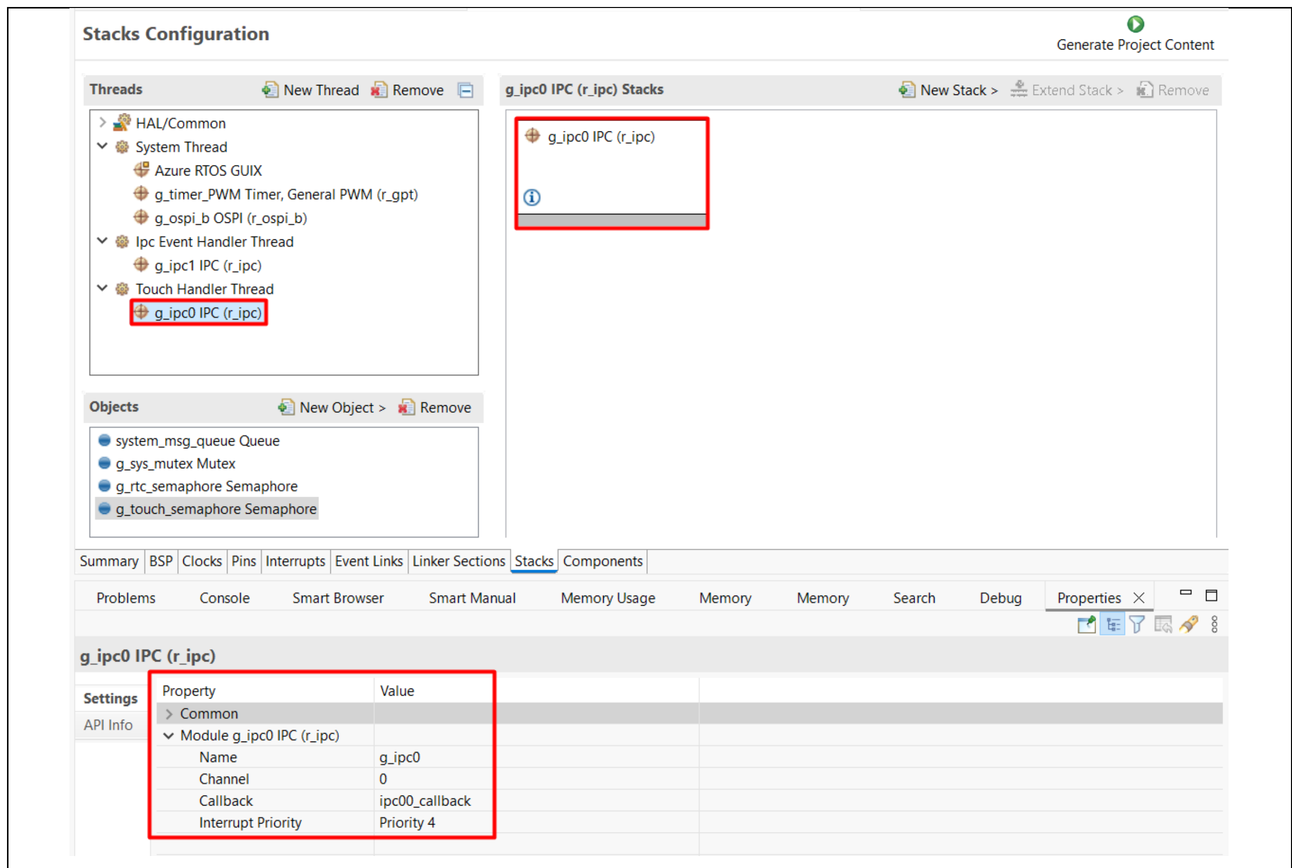


Figure 51. IPC Channel 0 configuration on CPU0

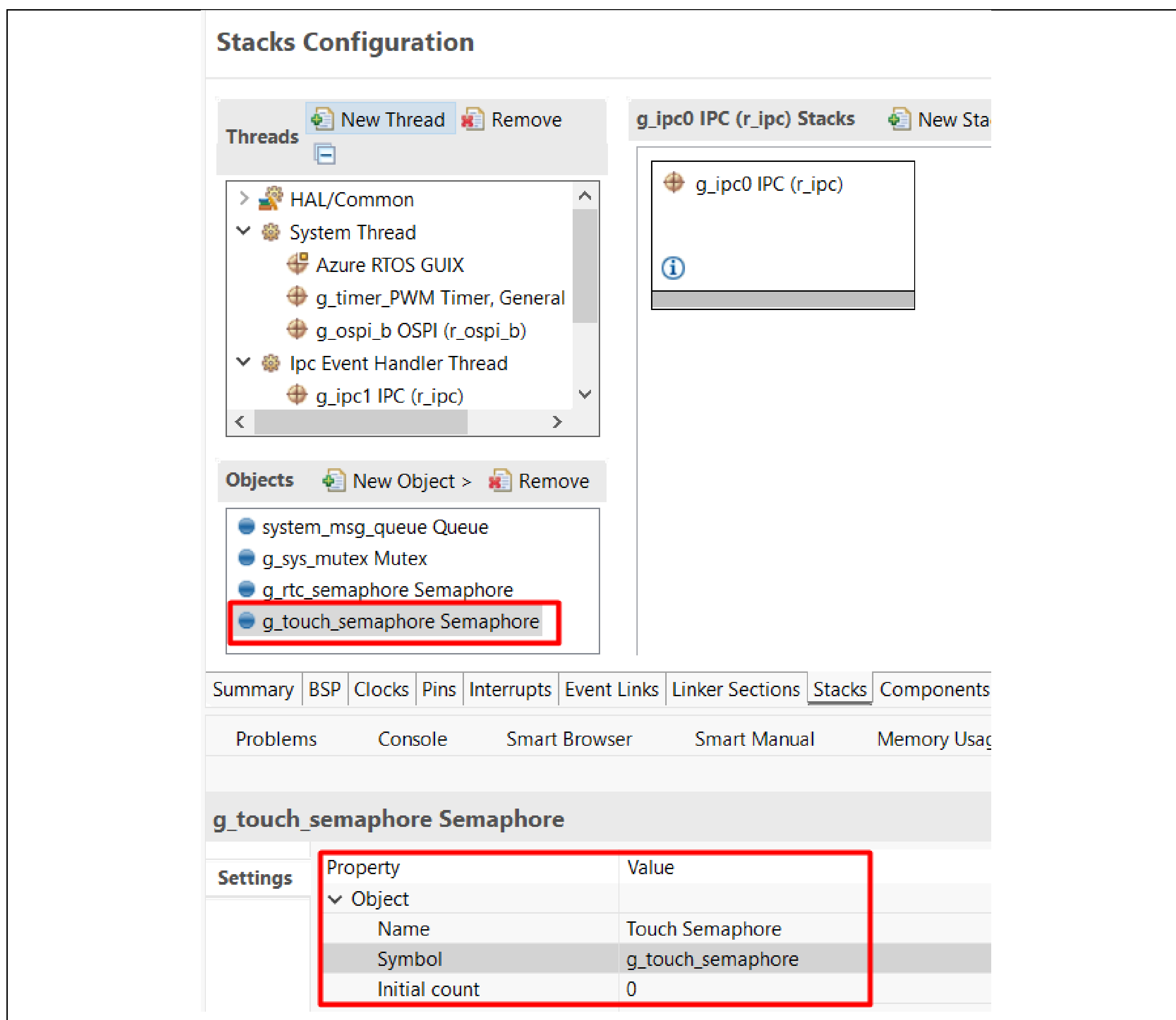


Figure 52. g_touch_semaphore configuration on CPU0

6.2.2 Code Highlights

The following code highlights illustrate how CPU0 receives touch data from CPU1 and updates the GUI accordingly.

After touch data is sent from CPU1, the ipc00_callback function is invoked to receive the message and notify the touch_handler_thread to process the touch event.

```
void ipc00_callback(ipc_callback_args_t *p_args)
{
    switch(p_args->event)
    {
        case IPC_EVENT_MESSAGE_RECEIVED:
        {
            /* Save touch data and wake handler thread */
            touch_message = p_args->message;
            tx_semaphore_put(&g_touch_semaphore);
            break;
        }
        default:
            break;
    }
}
```

Figure 53. ipc00_callback function

The touch_handler_thread is responsible for processing touch events and sending update requests to the graphics system.

```
void touch_handler_thread_entry(void)
{
    int err = 0;
    UINT status = 0;
    GX_EVENT gxe = {0};

    /* Open IPC0 channel for touch data communication with CPU1 */
    err = R_IPC_Open(&g_ipc0_ctrl, &g_ipc0_cfg);
    if (FSP_SUCCESS != err)
    {
        APP_ERR_TRAP(FSP_ERR_ASSERTION);
    }
    while (1)
    {
        /* Wait for touch data from CPU1 via IPC */
        status = tx_semaphore_get(&g_touch_semaphore, TX_WAIT_FOREVER);
        if (TX_SUCCESS != status)
        {
            APP_ERR_TRAP(FSP_ERR_ASSERTION);
        }

        /* Process touch coordinates and generate GUIX events */
        if(touch_message != 0xFFFFFFFF)
        {
            uint16_t touch_x = (uint16_t)(touch_message & 0xFFFF);
            uint16_t touch_y = (uint16_t)((touch_message >> 16) & 0xFFFF);
            gxe.gx_event_payload.gx_event_pointdata.gx_point_x = touch_x;
            gxe.gx_event_payload.gx_event_pointdata.gx_point_y = touch_y;
            gxe.gx_event_type = GX_EVENT_PEN_DOWN;
            gx_system_event_send(&gxe);
        }
        else
        {
            if (GX_EVENT_PEN_DOWN == gxe.gx_event_type)
            {
                gxe.gx_event_type = GX_EVENT_PEN_UP;
                gx_system_event_send(&gxe);
            }
        }
        tx_thread_sleep(1);
    }
}
```

Figure 54. touch_handler_thread Code

6.3 CPU1 – Touch Data Acquisition

6.3.1 Key Configuration

In this project, the display on the RA8D2 board uses the FT5316 touch controller. Since the application only requires single-point touch handling, the FT5x06 driver files from the "**Getting Started with GUIX Thermostat Application**" (R12AN0120) project are reused without modification.

