Renesas RA Family

# Driving Display Over SPI Bus on the RA4E1

## Introduction

This application note explains how to enable graphics applications on **any RA MCU** device by interfacing with an external display over the SPI bus.

While the information presented in this document is applicable to **all non-graphics-focused RA devices**, the concepts are demonstrated through a series of graphics applications that run on the FPB-RA4E1 and a selected external display.

The application note and its accompanying application projects explain two primary concepts:

1. **How to drive an external SPI display** with the RA4E1. The first half of the document explains how to understand the SPI interface options, identify compatible external SPI displays, configure the SPI bus, develop custom SPI display drivers built with the Flexible Software Package (FSP), and enable DTC/DMAC support on the SPI bus. The application project *ColorBars_SCI_SPI_ra4e1* demonstrates the concepts outlined in Sections 1-5.

2. **How to custom-integrate graphics libraries into any RA FSP project**. The second half of the document explains the procedure for each library included with the FSP: GUIX and emWin. It covers how to add the graphics library and supporting files to an FPB-RA4E1 e$^2$ studio project, match the graphics project and e$^2$ studio project settings, and design middleware that integrates the custom display drivers into the GUIX/emWin application stack. The application projects *HelloWorld_GUIX_SCI_SPI_ra4e1* and *HelloWorld_emWin_SCI_SPI_ra4e1* demonstrate the concepts outlined in Sections 6-8.

## Target Device

- RA4E1

This method can be used with any RA device and an external display that supports the SPI interface.

## Requirements

- To build and run the application projects accompanying this document, you will need the following:

## Evaluation Hardware

- FPB-RA4E1, available on Renesas website at https://www.renesas.com/fpb-ra4e1
- NewHave Display 1.5in OLED with Arduino shield, available on Newhaven website at https://newhavendisplay.com/1-5-inch-color-graphic-oled-display-arduino-shield/

## Evaluation Software

- e$^2$ studio v2025-04.1
- FSP v6.0.0
- LLVM for ARM v18.1.3

The FSP, e$^2$ studio, and LLVM compiler are bundled in a downloadable platform installer available on Renesas' website at: renesas.com/ra/fsp.

## Other Software

- GUIX Studio v6.4.0.0
- AppWizard v6.48.0

## Prerequisites & Intended Audience

This application note assumes you have some experience with the Renesas e$^2$ studio ISDE and RA Family Flexible Software Package (FSP). Before you perform the procedures in this application note, follow the stepsin the *FSP User Manual* to build and run the Blinky project. Doing so enables you to become familiar with e$^2$ studio and the FSP and validates that the debug connection to your board functions properly.

Additionally, this application note assumes that you have some theoretical background on graphical applications with LCD or OLED SPI controllers.

## Contents

# 1. Using FPB-RA4E1 for a Graphics Application

The Fast Prototyping Board (FPB) for the RA4E1 MCU Group is an evaluation board that uses the 100MHz Arm Cortex-M33 core and features 512 KB of code flash and 128 KB of SRAM.

For more detailed information on getting started with the FPB-RA4E1, refer to the FPB-RA4E1 Quick Start Guide and the RA4E1 Group User's Manual: Hardware.

The FPB-RA4E1 is suitable for light graphics applications that drive displays with resolutions such as 128x128 or 160x128 pixels at 16-bit color depth, and for graphics that consist of static images, icons, UI elements, or occasional small animations rather than full-screen video. The FPB-RA4E1 can comfortably draw and refresh these displays over the SPI bus without noticeable lag, provided full-screen updates are infrequent.

However, it is important to note that the FPB-RA4E1 has no dedicated graphics hardware and ships without a display. While it can be used for graphics, doing so requires identifying and sourcing an external display, developing custom display drivers, and manually integrating the custom display drivers with a graphics library such as emWin and GUIX. Applications with multiple images, icons, or animations may require external memory.

The FPB-RA4E1 kit differs significantly from other graphics-focused RA MCUs, such as the EK-RA6M3G and EK-RA8D1. Those evaluation kits come with dedicated LCD displays and graphics hardware options, such as GLCDC and MIPI DSI interfaces, along with 2D drawing acceleration enabled by D/AVE 2D. They have pre-integrated FSP stack support for graphical libraries such as emWin and GUIX Studio, enabling out-of-the-box operation.

This application note and the accompanying application projects explain two larger concepts for enabling graphics applications on RA MCUs:

1. **How to drive an external SPI display with the FPB-RA4E1.**
    1. **App Note Sections:** (ref) Sections 2-5
    2. **Application Project:** *ColorBars_SCI_SPI_ra4e1*
    3. **Major Topics:**
        - Understanding the SPI interface options on RA4E1 devices
        - Identifying compatible external SPI displays
        - Configuring the MCU's SPI bus to the display's interface specifications
        - Connecting the MCU and display
        - Developing custom SPI display drivers built with the FSP
        - Enabling DTC/DMAC support on the SPI bus

2. **How to perform custom integration of emWin and GUIX libraries to any RA FSP graphics project.**
    1. **App Note Sections:** Sections 4-8
    2. **Application Projects:**
        - *HelloWorld_emWin_SCI_SPI_ra4e1*
        - *HelloWorld_GUIX_SCI_SPI_ra4e1*
    3. **Major Topics:**
        - Adding graphics libraries (GUIX and emWin) to an e$^2$ studio project for a non-graphics-focused RA MCU
        - Integrating SPI display drivers into the GUIX stack
        - Integrating SPI display drivers into the emWin stack

# 2. SPI Interfaces on the RA4E1

Every RA MCU device will have at least one hardware module capable of implementing a form of SPI communication. Be aware that between MCU devices and SPI module interfaces, there are different underlying hardware implementations resulting in different communication and interface specifications. Always refer to the RA MCU's Hardware User's Manual and the FSP User's Manual for implementation details on the SPI module and device in question for operation details.

This application note first covers the SPI communication options available on the RA4E1 MCU, specifically in the context of driving an external graphics display over the SPI bus. This section details and compares the two different hardware modules on the RA4E1 that are capable of SPI communication: the Serial Peripheral Interface (SPI) and the Serial Communications Interface's Simple SPI (SCI SPI).

## 2.1 SPI on R_SPI

The SPI hardware module interface has a single channel that can provide high-speed, full-duplex, synchronous serial communications. Since the SPI module is highly configurable and has four SSL signals for the single channel, it can be used to communicate with multiple processors or peripheral devices on the same bus.

**Signals:** The SPI module uses the signals MOSI, MISO, SSL and RSPCK signals, so the module can implement either the 4-wire SPI method or the 3-wire clock-synchronous method. The interface can operate in master or slave mode.

**Format:** The data formats can be configured for different bit orders (MSB vs LSB) and 12 different bit lengths (8-16, 20, 24, or 32 bits). There are 128-bit transmission and reception double buffers that enable continuous communication.

**Clock:** The SPI interface supports 4 main clock configurations depending on the clock polarity and phase settings, which enable compatibility with a wide range of SPI devices. The clock speed can be adjusted to optimize baud rates according to the application's requirements. PCLKA provides the source clock that is divided into the RSPCK output with a divider range of 2 to 4096.

**FSP Support:** The **r_spi** software module from the Renesas FSP provides the Hardware Abstraction Layer (HAL) for interacting with the specific SPI hardware on the RA4E1.

## 2.2 SCI SPI on R_SCI_SPI

The Serial Communications Interface implements a variety of asynchronous and synchronous serial interfaces. This section is concerned only with the SCI module operating in "Simple SPI" mode, referred to as "SCI SPI".

The SCI SPI interface handles transfers among one or multiple master devices and multiple slave devices. The SCI interface has up to four channels that can operate in Simple SPI mode simultaneously, depending on the pin availability of your target application. On the RA4E1, the 4 channels are n=0,3,4,9.

**Signals:** The SCI SPI module uses the signals MOSI, MISO, and SCK, and data is transferred in synchronization with clock pulses, like clock synchronous mode. The interface can operate in master or slave mode. Full-duplex mode is possible with a shared clock signal, since the receiver and transmitter are independent in the SCI module. Note that in a single-master configuration, the slave select signal is unused. If an SS signal is required by the peripheral device's SPI specifications, you should control a GPIO pin for this functionality.

**Format:** The data format of SCI SPI is more rigid with 8-bit characters and no parity bit. Data has the option of being inverted. Both the SCI transmitter and receiver have the option of a FIFO buffer. When FIFO buffers are enabled and configured, write next data to the buffer during transmission and read previous data during reception to perform continuous communication.

**Clock:** The SCI SPI interface supports 4 main clock configurations depending on the clock polarity and phase settings, which enable compatibility with a wide range of SPI devices. In master mode, PCLKA provides the source clock that is divided into the SCK output with a divider range of 2 to 64.

**FSP Support:** The **r_sci_spi** FSP software module provides the HAL for interacting with the SCI SPI hardware on the RA4E1.

## 2.3 Key Differences Between SPI and SCI SPI

The R_SPI (Serial Peripheral Interface) and R_SCI_SPI (Serial Communication Interface with SPI mode) are two distinct interfaces available on the FPB-RA4E1 microcontroller, each offering unique functionalities. The following is a quick summary of their differences. You are strongly encouraged to also review the specifications of each interface in the RA4E1 Hardware User's Manual for full operation details.

**Table 1. RA4E1 SPI and SCI Module Comparison**

| Feature | SPI | SCI SPI |
|---|---|---|
| **Module Channel Availability** | Single channel (0) with 4 SSL pins | Up to 4 channels (n = 0, 3, 4, 9); availability depends on pin usage |
| **Bus Signals** | MOSI, MISO, SSL, and RSPCK; supports 3-wire SPI or 4-wire SPI clock synchronous operation. | MOSI, MISO, SCK in native 3-wire synchronous mode; GPIO-based SS enables custom 4-wire SPI |
| **SPI Modes Supported** | Master or slave mode; double buffering enables continuous communication; full-duplex or transmit-only selectable | Master or slave mode; buffered structure enables continuous transfer; full-duplex possible with shared clock (independent TX/RX) |
| **Data Packet Formats** | Highly configurable: MSB/LSB first, 12 packet lengths (8–16, 20, 24, 32 bits), byte swap, transmit/receive data inversion | Fixed 8-bit format with no parity; MSB/LSB first and data inversion selectable |
| **Clock Output (Master Mode)** | RSPCK sourced from PCLKA; clock divider range 2–4096; selectable polarity and phase | SCK sourced from PCLKA; clock divider range 2–64; selectable polarity and phase |
| **Interrupts Available** | Receive buffer full, transmit buffer empty, SPI error (mode fault, overrun, parity), SPI idle, transmission complete | Receive data full, transmit data empty, SPI error (overrun only), transmission end |

Overall, the SPI interface is more configurable and supports a wider range of baud rates than SCI SPI. Consequently, the SPI module is most suitable for applications that control 1-4 SPI peripherals with particular bus specifications, or for controlling devices with faster baud rate requirements, such as memory devices. The SCI SPI interface is ideal for applications where you need to control 1-4 SPI peripherals that use 8-bit data packets in a 3 or 4-wire setup.

# 3. Driving External Display with the SPI Bus

This section explains the process for interfacing an external SPI display to the FPB-RA4E1's SPI bus. The topics are explained using the accompanying graphic application project, *ColorBars_SCI_SPI_ra4e1*, which uses a particular external display model. The steps can be generalized to interface any external SPI peripheral to the RA4E1 SPI bus.

Topics include:

- Brief comparison of external display types, OLEDs vs LCDs.
- Overview of the external display used in accompanying application projects.
- Method for choosing between the SPI and SCI SPI interfaces on the RA4E1.
- Configuring the RA4E1 SPI bus to match the external display's SPI interface specifications.
- Mapping the RA4E1 SPI pins to the external display's pins.
- Controlling the external display by sending commands and data.
- Creating an initialization sequence for the external display using a series of commands and data.
- Sending a framebuffer of pixel data to the display.

## 3.1 Background: OLED vs LCD

The most common SPI displays for simple graphics applications are OLEDs (Organic Light Emitting Diode) and LCDs (Liquid Crystal Display). This section provides a brief comparison to help you select which is more appropriate for your application.

The main difference between these display types is how the images are produced. OLEDs generate images by applying electricity to organic materials inside the display. LCDs contain liquid crystals that will produce an image when light is passed through the display. By design, LCDs require a built-in backlight to make the graphic images visible, while OLEDs do not.

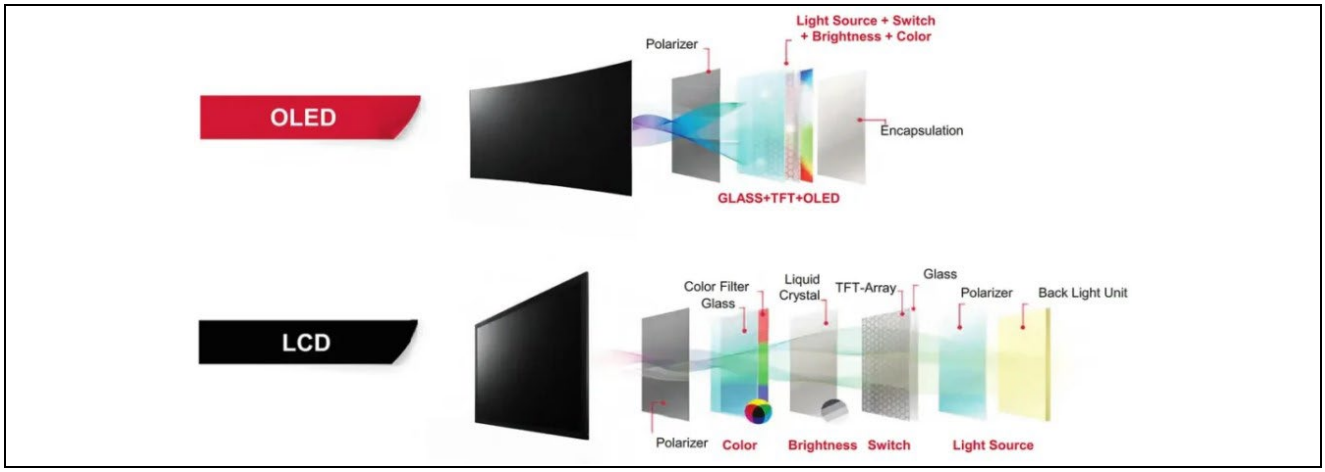


**Figure 1. LCDs need a background light source, unlike OLEDs**

The difference in technological implementations means there are different strengths and weaknesses between OLEDs and LCDs to consider. The main differences are summarized below, but note that these are general and each depends on the electrical characteristics of the device in question:

**Table 2. OLED vs LCD quick feature comparison**

| Feature | OLED | LCD |
|---|---|---|
| **Contrast** | Provides bolder, sharper images with higher contrast. Dark pixels are completely shut off for true blacks. | A dark pixel is still illuminated by the backlight, resulting in less intense blacks and lower contrast. |
| **Brightness** | Each pixel illuminates individually, producing intense brightness at the pixel level. May be less visible in direct sunlight. | Powered by backlights, LCDs can remain visible in bright conditions and direct sunlight, often outperforming OLEDs in overall screen brightness. |
| **Viewing Angles** | Typically offers a wider viewing angle; thinner design allows pixels closer to the surface, improving angles. | Viewing angles are generally narrower compared to OLEDs; always verify with the manufacturer for specifics. |
| **Lifespan** | May have a shorter lifespan for static images due to burn-in risks, but technology is rapidly improving and ideal for dynamic images. | Often outlasts OLEDs for static images, as LCDs are less susceptible to burn-in effects. |
| **Black Levels** | Black pixels are completely shut off, resulting in deeper blacks. | Black pixels remain illuminated by the backlight, so blacks are not as deep. |
| **Burn-In** | More prone to burn-in if static images remain for too long; permanent afterimages may occur. | More resilient to burn-in; static images do not cause permanent damage as easily. |

## 3.2 The Application Project's External OLED Display

The application projects accompanying the app note use the **Newhaven Display NHD-1.5-AU-Shield,** a 1.5-inch color OLED designed with an Arduino Uno shield header.

The on-board SSD1351 display controller handles pixel data, color control, and communication. Full specifications and command documentation for the SSD1351 are available on the New Haven website: Get SSD1351 Datasheet.

The OLED operates over a **4-wire SPI interface**, with built-in logic level shifting to accommodate both 3.3V and 5V systems. The OLED display has a **128×128 pixel resolution**, an internal 128×128×18-bit GDDRAM, and supports **262K colors** (6 bits per channel for RGB).

## 3.3 Matching the MCU's and External Display's SPI Interfaces

This section explains the process of operating the FPB-RA4E1's SPI bus within the NHD-1.5-AU-Shield's SPI communication specifications. Topics include choosing between SPI and SCI SPI, configuring the SPI bus in software, and physically connecting the external display to the RA4E1's SPI bus.

The NHD-1.5-AU-Shield's Datasheet states the requirements for the SPI interface, and the RA4E1 Group User's Manual: Hardware explains how to configure and control the SPI bus.

### 3.3.1 Choosing Between the SPI and SCI SPI Modules

First, analyze the external SPI peripheral's datasheet and compare it to the RA MCU Hardware User's Manual to determine which MCU module can satisfy the peripheral's SPI data format and signal timing requirements.

The SSD1351 display controller communicates via a 4-wire SPI interface. The data packets are specified as 8-bit length, no parity bit, and the baud rate requirement has a maximum of 4.54 MHz. (Based on the 220 ns maximum clock cycle time). Both the SPI and SCI SPI modules can satisfy the format and timing requirements on the RA4E1.

Second, determine the pin availability of SPI channels on the MCU based on the target application's needs.

All 3 application projects require a single 4-wire SPI module and a handful of GPIO pins. Both SPI and SCI SPI have pins available based on the total pin usage. However, note that the NHD-1.5-AU-Shield OLED has an Arduino Shield connector. The FPB-RA4E1 comes with a compatible Arduino Uno interface on which SCI SPI channel 0 is assigned.