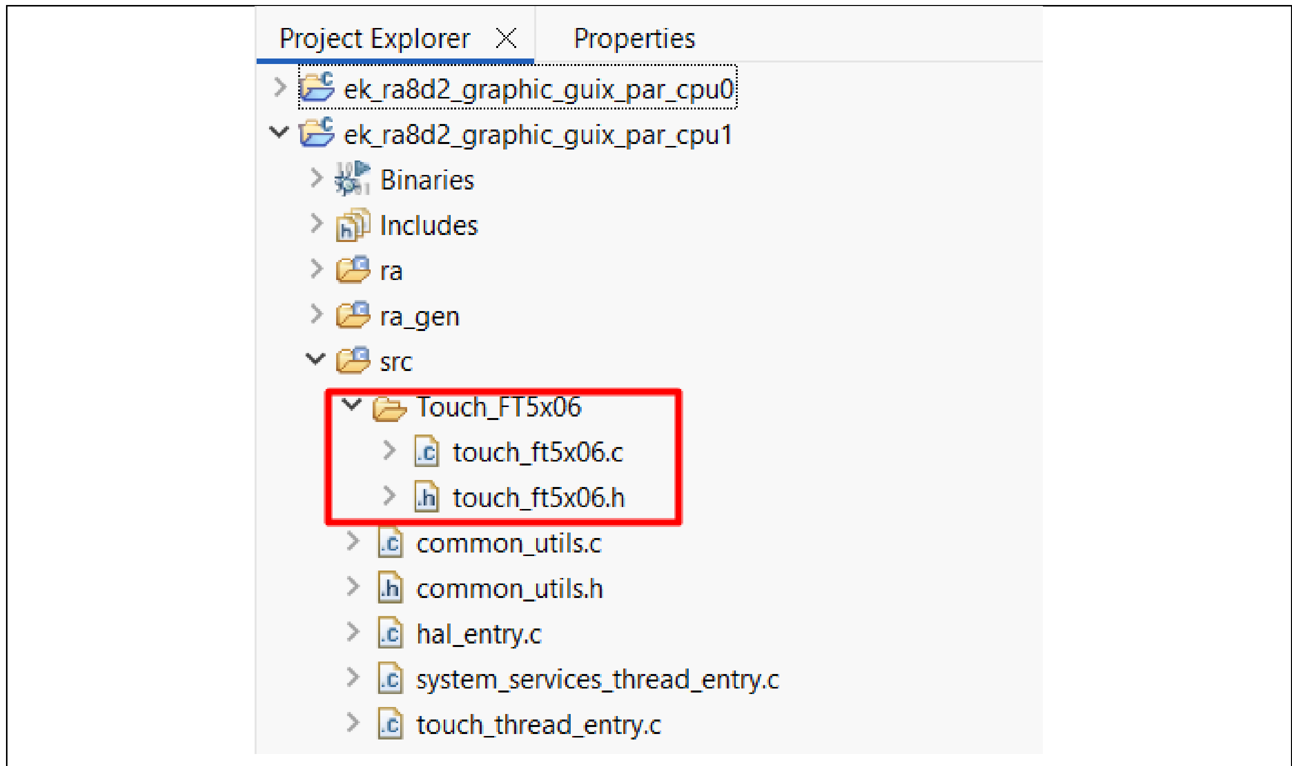


Figure 55. "touch_ft5x06" folder

The following highlights illustrate the key configuration steps for CPU1 in acquiring touch input and transmitting coordinates to CPU0.

The pins marked in red below are used for touch panel controller on the LCD board:

- IRQ19 interrupt (P111) is used to trigger touch events.
- I2C channel 1 (P512, P511) is used to read and write data to the touch controller. P606 is used to reset the touch controller.

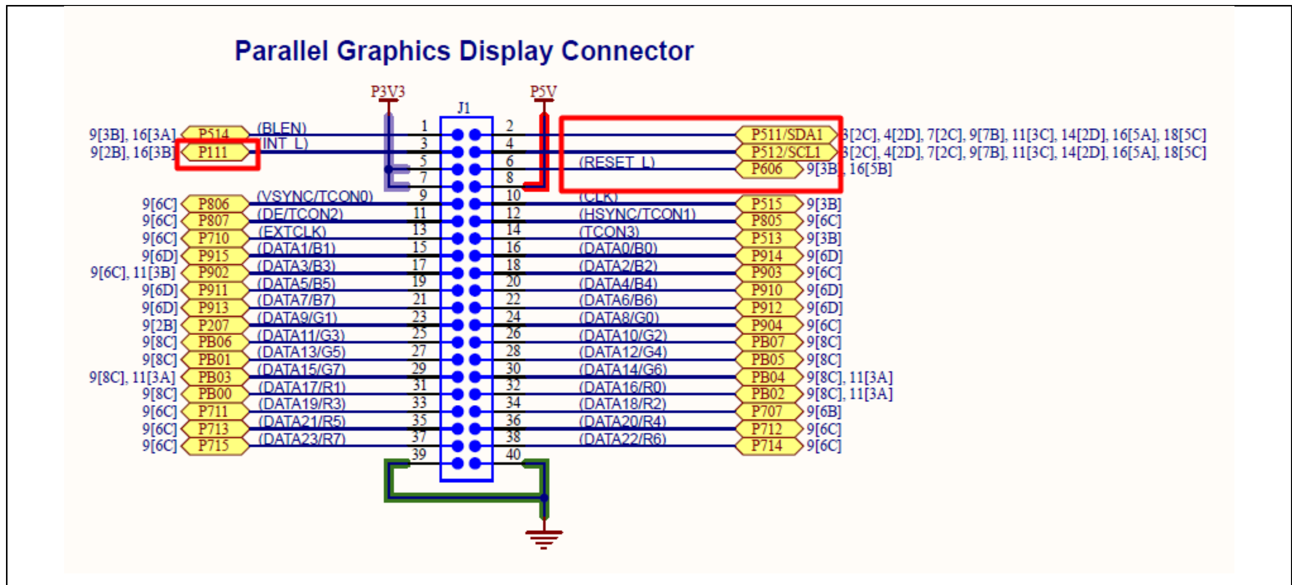


Figure 56. Pins Used in Touch Panel Controller

Setting the **External IRQ** Properties

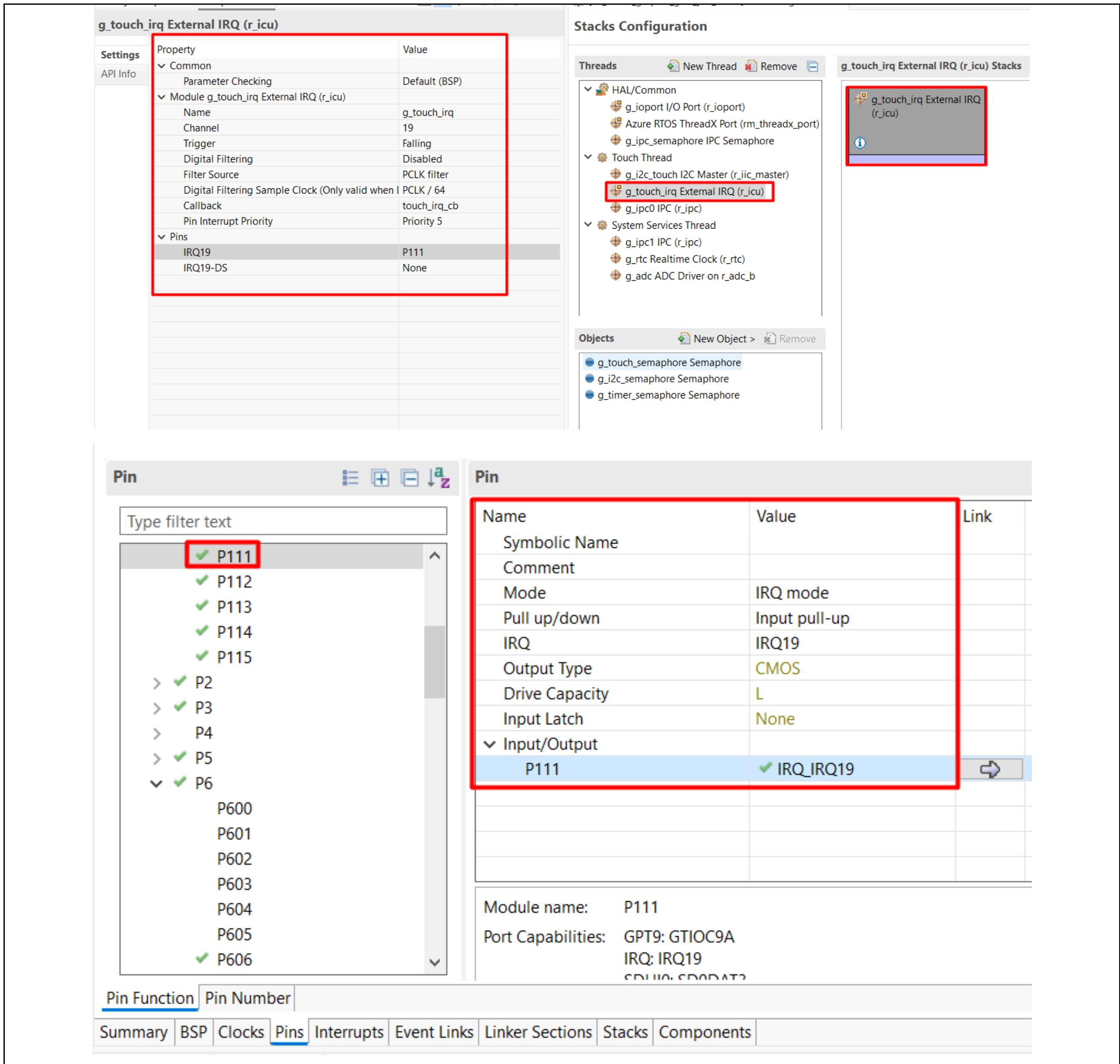


Figure 57. Settings External IRQ Properties

Setting Properties **I2C Master Driver** on **r_iic_master**

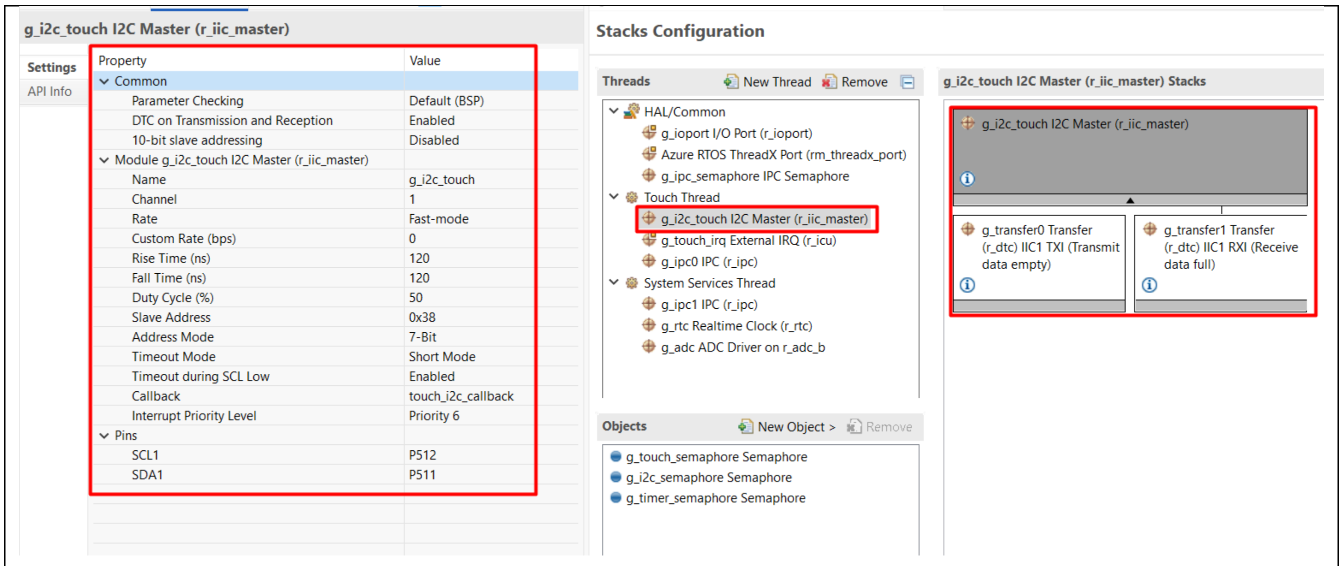


Figure 58. Setting Properties I2C Master Driver

In the project configuration, semaphores are used for I2C (by the ft5x06 driver to read data on touch-panel interrupts) and for Touch (to signal the Touch thread when a touch event occurs).