When comparing SPI and SCI SPI for the application projects, based on the pinout, SCI SPI is more advantageous than SPI.

### 3.3.2 Configuring the SPI Bus According to the External Display Specifications

Now that SCI SPI has been selected, use the following references to set up the SPI bus to communicate with the OLED in an e$^2$ studio project:

- RA4E1 Hardware User's Manual
  - Chapter 27 "Serial Communications Interface (SCI)"
  - Subchapters on "Simple SPI"
- FSP User's Manual v6.0.0
  - Chapter 7 "API Reference"
  - Modules > Connectivity > SPI on r_sci_spi

To add and configure an SCI SPI module in an e$^2$ studio RA C/C++ project, open the configuration.xml file, and open the **Stack tab** while in the **FSP Configuration View**. Add the module's stack using **New Stack > Search** and enter **"SCI SPI"**. Or add the stack with **New Stack > Connectivity > SPI (r_sci_spi)**.

While the stack is highlighted, the **Properties tab** will open with the configuration options. The default configurations are shown in the image below:

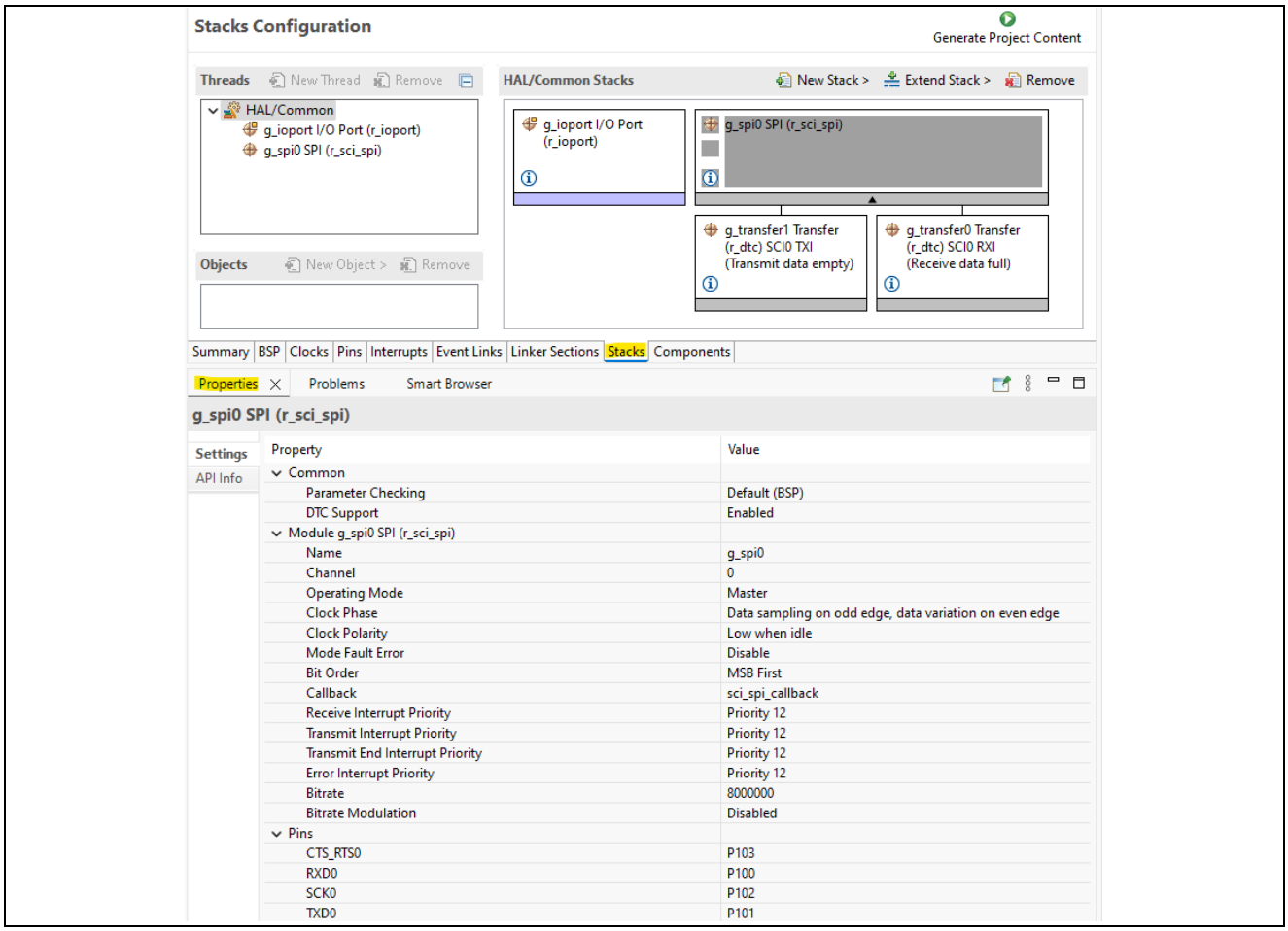


**Figure 2. Default configurations for the r_sci_spi stack on the FPB-RA4E1**

At this point, the following configurations are known:

- Name: g_spi
  - Provide a name for the stack.
- Channel: 0
  - Channel corresponds to the Arduino Uno pinout assignment.
- Operating Mode: Master
  - The RA4E1 MCU controls the SSD1351 slave device.
- Mode Fault Error: Disable
  - No need to detect master/slave conflicts with a single master setup.
- Callback & Priorities

The remaining stack settings for SCI SPI can be determined by analyzing the SSD1351 and packet specifications and signal timing for the 4-wire SPI interface.

The packet specification diagram is shown below. Because data is sent from D7 through D0, the r_sci_spi **Bit Order** should be configured to **MSB First**.

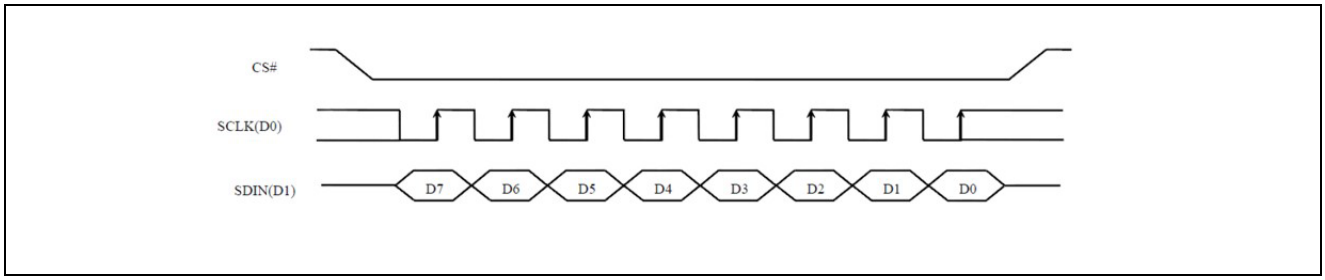


**Figure 3. Packet Diagram for SSD1351 4-Wire SPI**

The above diagram also provides information on valid configurations for the clock's idle state (high or low) and the sampling edge (even or odd). To satisfy SSD1351 specifications, the clock is valid to idle high or idle low, and in either case the data bits are sampled on the rising edge.

On the RA4E1, the SCI's SPI Mode Register (SPMR) has bitfield CKPH to set the clock phase and bitfield CKPOL to set the clock polarity. Compare the above packet specification diagram to Figure 4, which shows the relation between CKPH and CKPOL values.

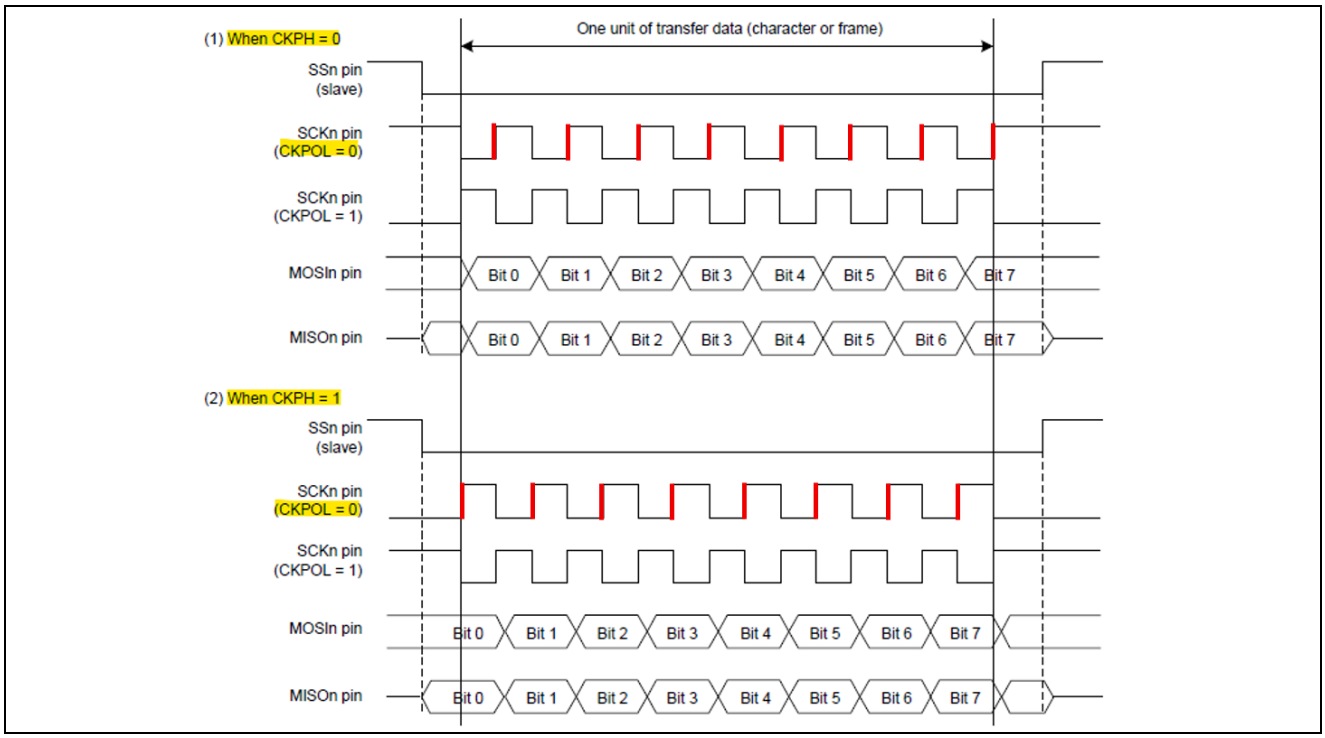


**Figure 4. Relation between CKPH and CKPOL for SCI in Simple SPI Mode**

The highlighted sets of SCK behavior both satisfy the SSD1351 specifications, idle high or idle low, with the red lines indicating the rising edge sample point.

The top set of values, CKPH = 0 and CKPOL = 0, is selected for use in all application projects to configure the r_sci_spi:

- Clock Phase: Data sampling on even edge, data variation on odd edge
- Clock Polarity: High when idle

The bottom set of values, CKPH = 1 and CKPOL = 0, is an alternative valid setting for r_sci_spi to interface with the NHD OLED. It was not used to configure the projects, but it is included here as a theoretical example:

- Clock Phase: Data sampling on odd edge, data variation on even edge
- Clock Polarity: Low when idle

The remaining settings require more timing information to determine. The timing diagram and table from the SSD1351 datasheet are shown below for reference:

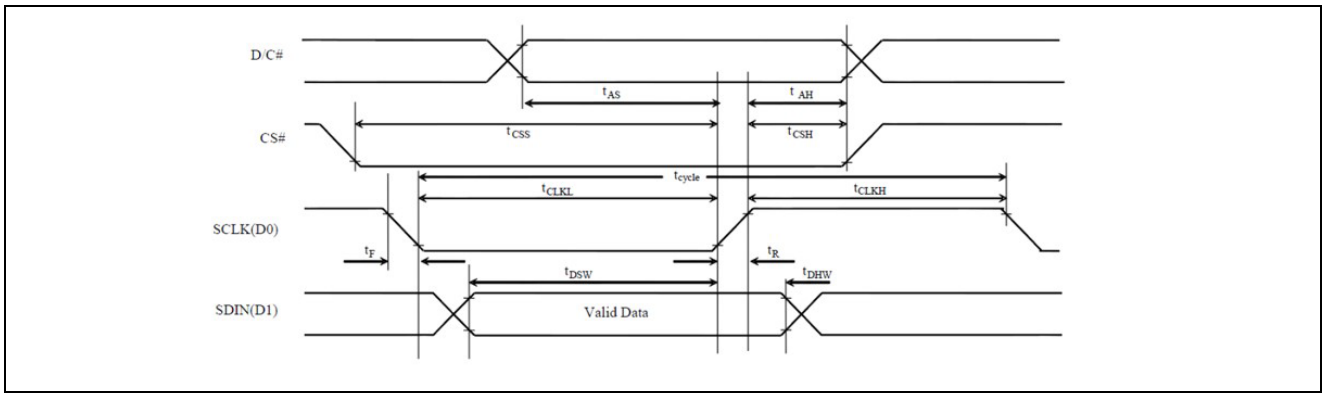


**Figure 5. Timing Diagram for SSD1351 4-Wire SPI**

**Table 3. Symbol Chart for SSD1351 Timing Diagram**

| Symbol | Parameter | Min | Typ | Max | Unit |
|---|---|---|---|---|---|
| **tcycle** | Clock Cycle Time | 220 | -- | -- | ns |
| **tAS** | Address Setup Time | 15 | -- | -- | ns |
| **tAH** | Address Hold Time | 42 | -- | -- | ns |
| **tCSS** | Chip Select Setup Time | 20 | -- | -- | ns |
| **tCSH** | Chip Select Hold Time | 10 | -- | -- | ns |
| **tDSW** | Write Data Setup Time | 15 | -- | -- | ns |
| **tDHW** | Write Data Hold Time | 20 | -- | -- | ns |
| **tCLKL** | Clock Low Time | 20 | -- | -- | ns |
| **tCLKH** | Clock High Time | 20 | -- | -- | ns |
| **tR** | Rise Time | -- | -- | 15 | ns |
| **tF** | Fall Time | -- | -- | 15 | ns |

The timing diagram specifies the remaining e$^2$ studio r_sci_spi stack configurations needed:

- Bitrate: 4000000
  - Selected 4 MHz bitrate. Based on the tcycle 220 ns minimum period, the maximum bitrate allowed is 4.545 MHz.
- Bitrate Modulation: Disabled
  - Unused

The timing diagram also indicates that there is an additional signal D/C# needed. This signal is used to indicate whether the SPI data contains an SSD1351 Data or Command packet. A GPIO pin can be assigned for the Data/Command select line control.

Also note that when SCI operates in Simple SPI Master Mode, there is no active SS line output. Another GPIO pin should be assigned and asserted low during the duration of the example applications to satisfy the CS# signal requirements.

A screenshot of the final configuration is shown below:

Properties | Problems | Smart Browser

**g_spi SPI (r_sci_spi)**

Settings | API Info

| Property | Value |
|---|---|
| Common | |
| Parameter Checking | Default (BSP) |
| DTC Support | Enabled |
| Module g_spi SPI (r_sci_spi) | |
| Name | g_spi |
| Channel | 0 |
| Operating Mode | Master |
| Clock Phase | Data sampling on even edge, data variation on odd edge |
| Clock Polarity | High when idle |
| Mode Fault Error | Disable |
| Bit Order | MSB First |
| Callback | sci_spi_callback |
| Receive Interrupt Priority | Priority 12 |
| Transmit Interrupt Priority | Priority 12 |
| Transmit End Interrupt Priority | Priority 12 |
| Error Interrupt Priority | Priority 12 |
| Bitrate | 4000000 |
| Bitrate Modulation | Disabled |
| Pins | |
| CTS_RTS0 | P103 |
| RXD0 | P100 |
| SCK0 | P102 |
| TXD0 | P101 |

**Figure 6. The r_sci_spi module is configured to the NHD OLED's SPI interface specifications**

### 3.3.3 Connecting the External Display to the MCU

This section details the pin connections between the external display and the MCU. Since the OLED comes equipped with an Arduino header, it will be connected to the Arduino Uno pinout on the FPB-RA4E1.

The following schematic is shown in Figure 7 is the configurable Digital headers half of the Arduino Uno pinout on the FPB-RA4E1. The other half consists of the Analog and Power pins. The Analog and Power pin assignments are NOT reconfigurable, so they do not need to be shown.

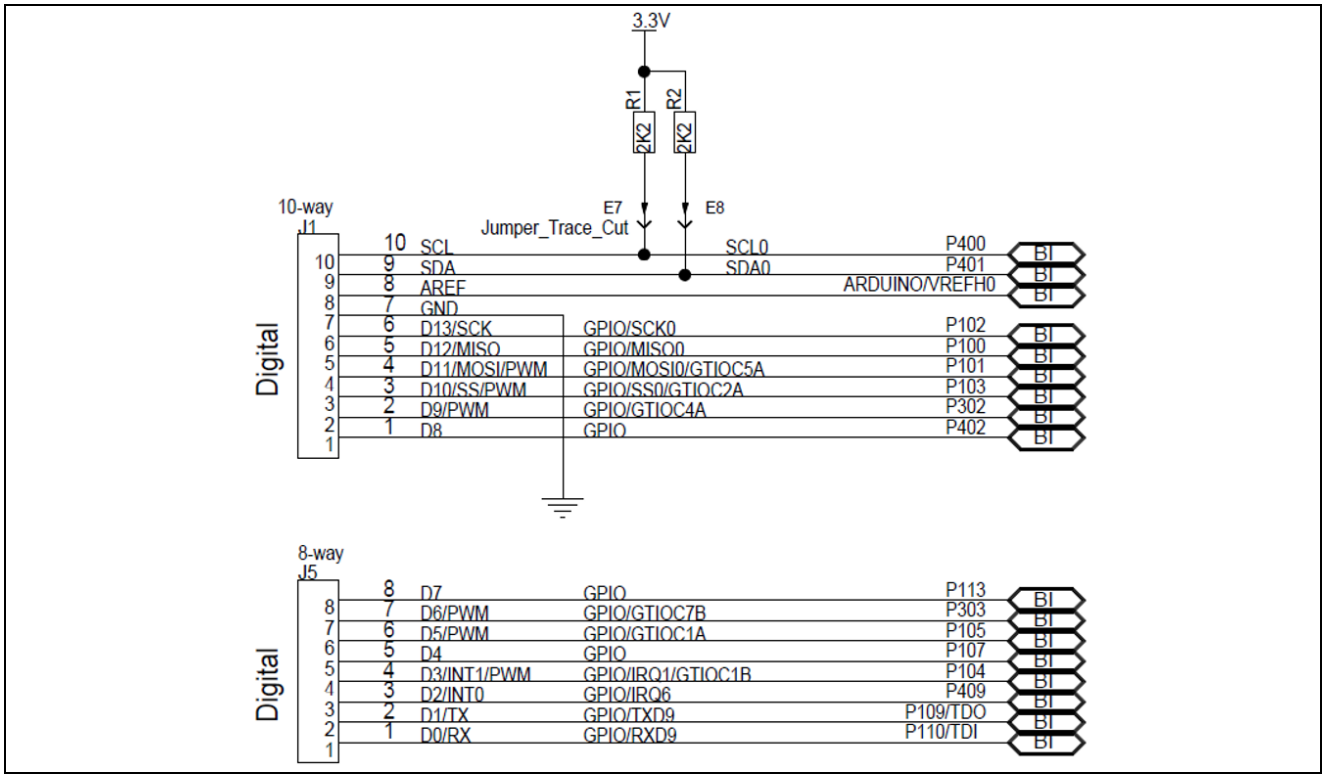


**Figure 7. Configure the FPB-RA4E1 Arduino Uno pins to match the OLED**

The above pins need to be configured in the configuration.xml Pins tab to match the pin interface requirements given in the NHD-1.5-AU-Shield OLED datasheet.

The tables below detail the pin interface that will match up to the 10-pin Digital header J1 (1-8) and the 8-pin Digital header J5 (1-8), respectively, on the FPB-RA4E1.

**Table 4. NHD-1.5-AU-Shield Arduino Interface Digital Pin Description**

| Shield Pin Symbol | Arduino UNO Pin Symbol | Function Description |
|---|---|---|
| AREF | AREF | No Connect |
| GND | GND | Ground |
| Digital 13 | 13 | Serial Clock signal |
| Digital 12 | 12 | Master In / Slave Out |
| Digital 11 | 11 | Master Out Slave In |
| Digital 10 | 10 | Micro SD Active LOW Chip Select signal |
| Digital 9 | 9 | No Connect |
| Digital 8 | 8 | No Connect |

| Shield Pin Symbol | Arduino UNO Pin Symbol | Function Description |
|---|---|---|
| Digital 7 | 7 | No Connect |
| Digital 6 | 6 | Active LOW Reset signal |
| Digital 5 | 5 | OLED Active LOW Chip Select signal |
| Digital 4 | 4 | Register Select signal. D/C=0: Command, D/C=1: Data |
| Digital 3 | 3 | No Connect |
| Digital 2 | 2 | No Connect |
| Digital 1 | 1 | No Connect |
| Digital 0 | 0 | No Connect |

By comparing the FPB-RA4E1 schematic to the pin tables of the NHD-1.5-AU-Shield OLED, the following pin assignment table below was created. Use the Pin tab of the configuration.xml to set up the assignment in an e$^2$ studio application.

**Table 5. Mapping the OLED Pins to the FPB-RA4E1**

| OLED Pins | Function | Arduino Shield Mapping | FPB-RA4E1 Pin Mapping | Notes |
|---|---|---|---|---|
| **MOSI** | Master Out | D11 | P101 - MOSI0 [J1:4] | Serial data from MCU to OLED. |
| **SCK** | Serial Clock | D13 | P102 - SCK0 [J1:6] | Serial clock signal. |
| **D/C#** | Data/Command Select | D4 | P107 - GPIO [J5:5] | OLED Data/Command register select signal.<br>0 - Command<br>1 - Data |
| **RES#** | Reset | D6 | P303 - GPIO [J5:7] | Active low OLED hardware reset signal. |

| | | | | |
|---|---|---|---|---|
| **OLEDCS** | OLED Chip Select | D5 | P105 - GPIO [J5:6] | Active low OLED chip select signal. |
| **SDCS** | MicroSD Chip Select | D10 | P103 - SS0<br>[J1:3] | Active low MicroSD chip select signal. Unused and kept held high. |
| **MISO** | Master In | D12 | P100 - MISO0 [J1:5] | SSD1351 does not output data; only SCI_SPI transmission is used in the example. |
| **VDD** | OLED Supply Voltage | 5V | 5V | Supply voltage for OLED and logic at 5V. |
| **GND** | Ground Reference | GND | GND | Common ground between OLED and MCU. |

## 3.4 Controlling the External Display Over SPI

At runtime, the FPB-RA4E1 microcontroller controls the OLED by sending the on-board SSD1351 controller a series of commands and data over the SPI bus. The main signals used for communication are D/C#, MOSI, SCK, OLEDCS, and RES#.

This section describes controlling the OLED at runtime through custom drivers composed of FSP and BSP APIs. The major topics include: creating drivers to send commands vs data, initializing the display with a startup sequence of commands, and sending a framebuffer to the display.

### 3.4.1 Sending Commands and Data Packets

This section explains how to create drivers to send commands and data to the display.

The commands instruct the display controller to perform an operation, and the data provides relevant parameters. The D/C# signal level will indicate whether the SPI bus is sending Data (high) or a Command (low). The full list of commands and their descriptions can be found in the SSD1351 specification.

The D/C# line is assigned to GPIO P107 (BSP_IO_PORT_01_PIN_07) and will need to be asserted before sending the 8-bit SPI packet. Utilize the BSP pin APIs, as shown in the example routine **pin_set** below, to properly control the D/C# level.

```
/* Change GPIO pin level. */
void pin_set(bsp_io_port_pin_t pin, bsp_io_level_t state)
{
    R_BSP_PinAccessEnable();
    R_BSP_PinWrite(pin, state);
    R_BSP_PinAccessDisable();
}
```

After setting the D/C# signal level appropriately, use the FSP API **R_SCI_SPI_Write** to send the Data or Command byte over the SPI bus. The **g_transfer_complete** flag is used to indicate a successful SCI SPI transfer complete.

```
#define DATA_CMD   (BSP_IO_PORT_01_PIN_07)
#define SEND_DATA (BSP_IO_LEVEL_HIGH)
#define SEND_CMD   (BSP_IO_LEVEL_LOW)

/* Send OLED command over SPI Bus. */
void OLED_Command(uint8_t command)
{
```

```
    pin_set(DATA_CMD, SEND_DATA);
    R_SCI_SPI_Write(&g_spi_ctrl, &command, 1, SPI_BIT_WIDTH_8_BITS);
    while(!g_transfer_complete);
    g_transfer_complete = false;
}

/* Send OLED data over SPI Bus. */
void OLED_Data(uint8_t data)
{
    pin_set(DATA_CMD, SEND_CMD);
    R_SCI_SPI_Write(&g_spi_ctrl, &data, 1, SPI_BIT_WIDTH_8_BITS);
    while(!g_transfer_complete);
    g_transfer_complete = false;
}

/* Block until transfer completes. */
void sci_spi_callback (spi_callback_args_t * p_args)
{
    switch (p_args->event)
    {
        case SPI_EVENT_TRANSFER_COMPLETE:
            g_transfer_complete = true;
            break;
        default:
            break;
    }
}
```

### 3.4.2 External Display Initialization Sequence

External displays need to be initialized after a power-on or hardware reset. This initialization procedure typically requires toggling the RESET line for a specified period, followed by sending a sequence of commands and data to the on-board controller.

This section explains the initialization procedure implemented in the accompanying application projects, but the steps can be generalized for a different target display. In the application projects, these functions are defined in the /src/oled_drivers.c, and supporting definitions are in /src/oled_driver.h.

First, to ensure the SSD1351 controller starts from a known state, implement an OLED reset. The /RESET line **RESET_OLED** is assigned to GPIO P303 (**BSP_IO_PORT_03_PIN_03**) and needs to be toggled low for 500 ms. Utilize the BSP API **R_BSP_SoftwareDelay** to introduce the required software delay:

```
pin_set(RESET_OLED, BSP_IO_LEVEL_LOW);
R_BSP_SoftwareDelay(500, BSP_DELAY_UNITS_MILLISECONDS);
pin_set(RESET_OLED, BSP_IO_LEVEL_HIGH);
R_BSP_SoftwareDelay(500, BSP_DELAY_UNITS_MILLISECONDS);
```

After resetting the OLED, the following command sequence configures the display. Some commands are **rigid** and must be set to specific values, while others are **flexible** and can be tuned based on application needs. The rigid commands configure the NHD OLED hardware to operate as expected electrically. The flexible commands configure the aesthetics of the display, based on application needs.

**Table 6. NHD OLED Initialization Command Sequence Explanation**

| | Command | Command Name & Macro Definition | Parameter Data & Reason | Classification | Command Explanation |
|---|---|---|---|---|---|
| **1** | 0xFD | Set Command Lock (CMD_CMD_LOCK) | 0x12 (Unlock OLED IC MCU interface) 0xB1 (Unlock specific commands: A2, B1, B3, BB, BE, C1) | Rigid | Mandatory initial step to ensure all necessary subsequent configuration commands are accessible to the MCU. |
| **2** | 0xAE | Set Sleep mode ON, Display OFF (CMD_DISPLAY_OFF) | N/A | Rigid | Display is turned OFF prior to configuration, which is standard procedure during initialization. |
| **3** | 0xB3 | Set Front Clock Divider & Oscillator Frequency (CMD_CLOCK_DIV_FREQ) | 0xF1 (Sets clock divide ratio and frequency) | Rigid | Sets fundamental timing parameters (DCLK and internal oscillator frequency) critical for stable chip operation and frame rate generation. |
| **4** | 0xCA | Set Multiplex Ratio (CMD_MUX_RATIO) | 0x7F (128 MUX ratio) | Rigid | The NHD-1.5-AU-SHIELD is a 128x128 pixel display; this command sets the driver to the required 1:128 multiplex mode. |
| **5** | 0xA2 | Set Display Offset (CMD_DISP_OFFSET) | 0x00 (No offset) | Flexible | While setting 0x00 is necessary for standard screen centering, the function itself is used to vertically shift the display content. |
| **6** | 0xA1 | Set Display Start Line (CMD_DISP_START) | 0x00 (0x00 start line) | Flexible | Sets the starting address of the display RAM to be mapped to the display. Setting this to 0x00 is usually required for the standard display view, but it allows for vertical scrolling. |

| | | | | | |
|---|---|---|---|---|---|
| **7** | 0xA0 | Set Re-map & Color Depth (CMD_REMAP) | 0xB4 (Sets 262k color mode and data mapping parameters) | Rigid | Sets the necessary color depth (262K for this module), and determines the physical mapping of RAM columns and color sequences (e.g., A B C vs. C B A). |
| **8** | 0xB5 | Set GPIO (CMD_GPIO) | 0x00 (GPIO pins disabled/HiZ) | Rigid | Disables or sets the state of the general-purpose I/O pins, typically disabled in standard operation. |
| **9** | 0xAB | Set Function Selection (CMD_FUNCTION_SEL) | 0x01 (8-bit interface, enables internal VDD regulator) | Rigid | Crucial command for enabling the internal VDD regulator and selecting the MCU interface bus width (8-bit parallel, 16-bit, or 18-bit). |
| **10** | 0xB4 | Set Segment Low Voltage (CMD_VSL) | 0xA0, 0xB5, 0x55 (External VSL) | Rigid | Configures the segment voltage reference (VSL). The example uses the external VSL mode, which is a hardware-dependent power configuration. |
| **11** | 0xC1 | Set Contrast Current for Colors A, B, C (CMD_CONTRAST_COLOR) | 0x8a, 0x51, 0x8a (Recommendation for this OLED) | Flexible but not advised | Sets the 256-step contrast value individually for the three color components A, B, and C. Controls the brightness of each color, giving the resulting color balance. |
| **12** | 0xC7 | Master Contrast Current Control (CMD_CONTRAST) | 0x0F (No change/highest factor) | Flexible | Controls the overall segment output current by applying a scaling factor (1/16 to 15/16, or no change 16/16=0xF) to all three colors. This is essentially a master brightness setting. |
| **13** | 0xB9 | Use Built-in Linear LUT (CMD_USE_GS_LUT) | N/A | Flexible | Selects the factory default linear Grayscale Look Up Table (LUT). This is related to gamma correction and visual appearance. |

| | | | | | |
|---|---|---|---|---|---|
| **14** | 0xB1 | Set Reset (Phase 1) / Pre-charge (Phase 2) period (CMD_PHASE_LENGTH) | 0x32 (Sets the length of Phase 1 and 2 periods) | Rigid | Configures the timing parameters (Phase 1 period for pixel reset and Phase 2 period for first pre-charge) essential for stable pixel driving of the OLED panel. |
| **15** | 0xBB | Set Pre-charge voltage (CMD_PRECHARGE_LVL) | 0x07 (Sets Phase 2 pre-charge voltage level) | Rigid | Sets the voltage level to which the pixel is driven during the first pre-charge phase, vital for accurate charging of the pixel capacitance. |
| **16** | 0xB2 | Display Enhancement (CMD_DISP_ENHANCE) | 0xa4, 0x00, 0x00 (Enable enhanced display performance) | Rigid | Enables features intended to enhance general display performance. |
| **17** | 0xB6 | Set Second Pre-charge Period (CMD_PRECHARGE_2) | 0x01 (1 DCLKS) | Rigid | Sets the timing for Phase 3 (second pre-charge) of the pixel driving cycle. |
| **18** | 0xBE | Set VCOMH Voltage (CMD_VCOMH_LVL) | 0x07 (Sets COM deselect voltage level) | Rigid | Sets the high voltage level for the common pins (VCOMH), which is an important operational voltage level configured with reference to VCC. |
| **19** | 0xA6 | Set Display Mode (Normal Display) (CMD_DISPLAY_NORMAL) | N/A | Rigid | Resets the display to normal mode, ensuring the display content reflects the GDDRAM data. |
| **20** | 0x15 | Set Column Address (CMD_SET_COLUMN) | 0x00, 0x7F (Start 0, End 127) | Flexible | Defines the boundaries (Start and End addresses) for the column address pointer, typically set to the full width (0x00 to 0x7F). |
| **21** | 0x75 | Set Row Address (CMD_SET_ROW) | 0x00, 0x7F (Start 0, End 127) | Flexible | Defines the boundaries (Start and End addresses) for the row address pointer, typically set to the full height (0x00 to 0x7F). |

| | | | | | |
|---|---|---|---|---|---|
| **22** | 0x5C | Write RAM Command (CMD_WRITE_GDDRAM) | N/A | Flexible | Prepares the display controller to accept pixel data for writing into the Graphic Display Data RAM (GDDRAM). This is application-dependent, based on when you want to draw data. |
| **23** | 0xAF | Set Sleep mode ON/OFF (Display ON) (CMD_DISPLAY_ON) | N/A | Rigid | Mandatory final step to bring the OLED display out of sleep mode and turn on the segment/common drivers. |

### 3.4.3 Drawing a Framebuffer

After the display is turned on and initialized to the appropriate application settings, writing to the OLED's GDDRAM at runtime will trigger the screen's pixels to update.

The GDDRAM Write Command (CMD_GDDRAM_WRITE, 0x5C) puts the OLED's controller in a state ready to receive pixel data.

For every active pixel, the MCU sends three bytes of data to specify the red, green, and blue values. Only the lower 6 bits of a SPI byte of color data is read by the display. The order of RGB vs BGR, along with the column and row order, is determined by the Re-Map and Color Command (CMD_REMAP pr 0xA0) in the initialization sequence.

When the display is initialized to RGB order, the following sequence of SPI packets shows the order and valid color data for filling the entire screen:

```
Packet 1 – Pixel1 – 00RRRRRR
Packet 2 – Pixel1 – 00GGGGGG
Packet 3 – Pixel1 – 00BBBBBB
Packet 4 – Pixel2 – 00RRRRRR
Packet 5 – Pixel2 – 00GGGGGG
Packet 6 – Pixel2 – 00BBBBBB
Packet 7 – Pixel3 – 00RRRRRR
Packet 8 – Pixel3 – 00GGGGGG
Packet 9 – Pixel3 – 00BBBBBB
            …
Packet 128*128*3 – Pixel 128 – 00BBBBBB
```

The code block below demonstrates using the functions defined in Section 3.4.1 to fill the screen with black pixels:

```
OLED_Command(CMD_WRITE_GDDRAM);
for (i = 0; i < DISPLAY_XRES; i++)
{
   for (j = 0; j < DISPLAY_YRES; j++)
   {
       OLED_Data(0x00);
       OLED_Data(0x00);
       OLED_Data(0x00);
```

```
    }
}
```

To send a full framebuffer of pixel data, first ensure that the framebuffer array is set up for the data packet format that the GDDRAM can accept, such as:

```
uint8_t framebuffer[128*128*3] = {P1 red: 00RRRRRR, P1 green: 00GGGGGG, P1 blue:
00BBBBBB, …, P16384 red: 00RRRRRR, P16384 green: 00GGGGGG, P16834 blue:
00BBBBBB};
```

Each time the graphics application draws a new framebuffer in the SRAM on the MCU, it needs to be sent over the SPI bus to the OLED's GDDRAM for the display to visually update.