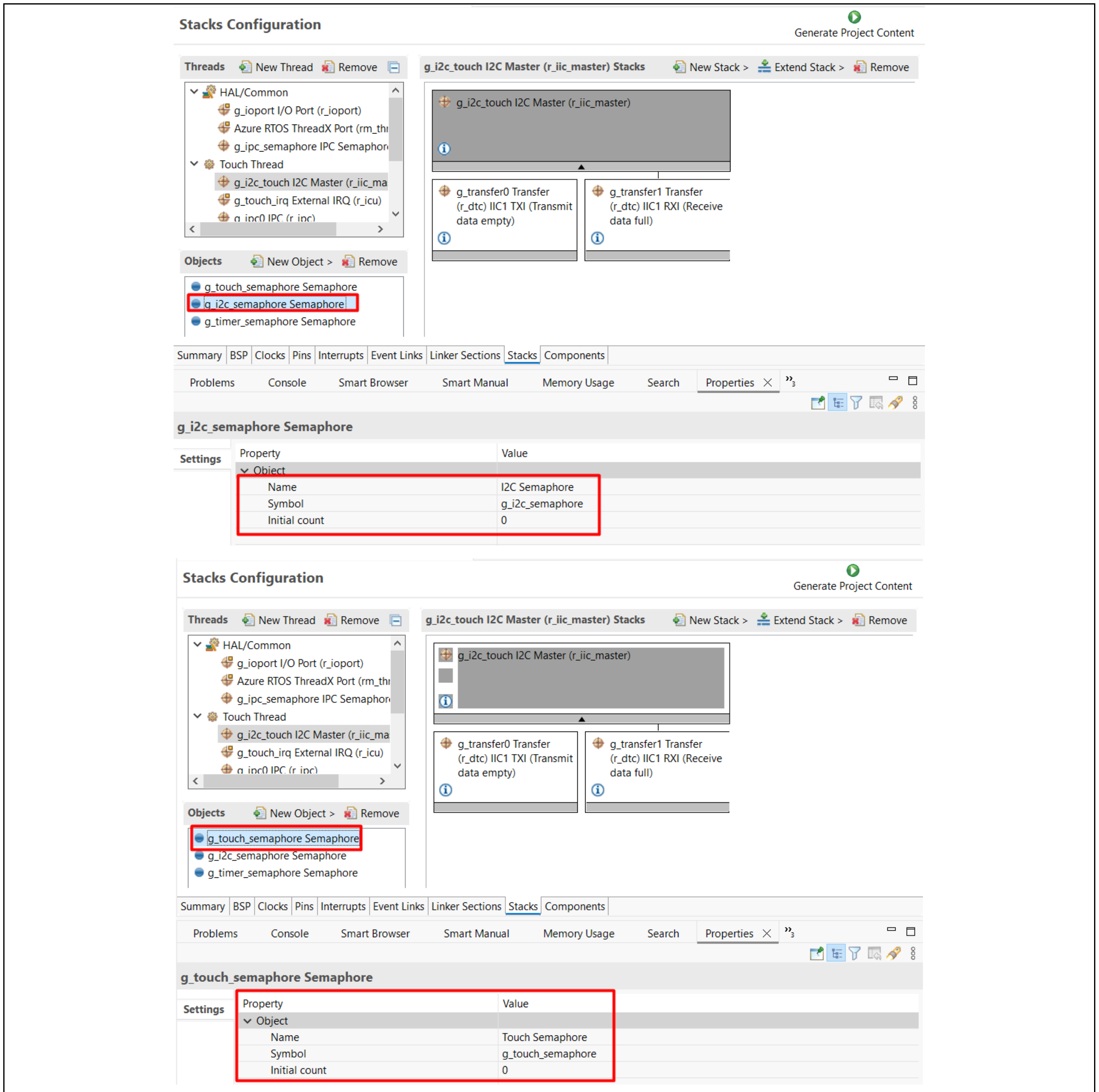


Figure 59. Configuration of g_touch_semaphore and g_i2c_semaphore on CPU1

Similarly to CPU0, IPC0 is also configured to exchange data between the two cores.

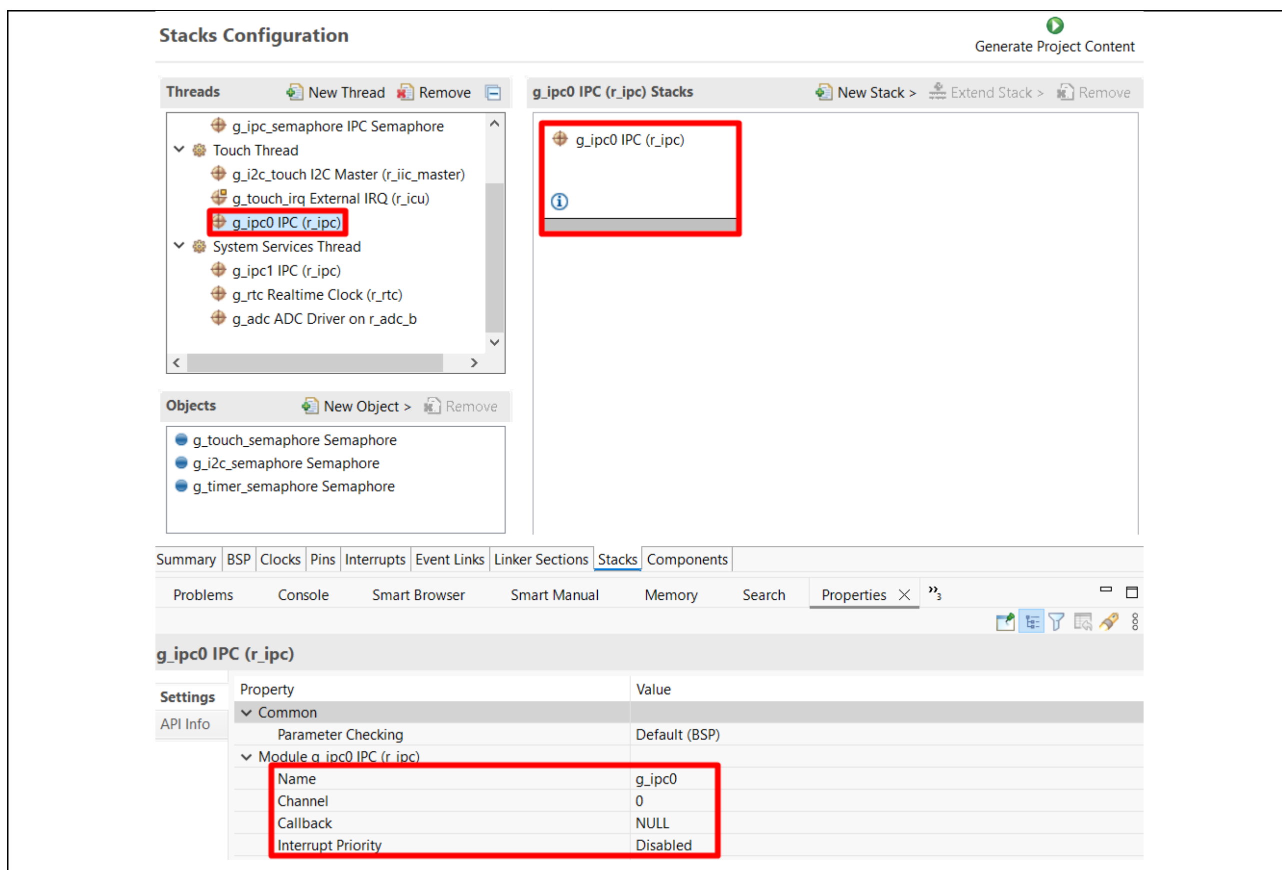


Figure 60. IPC Channel 0 configuration on CPU1

6.3.2 Code Highlights

The following code highlights show how CPU1 reads touch input and transmits coordinates to CPU0, ensuring timely and accurate data delivery.

```
while (1)
{
    /* Wait for touch interrupt */
    status = tx_semaphore_get(&g_touch_semaphore, TX_WAIT_FOREVER);
    if (TX_SUCCESS != status)
    {
        APP_ERR_TRAP(FSP_ERR_ASSERTION);
    }

    /* Read and encode touch data for IPC transmission */
    ft5x06_payload_get(&touch_data);
    uint32_t touch_message = 0;
    if (1 == touch_data.num_points)
    {
        touch_message = (uint32_t)touch_data.point[0].x |
                       ((uint32_t)touch_data.point[0].y << 16);
    }
    else
    {
        touch_message = 0xFFFFFFFF;
    }

    /* Send touch data to CPU0 */
    R_IPC_MessageSend(&g_ipc0_ctrl, touch_message);

    tx_thread_sleep(1);
}
```

Figure 61. Sending touch event to CPU0

7. Dual-Core RTC/ADC Processing

7.1 Overview

In this project, both the Real-Time Clock (RTC) and Analog-to-Digital Converter (ADC) are handled in a dual-core manner. CPU1 is responsible for acquiring raw data (such as time updates from the RTC or sampled values from the ADC), while CPU0 processes this information and updates the application logic accordingly. This section explains the configuration of RTC and ADC in a dual core environment and highlights the main implementation details.

7.2 CPU0 Processing

7.2.1 Key Configuration

The following highlights illustrate the key configuration steps for handling RTC and ADC events on CPU0. Similar to the touch input flow, IPC1 and g_rtc_semaphore are configured to notify the handler thread when data is sent from CPU1.