The code block below shows an example procedure for sending a framebuffer using the functions defined in Section 3.4.1 with some FSP APIs:

```
#define CMD_WRITE_GDDRAM      (0x5C)
#define DATA_CMD              (BSP_IO_PORT_01_PIN_07)
#define SEND_DATA             (BSP_IO_LEVEL_HIGH)
#define SEND_CMD              (BSP_IO_LEVEL_LOW)
#define PIXEL_BYTES           (128*128*3)

/* Call this function after the application has updated the uint8_t
framebuffer[PIXEL_BYTES] */
void UpdateOLED(void)
{
    OLED_Command(CMD_WRITE_GDDRAM);
    pin_set(DATA_CMD, SEND_DATA);
    R_SCI_SPI_Write(&g_spi_ctrl, (const void *) &framebuffer, PIXEL_BYTES,
                                                    SPI_BIT_WIDTH_BITS);
    while(!g_transfer_complete);
    g_transfer_complete = false;
}

void sci_spi_callback (spi_callback_args_t * p_args)
{
    switch (p_args->event)
    {
        case SPI_EVENT_TRANSFER_COMPLETE:
            g_transfer_complete = true;
            break;
        default:
            break;
    }
}
```

## 4. Graphics Enhancement by Offloading the CPU

On RA family devices, there are a few modules that can offload basic event-driven operations from the CPU. The modules work by channeling an event (or interrupt) triggered by the CPU, by a peripheral on the device, or by an external pin to generate a number of actions on the device.

The simplified diagram below shows how an interrupt/event can trigger a traditional interrupt with CPU intervention, or how it can trigger the Direct Memory Access Controller (DMAC), the Data Transfer Controller (DTC), or the Event Link Controller (ELC) without CPU intervention. The DMAC and DTC can transfer data between memory locations while the ELC links certain peripheral actions on the device.

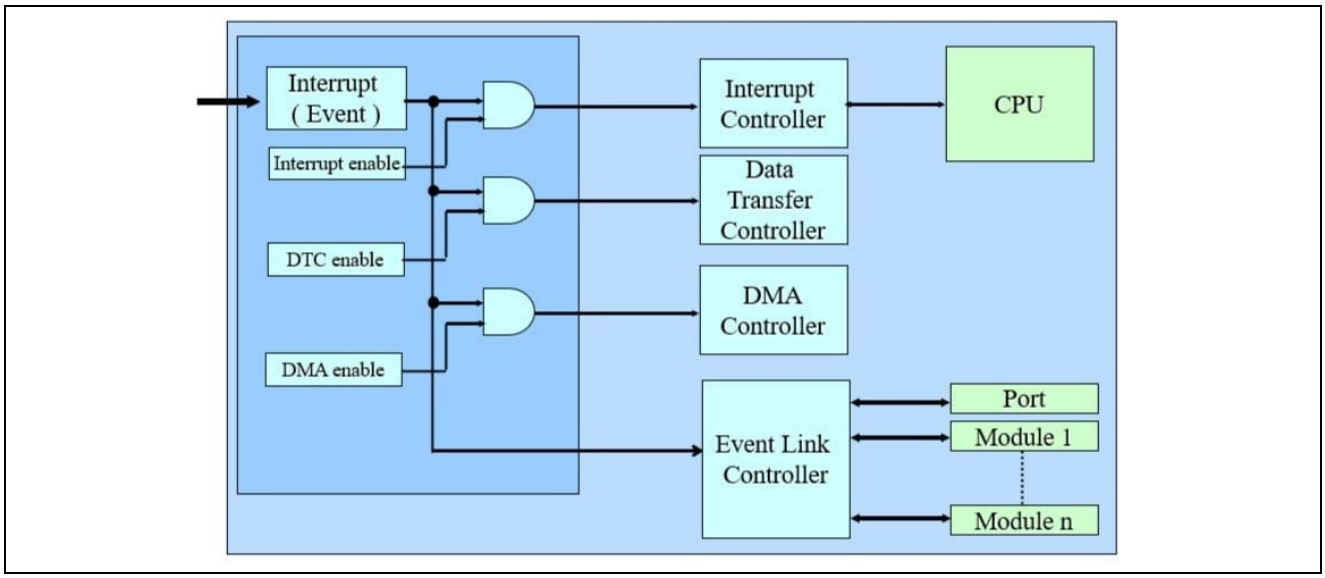


**Figure 8. Interrupts can be channeled with or without CPU intervention**

On the FPB-RA4E1, the DMAC has 8 dedicated hardware channels, while the DTC channel information is stored in the SRAM, so the number is limited by the application memory footprint. However, on the DTC, there is extra overhead from reading the configuration data in SRAM. The modules' characteristics are summarized below:

**Table 7. Feature Comparison of DMAC and DTC on the RA4E1**

| Feature | DMA | DTC |
|---|---|---|
| **Channels** | 8 independent hardware channels. | 1 shared hardware channel with flexible software vector mapping. |
| **Configuration** | Requires explicit source/destination setup. Channel configuration is stored in each activated channel's dedicated registers. | Uses interrupt vectors to trigger transfers. Vector configuration stored in SRAM array and loaded into the hardware channel at runtime. |
| **Bus Arbitration** | Fixed priority or round-robin. | Shares bus with CPU, arbitration handled internally. |
| **Integration with SCI_SPI** | Supported via DMAC activation sources. | Supported via DTC vector mapping to SPI TXI. |
| **Use Case Fit** | Best for large, continuous transfers. | Ideal for interrupt-driven, lightweight transfers. |

## 4.1 Using DTC/DMAC to Send the Framebuffer

It's advantageous to offload the CPU from performing large memory transfers when possible, so that the CPU can process the more intensive graphics tasks instead.

Sending the framebuffer to the OLED requires 128x128x3 = 49152 Byte transfers on the SPI bus. The DMAC or DTC can be configured to move the framebuffer from SRAM to the SPI transfer buffers, driven by the SPI transfer complete interrupt.

In the FSP, the **r_sci_spi** stack comes integrated with the **r_dtc** stack enabled by default. The DMA controller can be used with SCI SPI, but it must be manually set up and controlled. Therefore, the DTC was selected over DMAC for the graphic applications.

In the FSP, the **r_spi** stack has both the **r_dmac** and **r_dtc** selectable and integrated.

# 5. Build and Run the Projects

There are three application projects included to demonstrate the concepts in this application note:

- *ColorBars_SCI_SPI_ra4e1*: uses color bar LUT to draw to OLED in a baremetal e² studio project
- *HelloWorld_GUIX_SCI_SPI_ra4e1*: graphics project designed in GUIX Studio, e² studio project built with the GUIX library and Eclipse ThreadX
- *HelloWorld_emWin_SCI_SPI_ra4e1*: graphics project designed in AppWizard, e² studio project built with emWin library and FreeRTOS

The Color Bars baremetal application sends commands and pixel data to the OLED over the SPI bus with DTC support. The graphics are simple and created by a color bar LUT in Flash, so the code does not require integrating a graphic library. This project is meant to demonstrate configuring and operating the SPI bus, using the FSP SPI drivers, and creating custom OLED drivers.

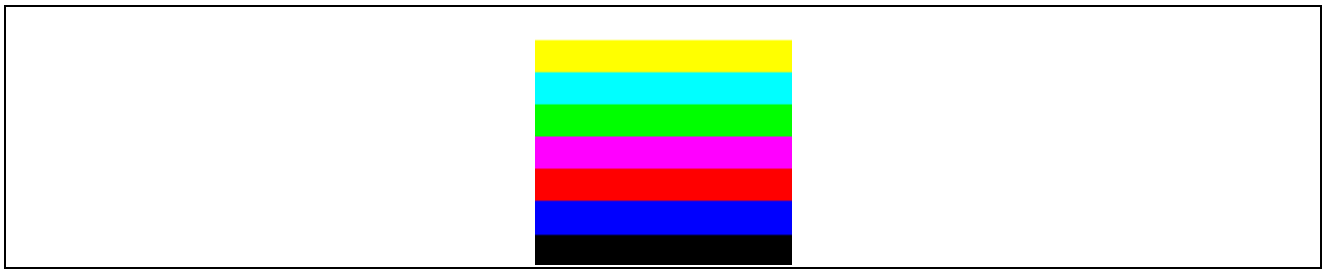

**Figure 9. The Color Bar application display image**

The Hello World projects are an extension of the Color Bars project. These projects use the GUIX and emWin graphics libraries to draw a background image on the display. These projects are meant to demonstrate how to add graphic libraries in RA MCUs that don't have native stack support, and how to integrate the custom OLED SPI drivers into each application's draw stack.


**Figure 10. The Hello World application display image**

Follow the steps in the next section to run and validate the *ColorBars_SCI_SPI_ra4e1,* which demonstrates all the ideas presented in Sections 1-4. **Skip Section 5.2 at this time.**

## 5.1 Steps to Run

The steps to build and run are the same for all projects.

1. Connect the FPB-RA4E1 to the NHD-1.5-AU-Shield by lining up the display to the Arduino header on the MCU:
2. Connect the FPB-RA4E1 to the host computer with the microUSB.
3. Download the application project folder and unzip it if needed.
4. Import the projects into the active e² studio workspace and build.
5. Click Debug, and press play twice.

## 5.2 Use Memory View to See the Framebuffer in SRAM

**Note: This section is only applicable to the project** *HelloWorld_GUIX_SCI_SPI_ra4e1.*

While in an active Debug session, the **Memory Browser** shows the real-time contents of the MCU's memory.

This View includes a **Raw Image Rendering** feature that is particularly useful for debugging graphics applications. Specify a framebuffer start address, the display resolution, and pixel color format to view the image in e² studio from the MCU memory. Image rendering validates that the graphics application is correctly updating and drawing the framebuffer.

Only the GUIX application project has a framebuffer in memory stored with a pixel format supported by the rendering feature. Follow the steps to view and validate the framebuffer:

1. **Pause** the application while in an active **Debug** session at the moment you wish to view the framebuffer in memory. (The call to guix_buffer_toggle_565rgb indicates GUIX has finished drawing the framebuffer.)
2. Open the **Memory View** while in **Debug View**. Navigate to the tab **Window > Show View > Memory.**
3. Click **Add New Monitor,** enter **&g_guix_framebuffer,** and press **OK.** The Browser will show the Hex Integer view of the memory region.
4. Click **Add New Rendering** to view the memory with a different render. Select **Raw Image**.
5. The Raw Image Format window will appear. Enter the dimensions **width** and **height** as **128 pixels.** Change the encoding to **RGB 16bpp (5:6:5)** and check that the **Start Position** is **Top.** Click **OK** to generate the rendering.

The following image shows a rendering of the **`g_guix_framebuffer`** in the FPB-RA4E1 SRAM, at a breakpoint when GUIX finishes drawing the framebuffer the first time.

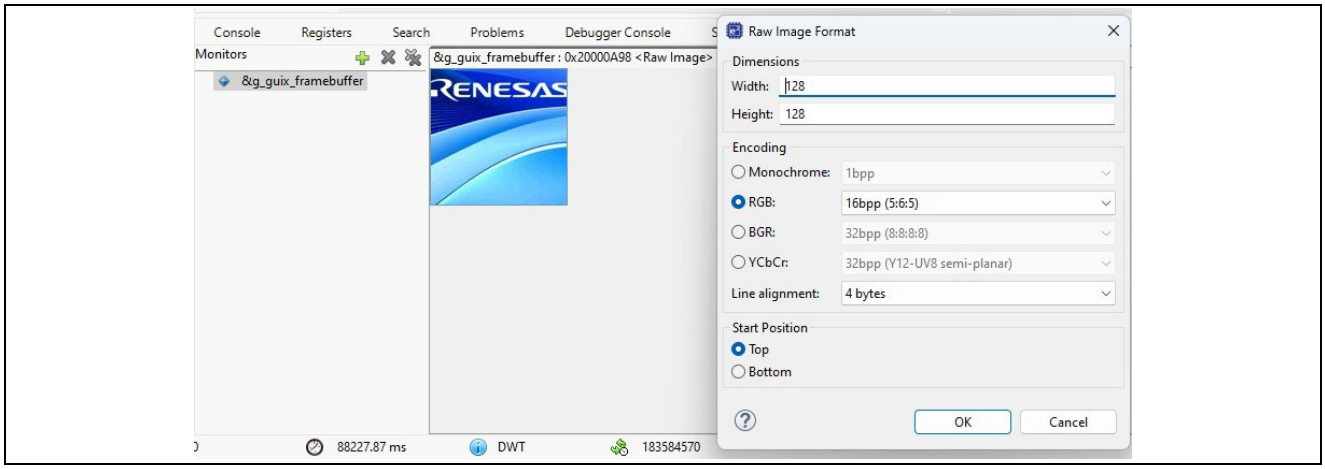


**Figure 11. Viewing the framebuffer in MCU memory ensures that the graphics library is drawing correctly**

## 6. Design a Graphics Application with the Renesas FSP

The Renesas FSP is the software framework for RA e² studio projects, providing HAL drivers, middleware, and libraries that are smoothly integrated into configurable peripheral stacks in e² studio.

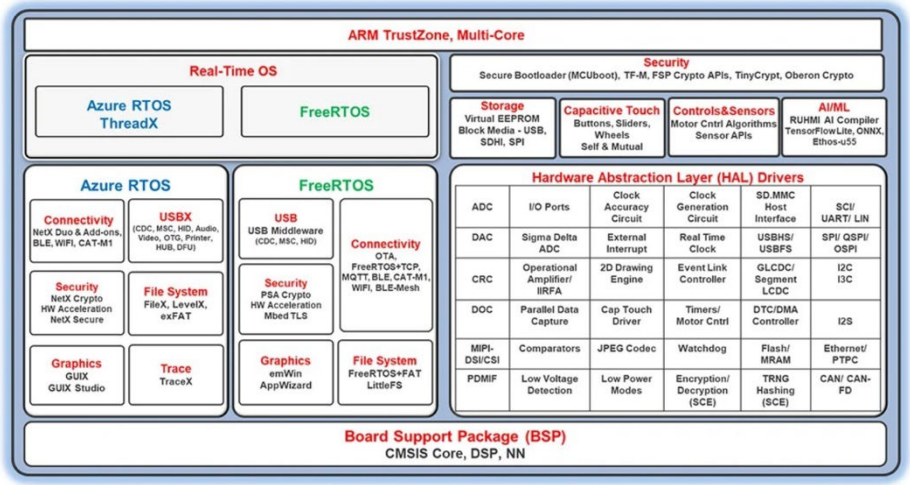


**Figure 12. Block Diagram of FSP Modules**

The FSP supports two major graphics libraries: GUIX and emWin, each offering distinct design environments and runtime capabilities.

- **GUIX and GUIX Studio**: Open-source embedded graphics library from Eclipse. Create projects with the GUIX Studio design environment, and run them with the GUIX library on ThreadX RTOS. The following resources are all downloadable from the Eclipse rtos-docs repository
    - **GUIX User Manual**
    - **GUIX Studio User Manual**
    - **ThreadX User Manual**
- **emWin and AppWizard**: Closed-source embedded graphics from SEGGER, with a free license available for non-commercial, evaluation, and educational use. Create projects with the AppWizard design environment, and run them with the emWin library on FreeRTOS.
    - emWin User Manual
    - AppWizard User Manual
    - FreeRTOS Documentation

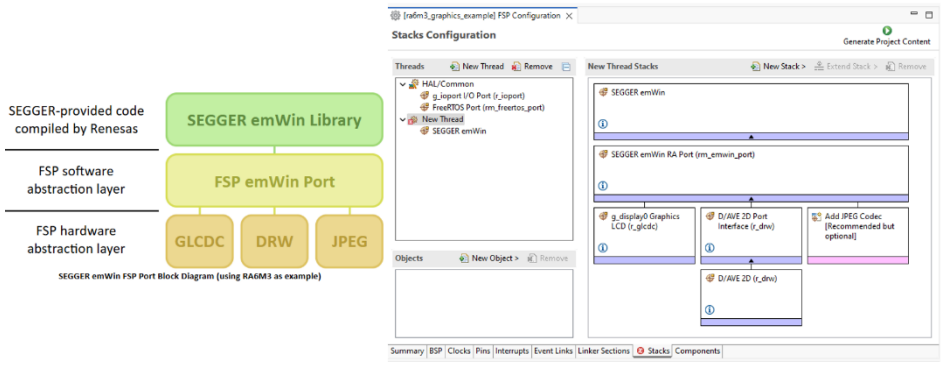


**Figure 13. On the EK-RA6M3G, emWin and GUIX are Part of a Larger Graphics Stack**

## 6.1 Overview of the Hello World Project

In comparison, when creating a graphics application on a non-graphics-focused RA MCU like the FPB-RA4E1, the emWin and GUIX libraries need to be manually added to the e² studio project. Users will need to create custom middleware to integrate the library with the underlying display drivers.

The Hello World application projects expand on the Color Bar project presented in the first half of this app note, by the addition of a graphics library. There are 2 project versions, one that uses GUIX Studio, GUIX library, and ThreadX RTOS, and one that uses AppWizard GUI, emWin library, and FreeRTOS:

- *HelloWorld_emWin_SCI_SPI_ra4e1*
- *HelloWorld_GUIX_SCI_SPI_ra4e1*

The Hello World application will initialize the OLED, fill the display screen with color bars, and start the emWin/GUIX application stack, which will, in turn, draw a simple background to the OLED.

The applications are intended to demonstrate:

1. How to manually add a graphics library to a RA e$^2$ studio project
2. How to design middleware for GUIX and emWin that integrates the custom OLED SCI SPI drivers within each library's application stack.

The next sections will walk through the process for each of the Hello World projects.

# 7. Hello World: GUIX

The application project titled *Hello_World_GUIX_SCI_SPI_ra4e1* uses the GUIX library running on ThreadX RTOS to execute a GUIX Studio graphics application titled “hello_world.gxp” onto the NHD-1.5-AU-Shield display.

The steps in the next subsections can be generalized to add and integrate a GUIX Studio project for any non-graphics-focused RA MCU and any external display.

## 7.1 Add GUIX Library to the e$^2$ studio Project

For RA MCUs that do not include native graphics hardware support, the GUIX library must be added manually to the e$^2$ studio project. Note that the e$^2$ studio project should be running with ThreadX.

Open the **configuration.xml** file in **the FSP Configuration View**, and click on the **Components** tab.

In the component tree, add the following, ensuring that the versions selected match the FSP version used in the project:

- **Microsoft > Azure > GUIX > gx**
- **Renesas > Middleware > all > rm_guix_port**

After adding the components, click **Generate Project Content** to include the component source files in your project.

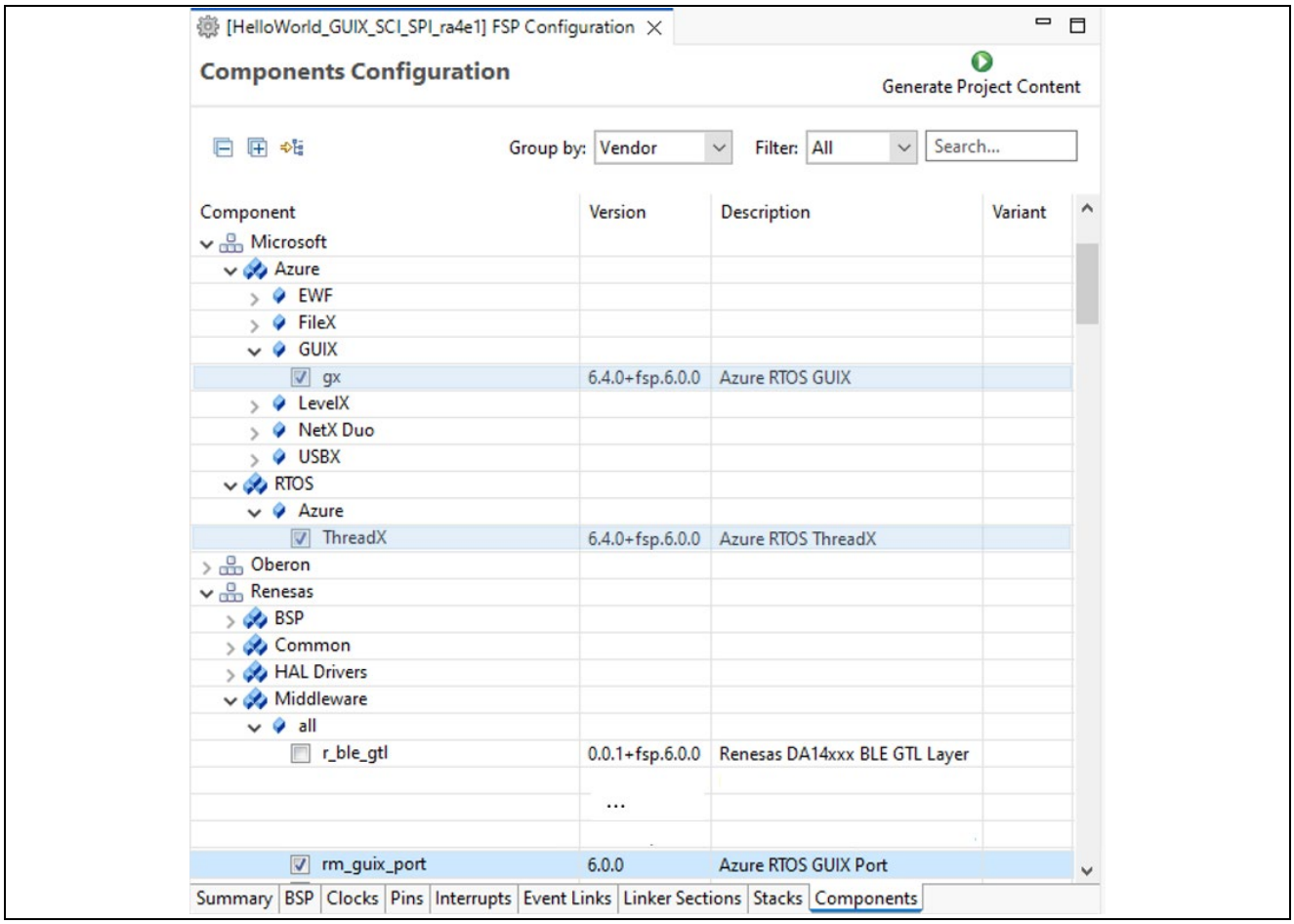


**Figure 14. Select rm_guix_port and gx in the components tab**

Some generated source files for the **rm_guix_port** component reference hardware modules that are not present on the FPB-RA4E1. They need to be removed from the project to avoid build errors.

Navigate to the generated folder **/ra/fsp/src/rm_guix_port**. For all files EXCEPT *gx_port.h* perform a **right-click** > **Resource Configurations > Exclude from Build…** After, the excluded files will be grayed out as shown below:

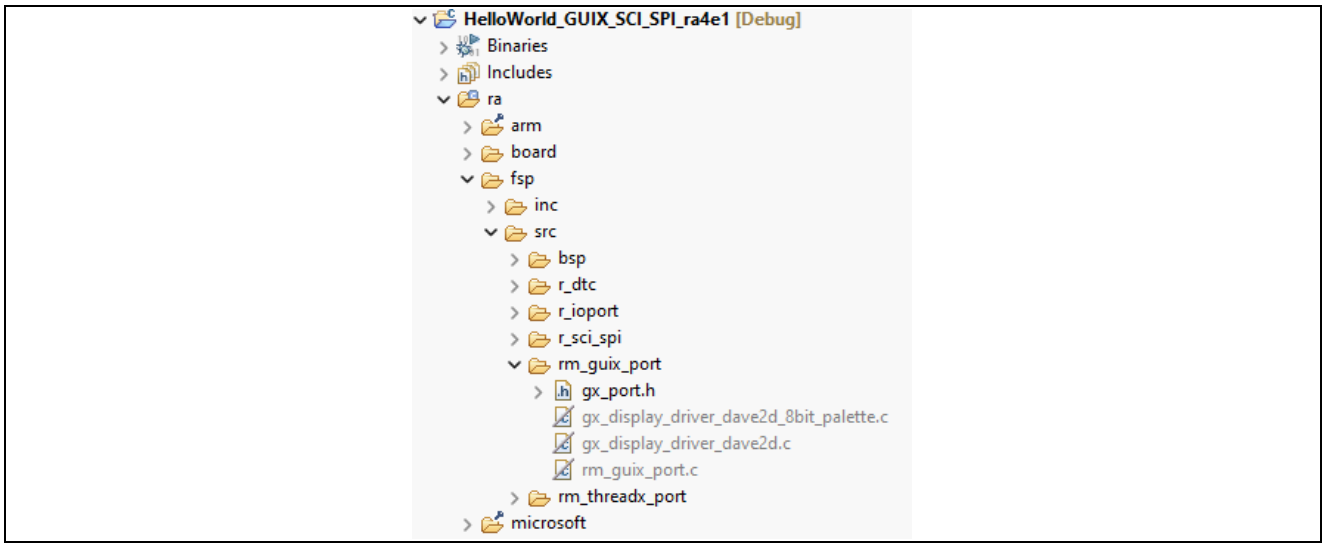


**Figure 15. Only the gx_port.h file is needed for a non-graphics RA MCU**

Now the e² studio project has included the GUIX Library and necessary supporting definitions from *gx_port.h*.

## 7.2 Integrating the GUIX Studio and e² studio Projects

The Hello World project is a simple non-animated graphics application that displays a custom JPEG image onto the NHD OLED. An overview of the GUIX Studio graphics application *hello_world.gxp* is shown in the screenshot below:

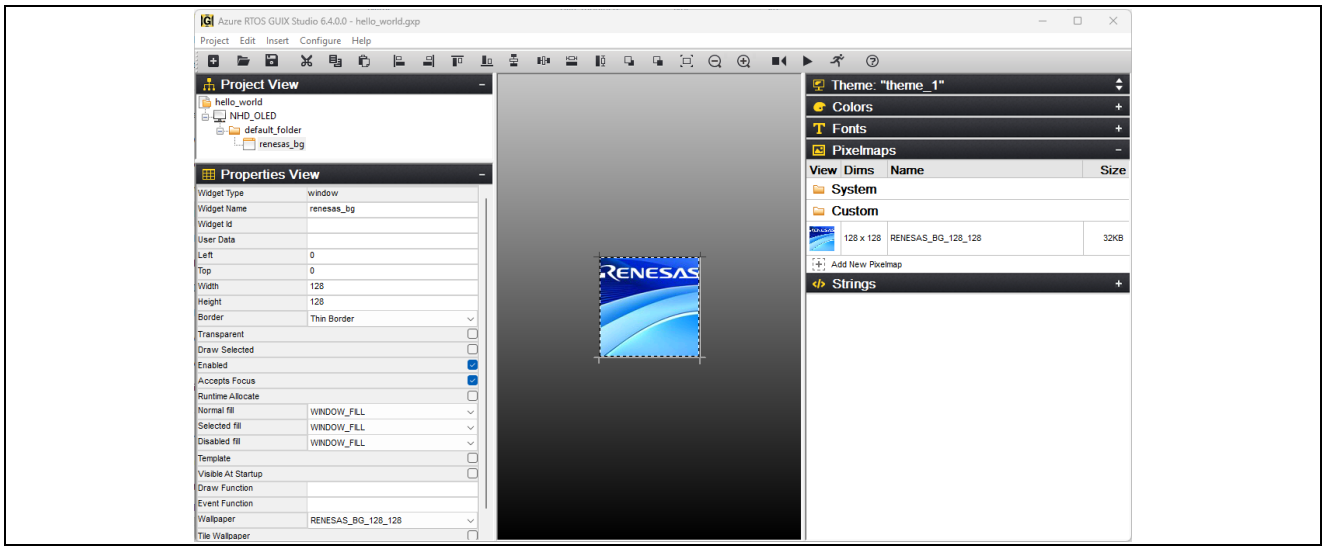


**Figure 16. Screenshot of Hello World GUIX Studio Project**

Review the GUIX Studio project properties under the tab **Configure > Project/Displays**, or in Figure 17 below.

The bottom half of the project properties is the **Display Configuration** section. It contains information about the external display settings, which will configure the way the GUIX system draws the display framebuffer at runtime. There is **1 Display** titled **NHD_OLED** and given the resolution of **128 Pixels X and 128 Pixels Y.** There is **NO** need to allocate canvas memory and **NO** screen rotation used in this application.

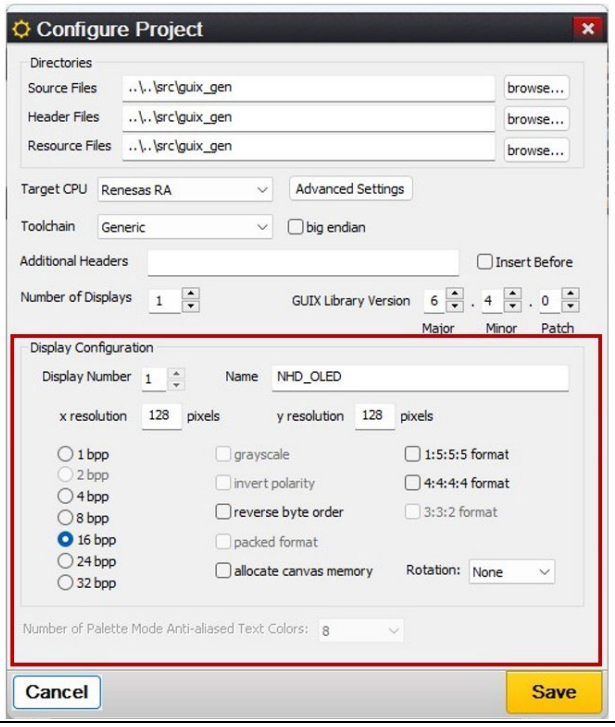


**Figure 17. The Display Configurations of the hello_world.gxp project**

Notice there is no exact match for the pixel color format options. The ideal format for the NHD OLED is 24 bits per pixel (bpp) using 666RGB in the LSB position: 00RRRRRR00GGGGGG00BBBBBB.

GUIX does NOT support custom color formats, so instead, a similar format must be selected from the project properties list. Then, at runtime, each time the GUIX system draws a framebuffer, it needs to be converted to match the OLED format before being sent on the SPI bus.

Compare the most similar formats available in the project properties:

1. 24bpp using 888RGB
    1. Framebuffer size: 24bpp (padded to 32 bits)*128 x pixels*128 y pixels = 65.536 KB
    2. Converting to 666RGB: The red, green, and blue channels will be reduced from 8 pixels to 6 pixels. With 2 bits of information lost on each color channel, the resulting image will have fewer colors and harsher gradients than the original.
2. 16bpp using 565RGB
    1. Framebuffer size: 16bpp*128 x pixels*128 y pixels = 32.768 KB
    2. Converting to 666RGB: The red and blue channels will be promoted from 5 pixels to 6 pixels, and the green channel will retain all color information. The red and blue interpolation will cause minimal visual artifacts, if any.

Based on the smaller framebuffer size and fewer visual differences to the original image, the **16bpp** using 565RGB was selected as the GUIX framebuffer format.

The top half of the GUIX Project Configurations is highlighted below in Figure 18 corresponds to the e² studio application project settings. The GUIX project's directories are all specified to be generated under the **\src\guix_gen** folder. The **Target CPU** is **Renesas RA**, the **Toolchain** is set to **Generic**, and the **GUIX Library version** is **6.4.0**.

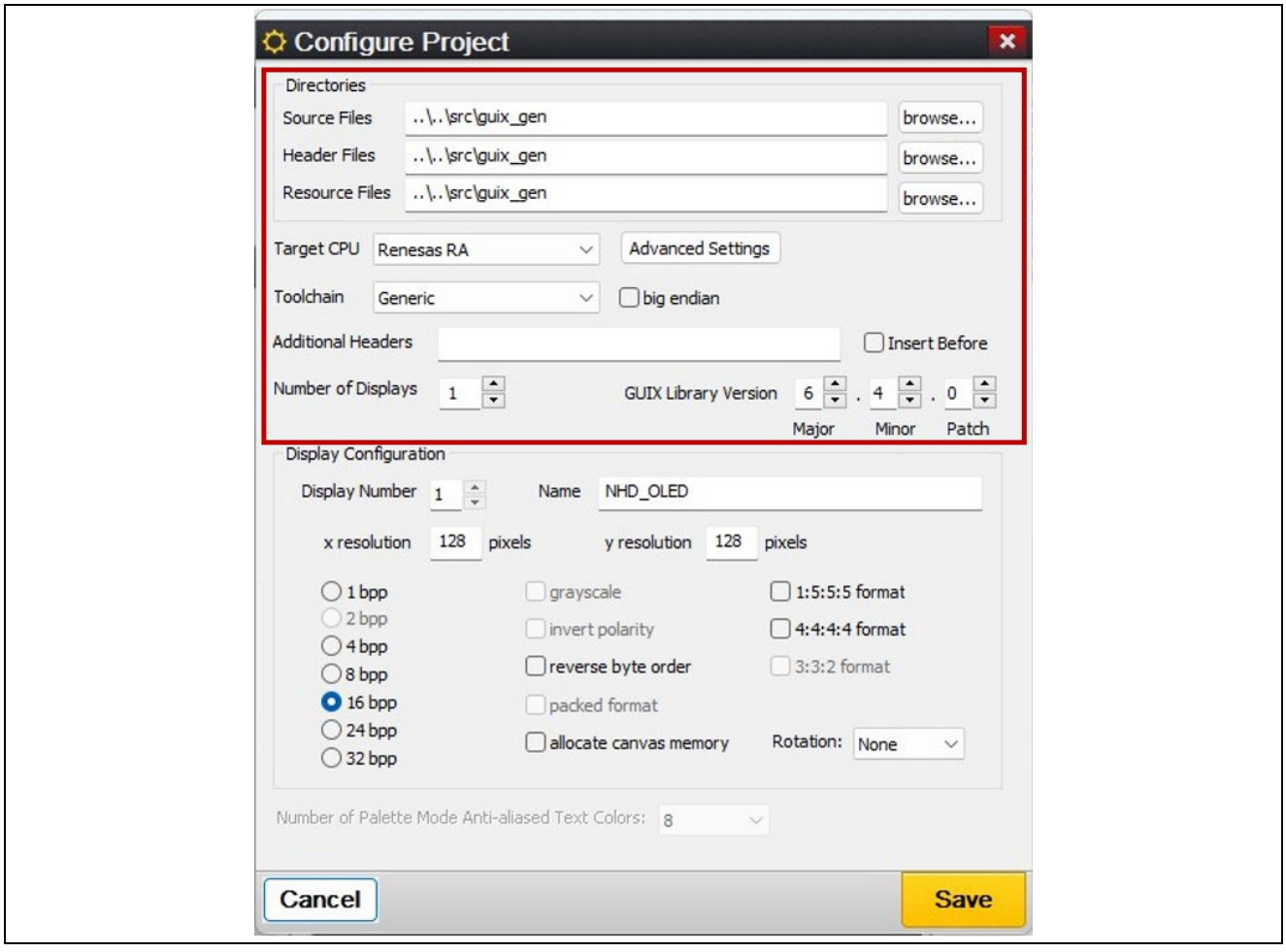


**Figure 18. The C Project File settings of the hello_world.gxp project**

The GUIX project's source files, header files, and resource file paths are specified relative to the *hello_world.gxp* project's location in the e² studio project file system. The following image of the

HelloWorld_GUI_SCI_SPI_ra4e1 workspace shows the relative location of the GUIX project in the **/guix_studio** folder (in blue), and the project's Source, Header, and Resource files in the **/guix_gen** folder (in red).