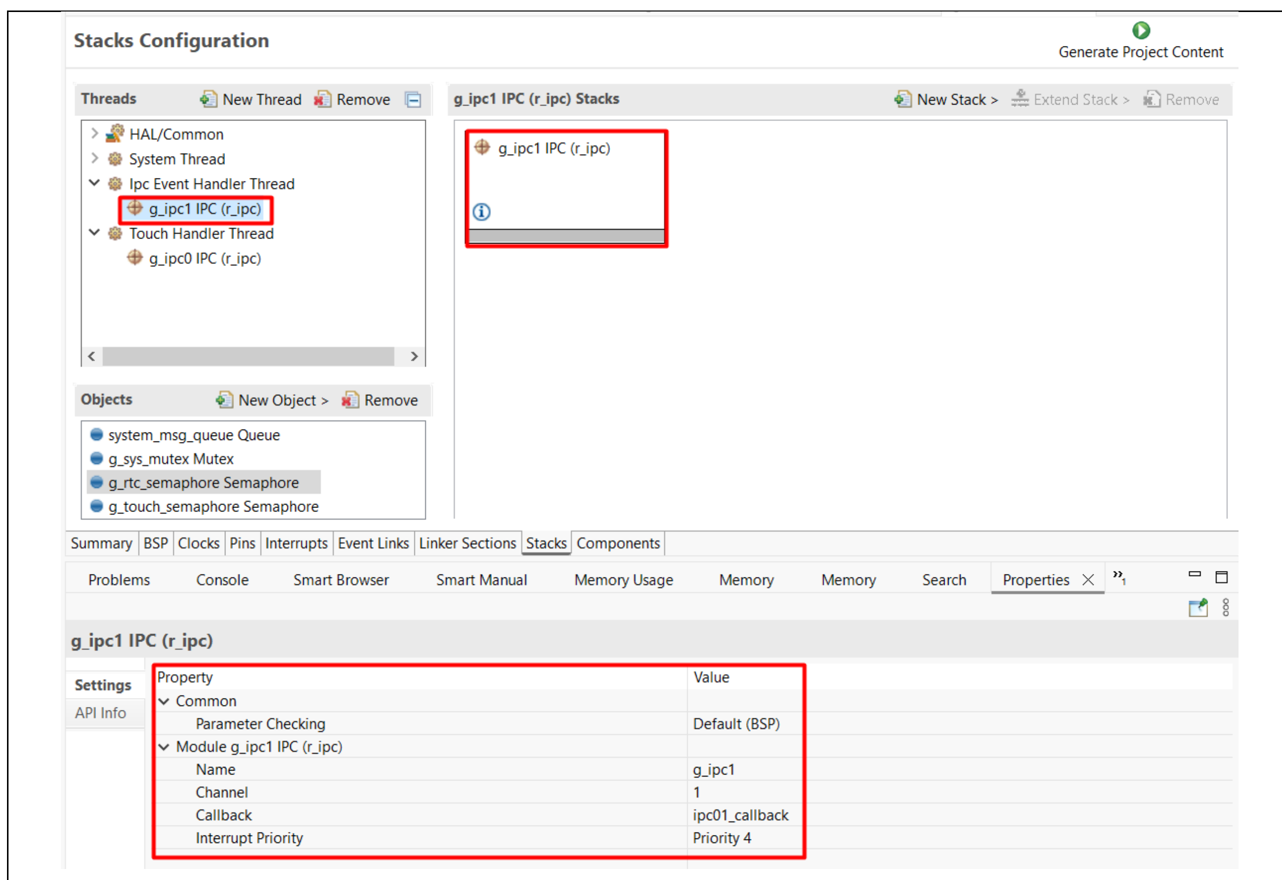


Figure 62. IPC Channel 1 configuration on CPU0

Figure 63. g_rtc_semaphore configuration on CPU0

To enable inter-core data exchange for RTC and temperature values, a shared memory region is configured. This memory space allows CPU1 to store the acquired data and CPU0 to access it for further processing and GUI updates.

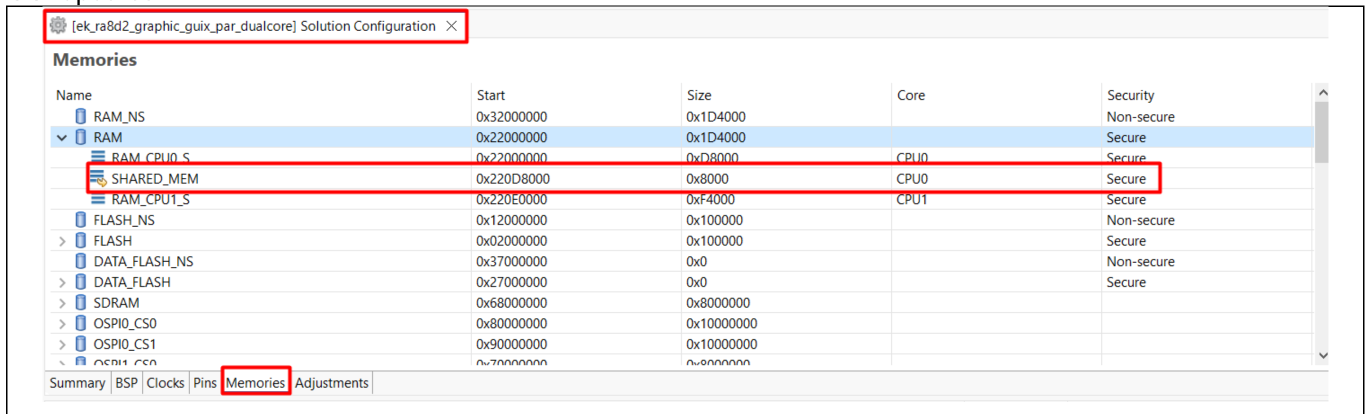


Figure 64. Shared memory configuration

7.2.2 Code Highlights

Upon receiving the time and temperature update event from CPU1, ipc01_callback retrieves the updated values from shared memory and signals ipc_event_handler_thread for display update.

```
void ipc01_callback(ipc_callback_args_t *p_args)
{
    switch (p_args->event)
    {
        case IPC_EVENT_IRQ0:
            /* Release semaphore when CPU1 acknowledges */
            R_BSP_IpcSemaphoreGive(&g_ipc_semaphore);
            break;
        case IPC_EVENT_IRQ1:
            /* Copy sensor data and wake handler thread */
            g_gui_state.time = share_memory->current_time;
            g_gui_state.temp_c = ADCTEMP_AS_C(share_memory->temperature);
            tx_semaphore_put(&g_rtc_semaphore);
            break;
        default:
            break;
    }
}
```

Figure 65. ipc01_callback function

```
void ipc_event_handler_thread_entry(void)
{
    fsp_err_t err = FSP_SUCCESS;
    UINT status   = 0;

    /* Open IPC1 channel to CPU1 for sensor data */
    err = R_IPC_Open(&g_ipc1_ctrl, &g_ipc1_cfg);
    if (FSP_SUCCESS != err)
    {
        APP_ERR_TRAP(FSP_ERR_ASSERTION);
    }

    while (1)
    {
        /* Wait for sensor data from CPU1 */
        status = tx_semaphore_get(&g_rtc_semaphore, TX_WAIT_FOREVER);
        if (TX_SUCCESS != status)
        {
            APP_ERR_TRAP(FSP_ERR_ASSERTION);
        }

        /* Update GUI with time and temperature */
        tx_mutex_get(&g_sys_mutex, TX_WAIT_FOREVER);
        send_hmi_message(GXEVENT_MSG_TIME_UPDATE);
        tx_mutex_put(&g_sys_mutex);
        send_hmi_message(GXEVENT_MSG_UPDATE_TEMPERATURE);

        /* Acknowledge CPU1 */
        R_IPC_EventGenerate(&g_ipc1_ctrl, IPC_GENERATE_EVENT_IRQ1);

        tx_thread_sleep(1);
    }
}
```

Figure 66. ipc_event_handler_thread code

In addition, CPU0 also receives user requests for time adjustment and forwards them to CPU1 for updating RTC.

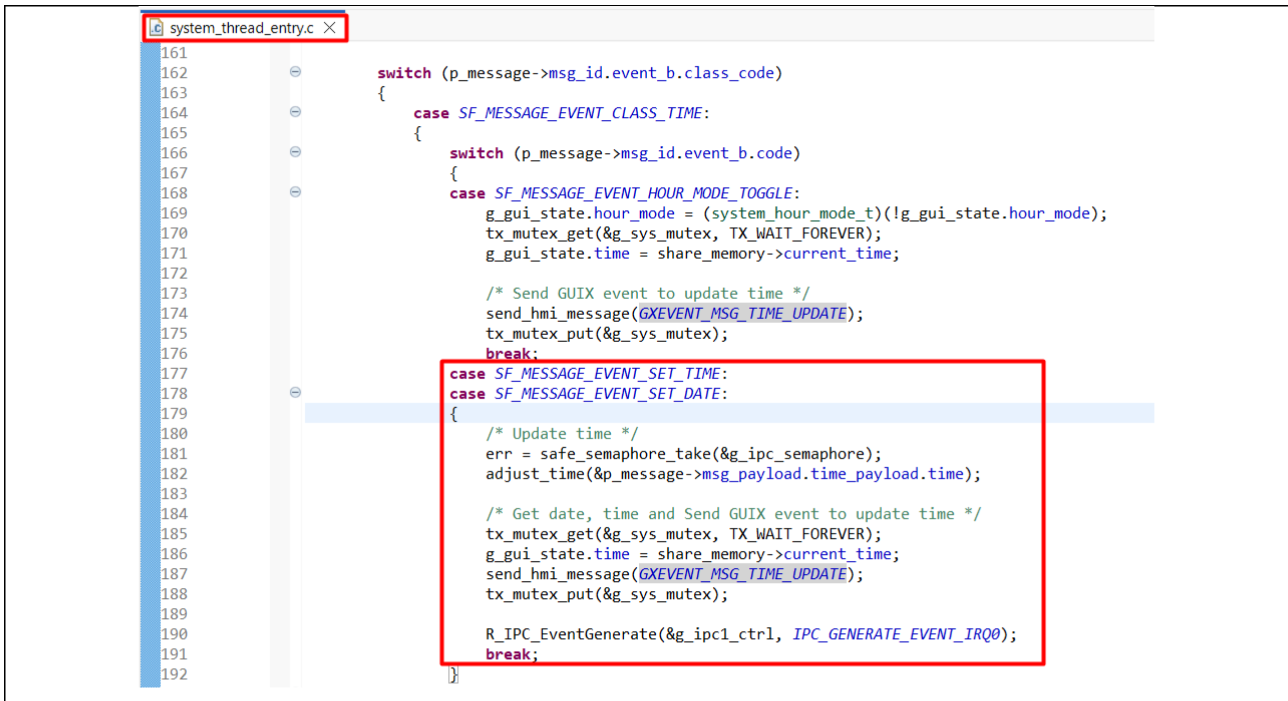


Figure 67. Event handling for SET_TIME/SET_DATE in CPU0

7.3 CPU1 Processing

7.3.1 Key Configuration

The following highlights illustrate the key configuration steps for enabling RTC and ADC acquisition on CPU1.