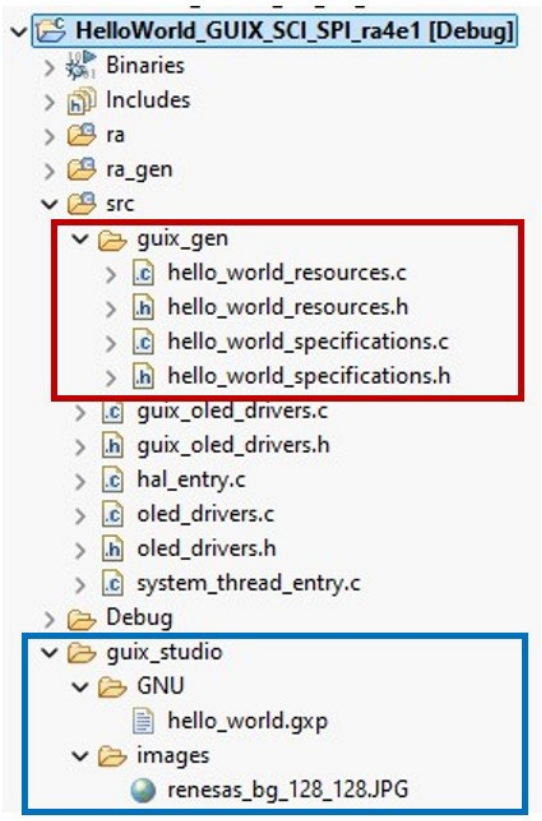


**Figure 19. Relative location of the GUIX project and its generated project files**

### 7.2.1 Create the GUIX Middleware

The GUIX Hello World application runs on ThreadX RTOS, and it executes the *hello_world.gxp* project on the NHD OLED using an extension of the OLED SCI SPI drivers, explained in Section 3.4.3 Drawing a Framebuffer.

The following image is the application block diagram, which shows the relationship between hardware and software modules used and the movement of the framebuffer data.

The GUIX system will draw updates to the **`g_framebuffer`** array over the CPU bus. The framebuffer is then redrawn by the GUIX middleware into the correct OLED color format **`g_framebuffer_rgb`**. The redrawn framebuffer is sent over the SPI bus to the NHD OLED using DMA support. Additional GPIO signals are used to communicate with the display.

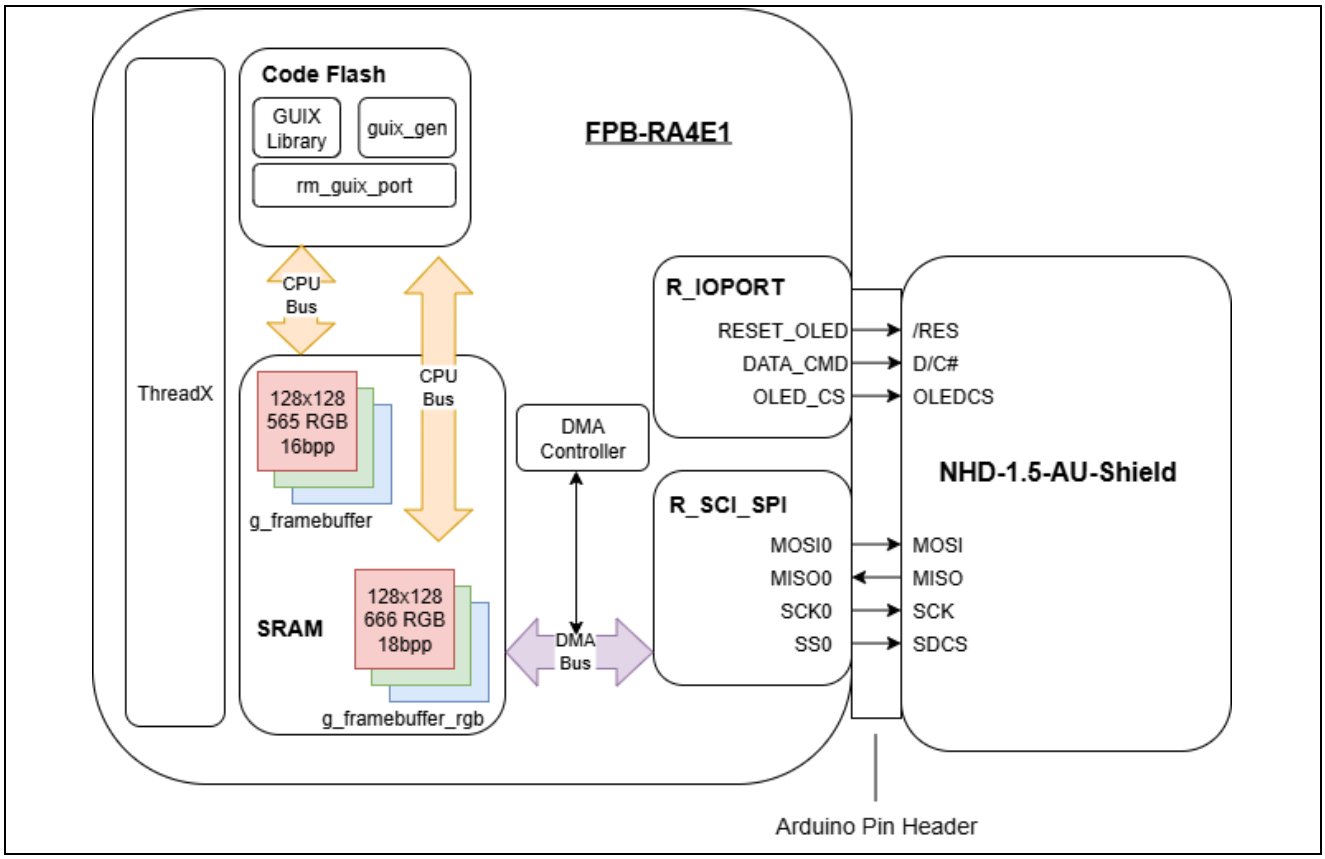


**Figure 20. The GUIX Hello World Application Block Diagram**

Custom-designed GUIX middleware integrates the low-level OLED SCI SPI display drivers with the GUIX library. The screenshot below of the e² studio workspace categorizes the project files:

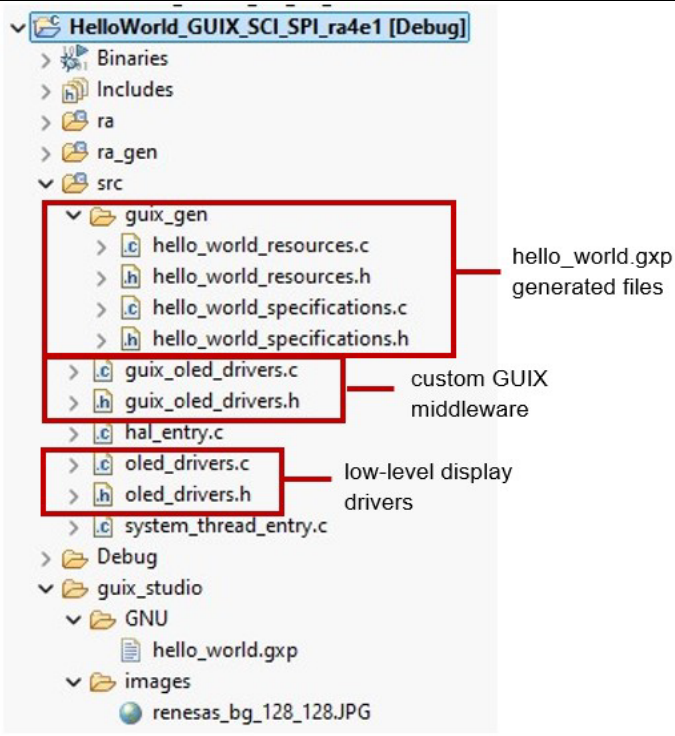


**Figure 21. The files comprising the GUIX middleware and low-level display drivers**

At runtime, the GUIX System Thread executes the .gxp graphics application, using the GUIX library functions to draw the framebuffer and the GUIX middleware to call the low-level display drivers. The main application

thread simply needs to initialize the MCU-related hardware, define the middleware, and start the GUIX System Thread.

The application flowchart is shown in greater detail below.

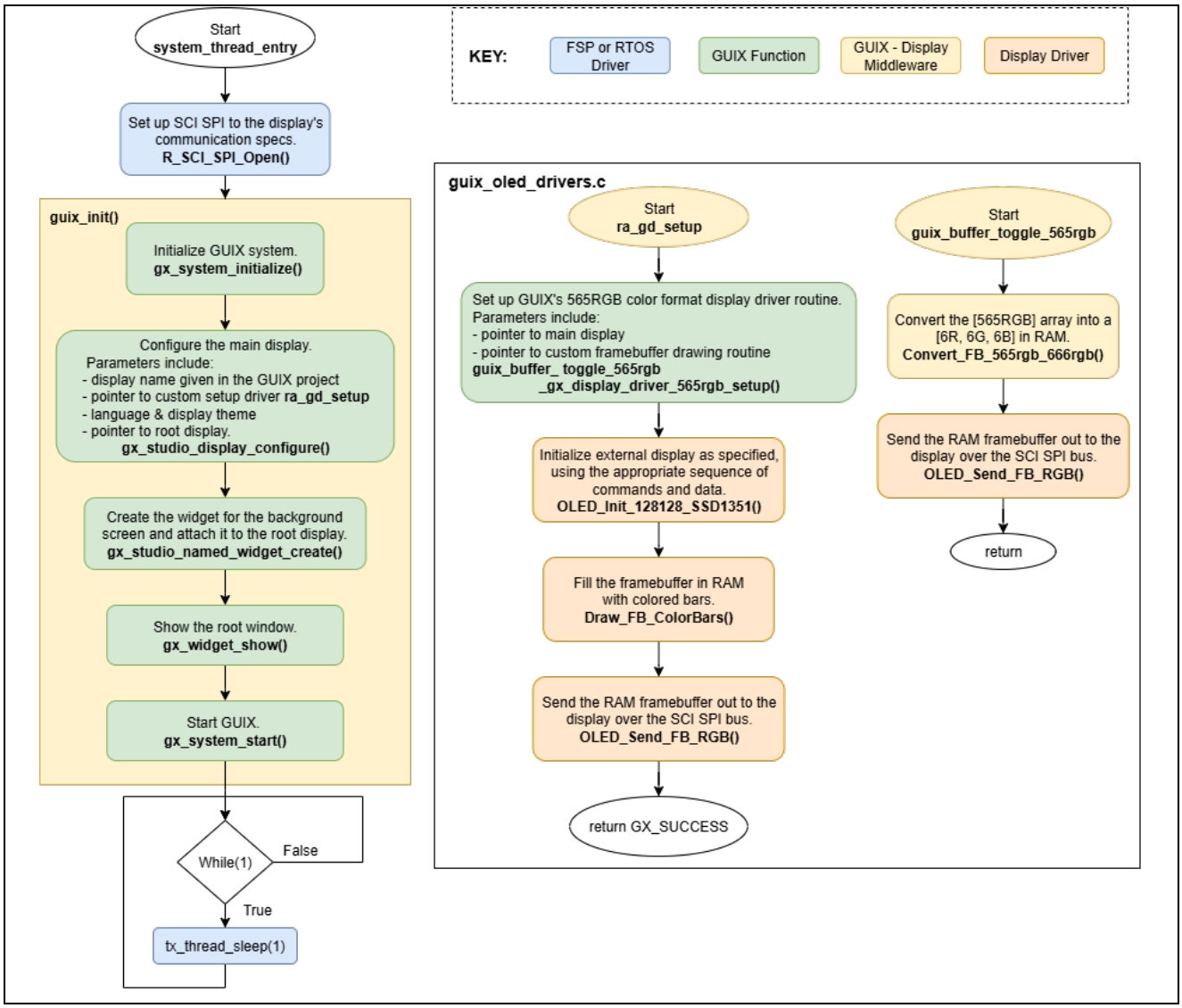


**Figure 22. Flowchart of the Hello World GUIX project**

In the main application thread **`system_thread_entry()`,** the SCI SPI hardware is initialized before calling the **`guix_init()`** middleware routine. The **`guix_init()`** routine initializes the GUIX system, configures the main display, defines the custom setup routine **`ra_gd_setup()`,** defines the custom buffer toggle routine **`guix_buffer_toggle_565rgb()`,** initializes the GUIX display window, and then starts the GUIX System Thread.

The GUIX System Thread calls the **`ra_gd_setup()`** routine once to initialize the display, using the sequence explained in Section 3.4.2. Each time the GUIX System Thread draws a new framebuffer, it then calls the **`guix_buffer_toggle_565rgb()`**. This routine redraws the framebuffer to the OLED color format before sending it out on the SPI bus to the display.

## 7.3 Example: Validate Hello World GUIX Project

After reading this section to understand the project mechanisms, run the *HelloWorld_GUIX_SCI_SPI_ra4e1* project using the steps provided in Section 5.

Use the steps in Section 5.2 to view the GUIX-drawn framebuffer from the MCU SRAM while debugging. This validates ONLY that the GUIX library and middleware are properly drawing the **`g_guix_framebuffer`**.

**NOTE: At this time, e² studio does not support viewing the redrawn framebuffer, which is sent over SPI to the OLED.**

# 8. Hello World: emWin

The application project titled *Hello_World_emWin_SCI_SPI_ra4e1* uses the emWin library running on FreeRTOS to execute an AppWizard graphics application titled appwizard_hello_world.AppWizard onto the NHD-1.5-AU-Shield display.

The steps in the next subsections can be generalized to add and integrate an AppWizard project for any non-graphics-focused RA MCU and any external display.

## 8.1 Add emWin Library through Components

For RA MCUs that do not include native graphics hardware support, the GUIX library must be added manually to the e² studio project. Note that the e² studio project should be running with FreeRTOS.

Open the **configuration.xml** file in **the FSP Configuration View**, and click on the **Components** tab.

In the component tree, add **SEGGER > GUI > all > emWin**, ensuring that the version selected matches the FSP version used in the project. Click **Generate Project Content** to include the emWin library in your project.

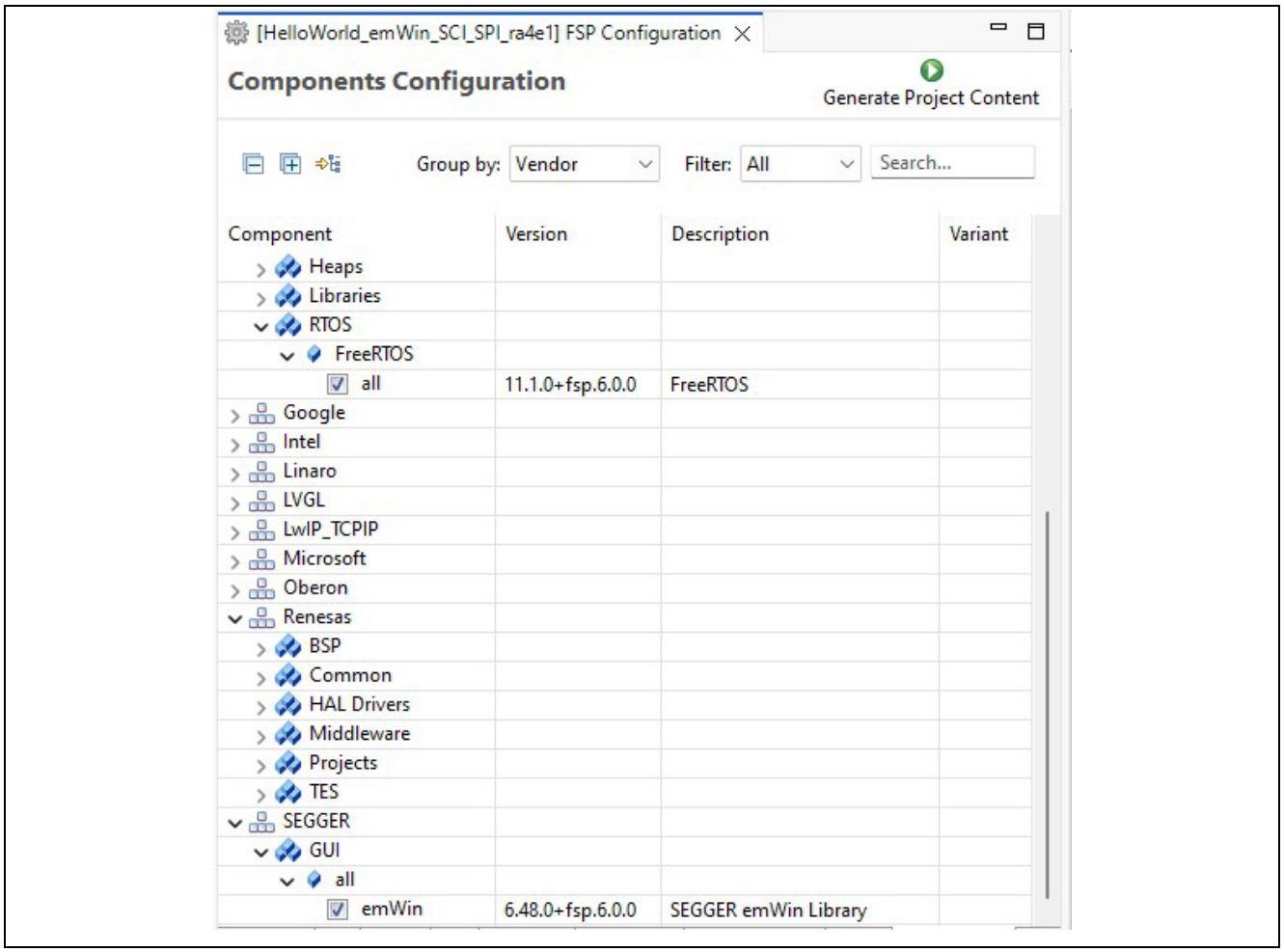


**Figure 23. Add the emWin library in the Components tab**

## 8.2 Integrating the AppWizard and e² studio Projects

The Hello World project is a simple non-animated graphics application that displays a custom JPEG image onto the NHD OLED. An overview of the AppWizard graphics application *appwizard_hello_world.AppWizard* is shown in the screenshot below:

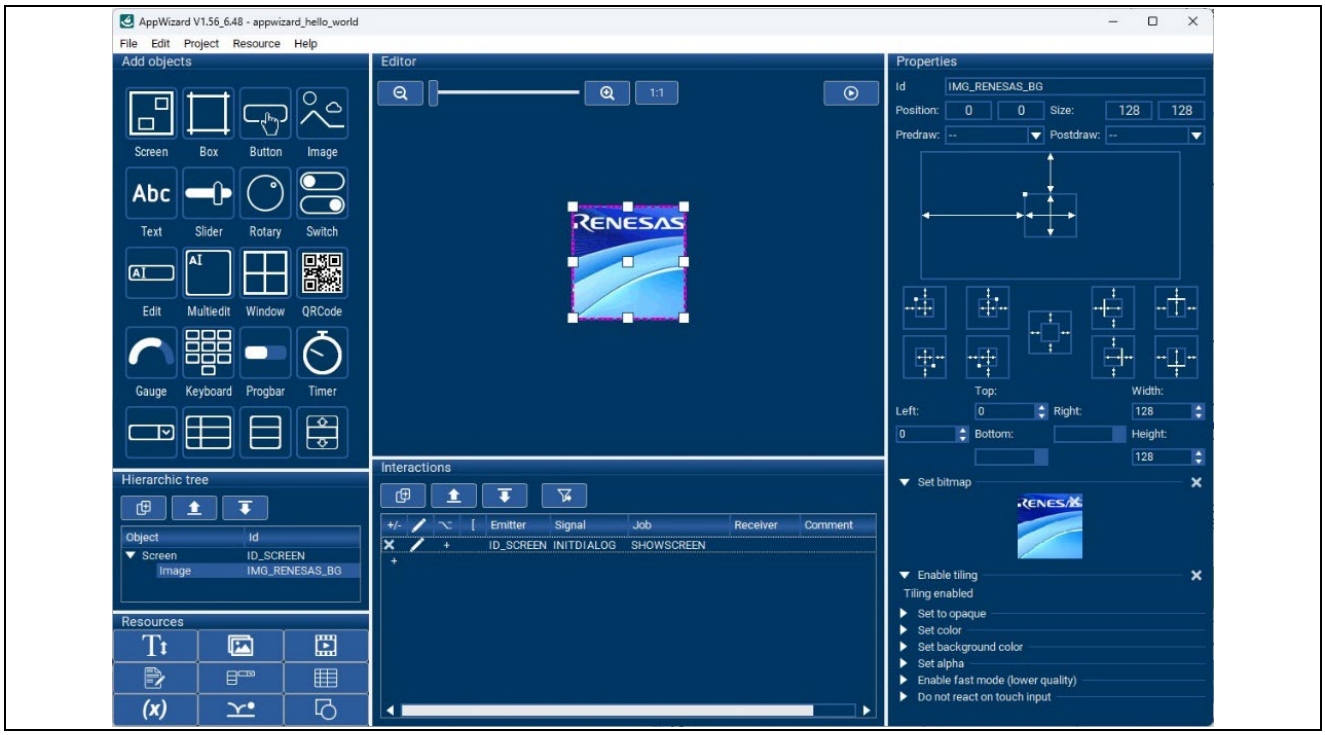


**Figure 24. Screenshot of the Hello World AppWizard project**

Review the AppWizard project options window from the tab **Project > Edit Options…**, shown in Figure 25 below.

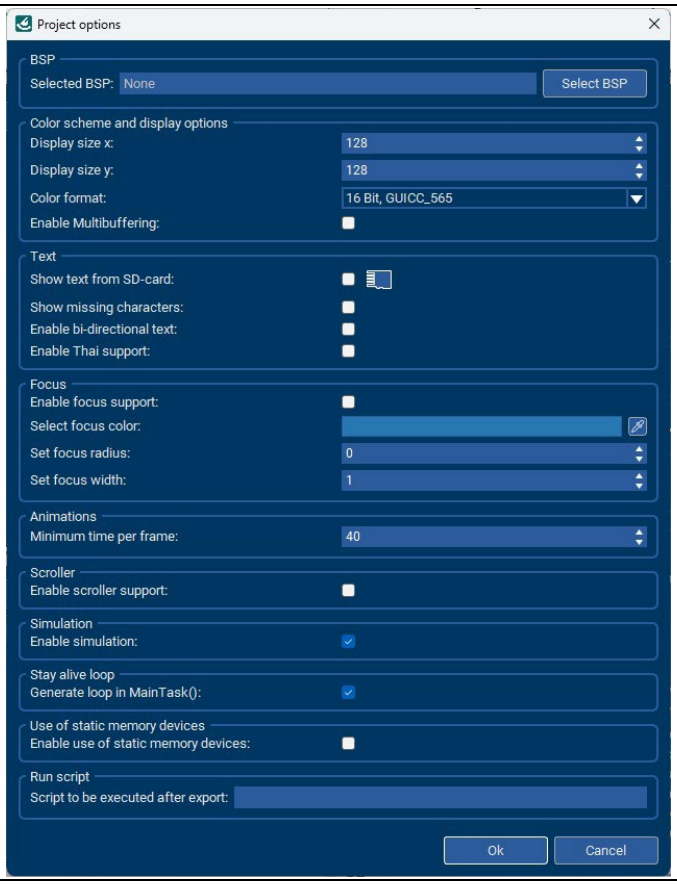

**Figure 25. The Project Options of the Hello World AppWizard Project**

The “Color scheme and display options” section contains information related to the external display settings and how emWin draws the framebuffer. The resolution has **Display size x** of **128 pixels** and **Display size y** of **128 pixels**. Since the GDDRAM on the OLED acts as a second framebuffer, **Enable Multibuffering** is **Disabled.**

Like with the GUIX project, there is NO option for the **Color format** that matches the OLED’s pixel color format. The ideal format for the NHD OLED is 24 bits per pixel (bpp) using 666RGB in the LSB position: 00RRRRRR00GGGGGG00BBBBBB.

For the AppWizard Project Options, it does NOT matter what is specified as the **Color format,** because this will be overridden at runtime by a custom application-defined color conversion created for the OLED.

The emWin User’s Manual Chapter 6.6.8 “Application-defined color conversion” explains how to create custom color conversion routines for an emWin application. The custom routines enable the emWin system to draw the framebuffer with the specified color format for the external display. The next subsection will explain this concept as part of the larger middleware.

### 8.2.1 Create the emWin Middleware

The emWin Hello World application runs on FreeRTOS, and it executes the *appwizard_hello_world.AppWizard* project on the NHD OLED using an extension of the OLED SCI SPI drivers explained in Section 3.4.3 Drawing a Framebuffer.

The following image is the application block diagram, which shows the relationship between hardware and software modules used and the movement of the framebuffer data.

The emWin system will draw updates to the **`g_framebuffer`** array over the CPU bus using the custom color conversion routines defined in *GUICC_666_LSB_18.c* to draw with the correct OLED format. The framebuffer is sent over the SPI bus to the NHD OLED using DMA support. Additional GPIO signals are used to communicate with the display.

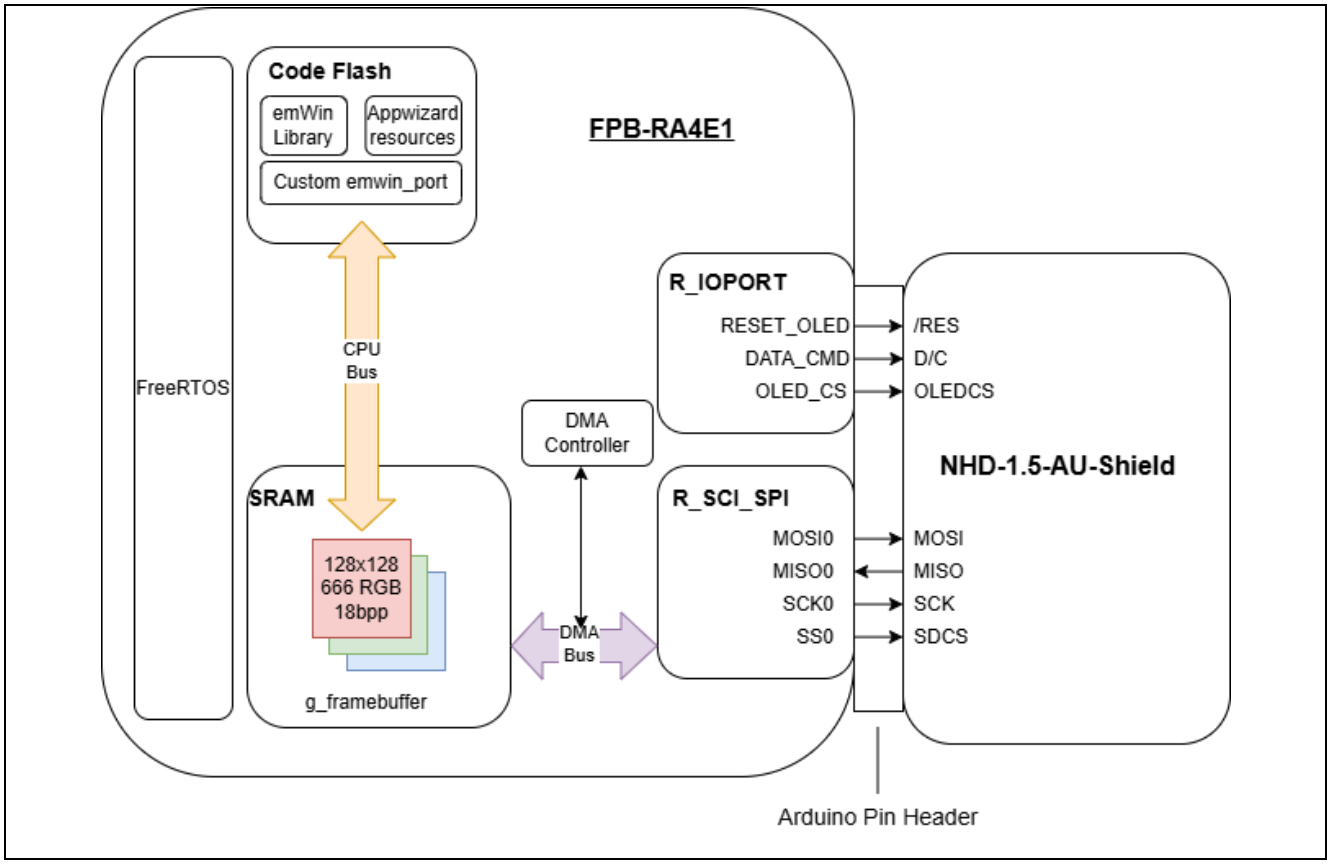


**Figure 26. The emWin Hello World Application Block Diagram**

The screenshot of the e² studio workspace below in Figure 27 categorizes the files in the e² studio project.

The **/src/appwizard_hello_world** folder contains the AppWizard-generated embedded C source and resource files along with the AppWizard project. The AppWizard project files are boxed in blue, and the generated files are boxed in red. Only the red folders/files are recursively included in the C project.

Chapter 3 of the emWin User's Manual describes how to "configure the software for using emWin on a target system". This means designing emWin middleware, which integrates the low-level OLED SCI SPI display drivers with the emWin library.

The **/src/emwin_port** folder contains emWin middleware and a custom color conversion file *GUICC_666_LSB_18.c.*

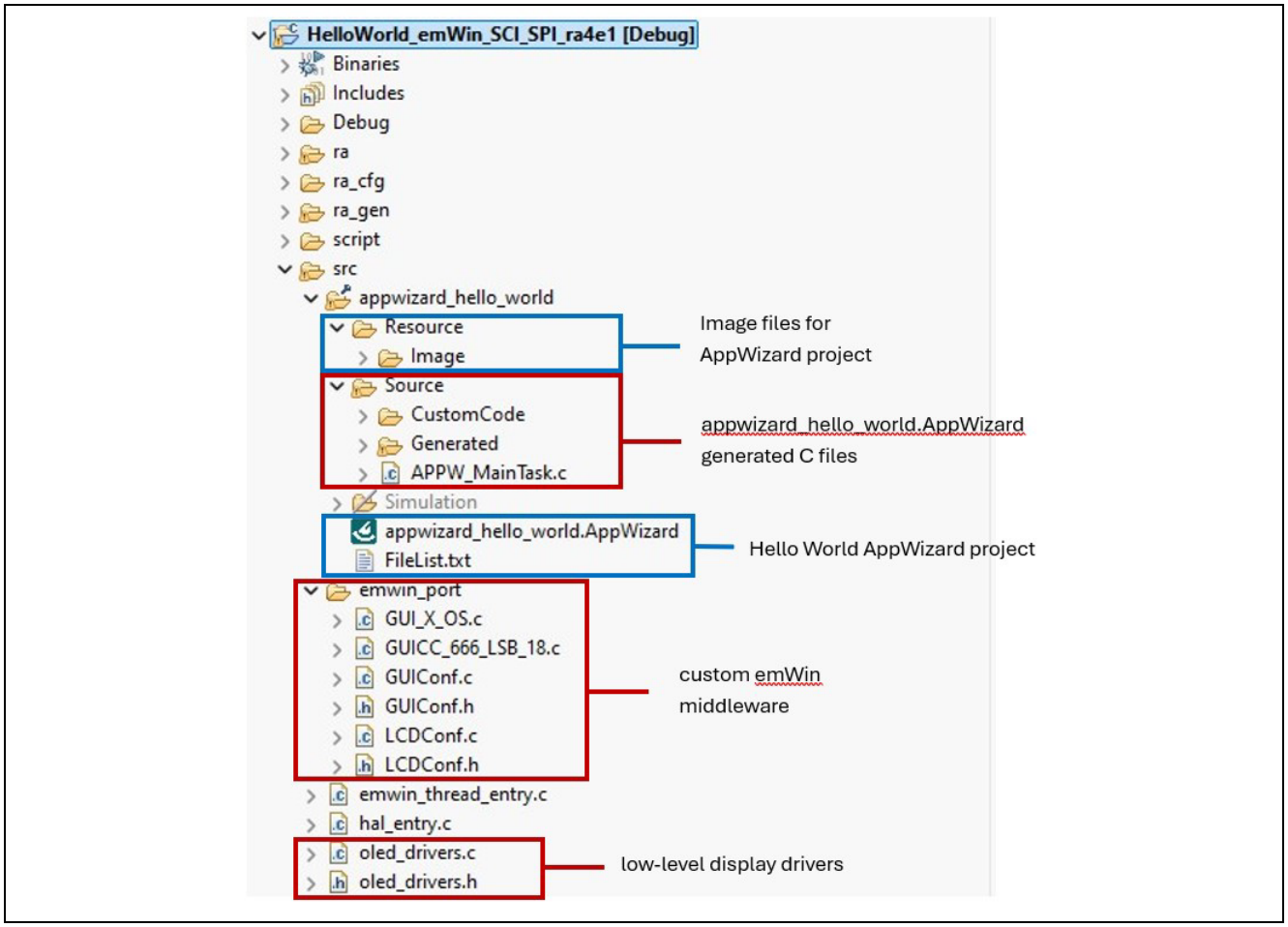


**Figure 27. Categorizing emWin Hello World e² studio project files**

At runtime, the emWin Main Task executes the *AppWizard* graphics application. It uses the emWin library and custom color conversion routines to draw the framebuffer, and it uses the emWin middleware to initialize the MCU-related hardware and call the low-level display drivers.

The main application thread flowchart is shown in Figure 28. The application loop is simple: start the Main Task and ensure that emWin continues to run.