Setting the **Realtime Clock Driver(r_rtc)** Properties

Stacks Configuration

Generate Project Content

Threads | New Thread | Remove

g_touch_irq External IRQ (r_icu)
g_ipc0 IPC (r_ipc)
System Services Thread
g_ipc1 IPC (r_ipc)
g_rtc Realtime Clock (r_rtc)
g_adc ADC Driver on r_adc_b

g_rtc Realtime Clock (r_rtc) Stacks | New Stack > | Extend Stack > | Remove

g_rtc Realtime Clock (r_rtc)

Objects | New Object > | Remove

g_touch_semaphore Semaphore
g_i2c_semaphore Semaphore

Summary | BSP | Clocks | Pins | Interrupts | Event Links | Linker Sections | Stacks | Components

Problems | Console | Smart Browser | Smart Manual | Memory Usage | Memory | Memory | Search | Debug | Properties

g_rtc Realtime Clock (r_rtc)

Settings
API Info

Property	Value
Common	
Parameter Checking	Default (BSP)
Set Source Clock in Open	Enabled
Module g_rtc Realtime Clock (r_rtc)	
Name	g_rtc
Clock Source	LOCO
Frequency Comparison Value (LOCO)	255
Automatic Adjustment Mode	Enabled
Automatic Adjustment Period	10 Seconds
Adjustment Type (Plus-Minus)	NONE
Error Adjustment Value	0
Callback	time_update_callback
Alarm Interrupt Priority	Disabled
Period Interrupt Priority	Priority 9
Carry Interrupt Priority	Priority 12
Pins	

Figure 68. Setting Realtime Clock Driver Properties

Setting the **ADC Driver (g_adc)** Properties

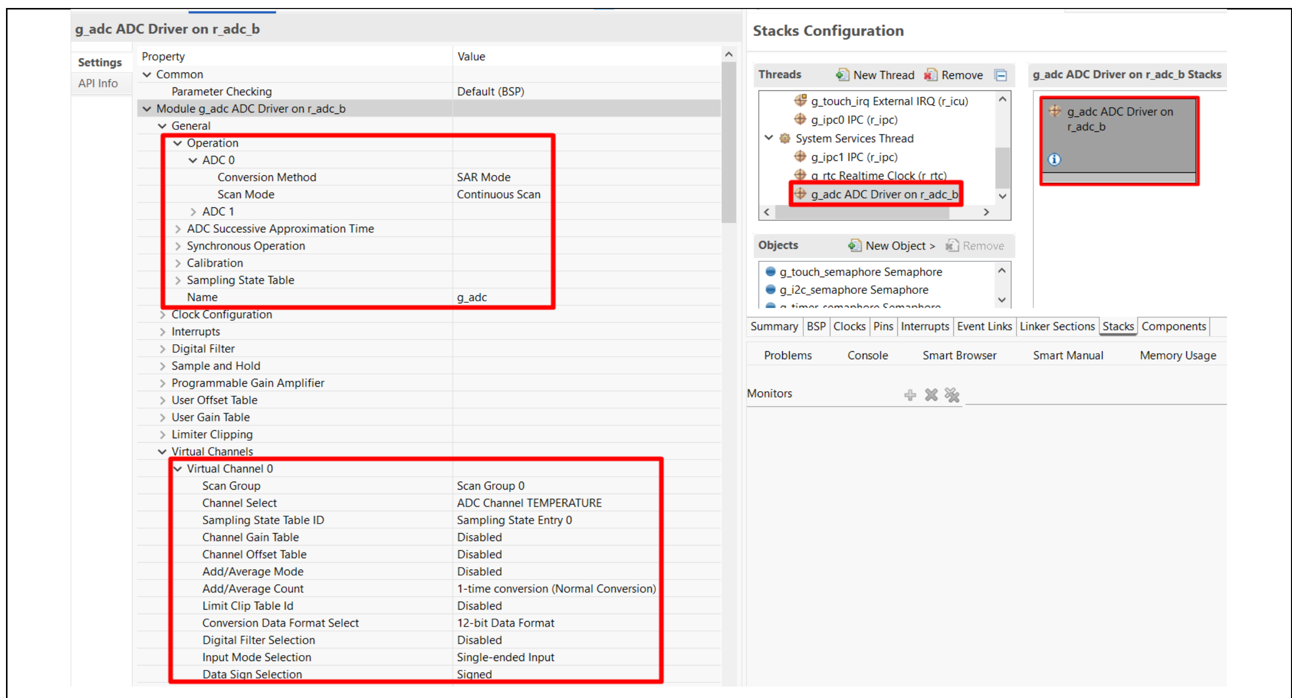


Figure 69. Setting the ADC Driver Properties

IPC1 and g_timer_semaphore are configured similarly to CPU0.

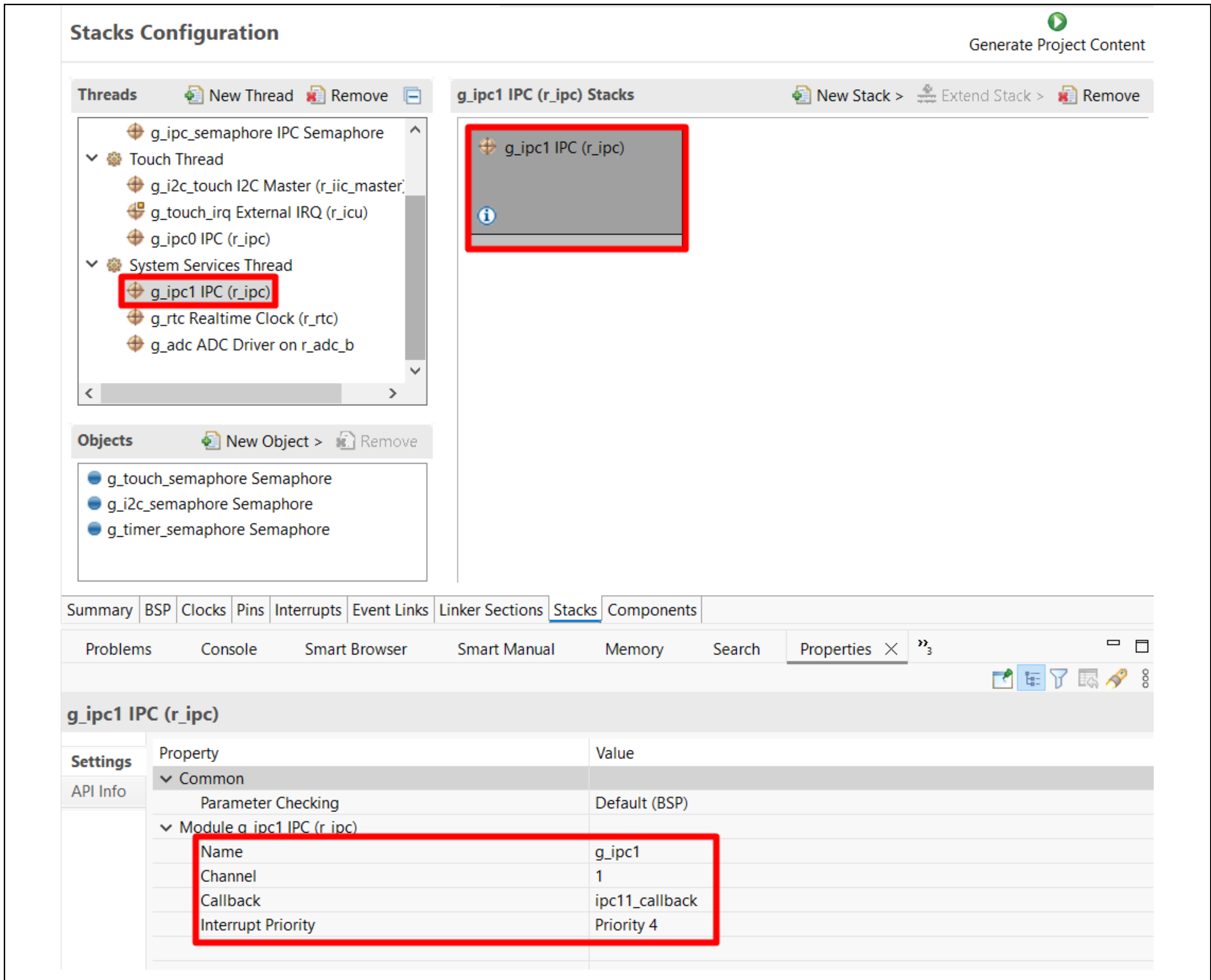


Figure 70. IPC Channel 1 configuration on CPU1

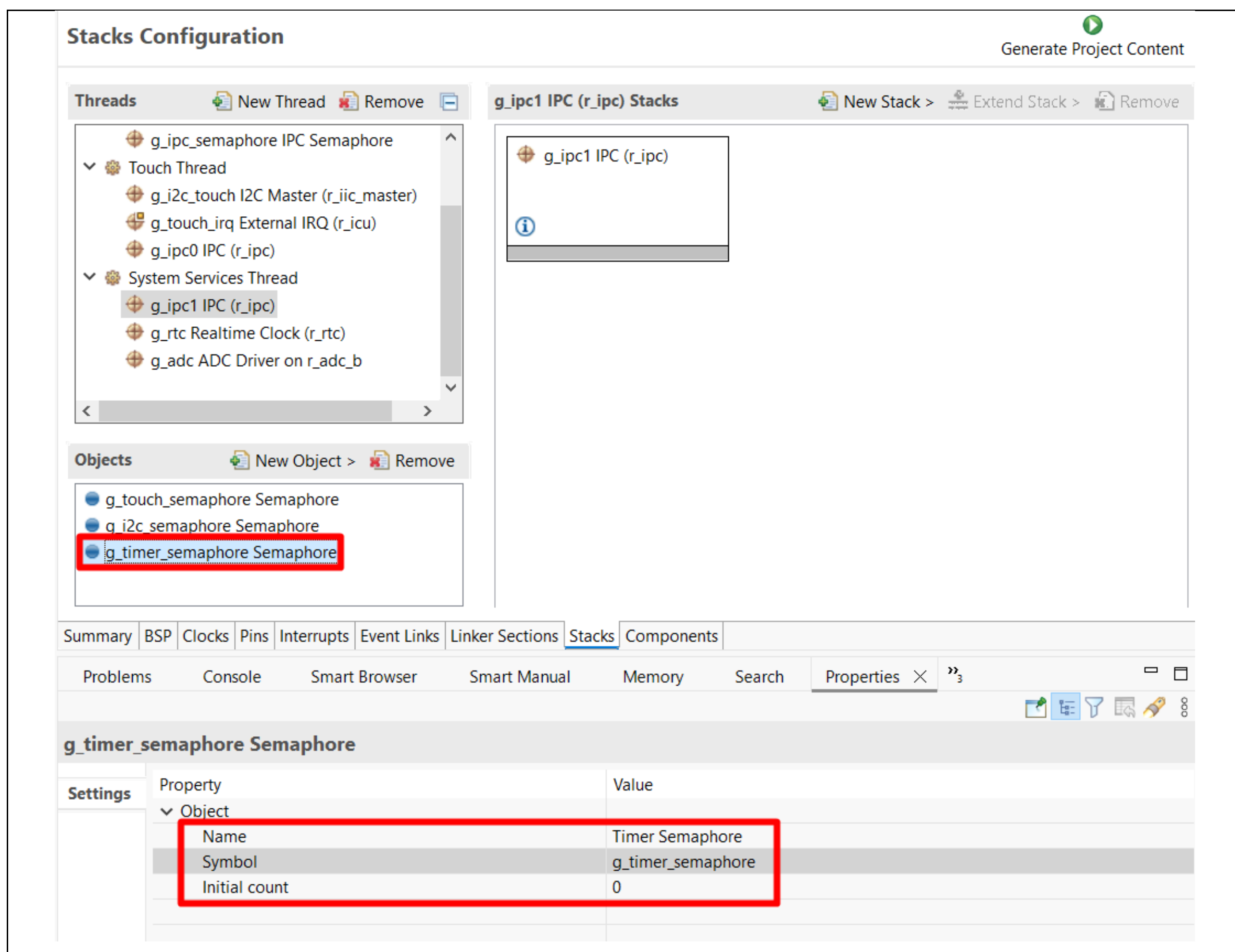


Figure 71. g_timer_semaphore configuration on CPU1

7.3.2 Code Highlights

CPU1 is responsible for updating the RTC and temperature every second, then writing the values to shared memory for CPU0 to display. The code below shows how data is read from the RTC and ADC and stored in shared memory.

```
while (1)
{
    /* Wait for RTC 1-second interrupt */
    status = tx_semaphore_get(&g_timer_semaphore, TX_WAIT_FOREVER);
    if (TX_SUCCESS != status)
    {
        APP_ERR_TRAP(FSP_ERR_ASSERTION);
    }

    /* Update shared memory with RTC time and temperature data */
    R_RTC_CalendarTimeGet(&g_rtc_ctrl, &time);
    safe_semaphore_take(&g_ipc_semaphore);
    memcpy( (void*)&share_memory.current_time, &time, sizeof(rtc_time_t));
    err = R_ADC_B_Read(&g_adc_ctrl, ADC_CHANNEL_TEMPERATURE, &adc_temp_data);
    if (FSP_SUCCESS != err)
    {
        APP_ERR_TRAP(err);
    }
    share_memory.temperature = adc_temp_data;

    /* Signal CPU0 that sensor data is ready */
    R_IPC_EventGenerate(&g_ipc1_ctrl, IPC_GENERATE_EVENT_IRQ1);

    tx_thread_sleep(1);
}
```

Figure 72. CPU1 RTC/ADC update to shared memory

Additionally, ipc11_callback is invoked upon a user request to set the RTC time.

```
void ipc11_callback(ipc_callback_args_t *p_args)
{
    switch (p_args->event)
    {
        case IPC_EVENT_IRQ0:
            /* Update RTC time and acknowledge */
            R_RTC_CalendarTimeSet(&g_rtc_ctrl, (rtc_time_t*)&share_memory.current_time);
            R_IPC_EventGenerate(&g_ipc1_ctrl, IPC_GENERATE_EVENT_IRQ0);
            break;
        case IPC_EVENT_IRQ1:
            /* Release shared memory semaphore */
            R_BSP_IpcSemaphoreGive(&g_ipc_semaphore);
            break;
        default:
            break;
    }
}
```

Figure 73. ipc11_callback function

Note: For Sections 6 and 7, to better understand dual-core data communication (via IPC or shared memory), refer to the application note Developing with RA8 Dual-Core MCU (R01AN7881).

8. Running the Thermostat Application

This section explains how to get the Graphics Thermostat Application for the EK-RA8D2 up and running, including steps for setting up the hardware and building and running the project in e^2 studio.

8.1 Hardware Setup

Connect J1 on the included Parallel Graphics Expansion Board to J1 on the EK-RA8D2. Use the included screw, if provided, to secure the display connection.

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

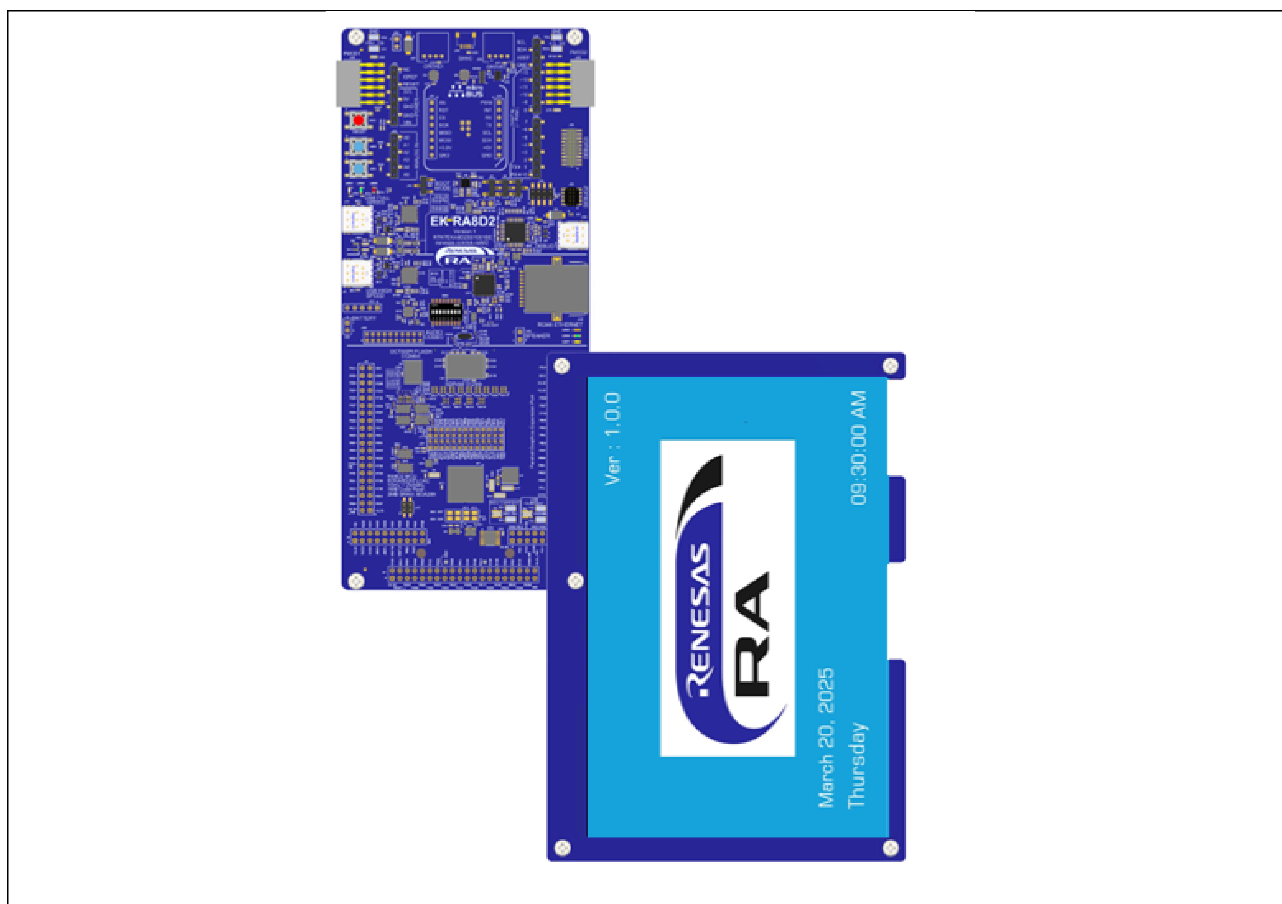


Figure 74. Parallel Graphics Expansion Board 1 connected to EK-RA8D2 Board

On the EK-RA8D2, there is a set of configuration switches (labeled SW4) that selects the operational peripheral pins on the board based on circuit groups. Refer to the board schematic for a complete understanding of the peripheral circuit groups.

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

Table 2. Required SW1 configurations for running the Thermostat Application Project

Location	Function	Setting	Function Restrictions
SW4-1	PMOD 1	OFF	-
SW4-2	PMOD 1	OFF	-
SW4-3	Octo-SPI Active	**OFF**	Arduino, mikroBUS and Pmod 1 (SPI, UART)
SW4-4	Arduino and mikroBUS Connectors Inactive	OFF	Conflict Octo-SPI
SW4-5	I2C Active	OFF	-
SW4-6	MIPI Display and Parallel Camera Active	**OFF**	Conflict I3C
SW4-7	Toggles USBFS between Host and Device mode	OFF	-
SW4-8	Toggles USBHS between Host and Device mode.	OFF	-

8.2 Import 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_guix_par_dual_core.zip* is downloaded onto your host PC.

2. Follow the connection steps from Section 5.1.
3. Open an instance of e² studio IDE.
4. In the workspace launcher, either create or browse to the workspace location of your choice and select it.
5. In e² studio, navigate to **File** > **Import**.
6. In the Import dialog box select **General** > **Existing Projects into Workspace.**
7. **Select root directory** *ek_ra8d2_graphic_guix_par_dual_core*.
8. Make sure the option **Copy projects into workspace** is selected. Click **Finish**.
9. Right-click the solution project "**ek_ra8d2_graphic_guix_par_dualcore**" and select Build Project. This process may take some time as it sequentially builds both subprojects: "**ek_ra8d2_graphic_guix_par_cpu0**" and "**ek_ra8d2_graphic_guix_par_cpu1**".

8.3 Downloading and Execute to the EK-RA8D2 Kit

To connect and run the code, follow these steps:

1. Connect your PC to the USB port DEBUG using a USB cable.
2. Go to **Run > Renesas Debug Tools > Renesas Device Partition Manager.**
3. Select **Initialize device**, choose **J-Link** as the connection method, and click **Run**.