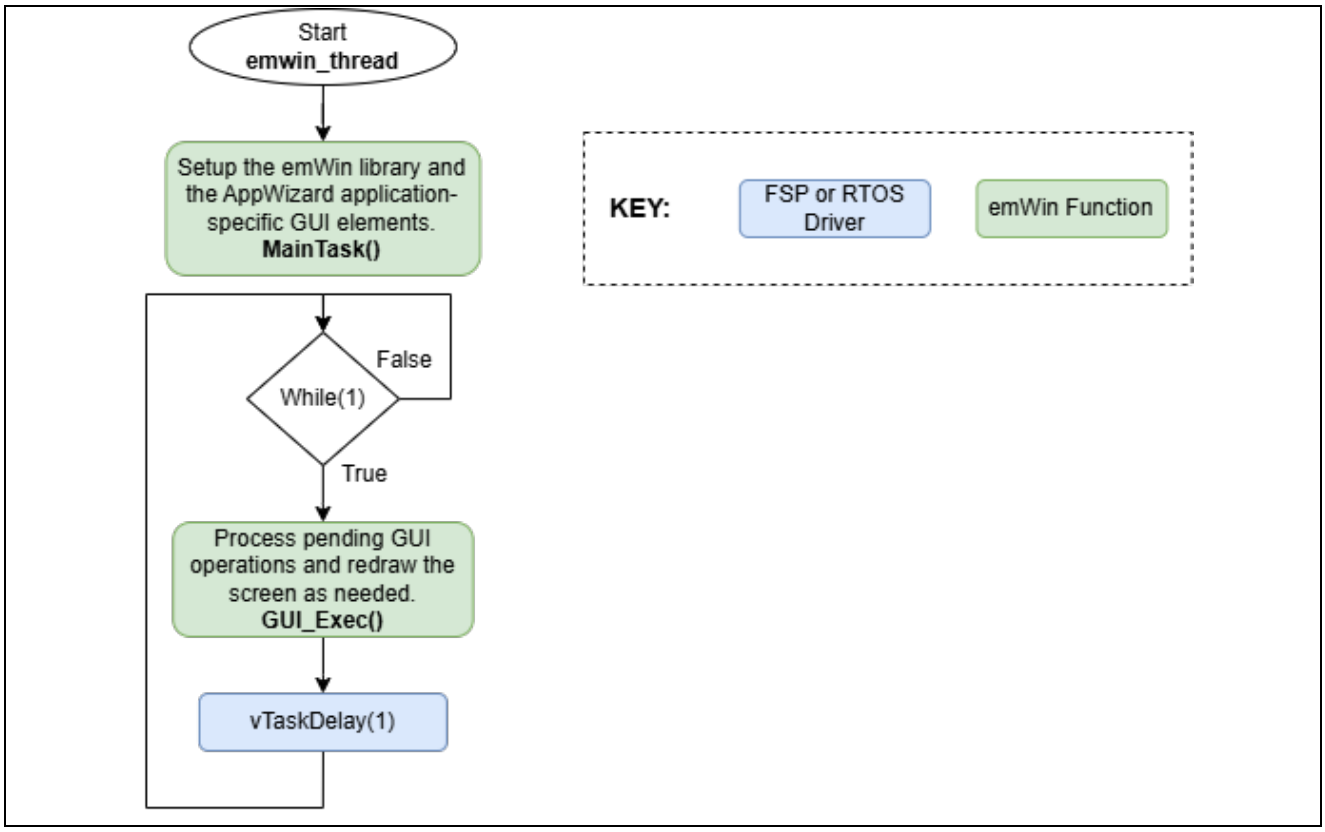


**Figure 28. The emWin Main Task will execute the AppWizard graphics application**

According to the emWin User's Manual, the middleware file *LCDConf.c* is required to provide emWin with the required display configuration routine and the callback function for the display driver. The following application-specific routines need to be defined in this file:

- **`LCD_X_Config():`** Configuration routine for creating the display driver device and setting the color conversion routines and display size.
- **`LCD_X_DisplayDriver():`** Callback routine called by the display driver for putting the display controller into operation.

The flowchart of LCDConf.c from the Hello World emWin project is shown in Figure 29 below. The required routines are defined as follows for this specific application:

- **`LCD_X_Config(void):`**
  - Sets the display driver to **`GUIDRV_LIN_24`** and the color conversion to the custom routine from the *GUICC_666_LSB_18.c* file.
  - Sets the display size to 128x128 pixels to match the NHD OLED.
  - Sets the framebuffer to the variable in SRAM that is used throughout the application.
  - Sets up a control hook callback routine called **UpdateDisplay**.
  - Ensures multibuffering is disabled.
- **`LCD_X_DisplayDriver( unsigned LayerIndex, unsigned Cmd, void * pData):`** Filters the command passed to the function by the emWin Main Task and handles each command's case accordingly. The following events required handling in this application:
  - **`LCD_X_INITCONTROLLER:`** Initializes the SCI SPI module to start the SPI bus, calls the OLED-specific initialization routine, and fills the screen with color bars to visually confirm initialization.
  - **`LCD_X_ON:`** Use the command to turn on the display.
  - **`LCD_X_OFF:`** Use the command to turn off the display.

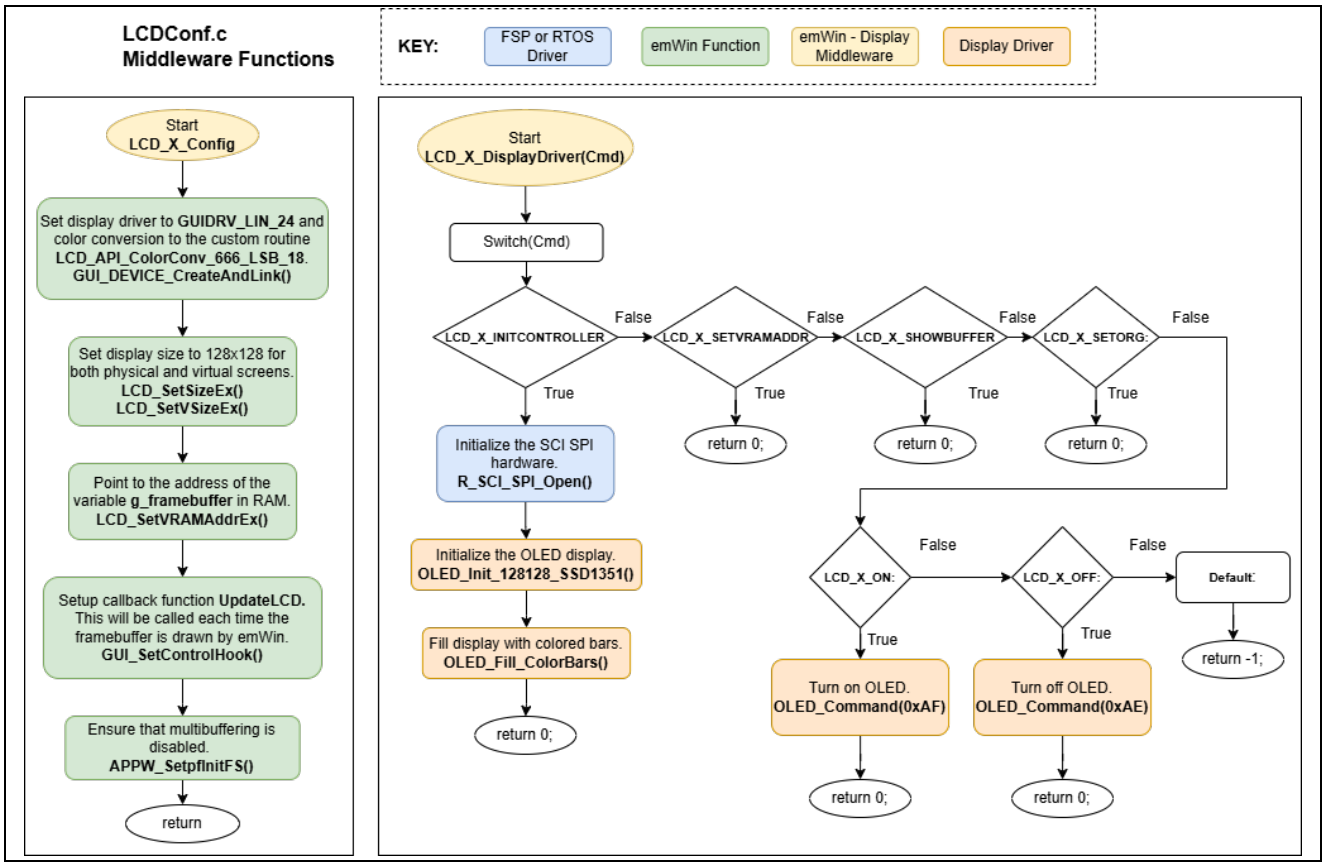


**Figure 29. The emWin Hello World project's LDC_X_Config and LCD_X_DisplayDriver**

The emWin library allows users to set hook functions for various purposes, like initialization or managing cache buffers. The full list of hook functions can be found in the emWin User's Manual Chapter 3.10.

The **`LCD_X_Config()`** function sets up a Control Hook routine called **`UpdateOLED()`**. A control hook is called immediately before a display driver cache operation should be executed. In the Hello World application, the control hook is used because **`GUIDRV_LIN_24`** manages the content of the layer, but custom display drivers are needed to transfer it to the display. A flowchart of the **`UpdateOLED()`** routine is shown below in Figure 30:

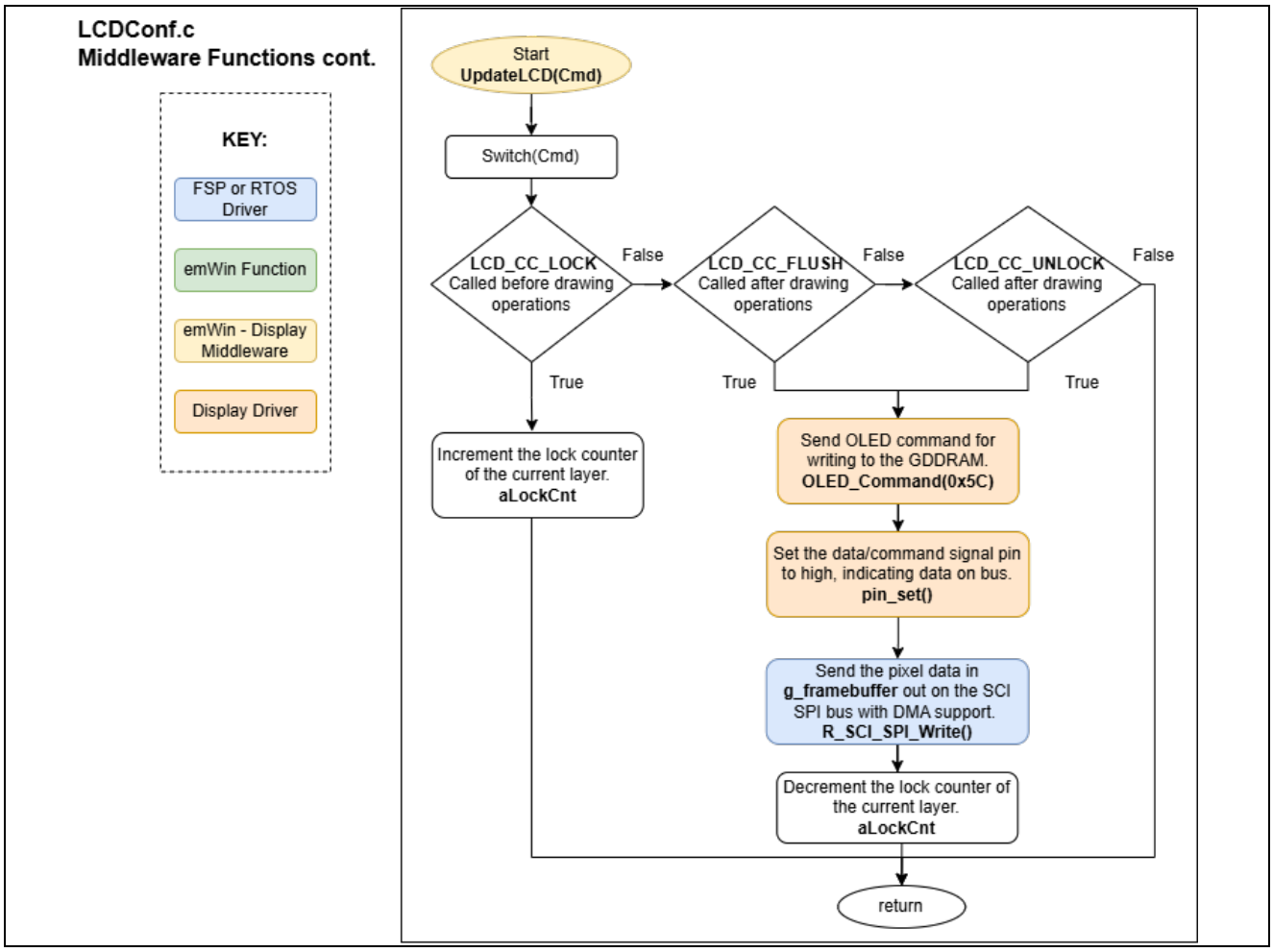


**Figure 30. The refresh control hook UpdateOLED sends the framebuffer to the OLED**

The command **LCD_CC_LOCK** is passed to the control hook before emWin updates a specified layer of the framebuffer. A lock counter is used to keep track of which layer is going to be updated.

The commands **LCD_CC_FLUSH** or **LCD_CC_UNLOCK** are passed to the control hook after emWin completes drawing operations for the specified layer. In other words, the events happen after emWin has finished drawing a new framebuffer in the MCU SRAM. At this time, the low-level display drivers are used to send the framebuffer variable to the NHD OLED over the SPI bus using DMA support.

### 8.2.2 Create the Custom Color Conversion Routines

The emWin User's Manual Chapter 6.6.8 describes how to create custom color conversion routines for an emWin application if none of the fixed palette modes match the needs of the external display.

The application file *GUICC_666_LSB_18.c was* created with support by Segger, and the routines enable the emWin system to draw the framebuffer with the specified color format for the NHD OLED. Together, the routines work to translate emWin's internal 32-bit logical color representation into the padded 666RGB format required by the display hardware.

1. **_Color2Index_666_LSB_18(LCD_COLOR Color)**
    - **Summary**: This static function converts a standard 32-bit logical color (LCD_COLOR) used by the application into an 18-bit pixel index value (LCD_PIXELINDEX).
    - **Conversion Mechanism:** It extracts the R, G, and B components from the input color (handling different byte orders if needed). It then uses a conversion array (**GUI__aConvert_255_63**) to map the 8-bit (0-255) color intensity values down to 6-bit (0-63) values. Finally, it combines these 6-bit R, G, and B values into a 32-bit result. Red is stored in bits 0-5, Green in bits 8-13, and Blue in bits 16-21, consistent with the LSB format.
2. **_Index2Color_666_LSB_18(LCD_PIXELINDEX Index)**

- **Summary**: This static function converts an 18-bit pixel index value from the display hardware format back into a standard 32-bit logical color (**`LCD_COLOR`**) for the application.
- **Conversion Mechanism:** It extracts the 6-bit R, G, and B values from the input Index using bit shifting and masking (0x3F). It uses a different conversion array (**`GUI__aConvert_63_255`**) to map these 6-bit values back up to 8-bit (0-255) color components. If the framework is using the ARGB color format (**`GUI_USE_ARGB`**), it sets the upper 8 bits to denote an opaque alpha channel (**`0xFF000000ul`**) when returning the logical color.

3. **`_GetIndexMask_666_LSB_18(void)`**
   - **Summary**: This static function returns the bitmask (**`LCD_PIXELINDEX`**) that defines which bits are used for color information in this mode.
   - **Mechanism**: It returns 0x003F3F3F. This mask consists of 6 bits (0x3F) for each color channel (R, G, and B) within the 32-bit index structure.
4. **`_Color2IndexBulk(...)`**
   - **Summary**: This static function is designed for bulk conversion of an array of logical colors into an array of pixel index values.
   - **Mechanism**: It performs the same logical conversion as **`_Color2Index_666_LSB_18`**, but optimizes the process for multiple colors/items (NumItems), often used when drawing large blocks of data.
5. **`_Index2ColorBulk(...)`**
   - **Summary**: This static function performs bulk conversion of an array of pixel index values back into an array of logical colors.
   - **Mechanism**: It iterates through the input index data (**`pIndex`**), performs the 6-bit to 8-bit component expansion using **`GUI__aConvert_63_255`**, and stores the resulting **`LCD_COLOR`** in the output buffer (**`pColor`**) for all **`NumItems`**.

The presence of both single-item and bulk conversion functions enhances performance, especially if hardware acceleration is available.

## 8.3 Example: Validate Hello World emWin

After understanding this section, run the *HelloWorld_emWin_SCI_SPI_ra4e1* project using the steps in Section 5, while skipping Section 5.1. Unfortunately, the Memory View's Raw Image Rendering does not support the color format with which the **`g_framebuffer`** is encoded.

# 9. Next Steps

Once your graphics application is up and running using Renesas FSP with GUIX or emWin, there are several exciting directions you can take to expand functionality, improve performance, or tailor the solution to specific use cases. Consider exploring the following topics:

- **Add PWM Backlight Control**
  While OLED displays like the SSD1351 do not require backlighting, this feature is essential for LCD-based designs. Implementing PWM control allows for dynamic brightness adjustment and power savings.
- **Integrate Touch Control**
  Consider adding application control with touch input. You can add a resistive or capacitive touch screen layer on top of a display, or you can choose display hardware with a touch screen already integrated. Touch controllers typically communicate with I2C or SPI, and drivers can be designed using the FSP APIs.
- **Perform Low Power Analysis**
  Evaluate your application's power consumption using Renesas' low-power modes and tools. Optimize display refresh rates, SPI activity, and GUI task scheduling to extend battery life.
- **Enable Multi-Language UI Support**
  Both GUIX and emWin support internationalization. Add font resources and language switching logic to make your application accessible to global users.

- **Add Animation and Transition Effects**
  Enhance user experience with smooth transitions, animated widgets, and screen effects. GUIX and emWin both offer APIs for animation control. The MCU memory will affect the number of images that can be stored, and the time to draw and send a full framebuffer will affect the frame rate.
- **Implement Dynamic Content Rendering**
  Use real-time data (e.g., sensor readings, network updates) to update GUI elements dynamically. This is especially useful for dashboards and monitoring applications.
- **Use External Flash for GUI Assets**
  Store large images, fonts, or GUI resources in external QSPI flash to free up internal memory and support richer interfaces.
- **Add GUI-Based Diagnostics or Debug Panel**
  Create a hidden or developer-accessible screen to display system status, logs, or performance metrics for field testing and troubleshooting.

## Website and Support

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

| | |
|---|---|
| EK-RA8P1 Resources | www.renesas.com/ek-ra8p1 |
| MCK-RA8T2 Resources | www.renesas.com/mck-ra8t2 |
| EK-RA8M2 Resources | www.renesas.com/ek-ra8m2 |
| EK-RA8D2 Resources | www.renesas.com/ek-ra8d2 |
| 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 | Jan.16.26 | - | Initial version |

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

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

1. Precaution against Electrostatic Discharge (ESD)

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

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

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

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

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

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

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

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

# Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

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

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

(Rev.5.0-1 October 2020)

## Corporate Headquarters

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

## Contact information

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

## Trademarks

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