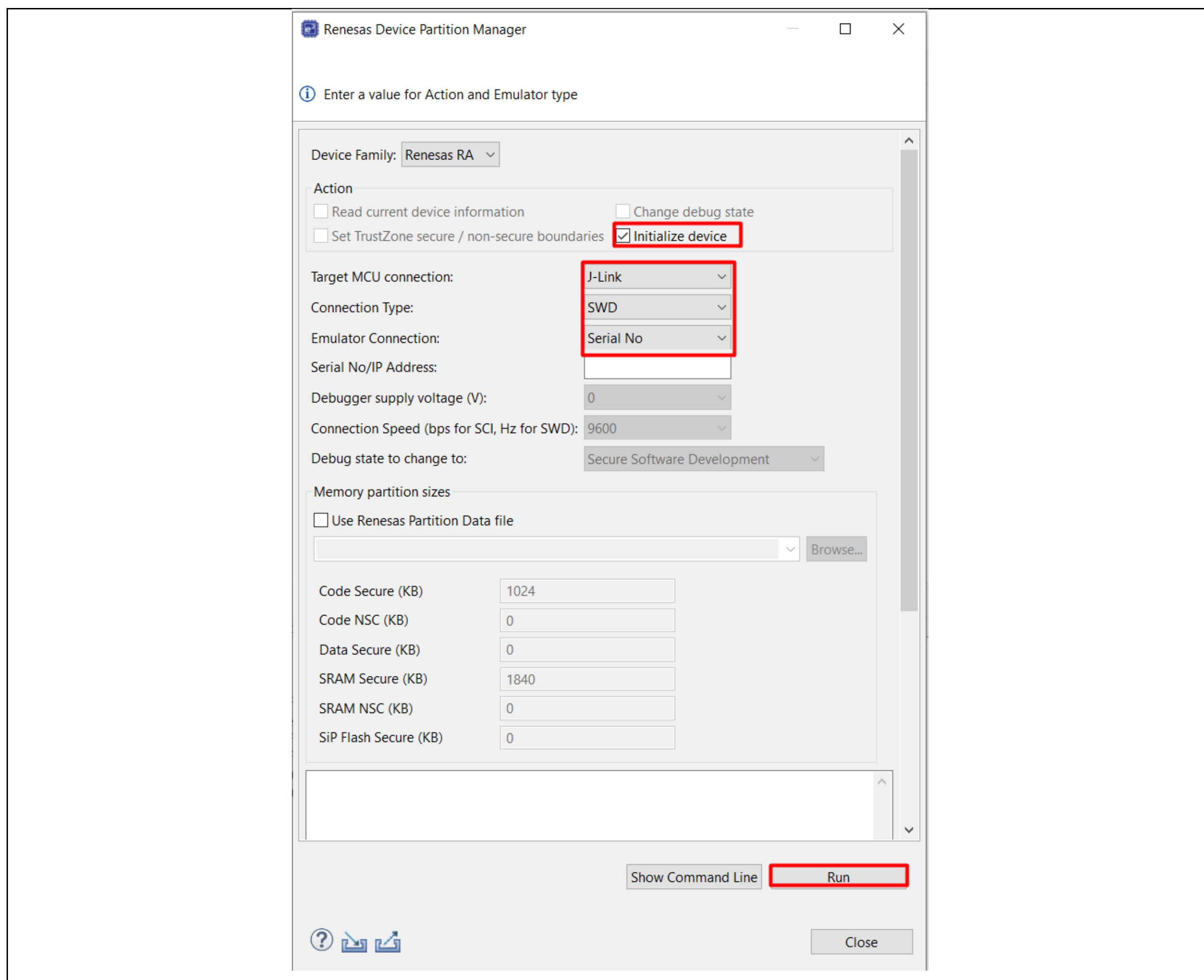


Figure 75. Initialize RA8D2 using Renesas Device Partition Manager

4. Power cycle the board, Go to **Run > Debug Configurations.**

5. Expand **Launch Group** Click **ek_ra8d2_graphic_guix_par_cpu1 Debug_Multicore Launch Group > Debug**.
6. Click **Switch** to the **Debug perspective** when prompted by the e² studio.
7. Click **Resume** > **Resume**.
8. You will see the transition from Splash screen to Main Page screen in about 3 seconds.

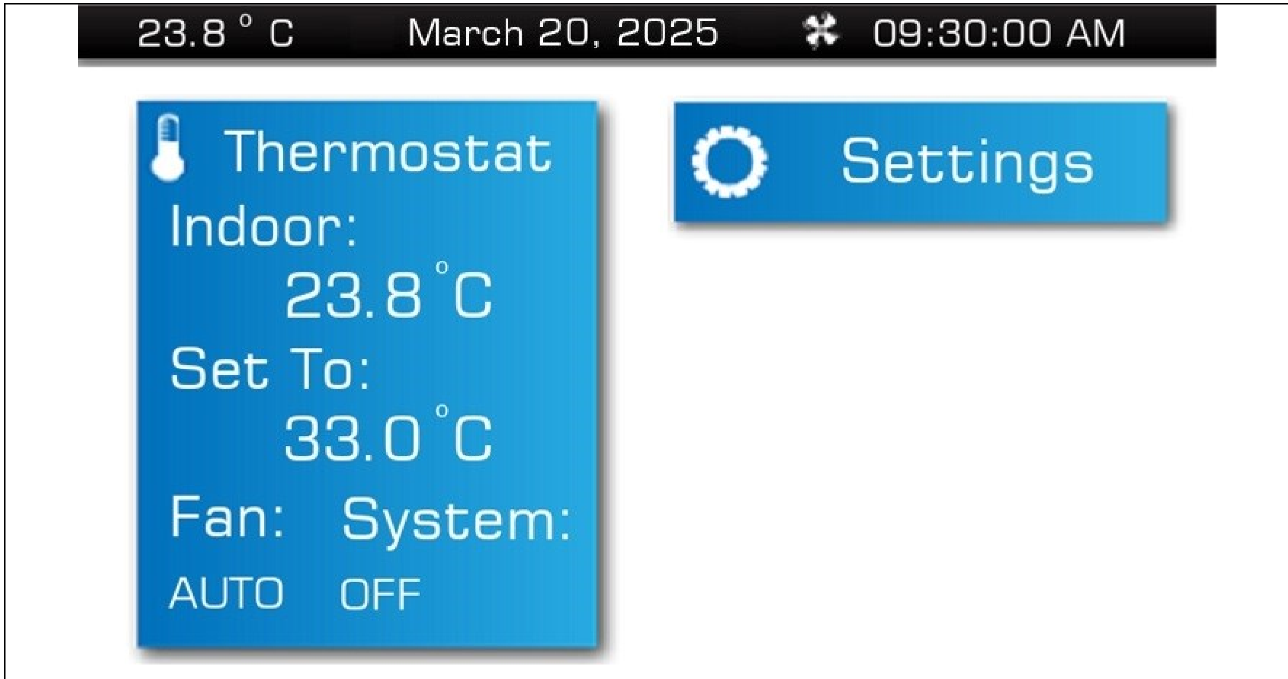


Figure 76. Main Page Screen

9. Graphics Tradeoffs on the RA8D2

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 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.

9.1 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 graphics subsystem to send pixel data via parallel RGB to the external LCD. While MIPI DSI communication is also supported, the connection details are outside the scope of this 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 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.

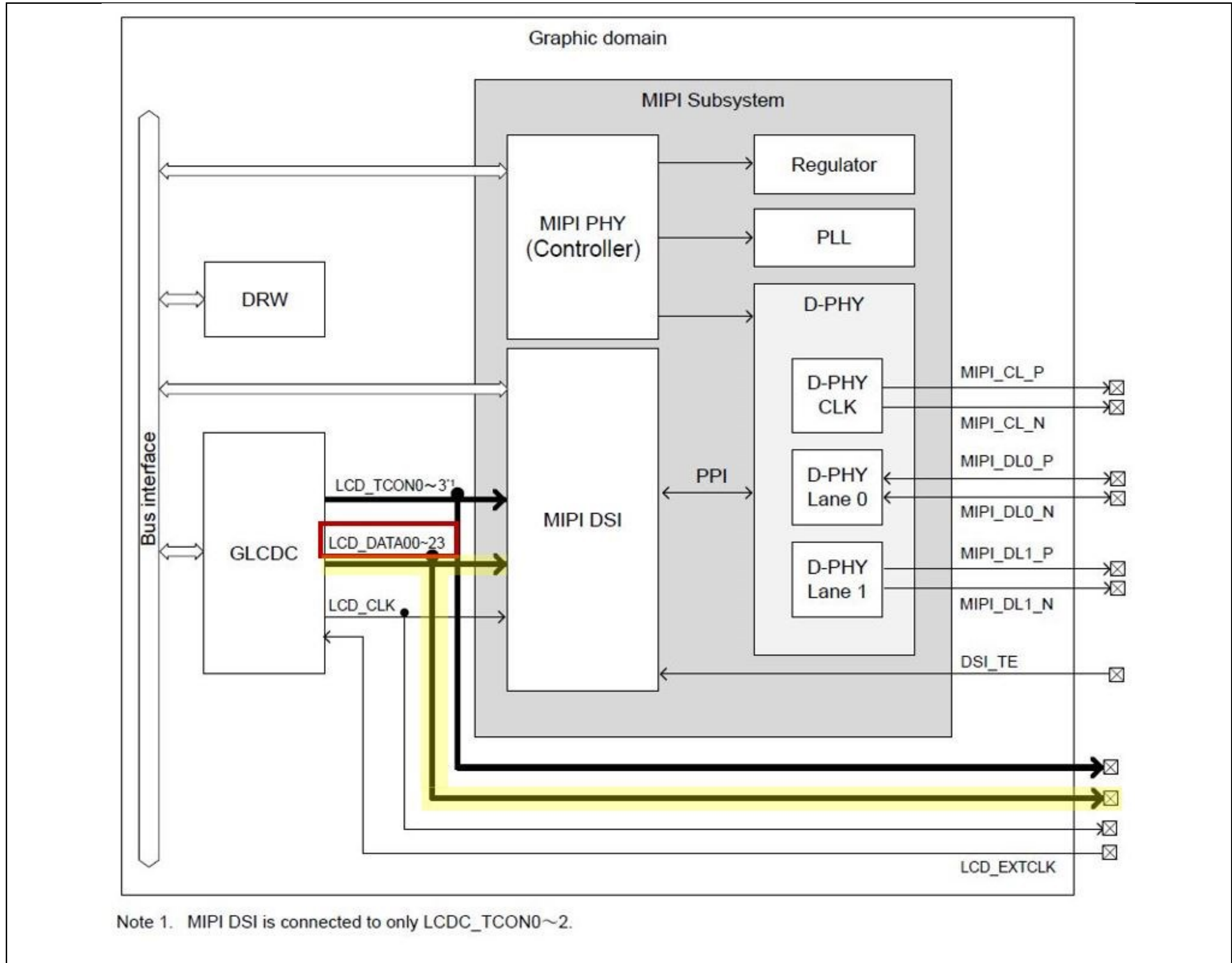


Figure 77. The GLCDC outputs parallel RGB data, routing to MCU output pins and to MIPI DSI input

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:

9.1.1 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.

9.1.2 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.

9.1.3 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.

9.1.4 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.

9.1.5 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.

9.2 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.

9.2.1 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 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. The EK-RA8D2 kit is equipped with a parallel landscape display that operates at a resolution of 1024×600 pixels. The GLCDC must be configured to match this display specification.

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.

9.2.2 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)

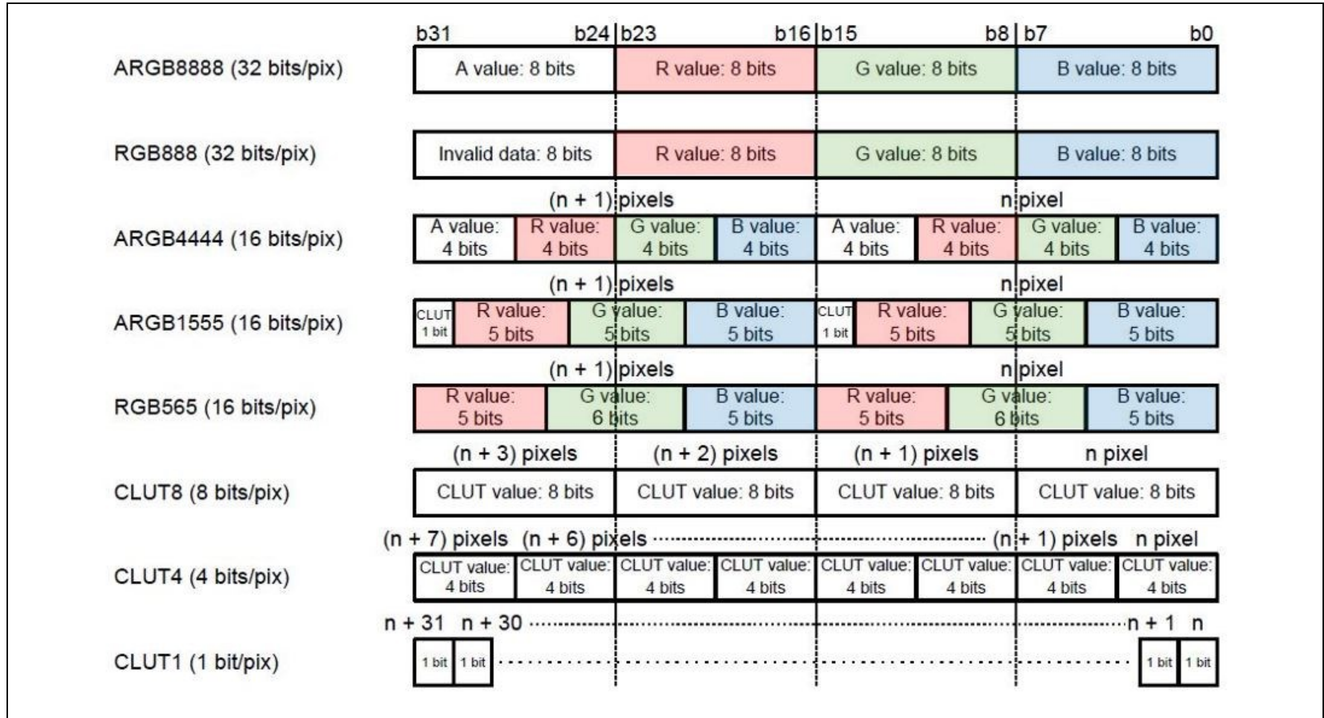


Figure 78. Pixel data formats of the RA8D2 GLCDC

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

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

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

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

9.2.3 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 GUIX. 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.

9.2.4 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 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:

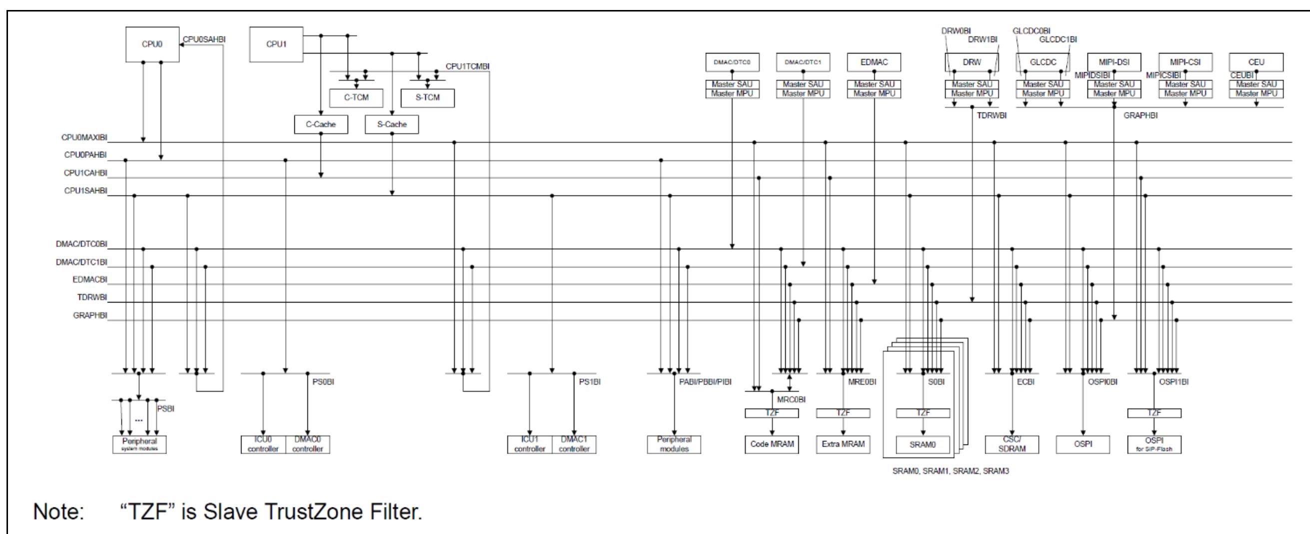


Figure 79. Bus map of the RA8D2 indicating commonly used modules for graphics applications

In the RA8D2 MCU group, there are devices with a 16-bit SDRAM bus and devices with a 32-bit SDRAM bus. The EK-RA8D2 board is based on the 32-bit variant. 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.

9.2.5 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 2 MB (256 KB of CM85 TCM RAM, 128 KB of CM33 TCM RAM, 1664 KB of user SRAM).

Since there is no hardware JPEG decoder on RA8D2 MCUs, if JPEG images are used, software decoding in the SRAM region is recommended for optimal performance. In general, the SRAM region is ideal for drawing framebuffer(s) and performing other intensive processes because it reduces the need to access slower external memory during rendering.

When using SRAM to render the framebuffer(s), the color format and resolution are directly related to the required SRAM size. For example, the following calculation compares the SRAM size required for a double framebuffer between 32-bit RGB888 and 16-bit RGB565 at a 1024×600-pixel resolution:

2 (framebuffers) × 1024×600 (pixels) × 32 bpp / 8 = 4.88 MB

2 (framebuffers) × 1024×600 (pixels) × 16 bpp / 8 = 2.44 MB

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

9.3 RA8D2 Memory Options

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

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

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

The RA8D2 supports the following memory options:

- 1MB MRAM
- 2 MB SRAM (256 KB of CM85 TCM RAM, 128 KB CM33 TCM RAM, 1664 KB of user SRAM)
- 512MB external Octo-SPI Flash.
- 128MB external SDRAM with 16/32-bit external memory bus interface.

9.3.1 EK-RA8D2 Memory Devices

The EK-RA8D2 has the following memory devices:

- 1MB MRAM
- 1664 KB internal SRAM
- 256 KB of CM85 TCM RAM.
- 128 KB CM33 TCM RAM
- 64MB external Octo-SPI Flash.
- 64MB external SDRAM with a 32-bit bus interface.

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

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

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

the single framebuffer scheme, but the benefit of a double buffer is achieving faster framerates and smoother visual responses.

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

Enabling the data cache can improve graphics performance, but cache maintenance must be carefully managed to ensure deterministic behavior and prevent visual artifacts.

9.4 Thermostat Application Best Case Design

The best-case design for the GUIX thermostat application depends on the system's predefined constraints and final image requirements. The design method is to balance resource tradeoffs according to priority to achieve smooth performance and responsiveness. This section provides insight into the trade-off considerations that led to the final application design.

It was decided that the project would be built on a dual-core RA8D2 implementation using Azure RTOS (ThreadX + GUIX and supporting FSP components). The graphical assets of the thermostat application, including full-screen weather backgrounds, icons, logos, fonts, and interactive controls, were defined in advance using GUIX Studio to lock visual scope and memory footprint early.

The display resolution was fixed by the 1024×600 parallel RGB panel specification; remaining visual decisions (color format and update behavior) were shaped by available memory bandwidth and the need to reserve internal MRAM for code/RTOS rather than large framebuffers. The GLCDC supports both RGB888 and RGB565; RGB888 was chosen to preserve gradient quality and icon clarity despite its higher buffer footprint.

To support high-resolution images in RGB888 format with smooth transitions, two framebuffers (double buffering) were placed in external SDRAM, providing sufficient capacity while reserving internal MRAM for time-critical code, RTOS objects, and small shared structures. In addition, the data cache was enabled to accelerate memory access and improve overall rendering performance. External Octal-SPI (OSPI) flash is used to store the GUIX constant graphical assets, significantly increasing available storage for high-quality graphics and larger image sets without exceeding on-chip limits. This configuration allows the retention of full-resolution images and multiple GUI objects, while maintaining smooth, tear-free rendering performance for the GUIX-based thermostat application.

10. References

- RA8D2 Group User's Manual: Hardware (R01UH1064).
- EK-RA8D2 Quick Start Guide (R20QS0077).
- Developing with RA8 Dual Core MCU (R01AN7881).
- Azure RTOS GUIX and GUIX Studio v6.4.0.0.
- EK-RA8D2-v1.0 Schematics.

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

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Dec.19.25	—	Initial release

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

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

1. Precaution against Electrostatic Discharge (ESD)

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

2. Processing at power-on

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

3. Input of signal during power-off state

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

4. Handling of unused pins

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

5. Clock signals

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

6. Voltage application waveform at input pin

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

7. Prohibition of access to reserved addresses

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

8. Differences between products

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

Notice

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

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

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

(Rev.5.0-1 October 2020)

Corporate Headquarters

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

www.renesas.com

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/.

© 2025 Renesas Electronics Corporation. All rights reserved.